

Book of Vaadin

Vaadin 7 Edition - 7th Revision



Marko Grönroos

2016
Vaadin Ltd

Book of Vaadin: Vaadin 7 Edition - 7th Revision

Vaadin Ltd

Marko Grönroos

Vaadin 7 Edition - 7th Revision Edition

Vaadin Framework 7.6

Published: 2016-04-19

Copyright © 2000-2016 Vaadin Ltd

Abstract

Vaadin is an AJAX web application development framework that enables developers to build high-quality user interfaces with Java, both on the server- and client-side. It provides a set of libraries of ready-to-use user interface components and a clean framework for creating your own components. The focus is on ease-of-use, re-usability, extensibility, and meeting the requirements of large enterprise applications.

All rights reserved. This work is licensed under the Creative Commons CC-BY-ND License Version 2.0.

Table of Contents

Preface	xix
1. Who is This Book For?	xix
2. Organization of This Book	xix
3. Supplementary Material	xxi
4. Support	xxii
5. About the Editor	xxii
6. Acknowledgements	xxiii
7. About Vaadin Ltd	xxiii
Chapter 1. Introduction	25
1.1. Overview	25
1.2. Example Application Walkthrough	27
1.3. Support for the Eclipse IDE	28
1.4. Goals and Philosophy	29
1.4.1. Right tool for the right purpose	29
1.4.2. Simplicity and maintainability	29
1.4.3. Choice between declarative and dynamic UIs	29
1.4.4. Tools should not limit your work	29
1.5. Background	29
1.5.1. Release 5 Into the Open	30
1.5.2. Birth of Vaadin Release 6	30
1.5.3. The Major Revision with Vaadin 7	30
Chapter 2. Installing the Development Toolchain	33
2.1. Overview	33
2.2. A Reference Toolchain	34
2.3. Installing Java SDK	35
2.3.1. Windows	35
2.3.2. Linux / UNIX	36
2.4. Installing a Web Server	36
2.4.1. Installing Apache Tomcat	37
2.5. Installing the Eclipse IDE and Plugin	37
2.5.1. Installing the Eclipse IDE	38
2.5.2. Installing the Vaadin Plugin	39
2.5.3. Notification Center	40
2.5.4. Updating the Plugins	42
2.6. Installing the NetBeans IDE and Plugin	43
2.6.1. Installing the NetBeans IDE	43
2.6.2. Installing the Vaadin Plug-in for NetBeans IDE	44
2.7. Installing and Configuring IntelliJ IDEA	44
2.7.1. Installing the Ultimate Edition	44
2.8. Installing Other Tools	46
2.8.1. Firefox and Firebug	46
Chapter 3. Creating a Vaadin Application	47
3.1. Overview	47
3.2. Vaadin Libraries	48
3.3. Overview of Maven Archetypes	49
3.4. Creating and Running a Project in Eclipse	49
3.4.1. Creating a Maven Project	50
3.4.2. Exploring the Project	52
3.4.3. Compiling the Widget Set and Theme	53

3.4.4. Coding Tips for Eclipse	53
3.4.5. Setting Up and Starting the Web Server	54
3.4.6. Running and Debugging	56
3.4.7. Updating the Vaadin Libraries in Maven Projects	57
3.4.8. Updating the Vaadin Libraries in Ivy Projects	58
3.5. Creating a Project with Maven	58
3.5.1. Working from Command-Line	59
3.5.2. Compiling and Running the Application	60
3.5.3. Using Add-ons and Custom Widget Sets	60
3.6. Creating a Project with the NetBeans IDE	60
3.6.1. Creating a Project	60
3.6.2. Exploring the Project	62
3.6.3. Running the Application	63
3.7. Creating a Project with IntelliJ IDEA	63
3.7.1. Creating a Vaadin Web Application Project	64
3.7.2. Creating a Maven Project	66
3.8. Vaadin Installation Package	68
3.8.1. Package Contents	68
3.8.2. Installing the Libraries	68
3.9. Using Vaadin with Scala	68
Chapter 4. Architecture	71
4.1. Overview	71
4.2. Technological Background	74
4.2.1. HTML and JavaScript	74
4.2.2. Styling with CSS and Sass	75
4.2.3. AJAX	75
4.2.4. Google Web Toolkit	75
4.2.5. Java Servlets	76
4.3. Client-Side Engine	77
4.4. Events and Listeners	78
Chapter 5. Writing a Server-Side Web Application	81
5.1. Overview	81
5.2. Building the UI	84
5.2.1. Application Architecture	86
5.2.2. Compositing Components	86
5.2.3. View Navigation	87
5.2.4. Accessing UI, Page, Session, and Service	88
5.3. Designing UIs Declaratively	88
5.3.1. Declarative Syntax	89
5.3.2. Component Elements	89
5.3.3. Component Attributes	91
5.3.4. Component Identifiers	92
5.3.5. Using Designs in Code	92
5.4. Handling Events with Listeners	93
5.4.1. Using Anonymous Classes	93
5.4.2. Handling Events in Java 8	94
5.4.3. Implementing a Listener in a Regular Class	95
5.4.4. Differentiating Between Event Sources	95
5.5. Images and Other Resources	96
5.5.1. Resource Interfaces and Classes	96
5.5.2. File Resources	97
5.5.3. Class Loader Resources	97

5.5.4. Theme Resources	98
5.5.5. Stream Resources	98
5.6. Handling Errors	100
5.6.1. Error Indicator and Message	100
5.6.2. Connection Fault	100
5.7. Notifications	100
5.7.1. Notification Type	101
5.7.2. Styling with CSS	102
5.8. Application Lifecycle	103
5.8.1. Deployment	103
5.8.2. Vaadin Servlet, Portlet, and Service	104
5.8.3. User Session	104
5.8.4. Loading a UI	105
5.8.5. UI Expiration	106
5.8.6. Closing UIs Explicitly	106
5.8.7. Session Expiration	107
5.8.8. Closing a Session	107
5.9. Deploying an Application	108
5.9.1. Creating Deployable WAR in Eclipse	109
5.9.2. Web Application Contents	109
5.9.3. Web Servlet Class	109
5.9.4. Using a web.xml Deployment Descriptor	110
5.9.5. Servlet Mapping with URL Patterns	111
5.9.6. Other Servlet Configuration Parameters	112
5.9.7. Deployment Configuration	114
Chapter 6. User Interface Components	117
6.1. Overview	118
6.2. Interfaces and Abstractions	120
6.2.1. Component Interface	121
6.2.2. AbstractComponent	121
6.3. Common Component Features	122
6.3.1. Caption	122
6.3.2. Description and Tooltips	124
6.3.3. Enabled	125
6.3.4. Icon	126
6.3.5. Locale	127
6.3.6. Read-Only	128
6.3.7. Style Name	129
6.3.8. Visible	129
6.3.9. Sizing Components	130
6.3.10. Managing Input Focus	132
6.4. Field Components	132
6.4.1. Field Interface	133
6.4.2. Data Binding and Conversions	134
6.4.3. Handling Field Value Changes	134
6.4.4. Field Buffering	135
6.4.5. Field Validation	135
6.5. Selection Components	138
6.5.1. Binding Selection Components to Data	138
6.5.2. Adding New Items	139
6.5.3. Item Captions	139
6.5.4. Getting and Setting Selection	141
6.5.5. Handling Selection Changes	142

6.5.6. Allowing Adding New Items	142
6.5.7. Multiple Selection	143
6.5.8. Item Icons	143
6.6. Component Extensions	144
6.7. Label	144
6.7.1. Text Width and Wrapping	145
6.7.2. Content Mode	145
6.7.3. Data Binding	147
6.7.4. CSS Style Rules	148
6.8. Link	148
6.8.1. CSS Style Rules	149
6.9. TextField	150
6.9.1. Data Binding	151
6.9.2. String Length	151
6.9.3. Handling Null Values	152
6.9.4. Text Change Events	152
6.9.5. CSS Style Rules	154
6.10. TextArea	154
6.10.1. Word Wrap	155
6.10.2. CSS Style Rules	156
6.11. PasswordField	156
6.11.1. CSS Style Rules	157
6.12. RichTextArea	157
6.12.1. CSS Style Rules	158
6.13. Date and Time Input with DateField	158
6.13.1. PopupDateField	159
6.13.2. InlineDateField	161
6.13.3. Date and Time Resolution	162
6.13.4. DateField Locale	162
6.14. Button	162
6.14.1. CSS Style Rules	163
6.15. CheckBox	163
6.15.1. CSS Style Rules	164
6.16. ComboBox	164
6.16.1. Filtered Selection	165
6.16.2. CSS Style Rules	167
6.17. ListSelect	167
6.17.1. CSS Style Rules	168
6.18. NativeSelect	168
6.18.1. CSS Style Rules	169
6.19. OptionGroup	169
6.19.1. Disabling Items	170
6.19.2. CSS Style Rules	171
6.20. TwinColSelect	172
6.20.1. CSS Style Rules	173
6.21. Table	173
6.21.1. Selecting Items in a Table	175
6.21.2. Table Features	177
6.21.3. Editing the Values in a Table	180
6.21.4. Column Headers and Footers	181
6.21.5. Generated Table Columns	184
6.21.6. Formatting Table Columns	184
6.21.7. CSS Style Rules	185

6.22. Tree	186
6.23. TreeTable	187
6.23.1. Expanding and Collapsing Items	189
6.24. Grid	189
6.24.1. Overview	189
6.24.2. Binding to Data	191
6.24.3. Handling Selection Changes	192
6.24.4. Configuring Columns	195
6.24.5. Generating Columns	197
6.24.6. Column Renderers	197
6.24.7. Header and Footer	200
6.24.8. Filtering	201
6.24.9. Sorting	202
6.24.10. Editing	203
6.24.11. Programmatic Scrolling	205
6.24.12. Generating Row or Cell Styles	205
6.24.13. Styling with CSS	206
6.25. MenuBar	207
6.25.1. Creating a Menu	207
6.25.2. Handling Menu Selection	208
6.25.3. CSS Style Rules	208
6.26. Upload	209
6.26.1. Receiving Upload Data	209
6.26.2. CSS Style Rules	211
6.27. ProgressBar	211
6.27.1. Indeterminate Mode	212
6.27.2. CSS Style Rules	212
6.28. Slider	213
6.28.1. CSS Style Rules	214
6.29. PopupView	214
6.29.1. CSS Style Rules	215
6.30. Calendar	215
6.30.1. Date Range and View Mode	216
6.30.2. Calendar Events	218
6.30.3. Getting Events from a Container	219
6.30.4. Backward and Forward Navigation	220
6.31. Composition with CustomComponent	220
6.32. Composite Fields with CustomField	222
6.33. Embedded Resources	222
6.33.1. Embedded Image	223
6.33.2. Adobe Flash Graphics	224
6.33.3. BrowserFrame	224
6.33.4. Generic Embedded Objects	224
Chapter 7. Managing Layout	227
7.1. Overview	228
7.2. UI, Window, and Panel Content	229
7.3. VerticalLayout and HorizontalLayout	230
7.3.1. Properties or Attributes	230
7.3.2. Spacing in Ordered Layouts	231
7.3.3. Sizing Contained Components	231
7.4. GridLayout	235
7.4.1. Sizing Grid Cells	236
7.4.2. CSS Style Rules	238

7.5. FormLayout	239
7.5.1. CSS Style Rules	240
7.6. Panel	241
7.6.1. Scrolling the Panel Content	241
7.7. Sub-Windows	243
7.7.1. Opening and Closing Sub-Windows	244
7.7.2. Window Positioning	245
7.7.3. Scrolling Sub-Window Content	245
7.7.4. Modal Sub-Windows	246
7.8. HorizontalSplitPanel and VerticalSplitPanel	246
7.8.1. CSS Style Rules	248
7.9. TabSheet	248
7.9.1. Adding Tabs	249
7.9.2. Tab Objects	249
7.9.3. Tab Change Events	250
7.9.4. Enabling and Handling Closing Tabs	250
7.10. Accordion	251
7.10.1. CSS Style Rules	252
7.11. AbsoluteLayout	253
7.11.1. Placing a Component in an Area	254
7.11.2. Proportional Coordinates	254
7.11.3. Styling with CSS	255
7.12. CssLayout	255
7.12.1. CSS Injection	256
7.12.2. Browser Compatibility	257
7.12.3. Styling with CSS	257
7.13. Layout Formatting	258
7.13.1. Layout Size	258
7.13.2. Expanding Components	259
7.13.3. Layout Cell Alignment	259
7.13.4. Layout Cell Spacing	262
7.13.5. Layout Margins	262
7.14. Custom Layouts	263
Chapter 8. Vaadin Designer	267
8.1. Overview	267
8.2. Installation	269
8.2.1. Installing Eclipse and Plug-Ins	269
8.2.2. License	269
8.2.3. Uninstalling	269
8.3. Getting Started	269
8.3.1. Creating a Design	270
8.3.2. Vaadin Designer GUI Overview	271
8.4. Designing	273
8.4.1. About Layouts	273
8.4.2. Starting from Blank	274
8.4.3. Using Templates	275
8.4.4. Adding Components	275
8.4.5. Previewing	278
8.5. Theming and Styling	281
8.5.1. Theme Based on Valo	281
8.5.2. Theme File	282
8.6. Wiring It Up	282
8.6.1. Declarative Code	283

8.6.2. Java Code	284
8.7. Limitations	285
Chapter 9. Themes	287
9.1. Overview	287
9.2. Introduction to Cascading Style Sheets	289
9.2.1. Applying CSS to HTML	289
9.2.2. Basic CSS Rules	289
9.2.3. Matching by Element Class	291
9.2.4. Matching by Descendant Relationship	292
9.2.5. Importance of Cascading	292
9.2.6. Style Class Hierarchy of a Vaadin UI	293
9.2.7. Notes on Compatibility	296
9.3. Syntactically Awesome Stylesheets (Sass)	296
9.3.1. Sass Overview	297
9.3.2. Sass Basics with Vaadin	298
9.4. Compiling Sass Themes	298
9.4.1. Compiling On the Fly	298
9.4.2. Compiling in Eclipse	299
9.4.3. Compiling with Maven	299
9.4.4. Compiling with Ant	300
9.5. Creating and Using Themes	301
9.5.1. Sass Themes	301
9.5.2. Plain Old CSS Themes	303
9.5.3. Styling Standard Components	303
9.5.4. Built-in Themes	303
9.5.5. Add-on Themes	304
9.6. Creating a Theme in Eclipse	305
9.7. Valo Theme	307
9.7.1. Basic Use	307
9.7.2. Common Settings	307
9.7.3. Valo Mixins and Functions	311
9.7.4. Valo Fonts	311
9.7.5. Component Styles	312
9.7.6. Theme Optimization	312
9.8. Font Icons	313
9.8.1. Loading Icon Fonts	313
9.8.2. Basic Use	313
9.8.3. Using Font icons in HTML	314
9.8.4. Using Font Icons in Other Text	315
9.8.5. Custom Font Icons	315
9.9. Custom Fonts	315
9.9.1. Loading Local Fonts	316
9.9.2. Loading Web Fonts	316
9.9.3. Using Custom Fonts	316
9.10. Responsive Themes	316
Chapter 10. Binding Components to Data	319
10.1. Overview	319
10.2. Properties	321
10.2.1. Property Viewers and Editors	322
10.2.2. ObjectProperty Implementation	323
10.2.3. Converting Between Property Type and Representation	323
10.3. Holding properties in Items	326

10.3.1. Wrapping a Bean in a BeanItem	326
10.4. Creating Forms by Binding Fields to Items	328
10.4.1. Simple Binding	328
10.4.2. Using a FieldFactory to Build and Bind Fields	329
10.4.3. Binding Member Fields	329
10.4.4. Buffering Forms	330
10.4.5. Binding Fields to a Bean	331
10.4.6. Bean Validation	331
10.5. Collecting Items in Containers	333
10.5.1. Basic Use of Containers	333
10.5.2. Container Subinterfaces	335
10.5.3. IndexedContainer	336
10.5.4. BeanContainer	336
10.5.5. BeanItemContainer	339
10.5.6. GeneratedPropertyContainer	340
10.5.7. Filterable Containers	341
Chapter 11. Vaadin SQLContainer	345
11.1. Architecture	346
11.2. Getting Started with SQLContainer	346
11.2.1. Creating a connection pool	347
11.2.2. Creating the TableQuery Query Delegate	347
11.2.3. Creating the Container	347
11.3. Filtering and Sorting	347
11.3.1. Filtering	348
11.3.2. Sorting	348
11.4. Editing	348
11.4.1. Adding items	348
11.4.2. Fetching generated row keys	349
11.4.3. Version column requirement	349
11.4.4. Auto-commit mode	350
11.4.5. Modified state	350
11.5. Caching, Paging and Refreshing	350
11.5.1. Container Size	350
11.5.2. Page Length and Cache Size	351
11.5.3. Refreshing the Container	351
11.6. Referencing Another SQLContainer	351
11.7. Making Freeform Queries	352
11.7.1. Getting started	352
11.7.2. Limitations	353
11.7.3. Creating your own FreeformStatementDelegate	353
11.8. Non-Implemented Methods	353
11.8.1. About the <code>getItemIds()</code> method	354
11.9. Known Issues and Limitations	354
Chapter 12. Advanced Web Application Topics	357
12.1. Handling Browser Windows	358
12.1.1. Opening Popup Windows	358
12.1.2. Closing Popup Windows	360
12.2. Embedding UIs in Web Pages	361
12.2.1. Embedding Inside a <code>div</code> Element	361
12.2.2. Embedding Inside an <code>iframe</code> Element	361
12.3. Debug Mode and Window	363
12.3.1. Enabling the Debug Mode	364

12.3.2. Opening the Debug Window	364
12.3.3. Debug Message Log	365
12.3.4. General Information	366
12.3.5. Inspecting Component Hierarchy	366
12.3.6. Communication Log	368
12.3.7. Debug Modes	369
12.4. Request Handlers	369
12.5. Shortcut Keys	370
12.5.1. Shortcut Keys for Default Buttons	370
12.5.2. Field Focus Shortcuts	371
12.5.3. Generic Shortcut Actions	371
12.5.4. Supported Key Codes and Modifier Keys	374
12.6. Printing	375
12.6.1. Printing the Browser Window	375
12.6.2. Opening a Print Window	375
12.6.3. Printing PDF	376
12.7. Google App Engine Integration	377
12.7.1. Rules and Limitations	377
12.8. Common Security Issues	378
12.8.1. Sanitizing User Input to Prevent Cross-Site Scripting	378
12.9. Navigating in an Application	378
12.9.1. Setting Up for Navigation	379
12.9.2. Implementing a View	380
12.9.3. Handling URI Fragment Path	380
12.10. Advanced Application Architectures	383
12.10.1. Layered Architectures	383
12.10.2. Model-View-Presenter Pattern	384
12.11. Managing URI Fragments	388
12.11.1. Setting the URI Fragment	388
12.11.2. Reading the URI Fragment	388
12.11.3. Listening for URI Fragment Changes	389
12.11.4. Supporting Web Crawling	389
12.12. Drag and Drop	390
12.12.1. Handling Drops	391
12.12.2. Dropping Items On a Tree	391
12.12.3. Dropping Items On a Table	393
12.12.4. Accepting Drops	394
12.12.5. Dragging Components	397
12.12.6. Dropping on a Component	397
12.12.7. Dragging Files from Outside the Browser	399
12.13. Logging	399
12.13.1. Logging in Apache Tomcat	399
12.13.2. Logging in Liferay	399
12.13.3. Piping to Log4j using SLF4J	399
12.13.4. Using Logger	400
12.14. JavaScript Interaction	400
12.14.1. Calling JavaScript	401
12.14.2. Handling JavaScript Function Callbacks	401
12.15. Accessing Session-Global Data	402
12.15.1. The Problem	403
12.15.2. Overview of Solutions	403
12.15.3. Passing References Around	403
12.15.4. Overriding <code>attach()</code>	404

12.15.5. ThreadLocal Pattern	404
12.16. Server Push	405
12.16.1. Installing the Push Support	406
12.16.2. Enabling Push for a UI	406
12.16.3. Accessing UI from Another Thread	407
12.16.4. Broadcasting to Other Users	409
12.17. Vaadin CDI Add-on	411
12.17.1. CDI Overview	411
12.17.2. Installing Vaadin CDI Add-on	412
12.17.3. Preparing Application for CDI	413
12.17.4. Injecting a UI with @CDIUI	413
12.17.5. Scopes	414
12.17.6. Deploying CDI UIs and Servlets	416
12.18. Vaadin Spring Add-on	417
12.18.1. Spring Overview	417
12.18.2. Quick Start with Vaadin Spring Boot	419
12.18.3. Installing Vaadin Spring Add-on	419
12.18.4. Preparing Application for Spring	420
12.18.5. Injecting a UI with @SpringUI	420
12.18.6. Scopes	421
12.18.7. Access Control	421
12.18.8. Deploying Spring UIs and Servlets	422
Chapter 13. Portal Integration	425
13.1. Overview	425
13.2. Creating a Generic Portlet in Eclipse	426
13.2.1. Creating a Project with Vaadin Plugin	426
13.3. Developing Vaadin Portlets for Liferay	428
13.3.1. Defining Liferay Profile for Maven	428
13.3.2. Creating a Portlet Project with Maven	430
13.3.3. Creating a Portlet Project in Liferay IDE	432
13.3.4. Removing the Bundled Installation	432
13.3.5. Installing Vaadin Resources	433
13.4. Portlet UI	434
13.5. Deploying to a Portal	436
13.5.1. Portlet Deployment Descriptor	436
13.5.2. Liferay Portlet Descriptor	437
13.5.3. Liferay Display Descriptor	438
13.5.4. Liferay Plugin Package Properties	438
13.5.5. Using a Single Widget Set	439
13.5.6. Building the WAR Package	439
13.5.7. Deploying the WAR Package	440
13.6. Vaadin IPC for Liferay	440
13.6.1. Installing the Add-on	441
13.6.2. Basic Communication	442
13.6.3. Considerations	442
13.6.4. Communication Through Session Attributes	443
13.6.5. Serializing and Encoding Data	444
13.6.6. Communicating with Non-Vaadin Portlets	445
Chapter 14. Client-Side Vaadin Development	447
14.1. Overview	447
14.2. Installing the Client-Side Development Environment	448
14.3. Client-Side Module Descriptor	448

14.3.1. Specifying a Stylesheet	449
14.3.2. Limiting Compilation Targets	449
14.4. Compiling a Client-Side Module	449
14.4.1. Vaadin Compiler Overview	450
14.4.2. Compiling in Eclipse	450
14.4.3. Compiling with Ant	450
14.4.4. Compiling with Maven	450
14.5. Creating a Custom Widget	450
14.5.1. A Basic Widget	450
14.5.2. Using the Widget	451
14.6. Debugging Client-Side Code	452
14.6.1. Launching SuperDevMode	452
14.6.2. Debugging Java Code in Chrome	453
Chapter 15. Client-Side Applications	455
15.1. Overview	455
15.2. Client-Side Module Entry-Point	457
15.2.1. Module Descriptor	457
15.3. Compiling and Running a Client-Side Application	458
15.4. Loading a Client-Side Application	458
Chapter 16. Client-Side Widgets	461
16.1. Overview	461
16.2. GWT Widgets	462
16.3. Vaadin Widgets	462
16.4. Grid	462
16.4.1. Renderers	463
Chapter 17. Integrating with the Server-Side	465
17.1. Overview	466
17.1.1. Project Structure	468
17.1.2. Integrating JavaScript Components	469
17.2. Starting It Simple With Eclipse	469
17.2.1. Creating a Widget	469
17.2.2. Compiling the Widget Set	471
17.3. Creating a Server-Side Component	472
17.3.1. Basic Server-Side Component	472
17.4. Integrating the Two Sides with a Connector	473
17.4.1. A Basic Connector	473
17.4.2. Communication with the Server-Side	473
17.5. Shared State	474
17.5.1. Location of Shared-State Classes	474
17.5.2. Accessing Shared State on Server-Side	474
17.5.3. Handling Shared State in a Connector	475
17.5.4. Handling Property State Changes with @OnStateChange	475
17.5.5. Delegating State Properties to Widget	476
17.5.6. Referring to Components in Shared State	476
17.5.7. Sharing Resources	477
17.6. RPC Calls Between Client- and Server-Side	477
17.6.1. RPC Calls to the Server-Side	478
17.7. Component and UI Extensions	479
17.7.1. Server-Side Extension API	479
17.7.2. Extension Connectors	480
17.8. Styling a Widget	481
17.8.1. Determining the CSS Class	481

17.8.2. Default Stylesheet	482
17.9. Component Containers	482
17.10. Advanced Client-Side Topics	482
17.10.1. Client-Side Processing Phases	482
17.11. Creating Add-ons	483
17.11.1. Exporting Add-on in Eclipse	484
17.12. Migrating from Vaadin 6	485
17.12.1. Quick (and Dirty) Migration	485
17.13. Integrating JavaScript Components and Extensions	486
17.13.1. Example JavaScript Library	486
17.13.2. A Server-Side API for a JavaScript Component	487
17.13.3. Defining a JavaScript Connector	488
17.13.4. RPC from JavaScript to Server-Side	489
Chapter 18. Using Vaadin Add-ons	491
18.1. Overview	491
18.2. Downloading Add-ons from Vaadin Directory	492
18.2.1. Compiling Widget Sets with an Ant Script	492
18.3. Installing Add-ons in Eclipse with Ivy	493
18.4. Using Add-ons in a Maven Project	494
18.4.1. Adding a Dependency	494
18.4.2. Compiling the Project Widget Set	496
18.4.3. Enabling Widget Set Compilation	496
18.5. Installing Commercial Vaadin Add-on Licence	497
18.5.1. Obtaining License Keys	497
18.5.2. Installing License Key in License File	499
18.5.3. Passing License Key as System Property	499
18.6. Troubleshooting	500
Chapter 19. Vaadin Charts	501
19.1. Overview	501
19.1.1. Licensing	504
19.2. Installing Vaadin Charts	504
19.2.1. Maven Dependency	504
19.2.2. Ivy Dependency	505
19.2.3. Installing License Key	505
19.3. Basic Use	506
19.3.1. Basic Chart Configuration	507
19.3.2. Plot Options	508
19.3.3. Chart Data Series	508
19.3.4. Axis Configuration	508
19.3.5. Displaying Multiple Series	509
19.3.6. Mixed Type Charts	510
19.3.7. 3D Charts	511
19.3.8. Chart Themes	514
19.4. Chart Types	515
19.4.1. Line and Spline Charts	515
19.4.2. Area Charts	515
19.4.3. Column and Bar Charts	516
19.4.4. Error Bars	517
19.4.5. Box Plot Charts	518
19.4.6. Scatter Charts	519
19.4.7. Bubble Charts	521
19.4.8. Pie Charts	522

19.4.9. Gauges	524
19.4.10. Solid Gauges	526
19.4.11. Area and Column Range Charts	528
19.4.12. Polar, Wind Rose, and Spiderweb Charts	529
19.4.13. Funnel and Pyramid Charts	529
19.4.14. Waterfall Charts	530
19.4.15. Heat Maps	530
19.4.16. Tree Maps	531
19.4.17. Polygons	531
19.4.18. Flags	531
19.4.19. OHLC and Candlestick Charts	533
19.5. Chart Configuration	535
19.5.1. Plot Options	536
19.5.2. Axes	538
19.5.3. Legend	541
19.5.4. Formatting Labels	542
19.6. Chart Data	543
19.6.1. List Series	543
19.6.2. Generic Data Series	544
19.6.3. Range Series	545
19.6.4. Container Data Series	546
19.6.5. Drill-Down	547
19.7. Advanced Uses	549
19.7.1. Server-Side Rendering and Exporting	549
19.8. Timeline	550
Chapter 20. Vaadin JPAContainer	553
20.1. Overview	553
20.1.1. Java Persistence API	554
20.1.2. JPAContainer Concepts	555
20.1.3. Documentation and Support	555
20.2. Installing	556
20.2.1. Downloading the Package	556
20.2.2. Installation Package Content	556
20.2.3. Downloading with Maven	557
20.2.4. Including Libraries in Your Project	557
20.2.5. Persistence Configuration	557
20.2.6. Troubleshooting	559
20.3. Defining a Domain Model	560
20.3.1. Persistence Metadata	560
20.4. Basic Use of JPAContainer	563
20.4.1. Creating JPAContainer with JPAContainerFactory	563
20.4.2. Creating and Accessing Entities	565
20.4.3. Nested Properties	566
20.4.4. Hierarchical Container	567
20.5. Entity Providers	568
20.5.1. Built-In Entity Providers	568
20.5.2. Using JNDI Entity Providers in JEE6 Environment	570
20.5.3. Entity Providers as Enterprise Beans	570
20.6. Filtering JPAContainer	571
20.7. Querying with the Criteria API	572
20.7.1. Filtering the Query	572
20.7.2. Compatibility	573
20.8. Automatic Form Generation	573

20.8.1. Configuring the Field Factory	573
20.8.2. Using the Field Factory	574
20.8.3. Master-Detail Editor	575
20.9. Using JPAContainer with Hibernate	575
20.9.1. Lazy loading	575
20.9.2. Joins in Hibernate vs EclipseLink	576
Chapter 21. Mobile Applications with TouchKit	577
21.1. Overview	578
21.1.1. TouchKit Demos	580
21.1.2. Licensing	580
21.2. Considerations Regarding Mobile Browsing	580
21.2.1. Mobile Human Interface	580
21.2.2. Bandwidth and Performance	581
21.2.3. Mobile Features	581
21.2.4. Compatibility	581
21.3. Installing Vaadin TouchKit	582
21.4. Importing the Parking Demo	582
21.5. Creating a New TouchKit Project	583
21.5.1. Using the Maven Archetype	583
21.5.2. Starting from a New Eclipse Project	584
21.6. Elements of a TouchKit Application	585
21.6.1. The Servlet Class	585
21.6.2. Defining Servlet and UI with web.xml Deployment Descriptor	586
21.6.3. TouchKit Settings	586
21.6.4. The UI	588
21.6.5. Mobile Widget Set	588
21.6.6. Mobile Theme	588
21.6.7. Using Font Icons	590
21.7. Mobile User Interface Components	591
21.7.1. NavigationView	592
21.7.2. Toolbar	593
21.7.3. NavigationManager	593
21.7.4. NavigationButton	595
21.7.5. Popover	597
21.7.6. SwipeView	601
21.7.7. Switch	602
21.7.8. VerticalComponentGroup	602
21.7.9. HorizontalButtonGroup	604
21.7.10. TabBarView	604
21.7.11. EmailField	605
21.7.12. NumberField	606
21.7.13. UrlField	607
21.8. Advanced Mobile Features	608
21.8.1. Providing a Fallback UI	608
21.8.2. Geolocation	609
21.8.3. Storing Data in the Local Storage	609
21.9. Offline Mode	610
21.9.1. Enabling the Cache Manifest	611
21.9.2. Disabling Offline Mode	611
21.9.3. Configuring Offline Mode	611
21.9.4. The Offline User Interface	612
21.9.5. Sending Data to Server	612
21.9.6. The Offline Theme	612

21.10. Building an Optimized Widget Set	613
21.11. Testing and Debugging on Mobile Devices	613
21.11.1. Debugging	613
Chapter 22. Vaadin Spreadsheet	615
22.1. Overview	615
22.1.1. Features	617
22.1.2. Spreadsheet Demo	618
22.1.3. Requirements	618
22.1.4. Limitations	618
22.1.5. Licensing	618
22.2. Installing Vaadin Spreadsheet	619
22.2.1. Installing License Key	619
22.2.2. Compiling Widget Set	619
22.2.3. Compiling Theme	619
22.2.4. Importing the Demo	619
22.3. Basic Use	620
22.3.1. Creating a Spreadsheet	620
22.3.2. Working with Sheets	621
22.4. Spreadsheet Configuration	621
22.4.1. Spreadsheet Elements	621
22.4.2. Frozen Row and Column Panes	622
22.5. Cell Content and Formatting	622
22.5.1. Cell Formatting	622
22.5.2. Cell Font Style	623
22.5.3. Cell Comments	623
22.5.4. Merging Cells	623
22.5.5. Components in Cells	624
22.5.6. Hyperlinks	624
22.5.7. Popup Buttons in Cells	624
22.6. Context Menus	624
22.6.1. Default Context Menu	624
22.6.2. Custom Context Menus	625
22.7. Tables Within Spreadsheets	625
22.7.1. Creating a Table	625
22.7.2. Filtering With a Table	626
Chapter 23. Vaadin TestBench	627
23.1. Overview	627
23.1.1. Vaadin TestBench in Software Development	628
23.1.2. Features	629
23.1.3. Based on Selenium	630
23.1.4. TestBench Components	630
23.1.5. Requirements	630
23.1.6. Continuous Integration Compatibility	631
23.1.7. Licensing and Trial Period	631
23.2. Quick Start	632
23.2.1. Installing License Key	632
23.2.2. Quick Start with Eclipse	633
23.2.3. Quick Start with Maven	634
23.3. Installing Vaadin TestBench	635
23.3.1. Test Development Setup	635
23.3.2. A Distributed Testing Environment	636
23.3.3. Installation Package Contents	637

23.3.4. TestBench Demo	637
23.3.5. Installing Browser Drivers	639
23.3.6. Test Node Configuration	639
23.4. Developing JUnit Tests	640
23.4.1. Basic Test Case Structure	640
23.4.2. Running JUnit Tests in Eclipse	643
23.5. Creating a Test Case	644
23.5.1. Test Setup	644
23.5.2. Basic Test Case Structure	644
23.5.3. Creating and Closing a Web Driver	645
23.6. Querying Elements	647
23.6.1. Generating Queries with Debug Window	647
23.6.2. Querying Elements by Component Type (\$)	648
23.6.3. Non-Recursive Component Queries (\$\$)	648
23.6.4. Element Classes	648
23.6.5. ElementQuery Objects	649
23.6.6. Query Terminators	649
23.7. Element Selectors	649
23.7.1. Finding by ID	650
23.7.2. Finding by CSS Class	650
23.8. Special Testing Topics	650
23.8.1. Waiting for Vaadin	650
23.8.2. Testing Tooltips	651
23.8.3. Scrolling	651
23.8.4. Testing Notifications	652
23.8.5. Testing Context Menus	652
23.8.6. Profiling Test Execution Time	653
23.9. Creating Maintainable Tests	654
23.9.1. Increasing Selector Robustness	655
23.9.2. The Page Object Pattern	656
23.10. Taking and Comparing Screenshots	658
23.10.1. Screenshot Parameters	658
23.10.2. Taking Screenshots on Failure	659
23.10.3. Taking Screenshots for Comparison	659
23.10.4. Practices for Handling Screenshots	662
23.10.5. Known Compatibility Problems	662
23.11. Running Tests	662
23.11.1. Running Tests with Ant	662
23.11.2. Running Tests with Maven	664
23.12. Running Tests in a Distributed Environment	665
23.12.1. Running Tests Remotely	665
23.12.2. Starting the Hub	666
23.12.3. Node Service Configuration	666
23.12.4. Starting a Grid Node	668
23.12.5. Mobile Testing	669
23.13. Parallel Execution of Tests	669
23.13.1. Local Parallel Execution	670
23.13.2. Multi-Browser Execution in a Grid	670
23.14. Headless Testing	671
23.14.1. Basic Setup for Running Headless Tests	671
23.14.2. Running Headless Tests in a Distributed Environment	671
23.15. Behaviour-Driven Development	672
23.16. Integration Testing with Maven	673

23.16.1. Project Structure	673
23.16.2. Overview of Lifecycle	673
23.16.3. Overview of Configuration	674
23.16.4. Vaadin Plugin Configuration	674
23.16.5. Configuring Integration Testing	674
23.16.6. Configuring Test Server	675
23.17. Known Issues	676
23.17.1. Running Firefox Tests on Mac OS X	676
Index	677

Preface

This book provides an overview of the Vaadin Framework and covers the most important topics that you might encounter while developing applications with it. The book is a compilation of the most important documentation available in Vaadin Docs at vaadin.com/docs. A more detailed documentation of the individual classes, interfaces, and methods is given in the Vaadin API Reference.

This edition mostly covers Vaadin Framework 7.6 released in 2016. The Volume 2 covers the latest versions of Vaadin Pro Tools.

Writing this manual is an ongoing work and it is rarely completely up-to-date with the quick-evolving product. Some features may not be included in this book yet. For the most current documentation, please see the Vaadin Docs available at vaadin.com/docs. You can also find PDF and EPUB versions of the book there. You may find the other versions more easily searchable than the printed book. The index in the book is incomplete and will be expanded later. The web edition also has some additional technical content, such as some example code and additional sections that you may need when actually doing development. The purpose of the slightly abridged print edition is more to be an introductory textbook to Vaadin, and still fit in your pocket.

1. Who is This Book For?

This book is intended for software developers who use or are considering to use Vaadin to develop web applications.

The book assumes that you have some experience with programming in Java. If not, it is at least as easy to begin learning Java with Vaadin as with any other UI framework. No knowledge of AJAX is needed as it is well hidden from the developer.

You may have used some desktop-oriented user interface frameworks for Java, such as AWT, Swing, or SWT, or a library such as Qt for C++. Such knowledge is useful for understanding the scope of Vaadin, the event-driven programming model, and other common concepts of UI frameworks, but not necessary.

If you do not have a web graphics designer at hand, knowing the basics of HTML and CSS can help so that you can develop presentation themes for your application. A brief introduction to CSS is provided. Knowledge of Google Web Toolkit (GWT) may be useful if you develop or integrate new client-side components.

2. Organization of This Book

The Book of Vaadin gives an introduction to what Vaadin is and how you use it to develop web applications.

Volume 1

Chapter 1, *Introduction*

The chapter gives an introduction to the application architecture supported by Vaadin, the core design ideas behind the framework, and some historical background.

Chapter 2, *Installing the Development Toolchain*

This chapter gives instructions for installing a toolchain for developing Vaadin applications. A toolchain typically includes an IDE, a Vaadin plugin for the IDE, and a development server. We cover the Eclipse IDE, NetBeans IDE, and IntelliJ IDEA. After this, you should have all the basic tools set up.

Chapter 3, *Creating a Vaadin Application*

This chapter gives practical instructions for creating a Vaadin application project in an IDE or otherwise, and for running it in an integrated development server.

Chapter 4, *Architecture*

This chapter gives an introduction to the architecture of Vaadin and its major technologies, including AJAX, Google Web Toolkit, and event-driven programming.

Chapter 5, *Writing a Server-Side Web Application*

This chapter gives all the practical knowledge required for creating applications with Vaadin, such as window management, application lifecycle, deployment in a servlet container, and handling events, errors, and resources.

Chapter 6, *User Interface Components*

This chapter gives the basic usage documentation for all the (non-layout) user interface components in Vaadin and their most significant features. The component sections include examples for using each component, as well as for styling with CSS/Sass.

Chapter 7, *Managing Layout*

This chapter describes the layout components, which are used for managing the layout of the user interface, just like in any desktop application frameworks.

Chapter 8, *Vaadin Designer*

This chapter gives instructions for using Vaadin Designer, a visual tool for the Eclipse IDE for creating composite designs, such as for UIs, views, or other composites.

Volume 2:

Chapter 9, *Themes*

This chapter gives an introduction to Cascading Style Sheets (CSS) and Sass and explains how you can use them to build custom visual themes for your application.

Chapter 10, *Binding Components to Data*

This chapter gives an overview of the built-in data model of Vaadin, consisting of properties, items, and containers.

Chapter 11, *Vaadin SQLContainer*

This chapter gives documentation for the SQLContainer, which allows binding Vaadin components to SQL queries.

Chapter 12, *Advanced Web Application Topics*

This chapter provides many special topics that are commonly needed in applications, such as opening new browser windows, embedding applications in regular web pages, low-level management of resources, shortcut keys, debugging, etc.

Chapter 13, *Portal Integration*

This chapter describes the development of Vaadin applications as portlets which you can deploy to any portal supporting Java Portlet API 2.0 (JSR-286). The chapter also describes the special support for Liferay and the Control Panel, IPC, and WSRP add-ons.

Chapter 14, *Client-Side Vaadin Development*

This chapter gives an introduction to creating and developing client-side applications and widgets, including installation, compilation, and debugging.

Chapter 15, *Client-Side Applications*

This chapter describes how to develop client-side applications and how to integrate them with a back-end service.

Chapter 16, *Client-Side Widgets*

This chapter describes the built-in widgets (client-side components) available for client-side development. The built-in widgets include Google Web Toolkit widgets as well as Vaadin widgets.

Chapter 17, *Integrating with the Server-Side*

This chapter describes how to integrate client-side widgets with their server-side counterparts for the purpose of creating new server-side components. The chapter also covers integrating JavaScript components.

Chapter 18, *Using Vaadin Add-ons*

This chapter gives instructions for downloading and installing add-on components from the Vaadin Directory.

Chapter 19, *Vaadin Charts*

This chapter documents the use of the Vaadin Charts add-on component for interactive charting with many diagram types. The add-on includes the Chart and Timeline components.

Chapter 20, *Vaadin JPAContainer*

This chapter gives documentation of the JPAContainer add-on, which allows binding Vaadin components directly to relational and other databases using Java Persistence API (JPA).

Chapter 21, *Mobile Applications with TouchKit*

This chapter gives examples and reference documentation for using the Vaadin TouchKit add-on for developing mobile applications.

Chapter 22, *Vaadin Spreadsheet*

This chapter gives documentation of the Vaadin Spreadsheet add-on, which provides a Microsoft Excel compatible spreadsheet component.

Chapter 23, *Vaadin TestBench*

This chapter gives the complete documentation of using the Vaadin TestBench tool for recording and executing user interface regression tests of Vaadin applications.

3. Supplementary Material

The Vaadin websites offer plenty of material that can help you understand what Vaadin is, what you can do with it, and how you can do it.

Demo Applications

The most important demo application for Vaadin is the Sampler, which demonstrates the use of all basic components and features. You can run it on-line at <http://demo.vaadin.com/> or download it as a WAR from the Vaadin download page.

Refcard

The six-page DZone Refcard gives an overview to application development with Vaadin. It includes a diagram of the user interface and data binding classes and interfaces. You can find more information about it at <https://vaadin.com/refcard>.

Address Book Tutorial

The Address Book is a sample application accompanied with a tutorial that gives detailed step-by-step instructions for creating a real-life web application with Vaadin. You can find the tutorial from the product website.

Developer's Website

Vaadin Developer's Site at <http://dev.vaadin.com/> provides various online resources, such as the ticket system, a development wiki, source repositories, activity timeline, development milestones, and so on.

The wiki provides instructions for developers, especially for those who wish to check-out and compile Vaadin itself from the source repository. The technical articles deal with integration of Vaadin applications with various systems, such as JSP, Maven, Spring, Hibernate, and portals. The wiki also provides answers to Frequently Asked Questions.

Online Documentation

You can read this book online at <http://vaadin.com/book>. Lots of additional material, including technical HOWTOs, answers to Frequently Asked Questions and other documentation is also available on Vaadin web-site.

4. Support

Stuck with a problem? No need to lose your hair over it, the Vaadin Framework developer community and the Vaadin company offer support to all of your needs.

Community Support Forum

You can find the user and developer community forum at <http://vaadin.com/forum>. Please use the forum to discuss any problems you might encounter, wishes for features, and so on. The answer to your problems may already lie in the forum archives, so searching the discussions is always the best way to begin.

Report Bugs

If you have found a possible bug in Vaadin, the demo applications, or the documentation, please report it by filing a ticket at the Vaadin developer's site at <http://dev.vaadin.com/>. You may want to check the existing tickets before filing a new one. You can make a ticket to make a request for a new feature as well, or to suggest modifications to an existing feature.

Commercial Support

Vaadin offers full commercial support and training services for the Vaadin Framework and related products. Read more about the commercial products at <http://vaadin.com/pro> for details.

5. About the Editor

Marko Grönroos is a professional writer and software developer working at Vaadin Ltd in Turku, Finland. He has been involved in web application development since 1994 and has worked on several application development frameworks in C, C++, and Java. He has been active in many

open source software projects and holds an M.Sc. degree in Computer Science from the University of Turku.

6. Acknowledgements

Much of the book is the result of close work within the development team at Vaadin Ltd. Joonas Lehtinen, CEO of Vaadin Ltd, wrote the first outline of the book, which became the basis for the first two chapters. Since then, Marko Grönroos was primary author for some time and is now working as the editor. The development teams have contributed several passages, answered numerous technical questions, reviewed the manual, and made many corrections.

The contributors are (in rough chronological order):

- Joonas Lehtinen
- Jani Laakso
- Marko Grönroos
- Jouni Koivuviita
- Matti Tahvonen
- Artur Signell
- Marc Englund
- Henri Sara
- Jonatan Kronqvist
- Mikael Grankvist (TestBench)
- Teppo Kurki (SQLContainer)
- Tomi Virtanen (Calendar)
- Risto Yrjänä (Calendar)
- John Ahlroos (Timeline)
- Petter Holmström (JPAContainer)
- Leif Åstrand
- Guillermo Alvarez (Charts)

7. About Vaadin Ltd

Vaadin Ltd is a Finnish software company specializing in the design and development of Rich Internet Applications. The company offers planning, implementation, and support services for the software projects of its customers, as well as sub-contract software development. Vaadin Framework and Vaadin Core Elements are the prominent open source products of the company, for which it provides premium professional add-on tools, commercial development, and support services.

Chapter 1

Introduction

1.1. Overview	25
1.2. Example Application Walkthrough	27
1.3. Support for the Eclipse IDE	28
1.4. Goals and Philosophy	29
1.5. Background	29

This chapter gives a brief introduction to software development with Vaadin. We also try to give some insight about the design philosophy behind Vaadin and its history.

1.1. Overview

Vaadin Framework is a Java web application development framework that is designed to make creation and maintenance of high quality web-based user interfaces easy. Vaadin supports two different programming models: server-side and client-side. The server-driven programming model is the more powerful one. It lets you forget the web and program user interfaces much like you would program a desktop application with conventional Java toolkits such as AWT, Swing, or SWT. But easier.

While traditional web programming is a fun way to spend your time learning new web technologies, you probably want to be productive and concentrate on the application logic. The server-side Vaadin framework takes care of managing the user interface in the browser and the *AJAX* communications between the browser and the server. With the Vaadin approach, you do not need to learn and deal directly with browser technologies, such as HTML or JavaScript.

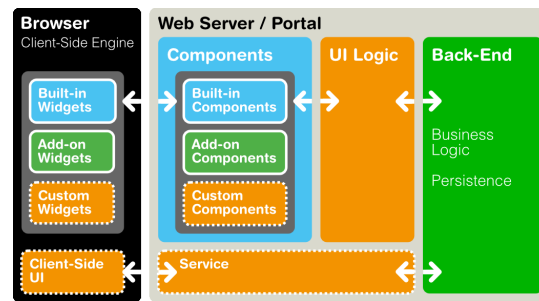
Figure 1.1. Vaadin Application Architecture

Figure 1.1, “Vaadin Application Architecture” illustrates the basic architectures of web applications made with Vaadin. The server-side application architecture consists of the *server-side framework* and a *client-side engine*. The engine runs in the browser as JavaScript code, rendering the user interface, and delivering user interaction to the server. The UI logic of an application runs as a Java Servlet in a Java application server.

As the client-side engine is executed as JavaScript in the browser, no browser plugins are needed for using applications made with Vaadin. This gives it an edge over frameworks based on Flash, Java Applets, or other plugins. Vaadin relies on the support of Google Web Toolkit for a wide range of browsers, so that the developer does not need to worry about browser support.

Because HTML, JavaScript, and other browser technologies are essentially invisible to the application logic, you can think of the web browser as only a thin client platform. A thin client displays the user interface and communicates user events to the server at a low level. The control logic of the user interface runs on a Java-based web server, together with your business logic. By contrast, a normal client-server architecture with a dedicated client application would include a lot of application specific communications between the client and the server. Essentially removing the user interface tier from the application architecture makes our approach a very effective one.

Behind the server-driven development model, Vaadin makes the best use of AJAX (*Asynchronous JavaScript and XML*, see Section 4.2.3, “AJAX” for a description) techniques that make it possible to create Rich Internet Applications (RIA) that are as responsive and interactive as desktop applications.

In addition to the server-side Java application development, you can develop on the client-side by making new widgets in Java, and even pure client-side applications that run solely in the browser. The Vaadin client-side framework includes Google Web Toolkit (GWT), which provides a compiler from Java to the JavaScript that runs in the browser, as well as a full-featured user interface framework. With this approach, Vaadin is pure Java on both sides.

Vaadin uses a client-side engine for rendering the user interface of a server-side application in the browser. All the client-server communications are hidden well under the hood. Vaadin is designed to be extensible, and you can indeed use any 3rd-party widgets easily, in addition to the component repertoire offered in Vaadin. In fact, you can find hundreds of add-ons in the Vaadin Directory.

Vaadin allows flexible separation between the appearance, structure, and interaction logic of the user interface. You can design the layouts either programmatically or declaratively, at the level of your choosing. The final appearance is defined in *themes* in CSS or Sass, as described in Chapter 9, *Themes*.

We hope that this is enough about the basic architecture and features of Vaadin for now. You can read more about it later in Chapter 4, *Architecture*, or jump straight to more practical things in Chapter 5, *Writing a Server-Side Web Application*.

1.2. Example Application Walkthrough

Let us follow the long tradition of first saying "Hello World!" when learning a new programming framework. First, using the primary server-side API.

```
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;

@Title("My UI")
@Theme("valo")
public class HelloWorld extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Create the content root layout for the UI
        VerticalLayout content = new VerticalLayout();
        setContent(content);

        // Display the greeting
        content.addComponent(new Label("Hello World!"));

        // Have a clickable button
        content.addComponent(new Button("Push Me!",
            new ClickListener() {
                @Override
                public void buttonClick(ClickEvent e) {
                    Notification.show("Pushed!");
                }
            }
        ));
    }
}
```

A Vaadin application has one or more *UIs* that extend the **com.vaadin.ui.UI** class. A UI is a part of the web page in which the Vaadin application runs. An application can have multiple UIs in the same page, especially in portals, or in different windows or tabs. A UI is associated with a user session, and a session is created for each user who uses the application. In the context of our Hello World UI, it is sufficient to know that the underlying session is created when the user first accesses the application by opening the page, and the `init()` method is invoked at that time.

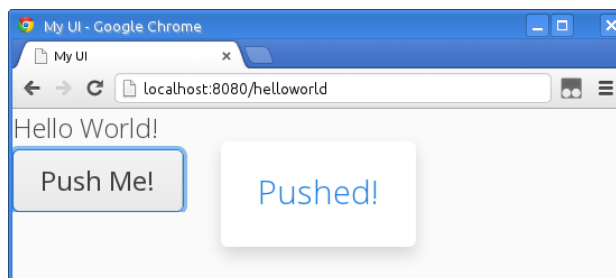
The page title, which is shown in the caption of the browser window or tab, is defined with an annotation. The example uses a layout component as the root content of the UI, as that is the case with most Vaadin applications, which normally have more than one component. It then creates a new **Label** user interface component, which displays simple text, and sets the text to "Hello World!". The label is added to the layout.

The example also shows how to create a button and handle button click events. Event handling is described in Section 4.4, "Events and Listeners" and on the practical side in Section 5.4, "Handling Events with Listeners". In addition to listeners, in Java 8 you can handle events with lambda expressions, which simplifies the handler code significantly.

```
content.addComponent(new Button("Push Me!",
    event -> Notification.show("Pushed!")));
```

The result of the Hello World application, when opened in a browser, is shown in Figure 1.2, “Hello World Application”.

Figure 1.2. Hello World Application



To run a program, you need to package it as a web application WAR package and deploy it to a server, as explained in Section 5.9, “Deploying an Application”. During development, you typically deploy to an application server integrated with the IDE.

Developing a pure client-side application, you could write a Hello World just as easily, and also in Java:

```
public class HelloWorld implements EntryPoint {
    @Override
    public void onModuleLoad() {
        RootPanel.get().add(new Label("Hello, world!"));
    }
}
```

We do not set the title here, because it is usually defined in the HTML page in which the code is executed. The application would be compiled into JavaScript with the Vaadin Client Compiler (or GWT Compiler). It is more typical, however, to write client-side widgets, which you can then use from a server-side Vaadin application. For more information regarding client-side development, see Chapter 14, *Client-Side Vaadin Development*.

1.3. Support for the Eclipse IDE

While Vaadin is not bound to any specific IDE, and you can in fact easily use it without any IDE altogether, we provide special support for the Eclipse IDE, which has become the most used environment for Java development.

Vaadin Plug-in for Eclipse helps you in:

- creating new Vaadin projects,
- creating custom themes,
- creating custom client-side widgets, and
- easily upgrading to a newer version of the Vaadin library.

Using the Vaadin plug-in for Eclipse is the recommended way of installing Vaadin for development. Downloading the installation package that contains the JARs or defining Vaadin as a Maven dependency is also possible.

Vaadin Designer is a commercial Eclipse plug-in that enables visual editing of Vaadin UIs and composites. See Chapter 8, *Vaadin Designer* for its complete reference.

Installation of the Eclipse IDE and the plug-in is covered in Section 2.5, “Installing the Eclipse IDE and Plug-in”. The creation of a new Vaadin project using the plug-in is covered in Section 3.4.1, “Creating a Maven Project”. See Section 9.6, “Creating a Theme in Eclipse” and Section 17.2, “Starting It Simple With Eclipse” for instructions on using the different features of the plug-in.

1.4. Goals and Philosophy

Simply put, Vaadin’s ambition is to be the best possible tool when it comes to creating web user interfaces for business applications. It is easy to adopt, as it is designed to support both entry-level and advanced programmers, as well as usability experts and graphic designers.

When designing Vaadin, we have followed the philosophy inscribed in the following rules.

1.4.1. Right tool for the right purpose

Because our goals are high, the focus must be clear. Vaadin is designed for creating web applications. It is not designed for creating websites or advertisement demos. You may find, for example, JSP/JSF or Flash more suitable for such purposes.

1.4.2. Simplicity and maintainability

We have chosen to emphasize robustness, simplicity, and maintainability. This involves following the well-established best practices in user interface frameworks and ensuring that our implementation represents an ideal solution for its purpose without clutter or bloat.

1.4.3. Choice between declarative and dynamic UIs

The Web is inherently document-centered and very much bound to the declarative presentation of user interfaces. Vaadin allows for declarative designs of views, layouts, and even entire UIs. Vaadin Designer enables creating such designs visually. Nevertheless, the programmatic approach by building the UIs from Java components frees the programmer from its limitations. To create highly dynamic views, it is more natural to create them by programming.

1.4.4. Tools should not limit your work

There should not be any limits on what you can do with the framework: if for some reason the user interface components do not support what you need to achieve, it must be easy to add new ones to your application. When you need to create new components, the role of the framework is critical: it makes it easy to create re-usable components that are easy to maintain.

1.5. Background

The Vaadin Framework was not written overnight. After working with web user interfaces since the beginning of the Web, a group of developers got together in 2000 to form IT Mill. The team had a desire to develop a new programming paradigm that would support the creation of real user interfaces for real applications using a real programming language.

The library was originally called Millstone Library. The first version was used in a large production application that IT Mill designed and implemented for an international pharmaceutical company.

IT Mill made the application already in the year 2001 and it is still in use. Since then, the company has produced dozens of large business applications with the library and it has proven its ability to solve hard problems easily.

The next generation of the library, IT Mill Toolkit Release 4, was released in 2006. It introduced an entirely new AJAX-based presentation engine. This allowed the development of AJAX applications without the need to worry about communications between the client and the server.

1.5.1. Release 5 Into the Open

IT Mill Toolkit 5, released initially at the end of 2007, took a significant step further into AJAX. The client-side rendering of the user interface was completely rewritten using GWT, the Google Web Toolkit.

IT Mill Toolkit 5 introduced many significant improvements both in the server-side API and in the functionality. Rewriting the Client-Side Engine with GWT allowed the use of Java both on the client and the server-side. The transition from JavaScript to GWT made the development and integration of custom components and customization of existing components much easier than before, and it also allows easy integration of existing GWT components. The adoption of GWT on the client-side did not, by itself, cause any changes in the server-side API, because GWT is a browser technology that is hidden well behind the API. Also theming was completely revised in IT Mill Toolkit 5.

The Release 5 was published under the Apache License 2, an unrestrictive open source license, to create faster expansion of the user base and to make the formation of a developer community possible.

1.5.2. Birth of Vaadin Release 6

IT Mill Toolkit was renamed as *Vaadin Framework*, or Vaadin in short, in spring 2009. Later IT Mill, the company, was also renamed as Vaadin Ltd. Vaadin means an adult female semi-domesticated mountain reindeer in Finnish.

With Vaadin 6, the number of developers using the framework exploded. Together with the release, the Vaadin Plugin for Eclipse was released, helping the creation of Vaadin projects. The introduction of Vaadin Directory in early 2010 gave it a further boost, as the number of available components multiplied almost overnight. Many of the originally experimental components have since then matured and are now used by thousands of developers. In 2013, we are seeing tremendous growth in the ecosystem around Vaadin. The size of the user community, at least if measured by forum activity, has already gone past the competing server-side frameworks and even GWT.

1.5.3. The Major Revision with Vaadin 7

Vaadin 7 was a major revision that changed the Vaadin API much more than Vaadin 6 did. It is certainly more web-oriented than Vaadin 6 was. We are doing everything we can to help Vaadin rise high in the web universe. Some of this work is easy and almost routine - fixing bugs and implementing features. But going higher also requires standing firmer. That was one of the aims of Vaadin 7 - redesigning the product so that the new architecture enables Vaadin to reach over many long-standing challenges. Many of the changes required breaking API compatibility with Vaadin 6, especially in the client-side, but they are made with a strong desire to avoid carrying unnecessary legacy burden far into the future.

Inclusion of the Google Web Toolkit in Vaadin 7 was a significant development, as it meant that Vaadin now provides support for GWT as well. When Google opened the GWT development in summer 2012, Vaadin (the company) joined the new GWT steering committee. As a member of the committee, Vaadin can work towards the success of GWT as a foundation of the Java web development community.

Chapter 2

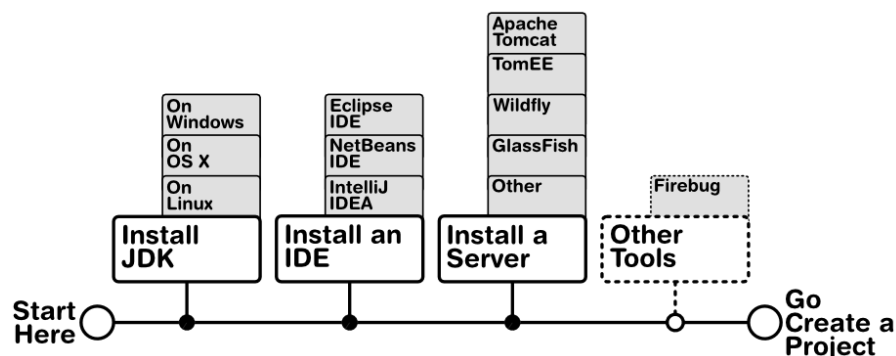
Installing the Development Toolchain

2.1. Overview	33
2.2. A Reference Toolchain	34
2.3. Installing Java SDK	35
2.4. Installing a Web Server	36
2.5. Installing the Eclipse IDE and Plugin	37
2.6. Installing the NetBeans IDE and Plugin	43
2.7. Installing and Configuring IntelliJ IDEA	44
2.8. Installing Other Tools	46

This chapter gives practical instructions for installing the development tools.

2.1. Overview

You can develop Vaadin applications in essentially any development environment that has the Java SDK and deploys to a Java Servlet container. You can use Vaadin with any Java IDE or no IDE at all. Vaadin has special support for the Eclipse and NetBeans IDEs, as well as for IntelliJ IDEA.

Figure 2.1. Vaadin installation steps

Managing Vaadin and other Java libraries can get tedious to do manually, so using a build system that manages dependencies automatically is advised. Vaadin is distributed in the Maven central repository, and can be used with any build or dependency management system that can access Maven repositories, such as Ivy or Gradle, in addition to Maven.

Vaadin has a multitude of installation options for different IDEs and dependency managers. You can also install it from an installation package:

- With the Eclipse IDE, use the Vaadin Plugin for Eclipse, as described in Vaadin Plugin for Eclipse
- With the Vaadin plugin for NetBeans IDE (Section 3.6, “Creating a Project with the NetBeans IDE”) or IntelliJ IDEA
- With Maven, Ivy, Gradle, or other Maven-compatible dependency manager, under Eclipse, NetBeans, IDEA, or using command-line, as described in Section 3.5, “Creating a Project with Maven”
- From installation package without dependency management, as described in Section 3.8, “Vaadin Installation Package”

2.2. A Reference Toolchain

This section presents a reference development environment. Vaadin supports a wide variety of tools, so you can use any IDE for writing the code, almost any Java web server for deploying the application, most web browsers for using it, and any operating system platform supported by Java.

In this example, we use the following toolchain:

- Windows, Linux, or Mac OS X
- Oracle Java SE 8 (Java 6 or newer is required)
- Eclipse IDE for Java EE Developers
- Apache Tomcat 8.0 (Core)
- Mozilla Firefox browser
- Firebug debug tool (optional)

- Vaadin Framework

The above reference toolchain is a good choice of tools, but you can use almost any tools you are comfortable with.

We recommend using Java 8 for Vaadin development, but you need to make sure that your entire toolchain supports it.

Figure 2.2. Development Toolchain and Process

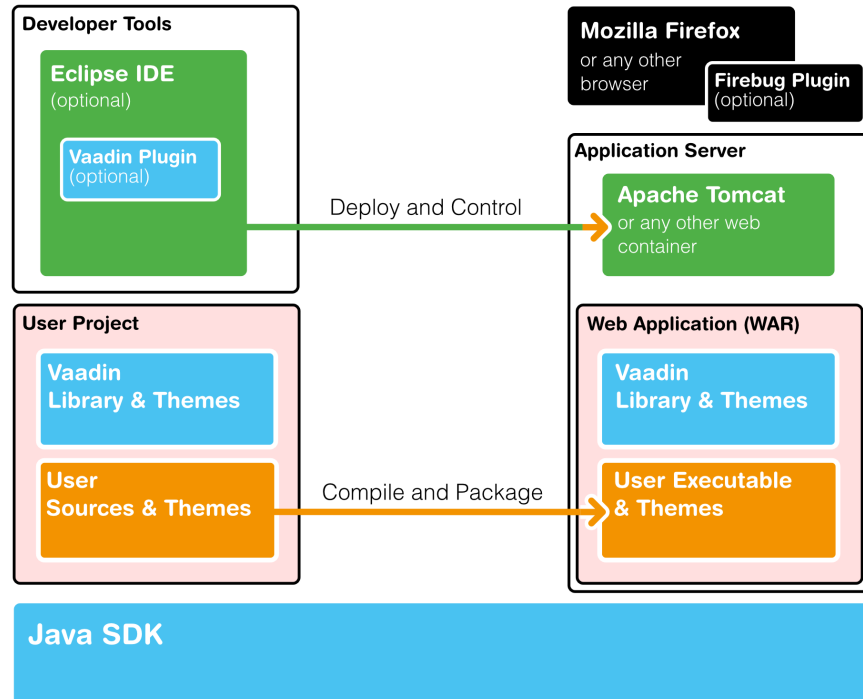


Figure 2.2, “Development Toolchain and Process” illustrates the development toolchain. You develop your application as an Eclipse project. The project must include, in addition to your source code, the Vaadin libraries. It can also include project-specific themes.

You need to compile and deploy a project to a web container before you can use it. You can deploy a project through the Web Tools Platform (WTP) for Eclipse (included in the Eclipse EE package), which allows automatic deployment of web applications from Eclipse. You can also deploy a project manually, by creating a web application archive (WAR) and deploying it to the web container.

2.3. Installing Java SDK

A Java SDK is required by Vaadin and also by any of the Java IDEs. Vaadin is compatible with Java 1.6 and later editions, but we recommend using Java 8 for Vaadin development. Java EE 7 is required for proper server push support with WebSockets.

2.3.1. Windows

Follow the following steps:

1. Download Oracle Java SE 8.0 from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
2. Install the Java SDK by running the installer. The default options are fine.

2.3.2. Linux / UNIX

Most Linux systems either have JDK preinstalled or allow installing it through a package management system. Notice however that they have OpenJDK as the default Java implementation. While it is known to have worked with Vaadin and possibly also with the development toolchain, we do not especially support it.

Regarding OS X, notice that JDK 1.6 or newer is included in OS X 10.6 and newer.

Otherwise:

1. Download Oracle Java SE 8.0 from <http://www.oracle.com/technetwork/java/javase/downloads/>
2. Decompress it under a suitable base directory, such as `/opt`. For example, for Java SDK, enter (either as root or with **sudo** in Linux):

```
# cd /opt
# sh <path>/jdk-<version>.bin
```

and follow the instructions in the installer.

3. Set up the `JAVA_HOME` environment variable to point to the Java installation directory. Also, include the `$JAVA_HOME/bin` in the `PATH`. How you do that varies by the UNIX variant. For example, in Linux and using the Bash shell, you would add lines such as the following to the `.bashrc` or `.profile` script in your home directory:

```
export JAVA_HOME=/opt/jdk1.8.0_31
export PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
```

You could also make the setting system-wide in a file such as `/etc/bash.bashrc`, `/etc/profile`, or an equivalent file. If you install Apache Ant or Maven, you may also want to set up those in the path.

Settings done in a `bashrc` file require that you open a new shell window. Settings done in a `profile` file require that you log in into the system. You can, of course, also give the commands in the current shell.

2.4. Installing a Web Server

You can run Vaadin applications in any Java servlet container that supports at least Servlet API 2.4. However, a server supporting Servlet API 3.0 is recommended. It is required for using Vaadin CDI, for which also a CDI container is required, a standard feature in Java EE 6 or newer servers. It is also required by the Vaadin Spring add-on. Server push can benefit from using communication modes, such as WebSocket, enabled by features in some latest servers. For Java EE containers, at least Wildfly, Glassfish, and Apache TomEE Web Profile are recommended.

Also, if you use Java 8 for Vaadin development, you need to make sure that the server supports it.

Some Java IDEs have server integration, so we describe installation of the server before the IDEs.

Some IDE bundles also include a development server; for example, NetBeans IDE includes GlassFish and Apache Tomcat.

You can also opt to install a development server from a Maven dependency and let the IDE control it through Maven executions.

2.4.1. Installing Apache Tomcat

Apache Tomcat is a lightweight Java web server suitable for both development and production. There are many ways to install it, but here we simply decompress the installation package.

Apache Tomcat should be installed with user permissions. During development, you will be running Eclipse or some other IDE with user permissions, but deploying web applications to a Tomcat server that is installed system-wide requires administrator or root permissions.

1. Download the installation package:

Apache Tomcat 8.0 (Core Binary Distribution) from <http://tomcat.apache.org/>

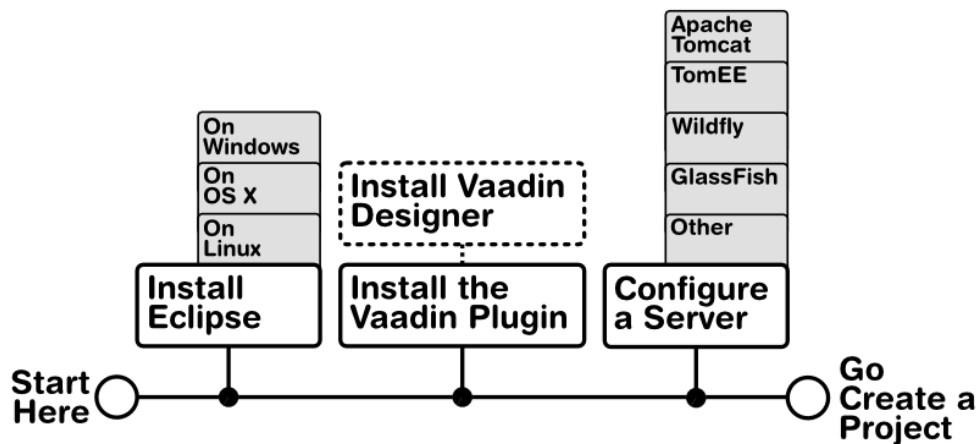
2. Decompress Apache Tomcat package to a suitable target directory, such as `C:\dev` (Windows) or `/opt` (Linux or Mac OS X). The Apache Tomcat home directory will be `C:\dev\apache-tomcat-8.0.x` or `/opt/apache-tomcat-8.0.x`, respectively.

Do not start the server. If you use an IDE integration, the IDE will control starting and stopping the server.

2.5. Installing the Eclipse IDE and Plugin

If you are using the Eclipse IDE, using the Vaadin Plugin for Eclipse helps greatly. The plugin includes wizards for creating new Vaadin-based projects, themes, and client-side widgets and widget sets. Notice that you can also create Vaadin projects as Maven projects in Eclipse.

Using Eclipse IDE for Vaadin development requires installing the IDE itself and the Vaadin Plugin for Eclipse. You are advised to also configure a web server in Eclipse. You can then use the server for running the Vaadin applications that you create.

Figure 2.3. Installation of the Eclipse IDE toolchain

Vaadin Designer is a visual design tool for professional developers. It allows for easy creation of declarative designs. It is also good as a sketching tool, as well as an easy way to learn about Vaadin components and layouts.

Once you have installed the Eclipse IDE and the plug-in, you can create a Vaadin application project as described in Section 3.4, “Creating and Running a Project in Eclipse”.

2.5.1. Installing the Eclipse IDE

Windows

1. Download the Eclipse IDE for Java EE Developers from <http://www.eclipse.org/downloads/>
2. Decompress the Eclipse IDE package to a suitable directory. You are free to select any directory and to use any ZIP decompressor, but in this example we decompress the ZIP file by just double-clicking it and selecting “Extract all files” task from Windows compressed folder task. In our installation example, we use `C:\dev` as the target directory.

Eclipse is now installed in `C:\dev\eclipse`. You can start it from there by double clicking `eclipse.exe`.

Linux / OS X / UNIX

We recommend that you install Eclipse manually in Linux and other UNIX variants. They may have it available from a package repository, but using such an installation may cause problems with installing plug-ins.

You can install Eclipse as follows:

1. Download Eclipse IDE for Java EE Developers from <http://www.eclipse.org/downloads/>
2. Decompress the Eclipse package into a suitable base directory. It is important to make sure that there is no old Eclipse installation in the target directory. Installing a new version on top of an old one probably renders Eclipse unusable.

3. Eclipse should normally be installed as a regular user, which makes installation of plug-ins easier. Eclipse also stores some user settings in the installation directory.

To install the package, enter:

```
$ tar xzf <path>/eclipse-jee-<version>.tar.gz
```

This will extract the package to a subdirectory with the name `eclipse`.

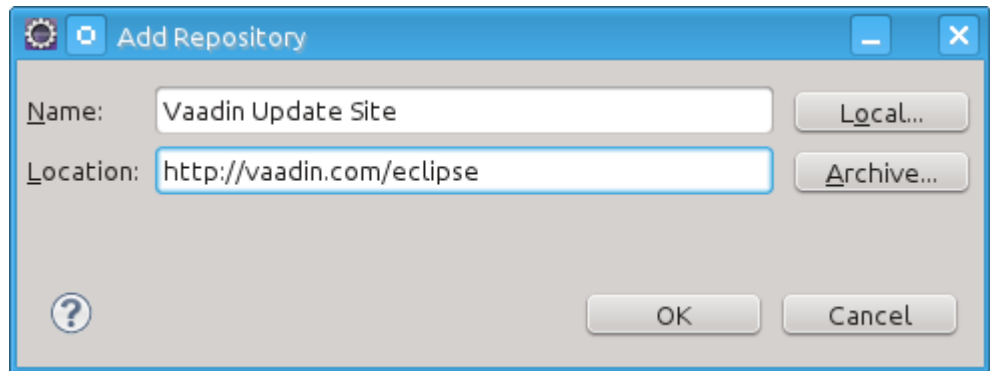
4. If you wish to enable starting Eclipse from command-line, you need to add the Eclipse installation directory to your system or user `PATH`, or make a symbolic link or script to point to the executable.

An alternative to the above procedure would be to use an Eclipse version available through the package management system of your operating system. It is, however, *not recommended*, because you will need write access to the Eclipse installation directory to install Eclipse plugins, and you may face incompatibility issues with Eclipse plugins installed by the package management of the operating system.

2.5.2. Installing the Vaadin Plugin

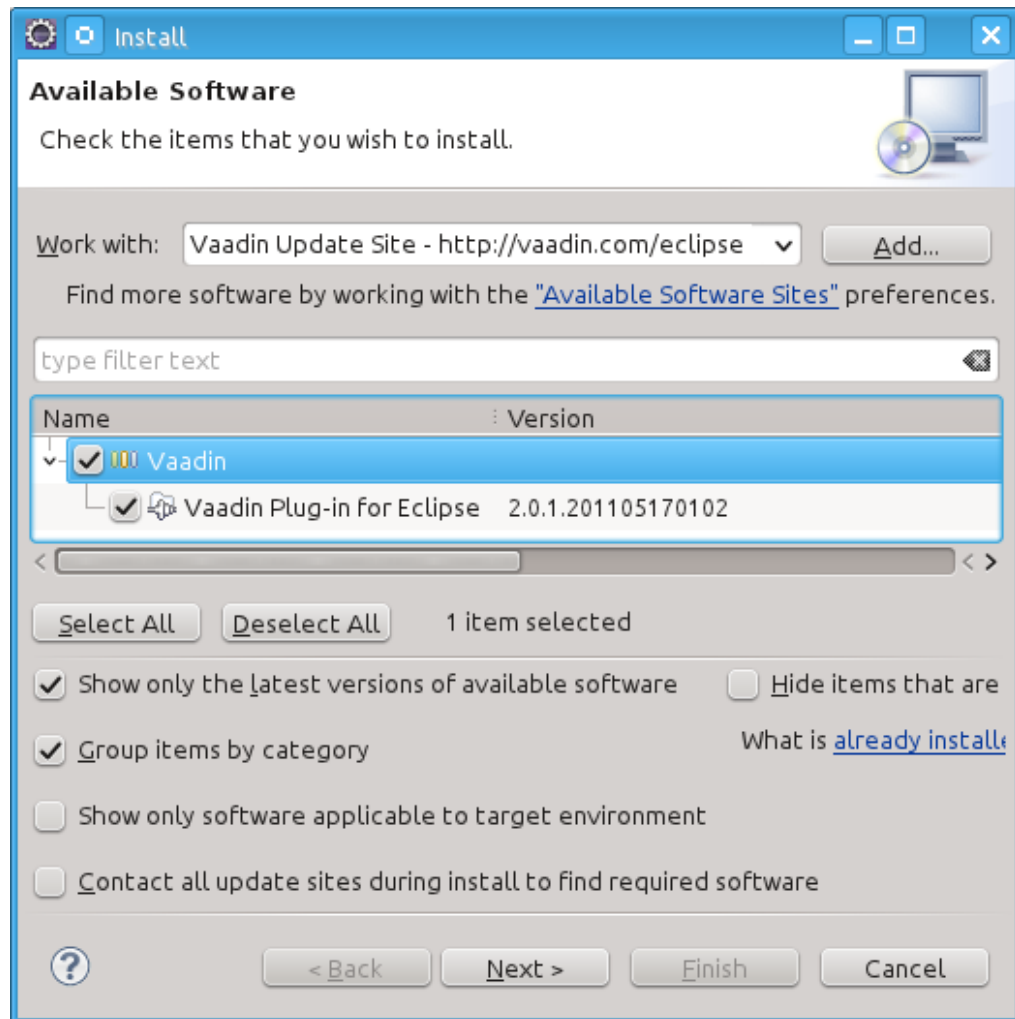
You can install the plugin as follows:

1. Select **Help** → **Install New Software...**
2. Add the Vaadin plugin update site by clicking **Add...** button.



Enter a name such as "Vaadin Update Site" and the URL of the update site: `http://vaadin.com/eclipse`. If you want or need to use the latest unstable plugin, which is usually more compatible with development and beta releases of Vaadin, you can use `http://vaadin.com/eclipse/experimental` and give it a distinctive name such as "Vaadin Experimental Site". Then click **OK**. The Vaadin site should now appear in the **Available Software** window.

3. Currently, if using the stable plugin, the **Group items by category** should be enabled. If using the experimental plugin, it should be disabled. This may change in future.
4. Select all the Vaadin plugins in the tree.



Then, click **Next**.

5. Review the installation details and click **Next**.
6. Accept or unaccept the license. Finally, click **Finish**.
7. After the plugin is installed, Eclipse will ask to restart itself. Click **Restart**.

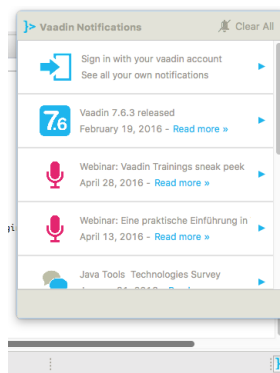
More installation instructions for the Eclipse plugin can be found at <http://vaadin.com/eclipse>.

2.5.3. Notification Center

The notification center is a feature of the Vaadin Eclipse plug-in. It displays notifications about new Vaadin releases as well as news about upcoming events, such as webinars. The notification center can be connected to your Vaadin account.

The plug-in adds an indicator in the bottom right corner. The indicator shows whether or not there are any pending notifications. The indicator turns red when there are new notifications.

Clicking the tray icon opens up the pop-up, as shown in Figure 2.4, "Overview of the notification center".

Figure 2.4. Overview of the notification center

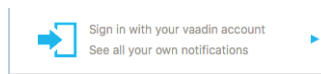
By clicking a notification in the list, you can open it up.

Clicking on the **Clear All** icon in the main pop-up clears all notifications and marks them all as read.

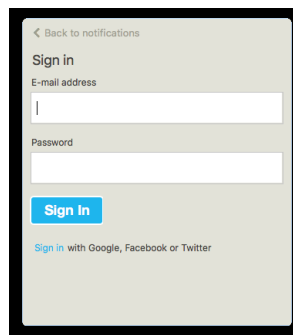
Signing in

The notification center uses your Vaadin account to determine which notifications you have acknowledged as read. If you want to keep the notification center in sync with your Vaadin account, you can sign in. If you have read a notification on the site, it will be marked as read in the notification center and vice versa.

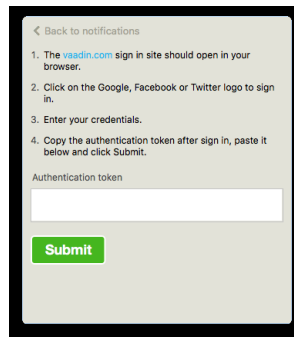
When you are not signed in, the top-most notification will be a notification that asks you to sign in.

Figure 2.5. The sign-in notification

When you click the sign-in item, a dialog opens for signing in.

Figure 2.6. The sign-in dialog.

You can then sign in with your Vaadin account. If you do not have one, you can sign in using a Google, Facebook, or Twitter account instead. First, click on the sign-in link. It opens a second dialog, as shown in Figure 2.7, "Sign-in authentication dialog".

Figure 2.7. Sign-in authentication dialog

By following the vaadin.com link and logging in to the Vaadin website using your preferred account, you can then copy the authentication token from the resulting page. After that, you can paste the authorization token into the dialog and click **Submit** to log in.

Notification Settings

You can open the settings by selecting **Eclipse... → Preferences**.

The options are as follows:

Enable Vaadin Notifications

Disabling this disables all notifications.

Inform me about new notifications using a popup

Disabling this stops pop-ups from appearing. The notifications can still be viewed by opening the notification center window.

In the **Update Schedule** panel, you can set the polling frequency.

Refresh notifications from scratch each time when list is shown

Forces the notifications center to poll all notifications every time.

Get notifications on IDE start

Immediately request notifications when the IDE starts without waiting for the polling interval.

Look for new notifications / Vaadin versions

Interval for polling new notifications.

2.5.4. Updating the Plugins

If you have automatic updates enabled in Eclipse (see **Window → Preferences → Install/Update → Automatic Updates**), the Vaadin plugin will be updated automatically along with other plugins. Otherwise, you can update the Vaadin plugin manually as follows:

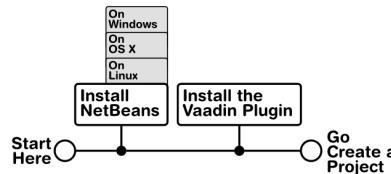
1. Select **Help → Check for Updates**. Eclipse will contact the update sites of the installed software.
2. After the updates are installed, Eclipse will ask to restart itself. Click **Restart**.

Notice that updating the Vaadin plugin only updates the plugin and *not* the Vaadin libraries, which are project specific. See below for instructions for updating the libraries.

2.6. Installing the NetBeans IDE and Plugin

Vaadin offers official support for the NetBeans IDE. The Vaadin Plug-in for NetBeans supports creating Vaadin projects, updating Vaadin libraries, compiling widget sets, and more. It also allows directly downloading Vaadin add-ons from the Vaadin Directory.

Figure 2.8. Installation of the NetBeans IDE toolchain



The installation bundle includes a web server, so you do not need that.

Once done with the installation, you can proceed to create a Vaadin project, as described in Section 3.6, “Creating a Project with the NetBeans IDE”.

2.6.1. Installing the NetBeans IDE

Download NetBeans IDE from the website at netbeans.org. We recommend using the *Java EE* download bundle, which includes support for Java EE, and also both the GlassFish and Tomcat application servers.

1. Run the installer
 - a. In OS X and Linux:

```
$ sh netbeans-<version>.sh
```
2. Select to install either GlassFish or Apache Tomcat, or both. GlassFish supports Java EE, which is required by Vaadin CDI and Vaadin Spring add-ons, while standard Tomcat does not support it. Click **Next**.

3. If you accept the license, click **Next**.

4. Choose installation folder and Java SDK.

In OS X and Linux, if you ran the installer with root permissions or can write to `/opt`, the `/opt` path is standard for such system-wide packages.

Click **Next**.

5. Choose the folder for installing the server.

6. Check the settings.

Click **Finish**. It takes a while to install the NetBeans IDE.

7. Finally, click **Done** to exit the installer.

You can now start NetBeans by starting the `bin/netbeans` from under the installation folder.

In Linux and OS X:

```
$ /opt/netbeans-8.1/bin/netbeans
```

You can now proceed to install the Vaadin Plug-in for NetBeans IDE.

2.6.2. Installing the Vaadin Plug-in for NetBeans IDE

You can install the plug-in from the NetBeans Plugin Portal Update Center as follows.

1. Select **Tools** → **Plugins** from the NetBeans main menu.
2. Select the **Available Plugins** tab.
 - a. Type "Vaadin" in the **Search** box and press `kbd:Enter[]`.
 - b. Select the **Install** check box for the **Vaadin Plugin for NetBeans**.
 - c. Click **Install**.
3. In the plugin installation window, click **Next**.
4. Accept the license if choose to do so. Click **Install**.
5. The Vaadin Plugin is not signed, so you need to verify the certificate. Click **Continue**.
6. In the final step, select **Restart IDE now** and click `[guibutton]#Finish`.

You can now proceed to create a Vaadin project, as described in Section 3.6, "Creating a Project with the NetBeans IDE".

The Vaadin Plug-in for NetBeans IDE can also be downloaded from the plug-in page at plugins.netbeans.org/plugin/50531/vaadin-plugin-in-for-netbeans.

2.7. Installing and Configuring IntelliJ IDEA

With IntelliJ IDEA, you have two choices: use the commercial Ultimate Edition or the free Community Edition. In the following, we cover the installation and configuration of them both.

The Ultimate Edition

The Ultimate Edition includes built-in support for creating Vaadin applications and running or debugging them in an integrated application server.

Community Edition

You can create a Vaadin application most easily with a Maven archetype and deploy it to a server using a Maven run/debug configuration.

You can get the both editions from the website at jetbrains.com/idea.

2.7.1. Installing the Ultimate Edition

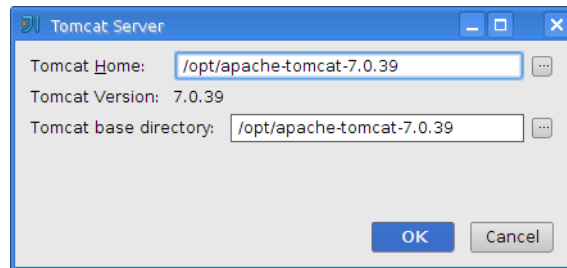
Follow the installation instructions given at the website.

Configuring an Application Server

To run a Vaadin application during development in the Ultimate Edition of IntelliJ IDEA, you first need to install and configure an application server that is integrated with the IDE. The edition includes integration with many commonly used application servers.

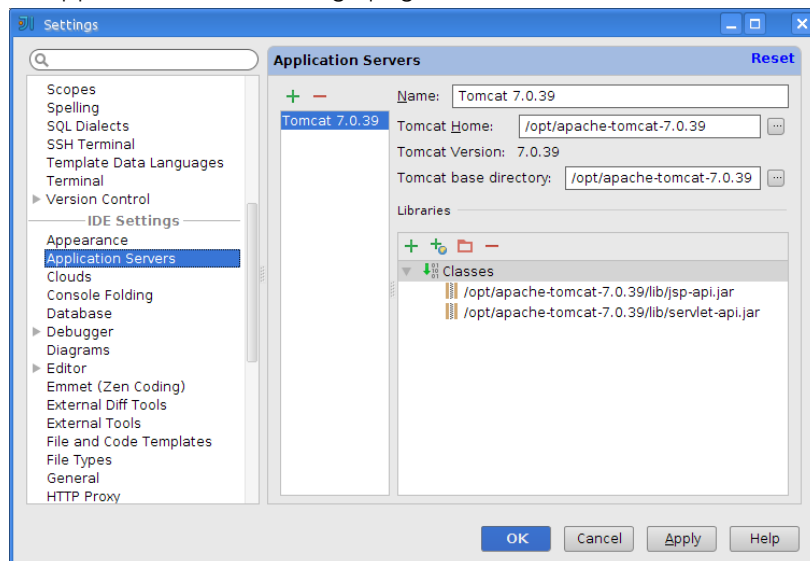
In the following, we configure Apache Tomcat:

1. Download and extract Tomcat installation package to a local directory, as instructed in Section 2.4.1, “Installing Apache Tomcat”.
2. Select **Configure** → **Settings**.
3. Select **IDE Settings** → **Application Servers**.
4. Click **+** and select **Tomcat Server** to add a Tomcat server, or any of the other supported servers. A WebSocket-enabled server, such as Glassfish or TomEE, is required for server push.
5. In the Tomcat Server dialog, specify the home directory for the server.



Click **OK**.

6. Review the application server settings page to check that it is OK.



Then, click **OK**.

2.8. Installing Other Tools

We recommend using a browser with either integrated or external development tools.

At least Mozilla Firefox and Google Chrome have an integrated web inspector, which supports inspecting the DOM structure of a web page, as well as CSS styles and debug JavaScript execution.

2.8.1. Firefox and Firebug

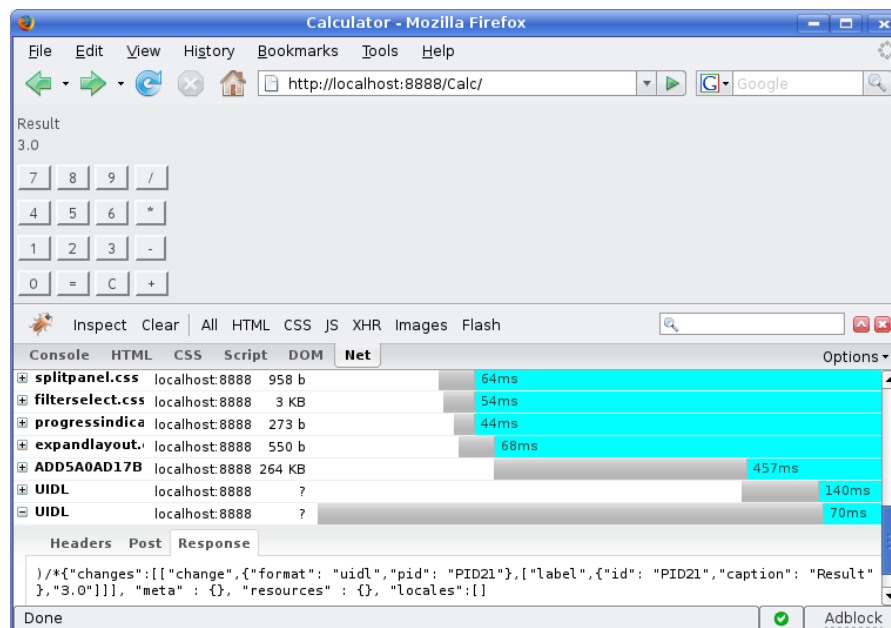
While Firefox has an integrated web inspector, the Firebug inspector has some additional features.

Using Firebug with Vaadin

After installing Firefox, use it to open <http://www.getfirebug.com/>. Follow the instructions on the site to install the latest stable version of Firebug available for the browser. You may need to allow Firefox to install the plugin by clicking the yellow warning bar at the top of the browser window.

After Firebug is installed, it can be enabled at any time from the Firefox toolbar. Figure 2.9, “Firebug Debugger for Firefox” shows Firebug in action.

Figure 2.9. Firebug Debugger for Firefox



The most important feature in Firebug is inspecting HTML elements. Right-click on an element and select **Inspect Element with Firebug** to inspect it. In addition to HTML tree, it also shows the CSS rules matching the element, which you can use for building themes. You can even edit the CSS styles live, to experiment with styling.

Chapter 3

Creating a Vaadin Application

3.1. Overview	47
3.2. Vaadin Libraries	48
3.3. Overview of Maven Archetypes	49
3.4. Creating and Running a Project in Eclipse	49
3.5. Creating a Project with Maven	58
3.6. Creating a Project with the NetBeans IDE	60
3.7. Creating a Project with IntelliJ IDEA	63
3.8. Vaadin Installation Package	68
3.9. Using Vaadin with Scala	68

This chapter gives practical instructions for creating a Vaadin application project and deploying it to a server to run it. We also consider topics such as debugging.

The instructions are given separately for the Eclipse IDE, NetBeans, and IntelliJ IDEA.

3.1. Overview

Once you have installed a development environment, as described in the previous chapter, creating a Vaadin project proceeds in the IDE that you have chosen.

The Vaadin core library and all Vaadin add-ons are available through Maven, a commonly used build and dependency management system.

The recommended way to create a Vaadin application project is to use a Maven archetype. The archetypes contain all the needed dependencies, which Maven takes care of. The Eclipse IDE plugin currently also supports creating a normal Eclipse web project using the Ivy dependency manager.

In this chapter, we give:

1. An overview of the Vaadin libraries
2. List the available Maven archetypes
3. Step-by-step instructions for creating a project in the Eclipse IDE, NetBeans IDE, and IntelliJ IDEA, as well as with command-line.

3.2. Vaadin Libraries

Vaadin comes as a set of library JARs, of which some are optional or alternative ones, depending on whether you are developing server-side or client-side applications, whether you use add-on components, or use CSS or Sass themes.

`vaadin-server-7.x.x.jar`

The main library for developing server-side Vaadin applications, as described in Chapter 5, *Writing a Server-Side Web Application*. It requires the `vaadin-shared` and the `vaadin-themes` libraries. You can use the pre-built `vaadin-client-compiled` for server-side development, unless you need add-on components or custom widgets.

`vaadin-shared-7.x.x.jar`

A shared library for server-side and client-side development. It is always needed.

`vaadin-client-7.x.x.jar`

The client-side Vaadin framework, including the basic GWT API and Vaadin-specific widgets and other additions. It is required when using the `vaadin-client-compiler` to compile client-side modules. It is not needed if you just use the server-side framework with the pre-compiled Client-Side Engine. You should not deploy it with a web application.

`vaadin-client-compiler-7.x.x.jar`

The Vaadin Client Compiler is a Java-to-JavaScript compiler that allows building client-side modules, such as the Client-Side Engine (widget set) required for server-side applications. The compiler is needed, for example, for compiling add-on components to the application widget set, as described in Chapter 18, *Using Vaadin Add-ons*.

For detailed information regarding the compiler, see Section 14.4, “Compiling a Client-Side Module”. Note that you should not deploy this library with a web application.

`vaadin-client-compiled-7.x.x.jar`

A pre-compiled Vaadin Client-Side Engine (widget set) that includes all the basic built-in widgets in Vaadin. This library is not needed if you compile the application widget set with the Vaadin Client Compiler.

`vaadin-themes-7.x.x.jar`

Vaadin built-in themes both as SCSS source files and precompiled CSS files. The library is required both for basic use with CSS themes and for compiling custom Sass themes.

`vaadin-sass-compiler-1.x.x.jar`

The Vaadin Sass Compiler compiles Sass themes to CSS, as described in Section 9.3, “Syntactically Awesome Stylesheets (Sass)”. It requires the `vaadin-themes-7.x.x.jar` library, which contains the Sass sources for the built-in themes. The library needs to be included in deployment in development mode to allow on-the-fly compilation of themes, but it is not needed in production deployment, when the themes are compiled before deployment.

Some of the libraries depend on each other as well as on the dependency libraries provided in the `lib` folder of the installation package, especially the `lib/vaadin-shared-deps.jar`.

The different ways to install the libraries are described in the subsequent sections.

Note that the `vaadin-client-compiler` and `vaadin-client` JARs should not be deployed with the web application by including them in `WEB-INF/lib`. Some other libraries, such as `vaadin-sass-compiler`, are not needed in production deployment.

3.3. Overview of Maven Archetypes

Vaadin currently offers the following Maven archetypes for different kinds of projects:

`vaadin-archetype-application`

This is a single-module project for simple applications. It is good for quick demos and trying out Vaadin. It is also useful when you are experienced with Vaadin and want to build all the aspects of the application yourself.

`vaadin-archetype-application-multimodule`

A complete Vaadin application development setup. It features separate production and development profiles.

`vaadin-archetype-application-example`

An example CRUD web application using multi-module project setup.

`vaadin-archetype-widget`

A multi-module project for a new Vaadin add-on. It has two modules: one for the add-on and another for a demo application.

`vaadin-archetype-touchkit`

A mobile development starter project using Vaadin TouchKit. See Chapter 21, *Mobile Applications with TouchKit*. Notice that this archetype uses the AGPL-licensed version of TouchKit, which requires that your project must also be licensed under the AGPL license.

`vaadin-archetype-liferay-portlet`

A portlet development setup for the open-source Liferay portal.

3.4. Creating and Running a Project in Eclipse

This section gives instructions for creating a new Eclipse project using the Vaadin Plugin. The task will include the following steps:

1. Create a new project
2. Write the source code

3. Configure and start Tomcat (or some other web server)
4. Open a web browser to use the web application

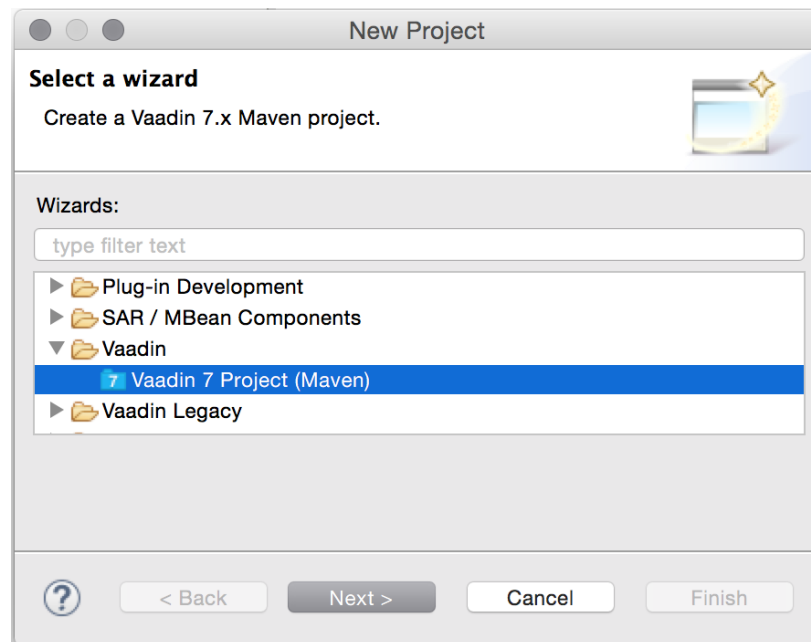
We also show how you can debug the application in the debug mode in Eclipse.

This walkthrough assumes that you have already installed the Eclipse IDE, the Vaadin Plugin, and a development server, as instructed in Section 2.5, “Installing the Eclipse IDE and Plugin”.

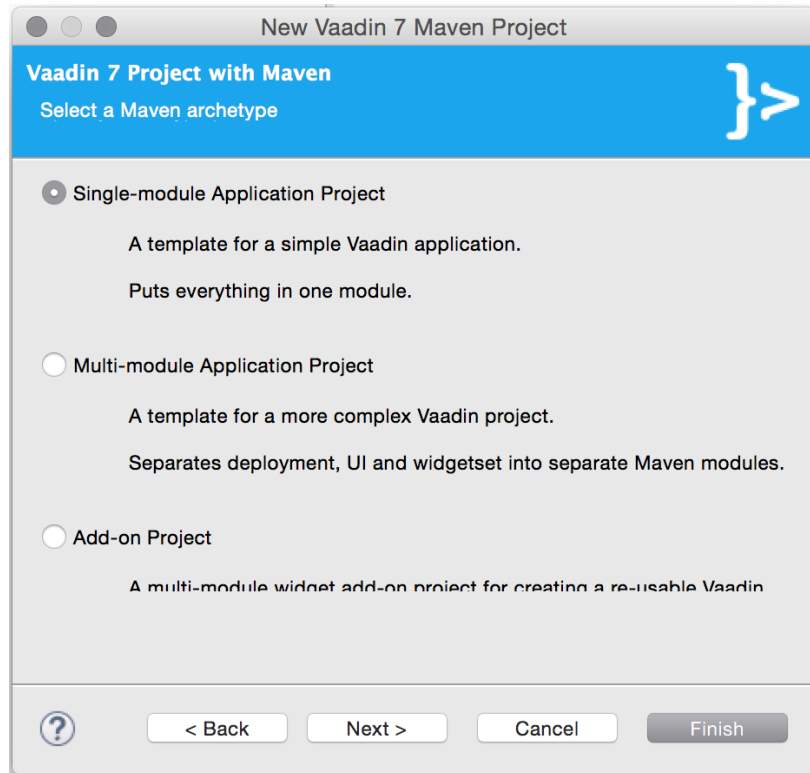
3.4.1. Creating a Maven Project

Let us create the first application project with the tools installed in the previous section. First, launch Eclipse and follow the following steps:

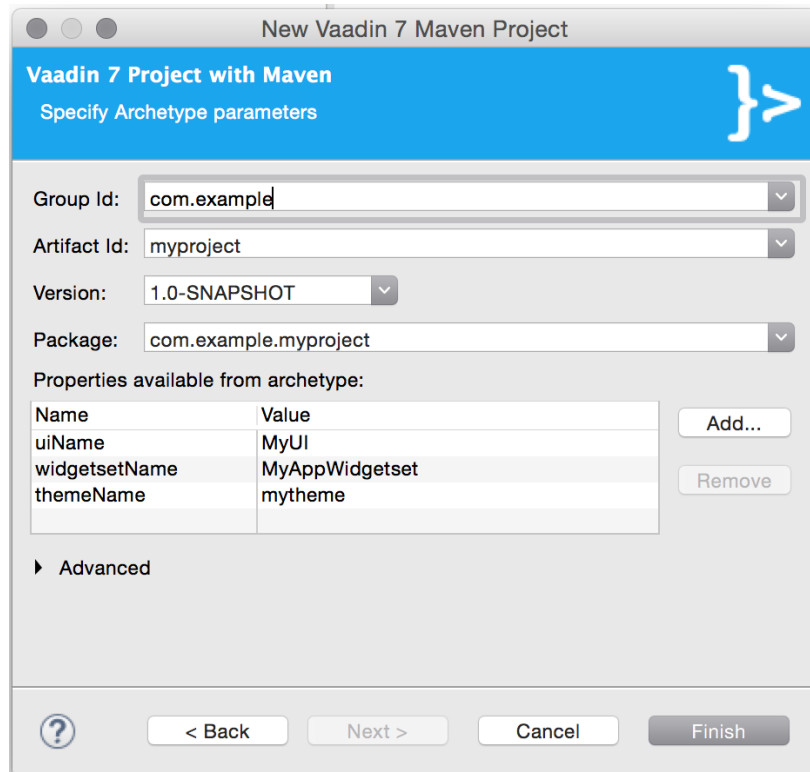
1. Start creating a new project by selecting from the menu **File → New → Project...**
2. In the **New Project** window that opens, select **Vaadin → Vaadin 7 Project (Maven)** and click **Next**.



3. In the **Select a Maven archetype** step, you need to select the project type. To create a simple test project, select the **Single-module Application Project**.



4. In the **Specify archetype parameters** step, you need to give at least the **Group Id** and the **Artifact Id**. The default values should be good for the other settings.



Group Id

Give the project an organization-level identifier, for example, `com.example`. It is used as a prefix for your Java package names, and hence must be a valid Java package name itself.

Artifact Id

Give the project a name, for example, `myproject`. The artifact ID must be a valid Java sub-package name.

Version

Give the project a Maven compatible version number, for example, `1.0-SNAPSHOT`. The version number should typically start with two or more integers separated with dots, and should not contain spaces.

Package

Give the base package name for the project, for example, `com.example.myproject`. It is by default generated from the group ID and the artifact ID.

Properties

Enter values for archetype-specific properties that control naming of various elements in the created project, such as the UI class name.

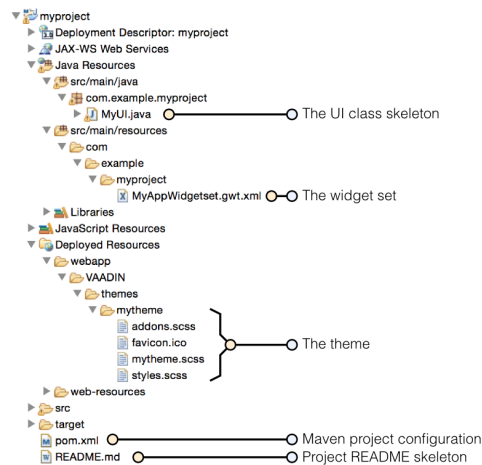
You can change the version later in the `pom.xml`.

Finally, click **Finish** to create the project.

3.4.2. Exploring the Project

After the **New Project** wizard exits, it has done all the work for you: a UI class skeleton has been written to the `src` directory. The project hierarchy shown in the Project Explorer is shown in Figure 3.1, “A new Vaadin Project”.

Figure 3.1. A new Vaadin Project



The Vaadin libraries and other dependencies are managed by Maven. Notice that the libraries are not stored under the project folder, even though they are listed in the **Java Resources** → **Libraries** → **Maven Dependencies** virtual folder.

The UI Class

The UI class created by the plug-in contains the following code:

```
package com.example.myproject;

import com.vaadin.ui.UI;
...

@Theme("mytheme")
@Widgetset("com.example.myproject.MyAppWidgetset")
public class MyUI extends UI {

    @Override
    protected void init(VaadinRequest vaadinRequest) {
        final VerticalLayout layout = new VerticalLayout();

        final TextField name = new TextField();
        name.setCaption("Type your name here:");

        Button button = new Button("Click Me");
        button.addClickListener( e -> {
            layout.addComponent(new Label("Thanks " + name.getValue()
                + ", it works!"));
        });

        layout.addComponents(name, button);
        layout.setMargin(true);
        layout.setSpacing(true);

        setContent(layout);
    }

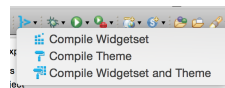
    @WebServlet(urlPatterns = "/*", name = "MyUIServlet", asyncSupported = true)

    @VaadinServletConfiguration(ui = MyUI.class, productionMode = false)
    public static class MyUIServlet extends VaadinServlet {
    }
}
```

3.4.3. Compiling the Widget Set and Theme

Before running the project for the first time, select **Compile Widgetset and Theme** from the menu shown in Figure 3.2, “Compile Widgetset and Theme Menu”.

Figure 3.2. Compile Widgetset and Theme Menu

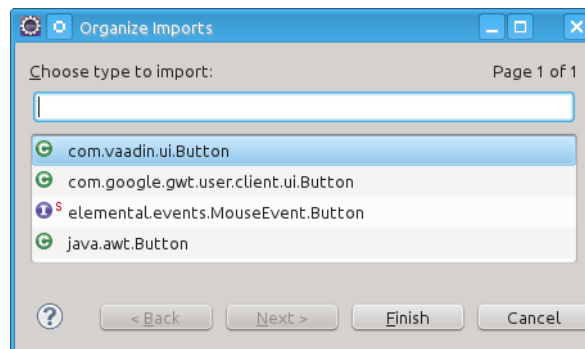


3.4.4. Coding Tips for Eclipse

One of the most useful features in Eclipse is *code completion*. Pressing **Ctrl+Space** in the editor will display a pop-up list of possible class name and method name completions, as shown in Figure 3.3, “Java Code Completion in Eclipse”, depending on the context of the cursor position.

Figure 3.3. Java Code Completion in Eclipse

To add an `import` statement for a class, such as **Button**, simply press **Ctrl+Shift+O** or click the red error indicator on the left side of the editor window. If the class is available in multiple packages, a list of the alternatives is displayed, as shown in Figure 3.4, “Importing Classes Automatically”. For server-side development, you should normally use the classes under the `com.vaadin.ui` or `com.vaadin.server` packages. You can not use client-side classes (under `com.vaadin.client`) or GWT classes for server-side development.

Figure 3.4. Importing Classes Automatically

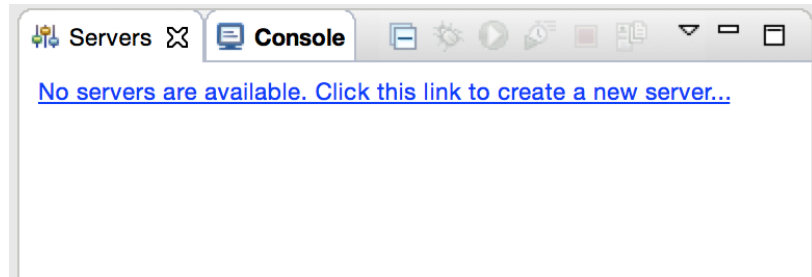
3.4.5. Setting Up and Starting the Web Server

Eclipse IDE for Java EE Developers has the Web Standard Tools package installed, which supports control of various web servers and automatic deployment of web content to the server when changes are made to a project.

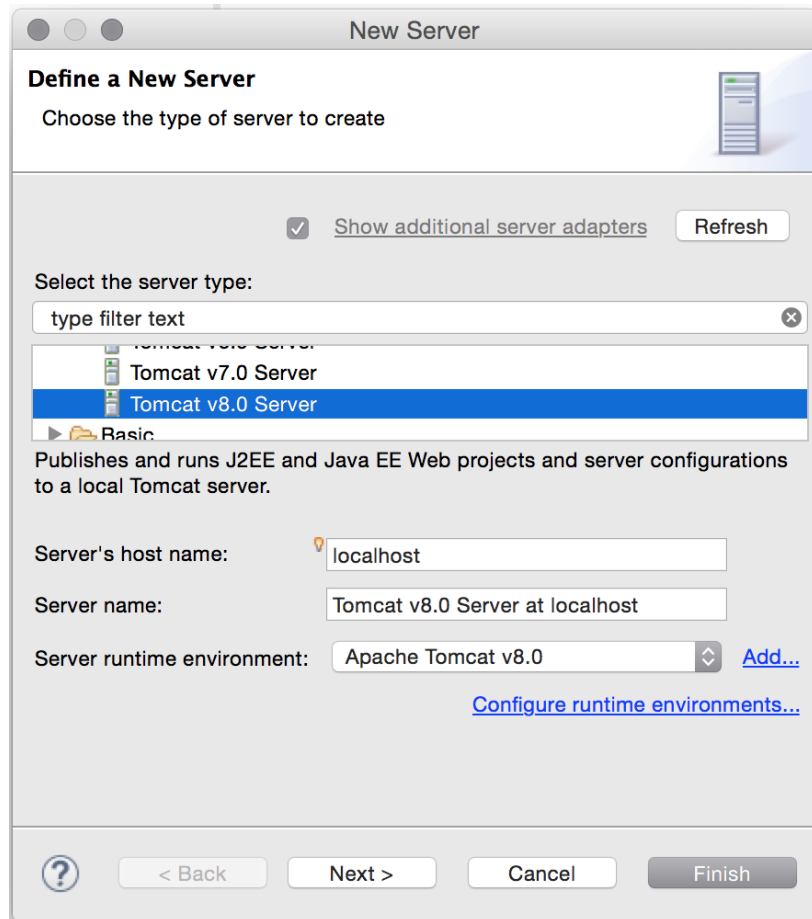
Make sure that Tomcat was installed with user permissions. Configuration of the web server in Eclipse will fail if the user does not have write permissions to the configuration and deployment directories under the Tomcat installation directory.

Follow the following steps:

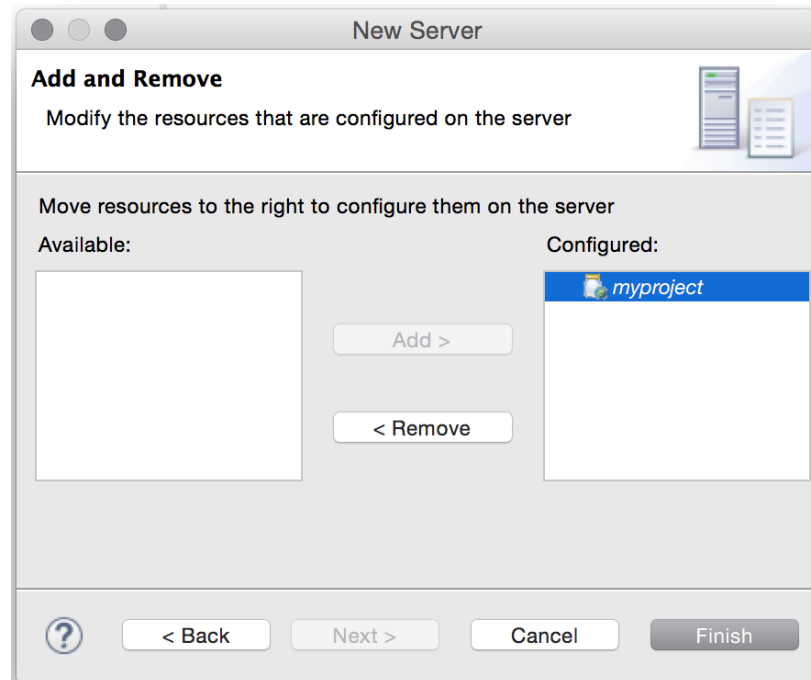
1. Switch to the Servers tab in the lower panel in Eclipse. List of servers should be empty after Eclipse is installed. Right-click on the empty area in the panel and select **New** → **Server**.



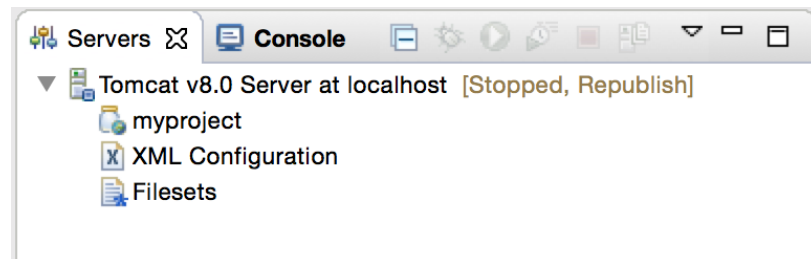
2. Select **Apache** → **Tomcat v7.0 Server** and set **Server's host name** as `localhost`, which should be the default. If you have only one Tomcat installed, **Server runtime** has only one choice. Click **Next**.



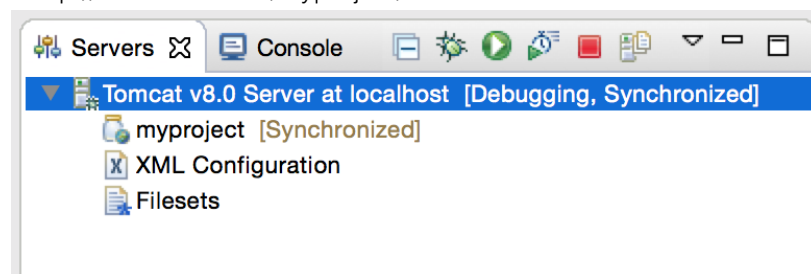
3. Add your project to the server by selecting it on the left and clicking **Add** to add it to the configured projects on the right. Click **Finish**.



4. The server and the project are now installed in Eclipse and are shown in the **Servers** tab. To start the server, right-click on the server and select **Debug**. To start the server in non-debug mode, select **Start**.

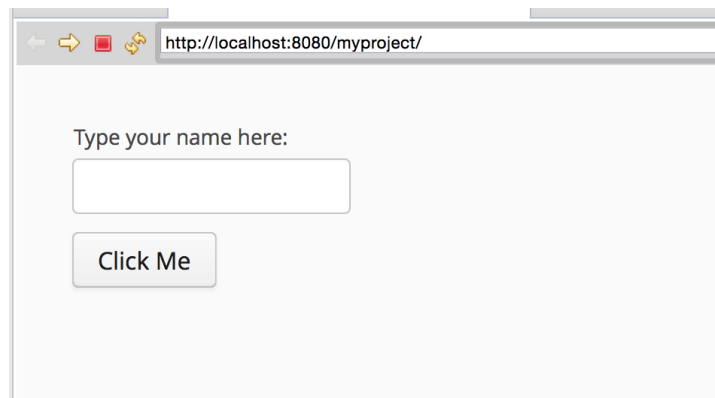


5. The server starts and the WebContent directory of the project is published to the server on <http://localhost:8080/myproject/>.

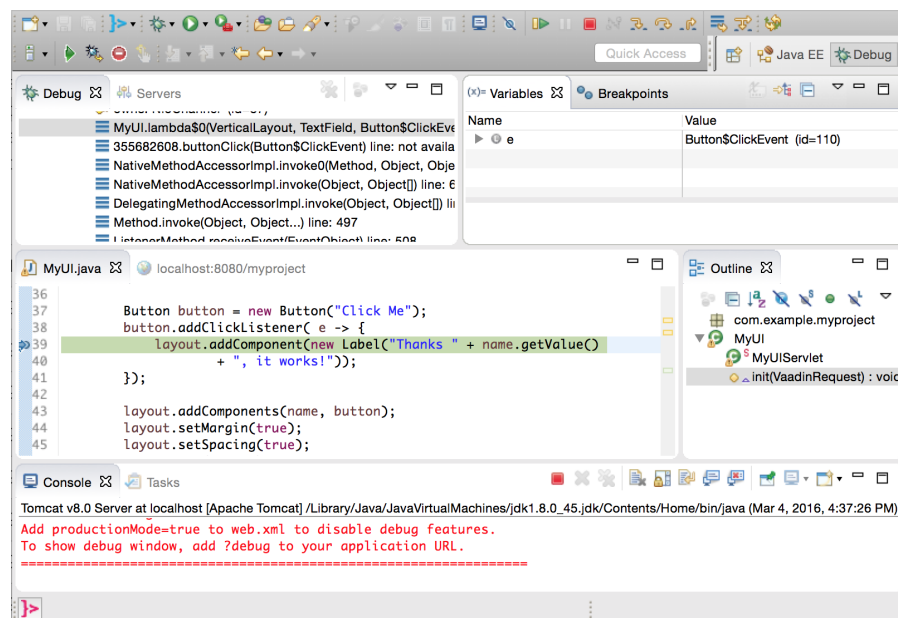


3.4.6. Running and Debugging

Starting your application is as easy as selecting **myproject** from the **Project Explorer** and then **Run** → **Debug As** → **Debug on Server**. Eclipse then opens the application in built-in web browser.

Figure 3.5. Running a Vaadin Application

You can insert break points in the Java code by double-clicking on the left margin bar of the source code window. For example, if you insert a breakpoint in the `buttonClick()` method and click the **What is the time?** button, Eclipse will ask to switch to the Debug perspective. Debug perspective will show where the execution stopped at the breakpoint. You can examine and change the state of the application. To continue execution, select **Resume** from **Run** menu.

Figure 3.6. Debugging a Vaadin Application

Above, we described how to debug a server-side application. Debugging client-side applications and widgets is described in Section 14.6, “Debugging Client-Side Code”.

3.4.7. Updating the Vaadin Libraries in Maven Projects

Updating the Vaadin plugin does not update Vaadin libraries. The libraries are project specific, as a different version might be required for different projects, so you have to update them separately for each project.

1. Open the `pom.xml` in an editor in Eclipse.

2. Edit the `vaadin.version` property to set the Vaadin version.

Updating the libraries can take several minutes. You can see the progress in the Eclipse status bar. You can get more details about the progress by clicking the indicator.

3. If you have compiled the widget set for your project, recompile it by clicking the **Compile Vaadin widgets** button in Eclipse toolbar.
4. Stop the integrated Tomcat (or other server) in Eclipse, clear its caches by right-clicking the server and selecting **Clean** as well as **Clean Tomcat Work Directory**, and restart it.

If you experience problems after updating the libraries, you can try using **Maven → Update Project**.

3.4.8. Updating the Vaadin Libraries in Ivy Projects

Updating the Vaadin plugin does not update Vaadin libraries. The libraries are project specific, as a different version might be required for different projects, so you have to update them separately for each project.

1. Open the `ivy.xml` in an editor in Eclipse.
2. Edit the entity definition at the beginning of the file to set the Vaadin version.

```
<!ENTITY vaadin.version "7.x.x">
```

You can specify either a fixed version number, as shown in the above example, or a dynamic revision tag such as `latest.release`. You can find more information about the dependency declarations in Ivy documentation.

3. Right-click the project and select **Ivy → Resolve**.

Updating the libraries can take several minutes. You can see the progress in the Eclipse status bar. You can get more details about the progress by clicking the indicator.

4. If you have compiled the widget set for your project, recompile it by clicking the **Compile Vaadin widgets** button in Eclipse toolbar.
5. Stop the integrated Tomcat (or other server) in Eclipse, clear its caches by right-clicking the server and selecting **Clean** as well as **Clean Tomcat Work Directory**, and restart it.

If you experience problems after updating the libraries, you can try clearing the Ivy resolution caches by right-clicking the project and selecting **Ivy → Clean all caches**. Then, do the **Ivy → Resolve** and other tasks again.

3.5. Creating a Project with Maven

In previous sections, we looked into creating a Vaadin Maven project in different IDEs. In this section, we look how to create such a project on command-line. You can then import such a project to your IDE.

In addition to regular Maven, you can use any Maven-compatible build or dependency management system, such as Ivy or Gradle. For Gradle, see the Gradle Vaadin Plugin. Vaadin Plugin for Eclipse uses Ivy for resolving dependencies in Vaadin projects, and it should provide you with the basic Ivy configuration.

For an interactive guide, see the instructions at vaadin.com/maven. It automatically generates you the command to create a new project based on archetype selection. It can also generate dependency declarations for Vaadin dependencies.

3.5.1. Working from Command-Line

You can create a new Maven project with the following command (given in one line):

```
$ mvn archetype:generate \
  -DarchetypeGroupId=com.vaadin \
  -DarchetypeArtifactId=vaadin-archetype-application \
  -DarchetypeVersion=7.x.x \
  -DgroupId=your.company \
  -DartifactId=project-name \
  -Dversion=0.1 \
  -Dpackaging=war
```

The parameters are as follows:

archetypeGroupId

The group ID of the archetype is `com.vaadin` for Vaadin archetypes.

archetypeArtifactId

The archetype ID. See the list of available archetypes in Section 3.3, “Overview of Maven Archetypes”.

archetypeVersion

Version of the archetype to use. This should be `LATEST` for normal Vaadin releases. For prerelease versions it should be the exact version number, such as `7.6.4`.

groupId

A Maven group ID for your project. It is normally your organization domain name in reverse order, such as `com.example`. The group ID is also used as a prefix for the Java package in the sources, so it should be Java compatible - only alphanumerics and an underscore.

artifactId

Identifier of the artifact, that is, your project. The identifier may contain alphanumerics, minus, and underscore. It is appended to the group ID to obtain the Java package name for the sources. For example, if the group ID is `com.example` and artifact ID is `myproject`, the project sources would be placed in `com.example.myproject` package.

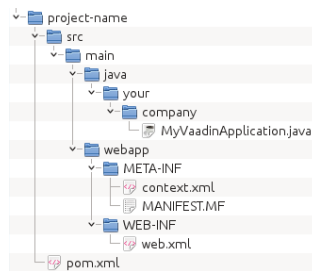
version

Initial version number of your application. The number must obey the Maven version numbering format.

packaging

How will the project be packaged. It is normally `war`.

Creating a project can take a while as Maven fetches all the dependencies. The created project structure is shown in Figure 3.7, “A New Vaadin Project with Maven”.

Figure 3.7. A New Vaadin Project with Maven

3.5.2. Compiling and Running the Application

Before the application can be deployed, it must be compiled and packaged as a WAR package. You can do this with the `package` goal as follows:

```
$ mvn package
```

The location of the resulting WAR package should be displayed in the command output. You can then deploy it to your favorite application server.

The easiest way to run Vaadin applications with Maven is to use the light-weight Jetty web server. After compiling the package, all you need to do is type:

```
$ mvn jetty:run
```

The special goal starts the Jetty server in port 8080 and deploys the application. You can then open it in a web browser at `http://localhost:8080/project-name`.

3.5.3. Using Add-ons and Custom Widget Sets

If you use Vaadin add-ons that include a widget set or make your custom widgets, you need to enable widget set compilation in the POM. The required configuration is described in Section 18.4, “Using Add-ons in a Maven Project”.

3.6. Creating a Project with the NetBeans IDE

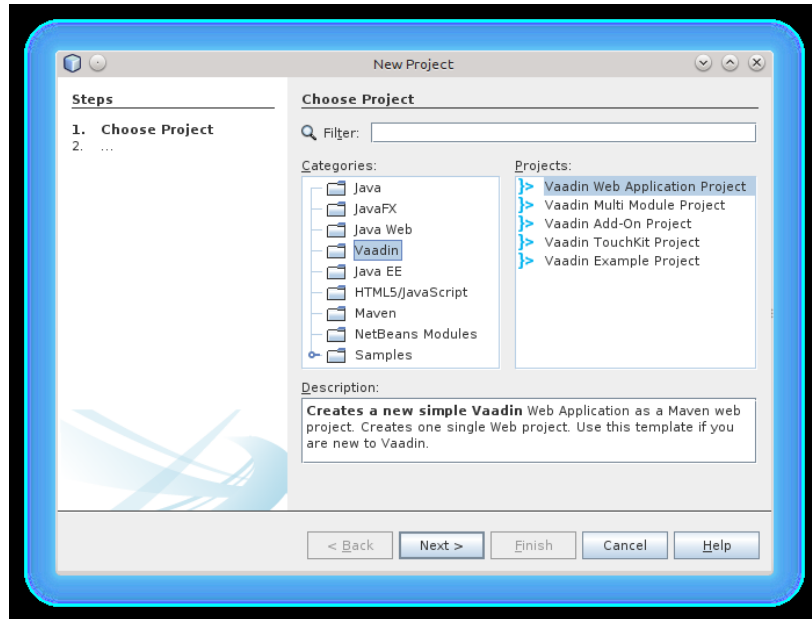
In the following, we walk you through the creation of a Vaadin project in NetBeans and show how to run it.

Installation of NetBeans and the Vaadin plugin is covered in Section 2.6, “Installing the NetBeans IDE and Plugin”.

Without the plugin, you can most easily create a Vaadin project as a Maven project using a Vaadin archetype. You can also create a Vaadin project as a regular web application project, but it requires many manual steps to install all the Vaadin libraries, create the UI class, configure the servlet, create theme, and so on.

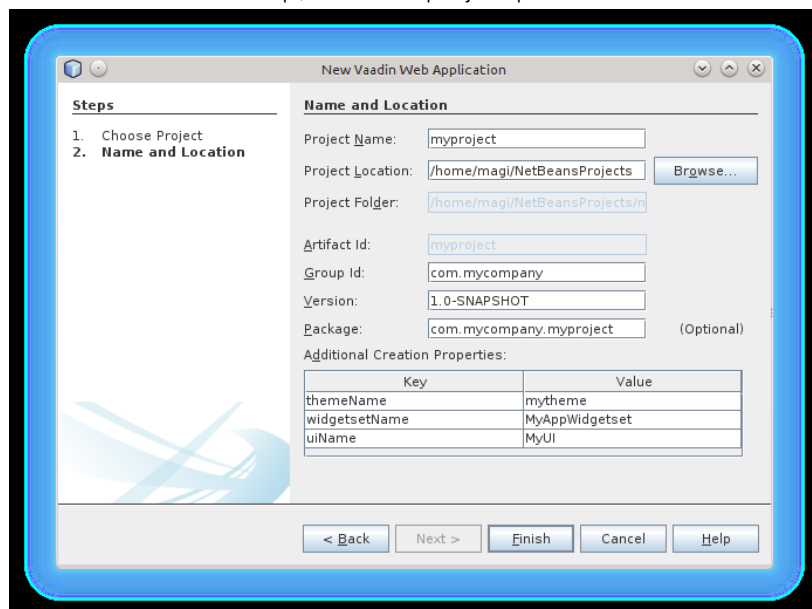
3.6.1. Creating a Project

1. Select **File** → **Net Project...** from the main menu or press **Ctrl+Shift+N**.
2. In the **New Project** window that opens, select the **Vaadin** category and one of the Vaadin archetypes from the right.



The archetypes are described in more detail in Section 3.3, “Overview of Maven Archetypes”.

3. In the **Name and Location** step, enter the project parameters.



Project Name

A project name. The name must be a valid identifier that may only contains alpha-numeric, minus, and underscore. It is appended to the group ID to obtain the Java package name for the sources.

Project Location

Path to the folder where the project is to be created.

Group Id

A Maven group ID for your project. It is normally your organization domain name in reverse order, such as `com.example`. The group ID is also used as a prefix for the Java source package, so it should be Java-compatible package name.

Version

Initial version of your application. The number must obey the Maven version numbering format.

Package

The Java package name to put sources in.

Additional Creation Properties

The properties control various names. They are specific to the archetype you chose.

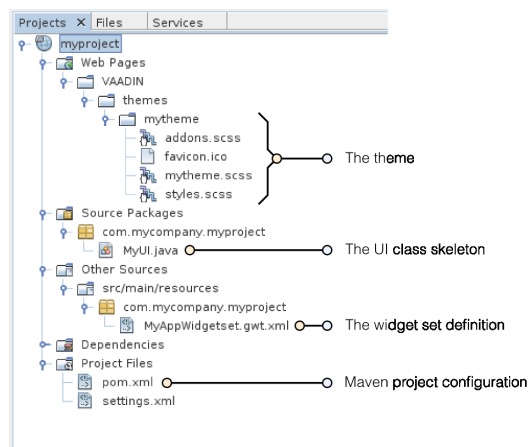
Click **Finish**.

Creating the project can take a while as Maven loads all the needed dependencies.

3.6.2. Exploring the Project

The project wizard has done all the work for you: a UI class skeleton has been written to the `src` directory. The project hierarchy shown in the Project Explorer is shown in Figure 3.8, “A new Vaadin project in NetBeans”.

Figure 3.8. A new Vaadin project in NetBeans



`mytheme`

The theme of the UI. See Chapter 9, *Themes* for information about themes.

`MyUI.java`

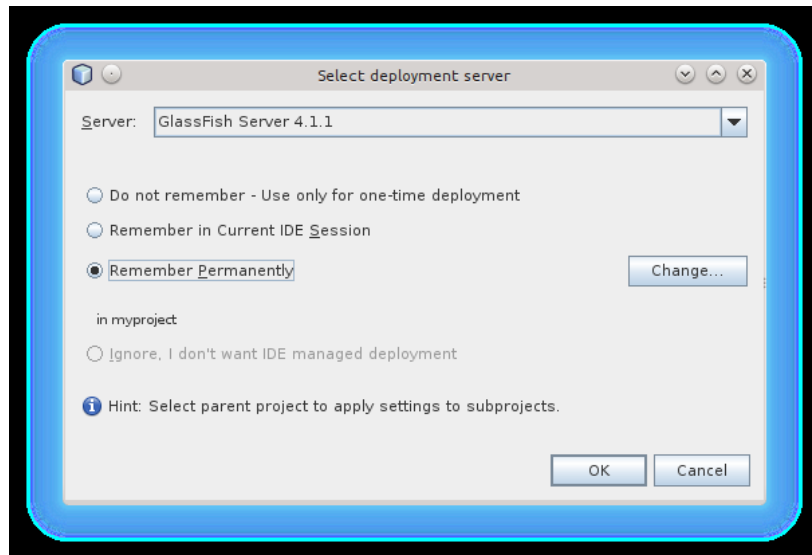
The UI class, which is the main entry-point of your application. See Chapter 5, *Writing a Server-Side Web Application* for information about the basic structure of Vaadin applications.

The Vaadin libraries and other dependencies are managed by Maven. Notice that the libraries are not stored under the project folder, even though they are listed in the **Java Resources** → **Libraries** → **Maven Dependencies** virtual folder.

3.6.3. Running the Application

Once created, you can run it in a server as follows.

1. In **Projects** tab, select the project and click in the **Run Project** button in the tool bar (or press **F6**).
2. In the **Select deployment server** window, select a server from the **Server** list. It should show either GlassFish or Apache Tomcat or both, depending on what you chose in NetBeans installation.



Also, select **Remember Permanently** if you want to use the same server also in future while developing applications.

Click **OK**.

The widget set will be compiled at this point, which may take a while.

If all goes well, NetBeans starts the server in port 8080 and, depending on your system configuration, launches the default browser to display the web application. If not, you can open it manually, for example, at <http://localhost:8080/myproject>. The project name is used by default as the context path of the application.

Now when you edit the UI class in the source editor and save it, NetBeans will automatically redeploy the application. After it has finished after a few seconds, you can reload the application in the browser.

3.7. Creating a Project with IntelliJ IDEA

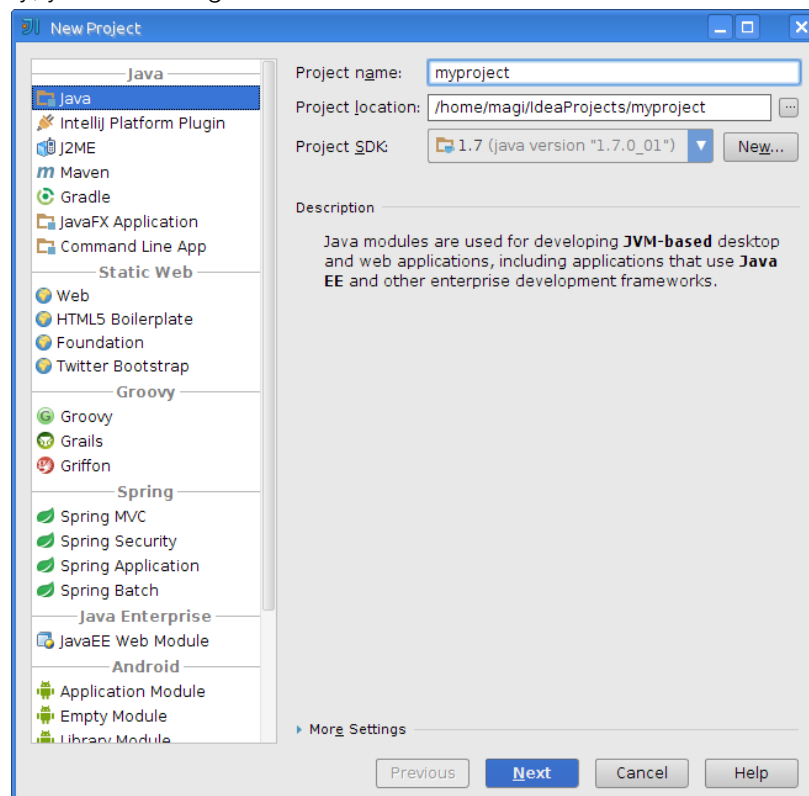
The Ultimate Edition of IntelliJ IDEA includes support for creating Vaadin applications and running or debugging them in an integrated application server.

With the Community Edition, you can create a Vaadin application most easily with a Maven archetype and deploy it to a server with a Maven run/debug configuration.

3.7.1. Creating a Vaadin Web Application Project

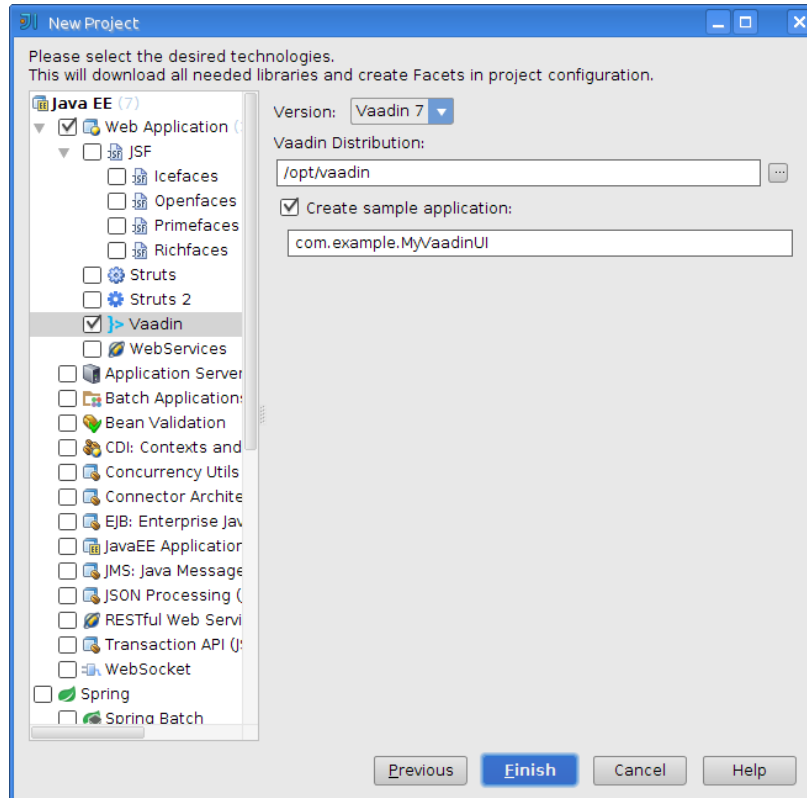
In the welcome page, do the following:

1. Download and extract the Vaadin installation package to a local folder, as instructed in Section 3.8, “Vaadin Installation Package”.
2. Select New Project
3. In the **New Project** window, select Java
4. Enter a **Project name** and **Project location**, and select the **Java SDK** to be used for the project. Vaadin requires at least Java 6. If you have not configured a Java SDK previously, you can configure it here.



Click **Next**.

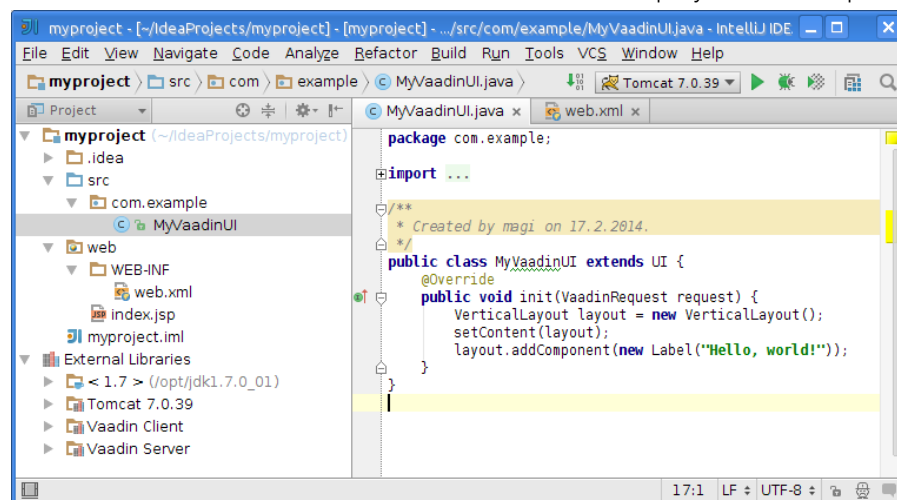
5. Select **Web Application** → **Vaadin** to add Vaadin technology to the project.
6. Select Vaadin **Version** and **Distribution** installation path. You probably also want an application stub, so select **Create sample application** and give a name for the generated UI class.



Do *not* click **Finish** yet.

7. Select **Application Server** in the same window. Set it as an integrated server that you have configured in IntelliJ IDEA, as described previously in the section called “Configuring an Application Server”.
8. Click **Finish**.

The project is created with the UI class stub and a `web.xml` deployment descriptor.



The wizard does not currently create a servlet class automatically, and uses Servlet 2.4 compatible deployment with a `web.xml` deployment descriptor.

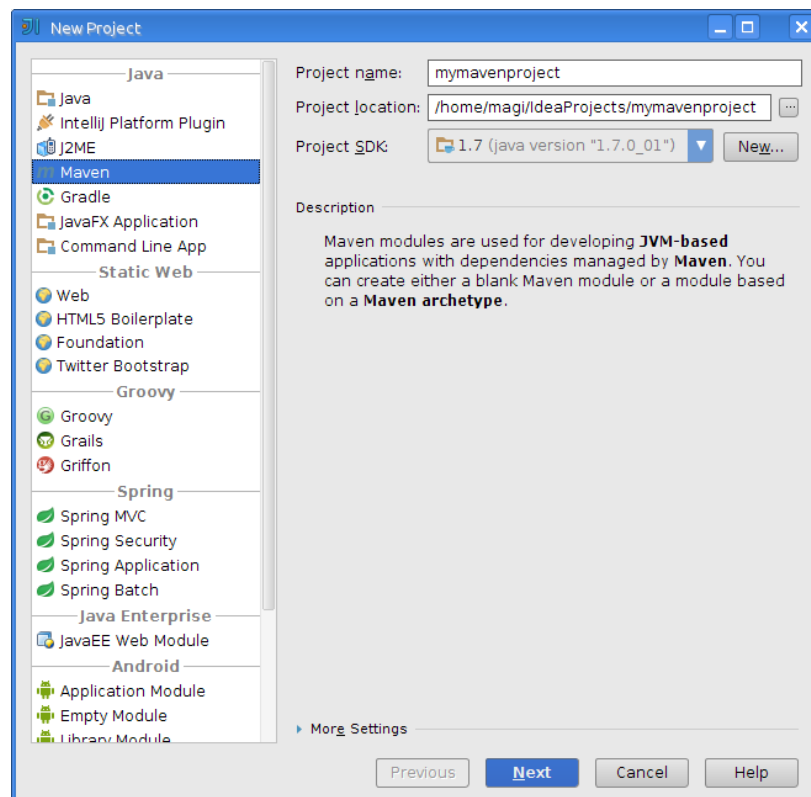
Deploying the Project

To deploy the application to the integrated web server, right-click the `index.jsp` file in the project and select `Run 'index.jsp'`. This starts the integrated server, if it was not already running, and launches the default browser with the application page.

3.7.2. Creating a Maven Project

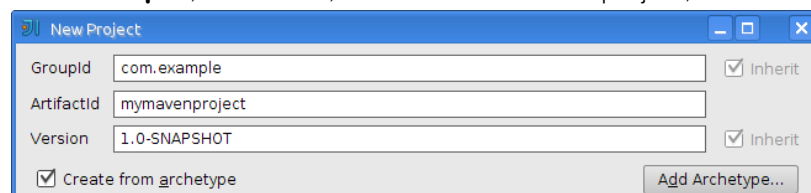
You can choose to create a Maven project in IntelliJ IDEA. This is the recommended way when using the Community Edition. You will not have the application server integration, but can deploy the application to an application server using a run/debug configuration.

1. Select New Project
2. In the **New Project** window, select Maven
3. Enter a project name, location, and the Java SDK to be used for the project. Vaadin requires at least Java 6.



Click **Next**.

4. Give a Maven **GroupId**, **ArtifactId**, and a **Version** for the project, or use the defaults.



5. Check **Create from archetype**
6. If the Vaadin archetype is not in the list, click **Add archetype**, enter **GroupId** `com.vaadin`, **ArtifactId** `vaadin-archetype-application`, and **Version** `LATEST` (or a specific version number).

Click **OK** in the dialog.
7. Select the `com.vaadin:vaadin-archetype-application`.

Click **Next**.
8. Review the general Maven settings and settings for the new project. You may need to override the settings, especially if you are creating a Maven project for the first time.

Click **Finish**.

Creating the Maven project takes some time as Maven fetches the dependencies. Once done, the project is created and the Maven POM is opened in the editor.

Compiling the Project

To compile a Vaadin application using Maven, you can define a run/debug configuration to execute a goal such as `package` to build the deployable WAR package. It will also compile the widget set and theme, if necessary. See Section 3.5.2, “Compiling and Running the Application” for more details.

Compilation is included in the following instructions for deploying the application.

Deploying to a Server

There exists Maven plugins for deploying to various application servers. For example, to deploy to Apache Tomcat, you can configure the `tomcat-maven-plugin` and then execute the `tomcat:deploy` goal. See the documentation of the plugin that you use for more details. If no Maven plugin exists for a particular server, you can always use some lower-level method to deploy the application, such as running an Ant task.

In the following, we create a run/debug configuration to build, deploy, and launch a Vaadin Maven application on the light-weight Jetty web server.

1. Select **Run** → **Edit Configurations**.
2. Click **+** and select **Maven** to create a new Maven run/debug configuration.
3. Enter a **Name** for the run configuration. For the **Command line**, enter `"`package jetty:run`#"` to first compile and package the project, and then launch Jetty to run it.

Click **OK**.
4. Select the run configuration in the toolbar and click the **Run** button beside it.

Compiling the project takes some time on the first time, as it compiles the widget set and theme. Once the run console pane informs that Jetty Server has been started, you can open the browser at the default URL `http://localhost:8080/`.

3.8. Vaadin Installation Package

While the recommended way to create a Vaadin project and install the libraries is to use an IDE plugin or a dependency management system, such as Maven, Vaadin is also available as a ZIP distribution package.

You can download the newest Vaadin installation package from the download page at <http://vaadin.com/download/>. Please use a ZIP decompression utility available in your operating system to extract the files from the ZIP package.

3.8.1. Package Contents

`README.TXT`

This README file gives simple instructions for installing Vaadin in your project.

`release-notes.html`

The Release Notes contain information about the new features in the particular release, give upgrade instructions, describe compatibility, etc. Please open the HTML file with a web browser.

`license.html`

Apache License version 2.0. Please open the HTML file with a web browser.

`lib` folder

All dependency libraries required by Vaadin are contained within the `lib` folder.

`*.jar`

Vaadin libraries, as described in Section 3.2, “Vaadin Libraries”.

3.8.2. Installing the Libraries

You can install the Vaadin ZIP package in a few simple steps:

1. Copy the JAR files at the package root folder to the `WEB-APP/lib` web library folder in the project. Some of the libraries are optional, as explained in Section 3.2, “Vaadin Libraries”.
2. Also copy the dependency JAR files at the `lib` folder to the `WEB-APP/lib` web library folder in the project.

The location of the `WEB-APP/lib` folder depends on the project organization, which depends on the development environment.

- In Eclipse Dynamic Web Application projects: `WebContent/WEB-INF/lib`.
- In Maven projects: `src/main/webapp/WEB-INF/lib`.

3.9. Using Vaadin with Scala

You can use Vaadin with any JVM compatible language, such as Scala or Groovy. There are, however, some caveats related to libraries and project set-up. In the following, we give instructions for creating a Scala UI in Eclipse, with the Scala IDE for Eclipse and the Vaadin Plugin for Eclipse.

1. Install the Scala IDE for Eclipse, either from an Eclipse update site or as a bundled Eclipse distribution.
2. Open an existing Vaadin Java project or create a new one as outlined in Section 3.4, “Creating and Running a Project in Eclipse”. You can delete the UI class created by the wizard.
3. Switch to the Scala perspective by clicking the perspective in the upper-right corner of the Eclipse window.
4. Right-click on the project folder in **Project Explorer** and select **Configure** → **Add Scala Nature**.
5. The web application needs `scala-library.jar` in its class path. If using Scala IDE, you can copy it from somewhere under your Eclipse installation to the class path of the web application, that is, either to the `WebContent/WEB-INF/lib` folder in the project or to the library path of the application server. If copying outside Eclipse to a project, refresh the project by selecting it and pressing **F5**.

You could also get it with an Ivy or Maven dependency, just make sure that the version is same as what the Scala IDE uses.

You should now be able to create a Scala UI class, such as the following:

```
@Theme("mytheme")
class MyScalaUI extends UI {
  override def init(request: VaadinRequest) = {
    val content: VerticalLayout = new VerticalLayout
    setContent(content)

    val label: Label = new Label("Hello, world!")
    content.addComponent(label)

    // Handle user interaction
    content.addComponent(new Button("Click Me!",
      new ClickListener {
        override def buttonClick(event: ClickEvent) =
          Notification.show("The time is " + new Date)
      })
  }
}
```

Eclipse and Scala IDE should be able to import the Vaadin classes automatically when you press **Ctrl+Shift+O**.

You need to define the Scala UI class either in a servlet class (in Servlet 3.0 project) or in a `web.xml` deployment descriptor, just like described in Section 3.4.2, “Exploring the Project” for Java UIs.

Chapter 4

Architecture

4.1. Overview	71
4.2. Technological Background	74
4.3. Client-Side Engine	77
4.4. Events and Listeners	78

In Chapter 1, *Introduction*, we gave a short introduction to the general architecture of Vaadin. This chapter looks deeper into the architecture at a more technical level.

4.1. Overview

Vaadin provides two development models for web applications: for the client-side (the browser) and for the server-side. The server-driven development model is the more powerful one, allowing application development solely on the server-side, by utilizing an AJAX-based Vaadin Client-Side Engine that renders the user interface in the browser. The client-side model allows developing widgets and applications in Java, which are compiled to JavaScript and executed in the browser. The two models can share their UI widgets, themes, and back-end code and services, and can be mixed together easily.

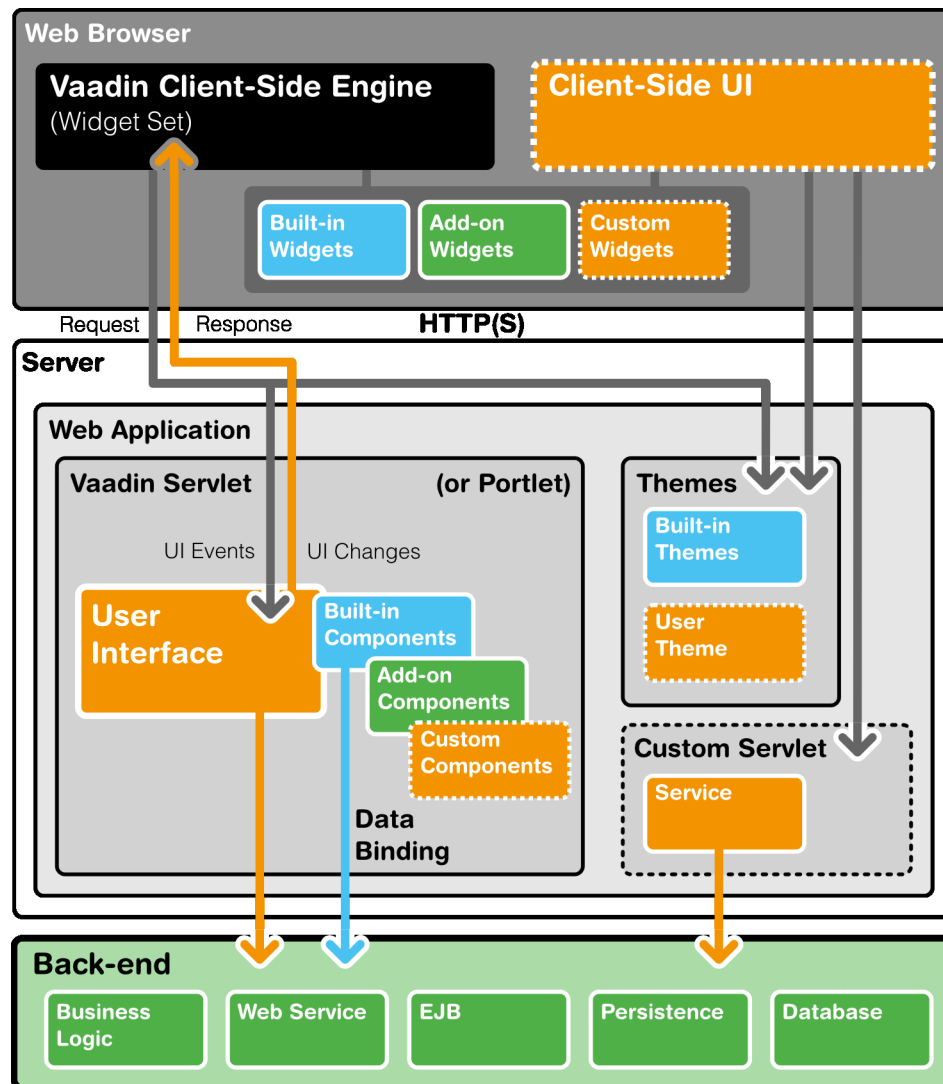
Figure 4.1. Vaadin Runtime Architecture

Figure 4.1, “Vaadin Runtime Architecture” gives a basic illustration of the client-side and server-side communications, in a running situation where the page with the client-side code (engine or application) has been initially loaded in the browser.

Vaadin Framework consists of a *server-side API*, a *client-side API*, a horde of *user interface components/widgets* on the both sides, *themes* for controlling the appearance, and a *data model* that allows binding the server-side components directly to data. For client-side development, it includes the Vaadin Compiler, which allows compiling Java to JavaScript.

A server-side Vaadin application runs as a servlet in a Java web server, serving HTTP requests. The **VaadinServlet** is normally used as the servlet class. The servlet receives client requests and interprets them as events for a particular user session. Events are associated with user interface components and delivered to the event listeners defined in the application. If the UI logic makes changes to the server-side user interface components, the servlet renders them in the web browser by generating a response. The client-side engine running in the browser receives the responses and uses them to make any necessary changes to the page in the browser.

The major parts of the server-driven development architecture and their function are as follows:

User Interface

Vaadin applications provide a user interface for the user to interface with the business logic and data of the application. At technical level, the UI is realized as a *UI* class that extends **com.vaadin.ui.UI**. Its main task is to create the initial user interface out of UI components and set up event listeners to handle user input. The UI can then be loaded in the browser using an URL, or can be embedded to any HTML page. For detailed information about implementing a **UI**, see Chapter 5, *Writing a Server-Side Web Application*.

Please note that the term "UI" is used throughout this book to refer both to the general UI concept as well as the technical UI class concept.

User Interface Components/Widgets

The user interface of a Vaadin application consists of components that are created and laid out by the application. Each server-side component has a client-side counterpart, a *widget*, by which it is rendered in the browser and with which the user interacts. The client-side widgets can also be used by client-side applications. The server-side components relay these events to the application logic. Field components that have a value, which the user can view or edit, can be bound to a data source (see below). For a more detailed description of the UI component architecture, see Chapter 6, *User Interface Components*.

Client-Side Engine

The Client-Side Engine of Vaadin manages the rendering of the UI in the web browser by employing various client-side *widgets*, counterparts of the server-side components. It communicates user interaction to the server-side, and then again renders the changes in the UI. The communications are made using asynchronous HTTP or HTTPS requests. See Section 4.3, "Client-Side Engine".

Vaadin Servlet

Server-side Vaadin applications work on top of the Java Servlet API (see Section 4.2.5, "Java Servlets"). The Vaadin servlet, or more exactly the **VaadinServlet** class, receives requests from different clients, determines which user session they belong to by tracking the sessions with cookies, and delegates the requests to their corresponding sessions. You can customize the Vaadin servlet by extending it.

Themes

Vaadin makes a separation between the appearance and component structure of the user interface. While the UI logic is handled as Java code, the presentation is defined in *themes* as CSS or Sass. Vaadin provides a number of default themes. User themes can, in addition to style sheets, include HTML templates that define custom layouts and other resources, such as images and fonts. Themes are discussed in detail in Chapter 9, *Themes*.

Events

Interaction with user interface components creates events, which are first processed on the client-side by the widgets, then passed all the way through the HTTP server, Vaadin servlet, and the user interface components to the event listeners defined in the application. See Section 4.4, "Events and Listeners".

Server Push

In addition to the event-driven programming model, Vaadin supports server push, where the UI changes are pushed directly from the server to the client without a client

request or an event. This makes it possible to update UIs immediately from other threads and other UIs, without having to wait for a request. See Section 12.16, “Server Push”.

Data Binding

In addition to the user interface model, Vaadin provides a *data model* for binding data presented in field components, such as text fields, check boxes and selection components, to a data source. Using the data model, the user interface components can update the application data directly, often without the need for any control code. All the field components in Vaadin use this data model internally, but any of them can be bound to a separate data source as well. For example, you can bind a table component to an SQL query response. For a complete overview of the Vaadin Data Model, please refer to Chapter 10, *Binding Components to Data*.

Client-Side Applications

In addition to server-side web applications, Vaadin supports client-side application modules, which run in the browser. Client-side modules can use the same widgets, themes, and back-end services as server-side Vaadin applications. They are useful when you have a need for highly responsive UI logic, such as for games or for serving a large number of clients with possibly stateless server-side code, and for various other purposes, such as offering an off-line mode for server-side applications. Please see Chapter 15, *Client-Side Applications* for further details.

Back-end

Vaadin is meant for building user interfaces, and it is recommended that other application layers should be kept separate from the UI. The business logic can run in the same servlet as the UI code, usually separated at least by a Java API, possibly as EJBs, or distributed to a remote back-end service. The data storage is usually distributed to a database management system, and is typically accessed through a persistence solution, such as JPA.

4.2. Technological Background

This section provides an introduction to the various technologies and designs, which Vaadin is based on. This knowledge is not necessary for using Vaadin, but provides some background if you need to make low-level extensions to Vaadin.

4.2.1. HTML and JavaScript

The World Wide Web, with all its websites and most of the web applications, is based on the use of the Hypertext Markup Language (HTML). HTML defines the structure and formatting of web pages, and allows inclusion of graphics and other resources. It is based on a hierarchy of elements marked with start and end tags, such as `<div> ... </div>`. Vaadin uses HTML version 5, although conservatively, to the extent supported by the major browsers, and their currently most widely used versions.

JavaScript, on the other hand, is a programming language for embedding programs in HTML pages. JavaScript programs can manipulate a HTML page through the Document Object Model (DOM) of the page. They can also handle user interaction events. The Client-Side Engine of Vaadin and its client-side widgets do exactly this, although it is actually programmed in Java, which is compiled to JavaScript with the Vaadin Client Compiler.

Vaadin largely hides the use of HTML, allowing you to concentrate on the UI component structure and logic. In server-side development, the UI is developed in Java using UI components and

rendered by the client-side engine as HTML, but it is possible to use HTML templates for defining the layout, as well as HTML formatting in many text elements. Also when developing client-side widgets and UIs, the built-in widgets in the framework hide most of HTML DOM manipulation.

4.2.2. Styling with CSS and Sass

While HTML defines the content and structure of a web page, *Cascading Style Sheet* (CSS) is a language for defining the visual style, such as colors, text sizes, and margins. CSS is based on a set of rules that are matched with the HTML structure by the browser. The properties defined in the rules determine the visual appearance of the matching HTML elements.

```
/* Define the color of labels in my view */
.myview .v-label {
  color: blue;
}
```

Sass, or *Syntactically Awesome Stylesheets*, is an extension of the CSS language, which allows the use of variables, nesting, and many other syntactic features that make the use of CSS easier and clearer. Sass has two alternative formats, SCSS, which is a superset of the syntax of CSS3, and an older indented syntax, which is more concise. The Vaadin Sass compiler supports the SCSS syntax.

Vaadin handles styling with *themes* defined with CSS or Sass, and associated images, fonts, and other resources. Vaadin themes are specifically written in Sass. In development mode, Sass files are compiled automatically to CSS. For production use, you compile the Sass files to CSS with the included compiler. The use of themes is documented in detail in Chapter 9, *Themes*, which also gives an introduction to CSS and Sass.

4.2.3. AJAX

AJAX, short for Asynchronous JavaScript and XML, is a technique for developing web applications with responsive user interaction, similar to traditional desktop applications. Conventional web applications, be they JavaScript-enabled or not, can get new page content from the server only by loading an entire new page. AJAX-enabled pages, on the other hand, handle the user interaction in JavaScript, send a request to the server asynchronously (without reloading the page), receive updated content in the response, and modify the page accordingly. This way, only small parts of the page data need to be loaded. This goal is achieved by the use of a certain set of technologies: HTML, CSS, DOM, JavaScript, and the XMLHttpRequest API in JavaScript. XML is just one way to serialize data between the client and the server, and in Vaadin it is serialized with the more efficient JSON.

The asynchronous requests used in AJAX are made possible by the XMLHttpRequest class in JavaScript. The API feature is available in all major browsers and is under way to become a W3C standard.

The communication of complex data between the browser and the server requires some sort of *serialization* (or *marshalling*) of data objects. The Vaadin servlet and the client-side engine handle the serialization of shared state objects from the server-side components to the client-side widgets, as well as serialization of RPC calls between the widgets and the server-side components.

4.2.4. Google Web Toolkit

The client-side framework of Vaadin is based on the Google Web Toolkit (GWT). Its purpose is to make it possible to develop web user interfaces that run in the browser easily with Java instead

of JavaScript. Client-side modules are developed with Java and compiled into JavaScript with the Vaadin Compiler, which is an extension of the GWT Compiler. The client-side framework also hides much of the HTML DOM manipulation and enables handling browser events in Java.

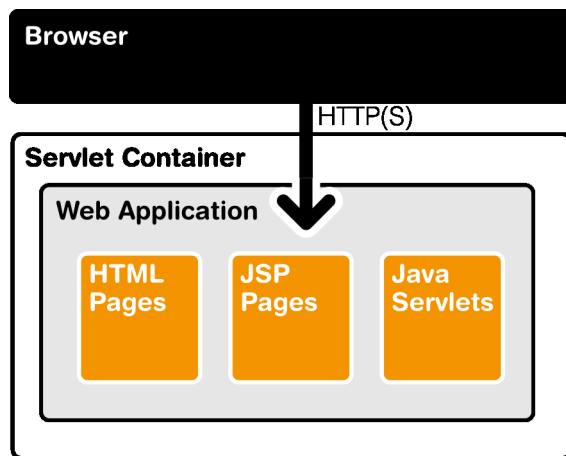
GWT is essentially a client-side technology, normally used to develop user interface logic in the web browser. Pure client-side modules still need to communicate with a server using RPC calls and by serializing any data. The server-driven development mode in Vaadin effectively hides all the client-server communications and allows handling user interaction logic in a server-side application. This makes the architecture of an AJAX-based web application much simpler. Nevertheless, Vaadin also allows developing pure client-side applications, as described in Chapter 15, *Client-Side Applications*.

See Section 4.3, “Client-Side Engine” for a description of how the client-side framework based on GWT is used in the Client-Side Engine of Vaadin. Chapter 14, *Client-Side Vaadin Development* provides information about the client-side development, and Chapter 17, *Integrating with the Server-Side* about the integration of client-side widgets with the server-side components.

4.2.5. Java Servlets

A Java Servlet is a class that is executed in a Java web server (a *Servlet container*) to extend the capabilities of the server. In practice, it is normally a part of a *web application*, which can contain HTML pages to provide static content, and JavaServer Pages (JSP) and Java Servlets to provide dynamic content. This is illustrated in Figure 4.2, “Java Web Applications and Servlets”.

Figure 4.2. Java Web Applications and Servlets



Web applications are usually packaged and deployed to a server as *WAR* (*Web application ARchive*) files, which are Java JAR packages, which in turn are ZIP compressed packages. The web application is defined in a `WEB-INF/web.xml` deployment descriptor, which defines the servlet classes and also the mappings from request URL paths to the servlets. This is described in more detail in Section 5.9.4, “Using a `web.xml` Deployment Descriptor”. The class path for the servlets and their dependencies includes the `WEB-INF/classes` and `WEB-INF/lib` folders. The `WEB-INF` is a special hidden folder that can not be accessed by its URL path.

The servlets are Java classes that handle HTTP requests passed to them by the server through the *Java Servlet API*. They can generate HTML or other content as a response. JSP pages, on the other hand, are HTML pages, which allow including Java source code embedded in the pages. They are actually translated to Java source files by the container and then compiled to servlets.

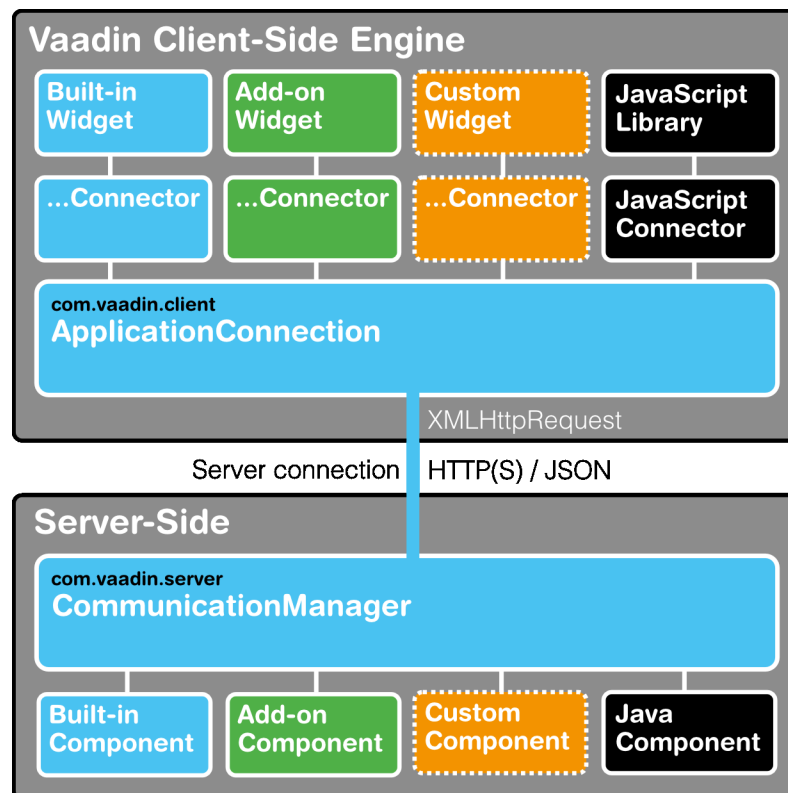
The UIs of server-side Vaadin applications run as servlets. They are wrapped inside a **Vaadin-Servlet** servlet class, which handles session tracking and other tasks. On the initial request, it returns an HTML loader page and then mostly JSON responses to synchronize the widgets and their server-side counterparts. It also serves various resources, such as themes. The server-side UIs are implemented as classes extending the **UI** class, as described in Chapter 5, *Writing a Server-Side Web Application*. The class is given as a parameter to the Vaadin Servlet in the `web.xml` deployment descriptor.

The Vaadin Client-Side Engine as well as client-side Vaadin applications are loaded to the browser as static JavaScript files. The client-side engine, or widget set in technical terms, needs to be located under the `VAADIN/widgetsets` path in the web application. The precompiled default widget set is served from the `vaadin-client-compiled` JAR by the Vaadin Servlet.

4.3. Client-Side Engine

The user interface of a server-side Vaadin application is rendered in the browser by the Vaadin Client-Side Engine. It is loaded in the browser when the page with the Vaadin UI is opened. The server-side UI components are rendered using *widgets* (as they are called in Google Web Toolkit) on the client-side. The client-side engine is illustrated in Figure 4.3, “Vaadin Client-Side Engine”.

Figure 4.3. Vaadin Client-Side Engine



The client-side framework includes two kinds of built-in widgets: GWT widgets and Vaadin-specific widgets. The two widget collections have significant overlap, where the Vaadin widgets provide a bit different features than the GWT widgets. In addition, many add-on widgets and their server-side counterparts exist, and you can easily download and install them, as described

in Chapter 18, *Using Vaadin Add-ons*. You can also develop your own widgets, as described in Chapter 14, *Client-Side Vaadin Development*.

The rendering with widgets, as well as the communication to the server-side, is handled in the **ApplicationConnection**. Connecting the widgets with their server-side counterparts is done in *connectors*, and there is one for each widget that has a server-side counterpart. The framework handles serialization of component state transparently, and includes an RPC mechanism between the two sides. Integration of widgets with their server-side counterpart components is described in Chapter 17, *Integrating with the Server-Side*.

4.4. Events and Listeners

Vaadin offers an event-driven programming model for handling user interaction. When a user does something in the user interface, such as clicks a button or selects an item, the application needs to know about it. Many Java-based user interface frameworks follow the *Event-Listener pattern* (also known as the Observer design pattern) to communicate user input to the application logic. So does Vaadin. The design pattern involves two kinds of elements: an object that generates ("fires" or "emits") events and a number of listeners that listen for the events. When such an event occurs, the object sends a notification about it to all the listeners. In a typical case, there is only one listener.

Events can serve many kinds of purposes. In Vaadin, the usual purpose of events is handling user interaction in a user interface. Session management can require special events, such as time-out, in which case the event would actually be the lack of user interaction. Time-out is a special case of timed or scheduled events, where an event occurs at a specific date and time or when a set time has passed.

To receive events of a particular type, an application must register a listener object with the event source. The listeners are registered in the components with an `add*Listener()` method (with a method name specific to the listener).

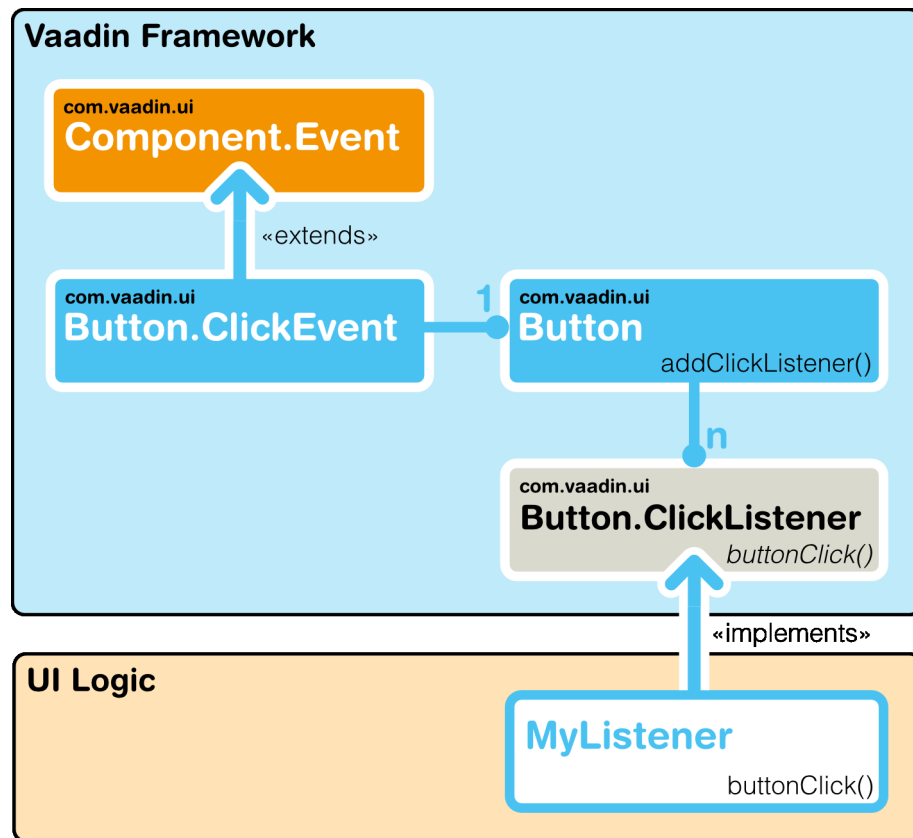
Most components that have related events define their own event class and the corresponding listener class. For example, the **Button** has **Button.ClickEvent** events, which can be listened to through the **Button.ClickListener** interface.

In the following, we handle button clicks with a listener implemented as an anonymous class:

```
final Button button = new Button("Push it!");

button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        button.setCaption("You pushed it!");
    }
});
```

Figure 4.4, "Class Diagram of a Button Click Listener" illustrates the case where an application-specific class inherits the **Button.ClickListener** interface to be able to listen for button click events. The application must instantiate the listener class and register it with `addClickListener()`. It can be an anonymous class, such as the one above. When an event occurs, an event object is instantiated, in this case a **Button.ClickEvent**. The event object knows the related UI component, in this case the **Button**.

Figure 4.4. Class Diagram of a Button Click Listener

In the ancient times of C programming, *callback functions* filled largely the same need as listeners do now. In object-oriented languages, we usually only have classes and methods, not functions, so the application has to give a class interface instead of a callback function pointer to the framework.

Section 5.4, “Handling Events with Listeners” goes into details of handling events in practice.

Chapter 5

Writing a Server-Side Web Application

5.1. Overview	81
5.2. Building the UI	84
5.3. Designing UIs Declaratively	88
5.4. Handling Events with Listeners	93
5.5. Images and Other Resources	96
5.6. Handling Errors	100
5.7. Notifications	100
5.8. Application Lifecycle	103
5.9. Deploying an Application	108

This chapter provides the fundamentals of server-side web application development with Vaadin, concentrating on the basic elements of an application from a practical point-of-view.

5.1. Overview

A server-side Vaadin application runs as a Java Servlet in a servlet container. The Java Servlet API is, however, hidden behind the framework. The user interface of the application is implemen-

ted as a *UI* class, which needs to create and manage the user interface components that make up the user interface. User input is handled with event listeners, although it is also possible to bind the user interface components directly to data. The visual style of the application is defined in themes as CSS or Sass. Icons, other images, and downloadable files are handled as *resources*, which can be external or served by the application server or the application itself.

Figure 5.1. Server-Side Application Architecture

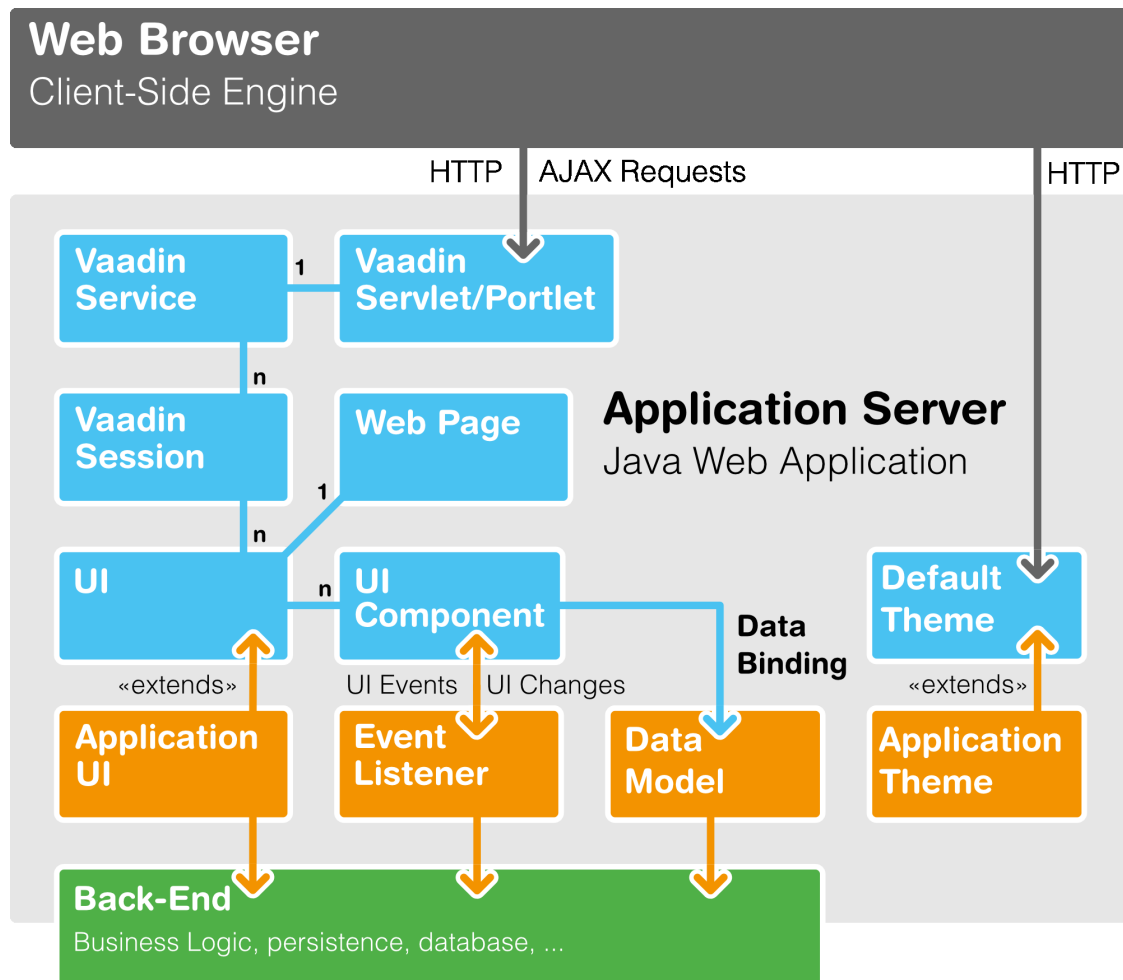


Figure 5.1, “Server-Side Application Architecture” illustrates the basic architecture of an application made with the Vaadin Framework, with all the major elements, which are introduced below and discussed in detail in this chapter.

First of all, a Vaadin application must have one or more UI classes that extend the abstract **com.vaadin.ui.UI** class and implement the **init()** method. A custom theme can be defined as an annotation for the UI.

```
@Theme("hellotheme")
public class HelloWorld extends UI {
    protected void init(VaadinRequest request) {
        ... initialization code goes here ...
    }
}
```


A UI is a viewport to a Vaadin application running in a web page. A web page can actually have multiple such UIs within it. Such situation is typical especially with portlets in a portal. An application can run in multiple browser windows, each having a distinct **UI** instance. The UIs of an application can be the same UI class or different.

Vaadin framework handles servlet requests internally and associates the requests with user sessions and a UI state. Because of this, you can develop Vaadin applications much like you would develop desktop applications.

The most important task in the initialization is the creation of the initial user interface. This, and the deployment of a UI as a Java Servlet in the Servlet container, as described in Section 5.9, “Deploying an Application”, are the minimal requirements for an application.

Below is a short overview of the other basic elements of an application besides UI:

UI

A *UI* represents an HTML fragment in which a Vaadin application runs in a web page. It typically fills the entire page, but can also be just a part of a page. You normally develop a Vaadin application by extending the **UI** class and adding content to it. A UI is essentially a viewport connected to a user session of an application, and you can have many such views, especially in a multi-window application. Normally, when the user opens a new page with the URL of the Vaadin UI, a new **UI** (and the associated **Page** object) is automatically created for it. All of them share the same user session.

The current UI object can be accessed globally with `UI.getCurrent()`. The static method returns the thread-local UI instance for the currently processed request.

Page

A **UI** is associated with a **Page** object that represents the web page as well as the browser window in which the UI runs.

The **Page** object for the currently processed request can be accessed globally from a Vaadin application with `Page.getCurrent()`. This is equivalent to calling `UI.getCurrent().getPage()`.

Vaadin Session

A **VaadinSession** object represents a user session with one or more UIs open in the application. A session starts when a user first opens a UI of a Vaadin application, and closes when the session expires in the server or when it is closed explicitly.

User Interface Components

The user interface consists of components that are created by the application. They are laid out hierarchically using special *layout components*, with a content root layout at the top of the hierarchy. User interaction with the components causes *events* related to the component, which the application can handle. *Field components* are intended for inputting values and can be directly bound to data using the Vaadin Data Model. You can make your own user interface components through either inheritance or composition. For a thorough reference of user interface components, see Chapter 6, *User Interface Components*, for layout components, see Chapter 7, *Managing Layout*, and for compositing components, see Section 6.31, “Composition with **CustomComponent**”.

Events and Listeners

Vaadin follows an event-driven programming paradigm, in which events, and listeners that handle the events, are the basis of handling user interaction in an application (although also server push is possible as described in Section 12.16, “Server Push”). Section 4.4, “Events and Listeners” gave an introduction to events and listeners from an architectural point-of-view, while Section 5.4, “Handling Events with Listeners” later in this chapter takes a more practical view.

Resources

A user interface can display images or have links to web pages or downloadable documents. These are handled as *resources*, which can be external or provided by the web server or the application itself. Section 5.5, “Images and Other Resources” gives a practical overview of the different types of resources.

Themes

The presentation and logic of the user interface are separated. While the UI logic is handled as Java code, the presentation is defined in *themes* as CSS or SCSS. Vaadin includes some built-in themes. User-defined themes can, in addition to style sheets, include HTML templates that define custom layouts and other theme resources, such as images. Themes are discussed in detail in Chapter 9, *Themes*, custom layouts in Section 7.14, “Custom Layouts”, and theme resources in Section 5.5.4, “Theme Resources”.

Data Binding

Field components are essentially views to data, represented in the *Vaadin Data Model*. Using the data model, the components can get their values from and update user input to the data model directly, without the need for any control code. A field component is always bound to a *property* and a group of fields to an *item* that holds the properties. Items can be collected in a *container*, which can act as a data source for some components such as tables or lists. While all the components have a default data model, they can be bound to a user-defined data source. For example, you can bind a **Table** component to an SQL query response. For a complete overview of data binding in Vaadin, please refer to Chapter 10, *Binding Components to Data*.

5.2. Building the UI

Vaadin user interfaces are built hierarchically from components, so that the leaf components are contained within layout components and other component containers. Building the hierarchy starts from the top (or bottom - whichever way you like to think about it), from the **UI** class of the application. You normally set a layout component as the content of the UI and fill it with other components.

```
public class MyHierarchicalUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // The root of the component hierarchy
        VerticalLayout content = new VerticalLayout();
        content.setSizeFull(); // Use entire window
        setContent(content);   // Attach to the UI

        // Add some component
        content.addComponent(new Label("Hello!"));

        // Layout inside layout
        HorizontalLayout hor = new HorizontalLayout();
```

```

        hor.setSizeFull(); // Use all available space

        // Couple of horizontally laid out components
        Tree tree = new Tree("My Tree",
            TreeExample.createTreeContent());
        hor.addComponent(tree);

        Table table = new Table("My Table",
            TableExample.generateContent());
        table.setSizeFull();
        hor.addComponent(table);
        hor.setExpandRatio(table, 1); // Expand to fill

        content.addComponent(hor);
        content.setExpandRatio(hor, 1); // Expand to fill
    }
}

```

The component hierarchy could be illustrated with a tree as follows:

```

UI
|-- VerticalLayout
|   |-- Label
|   |-- HorizontalLayout
|       |-- Tree
|       |-- Table

```

The result is shown in Figure 5.2, “Simple Hierarchical UI”.

Figure 5.2. Simple Hierarchical UI

name	city	year
Charles Darwin	Oxford	1 994
Charles Adams	London	1 884
Isaac Lovelace	Oxford	1 928
Charles Newton	Oxford	1 818
Charles Adams	London	1 883
Charles Adams	London	1 883

Instead of building the layout in Java, you can also use a declarative design, as described later in Section 5.3, “Designing UIs Declaratively”. The examples given for the declarative layouts give exactly the same UI layout as built from the components above.

The built-in components are described in Chapter 6, *User Interface Components* and the layout components in Chapter 7, *Managing Layout*.

The example application described above just is, it does not do anything. User interaction is handled with event listeners, as described a bit later in Section 5.4, “Handling Events with Listeners”.

5.2.1. Application Architecture

Once your application grows beyond a dozen or so lines, which is usually quite soon, you need to start considering the application architecture more closely. You are free to use any object-oriented techniques available in Java to organize your code in methods, classes, packages, and libraries. An architecture defines how these modules communicate together and what sort of dependencies they have between them. It also defines the scope of the application. The scope of this book, however, only gives a possibility to mention some of the most common architectural patterns in Vaadin applications.

The subsequent sections describe some basic application patterns. For more information about common architectures, see Section 12.10, “Advanced Application Architectures”, which discusses layered architectures, the Model-View-Presenter (MVP) pattern, and so forth.

5.2.2. Compositing Components

User interfaces typically contain many user interface components in a layout hierarchy. Vaadin provides many layout components for laying contained components vertically, horizontally, in a grid, and in many other ways. You can extend layout components to create composite components.

```
class MyView extends VerticalLayout {
    TextField entry = new TextField("Enter this");
    Label display = new Label("See this");
    Button click = new Button("Click This");

    public MyView() {
        addComponent(entry);
        addComponent(display);
        addComponent(click);

        // Configure it a bit
        setSizeFull();
        addStyleName("myview");
    }
}

// Use it
Layout myview = new MyView();
```

This composition pattern is especially supported for creating forms, as described in Section 10.4.3, “Binding Member Fields”.

While extending layouts is an easy way to make component composition, it is a good practice to encapsulate implementation details, such as the exact layout component used. Otherwise, the users of such a composite could begin to rely on such implementation details, which would make changes harder. For this purpose, Vaadin has a special **CustomComponent** wrapper, which hides the content representation.

```
class MyView extends CustomComponent {
    TextField entry = new TextField("Enter this");
    Label display = new Label("See this");
```

```

Button    click    = new Button("Click This");

public MyView() {
    Layout layout = new VerticalLayout();

    layout.addComponent(entry);
    layout.addComponent(display);
    layout.addComponent(click);

    setCompositionRoot(layout);

    setSizeFull();
}

// Use it
MyView myview = new MyView();

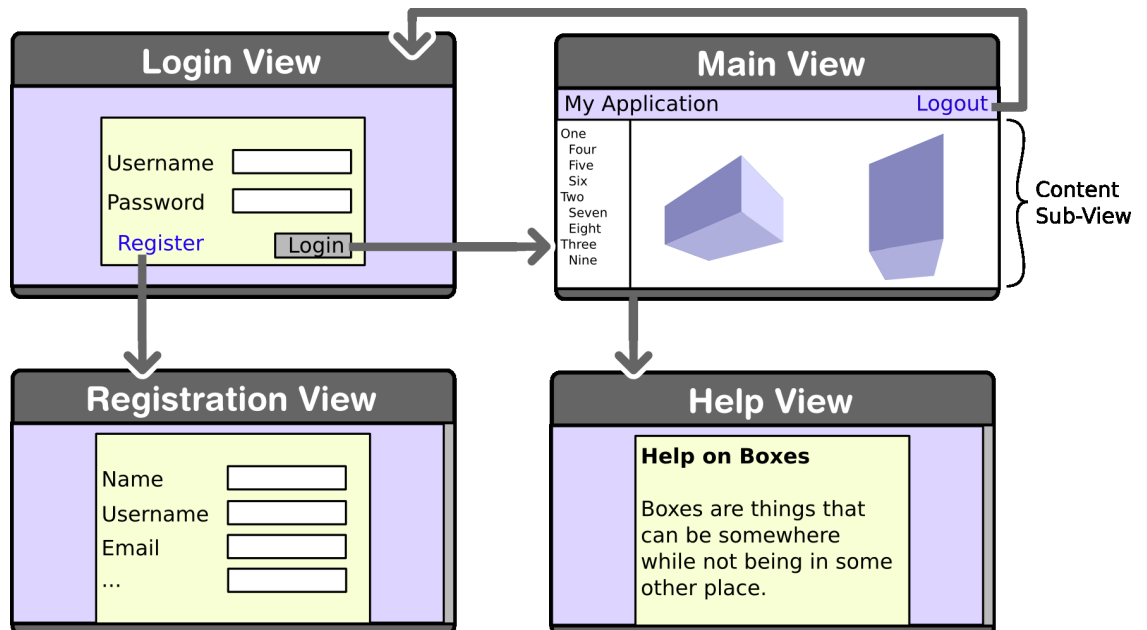
```

For a more detailed description of the **CustomComponent**, see Section 6.31, “Composition with **CustomComponent**”.

5.2.3. View Navigation

While the most simple applications have just a single *view* (or *screen*), perhaps most have many. Even in a single view, you often want to have sub-views, for example to display different content. Figure 5.3, “Navigation Between Views” illustrates a typical navigation between different top-level views of an application, and a main view with sub-views.

Figure 5.3. Navigation Between Views



The **Navigator** described in Section 12.9, “Navigating in an Application” is a view manager that provides a flexible way to navigate between views and sub-views, while managing the URI fragment in the page URL to allow bookmarking, linking, and going back in browser history.

Often Vaadin application views are part of something bigger. In such cases, you may need to integrate the Vaadin applications with the other website. You can use the embedding techniques described in Section 12.2, “Embedding UIs in Web Pages”.

5.2.4. Accessing UI, Page, Session, and Service

You can get the UI and the page to which a component is attached to with `getUI()` and `getPage()`.

However, the values are `null` until the component is attached to the UI, and typically, when you need it in constructors, it is not. It is therefore preferable to access the current UI, page, session, and service objects from anywhere in the application using the static `getCurrent()` methods in the respective **UI**, **Page**, **VaadinSession**, and **VaadinService** classes.

```
// Set the default locale of the UI
UI.getCurrent().setLocale(new Locale("en"));

// Set the page title (window or tab caption)
Page.getCurrent().setTitle("My Page");

// Set a session attribute
VaadinSession.getCurrent().setAttribute("myattrib", "hello");

// Access the HTTP service parameters
File baseDir = VaadinService.getCurrent().getBaseDirectory();
```

You can get the page and the session also from a **UI** with `getPage()` and `getSession()` and the service from **VaadinSession** with `getService()`.

The static methods use the built-in `ThreadLocal` support in the classes.

5.3. Designing UIs Declaratively

Declarative definition of composites and even entire UIs makes it easy for developers and especially graphical designers to work on visual designs without any coding. Designs can be modified even while the application is running, as can be the associated themes. A design is a representation of a component hierarchy, which can be accessed from Java code to implement dynamic UI logic, as well as data binding.

For example, considering the following layout in Java:

```
VerticalLayout vertical = new VerticalLayout ();
vertical.addComponent(new TextField("Name"));
vertical.addComponent(new TextField("Street address"));
vertical.addComponent(new TextField("Postal code"));
layout.addComponent(vertical);
```

You could define it declaratively with the following equivalent design:

```
<vaadin-vertical-layout>
  <vaadin-text-field caption="Name"/>
  <vaadin-text-field caption="Street address"/>
  <vaadin-text-field caption="Postal code"/>
</vaadin-vertical-layout>
```

Declarative designs can be crafted by hand, but are most conveniently created with the Vaadin Designer.

In the following, we first go through the syntax of the declarative design files, and then see how to use them in applications by binding them to data and handling user interaction events.

5.3.1. Declarative Syntax

A design is an HTML document with custom elements for representing components and their configuration. A design has a single root component inside the HTML body element. Enclosing `<html>`, `<head>`, and `<body>` are optional, but necessary if you need to make namespace definitions for custom components. Other regular HTML elements may not be used in the file, except inside components that specifically accept HTML content.

In a design, each nested element corresponds to a Vaadin component in a component tree. Components can have explicitly given IDs to enable binding them to variables in the Java code, as well as optional attributes.

```
<!DOCTYPE html>
<html>
  <body>
    <vaadin-vertical-layout size-full>
      <!-- Label with HTML content -->
      <vaadin-label><b>Hello!</b> -
        How are you?</vaadin-label>

      <vaadin-horizontal-layout size-full :expand>
        <vaadin-tree _id="mytree" caption="My Tree"
          width-auto height-full/>
        <vaadin-table _id="mytable" caption="My Table"
          size-full :expand/>
      </vaadin-horizontal-layout>
    </vaadin-vertical-layout>
  </body>
</html>
```

The DOCTYPE is not required, neither is the `<html>`, or `<body>` elements. Nevertheless, there may only be one design root element.

The above design defines the same UI layout as done earlier with Java code, and illustrated in Figure 5.2, “Simple Hierarchical UI”.

5.3.2. Component Elements

HTML elements of the declarative syntax are directly mapped to Vaadin components according to their Java class names. The tag of a component element has a namespace prefix separated by a dash. Vaadin core components, which are defined in the `com.vaadin.ui` package, have `vaadin-` prefix. The rest of an element tag is determined from the Java class name of the component, by making it lower-case, while adding a dash (-) before every previously upper-case letter as a word separator. For example, **ComboBox** component has declarative element tag `vaadin-combo-box`.

Component Prefix to Package Mapping

You can use any components in a design: components extending Vaadin components, composite components, and add-on components. To do so, you need to define a mapping from an element prefix to the Java package of the component. The prefix is used as a sort of a namespace.

The mappings are defined in `<meta name="package-mapping" ...>` elements in the HTML head. A *content* attribute defines a mapping, in notation with a prefix separated from the corresponding Java package name with a colon, such as `my:com.example.myapp`.

For example, consider that you have the following composite class **`com.example.myapp.ExampleComponent`**:

```
package com.example.myapp;

public class ExampleComponent extends CustomComponent {
    public ExampleComponent() {
        setCompositionRoot(new Label("I am an example."));
    }
}
```

You would make the package prefix mapping and then use the component as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="package-mapping"
          content="my:com.example.myapp" />
  </head>

  <body>
    <vaadin-vertical-layout>
      <vaadin-label><b>Hello!</b> -
        How are you?</vaadin-label>

      <!-- Use it here -->
      <my-example-component/>
    </vaadin-vertical-layout>
  </body>
</html>
```

Inline Content and Data

The element content can be used for certain default attributes, such as a button caption. For example:

```
<vaadin-button><b>OK</b></vaadin-button>
```

Some components, such as selection components, allow defining inline data within the element. For example:

```
<vaadin-native-select>
  <option>Mercury</option>
  <option>Venus</option>
  <option selected>Earth</option>
</vaadin-native-select>
```

The declarative syntax of each component type is described in the JavaDoc API documentation of Vaadin.

5.3.3. Component Attributes

Attribute-to-Property Mapping

Component properties are directly mapped to the attributes of the HTML elements according to the names of the properties. Attributes are written in lower-case letters and dash is used for word separation instead of upper-case letters in the Java methods, so that `input-prompt` attribute is equivalent to `setInputPrompt()`.

For example, the `caption` property, which you can set with `setCaption()`, is represented as `caption` attribute. You can find the component properties by the setter methods in the JavaDoc API documentation of the component classes.

```
<vaadin-text-field caption="Name" input-prompt="Enter Name"/>
```

Attribute Values

Attribute parameters must be enclosed in quotes and the value given as a string must be convertible to the type of the property (string, integer, boolean, or enumeration). Object types are not supported.

Some attribute names are given by a shorthand. For example, `alternateText` property of the **Image** component, which you would set with `setAlternateText()`, is given as the `alt` attribute.

Boolean values must be either `true` or `false`. The value can be omitted, in which case `true` is assumed. For example, the `enabled` attribute is boolean and has default value “true”, so `enabled="true"` and `enabled` are equivalent.

```
<vaadin-button enabled="false">OK</vaadin-button>
```

Parent Component Settings

Certain settings, such as a component's alignment in a layout, are not done in the component itself, but in the layout. Attributes prefixed with colon (:) are passed to the containing component, with the component as a target parameter. For example, `:expand="1"` given for a component `c` is equivalent to calling `setExpandRatio(c, 1)` for the containing layout.

```
<vaadin-vertical-layout size-full>
  <!-- Align right in the containing layout -->
  <vaadin-label width-auto :right>Hello!</vaadin-label>

  <!-- Expands to take up all remaining vertical space -->
  <vaadin-horizontal-layout size-full :expand>
    <!-- Automatic width - shrinks horizontally -->
    <vaadin-tree width-auto height-full/>

    <!-- Expands horizontally to take remaining space -->
    <vaadin-table size-full :expand/>
  </vaadin-horizontal-layout>
</vaadin-vertical-layout>
```

Again, compare the above declaration to the Java code given in Section 5.2, “Building the UI”.

5.3.4. Component Identifiers

Components can be identified by either an identifier or a caption. There are two types of identifiers: page-global and local. This allows accessing them from Java code and binding them to components, as described later in Section 5.3.5, “Using Designs in Code”.

The `id` attribute can be used to define a page-global identifier, which must be unique within the page. Another design or UI shown simultaneously in the same page may not have components sharing the same ID. Using global identifiers is therefore not recommended, except in special cases where uniqueness is ensured.

The `_id` attribute defines a local identifier used only within the design. This is the recommended way to identifying components.

```
<vaadin-tree _id="mytree" caption="My Tree"/>
```

5.3.5. Using Designs in Code

The main use of declarative designs is in building application views, sub-views, dialogs, and forms through composition. The two main tasks are filling the designs with application data and handling user interaction events.

Binding to a Design Root

You can bind any component container as the root component of a design with the **@DesignRoot** annotation. The class must match or extend the class of the root element in the design.

The member variables are automatically initialized from the design according to the component identifiers (see Section 5.3.4, “Component Identifiers”), which must match the variable names.

For example, the following class could be used to bind the design given earlier.

```
@DesignRoot
public class MyViewDesign extends VerticalLayout {
    Tree mytree;
    Table mytable;

    public MyViewDesign() {
        Design.read("MyDeclarativeUI.html", this);

        // Show some (example) data
        mytree.setContainerDataSource(
            TreeExample.createTreeContent());
        mytable.setContainerDataSource(
            TableExample.generateContent());

        // Some interaction
        mytree.addItemClickListener(event -> // Java 8
            Notification.show("Selected " +
                event.getItemId()));
    }
}
```

The design root class must match or extend the root element class of the design. For example, earlier we had `<vaadin-vertical-layout>` element in the HTML file, which can be bound to a class extending **VerticalLayout**.

Using a Design

The fact that a component is defined declaratively is not visible in its API, so you can create and use such it just like any other component.

For example, to use the previously defined design root component as the content of the entire UI:

```
public class DeclarativeViewUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new MyViewDesign());
    }
}
```

Designs in View Navigation

To use a design in view navigation, as described in Section 12.9, “Navigating in an Application”, you just need to implement the `View` interface.

```
@DesignRoot
public class MainView extends VerticalLayout
    implements View {
    public MainView() {
        Design.read(this);
        ...
    }
    ...
}

...
// Use the view by precreating it
navigator.addView(MAINVIEW, new MainView());
```

See Section 12.9.3, “Handling URI Fragment Path” for a complete example.

5.4. Handling Events with Listeners

Let us put into practice what we learned of event handling in Section 4.4, “Events and Listeners”. You can implement listener interfaces in a regular class, but it brings the problem with differentiating between different event sources. Using anonymous class for listeners is recommended in most cases.

5.4.1. Using Anonymous Classes

By far the easiest and the most common way to handle events in Java 6 and 7 is to use anonymous local classes. It encapsulates the handling of events to where the component is defined and does not require cumbering the managing class with interface implementations. The following example defines an anonymous class that inherits the **Button.ClickListener** interface.

```
// Have a component that fires click events
final Button button = new Button("Click Me!");

// Handle the events with an anonymous class
button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        button.setCaption("You made me click!");
    }
});
```

```
    }  
});
```

Local objects referenced from within an anonymous class, such as the **Button** object in the above example, must be declared `final`.

Most components allow passing a listener to the constructor, thereby losing a line or two. However, notice that if accessing the component that is constructed from an anonymous class, you must use a reference that is declared before the constructor is executed, for example as a member variable in the outer class. If it is declared in the same expression where the constructor is called, it doesn't yet exist. In such cases, you need to get a reference to the component from the event object.

```
final Button button = new Button("Click It!",  
    new Button.ClickListener() {  
        @Override  
        public void buttonClick(ClickEvent event) {  
            event.getButton().setCaption("Done!");  
        }  
    });
```

5.4.2. Handling Events in Java 8

Java 8 introduced lambda expressions, which offer a replacement for listeners. You can directly use lambda expressions in place of listeners that have only one method to implement.

For example, in the following, we use a lambda expression to handle button click events in the constructor:

```
layout.addComponent(new Button("Click Me!",  
    event -> event.getButton().setCaption("You made click!")));
```

Java 8 is the future that is already here, and as Vaadin API uses event listeners extensively, using lambda expressions makes UI code much more readable.

Directing events to handler methods is easy with method references:

```
public class Java8Buttons extends CustomComponent {  
    public Java8Buttons() {  
        setCompositionRoot(new HorizontalLayout(  
            new Button("OK", this::ok),  
            new Button("Cancel", this::cancel));  
    }  
  
    public void ok(ClickEvent event) {  
        event.getButton().setCaption ("OK!");  
    }  
  
    public void cancel(ClickEvent event) {  
        event.getButton().setCaption ("Not OK!");  
    }  
}
```

5.4.3. Implementing a Listener in a Regular Class

The following example follows a typical pattern where you have a **Button** component and a listener that handles user interaction (clicks) communicated to the application as events. Here we define a class that listens to click events.

```
public class MyComposite extends CustomComponent
    implements Button.ClickListener {
    Button button; // Defined here for access

    public MyComposite() {
        Layout layout = new HorizontalLayout();

        // Just a single component in this composition
        button = new Button("Do not push this");
        button.addClickListener(this);
        layout.addComponent(button);

        setCompositionRoot(layout);
    }

    // The listener method implementation
    public void buttonClick(ClickEvent event) {
        button.setCaption("Do not push this again");
    }
}
```

5.4.4. Differentiating Between Event Sources

If an application receives events of the same type from multiple sources, such as multiple buttons, it has to be able to distinguish between the sources. If using a regular class listener, distinguishing between the components can be done by comparing the source of the event with each of the components. The method for identifying the source depends on the event type.

```
public class TheButtons extends CustomComponent
    implements Button.ClickListener {
    Button onebutton;
    Button toobutton;

    public TheButtons() {
        onebutton = new Button("Button One", this);
        toobutton = new Button("A Button Too", this);

        // Put them in some layout
        Layout root = new HorizontalLayout();
        root.addComponent(onebutton);
        root.addComponent(toobutton);
        setCompositionRoot(root);
    }

    @Override
    public void buttonClick(ClickEvent event) {
        // Differentiate targets by event source
        if (event.getButton() == onebutton)
            onebutton.setCaption("Pushed one");
        else if (event.getButton() == toobutton)
            toobutton.setCaption("Pushed too");
    }
}
```

```
}
}
```

Other techniques exist for separating between event sources, such as using object properties, names, or captions to separate between them. Using captions or any other visible text is generally discouraged, as it may create problems for internationalization. Using other symbolic strings can also be dangerous, because the syntax of such strings is checked only at runtime.

5.5. Images and Other Resources

Web applications can display various *resources*, such as images, other embedded content, or downloadable files, that the browser has to load from the server. Image resources are typically displayed with the **Image** component or as component icons. Flash animations can be displayed with **Flash**, embedded browser frames with **BrowserFrame**, and other content with the **Embedded** component, as described in Section 6.33, “Embedded Resources”. Downloadable files are usually provided by clicking a **Link**.

There are several ways to how such resources can be provided by the web server. Static resources can be provided without having to ask for them from the application. For dynamic resources, the user application must be able to create them dynamically. The resource request interfaces in Vaadin allow applications to both refer to static resources as well as dynamically create them. The dynamic creation includes the **StreamResource** class and the `RequestHandler` described in Section 12.4, “Request Handlers”.

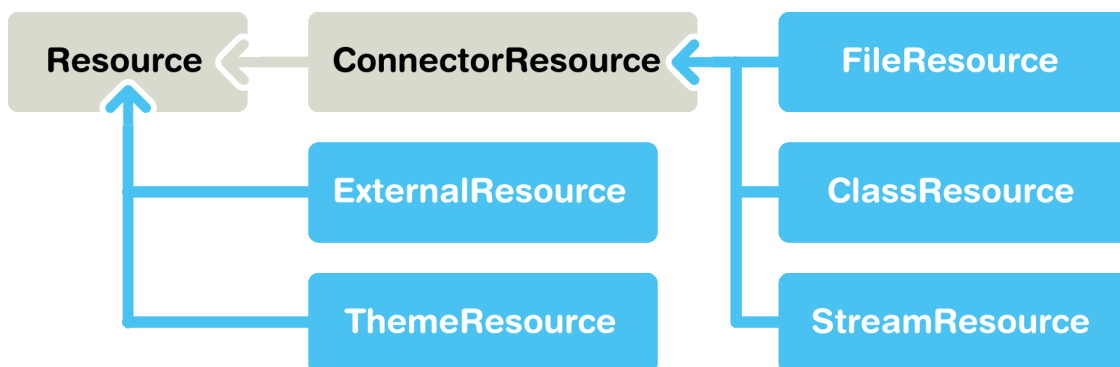
Vaadin also provides low-level facilities for retrieving the URI and other parameters of a HTTP request. We will first look into how applications can provide various kinds of resources and then look into low-level interfaces for handling URIs and parameters to provide resources and functionalities.

Notice that using request handlers to create “pages” is not normally meaningful in Vaadin or in AJAX applications generally. Please see Section 4.2.3, “AJAX” for a detailed explanation.

5.5.1. Resource Interfaces and Classes

The resource classes in Vaadin are grouped under two interfaces: a generic **Resource** interface and a more specific **ConnectorResource** interface for resources provided by the servlet.

Figure 5.4. Resource Interface and Class Diagram



5.5.2. File Resources

File resources are files stored anywhere in the file system. As such, they can not be retrieved by a regular URL from the server, but need to be requested through the Vaadin servlet. The use of file resources is typically necessary for persistent user data that is not packaged in the web application, which would not be persistent over redeployments.

A file object that can be accessed as a file resource is defined with the standard **java.io.File** class. You can create the file either with an absolute or relative path, but the base path of the relative path depends on the installation of the web server. For example, with Apache Tomcat, the default current directory would be the installation path of Tomcat.

In the following example, we provide an image resource from a file stored in the web application. Notice that the image is stored under the `WEB-INF` folder, which is a special folder that is never accessible using an URL, unlike the other folders of a web application. This is a security solution - another would be to store the resource elsewhere in the file system.

```
// Find the application directory
String basepath = VaadinService.getCurrent()
    .getBaseDirectory().getAbsolutePath();

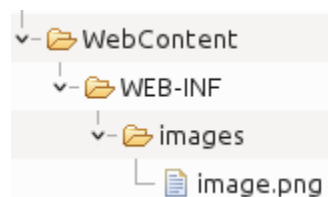
// Image as a file resource
FileResource resource = new FileResource(new File(basepath +
    "/WEB-INF/images/image.png"));

// Show the image in the application
Image image = new Image("Image from file", resource);

// Let the user view the file in browser or download it
Link link = new Link("Link to the image file", resource);
```

The result, as well as the folder structure where the file is stored under a regular Eclipse Vaadin project, is shown in Figure 5.5, “File Resource”.

Figure 5.5. File Resource



5.5.3. Class Loader Resources

The **ClassResource** allows resources to be loaded from the class path using Java Class Loader. Normally, the relevant class path entry is the `WEB-INF/classes` folder under the web application, where the Java compilation should compile the Java classes and copy other files from the source tree.

The one-line example below loads an image resource from the application package and displays it in an **Image** component.

```
layout.addComponent(new Image(null,
    new ClassResource("smiley.jpg"));
```

5.5.4. Theme Resources

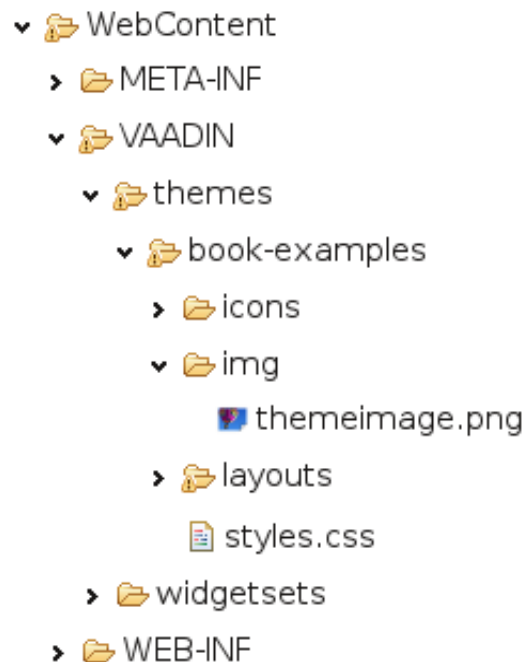
Theme resources of **ThemeResource** class are files, typically images, included in a theme. A theme is located with the path `VAADIN/themes/themename` in a web application. The name of a theme resource is given as the parameter for the constructor, with a path relative to the theme folder.

```
// A theme resource in the current theme ("mytheme")
// Located in: VAADIN/themes/mytheme/img/themeimage.png
ThemeResource resource = new ThemeResource("img/themeimage.png");

// Use the resource
Image image = new Image("My Theme Image", resource);
```

The result is shown in Figure 5.6, “Theme Resources”, also illustrating the folder structure for the theme resource file in an Eclipse project.

Figure 5.6. Theme Resources



To use theme resources, you must set the theme for the UI. See Chapter 9, *Themes* for more information regarding themes.

5.5.5. Stream Resources

Stream resources allow creating dynamic resource content. Charts are typical examples of dynamic images. To define a stream resource, you need to implement the **StreamResource.StreamSource** interface and its `getStream()` method. The method needs to return an **InputStream** from which the stream can be read.

The following example demonstrates the creation of a simple image in PNG image format.

```
import java.awt.image.*;
```



```
public class MyImageSource
    implements StreamResource.StreamSource {
    ByteArrayOutputStream imagebuffer = null;
    int reloads = 0;

    /* We need to implement this method that returns
     * the resource as a stream. */
    public InputStream getStream () {
        /* Create an image and draw something on it. */
        BufferedImage image = new BufferedImage (200, 200,
            BufferedImage.TYPE_INT_RGB);
        Graphics drawable = image.getGraphics();
        drawable.setColor(Color.lightGray);
        drawable.fillRect(0,0,200,200);
        drawable.setColor(Color.yellow);
        drawable.fillOval(25,25,150,150);
        drawable.setColor(Color.blue);
        drawable.drawRect(0,0,199,199);
        drawable.setColor(Color.black);
        drawable.drawString("Reloads="+reloads, 75, 100);
        reloads++;

        try {
            /* Write the image to a buffer. */
            imagebuffer = new ByteArrayOutputStream();
            ImageIO.write(image, "png", imagebuffer);

            /* Return a stream from the buffer. */
            return new ByteArrayInputStream(
                imagebuffer.toByteArray());
        } catch (IOException e) {
            return null;
        }
    }
}
```

The content of the generated image is dynamic, as it updates the reloads counter with every call. The **ImageIO.write()** method writes the image to an output stream, while we had to return an input stream, so we stored the image contents to a temporary buffer.

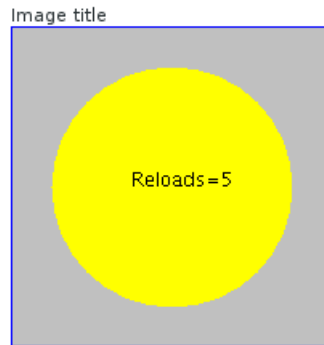
Below we display the image with the **Image** component.

```
// Create an instance of our stream source.
StreamResource.StreamSource imagesource = new MyImageSource ();

// Create a resource that uses the stream source and give it a name.
// The constructor will automatically register the resource in
// the application.
StreamResource resource =
    new StreamResource(imagesource, "myimage.png");

// Create an image component that gets its contents
// from the resource.
layout.addComponent(new Image("Image title", resource));
```

The resulting image is shown in Figure 5.7, “A Stream Resource”.

Figure 5.7. A Stream Resource

Another way to create dynamic content is a request handler, described in Section 12.4, “Request Handlers”.

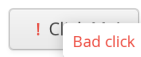
5.6. Handling Errors

5.6.1. Error Indicator and Message

All components have a built-in error indicator that is turned on if validating the component fails, and can be set explicitly with `setComponentError()`. Usually, the error indicator is placed right of the component caption. The error indicator is part of the component caption, so its placement is usually managed by the layout in which the component is contained, but some components handle it themselves. Hovering the mouse pointer over the field displays the error message.

```
textfield.setComponentError(new UserError("Bad value"));  
button.setComponentError(new UserError("Bad click"));
```

The result is shown in Figure 5.8, “Error Indicator Active”.

Figure 5.8. Error Indicator Active

5.6.2. Connection Fault

If the connection to the server is lost, Vaadin application shows a “lost connection” notification and tries to restore the connection. After several retries, an error message is shown. You can customize the messages, timeouts and the number of reconnect attempts using **ReconnectDialogConfiguration** class. Use `getReconnectDialogConfiguration` of your **UI** object.

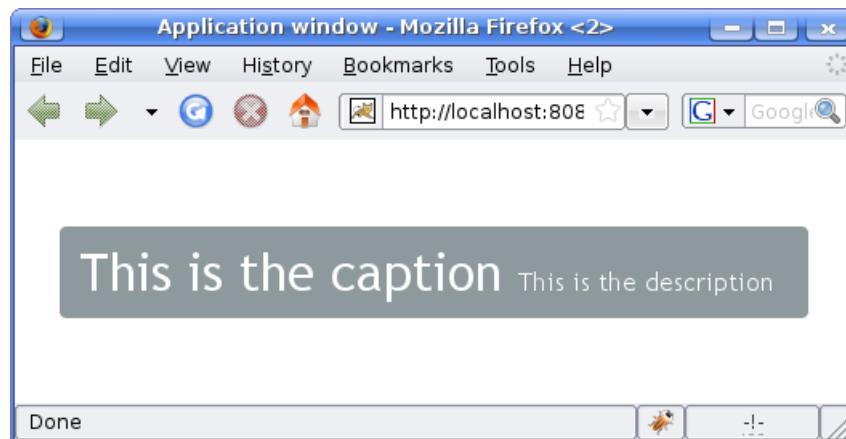
5.7. Notifications

Notifications are error or information boxes that appear briefly, typically at the center of the screen. A notification box has a caption and an optional description and icon. The box stays on the screen either for a preset time or until the user clicks it. The notification type defines the default appearance and behaviour of a notification.

There are two ways to create a notification. The easiest is to use a static shorthand `Notification.show()` method, which takes the caption of the notification as a parameter, and an optional description and notification type, and displays it in the current page.

```
Notification.show("This is the caption",
                 "This is the description",
                 Notification.Type.WARNING_MESSAGE);
```

Figure 5.9. Notification



For more control, you can create a **Notification** object. Different constructors exist for taking just the caption, and optionally the description, notification type, and whether HTML is allowed or not. Notifications are shown in a **Page**, typically the current page.

```
new Notification("This is a warning",
                 "<br/>This is the <i>last</i> warning",
                 Notification.Type.WARNING_MESSAGE, true)
    .show(Page.getCurrent());
```

The caption and description are by default written on the same line. If you want to have a line break between them, use the HTML line break markup "`<br/>`" if HTML is enabled, or "`\n`" if not. HTML is disabled by default, but can be enabled with `setHtmlContentAllowed(true)`. When enabled, you can use any HTML markup in the caption and description of a notification. If it is in any way possible to get the notification content from user input, you should either disallow HTML or sanitize the content carefully, as noted in Section 12.8.1, "Sanitizing User Input to Prevent Cross-Site Scripting".

Figure 5.10. Notification with HTML Formatting



5.7.1. Notification Type

The notification type defines the overall default style and behaviour of a notification. If no notification type is given, the "humanized" type is used as the default. The notification types, listed below, are defined in the **Notification.Type** class.

TYPE_HUMANIZED_MESSAGE image:[]

A user-friendly message that does not annoy too much: it does not require confirmation by clicking and disappears quickly. It is centered and has a neutral gray color.

TYPE_WARNING_MESSAGE image:[]

Warnings are messages of medium importance. They are displayed with colors that are neither neutral nor too distracting. A warning is displayed for 1.5 seconds, but the user can click the message box to dismiss it. The user can continue to interact with the application while the warning is displayed.

TYPE_ERROR_MESSAGE image:[]

Error messages are notifications that require the highest user attention, with alert colors, and they require the user to click the message to dismiss it. The error message box does not itself include an instruction to click the message, although the close box in the upper right corner indicates it visually. Unlike with other notifications, the user can not interact with the application while the error message is displayed.

TYPE_TRAY_NOTIFICATION image:[]

Tray notifications are displayed in the "system tray" area, that is, in the lower-right corner of the browser view. As they do not usually obscure any user interface, they are displayed longer than humanized or warning messages, 3 seconds by default. The user can continue to interact with the application normally while the tray notification is displayed.

5.7.2. Styling with CSS

```
.v-Notification {}
  .popupContent {}
    .gwt-HTML {}
      h1 {}
      p {}
```

The notification box is a floating `div` element under the `body` element of the page. It has an overall `v-Notification` style. The content is wrapped inside an element with `popupContent` style. The caption is enclosed within an `h1` element and the description in a `p` element.

To customize it, add a style for the **Notification** object with `setStyleName("mystyle")`, and make the settings in the theme, for example as follows:

```
.v-Notification.mystyle {
  background: #FFFF00;
  border: 10px solid #C00000;
  color: black;
}
```

The result is shown, with the icon set earlier in the customization example, in Figure 5.11, "A Styled Notification".

Figure 5.11. A Styled Notification

5.8. Application Lifecycle

In this section, we look into more technical details of application deployment, user sessions, and UI instance lifecycle. These details are not generally needed for writing Vaadin applications, but may be useful for understanding how they actually work and, especially, in what circumstances their execution ends.

5.8.1. Deployment

Before a Vaadin application can be used, it has to be deployed to a Java web server, as described in Section 5.9, “Deploying an Application”. Deploying reads the servlet classes annotated with the `@WebServlet` annotation (Servlet 3.0) or the `web.xml` deployment descriptor (Servlet 2.4) in the application to register servlets for specific URL paths and loads the classes. Deployment does not yet normally run any code in the application, although static blocks in classes are executed when they are loaded.

Undeploying and Redeploying

Applications are undeployed when the server shuts down, during redeployment, and when they are explicitly undeployed. Undeploying a server-side Vaadin application ends its execution, all application classes are unloaded, and the heap space allocated by the application is freed for garbage-collection.

If any user sessions are open at this point, the client-side state of the UIs is left hanging and an Out of Sync error is displayed on the next server request.

Redeployment and Serialization

Some servers, such as Tomcat, support *hot deployment*, where the classes are reloaded while preserving the memory state of the application. This is done by serializing the application state and then deserializing it after the classes are reloaded. This is, in fact, done with the basic Eclipse setup with Tomcat and if a UI is marked as **@PreserveOnRefresh**, you may actually need to give the `?restartApplication` URL parameter to force it to restart when you reload the page. Tools such as JRebel go even further by reloading the code in place without need for serialization. The server can also serialize the application state when shutting down and restarting, thereby preserving sessions over restarts.

Serialization requires that the applications are *serializable*, that is, all classes implement the `Serializable` interface. All Vaadin classes do. If you extend them or implement interfaces, you can provide an optional serialization key, which is automatically generated by Eclipse if you

use it. Serialization is also used for clustering and cloud computing, such as with Google App Engine.

5.8.2. Vaadin Servlet, Portlet, and Service

The **VaadinServlet**, or **VaadinPortlet** in a portal, receives all server requests mapped to it by its URL, as defined in the deployment configuration, and associates them with sessions. The sessions further associate the requests with particular UIs.

When servicing requests, the Vaadin servlet or portlet handles all tasks common to both servlets and portlets in a **VaadinService**. It manages sessions, gives access to the deployment configuration information, handles system messages, and does various other tasks. Any further servlet or portlet specific tasks are handled in the corresponding **VaadinServletService** or **VaadinPortletService**. The service acts as the primary low-level customization layer for processing requests.

Customizing Vaadin Servlet

Many common configuration tasks need to be done in the servlet class, which you already have if you are using the `@WebServlet` annotation for Servlet 3.0 to deploy the application. You can handle most customization by overriding the `servletInitialized()` method, where the **VaadinService** object is available with `getService()` (it would not be available in a constructor). You should always call `super.servletInitialized()` in the beginning.

```
public class MyServlet extends VaadinServlet {
    @Override
    protected void servletInitialized()
        throws ServletException {
        super.servletInitialized();

        ...
    }
}
```

To add custom functionality around request handling, you can override the `service()` method.

To use the custom servlet class in a Servlet 2.4 project, you need to define it in the `web.xml` deployment descriptor instead of the regular **VaadinServlet** class, as described in Section 5.9.4, “Using a `web.xml` Deployment Descriptor”.

5.8.3. User Session

A user session begins when a user first makes a request to a Vaadin servlet or portlet by opening the URL for a particular **UI**. All server requests belonging to a particular UI class are processed by the **VaadinServlet** or **VaadinPortlet** class. When a new client connects, it creates a new user session, represented by an instance of **VaadinSession**. Sessions are tracked using cookies stored in the browser.

You can obtain the **VaadinSession** of a **UI** with `getSession()` or globally with `VaadinSession.getCurrent()`. It also provides access to the lower-level session objects, `HttpSession` and `PortletSession`, through a **WrappedSession**. You can also access the deployment configuration through **VaadinSession**, as described in Section 5.9.7, “Deployment Configuration”.

A session ends after the last **UI** instance expires or is closed, as described later.

Handling Session Initialization and Destruction

You can handle session initialization and destruction by implementing a `SessionInitListener` or `SessionDestroyListener`, respectively, to the **VaadinService**. You can do that best by extending **VaadinServlet** and overriding the `servletInitialized()` method, as outlined in Section 5.8.2, “Vaadin Servlet, Portlet, and Service”.

```
public class MyServlet extends VaadinServlet
    implements SessionInitListener, SessionDestroyListener {

    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();
        getService().addSessionInitListener(this);
        getService().addSessionDestroyListener(this);
    }

    @Override
    public void sessionInit(SessionInitEvent event)
        throws ServiceException {
        // Do session start stuff here
    }

    @Override
    public void sessionDestroy(SessionDestroyEvent event) {
        // Do session end stuff here
    }
}
```

If using Servlet 2.4, you need to configure the custom servlet class in the `servlet-class` parameter in the `web.xml` descriptor instead of the **VaadinServlet**, as described in Section 5.9.4, “Using a `web.xml` Deployment Descriptor”.

5.8.4. Loading a UI

When a browser first accesses a URL mapped to the servlet of a particular UI class, the Vaadin servlet generates a loader page. The page loads the client-side engine (widget set), which in turn loads the UI in a separate request to the Vaadin servlet.

A **UI** instance is created when the client-side engine makes its first request. The servlet creates the UIs using a **UIProvider** registered in the **VaadinSession** instance. A session has at least a **DefaultUIProvider** for managing UIs opened by the user. If the application lets the user open popup windows with a **BrowserWindowOpener**, each of them has a dedicated special UI provider.

Once a new UI is created, its `init()` method is called. The method gets the request as a **VaadinRequest**.

Customizing the Loader Page

The HTML content of the loader page is generated as an HTML DOM object, which can be customized by implementing a `BootstrapListener` that modifies the DOM object. To do so, you need to extend the **VaadinServlet** and add a `SessionInitListener` to the service object, as outlined in Section 5.8.3, “User Session”. You can then add the bootstrap listener to a session with `addBootstrapListener()` when the session is initialized.

Loading the widget set is handled in the loader page with functions defined in a separate `vaadinBootstrap.js` script.

You can also use entirely custom loader code, such as in a static HTML page, as described in Section 12.2, “Embedding UIs in Web Pages”.

Custom UI Providers

You can create UI objects dynamically according to their request parameters, such as the URL path, by defining a custom `UIProvider`. You need to add custom UI providers to the session object which calls them. The providers are chained so that they are requested starting from the one added last, until one returns a UI (otherwise they return null). You can add a UI provider to a session most conveniently by implementing a custom servlet and adding the UI provider to sessions in a `SessionInitListener`.

You can find an example of custom UI providers in Section 21.8.1, “Providing a Fallback UI”.

Preserving UI on Refresh

Reloading a page in the browser normally spawns a new **UI** instance and the old UI is left hanging, until cleaned up after a while. This can be undesired as it resets the UI state for the user. To preserve the UI, you can use the **@PreserveOnRefresh** annotation for the UI class. You can also use a **UIProvider** with a custom implementation of `isUiPreserved()`.

```
@PreserveOnRefresh
public class MyUI extends UI {
```

Adding the `?restartApplication` parameter in the URL tells the Vaadin servlet to create a new **UI** instance when loading the page, thereby overriding the **@PreserveOnRefresh**. This is often necessary when developing such a UI in Eclipse, when you need to restart it after redeploying, because Eclipse likes to persist the application state between redeployments. If you also include a URI fragment, the parameter should be given before the fragment.

5.8.5. UI Expiration

UI instances are cleaned up if no communication is received from them after some time. If no other server requests are made, the client-side sends keep-alive heartbeat requests. A UI is kept alive for as long as requests or heartbeats are received from it. It expires if three consecutive heartbeats are missed.

The heartbeats occur at an interval of 5 minutes, which can be changed with the `heartbeatInterval` parameter of the servlet. You can configure the parameter in **@VaadinServletConfiguration** or in `web.xml` as described in Section 5.9.6, “Other Servlet Configuration Parameters”.

When the UI cleanup happens, a **DetachEvent** is sent to all **DetachListener**s added to the **UI**. When the `[classname]#UI` is detached from the session, `detach()` is called for it.

5.8.6. Closing UIs Explicitly

You can explicitly close a UI with `close()`. The method marks the UI to be detached from the session after processing the current request. Therefore, the method does not invalidate the UI instance immediately and the response is sent as usual.

Detaching a UI does not close the page or browser window in which the UI is running and further server request will cause error. Typically, you either want to close the window, reload it, or redirect

it to another URL. If the page is a regular browser window or tab, browsers generally do not allow closing them programmatically, but redirection is possible. You can redirect the window to another URL with `setLocation()`, as is done in the examples in Section 5.8.8, “Closing a Session”. You can close popup windows by making JavaScript `close()` call for them, as described in Section 12.1.2, “Closing Popup Windows”.

If you close other UI than the one associated with the current request, they will not be detached at the end of the current request, but after next request from the particular UI. You can make that occur quicker by making the UI heartbeat faster or immediately by using server push.

5.8.7. Session Expiration

A session is kept alive by server requests caused by user interaction with the application as well as the heartbeat monitoring of the UIs. Once all UIs have expired, the session still remains. It is cleaned up from the server when the session timeout configured in the web application expires.

If there are active UIs in an application, their heartbeat keeps the session alive indefinitely. You may want to have the sessions timeout if the user is inactive long enough, which is the original purpose of the session timeout setting. If the `closeIdleSessions` parameter of the servlet is set to `true` in the `web.xml`, as described in Section 5.9.4, “Using a `web.xml` Deployment Descriptor”, the session and all of its UIs are closed when the timeout specified by the `session-timeout` parameter of the servlet expires after the last non-heartbeat request. Once the session is gone, the browser will show an Out Of Sync error on the next server request. To avoid the ugly message, you may want to set a redirect URL for the UIs

The related configuration parameters are described in Section 5.9.6, “Other Servlet Configuration Parameters”.

You can handle session expiration on the server-side with a `SessionDestroyListener`, as described in Section 5.8.3, “User Session”.

5.8.8. Closing a Session

You can close a session by calling `close()` on the **VaadinSession**. It is typically used when logging a user out and the session and all the UIs belonging to the session should be closed. The session is closed immediately and any objects related to it are not available after calling the method.

When closing the session from a UI, you typically want to redirect the user to another URL. You can do the redirect using the `setLocation()` method in **Page**. This needs to be done before closing the session, as the UI or page are not available after that. In the following example, we display a logout button, which closes the user session.

```
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new Button("Logout", event -> { // Java 8
            // Redirect this page immediately
            getPage().setLocation("/myapp/logout.html");

            // Close the session
            getSession().close();
        }));
    }
}
```

```
    });  
  
    // Notice quickly if other UIs are closed  
    setPollInterval(3000);  
}  
}
```

This is not enough. When a session is closed from one UI, any other UIs attached to it are left hanging. When the client-side engine notices that a UI and the session are gone on the server-side, it displays a "Session Expired" message and, by default, reloads the UI when the message is clicked. You can customize the message and the redirect URL in the system messages

The client-side engine notices the expiration when user interaction causes a server request to be made or when the keep-alive heartbeat occurs. To make the UIs detect the situation faster, you need to make the heart beat faster, as was done in the example above. You can also use server push to close the other UIs immediately, as is done in the following example. Access to the UIs must be synchronized as described in Section 12.16, "Server Push".

```
@Push  
public class MyPushyUI extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        setContent(new Button("Logout", event -> { // Java 8  
            for (UI ui: VaadinSession.getCurrent().getUIs())  
                ui.access() -> {  
                    // Redirect from the page  
                    ui.getPage().setLocation("/logout.html");  
                });  
  
            getSession().close();  
        }));  
    }  
}
```

In the above example, we assume that all UIs in the session have push enabled and that they should be redirected; popups you might want to close instead of redirecting. It is not necessary to call `close()` for them individually, as we close the entire session afterwards.

5.9. Deploying an Application

Vaadin applications are deployed as *Java web applications*, which can contain a number of servlets, each of which can be a Vaadin application or some other servlet, and static resources such as HTML files. Such a web application is normally packaged as a WAR (Web application ARchive) file, which can be deployed to a Java application server (or a servlet container to be exact). A WAR file, which has the `.war` extension, is a subtype of JAR (Java ARchive), and like a regular JAR, is a ZIP-compressed file with a special content structure.

For a detailed tutorial on how web applications are packaged, please refer to any Java book that discusses Java Servlets.

In the Java Servlet parlance, a "web application" means a collection of Java servlets or portlets, JSP and static HTML pages, and various other resources that form an application. Such a Java web application is typically packaged as a WAR package for deployment. Server-side Vaadin UIs run as servlets within such a Java web application. There exists also other kinds of web

applications. To avoid confusion with the general meaning of "web application", we often refer to Java web applications with the slight misnomer "WAR" in this book.

5.9.1. Creating Deployable WAR in Eclipse

To deploy an application to a web server, you need to create a WAR package. Here we give the instructions for Eclipse.

1. Select **File** → **Export** and then **Web** → **WAR File**. Or, right-click the project in the Project Explorer and select **Web** → **WAR File**.
2. Select the **Web project** to export. Enter **Destination** file name (`.war`).
3. Make any other settings in the dialog, and click **Finish**.

5.9.2. Web Application Contents

The following files are required in a web application in order to run it.

`WEB-INF/web.xml` (optional with Servlet 3.0)

This is the web application descriptor that defines how the application is organized, that is, what servlets and such it has. You can refer to any Java book about the contents of this file. It is not needed if you define the Vaadin servlet with the `@WebServlet` annotation in Servlet API 3.0.

`WEB-INF/lib/*.jar`

These are the Vaadin libraries and their dependencies. They can be found in the installation package or as loaded by a dependency management system such as Maven or Ivy.

Your UI classes

You must include your UI classes either in a JAR file in `WEB-INF/lib` or as classes in `WEB-INF/classes`

Your own theme files (OPTIONAL)

If your application uses a special theme (look and feel), you must include it in `VAADIN/themes/themename` directory.

Widget sets (OPTIONAL)

If your application uses a project-specific widget set, it must be compiled in the `VAADIN/widgetset/` directory.

5.9.3. Web Servlet Class

When using the Servlet 3.0 API, you normally declare the Vaadin servlet classes with the `@WebServlet` annotation. The Vaadin UI associated with the servlet and other Vaadin-specific parameters are declared with a separate `@VaadinServletConfiguration` annotation.

```
@WebServlet(value = "/*",
            asyncSupported = true)
@VaadinServletConfiguration(
    productionMode = false,
    ui = MyProjectUI.class)
public class MyProjectServlet extends VaadinServlet {
}
```

The Vaadin Plugin for Eclipse creates the servlet class as a static inner class of the UI class. Normally, you may want to have it as a separate regular class.

The *value* parameter is the URL pattern for mapping request URLs to the servlet, as described in Section 5.9.5, “Servlet Mapping with URL Patterns”. The *ui* parameter is the UI class. Production mode is disabled by default, which enabled on-the-fly theme compilation, debug window, and other such development features. See the subsequent sections for details on the different servlet and Vaadin configuration parameters.

You can also use a `web.xml` deployment descriptor in Servlet 3.0 projects.

5.9.4. Using a `web.xml` Deployment Descriptor

A deployment descriptor is an XML file with the name `web.xml` in the `WEB-INF` sub-directory of a web application. It is a standard component in Java EE describing how a web application should be deployed. The descriptor is not required with Servlet API 3.0, where you can also define servlets with the **@WebServlet** annotation as described earlier, as web fragments, or programmatically. You can use both a `web.xml` and `WebServlet` in the same application. Settings in the `web.xml` override the ones given in annotations.

The following example shows the basic contents of a deployment descriptor for a Servlet 2.4 application. You simply specify the UI class with the *UI* parameter for the **com.vaadin.server.VaadinServlet**. The servlet is then mapped to a URL path in a standard way for Java Servlets.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  id="WebApp_ID" version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <servlet>
    <servlet-name>myservlet</servlet-name>
    <servlet-class>
      com.vaadin.server.VaadinServlet
    </servlet-class>

    <init-param>
      <param-name>UI</param-name>
      <param-value>com.ex.myprj.MyUI</param-value>
    </init-param>

    <!-- If not using the default widget set-->
    <init-param>
      <param-name>widgetset</param-name>
      <param-value>com.ex.myprj.MyWidgetSet</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>myservlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

The descriptor defines a servlet with the name `myservlet`. The servlet class, **`com.vaadin.server.VaadinServlet`**, is provided by Vaadin framework and is normally the same for all Vaadin projects. For some purposes, you may need to use a custom servlet class that extends the **`VaadinServlet`**. The class name must include the full package path.

Servlet API Version

The descriptor example given above was for Servlet 2.4. For a later version, such as Servlet 3.0, you should use:

```
<web-app
  id="WebApp_ID" version="3.0"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
```

Servlet 3.0 support is useful for at least server push.

Widget Set

If the UI uses add-on components or custom widgets, it needs a custom widget set, which can be specified with the `widgetset` parameter for the servlet. Alternatively, you can define it with the **`@WidgetSet`** annotation for the UI class. The parameter is a class name with the same path but without the `.gwt.xml` extension as the widget set definition file. If the parameter is not given, the **`com.vaadin.DefaultWidgetSet`** is used, which contains all the widgets for the built-in Vaadin components.

Unless using the default widget set (which is included in the `vaadin-client-compiled` JAR), the widget set must be compiled, as described in Chapter 18, *Using Vaadin Add-ons* or Section 14.4, “Compiling a Client-Side Module”, and properly deployed with the application.

5.9.5. Servlet Mapping with URL Patterns

The servlet needs to be mapped to an URL path, which requests it is to handle.

With **`@WebServlet`** annotation for the servlet class:

```
@WebServlet(value = "/*", asyncSupported = true)
```

In a `web.xml`:

```
<servlet-mapping>
  <servlet-name>myservlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

The URL pattern is defined in the above examples as `/*`. This matches any URL under the project context. We defined above the project context as `myproject` so the URL for the page of the UI will be `http://localhost:8080/myproject/`.

Mapping Sub-Paths

If an application has multiple UIs or servlets, they have to be given different paths in the URL, matched by a different URL pattern. Also, you may need to have statically served content under some path. Having an URL pattern `/myui/*` would match a URL such as `http://local-`

host:8080/myproject/myui/. Notice that the slash and the asterisk *must* be included at the end of the pattern. In such case, you also need to map URLs with `/VAADIN/*` to a servlet (unless you are serving it statically as noted below).

With a **@WebServlet** annotation for a servlet class, you can define multiple mappings as a list enclosed in curly braces as follows:

```
@WebServlet(value = {"/myui/*", "/VAADIN/*"},
            asyncSupported = true)
```

In a `web.xml`:

```
...
<servlet-mapping>
  <servlet-name>myservlet</servlet-name>
  <url-pattern>/myui/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>myservlet</servlet-name>
  <url-pattern>/VAADIN/*</url-pattern>
</servlet-mapping>
```

If you have multiple servlets, you should specify only one `/VAADIN/*` mapping. It does not matter which servlet you map the pattern to, as long as it is a Vaadin servlet.

You do not have to provide the above `/VAADIN/*` mapping if you serve both the widget sets and (custom and default) themes statically in the `/VAADIN` directory in the web application. The mapping simply allows serving them dynamically from the Vaadin JAR. Serving them statically is recommended for production environments as it is faster. If you serve the content from within the same web application, you may not have the root pattern `/*` for the Vaadin servlet, as then all the requests would be mapped to the servlet.

5.9.6. Other Servlet Configuration Parameters

The servlet class or deployment descriptor can have many parameters and options that control the execution of a servlet. You can find complete documentation of the basic servlet parameters in the appropriate Java Servlet Specification.

@VaadinServletConfiguration accepts a number of special parameters, as described below.

In a `web.xml`, you can set most parameters either as a `<context-param>` for the entire web application, in which case they apply to all Vaadin servlets, or as an `<init-param>` for an individual servlet. If both are defined, servlet parameters override context parameters.

Production Mode

By default, Vaadin applications run in *debug mode* (or *development mode*), which should be used during development. This enables various debugging features. For production use, you should have the `productionMode=true` setting in the **@VaadinServletConfiguration**, or in `web.xml`:

```
<context-param>
  <param-name>productionMode</param-name>
  <param-value>true</param-value>
  <description>Vaadin production mode</description>
</context-param>
```

The parameter and the debug and production modes are described in more detail in Section 12.3, “Debug Mode and Window”.

Custom UI Provider

Vaadin normally uses the **DefaultUIProvider** for creating **UI** class instances. If you need to use a custom UI provider, you can define its class with the *UIProvider* parameter. The provider is registered in the **VaadinSession**.

In a `web.xml`:

```
<servlet>
...
  <init-param>
    <param-name>UIProvider</param-name>
    <param-value>com.ex.my.MyUIProvider</param-value>
  </init-param>
```

The parameter is logically associated with a particular servlet, but can be defined in the context as well.

UI Heartbeat

Vaadin monitors UIs by using a heartbeat, as explained in Section 5.8.5, “UI Expiration”. If the user closes the browser window of a Vaadin application or navigates to another page, the Client-Side Engine running in the page stops sending heartbeat to the server, and the server eventually cleans up the **UI** instance.

The interval of the heartbeat requests can be specified in seconds with the *heartbeatInterval* parameter either as a context parameter for the entire web application or an init parameter for the individual servlet. The default value is 300 seconds (5 minutes).

In a `web.xml`:

```
<context-param>
  <param-name>heartbeatInterval</param-name>
  <param-value>300</param-value>
</context-param>
```

Session Timeout After User Inactivity

In normal servlet operation, the session timeout defines the allowed time of inactivity after which the server should clean up the session. The inactivity is measured from the last server request. Different servlet containers use varying defaults for timeouts, such as 30 minutes for Apache Tomcat. You can set the timeout under `<web-app>` with:

In a `web.xml`:

```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

The session timeout should be longer than the heartbeat interval or otherwise sessions are closed before the heartbeat can keep them alive. As the session expiration leaves the UIs in a state where they assume that the session still exists, this would cause an Out Of Sync error notification in the browser.

However, having a shorter heartbeat interval than the session timeout, which is the normal case, prevents the sessions from expiring. If the `closeIdleSessions` parameter for the servlet is enabled (disabled by default), Vaadin closes the UIs and the session after the time specified in the `session-timeout` parameter expires after the last non-heartbeat request.

In a `web.xml`:

```
<servlet>
  ...
  <init-param>
    <param-name>closeIdleSessions</param-name>
    <param-value>true</param-value>
  </init-param>
```

Push Mode

You can enable server push, as described in Section 12.16, “Server Push”, for a UI either with a **@Push** annotation for the UI or in the descriptor. The push mode is defined with a `pushmode` parameter. The `automatic` mode pushes changes to the browser automatically after `access()` finishes. With `manual` mode, you need to do the push explicitly with `push()`. If you use a Servlet 3.0 compatible server, you also want to enable asynchronous processing with the `async-supported` parameter.

In a `web.xml`:

```
<servlet>
  ...
  <init-param>
    <param-name>pushmode</param-name>
    <param-value>automatic</param-value>
  </init-param>
  <async-supported>true</async-supported>
```

Cross-Site Request Forgery Prevention

Vaadin uses a protection mechanism to prevent malicious cross-site request forgery (XSRF or CSRF), also called one-click attacks or session riding, which is a security exploit for executing unauthorized commands in a web server. This protection is normally enabled. However, it prevents some forms of testing of Vaadin applications, such as with JMeter. In such cases, you can disable the protection by setting the `disable-xsrf-protection` parameter to `true`.

In a `web.xml`:

```
<context-param>
  <param-name>disable-xsrf-protection</param-name>
  <param-value>true</param-value>
</context-param>
```

5.9.7. Deployment Configuration

The Vaadin-specific parameters defined in the deployment configuration are available from the **DeploymentConfiguration** object managed by the **VaadinSession**.

```
DeploymentConfiguration conf =
    getSession().getConfiguration();
```



```
// Heartbeat interval in seconds  
int heartbeatInterval = conf.getHeartbeatInterval();
```

Parameters defined in the Java Servlet definition, such as the session timeout, are available from the low-level **HttpSession** or **PortletSession** object, which are wrapped in a **WrappedSession** in Vaadin. You can access the low-level session wrapper with `getSession()` of the **VaadinSession**.

```
WrappedSession session = getSession().getSession();  
int sessionTimeout = session.getMaxInactiveInterval();
```

You can also access other **HttpSession** and **PortletSession** session properties through the interface, such as set and read session attributes that are shared by all servlets belonging to a particular servlet or portlet session.

Chapter 6

User Interface Components

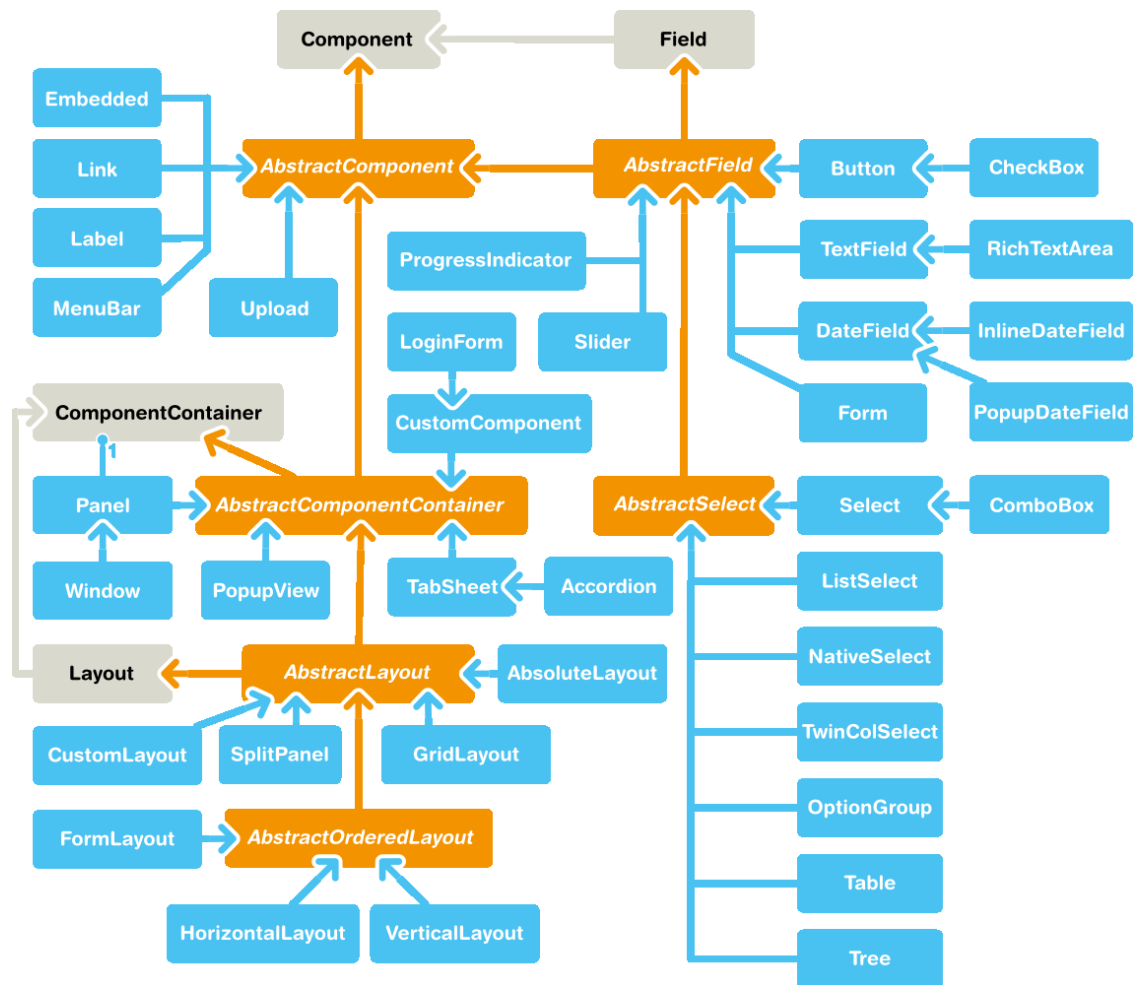
6.1. Overview	118
6.2. Interfaces and Abstractions	120
6.3. Common Component Features	122
6.4. Field Components	132
6.5. Selection Components	138
6.6. Component Extensions	144
6.7. Label	144
6.8. Link	148
6.9. TextField	150
6.10. TextArea	154
6.11. PasswordField	156
6.12. RichTextArea	157
6.13. Date and Time Input with DateField	158
6.14. Button	162
6.15. CheckBox	163
6.16. ComboBox	164
6.17. ListSelect	167
6.18. NativeSelect	168
6.19. OptionGroup	169
6.20. TwinColSelect	172

6.21. Table	173
6.22. Tree	186
6.23. TreeTable	187
6.24. Grid	189
6.25. MenuBar	207
6.26. Upload	209
6.27. ProgressBar	211
6.28. Slider	213
6.29. PopupView	214
6.30. Calendar	215
6.31. Composition with CustomComponent	220
6.32. Composite Fields with CustomField	222
6.33. Embedded Resources	222

This chapter provides an overview and a detailed description of all non-layout components in Vaadin.

6.1. Overview

Vaadin provides a comprehensive set of user interface components and allows you to define custom components. Figure 6.1, “User Interface Component Class Hierarchy” illustrates the inheritance hierarchy of the UI component classes and interfaces. Interfaces are displayed in gray, abstract classes in orange, and regular classes in blue. An annotated version of the diagram is featured in the *Vaadin Cheat Sheet*.

Figure 6.1. User Interface Component Class Hierarchy

At the top of the interface hierarchy, we have the **Component** interface. At the top of the class hierarchy, we have the **AbstractComponent** class. It is inherited by two other abstract classes: **AbstractField**, inherited further by field components, and **AbstractComponentContainer**, inherited by various container and layout components. Components that are not bound to a content data model, such as labels and links, inherit **AbstractComponent** directly.

The layout of the various components in a window is controlled, logically, by layout components, just like in conventional Java UI toolkits for desktop applications. In addition, with the **Custom-Layout** component, you can write a custom layout as an HTML template that includes the locations of any contained components. Looking at the inheritance diagram, we can see that layout components inherit the **AbstractComponentContainer** and the **Layout** interface. Layout components are described in detail in Chapter 7, *Managing Layout*.

Looking at it from the perspective of an object hierarchy, we would have a **Window** object, which contains a hierarchy of layout components, which again contain other layout components, field components, and other visible components.

You can browse the built-in UI components of Vaadin library in the Sampler application of the Vaadin Demo. The Sampler shows a description, JavaDoc documentation, and a code samples for each of the components.

In addition to the built-in components, many components are available as add-ons, either from the Vaadin Directory or from independent sources. Both commercial and free components exist. The installation of add-ons is described in Chapter 18, *Using Vaadin Add-ons*.



Vaadin Cheat Sheet and Refcard

Figure 6.1, “User Interface Component Class Hierarchy” is included in the Vaadin Cheat Sheet that illustrates the basic relationship hierarchy of the user interface components and data binding classes and interfaces. You can download it at <http://vaadin.com/book>.

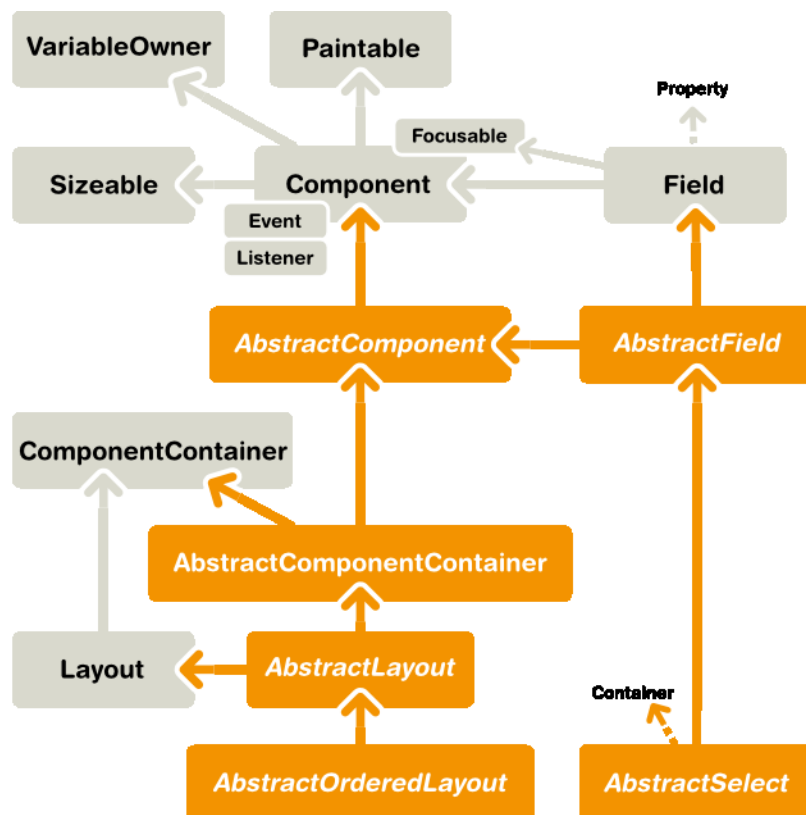
The diagram is also included in the six-page DZone Refcard, which you can find at <https://vaadin.com/refcard>.

6.2. Interfaces and Abstractions

Vaadin user interface components are built on a skeleton of interfaces and abstract classes that define and implement the features common to all components and the basic logic how the component states are serialized between the server and the client.

This section gives details on the basic component interfaces and abstractions. The layout and other component container abstractions are described in Chapter 7, *Managing Layout*. The interfaces that define the Vaadin data model are described in Chapter 10, *Binding Components to Data*.

Figure 6.2. Component Interfaces and Abstractions



All components also implement the **Paintable** interface, which is used for serializing ("painting") the components to the client, and the reverse **VariableOwner** interface, which is needed for deserializing component state or user interaction from the client.

In addition to the interfaces defined within the Vaadin framework, all components implement the **java.io.Serializable** interface to allow serialization. Serialization is needed in many clustering and cloud computing solutions.

6.2.1. Component Interface

The `Component` interface is paired with the **AbstractComponent** class, which implements all the methods defined in the interface.

Component Tree Management

Components are laid out in the user interface hierarchically. The layout is managed by layout components, or more generally components that implement the **ComponentContainer** interface. Such a container is the parent of the contained components.

The `getParent()` method allows retrieving the parent component of a component. While there is a `setParent()`, you rarely need it as you usually add components with the `addComponent()` method of the **ComponentContainer** interface, which automatically sets the parent.

A component does not know its parent when the component is still being created, so you can not refer to the parent in the constructor with `getParent()`.

Attaching a component to an UI triggers a call to its `attach()` method. Correspondingly, removing a component from a container triggers calling the `detach()` method. If the parent of an added component is already connected to the UI, the `attach()` is called immediately from `setParent()`.

```
public class AttachExample extends CustomComponent {
    public AttachExample() {

        @Override
        public void attach() {
            super.attach(); // Must call.

            // Now we know who ultimately owns us.
            ClassResource r = new ClassResource("smiley.jpg");
            Image image = new Image("Image:", r);
            setCompositionRoot(image);
        }
    }
}
```

The attachment logic is implemented in **AbstractComponent**, as described in Section 6.2.2, “**AbstractComponent**”.

6.2.2. AbstractComponent

AbstractComponent is the base class for all user interface components. It is the (only) implementation of the **Component** interface, implementing all the methods defined in the interface.

AbstractComponent has a single abstract method, `getTag()`, which returns the serialization identifier of a particular component class. It needs to be implemented when (and only when)

creating entirely new components. **AbstractComponent** manages much of the serialization of component states between the client and the server. Creation of new components and serialization is described in Chapter 17, *Integrating with the Server-Side*.

6.3. Common Component Features

The component base classes and interfaces provide a large number of features. Let us look at some of the most commonly needed features. Features not documented here can be found from the Java API Reference.

The interface defines a number of properties, which you can retrieve or manipulate with the corresponding setters and getters.

6.3.1. Caption

A caption is an explanatory textual label accompanying a user interface component, usually shown above, left of, or inside the component. The contents of a caption are automatically quoted, so no raw HTML can be rendered in a caption.

The caption text can usually be given as the first parameter of a constructor of a component or with `setCaption()`.

```
// New text field with caption "Name"
TextField name = new TextField("Name");
layout.addComponent(name);
```

The caption of a component is, by default, managed and displayed by the layout component or component container inside which the component is placed. For example, the **VerticalLayout** component shows the captions left-aligned above the contained components, while the **Form-Layout** component shows the captions on the left side of the vertically laid components, with the captions and their associated components left-aligned in their own columns. The **Custom-Component** does not manage the caption of its composition root, so if the root component has a caption, it will not be rendered.

Figure 6.3. Caption Management by `VerticalLayout` and `FormLayout`.

A Text Field

Second Text Field

Third Text Field

A Text Field

Second Text Field

Third Text Field

Some components, such as **Button** and **Panel**, manage the caption themselves and display it inside the component.

Icon (see Section 6.3.4, “Icon”) is closely related to caption and is usually displayed horizontally before or after it, depending on the component and the containing layout. Also the required indicator in field components is usually shown before or after the caption.

An alternative way to implement a caption is to use another component as the caption, typically a **Label**, a **TextField**, or a **Panel**. A **Label**, for example, allows highlighting a shortcut key with HTML markup or to bind the caption to a data source. The **Panel** provides an easy way to add both a caption and a border around a component.

CSS Style Rules

```
.v-caption {}  
.v-captiontext {}  
.v-caption-clearelem {}  
.v-required-field-indicator {}
```

A caption is rendered inside an HTML element that has the `v-caption` CSS style class. The containing layout may enclose a caption inside other caption-related elements.

Some layouts put the caption text in a `v-captiontext` element. A `v-caption-clearelem` is used in some layouts to clear a CSS `float` property in captions. An optional required indicator in field components is contained in a separate element with `v-required-field-indicator` style.

6.3.2. Description and Tooltips

All components (that inherit **AbstractComponent**) have a description separate from their caption. The description is usually shown as a tooltip that appears when the mouse pointer hovers over the component for a short time.

You can set the description with `setDescription()` and retrieve with `getDescription()`.

```
Button button = new Button("A Button");  
button.setDescription("This is the tooltip");
```

The tooltip is shown in Figure 6.4, “Component Description as a Tooltip”.

Figure 6.4. Component Description as a Tooltip



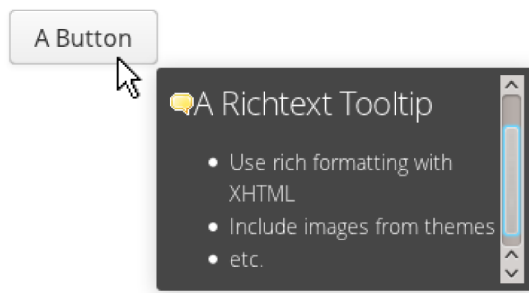
A description is rendered as a tooltip in most components.

When a component error has been set with `setComponentError()`, the error is usually also displayed in the tooltip, below the description. Components that are in error state will also display the error indicator. See Section 5.6.1, “Error Indicator and Message”.

The description is actually not plain text, but you can use HTML tags to format it. Such a rich text description can contain any HTML elements, including images.

```
button.setDescription(  
    "<h2><img src=\"../VAADIN/themes/sampler/icons/comment_yellow.gif\"/>"+  
    "A richtext tooltip</h2>"+  
    "<ul>"+  
    "  <li>Use rich formatting with HTML</li>"+  
    "  <li>Include images from themes</li>"+  
    "  <li>etc.</li>"+  
    "</ul>");
```

The result is shown in Figure 6.5, “A Rich Text Tooltip”.

Figure 6.5. A Rich Text Tooltip

Notice that the setter and getter are defined for all fields in the **Field** interface, not for all components in the **Component** interface.

6.3.3. Enabled

The *enabled* property controls whether the user can actually use the component. A disabled component is visible, but grayed to indicate the disabled state.

Components are always enabled by default. You can disable a component with `setEnabled(false)`.

```
Button enabled = new Button("Enabled");
enabled.setEnabled(true); // The default
layout.addComponent(enabled);
```

```
Button disabled = new Button("Disabled");
disabled.setEnabled(false);
layout.addComponent(disabled);
```

Figure 6.6, “An Enabled and Disabled **Button**” shows the enabled and disabled buttons.

Figure 6.6. An Enabled and Disabled Button

A disabled component is automatically put in read-only state. No client interaction with such a component is sent to the server and, as an important security feature, the server-side components do not receive state updates from the client in the read-only state. This feature exists in all built-in components in Vaadin and is automatically handled for all **Field** components for the field property value. For custom widgets, you need to make sure that the read-only state is checked on the server-side for all safety-critical variables.

CSS Style Rules

Disabled components have the `v-disabled` CSS style in addition to the component-specific style. To match a component with both the styles, you have to join the style class names with a dot as done in the example below.

```
.v-textfield.v-disabled {  
    border: dotted;  
}
```

This would make the border of all disabled text fields dotted.

\$v-button-disabled-opacity In Valo theme, the opacity of disabled components is specified with the \$v-disabled-opacity parameter

6.3.4. Icon

An icon is an explanatory graphical label accompanying a user interface component, usually shown above, left of, or inside the component. Icon is closely related to caption (see Section 6.3.1, “Caption”) and is usually displayed horizontally before or after it, depending on the component and the containing layout.

The icon of a component can be set with the `setIcon()` method. The image is provided as a resource, perhaps most typically a **ThemeResource**.

```
// Component with an icon from a custom theme  
TextField name = new TextField("Name");  
name.setIcon(new ThemeResource("icons/user.png"));  
layout.addComponent(name);  
  
// Component with an icon from another theme ('runo')  
Button ok = new Button("OK");  
ok.setIcon(new ThemeResource("../runo/icons/16/ok.png"));  
layout.addComponent(ok);
```

The icon of a component is, by default, managed and displayed by the layout component or component container in which the component is placed. For example, the **VerticalLayout** component shows the icons left-aligned above the contained components, while the **FormLayout** component shows the icons on the left side of the vertically laid components, with the icons and their associated components left-aligned in their own columns. The **CustomComponent** does not manage the icon of its composition root, so if the root component has an icon, it will not be rendered.

Figure 6.7. Displaying an Icon from a Theme Resource.



Some components, such as **Button** and **Panel**, manage the icon themselves and display it inside the component.

In addition to image resources, you can use *font icons*, which are icons included in special fonts, but which are handled as special resources. See Section 9.8, “Font Icons” for more details.

CSS Style Rules

An icon will be rendered inside an HTML element that has the `v-icon` CSS style class. The containing layout may enclose an icon and a caption inside elements related to the caption, such as `v-caption`.

6.3.5. Locale

The locale property defines the country and language used in a component. You can use the locale information in conjunction with an internationalization scheme to acquire localized resources. Some components, such as **DateField**, use the locale for component localization.

You can set the locale of a component (or the application) with `setLocale()` as follows:

```
// Component for which the locale is meaningful
InlineDateField date = new InlineDateField("Datum");

// German language specified with ISO 639-1 language
// code and ISO 3166-1 alpha-2 country code.
date.setLocale(new Locale("de", "DE"));

date.setResolution(Resolution.DAY);
layout.addComponent(date);
```

The resulting date field is shown in Figure 6.8, “Set Locale for **InlineDateField**”.

Figure 6.8. Set Locale for InlineDateField

Datum



6.3.6. Read-Only

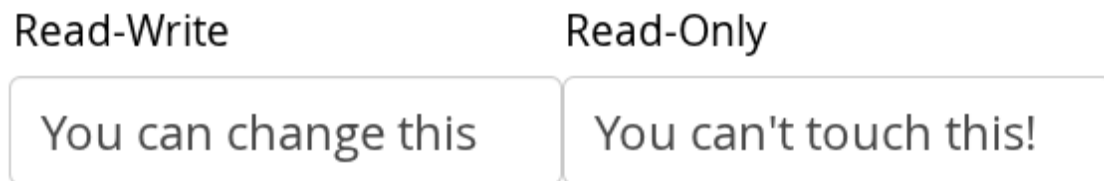
The property defines whether the value of a component can be changed. The property is mainly applicable to **Field** components, as they have a value that can be edited by the user.

```
TextField readwrite = new TextField("Read-Write");
readwrite.setValue("You can change this");
readwrite.setReadOnly(false); // The default
layout.addComponent(readwrite);
```

```
TextField readonly = new TextField("Read-Only");
readonly.setValue("You can't touch this!");
readonly.setReadOnly(true);
layout.addComponent(readonly);
```

The resulting read-only text field is shown in Figure 6.9, “A Read-Only Component.”.

Figure 6.9. A Read-Only Component.



Setting a layout or some other component container as read-only does not usually make the contained components read-only recursively. This is different from, for example, the disabled state, which is usually applied recursively.

Notice that the value of a selection component is the selection, not its items. A read-only selection component doesn't therefore allow its selection to be changed, but other changes are possible. For example, if you have a read-only **Table** in editable mode, its contained fields and the underlying data model can still be edited, and the user could sort it or reorder the columns.

Client-side state modifications will not be communicated to the server-side and, more importantly, server-side field components will not accept changes to the value of a read-only **Field** component. The latter is an important security feature, because a malicious user can not fabricate state changes in a read-only field. This is handled at the level of **AbstractField** in `setValue()`, so you can not change the value programmatically either. Calling `setValue()` on a read-only field results in **Property.ReadOnlyException**.

Also notice that while the read-only status applies automatically to the property value of a field, it does not apply to other component variables. A read-only component can accept some other variable changes from the client-side and some of such changes could be acceptable, such as change in the scroll bar position of a **Table**. Custom widgets should check the read-only state for variables bound to business data.

CSS Style Rules

Setting a normally editable component to read-only state can change its appearance to disallow editing the value. In addition to CSS styling, also the HTML structure can change. For example, **TextField** loses the edit box and appears much like a **Label**.

A read-only component will have the `v-readonly` style. The following CSS rule would make the text in all read-only **TextField** components appear in italic.

```
.v-textfield.v-readonly {  
    font-style: italic;  
}
```

6.3.7. Style Name

The *style name* property defines one or more custom CSS style class names for the component. The `getStyleName()` returns the current style names as a space-separated list. The `setStyleName()` replaces all the styles with the given style name or a space-separated list of style names. You can also add and remove individual style names with `addStyleName()` and `removeStyleName()`. A style name must be a valid CSS style name.

```
Label label = new Label("This text has a lot of style");  
label.addStyleName("mystyle");  
layout.addComponent(label);
```

The style name will appear in the component's HTML element in two forms: literally as given and prefixed with the component-specific style name. For example, if you add a style name *mystyle* to a **Button**, the component would get both *mystyle* and *v-button-mystyle* styles. Neither form may conflict with built-in style names of Vaadin. For example, *focus* style would conflict with a built-in style of the same name, and an *content* style for a **Panel** component would conflict with the built-in *v-panel-content* style.

The following CSS rule would apply the style to any component that has the *mystyle* style.

```
.mystyle {  
    font-family: fantasy;  
    font-style: italic;  
    font-size: 25px;  
    font-weight: bolder;  
    line-height: 30px;  
}
```

The resulting styled component is shown in Figure 6.10, "Component with a Custom Style"

Figure 6.10. Component with a Custom Style

This text has a lot of style

6.3.8. Visible

Components can be hidden by setting the *visible* property to *false*. Also the caption, icon and any other component features are made hidden. Hidden components are not just invisible, but their content is not communicated to the browser at all. That is, they are not made invisible cosmetically with only CSS rules. This feature is important for security if you have components that contain security-critical information that must only be shown in specific application states.

```
TextField invisible = new TextField("No-see-um");  
invisible.setValue("You can't see this!");  
invisible.setVisible(false);  
layout.addComponent(invisible);
```

The resulting invisible component is shown in Figure 6.11, “An Invisible Component.”.

Figure 6.11. An Invisible Component.

Beware that invisible beings can leave footprints. The containing layout cell that holds the invisible component will not go away, but will show in the layout as extra empty space. Also expand ratios work just like if the component was visible - it is the layout cell that expands, not the component.

If you need to make a component only cosmetically invisible, you should use a custom theme to set it `display: none` style. This is mainly useful for some special components that have effects even when made invisible in CSS. If the hidden component has undefined size and is enclosed in a layout that also has undefined size, the containing layout will collapse when the component disappears. If you want to have the component keep its size, you have to make it invisible by setting all its font and other attributes to be transparent. In such cases, the invisible content of the component can be made visible easily in the browser.

A component made invisible with the *visible* property has no particular CSS style class to indicate that it is hidden. The element does exist though, but has `display: none` style, which overrides any CSS styling.

6.3.9. Sizing Components

Vaadin components are sizeable; not in the sense that they were fairly large or that the number of the components and their features are sizeable, but in the sense that you can make them fairly large on the screen if you like, or small or whatever size.

The **Sizeable** interface, shared by all components, provides a number of manipulation methods and constants for setting the height and width of a component in absolute or relative units, or for leaving the size undefined.

The size of a component can be set with `setWidth()` and `setHeight()` methods. The methods take the size as a floating-point value. You need to give the unit of the measure as the second parameter for the above methods. The available units are listed in Table 6.1, “Size Units” below.

```
mycomponent.setWidth(100, Sizeable.UNITS_PERCENTAGE);  
mycomponent.setWidth(400, Sizeable.UNITS_PIXELS);
```

Alternatively, you can specify the size as a string. The format of such a string must follow the HTML/CSS standards for specifying measures.

```
mycomponent.setWidth("100%");  
mycomponent.setHeight("400px");
```

The "100%" percentage value makes the component take all available size in the particular direction (see the description of *Sizeable.UNITS_PERCENTAGE* in the table below). You can also use the shorthand method `setSizeFull()` to set the size to 100% in both directions.

The size can be *undefined* in either or both dimensions, which means that the component will take the minimum necessary space. Most components have undefined size by default, but some layouts have full size in horizontal direction. You can set the height or width as undefined with *Sizeable.SIZE_UNDEFINED* parameter for `setWidth()` and `setHeight()`.

You always need to keep in mind that *a layout with undefined size may not contain components with defined relative size*, such as "full size". See Section 7.13.1, "Layout Size" for details.

The Table 6.1, "Size Units" lists the available units and their codes defined in the **Sizeable** interface.

Table 6.1. Size Units

<code>Unit.PIXELS</code>	px	The <i>pixel</i> is the basic hardware-specific measure of one physical display pixel.
<code>Unit.POINTS</code>	pt	The <i>point</i> is a typographical unit, which is usually defined as 1/72 inches or about 0.35 mm. However, on displays the size can vary significantly depending on display metrics.
<code>Unit.PICAS</code>	pc	The <i>pica</i> is a typographical unit, defined as 12 points, or 1/7 inches or about 4.233 mm. On displays, the size can vary depending on display metrics.
<code>Unit.EM</code>	em	A unit relative to the used font, the width of the upper-case "M" letter.
<code>Unit.EX</code>	ex	A unit relative to the used font, the height of the lower-case "x" letter.
<code>Unit.MM</code>	mm	A physical length unit, millimeters on the surface of a display device. However, the actual size depends on the display, its metrics in the operating system, and the browser.
<code>Unit.CM</code>	cm	A physical length unit, <i>centimeter</i> on the surface of a display device. However, the actual size depends on the display, its metrics in the operating system, and the browser.
<code>Unit.INCH</code>	in	A physical length unit, <i>inch</i> on the surface of a display device. However, the actual size depends on the display, its metrics in the operating system, and the browser.
<code>Unit.PERCENTAGE</code>	%	A relative percentage of the available size. For example, for the top-level layout[parameter]#100%# would be the full width or height of the browser

		window. The percentage value must be between 0 and 100.

If a component inside **HorizontalLayout** or **VerticalLayout** has full size in the namesake direction of the layout, the component will expand to take all available space not needed by the other components. See Section 7.13.1, “Layout Size” for details.

6.3.10. Managing Input Focus

When the user clicks on a component, the component gets the *input focus*, which is indicated by highlighting according to style definitions. If the component allows inputting text, the focus and insertion point are indicated by a cursor. Pressing the Tab key moves the focus to the component next in the *focus order*.

Focusing is supported by all **Field** components and also by **Upload**.

The focus order or *tab index* of a component is defined as a positive integer value, which you can set with `setTabIndex()` and get with `getTabIndex()`. The tab index is managed in the context of the page in which the components are contained. The focus order can therefore jump between two any lower-level component containers, such as sub-windows or panels.

The default focus order is determined by the natural hierarchical order of components in the order in which they were added under their parents. The default tab index is 0 (zero).

Giving a negative integer as the tab index removes the component from the focus order entirely.

CSS Style Rules

The component having the focus will have an additional style class with the `-focus` suffix. For example, a **TextField**, which normally has the `v-textfield` style, would additionally have the `v-textfield-focus` style.

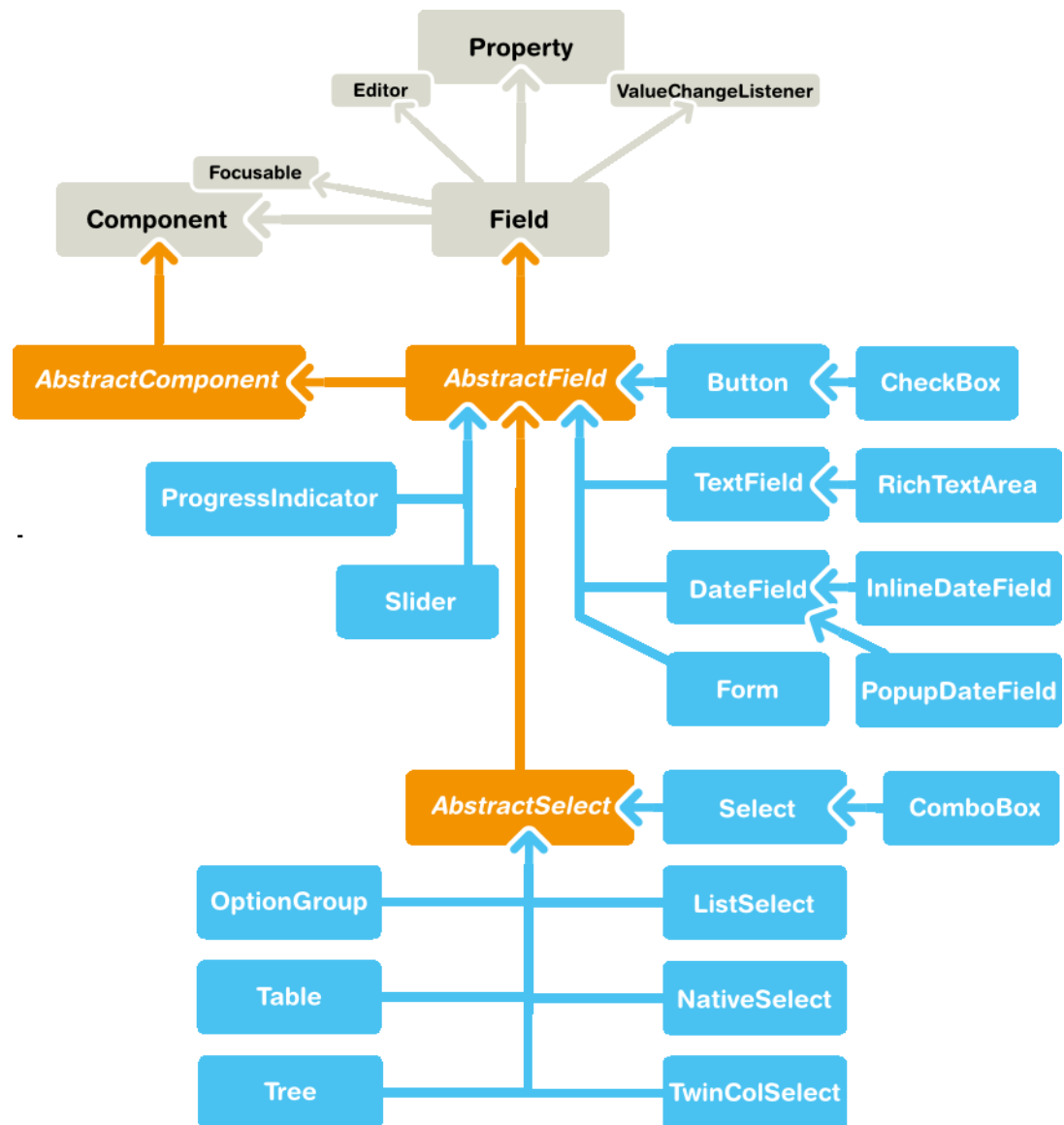
For example, the following would make a text field blue when it has focus.

```
.v-textfield-focus {  
    background: lightblue;  
}
```

6.4. Field Components

Fields are components that have a value that the user can change through the user interface. Figure 6.12, “Field Components” illustrates the inheritance relationships and the important interfaces and base classes.

Figure 6.12. Field Components

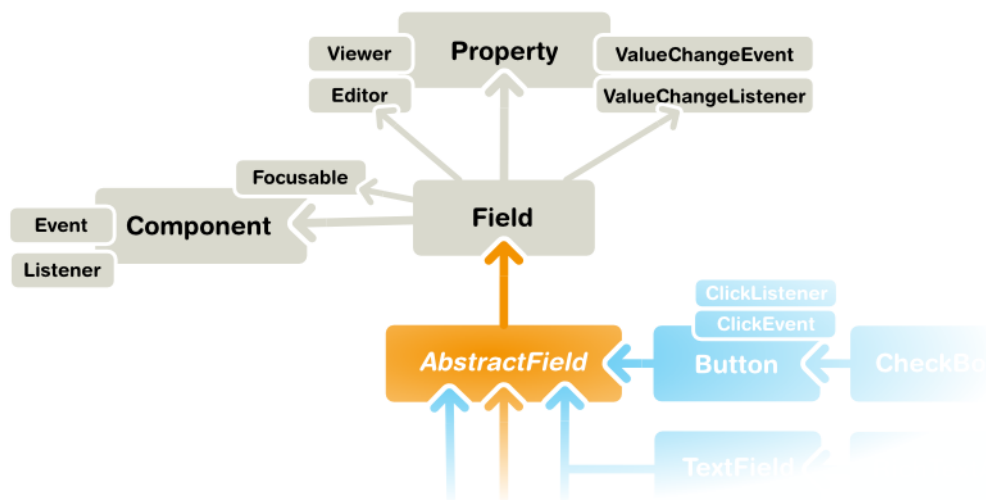


Field components are built upon the framework defined in the **Field** interface and the **AbstractField** base class. **AbstractField** is the base class for all field components. In addition to the component features inherited from **AbstractComponent**, it implements a number of features defined in **Property**, **Buffered**, **Validatable**, and **Component.Focusable** interfaces.

The description of the field interfaces and base classes is broken down in the following sections.

6.4.1. Field Interface

The **Field** interface inherits the **Component** superinterface and also the **Property** interface to have a value for the field. **AbstractField** is the only class implementing the **Field** interface directly. The relationships are illustrated in Figure 6.13, “**Field** Interface Inheritance Diagram”.

Figure 6.13. Field Interface Inheritance Diagram

You can set the field value with the `setValue()` and read with the `getValue()` method defined in the **Property** interface. The actual value type depends on the component.

The **Field** interface defines a number of properties, which you can access with the corresponding setters and getters.

`required`

When enabled, a required indicator (usually the asterisk `*` character) is displayed on the left, above, or right the field, depending on the containing layout and whether the field has a caption. If such fields are validated but are empty and the `requiredError` property (see below) is set, an error indicator is shown and the component error is set to the text defined with the error property. Without validation, the required indicator is merely a visual guide.

`requiredError`

Defines the error message to show when a value is required, but none is entered. The error message is set as the component error for the field and is usually displayed in a tooltip when the mouse pointer hovers over the error indicator.

6.4.2. Data Binding and Conversions

Fields are strongly coupled with the Vaadin data model. The field value is handled as a **Property** of the field component, as documented in Section 10.2, “Properties”. Selection fields allow management of the selectable items through the **Container** interface.

Fields are *editors* for some particular type. For example, **TextField** allows editing **String** values. When bound to a data source, the property type of the data model can be something different, say an **Integer**. *Converters* are used for converting the values between the representation and the model. They are described in Section 10.2.3, “Converting Between Property Type and Representation”.

6.4.3. Handling Field Value Changes

Field inherits **Property.ValueChangeListener** to allow listening for field value changes and **Property.Editor** to allow editing values.

When the value of a field changes, a **Property.ValueChangeEvent** is triggered for the field. You should not implement the `valueChange()` method in a class inheriting **AbstractField**, as it is already implemented in **AbstractField**. You should instead implement the method explicitly by adding the implementing object as a listener.

6.4.4. Field Buffering

Field components implement the `Buffered` and `BufferedValidatable` interfaces. When buffering is enabled for a field with `setBuffered(true)`, the value is not written to the property data source before the `commit()` method is called for the field. Calling `commit()` also runs validators added to the field, and if any fail (and the `invalidCommitted` is disabled), the value is not written.

```
form.addComponent(new Button("Commit",
    new Button.ClickListener() {
        @Override
        public void buttonClick(ClickEvent event) {
            try {
                editor.commit();
            } catch (InvalidValueException e) {
                Notification.show(e.getMessage());
            }
        }
    }
));
```

Calling `discard()` reads the value from the property data source to the current input.

If the fields are bound in a **FieldGroup** that has buffering enabled, calling `commit()` for the group runs validation on all fields in the group, and if successful, all the field values are written to the item data source. See Section 10.4.4, “Buffering Forms”.

6.4.5. Field Validation

The input for a field component can be syntactically or semantically invalid. Fields implement the `Validatable` interface, which allows checking validity of the input with *validators* that implement the `Validator` interface. You can add validators to fields with `addValidator()`.

```
TextField field = new TextField("Name");
field.addValidator(new StringLengthValidator(
    "The name must be 1-10 letters (was {0})",
    1, 10, true));
field.setNullRepresentation("");
field.setNullSettingAllowed(true);
layout.addComponent(field);
```

Failed validation is indicated with the error indicator of the field, described in Section 5.6.1, “Error Indicator and Message”, unless disabled with `setValidationVisible(false)`. Hovering mouse on the field displays the error message given as a parameter for the validator. If validated explicitly with `validate()`, as described later, the **InvalidValueException** is thrown if the validation fails, also carrying the error message. The value `{0}` in the error message string is replaced with the invalid input value.

Validators validate the property type of the field after a possible conversion, not the presentation type. For example, an **IntegerRangeValidator** requires that the value type of the property data source is **Integer**.

Built-in Validators

Vaadin includes the following built-in validators. The property value type is indicated.

BeanValidator

Validates a bean property according to annotations defined in the Bean Validation API 1.0 (JSR-303). This validator is usually not used explicitly, but they are created implicitly when binding fields in a **BeanFieldGroup**. Using bean validation requires an implementation library of the API. See Section 10.4.6, "Bean Validation" for details.

CompositeValidator

Combines validators using logical AND and OR operators.

DateRangeValidator:*[classname]Date*

Checks that the date value is within the range at or between two given dates/times.

DoubleRangeValidator:*[classname]Double*

Checks that the double value is at or between two given values.

EmailValidator:*[classname]String*

Checks that the string value is a syntactically valid email address. The validated syntax is close to the RFC 822 standard regarding email addresses.

IntegerRangeValidator:*[classname]Integer*

Checks that the integer value is at or between two given values.

NullValidator

Checks that the value is or is not a null value.

For the validator to be meaningful, the component must support inputting null values. For example, for selection components and **TextField**, inputting null values can be enabled with `setNullSettingAllowed()`. You also need to set the representation of null values: in selection components with `setNullSelectionItemId()` and in **TextField** with `setNullRepresentation()`.

RegexValidator:*[classname]String*

Checks that the value matches with the given regular expression.

StringLengthValidator:*[classname]String*

Checks that the length of the input string is at or between two given lengths.

Please see the API documentation for more details.

Automatic Validation

The validators are normally, when `validationVisible` is true for the field, executed implicitly on the next server request if the input has changed. If the field is in immediate mode, it (and any other fields with changed value) are validated immediately when the focus leaves the field.

```
TextField field = new TextField("Name");
field.addValidator(new StringLengthValidator(
    "The name must be 1-10 letters (was {0})",
    1, 10, true));
field.setImmediate(true);
field.setNullRepresentation("");
```

```
field.setNullSettingAllowed(true);
layout.addComponent(field);
```

Explicit Validation

The validators are executed when the `validate()` or `commit()` methods are called for the field.

```
// A field with automatic validation disabled
final TextField field = new TextField("Name");
field.setNullRepresentation("");
field.setNullSettingAllowed(true);
layout.addComponent(field);

// Define validation as usual
field.addValidator(new StringLengthValidator(
    "The name must be 1-10 letters (was {0})",
    1, 10, true));

// Run validation explicitly
Button validate = new Button("Validate");
validate.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        field.setValidationVisible(false);
        try {
            field.validate();
        } catch (InvalidValueException e) {
            Notification.show(e.getMessage());
            field.setValidationVisible(true);
        }
    }
});
layout.addComponent(validate);
```

Implementing a Custom Validator

You can create custom validators by implementing the `Validator` interface and implementing its `validate()` method. If the validation fails, the method should throw either **`InvalidValueException`** or **`EmptyValueException`**.

```
class MyValidator implements Validator {
    @Override
    public void validate(Object value)
        throws InvalidValueException {
        if (!(value instanceof String &&
            ((String)value).equals("hello")))
            throw new InvalidValueException("You're impolite");
    }
}

final TextField field = new TextField("Say hello");
field.addValidator(new MyValidator());
field.setImmediate(true);
layout.addComponent(field);
```

Validation in Field Groups

If the field is bound to a **FieldGroup**, described in Section 10.4, “Creating Forms by Binding Fields to Items”, calling `commit()` for the group runs the validation for all the fields in the group, and if successful, writes the input values to the data source.

6.5. Selection Components

Vaadin offers many alternative ways for selecting one or more items. The core library includes the following selection components, all based on the **AbstractSelect** class:

ComboBox (Section 6.16, “**ComboBox**”)

A drop-down list with a text box, where the user can type text to find matching items. The component also provides an input prompt and the user can enter new items.

ListSelect (Section 6.17, “**ListSelect**”)

A vertical list box for selecting items in either single or multiple selection mode.

NativeSelect (Section 6.18, “**NativeSelect**”)

Provides selection using the native selection component of the browser, typically a drop-down list for single selection and a multi-line list in multiselect mode. This uses the `<select>` element in HTML.

OptionGroup (Section 6.19, “**OptionGroup**”)

Shows the items as a vertically arranged group of radio buttons in the single selection mode and of check boxes in multiple selection mode.

TwinColSelect (Section 6.20, “**TwinColSelect**”)

Shows two list boxes side by side where the user can select items from a list of available items and move them to a list of selected items using control buttons.

In addition, the **Tree**, **Table**, and **TreeTable** components allow special forms of selection. They also inherit the **AbstractSelect**.

6.5.1. Binding Selection Components to Data

The selection components are strongly coupled with the Vaadin Data Model, described in Chapter 10, *Binding Components to Data*. The selectable items in all selection components are objects that implement the **Item** interface. The items are contained in a **Container**.

All selection components are containers themselves and simply forward all container operations to the underlying container data source. You can give the container in the constructor or set it with `setContainerDataSource()`. This is further described in Section 10.5.1, “Basic Use of Containers”.

```
// Have a container data source of some kind
IndexedContainer container = new IndexedContainer();
container.addContainerProperty("name", String.class, null);
...

// Create a selection component bound to the container
OptionGroup group = new OptionGroup("My Select", container);
```

If you do not bind a selection component to a container data source, a default container is used. It is usually either an **IndexedContainer** or a **HierarchicalContainer**.

The current selection of a selection component is bound to the **Property** interface, so you can get the current selection as the value of the selection component. Also selection changes are handled as value change events, as is described later.

6.5.2. Adding New Items

New items are added with the `addItem()` method defined in the **Container** interface, described in Section 10.5.1, “Basic Use of Containers”.

```
// Create a selection component
ComboBox select = new ComboBox("My ComboBox");

// Add items with given item IDs
select.addItem("Mercury");
select.addItem("Venus");
select.addItem("Earth");
```

The `addItem()` method creates an empty **Item**, which is identified by its *item identifier* (IID) object, given as the parameter. This item ID is by default used also as the caption of the item, as described in more detail later.

We emphasize that `addItem()` is a factory method that *takes an item ID, not the actual item* as the parameter - the item is returned by the method. The item is of a type that is specific to the container and has itself little relevance for most selection components, as the properties of an item may not be used in any way (except in **Table**), only the item ID.

The item identifier is typically a string, in which case it can be used as the caption, but can be any object type. We could as well have given integers for the item identifiers and set the captions explicitly with `setItemCaption()`. You could also add an item with the parameterless `addItem()`, which returns an automatically generated item ID.

```
// Create a selection component
ComboBox select = new ComboBox("My Select");

// Add an item with a generated ID
Object itemId = select.addItem();
select.setItemCaption(itemId, "The Sun");

// Select the item
select.setValue(itemId);
```

Some container types may support passing the actual data object to the add method. For example, you can add items to a **BeanItemContainer** with `addBean()`. Such implementations can use a separate item ID object, or the data object itself as the item ID, as is done in `addBean()`. In the latter case you can not depend on the default way of acquiring the item caption; see the description of the different caption modes later.

The next section describes the different options for determining the item captions.

6.5.3. Item Captions

The displayed captions of items in a selection component can be set explicitly with `setItemCaption()` or determined from the item IDs or item properties. The caption determination is defined with the *caption mode*, any of the modes in the **AbstractSelect.ItemCaptionMode** enum, which you can set with `setItemCaptionMode()`. The default mode is `EXPLICIT_DEFAULTS_ID`, which uses the item identifiers for the captions, unless given explicitly.

In addition to a caption, an item can have an icon. The icon is set with `setItemIcon()`.

The caption modes defined in **ItemCaptionMode** are the following:

EXPLICIT_DEFAULTS_ID

This is the default caption mode and its flexibility allows using it in most cases. By default, the item identifier will be used as the caption. The identifier object does not necessarily have to be a string; the caption is retrieved with `toString()` method. If the caption is specified explicitly with `setItemCaption()`, it overrides the item identifier.

```
// Create a selection component
ComboBox select = new ComboBox("Moons of Mars");
select.setItemCaptionMode(ItemCaptionMode.EXPLICIT_DEFAULTS_ID);

// Use the item ID also as the caption of this item
select.addItem(new Integer(1));

// Set item caption for this item explicitly
select.addItem(2); // same as "new Integer(2)"
select.setItemCaption(2, "Deimos");
```

EXPLICIT

Captions must be explicitly specified with `setItemCaption()`. If they are not, the caption will be empty. Such items with empty captions will nevertheless be displayed in the selection component as empty items. If they have an icon, they will be visible.

ICON_ONLY

Only icons are shown, captions are hidden.

ID

String representation of the item identifier object is used as caption. This is useful when the identifier is a string, and also when the identifier is an complex object that has a string representation. For example:

```
ComboBox select = new ComboBox("Inner Planets");
select.setItemCaptionMode(ItemCaptionMode.ID);

// A class that implements toString()
class PlanetId extends Object implements Serializable {
    String planetName;

    PlanetId (String name) {
        planetName = name;
    }

    public String toString () {
        return "The Planet " + planetName;
    }
}

// Use such objects as item identifiers
String planets[] = {"Mercury", "Venus", "Earth", "Mars"};
for (int i=0; i<planets.length; i++)
    select.addItem(new PlanetId(planets[i]));
```

INDEX

Index number of item is used as caption. This caption mode is applicable only to data sources that implement the **Container.Indexed** interface. If the interface is not

available, the component will throw a **ClassCastException**. The **AbstractSelect** itself does not implement this interface, so the mode is not usable without a separate data source. An **IndexedContainer**, for example, would work.

ITEM

String representation of item, acquired with `toString()`, is used as the caption. This is applicable mainly when using a custom **Item** class, which also requires using a custom **Container** that is used as a data source for the selection component.

PROPERTY

Item captions are read from the **String** representation of the property with the identifier specified with `setItemCaptionPropertyId()`. This is useful, for example, when you have a container that you use as the data source for the selection component, and you want to use a specific property for caption.

In the example below, we bind a selection component to a bean container and use a property of the bean as the caption.

```
/** A bean with a "name" property. */
public class Planet implements Serializable {
    int id;
    String name;

    public Planet(int id, String name) {
        this.id = id;
        this.name = name;
    }

    ... setters and getters ...
}

public void captionproperty(VerticalLayout layout) {
    // Have a bean container to put the beans in
    BeanItemContainer<Planet> container =
        new BeanItemContainer<Planet>(Planet.class);

    // Put some example data in it
    container.addItem(new Planet(1, "Mercury"));
    container.addItem(new Planet(2, "Venus"));
    container.addItem(new Planet(3, "Earth"));
    container.addItem(new Planet(4, "Mars"));

    // Create a selection component bound to the container
    ComboBox select = new ComboBox("Planets", container);

    // Set the caption mode to read the caption directly
    // from the 'name' property of the bean
    select.setItemCaptionMode(ItemCaptionMode.PROPERTY);
    select.setItemCaptionPropertyId("name");

    ...
}
```

6.5.4. Getting and Setting Selection

A selection component provides the current selection as the property of the component (with the **Property** interface). The property value is an item identifier object that identifies the selected item. You can get the identifier with `getValue()` of the **Property** interface.

You can select an item with the corresponding `setValue()` method. In multiselect mode, the property will be an unmodifiable set of item identifiers. If no item is selected, the property will be `null` in single selection mode or an empty collection in multiselect mode.

The **ComboBox** and **NativeSelect** will show empty selection when no actual item is selected. This is the *null selection item identifier*. You can set an alternative ID with `setNullSelectionItemId()`. Setting the alternative null ID is merely a visual text; the `getValue()` will still return `null` value if no item is selected, or an empty set in multiselect mode.

6.5.5. Handling Selection Changes

The item identifier of the currently selected item will be set as the property of the selection component. You can access it with the `getValue()` method of the **Property** interface of the component. Also, when handling selection changes with a **Property.ValueChangeListener**, the **ValueChangeEvent** will have the selected item as the property of the event, accessible with the `getProperty()` method.

```
// Create a selection component with some items
ComboBox select = new ComboBox("My Select");
select.addItem("Io", "Europa", "Ganymedes", "Callisto");

// Handle selection change
select.addValueChangeListener(event -> // Java 8
    layout.addComponent(new Label("Selected " +
        event.getProperty().getValue())));
```

The result of user interaction is shown in Figure 6.14, “Selected Item”.

Figure 6.14. Selected Item



6.5.6. Allowing Adding New Items

Some selection components can allow the user to add new items. Currently, only **ComboBox** allows it, when the user types in a value and presses **Enter**. You need to enable the mode with `setNewItemAllowed(true)`. Setting the component also in immediate mode may be necessary, as otherwise the item would not be added immediately when the user interacts with the component, but after some other component causes a server request.

```
myselect.setNewItemAllowed(true);
myselect.setImmediate(true);
```

The user interface for adding new items depends on the selection component. The regular **ComboBox** component allows you to simply type the new item in the combo box and hit **Enter** to add it.

Adding new items is not possible if the selection component is read-only or is bound to a **Container** that does not allow adding new items. An attempt to do so may result in an exception.

Handling New Items

Adding new items is handled by a `NewItemHandler`, which gets the item caption string as parameter for the `addNewItem()` method. The default implementation, **DefaultNewItemHandler**, checks for read-only state, adds the item using the entered caption as the item ID, and if the selection component gets the captions from a property, copies the caption to that property. It also selects the item. The default implementation may not be suitable for all container types, in which case you need to define a custom handler. For example, a **BeanItemContainer** expects the items to have the bean object itself as the ID, not a string.

6.5.7. Multiple Selection

Some selection components, such as **OptionGroup** and **ListSelect** support a multiple selection mode, which you can enable with `setMultiSelect()`. For **TwinColSelect**, which is especially intended for multiple selection, it is enabled by default.

```
myselect.setMultiSelect(true);
```

As in single selection mode, the property value of the component indicates the selection. In multiple selection mode, however, the property value is a **Collection** of the item IDs of the currently selected items. You can get and set the property with the `getValue()` and `setValue()` methods as usual.

A change in the selection will trigger a **ValueChangeEvent**, which you can handle with a **Property.ValueChangeListener**. As usual, you should use `setImmediate(true)` to trigger the event immediately when the user changes the selection. The following example shows how to handle selection changes with a listener.

```
// A selection component with some items
ListSelect select = new ListSelect("My Selection");
select.addItem("Mercury", "Venus", "Earth",
               "Mars", "Jupiter", "Saturn", "Uranus", "Neptune");

// Multiple selection mode
select.setMultiSelect(true);

// Feedback on value changes
select.addValueChangeListener(
    new Property.ValueChangeListener() {
        public void valueChange(ValueChangeEvent event) {
            // Some feedback
            layout.addComponent(new Label("Selected: " +
                event.getProperty().getValue().toString()));
        }
    });
select.setImmediate(true);
```

6.5.8. Item Icons

You can set an icon for each item with `setItemIcon()`, or define an item property that provides the icon resource with `setItemIconPropertyId()`, in a fashion similar to captions. Notice, however, that icons are not supported in **NativeSelect**, **TwinColSelect**, and some other selection

components and modes. This is because HTML does not support images inside the native `select` elements. Icons are also not really visually applicable.

6.6. Component Extensions

Components and UIs can have extensions which are attached to the component dynamically. Especially, many add-ons are extensions.

How a component is extended depends on the extension. Typically, they have an `extend()` method that takes the component to be extended as the parameter.

```
TextField tf = new TextField("Hello");
layout.addComponent(tf);

// Add a simple extension
new CapsLockWarning().extend(tf);

// Add an extension that requires some parameters
CSValidator validator = new CSValidator();
validator.setRegExp("[0-9]*");
validator.setErrorMessage("Must be a number");
validator.extend(tf);
```

Development of custom extensions is described in Section 17.7, “Component and UI Extensions”.

6.7. Label

Label component displays non-editable text. This text can be used for short simple labels or for displaying long text, such as paragraphs. The text can be formatted in HTML or as preformatted text, depending on the *content mode* of the label.

You can give the label text most conveniently in the constructor, as is done in the following. Label has 100% default width, so the containing layout must also have defined width.

```
// A container that is 100% wide by default
VerticalLayout layout = new VerticalLayout();

Label label = new Label("Labeling can be dangerous");
layout.addComponent(label);
```

Label implements the `Property` interface to allow accessing the text value, so you can get and set the text with `getValue()` and `setValue()`.

```
// Get the label's text to initialize a field
TextField editor = new TextField(null, // No caption
                                label.getValue());

// Change the label's text
editor.addValueChangeListener(event -> // Java 8
    label.setValue(editor.getValue()));
editor.setImmediate(true); // Send on Enter
```

Label also supports data binding to a property data source, as described later in Section 6.7.3, “Data Binding”. However, in that case the value can not be set through the label, as **Label** is not a `Property.Editor` and is not allowed to write to a bound property.

Even though **Label** is text and is often used as a caption, it is a normal component and therefore also has a caption that you can set with `setCaption()`. As with most other components, the caption is managed by the containing layout.

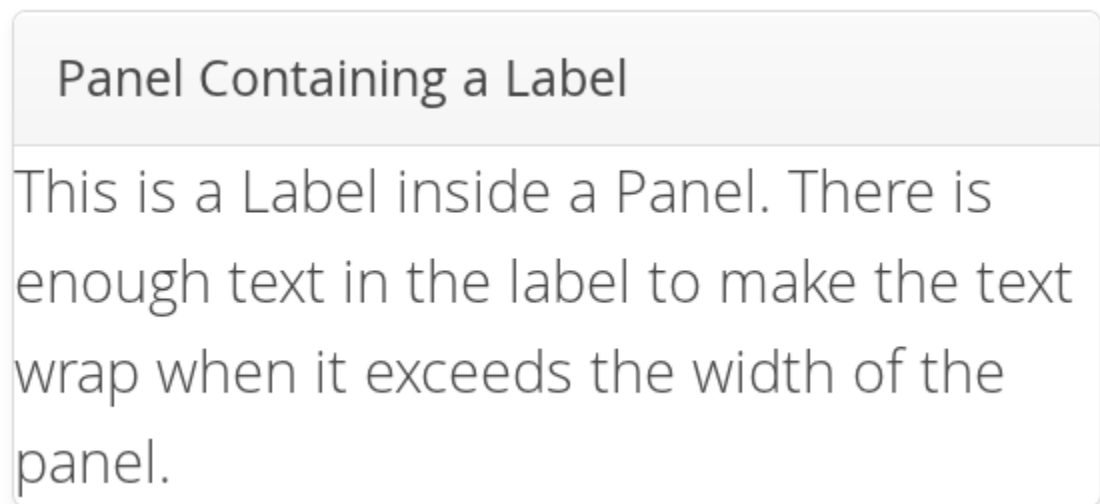
6.7.1. Text Width and Wrapping

Label has 100% default width, so the containing layout must also have a defined width. If the width of the label's text exceeds the width of the label, the text will wrap around and continue on the next line. Some layout components have undefined width by default, such as **HorizontalLayout**, so you need to pay special care with them.

```
// A container with a defined width.  
Panel panel = new Panel("Panel Containing a Label");  
panel.setWidth("300px");  
  
panel.setContent(  
    new Label("This is a Label inside a Panel. There is "  
              "enough text in the label to make the text "  
              "wrap when it exceeds the width of the panel.");
```

As the size of the **Panel** in the above example is fixed and the width of **Label** is the default 100%, the text in the **Label** will wrap to fit the panel, as shown in Figure 6.15, “The Label Component”.

Figure 6.15. The Label Component



Setting **Label** to undefined width will cause it to not wrap at the end of the line, as the width of the content defines the width. If placed inside a layout with defined width, the **Label** will overflow the layout horizontally and, normally, be truncated.

6.7.2. Content Mode

The content of a label is formatted depending on a *content mode*. By default, the text is assumed to be plain text and any contained XML-specific characters will be quoted appropriately to allow rendering the contents of a label in HTML in a web browser. The content mode can be set in the constructor or with `setContentMode()`, and can have the values defined in the **Content-Mode** enumeration type in `com.vaadin.shared.ui.label` package:

TEXT

The default content mode where the label contains only plain text. All characters are allowed, including the special <, >, and & characters in XML or HTML, which are quoted properly in HTML while rendering the component. This is the default mode.

PREFORMATTED

Content mode where the label contains preformatted text. It will be, by default, rendered with a fixed-width typewriter font. Preformatted text can contain line breaks, written in Java with the `\n` escape sequence for a newline character (ASCII 0x0a), or tabulator characters written with `\t` (ASCII 0x09).

HTML

Content mode where the label contains HTML.

Please note the following security and validity warnings regarding the HTML content mode.

**Cross-Site Scripting Warning**

Having **Label** in HTML content mode allows pure HTML content. If the content comes from user input, you should always carefully sanitize it to prevent cross-site scripting (XSS) attacks. Please see Section 12.8.1, “Sanitizing User Input to Prevent Cross-Site Scripting”.

Also, the validity of the HTML content is not checked when rendering the component and any errors can result in an error in the browser. If the content comes from an uncertain source, you should always validate it before displaying it in the component.

The following example demonstrates the use of **Label** in different modes.

```
Label textLabel = new Label(  
    "Text where formatting characters, such as \\n, " +  
    "and HTML, such as <b>here</b>, are quoted.",  
    ContentMode.TEXT);  
  
Label preLabel = new Label(  
    "Preformatted text is shown in an HTML <pre> tag.\\n" +  
    "Formatting such as\\n" +  
    " * newlines\\n" +  
    " * whitespace\\n" +  
    "and such are preserved. HTML tags, \\n"+  
    "such as <b>bold</b>, are quoted.",  
    ContentMode.PREFORMATTED);  
  
Label htmlLabel = new Label(  
    "In HTML mode, all HTML formatting tags, such as \\n" +  
    "<ul>"+  
    " <li><b>bold</b></li>"+  
    " <li>itemized lists</li>"+  
    " <li>etc.</li>"+  
    "</ul> "+  
    "are preserved.",  
    ContentMode.HTML);
```

The rendering will look as shown in Figure 6.16, “Label Content Modes”.

Figure 6.16. Label Content Modes

TEXT	Text where formatting characters, such as <code>\n</code> , and HTML, such as <code>here</code> , are quoted.
PREFORMATTED	Preformatted text is shown in an HTML <code><pre></code> tag. Formatting such as * newlines * whitespace and such are preserved. HTML tags, such as <code>bold</code> , are quoted.
HTML	In HTML mode, all HTML formatting tags, such as • bold • itemized lists • etc. are preserved.

6.7.3. Data Binding

While **Label** is not a field component, it is a `Property.Viewer` and can be bound to a property data source, described in Section 10.2, “Properties”. You can specify the data source either in the constructor or by the `setPropertyDataSource()` method.

```
// Some property
ObjectProperty<String> property =
    new ObjectProperty<String>("some value");

// Label that is bound to the property
Label label = new Label(property);
```

Further, as **Label** is a `Property`, you can edit its value with a property editor, such as a field:

```
Label label = new Label("some value");
TextField editor = new TextField();
editor.setPropertyDataSource(label);
editor.setImmediate(true);
```

However, **Label** is *not* a `Property.Editor`, so it is read-only when bound to a data source. Therefore, you can not use `setValue()` to set the value of a connected data source through a **Label** nor bind the label to an editor field, in which case writes would be delegated through the label.

6.7.4. CSS Style Rules

```
.v-label { }  
pre { } /* In PREFORMATTED content mode */
```

The **Label** component has a `v-label` overall style. In the *PREFORMATTED* content mode, the text is wrapped inside a `<pre>` element.

6.8. Link

The **Link** component allows making hyperlinks. References to locations are represented as resource objects, explained in Section 5.5, “Images and Other Resources”. The **Link** is a regular HTML hyperlink, that is, an `<a href>` anchor element that is handled natively by the browser. Unlike when clicking a **Button**, clicking a **Link** does not cause an event on the server-side.

Links to an arbitrary URL can be made by using an **ExternalResource** as follows:

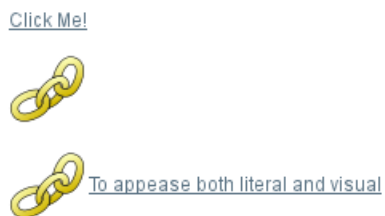
```
// Textual link  
Link link = new Link("Click Me!",  
    new ExternalResource("http://vaadin.com/"));
```

You can use `setIcon()` to make image links as follows:

```
// Image link  
Link iconic = new Link(null,  
    new ExternalResource("http://vaadin.com/"));  
iconic.setIcon(new ThemeResource("img/nicubunu_Chain.png"));  
  
// Image + caption  
Link combo = new Link("To appease both literal and visual",  
    new ExternalResource("http://vaadin.com/"));  
combo.setIcon(new ThemeResource("img/nicubunu_Chain.png"));
```

The resulting links are shown in Figure 6.17, “**Link** Example”. You could add a `display: block` style for the icon element to place the caption below it.

Figure 6.17. Link Example



With the simple constructor used in the above example, the resource is opened in the current window. Using the constructor that takes the target window as a parameter, or by setting the target window with `setTargetName()`, you can open the resource in another window, such as a popup browser window/tab. As the target name is an HTML `target` string managed by the browser, the target can be any window, including windows not managed by the application itself. You can use the special underscored target names, such as `_blank` to open the link to a new browser window or tab.

```
// Hyperlink to a given URL  
Link link = new Link("Take me a away to a faraway land",
```

```
new ExternalResource("http://vaadin.com/");

// Open the URL in a new window/tab
link.setTargetName("_blank");

// Indicate visually that it opens in a new window/tab
link.setIcon(new ThemeResource("icons/external-link.png"));
link.addStyleName("icon-after-caption");
```

Normally, the link icon is before the caption. You can have it right of the caption by reversing the text direction in the containing element.

```
/* Position icon right of the link caption. */
.icon-after-caption {
    direction: rtl;
}
/* Add some padding around the icon. */
.icon-after-caption .v-icon {
    padding: 0 3px;
}
```

The resulting link is shown in Figure 6.18, “Link That Opens a New Window”.

Figure 6.18. Link That Opens a New Window

[Take me a away to a faraway land](#) 

With the `_blank` target, a normal new browser window is opened. If you wish to open it in a popup window (or tab), you need to give a size for the window with `setTargetWidth()` and `setTargetHeight()`. You can control the window border style with `setTargetBorder()`, which takes any of the defined border styles `TARGET_BORDER_DEFAULT`, `TARGET_BORDER_MINIMAL`, and `TARGET_BORDER_NONE`. The exact result depends on the browser.

```
// Open the URL in a popup
link.setTargetName("_blank");
link.setTargetBorder(Link.TARGET_BORDER_NONE);
link.setTargetHeight(300);
link.setTargetWidth(400);
```

In addition to the **Link** component, Vaadin allows alternative ways to make hyperlinks. The **Button** component has a `Reindeer.BUTTON_LINK` style name that makes it look like a hyperlink, while handling clicks in a server-side click listener instead of in the browser. Also, you can make hyperlinks (or any other HTML) in a **Label** in HTML content mode.

6.8.1. CSS Style Rules

```
.v-link { }
  a { }
    .v-icon {}
    span {}
```

The overall style for the **Link** component is `v-link`. The root element contains the `<a href>` hyperlink anchor. Inside the anchor are the icon, with `v-icon` style, and the caption in a text span.

Hyperlink anchors have a number of *pseudo-classes* that are active at different times. An unvisited link has `a:link` class and a visited link `a:visited`. When the mouse pointer hovers over the

link, it will have `a: hover`, and when the mouse button is being pressed over the link, the `a: active` class. When combining the pseudo-classes in a selector, please notice that `a: hover` must come after an `a: link` and `a: visited`, and `a: active` after the `a: hover`.

6.9. TextField

TextField is one of the most commonly used user interface components. It is a **Field** component that allows entering textual values using keyboard.

The following example creates a simple text field:

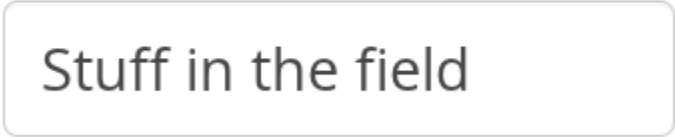
```
// Create a text field
TextField tf = new TextField("A Field");

// Put some initial content in it
tf.setValue("Stuff in the field");
```

The result is shown in Figure 6.19, “**TextField** Example”.

Figure 6.19. TextField Example

A Field



Value changes are handled with a **Property.ValueChangeListener**, as in most other fields. The value can be acquired with `getValue()` directly from the text field, as is done in the example below, or from the property reference of the event.

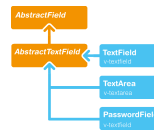
```
// Handle changes in the value
tf.addValueChangeListener(new Property.ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        // Assuming that the value type is a String
        String value = (String) event.getProperty().getValue();

        // Do something with the value
        Notification.show("Value is: " + value);
    }
});

// Fire value changes immediately when the field loses focus
tf.setImmediate(true);
```

As with other event listeners, you can use lambda expression with one parameter to handle the events in Java 8.

Much of the API of **TextField** is defined in **AbstractTextField**, which allows different kinds of text input fields, such as rich text editors, which do not share all the features of the single-line text fields.

Figure 6.20. Text Field Class Relationships

6.9.1. Data Binding

TextField edits **String** values, but you can bind it to any property type that has a proper converter, as described in Section 10.2.3, “Converting Between Property Type and Representation”.

```
// Have an initial data model. As Double is unmodifiable and
// doesn't support assignment from String, the object is
// reconstructed in the wrapper when the value is changed.
Double trouble = 42.0;

// Wrap it in a property data source
final ObjectProperty<Double> property =
    new ObjectProperty<Double>(trouble);

// Create a text field bound to it
// (StringToDoubleConverter is used automatically)
TextField tf = new TextField("The Answer", property);
tf.setImmediate(true);

// Show that the value is really written back to the
// data source when edited by user.
Label feedback = new Label(property);
feedback.setCaption("The Value");
```

When you put a **Table** in editable mode or create fields with a **FieldGroup**, the **DefaultFieldFactory** creates a **TextField** for almost every property type by default. You often need to make a custom factory to customize the creation and to set the field tooltip, validation, formatting, and so on.

See Chapter 10, *Binding Components to Data* for more details on data binding, field factories for **Table** in Section 6.21.3, “Editing the Values in a Table”, and Section 10.4, “Creating Forms by Binding Fields to Items” regarding forms.

6.9.2. String Length

The `setMaxLength()` method sets the maximum length of the input string so that the browser prevents the user from entering a longer one. As a security feature, the input value is automatically truncated on the server-side, as the maximum length setting could be bypassed on the client-side. The maximum length property is defined at **AbstractTextField** level.

Notice that the maximum length setting does not affect the width of the field. You can set the width with `setWidth()`, as with other components. Using *em* widths is recommended to better approximate the proper width in relation to the size of the used font, but the *em* width is not exactly the width of a letter and varies by browser and operating system. There is no standard way in HTML for setting the width exactly to a number of letters (in a monospaced font).

6.9.3. Handling Null Values

As with any field, the value of a **TextField** can be set as *null*. This occurs most commonly when you create a new field without setting a value for it or bind the field value to a data source that allows null values. In such case, you might want to show a special value that stands for the null value. You can set the null representation with the `setNullRepresentation()` method. Most typically, you use an empty string for the null representation, unless you want to differentiate from a string that is explicitly empty. The default null representation is " null ", which essentially warns that you may have forgotten to initialize your data objects properly.

The `setNullSettingAllowed()` controls whether the user can actually input a null value by using the null value representation. If the setting is *false*, which is the default, inputting the null value representation string sets the value as the literal value of the string, not null. This default assumption is a safeguard for data sources that may not allow null values.

```
// Have a property with null value
ObjectProperty<Double> dataModel =
    new ObjectProperty<Double>(new Double(0.0));
dataModel.setValue(null); // Have to set it null here

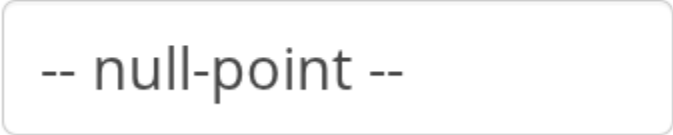
// Create a text field bound to the null data
TextField tf = new TextField("Field Energy (J)", dataModel);
tf.setNullRepresentation("-- null-point --");

// Allow user to input the null value by its representation
tf.setNullSettingAllowed(true);
```

The **Label**, which is bound to the value of the **TextField**, displays a null value as empty. The resulting user interface is shown in Figure 6.21, “Null Value Representation”.

Figure 6.21. Null Value Representation

Field Energy (J)



6.9.4. Text Change Events

Often you want to receive a change event immediately when the text field value changes. The *immediate* mode is not literally immediate, as the changes are transmitted only after the field loses focus. In the other extreme, using keyboard events for every keypress would make typing unbearably slow and also processing the keypresses is too complicated for most purposes. *Text change events* are transmitted asynchronously soon after typing and do not block typing while an event is being processed.

Text change events are received with a **TextChangeListener**, as is done in the following example that demonstrates how to create a text length counter:

```
// Text field with maximum length
final TextField tf = new TextField("My Eventful Field");
tf.setValue("Initial content");
```

```
tf.setMaxLength(20);

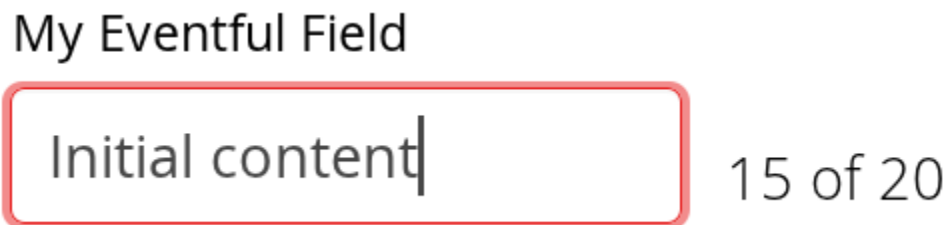
// Counter for input length
final Label counter = new Label();
counter.setValue(tf.getValue().length() +
    " of " + tf.getMaxLength());

// Display the current length interactively in the counter
tf.addChangeListener(new TextChangeListener() {
    public void textChange(TextChangeEvent event) {
        int len = event.getText().length();
        counter.setValue(len + " of " + tf.getMaxLength());
    }
});

// The lazy mode is actually the default
tf.setTextChangeEventMode(TextChangeEventMode.LAZY);
```

The result is shown in Figure 6.22, “Text Change Events”.

Figure 6.22. Text Change Events



The *text change event mode* defines how quickly the changes are transmitted to the server and cause a server-side event. Lazier change events allow sending larger changes in one event if the user is typing fast, thereby reducing server requests.

You can set the text change event mode of a **TextField** with `setTextChangeEventMode()`. The allowed modes are defined in **TextChangeEventMode** enum and are as follows:

TextChangeEventMode.LAZY(default)

An event is triggered when there is a pause in editing the text. The length of the pause can be modified with `setInputEventTimeout()`. As with the *TIMEOUT* mode, a text change event is forced before a possible **ValueChangeEvent**, even if the user did not keep a pause while entering the text.

This is the default mode.

TextChangeEventMode.TIMEOUT

A text change in the user interface causes the event to be communicated to the application after a timeout period. If more changes are made during this period, the event sent to the server-side includes the changes made up to the last change. The length of the timeout can be set with `setInputEventTimeout()`.

If a **ValueChangeEvent** would occur before the timeout period, a **TextChangeEvent** is triggered before it, on the condition that the text content has changed since the previous **TextChangeEvent**.

TextChangeEventMode.EAGER

An event is triggered immediately for every change in the text content, typically caused by a key press. The requests are separate and are processed sequentially one after another. Change events are nevertheless communicated asynchronously to the server, so further input can be typed while event requests are being processed.

6.9.5. CSS Style Rules

```
.v-textfield { }
```

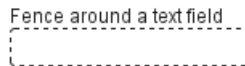
The HTML structure of **TextField** is extremely simple, consisting only of an element with the `v-textfield` style.

For example, the following custom style uses dashed border:

```
.v-textfield-dashing {  
    border:      thin dashed;  
    background: white; /* Has shading image by default */  
}
```

The result is shown in Figure 6.23, “Styling TextField with CSS”.

Figure 6.23. Styling TextField with CSS



The style name for **TextField** is also used in several components that contain a text input field, even if the text input is not an actual **TextField**. This ensures that the style of different text input boxes is similar.

6.10. TextArea

TextArea is a multi-line version of the **TextField** component described in Section 6.9, “**TextField**”.

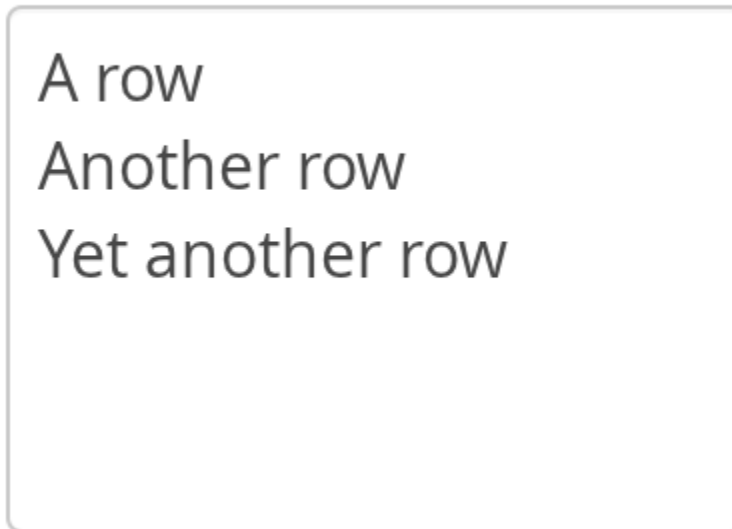
The following example creates a simple text area:

```
// Create the area  
TextArea area = new TextArea("Big Area");  
  
// Put some content in it  
area.setValue("A row\n"+  
              "Another row\n"+  
              "Yet another row");
```

The result is shown in Figure 6.24, “**TextArea** Example”.

Figure 6.24. TextArea Example

Big Area



You can set the number of visible rows with `setRows()` or use the regular `setHeight()` to define the height in other units. If the actual number of rows exceeds the number, a vertical scrollbar will appear. Setting the height with `setRows()` leaves space for a horizontal scrollbar, so the actual number of visible rows may be one higher if the scrollbar is not visible.

You can set the width with the regular `setWidth()` method. Setting the size with the *em* unit, which is relative to the used font size, is recommended.

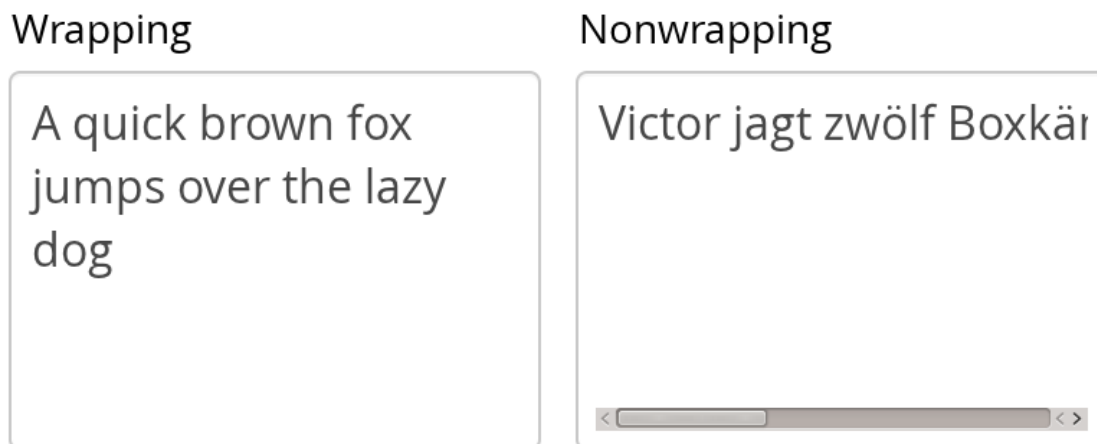
6.10.1. Word Wrap

The `setWordwrap()` sets whether long lines are wrapped (`true` - default) when the line length reaches the width of the writing area. If the word wrap is disabled (`false`), a vertical scrollbar will appear instead. The word wrap is only a visual feature and wrapping a long line does not insert line break characters in the field value; shortening a wrapped line will undo the wrapping.

```
TextArea area1 = new TextArea("Wrapping");
area1.setWordwrap(true); // The default
area1.setValue("A quick brown fox jumps over the lazy dog");

TextArea area2 = new TextArea("Nonwrapping");
area2.setWordwrap(false);
area2.setValue("Victor jagt zw\u00f6lf Boxk\u00e4mpfer quer "+
               "\u00ber den Sylter Deich");
```

The result is shown in Figure 6.25, “Word Wrap in **TextArea**”.

Figure 6.25. Word Wrap in TextArea

6.10.2. CSS Style Rules

```
.v-textarea { }
```

The HTML structure of **TextArea** is extremely simple, consisting only of an element with `v-textarea` style.

CSS Styling

6.11. PasswordField

The **PasswordField** is a variant of **TextField** that hides the typed input from visual inspection.

```
PasswordField tf = new PasswordField("Keep it secret");
```

The result is shown in Figure 6.26, "**PasswordField**".

Figure 6.26. PasswordField

You should note that the **PasswordField** hides the input only from "over the shoulder" visual observation. Unless the server connection is encrypted with a secure connection, such as HTTPS, the input is transmitted in clear text and may be intercepted by anyone with low-level access to the network. Also phishing attacks that intercept the input in the browser may be possible by exploiting JavaScript execution security holes in the browser.

6.11.1. CSS Style Rules

```
.v-textfield { }
```

The **PasswordField** does not have its own CSS style name but uses the same `v-textfield` style as the regular **TextField**. See Section 6.9.5, “CSS Style Rules” for information on styling it.

CSS Styling

6.12. RichTextArea

The **RichTextArea** field allows entering or editing formatted text. The toolbar provides all basic editing functionalities. The text content of **RichTextArea** is represented in HTML format. **RichTextArea** inherits **TextField** and does not add any API functionality over it. You can add new functionality by extending the client-side components **VRichTextArea** and **VRichTextToolbar**.

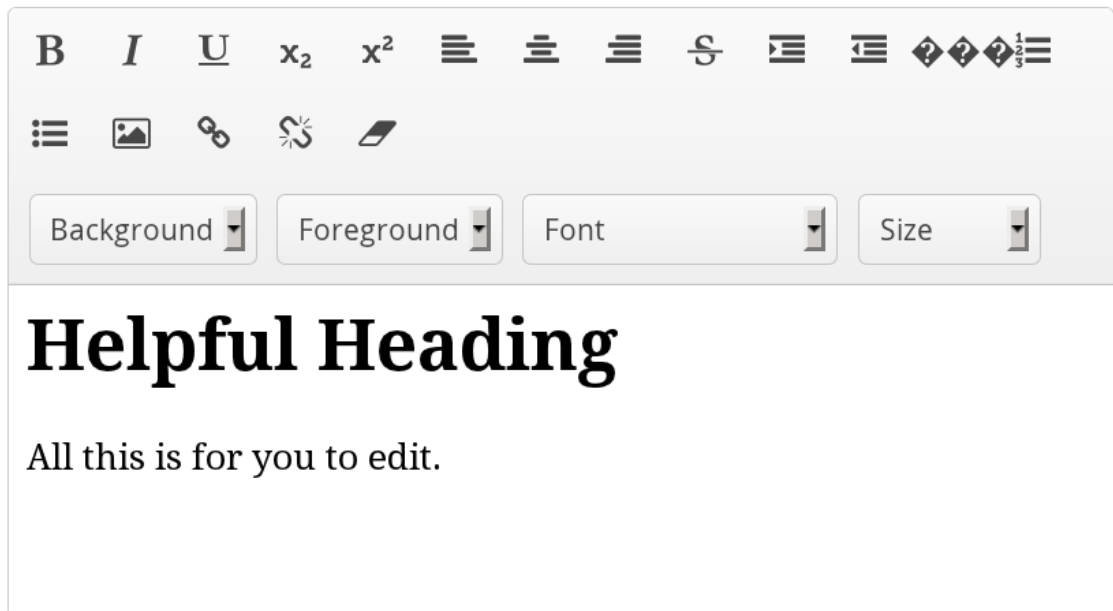
As with **TextField**, the textual content of the rich text area is the **Property** of the field and can be set with `setValue()` and read with `getValue()`.

```
// Create a rich text area
final RichTextArea rtarea = new RichTextArea();
rtarea.setCaption("My Rich Text Area");

// Set initial content as HTML
rtarea.setValue("<h1>Hello</h1>\n" +
    "<p>This rich text area contains some text.</p>");
```

Figure 6.27. Rich Text Area Component

My Rich Textarea



Above, we used context-specific tags such as `<h1>` in the initial HTML content. The rich text area component does not allow creating such tags, only formatting tags, but it does preserve

them unless the user edits them away. Any non-visible whitespace such as the new line character (`\n`) are removed from the content. For example, the value set above will be as follows when read from the field with `getValue()`:

```
<h1>Hello</h1> <p>This rich text area contains some text.</p>
```



Cross-Site Scripting Warning

The user input from a **RichTextArea** is transmitted as HTML from the browser to server-side and is not sanitized. As the entire purpose of the **RichTextArea** component is to allow input of formatted text, you can not sanitize it just by removing all HTML tags. Also many attributes, such as `style`, should pass through the sanitization.

See Section 12.8.1, “Sanitizing User Input to Prevent Cross-Site Scripting” for more details on Cross-Site scripting vulnerabilities and sanitization of user input.

6.12.1. CSS Style Rules

```
.v-richtextarea { }  
.v-richtextarea .gwt-RichTextToolbar { }  
.v-richtextarea .gwt-RichTextArea { }
```

The rich text area consists of two main parts: the toolbar with overall style `.gwt-RichTextToolbar` and the editor area with style `.gwt-RichTextArea`. The editor area obviously contains all the elements and their styles that the HTML content contains. The toolbar contains buttons and drop-down list boxes with the following respective style names:

```
.gwt-ToggleButton { }  
.gwt-ListBox { }
```

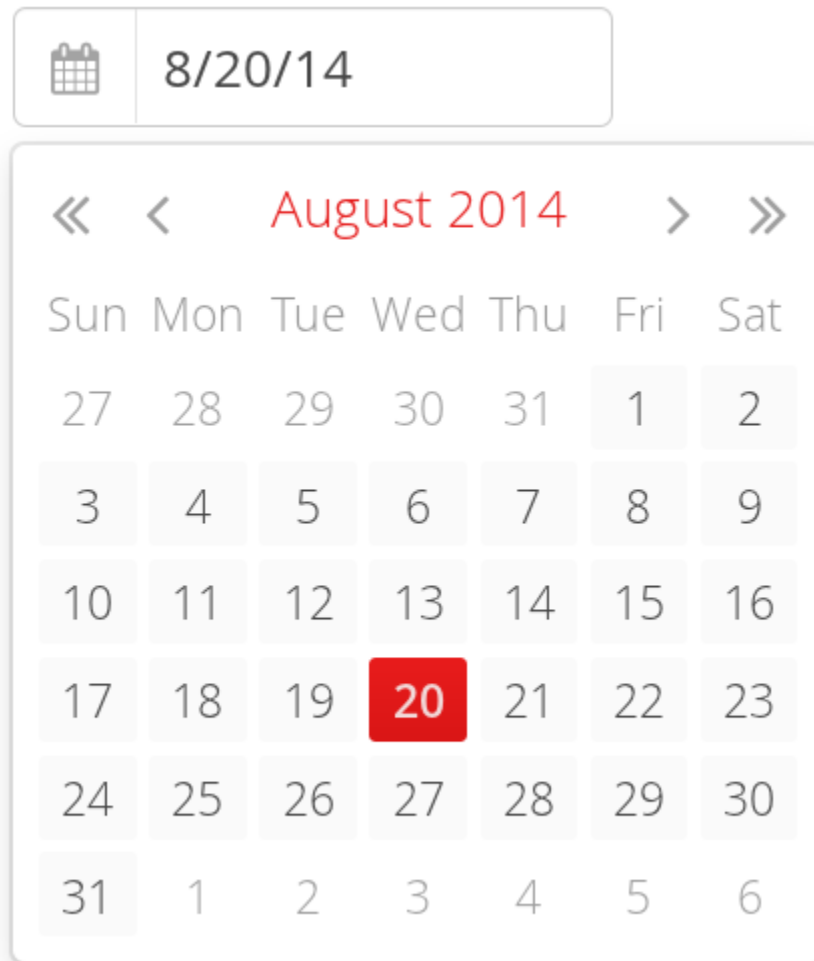
6.13. Date and Time Input with DateField

The **DateField** component provides the means to display and input date and time. The field comes in two variations: **PopupDateField**, with a numeric input box and a popup calendar view, and **InlineDateField**, with the calendar view always visible. The **DateField** base class defaults to the popup variation.

The example below illustrates the use of the **DateField** baseclass, which is equivalent to the **PopupDateField**. We set the initial time of the date field to current time by using the default constructor of the **java.util.Date** class.

```
// Create a DateField with the default style  
DateField date = new DateField();  
  
// Set the date and time to present  
date.setValue(new Date());
```

The result is shown in Figure 6.28, “**DateField (PopupDateField)** for Selecting Date and Time”.

Figure 6.28. DateField (PopupDateField) for Selecting Date and Time

6.13.1. PopupDateField

The **PopupDateField** provides date input using a text box for the date and time. As the **DateField** defaults to this component, the use is exactly the same as described earlier. Clicking the handle right of the date opens a popup view for selecting the year, month, and day, as well as time. Also the Down key opens the popup. Once opened, the user can navigate the calendar using the cursor keys.

The date and time selected from the popup are displayed in the text box according to the default date and time format of the current locale, or as specified with `setDateFormat()`. The same format definitions are used for parsing user input.

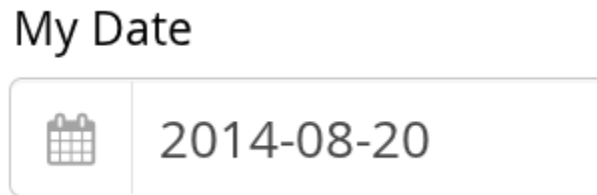
Date and Time Format

The date and time are normally displayed according to the default format for the current locale (see Section 6.3.5, “Locale”). You can specify a custom format with `setDateFormat()`. It takes a format string that follows the format of the **SimpleDateFormat** in Java.

```
// Display only year, month, and day in ISO format
date.setDateFormat("yyyy-MM-dd");
```

The result is shown in Figure 6.29, “Custom Date Format for **PopupDateField**”.

Figure 6.29. Custom Date Format for PopupDateField



The same format specification is also used for parsing user-input date and time, as described later.

Input Prompt

Like other fields that have a text box, **PopupDateField** allows an input prompt that is visible until the user has input a value. You can set the prompt with `setInputPrompt`.

```
PopupDateField date = new PopupDateField();
```

```
// Set the prompt
date.setInputPrompt("Select a date");
```

```
// Set width explicitly to accommodate the prompt
date.setWidth("10em");
```

The date field doesn’t automatically scale to accommodate the prompt, so you need to set it explicitly with `setWidth()`.

The input prompt is not available in the **DateField** superclass.

CSS Style Rules

```
.v-datefield, v-datefield-popupcalendar {}
  .v-textfield, v-datefield-textfield {}
  .v-datefield-button {}
```

The top-level element of **DateField** and all its variants have `v-datefield` style. The base class and the **PopupDateField** also have the `v-datefield-popupcalendar` style.

In addition, the top-level element has a style that indicates the resolution, with `v-datefield-basename` and an extension, which is one of `full`, `day`, `month`, or `year`. The `-full` style is enabled when the resolution is smaller than a day. These styles are used mainly for controlling the appearance of the popup calendar.

The text box has `v-textfield` and `v-datefield-textfield` styles, and the calendar button `v-datefield-button`.

Once opened, the calendar popup has the following styles at the top level:

```
.v-datefield-popup {}  
  .v-popupcontent {}  
    .v-datefield-calendarpanel {}
```

The top-level element of the floating popup calendar has `.v-datefield-popup` style. Observe that the popup frame is outside the HTML structure of the component, hence it is not enclosed in the `v-datefield` element and does not include any custom styles. 5752. The content in the *[literal]*`v-datefield-calendarpanel#` is the same as in **InlineDateField**, as described in Section 6.13.2, “**InlineDateField**”.

6.13.2. InlineDateField

The **InlineDateField** provides a date picker component with a month view. The user can navigate months and years by clicking the appropriate arrows. Unlike with the popup variant, the month view is always visible in the inline field.

```
// Create a DateField with the default style  
InlineDateField date = new InlineDateField();  
  
// Set the date and time to present  
date.setValue(new java.util.Date());
```

The result is shown in Figure 6.30, “Example of the **InlineDateField**”.

Figure 6.30. Example of the InlineDateField



The user can also navigate the calendar using the cursor keys.

CSS Style Rules

```
.v-datefield {}
.v-datefield-calendarpanel {}
.v-datefield-calendarpanel-header {}
.v-datefield-calendarpanel-prevyear {}
.v-datefield-calendarpanel-prevmonth {}
.v-datefield-calendarpanel-month {}
.v-datefield-calendarpanel-nextmonth {}
.v-datefield-calendarpanel-nextyear {}
.v-datefield-calendarpanel-body {}
.v-datefield-calendarpanel-weekdays,
.v-datefield-calendarpanel-weeknumbers {}
.v-datefield-calendarpanel-weeknumbers .v-first {}
.v-datefield-calendarpanel-weeknumbers .v-last {}
.v-datefield-calendarpanel-weeknumber {}
.v-datefield-calendarpanel-day {}
.v-datefield-calendarpanel-time {}
.v-datefield-time .v-select {}
.v-datefield-time .v-label {}
```

The top-level element has the `v-datefield` style. In addition, the top-level element has a style name that indicates the resolution of the calendar, with `v-datefield-` baseline and an extension, which is one of `full`, `day`, `month`, or `year`. The `-full` style is enabled when the resolution is smaller than a day.

The `v-datefield-calendarpanel-weeknumbers` and `v-datefield-calendarpanel-weeknumber` styles are enabled when the week numbers are enabled. The former controls the appearance of the weekday header and the latter the actual week numbers.

The other style names should be self-explanatory. For weekdays, the `v-first` and `v-last` styles allow making rounded endings for the weekday bar.

6.13.3. Date and Time Resolution

In addition to display a calendar with dates, **DateField** can also display the time in hours and minutes, or just the month or year. The visibility of the input components is controlled by *time resolution*, which you can set with `setResolution()`. The method takes as its parameters the lowest visible component, `DateField.Resolution.DAY` for just dates and `DateField.Resolution.MIN` for dates with time in hours and minutes. Please see the API Reference for the complete list of resolution parameters.

6.13.4. DateField Locale

The date and time are displayed according to the locale of the user, as reported by the browser. You can set a custom locale with the `setLocale()` method of **AbstractComponent**, as described in Section 6.3.5, “Locale”. Only Gregorian calendar is supported.

6.14. Button

The **Button** component is normally used for initiating some action, such as finalizing input in forms. When the user clicks a button, a **Button.ClickEvent** is fired, which can be handled with a `Button.ClickListener` in the `buttonClick()` method.

You can handle button clicks with an anonymous class as follows:


```

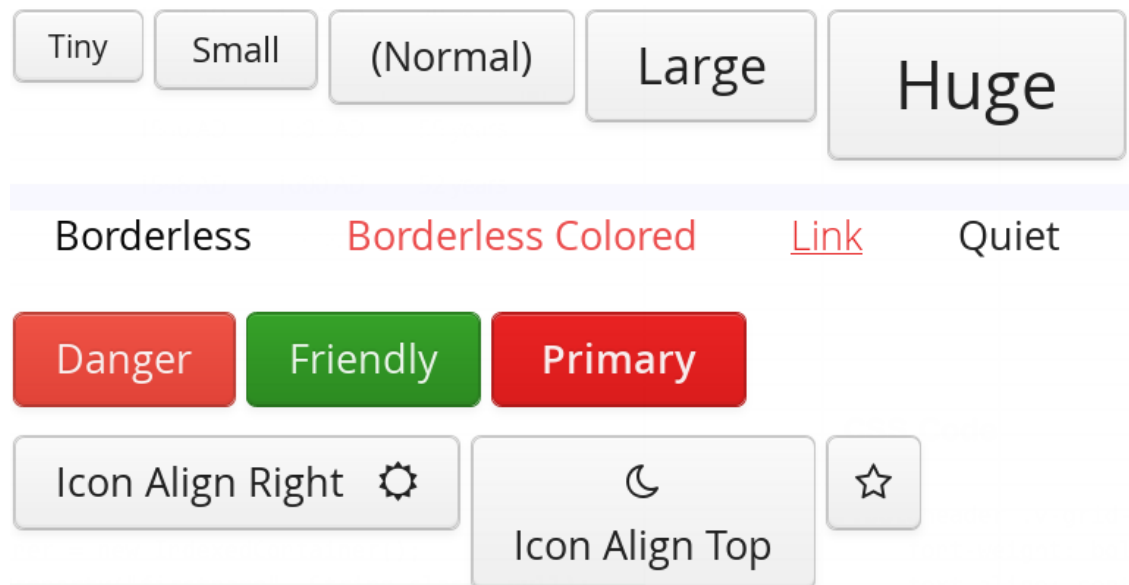
Button button = new Button("Do not press this button");

button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        Notification.show("Do not press this button again");
    }
});

```

The result is shown in Figure 6.31, “Button in Different Styles of Valo Theme”. The listener can also be given in the constructor, which is often perhaps simpler.

Figure 6.31. Button in Different Styles of Valo Theme



If you handle several buttons in the same listener, you can differentiate between them either by comparing the **Button** object reference returned by the `getButton()` method of **Button.ClickEvent** to a kept reference. For a detailed description of these patterns together with some examples, please see Section 4.4, “Events and Listeners”.

6.14.1. CSS Style Rules

```

.v-button { }
  .v-button-wrap { }
    .v-button-caption { }

```

A button has an overall `v-button` style. The caption has `v-button-caption` style. There is also an intermediate wrap element, which may help in styling in some cases.

Some built-in themes contain a small style, which you can enable by adding `Reindeer.BUTTON_SMALL`, etc. The **BaseTheme** also has a `BUTTON_LINK` style, which makes the button look like a hyperlink.

6.15. CheckBox

CheckBox is a two-state selection component that can be either checked or unchecked. The caption of the check box will be placed right of the actual check box. Vaadin provides two ways

to create check boxes: individual check boxes with the **CheckBox** component described in this section and check box groups with the **OptionGroup** component in multiple selection mode, as described in Section 6.19, “**OptionGroup**”.

Clicking on a check box will change its state. The state is a **Boolean** property that you can set with the `setValue()` method and obtain with the `getValue()` method of the **Property** interface. Changing the value of a check box will cause a **ValueChangeEvent**, which can be handled by a **ValueChangeListener**.

```
CheckBox checkbox1 = new CheckBox("Box with no Check");
CheckBox checkbox2 = new CheckBox("Box with a Check");

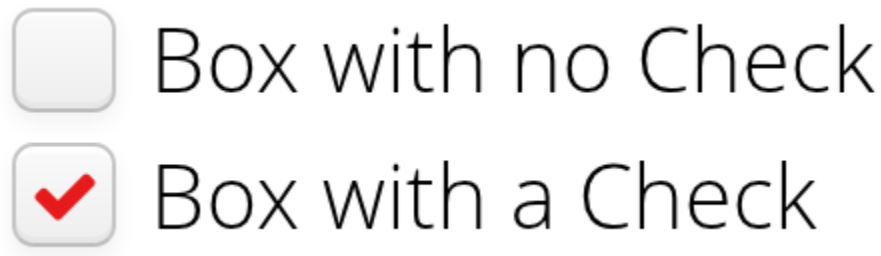
checkboxbox2.setValue(true);

checkboxbox1.addValueChangeListener(event -> // Java 8
    checkbox2.setValue(! checkbox1.getValue()));

checkboxbox2.addValueChangeListener(event -> // Java 8
    checkbox1.setValue(! checkbox2.getValue()));
```

The result is shown in Figure 6.32, “An Example of a Check Box”.

Figure 6.32. An Example of a Check Box



For an example on the use of check boxes in a table, see Section 6.21, “**Table**”.

6.15.1. CSS Style Rules

```
.v-checkbox { }
.v-checkbox > input { }
.v-checkbox > label { }
```

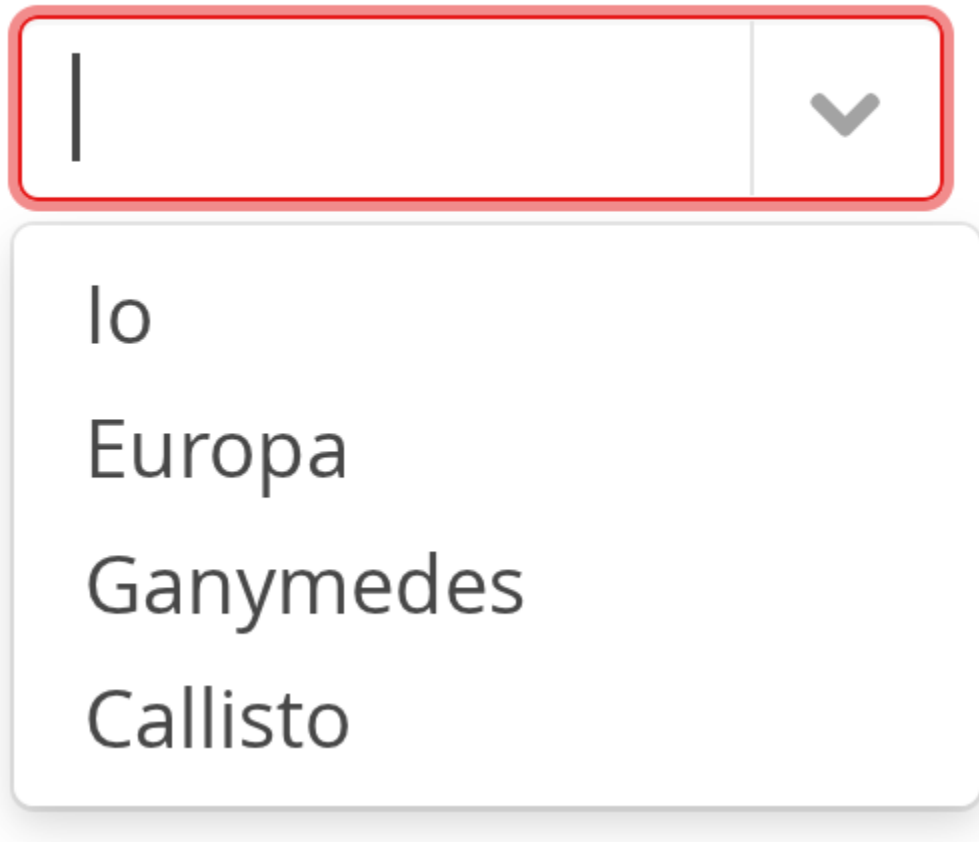
The top-level element of a **CheckBox** has the `v-checkbox` style. It contains two sub-elements: the actual check box `input` element and the `label` element. If you want to have the label on the left, you can change the positions with “`direction: rtl`” for the top element.

6.16. ComboBox

ComboBox is a selection component allows selecting an item from a drop-down list. The component also has a text field area, which allows entering search text by which the items shown in the drop-down list are filtered. Common selection component features are described in Section 6.5, “Selection Components”.

Figure 6.33. The ComboBox Component

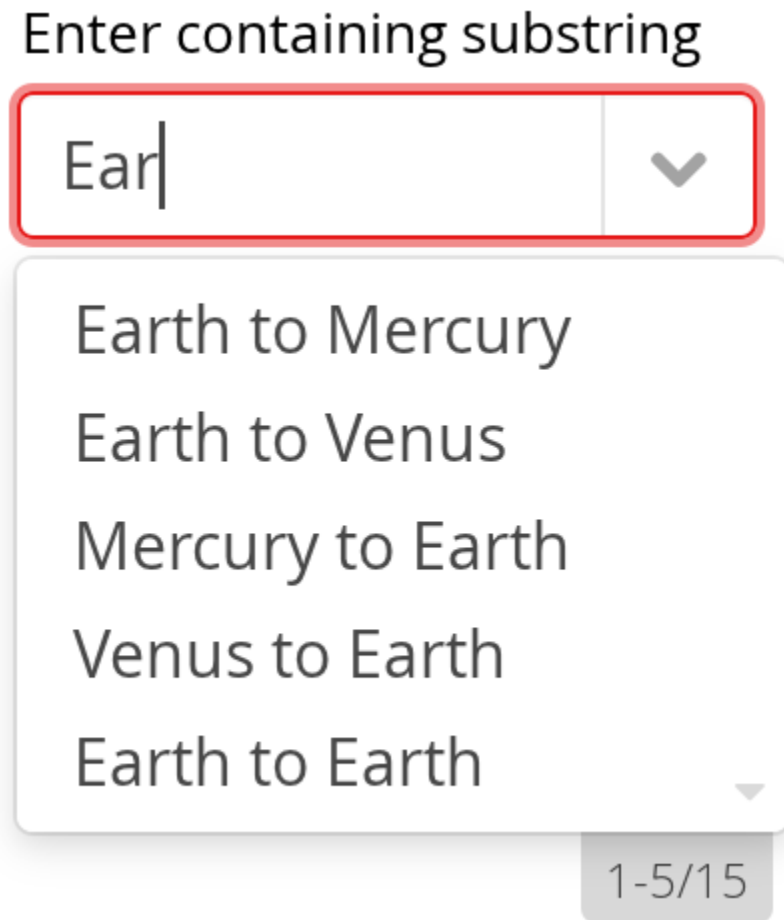
My Select



ComboBox supports adding new items when the user presses **Enter**.

6.16.1. Filtered Selection

ComboBox allows filtering the items available for selection in the drop-down list by the text entered in the input box.

Figure 6.34. Filtered Selection in ComboBox

Pressing **Enter** will complete the item in the input box. Pressing **Up** and **Down** arrow keys can be used for selecting an item from the drop-down list. The drop-down list is paged and clicking on the scroll buttons will change to the next or previous page. The list selection can also be done with the arrow keys on the keyboard. The shown items are loaded from the server as needed, so the number of items held in the component can be quite large. The number of matching items is displayed by the drop-down list.

Filtering is enabled by setting a *filtering mode* with `setFilteringMode()`.

```
cb.setFilteringMode(FilteringMode.CONTAINS);
```

The modes defined in the **FilteringMode** enum are as follows:

CONTAINS

Matches any item that contains the string given in the text field part of the component.

STARTSWITH

Matches only items that begin with the given string.

OFF(default)

Filtering is by default off and all items are shown all the time.

The above example uses the containment filter that matches to all items containing the input string. As shown in Figure 6.34, “Filtered Selection in **ComboBox**” below, when we type some text in the input area, the drop-down list will show all the matching items.

6.16.2. CSS Style Rules

```
.v-filterselect { }
  .v-filterselect-input { }
  .v-filterselect-button { }

// Under v-overlay-container
.v-filterselect-suggestpopup { }
  .popupContent { }
    .v-filterselect-prevpage,
    .v-filterselect-prevpage-off { }
    .v-filterselect-suggestmenu { }
    .gwt-MenuItem { }
    .v-filterselect-nextpage,
    .v-filterselect-nextpage-off { }
    .v-filterselect-status { }
```

In its default state, only the input field of the **ComboBox** component is visible. The entire component is enclosed in `v-filterselect` style (a legacy remnant), the input field has `v-filterselect-input` style and the button in the right end that opens and closes the drop-down result list has `v-filterselect-button` style.

The drop-down result list has an overall `v-filterselect-suggestpopup` style. It contains the list of suggestions with `v-filterselect-suggestmenu` style. When there are more items that fit in the menu, navigation buttons with `v-filterselect-prevpage` and `v-filterselect-nextpage` styles are shown. When they are not shown, the elements have `-off` suffix. The status bar in the bottom that shows the paging status has `v-filterselect-status` style.

6.17. ListSelect

The **ListSelect** component is list box that shows the selectable items in a vertical list. If the number of items exceeds the height of the component, a scrollbar is shown. The component allows both single and multiple selection modes, which you can set with `setMultiSelect()`. It is visually identical in both modes.

```
// Create the selection component
ListSelect select = new ListSelect("The List");

// Add some items (here by the item ID as the caption)
select.addItem("Mercury", "Venus", "Earth", ...);

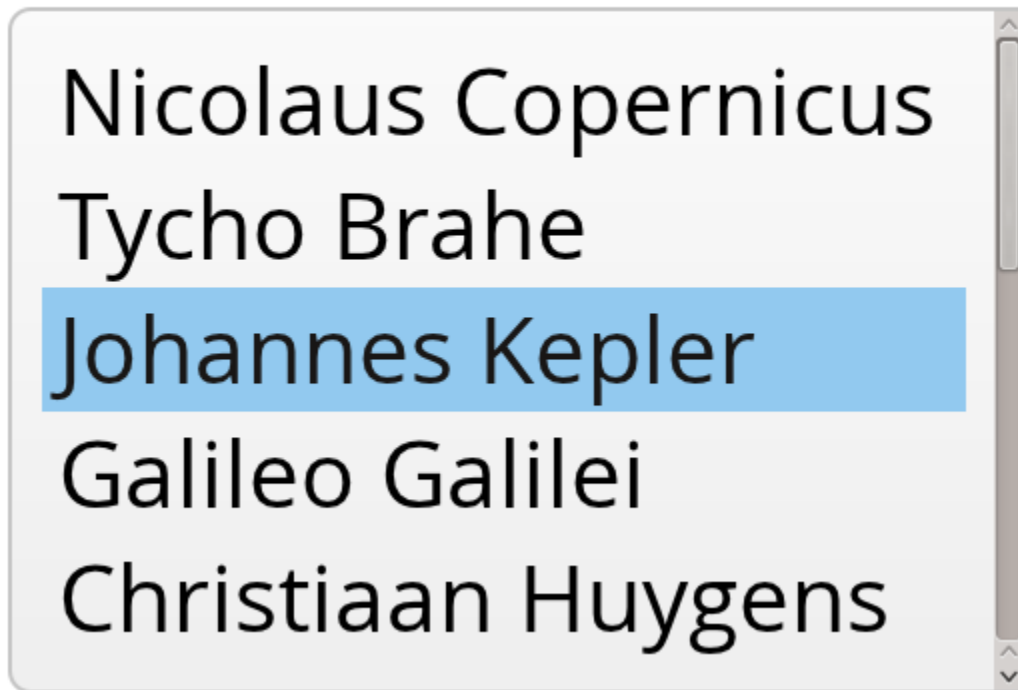
select.setNullSelectionAllowed(false);

// Show 5 items and a scrollbar if there are more
select.setRows(5);
```

The number of visible items is set with `setRows()`.

Figure 6.35. The ListSelect Component

The List



Common selection component features are described in Section 6.5, “Selection Components”.

6.17.1. CSS Style Rules

```
.v-select {}  
  .v-select-select {}  
    option {}
```

The component has an overall `v-select` style. The native `<select>` element has `v-select-select` style. The items are represented as `<option>` elements.

6.18. NativeSelect

NativeSelect is a drop-down selection component implemented with the native selection input of web browsers, using the HTML `<select>` element.

```
// Create the selection component  
NativeSelect select = new NativeSelect("Native Selection");  
  
// Add some items  
select.addItem("Mercury", "Venus", ...);
```

The `setColumns()` allows setting the width of the list as "columns", which is a measure that depends on the browser.

Figure 6.36. The NativeSelect Component

Common selection component features are described in Section 6.5, “Selection Components”.

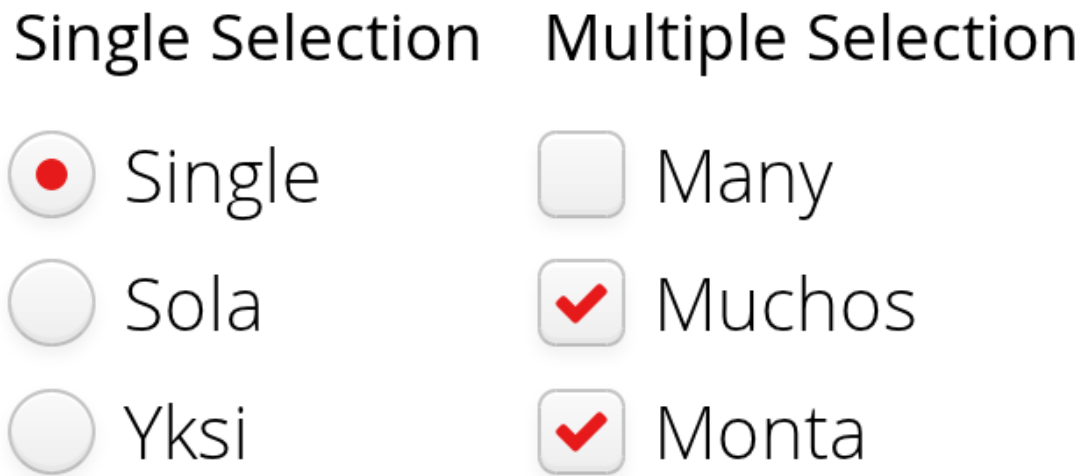
6.18.1. CSS Style Rules

```
.v-select {}  
  .v-select-select {}
```

The component has a `v-select` overall style. The native `select` element has `v-select-select` style.

6.19. OptionGroup

OptionGroup is a selection component that allows selection from a group of radio buttons in single selection mode. In multiple selection mode, the items show up as check boxes. The common selection component features are described in Section 6.5, “Selection Components”.

Figure 6.37. Option Button Group in Single and Multiple Selection Mode

Option group is by default in single selection mode. Multiple selection is enabled with `setMultiSelect()`.

```
// A single-select radio button group
OptionGroup single = new OptionGroup("Single Selection");
single.addItem("Single", "Sola", "Yksi");

// A multi-select check box group
OptionGroup multi = new OptionGroup("Multiple Selection");
multi.setMultiSelect(true);
multi.addItem("Many", "Muchos", "Monta");
```

Figure 6.37, “Option Button Group in Single and Multiple Selection Mode” shows the **OptionGroup** in both single and multiple selection mode.

You can also create check boxes individually using the **CheckBox** class, as described in Section 6.15, “**CheckBox**”. The advantages of the **OptionGroup** component are that as it maintains the individual check box objects, you can get an array of the currently selected items easily, and that you can easily change the appearance of a single component.

6.19.1. Disabling Items

You can disable individual items in an **OptionGroup** with `setItemEnabled()`. The user can not select or deselect disabled items in multi-select mode, but in single-select mode the user can change the selection from a disabled to an enabled item. The selections can be changed programmatically regardless of whether an item is enabled or disabled. You can find out whether an item is enabled with `isItemEnabled()`.

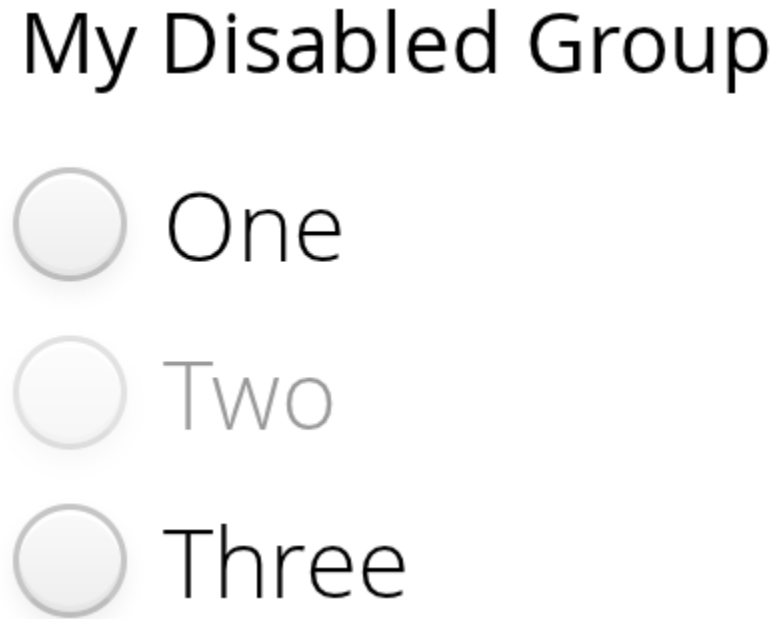
The `setItemEnabled()` identifies the item to be disabled by its item ID.

```
// Have an option group with some items
OptionGroup group = new OptionGroup("My Disabled Group");
group.addItem("One", "Two", "Three");

// Disable one item by its item ID
group.setItemEnabled("Two", false);
```


The item IDs are also used for the captions in this example. The result is shown in Figure 6.38, “**OptionGroup** with a Disabled Item”.

Figure 6.38. OptionGroup with a Disabled Item



Setting an item as disabled turns on the `v-disabled` style for it.

6.19.2. CSS Style Rules

```
.v-select-optiongroup {}  
  .v-select-option.v-checkbox {}  
  .v-select-option.v-radiobutton {}
```

The `v-select-optiongroup` is the overall style for the component. Each check box will have the `v-checkbox` style, borrowed from the **CheckBox** component, and each radio button the `v-radiobutton` style. Both the radio buttons and check boxes will also have the `v-select-option` style that allows styling regardless of the option type. Disabled items have additionally the `v-disabled` style.

Horizontal Layout

The options are normally laid out vertically. You can use horizontal layout by setting `display: inline-block` for the options. The `nowrap` setting for the overall element prevents wrapping if there is not enough horizontal space in the layout, or if the horizontal width is undefined.

```
/* Lay the options horizontally */  
.v-select-optiongroup-horizontal .v-select-option {  
  display: inline-block;  
}  
  
/* Avoid wrapping if the layout is too tight */  
.v-select-optiongroup-horizontal {  
  white-space: nowrap;  
}
```

```
/* Some extra spacing is needed */  
.v-select-optiongroup-horizontal  
  .v-select-option.v-radiobutton {  
    padding-right: 10px;  
  }  
}
```

Use of the above rules requires setting a custom `horizontal` style name for the component. The result is shown in Figure 6.39, “Horizontal **OptionGroup**”.

Figure 6.39. Horizontal OptionGroup

Horizontal Group

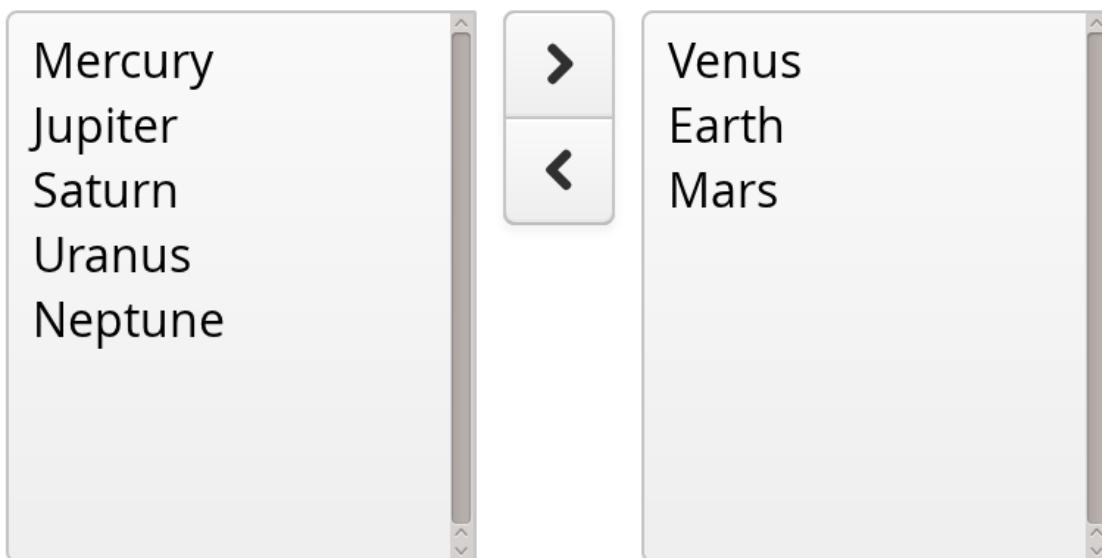


6.20. TwinColSelect

The **TwinColSelect** field provides a multiple selection component that shows two lists side by side, with the left column containing unselected items and the right column the selected items. The user can select items from the list on the left and click on the ">>" button to move them to the list on the right. Items can be deselected by selecting them in the right list and clicking on the "<<" button.

Figure 6.40. Twin Column Selection

Select Targets



TwinColSelect is always in multi-select mode, so its property value is always a collection of the item IDs of the selected items, that is, the items in the right column.

The selection columns can have their own captions, separate from the overall component caption, which is managed by the containing layout. You can set the column captions with `setLeftColumnCaption()` and `setRightColumnCaption()`.

```
TwinColSelect select = new TwinColSelect("Select Targets");

// Put some items in the select
select.addItem("Mercury", "Venus", "Earth", "Mars",
              "Jupiter", "Saturn", "Uranus", "Neptune");

// Few items, so we can set rows to match item count
select.setRows(select.size());

// Preselect a few items by creating a set
select.setValue(new HashSet<String>(
    Arrays.asList("Venus", "Earth", "Mars")));

// Handle value changes
select.addValueChangeListener(event -> // Java 8
    layout.addComponent(new Label("Selected: " +
        event.getProperty().getValue())));
```

The resulting component is shown in Figure 6.40, “Twin Column Selection”.

The `setRows()` method sets the height of the component by the number of visible items in the selection boxes. Setting the height with `setHeight()` to a defined value overrides the rows setting.

Common selection component features are described in Section 6.5, “Selection Components”.

6.20.1. CSS Style Rules

```
.v-select-twincol {}
.v-select-twincol-options-caption {}
.v-select-twincol-selections-caption {}
.v-select-twincol-options {}
.v-select-twincol-buttons {}
.v-button {}
.v-button-wrap {}
.v-button-caption {}
.v-select-twincol-deco {}
.v-select-twincol-selections {}
```

The **TwinColSelect** component has an overall `v-select-twincol` style. If set, the left and right column captions have `v-select-twincol-options-caption` and `v-select-twincol-selections-caption` style names, respectively. The left box, which displays the unselected items, has `v-select-twincol-options-caption` style and the right box, which displays the selected items, has `v-select-twincol-selections` style. Between them is the button area, which has overall `v-select-twincol-buttons` style; the actual buttons reuse the styles for the **Button** component. Between the buttons is a divider element with `v-select-twincol-deco` style.

6.21. Table

The **Table** component is intended for presenting tabular data organized in rows and columns. The **Table** is one of the most versatile components in Vaadin. Table cells can include text or

arbitrary UI components. You can easily implement editing of the table data, for example clicking on a cell could change it to a text field for editing.

The data contained in a **Table** is managed using the Data Model of Vaadin (see Chapter 10, *Binding Components to Data*), through the **Container** interface of the **Table**. This makes it possible to bind a table directly to a data source, such as a database query. Only the visible part of the table is loaded into the browser and moving the visible window with the scrollbar loads content from the server. While the data is being loaded, a tooltip will be displayed that shows the current range and total number of items in the table. The rows of the table are *items* in the container and the columns are *properties*. Each table row (item) is identified with an *item identifier* (IID), and each column (property) with a *property identifier* (PID).

When creating a table, you first need to define columns with `addContainerProperty()`. This method comes in two flavors. The simpler one takes the property ID of the column and uses it also as the caption of the column. The more complex one allows differing PID and header for the column. This may make, for example, internationalization of table headers easier, because if a PID is internationalized, the internationalization has to be used everywhere where the PID is used. The complex form of the method also allows defining an icon for the column from a resource. The "default value" parameter is used when new properties (columns) are added to the table, to fill in the missing values. (This default has no meaning in the usual case, such as below, where we add items after defining the properties.)

```
Table table = new Table("The Brightest Stars");

// Define two columns for the built-in container
table.addContainerProperty("Name", String.class, null);
table.addContainerProperty("Mag", Float.class, null);

// Add a row the hard way
Object newItemId = table.addItem();
Item row1 = table.getItem(newItemId);
row1.getItemProperty("Name").setValue("Sirius");
row1.getItemProperty("Mag").setValue(-1.46f);

// Add a few other rows using shorthand addItem()
table.addItem(new Object[] { "Canopus", -0.72f }, 2);
table.addItem(new Object[] { "Arcturus", -0.04f }, 3);
table.addItem(new Object[] { "Alpha Centauri", -0.01f }, 4);

// Show exactly the currently contained rows (items)
table.setPageLength(table.size());
```

In this example, we used an increasing **Integer** object as the Item Identifier, given as the second parameter to `addItem()`. The actual rows are given simply as object arrays, in the same order in which the properties were added. The objects must be of the correct class, as defined in the `addContainerProperty()` calls.

Figure 6.41. Basic Table Example

The Brightest Stars

Name	Mag
Sirius	-1,46
Canopus	-0,72
Arcturus	-0,04
Alpha Centauri	-0,01

Scalability of the **Table** is largely dictated by the container. The default **IndexedContainer** is relatively heavy and can cause scalability problems, for example, when updating the values. Use of an optimized application-specific container is recommended. Table does not have a limit for the number of items and is just as fast with hundreds of thousands of items as with just a few. With the current implementation of scrolling, there is a limit of around 500 000 rows, depending on the browser and the pixel height of rows.

Common selection component features are described in Section 6.5, “Selection Components”.

6.21.1. Selecting Items in a Table

The **Table** allows selecting one or more items by clicking them with the mouse. When the user selects an item, the IID of the item will be set as the property of the table and a **ValueChangeEvent** is triggered. To enable selection, you need to set the table *selectable*. You will also need to set it as *immediate* in most cases, as we do below, because without it, the change in the property will not be communicated immediately to the server.

The following example shows how to enable the selection of items in a **Table** and how to handle **ValueChangeEvent** events that are caused by changes in selection. You need to handle the event with the `valueChange()` method of the **Property.ValueChangeListener** interface.

```
// Allow selecting items from the table.
table.setSelectable(true);
```

```
// Send changes in selection immediately to server.
table.setImmediate(true);

// Shows feedback from selection.
final Label current = new Label("Selected: -");

// Handle selection change.
table.addValueChangeListener(new Property.ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        current.setValue("Selected: " + table.getValue());
    }
});
```

Figure 6.42. Table Selection Example

First Name	Last Name	Year	
Nicolaus	Copernicus	1473	
Tycho	Brahe	1546	
Giordano	Bruno	1548	
Galileo	Galilei	1564	
Johannes	Kepler	1571	

Selected: 2

If the user clicks on an already selected item, the selection will be deselected and the table property will have *null* value. You can disable this behaviour by setting `setNullSelectionAllowed(false)` for the table.

The selection is the value of the table's property, so you can get it with `getValue()`. You can get it also from a reference to the table itself. In single selection mode, the value is the item identifier of the selected item or *null* if no item is selected. In multiple selection mode (see below), the value is a **Set** of item identifiers. Notice that the set is unmodifiable, so you can not simply change it to change the selection.

Multiple Selection Mode

A table can also be in *multiselect* mode, where a user can select multiple items by clicking them with left mouse button while holding the **Ctrl** key (or **Meta** key) pressed. If **Ctrl** is not held, clicking an item will select it and other selected items are deselected. The user can select a range by selecting an item, holding the **Shift** key pressed, and clicking another item, in which case all the items between the two are also selected. Multiple ranges can be selected by first selecting a range, then selecting an item while holding **Ctrl**, and then selecting another item with both **Ctrl** and **Shift** pressed.

The multiselect mode is enabled with the `setMultiSelect()` method of the **AbstractSelect** superclass of **Table**. Setting table in multiselect mode does not implicitly set it as *selectable*, so it must be set separately.

The `setMultiSelectMode()` property affects the control of multiple selection: *MultiSelectMode.DEFAULT* is the default behaviour, which requires holding the **Ctrl** (or **Meta**) key pressed while selecting items, while in *MultiSelectMode.SIMPLE* holding the **Ctrl** key is not needed. In the simple mode, items can only be deselected by clicking them.

6.21.2. Table Features

Page Length and Scrollbar

The default style for **Table** provides a table with a scrollbar. The scrollbar is located at the right side of the table and becomes visible when the number of items in the table exceeds the page length, that is, the number of visible items. You can set the page length with `setPageLength()`.

Setting the page length to zero makes all the rows in a table visible, no matter how many rows there are. Notice that this also effectively disables buffering, as all the entire table is loaded to the browser at once. Using such tables to generate reports does not scale up very well, as there is some inevitable overhead in rendering a table with Ajax. For very large reports, generating HTML directly is a more scalable solution.

Resizing Columns

You can set the width of a column programmatically from the server-side with `setColumnWidth()`. The column is identified by the property ID and the width is given in pixels.

The user can resize table columns by dragging the resize handle between two columns. Resizing a table column causes a **ColumnResizeEvent**, which you can handle with a **Table.ColumnnResizeListener**. The table must be set in immediate mode if you want to receive the resize events immediately, which is typical.

```
table.addColumnResizeListener(new Table.ColumnResizeListener() {
    public void columnResize(ColumnResizeEvent event) {
        // Get the new width of the resized column
        int width = event.getCurrentWidth();

        // Get the property ID of the resized column
        String column = (String) event.getPropertyId();

        // Do something with the information
        table.setColumnFooter(column, String.valueOf(width) + "px");
    }
});

// Must be immediate to send the resize events immediately
table.setImmediate(true);
```

See Figure 6.43, “Resizing Columns” for a result after the columns of a table has been resized.

Figure 6.43. Resizing Columns

ColumnResize Events	
NAME ▼	BORN IN
Väisälä	1891
Valtaoja	1951
Galileo	1564
124px	248px

Reordering Columns

If `setColumnReorderingAllowed(true)` is set, the user can reorder table columns by dragging them with the mouse from the column header,

Collapsing Columns

When `setColumnCollapsingAllowed(true)` is set, the right side of the table header shows a drop-down list that allows selecting which columns are shown. Collapsing columns is different than hiding columns with `setVisibleColumns()`, which hides the columns completely so that they can not be made visible (uncollapsed) from the user interface.

You can collapse columns programmatically with `setColumnCollapsed()`. Collapsing must be enabled before collapsing columns with the method or it will throw an **`IllegalAccessExcep-tion`**.

```
// Allow the user to collapse and uncollapse columns
table.setColumnCollapsingAllowed(true);

// Collapse this column programmatically
try {
    table.setColumnCollapsed("born", true);
} catch (IllegalAccessException e) {
    // Can't occur - collapsing was allowed above
    System.err.println("Something horrible occurred");
}

// Give enough width for the table to accommodate the
// initially collapsed column later
table.setWidth("250px");
```

See Figure 6.44, “Collapsing Columns”.

Figure 6.44. Collapsing Columns

Column Collapsing

NAME	DIED
Galileo	1642
Väisälä	1971
Valtaoja	

☒ Name
 ☒ Died
 ☐ Born

If the table has undefined width, it minimizes its width to fit the width of the visible columns. If some columns are initially collapsed, the width of the table may not be enough to accomodate them later, which will result in an ugly horizontal scrollbar. You should consider giving the table enough width to accomodate columns uncollapsed by the user.

Components Inside a Table

The cells of a **Table** can contain any user interface components, not just strings. If the rows are higher than the row height defined in the default theme, you have to define the proper row height in a custom theme.

When handling events for components inside a **Table**, such as for the **Button** in the example below, you usually need to know the item the component belongs to. Components do not themselves know about the table or the specific item in which a component is contained. Therefore, the handling method must use some other means for finding out the Item ID of the item. There are a few possibilities. Usually the easiest way is to use the `setData()` method to attach an arbitrary object to a component. You can subclass the component and include the identity information there. You can also simply search the entire table for the item with the component, although that solution may not be so scalable.

The example below includes table rows with a **Label** in HTML content mode, a multiline **TextField**, a **CheckBox**, and a **Button** that shows as a link.

```
// Create a table and add a style to allow setting the row height in theme.
final Table table = new Table();
table.addStyleName("components-inside");

/* Define the names and data types of columns.
 * The "default value" parameter is meaningless here. */
table.addContainerProperty("Sum", Label.class, null);
table.addContainerProperty("Is Transferred", CheckBox.class, null);
table.addContainerProperty("Comments", TextField.class, null);
table.addContainerProperty("Details", Button.class, null);

/* Add a few items in the table. */
for (int i=0; i<100; i++) {
    // Create the fields for the current table row
    Label sumField = new Label(String.format(
        "Sum is <b>${%04.2f}</b><br/><i>(VAT incl.)</i>",
        new Object[] {new Double(Math.random()*1000)}),
        ContentMode.HTML);
    CheckBox transferredField = new CheckBox("is transferred");

    // Multiline text field. This required modifying the
    // height of the table row.
    TextField commentsField = new TextField();
    commentsField.setRows(3);

    // The Table item identifier for the row.
    Integer itemId = new Integer(i);

    // Create a button and handle its click. A Button does not
    // know the item it is contained in, so we have to store the
    // item ID as user-defined data.
    Button detailsField = new Button("show details");
    detailsField.setData(itemId);
    detailsField.addClickListener(new Button.ClickListener() {
        public void buttonClick(ClickEvent event) {
            // Get the item identifier from the user-defined data.
            Integer iid = (Integer)event.getButton().getData();
            Notification.show("Link " +
                iid.intValue() + " clicked.");
        }
    });
    detailsField.addStyleName("link");

    // Create the table row.
    table.addItem(new Object[] {sumField, transferredField,
        commentsField, detailsField},
        itemId);
}

// Show just three rows because they are so high.
table.setPageLength(3);
```

The row height has to be set higher than the default with a style rule such as the following:

```
/* Table rows contain three-row TextField components. */
.v-table-components-inside .v-table-cell-content {
```

```
height: 54px;
}
```

The table will look as shown in Figure 6.45, “Components in a Table”.

Figure 6.45. Components in a Table

Sum	Is Transferred	Comments	Details
Sum is \$777,60 (VAT incl.)	☒ is transferred	We sent this money already in last week.	show details
Sum is \$500,40 (VAT incl.)	☐ is transferred		show details
Sum is \$836,10 (VAT incl.)	☐ is transferred		show details

Iterating Over a Table

As the items in a **Table** are not indexed, iterating over the items has to be done using an iterator. The `getItemIds()` method of the **Container** interface of **Table** returns a **Collection** of item identifiers over which you can iterate using an **Iterator**. For an example about iterating over a **Table**, please see Section 10.5, “Collecting Items in Containers”. Notice that you may not modify the **Table** during iteration, that is, add or remove items. Changing the data is allowed.

Filtering Table Contents

A table can be filtered if its container data source implements the **Filterable** interface, as the default **IndexedContainer** does. See Section 10.5.7, “**Filterable** Containers”.

6.21.3. Editing the Values in a Table

Normally, a **Table** simply displays the items and their fields as text. If you want to allow the user to edit the values, you can either put them inside components as we did earlier or simply call `setEditable(true)`, in which case the cells are automatically turned into editable fields.

Let us begin with a regular table with a some columns with usual Java types, namely a **Date**, **Boolean**, and a **String**.

```
// Create a table. It is by default not editable.
Table table = new Table();

// Define the names and data types of columns.
table.addContainerProperty("Date",    Date.class, null);
table.addContainerProperty("Work",    Boolean.class, null);
table.addContainerProperty("Comments", String.class, null);

...
```

You could put the table in editable mode right away. We continue the example by adding a check box to switch the table between normal and editable modes:

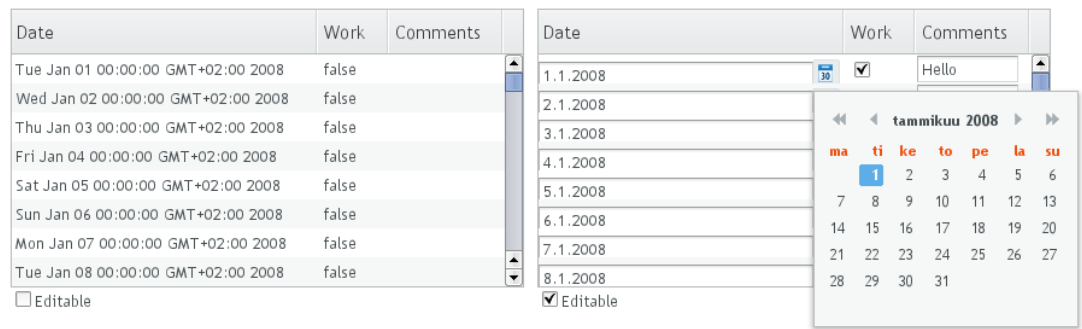
```

CheckBox editable = new CheckBox("Editable", true);
editable.addValueChangeListener(valueChange -> // Java 8
    table.setEditable((Boolean) editable.getValue()));

```

Now, when you check to checkbox, the components in the table turn into editable fields, as shown in Figure 6.46, “A Table in Normal and Editable Mode”.

Figure 6.46. A Table in Normal and Editable Mode



Field Factories

The field components that allow editing the values of particular types in a table are defined in a field factory that implements the **TableFieldFactory** interface. The default implementation is **DefaultFieldFactory**, which offers the following crude mappings:

Table 6.2. Type to Field Mappings in DefaultFieldFactory

Property Type	Mapped to Field Class
Date	<code>A[classname]DateField.</code>
Boolean	<code>A[classname]CheckBox.</code>
Item	<code>A[classname]Form</code> (deprecated in Vaadin 7). The fields of the form are automatically created from the item's properties using <code>a[classname]FormFieldFactory</code> . The normal use for this property type is inside <code>a[classname]Form</code> and is less useful inside <code>a[classname]#Table</code> .
<i>other</i>	<code>A[classname]TextField.</code> The text field manages conversions from the basic types, if possible.

Field factories are covered with more detail in Section 10.4, “Creating Forms by Binding Fields to Items”. You could just implement the **TableFieldFactory** interface, but we recommend that you extend the **DefaultFieldFactory** according to your needs. In the default implementation, the mappings are defined in the `createFieldByPropertyType()` method (you might want to look at the source code) both for tables and forms.

6.21.4. Column Headers and Footers

Table supports both column headers and footers; the headers are enabled by default.

Headers

The table header displays the column headers at the top of the table. You can use the column headers to reorder or resize the columns, as described earlier. By default, the header of a column is the property ID of the column, unless given explicitly with `setColumnHeader()`.

```
// Define the properties
table.addContainerProperty("lastname", String.class, null);
table.addContainerProperty("born", Integer.class, null);
table.addContainerProperty("died", Integer.class, null);

// Set nicer header names
table.setColumnHeader("lastname", "Name");
table.setColumnHeader("born", "Born");
table.setColumnHeader("died", "Died");
```

The text of the column headers and the visibility of the header depends on the *column header mode*. The header is visible by default, but you can disable it with `setColumnHeaderMode(Table.COLUMN_HEADER_MODE_HIDDEN)`.

Footers

The table footer can be useful for displaying sums or averages of values in a column, and so on. The footer is not visible by default; you can enable it with `setFooterVisible(true)`. Unlike in the header, the column headers are empty by default. You can set their value with `setColumnFooter()`. The columns are identified by their property ID.

The following example shows how to calculate average of the values in a column:

```
// Have a table with a numeric column
Table table = new Table("Custom Table Footer");
table.addContainerProperty("Name", String.class, null);
table.addContainerProperty("Died At Age", Integer.class, null);

// Insert some data
Object people[][] = { {"Galileo", 77},
                      {"Monnier", 83},
                      {"Vaisala", 79},
                      {"Oterma", 86}};
for (int i=0; i<people.length; i++)
    table.addItem(people[i], new Integer(i));

// Calculate the average of the numeric column
double avgAge = 0;
for (int i=0; i<people.length; i++)
    avgAge += (Integer) people[i][1];
avgAge /= people.length;

// Set the footers
table.setFooterVisible(true);
table.setColumnFooter("Name", "Average");
table.setColumnFooter("Died At Age", String.valueOf(avgAge));

// Adjust the table height a bit
table.setPageLength(table.size());
```

The resulting table is shown in Figure 6.47, “A Table with a Footer”.

Figure 6.47. A Table with a Footer

Custom Table Footer	
Name	Died At Age
Galileo	77
Monnier	83
Väisälä	79
Oterma	86
Average	81.25

Handling Mouse Clicks on Headers and Footers

Normally, when the user clicks a column header, the table will be sorted by the column, assuming that the data source is **Sortable** and sorting is not disabled. In some cases, you might want some other functionality when the user clicks the column header, such as selecting the column in some way.

Clicks in the header cause a **HeaderClickEvent**, which you can handle with a **Table.HeaderClickListener**. Click events on the table header (and footer) are, like button clicks, sent immediately to server, so there is no need to set `setImmediate()`.

```
// Handle the header clicks
table.addHeaderClickListener(new Table.HeaderClickListener() {
    public void headerClick(HeaderClickEvent event) {
        String column = (String) event.getPropertyId();
        Notification.show("Clicked " + column +
            "with " + event.getButtonName());
    }
});

// Disable the default sorting behavior
table.setSortDisabled(true);
```

Setting a click handler does not automatically disable the sorting behavior of the header; you need to disable it explicitly with `setSortDisabled(true)`. Header click events are not sent when the user clicks the column resize handlers to drag them.

The **HeaderClickEvent** object provides the identity of the clicked column with `getPropertyId()`. The `getButton()` reports the mouse button with which the click was made: `BUTTON_LEFT`, `BUTTON_RIGHT`, or `BUTTON_MIDDLE`. The `getButtonName()` a human-readable button name in English: "left", "right", or "middle". The `isShiftKey()`, `isCtrlKey()`, etc., methods indicate if the **Shift**, **Ctrl**, **Alt** or other modifier keys were pressed during the click.

Clicks in the footer cause a **FooterClickEvent**, which you can handle with a **Table.FooterClickListener**. Footers do not have any default click behavior, like the sorting in the header. Otherwise, handling clicks in the footer is equivalent to handling clicks in the header.

6.21.5. Generated Table Columns

A table can have generated columns which values can be calculated based on the values in other columns. The columns are generated with a class implementing the `Table.ColumnGenerator` interface.

The **GeneratedPropertyContainer** described in Section 10.5.6, "**GeneratedPropertyContainer**" is another way to accomplish the same task at container level. In addition to generating values, you can also use the feature for formatting or styling columns.

6.21.6. Formatting Table Columns

The displayed values of properties shown in a table are normally formatted using the `toString()` method of each property. Customizing the format in a table column can be done in several ways:

- Using **ColumnGenerator** to generate a second column that is formatted. The original column needs to be set invisible. See Section 6.21.5, "Generated Table Columns".
- Using a **Converter** to convert between the property data model and its representation in the table.
- Using a **GeneratedPropertyContainer** as a wrapper around the actual container to provide formatting.
- Overriding the default `formatPropertyValue()` in **Table**.

As using a **PropertyFormatter** is generally much more awkward than overriding the `formatPropertyValue()`, its use is not described here.

You can override `formatPropertyValue()` as is done in the following example:

```
// Create a table that overrides the default
// property (column) format
final Table table = new Table("Formatted Table") {
    @Override
    protected String formatPropertyValue(Object rowId,
        Object colId, Property property) {
        // Format by property type
        if (property.getType() == Date.class) {
            SimpleDateFormat df =
                new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
            return df.format((Date)property.getValue());
        }
    }
}
```

```

    }

    return super.formatPropertyValue(rowId, colId, property);
}
};

// The table has some columns
table.addContainerProperty("Time", Date.class, null);

... Fill the table with data ...

```

You can also distinguish between columns by the *colId* parameter, which is the property ID of the column. **DecimalFormat** is useful for formatting decimal values.

```

... in formatPropertyValue() ...
} else if ("Value".equals(pid)) {
    // Format a decimal value for a specific locale
    DecimalFormat df = new DecimalFormat("#.00",
        new DecimalFormatSymbols(locale));
    return df.format((Double) property.getValue());
}
...
table.addContainerProperty("Value", Double.class, null);

```

A table with the formatted date and decimal value columns is shown in Figure 6.48, “Formatted Table Columns”.

Figure 6.48. Formatted Table Columns

Formatted Table

TIME	VALUE	MESSAGE	
1970-01-01 02:00:00	708,42	Msg #55	▲
1970-03-29 05:16:33	44,31	Msg #33	
1970-07-22 04:40:13	741,61	Msg #1	
1970-09-01 12:11:49	757,91	Msg #36	
1970-12-21 02:32:50	793,82	Msg #92	
1971-01-13 05:25:15	700,79	Msg #65	▼

You can use CSS for further styling of table rows, columns, and individual cells by using a **CellStyleGenerator**. It is described in Section 6.21.7, “CSS Style Rules”.

6.21.7. CSS Style Rules

Styling the overall style of a **Table** can be done with the following CSS rules.

```

.v-table {}
.v-table-header-wrap {}
.v-table-header {}
.v-table-header-cell {}
.v-table-resizer {} /* Column resizer handle. */
.v-table-caption-container {}
.v-table-body {}
.v-table-row-spacer {}
.v-table-table {}
.v-table-row {}
.v-table-cell-content {}

```

Notice that some of the widths and heights in a table are calculated dynamically and can not be set in CSS.

6.22. Tree

The **Tree** component allows a natural way to represent data that has hierarchical relationships, such as filesystems or message threads. The **Tree** component in Vaadin works much like the tree components of most modern desktop user interface toolkits, for example in directory browsing.

The typical use of the **Tree** component is for displaying a hierarchical menu, like a menu on the left side of the screen, as in Figure 6.49, “A **Tree** Component as a Menu”, or for displaying filesystems or other hierarchical datasets. The *menu* style makes the appearance of the tree more suitable for this purpose.

```
final Object[][] planets = new Object[][]{
    new Object[]{"Mercury"},
    new Object[]{"Venus"},
    new Object[]{"Earth", "The Moon"},
    new Object[]{"Mars", "Phobos", "Deimos"},
    new Object[]{"Jupiter", "Io", "Europa", "Ganymedes",
        "Callisto"},
    new Object[]{"Saturn", "Titan", "Tethys", "Dione",
        "Rhea", "Iapetus"},
    new Object[]{"Uranus", "Miranda", "Ariel", "Umbriel",
        "Titania", "Oberon"},
    new Object[]{"Neptune", "Triton", "Proteus", "Nereid",
        "Larissa"}};

Tree tree = new Tree("The Planets and Major Moons");

/* Add planets as root items in the tree. */
for (int i=0; i<planets.length; i++) {
    String planet = (String) (planets[i][0]);
    tree.addItem(planet);

    if (planets[i].length == 1) {
        // The planet has no moons so make it a leaf.
        tree.setChildrenAllowed(planet, false);
    } else {
        // Add children (moons) under the planets.
        for (int j=1; j<planets[i].length; j++) {
            String moon = (String) planets[i][j];

            // Add the item as a regular item.
            tree.addItem(moon);

            // Set it to be a child.
            tree.setParent(moon, planet);

            // Make the moons look like leaves.
            tree.setChildrenAllowed(moon, false);
        }

        // Expand the subtree.
        tree.expandItemsRecursively(planet);
    }
}
```



```

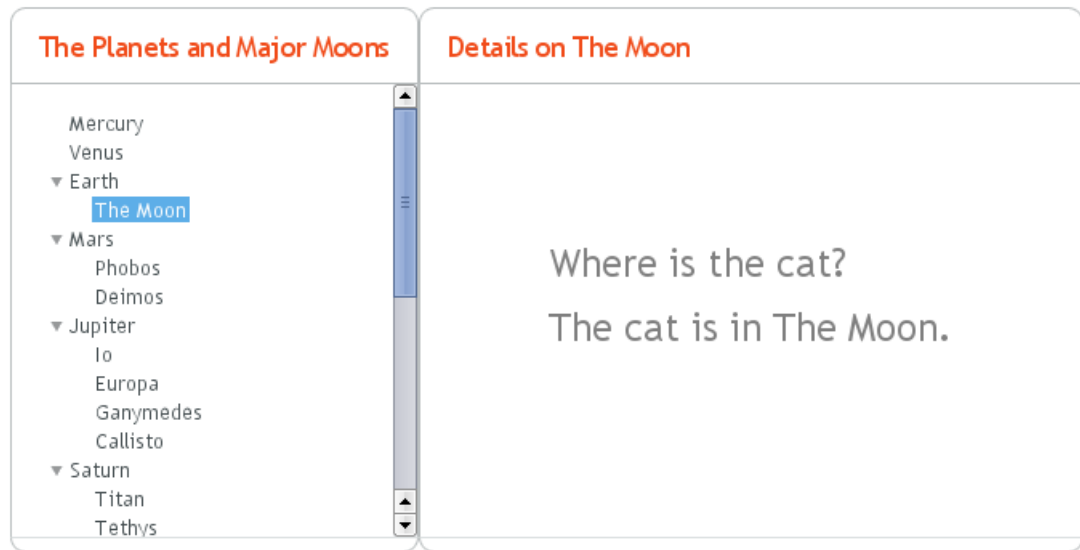
}

main.addComponent(tree);

```

Figure 6.49, “A **Tree** Component as a Menu” below shows the tree from the code example in a practical situation.

Figure 6.49. A Tree Component as a Menu



You can read or set the currently selected item by the `value` property of the **Tree** component, that is, with `getValue()` and `setValue()`. When the user clicks an item on a tree, the tree will receive an **ValueChangeEvent**, which you can catch with a **ValueChangeListener**. To receive the event immediately after the click, you need to set the tree as **setImmediate(true)**.

The **Tree** component uses **Container** data sources much like the **Table** component, with the addition that it also utilizes hierarchy information maintained by a **HierarchicalContainer**. The contained items can be of any item type supported by the container. The default container and its `addItem()` assume that the items are strings and the string value is used as the item ID.

6.23. TreeTable

TreeTable is an extension of the **Table** component with support for a tree-like hierarchy in the first column. As with **Tree**, the hierarchy is determined by the parent-children relationships defined in the `Container.Hierarchical` interface. The default container is **HierarchicalContainer**, but you can bind **TreeTable** to any container implementing the interface.

Figure 6.50. TreeTable Component**My TreeTable**

Name	Number
▼ Menu	
▼ Beverages	
Coffee	23
Tea	42
▼ Foods	
Bread	13
Cake	11

As with **Tree**, you can define the parent-child relationships with `setParent()`, as is shown in the following example with numeric item IDs:

```
TreeTable ttable = new TreeTable("My TreeTable");
ttable.addContainerProperty("Name", String.class, null);
ttable.addContainerProperty("Number", Integer.class, null);

// Create the tree nodes and set the hierarchy
ttable.addItem(new Object[]{"Menu", null}, 0);
ttable.addItem(new Object[]{"Beverages", null}, 1);
ttable.setParent(1, 0);
ttable.addItem(new Object[]{"Foods", null}, 2);
ttable.setParent(2, 0);
ttable.addItem(new Object[]{"Coffee", 23}, 3);
ttable.addItem(new Object[]{"Tea", 42}, 4);
ttable.setParent(3, 1);
ttable.setParent(4, 1);
```

```
ttable.addItem(new Object[]{"Bread", 13}, 5);
ttable.addItem(new Object[]{"Cake", 11}, 6);
ttable.setParent(5, 2);
ttable.setParent(6, 2);
```

Some container types may allow defining hierarchies if the container data itself, without explicitly setting the parent-child relationships with `setParent()`.

Unlike **Tree**, a **TreeTable** can have components in the hierarchical column, both when the property type is a component type and when the tree table is in editable mode.

For other features, we refer you to documentation for **Table**, as given in Section 6.21, “**Table**”.

6.23.1. Expanding and Collapsing Items

As in **Tree**, you can set the expanded/collapsed state of an item programmatically with `setCollapsed()`. Note that if you want to expand all items, there is no `expandItemsRecursively()` like in **Tree**. Moreover, the `getItemIds()` only returns the IDs of the currently visible items for ordinary **Hierarchical** (not **Collapsible**) containers. Hence you can not use that to iterate over all the items, but you need to get the IDs from the underlying container.

```
// Expand the tree
for (Object itemId: ttable.getContainerDataSource()
    .getItemIds()) {
    ttable.setCollapsed(itemId, false);

    // As we're at it, also disallow children from
    // the current leaves
    if (! ttable.hasChildren(itemId))
        ttable.setChildrenAllowed(itemId, false);
}
```

In large tables, this explicit setting becomes infeasible, as it needs to be stored in the **TreeTable** (more exactly, in the **HierarchicalStrategy** object) for all the contained items. You can use a **Collapsible** container to store the collapsed states in the container, thereby avoiding the explicit settings and memory overhead. There are no built-in collapsible containers in the Vaadin core framework, so you either need to use an add-on container or implement it yourself.

6.24. Grid

Grid is many things, and perhaps the most versatile and powerful component in Vaadin. Like **Table**, it allows presenting and editing tabular data, but escapes many of **Table**'s limitations. Efficient lazy loading of data while scrolling greatly improves performance. Grid is scalable, mobile friendly, and extensible.

6.24.1. Overview

Grid is for displaying and editing tabular data laid out in rows and columns. At the top, a *header* can be shown, and a *footer* at the bottom. In addition to plain text, the header and footer can contain HTML and components. Having components in the header allows implementing filtering easily. The grid data can be sorted by clicking on a column header; shift-clicking a column header enables secondary sorting criteria.

Figure 6.51. A Grid Component

	Names		Years		
	First Name ^ 2	Last Name ^ 1	Born In	Died In	Lived For
	Filter	Filter	Filter	Filter	Filter
☐	Tycho	Brahe	1546 AD	1601 AD	55 years
☐	Giordano	Bruno	1548 AD	1600 AD	52 years
☐	Nicolaus	Copernicus	1473 AD	1543 AD	70 years
☐	Galileo	Galilei	1564 AD	1642 AD	78 years
☐	Christiaan	Huygens	1629 AD	1695 AD	66 years
			1555,17	1618,50	63,33

The data area can be scrolled both vertically and horizontally. The leftmost columns can be frozen, so that they are never scrolled out of the view. The data is loaded lazily from the server, so that only the visible data is loaded. The smart lazy loading functionality gives excellent user experience even with low bandwidth, such as mobile devices.

The grid data can be edited with a row-based editor after double-clicking a row. The fields are generated with a field factory, or set explicitly, and bound to data with a field group.

Grid is fully themeable with CSS and style names can be set for all grid elements. For data rows and cells, the styles can be generated with a row or cell style generator.

Finally, **Grid** is designed to be extensible and used just as well for client-side development - its GWT API is nearly identical to the server-side API, including data binding.

Differences to Table

In addition to core features listed above, **Grid** has the following API-level and functional differences to Table:

- Grid is not a `Container` itself, even though it can be bound to a container data source. Consequently, columns are defined differently, and so forth.
- Rows can be added with `addRow()` shorthand (during initialization) instead of `addItem()`.
- Use `setHeightByRows()` and `setHeightMode()` instead of `setPageLength()` to set the height in number of rows.
- Grid does not extend **AbstractSelect** and is not a field, but has its own selection API. `addSelectionListener()` is called to define a `SelectionListener`. The listener also receives a collection of deselected items.

- Grid does not support having all cells in editable mode, it only supports row-based editing, with a row mini-editor that allows saving or discarding the changes.
- Grid has no generated columns. Instead, the container data source can be wrapped around a **GeneratedPropertyContainer**.
- No column icons; you can implement them in a column with an **ImageRenderer**.
- Components can not be shown in Grid cells; instead the much more efficient renderers can be used for the most common cases, and row editor for editing values.
- Limited support for drag and drop: the user can drag columns to reorder them.

In addition, Grid has the following visual changes:

- Multiple selection is indicated with check boxes in addition to highlighting.
- Grid does not show the row loading indicator like Table does.

6.24.2. Binding to Data

Grid is normally used by binding it to a container data source, described in Section 10.5, “Collecting Items in Containers”. The container must implement `Container.Indexed` interface. By default, it is bound to an **IndexedContainer**; Grid offers some shorthand methods to operate on the default container, as described later.

You can set the container in the constructor or with `setContainerDataSource()`.

For example, if you have a collection of beans, you could wrap them in a Vaadin **BeanContainer** or **BeanItemContainer**, and bind to a **Grid** as follows

```
// Have some data
Collection<Person> people = Lists.newArrayList(
    new Person("Nicolaus Copernicus", 1543),
    new Person("Galileo Galilei", 1564),
    new Person("Johannes Kepler", 1571));

// Have a container of some type to contain the data
BeanItemContainer<Person> container =
    new BeanItemContainer<Person>(Person.class, people);

// Create a grid bound to the container
Grid grid = new Grid(container);
grid.setColumnOrder("name", "born");
layout.addComponent(grid);
```

Note that you need to override `equals()` and `hashCode()` for the bean (**Person**) class to make the **BeanItemContainer** work properly.

Default Data Source and Shorthands

Sometimes, when you have just a few fixed items that you want to display, you can define the grid columns and add data rows manually. **Grid** is by default bound to a **IndexedContainer**. You can define new columns (container properties) with `addColumn()` and then add rows (items) with `addRow()`. The types in the row data must match the defined column types.

For example:

```
// Create a grid
Grid grid = new Grid();

// Define some columns
grid.addColumn("name", String.class);
grid.addColumn("born", Integer.class);

// Add some data rows
grid.addRow("Nicolaus Copernicus", 1543);
grid.addRow("Galileo Galilei", 1564);
grid.addRow("Johannes Kepler", 1571);

layout.addComponent(grid);
```

Or, if you have the data in an array:

```
// Have some data
Object[][] people = { {"Nicolaus Copernicus", 1543},
                      {"Galileo Galilei", 1564},
                      {"Johannes Kepler", 1571}};

for (Object[] person: people)
    grid.addRow(person);
```

Note that you can not use `addRow()` to add items if the container is read-only or has read-only columns, such as generated columns.

6.24.3. Handling Selection Changes

Selection in **Grid** is handled a bit differently from other selection components, as it is not an **AbstractSelect**. Grid supports both single and multiple selection, defined by the *selection mode*. Selection events can be handled with a `SelectionListener`.

Selection Mode

A **Grid** can be set to be in `SINGLE` (default), `MULTI`, or `NONE` selection mode, defined in the **Grid.SelectionMode** enum.

```
// Use single-selection mode (default)
grid.setSelectionMode(SelectionMode.SINGLE);
```

Empty (null) selection is allowed by default, but can be disabled with `setDeselectAllowed()` in single-selection mode.

The selection is handled with a different selection model object in each respective selection mode: **SingleSelectionModel**, **MultiSelectionModel**, and **NoSelectionModel** (in which selection is always empty).

```
// Pre-select an item
SingleSelectionModel selection =
    (SingleSelectionModel) grid.getSelectionModel();
selection.select( // Select 3rd item
    grid.getContainerDataSource().getIdByIndex(2));
```

Handling Selection

Changes in the selection can be handled with a `SelectionListener`. You need to implement the `select()` method, which gets a **SelectionEvent** as parameter. In addition to selection, you can handle clicks on rows or cells with a `ItemClickListener`.

You can get the new selection from the selection event with `getSelected()`, which returns a set of item IDs, or more simply from the grid or the selection model with `getSelectedRow()`, which returns the single selected item ID.

For example:

```
grid.addSelectionListener(selectionEvent -> { // Java 8
    // Get selection from the selection model
    Object selected = ((SingleSelectionModel)
        grid.getSelectionModel()).getSelectedRow();

    if (selected != null)
        Notification.show("Selected " +
            grid.getContainerDataSource().getItem(selected)
                .getItemProperty("name"));
    else
        Notification.show("Nothing selected");
});
```

The current selection can be obtained from the **Grid** object by `getSelectedRow()` or `getSelectedRows()`, which return one (in single-selection mode) or all (in multi-selection mode) selected items.



Warning

Note that changes to the item set of the container data source are not automatically reflected in the selection model and may cause the selection model to refer to stale item IDs. This always occurs, for example, when you delete the selected item or items. So, if you modify the item set of the container, you should synchronize or reset the selection with the container, such as by calling `reset()` on the selection model.

Multiple Selection

In the multiple selection mode, a user can select multiple items by clicking on the checkboxes in the leftmost column, or by using the **Space** to select/deselect the currently focused row. Space bar is the default key for toggling the selection, but it can be customized.

Figure 6.52. Multiple Selection in Grid

☐	Name	City	Year
☐	Charles Newton	Oxford	1 877
☒	Ada Lovelace	Innsbruck	1 815
☐	Charles Lovelace	London	1 878
☒	Ada Darwin	Innsbruck	1 861
☒	Charles Newton	Oxford	1 982
☐	Charles Adams	Innsbruck	1 940
☐	Ada Adams	Innsbruck	1 976

Delete Selected

The selection is managed through the **MultiSelectionMode** class. The currently selected rows can be set with `setSelected()` by a collection of item IDs, or you can use `select()` to add items to the selection.

```
// Grid in multi-selection mode
Grid grid = new Grid(exampleDataSource());
grid.setSelectionMode(SelectionMode.MULTI);

// Pre-select some items
MultiSelectionMode selection =
    (MultiSelectionMode) grid.getSelectionModel();
selection.setSelected( // Items 2-4
    grid.getContainerDataSource().getItemIds(2, 3));
```

The current selection can be read with `getSelectedRows()`; either in the **MultiSelectionMode** object or in the **Grid**.

```
// Allow deleting the selected items
Button delSelected = new Button("Delete Selected", e -> {
    // Delete all selected data items
    for (Object itemId: selection.getSelectedRows())
        grid.getContainerDataSource().removeItem(itemId);

    // Otherwise out of sync with container
    grid.getSelectionModel().reset();

    // Disable after deleting
    e.getButton().setEnabled(false);
});
```



```
});  
delSelected.setEnabled(grid.getSelectedRows().size() > 0);
```

Changes in the selection can be handled with a `SelectionListener`. The selection event object provides `getAdded()` and `getRemoved()` to allow determining the differences in the selection change.

```
// Handle selection changes  
grid.addSelectionListener(selection -> { // Java 8  
    Notification.show(selection.getAdded().size() +  
        " items added, " +  
        selection.getRemoved().size() +  
        " removed.");  
  
    // Allow deleting only if there's any selected  
    deleteSelected.setEnabled(  
        grid.getSelectedRows().size() > 0);  
});
```

Focus and Clicks

In addition to selecting rows, you can focus individual cells. The focus can be moved with arrow keys and, if editing is enabled, pressing **Enter** opens the editor. Normally, pressing **Tab** or **Shift+Tab** moves the focus to another component, as usual.

When editing or in unbuffered mode, **Tab** or **Shift+Tab** moves the focus to the next or previous cell. The focus moves from the last cell of a row forward to the beginning of the next row, and likewise, from the first cell backward to the end of the previous row. Note that you can extend **DefaultEditorEventHandler** to change this behavior.

With the mouse, you can focus a cell by clicking on it. The clicks can be handled with an `ItemClickListener`. The **ItemClickEvent** object contains various information, most importantly the ID of the clicked row and column.

```
grid.addItemClickListener(event -> // Java 8  
    Notification.show("Value: " +  
        container.getContainerProperty(event.getItemId(),  
            event.getPropertyId()).getValue().toString()));
```

The clicked grid cell is also automatically focused.

The focus indication is themed so that the focused cell has a visible focus indicator style by default, while the row doesn't. You can enable row focus, as well as disable cell focus, in a custom theme. See Section 6.24.13, "Styling with CSS".

6.24.4. Configuring Columns

Columns are normally defined in the container data source. The `addColumn()` method can be used to add columns to a container that supports it, such as the default **IndexedContainer**.

Column configuration is defined in **Grid.Column** objects, which can be obtained from the grid with `getColumn()` by the column (property) ID.

```
Grid.Column bornColumn = grid.getColumn("born");  
bornColumn.setHeaderCaption("Born");
```

In the following, we describe the basic column configuration.

Column Order

You can set the order of columns with `setColumnOrder()` for the grid. Columns that are not given for the method are placed after the specified columns in their natural order.

```
grid.setColumnOrder("firstname", "lastname", "born",  
                    "birthplace", "died");
```

Note that the method can not be used to hide columns. You can hide columns with the `removeColumn()`, as described later, or by hiding them in a **GeneratedPropertyContainer**.

Hiding Columns

Columns can be hidden by removing them with `removeColumn()`. You can remove all columns with `removeAllColumns()`. The removed columns are only removed from the grid, not from the container data source.

To restore a previously removed column, you can call `addColumn()` with the property ID. Instead of actually adding another column to the data source, it merely restores the previously removed one. However, column settings such as header or editor are not restored, but must be redone.

You can also hide columns at container-level. At least **GeneratedPropertyContainer** allows doing so, as described in Section 10.5.6, “**GeneratedPropertyContainer**”.

Column Captions

Column captions are displayed in the grid header. The default captions are generated automatically from the property ID. You can set the header caption explicitly through the column object with `setHeaderCaption()`.

```
Grid.Column bornColumn = grid.getColumn("born");  
bornColumn.setHeaderCaption("Born");
```

This is equivalent to setting it with `setText()` for the header cell; the **HeaderCell** also allows setting the caption in HTML or as a component, as well as styling it, as described later in Section 6.24.7, “Header and Footer”.

Column Widths

Columns have by default undefined width, which causes automatic sizing based on the widths of the displayed data. You can set column widths explicitly by pixel value with `setWidth()`, or relatively using expand ratios with `setExpandRatio()`.

When using expand ratios, the columns with a non-zero expand ratio use the extra space remaining from other columns, in proportion to the defined ratios.

You can specify minimum and maximum widths for the expanding columns with `setMinimumWidth()` and `setMaximumWidth()`, respectively.

The user can resize columns by dragging their separators with the mouse. When resized manually, all the columns widths are set to explicit pixel values, even if they had relative values before.

Frozen Columns

You can set the number of columns to be frozen with `setFrozenColumnCount()`, so that they are not scrolled off when scrolling horizontally.

```
grid.setFrozenColumnCount(2);
```

Setting the count to `0` disables frozen data columns; setting it to `-1` also disables the selection column in multi-selection mode.

6.24.5. Generating Columns

Columns with values computed from other columns or in some other way can be generated with a container or data model that generates the property values. The **GeneratedPropertyContainer** can be used for this purpose. It wraps around any indexed container to extend its properties with read-only generated properties. The generated properties can have same IDs as the original ones, thereby replacing them with formatted or converted values. See Section 10.5.6, “**GeneratedPropertyContainer**” for a detailed description of using it.

Generated columns are read-only, so you can not add grid rows with `addRow()`. In editable mode, editor fields are not generated for generated columns.

Note that, while **GeneratedPropertyContainer** implements `Container.Sortable`, the wrapped container might not, and also sorting on the generated properties requires special handling. In such cases, generated properties or the entire container might not actually be sortable.

6.24.6. Column Renderers

A *renderer* is a feature that draws the client-side representation of a data value. This allows having images, HTML, and buttons in grid cells.

Figure 6.53. Column Renderers: Image, Date, HTML, and Button

Picture	Name	Born	Link	Button
	Nicolaus Copernicus	March 19, 1473	more info	<button>Delete</button>
	Galileo Galilei	March 15, 1564	more info	<button>Delete</button>
	Johannes Kepler	January 27, 1572	more info	<button>Delete</button>

Renderers implement the `Renderer` interface. You set the column renderer in the **Grid.Column** object as follows:

```
grid.addColumn("born", Integer.class);  
...  
Grid.Column bornColumn = grid.getColumn("born");  
bornColumn.setRenderer(new NumberRenderer("born in %d AD"));
```

Renderers require a specific data type for the column. To convert to a property type to a type required by a renderer, you can pass an optional `Converter` to `setRenderer()`, as described later in this section. A converter can also be used to (pre)format the property values. The converter is run on the server-side, before sending the values to the client-side to be rendered with the renderer.

The following renderers are available, as defined in the server-side `com.vaadin.ui.renderers` package:

ButtonRenderer

Renders the data value as the caption of a button. A `RendererClickListener` can be given to handle the button clicks.

ImageRenderer

Renders the cell as an image. The column type must be a `Resource`, as described in Section 5.5, “Images and Other Resources”; only **ThemeResource** and **ExternalResource** are currently supported for images in **Grid**.

DateRenderer

Formats a column with a **Date** type using string formatter. The format string is same as for `String.format()` in Java API. The date is passed in the parameter index 1, which can be omitted if there is only one format specifier, such as “%tF”.

HTMLRenderer

Renders the cell as HTML. This allows formatting cell content, as well as using HTML features such as hyperlinks.

NumberRenderer

Formats column values with a numeric type extending **Number**: **Integer**, **Double**, etc. The format can be specified either by the subclasses of `java.text.NumberFormat`, namely **DecimalFormat** and **ChoiceFormat**, or by `String.format()`.

ProgressBarRenderer

Renders a progress bar in a column with a **Double** type. The value must be between 0.0 and 1.0.

TextRenderer

Displays plain text as is. Any HTML markup is quoted.

Custom Renderers

Renderers are component extensions that require a client-side counterpart. See Section 16.4.1, “Renderers” for information on implementing custom renderers.

Converting for Rendering

Optionally, you can give a `Converter` in the `setRenderer()`, or define it for the column, to convert the data value to an intermediary representation that is rendered by the renderer. For

example, when using an **ImageRenderer**, you could store the image file name in the data column, which the converter would convert to a resource, which would then be rendered by the renderer.

In the following example, we use a converter and **HTMLRenderer** to display boolean values as **FontAwesome** icons

```
// Have a column for hyperlink paths to Wikipedia
grid.addColumn("truth", Boolean.class);
Grid.Column truth = grid.getColumn("truth");
truth.setRenderer(new HtmlRenderer(),
    new StringToBooleanConverter(
        FontAwesome.CHECK_CIRCLE_O.getHtml(),
        FontAwesome.CIRCLE_O.getHtml()));
...
```

In this example, we use a converter to format URL paths to complete HTML hyperlinks with **HTMLRenderer**:

```
// Have a column for hyperlink paths to Wikipedia
grid.addColumn("link", String.class);

Grid.Column linkColumn = grid.getColumn("link");
linkColumn.setRenderer(new HtmlRenderer(),
    new Converter<String,String>() {

        @Override
        public String convertToModel(String value,
            Class<? extends String> targetType, Locale locale)
            throws Converter.ConversionException {
            return "not implemented";
        }

        @Override
        public String convertToPresentation(String value,
            Class<? extends String> targetType, Locale locale)
            throws Converter.ConversionException {
            return "<a href='http://en.wikipedia.org/wiki/" +
                value + "' target='_blank'>more info</a>";
        }

        @Override
        public Class<String> getModelType() {
            return String.class;
        }

        @Override
        public Class<String> getPresentationType() {
            return String.class;
        }
    });

// Data with a hyperlink path in the third column
grid.addRow("Nicolaus Copernicus", 1473,
    "Nicolaus_Copernicus");
...
```

A **GeneratedPropertyContainer** could be used for much the same purpose.

6.24.7. Header and Footer

A grid by default has a header, which displays column names, and can have a footer. Both can have multiple rows and neighbouring header row cells can be joined to feature column groups.

Adding and Removing Header and Footer Rows

A new header row is added with `prependHeaderRow()`, which adds it at the top of the header, `appendHeaderRow()`, which adds it at the bottom of the header, or with `addHeaderRowAt()`, which inserts it at the specified 0-base index. All of the methods return a **HeaderRow** object, which you can use to work on the header further.

```
// Group headers by joining the cells
HeaderRow groupingHeader = grid.prependHeaderRow();
...

// Create a header row to hold column filters
HeaderRow filterRow = grid.appendHeaderRow();
...
```

Similarly, you can add footer rows with `appendFooterRow()`, `prependFooterRow()`, and `addFooterRowAt()`.

Joining Header and Footer Cells

You can join two or more header or footer cells with the `join()` method. For header cells, the intention is usually to create column grouping, while for footer cells, you typically calculate sums or averages.

```
// Group headers by joining the cells
HeaderRow groupingHeader = grid.prependHeaderRow();
HeaderCell namesCell = groupingHeader.join(
    groupingHeader.getCell("firstname"),
    groupingHeader.getCell("lastname")).setText("Person");
HeaderCell yearsCell = groupingHeader.join(
    groupingHeader.getCell("born"),
    groupingHeader.getCell("died"),
    groupingHeader.getCell("lived")).setText("Dates of Life");
```

Text and HTML Content

You can set the header caption with `setText()`, in which case any HTML formatting characters are quoted to ensure security.

```
HeaderRow mainHeader = grid.getDefaultHeaderRow();
mainHeader.getCell("firstname").setText("First Name");
mainHeader.getCell("lastname").setText("Last Name");
mainHeader.getCell("born").setText("Born In");
mainHeader.getCell("died").setText("Died In");
mainHeader.getCell("lived").setText("Lived For");
```

To use raw HTML in the captions, you can use `setHtml()`.

```
namesCell.setHtml("<b>Names</b>");
yearsCell.setHtml("<b>Years</b>");
```

Components in Header or Footer

You can set a component in a header or footer cell with `setComponent()`. Often, this feature is used to allow filtering, as described in Section 6.24.8, “Filtering”, which also gives an example of the use.

6.24.8. Filtering

The ability to include components in the grid header can be used to create filters for the grid data. Filtering is done in the container data source, so the container must be of type that implements `Container.Filterable`.

Figure 6.54. Filtering Grid

Name	City	Year
Ada	Oxford	
Douglas Adams	Oxford	1 845
Ada Adams	Oxford	1 989
Ada Lovelace	Oxford	1 918
Ada Adams	Oxford	1 970
Charles Adams	Oxford	1 864
Ada Adams	Oxford	1 850

The filtering illustrated in Figure 6.54, “Filtering Grid” can be created as follows:

```
// Have a filterable container
IndexedContainer container = exampleDataSource();

// Create a grid bound to it
Grid grid = new Grid(container);
grid.setSelectionMode(SelectionMode.NONE);
grid.setWidth("500px");
grid.setHeight("300px");

// Create a header row to hold column filters
HeaderRow filterRow = grid.appendHeaderRow();

// Set up a filter for all columns
for (Object pid: grid.getContainerDataSource()
    .getContainerPropertyIds()) {
    HeaderCell cell = filterRow.getCell(pid);

    // Have an input field to use for filter
    TextField filterField = new TextField();
```

```

filterField.setColumns(8);

// Update filter When the filter input is changed
filterField.addChangeListener(change -> {
    // Can't modify filters so need to replace
    container.removeContainerFilters(pid);

    // (Re)create the filter if necessary
    if (! change.getText().isEmpty())
        container.addContainerFilter(
            new SimpleStringFilter(pid,
                change.getText(), true, false));
});
cell.setComponent(filterField);
}

```

6.24.9. Sorting

A user can sort the data in a grid on a column by clicking the column header. Clicking another time on the current sort column reverses the sort direction. Clicking on other column headers while keeping the Shift key pressed adds a secondary or more sort criteria.

Figure 6.55. Sorting Grid on Multiple Columns

Name ▲ 2	City ▲ 1	Year	
Ada Adams	Innsbruck	1 843	
Ada Darwin	Innsbruck	1 880	
Ada Lovelace	Innsbruck	1 868	
Ada Newton	Innsbruck	1 884	

Defining sort criteria programmatically can be done with the various alternatives of the `sort()` method. You can sort on a specific column with `sort(Object propertyId)`, which defaults to ascending sorting order, or `sort(Object propertyId, SortDirection direction)`, which allows specifying the sort direction.

```
grid.sort("name", SortDirection.DESENDING);
```

To sort on multiple columns, you need to use the fluid sort API with `sort(Sort)`, which allows chaining sorting rules. Sorting rules are created with the static `by()` method, which defines the primary sort column, and `then()`, which can be used to specify any secondary sort columns. They default to ascending sort order, but the sort direction can be given with an optional parameter.


```
// Sort first by city and then by name
grid.sort(Sort.by("city", SortDirection.ASCENDING)
        .then("name", SortDirection.DESENDING));
```

The container data source must support sorting. At least, it must implement `Container.Sortable`. Note that when using **GeneratedPropertyContainer**, as described in Section 6.24.5, “Generating Columns”, even though the container implements the interface, the wrapped container must also support it. Also, the generated properties are not normally sortable, but require special handling to enable sorting.

6.24.10. Editing

Grid supports line-based editing, where double-clicking a row opens the row editor. In the editor, the input fields can be edited, as well as navigated with **Tab** and **Shift+Tab** keys. If validation fails, an error is displayed and the user can correct the inputs.

To enable editing, you need to call `setEditorEnabled(true)` for the grid.

```
Grid grid = new Grid(GridExample.exampleDataSource());
grid.setEditorEnabled(true);
```

Grid supports two row editor modes - buffered and unbuffered. The default mode is buffered. The mode can be changed with `setBuffered(false)`

Buffered Mode

The editor has a **Save** button that commits the data item to the container data source and closes the editor. The **Cancel** button discards the changes and exits the editor.

A row under editing is illustrated in Figure 6.56, “Editing a Grid Row”.

Figure 6.56. Editing a Grid Row

Name	City	Year
Charles Lovelace	Innsbruck	1 965
Ada Lovelace	Turku	1 947
Charles Lovelace	Turku	1 968
Save Cancel		
Isaac Adams	Innsbruck	1 818
Isaac Newton	Innsbruck	1 804

Unbuffered Mode

The editor has no buttons and all changed data is committed directly to the container. If another row is clicked, the editor for the current row is closed and a row editor for the clicked row is opened.

Editor Fields

The editor fields are by default generated with a `FieldFactory` and bound to the container data source with a **FieldGroup**, which also handles tasks such as validation, as explained later.

To disable editing in a particular column, you can call `setEditable()` in the **Column** object with `false` parameter.

Customizing Editor Fields

The editor fields are normally created by the field factory of the editor's field group, which creates the fields according to the data types of their respective columns. To customize the editor fields of specific properties, such as to style them or to set up validation, you can provide them with `setEditorField()` in the respective columns.

In the following example, we configure a field with validation and styling:

```
TextField nameEditor = new TextField();

// Custom CSS style
nameEditor.addStyleName("nameeditor");

// Custom validation
nameEditor.addValidator(new RegexpValidator(
    "^\\p{Alpha}+ \\p{Alpha}+$",
    "Need first and last name"));

grid.getColumn("name").setEditorField(nameEditor);
```

Setting an editor field to `null` deletes the currently existing editor field, whether it was automatically generated or set explicitly with the setter. It will be regenerated with the factory the next time it is needed.

Binding to Data with a Field Group

Data binding to the item under editing is handled with a **FieldGroup**, which you need to set with `setEditorFieldGroup`. This is mostly useful when using special-purpose field groups, such as **BeanFieldGroup** to enable bean validation.

For example, assuming that we want to enable bean validation for a bean such as the following:

```
public class Person implements Serializable {
    @NotNull
    @Size(min=2, max=10)
    private String name;

    @Min(1)
    @Max(130)
    private int age;
    ...}
```

We can now use a **BeanFieldGroup** in the **Grid** as follows:

```
Grid grid = new Grid(exampleBeanDataSource());
grid.setColumnOrder("name", "age");
grid.setEditorEnabled(true);

// Enable bean validation for the data
grid.setEditorFieldGroup(
    new BeanFieldGroup<Person>(Person.class));

// Have some extra validation in a field
TextField nameEditor = new TextField();
nameEditor.addValidator(new RegexpValidator(
    "\\p{Alpha}+ \\p{Alpha}+$",
    "Need first and last name"));
grid.setEditorField("name", nameEditor);
```

To use bean validation as in the example above, you need to include an implementation of the Bean Validation API in the classpath, as described in Section 10.4.6, “Bean Validation”.

Editor Field Factory

The fields are generated by the **FieldFactory** of the field group; you can also set it with `setEditorFieldFactory()`. Alternatively, you can create the editor fields explicitly with `setEditorField()`.

6.24.11. Programmatic Scrolling

You can scroll to first item with `scrollToStart()`, to end with `scrollToEnd()`, or to a specific row with `scrollTo()`.

6.24.12. Generating Row or Cell Styles

You can style entire rows with a `RowStyleGenerator` or individual cells with a `CellStyleGenerator`.

Generating Row Styles

You set a `RowStyleGenerator` to a grid with `setRowStyleGenerator()`. The `getStyle()` method gets a **RowReference**, which contains various information about the row and a reference to the grid, and should return a style name or `null` if no style is generated.

For example, to add a style names to rows having certain values in one column, you can style them as follows:

```
grid.setRowStyleGenerator(rowRef -> { // Java 8
    if (! ((Boolean) rowRef.getItem()
        .getItemProperty("alive")
        .getValue()).booleanValue())
        return "grayed";
    else
        return null;
});
```

You could then style the rows with CSS as follows:

```
.v-grid-row.grayed {  
    color: gray;  
}
```

Generating Cell Styles

You set a `CellStyleGenerator` to a grid with `setCellStyleGenerator()`. The `getStyle()` method gets a **CellReference**, which contains various information about the cell and a reference to the grid, and should return a style name or `null` if no style is generated.

For example, to add a style name to a specific column, you can match on the property ID of the column as follows:

```
grid.setCellStyleGenerator(cellRef -> // Java 8  
    "born".equals(cellRef.getPropertyId())?  
        "rightalign" : null);
```

You could then style the cells with a CSS rule as follows:

```
.v-grid-cell.rightalign {  
    text-align: right;  
}
```

6.24.13. Styling with CSS

```
.v-grid {  
    .v-grid-scroller, .v-grid-scroller-horizontal { }  
    .v-grid-tablewrapper {  
        .v-grid-header {  
            .v-grid-row {  
                .v-grid-cell, .frozen, .v-grid-cell-focused { }  
            }  
        }  
        .v-grid-body {  
            .v-grid-row,  
            .v-grid-row-stripe,  
            .v-grid-row-has-data {  
                .v-grid-cell, .frozen, .v-grid-cell-focused { }  
            }  
        }  
        .v-grid-footer {  
            .v-grid-row {  
                .v-grid-cell, .frozen, .v-grid-cell-focused { }  
            }  
        }  
    }  
    .v-grid-header-deco { }  
    .v-grid-footer-deco { }  
    .v-grid-horizontal-scrollbar-deco { }  
    .v-grid-editor {  
        .v-grid-editor-cells { }  
        .v-grid-editor-footer {  
            .v-grid-editor-message { }  
            .v-grid-editor-buttons {  
                .v-grid-editor-save { }  
                .v-grid-editor-cancel { }  
            }  
        }  
    }  
}
```

```
}
}
```

A **Grid** has an overall `v-grid` style. The actual grid has three parts: a header, a body, and a footer. The scrollbar is a custom element with `v-grid-scroller` style. In addition, there are some decoration elements.

Grid cells, whether they are in the header, body, or footer, have a basic `v-grid-cell` style. Cells in a frozen column additionally have a `frozen` style. Rows have `v-grid-row` style, and every other row has additionally a `v-grid-row-stripe` style.

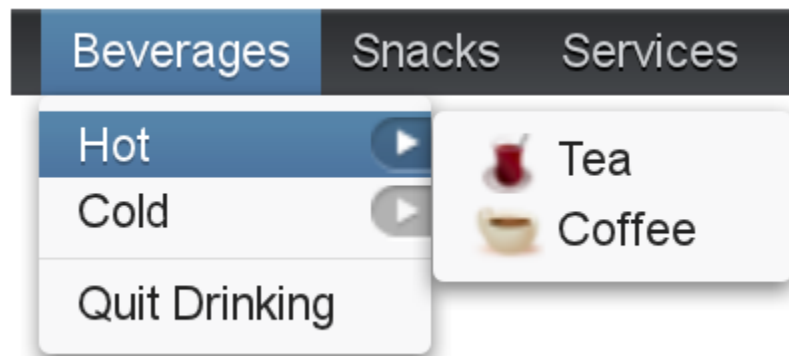
The focused row has additionally `v-grid-row-focused` style and focused cell `v-grid-cell-focused`. By default, cell focus is visible, with the border stylable with `$v-grid-cell-focused-border` parameter in Sass. Row focus has no visible styling, but can be made visible with the `$v-grid-row-focused-background-color` parameter or with a custom style rule.

In editing mode, a `v-grid-editor` overlay is placed on the row under editing. In addition to the editor field cells, it has an error message element, as well as the buttons.

6.25. MenuBar

The **MenuBar** component allows creating horizontal dropdown menus, much like the main menu in desktop applications.

Figure 6.57. Menu Bar



6.25.1. Creating a Menu

The actual menu bar component is first created as follows:

```
MenuBar barmenu = new MenuBar();
main.addComponent(barmenu);
```

You insert the top-level menu items to the **MenuBar** object with the `addItem()` method. It takes a string label, an icon resource, and a command as its parameters. The icon and command are not required and can be `null`. The `addItem()` method returns a **MenuBar.MenuItem** object, which you can use to add sub-menu items. The **MenuItem** has an identical `addItem()` method.

For example (the command is explained later):

```
// A top-level menu item that opens a submenu
MenuItem drinks = barmenu.addItem("Beverages", null, null);
```

```
// Submenu item with a sub-submenu
MenuItem hots = drinks.addItem("Hot", null, null);
hots.addItem("Tea",
    new ThemeResource("icons/tea-16px.png"), mycommand);
hots.addItem("Coffee",
    new ThemeResource("icons/coffee-16px.png"), mycommand);

// Another submenu item with a sub-submenu
MenuItem colds = drinks.addItem("Cold", null, null);
colds.addItem("Milk", null, mycommand);
colds.addItem("Weissbier", null, mycommand);

// Another top-level item
MenuItem snacks = barmenu.addItem("Snacks", null, null);
snacks.addItem("Weisswurst", null, mycommand);
snacks.addItem("Bratwurst", null, mycommand);
snacks.addItem("Currywurst", null, mycommand);

// Yet another top-level item
MenuItem servs = barmenu.addItem("Services", null, null);
servs.addItem("Car Service", null, mycommand);
```

6.25.2. Handling Menu Selection

Menu selection is handled by executing a *command* when the user selects an item from the menu. A command is a call-back class that implements the **MenuBar.Command** interface.

```
// A feedback component
final Label selection = new Label("-");
main.addComponent(selection);

// Define a common menu command for all the menu items.
MenuBar.Command mycommand = new MenuBar.Command() {
    public void menuSelected(MenuItem selectedItem) {
        selection.setValue("Ordered a " +
            selectedItem.getText() +
            " from menu.");
    }
};
```

6.25.3. CSS Style Rules

```
.v-menubar { }
.v-menubar-submenu { }
.v-menubar-menuitem { }
.v-menubar-menuitem-caption { }
.v-menubar-menuitem-selected { }
.v-menubar-submenu-indicator { }
```

The menu bar has the overall style name `.v-menubar`. Each menu item has `.v-menubar-menuitem` style normally and additionally `.v-menubar-selected` when the item is selected, that is, when the mouse pointer hovers over it. The item caption is inside a `v-menubar-menu-item-caption`. In the top-level menu bar, the items are directly under the component element.

Submenus are floating `v-menubar-submenu` elements outside the menu bar element. Therefore, you should not try to match on the component element for the submenu popups. In submenus, any further submenu levels are indicated with a `v-menubar-submenu-indicator`.

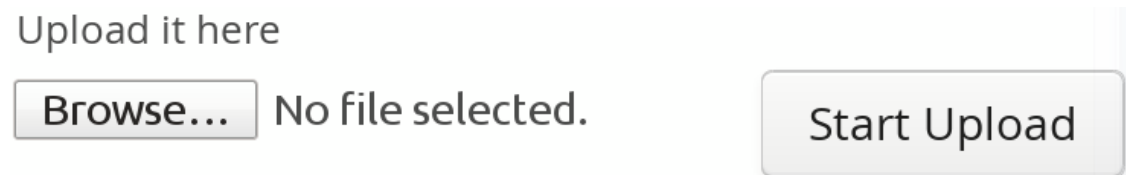
6.26. Upload

The **Upload** component allows a user to upload files to the server. It displays a file name entry box, a file selection button, and an upload submit button. The user can either write the filename in the text area or click the **Browse** button to select a file. After the file is selected, the user sends the file by clicking the upload submit button.

Uploading requires a receiver that implements `Upload.Receiver` to provide an output stream to which the upload is written by the server.

```
Upload upload = new Upload("Upload it here", receiver);
```

Figure 6.58. Upload Component



You can set the text of the upload button with `setButtonCaption()`. Note that it is difficult to change the caption or look of the **Browse** button. This is a security feature of web browsers. The language of the **Browse** button is determined by the browser, so if you wish to have the language of the **Upload** component consistent, you will have to use the same language in your application.

```
upload.setButtonCaption("Upload Now");
```

You can also hide the upload button with `.v-upload .v-button {display: none}` in theme, have custom logic for starting the upload, and call `startUpload()` to start it. If the upload component has `setImmediate(true)` enabled, uploading starts immediately after choosing the file.

6.26.1. Receiving Upload Data

The uploaded files are typically stored as files in a file system, in a database, or as temporary objects in memory. The upload component writes the received data to an **java.io.OutputStream** so you have plenty of freedom in how you can process the upload content.

To use the **Upload** component, you need to implement the **Upload.Receiver** interface. The `receiveUpload()` method of the receiver is called when the user clicks the submit button. The method must return an **OutputStream**. To do this, it typically creates a file or a memory buffer to which the stream is written. The method gets the file name and MIME type of the file, as reported by the browser.

While uploading, the upload progress can be monitored with an `Upload.ProgressListener`. The `updateProgress()` method gets the number of read bytes and the content length as parameters. The content length is reported by the browser, but the reported value is not reliable, and can also be unknown, in which case the value is -1. It is therefore recommended to follow the upload progress and check the allowed size in a progress listener. Upload can be terminated by calling `interruptUpload()` on the upload component. You may want to use a **ProgressBar** to visualize the progress, and in indeterminate mode if the content length is not known.

When an upload is finished, successfully or unsuccessfully, the **Upload** component will emit the **Upload.FinishedEvent** event, which you can handle with an **Upload.FinishedListener** added to the upload component. The event object will include the file name, MIME type, and final length of the file. More specific **Upload.FailedEvent** and **Upload.SucceededEvent** events will be called in the cases where the upload failed or succeeded, respectively.

The following example uploads images to /tmp/uploads directory in (UNIX) filesystem (the directory must exist or the upload fails). The component displays the uploaded image in an **Image** component.

```
// Show uploaded file in this placeholder
final Embedded image = new Embedded("Uploaded Image");
image.setVisible(false);

// Implement both receiver that saves upload in a file and
// listener for successful upload
class ImageUploader implements Receiver, SucceededListener {
    public File file;

    public OutputStream receiveUpload(String filename,
                                     String mimeType) {
        // Create upload stream
        FileOutputStream fos = null; // Stream to write to
        try {
            // Open the file for writing.
            file = new File("/tmp/uploads/" + filename);
            fos = new FileOutputStream(file);
        } catch (final java.io.FileNotFoundException e) {
            new Notification("Could not open file<br/>",
                            e.getMessage(),
                            Notification.Type.ERROR_MESSAGE)
                .show(Page.getCurrent());
            return null;
        }
        return fos; // Return the output stream to write to
    }

    public void uploadSucceeded(SucceededEvent event) {
        // Show the uploaded file in the image viewer
        image.setVisible(true);
        image.setSource(new FileResource(file));
    }
};

ImageUploader receiver = new ImageUploader();

// Create the upload with a caption and set receiver later
Upload upload = new Upload("Upload Image Here", receiver);
upload.setButtonCaption("Start Upload");
upload.addSucceededListener(receiver);

// Put the components in a panel
Panel panel = new Panel("Cool Image Storage");
Layout panelContent = new VerticalLayout();
panelContent.addComponents(upload, image);
panel.setContent(panelContent);
```

Note that the example does not check the type of the uploaded files in any way, which will cause an error if the content is anything else but an image. The program also assumes that the MIME

type of the file is resolved correctly based on the file name extension. After uploading an image, the component will look as shown in Figure 6.59, “Image Upload Example”.

Figure 6.59. Image Upload Example



6.26.2. CSS Style Rules

```
.v-upload { }
  .gwt-FileUpload { }
  .v-button { }
  .v-button-wrap { }
  .v-button-caption { }
```

The **Upload** component has an overall `v-upload` style. The upload button has the same structure and style as a regular **Button** component.

6.27. ProgressBar

The **ProgressBar** component allows displaying the progress of a task graphically. The progress is specified as a floating-point value between 0.0 and 1.0.

Figure 6.60. The Progress Bar Component



To display upload progress with the **Upload** component, you can update the progress bar in a `ProgressListener`.

When the position of a progress bar is done in a background thread, the change is not shown in the browser immediately. You need to use either polling or server push to update the browser. You can enable polling with `setPollInterval()` in the current UI instance. See Section 12.16, “Server Push” for instructions about using server push. Whichever method you use to update the UI, it is important to lock the user session by modifying the progress bar value inside `access()` call, as illustrated in the following example and described in Section 12.16.3, “Accessing UI from Another Thread”.

```
final ProgressBar bar = new ProgressBar(0.0f);
layout.addComponent(bar);

layout.addComponent(new Button("Increase",
    new ClickListener() {
        @Override
        public void buttonClick(ClickEvent event) {
            float current = bar.getValue();
            if (current < 1.0f)
                bar.setValue(current + 0.10f);
        }
    }));
```

6.27.1. Indeterminate Mode

In the indeterminate mode, a non-progressive indicator is displayed continuously. The indeterminate indicator is a circular wheel in the built-in themes. The progress value has no meaning in the indeterminate mode.

```
ProgressBar bar = new ProgressBar();
bar.setIndeterminate(true);
```

Figure 6.61. Indeterminate Progress Bar



6.27.2. CSS Style Rules

```
.v-progressbar, v-progressbar-indeterminate {}
.v-progressbar-wrapper {}
.v-progressbar-indicator {}
```

The progress bar has a `v-progressbar` base style. The animation is the background of the element with `v-progressbar-wrapper` style, by default an animated GIF image. The progress is an element with `v-progressbar-indicator` style inside the wrapper, and therefore displayed on top of it. When the progress element grows, it covers more and more of the animated background.

In the indeterminate mode, the top element also has the `v-progressbar-indeterminate` style. The built-in themes simply display the animated GIF in the top element and have the inner elements disabled.

6.28. Slider

The **Slider** is a vertical or horizontal bar that allows setting a numeric value within a defined range by dragging a bar handle with the mouse. The value is shown when dragging the handle.

Slider has a number of different constructors that take a combination of the caption, *minimum* and *maximum* value, *resolution*, and the *orientation* of the slider.

```
// Create a vertical slider
final Slider vertslider = new Slider(1, 100);
vertslider.setOrientation(SliderOrientation.VERTICAL);
```

min
Minimum value of the slider range. The default is 0.0.

max
Maximum value of the slider range. The default is 100.0.

resolution
The number of digits after the decimal point. The default is 0.

orientation
The orientation can be either horizontal (*SliderOrientation.HORIZONTAL*) or vertical (*SliderOrientation.VERTICAL*). The default is horizontal.

As the **Slider** is a field component, you can handle value changes with a **ValueChangeListener**. The value of the **Slider** field is a **Double** object.

```
// Shows the value of the vertical slider
final Label vertvalue = new Label();
vertvalue.setSizeUndefined();

// Handle changes in slider value.
vertslider.addValueChangeListener(
    new Property.ValueChangeListener() {
        public void valueChange(ValueChangeEvent event) {
            double value = (Double) vertslider.getValue();

            // Use the value
            box.setHeight((float) value, Sizeable.UNITS_PERCENTAGE);
            vertvalue.setValue(String.valueOf(value));
        }
    });

// The slider has to be immediate to send the changes
// immediately after the user drags the handle.
vertslider.setImmediate(true);
```

You can set the value with the `setValue()` method defined in **Slider** that takes the value as a native double value. The setter can throw a **ValueOutOfBoundsException**, which you must handle.

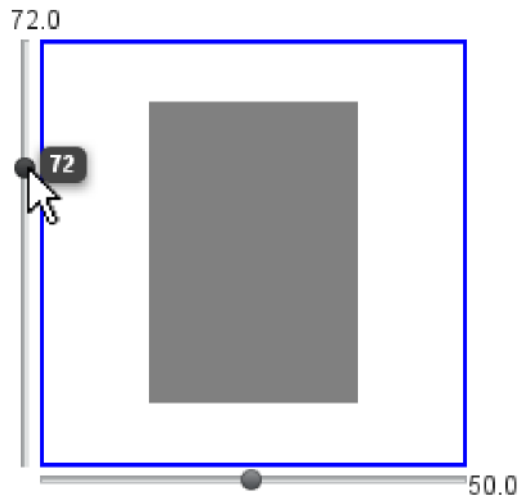
```
// Set the initial value. This has to be set after the
// listener is added if we want the listener to handle
// also this value change.
try {
    vertslider.setValue(50.0);
}
```

```
} catch (ValueOutOfBoundsException e) {  
}
```

Alternatively, you can use the regular `setValue(Object)`, which does not do bounds checking.

Figure 6.62, “The **Slider** Component” shows both vertical (from the code examples) and horizontal sliders that control the size of a box. The slider values are displayed also in separate labels.

Figure 6.62. The Slider Component



6.28.1. CSS Style Rules

```
.v-slider {}  
.v-slider-base {}  
.v-slider-handle {}
```

The enclosing style for the **Slider** is `v-slider`. The slider bar has style `v-slider-base`. Even though the handle is higher (for horizontal slider) or wider (for vertical slider) than the bar, the handle element is nevertheless contained within the slider bar element. The appearance of the handle comes from a background image defined in the *background* CSS property.

6.29. PopupView

The **PopupView** component allows opening a pop-up view either by clicking on a link or programmatically. The component has two representations: a minimized textual representation and the popped-up content. The view can contain any components. The view closes automatically when the mouse pointer moves outside the view.

In the following, we have a popup view with a text field and a button that opens automatically when the user clicks on a "Open the popup" link:

```
// Content for the PopupView  
VerticalLayout popupContent = new VerticalLayout();  
popupContent.addComponent(new TextField("Textfield"));  
popupContent.addComponent(new Button("Button"));  
  
// The component itself
```

```
PopupView popup = new PopupView("Pop it up", popupContent);
layout.addComponent(popup);
```

If the textual minimized representation is not given (a null is given), the component is invisible in the minimized state. The pop-up can be opened programmatically by calling `setPopupVisible(true)`. For example:

```
// A pop-up view without minimalized representation
PopupView popup = new PopupView(null,
    new Table(null, TableExample.generateContent()));

// A component to open the view
Button button = new Button("Show table", click -> // Java 8
    popup.setPopupVisible(true));

layout.addComponents(button, popup);
```

When the pop-up is opened or closed, a **PopupVisibilityEvent** is fired, which can be handled with a `PopupVisibilityListener` added with `setPopupVisibilityListener()`.

```
// Fill the pop-up content when it's popped up
popup.addPopupVisibilityListener(event -> {
    if (event.isPopupVisible()) {
        popupContent.removeAllComponents();
        popupContent.addComponent(new Table(null,
            TableExample.generateContent()));
    });
});
```

6.29.1. CSS Style Rules

```
.v-popupview {}
.v-overlay-container {
    .v-popupview-popup {
        .popupContent { }
    }
}
```

In minimalized state, the component has `v-popupview` style. When popped up, the pop-up content is shown in a `v-popupview-popup` overlay element under the `v-overlay-container`, which contains all floating overlays outside the component hierarchy.

6.30. Calendar

The **Calendar** component allows organizing and displaying calendar events. The main features of the calendar include:

- Monthly, weekly, and daily views
- Two types of events: all-day events and events with a time range
- Add events directly, from a **Container**, or with an event provider
- Control the range of the visible dates
- Selecting and editing date or time range by dragging
- Drag and drop events to calendar

- Support for localization and timezones

User interaction with the calendar elements, such as date and week captions as well as events, is handled with event listeners. Also date/time range selections, event dragging, and event resizing can be listened by the server. The weekly view has navigation buttons to navigate forward and backward in time. These actions are also listened by the server. Custom navigation can be implemented using event handlers

.

The data source of a calendar can be practically anything, as its events are queried dynamically by the component. You can bind the calendar to a Vaadin container, or to any other data source by implementing an *event provider*.

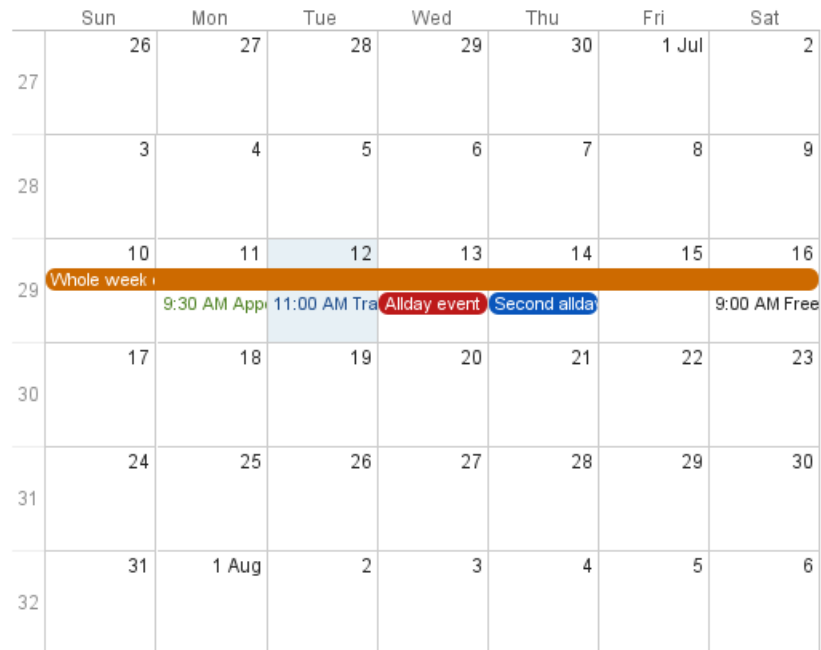
The **Calendar** has undefined size by default and you usually want to give it a fixed or relative size, for example as follows.

```
Calendar cal = new Calendar("My Calendar");
cal.setWidth("600px");
cal.setHeight("300px");
```

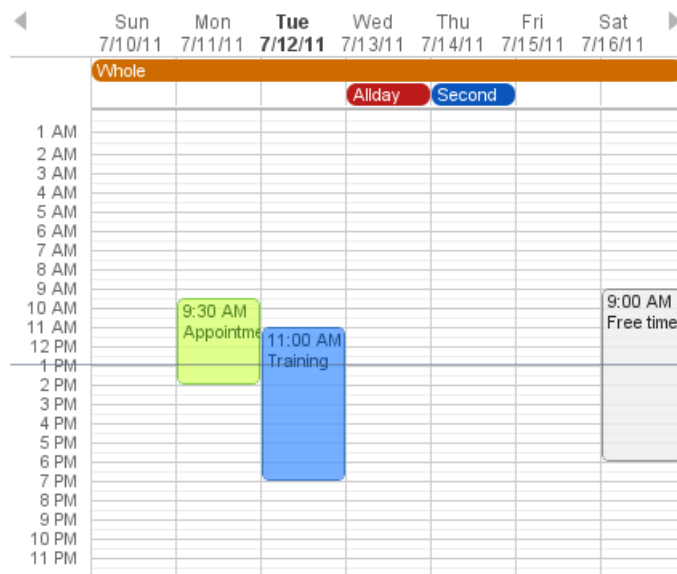
After creating the calendar, you need to set a time range for it, which also controls the view mode, and set up the data source for calendar events.

6.30.1. Date Range and View Mode

The Vaadin Calendar has two types of views that are shown depending on the date range of the calendar. The *weekly view* displays a week by default. It can show anything between one to seven days a week, and is also used as a single-day view. The view mode is determined from the *date range* of the calendar, defined by a start and an end date. Calendar will be shown in a *monthly view* when the date range is over than one week (seven days) long. The date range is always calculated in an accuracy of one millisecond.

Figure 6.63. Monthly view with All-Day and Normal Events

The monthly view, shown in Figure 6.63, “Monthly view with All-Day and Normal Events”, can easily be used to control all types of events, but it is best suited for events that last for one or more days. You can drag the events to move them. In the figure, you can see two longer events that are highlighted with a blue and green background color. Other markings are shorter day events that last less than a 24 hours. These events can not be moved by dragging in the monthly view.

Figure 6.64. Weekly View

In Figure 6.64, “Weekly View”, you can see four normal day events and also all-day events at the top of the time line grid.

In the following, we set the calendar to show only one day, which is the current day.

```
cal.setStartDate(new Date());  
cal.setEndDate(new Date());
```

Notice that although the range we set above is actually zero time long, the calendar still renders the time from 00:00 to 23:59. This is normal, as the Vaadin Calendar is guaranteed to render at least the date range provided, but may expand it. This behaviour is important to notice when we implement our own event providers.

6.30.2. Calendar Events

All occurrences in a calendar are represented as *events*. You have three ways to manage the calendar events:

- Add events directly to the **Calendar** object using the `addEvent()`
- Use a `Container` as a data source
- Use the *event provider* mechanism

You can add events with `addEvent()` and remove them with the `removeEvent()`. These methods will use the underlying event provider to write the modifications to the data source.

Event Interfaces and Providers

Events are handled through the `CalendarEvent` interface. The concrete class of the event depends on the specific **CalendarEventProvider** used in the calendar.

By default, **Calendar** uses a **BasicEventProvider** to provide events, which uses **BasicEvent** instances.

Calendar does not depend on any particular data source implementation. Events are queried by the **Calendar** from the provider that just has to implement the `CalendarEventProvider` interface. It is up to the event provider that **Calendar** gets the correct events.

You can bind any Vaadin **Container** to a calendar, in which case a **ContainerEventProvider** is used transparently. The container must be ordered by start date and time of the events. See Section 10.5, “Collecting Items in Containers” for basic information about containers.

Event Types

A calendar event requires a start time and an end time. These are the only mandatory properties. In addition, an event can also be set as an all-day event by setting the `all-day` property of the event. You can also set the `description` of an event, which is displayed as a tooltip in the user interface.

If the `all-day` field of the event is `true`, then the event is always rendered as an all-day event. In the monthly view, this means that no start time is displayed in the user interface and the event has an colored background. In the weekly view, all-day events are displayed in the upper part of the screen, and rendered similarly to the monthly view. In addition, when the time range of an event is 24 hours or longer, it is rendered as an all-day event in the monthly view.

When the time range of an event is equal or less than 24 hours, with the accuracy of one milli-second, the event is considered as a normal day event. Normal event has a start and end times that may be on different days.

Basic Events

The easiest way to add and manage events in a calendar is to use the *basic event* management API. Calendar uses by default a **BasicEventProvider**, which keeps the events in memory in an internal representation.

For example, the following adds a two-hour event starting from the current time. The standard Java **GregorianCalendar** provides various ways to manipulate date and time.

```
// Add a two-hour event
GregorianCalendar start = new GregorianCalendar();
GregorianCalendar end   = new GregorianCalendar();
end.add(java.util.Calendar.HOUR, 2);
calendar.addEvent(new BasicEvent("Calendar study",
    "Learning how to use Vaadin Calendar",
    start.getTime(), end.getTime()));
```

This adds a new event that lasts for 3 hours. As the **BasicEventProvider** and **BasicEvent** implement some optional event interfaces provided by the calendar package, there is no need to refresh the calendar. Just create events, set their properties and add them to the Event Provider.

6.30.3. Getting Events from a Container

You can use any Vaadin **Container** that implements the **Indexed** interface as the data source for calendar events. The **Calendar** will listen to change events from the container as well as write changes to the container. You can attach a container to a **Calendar** with `setContainerDataSource()`.

In the following example, we bind a **BeanItemContainer** that contains built-in **BasicEvent** events to a calendar.

```
// Create the calendar
Calendar calendar = new Calendar("Bound Calendar");

// Use a container of built-in BasicEvents
final BeanItemContainer<BasicEvent> container =
    new BeanItemContainer<BasicEvent>(BasicEvent.class);

// Create a meeting in the container
container.addBean(new BasicEvent("The Event", "Single Event",
    new GregorianCalendar(2012,1,14,12,00).getTime(),
    new GregorianCalendar(2012,1,14,14,00).getTime()));

// The container must be ordered by the start time. You
// have to sort the BIC every time after you have added
// or modified events.
container.sort(new Object[]{"start"}, new boolean[]{true});

calendar.setContainerDataSource(container, "caption",
    "description", "start", "end", "styleName");
```

The container must either use the default property IDs for event data, as defined in the **CalendarEvent** interface, or provide them as parameters for the `setContainerDataSource()` method, as we did in the example above.

Keeping the Container Ordered

The events in the container *must* be kept ordered by their start date/time. Failing to do so may and will result in the events not showing in the calendar properly.

Ordering depends on the container. With some containers, such as **BeanItemContainer**, you have to sort the container explicitly every time after you have added or modified events, usually with the `sort()` method, as we did in the example above. Some container, such as **JPAContainer**, keep the in container automatically order if you provide a sorting rule.

For example, you could order a **JPAContainer** by the following rule, assuming that the start date/time is held in the `startDate` property:

```
// The container must be ordered by start date. For JPAContainer
// we can just set up sorting once and it will stay ordered.
container.sort(new String[]{"startDate"}, new boolean[]{true});
```

6.30.4. Backward and Forward Navigation

Vaadin Calendar has only limited built-in navigation support. The weekly view has navigation buttons in the top left and top right corners.

You can handle backward and forward navigation with a `BackwardListener` and `ForwardListener`.

```
cal.setHandler(new BasicBackwardHandler() {
    protected void setDates(BackwardEvent event,
                             Date start, Date end) {

        java.util.Calendar calendar = event.getComponent()
                                      .getInternalCalendar();

        if (isThisYear(calendar, end)
            && isThisYear(calendar, start)) {
            super.setDates(event, start, end);
        }
    }
});
```

The forward navigation handler can be implemented in the same way. The example handler restricts the dates to the current year.

6.31. Composition with CustomComponent

The ease of making new user interface components is one of the core features of Vaadin. Typically, you simply combine existing built-in components to produce composite components. In many applications, such composite components make up the majority of the user interface.

As described earlier in Section 5.2.2, “Compositing Components”, you have two basic ways to create a composite - either by extending a layout component or the **CustomComponent**, which typically wraps around a layout component. The benefit of wrapping a layout composite in **CustomComponent** is mainly encapsulation - hiding the implementation details of the composition. Otherwise, a user of the composite could rely on implementation details, which would create an unwanted dependency.

To create a composite, you need to inherit the **CustomComponent** and set the *composition root* component in the constructor. The composition root is typically a layout component that contains other components.

For example:

```
class MyComposite extends CustomComponent {
    public MyComposite(String message) {
        // A layout structure used for composition
        Panel panel = new Panel("My Custom Component");
        VerticalLayout panelContent = new VerticalLayout();
        panelContent.setMargin(true); // Very useful
        panel.setContent(panelContent);

        // Compose from multiple components
        Label label = new Label(message);
        label.setSizeUndefined(); // Shrink
        panelContent.addComponent(label);
        panelContent.addComponent(new Button("Ok"));

        // Set the size as undefined at all levels
        panelContent.setSizeUndefined();
        panel.setSizeUndefined();
        setSizeUndefined();

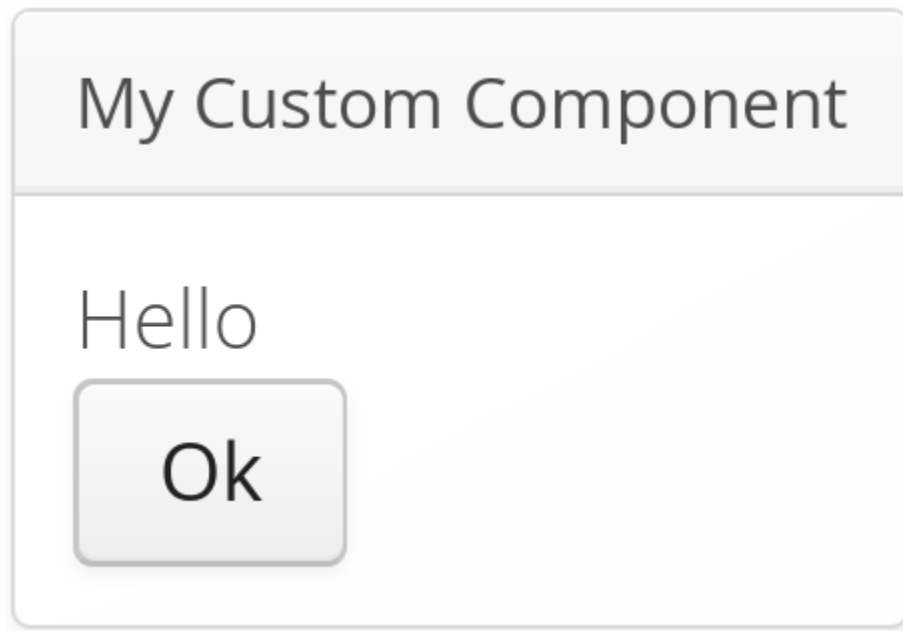
        // The composition root MUST be set
        setCompositionRoot(panel);
    }
}
```

Take note of the sizing when trying to make a customcomponent that shrinks to fit the contained components. You have to set the size as undefined at all levels; the sizing of the composite component and the composition root are separate.

You can use the component as follows:

```
MyComposite mycomposite = new MyComposite("Hello");
```

The rendered component is shown in Figure 6.65, “A Custom Composite Component”.

Figure 6.65. A Custom Composite Component

You can also inherit any other components, such as layouts, to attain similar composition. Even further, you can create entirely new low-level components, by integrating pure client-side components or by extending the client-side functionality of built-in components. Development of new components is covered in Chapter 17, *Integrating with the Server-Side*.

6.32. Composite Fields with CustomField

The **CustomField** is a way to create composite components like with **CustomComponent**, except that it implements the `Field` interface and inherits **AbstractField**, described in Section 6.4, “Field Components”. A field allows editing a property value in the Vaadin data model, and can be bound to data with field groups, as described in Section 10.4, “Creating Forms by Binding Fields to Items”. The field values are buffered and can be validated with validators.

A composite field class must implement the `getType()` and `initContent()` methods. The latter should return the content composite of the field. It is typically a layout component, but can be any component.

It is also possible to override `validate()`, `setInternalValue()`, `commit()`, `setPropertyDataSource()`, `isEmpty()` and other methods to implement different functionalities in the field. Methods overriding `setInternalValue()` should call the superclass method.

6.33. Embedded Resources

You can embed images in Vaadin UIs with the **Image** component, Adobe Flash graphics with **Flash**, and other web content with **BrowserFrame**. There is also a generic **Embedded** component for embedding other object types. The embedded content is referenced as *resources*, as described in Section 5.5, “Images and Other Resources”.

The following example displays an image as a class resource loaded with the class loader:

```
Image image = new Image("Yes, logo:",  
    new ClassResource("vaadin-logo.png"));  
main.addComponent(image);
```

The caption can be given as null to disable it. An empty string displays an empty caption which takes a bit space. The caption is managed by the containing layout.

You can set an alternative text for an embedded resource with `setAlternateText()`, which can be shown if images are disabled in the browser for some reason. The text can be used for accessibility purposes, such as for text-to-speech generation.

6.33.1. Embedded Image

The **Image** component allows embedding an image resource in a Vaadin UI.

```
// Serve the image from the theme  
Resource res = new ThemeResource("img/myimage.png");  
  
// Display the image without caption  
Image image = new Image(null, res);  
layout.addComponent(image);
```

The **Image** component has by default undefined size in both directions, so it will automatically fit the size of the embedded image. If you want scrolling with scroll bars, you can put the image inside a **Panel** that has a defined size to enable scrolling, as described in Section 7.6.1, “Scrolling the Panel Content”. You can also put it inside some other component container and set the `overflow: auto` CSS property for the container element in a theme to enable automatic scrollbars.

Generating and Reloading Images

You can also generate the image content dynamically using a **StreamResource**, as described in Section 5.5.5, “Stream Resources”, or with a **RequestHandler**.

If the image changes, the browser needs to reload it. Simply updating the stream resource is not enough. Because of how caching is handled in some browsers, you can cause a reload easiest by renaming the filename of the resource with a unique name, such as one including a timestamp. You should set cache time to zero with `setCacheTime()` for the resource object when you create it.//BUG #2470.

```
// Create the stream resource with some initial filename  
StreamResource imageResource =  
    new StreamResource(imageSource, "initial-filename.png");  
  
// Instruct browser not to cache the image  
imageResource.setCacheTime(0);  
  
// Display the image  
Image image = new Image(null, imageResource);
```

When refreshing, you also need to call `markAsDirty()` for the **Image** object.

```
// This needs to be done, but is not sufficient  
image.markAsDirty();  
  
// Generate a filename with a timestamp  
SimpleDateFormat df = new SimpleDateFormat("yyyyMMddHHmmssSSS");
```

```
String filename = "myfilename-" + df.format(new Date()) + ".png";

// Replace the filename in the resource
imageResource.setFilename(makeImageFilename());
```

6.33.2. Adobe Flash Graphics

The **Flash** component allows embedding Adobe Flash animations in Vaadin UIs.

```
Flash flash = new Flash(null,
    new ThemeResource("img/vaadin_spin.swf"));
layout.addComponent(flash);
```

You can set Flash parameters with `setParameter()`, which takes a parameter's name and value as strings. You can also set the `codeBase`, `archive`, and `standBy` attributes for the Flash object element in HTML.

6.33.3. BrowserFrame

The **BrowserFrame** allows embedding web content inside an HTML `<iframe>` element. You can refer to an external URL with **ExternalResource**.

As the **BrowserFrame** has undefined size by default, it is critical that you define a meaningful size for it, either fixed or relative.

```
BrowserFrame browser = new BrowserFrame("Browser",
    new ExternalResource("http://demo.vaadin.com/sampler/"));
browser.setWidth("600px");
browser.setHeight("400px");
layout.addComponent(browser);
```

Notice that web pages can prevent embedding them in an `<iframe>`.

6.33.4. Generic Embedded Objects

The generic **Embedded** component allows embedding all sorts of objects, such as SVG graphics, Java applets, and PDF documents, in addition to the images, Flash graphics, and browser frames which you can embed with the specialized components.

For example, to display a Flash animation:

```
// A resource reference to some object
Resource res = new ThemeResource("img/vaadin_spin.swf");

// Display the object
Embedded object = new Embedded("My Object", res);
layout.addComponent(object);
```

Or an SVG image:

```
// A resource reference to some object
Resource res = new ThemeResource("img/reindeer.svg");

// Display the object
Embedded object = new Embedded("My SVG", res);
object.setMimeType("image/svg+xml"); // Unnecessary
layout.addComponent(object);
```

The MIME type of the objects is usually detected automatically from the filename extension with the **FileTypeResolver** utility in Vaadin. If not, you can set it explicitly with `setMimeType()`, as was done in the example above (where it was actually unnecessary).

Some embeddable object types may require special support in the browser. You should make sure that there is a proper fallback mechanism if the browser does not support the embedded type.

Chapter 7

Managing Layout

7.1. Overview	228
7.2. UI, Window, and Panel Content	229
7.3. VerticalLayout and HorizontalLayout	230
7.4. GridLayout	235
7.5. FormLayout	239
7.6. Panel	241
7.7. Sub-Windows	243
7.8. HorizontalSplitPanel and VerticalSplitPanel	246
7.9. TabSheet	248
7.10. Accordion	251
7.11. AbsoluteLayout	253
7.12. CssLayout	255
7.13. Layout Formatting	258
7.14. Custom Layouts	263

Layout management of Vaadin is a direct successor of the web-based concept for separation of content and appearance and of the Java AWT solution for binding the layout and user interface components into objects in programs. Vaadin layout components allow you to position your UI components on the screen in a hierarchical fashion, much like in conventional Java UI toolkits such as AWT, Swing, or SWT. In addition, you can approach the layout from the direction of the web with the **CustomLayout** component, which you can use to write your layout as a template in HTML that provides locations of any contained components. The **AbsoluteLayout** allows the old-style pixel-position based layouting, but it also supports percentual values, which makes it usable for scalable layouts. It is also useful as an area on which the user can position items with drag and drop.

7.1. Overview

The user interface components in Vaadin can roughly be divided in two groups: components that the user can interact with and layout components for placing the other components to specific places in the user interface. The layout components are identical in their purpose to layout managers in regular desktop frameworks for Java and you can use plain Java to accomplish sophisticated component layouting.

You start by creating a content layout for the UI and then add other layout components hierarchically, and finally the interaction components as the leaves of the component tree.

```
// Set the root layout for the UI
VerticalLayout content = new VerticalLayout();
setContent(content);

// Add the topmost component.
content.addComponent(new Label("The Ultimate Cat Finder"));

// Add a horizontal layout for the bottom part.
HorizontalLayout bottom = new HorizontalLayout();
content.addComponent(bottom);

bottom.addComponent(new Tree("Major Planets and Their Moons"));
bottom.addComponent(new Panel());
...
```

Or in the declarative format:

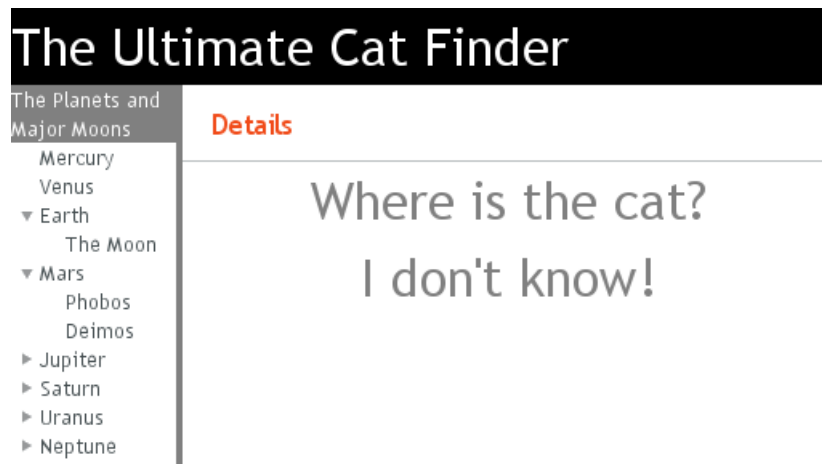
```
<vaadin-vertical-layout>
  <vaadin-label>The Ultimate Cat Finder</vaadin-label>

  <vaadin-horizontal-layout>
    <vaadin-tree caption="Major Planets and Their Moons"/>
    <vaadin-panel/>
  </vaadin-horizontal-layout>
</vaadin-vertical-layout>
```

You will usually need to tune the layout components a bit by setting sizes, expansion ratios, alignments, spacings, and so on. The general settings are described in Section 7.13, “Layout Formatting”.

Layouts are coupled with themes that specify various layout features, such as backgrounds, borders, text alignment, and so on. Definition and use of themes is described in Chapter 9, *Themes*.

You can see a finished version of the above example in Figure 7.1, “Layout Example”.

Figure 7.1. Layout Example

7.2. UI, Window, and Panel Content

The **UI**, **Window**, and **Panel** all have a single content component, which you need to set with `setContent()`. The content is usually a layout component, although any component is allowed.

```
Panel panel = new Panel("This is a Panel");
VerticalLayout panelContent = new VerticalLayout();
panelContent.addComponent(new Label("Hello!"));
panel.setContent(panelContent);
```

```
// Set the panel as the content of the UI
setContent(panel);
```

The size of the content is the default size of the particular layout component, for example, a **VerticalLayout** has 100% width and undefined height by default (this coincides with the defaults for **Panel** and **Label**). If such a layout with undefined height grows higher than the browser window, it will flow out of the view and scrollbars will appear. In many applications, you want to use the full area of the browser view. Setting the components contained inside the content layout to full size is not enough, and would actually lead to an invalid state if the height of the content layout is undefined.

```
// First set the root content for the UI
VerticalLayout content = new VerticalLayout();
setContent(content);

// Set the content size to full width and height
content.setSizeFull();

// Add a title area on top of the screen. This takes
// just the vertical space it needs.
content.addComponent(new Label("My Application"));

// Add a menu-view area that takes rest of vertical space
HorizontalLayout menuview = new HorizontalLayout();
menuview.setSizeFull();
content.addComponent(menuview);
```

See Section 7.13.1, “Layout Size” for more information about setting layout sizes.

7.3. VerticalLayout and HorizontalLayout

VerticalLayout and **HorizontalLayout** are ordered layouts for laying components out either vertically or horizontally, respectively. They both extend from **AbstractOrderedLayout**, together with the **FormLayout**. These are the two most important layout components in Vaadin, and typically you have a **VerticalLayout** as the root content of a UI.

VerticalLayout has 100% default width and undefined height, so it fills the containing layout (or UI) horizontally, and fits its content vertically. **HorizontalLayout** has undefined size in both dimensions.

Typical use of the layouts goes as follows:

```
VerticalLayout vertical = new VerticalLayout ();
vertical.addComponent(new TextField("Name"));
vertical.addComponent(new TextField("Street address"));
vertical.addComponent(new TextField("Postal code"));
layout.addComponent(vertical);
```

The component captions are placed above the component, so the layout will look as follows:

Name

Street address

Postal code

Using **HorizontalLayout** gives the following layout:

Name Street address Postal code

7.3.1. Properties or Attributes

Ordered layouts have the following properties:

Table 7.1. Properties and Declarative Attributes

Property	Declarative Attribute
<i>componentAlignment</i>	In child components: [literal]: <i>left</i> (default), : <i>center</i> , : <i>right</i> , : <i>top</i> (default), : <i>middle</i> , : <i>bottom</i>
<i>spacing</i>	<i>spacing</i> [replaceable][=<boolean>]
<i>margin</i>	<i>margin</i> [replaceable][=<boolean>]
<i>expandRatio</i>	In child components: [parameter]: <i>expand</i> =<integer># <i>or</i> [parameter]: <i>expand</i> #(implies ratio 1)

7.3.2. Spacing in Ordered Layouts

The ordered layouts can have spacing between the horizontal or vertical cells. The spacing can be enabled with `setSpacing(true)` or declaratively with the `spacing` attribute.

The spacing as a default height or width, which can be customized in CSS. You need to set the height or width for spacing elements with `v-spacing` style. You also need to specify an enclosing rule element in a CSS selector, such as `v-vertical` for a **VerticalLayout** or `v-horizontal` for a **HorizontalLayout**. You can also use `v-vertical` and `v-horizontal` for all vertically or horizontally ordered layouts, such as **FormLayout**.

For example, the following sets the amount of spacing for all **VerticalLayout**s, as well as **[classname]#FormLayout**, in the UI:

```
.v-vertical > .v-spacing {
    height: 30px;
}
```

Or for **HorizontalLayout**:

```
.v-horizontal > .v-spacing {
    width: 50px;
}
```

7.3.3. Sizing Contained Components

The components contained within an ordered layout can be laid out in a number of different ways depending on how you specify their height or width in the primary direction of the layout component.

Figure 7.2. Component Widths in HorizontalLayout

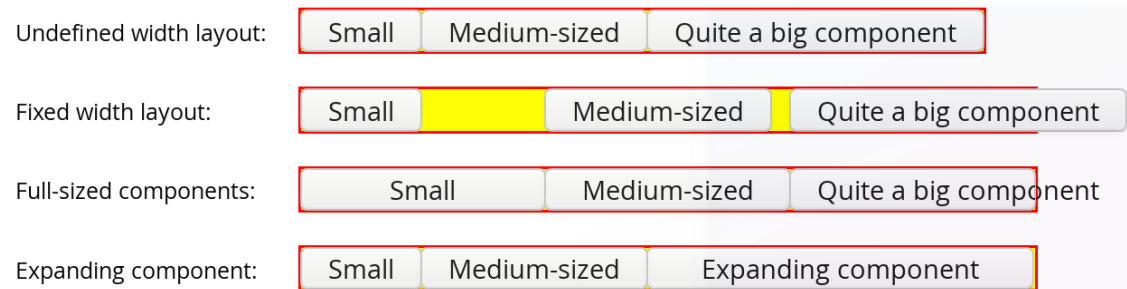


Figure 7.2, “Component Widths in **HorizontalLayout**” gives a summary of the sizing options for a **HorizontalLayout**. The figure is broken down in the following subsections.

Layout with Undefined Size

If a **VerticalLayout** has undefined height or **HorizontalLayout** undefined width, the layout will shrink to fit the contained components so that there is no extra space between them.

```
HorizontalLayout fittingLayout = new HorizontalLayout();
fittingLayout.setWidth(Sizeable.SIZE_UNDEFINED, 0); // Default
fittingLayout.addComponent(new Button("Small"));
fittingLayout.addComponent(new Button("Medium-sized"));
```

```
fittingLayout.addComponent(new Button("Quite a big component"));
parentLayout.addComponent(fittingLayout);
```

The both layouts actually have undefined height by default and **HorizontalLayout** has also undefined width, while **VerticalLayout** has 100% relative width.

If such a vertical layout with undefined height continues below the bottom of a window (a **Window** object), the window will pop up a vertical scroll bar on the right side of the window area. This way, you get a "web page". The same applies to **Panel**.



A layout that contains components with percentual size must have a defined size!

If a layout has undefined size and a contained component has, say, 100% size, the component would fill the space given by the layout, while the layout would shrink to fit the space taken by the component, which would be a paradox. This requirement holds for height and width separately. The debug window allows detecting such invalid cases; see Section 12.3.5, "Inspecting Component Hierarchy".

An exception to the above rule is a case where you have a layout with undefined size that contains a component with a fixed or undefined size together with one or more components with relative size. In this case, the contained component with fixed (or undefined) size in a sense defines the size of the containing layout, removing the paradox. That size is then used for the relatively sized components.

The technique can be used to define the width of a **VerticalLayout** or the height of a **HorizontalLayout**.

```
// Vertical layout would normally have 100% width
VerticalLayout vertical = new VerticalLayout();

// Shrink to fit the width of contained components
vertical.setWidth(Sizeable.SIZE_UNDEFINED, 0);

// Label has normally 100% width, but we set it as
// undefined so that it will take only the needed space
Label label =
    new Label("\u2190 The VerticalLayout shrinks to fit "+
              "the width of this Label \u2192");
label.setWidth(Sizeable.SIZE_UNDEFINED, 0);
vertical.addComponent(label);

// Button has undefined width by default
Button butt = new Button("\u2190 This Button takes 100% "+
                          "of the width \u2192");
butt.setWidth("100%");
vertical.addComponent(butt);
```

Figure 7.3. Defining the Size with a Component

This is a VerticalLayout with two components



Layout with Defined Size

If you set a **HorizontalLayout** to a defined size horizontally or a **VerticalLayout** vertically, and there is space left over from the contained components, the extra space is distributed equally between the component cells. The components are aligned within these cells according to their alignment setting, top left by default, as in the example below.

```
fixedLayout.setWidth("400px");
```

Using percentual sizes for components contained in a layout requires answering the question, "Percentage of what?" There is no sensible default answer for this question in the current implementation of the layouts, so in practice, you may not define "100%" size alone.

Expanding Components

Often, you want to have one component that takes all the available space left over from other components. You need to set its size as 100% and set it as *expanding* with `setExpandRatio()`. The second parameter for the method is an expansion ratio, which is relevant if there are more than one expanding component, but its value is irrelevant for a single expanding component.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");

// These buttons take the minimum size.
layout.addComponent(new Button("Small"));
layout.addComponent(new Button("Medium-sized"));

// This button will expand.
Button expandButton = new Button("Expanding component");

// Use 100% of the expansion cell's width.
expandButton.setWidth("100%");

// The component must be added to layout before setting the ratio.
layout.addComponent(expandButton);

// Set the component's cell to expand.
layout.setExpandRatio(expandButton, 1.0f);

parentLayout.addComponent(layout);
```

In the declarative format, you need to specify the `:expand` attribute in the child components. The attribute defaults to expand ratio 1.

Notice that you can not call `setExpandRatio()` before you have added the component to the layout, because it can not operate on an component that it doesn't yet have.

Expand Ratios

If you specify an expand ratio for multiple components, they will all try to use the available space according to the ratio.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");

// Create three equally expanding components.
String[] captions = { "Small", "Medium-sized",
                      "Quite a big component" };
```

```
for (int i = 1; i <= 3; i++) {
    Button button = new Button(captions[i-1]);
    button.setWidth("100%");
    layout.addComponent(button);

    // Have uniform 1:1:1 expand ratio.
    layout.setExpandRatio(button, 1.0f);
}
```

As the example used the same ratio for all components, the ones with more content may have the content cut. Below, we use differing ratios:

```
// Expand ratios for the components are 1:2:3.
layout.setExpandRatio(button, i * 1.0f);
```

If the size of the expanding components is defined as a percentage (typically "100%"), the ratio is calculated from the *overall* space available for the relatively sized components. For example, if you have a 100 pixels wide layout with two cells with 1.0 and 4.0 respective expansion ratios, and both the components in the layout are set as `setWidth("100%")`, the cells will have respective widths of 20 and 80 pixels, regardless of the minimum size of the components.

However, if the size of the contained components is undefined or fixed, the expansion ratio is of the *excess* available space. In this case, it is the excess space that expands, not the components.

```
for (int i = 1; i <= 3; i++) {
    // Button with undefined size.
    Button button = new Button(captions[i - 1]);

    layout4.addComponent(button);

    // Expand ratios are 1:2:3.
    layout4.setExpandRatio(button, i * 1.0f);
}
```

It is not meaningful to combine expanding components with percentually defined size and components with fixed or undefined size. Such combination can lead to a very unexpected size for the percentually sized components.

Percentage of Cells

A percentual size of a component defines the size of the component *within its cell*. Usually, you use "100%", but a smaller percentage or a fixed size (smaller than the cell size) will leave an empty space in the cell and align the component within the cell according to its alignment setting, top left by default.

```
HorizontalLayout layout50 = new HorizontalLayout();
layout50.setWidth("400px");

String[] captions1 = { "Small 50%", "Medium 50%",
                       "Quite a big 50%" };
for (int i = 1; i <= 3; i++) {
    Button button = new Button(captions1[i-1]);
    button.setWidth("50%");
    layout50.addComponent(button);

    // Expand ratios for the components are 1:2:3.
    layout50.setExpandRatio(button, i * 1.0f);
}
```



```
}  
parentLayout.addComponent(layout50);
```

7.4. GridLayout

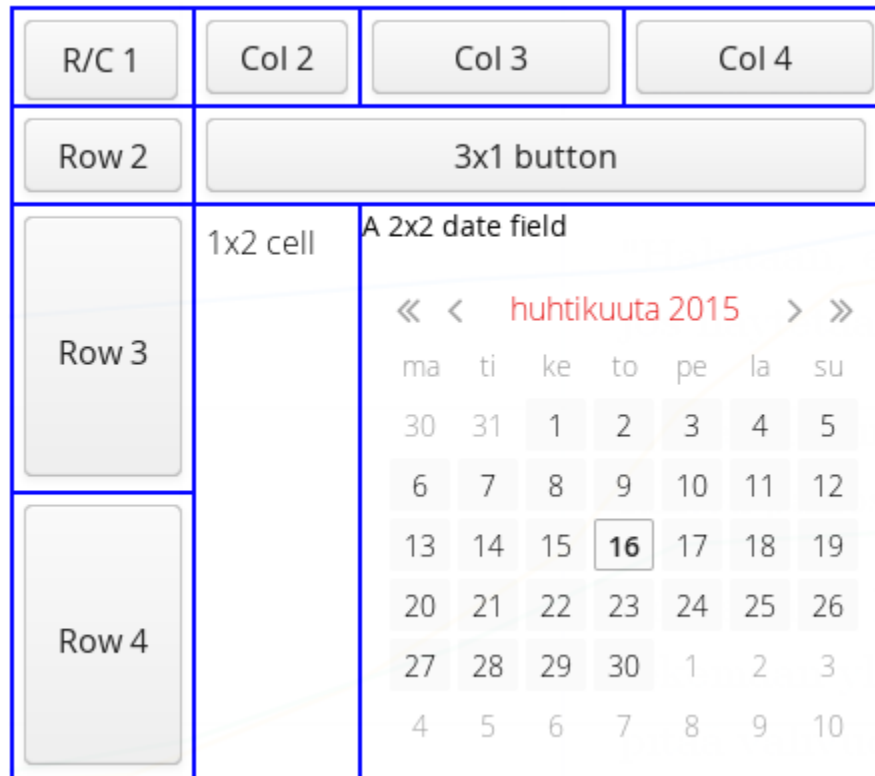
GridLayout container lays components out on a grid, defined by the number of columns and rows. The columns and rows of the grid serve as coordinates that are used for laying out components on the grid. Each component can use multiple cells from the grid, defined as an area (x1,y1,x2,y2), although they typically take up only a single grid cell.

The grid layout maintains a cursor for adding components in left-to-right, top-to-bottom order. If the cursor goes past the bottom-right corner, it will automatically extend the grid downwards by adding a new row.

The following example demonstrates the use of **GridLayout**. The `addComponent` takes a component and optional coordinates. The coordinates can be given for a single cell or for an area in x,y (column,row) order. The coordinate values have a base value of 0. If coordinates are not given, the cursor will be used.

```
// Create a 4 by 4 grid layout.  
GridLayout grid = new GridLayout(4, 4);  
grid.addStyleName("example-gridlayout");  
  
// Fill out the first row using the cursor.  
grid.addComponent(new Button("R/C 1"));  
for (int i = 0; i < 3; i++) {  
    grid.addComponent(new Button("Col " + (grid.getCursorX() + 1)));  
}  
  
// Fill out the first column using coordinates.  
for (int i = 1; i < 4; i++) {  
    grid.addComponent(new Button("Row " + i), 0, i);  
}  
  
// Add some components of various shapes.  
grid.addComponent(new Button("3x1 button"), 1, 1, 3, 1);  
grid.addComponent(new Label("1x2 cell"), 1, 2, 1, 3);  
InlineDateField date = new InlineDateField("A 2x2 date field");  
date.setResolution(DateField.RESOLUTION_DAY);  
grid.addComponent(date, 2, 2, 3, 3);
```

The resulting layout will look as follows. The borders have been made visible to illustrate the layout cells.

Figure 7.4. The Grid Layout Component

A component to be placed on the grid must not overlap with existing components. A conflict causes throwing a **GridLayout.OverlapsException**.

7.4.1. Sizing Grid Cells

You can define the size of both a grid layout and its components in either fixed or percentual units, or leave the size undefined altogether, as described in Section 6.3.9, “Sizing Components”. Section 7.13.1, “Layout Size” gives an introduction to sizing of layouts.

The size of the **GridLayout** component is undefined by default, so it will shrink to fit the size of the components placed inside it. In most cases, especially if you set a defined size for the layout but do not set the contained components to full size, there will be some unused space. The position of the non-full components within the grid cells will be determined by their *alignment*. See Section 7.13.3, “Layout Cell Alignment” for details on how to align the components inside the cells.

The components contained within a **GridLayout** layout can be laid out in a number of different ways depending on how you specify their height or width. The layout options are similar to **HorizontalLayout** and **VerticalLayout**, as described in Section 7.3, “**VerticalLayout** and **HorizontalLayout**”.



A layout that contains components with percentual size must have a defined size!

If a layout has undefined size and a contained component has, say, 100% size, the component would fill the space given by the layout, while the layout would shrink

to fit the space taken by the component, which is a paradox. This requirement holds for height and width separately. The debug mode allows detecting such invalid cases; see Section 12.3.1, “Enabling the Debug Mode”.

Often, you want to have one or more rows or columns that take all the available space left over from non-expanding rows or columns. You need to set the rows or columns as *expanding* with `setRowExpandRatio()` and `setColumnExpandRatio()`. The first parameter for these methods is the index of the row or column to set as expanding. The second parameter for the methods is an expansion ratio, which is relevant if there are more than one expanding row or column, but its value is irrelevant if there is only one. With multiple expanding rows or columns, the ratio parameter sets the relative portion how much a specific row/column will take in relation with the other expanding rows/columns.

```
GridLayout grid = new GridLayout(3,2);

// Layout containing relatively sized components must have
// a defined size, here is fixed size.
grid.setWidth("600px");
grid.setHeight("200px");

// Add some content
String labels [] = {
    "Shrinking column<br/>Shrinking row",
    "Expanding column (1:<br/>Shrinking row",
    "Expanding column (5:<br/>Shrinking row",
    "Shrinking column<br/>Expanding row",
    "Expanding column (1:<br/>Expanding row",
    "Expanding column (5:<br/>Expanding row"
};
for (int i=0; i<labels.length; i++) {
    Label label = new Label(labels[i], ContentMode.HTML);
    label.setWidth(null); // Set width as undefined
    grid.addComponent(label);
}

// Set different expansion ratios for the two columns
grid.setColumnExpandRatio(1, 1);
grid.setColumnExpandRatio(2, 5);

// Set the bottom row to expand
grid.setRowExpandRatio(1, 1);

// Align and size the labels.
for (int col=0; col<grid.getColumns(); col++) {
    for (int row=0; row<grid.getRows(); row++) {
        Component c = grid.getComponent(col, row);
        grid.setComponentAlignment(c, Alignment.TOP_CENTER);

        // Make the labels high to illustrate the empty
        // horizontal space.
        if (col != 0 || row != 0)
            c.setHeight("100%");
    }
}
```

Figure 7.5. Expanding Rows and Columns in GridLayout

Shrinking column Shrinking row	Expanding column (1:) Shrinking row	Expanding column (5:) Shrinking row
Shrinking column Expanding row	Expanding column (1:) Expanding row	Expanding column (5:) Expanding row

If the size of the contained components is undefined or fixed, the expansion ratio is of the *excess* space, as in Figure 7.5, “Expanding Rows and Columns in **GridLayout**” (excess horizontal space is shown in white). However, if the size of the all the contained components in the expanding rows or columns is defined as a percentage, the ratio is calculated from the *overall* space available for the percentually sized components. For example, if we had a 100 pixels wide grid layout with two columns with 1.0 and 4.0 respective expansion ratios, and all the components in the grid were set as `setWidth("100%")`, the columns would have respective widths of 20 and 80 pixels, regardless of the minimum size of their contained components.

7.4.2. CSS Style Rules

```
.v-gridlayout {}
.v-gridlayout-margin {}
```

The `v-gridlayout` is the root element of the **GridLayout** component. The `v-gridlayout-margin` is a simple element inside it that allows setting a padding between the outer element and the cells.

For styling the individual grid cells, you should style the components inserted in the cells. The implementation structure of the grid can change, so depending on it, as is done in the example below, is not generally recommended. Normally, if you want to have, for example, a different color for a certain cell, just make set the component inside it `setSizeFull()`, and add a style name for it. Sometimes you may need to use a layout component between a cell and its actual component just for styling.

The following example shows how to make the grid borders visible, as in Figure 7.5, “Expanding Rows and Columns in **GridLayout**”.

```
.v-gridlayout-gridexpandratio {
    background: blue; /* Creates a "border" around the grid. */
    margin:      10px; /* Empty space around the layout. */
}

/* Add padding through which the background color shows. */
.v-gridlayout-gridexpandratio .v-gridlayout-margin {
    padding: 2px;
}

/* Add cell borders and make the cell backgrounds white.
 * Warning: This depends heavily on the HTML structure. */
.v-gridlayout-gridexpandratio > div > div > div {
    padding: 2px; /* Layout background will show through. */
    background: white; /* The cells will be colored white. */
}
```

```
/* Components inside the layout are a safe way to style cells. */
.v-gridlayout-gridexpandratio .v-label {
    text-align: left;
    background: #ffffc0; /* Pale yellow */
}
```

You should beware of margin, padding, and border settings in CSS as they can mess up the layout. The dimensions of layouts are calculated in the Client-Side Engine of Vaadin and some settings can interfere with these calculations. For more information, on margins and spacing, see Section 7.13.4, “Layout Cell Spacing” and Section 7.13.5, “Layout Margins”

7.5. FormLayout

FormLayout lays the components and their captions out in two columns, with optional indicators for required fields and errors that can be shown for each field. The field captions can have an icon in addition to the text. **FormLayout** is an ordered layout and much like **VerticalLayout**. For description of margins, spacing, and other features in ordered layouts, see Section 7.3, “**VerticalLayout** and **HorizontalLayout**”.

The following example shows typical use of **FormLayout** in a form:

```
FormLayout form = new FormLayout();
TextField tf1 = new TextField("Name");
tf1.setIcon(FontAwesome.USER);
tf1.setRequired(true);
tf1.addValidator(new NullValidator("Must be given", false));
form.addComponent(tf1);

TextField tf2 = new TextField("Street address");
tf2.setIcon(FontAwesome.ROAD);
form.addComponent(tf2);

TextField tf3 = new TextField("Postal code");
tf3.setIcon(FontAwesome.ENVELOPE);
tf3.addValidator(new IntegerRangeValidator("Doh!", 1, 99999));
form.addComponent(tf3);
```

The resulting layout will look as follows. The error message shows in a tooltip when you hover the mouse pointer over the error indicator.

Figure 7.6. A FormLayout Layout for Forms

7.5.1. CSS Style Rules

```
.v-formlayout {}
.v-formlayout .v-caption {}

/* Columns in a field row. */
.v-formlayout-contentcell {} /* Field content. */
.v-formlayout-captioncell {} /* Field caption. */
.v-formlayout-errorcell {} /* Field error indicator. */

/* Overall style of field rows. */
.v-formlayout-row {}
.v-formlayout-firstrow {}
.v-formlayout-lastrow {}

/* Required field indicator. */
.v-formlayout .v-required-field-indicator {}
.v-formlayout-captioncell .v-caption
    .v-required-field-indicator {}

/* Error indicator. */
.v-formlayout-cell .v-errorindicator {}
.v-formlayout-error-indicator .v-errorindicator {}
```

The top-level element of **FormLayout** has the `v-formlayout` style. The layout is tabular with three columns: the caption column, the error indicator column, and the field column. These can be styled with `v-formlayout-captioncell`, `v-formlayout-errorcell`, and `v-formlayout-contentcell`, respectively. While the error indicator is shown as a dedicated column, the indicator for required fields is currently shown as a part of the caption column.

For information on setting margins and spacing, see also Section 7.3.2, “Spacing in Ordered Layouts” and Section 7.13.5, “Layout Margins”.

7.6. Panel

Panel is a single-component container with a frame around the content. It has an optional caption and an icon which are handled by the panel itself, not its containing layout. The panel itself does not manage the caption of its contained component. You need to set the content with `setContent()`.

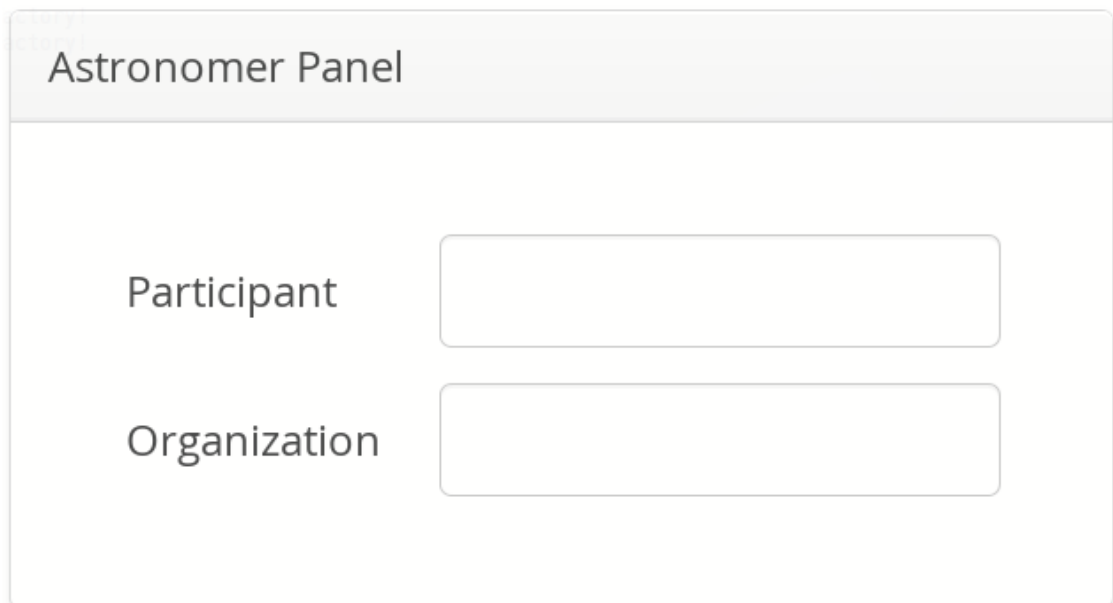
Panel has 100% width and undefined height by default. This corresponds with the default sizing of **VerticalLayout**, which is perhaps most commonly used as the content of a **Panel**. If the width or height of a panel is undefined, the content must have a corresponding undefined or fixed size in the same direction to avoid a sizing paradox.

```
Panel panel = new Panel("Astronomer Panel");
panel.addStyleName("mypanelexample");
panel.setSizeUndefined(); // Shrink to fit content
layout.addComponent(panel);

// Create the content
FormLayout content = new FormLayout();
content.addStyleName("mypanelcontent");
content.addComponent(new TextField("Participant"));
content.addComponent(new TextField("Organization"));
content.setSizeUndefined(); // Shrink to fit
content.setMargin(true);
panel.setContent(content);
```

The resulting layout is shown in Figure 7.7, “A Panel”.

Figure 7.7. A Panel



7.6.1. Scrolling the Panel Content

Normally, if a panel has undefined size in a direction, as it has by default vertically, it will fit the size of the content and grow as the content grows. However, if it has a fixed or percentual size

and its content becomes too big to fit in the content area, a scroll bar will appear for the particular direction. Scroll bars in a **Panel** are handled natively by the browser with the `overflow: auto` property in CSS.

In the following example, we have a 300 pixels wide and very high **Image** component as the panel content.

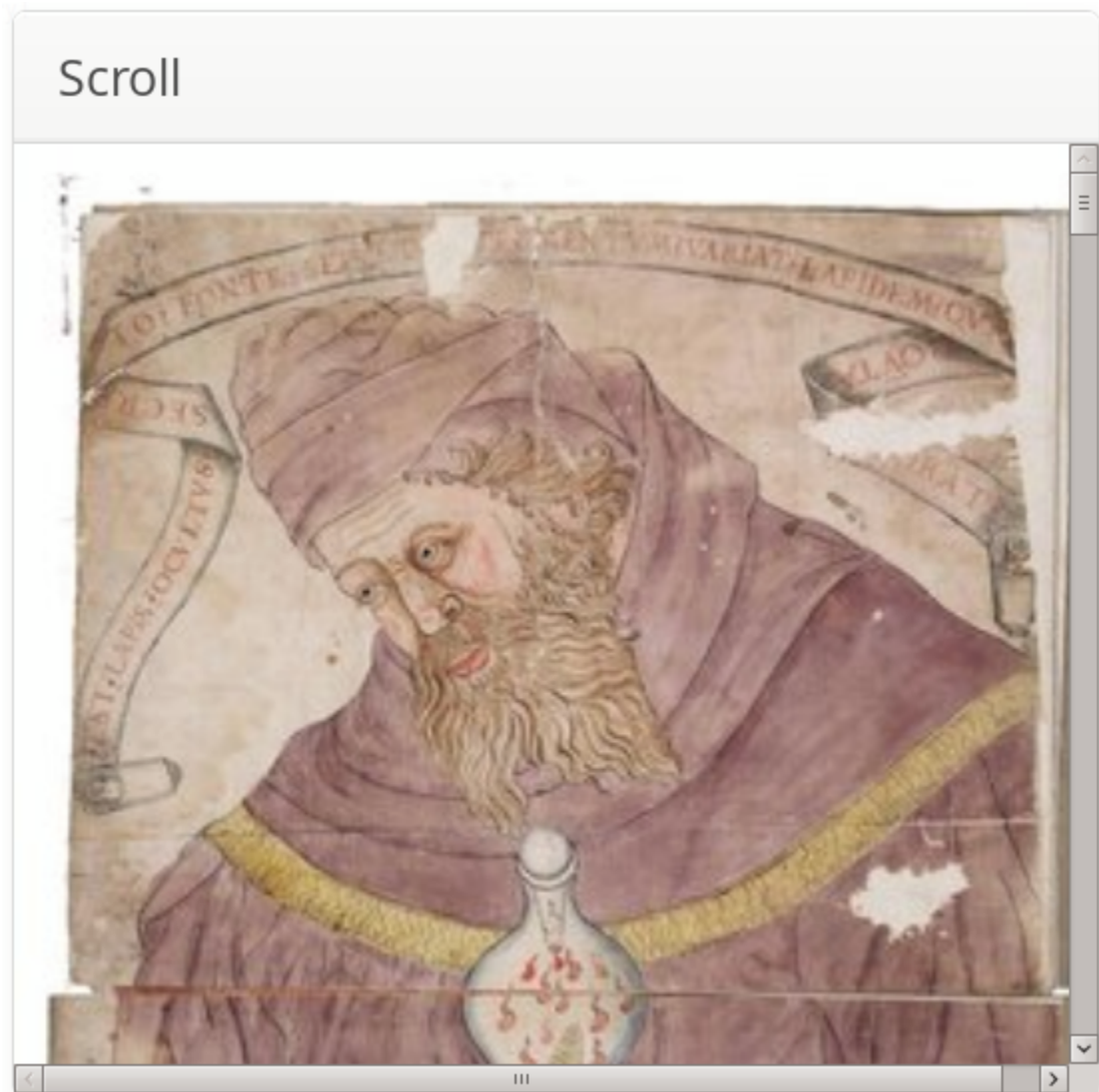
```
// Display an image stored in theme
Image image = new Image(null,
    new ThemeResource("img/Ripley_Scroll-300px.jpg"));

// To enable scrollbars, the size of the panel content
// must not be relative to the panel size
image.setSizeUndefined(); // Actually the default

// The panel will give it scrollbars.
Panel panel = new Panel("Scroll");
panel.setWidth("300px");
panel.setHeight("300px");
panel.setContent(image);

layout.addComponent(panel);
```

The result is shown in Figure 7.8, “Panel with Scroll Bars”. Notice that also the horizontal scrollbar has appeared even though the panel has the same width as the content (300 pixels) - the 300px width for the panel includes the panel border and vertical scrollbar.

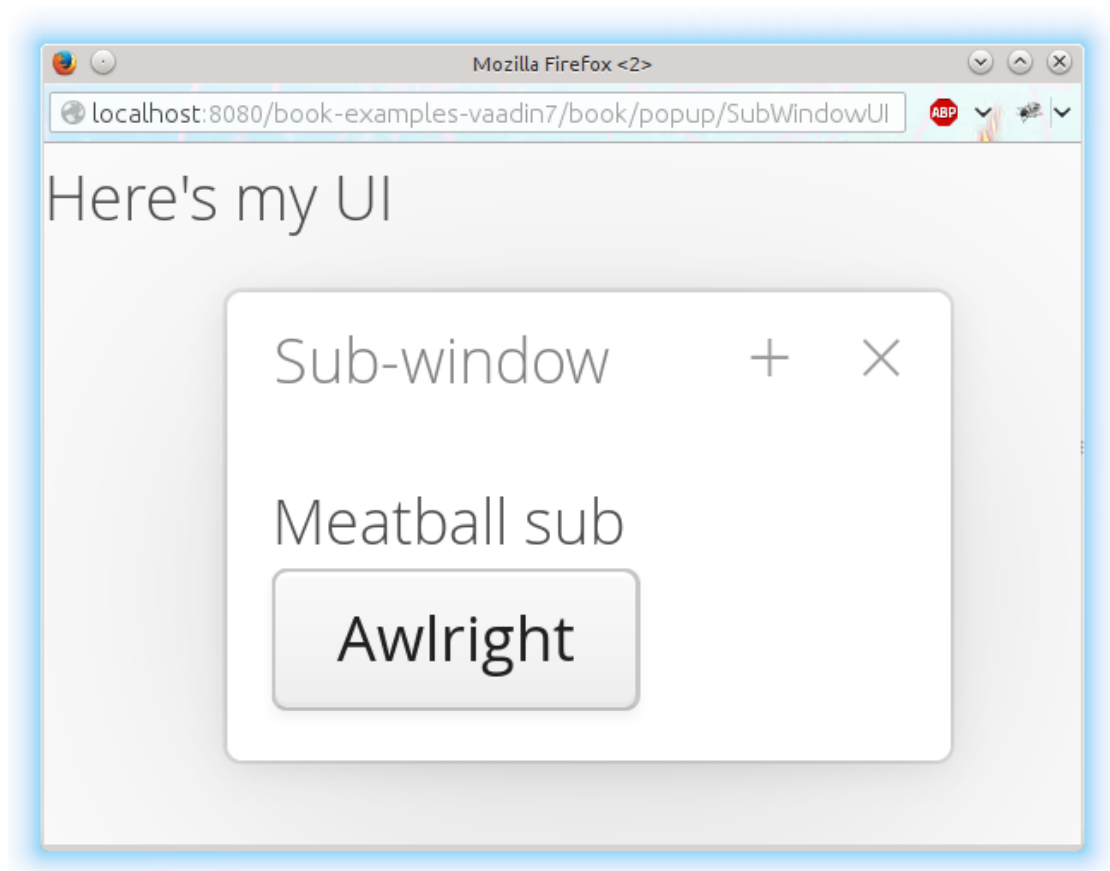
Figure 7.8. Panel with Scroll Bars

Programmatic Scrolling

Panel implements the `Scrollable` interface to allow programmatic scrolling. You can set the scroll position in pixels with `setScrollTop()` and `setScrollLeft()`. You can also get the scroll position set previously, but scrolling the panel in the browser does not update the scroll position to the server-side.

7.7. Sub-Windows

Sub-windows are floating panels within a native browser window. Unlike native browser windows, sub-windows are managed by the client-side runtime of Vaadin using HTML features. Vaadin allows opening, closing, resizing, maximizing and restoring sub-windows, as well as scrolling the window content.

Figure 7.9. A Sub-Window

Sub-windows are typically used for *Dialog Windows* and *Multiple Document Interface* applications. Sub-windows are by default not modal; you can set them modal as described in Section 7.7.4, “Modal Sub-Windows”.

7.7.1. Opening and Closing Sub-Windows

You can open a new sub-window by creating a new **Window** object and adding it to the UI with `addWindow()`, typically in some event listener. A sub-window needs a content component, which is typically a layout.

In the following, we display a sub-window immediately when a UI opens:

```
public static class SubWindowUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Some other UI content
        setContent(new Label("Here's my UI"));

        // Create a sub-window and set the content
        Window subWindow = new Window("Sub-window");
        VerticalLayout subContent = new VerticalLayout();
        subContent.setMargin(true);
        subWindow.setContent(subContent);
    }
}
```

```
// Put some components in it
subContent.addComponent(new Label("Meatball sub"));
subContent.addComponent(new Button("Awlright"));

// Center it in the browser window
subWindow.center();

// Open it in the UI
addWindow(subWindow);
}
}
```

The result was shown in Figure 7.9, “A Sub-Window”. Sub-windows by default have undefined size in both dimensions, so they will shrink to fit the content.

The user can close a sub-window by clicking the close button in the upper-right corner of the window. The button is controlled by the *closable* property, so you can disable it with `setClosable(false)`. You can also use keyboard shortcuts for closing a sub-window. You can manage the shortcuts with the `addCloseShortcut()`, `removeCloseShortcut()`, `removeAllCloseShortcuts()`, `hasCloseShortcut()` and `getCloseShortcuts()` methods.

You close a sub-window also programmatically by calling the `close()` for the sub-window, typically in a click listener for an **OK** or **Cancel** button. You can also call `removeWindow()` for the current **UI**.

7.7.2. Window Positioning

When created, a sub-window will have an undefined default size and position. You can specify the size of a window with `setHeight()` and `setWidth()` methods. You can set the position of the window with `setPositionX()` and `setPositionY()` methods.

```
// Create a new sub-window
mywindow = new Window("My Dialog");

// Set window size.
mywindow.setHeight("200px");
mywindow.setWidth("400px");

// Set window position.
mywindow.setPositionX(200);
mywindow.setPositionY(50);

UI.getCurrent().addWindow(mywindow);
```

7.7.3. Scrolling Sub-Window Content

If a sub-window has a fixed or percentual size and its content becomes too big to fit in the content area, a scroll bar will appear for the particular direction. On the other hand, if the sub-window has undefined size in the direction, it will fit the size of the content and never get a scroll bar. Scroll bars in sub-windows are handled with regular HTML features, namely `overflow: auto` property in CSS.

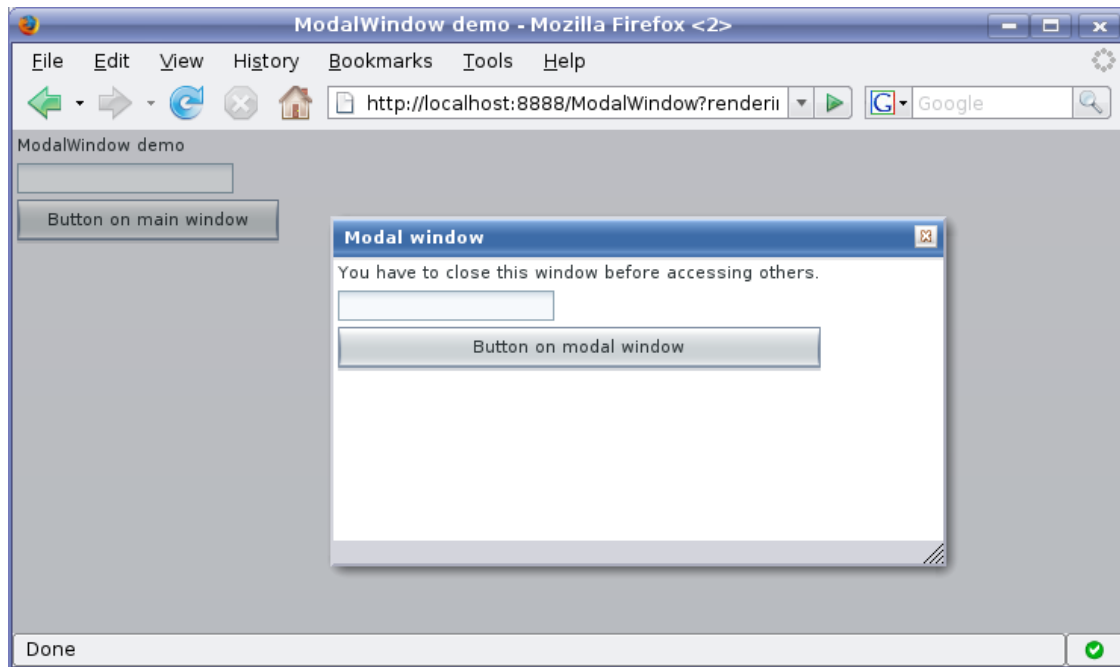
As **Window** extends **Panel**, windows are also `Scrollable`. Note that the interface defines *programmatic scrolling*, not scrolling by the user. Please see Section 7.6, “**Panel**”.

7.7.4. Modal Sub-Windows

A modal window is a sub-window that prevents interaction with the other UI. Dialog windows, as illustrated in Figure 7.10, “Modal Sub-Window”, are typical cases of modal windows. The advantage of modal windows is limiting the scope of user interaction to a sub-task, so changes in application state are more limited. The disadvantage of modal windows is that they can restrict workflow too much.

You can make a sub-window modal with `setModal(true)`.

Figure 7.10. Modal Sub-Window



Depending on the theme, the parent window may be grayed when the modal window is open.



Security Warning

Modality of child windows is purely a client-side feature and can be circumvented with client-side attack code. You should not trust in the modality of child windows in security-critical situations such as login windows.

7.8. HorizontalSplitPanel and VerticalSplitPanel

HorizontalSplitPanel and **VerticalSplitPanel** are a two-component containers that divide the available space into two areas to accommodate the two components. **HorizontalSplitPanel** makes the split horizontally with a vertical splitter bar, and **VerticalSplitPanel** vertically with a horizontal splitter bar. The user can drag the bar to adjust its position.

You can set the two components with the `setFirstComponent()` and `setSecondComponent()` methods, or with the regular `addComponent()` method.

```
// Have a panel to put stuff in
Panel panel = new Panel("Split Panels Inside This Panel");
```

```
// Have a horizontal split panel as its content
HorizontalSplitPanel hsplit = new HorizontalSplitPanel();
panel.setContent(hsplit);

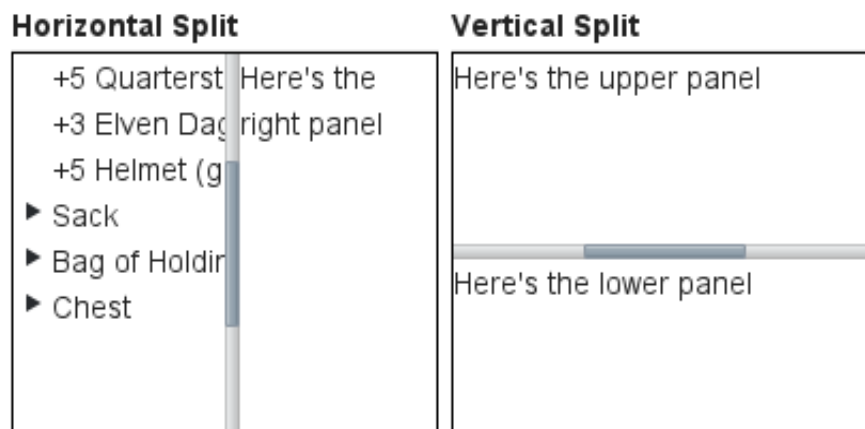
// Put a component in the left panel
Tree tree = new Tree("Menu", TreeExample.createTreeContent());
hsplit.setFirstComponent(tree);

// Put a vertical split panel in the right panel
VerticalSplitPanel vsplit = new VerticalSplitPanel();
hsplit.setSecondComponent(vsplit);

// Put other components in the right panel
vsplit.addComponent(new Label("Here's the upper panel"));
vsplit.addComponent(new Label("Here's the lower panel"));
```

The result is shown in Figure 7.11, “**HorizontalSplitPanel** and **VerticalSplitPanel**”. Observe that the tree is cut horizontally as it can not fit in the layout. If its height exceeds the height of the panel, a vertical scroll bar will appear automatically. If horizontal scroll bar is necessary, you could put the content in a **Panel**, which can have scroll bars in both directions.

Figure 7.11. HorizontalSplitPanel and VerticalSplitPanel



You can set the split position with `setSplitPosition()`. It accepts any units defined in the **Sizeable** interface, with percentual size relative to the size of the component.

```
// Have a horizontal split panel
HorizontalSplitPanel hsplit = new HorizontalSplitPanel();
hsplit.setFirstComponent(new Label("75% wide panel"));
hsplit.setSecondComponent(new Label("25% wide panel"));

// Set the position of the splitter as percentage
hsplit.setSplitPosition(75, Sizeable.UNITS_PERCENTAGE);
```

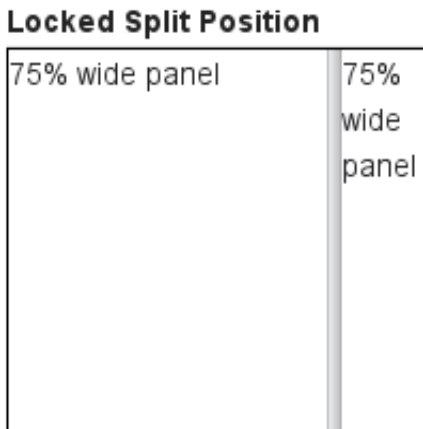
Another version of the `setSplitPosition()` method allows leaving out the unit, using the same unit as previously. The method also has versions take a boolean parameter, *reverse*, which allows defining the size of the right or bottom panel instead of the left or top panel.

The split bar allows the user to adjust the split position by dragging the bar with mouse. To lock the split bar, use `setLocked(true)`. When locked, the move handle in the middle of the bar is disabled.

```
// Lock the splitter
hsplit.setLocked(true);
```

Setting the split position programmatically and locking the split bar is illustrated in Figure 7.12, “A Layout With Nested SplitPanels”.

Figure 7.12. A Layout With Nested SplitPanels



Notice that the size of a split panel must not be undefined in the split direction.

7.8.1. CSS Style Rules

```
/* For a horizontal SplitPanel. */
.v-splitpanel-horizontal {}
.v-splitpanel-hsplitter {}
.v-splitpanel-hsplitter-locked {}

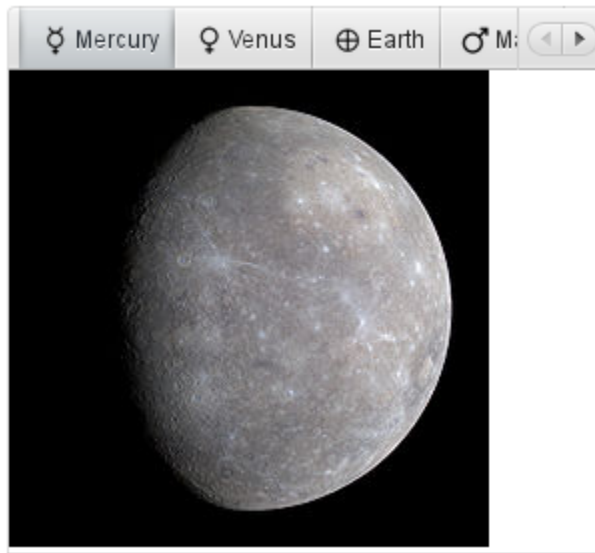
/* For a vertical SplitPanel. */
.v-splitpanel-vertical {}
.v-splitpanel-vsplitter {}
.v-splitpanel-vsplitter-locked {}

/* The two container panels. */
.v-splitpanel-first-container {} /* Top or left panel. */
.v-splitpanel-second-container {} /* Bottom or right panel. */
```

The entire split panel has the style `v-splitpanel-horizontal` or `v-splitpanel-vertical`, depending on the panel direction. The split bar or *splitter* between the two content panels has either the `...-splitter` or `...-splitter-locked` style, depending on whether its position is locked or not.

7.9. TabSheet

The **TabSheet** is a multicomponent container that allows switching between the components with “tabs”. The tabs are organized as a tab bar at the top of the tab sheet. Clicking on a tab opens its contained component in the main display area of the layout. If there are more tabs than fit in the tab bar, navigation buttons will appear.

Figure 7.13. A Simple TabSheet Layout

7.9.1. Adding Tabs

You add new tabs to a tab sheet with the `addTab()` method. The simple version of the method takes as its parameter the root component of the tab. You can use the root component to retrieve its corresponding **Tab** object. Typically, you put a layout component as the root component.

You can also give the caption and the icon as parameters for the `addTab()` method. The following example demonstrates the creation of a simple tab sheet, where each tab shows a different **Label** component. The tabs have an icon, which are (in this example) loaded as Java class loader resources from the application.

```
TabSheet tabsheet = new TabSheet();
layout.addComponent(tabsheet);

// Create the first tab
VerticalLayout tab1 = new VerticalLayout();
tab1.addComponent(new Embedded(null,
    new ThemeResource("img/planets/Mercury.jpg")));
tabsheet.addTab(tab1, "Mercury",
    new ThemeResource("img/planets/Mercury_symbol.png"));

// This tab gets its caption from the component caption
VerticalLayout tab2 = new VerticalLayout();
tab2.addComponent(new Embedded(null,
    new ThemeResource("img/planets/Venus.jpg")));
tab2.setCaption("Venus");
tabsheet.addTab(tab2).setIcon(
    new ThemeResource("img/planets/Venus_symbol.png"));
...
```

7.9.2. Tab Objects

Each tab in a tab sheet is represented as a **Tab** object, which manages the tab caption, icon, and attributes such as hidden and visible. You can set the caption with `setCaption()` and the icon with `setIcon()`. If the component added with `addTab()` has a caption or icon, it is

used as the default for the **Tab** object. However, changing the attributes of the root component later does not affect the tab, but you must make the setting through the **Tab** object. The `addTab()` returns the new **Tab** object, so you can easily set an attribute using the reference.

```
// Set an attribute using the returned reference
tabsheet.addTab(myTab).setCaption("My Tab");
```

Disabling and Hiding Tabs

A tab can be disabled by setting `setEnabled(false)` for the **Tab** object, thereby disallowing selecting it.

A tab can be made invisible by setting `setVisible(false)` for the **Tab** object. The `hideTabs()` method allows hiding the tab bar entirely. This can be useful in tabbed document interfaces (TDI) when there is only one tab.

7.9.3. Tab Change Events

Clicking on a tab selects it. This fires a **TabSheet.SelectedTabChangeEvent**, which you can handle by implementing the **TabSheet.SelectedTabChangeListener** interface. You can access the `tabsheet` of the event with `getTabSheet()`, and find the new selected tab with `getSelectedTab()`.

You can programmatically select a tab with `setSelectedTab()`, which also fires the **SelectedTabChangeEvent** (beware of recursive events). Reselecting the currently selected tab does not fire the event.

Notice that when the first tab is added, it is selected and the change event is fired, so if you want to catch that, you need to add your listener before adding any tabs.

7.9.4. Enabling and Handling Closing Tabs

You can enable a close button for individual tabs with the `closable` property in the **TabSheet.Tab** objects.

```
// Enable closing the tab
tabsheet.getTab(tabComponent).setClosable(true);
```

Figure 7.14. TabSheet with Closable Tabs



Handling Tab Close Events

You can handle closing tabs by implementing a custom **TabSheet.CloseHandler**. The default implementation simply calls `removeTab()` for the tab to be closed, but you can prevent the close by not calling it. This allows, for example, opening a dialog window to confirm the close.

```
tabsheet.setCloseHandler(new CloseHandler() {
    @Override
```



```

    public void onTabClose(TabSheet tabsheet,
                          Component tabContent) {
        Tab tab = tabsheet.getTab(tabContent);
        Notification.show("Closing " + tab.getCaption());

        // We need to close it explicitly in the handler
        tabsheet.removeTab(tab);
    }
});

```

7.10. Accordion

Accordion is a multicomponent container similar to **TabSheet**, except that the "tabs" are arranged vertically. Clicking on a tab opens its contained component in the space between the tab and the next one. You can use an **Accordion** identically to a **TabSheet**, which it actually inherits. See Section 7.9, "**TabSheet**" for more information.

The following example shows how you can create a simple accordion. As the **Accordion** is rather naked alone, we put it inside a Panel that acts as its caption and provides it a border.

```

// Create the accordion
Accordion accordion = new Accordion();

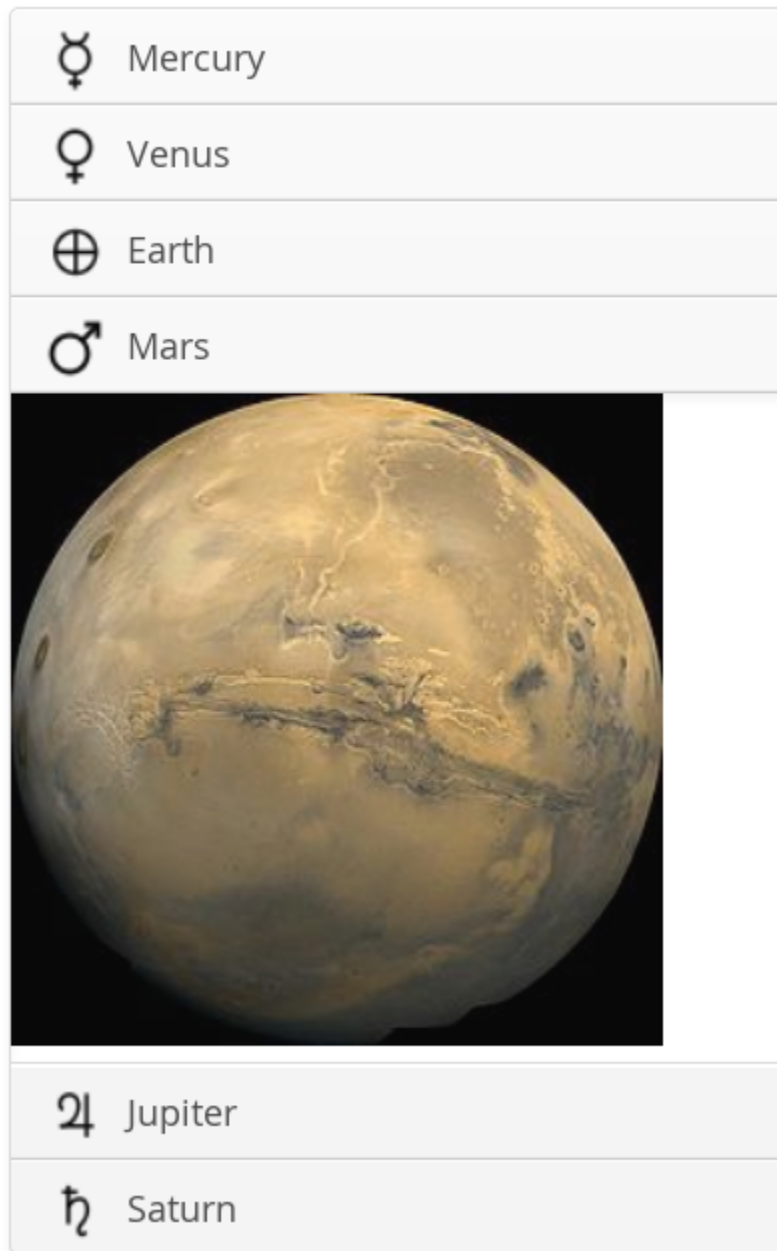
// Create the first tab, specify caption when adding
Layout tab1 = new VerticalLayout(); // Wrap in a layout
tab1.addComponent(new Image(null, // No component caption
    new ThemeResource("img/planets/Mercury.jpg")));
accordion.addTab(tab1, "Mercury",
    new ThemeResource("img/planets/Mercury_symbol.png"));

// This tab gets its caption from the component caption
Component tab2 = new Image("Venus",
    new ThemeResource("img/planets/Venus.jpg"));
accordion.addTab(tab2).setIcon(
    new ThemeResource("img/planets/Venus_symbol.png"));

// And so forth the other tabs...
String[] tabs = {"Earth", "Mars", "Jupiter", "Saturn"};
for (String caption: tabs) {
    String basename = "img/planets/" + caption;
    VerticalLayout tab = new VerticalLayout();
    tab.addComponent(new Embedded(null,
        new ThemeResource(basename + ".jpg")));
    accordion.addTab(tab, caption,
        new ThemeResource(basename + "_symbol.png"));
}

```

Figure 7.15, "An Accordion" shows what the example would look like with the default theme.

Figure 7.15. An Accordion

7.10.1. CSS Style Rules

```
.v-accordion {}  
.v-accordion-item,  
.v-accordion-item-open,  
.v-accordion-item-first {}  
.v-accordion-item-caption {}  
.v-caption {}  
.v-accordion-item-content {}  
.v-scrollable {}
```

The top-level element of **Accordion** has the `v-accordion` style. An **Accordion** consists of a sequence of item elements, each of which has a caption element (the tab) and a content area element.

The selected item (tab) has also the `v-accordion-open` style. The content area is not shown for the closed items.

7.11. **AbsoluteLayout**

AbsoluteLayout allows placing components in arbitrary positions in the layout area. The positions are specified in the `addComponent()` method with horizontal and vertical coordinates relative to an edge of the layout area. The positions can include a third depth dimension, the *z-index*, which specifies which components are displayed in front and which behind other components.

The positions are specified by a CSS absolute position string, using the `left`, `right`, `top`, `bottom`, and *z-index* properties known from CSS. In the following example, we have a 300 by 150 pixels large layout and position a text field 50 pixels from both the left and the top edge:

```
// A 400x250 pixels size layout
AbsoluteLayout layout = new AbsoluteLayout();
layout.setWidth("400px");
layout.setHeight("250px");

// A component with coordinates for its top-left corner
TextField text = new TextField("Somewhere someplace");
layout.addComponent(text, "left: 50px; top: 50px;");
```

The `left` and `top` specify the distance from the left and top edge, respectively. The `right` and `bottom` specify the distances from the right and top edge.

```
// At the top-left corner
Button button = new Button("left: 0px; top: 0px;");
layout.addComponent(button, "left: 0px; top: 0px;");

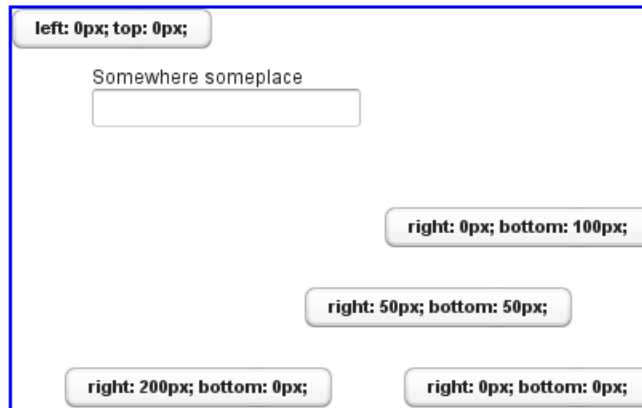
// At the bottom-right corner
Button buttCorner = new Button("right: 0px; bottom: 0px;");
layout.addComponent(buttCorner, "right: 0px; bottom: 0px;");

// Relative to the bottom-right corner
Button buttBrRelative = new Button("right: 50px; bottom: 50px;");
layout.addComponent(buttBrRelative, "right: 50px; bottom: 50px;");

// On the bottom, relative to the left side
Button buttBottom = new Button("left: 50px; bottom: 0px;");
layout.addComponent(buttBottom, "left: 50px; bottom: 0px;");

// On the right side, up from the bottom
Button buttRight = new Button("right: 0px; bottom: 100px;");
layout.addComponent(buttRight, "right: 0px; bottom: 100px;");
```

The result of the above code examples is shown in Figure 7.16, “Components Positioned Relative to Various Edges”.

Figure 7.16. Components Positioned Relative to Various Edges

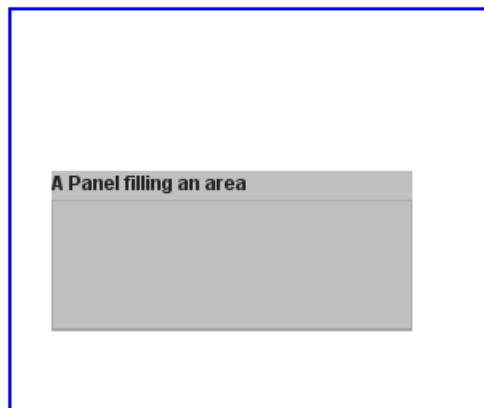
Drag and drop is very useful for moving the components contained in an **AbsoluteLayout**. Check out the example in Section 12.12.6, “Dropping on a Component”.

7.11.1. Placing a Component in an Area

Earlier, we had components of undefined size and specified the positions of components by a single pair of coordinates. The other possibility is to specify an area and let the component fill the area by specifying a proportional size for the component, such as "100%". Normally, you use `setSizeFull()` to take the entire area given by the layout.

```
// Specify an area that a component should fill
Panel panel = new Panel("A Panel filling an area");
panel.setSizeFull(); // Fill the entire given area
layout.addComponent(panel, "left: 25px; right: 50px; "+
    "top: 100px; bottom: 50px;");
```

The result is shown in Figure 7.17, “Component Filling an Area Specified by Coordinates”

Figure 7.17. Component Filling an Area Specified by Coordinates

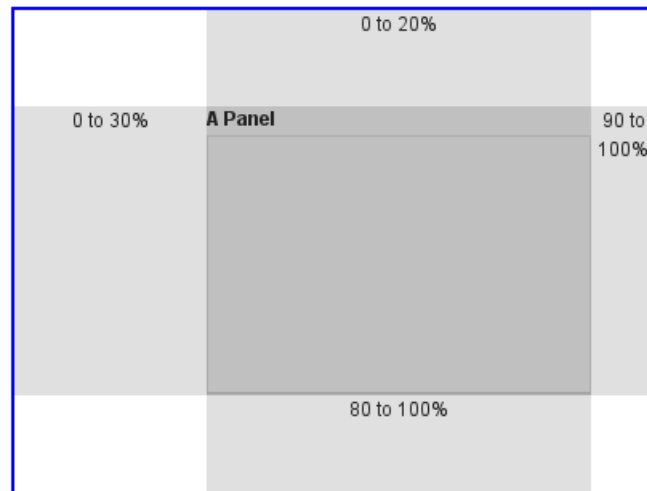
7.11.2. Proportional Coordinates

You can also use proportional coordinates to specify the placement of components:

```
// A panel that takes 30% to 90% horizontally and
// 20% to 80% vertically
Panel panel = new Panel("A Panel");
panel.setSizeFull(); // Fill the specified area
layout.addComponent(panel, "left: 30%; right: 10%;" +
    "top: 20%; bottom: 20%;");
```

The result is shown in Figure 7.18, “Specifying an Area by Proportional Coordinates”

Figure 7.18. Specifying an Area by Proportional Coordinates



7.11.3. Styling with CSS

```
.v-absolutelayout {}
.v-absolutelayout-wrapper {}
```

The **AbsoluteLayout** component has `v-absolutelayout` root style. Each component in the layout is contained within an element that has the `v-absolutelayout-wrapper`. The component captions are outside the wrapper elements, in a separate element with the usual `v-caption` style.

7.12. CssLayout

CssLayout allows strong control over styling of the components contained inside the layout. The components are contained in a simple DOM structure consisting of `<div>` elements. By default, the contained components are laid out horizontally and wrap naturally when they reach the width of the layout, but you can control this and most other behaviour with CSS. You can also inject custom CSS for each contained component. As **CssLayout** has a very simple DOM structure and no dynamic rendering logic, relying purely on the built-in rendering logic of the browsers, it is the fastest of the layout components.

The basic use of **CssLayout** is just like with any other layout component:

```
CssLayout layout = new CssLayout();

// Component with a layout-managed caption and icon
TextField tf = new TextField("A TextField");
tf.setIcon(new ThemeResource("icons/user.png"));
layout.addComponent(tf);
```

```
// Labels are 100% wide by default so must unset width
Label label = new Label("A Label");
label.setWidth(Sizeable.SIZE_UNDEFINED, 0);
layout.addComponent(label);
```

```
layout.addComponent(new Button("A Button"));
```

The result is shown in Figure 7.19, “Basic Use of **CssLayout**”. Notice that the default spacing and alignment of the layout is quite crude and CSS styling is nearly always needed.

Figure 7.19. Basic Use of **CssLayout**



The `display` attribute of **CssLayout** is inline-block by default, so the components are laid out horizontally following another. **CssLayout** has 100% width by default. If the components reach the width of the layout, they are wrapped to the next "line" just as text would be. If you add a component with 100% width, it will take an entire line by wrapping before and after the component.

7.12.1. CSS Injection

Overriding the `getCss()` method allows injecting custom CSS for each component. The CSS returned by the method is inserted in the `style` attribute of the `<div>` element of the component, so it will override any style definitions made in CSS files.

```
CssLayout layout = new CssLayout() {
    @Override
    protected String getCss(Component c) {
        if (c instanceof Label) {
            // Color the boxes with random colors
            int rgb = (int) (Math.random()*(1<<24));
            return "background: #" + Integer.toHexString(rgb);
        }
        return null;
    }
};
layout.setWidth("400px"); // Causes to wrap the contents

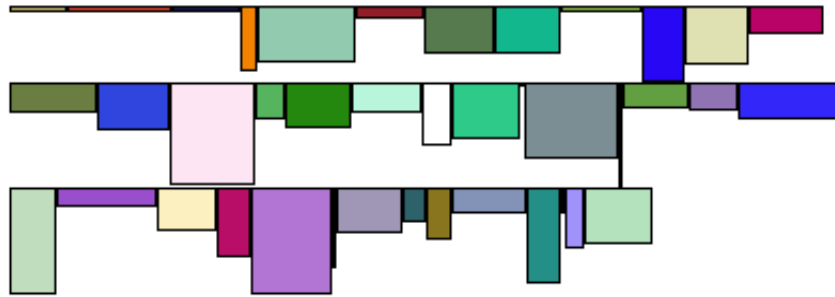
// Add boxes of various sizes
for (int i=0; i<40; i++) {
    Label box = new Label("&nbsp;", ContentMode.HTML);
    box.addStyleName("flowbox");
    box.setWidth((float) Math.random()*50,
        Sizeable.UNITS_PIXELS);
    box.setHeight((float) Math.random()*50,
        Sizeable.UNITS_PIXELS);
    layout.addComponent(box);
}
```

The style name added to the components allows making common styling in a CSS file:

```
.v-label-flowbox {
    border: thin black solid;
}
```

Figure 7.20, “Use of `getCss()` and line wrap” shows the rendered result.

Figure 7.20. Use of `getCss()` and line wrap



7.12.2. Browser Compatibility

The strength of the **CssLayout** is also its weakness. Much of the logic behind the other layout components is there to give nice default behaviour and to handle the differences in different browsers. Some browsers, no need to say which, are notoriously incompatible with the CSS standards, so they require a lot of custom CSS. You may need to make use of the browser-specific style classes in the root element of the application. Some features in the other layouts are not even solvable in pure CSS, at least in all browsers.

7.12.3. Styling with CSS

```
.v-csslayout {}
.v-csslayout-margin {}
.v-csslayout-container {}
```

The **CssLayout** component has `v-csslayout` root style. The margin element with `v-csslayout-margin` style is always enabled. The components are contained in an element with `v-csslayout-container` style.

For example, we could style the basic **CssLayout** example shown earlier as follows:

```
/* Have the caption right of the text box, bottom-aligned */
.csslayoutexample .mylayout .v-csslayout-container {
    direction: rtl;
    line-height: 24px;
    vertical-align: bottom;
}

/* Have some space before and after the caption */
.csslayoutexample .mylayout .v-csslayout-container .v-caption {
    padding-left: 3px;
    padding-right: 10px;
}
```

The example would now be rendered as shown in Figure 7.21, “Styling **CssLayout**”.

Figure 7.21. Styling **CssLayout**



Captions and icons that are managed by the layout are contained in an element with `v-caption` style. These caption elements are contained flat at the same level as the actual component elements. This may cause problems with wrapping in `inline-block` mode, as wrapping can occur between the caption and its corresponding component element just as well as between components. Such use case is therefore not feasible.

7.13. Layout Formatting

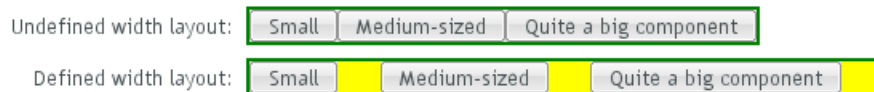
While the formatting of layouts is mainly done with style sheets, just as with other components, style sheets are not ideal or even possible to use in some situations. For example, CSS does not allow defining the spacing of table cells, which is done with the `cellspacing` attribute in HTML.

Moreover, as many layout sizes are calculated dynamically in the Client-Side Engine of Vaadin, some CSS settings can fail altogether.

7.13.1. Layout Size

The size of a layout component can be specified with the `setWidth()` and `setHeight()` methods defined in the **Sizeable** interface, just like for any component. It can also be undefined, in which case the layout shrinks to fit the component(s) inside it. Section 6.3.9, “Sizing Components” gives details on the interface.

Figure 7.22. HorizontalLayout with Undefined vs Defined size



Many layout components take 100% width by default, while they have the height undefined.

The sizes of components inside a layout can also be defined as a percentage of the space available in the layout, for example with `setWidth("100%")`; or with the (most commonly used method) `setFullSize()` that sets 100% size in both directions. If you use a percentage in a **HorizontalLayout**, **VerticalLayout**, or **GridLayout**, you will also have to set the component as *expanding*, as noted below.



Warning

A layout that contains components with percentual size must have a defined size!

If a layout has undefined size and a contained component has, say, 100% size, the component will try to fill the space given by the layout, while the layout will shrink to fit the space taken by the component, which is a paradox. This requirement holds for height and width separately. The debug mode allows detecting such invalid cases; see Section 12.3.5, “Inspecting Component Hierarchy”.

For example:

```
// This takes 100% width but has undefined height.
VerticalLayout layout = new VerticalLayout();

// A button that takes all the space available in the layout.
Button button = new Button("100%x100% button");
```



```
button.setSizeFull();
layout.addComponent(button);

// We must set the layout to a defined height vertically, in
// this case 100% of its parent layout, which also must
// not have undefined size.
layout.setHeight("100%");
```

If you have a layout with undefined height, such as **VerticalLayout**, in a **UI**, **Window**, or **Panel**, and put enough content in it so that it extends outside the bottom of the view area, scrollbars will appear. If you want your application to use all the browser view, nothing more or less, you should use `setFullSize()` for the root layout.

```
// Create the UI content
VerticalLayout content = new VerticalLayout();

// Use entire view area
content.setSizeFull();

setContent(content);
```

7.13.2. Expanding Components

If you set a **HorizontalLayout** to a defined size horizontally or a **VerticalLayout** vertically, and there is space left over from the contained components, the extra space is distributed equally between the component cells. The components are aligned within these cells, according to their alignment setting, top left by default, as in the example below.

Often, you don't want such empty space, but want one or more components to take all the leftover space. You need to set such a component to 100% size and use `setExpandRatio()`. If there is just one such expanding component in the layout, the ratio parameter is irrelevant.

If you set multiple components as expanding, the expand ratio dictates how large proportion of the available space (overall or excess depending on whether the components are sized as a percentage or not) each component takes. In the example below, the buttons have 1:2:3 ratio for the expansion.

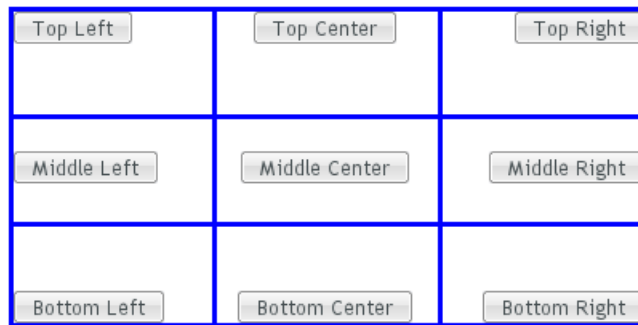
GridLayout has corresponding method for both of its directions, `setRowExpandRatio()` and `setColumnExpandRatio()`.

Expansion is dealt in detail in the documentation of the layout components that support it. See Section 7.3, “**VerticalLayout** and **HorizontalLayout**” and Section 7.4, “**GridLayout**” for details on components with relative sizes.

7.13.3. Layout Cell Alignment

When a component in a layout cell has size (width or height) smaller than the size of the cell, it will by default be aligned in the top-left corner of the cell. You can set the alignment with the `setComponentAlignment()` method. The method takes as its parameters the component contained in the cell to be formatted, and the horizontal and vertical alignment. The component must have been added to the layout before setting the alignment.

Figure 7.23, “Cell Alignments” illustrates the alignment of components within a **GridLayout**.

Figure 7.23. Cell Alignments

Note that *a component with 100% relative size can not be aligned*, as it will take the entire width or height of the cell, in which case alignment is meaningless. This should especially be noted with the **Label** component, which has 100% default width, and the text alignment *within* the component is separate, defined by the CSS `text-align` property.

The easiest way to set alignments is to use the constants defined in the **Alignment** class. Let us look how the buttons in the top row of the above **GridLayout** are aligned with constants:

```
// Create a grid layout
GridLayout grid = new GridLayout(3, 3);

grid.setWidth(400, Sizeable.UNITS_PIXELS);
grid.setHeight(200, Sizeable.UNITS_PIXELS);

Button topleft = new Button("Top Left");
grid.addComponent(topleft, 0, 0);
grid.setComponentAlignment(topleft, Alignment.TOP_LEFT);

Button topcenter = new Button("Top Center");
grid.addComponent(topcenter, 1, 0);
grid.setComponentAlignment(topcenter, Alignment.TOP_CENTER);

Button topright = new Button("Top Right");
grid.addComponent(topright, 2, 0);
grid.setComponentAlignment(topright, Alignment.TOP_RIGHT);
...
```

The following table lists all the **Alignment** constants by their respective locations:

Table 7.2. Alignment Constants

<code>TOP_LEFT</code>	<code>TOP_CENTER</code>	<code>TOP_RIGHT</code>
<code>MIDDLE_LEFT</code>	<code>MIDDLE_CENTER</code>	<code>MIDDLE_RIGHT</code>
<code>BOTTOM_LEFT</code>	<code>BOTTOM_CENTER</code>	<code>BOTTOM_RIGHT</code>

Another way to specify the alignments is to create an **Alignment** object and specify the horizontal and vertical alignment with separate constants. You can specify either of the directions, in which case the other alignment direction is not modified, or both with a bitmask operation between the two directions.

```
Button middleleft = new Button("Middle Left");
grid.addComponent(middleleft, 0, 1);
```

```
grid.setComponentAlignment(middleleft,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_LEFT));

Button middlecenter = new Button("Middle Center");
grid.addComponent(middlecenter, 1, 1);
grid.setComponentAlignment(middlecenter,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_HORIZONTAL_CENTER));

Button middleright = new Button("Middle Right");
grid.addComponent(middleright, 2, 1);
grid.setComponentAlignment(middleright,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_RIGHT));
```

Obviously, you may combine only one vertical bitmask with one horizontal bitmask, though you may leave either one out. The following table lists the available alignment bitmask constants:

Table 7.3. Alignment Bitmasks

Horizontal	<i>Bits.ALIGNMENT_LEFT</i>
<i>Bits.ALIGNMENT_HORIZONTAL_CENTER</i>	<i>Bits.ALIGNMENT_RIGHT</i>
Vertical	<i>Bits.ALIGNMENT_TOP</i>
<i>Bits.ALIGNMENT_VERTICAL_CENTER</i>	<i>Bits.ALIGNMENT_BOTTOM</i>

You can determine the current alignment of a component with `getComponentAlignment()`, which returns an **Alignment** object. The class provides a number of getter methods for decoding the alignment, which you can also get as a bitmask value.

Size of Aligned Components

You can only align a component that is smaller than its containing cell in the direction of alignment. If a component has 100% width, as many components have by default, horizontal alignment does not have any effect. For example, **Label** is 100% wide by default and can not therefore be horizontally aligned as such. The problem can be hard to notice, as the text inside a **Label** is left-aligned.

You usually need to set either a fixed size, undefined size, or less than a 100% relative size for the component to be aligned - a size that is smaller than the containing layout has.

For example, assuming that a **Label** has short content that is less wide than the containing **VerticalLayout**, you could center it as follows:

```
VerticalLayout layout = new VerticalLayout(); // 100% default width
Label label = new Label("Hello"); // 100% default width
label.setSizeUndefined();
layout.addComponent(label);
layout.setComponentAlignment(label, Alignment.MIDDLE_CENTER);
```

If you set the size as undefined and the component itself contains components, make sure that the contained components also have either undefined or fixed size. For example, if you set the size of a **Form** as undefined, its containing layout **FormLayout** has 100% default width, which you also need to set as undefined. But then, any components inside the **FormLayout** must have either undefined or fixed size.

7.13.4. Layout Cell Spacing

The **VerticalLayout**, **HorizontalLayout**, and **GridLayout** layouts offer a `setSpacing()` method to enable spacing between the cells of the layout.

For example:

```
VerticalLayout layout = new VerticalLayout();
layout.setSpacing(true);
layout.addComponent(new Button("Component 1"));
layout.addComponent(new Button("Component 2"));
layout.addComponent(new Button("Component 3"));
```

The effect of spacing in **VerticalLayout** and **HorizontalLayout** is illustrated in Figure 7.24, “Layout Spacings”.

Figure 7.24. Layout Spacings

		No spacing:	Vertical spacing:
No spacing:	Component 1 Component 2 Component 3	Component 1 Component 2 Component 3	Component 1 Component 2 Component 3
Horizontal spacing:	Component 1 Component 2 Component 3		

The exact amount of spacing is defined in CSS. If the default is not suitable, you can customize it in a custom theme.

In the Valo theme, you can specify the spacing with the `$v-layout-spacing-vertical` and `$v-layout-spacing-horizontal` parameters, as described in Section 9.7.2, “Common Settings”. The spacing defaults to the `$v-unit-size` measure.

When adjusting spacing in other themes, you should note that it is implemented in a bit different ways in different layouts. In the ordered layouts, it is done with spacer elements, while in the **GridLayout** it has special handling. Please see the sections on the individual components for more details.

7.13.5. Layout Margins

Most layout components do not have any margin around them by default. The ordered layouts, as well as **GridLayout**, support enabling a margin with `setMargin()`. This enables CSS classes for each margin in the HTML element of the layout component.

In the Valo theme, the margin sizes default to `$v-unit-size`. You can customize them with `$v-layout-margin-top`, `right`, `bottom`, and `left`. See Section 9.7.2, “Common Settings” for a description of the parameters.

To customize the default margins in other themes, you can define each margin with the `padding` property in CSS. You may want to have a custom CSS class for the layout component to enable specific customization of the margins, as is done in the following with the `mymargins` class:

```
.mymargins.v-margin-left {padding-left: 10px;}
.mymargins.v-margin-right {padding-right: 20px;}
```

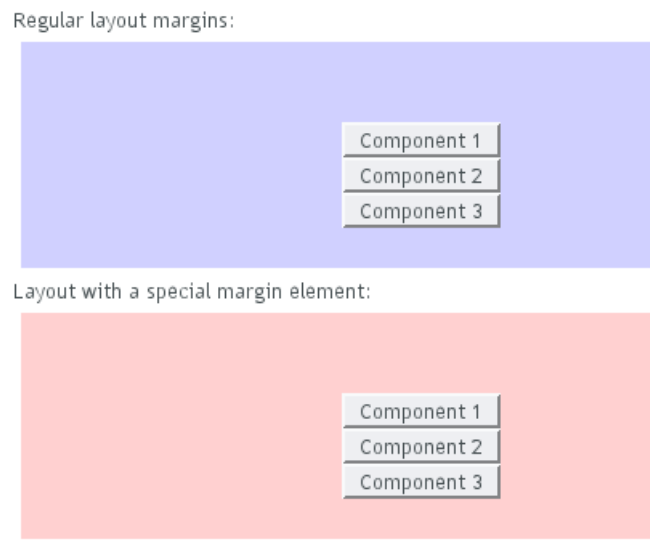
```
.mymargins.v-margin-top    {padding-top:    40px;}
.mymargins.v-margin-bottom {padding-bottom: 80px;}
```

You can enable only specific margins by passing a **MarginInfo** to the `setMargin()`. The margins are specified in clockwise order for top, right, bottom, and left margin. The following would enable the left and right margins:

```
layout.setMargin(new MarginInfo(false, true, false, true));
```

The resulting margins are shown in Figure 7.25, “Layout Margins” below. The two ways produce identical margins.

Figure 7.25. Layout Margins



7.14. Custom Layouts

While it is possible to create almost any typical layout with the standard layout components, it is sometimes best to separate the layout completely from code. With the **CustomLayout** component, you can write your layout as a template in HTML that provides locations of any contained components. The layout template is included in a theme. This separation allows the layout to be designed separately from code, for example using WYSIWYG web designer tools such as Adobe Dreamweaver.

A template is a HTML file located under `layouts` folder under a theme folder under the `WebContent/VAADIN/themes/` folder, for example, `WebContent/VAADIN/themes/themename/layouts/mylayout.html`. (Notice that the root path `WebContent/VAADIN/themes/` for themes is fixed.) A template can also be provided dynamically from an **InputStream**, as explained below. A template includes `<div>` elements with a `location` attribute that defines the location identifier. All custom layout HTML-files must be saved using UTF-8 character encoding.

```
<table width="100%" height="100%">
  <tr height="100%">
    <td>
      <table align="center">
        <tr>
```

```
        <td align="right">User&nbsp;name:</td>
        <td><div location="username"></div></td>
    </tr>
    <tr>
        <td align="right">Password:</td>
        <td><div location="password"></div></td>
    </tr>
</table>
</td>
</tr>
<tr>
    <td align="right" colspan="2">
        <div location="okbutton"></div>
    </td>
</tr>
</table>
```

The client-side engine of Vaadin will replace contents of the location elements with the components. The components are bound to the location elements by the location identifier given to `addComponent()`, as shown in the example below.

```
Panel loginPanel = new Panel("Login");
CustomLayout content = new CustomLayout("layoutname");
content.setSizeUndefined();
loginPanel.setContent(content);
loginPanel.setSizeUndefined();

// No captions for fields is they are provided in the template
content.addComponent(new TextField(), "username");
content.addComponent(new TextField(), "password");
content.addComponent(new Button("Login"), "okbutton");
```

The resulting layout is shown below in Figure 7.26, “Example of a Custom Layout Component”.

Figure 7.26. Example of a Custom Layout Component

The image shows a custom layout component for a login form. It has a light gray title bar at the top with the text "Login". Below the title bar, there are two labels: "User name:" and "Password:". To the right of each label is a rectangular input field. At the bottom right of the form is a button with the text "Login". The entire form is enclosed in a rounded rectangular border.

You can use `addComponent()` also to replace an existing component in the location given in the second parameter.

In addition to a static template file, you can provide a template dynamically with the **CustomLayout** constructor that accepts an **InputStream** as the template source. For example:

```
new CustomLayout(new ByteArrayInputStream("<b>Template</b>".getBytes()));
```

or

```
new CustomLayout(new FileInputStream(file));
```

Chapter 8

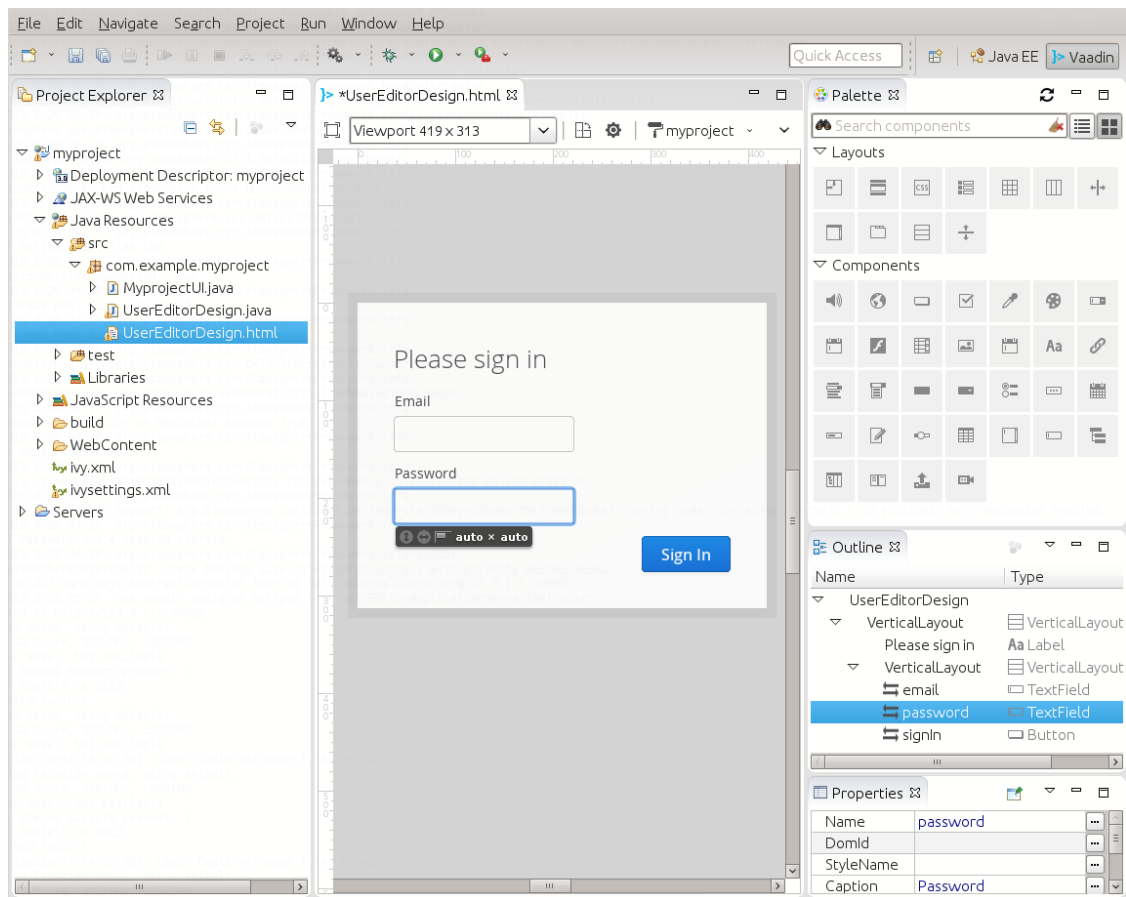
Vaadin Designer

8.1. Overview	267
8.2. Installation	269
8.3. Getting Started	269
8.4. Designing	273
8.5. Theming and Styling	281
8.6. Wiring It Up	282
8.7. Limitations	285

This chapter describes how to create designs using the Vaadin Designer.

8.1. Overview

Vaadin Designer is a visual WYSIWYG tool for creating Vaadin UIs and views by using drag&drop and direct manipulation. With features such as live external preview and a strong connection between the clean declarative format and the Java code, it allows you to design and layout your UIs with speed and confidence.

Figure 8.1. Vaadin Designer Views

Vaadin Designer is used to create two things:

1. A declarative file defining a UI (or part of a UI), also known as a *design* and
2. A *companion* Java file used to bind the UI components to Java logic.

The declarative format is a feature of the Vaadin Framework, and can be also used and edited without Vaadin Designer. See Section 5.3, “Designing UIs Declaratively” for a description of the format.

Vaadin Designer automatically creates and updates a Java file that exposes sub-components of the design as Java member variables, using variable names that you specify. This file provides the magic that creates a static binding between your design and your Java logic. It also enables Java syntax checking for using a design - if you remove from the design a component that your code needs, or change its variable name, you will get a compile-time error.

A design can be the whole UI or (more commonly) a smaller part of the UI, such as a view or its sub-component. A UI or view can contain many designs.

8.2. Installation

8.2.1. Installing Eclipse and Plug-Ins

You need to install the following to use Vaadin Designer:

1. Eclipse Luna SR2+ as described in Section 2.5, “Installing the Eclipse IDE and Plugin”
2. Vaadin Plug-in for Eclipse as described in Section 2.5.2, “Installing the Vaadin Plugin”
3. Vaadin Designer from vaadin.com/eclipse

Vaadin Designer is compatible with Eclipse Luna (and later) available from www.eclipse.org/downloads. We recommend choosing *Eclipse IDE for Java EE Developers*.

If you're using an existing install of Eclipse Luna, please make sure it is up-to-date. Eclipse Luna versions prior to the SR2 version had a nasty bug that will cause problems for several plug-ins.

Vaadin Designer is installed together with the Vaadin Plug-in for Eclipse, from the same Eclipse update-site. In Eclipse, do **Help** → **Install New Software**, press **Add...** next to the **Work with select**, enter `vaadin` as name and `http://vaadin.com/eclipse` as location.

If you already have the Vaadin plug-in installed, just choose to Work with the Vaadin update site. Make sure the whole Vaadin category is selected (or at least Vaadin Designer), then click **Next** to review licensing information and finalize the install. Please restart Eclipse when prompted.

Once installed, Vaadin Designer can be kept up-to-date by periodically running **Help** → **Check for Updates**.

8.2.2. License

The first time you start the Vaadin Designer, it will ask for a license key. You can obtain a free trial-license, purchase a stand-alone perpetual license, or use the license included with your Pro Tools subscription. See Section 18.5, “Installing Commercial Vaadin Add-on Licence” for instructions for installing the license.

Please note that a separate license key is required for each developer. If you choose not input any license (even trial), you will be unable to save your work.

If you for any reason need to remove or change a valid license, it is located in `~/.vaadin.designer.developer.license`.

8.2.3. Uninstalling

If you want to remove Vaadin Designer from your Eclipse installation, go to **Help** → **Installation Details**, select **Vaadin Designer** from the list, then click **Uninstall**.

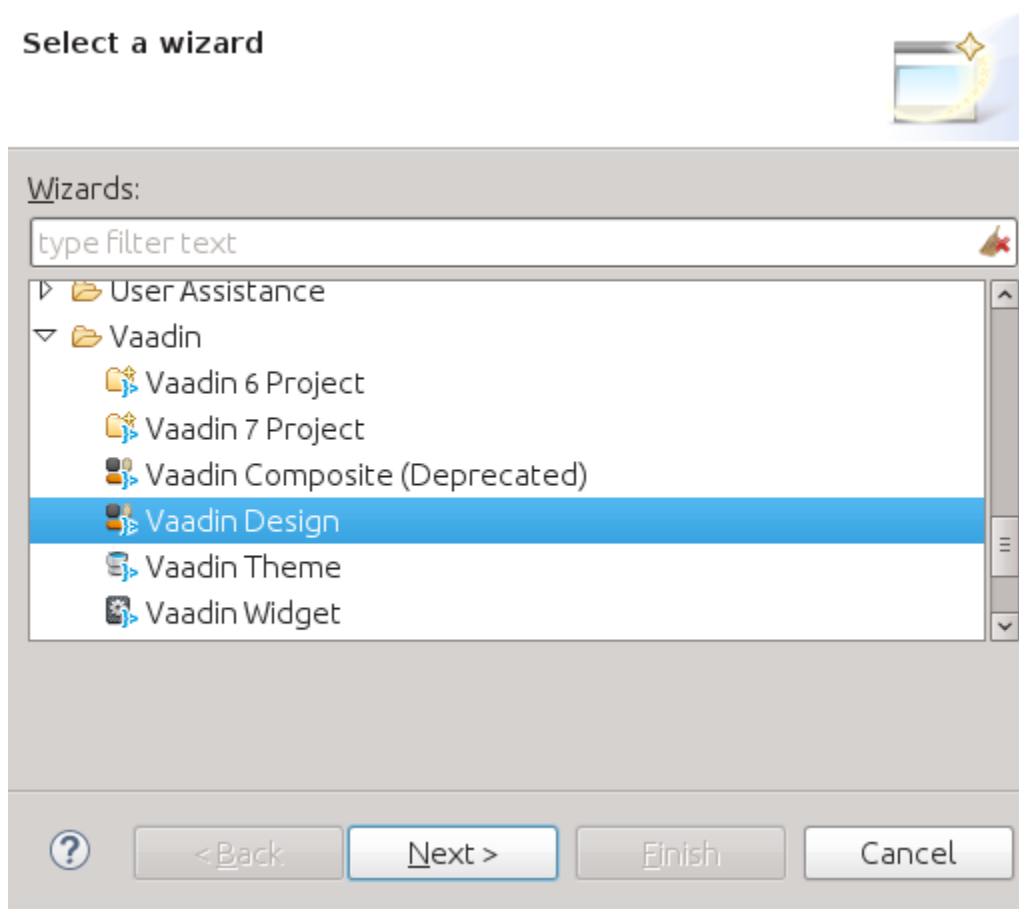
8.3. Getting Started

Vaadin Designer works projects using Vaadin 7.5 or later. In short, create a new project with **File** → **New** → **Vaadin 7 Project**, and choose 7.5 or later as Vaadin version, as described in Section 3.4, “Creating and Running a Project in Eclipse”.

8.3.1. Creating a Design

With your project selected, select **File** → **New** → **Other** (or press Ctrl+N), choose **Vaadin Design** from the list, and click **Next**.

Figure 8.2. Creating a New Vaadin Design



In the design parameters step, make sure the locations are correct (if you are using Maven they might point to different folders, otherwise probably not). You can put the design(s) in a specific package if you wish.

Figure 8.3. New Design Parameters

Vaadin Design

This wizard creates a Vaadin Design which can be edited with the Vaadin Designer.

Design location:

Java location:

Package:

Name:

Template:

Give your design a descriptive name. Using a naming convention to separate the design's companion Java file from the classes using it will make things easier for you later.

For example, the name **UserEditorDesign** will result in `UserEditorDesign.html` and `UserEditorDesign.java`. You could then create a **UserEditor** component that extends the **UserEditorDesign**, and perhaps a **UserEditorView** to place the editor component in a bigger context.

In another case, you could make a **LoginDesign** that is used in a **LoginWindow** (but not extended).

Finally, you can choose a template as starting point, or start from scratch (Blank).

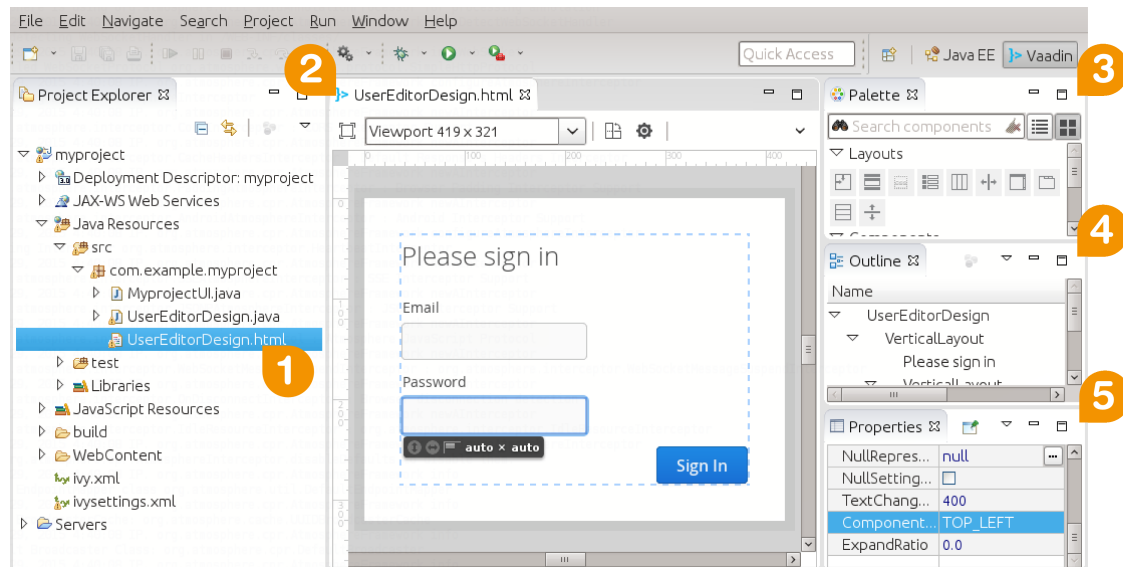
Choose **Finish** to create the design and open Vaadin Designer.

8.3.2. Vaadin Designer GUI Overview

Vaadin Designer is fully integrated with Eclipse and its views can therefore be freely moved and arranged as you wish. However, it is best to first try the Designer in its default setup by choosing **Show perspective** from the dialog that is presented when a new design is created.

To be able to successfully use the Designer, you will need the Outline, Palette and Properties views, in addition to the main editor. If you accidentally close a view, it can be opened from **Window** → **Show view**.

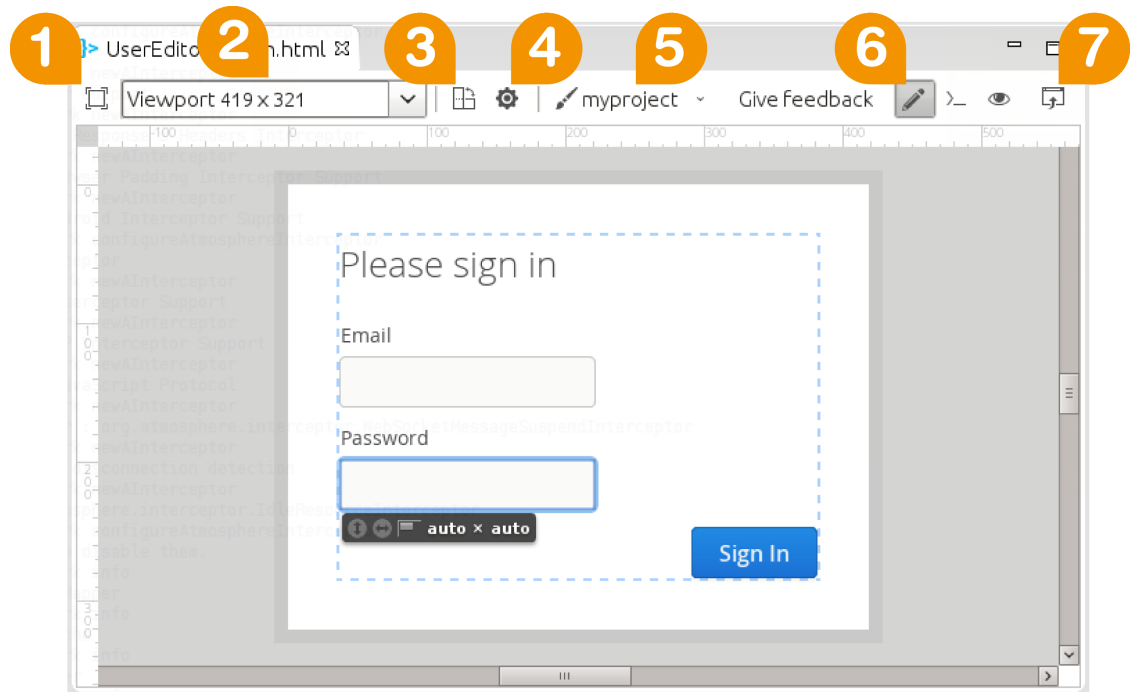
Figure 8.4. Panels in Vaadin Perspective



The elements of the perspective illustrated in Figure 8.4, “Panels in Vaadin Perspective” are as follows:

1. Design files
2. Editor (see below for close-up)
3. Component palette
4. Outline - component hierarchy
5. Properties for the selected component

In the editor view, illustrated in Figure 8.5, “Component Editor”, you have a number of controls in the toolbar.

Figure 8.5. Component Editor

1. Center viewport
2. Viewport size and presets
3. Rotate viewport (portrait / landscape)
4. Configure canvas; rulers, grids, snapping
5. Theme selector
6. Design / Code / Quick preview -modes
7. External preview

8.4. Designing

To add a component to your design, drag it from the component **Palette**, and drop it in the desired location - either in the canvas area, or in the hierarchical **Outline**. Dropping in the desired location on the canvas is the most common approach, but in some situations (especially with complex, deeply nested component hierarchies) dropping on the **Outline** might give more control.

8.4.1. About Layouts

Your designs should usually start with some sort of layout as the root component, or otherwise you are limited to a one-component design. You can also use a component that is not strictly a layout, but can still contain one (or several) components (or layouts) - this includes **TabSheet**, **Accordion**, **Panel**, etc.

There are three main types of layouts: ordered, absolute, and CSS.

Ordered layouts

Ordered layouts arrange the contained components in some ordered fashion, for instance vertically or horizontally with uniform spacing. This makes it easy to align components, and achieve a consistent look.

VerticalLayout, **HorizontalLayout**, and **FormLayout** fall into this category.

When you drop a component on a ordered layout, it will end up in a position determined by the layout, not exactly where you dropped it. Drop indicators help you estimate where the component will end up.

AbsoluteLayout

AbsoluteLayout allows free positioning of components, and supports anchoring freely in all directions. It is a powerful layout, but can be more challenging to use. You can use rulers, grids, guides and snapping to aid your work.

AbsoluteLayout allows you to position components freely - the component ends up where you drop it. However if you anchor the component elsewhere than to top/left, or use relative positioning, it might move when you change the size of the layout.

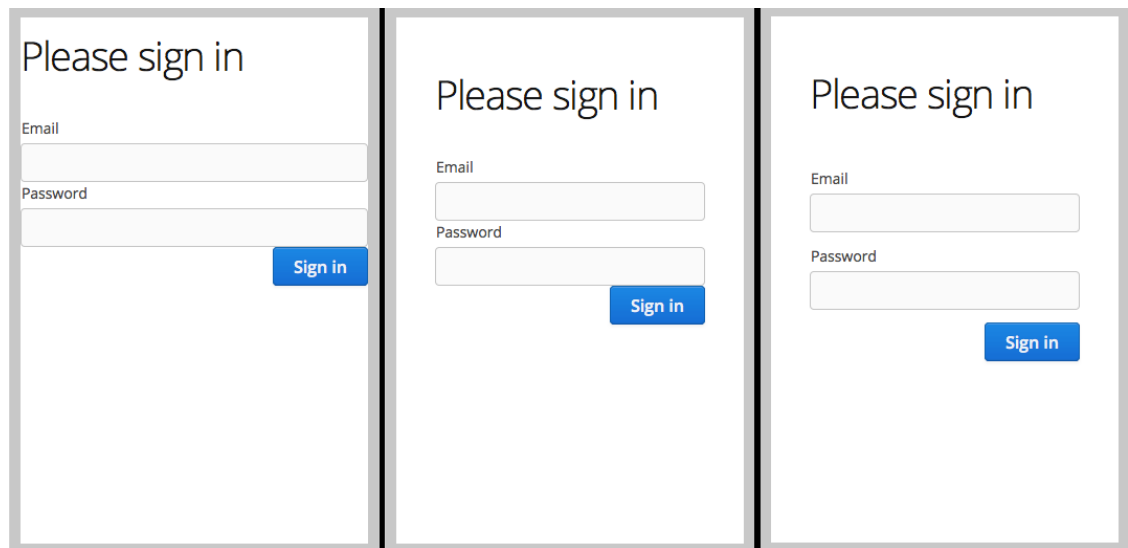
CssLayout

As the name indicates, **CssLayout** uses CSS to position components. It is very flexible, and with appropriate CSS, it can be used to achieve responsive layouts and a consistent look and feel. However, it requires CSS - either pre-made and copied to your theme, or hand-crafted by you.

8.4.2. Starting from Blank

When you add the first layout to your blank canvas, it will be sized 100% x 100%, filling the whole viewport. Whether or not this is a good idea depends on your design. For many UIs having a **VerticalLayout** as root, it makes sense to have the layout 100% wide, but *auto* high. This will make the layout grow vertically as you add components, instead of splitting the available vertical space evenly between components.

Most UIs will not look good without margin and spacing. You can enable them for ordered layouts in **Properties**. Figure 8.6, “Effects of Margin and Spacing” illustrates the same layout without margin or spacing, with margin, and with spacing.

Figure 8.6. Effects of Margin and Spacing

The *info bar*, illustrated in Figure 8.7, “Info Bar for Quick Adjustments”, lets you quickly toggle between *auto* sizing and 100%. You can try out the effect of these changes by grabbing just outside the viewport (canvas) corner and resizing it (add a few components to your layout first).

Figure 8.7. Info Bar for Quick Adjustments

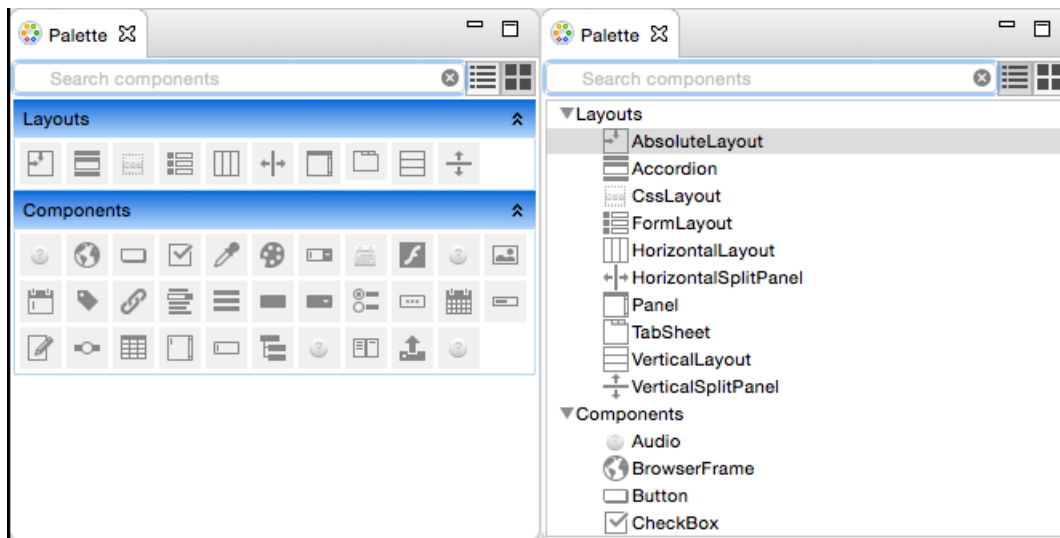
8.4.3. Using Templates

Templates provide a starting-point for your design - add, remove and modify the created design as you see fit.

You can also create templates of your own; just design a suitable starting point, then place the resulting HTML in `~/.vaadin/designer/templates`. It will show up in the **New Design** wizard as a template (without the `.html` extension).

8.4.4. Adding Components

Components can be added by dragging from the palette, either to the canvas, or to the Outline view.

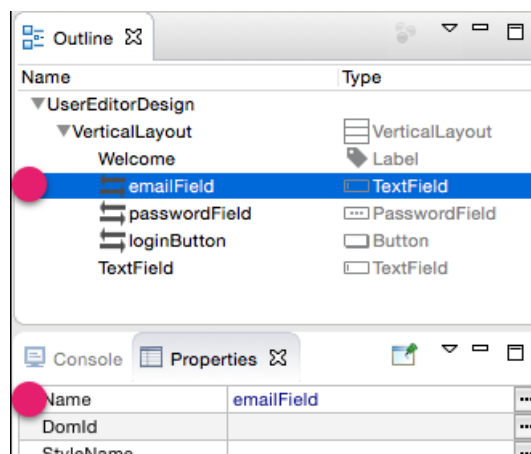
Figure 8.8. Component Palette (Alternate Layouts)

The component palette has a search field, and also two modes: list and tile. In the list mode you can see the component name next to the icon, which is convenient at first. The tile mode lets you see all components at once, which will speed up your work quite a bit. It requires a little patience, but is really worth your while in the long run. The component name can also be seen when hovering the icon.

The component you add will be selected in the editor view, and you can immediately edit its properties, such as the caption.

Editing Properties

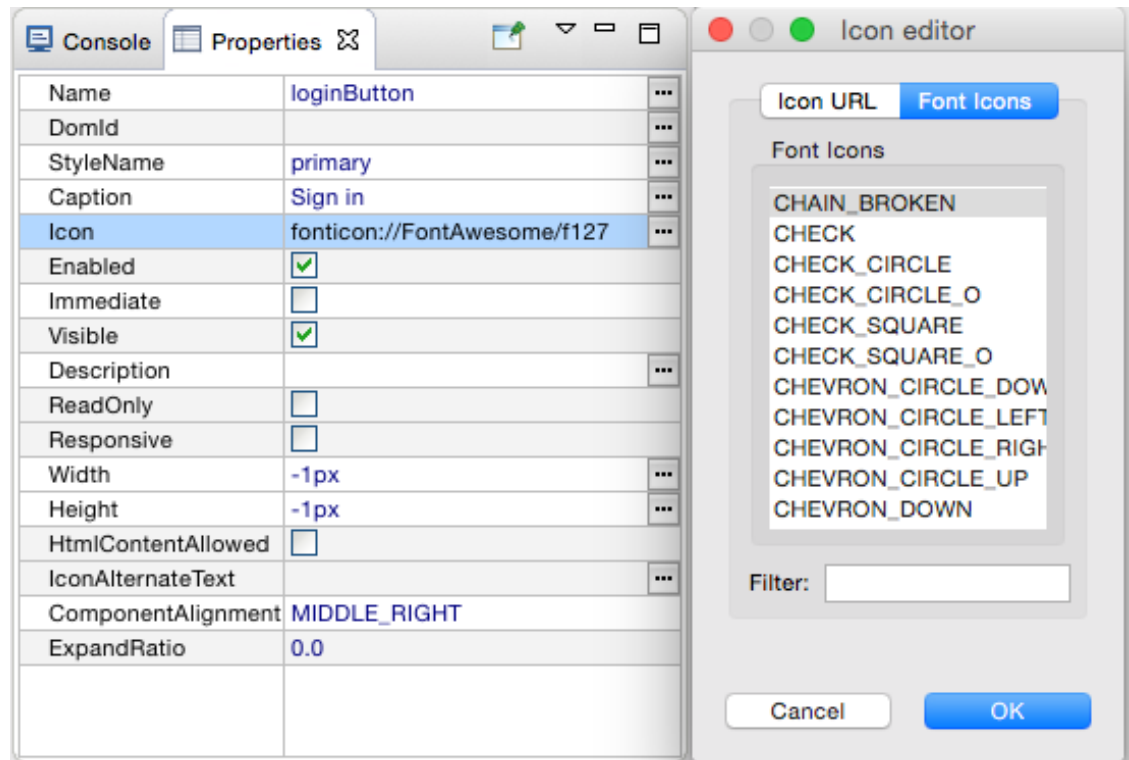
You can edit component properties in the **Properties** panel. It is a good idea to give components at least a **name** if they are to be used from Java code to add logic (such as click listeners for buttons). Generally, this is needed for most controls, but not for most layouts.

Figure 8.9. Property Panel

In addition to exporting the named components to Java, you will end up with things like `saveButton` and `emailField` in your outline, which will help you keep track of your components.

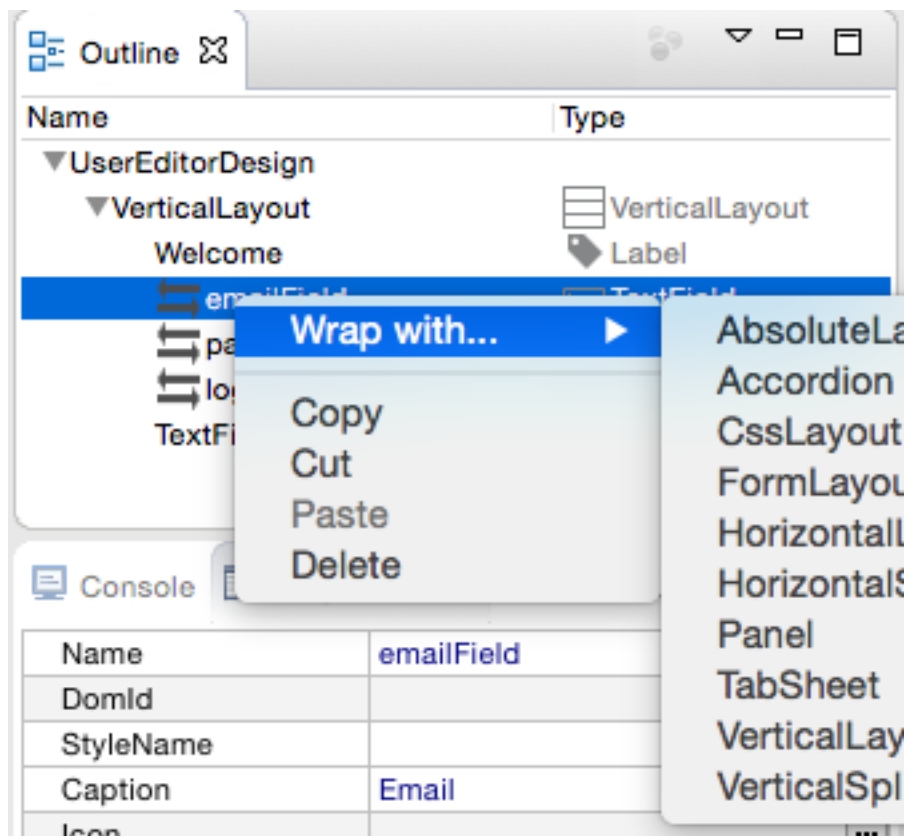
Note the ellipsis (...) button next to most properties - in many cases a more helpful editor is presented when you click it.

Figure 8.10. A Property Editor



Wrapping a Component

Once in a while, you may need to wrap a component with a layout, in order to achieve the desired result (quite often injecting a **HorizontalLayout** into a **VerticalLayout**, or vice versa). You can achieve this by right-clicking the component in the **Outline**, and choosing Wrap with in the context menu.

Figure 8.11. Wrapping a Component

8.4.5. Previewing

While creating a design, it is convenient to preview how the UI will behave in different sizes and on different devices. There are a number of features geared for this.

Resizing Viewport and Presets

The WYSIWYG canvas area also doubles as viewport. By resizing it, you can preview how your design will behave in different sizes, just as if it was displayed in a browser window that is being resized, or dropped in a Panel of a specific size.

You can manually resize the viewport by grabbing just outside of an edge or corner of the viewport, and dragging to the desired size. When you resize the viewport, you can see that the viewport control on the toolbar changes to indicate the current size.

By typing in the viewport control, you can also input a specific size (such as "200 x 200"), or open it up to reveal size presets. Choose a preset, such as **Phone** to instantly preview the design on that size.

Figure 8.12. Viewport Preset Sizes

You can also add your own presets - for instance known portlet or dashboard tile sizes, or other specific sizes you want to target.

To preview the design in the other orientation (portrait vs. landscape), press the icon right of the viewport size control.

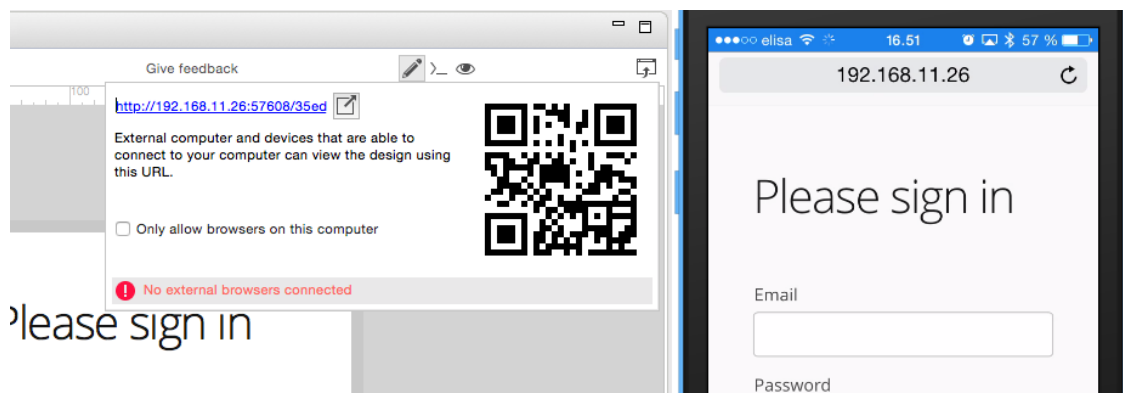
Quick preview

The **Quick preview** is one of the edit-modes available to the right in the toolbar (the other modes being **Design** and **Code**). In this mode, all designing tools and indicators are removed from the UI, and you can interact with components - type text, open dropdowns, check boxes, tab between fields, and so on. It allows you to quickly get a feel for (for instance) how a form will work when filling it in. Logic is still not run (hence "quick"), so no real data is shown and, for example, buttons do nothing.

External Preview

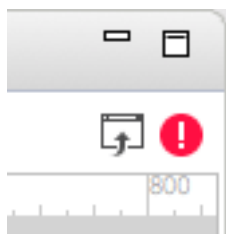
The external preview popup shows a QR code and its associated URL. By browsing to the URL with a browser or device that can access your computer (that is, on the same LAN), you can instantly see the design and interact with it. This view has no extra designer-specific controls or viewports added, instead it just shows the design as-is; the browser is the viewport.

Figure 8.13. External Preview



External preview allows multiple browsers and devices to be connected at once, and they are all updated live as you change the design in Eclipse. There is an indicator in the toolbar when the design is viewed externally.

Figure 8.14. Indicator for External Preview



This is an awesome way to instantly preview results on multiple devices and browsers, or to show off a design and collaborate on it - for instance in a meeting setting.

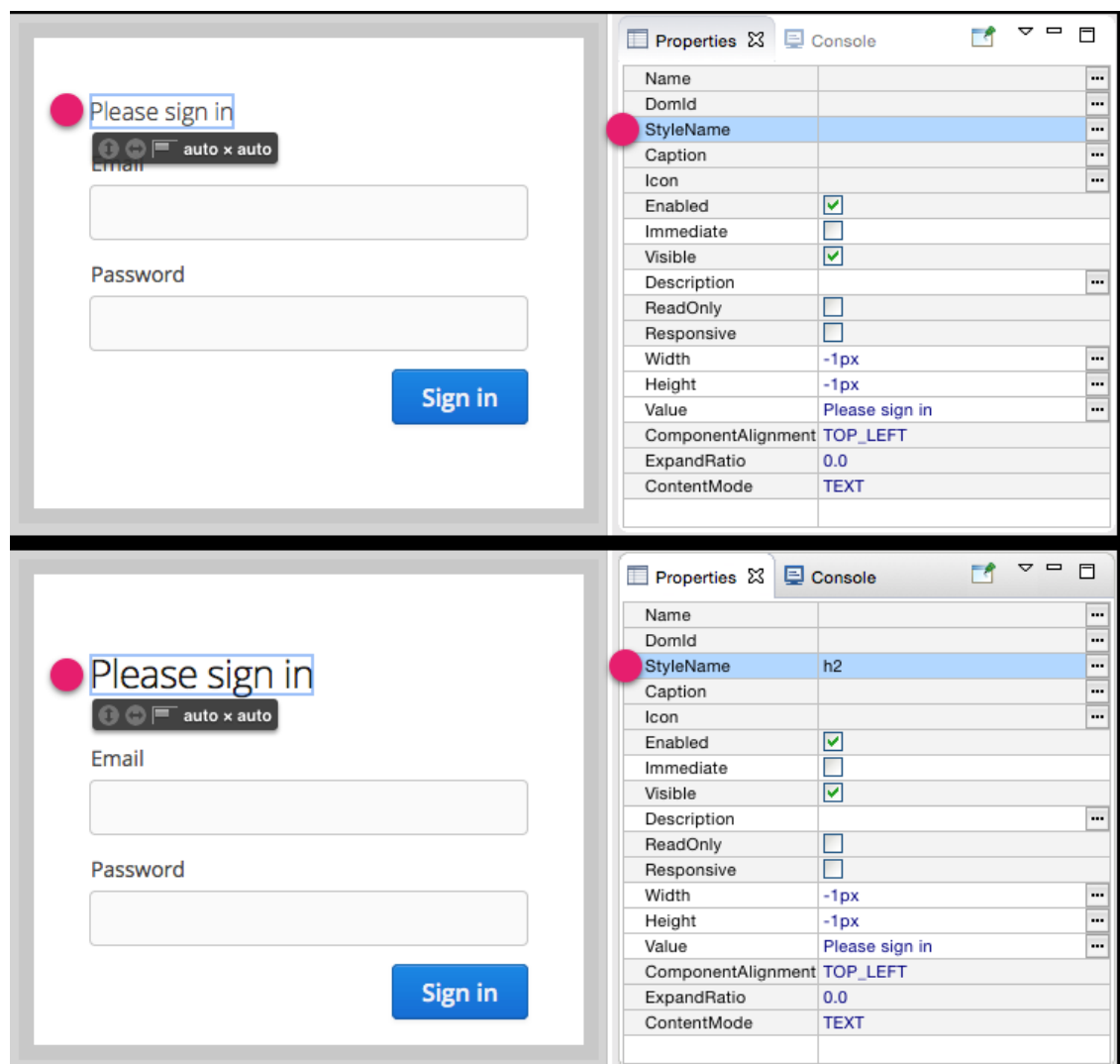
8.5. Theming and Styling

By default, Vaadin Designer shows your design using your application theme, so usually what you see is really what you get. You can also use the theme dropdown to apply a specific theme.

8.5.1. Theme Based on Valo

If your theme is based on the Valo theme (the default), you can make use of the built-in Valo features. For example, if you can apply the built-in component styles by adding the appropriate stylename. You will see the result instantly.

Figure 8.15. Adding Style Names



You can also modify the Valo settings by changing the parameters in your theme file (see below for more information about the theme file). See Section 9.7, "Valo Theme" for information about Valo theme.

8.5.2. Theme File

In a regular Vaadin application, you theme will be located in the `VAADIN/themes/<project-name>` folder, in the `<projectname>.scss` file. This is where you can modify Valo settings, and add you own styles.

When you make changes to this file (and save it), Vaadin Designer will notice and update the design, which is very convenient when styling your design, or generally when learning to make an application theme.

You can apply global styles (such as to style all buttons), or scoped as you wish. You can "scope" styles by specifying one or more space-separated style names in the `StyleName` property, then matching to that in CSS/Sass.

```
/* Applies to all buttons */
.v-button { ... }

/* Applies to components having the stylename "mybutton" */
.mybutton { ... }

/* Applies to all "mybutton" components within a "mydialog" layout */
.mydialog .mybutton { ... }
```

If you use the same stylenames in multiple designs, the same styles will be applied, allowing you to create a consistent look.

If you do not want some styles to apply to other designs, you should give your root layout a unique stylename (for instance matching the design name), and prefix all styles with that.

```
.usereditordesign .mybutton { ... }
```

8.6. Wiring It Up

Connecting Java logic to your design is made easy by splitting the UI definition and code into several layers, laying the foundation for a good separation of concerns.

From a coding perspective, a designs will have three separate parts:

1. A declarative definition of the UI
2. A "companion" class exposing select components a Java fields
3. Custom Java code connected to the components exposed to Java

The declarative file (1) is normally created by Vaadin Designer, but can be created and edited by hand as well, and changes you make will be reflected in the Designer.

The companion class (2) is auto-generated based on the declarative file by Vaadin Designer, and you should not edit this file - it will be overwritten and any changes you make will not be reflected in the design.

Finally, the custom Java code (3) is completely created and maintained by you (or some other programmer). As long as the companion class (2) is used to connect logic to components, you will notice if, for example, some component goes missing. In effect, you can safely edit the design with Vaadin Designer, because you will notice if you break the logic.

8.6.1. Declarative Code

The declarative format is based on HTML/WebComponents files, and is supported directly by the Vaadin Framework. The design files have the `.html` suffix.

Note that the format does still not support arbitrary HTML at the moment. See Section 5.3, “Designing UIs Declaratively” for more information regarding the declarative format.

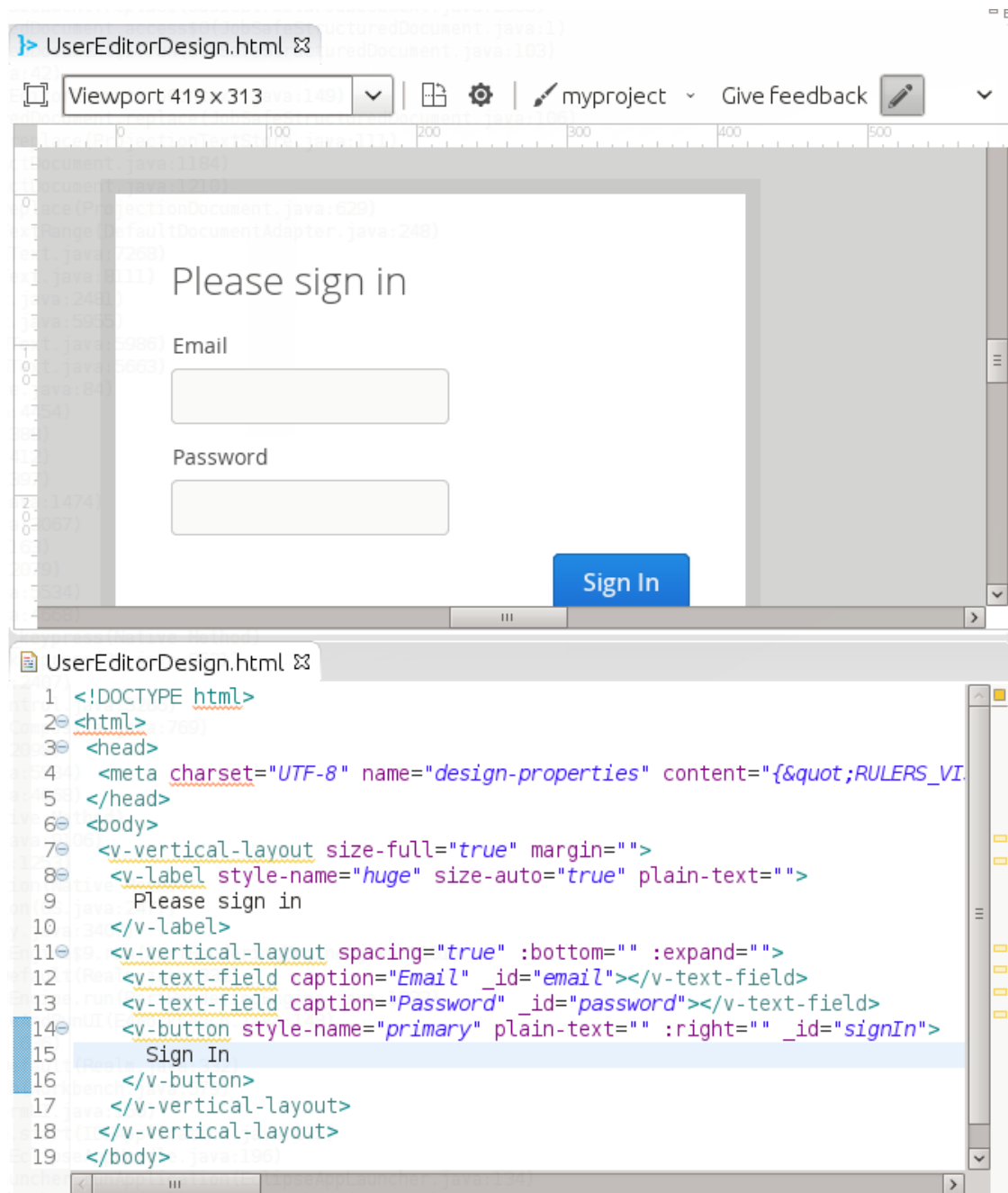
You can edit the declarative file with any text or HTML editor, but the Vaadin Designer is needed to automatically create and update the connection between declarative and Java.

Any changes you make to the declarative file are also reflected in the Designer.

Split View

In fact, you can keep the Designer open next to an HTML code editor, and see the changes you make visually reflected. This can be a powerful mode of operation.

You can open a code editor by right-clicking on a HTML design file and selecting **Open With** → **HTML Editor**. You can then drag the editor tab to under the Designer view to create a split view, as illustrated in Figure 8.16, “Split View with Designer”.

Figure 8.16. Split View with Designer

In a similar way, you can open your theme file (Sass or CSS) in a split view. When you save the file, Designer runs on-the-fly compilation for it and shows the changes to the visual appearance immediately.

8.6.2. Java Code

Vaadin Designer automatically creates a "companion" Java class, with all the components you choose to export from your design exposed as Java fields, all wired up and laid out according to your design.

The file will be overwritten by the Designer, and should not be edited.

This provides the compile-time connection between the design and Java code, as long as you are using Vaadin Designer to edit your UI. For instance, if you remove a component from the design that your code is using, you will immediately notice the error in Eclipse.

Exporting Components

Components are "exported" to Java by setting the "name" property in Vaadin Designer. The name is represented as a "`_id`" attribute in the declarative format (where it can also be manually set) and the corresponding field will be added to the Java companion class.

Note that the name is used as Java field name, so Java naming conventions are recommended.

If you change the name, the declarative file and the companion Java class will be updated, but custom code referencing the field will currently not.

Extending or Referencing

The companion Java class is overwritten and should not be edited. This is intentional, to create a clear and predictable separation of concerns. The declarative format configures the components, the companion class exposes the components to Java, and the logic goes in a separate file - either just referencing the companion class (in a composition) or by extending it.

In many cases, it is best to encapsulate the logic pertaining to a design by extending the companion class, and only exposing the API and events as needed. It might even make sense to place the designs in package(s) of their own.

8.7. Limitations

Vaadin Designer 1.0 has limitations that we hope to address as soon as possible.

- Multi-select
- Designs cannot be nested
- Custom components and add-ons are rendered as place-holders
- Custom widget sets are not used
- Data components, such as Table and Grid, all get the same dummy content
- You cannot dynamically enter real or mock data
- Advanced components, which cannot be properly configured with simple properties, are lacking features
- Component-specific edit mode is missing
- GridLayout column and rowspan cannot be adjusted
- There is no fluid/relative grid for responsive design
- No easy way to swap a layout, keeping content (can be done in code)

Chapter 9

Themes

9.1. Overview	287
9.2. Introduction to Cascading Style Sheets	289
9.3. Syntactically Awesome Stylesheets (Sass)	296
9.4. Compiling Sass Themes	298
9.5. Creating and Using Themes	301
9.6. Creating a Theme in Eclipse	305
9.7. Valo Theme	307
9.8. Font Icons	313
9.9. Custom Fonts	315
9.10. Responsive Themes	316

This chapter provides details about using and creating *themes* that control the visual look of web applications. Themes are created using Sass, which is an extension of CSS (Cascading Style Sheets), or with plain CSS. We provide an introduction to CSS, especially concerning the styling of HTML by element classes.

9.1. Overview

Vaadin separates the appearance of the user interface from its logic using *themes*. Themes can include Sass or CSS style sheets, custom HTML layouts, and any necessary graphics. Theme resources can also be accessed from application code as **ThemeResource** objects.

Custom themes are placed under the `VAADIN/themes/` folder of the web application (under `webContent` in Eclipse or `src/main/webapp` in Maven projects). This location is fixed — the `VAADIN` folder contains static resources that are served by the Vaadin servlet. The servlet augments the files stored in the folder by resources found from corresponding `VAADIN` folders

contained in JARs in the class path. For example, the built-in themes are stored in the `vaadin-themes.jar`.

Figure 9.1, “Contents of a Theme” illustrates the contents of a theme.

Figure 9.1. Contents of a Theme



The name of a theme folder defines the name of the theme. The name is used in the `@Theme` annotation that sets the theme. A theme must contain either a `styles.scss` for Sass themes, or `styles.css` stylesheet for plain CSS themes, but other contents have free naming. We recommend that you have the actual theme content in a SCSS file named after the theme, such as `mytheme.scss`, to make the names more unique.

We also suggest a convention for naming the folders as `img` for images, `layouts` for custom layouts, and `css` for additional stylesheets.

Custom themes need to extend a base theme, as described in Section 9.5, “Creating and Using Themes”. Copying and modifying an existing theme is also possible, but it is not recommended, as it may need more work to maintain if the modifications are small.

You use a theme by specifying it with the `@Theme` annotation for the UI class of the application as follows:

```
@Theme("mytheme")
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        ...
    }
}
```

A theme can contain alternate styles for user interface components, which can be changed as needed.

In addition to style sheets, a theme can contain HTML templates for custom layouts used with **CustomLayout**. See Section 7.14, “Custom Layouts” for details.

Resources provided in a theme can also be accessed using the **ThemeResource** class, as described in Section 5.5.4, “Theme Resources”. This allows displaying theme resources in component icons, in the **Image** component, and other such uses.

9.2. Introduction to Cascading Style Sheets

Cascading Style Sheets or CSS is the basic technique to separate the appearance of a web page from the content represented in HTML. In this section, we give an introduction to CSS and look how they are relevant to software development with Vaadin.

As we can only give a short intruction in this book, we encourage you to refer to the rich literature on CSS and the many resources available in the web. You can find the authoratitative specifications of CSS standards from the W3C website .

9.2.1. Applying CSS to HTML

Let us consider the following HTML document that contains various markup elements for formatting text. Vaadin UIs work in essentially similar documents, even though they use somewhat different elements to draw the user interface.

```
<html>
  <head>
    <title>My Page</title>
    <link rel="stylesheet" type="text/css"
        href="mystylesheet.css"/>
  </head>
  <body>
    <p>This is a paragraph</p>
    <p>This is another paragraph</p>
    <table>
      <tr>
        <td>This is a table cell</td>
        <td>This is another table cell</td>
      </tr>
    </table>
  </body>
</html>
```

The HTML elements that will be styled later by matching CSS rules are emphasized above.

The link element in the HTML header defines the CSS stylesheet. The definition is automatically generated by Vaadin in the HTML page that loads the UI of the application. A stylesheet can also be embedded in the HTML document itself, as is done when optimizing their loading in Vaadin TouchKit, for example.

9.2.2. Basic CSS Rules

A stylesheet contains a set of *rules* that can match the HTML elements in the page. Each rule consists of one or more *selectors*, separated with commas, and a *declaration block* enclosed

in curly braces. A declaration block contains a list of *property* statements. Each property has a label and a value, separated with a colon. A property statement ends with a semicolon.

Let us look at an example that matches certain elements in the simple HTML document given in the previous section:

```
p, td {  
    color: blue;  
}  
  
td {  
    background: yellow;  
    font-weight: bold;  
}
```

The `p` and `td` are element type selectors that match with `p` and `td` elements in HTML, respectively. The first rule matches with both elements, while the second matches only with `td` elements. Let us assume that you have saved the above style sheet with the name `mystylesheet.css` and consider the following HTML file located in the same folder.

Figure 9.2. Simple Styling by Element Type

This is a paragraph.

This is another paragraph.

This is a table cell	This is another table cell

Style Inheritance in CSS

CSS has *inheritance* where contained elements inherit the properties of their parent elements. For example, let us change the above example and define it instead as follows:

```
table {  
    color: blue;  
    background: yellow;  
}
```

All elements contained in the `table` element would have the same properties. For example, the text in the contained `td` elements would be in blue color.

HTML Element Types

HTML has a number of element types, each of which accepts a specific set of properties. The `div` elements are generic elements that can be used to create almost any layout and formatting that can be created with a specific HTML element type. Vaadin uses `div` elements extensively to draw the UI, especially in layout components.

Matching elements by their type as shown above is, however, rarely if ever used in style sheets for Vaadin applications. We used it above, because it is the normal way in regular HTML documents that use the various HTML elements for formatting text, but it is not applicable in Vaadin UIs that consist mostly of `div` elements. Instead, you need to match by element class, as described next.

9.2.3. Matching by Element Class

Matching HTML elements by the *class* attribute is the most common form of matching in Vaadin stylesheets. It is also possible to match with the *identifier* of a unique HTML element.

The class of an HTML element is defined with the *class* attribute as follows:

```
<html>
  <body>
    <p class="normal">This is the first paragraph</p>

    <p class="another">This is the second paragraph</p>

    <table>
      <tr>
        <td class="normal">This is a table cell</td>
        <td class="another">This is another table cell</td>
      </tr>
    </table>
  </body>
</html>
```

The class attributes of HTML elements can be matched in CSS rules with a selector notation where the class name is written after a period following the element name. This gives us full control of matching elements by their type and class.

```
p.normal {color: red;}
p.another {color: blue;}
td.normal {background: pink;}
td.another {background: yellow;}
```

The page would look as shown below:

Figure 9.3. Matching HTML Element Type and Class

This is a paragraph with "normal" class

This is a paragraph with "another" class

This is a table cell with "normal" class	This is a table cell with "another" class

We can also match solely by the class by using the universal selector *** for the element name, for example **.normal*. The universal selector can also be left out altogether so that we use just the class name following the period, for example *.normal*.

```
.normal {
  color: red;
}

.another {
  background: yellow;
}
```

In this case, the rule will match with all elements of the same class regardless of the element type. The result is shown in Figure 9.4, "Matching Only HTML Element Class". This example il-

illustrates a technique to make style sheets compatible regardless of the exact HTML element used in drawing a component.

Figure 9.4. Matching Only HTML Element Class

This is a paragraph with "normal" class

This is a paragraph with "another" class

This is a table cell with "normal" class	This is a table cell with "another" class

To ensure future compatibility, we recommend that you use only matching based on the classes and *do not* match for specific HTML element types in CSS rules, because Vaadin may change the exact HTML implementation how components are drawn in the future. For example, Vaadin earlier used `div` element to draw **Button** components, but later it was changed to use the special-purpose button element in HTML. Because of using the `v-button` style class in the CSS rules for the button, styling it has changed only very little.

9.2.4. Matching by Descendant Relationship

CSS allows matching HTML by their containment relationship. For example, consider the following HTML fragment:

```
<body>
  <p class="mytext">Here is some text inside a
    paragraph element</p>
  <table class="mytable">
    <tr>
      <td class="mytext">Here is text inside
        a table and inside a td element.</td>
    </tr>
  </table>
</body>
```

Matching by the class name `.mytext` alone would match both the `p` and `td` elements. If we want to match only the table cell, we could use the following selector:

```
.mytable .mytext {color: blue;}
```

To match, a class listed in a rule does not have to be an immediate descendant of the previous class, but just a descendant. For example, the selector `.v-panel .v-button` would match all elements with class `.v-button` somewhere inside an element with class `.v-panel`.

9.2.5. Importance of Cascading

CSS or Cascading Stylesheets are, as the name implies, about *cascading* stylesheets, which means applying the stylesheet rules according to their origin, importance, scope, specificity, and order.

For exact rules for cascading in CSS, see the section Cascading in the CSS specification.

Importance

Declarations in CSS rules can be made override declarations with otherwise higher priority by annotating them as `!important`. For example, an inline style setting made in the `style` attribute of an HTML element has a higher specificity than any rule in a CSS stylesheet.

```
<div class="v-button" style="height: 20px;">...
```

You can override the higher specificity with the `!important` annotation as follows:

```
.v-button {height: 30px !important;}
```

Specificity

A rule that specifies an element with selectors more closely overrides ones that specify it less specifically. With respect to the element class selectors most commonly used in Vaadin themes, the specificity is determined by the number of class selectors in the selector.

```
.v-button {}  
.v-verticallayout .v-button {}  
.v-app .v-verticallayout .v-button {}
```

In the above example, the last rule would have the highest specificity and would match.

As noted earlier, style declarations given in the `style` attribute of a HTML element have higher specificity than declarations in a CSS rule, except if the `!important` annotation is given.

See the CSS3 selectors module specification for details regarding how the specificity is computed.

Order

CSS rules given later have higher priority than ones given earlier. For example, in the following, the latter rule overrides the former and the color will be black:

```
.v-button {color: white}  
.v-button {color: black}
```

As specificity has a higher cascading priority than order, you could make the first rule have higher priority by adding specificity as follows:

```
.v-app .v-button {color: white}  
.v-button {color: black}
```

The order is important to notice in certain cases, because Vaadin does not guarantee the order in which CSS stylesheets are loaded in the browser, which can in fact be random and result in very unexpected behavior. This is not relevant for Sass stylesheets, which are compiled to a single stylesheet. For plain CSS stylesheets, such as add-on or TouchKit stylesheets, the order can be relevant.

9.2.6. Style Class Hierarchy of a Vaadin UI

Let us give a real case in a Vaadin UI by considering a simple Vaadin UI with a label and a button inside a vertical layout:

```
// UI has v-ui style class  
@Theme("mytheme")  
public class HelloWorld extends UI {
```

```

@Override
protected void init(VaadinRequest request) {
    // VerticalLayout has v-verticallayout style
    VerticalLayout content = new VerticalLayout();
    setContent(content);

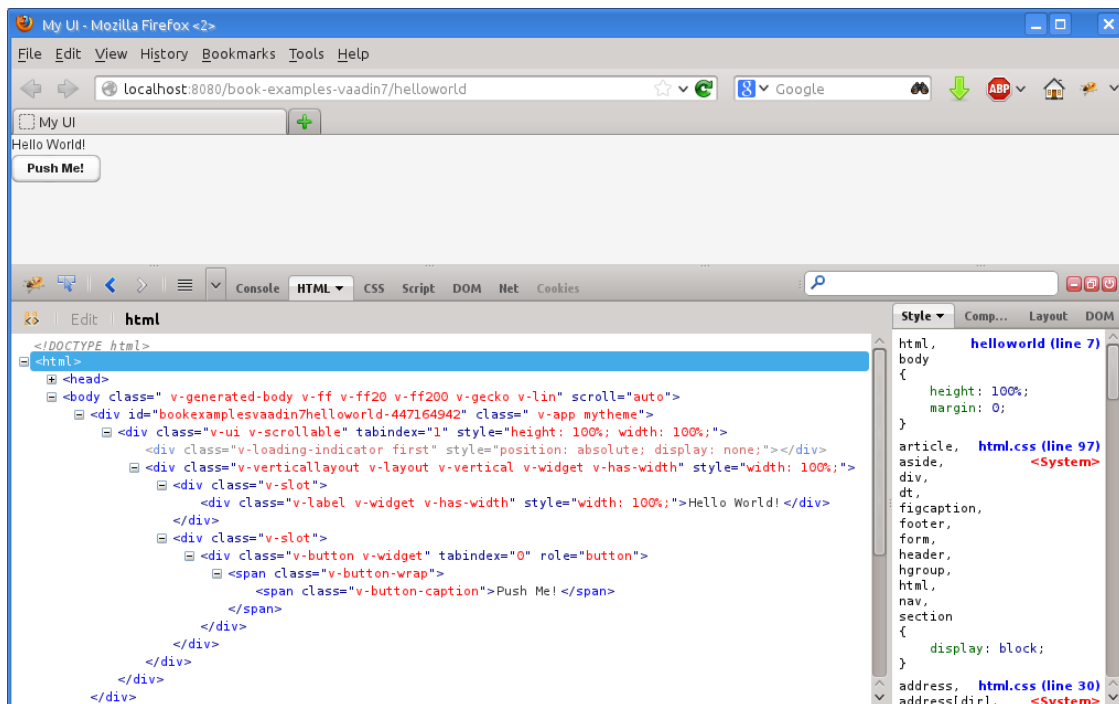
    // Label has v-label style
    content.addComponent(new Label("Hello World!"));

    // Button has v-button style
    content.addComponent(new Button("Push Me!",
        new Button.ClickListener() {
            @Override
            public void buttonClick(ClickEvent event) {
                Notification.show("Pushed!");
            }
        }
    ));
}
}

```

The UI will look by default as shown in Figure 9.5, “An Unthemed Vaadin UI”. By using a HTML inspector such as Firebug, you can view the HTML tree and the element classes and applied styles for each element.

Figure 9.5. An Unthemed Vaadin UI



Now, let us look at the HTML element class structure of the UI, as we can see it in the HTML inspector:

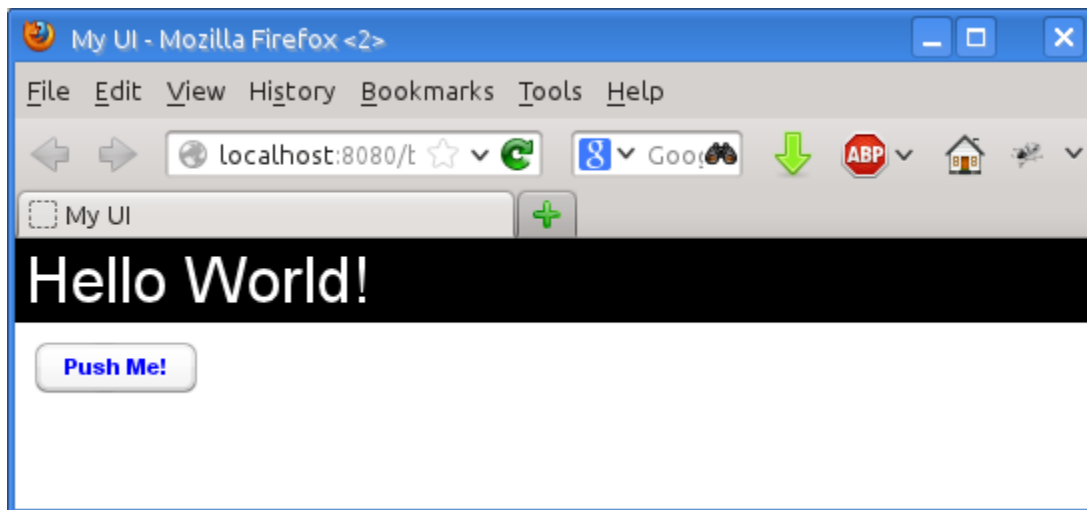
```

<body class="v-generated-body v-ff v-ff20 v-ff200 v-gecko v-lin"
    scroll="auto">
  <div id="bookexamplesvaadin7helloworld-447164942"
    class="v-app mytheme">

```

Now, consider the following theme where we set the colors and margins of various elements. The theme is actually a Sass theme.

The look has changed as shown in Figure 9.6, “Themed Vaadin UI”.

Figure 9.6. Themed Vaadin UI

An element can have multiple classes separated with a space. With multiple classes, a CSS rule matches an element if any of the classes match. This feature is used in many Vaadin components to allow matching based on the state of the component. For example, when the mouse is over a **Link** component, `over` class is added to the component. Most of such styling is a feature of Google Web Toolkit.

9.2.7. Notes on Compatibility

CSS is a standard continuously under development. It was first proposed in 1994. The specification of CSS is maintained by the CSS Working Group of World Wide Web Consortium (W3C). Versioned with backward-compatible "levels", CSS Level 1 was published in 1996, Level 2 in 1998, and the ongoing development of CSS Level 3 started in 1998. CSS3 is divided into a number of separate modules, each developed and progressing separately, and many of the modules are already Level 4.

While the support for CSS has been universal in all graphical web browsers since at least 1995, the support has been very incomplete at times and there still exists an unfortunate number of incompatibilities between browsers. While we have tried to take these incompatibilities into account in the built-in themes in Vaadin, you need to consider them while developing your own themes. Compatibility issues are detailed in various CSS handbooks.

9.3. Syntactically Awesome Stylesheets (Sass)

Vaadin uses Sass for stylesheets. Sass is an extension of CSS3 that adds nested rules, variables, mixins, selector inheritance, and other features to CSS. Sass supports two formats for stylesheet: Vaadin themes are written in SCSS (`.scss`), which is a superset of CSS3, but Sass also allows a more concise indented format (`.sass`).

Sass can be used in two basic ways in Vaadin applications, either by compiling SCSS files to CSS or by doing the compilation on the fly. The latter way is possible if the development mode is enabled for the Vaadin servlet, as described in Section 5.9.6, "Other Servlet Configuration Parameters".

9.3.1. Sass Overview

Variables

Sass allows defining variables that can be used in the rules.

```
$textcolor: blue;

.v-button-caption {
  color: $textcolor;
}
```

The above rule would be compiled to CSS as:

```
.v-button-caption {
  color: blue;
}
```

Also mixins can have variables as parameters, as explained later.

Nesting

Sass supports nested rules, which are compiled into inside-selectors. For example:

```
.v-app {
  background: yellow;

  .mybutton {
    font-style: italic;

    .v-button-caption {
      color: blue;
    }
  }
}
```

is compiled as:

```
.v-app {
  background: yellow;
}

.v-app .mybutton {
  font-style: italic;
}

.v-app .mybutton .v-button-caption {
  color: blue;
}
```

Mixins

Mixins are rules that can be included in other rules. You define a mixin rule by prefixing it with the `@mixin` keyword and the name of the mixin. You can then use `@include` to apply it to another rule. You can also pass parameters to it, which are handled as local variables in the mixin.

For example:

```
@mixin mymixin {  
  background: yellow;  
}  
  
@mixin othermixin($param) {  
  margin: $param;  
}  
  
.v-button-caption {  
  @include mymixin;  
  @include othermixin(10px);  
}
```

The above SCSS would translated to the following CSS:

```
.v-button-caption {  
  background: yellow;  
  margin: 10px;  
}
```

You can also have nested rules in a mixin, which makes them especially powerful. Mixing in rules is used when extending Vaadin themes, as described in Section 9.5.1, “Sass Themes”.

Vaadin themes are defined as mixins to allow for certain uses, such as different themes for different portlets in a portal.

9.3.2. Sass Basics with Vaadin

We are not going to give in-depth documentation of Sass and refer you to its excellent documentation at <http://sass-lang.com/>. In the following, we give just basic introduction to using it with Vaadin.

You can create a new Sass-based theme with the Eclipse plugin, as described in Section 9.6, “Creating a Theme in Eclipse”.

9.4. Compiling Sass Themes

Sass themes must be compiled to CSS understood by browsers. Compilation can be done with the Vaadin Sass Compiler, which you can run in Eclipse, Maven, or it can be run on-the-fly when the application is loaded in the browser. You can also use any other Sass compiler.

9.4.1. Compiling On the Fly

The easiest way to develop Sass themes is to compile them at runtime, when the page is loaded. The Vaadin servlet does this automatically if a compiled theme CSS file is not found from the theme folder. You need to have the SCSS source files placed in the theme folder. The theme is compiled when the `styles.css` is first requested from the server. If you edit the Sass theme, it is recompiled the next time you reload the page.

The on-the-fly compilation takes a bit time, so it is only available when the Vaadin servlet is in the development mode, as described in Section 5.9.6, “Other Servlet Configuration Parameters”. Also, it requires the theme compiler and all its dependencies to be in the class path of the servlet. At least for production, you must compile the theme to CSS, as described next.

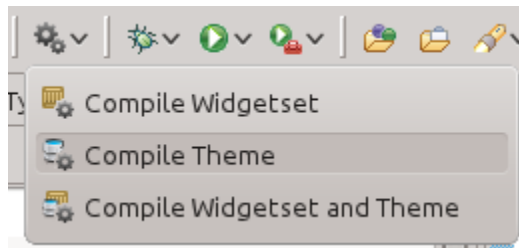
**Note**

If the on-the-fly compilation does not seem to work, ensure that your build script has not generated a pre-compiled CSS file. If `styles.css` file is found the servlet, the compilation is skipped. Delete such an existing `styles.css` file and disable theme compilation temporarily.

9.4.2. Compiling in Eclipse

If using Eclipse and the Vaadin Plugin for Eclipse, its project wizard creates a Sass theme. It includes Compile Theme command in the toolbar to compile the project theme to CSS. Another command compiles also the widget set.

Figure 9.7. Compiling Sass Theme



The `WebContent/VAADIN/mytheme/styles.scss` and any Sass sources included by it are compiled to `styles.css`.

9.4.3. Compiling with Maven

To compile the themes with Maven, you need to include the built-in themes as a dependency:

```
...
<dependencies>
  ...
  <dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-themes</artifactId>
    <version>${vaadin.version}</version>
  </dependency>
</dependencies>
...
```

This is automatically included at least in the `vaadin-archetype-application` archetype for Vaadin applications. The actual theme compilation is most conveniently done by the Vaadin Maven Plugin with `update-theme` and `compile-theme` goals.

```
...
<plugin>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-maven-plugin</artifactId>
  ...
  <executions>
    <execution>
      ...
      <goals>
        <goal>clean</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
        <goal>resources</goal>
        <goal>update-theme</goal>
        <goal>update-widgetset</goal>
        <goal>compile-theme</goal>
        <goal>compile</goal>
    </goals>
</execution>
</executions>
```

Once these are in place, the theme is compiled as part of relevant lifecycle phases, such as package.

```
mvn package
```

You can also compile just the theme with the compile-theme goal:

```
mvn vaadin:compile-theme
```

9.4.4. Compiling with Ant

With Apache Ant, you can easily resolve the dependencies with Ivy and compile the theme with the Theme Compiler included in Vaadin as follows. This build step can be conveniently included in a WAR build script.

Start with the following configuration:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
    name="My Project" basedir=".."
    default="package-war">

    <target name="configure">
        <!-- Where project source files are located -->
        <property name="src-location" value="src" />

        ... other project build definitions ...

        <!-- Name of the theme -->
        <property name="theme" value="book-examples"/>

        <!-- Compilation result directory -->
        <property name="result" value="build/result"/>
    </target>

    <!-- Initialize build -->
    <target name="init" depends="configure">
        <!-- Construct and check classpath -->
        <path id="compile.classpath">
            <!-- Source code to be compiled -->
            <pathelement path="${src-location}" />

            <!-- Vaadin libraries and dependencies -->
            <fileset dir="${result}/lib">
                <include name="*.jar"/>
            </fileset>
        </path>

        <mkdir dir="${result}"/>
    </target>
```

You should first resolve all Vaadin libraries to a single directory, which you can use for deployment, but also for theme compilation.

```
<target name="resolve" depends="init">
  <ivy:retrieve
    pattern="${result}/lib/[module]-[type]-[artifact]-[revision].[ext]"/>
</target>
```

Then, you can compile the theme as follows:

```
<!-- Compile theme -->
<target name="compile-theme"
  depends="init, resolve">
  <delete dir="${result}/VAADIN/themes/${theme}"/>
  <mkdir dir="${result}/VAADIN/themes/${theme}"/>

  <java classname="com.vaadin.sass.SassCompiler"
    fork="true">
    <classpath>
      <path refid="compile.classpath"/>
    </classpath>
    <arg value="WebContent/VAADIN/themes/${theme}/styles.scss"/>
    <arg value="${result}/VAADIN/themes/${theme}/styles.css"/>
  </java>

  <!-- Copy theme resources -->
  <copy todir="${result}/VAADIN/themes/${theme}">
    <fileset dir="WebContent/VAADIN/themes/${theme}">
      <exclude name="**/*.scss"/>
    </fileset>
  </copy>
</target>
</project>
```

9.5. Creating and Using Themes

Custom themes are placed in the `VAADIN/themes` folder of the web application, in an Eclipse project under the `WebContent` folder or `src/main/webapp` in Maven projects, as was illustrated in Figure 9.1, “Contents of a Theme”. This location is fixed. You need to have a theme folder for each theme you use in your application, although applications rarely need more than a single theme.

9.5.1. Sass Themes

You can use Sass themes in Vaadin in two ways, either by compiling them to CSS by yourself or by letting the Vaadin servlet compile them for you on-the-fly when the theme CSS is requested by the browser, as described in Section 9.4, “Compiling Sass Themes”.

To define a Sass theme with the name `mytheme`, you must place a file with name `styles.scss` in the theme folder `VAADIN/themes/mytheme`. If no `styles.css` exists in the folder, the Sass file is compiled on-the-fly when the theme is requested by a browser.

We recommend that you organize the theme in at least two SCSS files so that you import the actual theme from a Sass file that is named more uniquely than the `styles.scss`, to make it distinguishable in the editor. This organization is how the Vaadin Plugin for Eclipse creates a new theme.

If you use Vaadin add-ons that contain themes, Vaadin Plugin for Eclipse and Maven automatically add them to the `addons.scss` file.

Theme SCSS

We recommend that the rules in a theme should be prefixed with a selector for the theme name. You can do the prefixing in Sass by enclosing the rules in a nested rule with a selector for the theme name.

Themes are defined as Sass mixins, so after you import the mixin definitions, you can `@include` them in the theme rule as follows:

```
@import "addons.scss";
@import "mytheme.scss";

.mytheme {
  @include addons;
  @include mytheme;
}
```

However, this is mainly necessary if you use the UI in portlets, each of which can have its own theme, or in the special circumstance that the theme has rules that use empty parent selector `&` to refer to the theme name.

Otherwise, you can safely leave the nested theme selector out as follows:

```
@import "addons.scss";
@import "mytheme.scss";

@include addons;
@include mytheme;
```

The actual theme should be defined as follows, as a mixin that includes the base theme.

```
@import "../valo/valo.scss";

@mixin mytheme {
  @include valo;

  /* An actual theme rule */
  .v-button {
    color: blue;
  }
}
```

Add-on Themes

Some Vaadin add-ons include Sass styles that need to be compiled into the theme. These are managed in the `addons.scss` file in a theme, included from the `styles.scss`. The `addons.scss` file is automatically generated by the Vaadin Plugin for Eclipse or Maven.

```
/* This file is automatically managed and will be
   overwritten from time to time. /
/ Do not manually edit this file. /

/* Provided by vaadin-spreadsheet-1.0.0.beta1.jar */ @import "../VAADIN/addons/spreadsheet/spreadsheet.scss";
```

```
/ Import and include this mixin into your project
  theme to include the addon themes */
@mixin addons {
  @include spreadsheet;
}
```

9.5.2. Plain Old CSS Themes

In addition to Sass themes, you can create plain old CSS themes. CSS theme are more restricted than Sass styles - you can't parameterize CSS themes in any way, unlike you can Valo, for example. Further, an application can only have one CSS theme while you can have multiple Sass themes.

A CSS theme is defined in a `styles.css` file in the `VAADIN/themes/mytheme` folder. You need to import the `legacy-styles.css` of the built-in theme as follows:

```
@import "../reindeer/legacy-styles.css";

.v-app {
  background: yellow;
}
```

9.5.3. Styling Standard Components

Each user interface component in Vaadin has a CSS style class that you can use to control the appearance of the component. Many components have additional sub-elements that also allow styling. You can add context-specific stylenames with `addStyleName()`. Notice that `getStyleName()` returns only the custom stylenames, not the built-in stylenames for the component.

Please see the section on each component for a description of its styles. Most of the style names are determined in the client-side widget of each component. The easiest way to find out the styles of the elements is to use a HTML inspector such as FireBug.

Some client-side components or component styles can be shared by different server-side components. For example, `v-textfield` style is used for all text input boxes in components, in addition to **TextField**.

9.5.4. Built-in Themes

Vaadin currently includes the following built-in themes:

- `valo`, the primary theme since Vaadin 7.3
- `reindeer`, the primary theme in Vaadin 6 and 7
- `chameleon`, an easily customizable theme
- `runo`, the default theme in IT Mill Toolkit 5
- `liferay`, for Liferay portlets

In addition, there is the `base` theme, which should not be used directly, but is extended by the other built-in themes, except `valo`.

The built-in themes are provided in the respective `VAADIN/themes/<theme>/styles.scss` stylesheets in the `vaadin-themes` JAR. Also the precompiled CSS files are included, in case you want to use the themes directly.

Various constants related to the built-in themes are defined in the theme classes in `com.vaadin.ui.themes` package. These are mostly special style names for specific components.

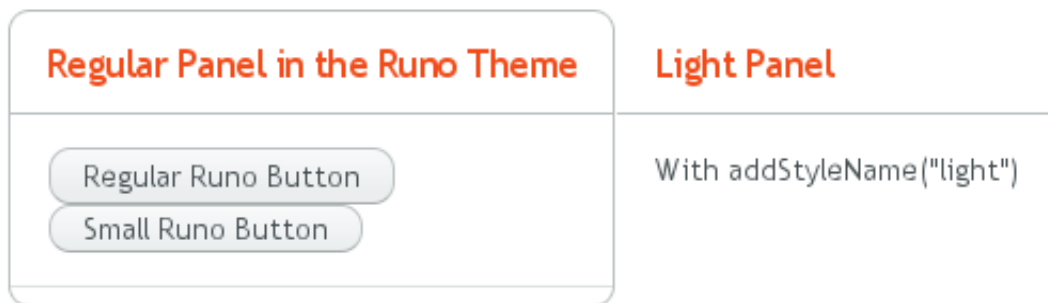
```
@Theme("runo")
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        ...
        Panel panel = new Panel("Regular Panel in the Runo Theme");
        panel.addComponent(new Button("Regular Runo Button"));

        // A button with the "small" style
        Button smallButton = new Button("Small Runo Button");
        smallButton.addStyleName(Runo.BUTTON_SMALL);

        Panel lightPanel = new Panel("Light Panel");
        lightPanel.addStyleName(Runo.PANEL_LIGHT);
        lightPanel.addComponent(
            new Label("With addStyleName(\"light\")"));
        ...
    }
}
```

The example with the Runo theme is shown in Figure 9.8, “Runo Theme”.

Figure 9.8. Runo Theme



The built-in themes come with a custom icon font, `FontAwesome`, which is used for icons in the theme, and which you can use as font icons, as described in Section 9.8, “Font Icons”.

Creation of a default theme for custom GWT widgets is described in Section 17.8, “Styling a Widget”.

9.5.5. Add-on Themes

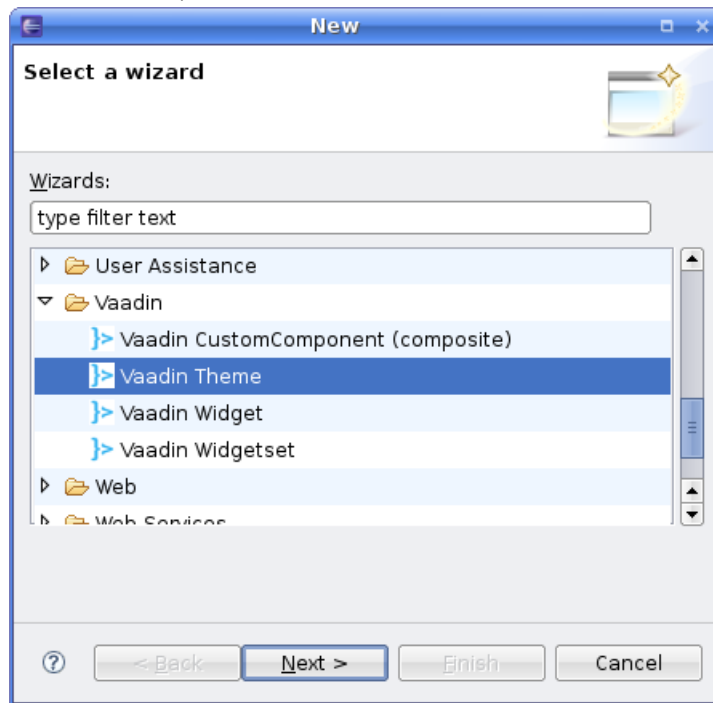
You can find more themes as add-ons from the Vaadin Directory. In addition, many component add-ons contain a theme for the components they provide.

The add-on themes need to be included in the project theme. Vaadin Plugin for Eclipse and Maven automatically include them in the `addons.scss` file in the project theme folder. It should be included by the project theme.

9.6. Creating a Theme in Eclipse

The Eclipse plugin automatically creates a theme stub for new Vaadin projects. It also includes a wizard for creating new custom themes. Do the following steps to create a new theme.

1. Select **File** → **New** → **Other...** in the main menu or right-click the **Project Explorer** and select **New** → **Other...**. A window will open.
2. In the **Select a wizard** step, select the **Vaadin** → **Vaadin Theme** wizard.



Click **Next** to proceed to the next step.

3. In the **Create a new Vaadin theme** step, you have the following settings:

Project(mandatory)

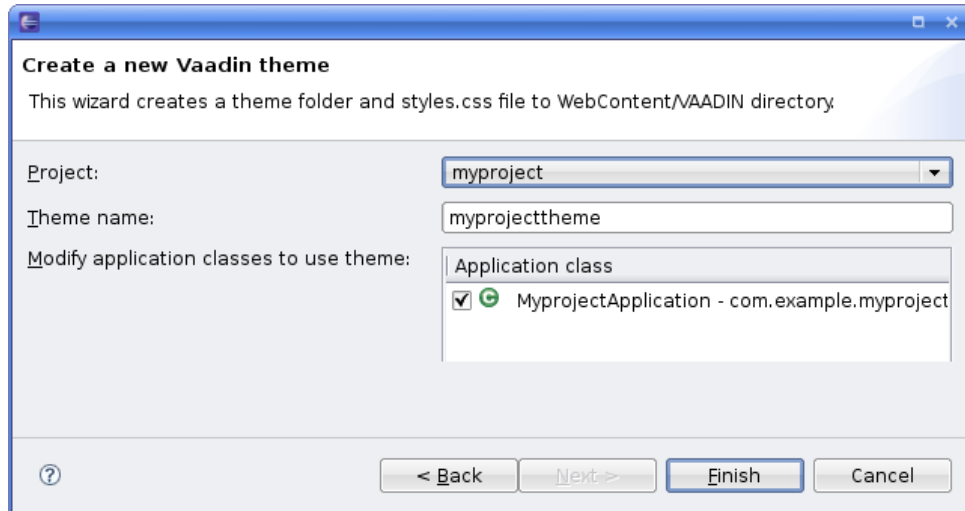
The project in which the theme should be created.

Theme name(mandatory)

The theme name is used as the name of the theme folder and in a CSS tag (prefixed with "v-theme-"), so it must be a proper identifier. Only latin alphanumerics, underscore, and minus sign are allowed.

Modify application classes to use theme(optional)

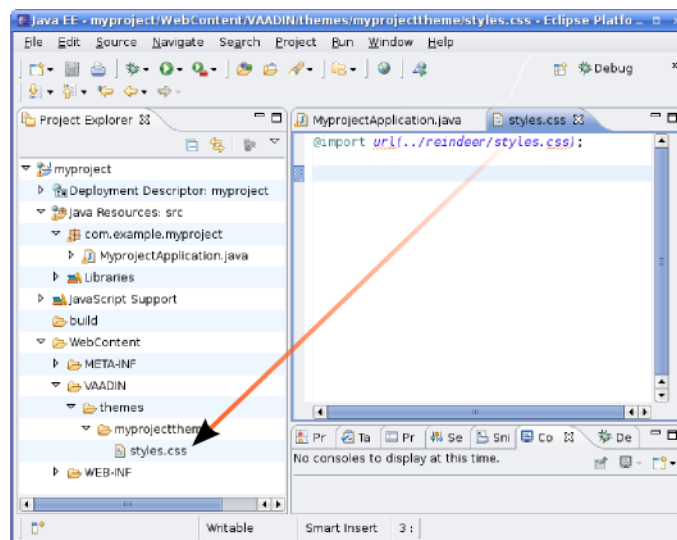
The setting allows the wizard to write a code statement that enables the theme in the constructor of the selected application (UI) class(es). If you need to control the theme with dynamic logic, you can leave the setting unchecked or change the generated line later.



Click **Finish** to create the theme.

The wizard creates the theme folder under the `WebContent/VAADIN/themes` folder and the actual style sheet as `mytheme.scss` and `styles.scss` files, as illustrated in Figure 9.9, “Newly Created Theme”.

Figure 9.9. Newly Created Theme



The created theme extends a built-in base theme with an `@import` statement. See the explanation of theme inheritance in Section 9.5, “Creating and Using Themes”. Notice that the `reindeer` theme is not located in the `widgetsets` folder, but in the Vaadin JAR. See Section 9.5.4, “Built-in Themes” for information for serving the built-in themes.

If you selected a UI class or classes in the **Modify application classes to use theme** in the theme wizard, the wizard will add the `@Theme` annotation to the UI class.

If you later rename the theme in Eclipse, notice that changing the name of the folder will not automatically change the `@Theme` annotation. You need to change such references to theme names in the calls manually.

9.7. Valo Theme

Valo is the word for light in Finnish. The Valo theme incorporates the use of light in its logic, in how it handles shades and highlights. It creates lines, borders, highlights, and shadows adaptively according to a background color, always with contrasts pleasant to human visual perception. Auxiliary colors are computed using an algorithmic color theory to blend gently with the background. The static art is complemented with responsive animations.

The true power of Valo lies in its configurability with parameters, functions, and Sass mixins. You can use the built-in definitions in your own themes or override the defaults. Detailed documentation of the available mixins, functions, and variables can be found in the Valo API documentation available at <http://vaadin.com/valo>.

9.7.1. Basic Use

Valo is used just like other themes. Its optional parameters must be given before the `@import` statement.

Your project theme file, such as `mytheme.scss`, included from the `styles.scss` file, could be as follows:

```
// Modify the base color of the theme
$v-background-color: hsl(200, 50%, 50%);

// Import valo after setting the parameters
@import "../valo/valo";

.mythemename {
  @include valo;

  // Your theme's rules go here
}
```

If you need to override mixins or function definitions in the valo theme, you must do that after the import statement, but before including the valo mixin. Also, with some configuration parameters, you can use variables defined in the theme. In this case, they need to be overridden after the import statement.

9.7.2. Common Settings

In the following, we describe the optional parameters that control the visual appearance of the Valo theme. In addition to the ones given here, component styles have their own parameters, listed in the sections describing the components in the other chapters.

General Settings

`$v-background-color`(default:[literal]*hsl(210, 0%, 98%)*)

The background color is the main control parameter for the Valo theme and it is used for computing all other colors in the theme. If the color is dark (has low luminance), light foreground colors that give high contrast with the background are automatically used.

You can specify the color in any way allowed in CSS: hexadecimal RGB color code, RGB/A value specified with `rgb()` or `rgba()`, HSL/A value specified with `hsl()` or

`hsla()`. You can also use color names, but it should be avoided, as not all CSS color names are currently supported.

`$v-app-background-color(default:$v-background-color)`

Background color of the UI's root element. You can specify the color in any way allowed in CSS.

`$v-app-loading-text(default:[literal] "")`

A static text that is shown under the loading spinner while the client-side engine is being loaded and started. The text must be given in quotes. The text can not be localized currently.

```
$v-app-loading-text: "Loading Resources...";
```

`$v-line-height(default:[literal] 1.55)`

Base line height for all widgets. It must be given a unitless number.

```
$v-line-height: 1.6;
```

Font Settings

`$v-font-size(default:[literal] 16px)`

Base font size. It should be specified in pixels.

```
$v-font-size: 18px;
```

`$v-font-weight(default:[literal] 300)`

Font weight for normal fonts. The size should be given as a numeric value, not symbolic.

```
$v-font-weight: 400;
```

`$v-font-color(default: computed)`

Foreground text color, specified as any CSS color value. The default is computed from the background color so that it gives a high contrast with the background.

`$v-font-family(default:[literal] "Open Sans", sans-serif)`

Font family and fallback fonts as a comma-separated list. Font names containing spaces must be quoted. The default font Open Sans is a web font included in the Valo theme. Other used Valo fonts must be specified in the list to enable them. See Section 9.7.4, "Valo Fonts".

```
$v-font-family: "Source Sans Pro", sans-serif;
```

`$v-caption-font-size(default:[literal] round($v-font-size * 0.9))`

Font size for component captions. The value should be a pixel value.

`$v-caption-font-weight(default:[literal] max(400, $v-font-weight))`

Font weight for captions. It should be defined with a numeric value instead of symbolic.

Layout Settings

`$v-unit-size (default: round(2.3 * $v-font-size))`

This is the base size for various layout measures. It is directly used in some measures, such as button height and layout margins, while other measures are derived from it. The value must be specified in pixels, with a suitable range of 18-50.

```
$v-unit-size: 40px;
```

`$v-layout-margin-top`, `$v-layout-margin-right`, `$v-layout-margin-bottom`,
`$v-layout-margin-left` (default: `$v-unit-size`)

Layout margin sizes for all built-in layout components, when the margin is enabled with `setMargin()`, as described in Section 7.13.5, “Layout Margins”.

`$v-layout-spacing-vertical` and `$v-layout-spacing-horizontal` (default: `round($v-unit-size/3)`)

Amount of vertical or horizontal space when spacing is enabled for a layout with `setSpacing()`, as described in Section 7.13.4, “Layout Cell Spacing”.

Component Features

The following settings apply to various graphical features of some components.

`$v-border`(default:[literal]*1px solid (v-shade 0.7)*)

Border specification for the components that have a border. The thickness measure must be specified in pixels. For the border color, you can specify any CSS color or one of the `v-tint`, `v-shade`, and `v-tone` keywords described later in this section.

`$v-border-radius`(default:[literal]*4px*)

Corner radius for components that have a border. The measure must be specified in pixels.

```
$v-border-radius: 8px;
```

`$v-gradient`(default:[literal]*v-linear 8%*)

Color gradient style for components that have a gradient. The gradient style may use the following keywords: `v-linear` and `v-linear-reverse`. The opacity must be given as percentage between 0% and 100%.

```
$v-gradient: v-linear 20%;
```

`$v-bevel`(default:[literal]*inset 0 1px 0 v-tint, inset 0 -1px 0 v-shade*)

Inset shadow style to define how some components are “raised” from the background. The value follows the syntax of CSS `box-shadow`, and should be a list of insets. For the bevel color, you can specify any CSS color or one of the `v-tint`, `v-shade`, and `v-tone` keywords described later in this section.

`$v-bevel-depth`(default:[literal]*30%*)

Specifies the “depth” of the bevel shadow, as applied to one of the color keywords for the bevel style. The actual amount of tint, shade, or tone is computed from the depth.

`$v-shadow`(default:[literal]*0 2px 3px v-shade*)

Default shadow style for all components. As with `$v-bevel`, the value follows the syntax of CSS `box-shadow`, but without the `inset`. For the shadow color, you can specify any CSS color or one of the `v-tint` or `v-shade` keywords described later in this section.

`$v-shadow-opacity`(default:[literal]*5%*)

Specifies the opacity of the shadow, as applied to one of the color keywords for the shadow style. The actual amount of tint or shade is computed from the depth.

`$v-focus-style(default:[literal]0 0 0 2px rgba($v-focus-color, .5))`

Box-shadow specification for the field focus indicator. The space-separated values are the horizontal shadow position in pixels, vertical shadow position in pixels, blur distance in pixels, spread distance in pixels, and the color. The color can be any CSS color. You can only specify the color, in which case defaults for the position are used. `rgba()` or `hsla()` can be used to enable transparency.

For example, the following creates a 2 pixels wide orange outline around the field:

```
$v-focus-style: 0 0 0 2px orange;
```

`$v-focus-color(default:[literal]valo-focus-color())`

Color for the field focus indicator. The `valo-focus-color()` function computes a high-contrast color from the context, which is usually the background color. The color can be any CSS color.

`$v-animations-enabled(default:[literal>true)`

Specifies whether various CSS animations are used.

`$v-hover-styles-enabled(default:[literal>true)`

Specifies whether various `:hover` styles are used for indicating that mouse pointer hovers over an element.

`$v-disabled-opacity(default:[literal]0.5)`

Opacity of disabled components, as described in Section 6.3.3, “Enabled”.

`$v-selection-color(default:[literal]$v-focus-color)`

Color for indicating selection in selection components.

`$v-default-field-width(default:[literal]$v-unit-size * 5)`

Default width of certain field components, unless overridden with `setWidth()`.

`$v-error-indicator-color(default:[literal]#ed473b)`

Color of the component error indicator, as described in Section 5.6.1, “Error Indicator and Message”.

`$v-required-field-indicator-color(default:[literal]$v-error-indicator-color)`

Color of the required indicator in field components, as described in Section 6.4.1, “**Field** Interface”.

Color specifications for `$v-border`, `$v-bevel`, and `$v-shadow` may use, in addition to CSS colors, the following keywords:

`v-tint`

Lighter than the background color.

`v-shade`

Darker than the background color.

`v-tone`

Adaptive color specification: darker on light background and lighter on dark background. Not usable in `$v-shadow`.

For example:

```
$v-border: 1px solid v-shade;
```

You can fine-tune the contrast by giving a weight parameter in parentheses:

```
$v-border: 1px solid (v-tint 2);  
  
$v-border: 1px solid (v-tone 0.5);
```

Theme Compilation and Optimization

`$v-relative-paths`(default:[literal]*true*)

This flag specifies whether relative URL paths are relative to the currently parsed SCSS file or to the compilation root file, so that paths are correct for different resources. Vaadin theme compiler parses URL paths differently from the regular Sass compiler (Vaadin modifies relative URL paths). Use `false` for Ruby compiler and `true` for Vaadin Sass compiler.

`$v-included-components`(default: component list)

Theme optimization parameter to specify the included component themes, as described in Section 9.7.6, “Theme Optimization”.

`$v-included-additional-styles`(default:[literal]*\$v-included-components*)

Theme optimization parameter that lists the components for which the additional component styles should be included. See Section 9.7.5, “Component Styles” for more details.

9.7.3. Valo Mixins and Functions

Valo uses Sass mixins and functions heavily to compute various theme features, such as colors and shades. Also, all component styles are mixins. You can use the built-in mixins or override them. For detailed documentation of the mixins and functions, please refer to the Valo API documentation available at <http://vaadin.com/valo/api>.

9.7.4. Valo Fonts

Valo includes the following custom fonts:

- Open Sans
- Source Sans Pro
- Roboto
- Lato
- Lora

The used fonts must be specified with the `$v-font-family` parameter for Valo, in a fallback order. A font family is used in decreasing order of preference, in case a font with higher preference is not available in the browser. You can specify any font families and generic families that browsers may support. In addition to the primary font family, you can use also others in your application. To enable using the fonts included in Valo, you need to list them in the variable.

```
$v-font-family: 'Open Sans', sans-serif, 'Source Sans Pro';
```

Above, we specify Open Sans as the preferred primary font, with any sans-serif font that the browser supports as a fallback. In addition, we include the Source Sans Pro as an auxiliary font that we can use in custom rules as follows:

```
.v-label pre {  
    font-family: 'Source Sans Pro', monospace;  
}
```

This would specify using the font in any **Label** component with the `PREFORMATTED` content mode.

9.7.5. Component Styles

Many components have component-specific styles to make them smaller, bigger, and so forth. You can specify the component styles with `addStyleName()` using the constants defined in the **ValoTheme** enum.

```
table.addStyleName(ValoTheme.TABLE_COMPACT);
```

For a complete up-to-date list of component-specific styles, please refer to Vaadin API documentation on the **ValoTheme** enum. Some are also described in the component-specific styling sections.

Disabling Component Styles

Component styles are optional, but all are enabled by default. They can be enabled on per-component basis with the `$v-included-additional-styles` parameter. It defaults to `$v-included-components` and can be customized in the same way, as described in Section 9.7.6, “Theme Optimization”.

Configuration Parameters

The following variables control some common component styles:

`$v-scaling-factor—tiny`(default:[literal]0.75)

A scaling multiplier for `TINY` component styles.

`$v-scaling-factor—small`(default:[literal]0.85)

A scaling multiplier for `SMALL` component styles.

`$v-scaling-factor—large`(default:[literal]1.2)

A scaling multiplier for `LARGE` component styles.

`$v-scaling-factor—huge`(default:[literal]1.6)

A scaling multiplier for `HUGE` component styles.

9.7.6. Theme Optimization

Valo theme allows optimizing the size of the compiled theme CSS by including the rules for only the components actually used in the application. The included component styles can be specified in the `$v-included-components` variable, which by default includes all components. The variable should include a comma-separated list of component names in lower-case letters. Likewise, you can specify which additional component styles, as described in Section 9.7.5, “Component Styles”, should be included using the `$v-included-additional-styles` parameter and the same format. The list of additional styles defaults to `$v-included-components`.

For example, if your UI contains just **VerticalLayout**, **TextField**, and **Button** components, you could define the variable as follows:

```
$v-included-components:  
    verticallylayout,  
    textfield,  
    button;
```

You can use the `remove()` function reversely to remove just some component themes from the standard selection.

For example, with the following you can remove the theme definitions for the **Calendar** component:

```
$v-included-components: remove($v-included-components, calendar);
```

Note that in this case, you need to give the statement *after* the `@import` statement for the Valo theme, because it overrides a variable by using its value that is defined in the theme.

9.8. Font Icons

Font icons are icons included in a font. Fonts have many advantages over bitmap images. Browsers are usually faster in rendering fonts than loading image files. Web fonts are vector graphics, so they are scalable. As font icons are text characters, you can define their color in CSS by the regular foreground color property.

9.8.1. Loading Icon Fonts

Vaadin currently comes with one custom icon font: FontAwesome. It is automatically enabled in the Valo theme. For other themes, you need to include it with the following line in your project theme, after importing the base theme:

```
@include fonticons;
```

If you use other icon fonts, as described in Section 9.8.5, “Custom Font Icons”, and the font is not loaded by a base theme, you need to load it with a `font` mixin in Sass, as described in Section 9.9.1, “Loading Local Fonts”.

9.8.2. Basic Use

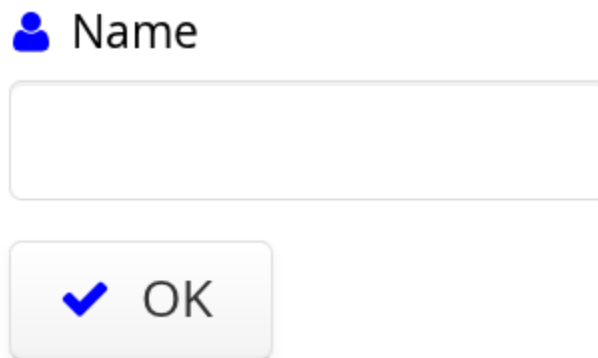
Font icons are resources of type **FontIcon**, which implements the `Resource` interface. You can use these special resources for component icons and such, but not as embedded images, for example.

Each icon has a Unicode codepoint, by which you can use it. Vaadin includes an awesome icon font, `FontAwesome`, which comes with an enumeration of all the icons included in the font.

Most typically, you set a component icon as follows:

```
TextField name = new TextField("Name");  
name.setIcon(FontAwesome.USER);  
layout.addComponent(name);  
  
// Button allows specifying icon resource in constructor  
Button ok = new Button("OK", FontAwesome.CHECK);  
layout.addComponent(ok);
```

The result is illustrated in Figure 9.10, “Basic Use of Font Icons”, with the color styling described next.

Figure 9.10. Basic Use of Font Icons

Styling the Icons

As font icons are regular text, you can specify their color with the `color` attribute in CSS to specify the foreground text color. All HTML elements that display icons in Vaadin have the `v-icon` style name.

```
.v-icon {  
    color: blue;  
}
```

If you use the font icon resources in other ways, such as in an **Image** component, the style name will be different.

9.8.3. Using Font icons in HTML

You can use font icons in HTML code, such as in a **Label**, by generating the HTML to display the icon with the `getHtml()` method.

```
Label label = new Label("I " +  
    FontAwesome.HEART.getHtml() + " Vaadin",  
    ContentMode.HTML);  
label.addStyleName("redicon");  
layout.addComponent(label);
```

The HTML code has the `v-icon` style, which you can modify in CSS:

```
.redicon .v-icon {  
    color: red;  
}
```

The result is illustrated in Figure 9.11, “Using Font Icons in Label”, with the color styling described next.

Figure 9.11. Using Font Icons in Label

You could have set the font color in the label's HTML code as well, or for all icons in the UI.

You can easily use font icons in HTML code in other ways as well. You just need to use the correct font family and then use the hex-formatted Unicode codepoint for the icon. See for example the implementation of the `getHtml()` method in **FontAwesome**:

```
@Override
public String getHtml() {
    return "<span class=\"v-icon\" style=\"font-family: \" +
        getFontFamily() + \";\>&#x\" +
        Integer.toHexString(codepoint) + \"</span>\";
}
```

9.8.4. Using Font Icons in Other Text

You can include a font icon in any text by its Unicode codepoint, which you can get with the `getCodePoint()` method. In such case, however, you need to use the same font for other text in the same string as well. The **FontAwesome** provided in Vaadin includes a basic character set.

```
TextField amount = new TextField("Amount (in " +
    new String(Character.toChars(
        FontAwesome.BTC.getCodepoint()))) +
    ")");
amount.addStyleName("awesomecaption");
layout.addComponent(amount);
```

You need to set the font family in CSS.

```
.v-caption-awesomecaption .v-captiontext {
    font-family: FontAwesome;
}
```

9.8.5. Custom Font Icons

You can easily use glyphs in existing fonts as icons, or create your own.

Creating New Icon Fonts With IcoMoon

You are free to use any of the many ways to create icons and embed them into fonts. Here, we give basic instructions for using the IcoMoon service, where you can pick icons from a large library of well-designed icons.

Font Awesome is included in IcoMoon's selection of icon libraries. Note that the codepoints of the icons are not fixed, so the **FontAwesome** enum is not compatible with such custom icon fonts.

After you have selected the icons that you want in your font, you can download them in a ZIP package. The package contains the icons in multiple formats, including WOFF, TTF, EOT, and SVG. Not all browsers support any one of them, so all are needed to support all the common browsers. Extract the `fonts` folder from the package to under your theme.

See Section 9.9.1, "Loading Local Fonts" for instructions for loading a custom font.

9.9. Custom Fonts

In addition to using the built-in fonts of the browser and the web fonts included in the Vaadin themes, you can use custom web fonts.

9.9.1. Loading Local Fonts

You can load locally served web fonts with the `font` mixin as follows:

```
@include font(MyFontFamily,
              '../..mytheme/fonts/myfontfamily');
```

The statement must be given in the `styles.scss` file *outside* the `.mytheme {}` block.

The first parameter is the name of the font family, which is used to identify the font. If the font family name contains spaces, you need to use single or double quotes around the name.

The second parameter is the base name of the font files without an extension, including a relative path. Notice that the path is relative to the base theme, where the mixin is defined, not the used theme. We recommend placing custom font files under a `fonts` folder in a theme.

Not all browsers support any single font file format, so the base name is appended with `.ttf`, `.eot`, `.woff`, or `.svg` suffix for the font file suitable for a user's browser.

9.9.2. Loading Web Fonts

You can load externally served web fonts such as Google Fonts simply by specifying the loading stylesheet for the UI with the **@StyleSheet** annotation.

For example, to load the "Cabin Sketch" font from Google Fonts:

```
@StyleSheet({"http://fonts.googleapis.com/css?family=Cabin+Sketch"})
public class MyUI extends UI {
    ...
}
```

9.9.3. Using Custom Fonts

After loaded, you can use a custom font, or actually font family, by its name in CSS and otherwise.

```
.mystyle {
    font-family: MyFontFamily;
}
```

Again, if the font family name contains spaces, you need to use single or double quotes around the name.

9.10. Responsive Themes

Vaadin includes support for responsive design which enables size range conditions in CSS selectors, allowing conditional CSS rules that respond to size changes in the browser window on the client-side.

You can use the **Responsive** extension to extend either a component, typically a layout, or the entire UI. You specify the component by the static `makeResponsive()` method.

```
// Have some component with an appropriate style name
Label c = new Label("Here be text");
c.addStyleName("myresponsive");
content.addComponent(c);
```

```
// Enable Responsive CSS selectors for the component
Responsive.makeResponsive(c);
```

You can now use width-range and height-range conditions in CSS selectors as follows:

```
/* Basic settings for all sizes */
.myresponsive {
    padding: 5px;
    line-height: 36pt;
}

/* Small size */
.myresponsive[width-range~="0-300px"] {
    background: orange;
    font-size: 16pt;
}

/* Medium size */
.myresponsive[width-range~="301px-600px"] {
    background: azure;
    font-size: 24pt;
}

/* Anything bigger */
.myresponsive[width-range~="601px-"] {
    background: palegreen;
    font-size: 36pt;
}
```

You can have overlapping size ranges, in which case all the selectors matching the current size are enabled.

Chapter 10

Binding Components to Data

10.1. Overview	319
10.2. Properties	321
10.3. Holding properties in Items	326
10.4. Creating Forms by Binding Fields to Items	328
10.5. Collecting Items in Containers	333

This chapter describes the Vaadin Data Model and shows how you can use it to bind components directly to data sources, such as database queries.

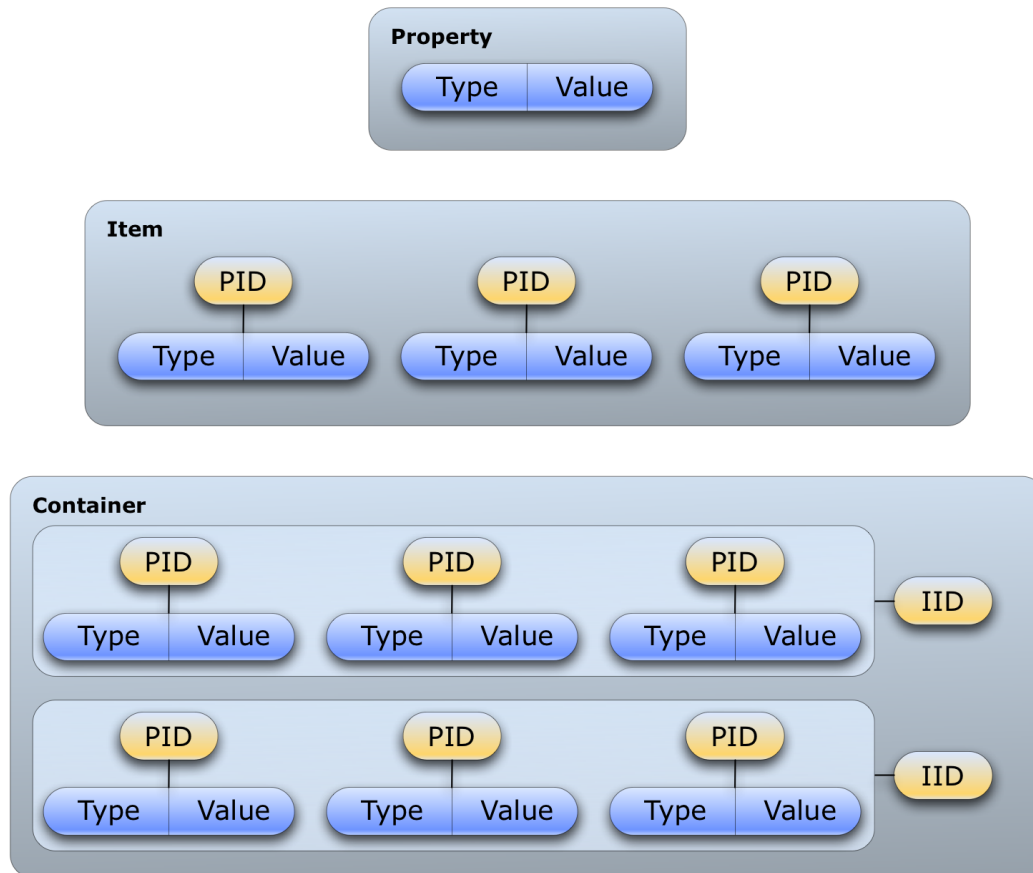
10.1. Overview

The Vaadin Data Model is one of the core concepts of the library. To allow the view (user interface components) to access the data model of an application directly, we have introduced a standard data interface.

The model allows binding user interface components directly to the data that they display and possibly allow to edit. There are three nested levels of hierarchy in the data model: *property*,

item, and *container*. Using a spreadsheet application as an analogy, these would correspond to a cell, a row, and a table, respectively.

Figure 10.1. Vaadin Data Model

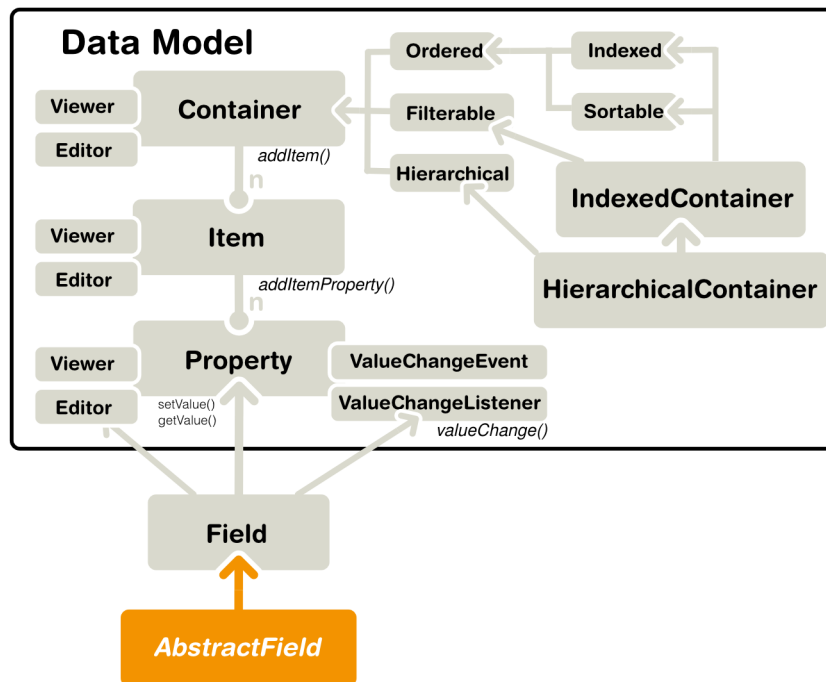


The Data Model is realized as a set of interfaces in the **com.vaadin.data** package. The package contains the **Property**, **Item**, and **Container** interfaces, along with a number of more specialized interfaces and classes.

Notice that the Data Model does not define data representation, but only interfaces. This leaves the representation fully to the implementation of the containers. The representation can be almost anything, such as a plain old Java object (POJO) structure, a filesystem, or a database query.

The Data Model is used heavily in the core user interface components of Vaadin, especially the field components, that is, components that implement the **Field** interface or more typically extend **AbstractField**, which defines many common features. A key feature of all the built-in field components is that they can either maintain their data by themselves or be bound to an external data source. The value of a field is always available through the **Property** interface. As more than one component can be bound to the same data source, it is easy to implement various viewer-editor patterns.

The relationships of the various interfaces are shown in Figure 10.2, “Interface Relationships in Vaadin Data Model”; the value change event and listener interfaces are shown only for the **Property** interface, while the notifier interfaces are omitted altogether.

Figure 10.2. Interface Relationships in Vaadin Data Model

The Data Model has many important and useful features, such as support for change notification. Especially containers have many helper interfaces, including ones that allow indexing, ordering, sorting, and filtering the data. Also **Field** components provide a number of features involving the data model, such as buffering, validation, and lazy loading.

Vaadin provides a number of built-in implementations of the data model interfaces. The built-in implementations are used as the default data models in many field components.

In addition to the built-in implementations, many data model implementations, such as containers, are available as add-ons, either from the Vaadin Directory or from independent sources. Both commercial and free implementations exist. The *JPAContainer*, described in Chapter 20, *Vaadin JPAContainer*, is the most often used commercial container add-on. The installation of add-ons is described in Chapter 18, *Using Vaadin Add-ons*. Notice that unlike with most regular add-on components, you do not need to compile a widget set for add-ons that include just data model implementations.

10.2. Properties

The `Property` interface is the base of the Vaadin Data Model. It provides a standardized API for a single data value object that can be read (get) and written (set). A property is always typed, but can optionally support data type conversions. The type of a property can be any Java class. Optionally, properties can provide value change events for following their changes.

You can set the value of a property with `setValue()` and read with `getValue()`.

In the following, we set and read the property value from a **TextField** component, which implements the `Property` interface to allow accessing the field value.

```
final TextField tf = new TextField("Name");
```

```
// Set the value
tf.setValue("The text field value");

// When the field value is edited by the user
tf.addValueChangeListener(
    new Property.ValueChangeListener() {
        public void valueChange(ValueChangeEvent event) {
            // Do something with the new value
            layout.addComponent(new Label(tf.getValue()));
        }
    });
```

Changes in the property value usually fire a **ValueChangeEvent**, which can be handled with a **ValueChangeListener**. The event object provides reference to the property with `getProperty()`. Note that its `getValue()` method returns the value with **Object** type, so you need to cast it to the proper type.

Properties are in themselves unnamed. They are collected in *items*, which associate the properties with names: the *Property Identifiers* or *PIDs*. Items can be further contained in containers and are identified with *Item Identifiers* or *IIDs*. In the spreadsheet analogy, *Property Identifiers* would correspond to column names and *Item Identifiers* to row names. The identifiers can be arbitrary objects, but must implement the `equals(Object)` and `hashCode()` methods so that they can be used in any standard Java **Collection**.

The **Property** interface can be utilized either by implementing the interface or by using some of the built-in property implementations. Vaadin includes a **Property** interface implementation for arbitrary function pairs and bean properties, with the **MethodProperty** class, and for simple object properties, with the **ObjectProperty** class, as described later.

In addition to the simple components, selection components provide their current selection as the property value. In single selection mode, the property is a single item identifier, while in multiple selection mode it is a set of item identifiers. See the documentation of the selection components for further details.

Components that can be bound to a property have an internal default data source object, typically a **ObjectProperty**, which is described later. As all such components are viewers or editors, also described later, so you can rebind a component to any data source with `setPropertyDataSource()`.

10.2.1. Property Viewers and Editors

The most important function of the **Property** as well as of the other data model interfaces is to connect classes implementing the interface directly to editor and viewer classes. This means connecting a data source (model) to a user interface component (views) to allow editing or viewing the data model.

A property can be bound to a component implementing the **Viewer** interface with `setPropertyDataSource()`.

```
// Have a data model
ObjectProperty property =
    new ObjectProperty("Hello", String.class);

// Have a component that implements Viewer
Label viewer = new Label();
```



```
// Bind it to the data
viewer.setPropertyDataSource(property);
```

You can use the same method in the **Editor** interface to bind a component that allows editing a particular property type to a property.

```
// Have a data model
ObjectProperty property =
    new ObjectProperty("Hello", String.class);

// Have a component that implements Viewer
TextField editor = new TextField("Edit Greeting");

// Bind it to the data
editor.setPropertyDataSource(property);
```

As all field components implement the **Property** interface, you can bind any component implementing the **Viewer** interface to any field, assuming that the viewer is able to view the object type of the field. Continuing from the above example, we can bind a **Label** to the **TextField** value:

```
Label viewer = new Label();
viewer.setPropertyDataSource(editor);

// The value shown in the viewer is updated immediately
// after editing the value in the editor (once it
// loses the focus)
editor.setImmediate(true);
```

If a field has validators, as described in Section 6.4.5, “Field Validation”, the validators are executed before writing the value to the property data source, or by calling the `validate()` or `commit()` for the field.

10.2.2. ObjectProperty Implementation

The **ObjectProperty** class is a simple implementation of the **Property** interface that allows storing an arbitrary Java object.

```
// Have a component that implements Viewer interface
final TextField tf = new TextField("Name");

// Have a data model with some data
String myObject = "Hello";

// Wrap it in an ObjectProperty
ObjectProperty property =
    new ObjectProperty(myObject, String.class);

// Bind the property to the component
tf.setPropertyDataSource(property);
```

10.2.3. Converting Between Property Type and Representation

Fields allow editing a certain type, such as a **String** or **Date**. The bound property, on the other hand, could have some entirely different type. Conversion between a representation edited by the field and the model defined in the property is handled with a converter that implements the `Converter` interface.

Most common type conversions, such as between string and integer, are handled by the default converters. They are created in a converter factory global in the application.

Basic Use of Converters

The `setConverter([interfacename]#Converter)#` method sets the converter for a field. The method is defined in **AbstractField**.

```
// Have an integer property
final ObjectProperty<Integer> property =
    new ObjectProperty<Integer>(42);

// Create a TextField, which edits Strings
final TextField tf = new TextField("Name");

// Use a converter between String and Integer
tf.setConverter(new StringToIntegerConverter());

// And bind the field
tf.setPropertyDataSource(property);
```

The built-in converters are the following:

Table 10.1. Built-in Converters

Converter	Representation	Model
StringToIntegerConverter	String	Integer
StringToDoubleConverter	String	Double
StringToNumberConverter	String	Number
StringToBooleanConverter	String	Boolean
StringToDateConverter	String	Date
DateToLongConverter	Date	Long

In addition, there is a **ReverseConverter** that takes a converter as a parameter and reverses the conversion direction.

If a converter already exists for a type, the `setConverter([interfacename]#Class)#` retrieves the converter for the given type from the converter factory, and then sets it for the field. This method is used implicitly when binding field to a property data source.

Implementing a Converter

A conversion always occurs between a *representation type*, edited by the field component, and a *model type*, that is, the type of the property data source. Converters implement the `Converter` interface defined in the `com.vaadin.data.util.converter` package.

For example, let us assume that we have a simple **Complex** type for storing complex values.

```
public class ComplexConverter
    implements Converter<String, Complex> {
    @Override
    public Complex convertToModel(String value, Locale locale)
        throws ConversionException {
        String parts[] =
```

```
        value.replaceAll("[\\(\\)]", "").split(",");
    if (parts.length != 2)
        throw new ConversionException(
            "Unable to parse String to Complex");
    return new Complex(Double.parseDouble(parts[0]),
        Double.parseDouble(parts[1]));
}

@Override
public String convertToPresentation(Complex value,
    Locale locale)
    throws ConversionException {
    return "("+value.getReal()+", "+value.getImag()+") ";
}

@Override
public Class<Complex> getModelType() {
    return Complex.class;
}

@Override
public Class<String> getPresentationType() {
    return String.class;
}
}
```

The conversion methods get the locale for the conversion as a parameter.

Converter Factory

If a field does not directly allow editing a property type, a default converter is attempted to create using an application-global converter factory. If you define your own converters that you wish to include in the converter factory, you need to implement one yourself. While you could implement the `ConverterFactory` interface, it is usually easier to just extend **DefaultConverterFactory**.

```
class MyConverterFactory extends DefaultConverterFactory {
    @Override
    public <PRESENTATION, MODEL> Converter<PRESENTATION, MODEL>
        createConverter(Class<PRESENTATION> presentationType,
            Class<MODEL> modelType) {
        // Handle one particular type conversion
        if (String.class == presentationType &&
            Complex.class == modelType)
            return (Converter<PRESENTATION, MODEL>)
                new ComplexConverter();

        // Default to the supertype
        return super.createConverter(presentationType,
            modelType);
    }
}

// Use the factory globally in the application
Application.getCurrentApplication().setConverterFactory(
    new MyConverterFactory());
```

10.3. Holding properties in Items

The **Item** interface provides access to a set of named properties. Each property is identified by a *property identifier* (PID) and a reference to such a property can be queried from an **Item** with `getItemProperty()` using the identifier.

Examples on the use of items include rows in a **Table**, with the properties corresponding to table columns, nodes in a **Tree**, and the the data bound to a **Form**, with item's properties bound to individual form fields.

Items are generally equivalent to objects in the object-oriented model, but with the exception that they are configurable and provide an event handling mechanism. The simplest way to utilize **Item** interface is to use existing implementations. Provided utility classes include a configurable property set (**PropertysetItem**) and a bean-to-item adapter (**BeanItem**). Also, a **Form** implements the interface and can therefore be used directly as an item.

In addition to being used indirectly by many user interface components, items provide the basic data model underlying the **Form** component. In simple cases, forms can even be generated automatically from items. The properties of the item correspond to the fields of the form.

The **Item** interface defines inner interfaces for maintaining the item property set and listening changes made to it. **PropertySetChangeEvent** events can be emitted by a class implementing the **PropertySetChangeNotifier** interface. They can be received through the **PropertySetChangeListener** interface.

10.3.1. Wrapping a Bean in a BeanItem

The **BeanItem** implementation of the **Item** interface is a wrapper for Java Bean objects. In fact, only the setters and getters are required while serialization and other bean features are not, so you can wrap almost any POJOs with minimal requirements.

```
// Here is a bean (or more exactly a POJO)
class Person {
    String name;
    int    age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age.intValue();
    }
}

// Create an instance of the bean
Person bean = new Person();
```

```
// Wrap it in a BeanItem
BeanItem<Person> item = new BeanItem<Person>(bean);

// Bind it to a component
Form form = new Form();
form.setItemDataSource(item);
```

You can use the `getBean()` method to get a reference to the underlying bean.

Nested Beans

You may often have composite classes where one class "has a" another class. For example, consider the following **Planet** class which "has a" discoverer:

```
// Here is a bean with two nested beans
public class Planet implements Serializable {
    String name;
    Person discoverer;

    public Planet(String name, Person discoverer) {
        this.name = name;
        this.discoverer = discoverer;
    }

    ... getters and setters ...
}

...
// Create an instance of the bean
Planet planet = new Planet("Uranus",
    new Person("William Herschel", 1738));
```

When shown in a **Form**, for example, you would want to list the properties of the nested bean along the properties of the composite bean. You can do that by binding the properties of the nested bean individually with a **MethodProperty** or **NestedMethodProperty**. You should usually hide the nested bean from binding as a property by listing only the bound properties in the constructor.

```
// Wrap it in a BeanItem and hide the nested bean property
BeanItem<Planet> item = new BeanItem<Planet>(planet,
    new String[]{"name"});

// Bind the nested properties.
// Use NestedMethodProperty to bind using dot notation.
item.addItemProperty("discoverername",
    new NestedMethodProperty(planet, "discoverer.name"));

// The other way is to use regular MethodProperty.
item.addItemProperty("discovererborn",
    new MethodProperty<Person>(planet.getDiscoverer(),
        "born"));
```

The difference is that **NestedMethodProperty** does not access the nested bean immediately but only when accessing the property values, while when using **MethodProperty** the nested bean is accessed when creating the method property. The difference is only significant if the nested bean can be null or be changed later.

You can use such a bean item for example in a **Form** as follows:

```
// Bind it to a component
Form form = new Form();
form.setItemDataSource(item);

// Nicer captions
form.getField("discoverername").setCaption("Discoverer");
form.getField("discovererborn").setCaption("Born");
```

Figure 10.3. A Form with Nested Bean Properties

Name	Uranus
Discoverer	William Herschel
Born	1738

The **BeanContainer** and **BeanItemContainer** allow easy definition of nested bean properties with `addNestedContainerProperty()`, as described in the section called “Nested Properties”.

10.4. Creating Forms by Binding Fields to Items

Most applications in existence have forms of some sort. Forms contain fields, which you want to bind to a data source, an item in the Vaadin data model. **FieldGroup** provides an easy way to bind fields to the properties of an item. You can use it by first creating a layout with some fields, and then call it to bind the fields to the data source. You can also let the **FieldGroup** create the fields using a field factory. It can also handle commits. Notice that **FieldGroup** is not a user interface component, so you can not add it to a layout.

10.4.1. Simple Binding

Let us start with a data model that has an item with a couple of properties. The item could be any item type, as described earlier.

```
// Have an item
PropertysetItem item = new PropertysetItem();
item.addItemProperty("name", new ObjectProperty<String>("Zaphod"));
item.addItemProperty("age", new ObjectProperty<Integer>(42));
```

Next, you would design a form for editing the data. The **FormLayout** (Section 7.5, “**FormLayout**”) is ideal for forms, but you could use any other layout as well.

```
// Have some layout and create the fields
FormLayout form = new FormLayout();

TextField nameField = new TextField("Name");
form.addComponent(nameField);

TextField ageField = new TextField("Age");
form.addComponent(ageField);
```

Then, we can bind the fields to the data as follows:

```
// Now create the binder and bind the fields
FieldGroup binder = new FieldGroup(item);
binder.bind(nameField, "name");
binder.bind(ageField, "age");
```

The above way of binding is not different from simply calling `setPropertyDataSource()` for the fields. It does, however, register the fields in the field group, which for example enables buffering or validation of the fields using the field group, as described in Section 10.4.4, “Buffering Forms”.

Next, we consider more practical uses for a **FieldGroup**.

10.4.2. Using a **FieldFactory** to Build and Bind Fields

Using the `buildAndBind()` methods, **FieldGroup** can create fields for you using a `FieldGroupFieldFactory`, but you still have to add them to the correct position in your layout.

```
// Have some layout
FormLayout form = new FormLayout();

// Now create a binder that can also create the fields
// using the default field factory
FieldGroup binder = new FieldGroup(item);
form.addComponent(binder.buildAndBind("Name", "name"));
form.addComponent(binder.buildAndBind("Age", "age"));
```

10.4.3. Binding Member Fields

The `bindMemberFields()` method in **FieldGroup** uses reflection to bind the properties of an item to field components that are member variables of a class. Hence, if you implement a form as a class with the fields stored as member variables, you can use this method to bind them super-easy.

The item properties are mapped to the members by the property ID and the name of the member variable. If you want to map a property with a different ID to a member, you can use the `@PropertyId` annotation for the member, with the property ID as the parameter.

For example:

```
// Have an item
PropertysetItem item = new PropertysetItem();
item.addItemProperty("name", new ObjectProperty<String>("Zaphod"));
item.addItemProperty("age", new ObjectProperty<Integer>(42));

// Define a form as a class that extends some layout
class MyForm extends FormLayout {
    // Member that will bind to the "name" property
    TextField name = new TextField("Name");

    // Member that will bind to the "age" property
    @PropertyId("age")
    TextField ageField = new TextField("Age");

    public MyForm() {
        // Customize the layout a bit
        setSpacing(true);

        // Add the fields
        addComponent(name);
        addComponent(ageField);
    }
}
```

```
// Create one
MyForm form = new MyForm();

// Now create a binder that can also creates the fields
// using the default field factory
FieldGroup binder = new FieldGroup(item);
binder.bindMemberFields(form);

// And the form can be used in an higher-level layout
layout.addComponent(form);
```

Encapsulating in CustomComponent

Using a **CustomComponent** can be better for hiding the implementation details than extending a layout. Also, the use of the **FieldGroup** can be encapsulated in the form class.

Consider the following as an alternative for the form implementation presented earlier:

```
// A form component that allows editing an item
class MyForm extends CustomComponent {
    // Member that will bind to the "name" property
    TextField name = new TextField("Name");

    // Member that will bind to the "age" property
    @PropertyId("age")
    TextField ageField = new TextField("Age");

    public MyForm(Item item) {
        FormLayout layout = new FormLayout();
        layout.addComponent(name);
        layout.addComponent(ageField);

        // Now use a binder to bind the members
        FieldGroup binder = new FieldGroup(item);
        binder.bindMemberFields(this);

        setCompositionRoot(layout);
    }
}

// And the form can be used as a component
layout.addComponent(new MyForm(item));
```

10.4.4. Buffering Forms

Just like for individual fields, as described in Section 6.4.4, “Field Buffering”, a **FieldGroup** can handle buffering the form content so that it is written to the item data source only when `commit()` is called for the group. It runs validation for all fields in the group and writes their values to the item data source only if all fields pass the validation. Edits can be discarded, so that the field values are reloaded from the data source, by calling `discard()`. Buffering is enabled by default, but can be disabled by calling `setBuffered(false)` for the **FieldGroup**.

```
// Have an item of some sort
final PropertysetItem item = new PropertysetItem();
item.addItemProperty("name", new ObjectProperty<String>("Q"));
item.addItemProperty("age", new ObjectProperty<Integer>(42));
```



```
// Have some layout and create the fields
Panel form = new Panel("Buffered Form");
form.setContent(new FormLayout());

// Build and bind the fields using the default field factory
final FieldGroup binder = new FieldGroup(item);
form.addComponent(binder.buildAndBind("Name", "name"));
form.addComponent(binder.buildAndBind("Age", "age"));

// Enable buffering (actually enabled by default)
binder.setBuffered(true);

// A button to commit the buffer
form.addComponent(new Button("OK", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        try {
            binder.commit();
            Notification.show("Thanks!");
        } catch (CommitException e) {
            Notification.show("You fail!");
        }
    }
}));

// A button to discard the buffer
form.addComponent(new Button("Discard", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        binder.discard();
        Notification.show("Discarded!");
    }
}));
```

10.4.5. Binding Fields to a Bean

The **BeanFieldGroup** makes it easier to bind fields to a bean. It also handles binding to nested beans properties. The build a field bound to a nested bean property, identify the property with dot notation. For example, if a **Person** bean has a `address` property with an **Address** type, which in turn has a `street` property, you could build a field bound to the property with `buildAndBind("Street", "address.street")`.

The input to fields bound to a bean can be validated using the Java Bean Validation API, as described in Section 10.4.6, “Bean Validation”. The **BeanFieldGroup** automatically adds a **BeanValidator** to every field if a bean validation implementation is included in the classpath.

10.4.6. Bean Validation

Vaadin allows using the Java Bean Validation API 1.0 (JSR-303) for validating input from fields bound to bean properties before the values are committed to the bean. The validation is done based on annotations on the bean properties, which are used for creating the actual validators automatically. See Section 6.4.5, “Field Validation” for general information about validation.

Using bean validation requires an implementation of the Bean Validation API, such as Hibernate Validator (`hibernate-validator-4.2.0.Final.jar` or later) or Apache Bean Validation. The implementation JAR must be included in the project classpath when using the bean validation, or otherwise an internal error is thrown.

Bean validation is especially useful when persisting entity beans with the Vaadin JPAContainer, described in Chapter 20, *Vaadin JPAContainer*.

Annotations

The validation constraints are defined as annotations. For example, consider the following bean:

```
// Here is a bean
public class Person implements Serializable {
    @NotNull
    @javax.validation.constraints.Size(min=2, max=10)
    String name;

    @Min(1)
    @Max(130)
    int age;

    // ... setters and getters ...
}
```

For a complete list of allowed constraints for different data types, please see the Bean Validation API documentation.

Validating the Beans

Validating a bean is done with a **BeanValidator**, which you initialize with the name of the bean property it should validate and add it to the editor field.

In the following example, we validate a single unbuffered field:

```
Person bean = new Person("Mung bean", 100);
BeanItem<Person> item = new BeanItem<Person> (bean);

// Create an editor bound to a bean field
TextField firstName = new TextField("First Name",
    item.getItemProperty("name"));

// Add the bean validator
firstName.addValidator(new BeanValidator(Person.class, "name"));

firstName.setImmediate(true);
layout.addComponent(firstName);
```

In this case, the validation is done immediately after focus leaves the field. You could do the same for the other field as well.

Bean validators are automatically created when using a **BeanFieldGroup**.

```
// Have a bean
Person bean = new Person("Mung bean", 100);

// Form for editing the bean
final BeanFieldGroup<Person> binder =
    new BeanFieldGroup<Person>(Person.class);
binder.setItemDataSource(bean);
layout.addComponent(binder.buildAndBind("Name", "name"));
layout.addComponent(binder.buildAndBind("Age", "age"));

// Buffer the form content
```

```
binder.setBuffered(true);
layout.addComponent(new Button("OK", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        try {
            binder.commit();
        } catch (CommitException e) {
        }
    }
}));
```

Locale Setting for Bean Validation

The validation error messages are defined in the bean validation implementation, in a `ValidationMessages.properties` file. The message is shown in the language specified with the locale setting for the form. The default language is English, but for example Hibernate Validator contains translations of the messages for a number of languages. If other languages are needed, you need to provide a translation of the properties file.

10.5. Collecting Items in Containers

The **Container** interface is the highest containment level of the Vaadin data model, for containing items (rows) which in turn contain properties (columns). Containers can therefore represent tabular data, which can be viewed in a **Table** or some other selection component, as well as hierarchical data.

The items contained in a container are identified by an *item identifier* or *IID*, and the properties by a *property identifier* or *PID*.

10.5.1. Basic Use of Containers

The basic use of containers involves creating one, adding items to it, and binding it as a container data source of a component.

Default Containers and Delegation

Before saying anything about creation of containers, it should be noted that all components that can be bound to a container data source are by default bound to a default container. For example, **Table** is bound to a **IndexedContainer**, **Tree** to a **HierarchicalContainer**, and so forth.

All of the user interface components using containers also implement the relevant container interfaces themselves, so that the access to the underlying data source is delegated through the component.

```
// Create a table with one column
Table table = new Table("My Table");
table.addContainerProperty("col1", String.class, null);

// Access items and properties through the component
table.addItem("row1"); // Create item by explicit ID
Item item1 = table.getItem("row1");
Property property1 = item1.getItemProperty("col1");
property1.setValue("some given value");

// Equivalent access through the container
Container container = table.getContainerDataSource();
```

```
container.addItem("row2");
Item item2 = container.getItem("row2");
Property property2 = item2.getItemProperty("col1");
property2.setValue("another given value");
```

Creating and Binding a Container

A container is created and bound to a component as follows:

```
// Create a container of some type
Container container = new IndexedContainer();

// Initialize the container as required by the container type
container.addContainerProperty("name", String.class, "none");
container.addContainerProperty("volume", Double.class, 0.0);

... add items ...

// Bind it to a component
Table table = new Table("My Table");
table.setContainerDataSource(container);
```

Most components that can be bound to a container allow passing it also in the constructor, in addition to using `setContainerDataSource()`. Creation of the container depends on its type. For some containers, such as the **IndexedContainer**, you need to define the contained properties (columns) as was done above, while some others determine them otherwise. The definition of a property with `addContainerProperty()` requires a unique property ID, type, and a default value. You can also give `null`. If the container of a component is replaced and the new container contains a different set of columns, such as a property with the same ID but a different data type, the component should be reinitialized. For a table or grid, it means redefining their columns.

Vaadin has a several built-in in-memory container implementations, such as **IndexedContainer** and **BeanItemContainer**, which are easy to use for setting up nonpersistent data storages. For persistent data, either the built-in **SQLContainer** or the **JPAContainer** add-on container can be used.

Adding Items and Accessing Properties

Items can be added to a container with the `addItem()` method. The parameterless version of the method automatically generates the item ID.

```
// Create an item
Object itemId = container.addItem();
```

Properties can be requested from container by first requesting an item with `getItem()` and then getting the properties from the item with `getItemProperty()`.

```
// Get the item object
Item item = container.getItem(itemId);

// Access a property in the item
Property<String> nameProperty =
    item.getItemProperty("name");

// Do something with the property
nameProperty.setValue("box");
```

You can also get a property directly by the item and property ids with `getContainerProperty()`.

```
container.getContainerProperty(itemId, "volume").setValue(5.0);
```

Adding Items by Given ID

Some containers, such as **IndexedContainer** and **HierarchicalContainer**, allow adding items by a given ID, which can be any **Object**.

```
Item item = container.addItem("agivenid");
item.getItemProperty("name").setValue("barrel");
Item.getItemProperty("volume").setValue(119.2);
```

Notice that the actual item *is not* given as a parameter to the method, only its ID, as the interface assumes that the container itself creates all the items it contains. Some container implementations can provide methods to add externally created items, and they can even assume that the item ID object is also the item itself. Lazy containers might not create the item immediately, but lazily when it is accessed by its ID.

10.5.2. Container Subinterfaces

The **Container** interface contains inner interfaces that container implementations can implement to fulfill different features required by components that present container data.

`Container.Filterable`

Filterable containers allow filtering the contained items by filters, as described in Section 10.5.7, “**Filterable** Containers”.

`Container.Hierarchical`

Hierarchical containers allow representing hierarchical relationships between items and are required by the **Tree** and **TreeTable** components. The **HierarchicalContainer** is a built-in in-memory container for hierarchical data, and is used as the default container for the tree components. The **FilesystemContainer** provides access to browsing the content of a file system. Also **JPAContainer** is hierarchical, as described in Section 20.4.4, “Hierarchical Container”.

`Container.Indexed`

An indexed container allows accessing items by an index number, not just their item ID. This feature is required by some components, especially **Table**, which needs to provide lazy access to large containers. The **IndexedContainer** is a basic in-memory implementation, as described in Section 10.5.3, “**IndexedContainer**”.

`Container.Ordered`

An ordered container allows traversing the items in successive order in either direction. Most built-in containers are ordered.

`Container.SimpleFilterable`

This interface enables filtering a container by string matching with `addContainerFilter()`. The filtering is done by either searching the given string anywhere in a property value, or as its prefix.

`Container.Sortable`

A sortable container is required by some components that allow sorting the content, such as **Table**, where the user can click a column header to sort the table by the

column. Some other components, such as **Calendar**, may require that the content is sorted to be able to display it properly. Depending on the implementation, sorting can be done only when the `sort()` method is called, or the container is automatically kept in order according to the last call of the method.

See the API documentation for a detailed description of the interfaces.

10.5.3. IndexedContainer

The **IndexedContainer** is an in-memory container that implements the `Indexed` interface to allow referencing the items by an index. **IndexedContainer** is used as the default container in most selection components in Vaadin.

The properties need to be defined with `addContainerProperty()`, which takes the property ID, type, and a default value. This must be done before any items are added to the container.

```
// Create the container
IndexedContainer container = new IndexedContainer();

// Define the properties (columns)
container.addContainerProperty("name", String.class, "noname");
container.addContainerProperty("volume", Double.class, -1.0d);

// Add some items
Object content[][] = { {"jar", 2.0}, {"bottle", 0.75},
                       {"can", 1.5}};
for (Object[] row: content) {
    Item newItem = container.getItem(container.addItem());
    newItem.getItemProperty("name").setValue(row[0]);
    newItem.getItemProperty("volume").setValue(row[1]);
}
```

New items are added with `addItem()`, which returns the item ID of the new item, or by giving the item ID as a parameter as was described earlier. Note that the **Table** component, which has **IndexedContainer** as its default container, has a convenience `addItem()` method that allows adding items as object vectors containing the property values.

The container implements the `Container.Indexed` feature to allow accessing the item IDs by their index number, with `getIdByIndex()`, etc. The feature is required mainly for internal purposes of some components, such as **Table**, which uses it to enable lazy transmission of table data to the client-side.

10.5.4. BeanContainer

The **BeanContainer** is an in-memory container for JavaBean objects. Each contained bean is wrapped inside a **BeanItem** wrapper. The item properties are determined automatically by inspecting the getter and setter methods of the class. This requires that the bean class has public visibility, local classes for example are not allowed. Only beans of the same type can be added to the container.

The generic has two parameters: a bean type and an item identifier type. The item identifiers can be obtained by defining a custom resolver, using a specific item property for the IDs, or by giving item IDs explicitly. As such, it is more general than the **BeanItemContainer**, which uses the bean object itself as the item identifier, making the use usually simpler. Managing the item IDs makes **BeanContainer** more complex to use, but it is necessary in some cases where the `equals()` or `hashCode()` methods have been reimplemented in the bean.

```
// Here is a JavaBean
public class Bean implements Serializable {
    String name;
    double energy; // Energy content in kJ/100g

    public Bean(String name, double energy) {
        this.name = name;
        this.energy = energy;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getEnergy() {
        return energy;
    }

    public void setEnergy(double energy) {
        this.energy = energy;
    }
}

void basic(VerticalLayout layout) {
    // Create a container for such beans with
    // strings as item IDs.
    BeanContainer<String, Bean> beans =
        new BeanContainer<String, Bean>(Bean.class);

    // Use the name property as the item ID of the bean
    beans.setBeanIdProperty("name");

    // Add some beans to it
    beans.addBean(new Bean("Mung bean", 1452.0));
    beans.addBean(new Bean("Chickpea", 686.0));
    beans.addBean(new Bean("Lentil", 1477.0));
    beans.addBean(new Bean("Common bean", 129.0));
    beans.addBean(new Bean("Soybean", 1866.0));

    // Bind a table to it
    Table table = new Table("Beans of All Sorts", beans);
    layout.addComponent(table);
}
```

To use explicit item IDs, use the methods `addItem(Object, Object)`, `addItemAfter(Object, Object, Object)`, and `addItemAt(int, Object, Object)`.

It is not possible to add additional properties to the container, except properties in a nested bean.

Nested Properties

If you have a nested bean with an 1:1 relationship inside a bean type contained in a **BeanContainer** or **BeanItemContainer**, you can add its properties to the container by spe-

cifying them with `addNestedContainerProperty()`. The feature is defined at the level of **AbstractBeanContainer**.

As with the bean in a bean container, also a nested bean must have public visibility or otherwise an access exception is thrown. An intermediate reference from a bean in the bean container to a nested bean may have a null value.

For example, let us assume that we have the following two beans with the first one nested inside the second one.

```
/** Bean to be nested */
public class EqCoord implements Serializable {
    double rightAscension; /* In angle hours */
    double declination;    /* In degrees    */

    ... setters and getters for the properties ...
}

/** Bean referencing a nested bean */
public class Star implements Serializable {
    String name;
    EqCoord equatorial; /* Nested bean */

    ... setters and getters for the properties ...
}
```

After creating the container, you can declare the nested properties by specifying their property identifiers with the `addNestedContainerProperty()` in dot notation.

```
// Create a container for beans
BeanItemContainer<Star> stars =
    new BeanItemContainer<Star>(Star.class);

// Declare the nested properties to be used in the container
stars.addNestedContainerProperty("equatorial.rightAscension");
stars.addNestedContainerProperty("equatorial.declination");

// Add some items
stars.addBean(new Star("Sirius", new EqCoord(6.75, 16.71611)));
stars.addBean(new Star("Polaris", new EqCoord(2.52, 89.26417)));

// Here the nested bean reference is null
stars.addBean(new Star("Vega", null));
```

If you bind such a container to a **Table**, you probably also need to set the column headers. Notice that the entire nested bean itself is still a property in the container and would be displayed in its own column. The `toString()` method is used for obtaining the displayed value, which is by default an object reference. You normally do not want this, so you can hide the column with `setVisibleColumns()`.

```
// Put them in a table
Table table = new Table("Stars", stars);
table.setColumnHeader("equatorial.rightAscension", "RA");
table.setColumnHeader("equatorial.declination", "Decl");
table.setPageLength(table.size());

// Have to set explicitly to hide the "equatorial" property
table.setVisibleColumns("name",
    "equatorial.rightAscension", "equatorial.declination");
```


The resulting table is shown in Figure 10.4, “**Table Bound to a BeanContainer with Nested Properties**”.

Figure 10.4. Table Bound to a BeanContainer with Nested Properties

name	RA	Decl
Sirius	6,75	16,716
Polaris	2,52	89,264
Vega		

The bean binding in **AbstractBeanContainer** normally uses the **MethodProperty** implementation of the **Property** interface to access the bean properties using the setter and getter methods. For nested properties, the **NestedMethodProperty** implementation is used.

10.5.5. BeanItemContainer

BeanItemContainer is a container for JavaBean objects where each bean is wrapped inside a **BeanItem** wrapper. The item properties are determined automatically by inspecting the getter and setter methods of the class. This requires that the bean class has public visibility, local classes for example are not allowed. Only beans of the same type can be added to the container.

BeanItemContainer is a specialized version of the **BeanContainer** described in Section 10.5.4, “**BeanContainer**”. It uses the bean itself as the item identifier, which makes it a bit easier to use than **BeanContainer** in many cases. The latter is, however, needed if the bean has reimplemented the `equals()` or `hashCode()` methods.

Let us revisit the example given in Section 10.5.4, “**BeanContainer**” using the **BeanItemContainer**.

```
// Create a container for the beans
BeanItemContainer<Bean> beans =
    new BeanItemContainer<Bean>(Bean.class);

// Add some beans to it
beans.addBean(new Bean("Mung bean", 1452.0));
beans.addBean(new Bean("Chickpea", 686.0));
```

```
beans.addBean(new Bean("Lentil", 1477.0));
beans.addBean(new Bean("Common bean", 129.0));
beans.addBean(new Bean("Soybean", 1866.0));

// Bind a table to it
Table table = new Table("Beans of All Sorts", beans);
```

It is not possible to add additional properties to a **BeanItemContainer**, except properties in a nested bean, as described in Section 10.5.4, “**BeanContainer**”.

10.5.6. GeneratedPropertyContainer

GeneratedPropertyContainer is a container wrapper that allows defining generated values for properties (columns). The generated properties can shadow properties with the same IDs in the wrapped container. Removing a property from the wrapper hides it.

The container is especially useful with **Grid**, which does not support generated columns or hiding columns like **Table** does.

Wrapping a Container

A container to be wrapped must be a `Container.Indexed`. It can optionally also implement `Container.Sortable` or `Container.Filterable` to enable sorting and filtering the container, respectively.

For example, let us consider the following container with some regular columns:

```
IndexedContainer container = new IndexedContainer();
container.addContainerProperty("firstname", String.class, null);
container.addContainerProperty("lastname", String.class, null);
container.addContainerProperty("born", Integer.class, null);
container.addContainerProperty("died", Integer.class, null);

// Wrap it
GeneratedPropertyContainer gpcontainer =
    new GeneratedPropertyContainer(container);
```

Generated Properties

Now, you can add generated properties in the container with `addGeneratedProperty()` by specifying a property ID and a `PropertyValueGenerator`. The method takes the ID of the generated property as first parameter; you can use a same ID as in the wrapped container to shadow its properties.

You need to implement `getType()`, which must return the class object of the value type of the property, and `getValue()`, which returns the property value for the given item. The item ID and the property ID of the generated property are also given in case they are needed. You can access other properties of the item to compute the property value.

```
gpcontainer.addGeneratedProperty("lived",
    new PropertyValueGenerator<Integer>() {
        @Override
        public Integer getValue(Item item, Object itemId,
                                Object propertyId) {
            int born = (Integer)
                item.getItemProperty("born").getValue();
            int died = (Integer)
```

```
        item.getItemProperty("died").getValue();
        return Integer.valueOf(died - born);
    }

    @Override
    public Class<Integer> getType() {
        return Integer.class;
    }
});
```

You can access other items in the container, also their generated properties, although you should beware of accidental recursion.

Using GeneratedPropertyContainer

Finally, you need to bind the **GeneratedPropertyContainer** to the component instead of the wrapped container.

```
Grid grid = new Grid(gpcontainer);
```

When using **GeneratedPropertyContainer** in **Grid**, notice that generated columns are read-only, so you can not add grid rows with `addRow()`. In editable mode, editor fields are not generated for generated columns.

Sorting

Even though the **GeneratedPropertyContainer** implements `Container.Sortable`, the wrapped container must also support it or otherwise sorting is disabled. Also, the generated properties are not normally sortable, but require special handling to enable sorting.

10.5.7. Filterable Containers

Containers that implement the **Container.Filterable** interface can be filtered. For example, the built-in **IndexedContainer** and the bean item containers implement it. Filtering is typically used for filtering the content of a **Table**.

Filters implement the **Filter** interface and you add them to a filterable container with the `addContainerFilter()` method. Container items that pass the filter condition are kept and shown in the filterable component.

```
Filter filter = new SimpleStringFilter("name",
    "Douglas", true, false);
table.addContainerFilter(filter);
```

If multiple filters are added to a container, they are evaluated using the logical AND operator so that only items that are passed by all the filters are kept.

Atomic and Composite Filters

Filters can be classified as *atomic* and *composite*. Atomic filters, such as **SimpleStringFilter**, define a single condition, usually for a specific container property. Composite filters make filtering decisions based on the result of one or more other filters. The built-in composite filters implement the logical operators AND, OR, or NOT.

For example, the following composite filter would filter out items where the `name` property contains the name "Douglas" somewhere *or* where the `age` property has value less than 42. The properties must have **String** and **Integer** types, respectively.

```
filter = new Or(new SimpleStringFilter("name",  
    "Douglas", true, false),  
    new Compare.Less("age", 42));
```

Built-In Filter Types

The built-in filter types are the following:

SimpleStringFilter

Passes items where the specified property, that must be of **String** type, contains the given *filterString* as a substring. If *ignoreCase* is *true*, the search is case insensitive. If the *onlyMatchPrefix* is *true*, the substring may only be in the beginning of the string, otherwise it may be elsewhere as well.

IsNull

Passes items where the specified property has null value. For in-memory filtering, a simple `==` check is performed. For other containers, the comparison implementation is container dependent, but should correspond to the in-memory null check.

Equal, Greater, Less, GreaterOrEqual, and LessOrEqual

The comparison filter implementations compare the specified property value to the given constant and pass items for which the comparison result is true. The comparison operators are included in the abstract **Compare** class.

For the **Equal** filter, the `equals()` method for the property is used in built-in in-memory containers. In other types of containers, the comparison is container dependent and may use, for example, database comparison operations.

For the other filters, the property value type must implement the **Comparable** interface to work with the built-in in-memory containers. Again for the other types of containers, the comparison is container dependent.

And and Or

These logical operator filters are composite filters that combine multiple other filters.

Not

The logical unary operator filter negates which items are passed by the filter given as the parameter.

Implementing Custom Filters

A custom filter needs to implement the **Container.Filter** interface.

A filter can use a single or multiple properties for the filtering logic. The properties used by the filter must be returned with the `appliesToProperty()` method. If the filter applies to a user-defined property or properties, it is customary to give the properties as the first argument for the constructor of the filter.

```
class MyCustomFilter implements Container.Filter {
    protected String propertyId;
    protected String regex;

    public MyCustomFilter(String propertyId, String regex) {
        this.propertyId = propertyId;
        this.regex      = regex;
    }

    /** Tells if this filter works on the given property. */
    @Override
    public boolean appliesToProperty(Object propertyId) {
        return propertyId != null &&
            propertyId.equals(this.propertyId);
    }
}
```

The actual filtering logic is done in the `passesFilter()` method, which simply returns `true` if the item should pass the filter and `false` if it should be filtered out.

```
/** Apply the filter on an item to check if it passes. */
@Override
public boolean passesFilter(Object itemId, Item item)
    throws UnsupportedOperationException {
    // Acquire the relevant property from the item object
    Property p = item.getItemProperty(propertyId);

    // Should always check validity
    if (p == null || !p.getType().equals(String.class))
        return false;
    String value = (String) p.getValue();

    // The actual filter logic
    return value.matches(regex);
}
}
```

You can use such a custom filter just like any other:

```
c.addContainerFilter(
    new MyCustomFilter("Name", (String) tf.getValue()));
```

Chapter 11

Vaadin SQLContainer

11.1. Architecture	346
11.2. Getting Started with SQLContainer	346
11.3. Filtering and Sorting	347
11.4. Editing	348
11.5. Caching, Paging and Refreshing	350
11.6. Referencing Another SQLContainer	351
11.7. Making Freeform Queries	352
11.8. Non-Implemented Methods	353
11.9. Known Issues and Limitations	354

Vaadin SQLContainer is a container implementation that allows easy and customizable access to data stored in various SQL-speaking databases.

SQLContainer supports two types of database access. Using **TableQuery**, the pre-made query generators will enable fetching, updating, and inserting data directly from the container into a database table - automatically, whereas **FreeformQuery** allows the developer to use their own, probably more complex query for fetching data and their own optional implementations for writing, filtering and sorting support - item and property handling as well as lazy loading will still be handled automatically.

In addition to the customizable database connection options, SQLContainer also extends the Vaadin **Container** interface to implement more advanced and more database-oriented filtering

rules. Finally, the add-on also offers connection pool implementations for JDBC connection pooling and JEE connection pooling, as well as integrated transaction support; auto-commit mode is also provided.

The purpose of this section is to briefly explain the architecture and some of the inner workings of SQLContainer. It will also give the readers some examples on how to use SQLContainer in their own applications. The requirements, limitations and further development ideas are also discussed.

SQLContainer is available from the Vaadin Directory under the same unrestrictive Apache License 2.0 as the Vaadin Framework itself.

11.1. Architecture

The architecture of SQLContainer is relatively simple. **SQLContainer** is the class implementing the Vaadin **Container** interfaces and providing access to most of the functionality of this add-on. The standard Vaadin **Property** and **Item** interfaces have been implemented as the **ColumnProperty** and **RowItem** classes. Item IDs are represented by **RowId** and **TemporaryRowId** classes. The **RowId** class is built based on the primary key columns of the connected database table or query result.

In the connection package, the **JDBCConnectionPool** interface defines the requirements for a connection pool implementation. Two implementations of this interface are provided: **SimpleJDBCConnectionPool** provides a simple yet very usable implementation to pool and access JDBC connections. **J2EEConnectionPool** provides means to access J2EE DataSources.

The query package contains the **QueryDelegate** interface, which defines everything the SQLContainer needs to enable reading and writing data to and from a database. As discussed earlier, two implementations of this interface are provided: **TableQuery** for automatic read-write support for a database table, and **FreeformQuery** for customizing the query, sorting, filtering and writing; this is done by implementing relevant methods of the **FreeformStatementDelegate** interface.

The query package also contains **Filter** and **OrderBy** classes which have been written to provide an alternative to the standard Vaadin container filtering and make sorting non-String properties a bit more user friendly.

Finally, the generator package contains a **SQLGenerator** interface, which defines the kind of queries that are required by the **TableQuery** class. The provided implementations include support for HSQLDB, MySQL, PostgreSQL (**DefaultSQLGenerator**), Oracle (**OracleGenerator**) and Microsoft SQL Server (**MSSQLGenerator**). A new or modified implementations may be provided to gain compatibility with older versions or other database servers.

For further details, please refer to the SQLContainer API documentation.

11.2. Getting Started with SQLContainer

Getting development going with the SQLContainer is easy and quite straight-forward. The purpose of this section is to describe how to create the required resources and how to fetch data from and write data to a database table attached to the container.

11.2.1. Creating a connection pool

First, we need to create a connection pool to allow the SQLContainer to connect to a database. Here we will use the **SimpleJDBCConnectionPool**, which is a basic implementation of connection pooling with JDBC data sources. In the following code, we create a connection pool that uses the HSQLDB driver together with an in-memory database. The initial amount of connections is 2 and the maximum amount is set at 5. Note that the database driver, connection url, username, and password parameters will vary depending on the database you are using.

```
JDBCConnectionPool pool = new SimpleJDBCConnectionPool(  
    "org.hsqldb.jdbc.JDBCDriver",  
    "jdbc:hsqldb:mem:sqlcontainer", "SA", "", 2, 5);
```

11.2.2. Creating the TableQuery Query Delegate

After the connection pool is created, we'll need a query delegate for the SQLContainer. The simplest way to create one is by using the built-in **TableQuery** class. The **TableQuery** delegate provides access to a defined database table and supports reading and writing data out-of-the-box. The primary key(s) of the table may be anything that the database engine supports, and are found automatically by querying the database when a new **TableQuery** is instantiated. We create the **TableQuery** with the following statement:

```
TableQuery tq = new TableQuery("tablename", connectionPool);
```

In order to allow writes from several user sessions concurrently, we must set a version column to the **TableQuery** as well. The version column is an integer- or timestamp-typed column which will either be incremented or set to the current time on each modification of the row. **TableQuery** assumes that the database will take care of updating the version column; it just makes sure the column value is correct before updating a row. If another user has changed the row and the version number in the database does not match the version number in memory, an **OptimisticLockException** is thrown and you can recover by refreshing the container and allow the user to merge the data. The following code will set the version column:

```
tq.setVersionColumn("OPTLOCK");
```

11.2.3. Creating the Container

Finally, we may create the container itself. This is as simple as stating:

```
SQLContainer container = new SQLContainer(tq);
```

After this statement, the **SQLContainer** is connected to the table **tablename** and is ready to use for example as a data source for a Vaadin **Table** or a Vaadin **Form**.

11.3. Filtering and Sorting

Filtering and sorting the items contained in an SQLContainer is, by design, always performed in the database. In practice this means that whenever the filtering or sorting rules are modified, at least some amount of database communication will take place (the minimum is to fetch the updated row count using the new filtering/sorting rules).

11.3.1. Filtering

Filtering is performed using the filtering API in Vaadin, which allows for very complex filtering to be easily applied. More information about the filtering API can be found in Section 10.5.7, “**Filterable** Containers”.

In addition to the filters provided by Vaadin, SQLContainer also implements the **Like** filter as well as the **Between** filter. Both of these map to the equally named WHERE-operators in SQL. The filters can also be applied on items that reside in memory, for example, new items that have not yet been stored in the database or rows that have been loaded and updated, but not yet stored.

The following is an example of the types of complex filtering that are possible with the new filtering API. We want to find all people named Paul Johnson that are either younger than 18 years or older than 65 years and all Johnsons whose first name starts with the letter "A":

```
mySQLContainer.addContainerFilter(
    new Or(new And(new Equal("NAME", "Paul"),
        new Or(new Less("AGE", 18),
            new Greater("AGE", 65))),
        new Like("NAME", "A%")));
mySQLContainer.addContainerFilter(
    new Equal("LASTNAME", "Johnson"));
```

This will produce the following WHERE clause:

```
WHERE (("NAME" = "Paul" AND ("AGE" < 18 OR "AGE" > 65)) OR "NAME" LIKE "A%")
AND "LASTNAME" = "Johnson"
```

11.3.2. Sorting

Sorting can be performed using standard Vaadin, that is, using the sort method from the **Container.Sortable** interface. The *propertyId* parameter refers to column names.

```
public void sort(Object[] propertyId, boolean[] ascending)
```

In addition to the standard method, it is also possible to directly add an **OrderBy** to the container via the `addOrderBy()` method. This enables the developer to insert sorters one by one without providing the whole array of them at once.

All sorting rules can be cleared by calling the sort method with null or an empty array as the first argument.

11.4. Editing

Editing the items (**RowItem**s) of **SQLContainer** can be done similarly to editing the items of any Vaadin container. `[classname]#ColumnProperties` of a **RowItem** will automatically notify SQLContainer to make sure that changes to the items are recorded and will be applied to the database immediately or on commit, depending on the state of the auto-commit mode.

11.4.1. Adding items

Adding items to an **SQLContainer** object can only be done via the `addItem()` method. This method will create a new **Item** based on the connected database table column properties. The

new item will either be buffered by the container or committed to the database through the query delegate depending on whether the auto commit mode (see the next section) has been enabled.

When an item is added to the container it is impossible to precisely know what the primary keys of the row will be, or will the row insertion succeed at all. This is why the `SQLContainer` will assign an instance of **TemporaryRowId** as a **RowId** for the new item. We will later describe how to fetch the actual key after the row insertion has succeeded.

If auto-commit mode is enabled in the **SQLContainer**, the `addItem()` method will return the final **RowId** of the new item.

11.4.2. Fetching generated row keys

Since it is a common need to fetch the generated key of a row right after insertion, a listener/notifier has been added into the **QueryDelegate** interface. Currently only the **TableQuery** class implements the **RowIdChangeNotifier** interface, and thus can notify interested objects of changed row IDs. The events will be fired after `commit()` in **TableQuery** has finished; this method is called by **SQLContainer** when necessary.

To receive updates on the row IDs, you might use the following code (assuming container is an instance of **SQLContainer**). Note that these events are not fired if auto commit mode is enabled.

```
app.getDbHelp().getCityContainer().addListener(
    new QueryDelegate.RowIdChangeListener() {
        public void rowIdChange(RowIdChangeEvent event) {
            System.err.println("Old ID: " + event.getOldRowId());
            System.err.println("New ID: " + event.getNewRowId());
        }
    });
```

11.4.3. Version column requirement

If you are using the **TableQuery** class as the query delegate to the **SQLContainer** and need to enable write support, there is an enforced requirement of specifying a version column name to the **TableQuery** instance. The column name can be set to the **TableQuery** using the following statement:

```
tq.setVersionColumn("OPTLOCK");
```

The version column is preferably an integer or timestamp typed column in the table that is attached to the **TableQuery**. This column will be used for optimistic locking; before a row modification the **TableQuery** will check before that the version column value is the same as it was when the data was read into the container. This should ensure that no one has modified the row inbetween the current user's reads and writes.

Note! **TableQuery** assumes that the database will take care of updating the version column by either using an actual `VERSION` column (if supported by the database in question) or by a trigger or a similar mechanism.

If you are certain that you do not need optimistic locking, but do want to enable write support, you may point the version column to, for example, a primary key column of the table.

11.4.4. Auto-commit mode

SQLContainer is by default in transaction mode, which means that actions that edit, add or remove items are recorded internally by the container. These actions can be either committed to the database by calling `commit()` or discarded by calling `rollback()`.

The container can also be set to auto-commit mode. When this mode is enabled, all changes will be committed to the database immediately. To enable or disable the auto-commit mode, call the following method:

```
public void setAutoCommit(boolean autoCommitEnabled)
```

It is recommended to leave the auto-commit mode disabled, as it ensures that the changes can be rolled back if any problems are noticed within the container items. Using the auto-commit mode will also lead to failure in item addition if the database table contains non-nullable columns.

11.4.5. Modified state

When used in the transaction mode it may be useful to determine whether the contents of the **SQLContainer** have been modified or not. For this purpose the container provides an `isModified()` method, which will tell the state of the container to the developer. This method will return true if any items have been added to or removed from the container, as well as if any value of an existing item has been modified.

Additionally, each **RowItem** and each **ColumnProperty** have `isModified()` methods to allow for a more detailed view over the modification status. Do note that the modification statuses of **RowItem** and **ColumnProperty** objects only depend on whether or not the actual **Property** values have been modified. That is, they do not reflect situations where the whole **RowItem** has been marked for removal or has just been added to the container.

11.5. Caching, Paging and Refreshing

To decrease the amount of queries made to the database, **SQLContainer** uses internal caching for database contents. The caching is implemented with a size-limited **LinkedHashMap** containing a mapping from `[classname]#RowId#s` to `[classname]#RowItem#s`. Typically developers do not need to modify caching options, although some fine-tuning can be done if required.

11.5.1. Container Size

The **SQLContainer** keeps continuously checking the amount of rows in the connected database table in order to detect external addition or removal of rows. By default, the table row count is assumed to remain valid for 10 seconds. This value can be altered from code; with `setSizeValidityMilliseconds()` in **SQLContainer**.

If the size validity time has expired, the row count will be automatically updated on:

- A call to `getItemIds()` method
- A call to `size()` method
- Some calls to `indexOfId(Object itemId)` method
- A call to `firstItemId()` method
- When the container is fetching a set of rows to the item cache (lazy loading)

11.5.2. Page Length and Cache Size

The page length of the **SQLContainer** dictates the amount of rows fetched from the database in one query. The default value is 100, and it can be modified with the `setPageLength()` method. To avoid constant queries it is recommended to set the page length value to at least 5 times the amount of rows displayed in a Vaadin **Table**; obviously, this is also dependent on the cache ratio set for the **Table** component.

The size of the internal item cache of the **SQLContainer** is calculated by multiplying the page length with the cache ratio set for the container. The cache ratio can only be set from the code, and the default value for it is 2. Hence with the default page length of 100 the internal cache size becomes 200 items. This should be enough even for larger [classname]#Table#s while ensuring that no huge amounts of memory will be used on the cache.

11.5.3. Refreshing the Container

Normally, the **SQLContainer** will handle refreshing automatically when required. However, there may be situations where an implicit refresh is needed, for example, to make sure that the version column is up-to-date prior to opening the item for editing in a form. For this purpose a `refresh()` method is provided. This method simply clears all caches, resets the current item fetching offset and sets the container size dirty. Any item-related call after this will inevitably result into row count and item cache update.

Note that a call to the refresh method will not affect or reset the following properties of the container:

- The **QueryDelegate** of the container
- Auto-commit mode
- Page length
- Filters
- Sorting

11.6. Referencing Another SQLContainer

When developing a database-connected application, there is usually a need to retrieve data related to one table from one or more other tables. In most cases, this relation is achieved with a foreign key reference, where a column of the first table contains a primary key or candidate key of a row in another table.

SQLContainer offers limited support for this kind of referencing relation, although all referencing is currently done on the Java side so no constraints need to be made in the database. A new reference can be created by calling the following method:

```
public void addReference(SQLContainer refdCont,  
                        String refingCol, String refdCol);
```

This method should be called on the source container of the reference. The target container should be given as the first parameter. The `refingCol` is the name of the 'foreign key' column in the source container, and the `refdCol` is the name of the referenced key column in the target container.

*Note: For any **SQLContainer**, all the referenced target containers must be different. You can not reference the same container from the same source twice.*

Handling the referenced item can be done through the three provided set/get methods, and the reference can be completely removed with the `removeReference()` method. Signatures of these methods are listed below:

```
public boolean setReferencedItem(Object itemId,
                                Object refdItemId, SQLContainer refdCont)
public Object getReferencedItemId(Object itemId,
                                SQLContainer refdCont)
public Item getReferencedItem(Object itemId,
                              SQLContainer refdCont)
public boolean removeReference(SQLContainer refdCont)
```

The setter method should be given three parameters: `itemId` is the ID of the referencing item (from the source container), `refdItemId` is the referenced `itemId` (from the target container) and `refdCont` is a reference to the target container that identifies the reference. This method returns true if the setting of the referenced item was successful. After setting the referenced item you must normally call `commit()` on the source container to persist the changes to the database.

The `getReferencedItemId()` method will return the item ID of the referenced item. As parameters this method needs the item ID of the referencing item and a reference to the target container as an identifier. **SQLContainer** also provides a convenience method `getReferencedItem()`, which directly returns the referenced item from the target container.

Finally, the referencing can be removed from the source container by calling the `removeReference()` method with the target container as parameter. Note that this does not actually change anything in the database; it merely removes the logical relation that exists only on the Java-side.

11.7. Making Freeform Queries

In most cases, the provided **TableQuery** will be enough to allow a developer to gain effortless access to an SQL data source. However there may arise situations when a more complex query with, for example, join expressions is needed. Or perhaps you need to redefine how the writing or filtering should be done. The **FreeformQuery** query delegate is provided for this exact purpose. Out of the box the **FreeformQuery** supports read-only access to a database, but it can be extended to allow writing also.

11.7.1. Getting started

Getting started with the **FreeformQuery** may be done as shown in the following. The connection pool initialization is similar to the **TableQuery** example so it is omitted here. Note that the name(s) of the primary key column(s) must be provided to the **FreeformQuery** manually. This is required because depending on the query the result set may or may not contain data about primary key columns. In this example, there is one primary key column with a name 'ID'.

```
FreeformQuery query = new FreeformQuery(
    "SELECT * FROM SAMPLE", pool, "ID");
SQLContainer container = new SQLContainer(query);
```

11.7.2. Limitations

While this looks just as easy as with the **TableQuery**, do note that there are some important caveats here. Using **FreeformQuery** like this (without providing **FreeformQueryDelegate** or **FreeformStatementDelegate** implementation) it can only be used as a read-only window to the resultset of the query. Additionally filtering, sorting and lazy loading features will not be supported, and the row count will be fetched in quite an inefficient manner. Bearing these limitations in mind, it becomes quite obvious that the developer is in reality meant to implement the **FreeformQueryDelegate** or **FreeformStatementDelegate** interface.

The **FreeformStatementDelegate** interface is an extension of the **FreeformQueryDelegate** interface, which returns **StatementHelper** objects instead of pure query **String**s. **This enables the developer to use prepared statements instead of regular statements. It is highly recommended to use the [classname]#FreeformStatementDelegate in all implementations.** From this chapter onwards, we will only refer to the **FreeformStatementDelegate** in cases where **FreeformQueryDelegate** could also be applied.

11.7.3. Creating your own FreeformStatementDelegate

To create your own delegate for **FreeformQuery** you must implement some or all of the methods from the **FreeformStatementDelegate** interface, depending on which ones your use case requires. The interface contains eight methods which are shown below. For more detailed requirements, see the JavaDoc documentation of the interface.

```
// Read-only queries
public StatementHelper getCountStatement()
public StatementHelper getQueryStatement(int offset, int limit)
public StatementHelper getContainsRowQueryStatement(Object... keys)

// Filtering and sorting
public void setFilters(List<Filter> filters)
public void setFilters(List<Filter> filters,
                      FilteringMode filteringMode)
public void setOrderBy(List<OrderBy> orderBys)

// Write support
public int storeRow(Connection conn, RowItem row)
public boolean removeRow(Connection conn, RowItem row)
```

A simple demo implementation of this interface can be found in the SQLContainer package, more specifically in the class **com.vaadin.addon.sqlcontainer.demo.DemoFreeformQueryDelegate**.

11.8. Non-Implemented Methods

Due to the database connection inherent to the SQLContainer, some of the methods from the container interfaces of Vaadin can not (or would not make sense to) be implemented. These methods are listed below, and they will throw an **UnsupportedOperationException** on invocation.

```
public boolean addContainerProperty(Object propertyId,
                                   Class<?> type,
                                   Object defaultValue)
public boolean removeContainerProperty(Object propertyId)
public Item addItem(Object itemId)
public Object addItemAt(int index)
```

```
public Item addItemAt(int index, Object newItemId)
public Object addItemAfter(Object previousItemId)
public Item addItemAfter(Object previousItemId, Object newItemId)
```

Additionally, the following methods of the **Item** interface are not supported in the **RowItem** class:

```
public boolean addItemProperty(Object id, Property property)
public boolean removeItemProperty(Object id)
```

11.8.1. About the `getItemIds()` method

To properly implement the Vaadin **Container** interface, a `getItemIds()` method has been implemented in the **SQLContainer**. By definition, this method returns a collection of all the item IDs present in the container. What this means in the **SQLContainer** case is that the container has to query the database for the primary key columns of all the rows present in the connected database table.

It is obvious that this could potentially lead to fetching tens or even hundreds of thousands of rows in an effort to satisfy the method caller. This will effectively kill the lazy loading properties of **SQLContainer** and therefore the following warning is expressed here:



Warning

It is highly recommended not to call the `getItemIds()` method, unless it is known that in the use case in question the item ID set will always be of reasonable size.

11.9. Known Issues and Limitations

At this point, there are still some known issues and limitations affecting the use of SQLContainer in certain situations. The known issues and brief explanations are listed below:

- Some SQL data types do not have write support when using TableQuery: ** All binary types
- All custom types
- CLOB (if not converted automatically to a **String** by the JDBC driver in use)
- See **com.vaadin.addon.sqlcontainer.query.generator.StatementHelper** for details.
- When using Oracle or MS SQL database, the column name "`rownum`" can not be used as a column name in a table connected to **SQLContainer**.

This limitation exists because the databases in question do not support limit/offset clauses required for paging. Instead, a generated column named 'rownum' is used to implement paging support.

The permanent limitations are listed below. These can not or most probably will not be fixed in future versions of SQLContainer.

- The `getItemIds()` method is very inefficient - avoid calling it unless absolutely required!

- When using **FreeformQuery** without providing a **FreeformStatementDelegate**, the row count query is very inefficient - avoid using **FreeformQuery** without implementing at least the count query properly.
- When using **FreeformQuery** without providing a **FreeformStatementDelegate**, writing, sorting and filtering will not be supported.
- When using Oracle database most or all of the numeric types are converted to **java.math.BigDecimal** by the Oracle JDBC Driver.

This is a feature of how Oracle DB and the Oracle JDBC Driver handles data types.

Chapter 12

Advanced Web Application Topics

12.1. Handling Browser Windows	358
12.2. Embedding UIs in Web Pages	361
12.3. Debug Mode and Window	363
12.4. Request Handlers	369
12.5. Shortcut Keys	370
12.6. Printing	375
12.7. Google App Engine Integration	377
12.8. Common Security Issues	378
12.9. Navigating in an Application	378
12.10. Advanced Application Architectures	383
12.11. Managing URI Fragments	388
12.12. Drag and Drop	390
12.13. Logging	399
12.14. JavaScript Interaction	400
12.15. Accessing Session-Global Data	402
12.16. Server Push	405
12.17. Vaadin CDI Add-on	411
12.18. Vaadin Spring Add-on	417

This chapter covers various features and topics often needed in applications.

12.1. Handling Browser Windows

The UI of a Vaadin application runs in a web page displayed in a browser window or tab. An application can be used from multiple UIs in different windows or tabs, either opened by the user using an URL or by the Vaadin application.

In addition to native browser windows, Vaadin has a **Window** component, which is a floating panel or *sub-window* inside a page, as described in Section 7.7, “Sub-Windows”.

- *Native popup windows.* An application can open popup windows for sub-tasks.
- *Page-based browsing.* The application can allow the user to open certain content to different windows. For example, in a messaging application, it can be useful to open different messages to different windows so that the user can browse through them while writing a new message.
- *Bookmarking.* Bookmarks in the web browser can provide an entry-point to some content provided by an application.
- *Embedding UIs.* UIs can be embedded in web pages, thus making it possible to provide different views to an application from different pages or even from the same page, while keeping the same session. See Section 12.2, “Embedding UIs in Web Pages”.

Use of multiple windows in an application may require defining and providing different UIs for the different windows. The UIs of an application share the same user session, that is, the **VaadinSession** object, as described in Section 5.8.3, “User Session”. Each UI is identified by a URL that is used to access it, which makes it possible to bookmark application UIs. UI instances can even be created dynamically based on the URLs or other request parameters, such as browser information, as described in Section 5.8.4, “Loading a UI”.

Because of the special nature of AJAX applications, use of multiple windows uses require some caveats.

12.1.1. Opening Popup Windows

Popup windows are native browser windows or tabs opened by user interaction with an existing window. Due to browser security reasons, it is made inconvenient for a web page to open popup windows using JavaScript commands. At the least, the browser will ask for a permission to open the popup, if it is possible at all. This limitation can be circumvented by letting the browser open the new window or tab directly by its URL when the user clicks some target. This is realized in Vaadin with the **BrowserWindowOpener** component extension, which causes the browser to open a window or tab when the component is clicked.

The Popup Window UI

A popup Window displays an **UI**. The UI of a popup window is defined just like a main UI in a Vaadin application, and it can have a theme, title, and so forth.

For example:

```
@Theme("book-examples")
public static class MyPopupUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        getPage().setTitle("Popup Window");
    }
}
```

```
// Have some content for it
VerticalLayout content = new VerticalLayout();
Label label =
    new Label("I just popped up to say hi!");
label.setSizeUndefined();
content.addComponent(label);
content.setComponentAlignment(label,
    Alignment.MIDDLE_CENTER);
content.setSizeFull();
setContent(content);
}
}
```

Popping It Up

A popup window is opened using the **BrowserWindowOpener** extension, which you can attach to any component. The constructor of the extension takes the class object of the UI class to be opened as a parameter.

You can configure the features of the popup window with `setFeatures()`. It takes as its parameter a comma-separated list of window features, as defined in the HTML specification.

`status=0|1`

Whether the status bar at the bottom of the window should be enabled.

`[parameter]##, scrollbars`

Enables scrollbars in the window if the document area is bigger than the view area of the window.

`resizable`

Allows the user to resize the browser window (no effect for tabs).

`menubar`

Enables the browser menu bar.

`location`

Enables the location bar.

`toolbar`

Enables the browser toolbar.

`height=value`

Specifies the height of the window in pixels.

`width=value`

Specifies the width of the window in pixels.

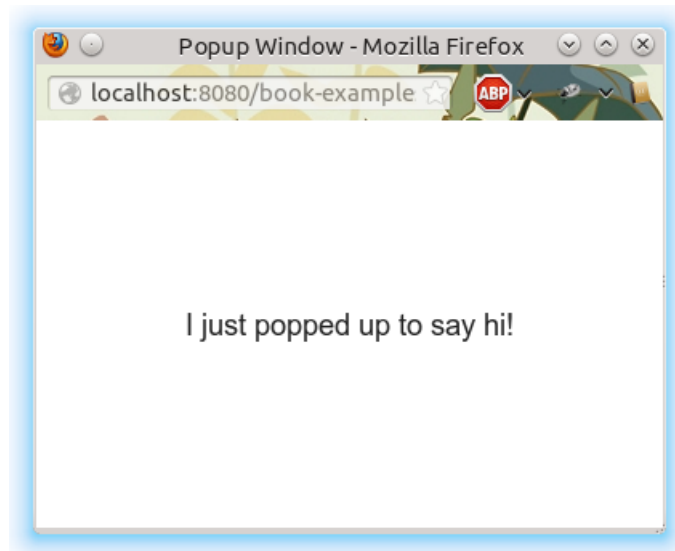
For example:

```
// Create an opener extension
BrowserWindowOpener opener =
    new BrowserWindowOpener(MyPopupUI.class);
opener.setFeatures("height=200,width=300,resizable");

// Attach it to a button
Button button = new Button("Pop It Up");
opener.extend(button);
```

The resulting popup window, which appears when the button is clicked, is shown in Figure 12.1, “A Popup Window”.

Figure 12.1. A Popup Window



Popup Window Name (Target)

The target name is one of the default HTML target names (*_new*, *_blank*, *_top*, etc.) or a custom target name. How the window is exactly opened depends on the browser. Browsers that support tabbed browsing can open the window in another tab, depending on the browser settings.

URL and Session

The URL path for a popup window UI is by default determined from the UI class name, by prefixing it with "popup/". For example, for the example UI given earlier, the URL would be `/book-examples/book/popup/MyPopupUI`.

12.1.2. Closing Popup Windows

Besides closing popup windows from a native window close button, you can close them programmatically by calling the JavaScript `close()` method as follows.

```
public class MyPopup extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new Button("Close Window", event -> { // Java 8
            // Close the popup
            JavaScript.eval("close()");

            // Detach the UI from the session
            getUI().close();
        }));
    }
}
```

12.2. Embedding UIs in Web Pages

Many web sites are not all Vaadin, but Vaadin UIs are used only for specific functionalities. In practice, many web applications are a mixture of dynamic web pages, such as JSP, and Vaadin UIs embedded in such pages.

Embedding Vaadin UIs in web pages is easy and there are several different ways to embed them. One is to have a `<div>` placeholder for the UI and load the Vaadin Client-Side Engine with some simple JavaScript code. Another method is even easier, which is to simply use the `<iframe>` element. Both of these methods have advantages and disadvantages. One disadvantage of the `<iframe>` method is that the size of the `<iframe>` element is not flexible according to the content while the `<div>` method allows such flexibility. The following sections look closer into these two embedding methods.

12.2.1. Embedding Inside a `div` Element

You can embed one or more Vaadin UIs inside a web page with a method that is equivalent to loading the initial page content from the Vaadin servlet in a non-embedded UI. Normally, the **VaadinServlet** generates an initial page that contains the correct parameters for the specific UI. You can easily configure it to load multiple Vaadin UIs in the same page. They can have different widget sets and different themes.

Embedding an UI requires the following basic tasks:

- Set up the page header
- Include a GWT history frame in the page
- Call the `vaadinBootstrap.js` file
- Define the `<div>` element for the UI
- Configure and initialize the UI

Notice that you can view the loader page for the UI easily by opening the UI in a web browser and viewing the HTML source code of the page. You could just copy and paste the embedding code from the page, but some modifications and additional settings are required, mainly related to the URLs that have to be made relative to the page instead of the servlet URL.

12.2.2. Embedding Inside an `iframe` Element

Embedding a Vaadin UI inside an `<iframe>` element is even easier than the method described above, as it does not require definition of any Vaadin specific definitions.

You can embed an UI with an element such as the following:

```
<iframe src="/myapp/myui"></iframe>
```

The `<iframe>` elements have several downsides for embedding. One is that their size of is not flexible depending on the content of the frame, but the content must be flexible to accommodate in the frame. You can set the size of an `<iframe>` element with `height` and `width` attributes. Other issues arise from themeing and communication with the frame content and the rest of the page.

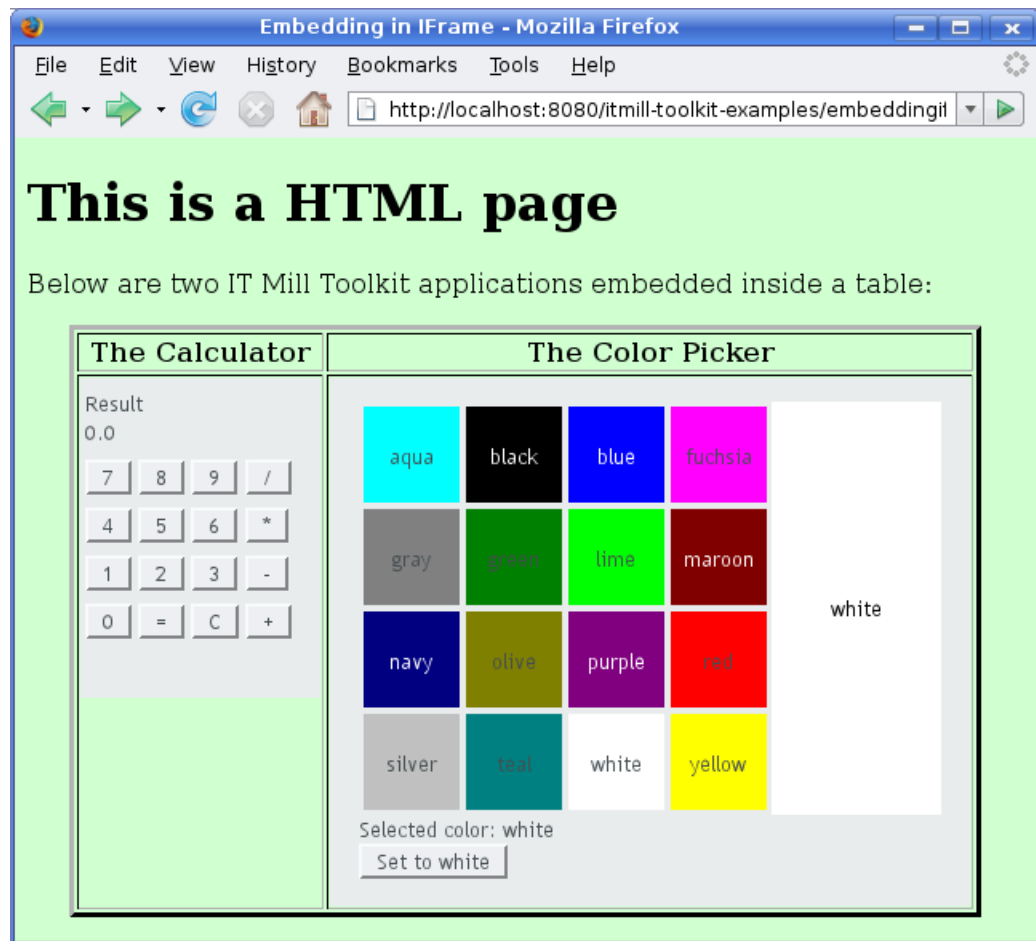
Below is a complete example of using the `<iframe>` to embed two applications in a web page.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Embedding in IFrame</title>
  </head>

  <body style="background: #d0ffd0;">
    <h1>This is a HTML page</h1>
    <p>Below are two Vaadin applications embedded inside
      a table:</p>

    <table align="center" border="3">
      <tr>
        <th>The Calculator</th>
        <th>The Color Picker</th>
      </tr>
      <tr valign="top">
        <td>
          <iframe src="/vaadin-examples/Calc" height="200"
            width="150" frameborder="0"></iframe>
        </td>
        <td>
          <iframe src="/vaadin-examples/colorpicker"
            height="330" width="400"
            frameborder="0"></iframe>
        </td>
      </tr>
    </table>
  </body>
</html>
```

The page will look as shown in Figure 12.2, “Vaadin Applications Embedded Inside IFrames” below.

Figure 12.2. Vaadin Applications Embedded Inside IFrames

You can embed almost anything in an iframe, which essentially acts as a browser window. However, this creates various problems. The iframe must have a fixed size, inheritance of CSS from the embedding page is not possible, and neither is interaction with JavaScript, which makes mashups impossible, and so on. Even bookmarking with URI fragments will not work.

Note also that websites can forbid iframe embedding by specifying an `X-Frame-Options: SAMEORIGIN` header in the HTTP response.

12.3. Debug Mode and Window

Vaadin applications can be run in two modes: *debug mode* and *production mode*. The debug mode, which is on by default, enables a number of built-in debug features for Vaadin developers:

- Debug Window
- Display debug information in the Debug Window and server console
- On-the-fly compilation of Sass themes
- Timings of server calls for Vaadin TestBench

It is recommended to always deploy production applications in production mode for security reasons.

12.3.1. Enabling the Debug Mode

The debug mode is enabled and production mode disabled by default in the UI templates created with the Eclipse plugin or the Maven archetypes. Some archetypes have a separate module and profile for producing a production mode application. The debug mode can be enabled by giving a *productionMode=false* parameter to the Vaadin servlet configuration:

```
@VaadinServletConfiguration(  
    productionMode = false,  
    ui = MyprojectUI.class)
```

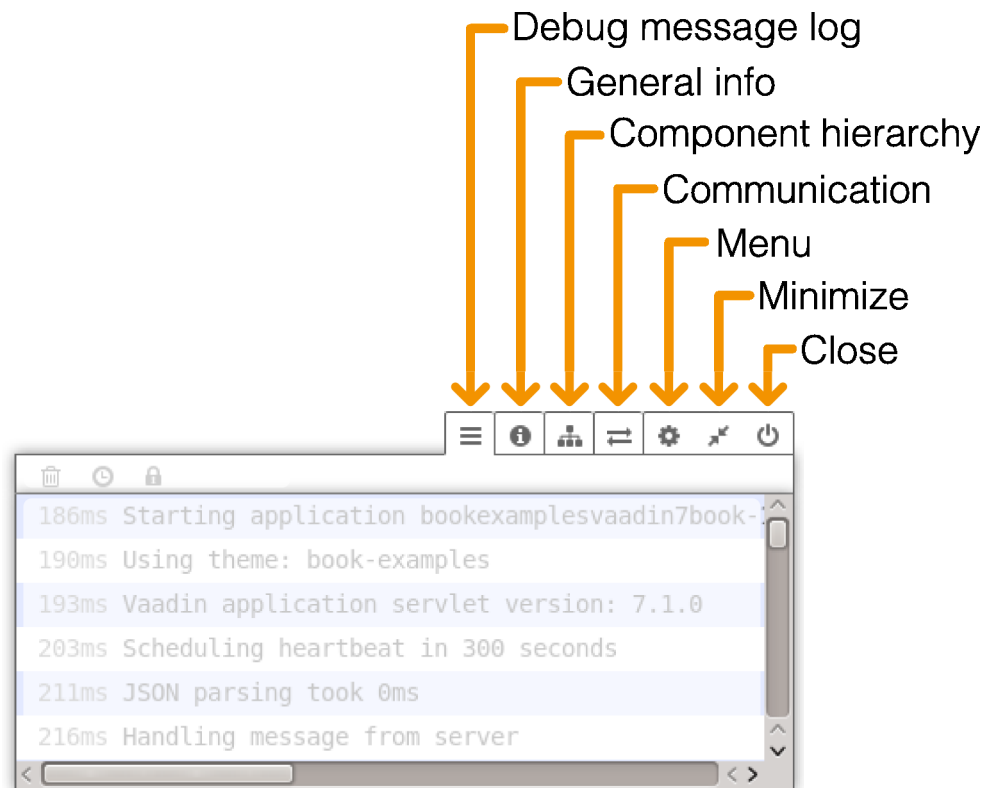
Or with a context parameter in the `web.xml` deployment descriptor:

```
<context-param>  
  <description>Vaadin production mode</description>  
  <param-name>productionMode</param-name>  
  <param-value>false</param-value>  
</context-param>
```

Enabling the production mode disables the debug features, thereby preventing users from easily inspecting the inner workings of the application from the browser.

12.3.2. Opening the Debug Window

Running an application in the debug mode enables the client-side Debug Window in the browser. You can open the Debug Window by adding " ?debug" parameter to the URL of the UI, for example, `http://localhost:8080/myapp/?debug`. The Debug Window has buttons for controlling the debugging features and a scrollable log of debug messages.

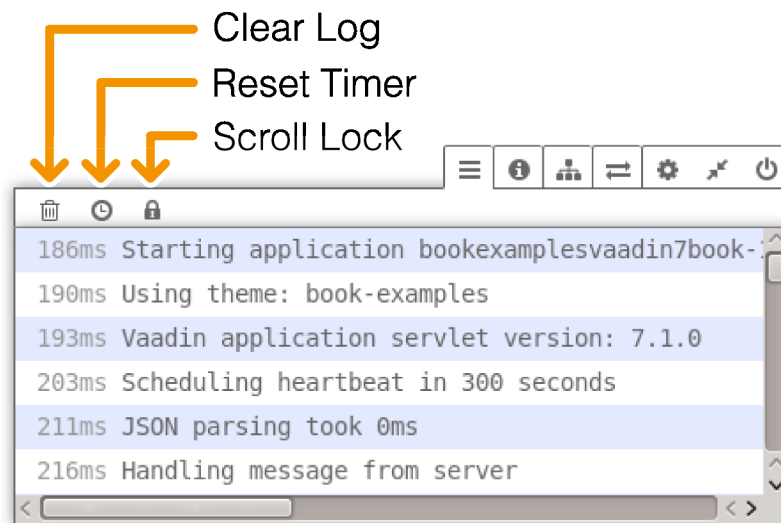
Figure 12.3. Debug Window

The functionalities are described in detail in the subsequent sections. You can move the window by dragging it from the title bar and resize it from the corners. The **Minimize** button minimizes the debug window in the corner of the browser window, and the **Close** button closes it.

If you use the Firebug plugin for Firefox or the Developer Tools console in Chrome, the log messages will also be printed to the Firebug console. In such a case, you may want to enable client-side debugging without showing the Debug Window with " ?debug=quiet" in the URL. In the quiet debug mode, log messages will only be printed to the console of the browser debugger.

12.3.3. Debug Message Log

The debug message log displays client-side debug messages, with time counter in milliseconds. The control buttons allow you to clear the log, reset the timer, and lock scrolling.

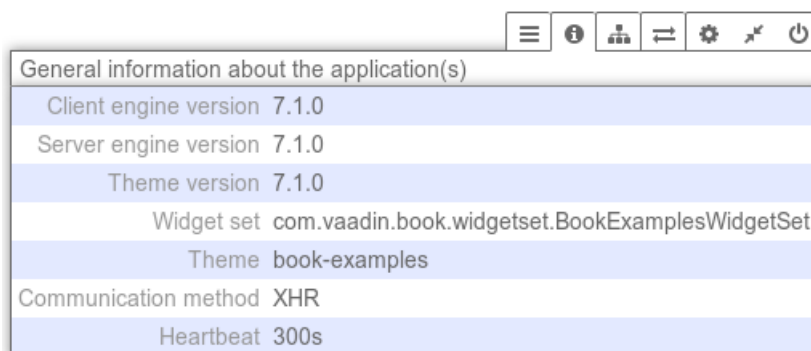
Figure 12.4. Debug Message Log

Logging to Debug Window

You can take advantage of the debug mode when developing client-side components, by using the standard Java **Logger** to write messages to the log. The messages will be written to the debug window and Firebug console. No messages are written if the debug window is not open or if the application is running in production mode.

12.3.4. General Information

The **General information about the application(s)** tab displays various information about the UI, such as version numbers of the client and servlet engine, and the theme. If they do not match, you may need to compile the widget set or theme.

Figure 12.5. General Information

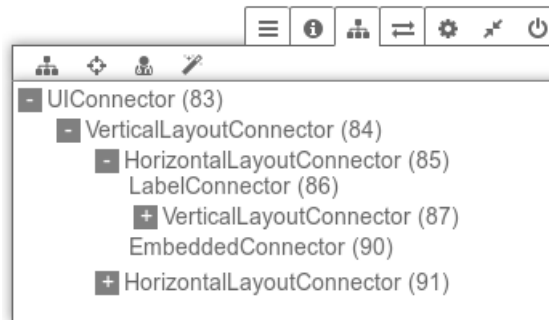
12.3.5. Inspecting Component Hierarchy

The **Component Hierarchy** tab has several sub-modes that allow debugging the component tree in various ways.

Connector Hierarchy Tree

The **Show the connector hierarchy tree** button displays the client-side connector hierarchy. As explained in Chapter 17, *Integrating with the Server-Side*, client-side widgets are managed by connectors that handle communication with the server-side component counterparts. The connector hierarchy therefore corresponds with the server-side component tree, but the client-side widget tree and HTML DOM tree have more complexity.

Figure 12.6. Connector Hierarchy Tree



Clicking on a connector highlights the widget in the UI.

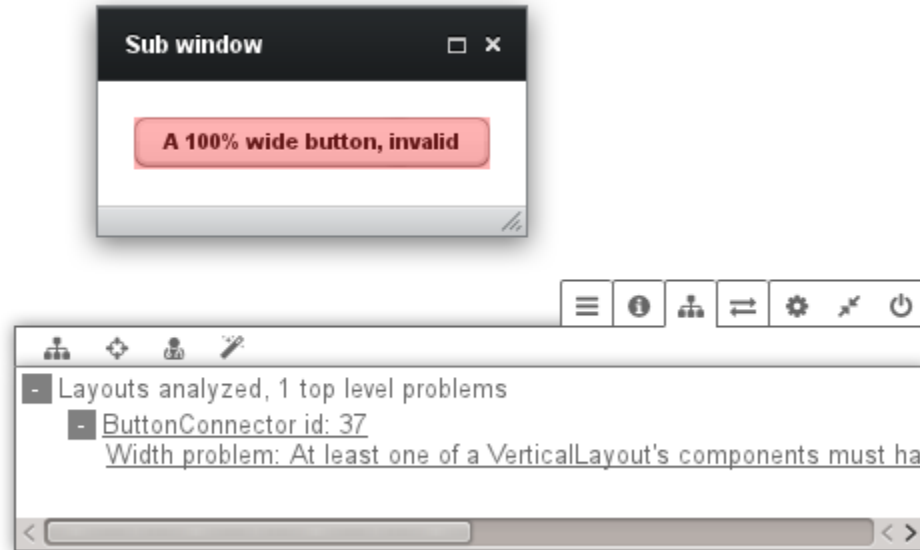
Inspecting a Component

The **Select a component in the page to inspect it** button lets you select a component in the UI by clicking it and display its client-side properties.

To view the HTML structure and CSS styles in more detail, you can use Firebug in Firefox, or the Developer Tools in Chrome, as described in Section 2.8.1, "Firefox and Firebug". Firefox also has a built-in feature for inspecting HTML and CSS.

Analyzing Layout Problems

The **Check layouts for potential problems** button analyzes the currently visible UI and makes a report of possible layout related problems. All detected layout problems are displayed in the log and also printed to the console.

Figure 12.7. Debug window showing the result of layout analysis.

Clicking on a reported problem highlights the component with the problem in the UI.

The most common layout problem is caused by placing a component that has a relative size inside a container (layout) that has undefined size in the particular direction (height or width). For example, adding a **Button** with 100% width inside a **VerticalLayout** with undefined width. In such a case, the error would look as shown in Figure 12.7, “Debug window showing the result of layout analysis.”.

CustomLayout components can not be analyzed in the same way as other layouts. For custom layouts, the button analyzes all contained relative-sized components and checks if any relative dimension is calculated to zero so that the component will be invisible. The error log will display a warning for each of these invisible components. It would not be meaningful to emphasize the component itself as it is not visible, so when you select such an error, the parent layout of the component is emphasized if possible.

Displaying Used Connectors

The last button, **Show used connectors and how to optimize widget set**, displays a list of all currently visible connectors. It also generates a connector bundle loader factory, which you can use to optimize the widget set so that it only contains the widgets actually used in the UI. Note, however, that it only lists the connectors visible in the current UI state, and you usually have more connectors than that.

12.3.6. Communication Log

The **Communication** tab displays all server requests. You can unfold the requests to view details, such as the connectors involved. Clicking on a connector highlights the corresponding element in the UI.

You can use Firebug or Developer Tools in Firefox or Chrome, respectively, to get more detailed information about the requests and responses.

12.3.7. Debug Modes

The **Menu** tab in the window opens a sub-menu to select various settings. Here you can also launch the GWT SuperDevMode, as described in Section 14.6, “Debugging Client-Side Code”.

12.4. Request Handlers

Request handlers are useful for catching request parameters or generating dynamic content, such as HTML, images, PDF, or other content. You can provide HTTP content also with stream resources, as described in Section 5.5.5, “Stream Resources”. The stream resources, however, are only usable from within a Vaadin application, such as in an **Image** component. Request handlers allow responding to HTTP requests made with the application URL, including GET or POST parameters. You could also use a separate servlet to generate dynamic content, but a request handler is associated with the user session and it can easily access data associated with the session and the user.

To handle requests, you need to implement the `RequestHandler` interface. The `handleRequest()` method gets the session, request, and response objects as parameters.

If the handler writes a response, it must return `true`. This stops running other possible request handlers. Otherwise, it should return `false` so that another handler could return a response. Eventually, if no other handler writes a response, a UI will be created and initialized.

In the following example, we catch requests for a sub-path in the URL for the servlet and write a plain text response. The servlet path consists of the context path and the servlet (sub-)path. Any additional path is passed to the request handler in the `pathInfo` of the request. For example, if the full path is `/myapp/myui/rhexample`, the path info will be `/rhexample`. Also, request parameters are available.

```
// A request handler for generating some content
VaadinSession.getCurrent().addRequestHandler(
    new RequestHandler() {
        @Override
        public boolean handleRequest(VaadinSession session,
                                    VaadinRequest request,
                                    VaadinResponse response)
            throws IOException {
            if ("/rhexample".equals(request.getPathInfo())) {
                // Generate a plain text document
                response.setContentType("text/plain");
                response.getWriter().append(
                    "Here's some dynamically generated content.\n");
                response.getWriter().format(Locale.ENGLISH,
                    "Time: %Tc\n", new Date());

                // Use shared session data
                response.getWriter().format("Session data: %s\n",
                    session.getAttribute("mydata"));

                return true; // We wrote a response
            } else
                return false; // No response was written
        }
    });
```

A request handler can be used by embedding it in a page or opening a new page with a link or a button. In the following example, we pass some data to the handler through a session attribute.

```
// Input some shared data in the session
TextField dataInput = new TextField("Some data");
dataInput.addValueChangeListener(event ->
    VaadinSession.getCurrent().setAttribute("mydata",
        event.getProperty().getValue()));
dataInput.setValue("Here's some");

// Determine the base path for the servlet
String servletPath = VaadinServlet.getCurrent()
    .getServletContext().getContextPath()
    + "/book"; // Servlet

// Display the page in a pop-up window
Link open = new Link("Click to Show the Page",
    new ExternalResource(servletPath + "/rhexample"),
    "_blank", 500, 350, BorderStyle.DEFAULT);

layout.addComponents(dataInput, open);
```

12.5. Shortcut Keys

Vaadin provides simple ways to define shortcut keys for field components, as well as to a default button, and a lower-level generic shortcut API based on actions.

A *shortcut* is an action that is executed when a key or key combination is pressed within a specific scope in the UI. The scope can be the entire **UI** or a **Window** inside it.

12.5.1. Shortcut Keys for Default Buttons

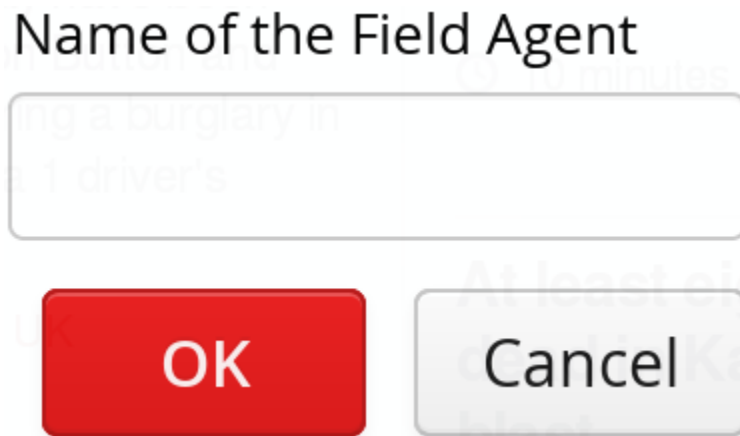
You can add a *click shortcut* to a button to set it as "default" button; pressing the defined key, typically **Enter**, in any component in the scope (sub-window or UI) causes a click event for the button to be fired.

You can define a click shortcut with the `setClickShortcut()` shorthand method:

```
// Have an OK button and set it as the default button
Button ok = new Button("OK");
ok.setClickShortcut(KeyCode.ENTER);
ok.addStyleName(ValoTheme.BUTTON_PRIMARY);
```

The `setClickShortcut()` is a shorthand method to create, add, and manage a **ClickShortcut**, rather than to add it with `addShortcutListener()`.

Themes offer special button styles to show that a button is special. In the Valo theme, you can use the `BUTTON_PRIMARY` style name. The result can be seen in Figure 12.8, "Default Button with Click Shortcut".

Figure 12.8. Default Button with Click Shortcut

12.5.2. Field Focus Shortcuts

You can define a shortcut key that sets the focus to a field component (any component that inherits **AbstractField**) by adding a **FocusShortcut** as a shortcut listener to the field.

The constructor of the **FocusShortcut** takes the field component as its first parameter, followed by the key code, and an optional list of modifier keys, as listed in Section 12.5.4, “Supported Key Codes and Modifier Keys”.

```
// A field with Alt+N bound to it
TextField name = new TextField("Name (Alt+N)");
name.addShortcutListener(
    new AbstractField.FocusShortcut(name, KeyCode.N,
                                    ModifierKey.ALT));
layout.addComponent(name);
```

You can also specify the shortcut by a shorthand notation, where the shortcut key is indicated with an ampersand (&).

```
// A field with Alt+A bound to it, using shorthand notation
TextField address = new TextField("Address (Alt+A)");
address.addShortcutListener(
    new AbstractField.FocusShortcut(address, "&Address"));
```

This is especially useful for internationalization, so that you can determine the shortcut key from the localized string.

12.5.3. Generic Shortcut Actions

Shortcut keys can be defined as *actions* using the **ShortcutAction** class. It extends the generic **Action** class that is used for example in **Tree** and **Table** for context menus. Currently, the only classes that accept **ShortcutActions** are **Window** and **Panel**.

To handle key presses, you need to define an action handler by implementing the **Handler** interface. The interface has two methods that you need to implement: `getActions()` and `handleAction()`.

The `getActions()` method must return an array of **Action** objects for the component, specified with the second parameter for the method, the *sender* of an action. For a keyboard shortcut, you use a **ShortcutAction**. The implementation of the method could be following:

```
// Have the unmodified Enter key cause an event
Action action_ok = new ShortcutAction("Default key",
    ShortcutAction.KeyCode.ENTER, null);

// Have the C key modified with Alt cause an event
Action action_cancel = new ShortcutAction("Alt+C",
    ShortcutAction.KeyCode.C,
    new int[] { ShortcutAction.ModifierKey.ALT });

Action[] actions = new Action[] {action_cancel, action_ok};

public Action[] getActions(Object target, Object sender) {
    if (sender == myPanel)
        return actions;

    return null;
}
```

The returned **Action** array may be static or you can create it dynamically for different senders according to your needs.

The constructor of **ShortcutAction** takes a symbolic caption for the action; this is largely irrelevant for shortcut actions in their current implementation, but might be used later if implementors use them both in menus and as shortcut actions. The second parameter is the key code and the third a list of modifier keys, which are listed in Section 12.5.4, “Supported Key Codes and Modifier Keys”.

The following example demonstrates the definition of a default button for a user interface, as well as a normal shortcut key, **Alt+C** for clicking the **Cancel** button.

```
public class DefaultButtonExample extends CustomComponent
    implements Handler {
    // Define and create user interface components
    Panel panel = new Panel("Login");
    FormLayout formlayout = new FormLayout();
    TextField username = new TextField("Username");
    TextField password = new TextField("Password");
    HorizontalLayout buttons = new HorizontalLayout();

    // Create buttons and define their listener methods.
    Button ok = new Button("OK", this, "okHandler");
    Button cancel = new Button("Cancel", this, "cancelHandler");

    // Have the unmodified Enter key cause an event
    Action action_ok = new ShortcutAction("Default key",
        ShortcutAction.KeyCode.ENTER, null);

    // Have the C key modified with Alt cause an event
    Action action_cancel = new ShortcutAction("Alt+C",
        ShortcutAction.KeyCode.C,
        new int[] { ShortcutAction.ModifierKey.ALT });

    public DefaultButtonExample() {
        // Set up the user interface
    }
}
```

```
        setCompositionRoot(panel);
        panel.addComponent(formlayout);
        formlayout.addComponent(username);
        formlayout.addComponent(password);
        formlayout.addComponent(buttons);
        buttons.addComponent(ok);
        buttons.addComponent(cancel);

        // Set focus to username
        username.focus();

        // Set this object as the action handler
        panel.addActionHandler(this);
    }

    /**
     * Retrieve actions for a specific component. This method
     * will be called for each object that has a handler; in
     * this example just for login panel. The returned action
     * list might as well be static list.
     */
    public Action[] getActions(Object target, Object sender) {
        System.out.println("getActions()");
        return new Action[] { action_ok, action_cancel };
    }

    /**
     * Handle actions received from keyboard. This simply directs
     * the actions to the same listener methods that are called
     * with ButtonClick events.
     */
    public void handleAction(Action action, Object sender,
                             Object target) {
        if (action == action_ok) {
            okHandler();
        }
        if (action == action_cancel) {
            cancelHandler();
        }
    }

    public void okHandler() {
        // Do something: report the click
        formlayout.addComponent(new Label("OK clicked. "
            + "User=" + username.getValue() + ", password="
            + password.getValue()));
    }

    public void cancelHandler() {
        // Do something: report the click
        formlayout.addComponent(new Label("Cancel clicked. User="
            + username.getValue() + ", password="
            + password.getValue()));
    }
}
```

Notice that the keyboard actions can currently be attached only to **Panels** and **Windows**. This can cause problems if you have components that require a certain key. For example, multi-line

TextField requires the **Enter** key. There is currently no way to filter the shortcut actions out while the focus is inside some specific component, so you need to avoid such conflicts.

12.5.4. Supported Key Codes and Modifier Keys

The shortcut key definitions require a key code to identify the pressed key and modifier keys, such as **Shift**, **Alt**, or **Ctrl**, to specify a key combination.

The key codes are defined in the **ShortcutAction.KeyCode** interface and are:

Keys *A* to *Z*
Normal letter keys

F1 to *F12*
Function keys

BACKSPACE, *DELETE*, *ENTER*, *ESCAPE*, *INSERT*, *TAB*
Control keys

NUM0 to *NUM9*
Number pad keys

ARROW_DOWN, *ARROW_UP*, *ARROW_LEFT*, *ARROW_RIGHT*
Arrow keys

HOME, *END*, *PAGE_UP*, *PAGE_DOWN*
Other movement keys

Modifier keys are defined in **ShortcutAction.ModifierKey** and are:

ModifierKey.ALT
Alt key

ModifierKey.CTRL
Ctrl key

ModifierKey.SHIFT
Shift key

All constructors and methods accepting modifier keys take them as a variable argument list following the key code, separated with commas. For example, the following defines a **Ctrl+Shift+N** key combination for a shortcut.

```
TextField name = new TextField("Name (Ctrl+Shift+N)");
name.addShortcutListener(
    new AbstractField.FocusShortcut(name, KeyCode.N,
                                    ModifierKey.CTRL,
                                    ModifierKey.SHIFT));
```

Supported Key Combinations

The actual possible key combinations vary greatly between browsers, as most browsers have a number of built-in shortcut keys, which can not be used in web applications. For example, Mozilla Firefox allows binding almost any key combination, while Opera does not even allow binding **Alt** shortcuts. Other browsers are generally in between these two. Also, the operating

system can reserve some key combinations and some computer manufacturers define their own system key combinations.

12.6. Printing

Vaadin does not have any special support for printing. There are two basic ways to print - in a printer controlled by the application server or by the user from the web browser. Printing in the application server is largely independent of the UI, you just have to take care that printing commands do not block server requests, possibly by running the print commands in another thread.

For client-side printing, most browsers support printing the web page. You can either print the current or a special print page that you open. The page can be styled for printing with special CSS rules, and you can hide unwanted elements. You can also print other than Vaadin UI content, such as HTML or PDF.

12.6.1. Printing the Browser Window

Vaadin does not have special support for launching the printing in browser, but you can easily use the JavaScript `print()` method that opens the print window of the browser.

```
Button print = new Button("Print This Page");
print.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        // Print the current page
        JavaScript.getCurrent().execute("print();");
    }
});
```

The button in the above example would print the current page, including the button itself. You can hide such elements in CSS, as well as otherwise style the page for printing. Style definitions for printing are defined inside a `@media print {}` block in CSS.

12.6.2. Opening a Print Window

You can open a browser window with a special UI for print content and automatically launch printing the content.

```
public static class PrintUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Have some content to print
        setContent(new Label(
            "<h1>Here's some dynamic content</h1>\n" +
            "<p>This is to be printed.</p>",
            ContentMode.HTML));

        // Print automatically when the window opens
        JavaScript.getCurrent().execute(
            "setTimeout(function() {" +
            "    print(); self.close();}, 0);");
    }
    ...

    // Create an opener extension
```

```
BrowserWindowOpener opener =
    new BrowserWindowOpener(PrintUI.class);
opener.setFeatures("height=200,width=400,resizable");

// A button to open the printer-friendly page.
Button print = new Button("Click to Print");
opener.extend(print);
```

How the browser opens the window, as an actual (popup) window or just a tab, depends on the browser. After printing, we automatically close the window with JavaScript `close()` call.

12.6.3. Printing PDF

To print content as PDF, you need to provide the downloadable content as a static or a dynamic resource, such as a **StreamResource**.

You can let the user open the resource using a **Link** component, or some other component with a **PopupWindowOpener** extension. When such a link or opener is clicked, the browser opens the PDF in the browser, in an external viewer (such as Adobe Reader), or lets the user save the document.

It is crucial to notice that clicking a **Link** or a **PopupWindowOpener** is a client-side operation. If you get the content of the dynamic PDF from the same UI state, you can not have the link or opener enabled, as then clicking it would not get the current UI content. Instead, you have to create the resource object before the link or opener are clicked. This usually requires a two-step operation, or having the print operation available in another view.

```
// A user interface for a (trivial) data model from which
// the PDF is generated.
final TextField name = new TextField("Name");
name.setValue("Slartibartfast");

// This has to be clicked first to create the stream resource
final Button ok = new Button("OK");

// This actually opens the stream resource
final Button print = new Button("Open PDF");
print.setEnabled(false);

ok.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        // Create the PDF source and pass the data model to it
        StreamSource source =
            new MyPdfSource((String) name.getValue());

        // Create the stream resource and give it a file name
        String filename = "pdf_printing_example.pdf";
        StreamResource resource =
            new StreamResource(source, filename);

        // These settings are not usually necessary. MIME type
        // is detected automatically from the file name, but
        // setting it explicitly may be necessary if the file
        // suffix is not ".pdf".
        resource.setMimeType("application/pdf");
        resource.getStream().setParameter(
            "Content-Disposition",
```

```
        "attachment; filename="+filename);

    // Extend the print button with an opener
    // for the PDF resource
    BrowserWindowOpener opener =
        new BrowserWindowOpener(resource);
    opener.extend(print);

    name.setEnabled(false);
    ok.setEnabled(false);
    print.setEnabled(true);
}
});

layout.addComponent(name);
layout.addComponent(ok);
layout.addComponent(print);
```

12.7. Google App Engine Integration

This section is not yet fully updated to Vaadin 7.

Vaadin includes support to run Vaadin applications in the Google App Engine (GAE). The most essential requirement for GAE is the ability to serialize the application state. Vaadin applications are serializable through the **java.io.Serializable** interface.

To run as a GAE application, an application must use **GAEVaadinServlet** instead of **VaadinServlet**, and of course implement the **java.io.Serializable** interface for all persistent classes. You also need to enable session support in `appengine-web.xml` with:

```
<sessions-enabled>true</sessions-enabled>
```

The Vaadin Project wizard can create the configuration files needed for GAE deployment. See Section 3.4.1, “Creating a Maven Project”. When the Google App Engine deployment configuration is selected, the wizard will create the project structure following the GAE Servlet convention instead of the regular Servlet convention. The main differences are:

- Source directory: `src/main/java`
- Output directory: `war/WEB-INF/classes`
- Content directory: `war`

12.7.1. Rules and Limitations

Running Vaadin applications in Google App Engine has the following rules and limitations:

- Avoid using the session for storage, usual App Engine limitations apply (no synchronization, that is, it is unreliable).
- Vaadin uses memcache for mutex, the key is of the form `_vmutex<sessionid>`.
- The Vaadin **WebApplicationContext** class is serialized separately into memcache and datastore; the memcache key is `_vac<sessionid>` and the datastore entity kind is `_vac` with identifiers of the type `_vac<sessionid>`.

- *Do not* update the application state when serving an **ConnectorResource** (such as **ClassResource**.`getStream()`).
- *Avoid* (or be very careful when) updating application state in a **TransactionListener** - it is called even when the application is not locked and won't be serialized (such as with **ConnectorResource**), and changes can therefore be lost (it should be safe to update things that can be safely discarded later, that is, valid only for the current request).
- The application remains locked during uploads - a progress bar is not possible.

12.8. Common Security Issues

12.8.1. Sanitizing User Input to Prevent Cross-Site Scripting

You can put raw HTML content in many components, such as the **Label** and **CustomLayout**, as well as in tooltips and notifications. In such cases, you should make sure that if the content has any possibility to come from user input, you must make sure that the content is safe before displaying it. Otherwise, a malicious user can easily make a cross-site scripting attack by injecting offensive JavaScript code in such components. See other sources for more information about cross-site scripting.

Offensive code can easily be injected with `<script>` markup or in tag attributes as events, such as `onLoad`.

Cross-site scripting vulnerabilities are browser dependent, depending on the situations in which different browsers execute scripting markup.

Therefore, if the content created by one user is shown to other users, the content must be sanitized. There is no generic way to sanitize user input, as different applications can allow different kinds of input. Pruning (X)HTML tags out is somewhat simple, but some applications may need to allow (X)HTML content. It is therefore the responsibility of the application to sanitize the input.

Character encoding can make sanitization more difficult, as offensive tags can be encoded so that they are not recognized by a sanitizer. This can be done, for example, with HTML character entities and with variable-width encodings such as UTF-8 or various CJK encodings, by abusing multiple representations of a character. Most trivially, you could input `<` and `>` with `<` and `>`, respectively. The input could also be malformed and the sanitizer must be able to interpret it exactly as the browser would, and different browsers can interpret malformed HTML and variable-width character encodings differently.

Notice that the problem applies also to user input from a **RichTextArea** is transmitted as HTML from the browser to server-side and is not sanitized. As the entire purpose of the **RichTextArea** component is to allow input of formatted text, you can not just remove all HTML tags. Also many attributes, such as `style`, should pass through the sanitization.

12.9. Navigating in an Application

Plain Vaadin applications do not have normal web page navigation as they usually run on a single page, as all Ajax applications do. Quite commonly, however, applications have different views between which the user should be able to navigate. The **Navigator** in Vaadin can be used for most cases of navigation. Views managed by the navigator automatically get a distinct URI fragment, which can be used to be able to bookmark the views and their states and to go back and forward in the browser history.

12.9.1. Setting Up for Navigation

The **Navigator** class manages a collection of *views* that implement the `View` interface. The views can be either registered beforehand or acquired from a *view provider*. When registering, the views must have a name identifier and be added to a navigator with `addView()`. You can register new views at any point. Once registered, you can navigate to them with `navigateTo()`.

Navigator manages navigation in a component container, which can be either a `ComponentContainer` (most layouts) or a `SingleComponentContainer` (**UI**, **Panel**, or **Window**). The component container is managed through a `ViewDisplay`. Two view displays are defined: **ComponentContainerViewDisplay** and **SingleComponentContainerViewDisplay**, for the respective component container types. Normally, you can let the navigator create the view display internally, as we do in the example below, but you can also create it yourself to customize it.

Let us consider the following UI with two views: start and main. Here, we define their names with enums to be typesafe. We manage the navigation with the UI class itself, which is a `SingleComponentContainer`.

```
public class NavigatorUI extends UI {
    Navigator navigator;
    protected static final String MAINVIEW = "main";

    @Override
    protected void init(VaadinRequest request) {
        getPage().setTitle("Navigation Example");

        // Create a navigator to control the views
        navigator = new Navigator(this, this);

        // Create and register the views
        navigator.addView("", new StartView());
        navigator.addView(MAINVIEW, new MainView());
    }
}
```

The **Navigator** automatically sets the URI fragment of the application URL. It also registers a `URIFragmentChangeListener` in the page

to show the view identified by the URI fragment if entered or navigated to in the browser. This also enables browser navigation history in the application.

View Providers

You can create new views dynamically using a *view provider* that implements the `ViewProvider` interface. A provider is registered in **Navigator** with `addProvider()`.

The `ClassBasedViewProvider` is a view provider that can dynamically create new instances of a specified view class based on the view name.

The `StaticViewProvider` returns an existing view instance based on the view name. The `addView()` in **Navigator** is actually just a shorthand for creating a static view provider for each registered view.

View Change Listeners

You can handle view changes also by implementing a `ViewChangeListener` and adding it to a **Navigator**. When a view change occurs, a listener receives a **ViewChangeEvent** object, which has references to the old and the activated view, the name of the activated view, as well as the fragment parameters.

12.9.2. Implementing a View

Views can be any objects that implement the `View` interface. When the `navigateTo()` is called for the navigator, or the application is opened with the URI fragment associated with the view, the navigator switches to the view and calls its `enter()` method.

To continue with the example, consider the following simple start view that just lets the user to navigate to the main view. It only pops up a notification when the user navigates to it and displays the navigation button.

```
/** A start view for navigating to the main view */
public class StartView extends VerticalLayout implements View {
    public StartView() {
        setSizeFull();

        Button button = new Button("Go to Main View",
            new Button.ClickListener() {
                @Override
                public void buttonClick(ClickEvent event) {
                    navigator.navigateTo(MAINVIEW);
                }
            });
        addComponent(button);
        setComponentAlignement(button, Alignment.MIDDLE_CENTER);
    }

    @Override
    public void enter(ViewChangeEvent event) {
        Notification.show("Welcome to the Animal Farm");
    }
}
```

You can initialize the view content in the constructor, as was done in the example above, or in the `enter()` method. The advantage with the latter method is that the view is attached to the view container as well as to the UI at that time, which is not the case in the constructor.

12.9.3. Handling URI Fragment Path

URI fragment part of a URL is the part after a hash `#` character. Is used for within-UI URLs, because it is the only part of the URL that can be changed with JavaScript from within a page without reloading the page. The URLs with URI fragments can be used for hyperlinking and bookmarking, as well as browser history, just like any other URLs. In addition, an exclamation mark `#!` after the hash marks that the page is a stateful AJAX page, which can be crawled by search engines. Crawling requires that the application also responds to special URLs to get the searchable content. URI fragments are managed by **Page**, which provides a low-level API.

URI fragments can be used with **Navigator** in two ways: for navigating to a view and to a state within a view. The URI fragment accepted by `navigateTo()` can have the view name at the

root, followed by fragment parameters after a slash (" /"). These parameters are passed to the `enter()` method in the `View`.

In the following example, we implement within-view navigation. Here we use the following declarative design for the view:

```
<vaadin-vertical-layout size-full>
  <vaadin-horizontal-layout size-full :expand>
    <vaadin-panel caption="List of Equals" height-full width-auto>
      <vaadin-vertical-layout _id="menuContent" width-auto margin/>
    </vaadin-panel>

    <vaadin-panel _id="equalPanel" caption="An Equal" size-full :expand/>
  </vaadin-horizontal-layout>

  <vaadin-button _id="logout">Logout</vaadin-button>
</vaadin-vertical-layout>
```

The view's logic code would be as follows:

```
/** Main view with a menu (with declarative layout design) */
@DesignRoot
public class MainView extends VerticalLayout implements View {
    // Menu navigation button listener
    class ButtonListener implements Button.ClickListener {
        String menuitem;
        public ButtonListener(String menuitem) {
            this.menuitem = menuitem;
        }

        @Override
        public void buttonClick(ClickEvent event) {
            // Navigate to a specific state
            navigator.navigateTo(MAINVIEW + "/" + menuitem);
        }
    }

    VerticalLayout menuContent;
    Panel equalPanel;
    Button logout;

    public MainView() {
        Design.read(this);

        menuContent.addComponent(new Button("Pig",
            new ButtonListener("pig")));
        menuContent.addComponent(new Button("Cat",
            new ButtonListener("cat")));
        menuContent.addComponent(new Button("Dog",
            new ButtonListener("dog")));
        menuContent.addComponent(new Button("Reindeer",
            new ButtonListener("reindeer")));
        menuContent.addComponent(new Button("Penguin",
            new ButtonListener("penguin")));
        menuContent.addComponent(new Button("Sheep",
            new ButtonListener("sheep")));

        // Allow going back to the start
        logout.addClickListener(event -> // Java 8
```

```
        navigator.navigateTo("");
    }

    @DesignRoot
    class AnimalViewer extends VerticalLayout {
        Label watching;
        Embedded pic;
        Label back;

        public AnimalViewer(String animal) {
            Design.read(this);

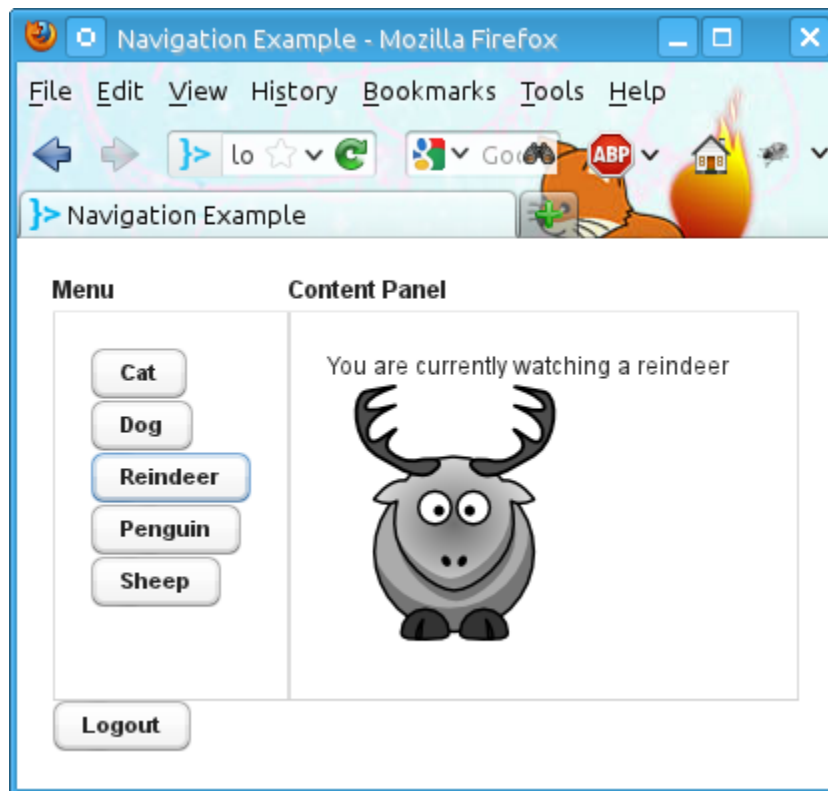
            watching.setValue("You are currently watching a " +
                               animal);
            pic.setSource(new ThemeResource(
                "img/" + animal + "-128px.png"));
            back.setValue("and " + animal +
                           " is watching you back");
        }
    }

    @Override
    public void enter(ViewChangeEvent event) {
        if (event.getParameters() == null
            || event.getParameters().isEmpty()) {
            equalPanel.setContent(
                new Label("Nothing to see here, " +
                           "just pass along."));
            return;
        } else
            equalPanel.setContent(new AnimalViewer(
                event.getParameters()));
    }
}
```

The animal sub-view would have the following declarative design:

```
<vaadin-vertical-layout size-full>
  <vaadin-label _id="watching" size-auto :middle :center/>
  <vaadin-embedded _id="pic" :middle :center :expand/>
  <vaadin-label _id="back" size-auto :middle :center/>
</vaadin-vertical-layout>
```

The main view is shown in Figure 12.9, “Navigator Main View”. At this point, the URL would be `http://localhost:8080/myapp#!main/reindeer`.

Figure 12.9. Navigator Main View

12.10. Advanced Application Architectures

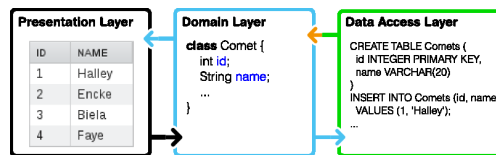
In this section, we continue from the basic application architectures described in Section 5.2, "Building the UI" and discuss some of the more advanced patterns that are often used in Vaadin applications.

12.10.1. Layered Architectures

Layered architectures, where each layer has a clearly distinct responsibility, are probably the most common architectures. Typically, applications follow at least a three-layer architecture:

- User interface (or presentation) layer
- Domain layer
- Data store layer

Such an architecture starts from a *domain model*, which defines the data model and the "business logic" of the application, typically as beans or POJOs. A user interface is built on top of the domain model, in our context with the Vaadin Framework. The Vaadin user interface could be bound directly to the data model through the Vaadin Data Model, described in Chapter 10, *Binding Components to Data*. Beneath the domain model lies a data store, such as a relational database. The dependencies between the layers are restricted so that a higher layer may depend on a lower one, but never the other way around.

Figure 12.10. Three-layer architecture

An *application layer* (or *service layer*) is often distinguished from the domain layer, offering the domain logic as a service, which can be used by the user interface layer, as well as for other uses. In Java EE development, Enterprise JavaBeans (EJBs) are typically used for building this layer.

An *infrastructure layer* (or *data access layer*) is often distinguished from the data store layer, with a purpose to abstract the data store. For example, it could involve a persistence solution such as JPA and an EJB container. This layer becomes relevant with Vaadin when binding Vaadin components to data with the JPAContainer, as described in Chapter 20, *Vaadin JPAContainer*.

12.10.2. Model-View-Presenter Pattern

The Model-View-Presenter (MVP) pattern is one of the most common patterns in developing large applications with Vaadin. It is similar to the older Model-View-Controller (MVC) pattern, which is not as meaningful in Vaadin development. Instead of an implementation-aware controller, there is an implementation-agnostic presenter that operates the view through an interface. The view does not interact directly with the model. This isolates the view implementation better than in MVC and allows easier unit testing of the presenter and model.

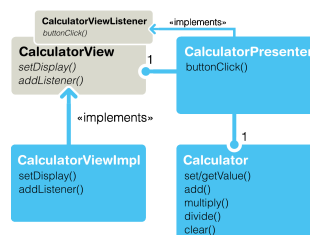
Figure 12.11. Model-View-Presenter pattern

Figure 12.11, “Model-View-Presenter pattern” illustrates the MVP pattern with a simple calculator. The domain model is realized in the **Calculator** class, which includes a data model and some model logic operations. The **CalculatorViewImpl** is a Vaadin implementation of the view, defined in the `CalculatorView` interface. The **CalculatorPresenter** handles the user interface logic. User interaction events received in the view are translated into implementation-independent events for the presenter to handle (the view implementation could also just call the presenter).

Let us first look how the model and view are bound together by the presenter in the following example:

```
// Create the model and the Vaadin view implementation
CalculatorModel    model = new CalculatorModel();
CalculatorViewImpl view = new CalculatorViewImpl();

// The presenter binds the model and view together
new CalculatorPresenter(model, view);
```

```
// The view implementation is a Vaadin component
layout.addComponent(view);
```

You could add the view anywhere in a Vaadin application, as it is a composite component.

The Model

Our business model is quite simple, with one value and a number of operations for manipulating it.

```
/** The model */
class CalculatorModel {
    private double value = 0.0;

    public void clear() {
        value = 0.0;
    }

    public void add(double arg) {
        value += arg;
    }

    public void multiply(double arg) {
        value *= arg;
    }

    public void divide(double arg) {
        if (arg != 0.0)
            value /= arg;
    }

    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
    }
}
```

The View

The purpose of the view in MVP is to display data and receive user interaction. It relays the user interaction to the presenter in a fashion that is independent of the view implementation, that is, no Vaadin events. It is defined as a UI framework interface that can have multiple implementations.

```
interface CalculatorView {
    public void setDisplay(double value);

    interface CalculatorViewListener {
        void buttonClick(char operation);
    }

    public void addListener(CalculatorViewListener listener);
}
```

There are design alternatives for the view. It could receive the listener in its constructor, or it could just know the presenter. Here, we forward button clicks as an implementation-independent event.

As we are using Vaadin, we make a Vaadin implementation of the interface as follows:

```
class CalculatorViewImpl extends CustomComponent
    implements CalculatorView, ClickListener {
    private Label display = new Label("0.0");

    public CalculatorViewImpl() {
        GridLayout layout = new GridLayout(4, 5);

        // Create a result label that spans over all
        // the 4 columns in the first row
        layout.addComponent(display, 0, 0, 3, 0);

        // The operations for the calculator in the order
        // they appear on the screen (left to right, top
        // to bottom)
        String[] operations = new String[] {
            "7", "8", "9", "/", "4", "5", "6",
            "*", "1", "2", "3", "-", "0", "=", "C", "+" };

        // Add buttons and have them send click events
        // to this class
        for (String caption: operations)
            layout.addComponent(new Button(caption, this));

        setCompositionRoot(layout);
    }

    public void setDisplay(double value) {
        display.setValue(Double.toString(value));
    }

    /* Only the presenter registers one listener... */
    List<CalculatorViewListener> listeners =
        new ArrayList<CalculatorViewListener>();

    public void addListener(CalculatorViewListener listener) {
        listeners.add(listener);
    }

    /** Relay button clicks to the presenter with an
     *  implementation-independent event */
    @Override
    public void buttonClick(ClickEvent event) {
        for (CalculatorViewListener listener: listeners)
            listener.buttonClick(event.getButton()
                                   .getCaption().charAt(0));
    }
}
```


The Presenter

The presenter in MVP is a middle-man that handles all user interaction logic, but in an implementation-independent way, so that it doesn't actually know anything about Vaadin. It shows data in the view and receives user interaction back from it.

```
class CalculatorPresenter
    implements CalculatorView.CalculatorViewListener {
    CalculatorModel model;
    CalculatorView view;

    private double current = 0.0;
    private char lastOperationRequested = 'C';

    public CalculatorPresenter(CalculatorModel model,
                              CalculatorView view) {
        this.model = model;
        this.view = view;

        view.setDisplay(current);
        view.addListener(this);
    }

    @Override
    public void buttonClick(char operation) {
        // Handle digit input
        if ('0' <= operation && operation <= '9') {
            current = current * 10
                + Double.parseDouble("" + operation);
            view.setDisplay(current);
            return;
        }

        // Execute the previously input operation
        switch (lastOperationRequested) {
            case '+':
                model.add(current);
                break;
            case '-':
                model.add(-current);
                break;
            case '/':
                model.divide(current);
                break;
            case '*':
                model.multiply(current);
                break;
            case 'C':
                model.setValue(current);
                break;
        } // '=' is implicit

        lastOperationRequested = operation;

        current = 0.0;
        if (operation == 'C')
            model.clear();
        view.setDisplay(model.getValue());
    }
}
```

```
}  
}
```

In the above example, we held some state information in the presenter. Alternatively, we could have had an intermediate controller between the presenter and the model to handle the low-level button logic.

12.11. Managing URI Fragments

A major issue in AJAX applications is that as they run in a single web page, bookmarking the application URL (or more generally the *URI*) can only bookmark the application, not an application state. This is a problem for many applications, such as product catalogs and discussion forums, in which it would be good to provide links to specific products or messages. Consequently, as browsers remember the browsing history by URI, the history and the **Back** button do not normally work. The solution is to use the *fragment identifier* part of the URI, which is separated from the primary part (address + path + optional query parameters) of the URI with the hash (#) character. For example:

```
http://example.com/path#myfragment
```

The exact syntax of the fragment identifier part is defined in RFC 3986 (Internet standard STD 66) that defines the URI syntax. A fragment may only contain the regular URI *path characters* (see the standard) and additionally the slash and the question mark.

Vaadin offers two ways to enable the use of URI fragments: the high-level **Navigator** utility described in Section 12.9, “Navigating in an Application” and the low-level API described here.

12.11.1. Setting the URI Fragment

You can set the current fragment identifier with the `setUriFragment()` method in the **Page** object.

```
Page.getCurrent().setUriFragment("mars");
```

Setting the URI fragment causes an `UriFragmentChangeEvent`, which is processed in the same server request. As with UI rendering, the URI fragment is changed in the browser after the currently processed server request returns the response.

Prefixing the fragment identifier with an exclamation mark enables the web crawler support described in Section 12.11.4, “Supporting Web Crawling”.

12.11.2. Reading the URI Fragment

The current URI fragment can be acquired with the `getUriFragment()` method from the current **Page** object. The fragment is known when the `init()` method of the UI is called.

```
// Read initial URI fragment to create UI content  
String fragment = getPage().getUriFragment();  
enter(fragment);
```

To enable reusing the same code when the URI fragment is changed, as described next, it is usually best to build the relevant part of the UI in a separate method. In the above example, we called an `enter()` method, in a way that is similar to handling view changes with **Navigator**.

12.11.3. Listening for URI Fragment Changes

After the UI has been initialized, changes in the URI fragment can be handled with a `UriFragmentChangeListener`. The listeners are called when the URI fragment changes, but not when the UI is initialized, where the current fragment is available from the page object as described earlier.

For example, we could define the listener as follows in the `init()` method of a UI class:

```
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        getPage().addUriFragmentChangeListener(
            new UriFragmentChangeListener() {
                public void uriFragmentChanged(
                    UriFragmentChangedEvent source) {
                    enter(source.getUriFragment());
                }
            }
        );

        // Read the initial URI fragment
        enter(getPage().getUriFragment());
    }

    void enter(String fragment) {
        ... initialize the UI ...
    }
}
```

Figure 12.12, “Application State Management with URI Fragment Utility” shows an application that allows specifying the menu selection with a URI fragment and correspondingly sets the fragment when the user selects a menu item.

Figure 12.12. Application State Management with URI Fragment Utility



12.11.4. Supporting Web Crawling

Stateful AJAX applications can not normally be crawled by a search engine, as they run in a single page and a crawler can not navigate the states even if URI fragments are enabled. The Google search engine and crawler support a convention where the fragment identifiers are prefixed with exclamation mark, such as `#!myfragment`. The servlet needs to have a separate searchable content page accessible with the same URL, but with a `_escaped_fragment_` parameter. For example, for `/myapp/myui#!myfragment` it would be `/myapp/myui?_escaped_fragment_=myfragment`.

You can provide the crawl content by overriding the `service()` method in a custom servlet class. For regular requests, you should call the super implementation in the **VaadinServlet** class.

```
public class MyCustomServlet extends VaadinServlet
    @Override
    protected void service(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        String fragment = request
            .getParameter("_escaped_fragment_");
        if (fragment != null) {
            response.setContentType("text/html");
            Writer writer = response.getWriter();
            writer.append("<html><body>"+
                "<p>Here is some crawlable "+
                "content about " + fragment + "</p>");

            // A list of all crawlable pages
            String items[] = {"mercury", "venus",
                             "earth", "mars"};
            writer.append("<p>Index of all content:</p><ul>");
            for (String item: items) {
                String url = request.getContextPath() +
                    request.getServletPath() +
                    request.getPathInfo() + "#!" + item;
                writer.append("<li><a href='" + url + "'" +
                    item + "</a></li>");
            }
            writer.append("</ul></body>");
        } else
            super.service(request, response);
    }
}
```

The crawlable content does not need to be human readable. It can provide an index of links to other application states, as we did in the example above. The links should use the "#!" notation, but can not be relative to avoid having the `_escaped_fragment_` parameter.

You need to use the custom servlet class in the `web.xml` deployment descriptor instead of the normal **VaadinServlet** class, as described in Section 5.9.4, "Using a `web.xml` Deployment Descriptor".

12.12. Drag and Drop

Dragging an object from one location to another by grabbing it with mouse, holding the mouse button pressed, and then releasing the button to "drop" it to the other location is a common way to move, copy, or associate objects. For example, most operating systems allow dragging and dropping files between folders or dragging a document on a program to open it. In Vaadin, it is possible to drag and drop components and parts of certain components.

Dragged objects, or *transferables*, are essentially data objects. You can drag and drop rows in **Table** and nodes in **Tree** components, either within or between the components. You can also drag entire components by wrapping them inside **DragAndDropWrapper**.

Dragging starts from a *drag source*, which defines the transferable. Transferables implement the **Transferable** interfaces. For trees and tables, which are bound to **Container** data sources,

a node or row transferable is a reference to an **Item** in the Vaadin Data Model. Dragged components are referenced with a **WrapperTransferable**. Starting dragging does not require any client-server communication, you only need to enable dragging. All drag and drop logic occurs in two operations: determining (*accepting*) where dropping is allowed and actually dropping. Drops can be done on a *drop target*, which implements the **DropTarget** interface. Three components implement the interface: **Tree**, **Table**, and **DragAndDropWrapper**. These accept and drop operations need to be provided in a *drop handler*. Essentially all you need to do to enable drag and drop is to enable dragging in the drag source and implement the `getAcceptCriterion()` and `drop()` methods in the **DropHandler** interface.

The client-server architecture of Vaadin causes special requirements for the drag and drop functionality. The logic for determining where a dragged object can be dropped, that is, *accepting* a drop, should normally be done on the client-side, in the browser. Server communications are too slow to have much of such logic on the server-side. The drag and drop feature therefore offers a number of ways to avoid the server communications to ensure a good user experience.

12.12.1. Handling Drops

Most of the user-defined drag and drop logic occurs in a *drop handler*, which is provided by implementing the `drop()` method in the **DropHandler** interface. A closely related definition is the drop accept criterion, which is defined in the `getAcceptCriterion()` method in the same interface. It is described in Section 12.12.4, “Accepting Drops” later.

The `drop()` method gets a **DragAndDropEvent** as its parameters. The event object provides references to two important object: **Transferable** and **TargetDetails**.

A **Transferable** contains a reference to the object (component or data item) that is being dragged. A tree or table item is represented as a **TreeTransferable** or **TableTransferable** object, which carries the item identifier of the dragged tree or table item. These special transferables, which are bound to some data in a container, are **DataBoundTransferable**. Dragged components are represented as **WrapperTransferable** objects, as the components are wrapped in a **DragAndDropWrapper**.

The **TargetDetails** object provides information about the exact location where the transferable object is being dropped. The exact class of the details object depends on the drop target and you need to cast it to the proper subclass to get more detailed information. If the target is selection component, essentially a tree or a table, the **AbstractSelectTargetDetails** object tells the item on which the drop is being made. For trees, the **TreeTargetDetails** gives some more details. For wrapped components, the information is provided in a **WrapperDropDetails** object. In addition to the target item or component, the details objects provide a *drop location*. For selection components, the location can be obtained with the `getDropLocation()` and for wrapped components with `verticalDropLocation()` and `horizontalDropLocation()`. The locations are specified as either **VerticalDropLocation** or **HorizontalDropLocation** objects. The drop location objects specify whether the transferable is being dropped above, below, or directly on (at the middle of) a component or item.

Dropping on a **Tree**, **Table**, and a wrapped component is explained further in the following sections.

12.12.2. Dropping Items On a Tree

You can drag items from, to, or within a **Tree**. Making tree a drag source requires simply setting the drag mode with `setDragMode()`. **Tree** currently supports only one drag mode, `TreeDragMode.NODE`, which allows dragging single tree nodes. While dragging, the dragged

node is referenced with a **TreeTransferable** object, which is a **DataBoundTransferable**. The tree node is identified by the item ID of the container item.

When a transferable is dropped on a tree, the drop location is stored in a **TreeTargetDetails** object, which identifies the target location by item ID of the tree node on which the drop is made. You can get the item ID with `getItemIdOver()` method in **AbstractSelectTargetDetails**, which the **TreeTargetDetails** inherits. A drop can occur directly on or above or below a node; the exact location is a **VerticalDropLocation**, which you can get with the `getDropLocation()` method.

In the example below, we have a **Tree** and we allow reordering the tree items by drag and drop.

```
final Tree tree = new Tree("Inventory");
tree.setContainerDataSource(TreeExample.createTreeContent());
layout.addComponent(tree);

// Expand all items
for (Iterator<?> it = tree.rootItemIds().iterator(); it.hasNext();)
    tree.expandItemsRecursively(it.next());

// Set the tree in drag source mode
tree.setDragMode(TreeDragMode.NODE);

// Allow the tree to receive drag drops and handle them
tree.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }

    public void drop(DragAndDropEvent event) {
        // Wrapper for the object that is dragged
        Transferable t = event.getTransferable();

        // Make sure the drag source is the same tree
        if (t.getSourceComponent() != tree)
            return;

        TreeTargetDetails target = (TreeTargetDetails)
            event.getTargetDetails();

        // Get ids of the dragged item and the target item
        Object sourceItemId = t.getData("itemId");
        Object targetItemId = target.getItemIdOver();

        // On which side of the target the item was dropped
        VerticalDropLocation location = target.getDropLocation();

        HierarchicalContainer container = (HierarchicalContainer)
            tree.getContainerDataSource();

        // Drop right on an item -> make it a child
        if (location == VerticalDropLocation.MIDDLE)
            tree.setParent(sourceItemId, targetItemId);

        // Drop at the top of a subtree -> make it previous
        else if (location == VerticalDropLocation.TOP) {
            Object parentId = container.getParent(targetItemId);
            container.setParent(sourceItemId, parentId);
        }
    }
});
```

```
        container.moveAfterSibling(sourceItemId, targetItemId);
        container.moveAfterSibling(targetItemId, sourceItemId);
    }

    // Drop below another item -> make it next
    else if (location == VerticalDropLocation.BOTTOM) {
        Object parentId = container.getParent(targetItemId);
        container.setParent(sourceItemId, parentId);
        container.moveAfterSibling(sourceItemId, targetItemId);
    }
}
});
```

Accept Criteria for Trees

Tree defines some specialized accept criteria for trees.

TargetInSubtree(client-side)

Accepts if the target item is in the specified sub-tree. The sub-tree is specified by the item ID of the root of the sub-tree in the constructor. The second constructor includes a depth parameter, which specifies how deep from the given root node are drops accepted. Value `-1` means infinite, that is, the entire sub-tree, and is therefore the same as the simpler constructor.

TargetItemAllowsChildren(client-side)

Accepts a drop if the tree has `setChildrenAllowed()` enabled for the target item. The criterion does not require parameters, so the class is a singleton and can be acquired with `Tree.TargetItemAllowsChildren.get()`. For example, the following composite criterion accepts drops only on nodes that allow children, but between all nodes:

```
return new Or (Tree.TargetItemAllowsChildren.get(), new Not(Vertical-
LocationIs.MIDDLE));
```

TreeDropCriterion(server-side)

Accepts drops on only some items, which as specified by a set of item IDs. You must extend the abstract class and implement the `getAllowedItemIds()` to return the set. While the criterion is server-side, it is lazy-loading, so that the list of accepted target nodes is loaded only once from the server for each drag operation. See Section 12.12.4, “Accepting Drops” for an example.

In addition, the accept criteria defined in **AbstractSelect** are available for a **Tree**, as listed in Section 12.12.4, “Accepting Drops”.

12.12.3. Dropping Items On a Table

You can drag items from, to, or within a **Table**. Making table a drag source requires simply setting the drag mode with `setDragMode()`. **Table** supports dragging both single rows, with `TableDragMode.ROW`, and multiple rows, with `TableDragMode.MULTIROW`. While dragging, the dragged node or nodes are referenced with a **TreeTransferable** object, which is a **DataBoundTransferable**. Tree nodes are identified by the item IDs of the container items.

When a transferable is dropped on a table, the drop location is stored in a **AbstractSelectTargetDetails** object, which identifies the target row by its item ID. You can get the item ID with `getItemIdOver()` method. A drop can occur directly on or above or below a row; the exact

location is a **VerticalDropLocation**, which you can get with the `getDropLocation()` method from the details object.

Accept Criteria for Tables

Table defines one specialized accept criterion for tables.

TableDropCriterion(server-side)

Accepts drops only on (or above or below) items that are specified by a set of item IDs. You must extend the abstract class and implement the `getAllowedItemIds()` to return the set. While the criterion is server-side, it is lazy-loading, so that the list of accepted target items is loaded only once from the server for each drag operation.

12.12.4. Accepting Drops

You can not drop the objects you are dragging around just anywhere. Before a drop is possible, the specific drop location on which the mouse hovers must be *accepted*. Hovering a dragged object over an accepted location displays an *accept indicator*, which allows the user to position the drop properly. As such checks have to be done all the time when the mouse pointer moves around the drop targets, it is not feasible to send the accept requests to the server-side, so drops on a target are normally accepted by a client-side *accept criterion*.

A drop handler must define the criterion on the objects which it accepts to be dropped on the target. The criterion needs to be provided in the **getAcceptCriterion()** method of the **DropHandler** interface. A criterion is represented in an **AcceptCriterion** object, which can be a composite of multiple criteria that are evaluated using logical operations. There are two basic types of criteria: *client-side* and *server-side* criteria. The various built-in criteria allow accepting drops based on the identity of the source and target components, and on the *data flavor* of the dragged objects.

To allow dropping any transferable objects, you can return a universal accept criterion, which you can get with `AcceptAll.get()`.

```
tree.setDropHandler(new DropHandler() {  
    public AcceptCriterion getAcceptCriterion() {  
        return AcceptAll.get();  
    }  
    ...  
});
```

Client-Side Criteria

The *client-side* criteria, which inherit the **ClientSideCriterion**, are verified on the client-side, so server requests are not needed for verifying whether each component on which the mouse pointer hovers would accept a certain object.

The following client-side criteria are define in `com.vaadin.event.dd.acceptcriterion`:

AcceptAll

Accepts all transferables and targets.

And

Performs the logical AND operation on two or more client-side criteria; accepts the transferable if all the given sub-criteria accept it.

ContainsDataFlavour

The transferable must contain the defined data flavour.

Not

Performs the logical NOT operation on a client-side criterion; accepts the transferable if and only if the sub-criterion does not accept it.

Or

Performs the logical OR operation on two or more client-side criteria; accepts the transferable if any of the given sub-criteria accepts it.

Sources

Accepts all transferables from any of the given source components

SourcesTarget

Accepts the transferable only if the source component is the same as the target. This criterion is useful for ensuring that items are dragged only within a tree or a table, and not from outside it.

TargetDetails

Accepts any transferable if the target detail, such as the item of a tree node or table row, is of the given data flavor and has the given value.

In addition, target components such as **Tree** and **Table** define some component-specific client-side accept criteria. See Section 12.12.2, “Dropping Items On a **Tree**” for more details.

AbstractSelect defines the following criteria for all selection components, including **Tree** and **Table**.

AcceptItem

Accepts only specific items from a specific selection component. The selection component, which must inherit **AbstractSelect**, is given as the first parameter for the constructor. It is followed by a list of allowed item identifiers in the drag source.

AcceptItem.ALL

Accepts all transferables as long as they are items.

TargetItems

Accepts all drops on the specified target items. The constructor requires the target component (**AbstractSelect**) followed by a list of allowed item identifiers.

VerticalLocationIs.MIDDLE, TOP, and [classname]BOTTOM

The three static criteria accepts drops on, above, or below an item. For example, you could accept drops only in between items with the following:

```
public AcceptCriterion getAcceptCriterion() {  
    return new Not (VerticalLocationIs.MIDDLE);  
}
```

Server-Side Criteria

The *server-side criteria* are verified on the server-side with the `accept()` method of the **ServerSideCriterion** class. This allows fully programmable logic for accepting drops, but the negative side is that it causes a very large amount of server requests. A request is made for every target position on which the pointer hovers. This problem is eased in many cases by the component-specific lazy loading criteria **TableDropCriterion** and **TreeDropCriterion**. They

do the server visit once for each drag and drop operation and return all accepted rows or nodes for current **Transferable** at once.

The `accept()` method gets the drag event as a parameter so it can perform its logic much like in `drop()`.

```
public AcceptCriterion getAcceptCriterion() {
    // Server-side accept criterion that allows drops on any other
    // location except on nodes that may not have children
    ServerSideCriterion criterion = new ServerSideCriterion() {
        public boolean accept(DragAndDropEvent dragEvent) {
            TreeTargetDetails target = (TreeTargetDetails)
                dragEvent.getTargetDetails();

            // The tree item on which the load hovers
            Object targetItemId = target.getItemIdOver();

            // On which side of the target the item is hovered
            VerticalDropLocation location = target.getDropLocation();
            if (location == VerticalDropLocation.MIDDLE)
                if (! tree.areChildrenAllowed(targetItemId))
                    return false; // Not accepted

            return true; // Accept everything else
        }
    };
    return criterion;
}
```

The server-side criteria base class **ServerSideCriterion** provides a generic `accept()` method. The more specific **TableDropCriterion** and **TreeDropCriterion** are convenience extensions that allow defining allowed drop targets as a set of items. They also provide some optimization by lazy loading, which reduces server communications significantly.

```
public AcceptCriterion getAcceptCriterion() {
    // Server-side accept criterion that allows drops on any
    // other tree node except on node that may not have children
    TreeDropCriterion criterion = new TreeDropCriterion() {
        @Override
        protected Set<Object> getAllowedItemIds(
            DragAndDropEvent dragEvent, Tree tree) {
            HashSet<Object> allowed = new HashSet<Object>();
            for (Iterator<Object> i =
                tree.getItemIds().iterator(); i.hasNext();) {
                Object itemId = i.next();
                if (tree.hasChildren(itemId))
                    allowed.add(itemId);
            }
            return allowed;
        }
    };
    return criterion;
}
```

Accept Indicators

When a dragged object hovers on a drop target, an *accept indicator* is displayed to show whether or not the location is accepted. For *MIDDLE* location, the indicator is a box around the

target (tree node, table row, or component). For vertical drop locations, the accepted locations are shown as horizontal lines, and for horizontal drop locations as vertical lines.

For **DragAndDropWrapper** drop targets, you can disable the accept indicators or *drag hints* with the *no-vertical-drag-hints*, *no-horizontal-drag-hints*, and *no-box-drag-hints* styles. You need to add the styles to the *layout that contains* the wrapper, not to the wrapper itself.

```
// Have a wrapper
DragAndDropWrapper wrapper = new DragAndDropWrapper(c);
layout.addComponent(wrapper);

// Disable the hints
layout.addStyleName("no-vertical-drag-hints");
layout.addStyleName("no-horizontal-drag-hints");
layout.addStyleName("no-box-drag-hints");
```

12.12.5. Dragging Components

Dragging a component requires wrapping the source component within a **DragAndDropWrapper**. You can then allow dragging by putting the wrapper (and the component) in drag mode with `setDragStartMode()`. The method supports two drag modes: `DragStartMode.WRAPPER` and `DragStartMode.COMPONENT`, which defines whether the entire wrapper is shown as the drag image while dragging or just the wrapped component.

```
// Have a component to drag
final Button button = new Button("An Absolute Button");

// Put the component in a D&D wrapper and allow dragging it
final DragAndDropWrapper buttonWrap = new DragAndDropWrapper(button);
buttonWrap.setDragStartMode(DragStartMode.COMPONENT);

// Set the wrapper to wrap tightly around the component
buttonWrap.setSizeUndefined();

// Add the wrapper, not the component, to the layout
layout.addComponent(buttonWrap, "left: 50px; top: 50px;");
```

The default height of **DragAndDropWrapper** is undefined, but the default width is 100%. If you want to ensure that the wrapper fits tightly around the wrapped component, you should call `setSizeUndefined()` for the wrapper. Doing so, you should make sure that the wrapped component does not have a relative size, which would cause a paradox.

Dragged components are referenced in the **WrapperTransferable**. You can get the reference to the dragged component with `getDraggedComponent()`. The method will return `null` if the transferable is not a component. Also HTML 5 drags (see later) are held in wrapper transferables.

12.12.6. Dropping on a Component

Drops on a component are enabled by wrapping the component in a **DragAndDropWrapper**. The wrapper is an ordinary component; the constructor takes the wrapped component as a parameter. You just need to define the **DropHandler** for the wrapper with `setDropHandler()`.

In the following example, we allow moving components in an absolute layout. Details on the drop handler are given later.

```
// A layout that allows moving its contained components
// by dragging and dropping them
final AbsoluteLayout absLayout = new AbsoluteLayout();
absLayout.setWidth("100%");
absLayout.setHeight("400px");

... put some (wrapped) components in the layout ...

// Wrap the layout to allow handling drops
DragAndDropWrapper layoutWrapper =
    new DragAndDropWrapper(absLayout);

// Handle moving components within the AbsoluteLayout
layoutWrapper.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }

    public void drop(DragAndDropEvent event) {
        ...
    }
});
```

Target Details for Wrapped Components

The drop handler receives the drop target details in a **WrapperTargetDetails** object, which implements the **TargetDetails** interface.

```
public void drop(DragAndDropEvent event) {
    WrapperTransferable t =
        (WrapperTransferable) event.getTransferable();
    WrapperTargetDetails details =
        (WrapperTargetDetails) event.getTargetDetails();
```

The wrapper target details include a **MouseEventDetails** object, which you can get with `getMouseEvent()`. You can use it to get the mouse coordinates for the position where the mouse button was released and the drag ended. Similarly, you can find out the drag start position from the transferable object (if it is a **WrapperTransferable**) with `getMouseDownEvent()`.

```
// Calculate the drag coordinate difference
int xChange = details.getMouseEvent().getClientX()
    - t.getMouseDownEvent().getClientX();
int yChange = details.getMouseEvent().getClientY()
    - t.getMouseDownEvent().getClientY();

// Move the component in the absolute layout
ComponentPosition pos =
    absLayout.getPosition(t.getSourceComponent());
pos.setLeftValue(pos.getLeftValue() + xChange);
pos.setTopValue(pos.getTopValue() + yChange);
```

You can get the absolute x and y coordinates of the target wrapper with `getAbsoluteLeft()` and `getAbsoluteTop()`, which allows you to translate the absolute mouse coordinates to coordinates relative to the wrapper. Notice that the coordinates are really the position of the wrapper, not the wrapped component; the wrapper reserves some space for the accept indicators.

The `verticalDropLocation()` and `horizontalDropLocation()` return the more detailed drop location in the target.

12.12.7. Dragging Files from Outside the Browser

The **DragAndDropWrapper** allows dragging files from outside the browser and dropping them on a component wrapped in the wrapper. Dropped files are automatically uploaded to the application and can be acquired from the wrapper with `getFiles()`. The files are represented as **Html5File** objects as defined in the inner class. You can define an upload **Receiver** to receive the content of a file to an **OutputStream**.

Dragging and dropping files to browser is supported in HTML 5 and requires a compatible browser, such as Mozilla Firefox 3.6 or newer.

12.13. Logging

You can do logging in Vaadin application using the standard `java.util.logging` facilities. Configuring logging is as easy as putting a file named `logging.properties` in the default package of your Vaadin application (`src` in an Eclipse project or `src/main/java` or `src/main/resources` in a Maven project). This file is read by the **Logger** class when a new instance of it is initialize.

12.13.1. Logging in Apache Tomcat

For logging Vaadin applications deployed in Apache Tomcat, you do not need to do anything special to log to the same place as Tomcat itself. If you need to write the Vaadin application related messages elsewhere, just add a custom `logging.properties` file to the default package of your Vaadin application.

If you would like to pipe the log messages through another logging solution, see Section 12.13.3, “Piping to Log4j using SLF4J” below.

12.13.2. Logging in Liferay

Liferay mutes logging through `java.util.logging` by default. In order to enable logging, you need to add a `logging.properties` file of your own to the default package of your Vaadin application. This file should define at least one destination where to save the log messages.

You can also log through SLF4J, which is used in and bundled with Liferay. Follow the instructions in Section 12.13.3, “Piping to Log4j using SLF4J”.

12.13.3. Piping to Log4j using SLF4J

Piping output from `java.util.logging` to Log4j is easy with SLF4J (<http://slf4j.org/>). The basic way to go about this is to add the SLF4J JAR file as well as the `jul-to-slf4j.jar` file, which implements the bridge from `java.util.logging`, to SLF4J. You will also need to add a third logging implementation JAR file, that is, `slf4j-log4j12-x.x.x.jar`, to log the actual messages using Log4j. For more info on this, please visit the SLF4J site.

In order to get the `java.util.logging` to SLF4J bridge installed, you need to add the following snippet of code to your **UI** class at the very top://TODO: Sure it's UI class and not the servlet?

```
static {
    SLF4JBridgeHandler.install();
}
```

This will make sure that the bridge handler is installed and working before Vaadin starts to process any logging calls.



Please note!

This can seriously impact on the cost of disabled logging statements (60-fold increase) and a measurable impact on enabled log statements (20% overall increase). However, Vaadin doesn't log very much, so the effect on performance will be negligible.

12.13.4. Using Logger

You can do logging with a simple pattern where you register a static logger instance in each class that needs logging, and use this logger wherever logging is needed in the class. For example:

```
public class MyClass {
    private final static Logger logger =
        Logger.getLogger(MyClass.class.getName());

    public void myMethod() {
        try {
            // do something that might fail
        } catch (Exception e) {
            logger.log(Level.SEVERE, "FAILED CATASTROPHICALLY!", e);
        }
    }
}
```

Having a static logger instance for each class needing logging saves a bit of memory and time compared to having a logger for every logging class instance. However, it could cause the application to leak PermGen memory with some application servers when redeploying the application. The problem is that the **Logger** may maintain hard references to its instances. As the **Logger** class is loaded with a classloader shared between different web applications, references to classes loaded with a per-application classloader would prevent garbage-collecting the classes after redeploying, hence leaking memory. As the size of the PermGen memory where class objects are stored is fixed, the leakage will lead to a server crash after many redeployments. The issue depends on the way how the server manages classloaders, on the hardness of the back-references, and may also be different between Java 6 and 7. So, if you experience PermGen issues, or want to play it on the safe side, you should consider using non-static **Logger** instances.//As discussed in Forum thread 1175841 (24.2.2012).

12.14. JavaScript Interaction

Vaadin supports two-direction JavaScript calls from and to the server-side. This allows interfacing with JavaScript code without writing client-side integration code.

12.14.1. Calling JavaScript

You can make JavaScript calls from the server-side with the `execute()` method in the **JavaScript** class. You can get a **JavaScript** instance from the current **Page** object with `getJavaScript()`. //TODO Check that the API is so.

```
// Execute JavaScript in the currently processed page
Page.getCurrent().getJavaScript().execute("alert('Hello')");
```

The **JavaScript** class itself has a static shorthand method `getCurrent()` to get the instance for the currently processed page.

```
// Shorthand
JavaScript.getCurrent().execute("alert('Hello')");
```

The JavaScript is executed after the server request that is currently processed returns. If multiple JavaScript calls are made during the processing of the request, they are all executed sequentially after the request is done. Hence, the JavaScript execution does not pause the execution of the server-side application and you can not return values from the JavaScript.

12.14.2. Handling JavaScript Function Callbacks

You can make calls with JavaScript from the client-side to the server-side. This requires that you register JavaScript call-back methods from the server-side. You need to implement and register a **JavaScriptFunction** with `addFunction()` in the current **JavaScript** object. A function requires a name, with an optional package path, which are given to the `addFunction()`. You only need to implement the `call()` method to handle calls from the client-side JavaScript.

```
JavaScript.getCurrent().addFunction("com.example.foo.myfunc",
                                   new JavaScriptFunction() {
    @Override
    public void call(JsonArray arguments) {
        Notification.show("Received call");
    }
});

Link link = new Link("Send Message", new ExternalResource(
    "javascript:com.example.foo.myfunc()");
```

Parameters passed to the JavaScript method on the client-side are provided in a **JSONArray** passed to the `call()` method. The parameter values can be acquired with the `get()` method by the index of the parameter, or any of the type-casting getters. The getter must match the type of the passed parameter, or an exception is thrown.

```
JavaScript.getCurrent().addFunction("com.example.foo.myfunc",
                                   new JavaScriptFunction() {
    @Override
    public void call(JsonArray arguments) {
        try {
            String message = arguments.getString(0);
            int value = arguments.getInt(1);
            Notification.show("Message: " + message +
                             ", value: " + value);
        } catch (Exception e) {
            Notification.show("Error: " + e.getMessage());
        }
    }
});
```

```
});
```

```
Link link = new Link("Send Message", new ExternalResource(  
    "javascript:com.example.foo.myfunc(prompt('Message'), 42)"));
```

The function callback mechanism is the same as the RPC mechanism used with JavaScript component integration, as described in Section 17.13.4, “RPC from JavaScript to Server-Side”.

12.15. Accessing Session-Global Data

This section is mostly up-to-date with Vaadin 7, but has some information which still needs to be updated.

Applications typically need to access some objects from practically all user interface code, such as a user object, a business data model, or a database connection. This data is typically initialized and managed in the UI class of the application, or in the session or servlet.

For example, you could hold it in the UI class as follows:

```
class MyUI extends UI {  
    UserData userData;  
  
    public void init() {  
        userData = new UserData();  
    }  
  
    public UserData getUserData() {  
        return userData;  
    }  
}
```

Vaadin offers two ways to access the UI object: with `getUI()` method from any component and the global `UI.getCurrent()` method.

The `getUI()` works as follows:

```
data = ((MyUI) component.getUI()).getUserData();
```

This does not, however work in many cases, because it requires that the components are attached to the UI. That is not the case most of the time when the UI is still being built, such as in constructors.

```
class MyComponent extends CustomComponent {  
    public MyComponent() {  
        // This fails with NullPointerException  
        Label label = new Label("Country: " +  
            getUI().getLocale().getCountry());  
  
        setCompositionRoot(label);  
    }  
}
```

The global access methods for the currently served servlet, session, and UI allow an easy way to access the data:

```
data = ((MyUI) UI.getCurrent()).getUserData();
```


12.15.1. The Problem

The basic problem in accessing session-global data is that the `getUI()` method works only after the component has been attached to the application. Before that, it returns `null`. This is the case in constructors of components, such as a **CustomComponent**:

Using a static variable or a singleton implemented with such to give a global access to user session data is not possible, because static variables are global in the entire web application, not just the user session. This can be handy for communicating data between the concurrent sessions, but creates a problem within a session.

The data would be shared by all users and be reinitialized every time a new user opens the application.

12.15.2. Overview of Solutions

To get the application object or any other global data, you have the following solutions:

- Pass a reference to the global data as a parameter
- Initialize components in `attach()` method
- Initialize components in the `enter()` method of the navigation view (if using navigation)
- Store a reference to global data using the *ThreadLocal Pattern*

Each solution is described in the following sections.

12.15.3. Passing References Around

You can pass references to objects as parameters. This is the normal way in object-oriented programming.

```
class MyApplication extends Application {
    UserData userData;

    public void init() {
        Window mainWindow = new Window("My Window");
        setMainWindow(mainWindow);

        userData = new UserData();

        mainWindow.addComponent(new MyComponent(this));
    }

    public UserData getUserData() {
        return userData;
    }
}

class MyComponent extends CustomComponent {
    public MyComponent(MyApplication app) {
        Label label = new Label("Name: " +
            app.getUserData().getName());

        setCompositionRoot(label);
    }
}
```

```
    }  
}
```

If you need the reference in other methods, you either have to pass it again as a parameter or store it in a member variable.

The problem with this solution is that practically all constructors in the application need to get a reference to the application object, and passing it further around in the classes is another hard task.

12.15.4. Overriding `attach()`

The `attach()` method is called when the component is attached to the UI through containment hierarchy. The `getUI()` method always works.

```
class MyComponent extends CustomComponent {  
    public MyComponent() {  
        // Must set a dummy root in constructor  
        setCompositionRoot(new Label(""));  
    }  
  
    @Override  
    public void attach() {  
        Label label = new Label("Name: " +  
            (MyUI) component.getUI()  
                .getUserData().getName());  
  
        setCompositionRoot(label);  
    }  
}
```

While this solution works, it is slightly messy. You may need to do some initialization in the constructor, but any construction requiring the global data must be done in the `attach()` method. Especially, **CustomComponent** requires that the `setCompositionRoot()` method is called in the constructor. If you can't create the actual composition root component in the constructor, you need to use a temporary dummy root, as is done in the example above.

Using `getUI()` also needs casting if you want to use methods defined in your UI class.

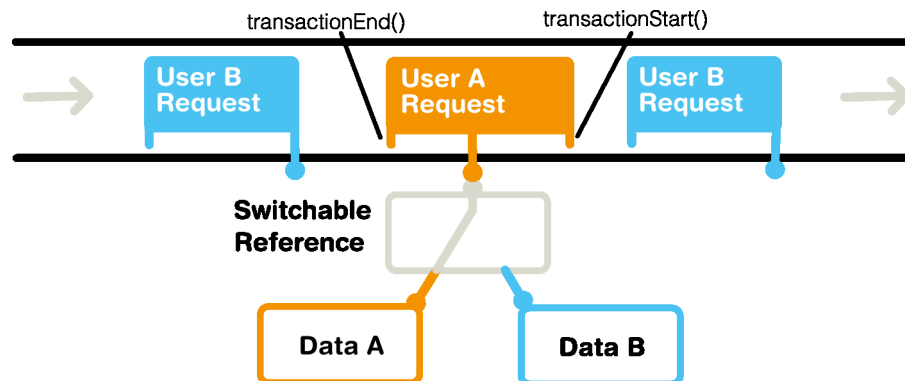
12.15.5. ThreadLocal Pattern

Vaadin uses the ThreadLocal pattern for allowing global access to the **UI**, and **Page** objects of the currently processed server request with a static `getCurrent()` method in all the respective classes. This section explains why the pattern is used in Vaadin and how it works. You may also need to reimplement the pattern for some purpose.

The ThreadLocal pattern gives a solution to the global access problem by solving two sub-problems of static variables.

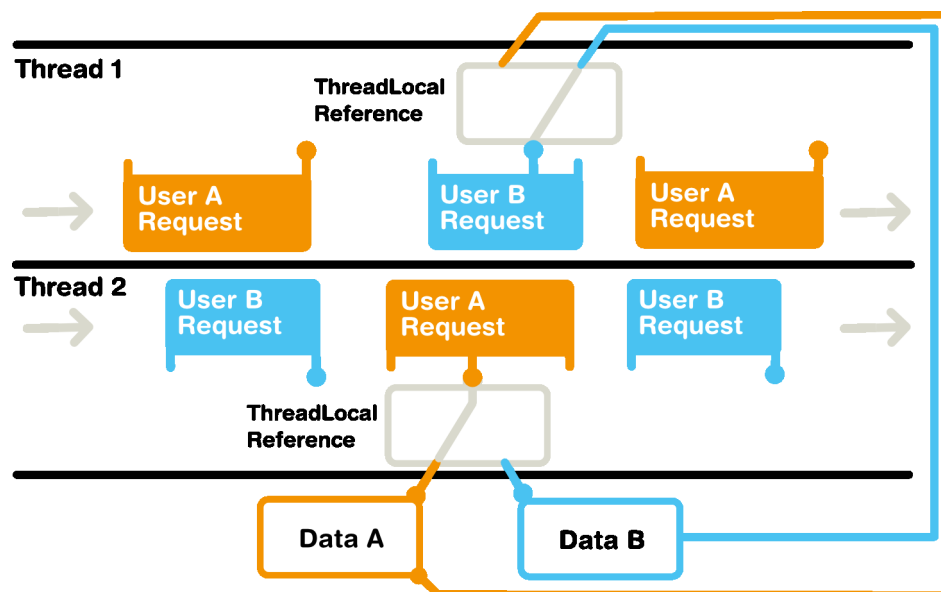
As the first problem, assume that the servlet container processes requests for many users (sessions) sequentially. If a static variable is set in a request belonging one user, it could be read or re-set by the next incoming request belonging to another user. This can be solved by setting the global reference at the beginning of each HTTP request to point to data of the current user, as illustrated in Figure Figure 12.13, "Switching a static (or ThreadLocal) reference during sequential processing of requests".

Figure 12.13. Switching a static (or ThreadLocal) reference during sequential processing of requests



The second problem is that servlet containers typically do thread pooling with multiple worker threads that process requests. Therefore, setting a static reference would change it in all threads running concurrently, possibly just when another thread is processing a request for another user. The solution is to store the reference in a thread-local variable instead of a static. You can do so by using the **ThreadLocal** class in Java for the switch reference.

Figure 12.14. Switching ThreadLocal references during concurrent processing of requests



12.16. Server Push

When you need to update a UI from another UI, possibly of another user, or from a background thread running in the server, you usually want to have the update show immediately, not when the browser happens to make the next server request. For this purpose, you can use *server push* that sends the data to the browser immediately. Push is based on a client-server connection, usually a WebSocket connection, that the client establishes and the server can then use to send updates to the client.

The server-client communication is done by default with a WebSocket connection if the browser and the server support it. If not, Vaadin will fall back to a method supported by the browser. Vaadin Push uses a custom build of the Atmosphere framework for client-server communication.

12.16.1. Installing the Push Support

The server push support in Vaadin requires the separate Vaadin Push library. It is included in the installation package as `vaadin-push.jar`.

Retrieving with Ivy

With Ivy, you can get it with the following declaration in the `ivy.xml`:

```
<dependency org="com.vaadin" name="vaadin-push"
            rev="&vaadin.version;" conf="default->default"/>
```

In some servers, you may need to exclude a `slf4j` dependency as follows:

```
<dependency org="com.vaadin" name="vaadin-push"
            rev="&vaadin.version;" conf="default->default">
  <exclude org="org.slf4j" name="slf4j-api"/>
</dependency>
```

Pay note that the Atmosphere library is a bundle, so if you retrieve the libraries with Ant, for example, you need to retrieve `type="jar,bundle"`.

Retrieving with Maven

In Maven, you can get the push library with the following dependency in the POM:

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-push</artifactId>
  <version>${vaadin.version}</version>
</dependency>
```

12.16.2. Enabling Push for a UI

To enable server push, you need to define the push mode either in the deployment descriptor or with the **@Push** annotation for the UI.

Push Modes and Transports

You can use server push in two modes: `automatic` and `manual`. The automatic mode pushes changes to the browser automatically after `access()` finishes. With the manual mode, you can do the push explicitly with `push()`, which allows more flexibility.

Sever push can use several transports - WebSockets, long polling, or combined WebSockets+XHR. Default is WebSockets.

The @Push annotation

You can enable server push for a UI with the **@Push** annotation as follows. It defaults to automatic mode (`PushMode.AUTOMATIC`).

```
@Push
public class PushyUI extends UI {
```

To enable manual mode, you need to give the `PushMode.MANUAL` parameter as follows:

```
@Push(PushMode.MANUAL)
public class PushyUI extends UI {
```

To use the long polling transport, you need to set the `Transport.LONG_POLLING` parameter as follows:

```
@Push(transport=Transport.LONG_POLLING)
public class PushyUI extends UI {
```

Servlet Configuration

You can enable the server push and define the push mode also in the servlet configuration with the `pushmode` parameter for the servlet in the `web.xml` deployment descriptor. If you use a Servlet 3.0 compatible server, you also want to enable asynchronous processing with the `async-supported` parameter. Note the use of Servlet 3.0 schema in the deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  id="WebApp_ID" version="3.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>Pushy UI</servlet-name>
    <servlet-class>
      com.vaadin.server.VaadinServlet</servlet-class>

    <init-param>
      <param-name>UI</param-name>
      <param-value>com.example.my.PushyUI</param-value>
    </init-param>

    <!-- Enable server push -->
    <init-param>
      <param-name>pushmode</param-name>
      <param-value>automatic</param-value>
    </init-param>
    <async-supported>true</async-supported>
  </servlet>
</web-app>
```

12.16.3. Accessing UI from Another Thread

Making changes to a **UI** object from another thread and pushing them to the browser requires locking the user session when accessing the UI. Otherwise, the UI update done from another thread could conflict with a regular event-driven update and cause either data corruption or deadlocks. Because of this, you may only access an UI using the `access()` method, which locks the session to prevent conflicts. It takes a `Runnable` which it executes as its parameter.

For example:

```
ui.access(new Runnable() {  
    @Override  
    public void run() {  
        series.add(new DataSeriesItem(x, y));  
    }  
});
```

In Java 8, where a parameterless lambda expression creates a runnable, you could simply write:

```
ui.access(() ->  
    series.add(new DataSeriesItem(x, y)));
```

If the push mode is manual, you need to push the pending UI changes to the browser explicitly with the `push()` method.

```
ui.access(new Runnable() {  
    @Override  
    public void run() {  
        series.add(new DataSeriesItem(x, y));  
        ui.push();  
    }  
});
```

Below is a complete example of a case where we make UI changes from another thread.

```
public class PushyUI extends UI {  
    Chart chart = new Chart(ChartType.AREASPLINE);  
    DataSeries series = new DataSeries();  
  
    @Override  
    protected void init(VaadinRequest request) {  
        chart.setSizeFull();  
        setContent(chart);  
  
        // Prepare the data display  
        Configuration conf = chart.getConfiguration();  
        conf.setTitle("Hot New Data");  
        conf.setSeries(series);  
  
        // Start the data feed thread  
        new FeederThread().start();  
    }  
  
    class FeederThread extends Thread {  
        int count = 0;  
  
        @Override  
        public void run() {  
            try {  
                // Update the data for a while  
                while (count < 100) {  
                    Thread.sleep(1000);  
  
                    access(new Runnable() {  
                        @Override  
                        public void run() {  
                            double y = Math.random();  
                            series.add(  
                                new DataSeriesItem(count++, y),  

```

```
        true, count > 10);
    }
    });
}

// Inform that we have stopped running
access(new Runnable() {
    @Override
    public void run() {
        setContent(new Label("Done!"));
    }
});
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}
```

When sharing data between UIs or user sessions, you need to consider the message-passing mechanism more carefully, as explained next.

12.16.4. Broadcasting to Other Users

Broadcasting messages to be pushed to UIs in other user sessions requires having some sort of message-passing mechanism that sends the messages to all UIs that register as recipients. As processing server requests for different UIs is done concurrently in different threads of the application server, locking the threads properly is very important to avoid deadlock situations.

The Broadcaster

The standard pattern for sending messages to other users is to use a *broadcaster* singleton that registers the UIs and broadcasts messages to them safely. To avoid deadlocks, it is recommended that the messages should be sent through a message queue in a separate thread. Using a Java **ExecutorService** running in a single thread is usually the easiest and safest way.

```
public class Broadcaster implements Serializable {
    static ExecutorService executorService =
        Executors.newSingleThreadExecutor();

    public interface BroadcastListener {
        void receiveBroadcast(String message);
    }

    private static LinkedList<BroadcastListener> listeners =
        new LinkedList<BroadcastListener>();

    public static synchronized void register(
        BroadcastListener listener) {
        listeners.add(listener);
    }

    public static synchronized void unregister(
        BroadcastListener listener) {
        listeners.remove(listener);
    }

    public static synchronized void broadcast(
```

```
        final String message) {
    for (final BroadcastListener listener: listeners)
        executorService.execute(new Runnable() {
            @Override
            public void run() {
                listener.receiveBroadcast(message);
            }
        });
    }
}
```

In Java 8, you could use lambda expressions for the listeners instead of the interface, and a parameterless expression to create the runnable:

```
for (final Consumer<String> listener: listeners)
    executorService.execute(() ->
        listener.accept(message));
```

Receiving Broadcasts

The receivers need to implement the receiver interface and register to the broadcaster to receive the broadcasts. A listener should be unregistered when the UI expires. When updating the UI in a receiver, it should be done safely as described earlier, by executing the update through the `access()` method of the UI.

```
@Push
public class PushAroundUI extends UI
    implements Broadcaster.BroadcastListener {

    VerticalLayout messages = new VerticalLayout();

    @Override
    protected void init(VaadinRequest request) {
        ... build the UI ...

        // Register to receive broadcasts
        Broadcaster.register(this);
    }

    // Must also unregister when the UI expires
    @Override
    public void detach() {
        Broadcaster.unregister(this);
        super.detach();
    }

    @Override
    public void receiveBroadcast(final String message) {
        // Must lock the session to execute logic safely
        access(new Runnable() {
            @Override
            public void run() {
                // Show it somehow
                messages.addComponent(new Label(message));
            }
        });
    }
}
```


Sending Broadcasts

To send broadcasts with a broadcaster singleton, such as the one described above, you would only need to call the `broadcast()` method as follows.

```
final TextField input = new TextField();
sendBar.addComponent(input);

Button send = new Button("Send");
send.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        // Broadcast the message
        Broadcaster.broadcast(input.getValue());

        input.setValue("");
    }
});
```

12.17. Vaadin CDI Add-on

Vaadin CDI add-on makes it easier to use contexts and dependency injection (CDI) in Vaadin applications. CDI is a Java EE feature especially targeted for web applications, which have well-defined contextual scopes, such as sessions, views, requests, and so forth. The lifecycle of objects, such as beans, can be managed by binding their lifecycles to such contexts. Vaadin CDI enables these features with two additional kinds of Vaadin-specific contextual scopes: UIs and navigation views.

To learn more about Vaadin CDI, the link:[Vaadin CDI Tutorial] gives a hands-on introduction. The source code of the CDI Tutorial demo is available for browsing or cloning at <https://github.com/vaadin/cdi-tutorial>.

12.17.1. CDI Overview

Contexts and dependency injection, defined in the JSR-299 standard, is a Java EE feature that, through a set of services, helps in improving application architecture by decoupling the management of service object lifecycles from client objects using them. The lifecycle of objects stored in a CDI container is defined by a context. The managed objects or beans are accessed using dependency injection.

CDI builds on the Java concept of beans, but with somewhat different definition and requirements.

Regarding general CDI topics, such as use of qualifiers, interceptors, decorators, event notifications, and other CDI features, we refer you to CDI documentation.

Dependency Injection

Dependency injection is a way to pass dependencies (service objects) to dependent objects (clients) by injecting them in member variables or initializer parameters, instead of managing the lifecycle in the clients or passing them explicitly as parameters. In CDI, injection of a service object to a client is specified by the **@Inject** annotation.

For example, if we have a UI view that depends on user data, we could inject the data in the client as follows:

```
public class MainView extends CustomComponent implements View {
    @Inject
    User user;

    ...
    @Override
    public void enter(ViewChangeEvent event) {
        greeting.setValue("Hello, " + user.getName());
    }
}
```

In addition to injecting managed beans with the annotation, you can query for them from the bean manager.

Contexts and Scopes

Contexts in CDI are services that manage the lifecycle of objects and handle their injection. Generally speaking, a context is a situation in which an instance is used with a unique identity. Such objects are essentially "singletons" in the context. While conventional singletons are application-wide, objects managed by a CDI container can be "singletons" in a more narrow *scope*: a user session, a particular UI instance associated with the session, a view within the UI, or even just a single request. Such a context defines the lifecycle of the object: its creation, use, and finally its destruction.

As a very typical example in a web application, you would have a user data object associated with a user session.

```
@SessionScoped
public class User {
    private String name;

    public void setName(String name) {this.name = name;}
    public String getName() {return name;}
}
```

Now, when you need to refer to the user, you can use CDI injection to inject the session-scoped "singleton" to a member variable or a constructor parameter.

```
public class MainView extends CustomComponent implements View {
    @Inject
    User user;

    ...

    @Override
    public void enter(ViewChangeEvent event) {
        greeting.setValue("Hello, " + user.getName());
    }
}
```

12.17.2. Installing Vaadin CDI Add-on

Vaadin CDI requires a Java EE 7 compatible servlet container, such as Glassfish or Apache TomEE Web Profile, as mentioned for the reference toolchain in Section 2.2, "A Reference Toolchain".

To install the Vaadin CDI add-on, either define it as an Ivy or Maven dependency or download it from the Vaadin Directory add-on page at <<,vaadin.com/directory#addon/vaadin-cdi>>. See Chapter 18, *Using Vaadin Add-ons* for general instructions for installing and using Vaadin add-ons.

The Ivy dependency is as follows:

```
<dependency org="com.vaadin" name="vaadin-cdi"
            rev="latest.release"/>
```

The Maven dependency is as follows:

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-cdi</artifactId>
  <version>[replaceable]LATEST</version>
</dependency>
<dependency>
  <groupId>javax.enterprise</groupId>
  <artifactId>cdi-api</artifactId>
  <version>[replaceable]1.2</version>
</dependency>
```

12.17.3. Preparing Application for CDI

A Vaadin application that uses CDI must have a file named `beans.xml` in the `WEB-INF` directory. The file can be completely empty (it has content only in certain limited situations), but it must be present.

The application should not have a servlet extending **VaadinServlet**, as Vaadin servlet has its own **VaadinCDIServlet** that is deployed automatically. If you need multiple servlets or need to customize the Vaadin CDI servlet, see Section 12.17.6, “Deploying CDI UIs and Servlets”.

12.17.4. Injecting a UI with @CDIUI

Vaadin CDI offers an easier way to instantiate UIs and to define the URL mapping for them than the usual ways described in Section 5.9, “Deploying an Application”. To define a UI class that should be instantiated for a given URL, you simply need to annotate the class with **@CDIUI**. It takes an optional URL path as parameter.

```
@CDIUI("myniceui")
@Theme("valo")
public class MyNiceUI extends UI {
    ...
}
```

Giving empty UI path maps the UI to the root of the application context.

```
@CDIUI("")
```

If the optional UI path is not given, the path is determined automatically from the class name by removing a possible “-UI” suffix in the class name, making it lower-case, and for capitalized letters, a hyphen is added. For example, a UI with class name **MyNiceUI** would have path `my-nice`. The URL consists of the server address, application context, and the UI path. For example, when running a Vaadin application in a development workstation, you would have URL such as `http://localhost:8080/myproject/my-nice`.

UI path mappings are reported in the server log during deployment.

See Section 12.17.6, “Deploying CDI UIs and Servlets” for how to handle servlet URL mapping of CDI UIs when working with multiple servlets in the same web application.

12.17.5. Scopes

As in programming languages, where a variable name refers to a unique object within the scope of the variable, a CDI scope is a context in which an object has unique identity. In CDI, objects to be injected are identified by their type and any qualifiers they may have. The scope can be defined as an annotation to the service class as follows:

```
@SessionScoped
public class User {
    ...
}
```

CDI defines a number of scopes. Note that the standard CDI scopes are defined under the `javax.enterprise.context` package and Vaadin CDI scopes under `com.vaadin.cdi`, while JSF scopes are defined in `javax.faces.bean`.

UI Scope

UI-scoped beans are uniquely identified within a UI instance, that is, a browser window or tab.

Vaadin CDI provides two annotations for the UI scope, differing in how they enable proxies, as explained later.

@UIScoped(`com.vaadin.cdi`)

Injection with this annotation will create a direct reference to the bean rather than a proxy. There are some limitations when not using proxies. Circular references (injecting A to B and B to A) will not work, and neither do CDI interceptors and decorators.

@NormalUIScoped(`com.vaadin.cdi`): As **@UIScoped**, but injecting a managed bean having this annotation injects a proxy for the bean instead of a direct reference. This is the normal behaviour with CDI, as many CDI features utilize the proxy.

Defining a CDI view (annotated with **@CDIView** as described later) as **@UIScoped** makes the view retain the same instance when the user navigates away and back to the view.

View Scopes

The lifecycle of a view-scoped bean starts when the user navigates to a view referring to the object and ends when the user navigates out of the view (or when the UI is closed or expires).

Vaadin CDI provides two annotations for the view scope, differing in how they enable proxies, as explained later.

@ViewScoped(`com.vaadin.cdi`)

Injection with this annotation will create a direct reference to the bean rather than a proxy. There are some limitations when not using proxies. Circular references (injecting A to B and B to A) will not work, and neither do CDI interceptors and decorators.

@NormalViewScoped(`com.vaadin.cdi`)

As **@NormalScoped**, except that injecting with this annotation will create a proxy for the contextual instance rather than provide the contextual instance itself. See the explanation of proxies below.

Standard CDI Scopes

@ApplicationScoped

Application-scoped beans are shared by all servlets in the web application, and are essentially equal to singletons.//TODO This is just a guess - is it true? Note that referencing application-scoped beans is not thread-safe and access must be synchronized.

@SessionScoped::

+ The lifecycle and visibility of session-scoped beans is bound to a HTTP or user session, which in Vaadin applications is associated with the **VaadinSession** (see Section 5.8.3, “User Session”). This is a very typical scope to store user data, as is done in many examples in this section, or database connections. The lifecycle of session-scoped beans starts when a user opens the page for a UI in the browser, and ends when the session expires after the last UI in the session is closed.

Proxies vs Direct References

CDI uses proxy objects to enable many of the CDI features, by hooking into message-passing from client to service beans. Under the hood, a proxy is an instance of an automatically generated class that extends the proxied bean type, so communicating through a proxy occurs transparently, as it has the same polymorphic type as the actual bean. Whether proxying is enabled or not is defined in the scope: CDI scopes are either *normal scopes*, which can be proxied, or *pseudo-scopes*, which use direct references to injected beans.

The proxying mechanism creates some requirements for injecting objects in normal scope:

- The objects may not be primitive types or arrays
- The bean class must not be final
- The bean class must not have final methods

Beans annotated with **@UIScoped** or **@ViewScoped** use a pseudoscope, and are therefore injected with direct references to the bean instances, while **@NormalUIScoped** and **@NormalViewScoped** beans will use a proxy for communicating with the beans.

When using proxies, be aware that it is not guaranteed that the `hashCode()` or `equals()` will match when comparing a proxy to its underlying instance. It is imperative to be aware of this when, for example, adding proxies to a `Collection`.

You should avoid using normal scopes with Vaadin components, as proxies may not work correctly within the Vaadin framework. If Vaadin CDI plugin detects such use, it displays a warning such as the following:

```
INFO: The following Vaadin components are injected
into normal scoped contexts:
    @NormalUIScoped      org.example.User
This approach uses proxy objects and has not been
extensively tested with the framework. Please report
any unexpected behavior. Switching to a pseudo-scoped
context may also resolve potential issues.
```

12.17.6. Deploying CDI UIs and Servlets

Vaadin CDI hooks into Vaadin framework by using a special **VaadinCDIServlet**. As described earlier, you do not need to map an URL path to a UI, as it is handled by Vaadin CDI. However, in the following, we go through some cases where you need to customize the servlet or use CDI with non-CDI servlets and UIs in a web application.

Defining Servlet Root with @URLMapping

CDI UIs are managed by a CDI servlet (**VaadinCDIServlet**), which is by default mapped to the root of the application context. For example, if the name of a CDI UI is "my-cdi" and application context is /myproject, the UI would by default have URL " /myproject/my-cdi". If you do not want to have the servlet mapped to context root, you can use the **@URLMapping** annotation to map all CDI UIs to a sub-path. The annotation must be given to only one CDI UI, usually the one with the default ("") path.

For example, if we have a root UI and another:

```
@CDIUI("") // At CDI servlet root
@URLMapping("mycdiuis") // Define CDI Servlet root
public class MyCDIRootUI extends UI {...}

@CDIUI("another")
public class AnotherUI extends UI {...}
```

These two UIs would have URLs /myproject/mycdiuis and /myproject/mycdiuis/another, respectively.

You can also map the CDI servlet to another URL in servlet definition in web.xml, as described the following.

Mixing With Other Servlets

The **VaadinCDIServlet** is normally used as the default servlet, but if you have other servlets in the application, such as for non-CDI UIs, you need to define the CDI servlet explicitly in the web.xml. You can map the servlet to any URL path, but perhaps typically, you define it as the default servlet as follows, and map the other servlets to other URL paths:

```
<web-app>
...

<servlet>
  <servlet-name>Default</servlet-name>
  <servlet-class>
    com.vaadin.cdi.internal.VaadinCDIServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Default</servlet-name>
  <url-pattern>[replaceable]/mycdiuis</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>Default</servlet-name>
  <url-pattern>/VAADIN</url-pattern>
```

```
</servlet-mapping>  
</web-app>
```

With such a setting, paths to CDI UIs would have base path `/myapp/mycdiuis`, to which the (optional) UI path would be appended. The `/VAADIN/*` only needs to be mapped to the servlet if there are no other Vaadin servlets.

Custom Servlets

When customizing the Vaadin servlet, as outlined in Section 5.8.2, “Vaadin Servlet, Portlet, and Service”, you simply need to extend **`com.vaadin.cdi.internal.VaadinCDIServlet`** instead of **`com.vaadin.servlet.VaadinServlet`**.

The custom servlet must not have **`@WebServlet`** annotation or **`@VaadinServletConfiguration`**, as you would normally with a Vaadin servlet, as described in Section 5.9, “Deploying an Application”.

12.18. Vaadin Spring Add-on

Vaadin Spring and Vaadin Spring Boot add-ons make it easier to use Spring in Vaadin applications. Vaadin Spring enables Spring dependency injection with custom UI and view providers, and provides three custom scopes: **`UIScope`**, **`ViewScope`**, and **`VaadinSessionScope`**.

API documentation for add-ons is available at:

- Vaadin Spring API at vaadin.com/api/vaadin-spring
- Vaadin Spring Boot API at vaadin.com/api/vaadin-spring-boot

To learn more about Vaadin Spring, the Vaadin Spring Tutorial gives a hands-on introduction. The source code of the Spring Tutorial demo is available for browsing or cloning at github.com/Vaadin/spring-tutorial.

12.18.1. Spring Overview

Spring Framework is a Java application framework that provides many useful services for building applications, such as authentication, authorization, data access, messaging, testing, and so forth. In the Spring core, one of the central features is dependency injection, which accomplishes inversion of control for dependency management in managed beans. Other Spring features rely on it extensively. As such, Spring offers capabilities similar to CDI, but with integration with other Spring services. Spring is well-suited for applications where Vaadin provides the UI layer and Spring is used for other aspects of the application logic.

Spring Boot is a Spring application development tool that allows creating Spring applications easily. *Vaadin Spring Boot* builds on Spring Boot to allow creating Vaadin Spring applications easily. It starts up a servlet, handles the configuration of the application context, registers a UI provider, and so forth.

Regarding general Spring topics, we recommend the following Spring documentation:

Dependency Injection

Dependency injection is a way to pass dependencies (service objects) to dependent objects (clients) by injecting them in member variables or initializer parameters, instead of managing

the lifecycle in the clients or passing them explicitly as parameters. In Spring, injection of a service object to a client is configured with the **@Autowired** annotation.

For a very typical example in a web application, you could have a user data object associated with a user session:

```
@SpringComponent
@VaadinSessionScope
public class User implements Serializable {
    private String name;

    public void setName(String name) {this.name = name;}
    public String getName() {return name;}
}
```

The **@SpringComponent** annotation allows for automatic detection of managed beans by Spring. (The annotation is exactly the same as the regular Spring **@Component**, but has been given an alias, because Vaadin has a `Component` interface, which can cause trouble.)

Now, if we have a UI view that depends on user data, we could inject the data in the client as follows:

```
public class MainView extends CustomComponent implements View {
    User user;
    Label greeting = new Label();

    @Autowired
    public MainView(User user) {
        this.user = user;
        ...
    }

    ...
    @Override
    public void enter(ViewChangeEvent event) {
        // Then you can use the injected data
        // for some purpose
        greeting.setValue("Hello, " + user.getName());
    }
}
```

Here, we injected the user object in the constructor. The user object would be created automatically when the view is created, with all such references referring to the same shared instance in the scope of the Vaadin user session.

Contexts and Scopes

Contexts in Spring are services that manage the lifecycle of objects and handle their injection. Generally speaking, a context is a situation in which an instance is used with a unique identity. Such objects are essentially "singletons" in the context. While conventional singletons are application-wide, objects managed by a Spring container can be "singletons" in a more narrow *scope*: a user session, a particular UI instance associated with the session, a view within the UI, or even just a single request. Such a context defines the lifecycle of the object: its creation, use, and finally its destruction.

Earlier, we introduced a user class defined as session-scoped:


```
@SpringComponent
@VaadinSessionScope
public class User {
```

Now, when you need to refer to the user, you can use Spring injection to inject the session-scoped "singleton" to a member variable or a constructor parameter; the former in the following:

```
public class MainView extends CustomComponent implements View {
    @Autowired
    User user;

    Label greeting = new Label();
    ...

    @Override
    public void enter(ViewChangeEvent event) {
        greeting.setValue("Hello, " + user.getName());
    }
}
```

12.18.2. Quick Start with Vaadin Spring Boot

The Vaadin Spring Boot is an add-on that allows for easily creating a project that uses Vaadin Spring. It is meant to be used together with Spring Boot, which enables general Spring functionalities in a web application.

You can use the Spring Initializr at start.spring.io website to generate a project, which you can then download as a package and import in your IDE. The generated project is a Spring application stub; you need to add at least Vaadin dependencies (`vaadin-spring-boot`, `vaadin-themes`, and `vaadin-client-compiled`) and a UI class to the generated project, as well as a theme, and so forth.

See the Vaadin Spring Tutorial for detailed instructions for using Spring Boot.

12.18.3. Installing Vaadin Spring Add-on

Vaadin Spring requires a Java EE 7 compatible servlet container, such as Glassfish or Apache TomEE Web Profile, as mentioned for the reference toolchain in Section 2.2, "A Reference Toolchain".

To install the Vaadin Spring and/or Vaadin Spring Boot add-ons, either define them as an Ivy or Maven dependency or download from the Vaadin Directory add-on page at <<vaadin.com/directory#addon/vaadin-spring>> or <<vaadin.com/directory#addon/vaadin-spring-boot>>. See Chapter 18, *Using Vaadin Add-ons* for general instructions for installing and using Vaadin add-ons.

The Ivy dependency is as follows:

```
<dependency org="com.vaadin" name="vaadin-spring"
    rev="latest.release"/>
```

The Maven dependency is as follows:

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-spring</artifactId>
```

```
<version>LATEST</version>
</dependency>
```

12.18.4. Preparing Application for Spring

A Vaadin application that uses Spring must have a file named `applicationContext.xml` in the `WEB-INF` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.1.xsd">

  <!-- Configuration object -->
  <bean class="com.example.myapp.MySpringUI.MyConfiguration" />

  <!-- Location for automatically scanned beans -->
  <context:component-scan
    base-package="com.example.myapp.domain" />
</beans>
```

The application should not have a servlet extending **VaadinServlet**, as Vaadin servlet has its own **VaadinSpringServlet** that is deployed automatically. If you need multiple servlets or need to customize the Vaadin Spring servlet, see Section 12.18.8, “Deploying Spring UIs and Servlets”.

You can configure managed beans explicitly in the file, or configure them to be scanned using the annotations, which is the preferred way described in this section.

12.18.5. Injecting a UI with @SpringUI

Vaadin Spring offers an easier way to instantiate UIs and to define the URL mapping for them than the usual ways described in Section 5.9, “Deploying an Application”. It is also needed for enabling Spring features in the UI. To define a UI class that should be instantiated for a given URL, you simply need to annotate the class with **@SpringUI**. It takes an optional path as parameter.

```
@SpringUI(path="/myniceui")
@Theme("valo")
public class MyNiceUI extends UI {
    ...
}
```

The path in the URL for accessing the UI consists of the context path of the application and the UI path, for example, `/myapp/myniceui`. Giving empty UI path maps the UI to the root of the application context, for example, `/myapp`.

```
@SpringUI
```

See Section 12.18.8, “Deploying Spring UIs and Servlets” for how to handle servlet URL mapping of Spring UIs when working with multiple servlets in the same web application.

12.18.6. Scopes

As in programming languages, where a variable name refers to a unique object within the scope of the variable, an object has unique identity within a scope in Spring. However, instead of identifying the objects by variable names, they are identified by their type (object class) and any qualifiers they may have.

The scope of an object can be defined with an annotation to the class as follows:

```
@VaadinSessionScope
public class User {
    ...
}
```

Defining a scope in Spring is normally done with the **@Scope** annotation. For example, `@Scope("prototype")` creates a new instance every time one is requested/auto-wired. Such standard scopes can be used with some limitations. For example, Spring session and request scopes do not work in background threads and with certain push transport modes.

Vaadin Spring provides three scopes useful in Vaadin applications: a session scope, a UI scope, a view scope, all defined in the `com.vaadin.spring.annotation` package.

@VaadinSessionScope

The session scope is the broadest of the custom scopes defined in Vaadin Spring. Objects in the Vaadin session scope are unique in a user session, and shared between all UIs open in the session. This is the most basic scope in Vaadin applications, useful for accessing data for the user associated with the session. It is also useful when updating UIs from a background thread, as in those cases the UI access is locked on the session and also data should be in that scope.

@UIScope

UI-scoped beans are uniquely identified within a UI instance, that is, a browser window or tab. The lifecycle of UI-scoped beans is bound between to the initialization and closing of a UI. Whenever you inject a bean, as long as you are within the same UI, you will get the same instance.

Annotating a Spring view (annotated with **@SpringView** as described later) also as **@UIScoped** makes the view retain the same instance when the user navigates away and back to the view.

@ViewScope

The annotation enables the view scope in a bean. The lifecycle of such a bean starts when the user navigates to a view referring to the object and ends when the user navigates out of the view (or when the UI is closed or expires).

Views themselves are by default view-scoped, so a new instance is created every time the user navigates to the view.

12.18.7. Access Control

Access control for views can be implemented by registering beans implementing `ViewAccessControl` or `ViewInstanceAccessControl`, which can restrict access to the view either before or after a view instance is created.

Integration with authorization solutions, such as Spring Security, is provided by additional unofficial add-ons on top of Vaadin Spring.

Access Denied View

By default, the view provider acts as if a denied view didn't exist. You can set up an "Access Denied" view that is shown if the access is denied with `setAccessDeniedView()` in **SpringViewProvider**.

```
@Autowired
SpringViewProvider viewProvider;

@Override
protected void init(VaadinRequest request) {
    Navigator navigator = new Navigator(this, this);
    navigator.addProvider(viewProvider);

    // Set up access denied view
    viewProvider.setAccessDeniedViewClass(
        MyAccessDeniedView.class);
}
```

12.18.8. Deploying Spring UIs and Servlets

Vaadin Spring hooks into Vaadin framework by using a special **VaadinSpringServlet**. As described earlier, you do not need to map an URL path to a UI, as it is handled by Vaadin Spring. However, in the following, we go through some cases where you need to customize the servlet or use Spring with non-Spring servlets and UIs in a web application.

Custom Servlets

When customizing the Vaadin servlet, as outlined in Section 5.8.2, "Vaadin Servlet, Portlet, and Service", you simply need to extend **com.vaadin.spring.internal.VaadinSpringServlet** instead of **com.vaadin.servlet.VaadinServlet**.

```
@WebServlet(value = "/*", asyncSupported = true)
public class MySpringServlet extends SpringVaadinServlet {
}
```

The custom servlet must not have **@VaadinServletConfiguration**, as you would normally with a Vaadin servlet, as described in Section 5.9, "Deploying an Application".

Defining Servlet Root

Spring UIs are managed by a Spring servlet (**VaadinSpringServlet**), which is by default mapped to the root of the application context. For example, if the name of a Spring UI is "my-spring-ui" and application context is /myproject, the UI would by default have URL "/myproject/my-spring-ui". If you do not want to have the servlet mapped to context root, you can use a **@WebServlet** annotation for the servlet or a `web.xml` definition to map all Spring UIs to a sub-path.

For example, if we have a root UI and another:

```
@SpringUI(path="") // At Spring servlet root
public class MySpringRootUI extends UI {...}

@SpringUI("another")
public class AnotherUI extends UI {...}
```

Then define a path for the servlet by defining a custom servlet:

```
@WebServlet(value = "/myspringuis/*", asyncSupported = true)
public class MySpringServlet extends SpringVaadinServlet {
}
```

These two UIs would have URLs `/myproject/myspringuis` and `/myproject/myspringuis/another`, respectively.

You can also map the Spring servlet to another URL in servlet definition in `web.xml`, as described the following.

Mixing With Other Servlets

The **VaadinSpringServlet** is normally used as the default servlet, but if you have other servlets in the application, such as for non-Spring UIs, you need to define the Spring servlet explicitly in the `web.xml`. You can map the servlet to any URL path, but perhaps typically, you define it as the default servlet as follows, and map the other servlets to other URL paths:

```
<web-app>
...
  <servlet>
    <servlet-name>Default</servlet-name>
    <servlet-class>
      com.vaadin.spring.internal.VaadinSpringServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Default</servlet-name>
    <url-pattern>/myspringuis/*</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>Default</servlet-name>
    <url-pattern>/VAADIN/*</url-pattern>
  </servlet-mapping>
</web-app>
```

With such a setting, paths to Spring UIs would have base path `/myapp/myspringuis`, to which the (optional) UI path would be appended. The `/VAADIN/*` only needs to be mapped to the servlet if there are no other Vaadin servlets.

Chapter 13

Portal Integration

13.1. Overview	425
13.2. Creating a Generic Portlet in Eclipse	426
13.3. Developing Vaadin Portlets for Liferay	428
13.4. Portlet UI	434
13.5. Deploying to a Portal	436
13.6. Vaadin IPC for Liferay	440

13.1. Overview

Vaadin supports running UIs as portlets in a portal, as defined in the JSR-286 (Java Portlet API 2.0) standard. A portlet UI is defined just as a regular UI, but deploying to a portal is somewhat different from deployment of regular web applications, requiring special portlet descriptors, etc. Creating the portlet project with the Vaadin Plugin for Eclipse or a Maven archetype automatically generates the necessary descriptors.

In addition to providing user interface through the Vaadin UI, portlets can integrate with the portal to switch between portlet modes and process special portal requests, such as actions and events.

While providing generic support for all portals implementing the standard, Vaadin especially supports the Liferay portal and the needed portal-specific configuration in this chapter is given for Liferay. Vaadin also has a special Liferay IPC add-on to enable communication between portlets.

13.2. Creating a Generic Portlet in Eclipse

Here we describe the creation of a generic portlet project in Eclipse. You can use the Maven archetypes also in other IDEs or without an IDE.

For Liferay portlet development, you may instead want to use the Maven archetype or Liferay IDE to create the project, as described in Section 13.3, “Developing Vaadin Portlets for Liferay”.

13.2.1. Creating a Project with Vaadin Plugin

The Vaadin Plugin for Eclipse has a wizard for easy creation of generic portlet projects. It creates a UI class and all the necessary descriptor files.

Creating a portlet project is almost identical to the creation of a regular Vaadin servlet application project. For a full treatment of the New Project Wizard and the possible options, please see Section 3.4.1, “Creating a Maven Project”.

1. Start creating a new project by selecting from the menu **File** → **New** → **Project...+**
1. In the **New Project** window that opens, select **Web** → **Vaadin 7 Project** and click **Next**.
2. In the **Vaadin Project** step, you need to set the basic web project settings. You need to give at least the project name, the runtime, select **Generic Portlet** for the **Deployment configuration**; the default values should be good for the other settings.

You can click **Finish** here to use the defaults for the rest of the settings, or click **Next**.
3. The settings in the **Web Module** step define the basic servlet-related settings and the structure of the web application project. All the settings are pre-filled, and you should normally accept them as they are and click **Next**.
4. The **Vaadin project** step page has various Vaadin-specific application settings. These are largely the same as for regular applications. Setting them here is easiest - later some of the changes require changes in several different files. The **Create portlet template** option should be automatically selected. You can give another portlet title of you want. You can change most of the settings afterward.

New Vaadin 7 Project

Vaadin project
Configure Vaadin specific project details

Application

☒ Create project template

Application name: Myportlet Application

Base package name: com.example.myporlet

Application class name: MyportletUI

Theme name: myportlet

Portlet

Portlet version: Portlet 2.0

Portlet title: myportlet

Vaadin Version

Vaadin version: 7.1.0

? < Back Next > Finish Cancel

Create project template

Creates a UI class and all the needed portlet deployment descriptors.

Application name

The application name is used in the title of the browser window, which is usually invisible in portlets, and as an identifier, either as is or with a suffix, in various deployment descriptors.

Base package name

Java package for the UI class.

Application class name

Name of the UI class. The default is derived from the project name.

Theme name

Name of the custom portlet theme to use.

Portlet version

Same as in the project settings.

Portlet title

The portlet title, defined in `portlet.xml`, can be used as the display name of the portlet (at least in Liferay). The default value is the project name. The title is also used as a short description in `liferay-plugin-package.properties`.

Vaadin version

Same as in the project settings.

Finally, click **Finish** to create the project.

5. Eclipse may ask you to switch to J2EE perspective. A Dynamic Web Project uses an external web server and the J2EE perspective provides tools to control the server and manage application deployment. Click **Yes**.

13.3. Developing Vaadin Portlets for Liferay

A Vaadin portlet requires resources such as the server-side Vaadin libraries, a theme, and a widget set. You have two basic ways to deploy these: either globally in Liferay, so that the resources are shared between all Vaadin portlets, or as self-contained WARs, where each portlet carries their own resources.

The self-contained way is easier and more flexible to start with, as the different portlets may have different versions of the resources. Currently, the latest Maven archetypes support the self-contained portlets, while with portlets created with the Vaadin Plugin for Eclipse only support globally deployed resources.

Using shared resources is more efficient when you have multiple Vaadin portlets on the same page, as they can share the common resources. However, they must use exactly same Vaadin version. This is recommended for production environments, where you can even serve the theme and widget set from a front-end server. You can install the shared resources as described in Section 13.3.5, “Installing Vaadin Resources”.

At the time of writing, the latest Liferay release 6.2 is bundled with a version of Vaadin release 6. If you want to use Vaadin 7 portlets with shared resources, you first need to remove the old ones as described in Section 13.3.4, “Removing the Bundled Installation”.

13.3.1. Defining Liferay Profile for Maven

When creating a Liferay portlet project with a Maven archetype or the Liferay IDE, you need to define a Liferay profile. With the Liferay IDE, you can create it when you create the project, as described in Section 13.3.3, “Creating a Portlet Project in Liferay IDE”, but for creating a project from the Maven archetype, you need to define in manually.

Defining Profile in `settings.xml`

Liferay profile can be defined either in the user or in the global `settings.xml` file for Maven. The global settings file is located in `${MAVEN_HOME}/conf/settings.xml` and the user settings file in `${USER_HOME}/.m2/settings.xml`. To create a user settings file, copy at least the relevant headers and root element from the global settings file.

```
...
<profile>
  <id>liferay</id>
  <properties>
    <liferayinstall>/opt/liferay-portal-6.2-ce-ga2
    </liferayinstall>
    <plugin.type>portlet</plugin.type>
    <liferay.version>6.2.1</liferay.version>
    <liferay.maven.plugin.version>6.2.1
    </liferay.maven.plugin.version>
    <liferay.auto.deploy.dir>${liferayinstall}/deploy
    </liferay.auto.deploy.dir>

    <!-- Application server version - here for Tomcat -->
    <liferay.tomcat.version>7.0.42</liferay.tomcat.version>
    <liferay.tomcat.dir>
      ${liferayinstall}/tomcat-${liferay.tomcat.version}
    </liferay.tomcat.dir>

    <liferay.app.server.deploy.dir>${liferay.tomcat.dir}/webapps
    </liferay.app.server.deploy.dir>
    <liferay.app.server.lib.global.dir>${liferay.tomcat.dir}/lib/ext
    </liferay.app.server.lib.global.dir>
    <liferay.app.server.portal.dir>${liferay.tomcat.dir}/webapps/ROOT
    </liferay.app.server.portal.dir>
  </properties>
</profile>
```

The parameters are as follows:

liferayinstall

Full (absolute) path to the Liferay installation directory.

liferay.version

Liferay version by the Maven version numbering scheme. The first two (major and minor) numbers are same as in the installation package. The third (maintenance) number starts from 0 with first GA (general availability) release.

liferay.maven.plugin.version

This is usually the same as the Liferay version.

liferay.auto.deploy.dir

The Liferay auto-deployment directory. It is by default `deploy` under the Liferay installation path.

liferay.tomcat.version(optional)

If using Tomcat, its version number.

liferay.tomcat.dir

Full (absolute) path to Tomcat installation directory. For Tomcat bundled with Liferay, this is under the Liferay installation directory.

liferay.app.server.deploy.dir

Directory where portlets are deployed in the application server used for Liferay. This depends on the server - for Tomcat it is the `webapps` directory under the Tomcat installation directory.

`liferay.app.server.lib.global.dir`

Library path where libraries globally accessible in the application server should be installed.

`liferay.app.server.portal.dir`

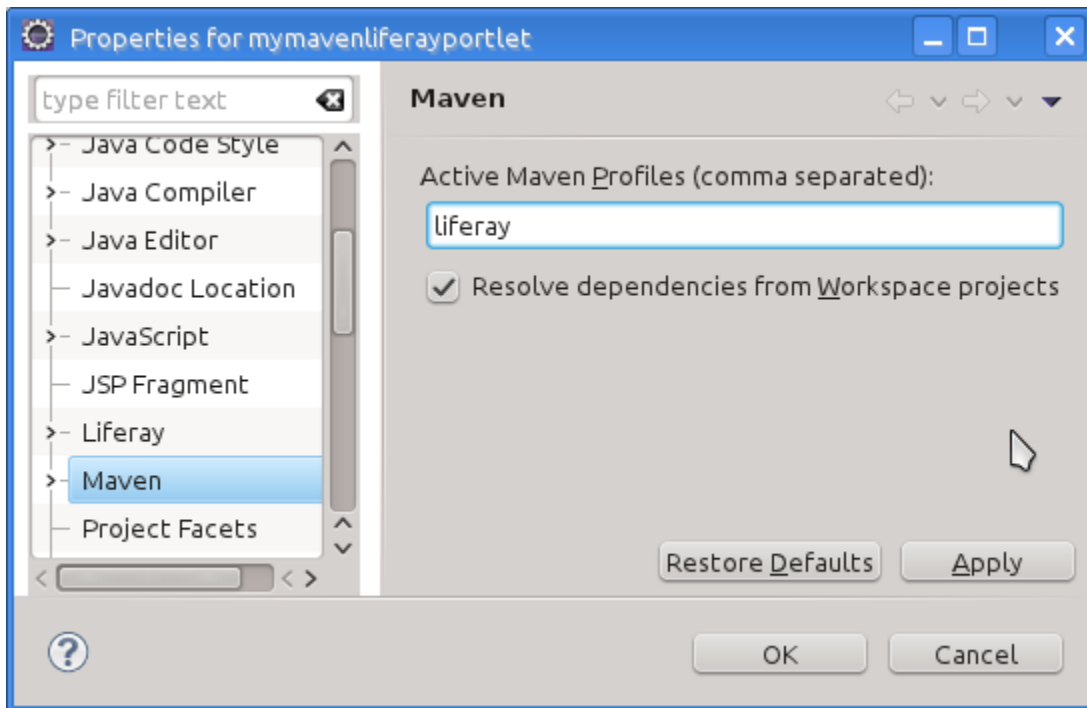
Deployment directory for static resources served by the application server, under the root path of the server.

If you modify the settings after the project is created, you need to touch the POM file in the project to have the settings reloaded.

Activating the Maven Profile

The Maven 2 Plugin for Eclipse (m2e) must know which Maven profiles you use in a project. This is configured in the Maven section of the project properties. In the **Active Maven Profiles** field, enter the profile ID defined in the `settings.xml` file, as illustrated in Figure 13.1, “Activating Maven Liferay Profile”.

Figure 13.1. Activating Maven Liferay Profile



13.3.2. Creating a Portlet Project with Maven

Creation of Vaadin a Maven project is described in Section 3.5, “Creating a Project with Maven”. For a Liferay project, you should use the `vaadin-archetype-liferay-portlet`.

Archetype Parameters

The archetype has a number of parameters. If you use Maven Plugin for Eclipse (m2e) to create the project, you get to enter the parameters after selecting the archetype, as shown in Figure 13.2, “Liferay Project Archetype Parameters”.

Minimally, you just need to enter the artifact ID. To activate the Maven profile created as described earlier in Section 13.3.1, “Defining Liferay Profile for Maven”, you need to specify the profile in the **Profiles** field under the **Advanced** section.

Figure 13.2. Liferay Project Archetype Parameters

New Maven Project

Specify Archetype parameters

Group Id:

Artifact Id:

Version:

Package:

Properties available from archetype:

Name	Value
vaadinVersion	7.2.0
uiClassName	MyPortletUI
theme	mytheme
portletTitle	My Vaadin portlet
portletShortTitle	Hello World

Advanced

☒ Resolve Workspace projects

Profiles:

Name template:

The other parameters are the following:

vaadinVersion
Vaadin release version for the Maven dependency.

uiClassName
Class name of the UI class stub to be created.

theme

Theme to use. You can use either a project theme, which must be compiled before deployment, or use the `liferay` theme.

portletTitle

Title shown in the portlet title bar.

portletShortTitle

Title shown in contexts where a shorter title is preferred.

portletKeywords

Keywords for finding the portlet in Liferay.

portletDescription

A description of the portlet.

portletName

Identifier for the portlet, used for identifying it in the configuration files.

portletDisplayName

Name of the portlet for contexts where it is displayed.

13.3.3. Creating a Portlet Project in Liferay IDE

Liferay IDE, which you install in Eclipse as plugins just like the Vaadin plugin, enables a development environment for Liferay portlets. Liferay IDE allows integrated deployment of portlets to Liferay, just like you would deploy servlets to a server in Eclipse. The project creation wizard supports creation of Vaadin portlets.

Loading widget sets, themes, and the Vaadin JAR from a portlet is possible as long as you have a single portlet, but causes a problem if you have multiple portlets. To solve this, Vaadin portlets need to use a globally installed widget set, theme, and Vaadin libraries.

Liferay 6.2, which is the latest Liferay version at the time of publication of this book, comes bundled with an older Vaadin 6 version. If you want to use Vaadin 7, you need to remove the bundled version and install the newer one manually as described in this chapter.

In these instructions, we assume that you use Liferay bundled with Apache Tomcat, although you can use almost any other application server with Liferay just as well. The Tomcat installation is included in the Liferay installation package, under the `tomcat-x.x.x` directory.

13.3.4. Removing the Bundled Installation

Before installing a new Vaadin version, you need to remove the version bundled with Liferay. You need to remove the Vaadin library JAR from the library directory of the portal and the `VAADIN` directory from under the root context. For example, with Liferay bundled with Tomcat, they are usually located as follows:

- `tomcat-x.x.x/webapps/ROOT/html/VAADIN`
- `tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/vaadin.jar`

13.3.5. Installing Vaadin Resources

To use common resources needed by multiple Vaadin portlets, you can install them globally as shared resources as described in the following.

If you are installing Vaadin in a Liferay version that comes bundled with an older version of Vaadin, you first need to remove the resources as described in Section 13.3.4, “Removing the Bundled Installation”.

In the following, we assume that you use only the built-in “liferay” theme in Vaadin and the default widget set.

1. Get the Vaadin installation package from the Vaadin download page
2. Extract the following Vaadin JARs from the installation package: `vaadin-server.jar` and `vaadin-shared.jar`, as well as the `vaadin-shared-deps.jar` and `jsoup.jar` dependencies from the `lib` folder
3. Rename the JAR files as they were listed above, without the version number
4. Put the libraries in `tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/`
5. Extract the `VAADIN` folders from `vaadin-server.jar`, `vaadin-themes.jar`, and `vaadin-client-compiled.jar` and copy their contents to `tomcat-x.x.x/webapps/ROOT/html/VAADIN`.

```
$ cd tomcat-x.x.x/webapps/ROOT/html
$ unzip path-to/vaadin-server-7.1.0.jar 'VAADIN/*'
$ unzip path-to/vaadin-themes-7.1.0.jar 'VAADIN/*'
$ unzip path-to/vaadin-client-compiled-7.1.0.jar 'VAADIN/*'
```

You need to define the widget set, the theme, and the JAR in the `portal-ext.properties` configuration file for Liferay, as described earlier. The file should normally be placed in the Liferay installation directory. See Liferay documentation for details on the configuration file.

Below is an example of a `portal-ext.properties` file:

```
# Path under which the VAADIN directory is located.
# (/html is the default so it is not needed.)
# vaadin.resources.path=/html

# Portal-wide widget set
vaadin.widgetset=com.vaadin.server.DefaultWidgetSet

# Theme to use
vaadin.theme=liferay
```

The allowed parameters are:

vaadin.resources.path
Specifies the resource root path under the portal context. This is `/html` by default. Its actual location depends on the portal and the application server; in Liferay with Tomcat it would be located at `webapps/ROOT/html` under the Tomcat installation directory.

vaadin.widgetset

The widget set class to use. Give the full path to the class name in the dot notation. If the parameter is not given, the default widget set is used.

vaadin.theme

Name of the theme to use. If the parameter is not given, the default theme is used, which is `reindeer` in Vaadin 6.

You will need to restart Liferay after creating or modifying the `portal-ext.properties` file.

13.4. Portlet UI

A portlet UI is just like in a regular Vaadin application, a class that extends **`com.vaadin.ui.UI`**.

```
@Theme("myportlet")
public class MyportletUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        final VerticalLayout layout = new VerticalLayout();
        layout.setMargin(true);
        setContent(layout);

        Button button = new Button("Click Me");
        button.addClickListener(new Button.ClickListener() {
            public void buttonClick(ClickEvent event) {
                layout.addComponent(
                    new Label("Thank you for clicking"));
            }
        });
        layout.addComponent(button);
    }
}
```

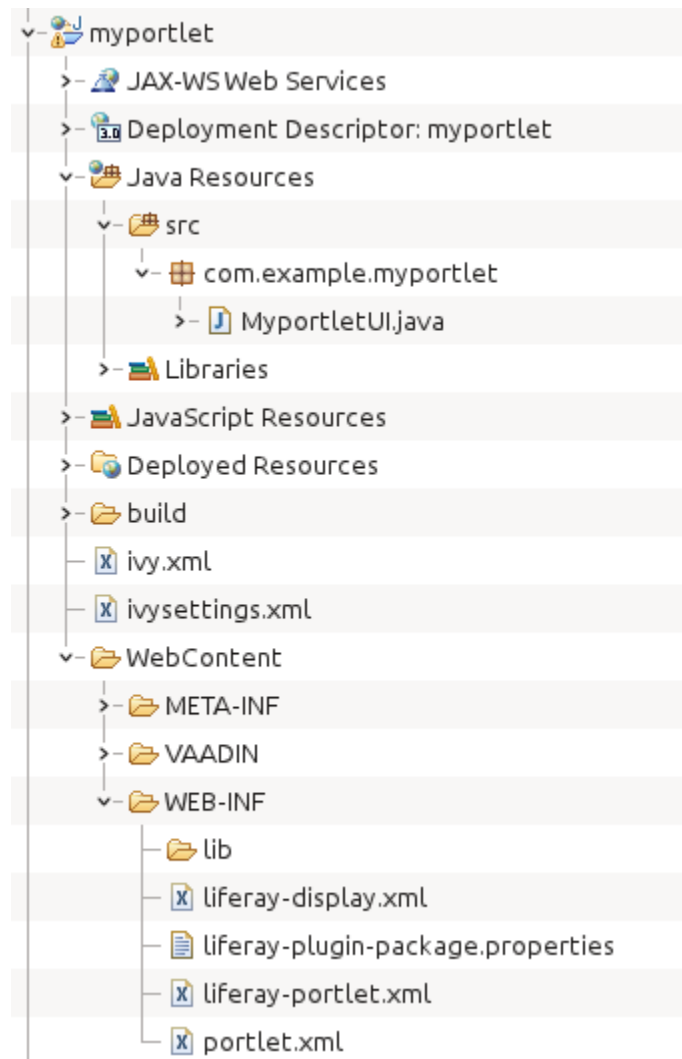
If you created the project as a Servlet 3.0 project, the generated UI stub includes a static servlet class annotated with **`@WebServlet`**, as described in Section 3.4.2, “Exploring the Project”.

```
@WebServlet(value = "/*", asyncSupported = true)
@VaadinServletConfiguration(productionMode = false,
                           ui = MyportletUI.class)
public static class Servlet extends VaadinServlet {
}
```

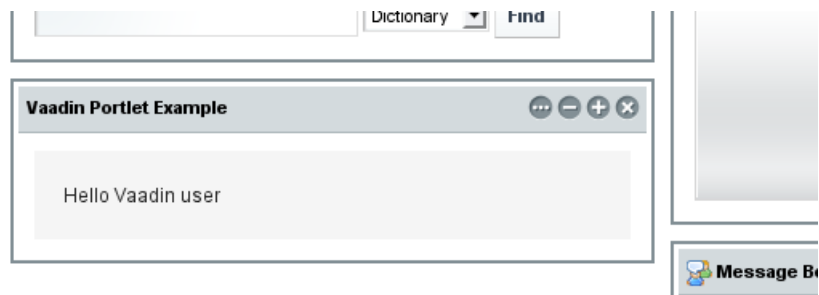
This enables running the portlet UI in a servlet container while developing it, which may be easier than deploying to a portal. For Servlet 2.4 projects, a `web.xml` is created.

The portlet theme is defined with the **`@Theme`** annotation as usual. The theme for the UI must match a theme installed in the portal. You can use any of the built-in themes in Vaadin. For Liferay theme compatibility, there is a special `liferay` theme. If you use a custom theme, you need to compile it to CSS with the theme compiler and install it in the portal under the `VAADIN/themes` context to be served statically.

In addition to the UI class, you need the portlet descriptor files, Vaadin libraries, and other files as described later. Figure 13.3, “Portlet Project Structure in Eclipse” shows the complete project structure under Eclipse.

Figure 13.3. Portlet Project Structure in Eclipse

Installed as a portlet in Liferay from the **Add Application** menu, the application will show as illustrated in Figure 13.4, “Hello World Portlet”.

Figure 13.4. Hello World Portlet

13.5. Deploying to a Portal

To deploy a portlet WAR in a portal, you need to provide a `portlet.xml` descriptor specified in the Java Portlet API 2.0 standard (JSR-286). In addition, you may need to include possible portal vendor specific deployment descriptors. The ones required by Liferay are described below.

Deploying a Vaadin UI as a portlet is essentially just as easy as deploying a regular application to an application server. You do not need to make any changes to the UI itself, but only the following:

- Application packaged as a WAR
 - `WEB-INF/portlet.xml` descriptor
 - `WEB-INF/liferay-portlet.xml` descriptor for Liferay
 - `WEB-INF/liferay-display.xml` descriptor for Liferay
 - `WEB-INF/liferay-plugin-package.properties` for Liferay
- Widget set installed to portal (optional)
- Themes installed to portal (optional)
- Vaadin libraries installed to portal (optional)
- Portal configuration settings (optional)

The Vaadin Plugin for Eclipse creates these files for you, when you create a portlet project as described in Section 13.2, “Creating a Generic Portlet in Eclipse”.

Installing the widget set and themes to the portal is required for running two or more Vaadin portlets simultaneously in a single portal page. As this situation occurs quite easily, we recommend installing them in any case. Instructions for Liferay are given in Section 13.3, “Developing Vaadin Portlets for Liferay” and the procedure is similar for other portals.

In addition to the Vaadin libraries, you will need to have the `portlet.jar` in your project classpath. However, notice that you must *not* put the `portlet.jar` in the same `WEB-INF/lib` directory as the Vaadin JAR or otherwise include it in the WAR to be deployed, because it would create a conflict with the internal portlet library of the portal. The conflict would cause errors such as `" ClassCastException: ...VaadinPortlet cannot be cast to javax.portlet.Portlet"`.

13.5.1. Portlet Deployment Descriptor

The portlet WAR must include a portlet descriptor located at `WEB-INF/portlet.xml`. A portlet definition includes the portlet name, mapping to a servlet, modes supported by the portlet, and other configuration. Below is an example of a simple portlet definition in `portlet.xml` descriptor.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<portlet-app
  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="2.0"
  xsi:schemaLocation=
```

```
"http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd">

<portlet>
  <portlet-name>Portlet Example portlet</portlet-name>
  <display-name>Vaadin Portlet Example</display-name>

  <!-- Map portlet to a servlet. -->
  <portlet-class>
    com.vaadin.server.VaadinPortlet
  </portlet-class>
  <init-param>
    <name>UI</name>

    <!-- The application class with package name. -->
    <value>com.example.myportlet.MyportletUI</value>
  </init-param>

  <!-- Supported portlet modes and content types. -->
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>view</portlet-mode>
    <portlet-mode>edit</portlet-mode>
    <portlet-mode>help</portlet-mode>
  </supports>

  <!-- Not always required but Liferay requires these. -->
  <portlet-info>
    <title>Vaadin Portlet Example</title>
    <short-title>Portlet Example</short-title>
  </portlet-info>
</portlet>
</portlet-app>
```

The mode definitions enable the corresponding portlet controls in the portal. The portlet controls allow changing the mode of the portlet, as described later.

13.5.2. Liferay Portlet Descriptor

Liferay requires a special `liferay-portlet.xml` descriptor file that defines Liferay-specific parameters. Especially, Vaadin portlets must be defined as "*instanceable*", but not "*ajaxable*".

Below is an example descriptor for the earlier portlet example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE liferay-portlet-app PUBLIC
  "-//Liferay//DTD Portlet Application 4.3.0//EN"
  "http://www.liferay.com/dtd/liferay-portlet-app_4_3_0.dtd">

<liferay-portlet-app>
  <portlet>
    <!-- Matches definition in portlet.xml. -->
    <!-- Note: Must not be the same as servlet name. -->
    <portlet-name>Portlet Example portlet</portlet-name>

    <instanceable>true</instanceable>
    <ajaxable>false</ajaxable>
  </portlet>
</liferay-portlet-app>
```

See Liferay documentation for further details on the `liferay-portlet.xml` deployment descriptor.

13.5.3. Liferay Display Descriptor

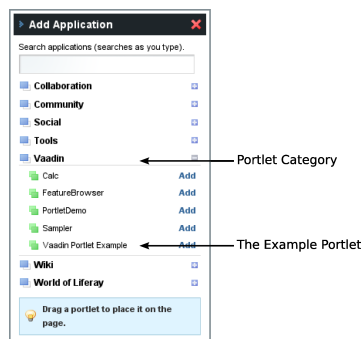
The `WEB-INF/liferay-display.xml` file defines the portlet category under which portlets are located in the **Add Application** window in Liferay. Without this definition, portlets will be organized under the "Undefined" category.

The following display configuration, which is included in the demo WAR, puts the Vaadin portlets under the "Vaadin" category, as shown in Figure 13.5, "Portlet Categories in Add Application Window".

```
<?xml version="1.0"?>
<!DOCTYPE display PUBLIC
    "-//Liferay//DTD Display 4.0.0//EN"
    "http://www.liferay.com/dtd/liferay-display_4_0_0.dtd">

<display>
    <category name="Vaadin">
        <portlet id="Portlet Example portlet" />
    </category>
</display>
```

Figure 13.5. Portlet Categories in Add Application Window



See Liferay documentation for further details on how to configure the categories in the `liferay-display.xml` deployment descriptor.

13.5.4. Liferay Plugin Package Properties

The `liferay-plugin-package.properties` file defines a number of settings for the portlet, most importantly the Vaadin JAR to be used.

```
name=Portlet Example portlet
short-description=myportlet
module-group-id=Vaadin
module-incremental-version=1
#change-log=
#page-uri=
#author=
license=Proprietary
portal-dependency-jars=\
    vaadin.jar
```

name

The plugin name must match the portlet name.

short-description

A short description of the plugin. This is by default the project name.

module-group-id

The application group, same as the category id defined in `liferay-display.xml`.

license

The plugin license type; "proprietary" by default.

portal-dependency-jars

The JAR libraries on which this portlet depends. This should have value `vaadin.jar`, unless you need to use a specific version. The JAR must be installed in the portal, for example, in Liferay bundled with Tomcat to `tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/vaadin.jar`.

13.5.5. Using a Single Widget Set

If you have just one Vaadin application that you ever need to run in your portal, you can just deploy the WAR as described above and that's it. However, if you have multiple applications, especially ones that use different custom widget sets, you run into problems, because a portal window can load only a single Vaadin widget set at a time. You can solve this problem by combining all the different widget sets in your different applications into a single widget set using inheritance or composition.

For example, if using the default widget set for portlets, you should have the following for all portlets so that they will all use the same widget set:

```
<portlet>
...
<!-- Use the portal default widget set for all portal demos. -->
<init-param>
  <name>widgetset</name>
  <value>com.vaadin.portal.PortalsDefaultWidgetSet</value>
</init-param>
...
```

The **PortalsDefaultWidgetSet** extends **SamplerWidgetSet**, which extends the **DefaultWidgetSet**. The **DefaultWidgetSet** is therefore essentially a subset of **PortalsDefaultWidgetSet**, which contains also the widgets required by the Sampler demo. Other applications that would otherwise require only the regular **DefaultWidgetSet**, and do not define their own widgets, can just as well use the larger set, making them compatible with the demos. The **PortalsDefaultWidgetSet** will also be the default Vaadin widgetset bundled in Liferay 5.3 and later.

If your portlets are contained in multiple WARs, which can happen quite typically, you need to install the widget set and theme portal-wide so that all the portlets can use them. See Section 13.3, "Developing Vaadin Portlets for Liferay" on configuring the widget sets in the portal itself.

13.5.6. Building the WAR Package

To deploy the portlet, you need to build a WAR package. For production deployment, you probably want to either use Maven or an Ant script to build the package. In Eclipse, you can right-click on the project and select **Export** → **WAR**. Choose a name for the package and a

target. If you have installed Vaadin in the portal as described in Section 13.3, “Developing Vaadin Portlets for Liferay”, you should exclude all the Vaadin libraries, as well as widget set and themes from the WAR.

13.5.7. Deploying the WAR Package

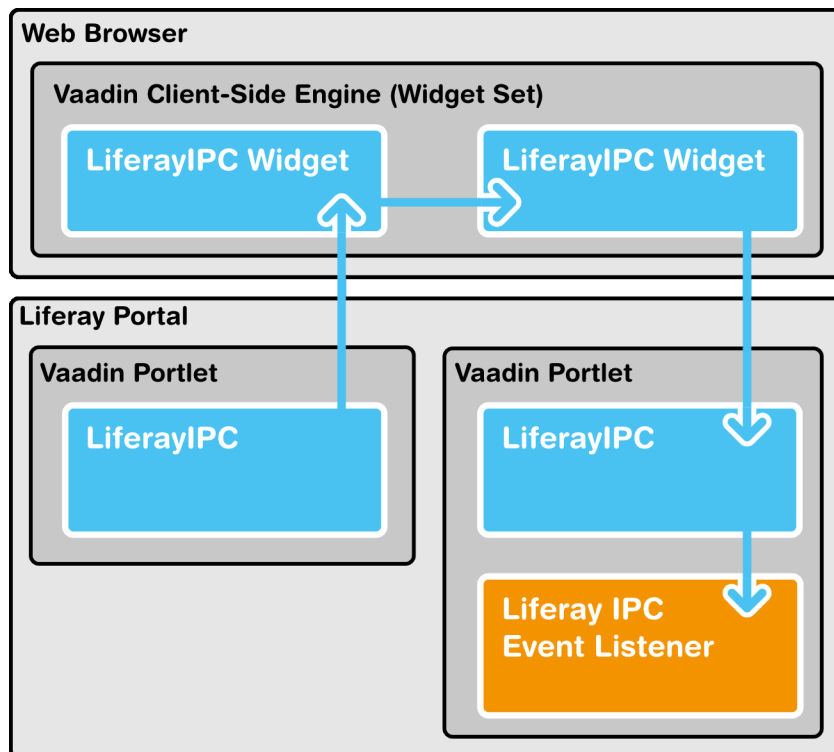
How you actually deploy a WAR package depends on the portal. In Liferay, you simply drop it to the `deploy` subdirectory under the Liferay installation directory. The deployment depends on the application server under which Liferay runs; for example, if you use Liferay bundled with Tomcat, you will find the extracted package in the `webapps` directory under the Tomcat installation directory included in Liferay.

13.6. Vaadin IPC for Liferay

Portlets rarely live alone. A page can contain multiple portlets and when the user interacts with one portlet, you may need to have the other portlets react to the change immediately. This is not normally possible with Vaadin portlets, as Vaadin applications need to get an Ajax request from the client-side to change their user interface. On the other hand, the regular inter-portlet communication (IPC) mechanism in Portlet 2.0 Specification requires a complete page reload, but that is not appropriate with Vaadin or in general Ajax applications, which do not require a page reload. One solution is to communicate between the portlets on the server-side and then use a server-push mechanism to update the client-side.

The Vaadin IPC for Liferay Add-on takes another approach by communicating between the portlets through the client-side. Events (messages) are sent through the **LiferayIPC** component and the client-side widget relays them to the other portlets, as illustrated in Figure 13.6, “Vaadin IPC for Liferay Architecture”.

Figure 13.6. Vaadin IPC for Liferay Architecture

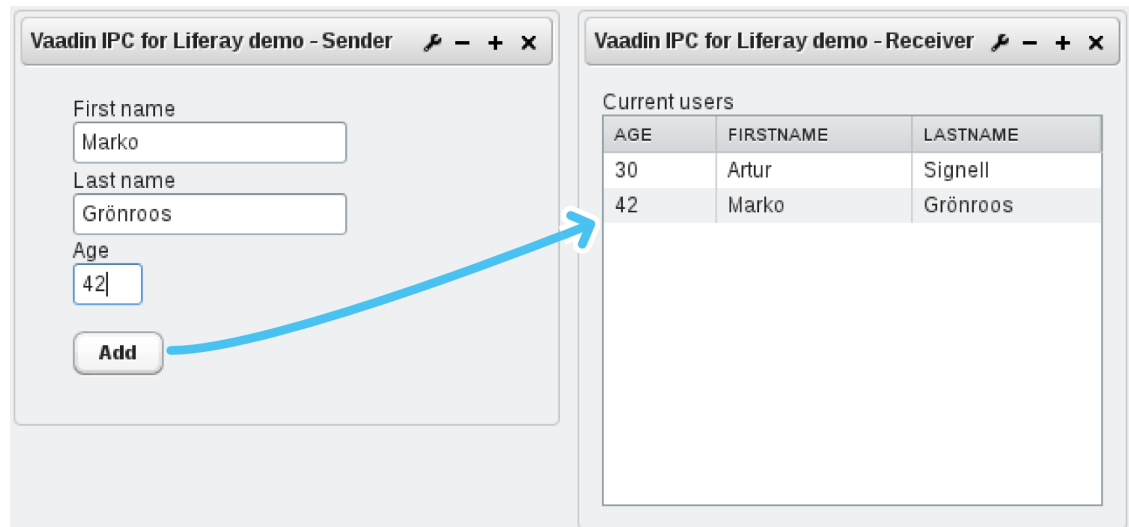


Vaadin IPC for Liferay uses the Liferay JavaScript event API for client-side inter-portlet communication, so you can communicate just as easily with other Liferay portlets.

Notice that you can use this communication only between portlets on the same page.

Figure 13.7, “Vaadin IPC Add-on Demo with Two Portlets” shows Vaadin IPC for Liferay in action. Entering a new item in one portlet is updated interactively in the other.

Figure 13.7. Vaadin IPC Add-on Demo with Two Portlets



13.6.1. Installing the Add-on

The Vaadin IPC for Liferay add-on is available from the Vaadin Directory as well as from a Maven repository. To download the installation package or find out the Maven or Ivy dependency, see the add-on page at Vaadin Directory, and install the add-on as described in Chapter 18, *Using Vaadin Add-ons*.

The contents of the installation package are as follows:

`vaadin-ipc-for-liferay-x.x.x.jar`

The add-on JAR in the installation package must be installed in the `WEB-INF/lib` directory under the root context. The location depends on the server - for example in Liferay running in Tomcat it is located under the `webapps/ROOT` folder of the server.

`doc`

The documentation folder includes a `README.TXT` file that describes the contents of the installation package briefly, and `licensing.txt` and `license-asl-2.0.txt`, which describe the licensing under the Apache License 2.0. Under the `doc/api` folder is included the complete JavaDoc API documentation for the add-on.

`vaadin-ipc-for-liferay-x.x.x-demo.war`

A WAR containing demo portlets. After installing the add-on library and compiling the widget set, as described below, you can deploy the WAR to Liferay and add the two demo portlets to a page, as shown in Figure 13.7, “Vaadin IPC Add-on Demo with Two Portlets”. The source of the demo is available at dev.vaadin.com/svn/addons/IP-CforLiferay/trunk/.

The add-on contains a widget set, which you must compile into the Vaadin widget set installed in the portal.

13.6.2. Basic Communication

LiferayIPC is an invisible user interface component that can be used to send messages between two or more Vaadin portlets. You add it to an application layout as you would any regular user interface component.

```
LiferayIPC liferayipc = new LiferayIPC();
layout.addComponent(liferayipc);
```

You should be careful not to remove the invisible component from the portlet later if you modify the layout of the portlet.

The component can be used both for sending and receiving messages, as described next.

Sending Events

You can send an event (a message) with the `sendEvent()` method, which takes an event ID and the message data as parameters. The event is broadcast to all listening portlets. The event ID is a string that can be used to identify the recipient of an event or the event type.

```
liferayipc.sendEvent("hello", "This is Data");
```

If you need to send more complex data, you need to format or serialize it to a string representation as described in Section 13.6.5, “Serializing and Encoding Data”.

Receiving Events

A portlet wishing to receive events (messages) from other portlets needs to register a listener in the component with `addListener()`. The listener receives the messages in a **LiferayIPCEvent** object. Filtering events by the ID is built in into the listener handler, you give the listened event ID as the first parameter for the `addListener()`. The actual message data is held in the `data` property, which you can read with `getData()`.

```
liferayipc.addListener("hello", new LiferayIPCEventListener() {
    public void eventReceived(LiferayIPCEvent event) {
        // Do something with the message data
        String data = event.getData();
        Notification.show("Received hello: " + data);
    }
});
```

A listener added to a **LiferayIPC** can be removed with `removeListener()`.

13.6.3. Considerations

Both security and efficiency should be considered with inter-portlet communications when using the Vaadin IPC for Liferay.

Browser Security

As the message data is passed through the client-side (browser), any code running in the browser has access to the data. You should be careful not to expose any security-critical data in client-side messaging. Also, malicious code running in the browser could alter or fake mes-

sages. Sanitization can help with the latter problem and encryption to solve the both issues. You can also share the sensitive data through session attributes or a database and use the client-side IPC only to notify that the data is available.

Efficiency

Sending data through the browser requires loading and sending it in HTTP requests. The data is held in the memory space of the browser, and handling large data in the client-side JavaScript code can take time. Noticeably large message data can therefore reduce the responsiveness of the application and could, in extreme cases, go over browser limits for memory consumption or JavaScript execution time.

13.6.4. Communication Through Session Attributes

In many cases, such as when considering security or efficiency, it is better to pass the bulk data on the server-side and use the client-side IPC only for notifying the other portlet(s) that the data is available. Session attributes are a convenient way of sharing data on the server-side. You can also share objects through them, not just strings.

The session variables have a *scope*, which should be `APPLICATION_SCOPE`. The "application" refers to the scope of the Java web application (WAR) that contains the portlets.

If the communicating portlets are in the same Java web application (WAR), no special configuration is needed. You can also communicate between portlets in different WARs, in which case you need to disable the `private-session-attributes` parameter in `liferay-portlet.xml` by setting it to `false`. Please see Liferay documentation for more information regarding the configuration.

You can also share Java objects between the portlets in the same WAR, not just strings. If the portlets are in different WARs, they normally have different class loaders, which could cause incompatibilities, so you can only communicate with strings and any object data needs to be serialized.

Session attributes are accessible through the **PortletSession** object, which you can access through the portlet context from the Vaadin **Application** class.

```
Person person = new Person(firstname, lastname, age);
...

PortletSession session =
    ((PortletApplicationContext2)getContext()).
        getPortletSession();

// Share the object
String key = "IPCDEMO_person";
session.setAttribute(key, person,
    PortletSession.APPLICATION_SCOPE);

// Notify that it's available
liferayipc.sendEvent("ipc_demodata_available", key);
```

You can then receive the attribute in a **LiferayIPCEventListener** as follows:

```
public void eventReceived(LiferayIPCEvent event) {
    String key = event.getData();

    PortletSession session =
```

```
((PortletApplicationContext2)getContext()).
    getPortletSession();

// Get the object reference
Person person = (Person) session.getAttribute(key);

// We can now use the object in our application
BeanItem<Person> item = new BeanItem<Person> (person);
form.setItemDataSource(item);
}
```

Notice that changes to a shared object bound to a user interface component are not updated automatically if it is changed in another portlet. The issue is the same as with double-binding in general.

13.6.5. Serializing and Encoding Data

The IPC events support transmitting only plain strings, so if you have object or other non-string data, you need to format or serialize it to a string representation. For example, the demo application formats the trivial data model as a semicolon-separated list as follows:

```
private void sendPersonViaClient(String firstName,
                                String lastName, int age) {
    liferayIPC_1.sendEvent("newPerson", firstName + ";" +
                           lastName + ";" + age);
}
```

You can use standard Java serialization for any classes that implement the `Serializable` interface. The transmitted data may not include any control characters, so you also need to encode the string, for example by using Base64 encoding.

```
// Some serializable object
MyBean mybean = new MyBean();
...

// Serialize
ByteArrayOutputStream baostr = new ByteArrayOutputStream();
ObjectOutputStream oostr;
try {
    oostr = new ObjectOutputStream(baostr);
    oostr.writeObject(mybean); // Serialize the object
    oostr.close();
} catch (IOException e) {
    Notification.show("IO PAN!"); // Complain
}

// Encode
BASE64Encoder encoder = new BASE64Encoder();
String encoded = encoder.encode(baostr.toByteArray());

// Send the IPC event to other portlet(s)
liferayipc.sendEvent("mybeanforyou", encoded);
```

You can then deserialize such a message at the receiving end as follows:

```
public void eventReceived(LiferayIPCEvent event) {
    String encoded = event.getData();

    // Decode and deserialize it
}
```

```
BASE64Decoder decoder = new BASE64Decoder();
try {
    byte[] data = decoder.decodeBuffer(encoded);
    ObjectInputStream ois =
        new ObjectInputStream(
            new ByteArrayInputStream(data));

    // The deserialized bean
    MyBean deserialized = (MyBean) ois.readObject();
    ois.close();

    ... do something with the bean ...

} catch (IOException e) {
    e.printStackTrace(); // Handle somehow
} catch (ClassNotFoundException e) {
    e.printStackTrace(); // Handle somehow
}
```

13.6.6. Communicating with Non-Vaadin Portlets

You can use the Vaadin IPC for Liferay to communicate also between a Vaadin application and other portlets, such as JSP portlets. The add-on passes the events as regular Liferay JavaScript events. The demo WAR includes two JSP portlets that demonstrate the communication.

When sending events from non-Vaadin portlet, fire the event using the JavaScript `Liferay.fire()` method with an event ID and message. For example, in JSP you could have:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0"
    prefix="portlet" %>
<portlet:defineObjects />

<script>
function send_message() {
    Liferay.fire('hello', "Hello, I'm here!");
}
</script>

<input type="button" value="Send message"
    onclick="send_message()" />
```

You can receive events using a Liferay JavaScript event handler. You define the handler with the `on()` method in the Liferay object. It takes the event ID and a callback function as its parameters. Again in JSP you could have:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0"
    prefix="portlet" %>
<portlet:defineObjects />

<script>
Liferay.on('hello', function(event, data) {
    alert("Hello: " + data);
});
</script>
```

Chapter 14

Client-Side Vaadin Development

14.1. Overview	447
14.2. Installing the Client-Side Development Environment	448
14.3. Client-Side Module Descriptor	448
14.4. Compiling a Client-Side Module	449
14.5. Creating a Custom Widget	450
14.6. Debugging Client-Side Code	452

This chapter gives an overview of the Vaadin client-side framework, its architecture, and development tools.

14.1. Overview

As noted in the introduction, Vaadin supports two development models: server-side and client-side. Client-side Vaadin code is executed in the web browser as JavaScript code. The code is written in Java, like all Vaadin code, and then compiled to JavaScript with the *Vaadin Client Compiler*. You can develop client-side widgets and integrate them with server-side counterpart components to allow using them in server-side Vaadin applications. That is how the components in the server-side framework and in most add-ons are done. Alternatively, you can create pure client-side GWT applications, which you can simply load in the browser from an HTML page and use even without server-side connectivity.

The client-side framework is based on the Google Web Toolkit (GWT), with added features and bug fixes. Vaadin is compatible with GWT to the extent of the basic GWT feature set. Vaadin

Ltd is a member of the GWT Steering Committee, working on the future direction of GWT together with Google and other supporters of GWT.



Widgets and Components

Google Web Toolkit uses the term *widget* for user interface components. In this book, we use the term *widget* to refer to client-side components, while using the term *component* in a general sense and also in the special sense for server-side components.

The main idea in server-side Vaadin development is to render the server-side components in the browser with the Client-Side Engine. The engine is essentially a set of widgets paired with *connectors* that serialize their state and events with the server-side counterpart components. The client-side engine is technically called a *widget set*, to describe the fact that it mostly consists of widgets and that widget sets can be combined, as described later.

14.2. Installing the Client-Side Development Environment

The installation of the client-side development libraries is described in Chapter 3, *Creating a Vaadin Application*. You especially need the `vaadin-client` library, which contains the client-side Java API, and `vaadin-client-compiler`, which contains the Vaadin Client Compiler for compiling Java to JavaScript.

14.3. Client-Side Module Descriptor

Client-side Vaadin modules, such as the Vaadin Client-Side Engine (widget set) or pure client-side applications, that are to be compiled to JavaScript, are defined in a *module descriptor* (`.gwt.xml`) file.

When defining a widget set to build the Vaadin client-side engine, the only necessary task is to inherit a base widget set. If you are developing a regular widget set, you should normally inherit the **DefaultWidgetSet**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
    "-//Google Inc.//DTD Google Web Toolkit 1.7.0//EN"
    "http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-
source/core/src/gwt-module.dtd">

<module>
    <!-- Inherit the default widget set -->
    <inherits name="com.vaadin.DefaultWidgetSet" />
</module>
```

If you are developing a pure client-side application, you should instead inherit **com.vaadin.Vaadin**, as described in Chapter 15, *Client-Side Applications*. In that case, the module descriptor also needs an entry-point.

If you are using the Eclipse IDE, the New Vaadin Widget wizard will automatically create the GWT module descriptor. See Section 17.2.1, “Creating a Widget” for detailed instructions.

14.3.1. Specifying a Stylesheet

A client-side module can include CSS stylesheets. When the module is compiled, these stylesheets are copied to the output target. In the module descriptor, define a `stylesheet` element.

For example, if you are developing a custom widget and want to have a default stylesheet for it, you could define it as follows:

```
<stylesheet src="mywidget/styles.css"/>
```

The specified path is relative to the *public* folder under the folder of the module descriptor.

14.3.2. Limiting Compilation Targets

Compiling widget sets takes considerable time. You can reduce the compilation time significantly by compiling the widget sets only for your browser, which is useful during development. You can do this by setting the *user.agent* property in the module descriptor.

```
<set-property name="user.agent" value="gecko1_8"/>
```

The *value* attribute should match your browser. The browsers supported by GWT depend on the GWT version, below is a list of browser identifiers supported by GWT.

Table 14.1. GWT User Agents

Identifier	Name
ie6	Internet Explorer 6
ie8	Internet Explorer 8
gecko1_8	Mozilla Firefox 1.5 and later
safari	Apple Safari and other Webkit-based browsers including Google Chrome
opera	Opera
ie9	Internet Explorer 9

For more information about the GWT Module XML Format, please see Google Web Toolkit Developer Guide.

14.4. Compiling a Client-Side Module

A client-side module, either a widget set or a pure client-side module, needs to be compiled to JavaScript using the Vaadin Client Compiler. During development, the Development Mode makes the compilation automatically when you reload the page, provided that the module has been initially compiled once with the compiler.

As most Vaadin add-ons include widgets, widget set compilation is usually needed when using add-ons. In that case, the widget sets from different add-ons are compiled into a *project widget set*, as described in Chapter 18, *Using Vaadin Add-ons*.

14.4.1. Vaadin Compiler Overview

The Vaadin Client Compiler compiles Java to JavaScript. It is provided as the `vaadin-client-compiler` JAR, which you can execute with the `-jar` parameter for the Java runtime. It requires the `vaadin-client` JAR, which contains the Vaadin client-side framework.

The compiler compiles a *client module*, which can be either a pure client-side module or a Vaadin widget set, that is, the Vaadin Client-Side Engine that includes the widgets used in the application. The client module is defined with a module descriptor, which was described in Section 14.3, “Client-Side Module Descriptor”.

The compiler writes the compilation result to a target folder that will include the compiled JavaScript with any static resources included in the module.

14.4.2. Compiling in Eclipse

When the Vaadin Plugin is installed in Eclipse, you can simply click the button in the toolbar. It will compile the widget set it finds from the project. If the project has multiple widget sets, such as one for custom widgets and another one for the project, you need to select the module descriptor of the widget set to compile before clicking the button.

The compilation with Vaadin Plugin for Eclipse currently requires that the module descriptor has suffix `Widgetset.gwt.xml`, although you can use it to compile also other client-side modules than widget sets. The result is written under `WebContent/VAADIN/widgetsets` folder.

14.4.3. Compiling with Ant

You can find a script template for compiling widget sets with Ant and Ivy at the Vaadin download page. You can copy the build script to your project and, once configured, run it with Ant.

14.4.4. Compiling with Maven

You can compile the widget set with the `vaadin:compile` goal as follows:

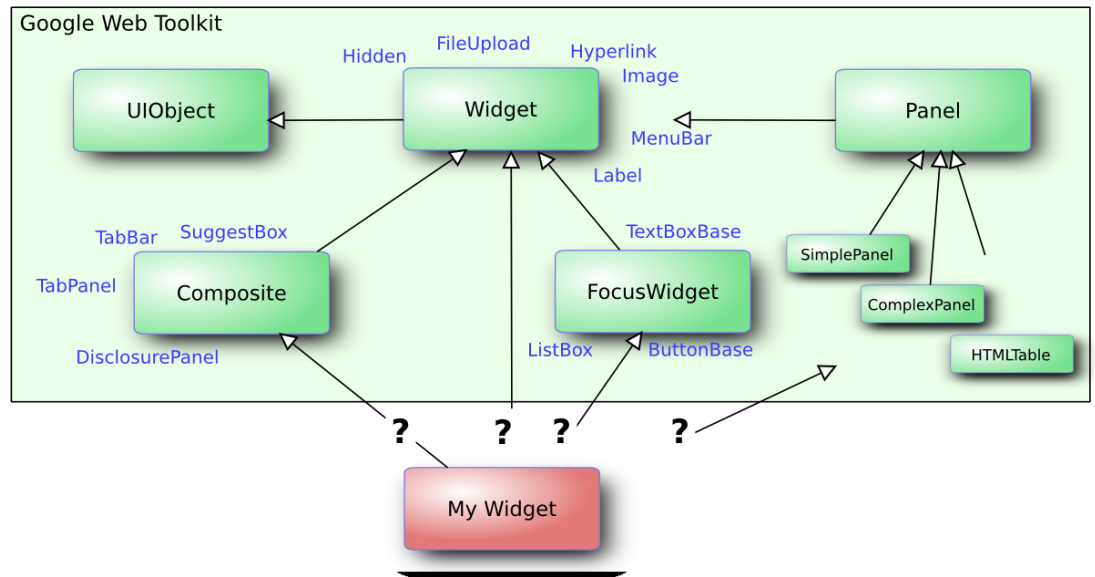
```
$ mvn vaadin:compile
```

14.5. Creating a Custom Widget

Creating a new Vaadin component usually begins from making a client-side widget, which is later integrated with a server-side counterpart to enable server-side development. In addition, you can also choose to make pure client-side widgets, a possibility which we also describe later in this section.

14.5.1. A Basic Widget

All widgets extend the **Widget** class or some of its subclasses. You can extend any core GWT or supplementary Vaadin widgets. Perhaps typically, an abstraction such as **Composite**. The basic GWT widget component hierarchy is illustrated in Figure 14.1, “GWT widget base class hierarchy”. Please see the GWT API documentation for a complete description of the widget classes.

Figure 14.1. GWT widget base class hierarchy

For example, we could extend the **Label** widget to display some custom text.

```
package com.example.myapp.client;

import com.google.gwt.user.client.ui.Label;

public class MyWidget extends Label {
    public static final String CLASSNAME = "mywidget";

    public MyWidget() {
        setStyleName(CLASSNAME);
        setText("This is MyWidget");
    }
}
```

The above example is largely what the Eclipse plugin generates as a widget stub. It is a good practice to set a distinctive style class for the widget, to allow styling it with CSS.

The client-side source code *must* be contained in a `client` package under the package of the descriptor file, which is covered later.

14.5.2. Using the Widget

You can use a custom widget just like you would use any widget, possibly integrating it with a server-side component, or in pure client-side modules such as the following:

```
public class MyEntryPoint implements EntryPoint {
    @Override
    public void onModuleLoad() {
        // Use the custom widget
        final MyWidget mywidget = new MyWidget();
        RootPanel.get().add(mywidget);
    }
}
```

14.6. Debugging Client-Side Code

Vaadin currently includes SuperDevMode for debugging client-side code right in the browser.

The predecessor of SuperDevMode, the GWT Development Mode, no longer works in recent versions of Firefox and Chrome, because of certain API changes in the browsers. There exists workarounds on some platforms, but for the sake of simplicity, we recommend using the SuperDevMode.

14.6.1. Launching SuperDevMode

The SuperDevMode is much like the old Development Mode, except that it does not require a browser plugin. Compilation from Java to JavaScript is done incrementally, reducing the compilation time significantly. It also allows debugging JavaScript and even Java right in the browser (currently only supported in Chrome).

You can enable SuperDevMode as follows:

1. You need to set a redirect property in the `.gwt.xml` module descriptor as follows:

```
<set-configuration-property name="devModeRedirectEnabled" value="true" />
```

In addition, you need the `xsiframe` linker. It is included in the **com.vaadin.DefaultWidgetSet** as well as in the **com.vaadin.Vaadin** module. Otherwise, you need to include it with:

```
<add-linker name="xsiframe" />
```

2. Compile the module (that is, the widget set), for example by clicking the button in Eclipse.
3. If you are using Eclipse, create a launch configuration for the SuperDevMode by clicking the **Create SuperDevMode launch** in the **Vaadin** section of the project properties.
 - a. The main class to execute should be **com.google.gwt.dev.codeserver.CodeServer**.
 - b. The application takes the fully-qualified class name of the module (or widget set) as parameter, for example, **com.example.myproject.widgetset.MyprojectWidgetset**.
 - c. Add project sources to the class path of the launch if they are not in the project class path.

The above configuration only needs to be done once to enable the SuperDevMode. After that, you can launch the mode as follows:

1. Run the SuperDevMode Code Server with the launch configuration that you created above. This performs the initial compilation of your module or widget set.
2. Launch the servlet container for your application, for example, Tomcat.
3. Open your browser with the application URL and add `?superdevmode` parameter to the URL (see the notice below if you are not extending **DefaultWidgetSet**). This re-

compiles the code, after which the page is reloaded with the SuperDevMode. You can also use the `?debug` parameter and then click the **SDev** button in the debug console.

If you make changes to the client-side code and refresh the page in the browser, the client-side is recompiled and you see the results immediately.

The Step 3 above assumes that you extend **DefaultWidgetSet** in your module. If that is not the case, you need to add the following at the start of the `onModuleLoad()` method of the module:

```
if (SuperDevMode.enableBasedOnParameter()) { return; }
```

Alternatively, you can use the bookmarklets provided by the code server. Go to <http://localhost:9876/> and drag the bookmarklets "**Dev Mode On**" and "**Dev Mode Off**" to the bookmarks bar

14.6.2. Debugging Java Code in Chrome

Chrome supports source maps, which allow debugging Java source code from which the JavaScript was compiled.

Open the Chrome Inspector by right-clicking and selecting **Inspect Element**. Click the settings icon in the lower corner of the window and check the **Scripts** → **Enable source maps** option. Refresh the page with the Inspector open, and you will see Java code instead of JavaScript in the scripts tab.

Chapter 15

Client-Side Applications

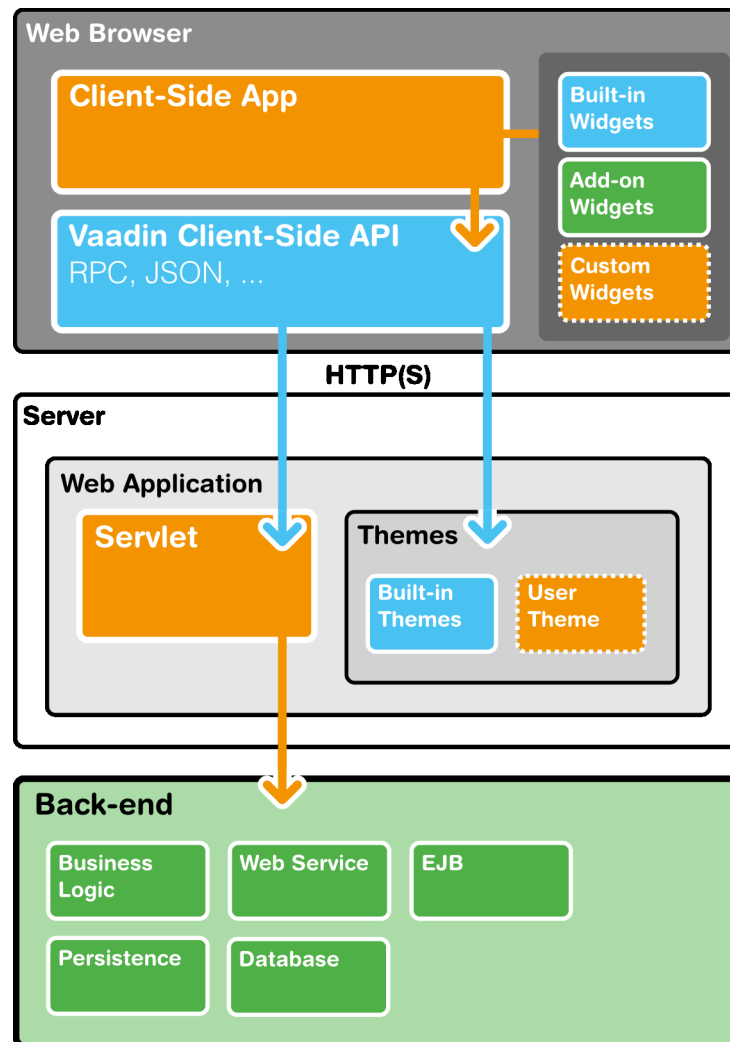
15.1. Overview	455
15.2. Client-Side Module Entry-Point	457
15.3. Compiling and Running a Client-Side Application	458
15.4. Loading a Client-Side Application	458

This chapter describes how to develop client-side Vaadin applications.

We only give a brief introduction to the topic in this book. Please refer to the GWT documentation for a more complete treatment of the many GWT features.

15.1. Overview

Vaadin allows developing client-side modules that run in the browser. Client-side modules can use all the GWT widgets and some Vaadin-specific widgets, as well as the same themes as server-side Vaadin applications. Client-side applications run in the browser, even with no further server communications. When paired with a server-side service to gain access to data storage and server-side business logic, client-side applications can be considered "fat clients", in comparison to the "thin client" approach of the server-side Vaadin applications. The services can use the same back-end services as server-side Vaadin applications. Fat clients are useful for a range of purposes when you have a need for highly responsive UI logic, such as for games or for serving a huge number of clients with possibly stateless server-side code.

Figure 15.1. Client-Side Application Architecture

A client-side application is defined as a *module*, which has an *entry-point* class. Its `onModuleLoad()` method is executed when the JavaScript of the compiled module is loaded in the browser.

Consider the following client-side application:

```
public class HelloWorld implements EntryPoint {
    @Override
    public void onModuleLoad() {
        RootPanel.get().add(new Label("Hello, world!"));
    }
}
```

The user interface of a client-side application is built under a HTML *root element*, which can be accessed by `RootPanel.get()`. The purpose and use of the entry-point is documented in more detail in Section 15.2, “Client-Side Module Entry-Point”. The user interface is built from *widgets* hierarchically, just like with server-side Vaadin UIs. The built-in widgets and their relationships are catalogued in Chapter 16, *Client-Side Widgets*. You can also use many of the widgets in Vaadin add-ons that have them, or make your own.

A client-side module is defined in a *module descriptor*, as described in Section 14.3, “Client-Side Module Descriptor”. A module is compiled from Java to JavaScript using the Vaadin Compiler, of which use was described in Section 14.4, “Compiling a Client-Side Module”. The Section 15.3, “Compiling and Running a Client-Side Application” in this chapter gives further information about compiling client-side applications. The resulting JavaScript can be loaded to any web page, as described in Section 15.4, “Loading a Client-Side Application”.

The client-side user interface can be built declaratively using the included *UI Binder*.

The servlet for processing RPC calls from the client-side can be generated automatically using the included compiler.

Even with regular server-side Vaadin applications, it may be useful to provide an off-line mode if the connection is closed. An off-line mode can persist data in a local store in the browser, thereby avoiding the need for server-side storage, and transmit the data to the server when the connection is again available. Such a pattern is commonly used with Vaadin TouchKit.

15.2. Client-Side Module Entry-Point

A client-side application requires an *entry-point* where the execution starts, much like the `init()` method in server-side Vaadin UIs.

Consider the following application:

```
package com.example.myapp.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.RootPanel;
import com.vaadin.ui.VButton;

public class MyEntryPoint implements EntryPoint {
    @Override
    public void onModuleLoad() {
        // Create a button widget
        Button button = new Button();
        button.setText("Click me!");
        button.addClickHandler(new ClickHandler() {
            @Override
            public void onClick(ClickEvent event) {
                mywidget.setText("Hello, world!");
            }
        });
        RootPanel.get().add(button);
    }
}
```

Before compiling, the entry-point needs to be defined in a module descriptor, as described in the next section.

15.2.1. Module Descriptor

The entry-point of a client-side application is defined, along with any other configuration, in a client-side module descriptor, described in Section 14.3, “Client-Side Module Descriptor”. The descriptor is an XML file with suffix `.gwt.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
"-//Google Inc.//DTD Google Web Toolkit 1.7.0//EN"
"http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-
source/core/src/gwt-module.dtd">
<module>
  <!-- Builtin Vaadin and GWT widgets -->
  <inherits name="com.vaadin.Vaadin" />

  <!-- The entry-point for the client-side application -->
  <entry-point class="com.example.myapp.client.MyEntryPoint"/>
</module>
```

You might rather want to inherit the **com.google.gwt.user.User** to get just the basic GWT widgets, and not the Vaadin-specific widgets and classes, most of which are unusable in pure client-side applications.

You can put static resources, such as images or CSS stylesheets, in a `public` folder (not a Java package) under the folder of the descriptor file. When the module is compiled, the resources are copied to the output folder. Normally in pure client-side application development, it is easier to load them in the HTML host file or in a **ClientBundle** (see GWT documentation), but these methods are not compatible with server-side component integration, if you use the resources for that purpose as well.

15.3. Compiling and Running a Client-Side Application

Compilation of client-side modules other than widget sets with the Vaadin Plugin for Eclipse has recent changes and limitations at the time of writing of this edition and the information given here may not be accurate.

The application needs to be compiled into JavaScript to run it in a browser. For deployment, and also initially for the first time when running the Development Mode, you need to do the compilation with the Vaadin Client Compiler, as described in Section 14.4, “Compiling a Client-Side Module”.

During development, it is easiest to use the SuperDevMode, which also quickly launching the client-side code and also allows debugging. See Section 14.6, “Debugging Client-Side Code” for more details.

15.4. Loading a Client-Side Application

You can load the JavaScript code of a client-side application in an HTML *host page* by including it with a `<script>` tag, for example as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />

    <title>Embedding a Vaadin Application in HTML Page</title>

    <!-- Load the Vaadin style sheet -->
    <link rel="stylesheet"
          type="text/css"
          href="/myproject/VAADIN/themes/reindeer/legacy-styles.css"/>
  </head>
```



```
<body>
  <h1>A Pure Client-Side Application</h1>

  <script type="text/javascript" language="javascript"
    src="clientside/com.example.myapp.MyModule/
      com.example.myapp.MyModule.nocache.js">
  </script>
</body>
</html>
```

The JavaScript module is loaded in a `<script>` element. The `src` parameter should be a relative link from the host page to the compiled JavaScript module.

If the application uses any supplementary Vaadin widgets, and not just core GWT widgets, you need to include the Vaadin theme as was done in the example. The exact path to the style file depends on your project structure - the example is given for a regular Vaadin application where themes are contained in the `VAADIN` folder in the WAR.

In addition to CSS and scripts, you can load any other resources needed by the client-side application in the host page.

Chapter 16

Client-Side Widgets

16.1. Overview	461
16.2. GWT Widgets	462
16.3. Vaadin Widgets	462
16.4. Grid	462

This chapter gives basic documentation on the use of the Vaadin client-side framework for the purposes of creating client-side applications and writing your own widgets.

We only give a brief introduction to the topic in this chapter. Please refer to the GWT documentation for a more complete treatment of the GWT widgets.

16.1. Overview

The Vaadin client-side API is based on the Google Web Toolkit. It involves *widgets* for representing the user interface as Java objects, which are rendered as a HTML DOM in the browser. Events caused by user interaction with the page are delegated to event handlers, where you can implement your UI logic.

In general, the client-side widgets come in two categories - basic GWT widgets and Vaadin-specific widgets. The library includes *connectors* for integrating the Vaadin-specific widgets with the server-side components, thereby enabling the server-side development model of Vaadin. The integration is described in Chapter 17, *Integrating with the Server-Side*.

The layout of the client-side UI is managed with *panel*/widgets, which correspond in their function with layout components in the Vaadin server-side API.

In addition to the rendering API, the client-side API includes facilities for making HTTP requests, logging, accessibility, internationalization, and testing.

For information about the basic GWT framework, please refer to <https://developers.google.com/web-toolkit/overview>.

16.2. GWT Widgets

GWT widgets are user interface elements that are rendered as HTML. Rendering is done either by manipulating the HTML Document Object Model (DOM) through the lower-level DOM API, or simply by injecting the HTML with `setInnerHTML()`. The layout of the user interface is managed using special panel widgets.

For information about the basic GWT widgets, please refer to the GWT Developer's Guide at <https://developers.google.com/web-toolkit/doc/latest/DevGuideUi>.

16.3. Vaadin Widgets

Vaadin comes with a number of Vaadin-specific widgets in addition to the GWT widgets, some of which you can use in pure client-side applications. The Vaadin widgets have somewhat different feature set from the GWT widgets and are foremost intended for integration with the server-side components, but some may prove useful for client-side applications as well.

```
public class MyEntryPoint implements EntryPoint {
    @Override
    public void onModuleLoad() {
        // Add a Vaadin button
        VButton button = new VButton();
        button.setText("Click me!");
        button.addClickHandler(new ClickHandler() {
            @Override
            public void onClick(ClickEvent event) {
                mywidget.setText("Clicked!");
            }
        });

        RootPanel.get().add(button);
    }
}
```

16.4. Grid

The **Grid** widget is the client-side counterpart for the server-side **Grid** component described in Section 6.24, “**Grid**”.

The client-side API is almost identical to the server-side API, so its documentation is currently omitted here and we refer you to the API documentation. In the following, we go through some customization features of **Grid**.

16.4.1. Renderers

As described in Section 6.24.6, “Column Renderers”, renderers draw the visual representation of data values on the client-side. They implement `Renderer` interface and its `render()` method. The method gets a reference to the element of the grid cell, as well as the data value to be rendered. An implementation needs to modify the element as needed.

For example, **TextRenderer** is implemented simply as follows:

```
public class TextRenderer implements Renderer<String> {
    @Override
    public void render(RendererCellReference cell,
                      String text) {
        cell.getElement().setInnerText(text);
    }
}
```

The server-side renderer API should extend **AbstractRenderer** or **ClickableRenderer** with the data type accepted by the renderer. The data type also must be given for the superclass constructor.

```
public class TextRenderer extends AbstractRenderer<String> {
    public TextRenderer() {
        super(String.class);
    }
}
```

The client-side and server-side renderer need to be connected with a connector extending from **AbstractRendererConnector**.

```
@Connect(com.vaadin.ui.renderer.TextRenderer.class)
public class TextRendererConnector
    extends AbstractRendererConnector<String> {
    @Override
    public TextRenderer getRenderer() {
        return (TextRenderer) super.getRenderer();
    }
}
```

Renderers can have parameters, for which normal client-side communication of extension parameters can be used. Please see the implementations of different renderers for examples.

Chapter 17

Integrating with the Server-Side

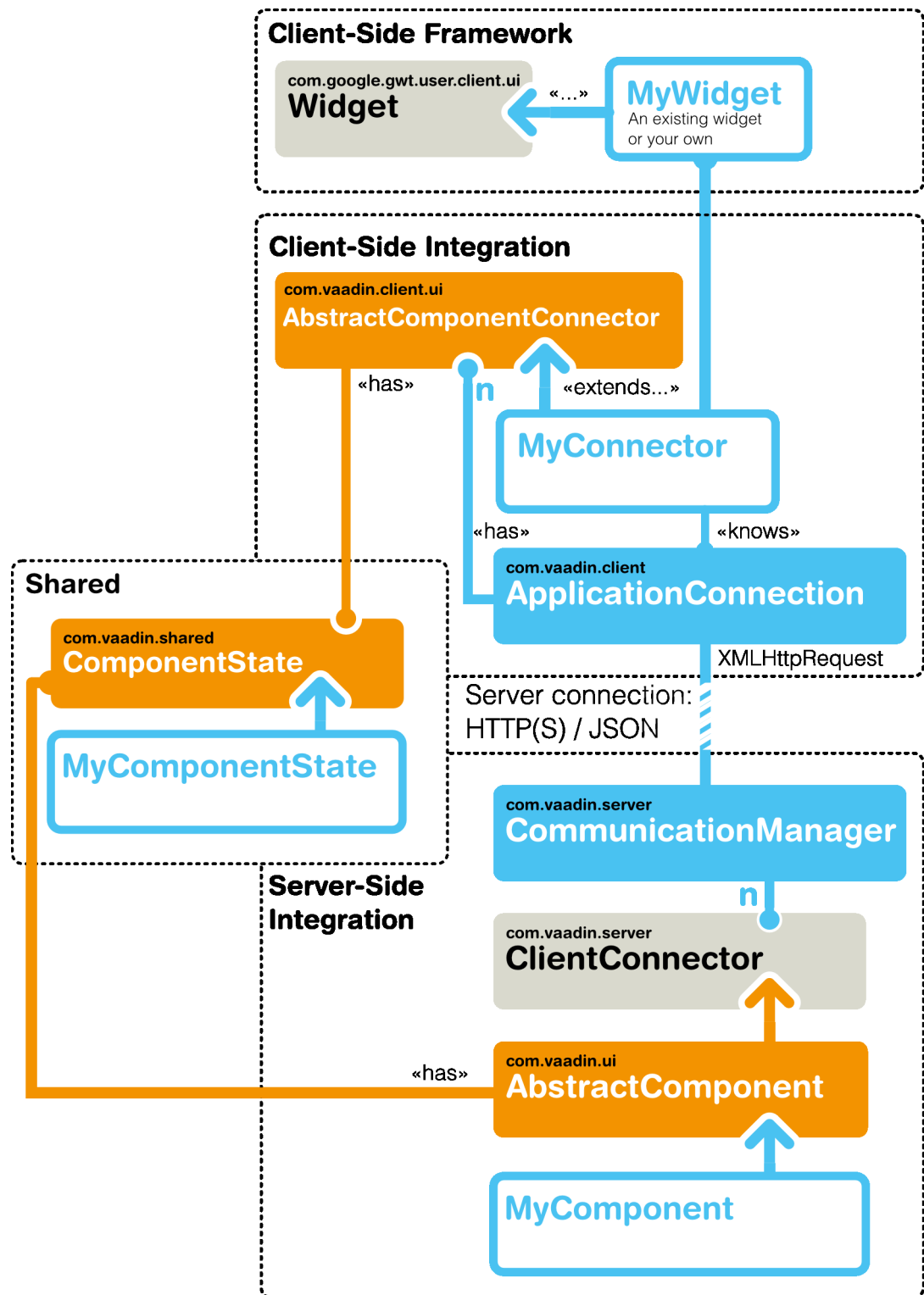
17.1. Overview	466
17.2. Starting It Simple With Eclipse	469
17.3. Creating a Server-Side Component	472
17.4. Integrating the Two Sides with a Connector	473
17.5. Shared State	474
17.6. RPC Calls Between Client- and Server-Side	477
17.7. Component and UI Extensions	479
17.8. Styling a Widget	481
17.9. Component Containers	482
17.10. Advanced Client-Side Topics	482
17.11. Creating Add-ons	483
17.12. Migrating from Vaadin 6	485
17.13. Integrating JavaScript Components and Extensions	486

This chapter describes how you can integrate client-side widgets or JavaScript components with a server-side component. The client-side implementations of all standard server-side components in Vaadin use the same client-side interfaces and patterns.

17.1. Overview

Vaadin components consist of two parts: a server-side and a client-side component. The latter are also called *widgets* in Google Web Toolkit (GWT) parlance. A Vaadin application uses the API of the server-side component, which is rendered as a client-side widget in the browser. As on the server-side, the client-side widgets form a hierarchy of layout widgets and regular widgets as the leaves.

Figure 17.1. Integration of Client-Side Widgets



The communication between a client-side widget and a server-side component is managed with a *connector* that handles synchronizing the widget state and events to and from the server-side.

When rendering the user interface, a client-side connector and a widget are created for each server-side component. The mapping from a component to a connector is defined in the connector class with a `@Connect` annotation, and the widget is created by the connector class.

The state of a server-side component is synchronized automatically to the client-side widget using a *shared state* object. A shared state object extends **AbstractComponentState** and it is used both in the server-side and the client-side component. On the client-side, a connector always has access to its state instance, as well to the state of its parent component state and the states of its children.

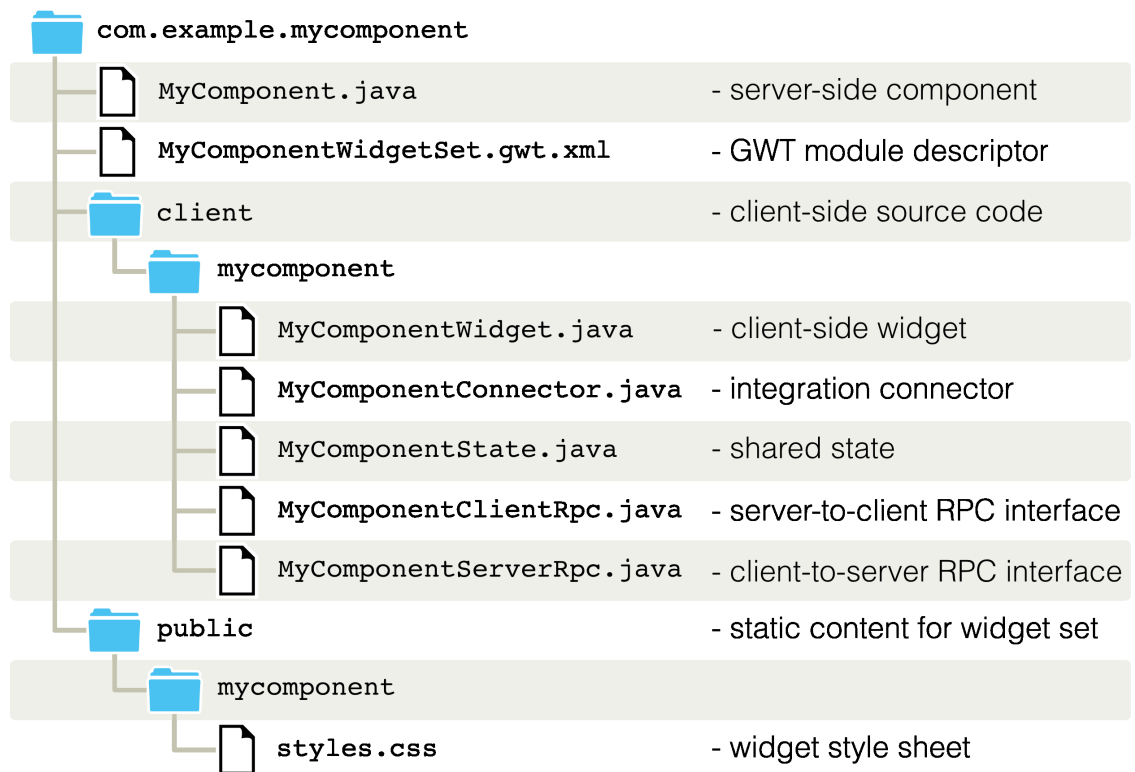
The state sharing assumes that state is defined with standard Java types, such as primitive and boxed primitive types, **String**, arrays, and certain collections (**List**, **Set**, and **Map**) of the supported types. Also the Vaadin **Connector** and some special internal types can be shared.

In addition to state, both server- and client-side can make remote procedure calls (RPC) to the other side. RPC is used foremost for event notifications. For example, when a client-side connector of a button receives a click, it sends the event to the server-side using RPC.

17.1.1. Project Structure

Widget set compilation, as described in Section 14.3, “Client-Side Module Descriptor”, requires using a special project structure, where the client-side classes are located under a `client` package under the package of the module descriptor. Any static resources, such as stylesheets and images, should be located under a `public` folder (not Java package). The source for the server-side component may be located anywhere, except not in the client-side package.

The basic project structure is illustrated in Figure 17.2, “Basic Widget Integration Project Structure”.

Figure 17.2. Basic Widget Integration Project Structure

The Eclipse wizard, described in Section 17.2, “Starting It Simple With Eclipse”, creates a widget integration skeleton with the above structure.

17.1.2. Integrating JavaScript Components

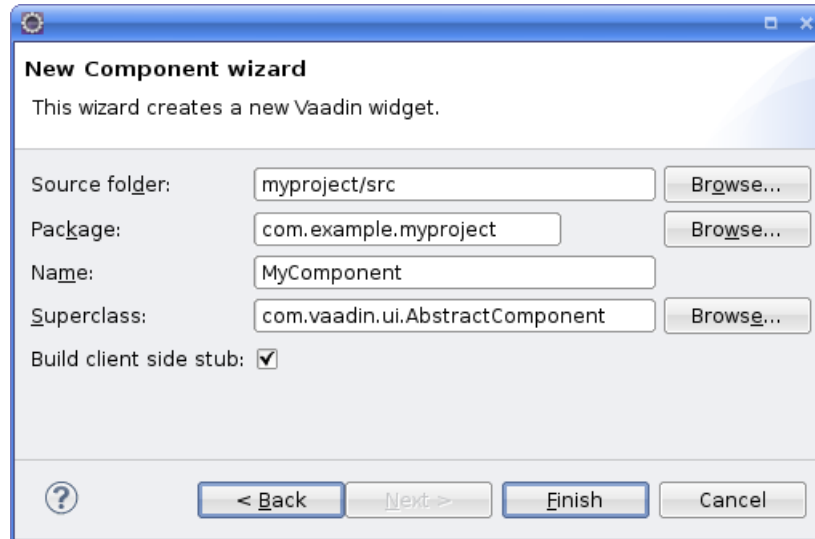
In addition to the GWT widget integration, Vaadin offers a simplified way to integrate pure JavaScript components. The JavaScript connector code is published from the server-side. As the JavaScript integration does not involve GWT programming, no widget set compilation is needed.

17.2. Starting It Simple With Eclipse

Let us first take the easy way and create a simple component with Eclipse. While you can develop new widgets with any IDE or even without, you may find Eclipse and the Vaadin Plugin for it useful, as it automates all the basic routines of widget development, most importantly the creation of new widgets.

17.2.1. Creating a Widget

1. Right-click the project in the Project Explorer and select **New** → **Other...**
2. In the wizard selection, select **Vaadin** → **Vaadin Widget** and click **Next**.
3. In the **New Component Wizard**, make the following settings.



Source folder

The root folder of the entire source tree. The default value is the default source tree of your project, and you should normally leave it unchanged unless you have a different project structure.

Package

The parent package under which the new server-side component should be created. If the project does not already have a widget set, one is created under this package in the widgetset subpackage. The subpackage will contain the `.gwt.xml` descriptor that defines the widget set and the new widget stub under the widgetset.client subpackage.

Name

The class name of the new *server-side component*. The name of the client-side widget stub will be the same but with "- **Widget**" suffix, for example, **MyComponentWidget**. You can rename the classes afterwards.

Superclass

The superclass of the server-side component. It is **AbstractComponent** by default, but **com.vaadin.ui.AbstractField** or **com.vaadin.ui.AbstractSelect** are other commonly used superclasses. If you are extending an existing component, you should select it as the superclass. You can easily change the superclass later.

Template

Select which template to use. The default is **Full fledged**, which creates the server-side component, the client-side widget, the connector, a shared state object, and an RPC object. The **Connector only** leaves the shared state and RPC objects out.

Finally, click **Finish** to create the new component.

The wizard will:

- Create a server-side component stub in the base package
- If the project does not already have a widget set, the wizard creates a GWT module descriptor file (`.gwt.xml`) in the base package and modifies the servlet class or the

`web.xml` deployment descriptor to specify the widget set class name parameter for the application

- Create a client-side widget stub (along with the connector and shared state and RPC stubs) in the `client.componentname` package under the base package

The structure of the server-side component and the client-side widget, and the serialization of component state between them, is explained in the subsequent sections of this chapter.

To compile the widget set, click the **Compile widget set** button in the Eclipse toolbar. See Section 17.2.2, “Compiling the Widget Set” for details. After the compilation finishes, you should be able to run your application as before, but using the new widget set. The compilation result is written under the `WebContent/VAADIN/widgetsets` folder. When you need to recompile the widget set in Eclipse, see Section 17.2.2, “Compiling the Widget Set”. For detailed information on compiling widget sets, see Section 14.4, “Compiling a Client-Side Module”.

The following setting is inserted in the `web.xml` deployment descriptor to enable the widget set:

```
<init-param>
  <description>Application widgetset</description>
  <param-name>widgetset</param-name>
  <param-value>com.example.myproject.widgetset.MyprojectApplicationWidget-
set</param-value>
</init-param>
```

You can refactor the package structure if you find need for it, but GWT compiler requires that the client-side code *must* always be stored under a package named "client" or a package defined with a `source` element in the widget set descriptor.

17.2.2. Compiling the Widget Set

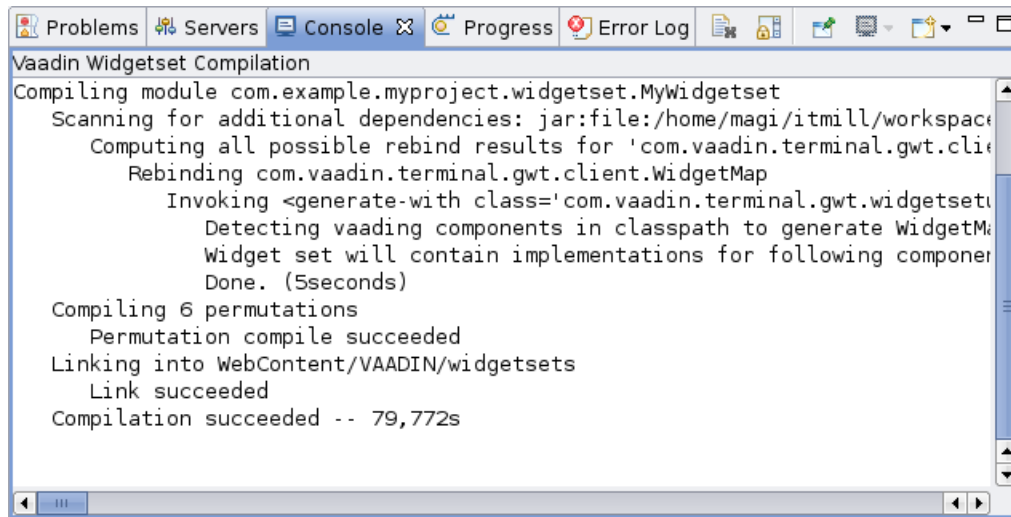
After you edit a widget, you need to compile the widget set. The Vaadin Plugin for Eclipse automatically suggests to compile the widget set in various situations, such as when you save a client-side source file. If this gets annoying, you can disable the automatic recompilation in the Vaadin category in project settings, by selecting the **Suspend automatic widgetset builds** option.

You can compile the widget set manually by clicking the **Compile widgetset** button in the Eclipse toolbar, shown in Figure 17.3, “The **Compile Widgetset** Button in Eclipse Toolbar”, while the project is open and selected. If the project has multiple widget set definition files, you need to select the one to compile in the Project Explorer.

Figure 17.3. The Compile Widgetset Button in Eclipse Toolbar



The compilation progress is shown in the **Console** panel in Eclipse, illustrated in Figure 17.4, “Compiling a Widget Set”. You should note especially the list of widget sets found in the class path.

Figure 17.4. Compiling a Widget Set

The compilation output is written under the `WebContent/VAADIN/widgetsets` folder, in a widget set specific folder.

You can speed up the compilation significantly by compiling the widget set only for your browser during development. The generated `.gwt.xml` descriptor stub includes a disabled element that specifies the target browser. See Section 14.3.2, “Limiting Compilation Targets” for more details on setting the `user-agent` property.

For more information on compiling widget sets, see Section 14.4, “Compiling a Client-Side Module”. Should you compile a widget set outside Eclipse, you need to refresh the project by selecting it in **Project Explorer** and pressing F5.

17.3. Creating a Server-Side Component

Typical server-side Vaadin applications use server-side components that are rendered on the client-side using their counterpart widgets. A server-side component must manage state synchronization between the widget on the client-side, in addition to any server-side logic.

17.3.1. Basic Server-Side Component

The component state is usually managed by a *shared state*, described later in Section 17.5, “Shared State”.

```

public class MyComponent extends AbstractComponent {
    public MyComponent() {
        getState().setText("This is MyComponent");
    }

    @Override
    protected MyComponentState getState() {
        return (MyComponentState) super.getState();
    }
}

```

17.4. Integrating the Two Sides with a Connector

A client-side widget is integrated with a server-side component with a *connector*. A connector is a client-side class that communicates changes to the widget state and events to the server-side.

A connector normally gets the state of the server-side component by the *shared state*, described later in Section 17.5, “Shared State”.

17.4.1. A Basic Connector

The basic tasks of a connector is to hook up to the widget and handle events from user interaction and changes received from the server. A connector also has a number of routine infrastructure methods which need to be implemented.

```
@Connect(MyComponent.class)
public class MyComponentConnector
    extends AbstractComponentConnector {

    @Override
    public MyComponentWidget getWidget() {
        return (MyComponentWidget) super.getWidget();
    }

    @Override
    public MyComponentState getState() {
        return (MyComponentState) super.getState();
    }

    @Override
    public void onStateChanged(StateChangeEvent stateChangeEvent)
    {
        super.onStateChanged(stateChangeEvent);

        // Do something useful
        final String text = getState().text;
        getWidget().setText(text);
    }
}
```

Here, we handled state change with the crude `onStateChanged()` method that is called when any of the state properties is changed. A finer and simpler handling is achieved by using the **@OnStateChange** annotation on a handler method for each property, or by **@DelegateToWidget** on a shared state property, as described later in Section 17.5, “Shared State”.

17.4.2. Communication with the Server-Side

The main task of a connector is to communicate user interaction with the widget to the server-side and receive state changes from the server-side and relay them to the widget.

Server-to-client communication is normally done using a *shared state*, as described in Section 17.5, “Shared State”, as well as RPC calls. The serialization of the state data is handled completely transparently.

For client-to-server communication, a connector can make remote procedure calls (RPC) to the server-side. Also, the server-side component can make RPC calls to the connector. For a thorough

description of the RPC mechanism, refer to Section 17.6, “RPC Calls Between Client- and Server-Side”.

17.5. Shared State

The basic communication from a server-side component to its the client-side widget counterpart is handled using a *shared state*. The shared state is serialized transparently. It should be considered read-only on the client-side, as it is not serialized back to the server-side.

A shared state object simply needs to extend the **AbstractComponentState**. The member variables should normally be declared as public.

```
public class MyComponentState extends AbstractComponentState {  
    public String text;  
}
```

A shared state should never contain any logic. If the members have private visibility for some reason, you can also use public setters and getters, in which case the property must not be public.

17.5.1. Location of Shared-State Classes

The shared-state classes are used by both server- and client-side classes, but widget set compilation requires that they must be located in a client-side source package. The default location is under a `client` package under the package of the `.gwt.xml` descriptor. If you wish to organize the shared classes separately from other client-side code, you can define separate client-side source packages for pure client-side classes and any shared classes. In addition to shared state classes, shared classes could include enumerations and other classes needed by shared-state or RPC communication.

For example, you could have the following definitions in the `.gwt.xml` descriptor:

```
<source path="client" />  
<source path="shared" />
```

The paths are relative to the package containing the descriptor.

17.5.2. Accessing Shared State on Server-Side

A server-side component can access the shared state with the `getState()` method. It is required that you override the base implementation with one that returns the shared state object cast to the proper type, as follows:

```
@Override  
public MyComponentState getState() {  
    return (MyComponentState) super.getState();  
}
```

You can then use the `getState()` to access the shared state object with the proper type.

```
public MyComponent() {  
    getState().setText("This is the initial state");  
    ....  
}
```


17.5.3. Handling Shared State in a Connector

A connector can access a shared state with the `getState()` method. The access should be read-only. It is required that you override the base implementation with one that returns the proper shared state type, as follows:

```
@Override
public MyComponentState getState() {
    return (MyComponentState) super.getState();
}
```

State changes made on the server-side are communicated transparently to the client-side. When a state change occurs, the `onStateChanged()` method in the connector is called. You should always call the superclass method before anything else to handle changes to common component properties.

```
@Override
public void onStateChanged(StateChangeEvent stateChangeEvent) {
    super.onStateChanged(stateChangeEvent);

    // Copy the state properties to the widget properties
    final String text = getState().getText();
    getWidget().setText(text);
}
```

The crude `onStateChanged()` method is called when any of the state properties is changed, allowing you to have even complex logic in how you manipulate the widget according to the state changes. In most cases, however, you can handle the property changes more easily and also more efficiently by using instead the **@OnStateChange** annotation on the handler methods for each property, as described next in Section 17.5.4, “Handling Property State Changes with **@OnStateChange**”, or by delegating the property value directly to the widget, as described in Section 17.5.5, “Delegating State Properties to Widget”.

17.5.4. Handling Property State Changes with @OnStateChange

The **@OnStateChange** annotation can be used to mark a connector method that handles state change on a particular property, given as parameter for the annotation. In addition to higher clarity, this avoids handling all property changes if a state change occurs in only one or some of them. However, if a state change can occur in multiple properties, you can only use this technique if the properties do not have interaction that prevents handling them separately in arbitrary order.

We can replace the `onStateChange()` method in the earlier connector example with the following:

```
@OnStateChange("text")
void updateText() {
    getWidget().setText(getState().text);
}
```

If the shared state property and the widget property have same name and do not require any type conversion, as is the case in the above example, you could simplify this even further by using the **@DelegateToWidget** annotation for the shared state property, as described in Section 17.5.5, “Delegating State Properties to Widget”.

17.5.5. Delegating State Properties to Widget

The **@DelegateToWidget** annotation for a shared state property defines automatic delegation of the property value to the corresponding widget property of the same name and type, by calling the respective setter for the property in the widget.

```
public class MyComponentState extends AbstractComponentState {
    @DelegateToWidget
    public String text;
}
```

This is equivalent to handling the state change in the connector, as done in the example in Section 17.5.4, “Handling Property State Changes with **@OnStateChange**”.

If you want to delegate a shared state property to a widget property of another name, you can give the property name as a string parameter for the annotation.

```
public class MyComponentState extends AbstractComponentState {
    @DelegateToWidget("description")
    public String text;
}
```

17.5.6. Referring to Components in Shared State

While you can pass any regular Java objects through a shared state, referring to another component requires special handling because on the server-side you can only refer to a server-side component, while on the client-side you only have widgets. References to components can be made by referring to their connectors (all server-side components implement the `Connector` interface).

```
public class MyComponentState extends AbstractComponentState {
    public Connector otherComponent;
}
```

You could then access the component on the server-side as follows:

```
public class MyComponent {
    public void MyComponent(Component otherComponent) {
        getState().otherComponent = otherComponent;
    }

    public Component getOtherComponent() {
        return (Component) getState().otherComponent;
    }

    // And the cast method
    @Override
    public MyComponentState getState() {
        return (MyComponentState) super.getState();
    }
}
```

On the client-side, you should cast it in a similar fashion to a **ComponentConnector**, or possibly to the specific connector type if it is known.

17.5.7. Sharing Resources

Resources, which commonly are references to icons or other images, are another case of objects that require special handling. A `Resource` object exists only on the server-side and on the client-side you have an URL to the resource. You need to use the `setResource()` and `getResource()` on the server-side to access a resource, which is serialized to the client-side separately.

Let us begin with the server-side API:

```
public class MyComponent extends AbstractComponent {
    ...

    public void setMyIcon(Resource myIcon) {
        setResource("myIcon", myIcon);
    }

    public Resource getMyIcon() {
        return getResource("myIcon");
    }
}
```

On the client-side, you can then get the URL of the resource with `getResourceUrl()`.

```
@Override
public void onStateChanged(StateChangeEvent stateChangeEvent) {
    super.onStateChanged(stateChangeEvent);
    ...

    // Get the resource URL for the icon
    getWidget().setMyIcon(getResourceUrl("myIcon"));
}
```

The widget could then use the URL, for example, as follows:

```
public class MyWidget extends Label {
    ...

    Element imgElement = null;

    public void setMyIcon(String url) {
        if (imgElement == null) {
            imgElement = DOM.createImg();
            getElement().appendChild(imgElement);
        }

        DOM.setElementAttribute(imgElement, "src", url);
    }
}
```

17.6. RPC Calls Between Client- and Server-Side

Vaadin supports making Remote Procedure Calls (RPC) between a server-side component and its client-side widget counterpart. RPC calls are normally used for communicating stateless events, such as button clicks or other user interaction, in contrast to changing the shared state. Either party can make an RPC call to the other side. When a client-side widget makes a call, a server request is made. Calls made from the server-side to the client-side are communicated in the response of the server request during which the call was made.

If you use Eclipse and enable the "Full-Fledged" widget in the New Vaadin Widget wizard, it automatically creates a component with an RPC stub.

17.6.1. RPC Calls to the Server-Side

RPC calls from the client-side to the server-side are made through an RPC interface that extends the `ServerRpc` interface. A server RPC interface simply defines any methods that can be called through the interface.

For example:

```
public interface MyComponentServerRpc extends ServerRpc {
    public void clicked(String buttonName);
}
```

The above example defines a single `clicked()` RPC call, which takes a **String** object as the parameter.

You can pass the most common standard Java types, such as primitive and boxed primitive types, **String**, and arrays and some collections (**List**, **Set**, and **Map**) of the supported types. Also the Vaadin **Connector** and some special internal types can be passed.

An RPC method must return void - the widget set compiler should complain if it doesn't.

Making a Call

Before making a call, you need to instantiate the server RPC object with `RpcProxy.create()`. After that, you can make calls through the server RPC interface that you defined, for example as follows:

```
@Connect(MyComponent.class)
public class MyComponentConnector
    extends AbstractComponentConnector {

    public MyComponentConnector() {
        getWidget().addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                final MouseEventDetails mouseDetails =
                    MouseEventDetailsBuilder
                        .buildMouseEventDetails(
                            event.getNativeEvent(),
                            getWidget().getElement());
                MyComponentServerRpc rpc =
                    getRpcProxy(MyComponentServerRpc.class);

                // Make the call
                rpc.clicked(mouseDetails.getButtonName());
            }
        });
    }
}
```

Handling a Call

RPC calls are handled in a server-side implementation of the server RPC interface. The call and its parameters are serialized and passed to the server in an RPC request transparently.

```
public class MyComponent extends AbstractComponent {
    private MyComponentServerRpc rpc =
        new MyComponentServerRpc() {
            private int clickCount = 0;

            public void clicked(String buttonName) {
                Notification.show("Clicked " + buttonName);
            }
        };

    public MyComponent() {
        ...
        registerRpc(rpc);
    }
}
```

17.7. Component and UI Extensions

Adding features to existing components by extending them by inheritance creates a problem when you want to combine such features. For example, one add-on could add spell-check to a **TextField**, while another could add client-side validation. Combining such add-on features would be difficult if not impossible. You might also want to add a feature to several or even to all components, but extending all of them by inheritance is not really an option. Vaadin includes a component plug-in mechanism for these purposes. Such plug-ins are simply called *extensions*.

Also a UI can be extended in a similar fashion. In fact, some Vaadin features such as the JavaScript execution are UI extensions.

Implementing an extension requires defining a server-side extension class and a client-side connector. An extension can have a shared state with the connector and use RPC, just like a component could.

17.7.1. Server-Side Extension API

The server-side API for an extension consists of class that extends (in the Java sense) the **AbstractExtension** class. It typically has an *extend()* method, a constructor, or a static helper method that takes the extended component or UI as a parameter and passes it to *super.extend()*.

For example, let us have a trivial example with an extension that takes no special parameters, and illustrates the three alternative APIs:

```
public class CapsLockWarning extends AbstractExtension {
    // You could pass it in the constructor
    public CapsLockWarning(PasswordField field) {
        super.extend(field);
    }

    // Or in an extend() method
    public void extend(PasswordField field) {
        super.extend(field);
    }

    // Or with a static helper
    public static addTo(PasswordField field) {
        new CapsLockWarning().extend(field);
    }
}
```

The extension could then be added to a component as follows:

```
PasswordField password = new PasswordField("Give it");

// Use the constructor
new CapsLockWarning(password);

// ... or with the extend() method
new CapsLockWarning().extend(password);

// ... or with the static helper
CapsLockWarning.addTo(password);

layout.addComponent(password);
```

Adding a feature in such a "reverse" way is a bit unusual in the Vaadin API, but allows type safety for extensions, as the method can limit the target type to which the extension can be applied, and whether it is a regular component or a UI.

17.7.2. Extension Connectors

An extension does not have a corresponding widget on the client-side, but only an extension connector that extends the **AbstractExtensionConnector** class. The server-side extension class is specified with a `@Connect` annotation, just like in component connectors.

An extension connector needs to implement the `extend()` method, which allows hooking to the extended component. The normal extension mechanism is to modify the extended component as needed and add event handlers to it to handle user interaction. An extension connector can share a state with the server-side extension as well as make RPC calls, just like with components.

In the following example, we implement a "Caps Lock warning" extension. It listens for changes in Caps Lock state and displays a floating warning element over the extended component if the Caps Lock is on.

```
@Connect(CapsLockWarning.class)
public class CapsLockWarningConnector
    extends AbstractExtensionConnector {

    @Override
    protected void extend(ServerConnector target) {
        // Get the extended widget
        final Widget pw =
            ((ComponentConnector) target).getWidget();

        // Preparations for the added feature
        final VOverlay warning = new VOverlay();
        warning.setOwner(pw);
        warning.add(new HTML("Caps Lock is enabled!"));

        // Add an event handler
        pw.addDomHandler(new KeyPressHandler() {
            public void onKeyPress(KeyPressEvent event) {
                if (isEnabled() && isCapsLockOn(event)) {
                    warning.showRelativeTo(passwordWidget);
                } else {
                    warning.hide();
                }
            }
        }, Vaadin.EventListeners.CLICK);
    }
}
```

```
    }, KeyPressEvent.getType());
}

private boolean isCapsLockOn(KeyPressEvent e) {
    return e.isShiftKeyDown() ^
        Character.isUpperCase(e.getCharCode());
}
}
```

The `extend()` method gets the connector of the extended component as the parameter, in the above example a **PasswordFieldConnector**. It can access the widget with the `getWidget()`.

An extension connector needs to be included in a widget set. The class must therefore be defined under the `client` package of a widget set, just like with component connectors.

17.8. Styling a Widget

To make your widget look stylish, you need to style it. There are two basic ways to define CSS styles for a component: in the widget sources and in a theme. A default style should be defined in the widget sources, and different themes can then modify the style.

17.8.1. Determining the CSS Class

The CSS class of a widget element is normally defined in the widget class and set with `setStyleName()`. A widget should set the styles for its sub-elements as it desires.

For example, you could style a composite widget with an overall style and with separate styles for the sub-widgets as follows:

```
public class MyPickerWidget extends ComplexPanel {
    public static final String CLASSNAME = "mypicker";

    private final TextBox textBox = new TextBox();
    private final PushButton button = new PushButton("...");

    public MyPickerWidget() {
        setElement(Document.get().createDivElement());
        setStylePrimaryName(CLASSNAME);

        textBox.setStylePrimaryName(CLASSNAME + "-field");
        button.setStylePrimaryName(CLASSNAME + "-button");

        add(textBox, getElement());
        add(button, getElement());

        button.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                Window.alert("Calendar picker not yet supported!");
            }
        });
    }
}
```

In addition, all Vaadin components get the `v-widget` class. If it extends an existing Vaadin or GWT widget, it will inherit CSS classes from that as well.

17.8.2. Default Stylesheet

A client-side module, which is normally a widget set, can include stylesheets. They must be placed under the `public` folder under the folder of the widget set, as described in Section 14.3.1, “Specifying a Stylesheet”.

For example, you could style the widget described above as follows:

```
.mypicker {
  white-space: nowrap;
}

.mypicker-button {
  display: inline-block;
  border: 1px solid black;
  padding: 3px;
  width: 15px;
  text-align: center;
}
```

Notice that some size settings may require more complex handling and calculating the sizes dynamically.

17.9. Component Containers

Component containers, such as layout components, are a special group of components that require some consideration. In addition to handling state, they need to manage communicating the hierarchy of their contained components to the other side.

The easiest way to implement a component container is extend the **AbstractComponentContainer**, which handles the synchronization of the container server-side components to the client-side.

17.10. Advanced Client-Side Topics

In the following, we mention some topics that you may encounter when integrating widgets.

17.10.1. Client-Side Processing Phases

Vaadin’s client-side engine reacts to changes from the server in a number of phases, the order of which can be relevant for a connector. The processing occurs in the `handleUIDLMessage()` method in **ApplicationConnection**, but the logic can be quite overwhelming, so we describe the phases in the following summary.

1. Any dependencies defined by using **@JavaScript** or **@StyleSheet** on the server-side class are loaded. Processing does not continue until the browser confirms that they have been loaded.
2. New connectors are instantiated and `init()` is run for each `Connector`.
3. State objects are updated, but no state change event is fired yet.
4. The connector hierarchy is updated, but no hierarchy change event is fired yet. `setParent()` and `setChildren()` are run in this phase.

5. Hierarchy change events are fired. This means that all state objects and the entire hierarchy are already up to date when this happens. The DOM hierarchy should in theory be up to date after all hierarchy events have been handled, although there are some built-in components that for various reasons do not always live up to this promise.
6. Captions are updated, causing `updateCaption()` to be invoked on layouts as needed.
7. **@DelegateToWidget** is handled for all changed state objects using the annotation.
8. State change events are fired for all changed state objects.
9. `updateFromUIDL()` is called for legacy connectors.
10. All RPC methods received from the server are invoked.
11. Connectors that are no longer included in the hierarchy are unregistered. This calls `onUnregister()` on the Connector.
12. The layout phase starts, first checking the sizes and positions of all elements, and then notifying any `ElementResizeListener`s, as well as calling the appropriate layout method for the connectors that implement either `[classname]#SimpleManagedLayout` or **DirectionalManagedLayout** interface.

17.11. Creating Add-ons

Add-ons are the most convenient way to reuse Vaadin code, either commercially or free. Vaadin Directory serves as the store for the add-ons. You can distribute add-ons both as JAR libraries and Zip packages.

Creating a typical add-on package involves the following tasks:

- Compile server-side classes
- Compile JavaDoc (optional)
- Build the JAR
 - Include Vaadin add-on manifest
 - Include the compiled server-side classes
 - Include the compiled JavaDoc (optional)
 - Include sources of client-side classes for widget set compilation (optional)
 - Include any JavaScript dependency libraries (optional)
 - Exclude any test or demo code in the project

The exact contents depend on the add-on type. Component add-ons often include a widget set, but not always, such as JavaScript components or pure server-side components. You can also have data container and theme add-ons, as well as various tools.

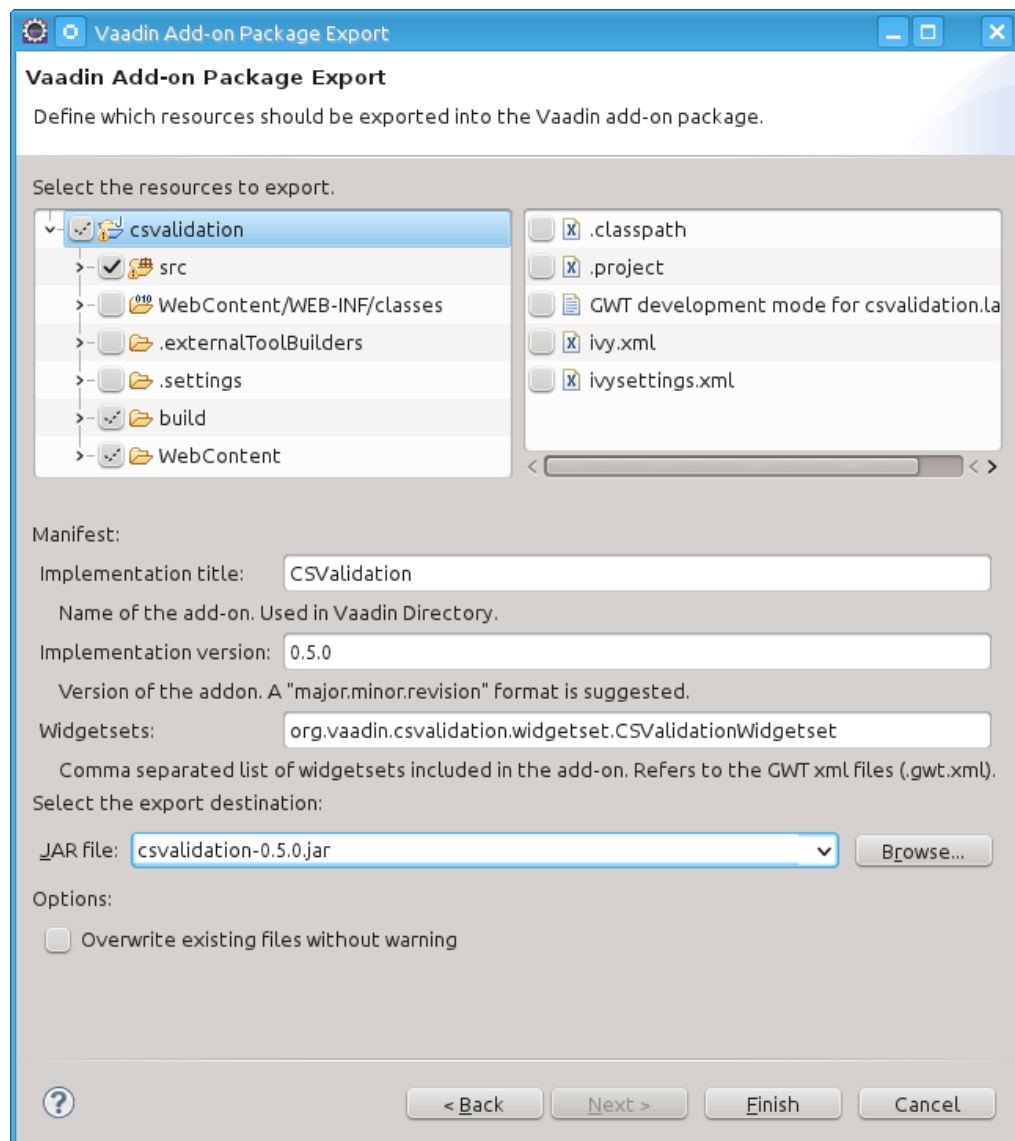
It is common to distribute the JavaDoc in a separate JAR, but you can also include it in the same JAR.

17.11.1. Exporting Add-on in Eclipse

If you use the Vaadin Plugin for Eclipse for your add-on project, you can simply export the add-on from Eclipse.

1. Select the project and then **File** → **Export** from the menu
2. In the export wizard that opens, select **Vaadin** → **Vaadin Add-on Package**, and click **Next**
3. In the **Select the resources to export** panel, select the content that should be included in the add-on package. In general, you should include sources in `src` folder (at least for the client-side package), compiled server-side classes, themes in `WebContent/VAADIN/themes`. These are all included automatically. You probably want to leave out any demo or example code.

Figure 17.5. Exporting a Vaadin Add-on



If you are submitting the add-on to Vaadin Directory, the **Implementation title** should be exactly the name of the add-on in Directory. The name may contain spaces and most other letters. Notice that *it is not possible to change the name later*.

The **Implementation version** is the version of your add-on. Typically experimental or beta releases start from 0.1.0, and stable releases from 1.0.0.

The **Widgetsets** field should list the widget sets included in the add-on, separated by commas. The widget sets should be listed by their class name, that is, without the `.gwt.xml` extension.

The **JAR file** is the file name of the exported JAR file. It should normally include the version number of the add-on. You should follow the Maven format for the name, such as `myaddon-1.0.0.jar`.

Finally, click **Finish**.

17.12. Migrating from Vaadin 6

The client-side architecture was redesigned almost entirely in Vaadin 7. In Vaadin 6, state synchronization was done explicitly by serializing and deserializing the state on the server- and client-side. In Vaadin 7, the serialization is handled automatically by the framework using state objects.

In Vaadin 6, a server-side component serialized its state to the client-side using the `Paintable` interface in the client-side and deserialized the state through the `VariableOwner` interface. In Vaadin 7, these are done through the `ClientConnector` interface.

On the client-side, a widget deserialized its state through the `Paintable` interface and sent state changes through the `ApplicationConnection` object. In Vaadin 7, these are replaced with the `ServerConnector`.

In addition to state synchronization, Vaadin 7 has an RPC mechanism that can be used for communicating events. They are especially useful for events that are not associated with a state change, such as a button click.

The framework ensures that the connector hierarchy and states are up-to-date when listeners are called.

17.12.1. Quick (and Dirty) Migration

Vaadin 7 has a compatibility layer that allows quick conversion of a widget.

1. Create a connector class, such as **MyConnector**, that extends **LegacyConnector**. Implement the `getWidget()` method.
2. Move the `@ClientWidget(MyWidget.class)` from the server-side component, say **MyComponent**, to the **MyConnector** class and make it `@Connect(MyComponent.class)`.
3. Have the server-side component implement the `LegacyComponent` interface to enable compatibility handling.
4. Remove any calls to `super.paintContent()`

5. Update any imports on the client-side

17.13. Integrating JavaScript Components and Extensions

Vaadin allows simplified integration of pure JavaScript components, as well as component and UI extensions. The JavaScript connector code is published from the server-side. As the JavaScript integration does not involve GWT programming, no widget set compilation is needed.

17.13.1. Example JavaScript Library

There are many kinds of component libraries for JavaScript. In the following, we present a simple library that provides one object-oriented JavaScript component. We use this example later to show how to integrate it with a server-side Vaadin component.

The example library includes a single **MyComponent** component, defined in `mylibrary.js`.

```
// Define the namespace
var mylibrary = mylibrary || {};

mylibrary.MyComponent = function (element) {
  element.innerHTML =
    "<div class='caption'>Hello, world!</div>" +
    "<div class='textinput'>Enter a value: " +
    "<input type='text' name='value'>" +
    "<input type='button' value='Click'>" +
    "</div>";

  // Style it
  element.style.border = "thin solid red";
  element.style.display = "inline-block";

  // Getter and setter for the value property
  this.getValue = function () {
    return element.
      getElementsByTagName("input")[0].value;
  };
  this.setValue = function (value) {
    element.getElementsByTagName("input")[0].value =
      value;
  };

  // Default implementation of the click handler
  this.click = function () {
    alert("Error: Must implement click() method");
  };

  // Set up button click
  var button = element.getElementsByTagName("input")[1];
  var self = this; // Can't use this inside the function
  button.onclick = function () {
    self.click();
  };
};
```

When used in an HTML page, the library would be included with the following definition:

```
<script type="text/javascript"
  src="mylibrary.js"></script>
```

You could then use it anywhere in the HTML document as follows:

```
<!-- Placeholder for the component -->
<div id="foo"></div>

<!-- Create the component and bind it to the placeholder -->
<script type="text/javascript">
    window.foo = new mylibrary.MyComponent(
        document.getElementById("foo"));
    window.foo.click = function () {
        alert("Value is " + this.getValue());
    }
</script>
```

Figure 17.6. A JavaScript Component Example



You could interact with the component with JavaScript for example as follows:

```
<a href="javascript:foo.setValue('New value')">Click here</a>
```

17.13.2. A Server-Side API for a JavaScript Component

To begin integrating such a JavaScript component, you would need to sketch a bit how it would be used from a server-side Vaadin application. The component should support writing the value as well as listening for changes to it.

```
final MyComponent mycomponent = new MyComponent();

// Set the value from server-side
mycomponent.setValue("Server-side value");

// Process a value input by the user from the client-side
mycomponent.addValueChangeListener(
    new MyComponent.ValueChangeListener() {
        @Override
        public void valueChange() {
            Notification.show("Value: " + mycomponent.getValue());
        }
    });

layout.addComponent(mycomponent);
```

Basic Server-Side Component

A JavaScript component extends the **AbstractJavaScriptComponent**, which handles the shared state and RPC for the component.

```
package com.vaadin.book.examples.client.js;

@JavaScript({"mylibrary.js", "mycomponent-connector.js"})
public class MyComponent extends AbstractJavaScriptComponent {
    public interface ValueChangeListener extends Serializable {
        void valueChange();
    }
}
```

```
ArrayList<ValueChangeListener> listeners =
    new ArrayList<ValueChangeListener>();
public void addValueChangeListener(
    ValueChangeListener listener) {
    listeners.add(listener);
}

public void setValue(String value) {
    getState().value = value;
}

public String getValue() {
    return getState().value;
}

@Override
protected MyComponentState getState() {
    return (MyComponentState) super.getState();
}
}
```

Notice later when creating the JavaScript connector that its name must match the package name of this server-side class.

The shared state of the component is as follows:

```
public class MyComponentState extends JavaScriptComponentState {
    public String value;
}
```

If the member variables are private, you need to have public setters and getters for them, which you can use in the component.

17.13.3. Defining a JavaScript Connector

A JavaScript connector is a function that initializes the JavaScript component and handles communication between the server-side and the JavaScript code.//TOD Clarify - code?

A connector is defined as a connector initializer function that is added to the `window` object. The name of the function must match the server-side class name, with the full package path. Instead of the Java dot notation for the package name, underscores need to be used as separators.

The Vaadin client-side framework adds a number of methods to the connector function. The `this.getElement()` method returns the HTML DOM element of the component. The `this.getState()` returns a shared state object with the current state as synchronized from the server-side.

```
window.com_vaadin_book_examples_client_js_MyComponent =
function() {
    // Create the component
    var mycomponent =
        new mylibrary.MyComponent(this.getElement());

    // Handle changes from the server-side
    this.onStateChange = function() {
        mycomponent.setValue(this.getState().value);
    };
};
```

```
// Pass user interaction to the server-side
var self = this;
mycomponent.click = function() {
    self.onClick(mycomponent.getValue());
};
};
```

In the above example, we pass user interaction using the JavaScript RPC mechanism, as described in the next section.

17.13.4. RPC from JavaScript to Server-Side

User interaction with the JavaScript component has to be passed to the server-side using an RPC (Remote Procedure Call) mechanism. The JavaScript RPC mechanism is almost equal to regular client-side widgets, as described in Section 17.6, “RPC Calls Between Client- and Server-Side”.

Handling RPC Calls on the Server-Side

Let us begin with the RPC function registration on the server-side. RPC calls are handled on the server-side in function handlers that implement the `JavaScriptFunction` interface. A server-side function handler is registered with the `addFunction()` method in **AbstractJavaScriptComponent**. The server-side registration actually defines a JavaScript method that is available in the client-side connector object.

Continuing from the server-side **MyComponent** example we defined earlier, we add a constructor to it that registers the function.

```
public MyComponent() {
    addFunction("onClick", new JavaScriptFunction() {
        @Override
        public void call(JsonArray arguments) {
            getState().setValue(arguments.getString(0));
            for (ValueChangeListener listener: listeners)
                listener.valueChange();
        }
    });
}
```

Making an RPC Call from JavaScript

An RPC call is made simply by calling the RPC method in the connector. In the constructor function of the JavaScript connector, you could write as follows (the complete connector code was given earlier):

```
window.com_vaadin_book_examples_gwt_js_MyComponent =
function() {
    ...
    var connector = this;
    mycomponent.click = function() {
        connector.onClick(mycomponent.getValue());
    };
};
```

Here, the `mycomponent.click` is a function in the example JavaScript library, as described in Section 17.13.1, “Example JavaScript Library”. The `onClick()` is the method we defined on the server-side. We pass a simple string parameter in the call.

You can pass anything that is valid in JSON notation in the parameters.

Chapter 18

Using Vaadin Add-ons

18.1. Overview	491
18.2. Downloading Add-ons from Vaadin Directory	492
18.3. Installing Add-ons in Eclipse with Ivy	493
18.4. Using Add-ons in a Maven Project	494
18.5. Installing Commercial Vaadin Add-on Licence	497
18.6. Troubleshooting	500

This chapter describes the installation of add-on components, themes, containers, and other tools from the Vaadin Directory and the use of commercial add-ons offered by Vaadin.

18.1. Overview

In addition to the components, layouts, themes, and data sources built in into the core Vaadin library, many others are available as add-ons. Vaadin Directory provides a rich collection of add-ons for Vaadin, and you may find others from independent sources. Add-ons are also one way to share your own components between projects.

Installation of add-ons from Vaadin Directory is simple, just adding an Ivy or Maven dependency, or downloading the JAR package and dropping it in the web library folder of the project. Most add-ons include a widget set, which you need to compile, but it's usually just a click of a button or a single command.

After trying out an add-on, you can give some feedback to the author of the add-on by rating the add-on with one to five stars and optionally leaving a comment. Most add-ons also have a discussion forum thread for user feedback and questions.

Add-ons available from Vaadin Directory are distributed under different licenses, of which some are commercial. While the add-ons can be downloaded directly, you should note their license and other terms and conditions. Many are offered under a dual licensing agreement so that they can be used in open source projects for free, and many have a trial period for closed-source development. Commercial Vaadin add-ons distributed under the CVAL license require installing a license key as instructed in Section 18.5, “Installing Commercial Vaadin Add-on Licence”.

18.2. Downloading Add-ons from Vaadin Directory

If you are not using Maven or a Maven-compatible dependency manager such as Ivy, or want to manage for your libraries manually, you can download add-on packages from the details page of an add-on in Vaadin Directory.

1. Select the version; some add-ons have several versions available. The latest is shown by default, but you can choose another the version to download from the dropdown menu in the header of the details page.
2. Click **Download Now** and save the JAR or Zip file on your computer.
3. If the add-on is packaged in a Zip package, unzip the package and follow any instructions provided inside the package. Typically, you just need to copy a JAR file to your web project under the `WEB-INF/lib` directory.

Note that some add-ons may require other libraries. You can resolve such dependencies manually, but we recommend using a dependency manager such as Ivy or Maven in your project.

4. Update and recompile your project. In Eclipse, select the project and press F5.
 5. You may need to compile the client-side implementations of the add-on components, that is, a *widget set*. This is the case for majority of add-ons, except for pure server-side, theme, or data binding add-ons. Compiling the widget set depends on the build environment. See Section 18.2.1, “Compiling Widget Sets with an Ant Script”, or later in this chapter for instructions for compiling the widget set with Eclipse and Maven.
1. Update the project in your development web server and possibly restart the server.

18.2.1. Compiling Widget Sets with an Ant Script

If you need to compile the widget set with an Ant script, you can find a script template package at the Vaadin download page. You can copy the files in the package to your project and, once configured, use it by running Ant in the directory.

If you are using an IDE such as Eclipse, *always* remember to refresh the project to synchronize it with the filesystem after compiling the widget set outside the IDE.

18.3. Installing Add-ons in Eclipse with Ivy

The Vaadin Plugin for Eclipse uses Apache Ivy to resolve dependencies. The dependencies should be listed in the `ivy.xml` file in the project root. The Vaadin Directory allows downloading add-ons from a Maven repository, which can be accessed also by Ivy.

You can also use Ivy to resolve dependencies in an Ant script.

1. Open the add-on page in Vaadin Directory.
2. Select the version. The latest is shown by default, but you can choose another the version from the dropdown menu in the header of the add-on details page.
3. Click the **Maven/Ivy** to display the Ivy dependency declaration, as illustrated in Figure 18.1, "Ivy Dependency Declaration". If the add-on is available with multiple licenses, you will be prompted to select a license for the dependency.

Figure 18.1. Ivy Dependency Declaration



4. Open the `ivysettings.xml` in your Eclipse project either in the XML or Ivy Editor (either double-click the file or right-click it and select **Open With** → **Ivy Editor**).

Check that the settings file has the `<ibiblio>` entry given in the Directory page. It should be, if the file was created by the Vaadin project wizard in Eclipse. If not, copy it there.

```
<chain name="default">
  ...
  <ibiblio name="vaadin-addons"
    usepoms="true"
    m2compatible="true"
    root="http://maven.vaadin.com/vaadin-addons"/>
```

```
...
</chain>
```

If you get Vaadin addons from another repository, such as the local repository if you have compiled them yourself, you need to define a resolver for the repository in the settings file.

5. Open the `ivy.xml` in your Eclipse project and copy the Ivy dependency to inside the `dependencies` element. It should be as follows:

```
<dependencies>
...
  <dependency org="com.vaadin.addon"
              name="vaadin-charts"
              rev="1.0.0"/>
</dependencies>
```

You can specify either a fixed version number or a dynamic revision tag, such as `latest.release`. You can find more information about the dependency declarations in Ivy documentation.

If the `ivy.xml` does not have a `<configurations defaultconfmapping="default->default">` defined, you also need to have `conf="default->default"` in the dependency to resolve transient dependencies correctly.

IvyIDE immediately resolves the dependencies when you save the file.

6. Compile the add-on widget set

by clicking the **Compile Vaadin widgets** button in the toolbar.

+

Figure 18.2. Compiling Widget Set in Eclipse



If you experience problems with Ivy, first check all the dependency parameters. IvyDE can sometimes cause unexpected problems. You can clear the Ivy dependency cache by right-clicking the project and selecting **Ivy** → **Clean all caches**. To refresh Ivy configuration, select **Ivy** → **Refresh**. To try resolving again Ivy, select **Ivy** → **Resolve**.

18.4. Using Add-ons in a Maven Project

To use add-ons in a Maven project, you simply have to add them as dependencies in the project POM. Most add-ons include a widget set, which are compiled to the project widget set.

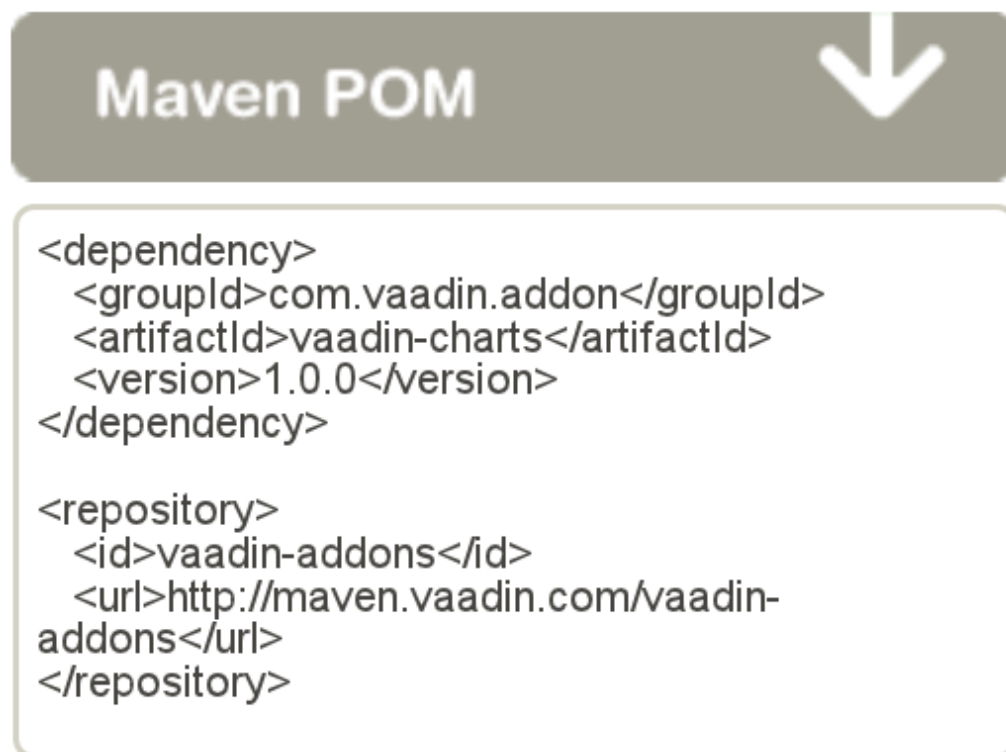
Creating, compiling, and packaging a Vaadin project with Maven was described in Section 3.5, “Creating a Project with Maven”.

18.4.1. Adding a Dependency

Vaadin Directory provides a Maven repository for all the add-ons in the Directory.

1. Open the add-on page in Vaadin Directory.
2. Select the version. The latest is shown by default, but you can choose another the version from the dropdown menu in the header of the add-on details page.
3. Click the **Maven/Ivy** to display the Maven dependency declaration, as illustrated in Figure Figure 18.3, “Maven POM Definitions”. If the add-on is available with multiple licenses, you will be prompted to select a license for the dependency.

Figure 18.3. Maven POM Definitions



4. Copy the dependency declaration to the `pom.xml` file in your project, under the `dependencies` element.

```
...
<dependencies>
  ...
  <dependency>
    <groupId>com.vaadin.addon</groupId>
    <artifactId>vaadin-charts</artifactId>
    <version>1.0.0</version>
  </dependency>
</dependencies>
```

You can use an exact version number, as is done in the example above, or `LATEST` to always use the latest version of the add-on.

The POM excerpt given in Directory includes also a repository definition, but if you have used the `vaadin-archetype-application` to create your project, it already includes the definition.

5. Compile the widget set as described in the following section.

18.4.2. Compiling the Project Widget Set

If you have used the `vaadin-archetype-application` to create the project, the `pom.xml` includes all necessary declarations to compile the widget set. The widget set compilation occurs in standard Maven build phase, such as with `package` or `install` goal.

```
$ mvn package
```

Then, just deploy the WAR to your application server.

Recompiling the Widget Set

The Vaadin plugin for Maven tries to avoid recompiling the widget set unless necessary, which sometimes means that it is not compiled even when it should. Running the `clean` goal usually helps, but causes a full recompilation. You can compile the widget set manually by running the `vaadin:compile` goal.

```
$ mvn vaadin:compile
```

Note that this does not update the project widget set by searching new widget sets from the class path. It must be updated if you add or remove add-ons, for example. You can do that by running the `vaadin:update-widgetset` goal in the project directory.

```
$ mvn vaadin:update-widgetset
...
[INFO] auto discovered modules [your.company.gwt.ProjectNameWidgetSet]
[INFO] Updating widgetset your.company.gwt.ProjectNameWidgetSet
[ERROR] 27.10.2011 19:22:34 com.vaadin.terminal.gwt.widgetsetutils.ClassPathEx-
plorer getAvailableWidgetSets
[ERROR] INFO: Widgetsets found from classpath:
...
```

Do not mind the "ERROR" labels, they are just an issue with the Vaadin Plugin for Maven.

After running the update, you need to run the `vaadin:compile` goal to actually compile the widget set.

18.4.3. Enabling Widget Set Compilation

If you are not using a POM created with the proper Vaadin archetype, you may need to enable widget set compilation manually. The simplest way to do that is to copy the definitions from a POM created with the archetype. Specifically, you need to copy the `plugin` definitions. You also need the Vaadin dependencies.

You need to create an empty widget set definition file, which the widget set compilation will populate with widget sets found from the class path. Create a `src/main/java/com/example/AppWidgetSet.gwt.xml` file (in the project package) with an empty `<module>` element as follows:

```
<module>
</module>
```

Enabling the Widget Set in the UI

If you have previously used the default widget set in the project, you need to enable the project widget set in the `web.xml` deployment descriptor. Edit the `src/main/webapp/WEB-INF/web.xml` file and add or modify the `widgetset` parameter for the servlet as follows.

```
<servlet>
  ...
  <init-param>
    <description>Widget Set to Use</description>
    <param-name>widgetset</param-name>
    <param-value>com.example.AppWidgetSet</param-value>
  </init-param>
</servlet>
```

The parameter is the class name of the widget set, that is, without the `.gwt.xml` extension and with the Java dot notation for class names that include the package name.

18.5. Installing Commercial Vaadin Add-on Licence

The commercial Vaadin add-ons require installing a license key before using them. The license keys are development licenses and checked during widget set compilation, or in Vaadin Test-Bench when executing tests, so you do not need them when deploying the application.

18.5.1. Obtaining License Keys

You can purchase add-ons or obtain a free trial key from the Vaadin website. You need to register in the website to obtain a key.

You can get license keys from vaadin.com/pro/licenses, where you can navigate by selecting in the Vaadin website **My Account** → **Licenses** or directly Licenses if you are a Pro Tools subscriber.

Figure 18.4. Obtaining CVAL License

[My Account](#)
[Certification](#)
[Licenses](#)
[Support](#)
[Bugfix](#)
[Feature Votes](#)

Licenses

Vaadin Framework

67

Select which version of Vaadin Framework are you using.

Subscription Licenses (CVAL)

Official Vaadin add-ons are licensed under the [Commercial Vaadin Add-on License](#).
Get your subscription licenses below.

	License key
 Spreadsheet 1.0.0.beta1 Spreadsheet component for business applications	
 TestBench 4.0.2 An environment for automated user interface regression testing of Vaadin applications on multiple platforms and browsers	
 Charts 2.0.0 The most comprehensive visualization library available for Vaadin.	
 TouchKit 4.0.0 Build touch-enabled applications for iOS and Android mobile devices using Vaadin	

Click on a license key to obtain the purchased or trial key.

Figure 18.5. Obtaining CVAL License Key


Spreadsheet 1.0.0.beta1
 Spreadsheet component for business applications



L1cen5e-c0de
[Download](#)

Copy-paste the license key to a file called `vaadin.spreadsheet.developer.license` or download it directly.
After downloading you need to copy the file to your home directory (`/home/<username>/`).

18.5.2. Installing License Key in License File

To install the license key in a development workstation, you can copy and paste it verbatim to a file in your home directory.

License for each product has a separate license file as follows:

```
Vaadin Charts
    .vaadin.charts.developer.license

Vaadin Spreadsheet
    .vaadin.spreadsheet.developer.license

Vaadin TestBench
    .vaadin.testbench.developer.license

Vaadin TouchKit
    .vaadin.touchkit.developer.license
```

For example, in Linux and OS X:

```
$ echo "L1cen5e-c0de" > ~/.vaadin.#<product>#.developer.license
```

18.5.3. Passing License Key as System Property

You can also pass the key as a system property to the widget set compiler, usually with a `-D` option. For example, on the command-line:

```
$ java -D[parameter]vaadin.#<product>#.developer.license=L1cen5e-c0de ...
```

Passing License Key in Different Environments

How you actually pass the parameter to the widget set compiler depends on the development environment and the build system that you use to compile the widget set. Below are listed a few typical environments:

Eclipse IDE

To install the license key for all projects, select **Window** → **Preferences** and navigate to the **Java** → **Installed JREs** section. Select the JRE version that you use for the application and click **Edit**. In the **Default VM arguments**, give the `-D` expression as shown above.

Apache Ant

If compiling the project with Apache Ant, you could set the key in the Ant script as follows:

```
<sysproperty key="vaadin.<product>.developer.license"
    value="L1cen5e-c0de"/>
```

However, you should never store license keys in a source repository, so if the Ant script is stored in a source repository, you should pass the license key to Ant as a property that you then use in the script for the value argument of the `<sysproperty>` as follows:

```
<sysproperty key="vaadin.<product>.developer.license"
    value="${vaadin.<product>.developer.license}"/>
```

When invoking Ant from the command-line, you can pass the property with a `-D` parameter to Ant.

Apache Maven

If building the project with Apache Maven, you can pass the license key with a `-D` parameter to Maven:

```
$ mvn -D[parameter]vaadin.#{product}#.developer.license=L1cen5e-c0de
package
```

Continuous Integration Systems

In CIS systems, you can pass the license key to build runners as a system property in the build configuration. However, this only passes it to a runner. As described above, Ant does not pass it to sub-processes implicitly, so you need to forward it explicitly as described earlier.

18.6. Troubleshooting

If you experience problems with using add-ons, you can try the following:

- Check the `.gwt.xml` descriptor file under the the project root package. For example, if the project root package is `com.example.myproject`, the widget set definition file is typically at `com/example/project/AppWidgetset.gwt.xml`. The location is not fixed and it can be elsewhere, as long as references to it match. See Section 14.3, “Client-Side Module Descriptor” for details on the contents of the client-side module descriptor, which is used to define a widget set.
- Check the `WEB-INF/web.xml` deployment descriptor and see that the servlet for your UI has a widget set parameter, such as the following:

```
<init-param>
  <description>UI widgetset</description>
  <param-name>widgetset</param-name>
  <param-value>com.example.project.AppWidgetSet</param-value>
</init-param>
```

Check that the widget set class corresponds with the `.gwt.xml` file in the source tree.

- See the `VAADIN/widgetsets` directory and check that the widget set appears there. You can remove it and recompile the widget set to see that the compilation works properly.
- Use the **Net** tab in Firebug to check that the widget set (and theme) is loaded properly.
- Use the `?debug` parameter for the application to open the debug window and check if there is any version conflict between the widget set and the Vaadin library, or the themes. See Section 12.3, “Debug Mode and Window” for details.
- Refresh and recompile the project. In Eclipse, select the project and press F5, stop the server, clean the server temporary directories, and restart it.
- Check the Error Log view in Eclipse (or in the IDE you use).

Chapter 19

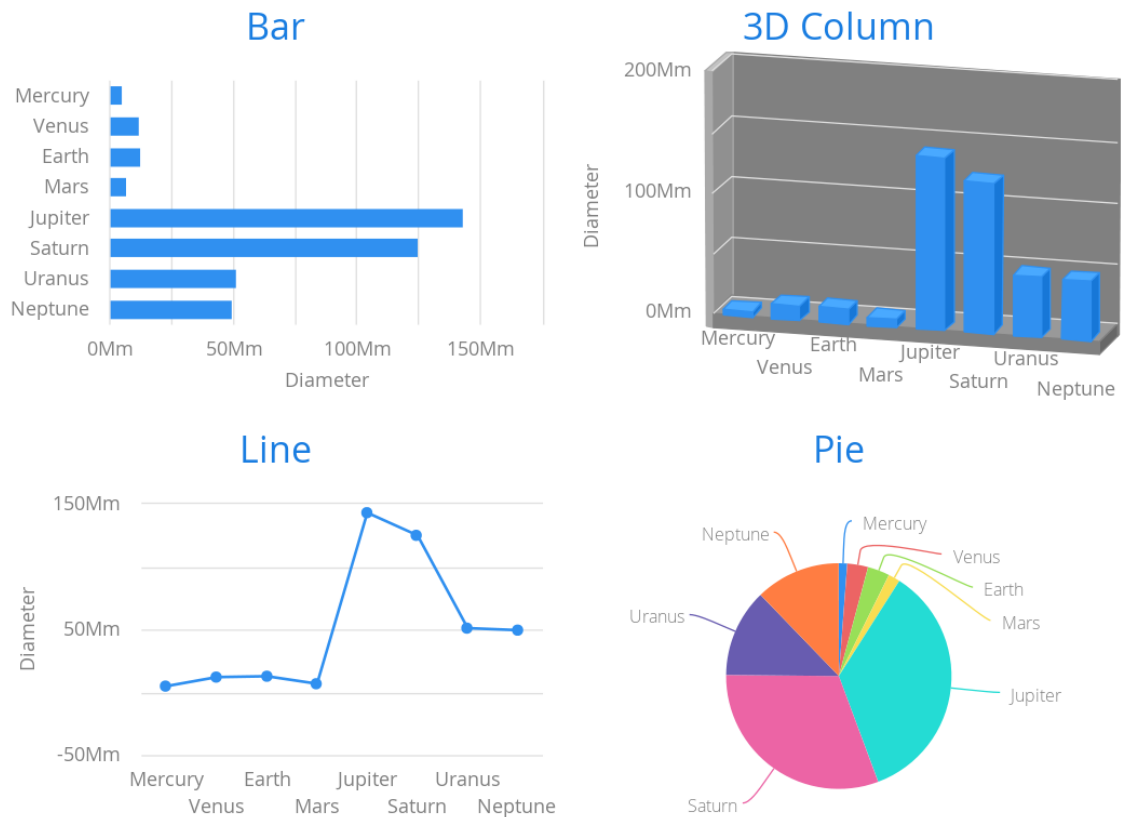
Vaadin Charts

19.1. Overview	501
19.2. Installing Vaadin Charts	504
19.3. Basic Use	506
19.4. Chart Types	515
19.5. Chart Configuration	535
19.6. Chart Data	543
19.7. Advanced Uses	549
19.8. Timeline	550

This chapter provides the documentation for the Vaadin Charts add-on.

19.1. Overview

Vaadin Charts is a feature-rich interactive charting library for Vaadin. It provides a **Chart** component. The **Chart** can visualize one- and two-dimensional numeric data in many available chart types. The charts allow flexible configuration of all the chart elements as well as the visual style. The library includes a number of built-in visual themes, which you can extend further. The basic functionalities allow the user to interact with the chart elements in various ways, and you can define custom interaction with click events.

Figure 19.1. Vaadin Charts

The data displayed in a chart can be one- or two dimensional tabular data, or scatter data with free X and Y values. Data displayed in range charts has minimum and maximum values instead of singular values.

This chapter covers the basic use of Vaadin Charts and the chart configuration. For detailed documentation of the configuration parameters and classes, please refer to the JavaDoc API documentation of the library.

In the following basic examples, which we study further in Section 19.3, “Basic Use” and Section 19.3, “Basic Use”, we demonstrate how to display one-dimensional data in a column graph and customize the X and Y axis labels and titles.

Java:

```
Chart chart = new Chart(ChartType.BAR);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Planets");
conf.setSubTitle("The bigger they are the harder they pull");
conf.getLegend().setEnabled(false); // Disable legend

// The data
ListSeries series = new ListSeries("Diameter");
series.setData(4900, 12100, 12800,
```

```

        6800, 143000, 125000,
        51100, 49500);
conf.addSeries(series);

// Set the category labels on the axis correspondingly
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                  "Mars", "Jupiter", "Saturn",
                  "Uranus", "Neptune");
xaxis.setTitle("Planet");
conf.addXAxis(xaxis);

// Set the Y axis title
YAxis yaxis = new YAxis();
yaxis.setTitle("Diameter");
yaxis.getLabels().setFormatter(
    "function() {return Math.floor(this.value/1000) + \" Mm\";}");
yaxis.getLabels().setStep(2);
conf.addYAxis(yaxis);

layout.addComponent(chart);

```

Web Components:

```

<!DOCTYPE html>
<html>
<head>
  <script src="bower_components/webcomponentsjs/webcomponents-
lite.min.js"></script>
  <link rel="import" href="bower_components/vaadin-charts/vaadin-bar-chart.html">
</head>
<body>
  <div style="width:400px;height:300px">
    <vaadin-bar-chart>
      <chart-title>Planets</chart-title>
      <subtitle>The bigger they are the harder they pull</subtitle>
      <legend>
        <enabled>false</enabled>
      </legend>
      <x-axis>
        <chart-title>Planet</chart-title>
        <categories>Mercury, Venus, Earth, Mars,
                    Jupiter, Saturn, Uranus, Neptune</categories>
      </x-axis>
      <y-axis>
        <chart-title>Diameter</chart-title>
        <labels formatter = "function () { return this.value / 1000 + 'Mm';}">

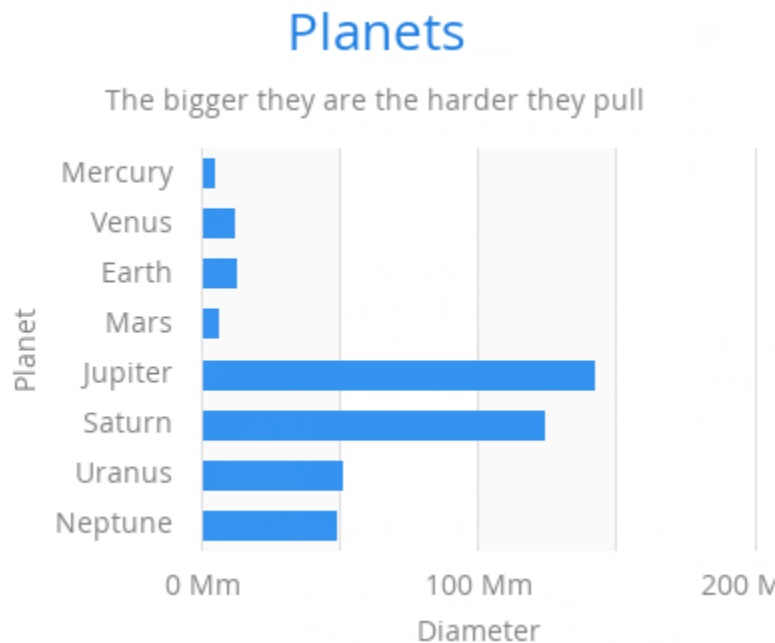
          <step>2</step>
        </labels>
      </y-axis>
      <data-series name="Diameter">
        <data>
          4900, 12100, 12800, 6800,
          143000, 125000, 51100, 49500
        </data>
      </data-series>
    </vaadin-bar-chart>
  </div>

```

```
</body>
</html>
```

The resulting chart is shown in Figure 19.2, “Basic Chart Example”.

Figure 19.2. Basic Chart Example



19.1.1. Licensing

Vaadin Charts is a commercial product licensed under the CVAL License (Commercial Vaadin Add-On License). A license needs to be purchased for all use, including web deployments as well as intranet use.

The commercial licenses can be purchased from the Vaadin Directory, where you can also find the license details and download the Vaadin Charts.

19.2. Installing Vaadin Charts

As with most Vaadin add-ons, you can install Vaadin Charts as a Maven or Ivy dependency in your project, or from an installation package. For general instructions on installing add-ons, please see Chapter 18, *Using Vaadin Add-ons*.

Vaadin Charts requires Vaadin 7.4 or later.

Using Vaadin Charts requires a license key, which you must install before compiling the widget set. The widget set must be compiled after setting up the dependency or library JARs.

19.2.1. Maven Dependency

The Maven dependency for Vaadin Charts is as follows:

```
<dependency>
  <groupId>com.vaadin.addon</groupId>
```

```
<artifactId>vaadin-charts</artifactId>
<version>3.0.0-beta1</version>
</dependency>
```

You also need to define the Vaadin add-ons repository if not already defined:

```
<repository>
  <id>vaadin-addons</id>
  <url>http://maven.vaadin.com/vaadin-addons</url>
</repository>
```

19.2.2. Ivy Dependency

The Ivy dependency, to be defined in `ivy.xml`, would be as follows:

```
<dependency org="com.vaadin" name="vaadin-charts"
  rev="3.0.0-beta1" />
```

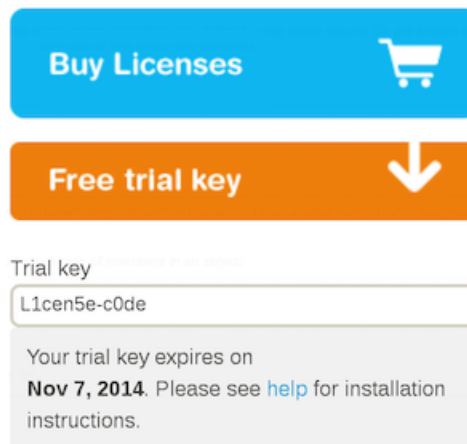
It is generally recommended to use a fixed version number, but you can also use `latest.release` to get the latest release.

19.2.3. Installing License Key

You need to install a license key before compiling the widget set. The license key is checked during widget set compilation, so you do not need it when deploying the application.

You can purchase Vaadin Charts or obtain a free trial key from the Vaadin Charts download page in Vaadin Directory. You need to register in Vaadin Directory to obtain the key.

Figure 19.3. Obtaining License Key from Vaadin Directory



To install the license key in a development workstation, you can copy and paste it verbatim to a `.vaadin.charts.developer.license` file in your home directory. For example, in Linux and OS X:

```
$ echo "L1cen5e-c0de" > ~/.vaadin.charts.developer.license
```

You can also pass the key as a system property to the widget set compiler, usually with a `-D` option. For example, on the command-line:

```
$ java -Dvaadin.charts.developer.license=L1cen5e-c0de ...
```

Passing License Key in Different Environments

How you actually pass the parameter to the widget set compiler depends on the development environment and the build system that you use to compile the widget set. Below are listed a few typical environments:

Eclipse IDE

To install the license key for all projects, select **Window** → **Preferences** and navigate to the **Java** → **Installed JREs** section. Select the JRE version that you use for the application and click **Edit**. In the **Default VM arguments**, give the `-D` expression as shown above.

Apache Ant

If compiling the project with Apache Ant, you could set the key in the Ant script as follows:

```
<sysproperty key="vaadin.charts.developer.license"
             value="L1cen5e-c0de"/>
```

However, you should never store license keys in a source repository, so if the Ant script is stored in a source repository, you should pass the license key to Ant as a property that you then use in the script for the value argument of the `<sysproperty>` as follows:

```
<sysproperty key="vaadin.charts.developer.license"
             value="${vaadin.charts.developer.license}"/>
```

When invoking Ant from the command-line, you can pass the property with a `-D` parameter to Ant.

Apache Maven

If building the project with Apache Maven, you can pass the license key with a `-D` parameter to Maven:

```
$ mvn -Dvaadin.charts.developer.license=L1cen5e-c0de package
```

Continuous Integration Systems

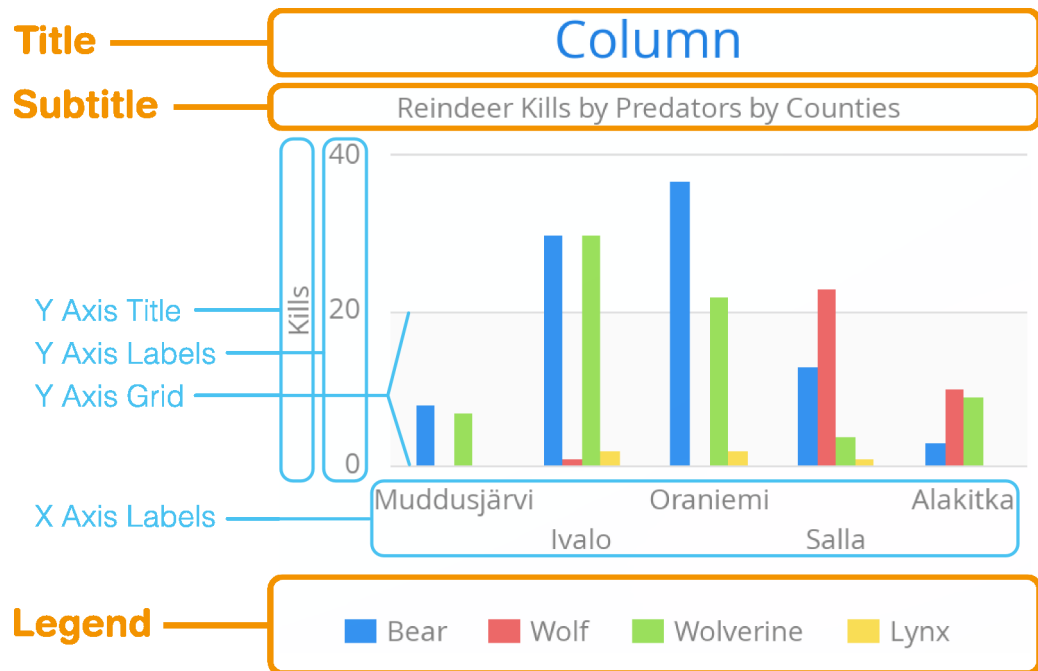
In CIS systems, you can pass the license key to build runners as a system property in the build configuration. However, this only passes it to a runner. As described above, Ant does not pass it to sub-processes implicitly, so you need to forward it explicitly as described earlier.

19.3. Basic Use

The **Chart** is a regular Vaadin component, which you can add to a layout. You can give the chart type in the constructor or set it later in the chart model. A chart has a height of 400 pixels and takes full width by default, which settings you may often need to customize.

```
Chart chart = new Chart(ChartType.COLUMN);
chart.setWidth("400px"); // 100% by default
chart.setHeight("300px"); // 400px by default
...
layout.addComponent(chart);
```

The chart types are described in Section 19.4, “Chart Types”. The main parts of a chart are illustrated in Figure 19.4, “Chart Elements”.

Figure 19.4. Chart Elements

To actually display something in a chart, you typically need to configure the following aspects:

- Basic chart configuration
- Configure *plot options* for the chart type
- Configure one or more *data series* to display
- Configure *axes*

The plot options can be configured for each data series individually, or for different chart types in mixed-type charts.

19.3.1. Basic Chart Configuration

After creating a chart, you need to configure it further. At the least, you need to specify the data series to be displayed in the configuration.

Most methods available in the **Chart** object handle its basic Vaadin component properties. All the chart-specific properties are in a separate **Configuration** object, which you can access with the `getConfiguration()` method.

```
Configuration conf = chart.getConfiguration();
conf.setTitle("Reindeer Kills by Predators");
conf.setSubTitle("Kills Grouped by Counties");
```

The configuration properties are described in more detail in Section 19.5, “Chart Configuration”.

19.3.2. Plot Options

Many chart settings can be configured in the *plot options* of the chart or data series. Some of the options are chart type specific, as described later for each chart type, while many are shared.

For example, for line charts, you could disable the point markers as follows:

```
// Disable markers from lines
PlotOptionsLine plotOptions = new PlotOptionsLine();
plotOptions.setMarker(new Marker(false));
conf.setPlotOptions(plotOptions);
```

You can set the plot options for the entire chart or for each data series separately, allowing also mixed-type charts, as described in Section 19.3.6, “Mixed Type Charts”.

The shared plot options are described in Section 19.5.1, “Plot Options”.

19.3.3. Chart Data Series

The data displayed in a chart is stored in the chart configuration as a list of **Series** objects. A new data series is added in a chart with the `addSeries()` method.

```
ListSeries series = new ListSeries("Diameter");
series.setData(4900, 12100, 12800,
               6800, 143000, 125000,
               51100, 49500);
conf.addSeries(series);
```

The data can be specified with a number of different series types **DataSetSeries**, **ListSeries**, **AreaListSeries**, **RangeSeries**, and **ContainerDataSetSeries**.

Data point features, such as color and size, can be defined in the versatile **DataSetSeries**, which contains **DataSetSeriesItem** items. Special chart types, such as box plots and 3D scatter charts require using their own special data point type.

The data series configuration is described in more detail in Section 19.6, “Chart Data”.

19.3.4. Axis Configuration

One of the most common tasks for charts is customizing its axes. At the least, you usually want to set the axis titles. Usually you also want to specify labels for data values in the axes.

When an axis is categorical rather than numeric, you can define category labels for the items. They must be in the same order and the same number as you have values in your data series.

```
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                  "Mars", "Jupiter", "Saturn",
                  "Uranus", "Neptune");
xaxis.setTitle("Planet");
conf.addXAxis(xaxis);
```

Formatting of numeric labels can be done with JavaScript expressions, for example as follows:

```
// Set the Y axis title
YAxis yaxis = new YAxis();
yaxis.setTitle("Diameter");
```

```
yaxis.getLabels().setFormatter(
    "function() {return Math.floor(this.value/1000) + 'Mm\';}");
yaxis.getLabels().setStep(2);
conf.addyAxis(yaxis);
```

19.3.5. Displaying Multiple Series

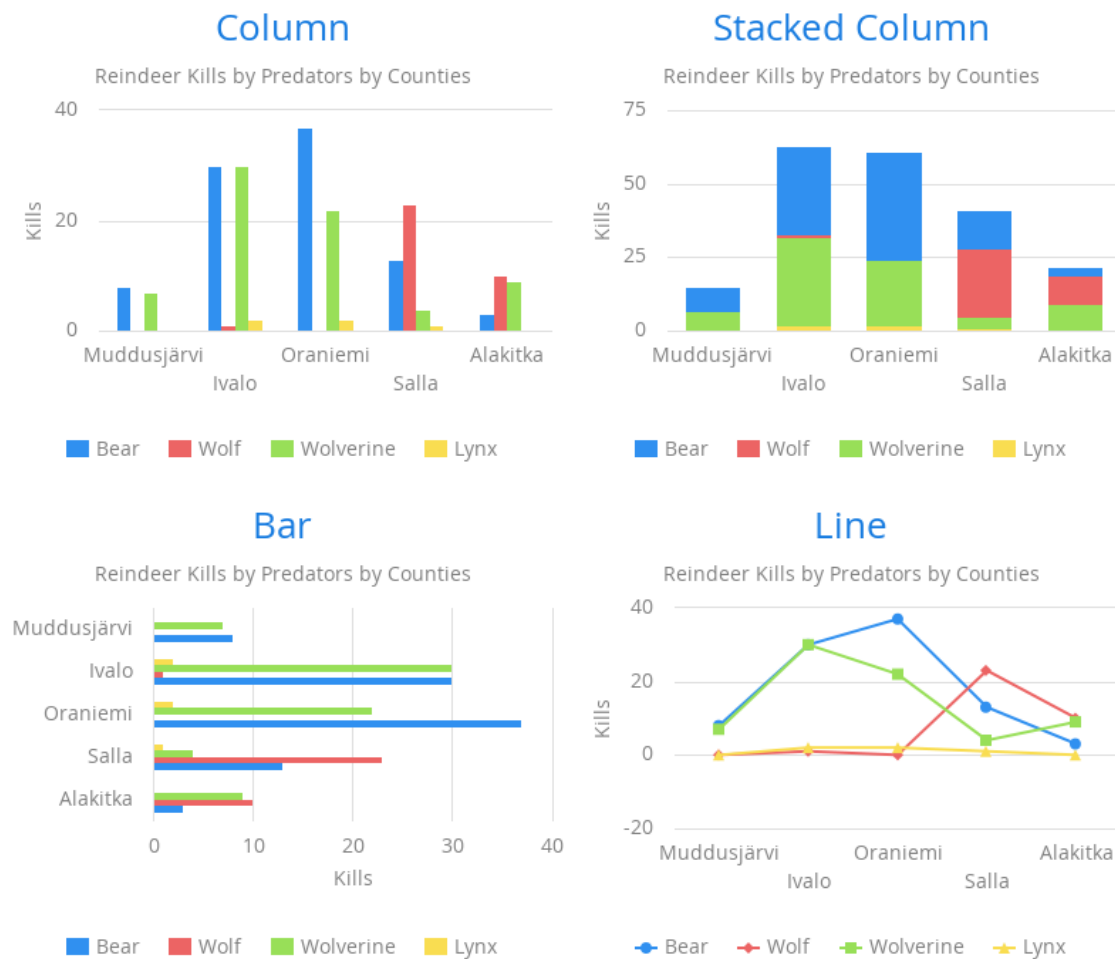
The simplest data, which we saw in the examples earlier in this chapter, is one-dimensional and can be represented with a single data series. Most chart types support multiple data series, which are used for representing two-dimensional data. For example, in line charts, you can have multiple lines and in column charts the columns for different series are grouped by category. Different chart types can offer alternative display modes, such as stacked columns. The legend displays the symbols for each series.

```
// The data
// Source: V. Maijala, H. Norberg, J. Kumpula, M. Nieminen
// Calf production and mortality in the Finnish
// reindeer herding area. 2002.
String predators[] = { "Bear", "Wolf", "Wolverine", "Lynx" };
int kills[][] = { // Location:
    {8, 0, 7, 0}, // Muddusjarvi
    {30, 1, 30, 2}, // Ivalo
    {37, 0, 22, 2}, // Oraniemi
    {13, 23, 4, 1}, // Salla
    {3, 10, 9, 0}, // Alakitka
};

// Create a data series for each numeric column in the table
for (int predator = 0; predator < 4; predator++) {
    ListSeries series = new ListSeries();
    series.setName(predators[predator]);

    // The rows of the table
    for (int location = 0; location < kills.length; location++)
        series.addData(kills[location][predator]);
    conf.addSeries(series);
}
```

The result for both regular and stacked column chart is shown in Figure 19.5, “Multiple Series in a Chart”. Stacking is enabled with `setStacking()` in **PlotOptionsColumn**.

Figure 19.5. Multiple Series in a Chart

19.3.6. Mixed Type Charts

You can enable mixed charts by setting the chart type in the **PlotOptions** object for a data series, which overrides the default chart type set in the **Chart** object. You can also make color and other settings for the series in the plot options.

For example, to get a line chart, you need to use **PlotOptionsLine**.

```
// A data series as column graph
DataSeries series1 = new DataSeries();
PlotOptionsColumn options1 = new PlotOptionsColumn();
options1.setColor(SolidColor.BLUE);
series1.setPlotOptions(options1);
series1.setData(4900, 12100, 12800,
    6800, 143000, 125000, 51100, 49500);
conf.addSeries(series1);

// A data series as line graph
ListSeries series2 = new ListSeries("Diameter");
PlotOptionsLine options2 = new PlotOptionsLine();
options2.setColor(SolidColor.RED);
series2.setPlotOptions(options2);
```

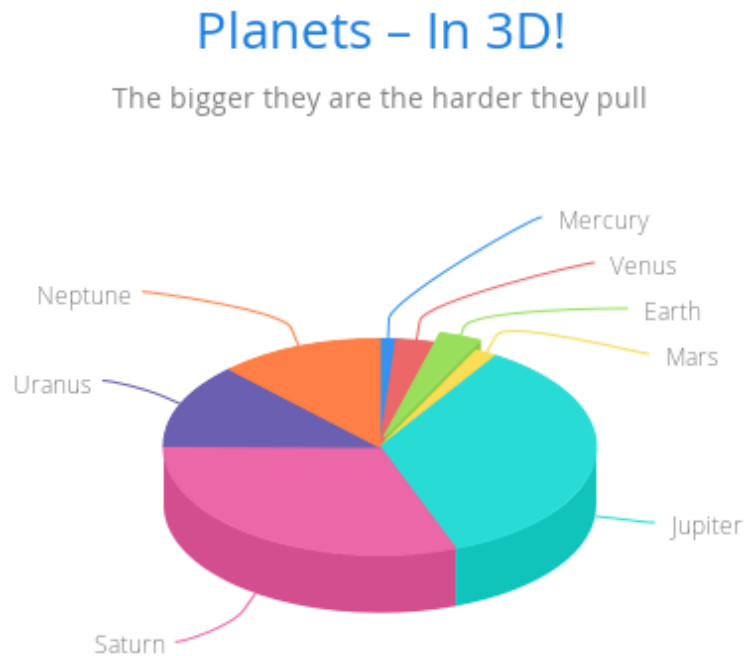
```
series2.setData(4900, 12100, 12800,
                6800, 143000, 125000, 51100, 49500);
conf.addSeries(series2);
```

In the above case, where we set the chart type for each series, the overall chart type is irrelevant.

19.3.7. 3D Charts

Most chart types can be made 3-dimensional by adding 3D options to the chart. You can rotate the charts, set up the view distance, and define the thickness of the chart features, among other things. You can also set up a 3D axis frame around a chart.

Figure 19.6. 3D Charts



3D Options

3D view has to be enabled in the **Options3d** configuration, along with other parameters. Minimally, to have some 3D effect, you need to rotate the chart according to the *alpha* and *beta* parameters.

Let us consider a basic scatter chart for an example. The basic configuration for scatter charts is described elsewhere, but let us look how to make it 3D.

```
Chart chart = new Chart(ChartType.SCATTER);
Configuration conf = chart.getConfiguration();
... other chart configuration ...

// In 3D!
Options3d options3d = new Options3d();
options3d.setEnabled(true);
options3d.setAlpha(10);
options3d.setBeta(30);
options3d.setDepth(135); // Default is 100
```

```
options3d.setViewDistance(100); // Default
conf.getChart().setOptions3d(options3d);
```

The 3D options are as follows:

alpha

The vertical tilt (pitch) in degrees.

beta

The horizontal tilt (yaw) in degrees.

depth

Depth of the third (Z) axis in pixel units.

enabled

Whether 3D plot is enabled. Default is *false*.

frame

Defines the 3D frame, which consists of a back, bottom, and side panels that display the chart grid.

```
+
----
Frame frame = new Frame();
Back back=new Back();
back.setColor(SolidColor.BEIGE);
back.setSize(1);
frame.setBack(back);
options3d.setFrame(frame);
----
[parameter]#viewDistance#:: View distance for creating perspective distortion.
Default is 100.
```

3D Plot Options

The above sets up the general 3D view, but you also need to configure the 3D properties of the actual chart type. The 3D plot options are chart type specific. For example, a pie has *depth* (or thickness), which you can configure as follows:

```
// Set some plot options
PlotOptionsPie options = new PlotOptionsPie();
... Other plot options for the chart ...

options.setDepth(45); // Our pie is quite thick

conf.setPlotOptions(options);
```

3D Data

For some chart types, such as pies and columns, the 3D view is merely a visual representation for one- or two-dimensional data. Some chart types, such as scatter charts, also feature a third, *depth axis*, for data points. Such data points can be given as **DataSeriesItem3d** objects.

The Z parameter is *depth* and is not scaled; there is no configuration for the depth or Z axis. Therefore, you need to handle scaling yourself as is done in the following.

```
// Orthogonal data points in 2x2x2 cube
double[][] points = { {0.0, 0.0, 0.0}, // x, y, z
```

```
        {1.0, 0.0, 0.0},
        {0.0, 1.0, 0.0},
        {0.0, 0.0, 1.0},
        {-1.0, 0.0, 0.0},
        {0.0, -1.0, 0.0},
        {0.0, 0.0, -1.0}};

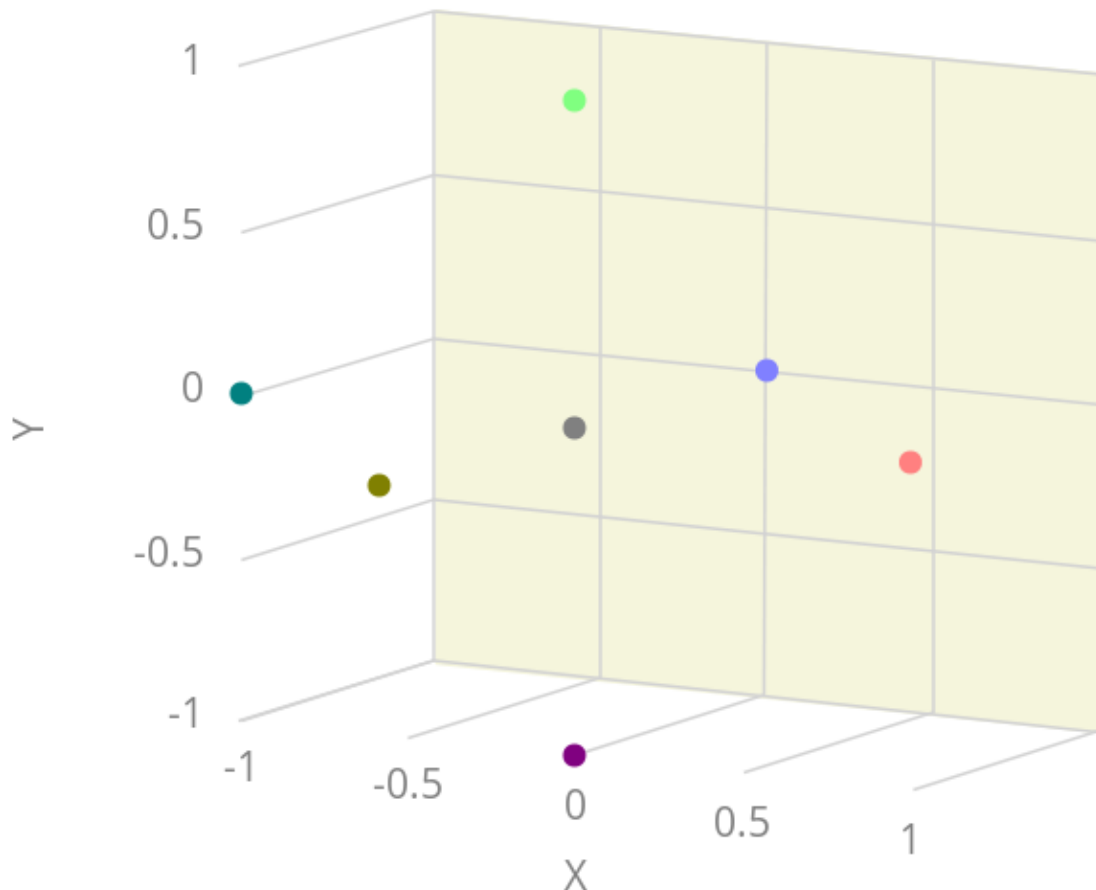
DataSeries series = new DataSeries();
for (int i=0; i<points.length; i++) {
    double x = points[i][0];
    double y = points[i][1];
    double z = points[i][2];

    // Scale the depth coordinate, as the depth axis is
    // not scaled automatically
    DataSeriesItem3d item = new DataSeriesItem3d(x, y,
        z * options3d.getDepth().doubleValue());
    series.add(item);
}
conf.addSeries(series);
```

Above, we defined 7 orthogonal data points in the 2x2x2 cube centered in origo. The 3D depth was set to 135 earlier. The result is illustrated in Figure 19.7, “3D Scatter Chart”.

Figure 19.7. 3D Scatter Chart

Scatter – in 3D!



19.3.8. Chart Themes

The visual style and essentially any other chart configuration can be defined in a *theme*. All charts shown in a UI may have only one theme, which can be set with `setTheme()` in the **ChartOptions**.

The **ChartOptions** is a **UI** extension that is created and referenced by calling the `get()` as follows:

```
// Set Charts theme for the current UI
ChartOptions.get().setTheme(new SkiesTheme());
```

The **VaadinTheme** is the default chart theme in Vaadin Charts. Other available themes are **GrayTheme**, **GridTheme**, **SkiesTheme**, and **HighChartsDefaultTheme**.

A theme is a Vaadin Charts configuration that is used as a template for the configuration when rendering the chart.

19.4. Chart Types

Vaadin Charts comes with over a dozen different chart types. You normally specify the chart type in the constructor of the **Chart** object. The available chart types are defined in the **ChartType** enum. You can later read or set the chart type with the `chartType` property of the chart model, which you can get with `getConfiguration().getChart()`.

The supported chart types are:

area	arearange	areaspline	areasplinerange
bar	boxplot	bubble	column
columnrange	errorbar	flags	funnel
gauge	heatmap	line	pie
polygon	pyramid	scatter	solidgauge
sparkline	spline	treemap	waterfall

Each chart type has its specific plot options and support its specific collection of chart features. They also have specific requirements for the data series.

The basic chart types and their variants are covered in the following subsections.

19.4.1. Line and Spline Charts

Line charts connect the series of data points with lines. In the basic line charts the lines are straight, while in spline charts the lines are smooth polynomial interpolations between the data points.

Table 19.1. Line Chart Subtypes

ChartType	Plot Options Class
<i>LINE</i>	PlotOptionsLine
<i>SPLINE</i>	PlotOptionsSpline

Plot Options

The `color` property in the line plot options defines the line color, `lineWidth` the line width, and `dashStyle` the dash pattern for the lines.

See Section 19.4.6, “Scatter Charts” for plot options regarding markers and other data point properties. The markers can also be configured for each data point.

19.4.2. Area Charts

Area charts are like line charts, except that they fill the area between the line and some threshold value on Y axis. The threshold depends on the chart type. In addition to the base type, chart type combinations for spline interpolation and ranges are supported.

Table 19.2. Area Chart Subtypes

ChartType	Plot Options Class
<i>AREA</i>	PlotOptionsArea
<i>AREASPLINE</i>	PlotOptionsAreaSpline
<i>AREARANGE</i>	PlotOptionsAreaRange
<i>AREASPLINERANGE</i>	PlotOptionsAreaSplineRange

In area range charts, the area between a lower and upper value is painted with a transparent color. The data series must specify the minimum and maximum values for the Y coordinates, defined either with **RangeSeries**, as described in Section 19.6.3, “Range Series”, or with **DataSeries**, described in Section 19.6.2, “Generic Data Series”.

Plot Options

Area charts support *stacking*, so that multiple series are piled on top of each other. You enable stacking from the plot options with `setStacking()`. The *Stacking.NORMAL* stacking mode does a normal summative stacking, while the *Stacking.PERCENT* handles them as proportions.

The fill color for the area is defined with the *fillColor* property and its transparency with *fillOpacity* (the opposite of transparency) with a value between 0.0 and 1.0.

The *color* property in the line plot options defines the line color, *lineWidth* the line width, and *dashStyle* the dash pattern for the lines.

See Section 19.4.6, “Scatter Charts” for plot options regarding markers and other data point properties. The markers can also be configured for each data point.

19.4.3. Column and Bar Charts

Column and bar charts illustrate values as vertical or horizontal bars, respectively. The two chart types are essentially equivalent, just as if the orientation of the axes was inverted.

Multiple data series, that is, two-dimensional data, are shown with thinner bars or columns grouped by their category, as described in Section 19.3.5, “Displaying Multiple Series”. Enabling stacking with `setStacking()` in plot options stacks the columns or bars of different series on top of each other.

You can also have *COLUMNRANGE* charts that illustrate a range between a lower and an upper value, as described in Section 19.4.11, “Area and Column Range Charts”. They require the use of **RangeSeries** for defining the lower and upper values.

Table 19.3. Column and Bar Chart Subtypes

ChartType	Plot Options Class
<i>COLUMN</i>	PlotOptionsColumn
<i>COLUMNRANGE</i>	PlotOptionsColumnRange
<i>BAR</i>	PlotOptionsBar

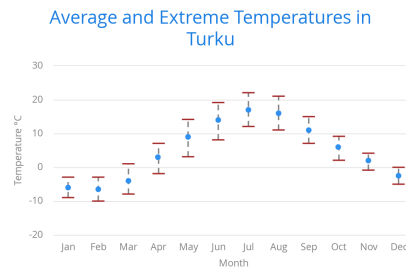
See the API documentation for details regarding the plot options.

19.4.4. Error Bars

An error bars visualize errors, or high and low values, in statistical data. They typically represent high and low values in data or a multitude of standard deviation, a percentile, or a quantile. The high and low values are represented as horizontal lines, or "whiskers", connected by a vertical stem.

While error bars technically are a chart type (`ChartType.ERRORBAR`), you normally use them together with some primary chart type, such as a scatter or column chart.

Figure 19.8. Error Bars in a Scatter Chart



To display the error bars for data points, you need to have a separate data series for the low and high values. The data series needs to use the **PlotOptionsErrorBar** plot options type.

```
// Create a chart of some primary type
Chart chart = new Chart(ChartType.SCATTER);
chart.setWidth("600px");
chart.setHeight("400px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Average Temperatures in Turku");
conf.getLegend().setEnabled(false);

// The primary data series
ListSeries averages = new ListSeries(
    -6, -6.5, -4, 3, 9, 14, 17, 16, 11, 6, 2, -2.5);

// Error bar data series with low and high values
DataSeries errors = new DataSeries();
errors.add(new DataSeriesItem(0, -9, -3));
errors.add(new DataSeriesItem(1, -10, -3));
errors.add(new DataSeriesItem(2, -8, 1));
...

// Configure the stem and whiskers in error bars
PlotOptionsErrorbar barOptions = new PlotOptionsErrorbar();
barOptions.setStemColor(SolidColor.GREY);
barOptions.setStemWidth(2);
barOptions.setStemDashStyle(DashStyle.DASH);
barOptions.setWhiskerColor(SolidColor.BROWN);
barOptions.setWhiskerLength(80, Sizeable.Unit.PERCENTAGE); // 80% of category
width
barOptions.setWhiskerWidth(2); // Pixels
errors.setPlotOptions(barOptions);

// The errors should be drawn lower
```

```
conf.addSeries(errors);
conf.addSeries(averages);
```

Note that you should add the error bar series first, to have it rendered lower in the chart.

Plot Options

Plot options for error bar charts have type **PlotOptionsErrorBar**. It has the following chart-specific plot option properties:

whiskerColor, *whiskerWidth*, and *whiskerLength*

The color, width (vertical thickness), and length of the horizontal "whiskers" that indicate high and low values.

stemColor, *stemWidth*, and *stemDashStyle*

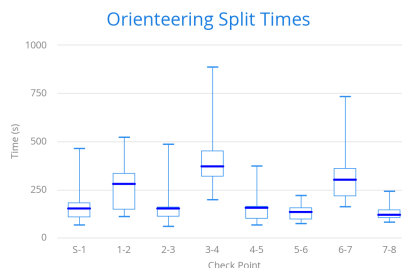
The color, width (thickness), and line style of the vertical "stems" that connect the whiskers. In box plot charts, which also have stems, they extend from the quadrantile box.

19.4.5. Box Plot Charts

Box plot charts display the distribution of statistical variables. A data point has a median, represented with a horizontal line, upper and lower quartiles, represented by a box, and a low and high value, represented with T-shaped "whiskers". The exact semantics of the box symbols are up to you.

Box plot chart is closely related to the error bar chart described in Section 19.4.4, "Error Bars", sharing the box and whisker elements.

Figure 19.9. Box Plot Chart



The chart type for box plot charts is `ChartType.BOXPLOT`. You normally have just one data series, so it is meaningful to disable the legend.

```
Chart chart = new Chart(ChartType.BOXPLOT);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Orienteering Split Times");
conf.getLegend().setEnabled(false);
```

Plot Options

The plot options for box plots have type **PlotOptionsBoxPlot**, which extends the slightly more generic **PlotOptionsErrorBar**. They have the following plot option properties:

medianColor, medianWidth

Color and width (vertical thickness) of the horizontal median indicator line.

For example:

```
// Set median line color and thickness
PlotOptionsBoxplot plotOptions = new PlotOptionsBoxplot();
plotOptions.setMedianColor(SolidColor.BLUE);
plotOptions.setMedianWidth(3);
conf.setPlotOptions(plotOptions);
```

Data Model

As the data points in box plots have five different values instead of the usual one, they require using a special **BoxPlotItem**. You can give the different values with the setters, or all at once in the constructor.

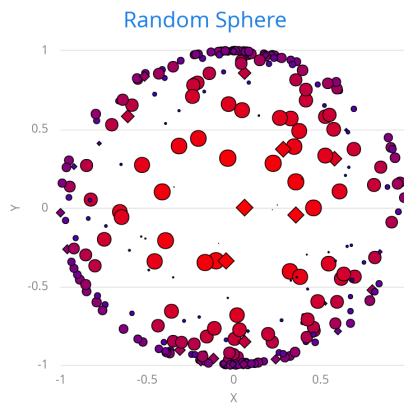
```
// Orienteering control point times for runners
double data[][] = orienteeringdata();

DataSeries series = new DataSeries();
for (double cpointtimes[]: data) {
    StatAnalysis analysis = new StatAnalysis(cpointtimes);
    series.add(new BoxPlotItem(analysis.low(),
                              analysis.firstQuartile(),
                              analysis.median(),
                              analysis.thirdQuartile(),
                              analysis.high()));
}
conf.setSeries(series);
```

If the "low" and "high" attributes represent an even smaller quantile, or a larger multiple of standard deviation, you can have outliers. You can plot them with a separate data series, with

19.4.6. Scatter Charts

Scatter charts display a set of unconnected data points. The name refers to freely given X and Y coordinates, so the **DataSeries** or **ContainerSeries** are usually the most meaningful data series types for scatter charts.

Figure 19.10. Scatter Chart

The chart type of a scatter chart is `ChartType.SCATTER`. Its options can be configured in a **PlotOptionsScatter** object, although it does not have any chart-type specific options.

```
Chart chart = new Chart(ChartType.SCATTER);
chart.setWidth("500px");
chart.setHeight("500px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Random Sphere");
conf.getLegend().setEnabled(false); // Disable legend

PlotOptionsScatter options = new PlotOptionsScatter();
// ... Give overall plot options here ...
conf.setPlotOptions(options);

DataSeries series = new DataSeries();
for (int i=0; i<300; i++) {
    double lng = Math.random() * 2 * Math.PI;
    double lat = Math.random() * Math.PI - Math.PI/2;
    double x = Math.cos(lat) * Math.sin(lng);
    double y = Math.sin(lat);
    double z = Math.cos(lng) * Math.cos(lat);

    DataSeriesItem point = new DataSeriesItem(x,y);
    Marker marker = new Marker();
    // Make settings as described later
    point.setMarker(marker);
    series.add(point);
}
conf.addSeries(series);
```

The result was shown in Figure 19.10, “Scatter Chart”.

Data Point Markers

Scatter charts and other charts that display data points, such as line and spline charts, visualize the points with *markers*. The markers can be configured with the **Marker** property objects available from the plot options of the relevant chart types, as well as at the level of each data point, in the **DataSeriesItem**. You need to create the marker and apply it with the `setMarker()` method in the plot options or the data series item.

For example, to set the marker for an individual data point:

```

DataSeriesItem point = new DataSeriesItem(x,y);
Marker marker = new Marker();
// ... Make any settings ...
point.setMarker(marker);
series.add(point);

```

Marker Shape Properties

A marker has a *lineColor* and a *fillColor*, which are set using a **Color** object. Both solid colors and gradients are supported. You can use a **SolidColor** to specify a solid fill color by RGB values or choose from a selection of predefined colors in the class.

```

// Set line width and color
marker.setLineWidth(1); // Normally zero width
marker.setLineColor(SolidColor.BLACK);

// Set RGB fill color
int level = (int) Math.round((1-z)*127);
marker.setFillColors(
    new SolidColor(255-level, 0, level));
point.setMarker(marker);
series.add(point);

```

You can also use a color gradient with **GradientColor**. Both linear and radial gradients are supported, with multiple color stops.

Marker size is determined by the *radius* parameter, which is given in pixels. The actual visual radius includes also the line width.

```
marker.setRadius((z+1)*5);
```

Marker Symbols

Markers are visualized either with a shape or an image symbol. You can choose the shape from a number of built-in shapes defined in the **MarkerSymbolEnum** enum (*CIRCLE*, *SQUARE*, *DIAMOND*, *TRIANGLE*, or *TRIANGLE_DOWN*). These shapes are drawn with a line and fill, which you can set as described above.

```
marker.setSymbol(MarkerSymbolEnum.DIAMOND);
```

You can also use any image accessible by a URL by using a **MarkerSymbolUrl** symbol. If the image is deployed with your application, such as in a theme folder, you can determine its URL as follows:

```

String url = VaadinServlet.getCurrent().getServletContext()
    .getContextPath() + "/VAADIN/themes/mytheme/img/smiley.png";
marker.setSymbol(new MarkerSymbolUrl(url));

```

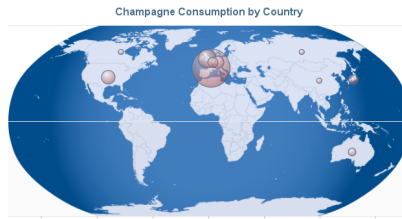
The line, radius, and color properties are not applicable to image symbols.

19.4.7. Bubble Charts

Bubble charts are a special type of scatter charts for representing three-dimensional data points with different point sizes. We demonstrated the same possibility with scatter charts in Section 19.4.6, “Scatter Charts”, but the bubble charts make it easier to define the size of a point

by its third (Z) dimension, instead of the radius property. The bubble size is scaled automatically, just like for other dimensions. The default point style is also more bubbly.

Figure 19.11. Bubble Chart



The chart type of a bubble chart is `ChartType.BUBBLE`. Its options can be configured in a **PlotOptionsBubble** object, which has a single chart-specific property, `displayNegative`, which controls whether bubbles with negative values are displayed at all. More typically, you want to configure the bubble `marker`. The bubble tooltip is configured in the basic configuration. The Z coordinate value is available in the formatter JavaScript with `this.point.z` reference.

The bubble radius is scaled linearly between a minimum and maximum radius. If you would rather scale by the area of the bubble, you can approximate that by taking square root of the Z values.

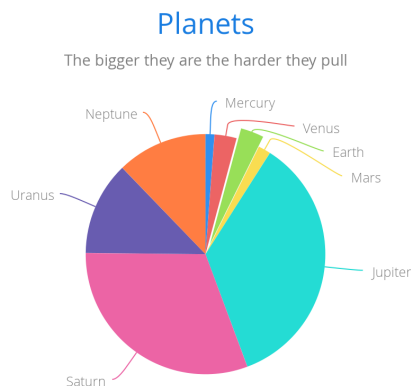
19.4.8. Pie Charts

A pie chart illustrates data values as sectors of size proportionate to the sum of all values. The pie chart is enabled with `ChartType.PIE` and you can make type-specific settings in the **PlotOptionsPie** object as described later.

```
Chart chart = new Chart(ChartType.PIE);
Configuration conf = chart.getConfiguration();
...
```

A ready pie chart is shown in Figure 19.12, “Pie Chart”.

Figure 19.12. Pie Chart



Plot Options

The chart-specific options of a pie chart are configured with a **PlotOptionsPie**.


```
PlotOptionsPie options = new PlotOptionsPie();
options.setInnerSize("0");
options.setSize("75%"); // Default
options.setCenter("50%", "50%"); // Default
conf.setPlotOptions(options);
```

innerSize

A pie with inner size greater than zero is a "donut". The inner size can be expressed either as number of pixels or as a relative percentage of the chart area with a string (such as "60%") See the section later on donuts.

size

The size of the pie can be expressed either as number of pixels or as a relative percentage of the chart area with a string (such as "80%"). The default size is 75%, to leave space for the labels.

center

The X and Y coordinates of the center of the pie can be expressed either as numbers of pixels or as a relative percentage of the chart sizes with a string. The default is "50%", "50%".

Data Model

The labels for the pie sectors are determined from the labels of the data points. The **DataSeries** or **ContainerSeries**, which allow labeling the data points, should be used for pie charts.

```
DataSet series = new DataSet();
series.add(new DataSetItem("Mercury", 4900));
series.add(new DataSetItem("Venus", 12100));
...
conf.addSeries(series);
```

If a data point, as defined as a **DataSetItem** in a **DataSet**, has the *sliced* property enabled, it is shown as slightly cut away from the pie.

```
// Slice one sector out
DataSetItem earth = new DataSetItem("Earth", 12800);
earth.setSliced(true);
series.add(earth);
```

Donut Charts

Setting the *innerSize* of the plot options of a pie chart to a larger than zero value results in an empty hole at the center of the pie.

```
PlotOptionsPie options = new PlotOptionsPie();
options.setInnerSize("60%");
conf.setPlotOptions(options);
```

As you can set the plot options also for each data series, you can put two pie charts on top of each other, with a smaller one fitted in the "hole" of the donut. This way, you can make pie charts with more details on the outer rim, as done in the example below:

```
// The inner pie
DataSet innerSeries = new DataSet();
innerSeries.setName("Browsers");
PlotOptionsPie innerPieOptions = new PlotOptionsPie();
innerPieOptions.setSize("60%");
```

```

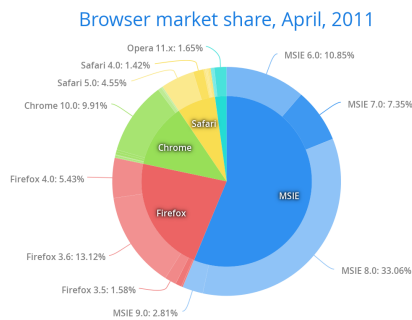
innerSeries.setPlotOptions(innerPieOptions);
...

DataSeries outerSeries = new DataSeries();
outerSeries.setName("Versions");
PlotOptionsPie outerSeriesOptions = new PlotOptionsPie();
outerSeriesOptions.setInnerSize("60%");
outerSeries.setPlotOptions(outerSeriesOptions);
...

```

The result is illustrated in Figure 19.13, “Overlaid Pie and Donut Chart”.

Figure 19.13. Overlaid Pie and Donut Chart



19.4.9. Gauges

A gauge is an one-dimensional chart with a circular Y-axis, where a rotating pointer points to a value on the axis. A gauge can, in fact, have multiple Y-axes to display multiple scales.

A *solid gauge* is otherwise like a regular gauge, except that a solid color arc is used to indicate current value instead of a pointer. The color of the indicator arc can be configured to change according to color stops.

Let us consider the following gauge:

```

Chart chart = new Chart(ChartType.GAUGE);
chart.setWidth("400px");
chart.setHeight("400px");

```

After the settings done in the subsequent sections, it will show as in Figure 19.14, “A Gauge”.

Figure 19.14. A Gauge

Speedometer



Gauge Configuration

The start and end angles of the gauge can be configured in the **Pane** object of the chart configuration. The angles can be given as -360 to 360 degrees, with 0 at the top of the circle.

```
Configuration conf = chart.getConfiguration();
conf.setTitle("Speedometer");
conf.getPane().setStartAngle(-135);
conf.getPane().setEndAngle(135);
```

Axis Configuration

A gauge has only an Y-axis. You need to provide both a minimum and maximum value for it.

```
YAxis yaxis = new YAxis();
yaxis.setTitle("km/h");

// The limits are mandatory
yaxis.setMin(0);
yaxis.setMax(100);

// Other configuration
yaxis.getLabels().setStep(1);
yaxis.setTickInterval(10);
yaxis.setTickLength(10);
yaxis.setTickWidth(1);
yaxis.setMinorTickInterval("1");
yaxis.setMinorTickLength(5);
yaxis.setMinorTickWidth(1);
yaxis.setPlotBands(new PlotBand[]{
    new PlotBand(0, 60, SolidColor.GREEN),
    new PlotBand(60, 80, SolidColor.YELLOW),
    new PlotBand(80, 100, SolidColor.RED)});
yaxis.setGridLineWidth(0); // Disable grid

conf.addYAxis(yaxis);
```

You can do all kinds of other configuration to the axis - please see the API documentation for all the available parameters.

Setting and Updating Gauge Data

A gauge only displays a single value, which you can define as a data series of length one, such as as follows:

```
ListSeries series = new ListSeries("Speed", 80);
conf.addSeries(series);
```

Gauges are especially meaningful for displaying changing values. You can use the `updatePoint()` method in the data series to update the single value.

```
final TextField tf = new TextField("Enter a new value");
layout.addComponent(tf);

Button update = new Button("Update", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        Integer newValue = new Integer((String)tf.getValue());
```

```

        series.updatePoint(0, newValue);
    }
});
layout.addComponent(update);

```

19.4.10. Solid Gauges

A solid gauge is much like a regular gauge described previously; a one-dimensional chart with a circular Y-axis. However, instead of a rotating pointer, the value is indicated by a rotating arc with solid color. The color of the indicator arc can be configured to change according to the value using color stops.

Let us consider the following gauge:

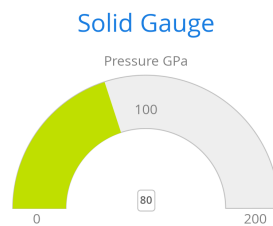
```

Chart chart = new Chart(ChartType.SOLIDGAUGE);
chart.setWidth("400px");
chart.setHeight("400px");

```

After the settings done in the subsequent sections, it will show as in Figure 19.15, “A Solid Gauge”.

Figure 19.15. A Solid Gauge



While solid gauge is much like a regular gauge, the configuration differs

Configuration

The solid gauge must be configured in the drawing **Pane** of the chart configuration. The gauge arc spans an angle, which is specified as -360 to 360 degrees, with 0 degrees at the top of the arc. Typically, a semi-arc is used, where you use -90 and 90 for the angles, and move the center lower than you would have with a full circle. You can also adjust the size of the gauge pane; enlargening it allows positioning tick labels better.

```

Configuration conf = chart.getConfiguration();
conf.setTitle("Solid Gauge");

Pane pane = conf.getPane();
pane.setSize("125%"); // For positioning tick labels
pane.setCenter("50%", "70%"); // Move center lower
pane.setStartAngle(-90); // Make semi-circle
pane.setEndAngle(90); // Make semi-circle

```

The shape of the gauge display is defined as the background of the pane. You at least need to set the shape as either "arc" or "solid". You typically also want to set background color and inner and outer radius.

```

Background bkg = new Background();
bkg.setBackgroundColor(new SolidColor("#eeeeee")); // Gray
bkg.setInnerRadius("60%"); // To make it an arc and not circle

```

```
bkg.setOuterRadius("100%"); // Default - not necessary
bkg.setShape("arc"); // solid or arc
pane.setBackground(bkg);
```

Axis Configuration

A gauge only has an Y-axis. You must define the value range (*min* and *max*).

```
YAxis yaxis = new YAxis();
yaxis.setTitle("Pressure GPa");
yaxis.getTitle().setY(-80); // Move 70 px upwards from center

// The limits are mandatory
yaxis.setMin(0);
yaxis.setMax(200);

// Configure ticks and labels
yaxis.setTickInterval(100); // At 0, 100, and 200
yaxis.getLabels().setY(-16); // Move 16 px upwards
yaxis.setGridLineWidth(0); // Disable grid
```

You can configure color stops for the indicator arc. The stops are defined with **Stop** objects having stop points from 0.0 to 1.0 and color values.

```
yaxis.setStops(new Stop(0.1f, SolidColor.GREEN),
               new Stop(0.5f, SolidColor.YELLOW),
               new Stop(0.9f, SolidColor.RED));

conf.addYAxis(yaxis);
```

Setting `yaxis.getLabels().setRotationPerpendicular()` makes gauge labels rotate perpendicular to the center.

You can do all kinds of other configuration to the axis - please see the API documentation for all the available parameters.

Plot Options

Solid gauges do not currently have any chart type specific plot options. See Section 19.5.1, "Plot Options" for common options.

```
PlotOptionsSolidgauge options = new PlotOptionsSolidgauge();

// Move the value display box at the center a bit higher
Labels dataLabels = new Labels();
dataLabels.setY(-20);
options.setDataLabels(dataLabels);

conf.setPlotOptions(options);
```

Setting and Updating Gauge Data

A gauge only displays a single value, which you can define as a data series of length one, such as as follows:

```
ListSeries series = new ListSeries("Pressure MPa", 80);
conf.addSeries(series);
```

Gauges are especially meaningful for displaying changing values. You can use the `updatePoint()` method in the data series to update the single value.

```
final TextField tf = new TextField("Enter a new value");
layout.addComponent(tf);

Button update = new Button("Update", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        Integer newValue = new Integer((String)tf.getValue());
        series.updatePoint(0, newValue);
    }
});
layout.addComponent(update);
```

19.4.11. Area and Column Range Charts

Ranged charts display an area or column between a minimum and maximum value, instead of a singular data point. They require the use of **RangeSeries**, as described in Section 19.6.3, “Range Series”. An area range is created with `AREARANGE` chart type, and a column range with `COLUMNRANGE` chart type.

Consider the following example:

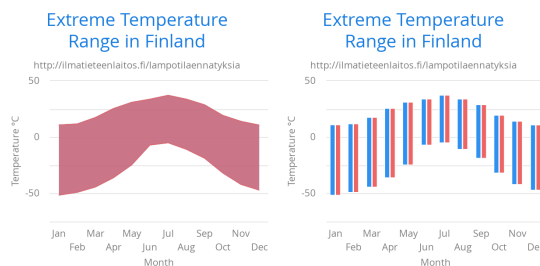
```
Chart chart = new Chart(ChartType.AREARANGE);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Extreme Temperature Range in Finland");
...

// Create the range series
// Source: http://ilmatieteenlaitos.fi/lampotilaennatyksia
RangeSeries series = new RangeSeries("Temperature Extremes",
    new Double[]{-51.5,10.9},
    new Double[]{-49.0,11.8},
    ...
    new Double[]{-47.0,10.8}); //
conf.addSeries(series);
```

The resulting chart, as well as the same chart with a column range, is shown in Figure 19.16, “Area and Column Range Chart”.

Figure 19.16. Area and Column Range Chart



19.4.12. Polar, Wind Rose, and Spiderweb Charts

Most chart types having two axes can be displayed in *polar* coordinates, where the X axis is curved on a circle and Y axis from the center of the circle to its rim. Polar chart is not a chart type in itself, but can be enabled for most chart types with `setPolar(true)` in the chart model parameters. Therefore all chart type specific features are usable with polar charts.

Vaadin Charts allows many sorts of typical polar chart types, such as *wind rose*, a polar column graph, or *spiderweb*, a polar chart with categorical data and a more polygonal visual style.

```
// Create a chart of some type
Chart chart = new Chart(ChartType.LINE);

// Enable the polar projection
Configuration conf = chart.getConfiguration();
conf.getChart().setPolar(true);
```

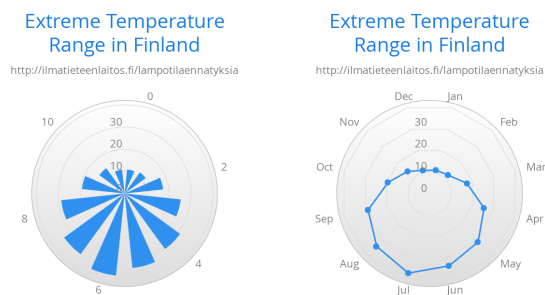
You need to define the sector of the polar projection with a **Pane** object in the configuration. The sector is defined as degrees from the north direction. You also need to define the value range for the X axis with `setMin()` and `setMax()`.

```
// Define the sector of the polar projection
Pane pane = new Pane(0, 360); // Full circle
conf.addPane(pane);

// Define the X axis and set its value range
XAxis axis = new XAxis();
axis.setMin(0);
axis.setMax(360);
```

The polar and spiderweb charts are illustrated in Figure 19.17, “Wind Rose and Spiderweb Charts”.

Figure 19.17. Wind Rose and Spiderweb Charts



Spiderweb Charts

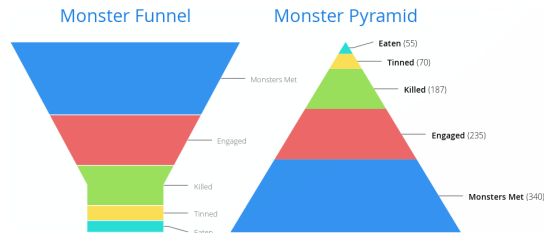
A *spiderweb* chart is a commonly used visual style of a polar chart with a polygonal shape rather than a circle. The data and the X axis should be categorical to make the polygonal interpolation meaningful. The sector is assumed to be full circle, so no angles for the pane need to be specified.

19.4.13. Funnel and Pyramid Charts

Funnel and pyramid charts are typically used to visualize stages in a sales processes, and for other purposes to visualize subsets of diminishing size. A funnel or pyramid chart has layers

much like a stacked column: in funnel from top-to-bottom and in pyramid from bottom-to-top. The top of the funnel has width of the drawing area of the chart, and diminishes in size down to a funnel "neck" that continues as a column to the bottom. A pyramid diminishes from bottom to top and does not have a neck.

Figure 19.18. Funnel and Pyramid Charts



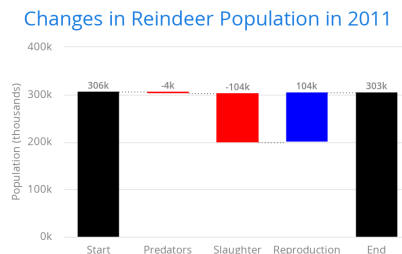
Funnels have chart type *FUNNEL*, pyramids have *PYRAMID*.

The labels of the funnel blocks are by default placed on the right side of the blocks, together with a connector. You can configure their style in the plot options.

19.4.14. Waterfall Charts

Waterfall charts are used for visualizing level changes from an initial level to a final level through a number of changes in the level. The changes are given as delta values, and you can have a number of intermediate totals, which are calculated automatically.

Figure 19.19. Waterfall Charts

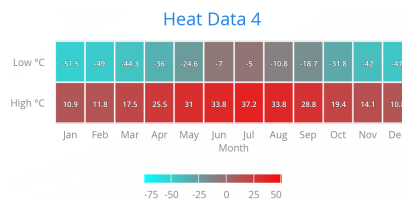


Waterfall charts have chart type *WATERFALL*.

19.4.15. Heat Maps

A heat map is a two-dimensional grid, where the color of a grid cell indicates a value.

Figure 19.20. Heat Maps

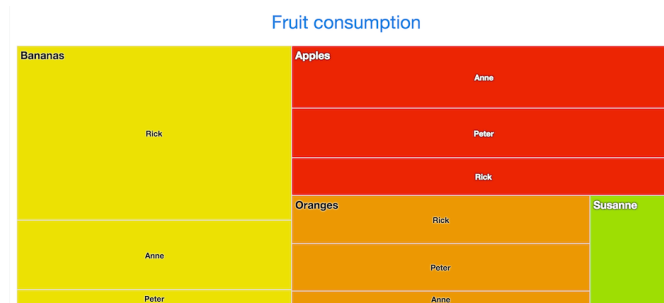


Heat maps have chart type *HEATMAP*.

19.4.16. Tree Maps

A tree map is used to display hierarchical data. It consists of a group of rectangles that contains other rectangles, where the size of a rectangle indicates the item value.

Figure 19.21. Tree Maps



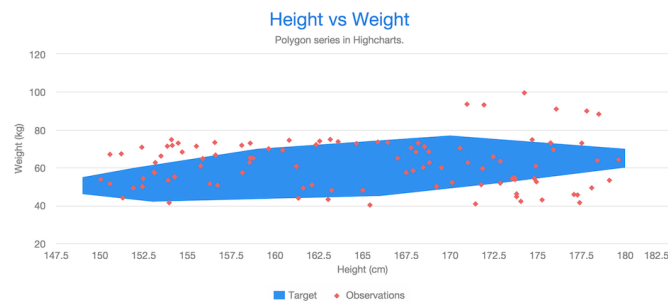
Tree maps have chart type `TREEMAP`.

19.4.17. Polygons

A polygon can be used to draw any freeform filled or stroked shape in the Cartesian plane.

Polygons consist of connected data points. The **DataSeries** or **ContainerSeries** are usually the most meaningful data series types for polygon charts. In both cases, the `x` and `y` properties should be set.

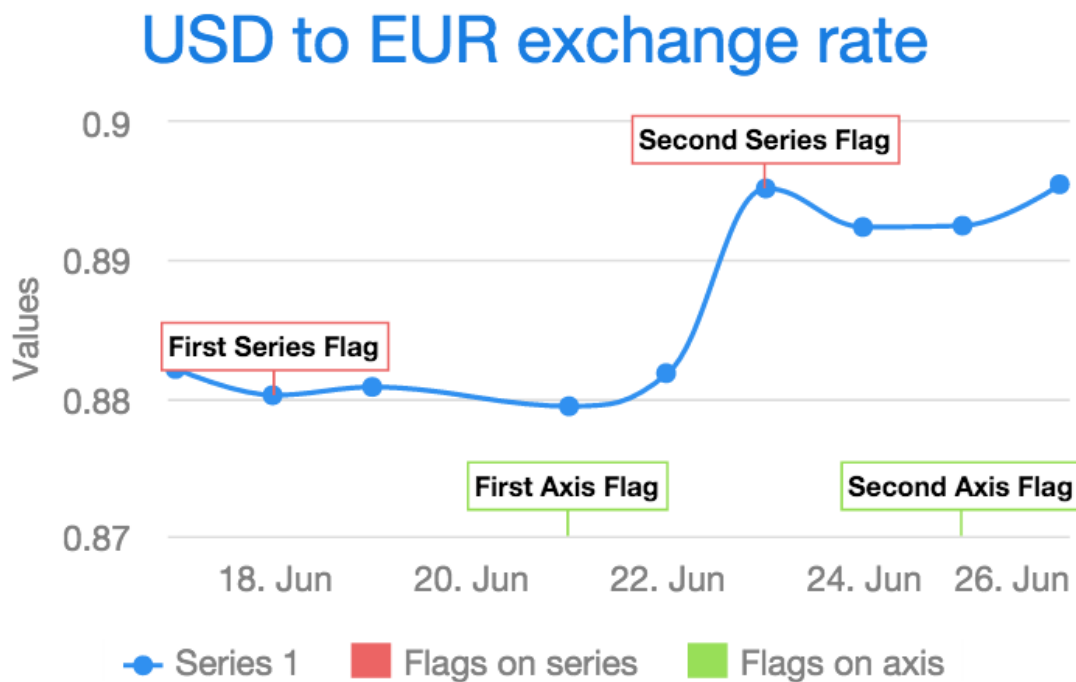
Figure 19.22. Polygon combined with Scatter



Polygons have chart type `POLYGON`.

19.4.18. Flags

Flags is a special chart type for annotating a series or the X axis with callout labels. Flags indicate interesting points or events on the series or axis. The flags are defined as items in a data series separate from the annotated series or axis.

Figure 19.23. Flags placed on an axis and a series

Flags are normally used in a chart that has one or more normal data series.

Plot Options

The flags are defined in a series that has its chart type specified by setting its plot options as **PlotOptionsFlags**. In addition to the common plot options properties, flag charts also have the following properties:

shape

defines the shape of the marker. It can be one of `FLAG`, `CIRCLEPIN`, `SQUAREPIN`, or `CALLOUT`.

stackDistance

defines the vertical offset between flags on the same value in the same series. Defaults to 12.

onSeries

defines the ID of the series where the flags should be drawn on. If no ID is given, the flags are drawn on the X axis.

onKey

in chart types that have multiple keys (Y values) for a data point, the property defines on which key the flag is placed. Line and column series have only one key, `y`. In range, OHLC, and candlestick series, the flag can be placed on the `open`, `high`, `low`, or `close` key. Defaults to `y`.

Data

The data for flags series require x and title properties, but can also have text property indicating the tooltip text. The easiest way to set these properties is to use **FlagItem**.

19.4.19. OHLC and Candlestick Charts

An Open-High-Low-Close (OHLC) chart displays the change in price over a period of time. The OHLC charts have chart type `OHLC`. An OHLC chart consist of vertical lines, each having a horizontal tickmark both on the left and the right side. The top and bottom ends of the vertical line indicate the highest and lowest prices during the time period. The tickmark on the left side of the vertical line shows the opening price and the tickmark on the right side the closing price.

Figure 19.24. OHLC Chart.



A candlestick chart is another way to visualize OHLC data. A candlestick has a body and two vertical lines, called *wicks*. The body represents the opening and closing prices. If the body is filled, the top edge of the body shows the opening price and the bottom edge shows the closing price. If the body is unfilled, the top edge shows the closing price and the bottom edge the opening price. In other words, if the body is filled, the opening price is higher than the closing price, and if not, lower. The upper wick represents the highest price during the time period and the lower wick represents the lowest price. A candlestick chart has chart type `CANDLESTICK`.

Figure 19.25. Candlestick Chart.

To attach data to an OHLC or a candlestick chart, you need to use a **DataSet** or a **ContainerSeries**. See Section 19.6, “Chart Data” for more details. A data series for an OHLC chart must contain **OhlcItem** objects. An **OhlcItem** contains a date and the open, highest, lowest, and close price on that date.

```
Chart chart = new Chart(ChartType.OHLC);
chart.setTimeline(true);

Configuration configuration = chart.getConfiguration();
configuration.getTitle().setText("AAPL Stock Price");
DataSet dataSet = new DataSet();
for (StockPrices.OhlcData data : StockPrices.fetchAaplOhlcPrice()) {
    OhlcItem item = new OhlcItem();
    item.setX(data.getDate());
    item.setLow(data.getLow());
    item.setHigh(data.getHigh());
    item.setClose(data.getClose());
    item.setOpen(data.getOpen());
    dataSet.add(item);
}
configuration.setSeries(dataSet);
chart.drawChart(configuration);
```

When using **ContainerSeries**, you need to specify the property IDs of the container properties containing OHLC values: `setXPropertyId()`, `setOpenPropertyId()`, `setClosePropertyId()`, `setHighPropertyId()`, and `setLowPropertyId()`.

```
// Create a container filled with stock price data
IndexedContainer container= initContainer();
int i=0;
for (StockPrices.OhlcData data : StockPrices.fetchAaplOhlcPrice()) {
    Item ie = container.addItem(i);
```

```

        ie.getItemProperty("x").setValue(data.getDate());
        ie.getItemProperty("low").setValue(data.getLow());
        ie.getItemProperty("high").setValue(data.getHigh());
        ie.getItemProperty("open").setValue(data.getOpen());
        ie.getItemProperty("close").setValue(data.getClose());
        i++;
    }

    // Wrap the container in a data series
    ContainerDataSeries dataSeries = new ContainerDataSeries(container);
    configuration.setSeries(dataSeries);
    dataSeries.setHighPropertyId("high");
    dataSeries.setLowPropertyId("low");
    dataSeries.setXPropertyId("x");
    dataSeries.setOpenPropertyId("open");
    dataSeries.setClosePropertyId("close");

    PlotOptionsOhlc plotOptionsOhlc = new PlotOptionsOhlc();
    plotOptionsOhlc.setTurboThreshold(0);
    dataSeries.setPlotOptions(plotOptionsOhlc);

```

Typically the OHLC and candlestick charts contain a lot of data, so it is useful to use them with the timeline feature enabled. The timeline feature is described in Section 19.8, “Timeline”.

Plot Options

You can use a **DataGrouping** object to configure data grouping properties. You set it in the plot options with `setDataGrouping()`. If the data points in a series are so dense that the spacing between two or more points is less than value of the `groupPixelWidth` property in the **DataGrouping**, the points will be grouped into appropriate groups so that each group is more or less two pixels wide. The `approximation` property in **DataGrouping** specifies which data point value should represent the group. The possible values are: `average`, `open`, `high`, `low`, `close`, and `sum`.

Using `setUpColor()` and `setUpLineColor()` allow setting the fill and border colors of the candlestick that indicate rise in the values. The default colors are white.

19.5. Chart Configuration

All the chart content configuration of charts is defined in a *chart model* in a **Configuration** object. You can access the model with the `getConfiguration()` method.

The configuration properties in the **Configuration** class are summarized in the following:

- `credits`: **Credits** (text, position, href, enabled)
- `labels`: **HTMLLabels** (html, style)
- `lang`: **Lang** (decimalPoint, thousandsSep, loading)
- `legend`: **Legend** (see Section 19.5.3, “Legend”)
- `pane`: **Pane**
- `plotoptions`: **PlotOptions** (see Section 19.5.1, “Plot Options”)
- `series`: **Series**

- `subTitle`: **SubTitle**
- `title`: **Title**
- `tooltip`: **Tooltip**
- `xAxis`: **XAxis** (see Section 19.5.2, “Axes”)
- `yAxis`: **YAxis** (see Section 19.5.2, “Axes”)

For data configuration, see Section 19.6, “Chart Data”.

19.5.1. Plot Options

The plot options are used to configure the data series in the chart. For example, line color could be specified for each line series. Plot options can be set in the configuration of the entire chart or for each data series separately with `setPlotOptions()`. When the plot options are set to the entire chart, it will be applied to all the series in the chart.

For example, the following enables stacking in column charts:

```
Chart chart = new Chart();
Configuration configuration = chart.getConfiguration();
PlotOptionsColumn plotOptions = new PlotOptionsColumn();
plotOptions.setStacking(Stacking.NORMAL);
configuration.setPlotOptions(plotOptions);
```

Chart can contain multiple plot options which can be added dynamically with `addPlotOptions()`.

The developer can specify also the plot options for the particular data series as follows:

```
ListSeries series = new ListSeries(50, 60, 70, 80);
PlotOptionsColumn plotOptions = new PlotOptionsColumn();
plotOptions.setStacking(Stacking.NORMAL);
series.setPlotOptions(plotOptions);
```

The plot options are defined in type-specific options classes or in a **PlotOptionsSeries** class which contains general options for all series types. Type specific classes are applied to all the series with the same type in the chart. If **PlotOptionsSeries** is used, it will be applied to all the series in the chart regardless of the type.

Chart types are divided into several groups with common properties. These groups are presented as abstract classes, that allow to use polymorphism for setting common properties for specific implementations. The abstract classes and groups are the following:

- **AreaOptions** **PlotOptionsArea**, **PlotOptionsArearange**, **PlotOptionsAreaspline**, **PlotOptionsAreasplinerange**
- **ColumnOptions** **PlotOptionsBar**, **PlotOptionsColumn**, **PlotOptionsColumnrange**
- **GaugeOptions** **PlotOptionsGauge**, **PlotOptionsSolidgauge**
- **PointOptions** **PlotOptionsLine**, **PlotOptionsSpline**, **PlotOptionsScatter**
- **PyramidOptions** **PlotOptionsPyramid**, **PlotOptionsFunnel**

- **OhlcOptions PlotOptionsOhlc, PlotOptionsCandlestick**

For example, to set the same `lineWidth` for **PlotOptionsLine** and **PlotOptionsSpline** use **PointOptions**.

```
private void setCommonProperties(PointOptions options) {
    options.setLineWidth(5);
    options.setColor(SolidColor.RED);
    options.setAnimation(false);
}
...
PlotOptionsSpline lineOptions = new PlotOptionsSpline();
PlotOptionsLine splineOptions = new PlotOptionsLine();
setCommonProperties(lineOptions);
setCommonProperties(splineOptions);
configuration.setPlotOptions(lineOptions, splineOptions);
```

See the API documentation of each chart type and its plot options class for more information about the chart-specific options.

Other Options

The following options are supported by some chart types.

width

Defines the width of the chart either by pixels or as a percentual proportion of the drawing area.

height

Defines the height of the chart either by pixels or as a percentual proportion of the drawing area.

depth

Specifies the thickness of the chart in 3D mode.

allowPointSelect

Specifies whether data points, in whatever way they are visualized in the particular chart type, can be selected by clicking on them. Defaults to *false*.

borderColor

Defines the border color of the chart elements.

borderWidth

Defines the width of the border in pixels.

center

Defines the center of the chart within the chart area by left and top coordinates, which can be specified either as pixels or as a percentage (as string) of the drawing area. The default is top 50% and left 50%.

slicedOffset

In chart types that support slices, such as pie and pyramid charts, specifies the offset for how far a slice is detached from other items. The amount is given in pixels and defaults to 10 pixels.

visible

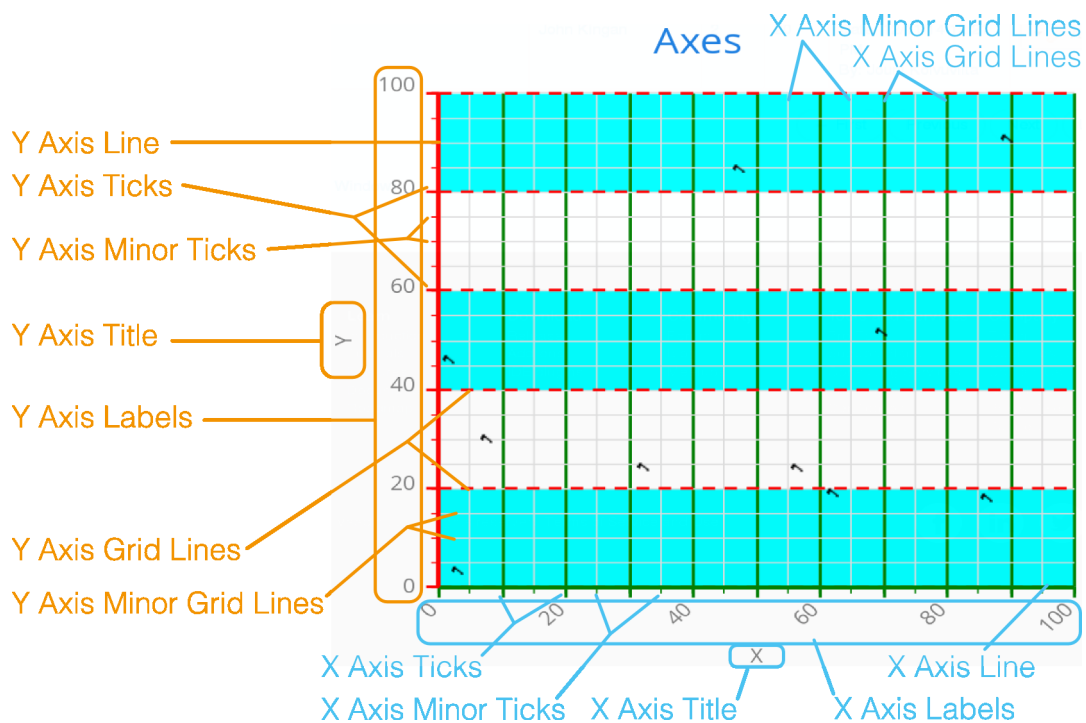
Specifies whether or not a chart is visible. Defaults to *true*.

19.5.2. Axes

Different chart types may have one, two, or three axes; in addition to X and Y axes, some chart types may have a color axis. These are represented by **XAxis**, **YAxis**, and **ColorAxis**, respectively. The X axis is usually horizontal, representing the iteration over the data series, and Y vertical, representing the values in the data series. Some chart types invert the axes and they can be explicitly inverted with `getChart().setInverted()` in the chart configuration. An axis has a caption and tick marks at intervals indicating either numeric values or symbolic categories. Some chart types, such as gauge, have only Y-axis, which is circular in the gauge, and some such as a pie chart have none.

The basic elements of X and Y axes are illustrated in Figure 19.26, “Chart Axis Elements”.

Figure 19.26. Chart Axis Elements



Axis objects are created and added to the configuration object with `addXAxis()` and `addYAxis()`.

```
XAxis xaxis = new XAxis();
xaxis.setTitle("Axis title");
conf.addXAxis(xaxis);
```

A chart can have more than one Y-axis, usually when different series displayed in a graph have different units or scales. The association of a data series with an axis is done in the data series object with `setYAxis()`.

For a complete reference of the many configuration parameters for the axes, please refer to the JavaDoc API documentation of Vaadin Charts.

Axis Type

Axes can be one of the following types, which you can set with `setType()`. The axis types are enumerated under **AxisType**. *LINEAR* is the default.

LINEAR(default)

For numeric values in linear scale.

LOGARITHMIC

For numerical values, as in the linear axis, but the axis will be scaled in the logarithmic scale. The minimum for the axis *must* be a positive non-zero value (`log(0)` is not defined, as it has limit at negative infinity when the parameter approaches zero).

DATETIME

Enables date/time mode in the axis. The date/time values are expected to be given either as a **Date** object or in milliseconds since the Java (or Unix) date epoch on January 1st 1970 at 00:00:00 GMT. You can get the millisecond representation of Java **Date** with `getTime()`.

CATEGORY

Enables using categorical data for the axis, as described in more detail later. With this axis type, the category labels are determined from the labels of the data points in the data series, without need to set them explicitly with `setCategories()`.

Categories

The axes display, in most chart types, tick marks and labels at some numeric interval by default. If the items in a data series have a symbolic meaning rather than numeric, you can associate *categories* with the data items. The category label is displayed between two axis tick marks and aligned with the data point. In certain charts, such as column chart, where the corresponding values in different data series are grouped under the same category. You can set the category labels with `setCategories()`, which takes the categories as (an ellipsis) parameter list, or as an iterable. The list should match the items in the data series.

```
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                  "Mars", "Jupiter", "Saturn",
                  "Uranus", "Neptune");
```

You can only set the category labels from the data point labels by setting the axis type to *CATEGORY*, as described earlier.

Labels

The axes display, in most chart types, tick marks and labels at some numeric interval by default. The format and style of labels in an axis is defined in a **Labels** object, which you can get with `getLabels()` from the axis.

```
XAxis xaxis = new XAxis();
...
Labels xlabel = xaxis.getLabels();
xlabel.setAlign(HorizontalAlign.CENTER); // Default
xlabel.getStyle().setColor(SolidColor.GREEN);
xlabel.getStyle().setFontWeight(FontWeight.BOLD);
xlabel.setRotation(-45);
xlabel.setStep(2); // Every 2 major tick
```

Axis labels have the following configuration properties:

align

Defines the alignment of the labels relative to the centers of the ticks. On left alignment, the left edges of labels are aligned at the tickmarks, and correspondingly the right side on right alignment. The default is determined automatically based on the direction of the axis and rotation of the labels.

distance(only in polar charts)

Distance of labels from the perimeter of the plot area, in pixels.

enabled

Whether labels are enabled or not. Defaults to *true*.

format

Formatting string for labels, as described in Section 19.5.4, “Formatting Labels”. Defaults to “{value}”.

formatter

A JavaScript formatter for the labels, as described in Section 19.5.4, “Formatting Labels”. The value is available in the *this.value* property. The *this* object also has *axis*, *chart*, *isFirst*, and *isLast* properties. Defaults to:

```
function() {return this.value;}
```

maxStaggerLines(only horizontal axis)

When labels on the horizontal (usually X) axis are displayed so densely that they would overlap, they are automatically placed on alternating lines in “staggered” fashion. When number of lines is not set manually with *staggerLines*, this parameter defines the maximum number of such lines; value 1 disables automatic staggering. Default is 5 lines.

rotation

Defines rotation of labels in degrees. A positive value indicates rotation in clockwise direction. Labels are rotated at their alignment point. Defaults to 0.

```
Labels xlabels = xaxis.getLabels();
xlabels.setAlign(HorizontalAlign.RIGHT);
xlabels.setRotation(-45); // Tilt 45 degrees CCW
```

staggerLines

Defines number of lines for placing the labels to avoid overlapping. By default undefined, and the number of lines is automatically determined up to *maxStaggerLines*.

step

Defines tick interval for showing labels, so that labels are shown at every *n*th tick. The default step is automatically determined, along with staggering, to avoid overlap.

```
Labels xlabels = xaxis.getLabels();
xlabels.setStep(2); // Every 2 major tick
```

style

Defines style for labels. The property is a **Style** object, which has to be created and set.

```
Labels xlabels = xaxis.getLabels();
Style xlabelsstyle = new Style();
xlabelsstyle.setColor(SolidColor.GREEN);
xlabels.setStyle(xlabelsstyle);
```

useHTML

Allows using HTML in custom label formats. Otherwise, HTML is quoted. Defaults to false.

x,y

Offsets for the label's position, relative to the tick position. X offset defaults to 0, but Y to null, which enables automatic positioning based on font size.

Gauge, pie, and polar charts allow additional properties.

For a complete reference of the many configuration parameters for the labels, please refer to the JavaDoc API documentation of Vaadin Charts.

Axis Range

The axis range is normally set automatically to fit the data, but can also be set explicitly. The *extremes* property in the axis configuration defines the minimum and maximum values of the axis range. You can set them either individually with `setMin()` and `setMax()`, or together with `setExtremes()`. Changing the extremes programmatically requires redrawing the chart with `drawChart()`.

19.5.3. Legend

The legend is a box that describes the data series shown in the chart. It is enabled by default and is automatically populated with the names of the data series as defined in the series objects, and the corresponding color symbol of the series.

alignment

Specifies the horizontal alignment of the legend box within the chart area. Defaults to `HorizontalAlign.CENTER`.

enabled

Enables or disables the legend. Defaults to `true`.

layout

Specifies the layout direction of the legend items. Defaults to `LayoutDirection.HORIZONTAL`.

title

Specifies the title of the legend.

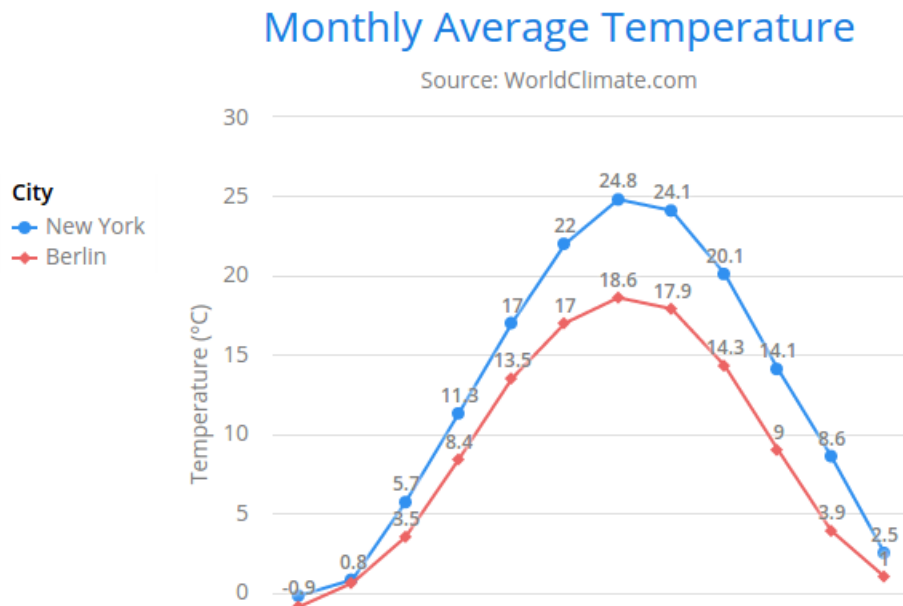
verticalAlign

Specifies the vertical alignment of the legend box within the chart area. Defaults to `VerticalAlign.BOTTOM`.

```
Legend legend = configuration.getLegend();
legend.getTitle().setText("City");
legend.setLayout(LayoutDirection.VERTICAL);
legend.setAlign(HorizontalAlign.LEFT);
legend.setVerticalAlign(VerticalAlign.TOP);
```

The result can be seen in Figure 19.27, “Legend example”.

Figure 19.27. Legend example



19.5.4. Formatting Labels

Data point values, tooltips, and tick labels are formatted according to formatting configuration for the elements, with configuration properties described earlier for each element. Formatting can be set up in the overall configuration, for a data series, or for individual data points. The format can be defined either by a format string or by JavaScript formatter, which are described in the following.

Using Format Strings

A formatting string contain free-form text mixed with variables. Variables are enclosed in brackets, such as "Here {point.y} is a value at {point.x}". In different contexts, you have at least the following variables available:

- *value* in axis labels
- *point.x*, *point.x* in data points and tooltips
- *series.name* in data points and tooltips
- *series.color* in data points and tooltips

Values can be formatted according to a formatting string, separated from the variable name by a colon.

For numeric values, a subset of C printf formatting specifiers is supported. For example, "{point.y:%02.2f}" would display a floating-point value with two decimals and two leading zeroes, such as 02.30.

For dates, you can use a subset of PHP `strftime()` formatting specifiers. For example, "`{value:%Y-%m-%d %H:%M:%S}`" would format a date and time in the ISO 8601 format.

Using a JavaScript Formatter

A JavaScript formatter is given in a string that defines a JavaScript function that returns the formatted string. The value to be formatted is available in `this.value` for axis labels, or `this.x`, `this.y` for data points.

For example, to format tick labels on a chart axis, you could have:

```
YAxis yaxis = new YAxis();
Labels ylabels = yaxis.getLabels();
ylabels.setFormatter("function() {return this.value + ' km';}");
```

Simplified Formatting

Some contexts that display labels allow defining simple formatting for the labels. For example, data point tooltips allow defining prefix, suffix, and floating-point precision for the values.

19.6. Chart Data

Chart data is stored in a data series model that contains information about the visual representation of the data points in addition to their values. There are a number of different types of series - **DataSeries**, **ListSeries**, **AreaListSeries**, and **RangeSeries**.

19.6.1. List Series

The **ListSeries** is essentially a helper type that makes the handling of simple sequential data easier than with **DataSeries**. The data points are assumed to be at a constant interval on the X axis, starting from the value specified with the `pointStart` property (default is 0) at intervals specified with the `pointInterval` property (default is 1.0). The two properties are defined in the **PlotOptions** for the series.

The Y axis values are given in a **List<Number>**, or with ellipsis or an array.

```
ListSeries series = new ListSeries(
    "Total Reindeer Population",
    181091, 201485, 188105, ...);
PlotOptionsLine plotOptions = new PlotOptionsLine();
plotOptions.setPointStart(1959);
series.setPlotOptions(plotOptions);
conf.addSeries(series);
```

You can also add them one by one with the `addData()` method, which is typical when converting from some other representation.

```
// Original representation
int data[][] = reindeerData();

// Create a list series with X values starting from 1959
ListSeries series = new ListSeries("Reindeer Population");
PlotOptionsLine plotOptions = new PlotOptionsLine();
plotOptions.setPointStart(1959);
series.setPlotOptions(plotOptions);
```

```
// Add the data points
for (int row[]: data)
    series.addData(row[1]);

conf.addSeries(series);
```

If the chart has multiple Y axes, you can specify the axis for the series by its index number with `setYAxis()`.

19.6.2. Generic Data Series

The **DataSeries** can represent a sequence of data points at an interval as well as scatter data. Data points are represented with the **DataSeriesItem** class, which has *x* and *y* properties for representing the data value. Each item can be given a category name.

```
DataSet series = new DataSet();
series.setName("Total Reindeer Population");
series.add(new DataSetItem(1959, 181091));
series.add(new DataSetItem(1960, 201485));
series.add(new DataSetItem(1961, 188105));
series.add(new DataSetItem(1962, 177206));

// Modify the color of one point
series.get(2).getMarker().setFillColor(SolidColor.RED);
conf.addSeries(series);
```

Data points are associated with some visual representation parameters: marker style, selected state, legend index, and dial style (for gauges). Most of them can be configured at the level of individual data series items, the series, or in the overall plot options for the chart. The configuration options are described in Section 19.5, “Chart Configuration”. Some parameters, such as the *sliced* option for pie charts is only meaningful to configure at item level.

Adding and Removing Data Items

New **DataSetItem** items are added to a series with the `add()` method. The basic method takes just the data item, but the other method takes also two boolean parameters. If the *updateChart* parameter is *false*, the chart is not updated immediately. This is useful if you are adding many points in the same request.

The *shift* parameter, when *true*, causes removal of the first data point in the series in an optimized manner, thereby allowing an animated chart that moves to left as new points are added. This is most meaningful with data with even intervals.

You can remove data points with the `remove()` method in the series. Removal is generally not animated, unless a data point is added in the same change, as is caused by the *shift* parameter for the `add()`.

Updating Data Items

If you update the properties of a **DataSetItem** object, you need to call `update()` method for the series with the item as the parameter. Changing the coordinates of a data point in this way causes animation of the change.

Range Data

Range charts expect the Y values to be specified as minimum-maximum value pairs. The **DataSeriesItem** provides `setLow()` and `setHigh()` methods to set the minimum and maximum values of a data point, as well as a number of constructors that accept the values.

```
RangeSeries series =
    new RangeSeries("Temperature Extremes");

// Give low-high values in constructor
series.add(new DataSeriesItem(0, -51.5, 10.9));
series.add(new DataSeriesItem(1, -49.0, 11.8));

// Set low-high values with setters
DataSeriesItem point = new DataSeriesItem();
point.setX(2);
point.setLow(-44.3);
point.setHigh(17.5);
series.add(point);
```

The **RangeSeries** offers a slightly simplified way of adding ranged data points, as described in Section 19.6.3, “Range Series”.

19.6.3. Range Series

The **RangeSeries** is a helper class that extends **DataSeries** to allow specifying interval data a bit easier, with a list of minimum-maximum value ranges in the Y axis. You can use the series in range charts, as described in Section 19.4.11, “Area and Column Range Charts”.

For X axis, the coordinates are generated at fixed intervals starting from the value specified with the `pointStart` property (default is 0) at intervals specified with the `pointInterval` property (default is 1.0).

Setting the Data

The data in a **RangeSeries** is given as an array of minimum-maximum value pairs for the Y value axis. The pairs are also represented as arrays. You can pass the data using the ellipsis in the constructor or the `setData()`:

```
RangeSeries series =
    new RangeSeries("Temperature Ranges",
        new Double[]{-51.5, 10.9},
        new Double[]{-49.0, 11.8},
        ...
        new Double[]{-47.0, 10.8});
conf.addSeries(series);
```

Or, as always with variable arguments, you can also pass them in an array, in the following for the `setData()`:

```
series.setRangeData(new Double[][] {
    new Double[]{-51.5, 10.9},
    new Double[]{-49.0, 11.8},
    ...
    new Double[]{-47.0, 10.8}});
```

19.6.4. Container Data Series

The **ContainerDataSeries** is an adapter for binding Vaadin `Container` data sources to charts. The container needs to have properties that define the name, X-value, and Y-value of a data point. The default property IDs of the three properties are "name", "x", and "y", respectively. You can set the property IDs with `setNamePropertyId()`, `setYPropertyId()`, and `setXPropertyId()`, respectively. If the container has no x property, the data is assumed to be categorical.

In the following example, we have a **BeanItemContainer** with **Planet** items, which have a `name` and `diameter` property. We display the container data both in a Vaadin **Table** and a chart.

```
// The data
BeanItemContainer<Planet> container =
    new BeanItemContainer<Planet>(Planet.class);
container.addBean(new Planet("Mercury", 4900));
container.addBean(new Planet("Venus", 12100));
container.addBean(new Planet("Earth", 12800));
...

// Display it in a table
Table table = new Table("Planets", container);
table.setPageLength(container.size());
table.setVisibleColumns("name", "diameter");
layout.addComponent(table);

// Display it in a chart
Chart chart = new Chart(ChartType.COLUMN);
... Configure it ...

// Wrap the container in a data series
ContainerDataSeries series =
    new ContainerDataSeries(container);

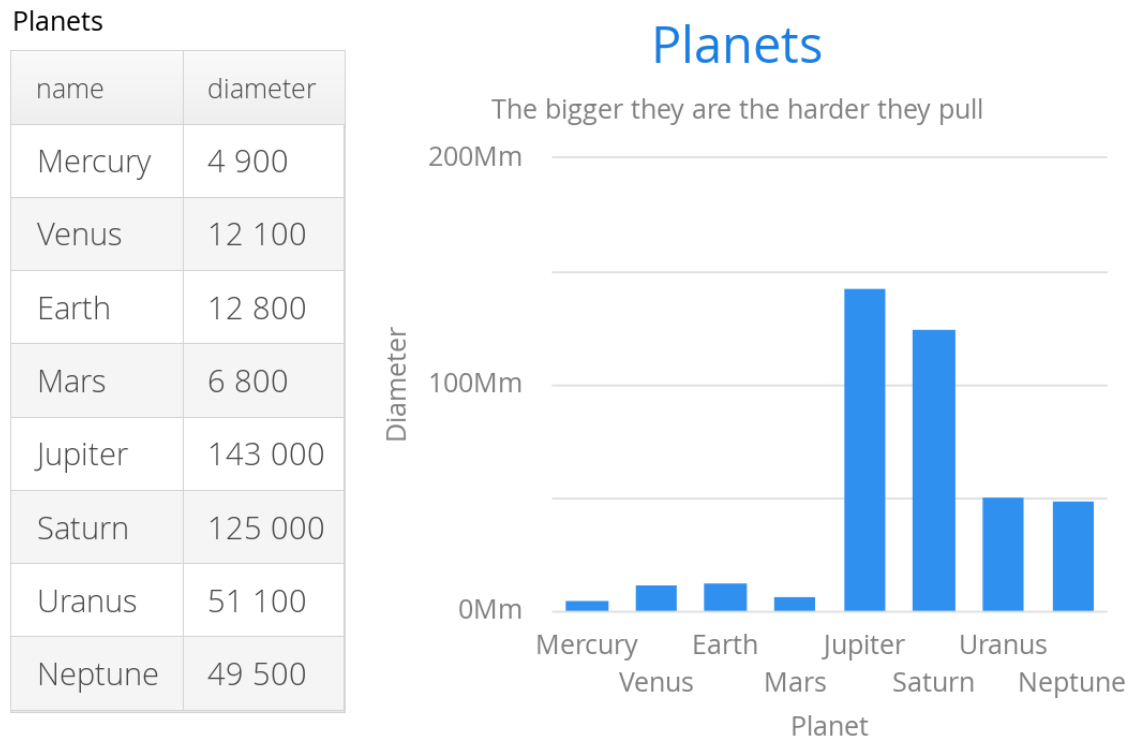
// Set up the name and Y properties
series.setNamePropertyId("name");
series.setYPropertyId("diameter");

conf.addSeries(series);
```

As the X axis holds categories rather than numeric values, we need to set up the category labels with an array of string. There are a few ways to do that, some more efficient than others, below is one way:

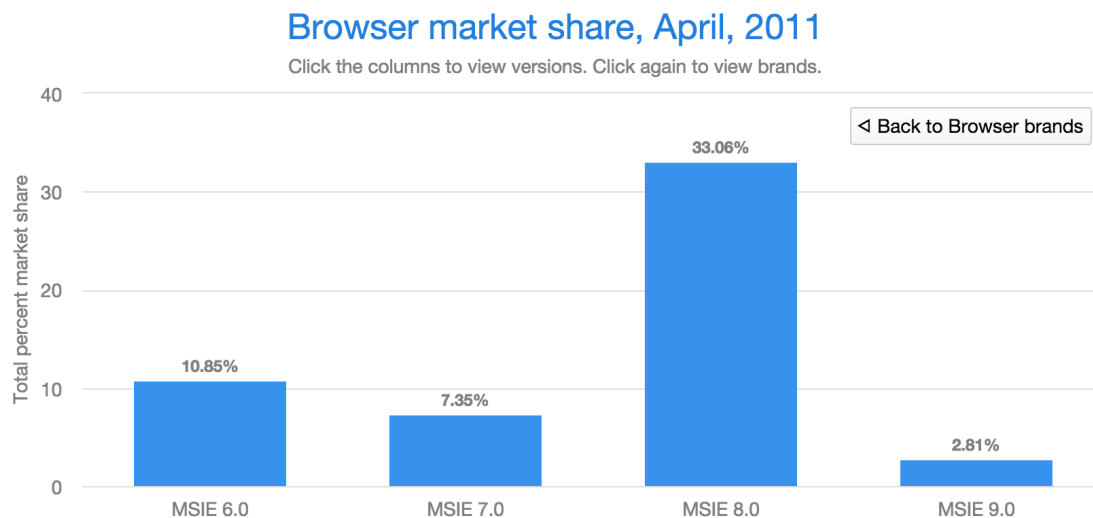
```
// Set the category labels on the axis correspondingly
XAxis xaxis = new XAxis();
String names[] = new String[container.size()];
List<Planet> planets = container.getItemIds();
for (int i=0; i<planets.size(); i++)
    names[i] = planets.get(i).getName();
xaxis.setCategories(names);
xaxis.setTitle("Planet");
conf.addXAxis(xaxis);
```

The result can be seen in Figure 19.28, "Table and Chart Bound to a Container".

Figure 19.28. Table and Chart Bound to a Container

19.6.5. Drill-Down

Vaadin Charts allows drilling down from a chart to a more detailed view by clicking an item in the top-level view. To enable the feature, you need to provide a separate data series for each of the detailed views by calling the `addItemWithDrilldown()` method. When the user clicks on a drill-down item, the current series is animated into the the linked drill-down series. A customizable back button is provided to navigate back to the main series, as shown in Figure 19.29, “Detailed series after a drill-down”.

Figure 19.29. Detailed series after a drill-down

There are two ways to use drill-down: synchronous and asynchronous.

Synchronous

When using synchronous drill-down, you provide the top-level series and all the series below it beforehand. The data is transferred to the client-side at the same time and no client-server communication needs to happen for the drill-down. The drill-down series must have an identifier, set with `setId()`, as shown below.

```

DataSeries series = new DataSeries();

DataSeriesItem mainItem = new DataSeriesItem("MSIE", 55.11);

DataSeries drillDownSeries = new DataSeries("MSIE versions");
drillDownSeries.setId("MSIE");

drillDownSeries.add(new DataSeriesItem("MSIE 6.0", 10.85));
drillDownSeries.add(new DataSeriesItem("MSIE 7.0", 7.35));
drillDownSeries.add(new DataSeriesItem("MSIE 8.0", 33.06));
drillDownSeries.add(new DataSeriesItem("MSIE 9.0", 2.81));

series.addItemWithDrilldown(mainItem, drillDownSeries);

```

Asynchronous

When using asynchronous drill-down, you omit the drill-down series parameter. Instead, you provide a callback method with `Chart.setDrillDownCallback()`. When the user clicks an item in the series, the callback is called to provide a drill-down series.

```

DataSeries series = new DataSeries();

DataSeriesItem mainItem = new DataSeriesItem("MSIE", 55.11);

series.addItemWithDrilldown(mainItem);

chart.setDrilldownCallback(new DrilldownCallback() {

```

```
@Override
public Series handleDrilldown(DrilldownEvent event) {
    DataSeries drillDownSeries = new DataSeries("MSIE versions");

    drillDownSeries.add(new DataSeriesItem("MSIE 6.0", 10.85));
    drillDownSeries.add(new DataSeriesItem("MSIE 7.0", 7.35));
    drillDownSeries.add(new DataSeriesItem("MSIE 8.0", 33.06));
    drillDownSeries.add(new DataSeriesItem("MSIE 9.0", 2.81));

    return drillDownSeries;
}
});
```

You can use the event to decide what kind of series you want to return. The event contains, for example, a reference to the item that was clicked. Note that the same callback is used for all items. The callback can also return null. Returning null will not trigger a drilldown.

19.7. Advanced Uses

19.7.1. Server-Side Rendering and Exporting

In addition to using charts in Vaadin UIs, you may also need to provide them as images or in downloadable documents. Vaadin Charts can be rendered on the server-side using a headless JavaScript execution environment, such as PhantomJS.

Vaadin Charts supports a HighCharts remote export service, but the SVG Generator based on PhantomJS is almost as easy to use and allows much more powerful uses.

Using a Remote Export Service

Vaadin Charts has a simple built-in export functionality that does the export in a remote export server. Vaadin Charts provides a default export service, but you can also configure your own.

You can enable the built-in export function by setting `setExporting(true)` in the chart configuration.

```
chart.getConfiguration().setExporting(true);
```

To configure it further, you can provide a **Exporting** object with custom settings.

```
// Create the export configuration
Exporting exporting = new Exporting(true);

// Customize the file name of the download file
exporting.setFilename("mychartfile.pdf");

// Use the exporting configuration in the chart
chart.getConfiguration().setExporting(exporting);
```

The functionality uses a HighCharts export service by default. To use your own, you need to set up one and then configure it in the exporting configuration as follows:

```
exporting.setUrl("http://my.own.server.com");
```

Using the SVG Generator

The **SVGGenerator** in Vaadin Charts provides an advanced way to render the Chart into SVG format on the server-side. SVG is well supported by many applications, can be converted to virtually any other graphics format, and can be passed to PDF report generators.

The generator uses PhantomJS to render the chart on the server-side. You need to install it from phantomjs.org. After installation, PhantomJS should be in your system path. If not, you can set the `phantom.exec` system property for the JRE to point to the PhantomJS binary.

To generate the SVG image content as a string (it's XML), simply call the `generate()` method in the **SVGGenerator** singleton and pass it the chart configuration.

```
String svg = SVGGenerator.getInstance()  
    .generate(chart.getConfiguration());
```

You can then use the SVG image as you like, for example, for download from a **StreamResource**, or include it in a HTML, PDF, or other document. You can use SVG tools such as the Batik or iText libraries to generate documents. For a complete example, you can check out the Charts Export Demo from the Subversion repository at <https://github.com/vaadin/charts/tree/master/chart-export-demo>.

19.8. Timeline

A charts timeline feature allows selecting different time ranges for which to display the chart data, as well as navigating between such ranges. It is especially useful when working with large time Section 19.3.3, “Chart Data Series”. Adding a timeline to your chart is very easy - just set the 'timeline' property to 'true', that is, call `setTimeline(true)`. You can enable the timeline in a chart that displays one or more time series. Most of the chart types support the timeline. There are few exceptions which are listed here: Section 19.4.8, “Pie Charts”, Section 19.4.9, “Gauges”, Section 19.4.10, “Solid Gauges”, Section 19.4.13, “Funnel and Pyramid Charts”, and Section 19.4.13, “Funnel and Pyramid Charts”. Enabling the timeline in these chart types will raise a runtime exception.

You can change the time range using the navigator at the bottom of the chart. To be able to use the navigator, the X values of the corresponding data series should be of the type **Date**. Also integer values can be used, in which case they are interpreted as milliseconds since the 01/01/1970 epoch. If you have multiple series, the first one is presented in the navigator.

Figure 19.30. Vaadin chart with a timeline.

Another way to change the time range is to use the range selector. The range selector includes a set of predefined time ranges for easier navigation, for example, 1 month, 3 month, 6 month etc. To specify a custom time range, you can use range selector text fields for setting start and end of the time interval.

You can configure the range navigator and selector in the chart configuration. To show or hide the navigator, call `setEnabled()`. You can use **Navigator** and **PlotOptionsSeries** to change the appearance of the navigator.

```
Navigator navigator = configuration.getNavigator();
navigator.setEnabled(true);
navigator.setMargin(75);
PlotOptionsSeries plotOptions=new PlotOptionsSeries();
plotOptions.setColor(SolidColor.BROWN);
navigator.setSeries(plotOptions);
```

You can change the style of the range selector buttons with the `setButtonTheme(theme)` method and specify the index of the button to appear pre-selected with the `setSelected(index)` method.

```
RangeSelector rangeSelector = new RangeSelector();
rangeSelector.setSelected(4);
ButtonTheme theme = new ButtonTheme();
Style style = new Style();
style.setColor(new SolidColor("#0766d8"));
style.setFontWeight(FontWeight.BOLD);
theme.setStyle(style);
rangeSelector.setButtonTheme(theme);
```

```

Chart chart = new Chart();
chart.setTimeline(true);
Configuration configuration = chart.getConfiguration();
configuration.setRangeSelector(rangeSelector);
chart.drawChart(configuration);

```

You can customize the date format for the time range input fields by specifying formatter strings for displaying and editing the dates, as well as a corresponding JavaScript parser function to parse edited values:

```

RangeSelector rangeSelector = new RangeSelector();
rangeSelector.setInputDateFormat("%YYYY-%MM-%DD:%H:%M");
rangeSelector.setInputEditDateFormat("%YYYY-%MM-%DD:%H:%M");
rangeSelector.setInputDateParser(
    "function(value) {" +
    "    value = value.split(/[:\\-]/);\\n" +
    "    return Date.UTC(\\n" +
    "        parseInt(value[0], 10),\\n" +
    "        parseInt(value[1], 10),\\n" +
    "        parseInt(value[2], 10),\\n" +
    "        parseInt(value[3], 10),\\n" +
    "        parseInt(value[4], 10),\\n" +
    "    );}" );
configuration.setRangeSelector(rangeSelector);

```

Timeline charts allow comparing the charts series against each other. Setting the compare property to either Compare.PERCENT or Compare.VALUE will show the difference between charts data series in percentage or absolute values respectively.

```

PlotOptionsSeries plotOptions = new PlotOptionsSeries();
plotOptions.setCompare(Compare.PERCENT);
configuration.setPlotOptions(plotOptions);

```

Figure 19.31. Vaadin chart with a percentage comparison between series.



You can find more examples in the Timeline section of Vaadin Charts Demo.

Chapter 20

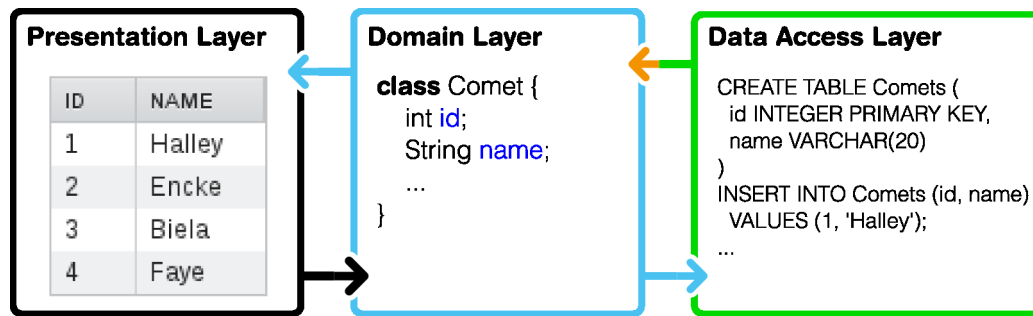
Vaadin JPAContainer

20.1. Overview	553
20.2. Installing	556
20.3. Defining a Domain Model	560
20.4. Basic Use of JPAContainer	563
20.5. Entity Providers	568
20.6. Filtering JPAContainer	571
20.7. Querying with the Criteria API	572
20.8. Automatic Form Generation	573
20.9. Using JPAContainer with Hibernate	575

This chapter describes the use of the Vaadin JPAContainer add-on.

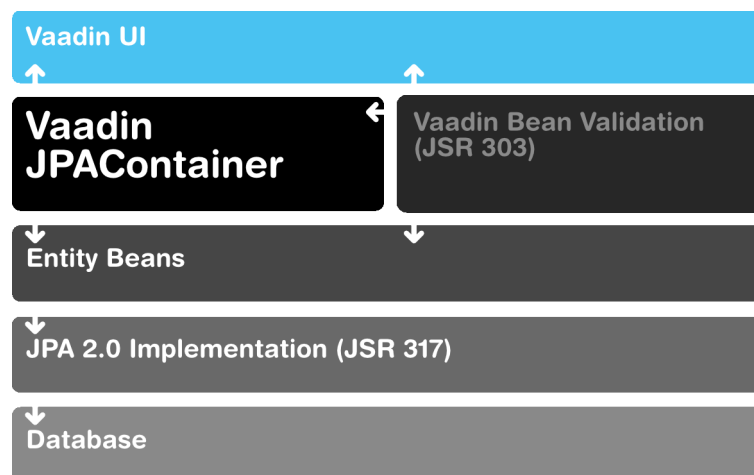
20.1. Overview

Vaadin JPAContainer add-on makes it possible to bind user interface components to a database easily using the Java Persistence API (JPA). It is an implementation of the `Container` interface described in Section 10.5, “Collecting Items in Containers”. It supports a typical three-layer application architecture with an intermediate *domain model* between the user interface and the data access layer.

Figure 20.1. Three-Layer Architecture Using JPAContainer And JPA

The role of Java Persistence API is to handle persisting the domain model in the database. The database is typically a relational database. Vaadin JPAContainer binds the user interface components to the domain model and handles database access with JPA transparently.

JPA is really just an API definition and has many alternative implementations. Vaadin JPAContainer supports especially EclipseLink, which is the reference implementation of JPA, and Hibernate. Any other compliant implementation should work just as well. The architecture of an application using JPAContainer is shown in Figure 20.2, “JPAContainer Architecture”.

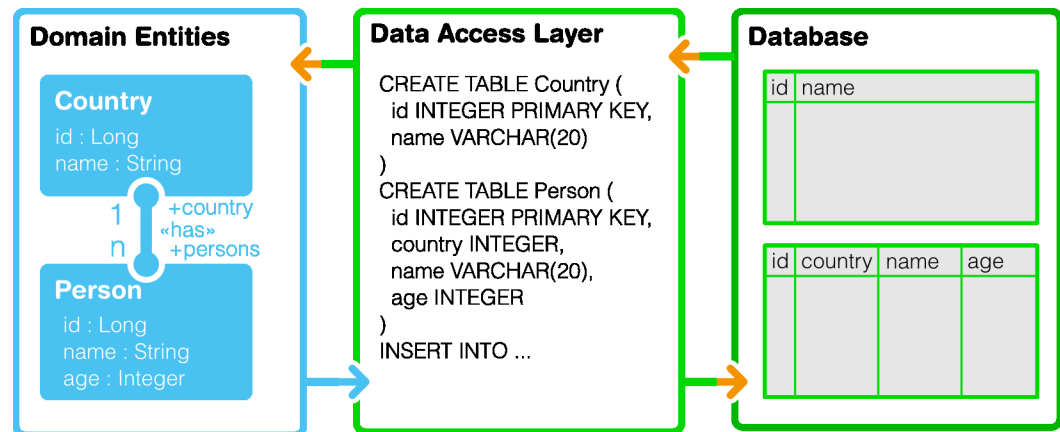
Figure 20.2. JPAContainer Architecture

Vaadin JPAContainer also plays together with the Vaadin support for Java Bean Validation (JSR 303).

20.1.1. Java Persistence API

Java Persistence API (JPA) is an API for object-relational mapping (ORM) of Java objects to a relational database. In JPA and entity-relationship modeling in general, a Java class is considered an *entity*. Class (or entity) instances correspond with a row in a database table and member variables of a class with columns. Entities can also have relationships with other entities.

The object-relational mapping is illustrated in Figure 20.3, “Object-Relational Mapping” with two entities with a one-to-many relationship.

Figure 20.3. Object-Relational Mapping

The entity relationships are declared with metadata. With Vaadin JPAContainer, you provide the metadata with annotations in the entity classes. The JPA implementation uses reflection to read the annotations and defines a database model automatically from the class definitions. Definition of the domain model and the annotations are described in Section 20.3.1, “Persistence Metadata”.

The main interface in JPA is the `EntityManager`, which allows making different kinds of queries either with the Java Persistence Query Language (JPQL), native SQL, or the Criteria API in JPA 2.0. You can always use the interface directly as well, using Vaadin JPAContainer only for binding the data to the user interface.

Vaadin JPAContainer supports JPA 2.0 (JSR 317). It is available under the Apache License 2.0.

20.1.2. JPAContainer Concepts

The **JPAContainer** is an implementation of the `Vaadin Container` interface that you can bind to user interface components such as **Table**, **ComboBox**, etc.

The data access to the persistent entities is handled with a *entity provider*, as defined in the `EntityProvider` interface. JPAContainer provides a number of different entity providers for different use cases and optimizations. The built-in providers are described in Section 20.5, “Entity Providers”.

JPAContainer is by default *unbuffered*, so that any entity property changes are written immediately to the database when you call `setValue()` for a property, or when a user edits a bound field. A container can be set as *buffered*, so that changes are written on calling `commit()`. Buffering can be done both at item level, such as when updating item property values, or at container level, such as when adding or deleting items. Only *batchable* containers, that is, containers with a batchable entity provider, can be buffered. Note that buffering is recommended for situations where two users could update the same entity simultaneously, and when this would be a problem. In an unbuffered container, the entity is refreshed before writing an update, so the last write wins and a conflicting simultaneous update written before it is lost. A buffered container throws an **OptimisticLockException** when two users edit the same item (an unbuffered container never throws it), thereby allowing to handle the situation with application logic.

20.1.3. Documentation and Support

In addition to this chapter in the book, the installation package includes the following documentation about JPAContainer:

- [API Documentation](#)
- [JPAContainer Tutorial](#)
- [JPAContainer AddressBook Demo](#)
- [JPAContainer Demo](#)

20.2. Installing

Vaadin JPAContainer can be installed either as an installation package, downloaded from the Vaadin Directory, or as a Maven dependency. You can also create a new JPAContainer-enabled Vaadin project using a Maven archetype.

20.2.1. Downloading the Package

Vaadin JPAContainer is available for download from the Vaadin Directory. Please see Section 18.2, “Downloading Add-ons from Vaadin Directory” for basic instructions for downloading from Directory. The download page also gives the dependency declaration needed for retrieving the library with Maven.

JPAContainer is a purely server-side component, so it does not include a widget set that you would need to compile.

20.2.2. Installation Package Content

Once extracted to a local folder, the contents of the installation directory are as follows:

README

A readme file describing the package contents.

LICENSE

The full license text for the library.

vaadin-jpacontainer-3.x.x.jar

The actual Vaadin JPAContainer library.

vaadin-jpacontainer-3.x.x-sources.jar

Source JAR for the library. You can use it for example in Eclipse by associating the JavaDoc JAR with the JPAContainer JAR in the build path settings of your project.

jpacontainer-tutorial.pdf

The tutorial in PDF format.

jpacontainer-tutorial-html

The tutorial in HTML format.

jpacontainer-addressbook-demo

The JPAContainer AddressBook Demo project covered in this tutorial. You can compile and package the application as a WAR with "**mvn** *package*" or launch it in the Jetty web browser with "**mvn** *jetty:run*". You can also import the demo project in Eclipse.

20.2.3. Downloading with Maven

The download page in Vaadin Directory gives the dependency declaration needed for retrieving the Vaadin JPAContainer library with Maven.

```
<dependency>
  <groupId>com.vaadin.addon</groupId>
  <artifactId>jpacontainer</artifactId>
  <version>3.2.0</version>
</dependency>
```

Use the `LATEST` version tag to automatically download the latest stable release or use a specific version number as done above.

See Section 18.4, “Using Add-ons in a Maven Project” for detailed instructions for using a Vaadin add-on with Maven.

Using the Maven Archetype

If you wish to create a new JPAContainer-enabled Vaadin project with Maven, you can use the `vaadin-archetype-jpacontainer` archetype. Please see Section 3.5, “Creating a Project with Maven” for details on creating a Vaadin project with a Maven archetype.

20.2.4. Including Libraries in Your Project

The Vaadin JPAContainer JAR must be included in the library folder of the web application. It is located in `WEB-INF/lib` path in a web application. In a normal Eclipse web projects the path is `WebContent/WEB-INF/lib`. In Maven projects the JARs are automatically included in the folder, as long as the dependencies are defined correctly.

You will need the following JARs:

- Vaadin Framework Library
- Vaadin JPAContainer
- Java Persistence API 2.0 (javax.persistence package)
- JPA implementation (EclipseLink, Hibernate, ...)
- Database driver or embedded engine (H2, HSQLDB, MySQL, PostgreSQL, ...)

If you use Eclipse, the Vaadin Framework library is automatically downloaded and updated by the Vaadin Plugin for Eclipse.

To use bean validation, you need an implementation of the Bean Validation, such as Hibernate Validator.*//TODO elaborate*

20.2.5. Persistence Configuration

Persistence configuration is done in a `persistence.xml` file. In a regular Eclipse project, it should be located in `WebContent/WEB-INF/classes/META-INF`. In a Maven project, it should be in `src/main/resources/META-INF`. The configuration includes the following:

- The persistence unit

- The persistence provider
- The database driver and connection
- Logging

The `persistence.xml` file is packaged as `WEB-INF/classes/META-INF/persistence.xml` in the WAR. This is done automatically in a Maven build at the package phase.

Persistence XML Schema

The beginning of a `persistence.xml` file defines the used schema and namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
```

Defining the Persistence Unit

The root element of the persistence definition is `persistence-unit`. The name of the persistence unit is needed for creating **JPAContainer** instances from a **JPAContainerFactory**, as described in Section 20.4.1, “Creating **JPAContainer** with **JPAContainerFactory**” or when creating a JPA entity manager.

```
<persistence-unit name="addressbook">
```

Persistence provider is the JPA provider implementation used. For example, the JPAContainer AddressBook demo uses the EclipseLink JPA, which is defined as follows:

```
<provider>
  org.eclipse.persistence.jpa.PersistenceProvider
</provider>
```

The persistent classes need to be listed with a `<class>` element. Alternatively, you can allow including unlisted classes for persistence by overriding the `exclude-unlisted-classes` default as follows:

```
<exclude-unlisted-classes>false</exclude-unlisted-classes>
```

JPA provider specific parameters are given under the `properties` element.

```
<properties>
  ...
```

In the following section we give parameters for the EclipseLink JPA and H2 database used in the JPAContainer AddressBook Demo. Please refer to the documentation of the JPA provider you use for a complete reference of parameters.

Database Connection

EclipseLink allows using JDBC for database connection. For example, if we use the the H2 database, we define its driver here as follows:

```
<property name="eclipselink.jdbc.platform"
  value="org.eclipse.persistence.platform.database.H2Platform"/>
<property name="eclipselink.jdbc.driver"
  value="org.h2.Driver" />
```

Database connection is specified with a URL. For example, using an embedded H2 database stored in the home directory it would be as follows:

```
<property name="eclipselink.jdbc.url"
  value="jdbc:h2:~/my-app-h2db"/>
```

A hint: when using an embedded H2 database while developing a Vaadin application in Eclipse, you may want to add `;FILE_LOCK=NO` to the URL to avoid locking issues when redeploying.

We can just use the default user name and password for the H2 database:

```
<property name="eclipselink.jdbc.user" value="sa"/>
<property name="eclipselink.jdbc.password" value="sa"/>
```

Logging Configuration

JPA implementations as well as database engines like to produce logs and they should be configured in the persistence configuration. For example, if using EclipseLink JPA, you can get log that includes all SQL statements with the `FINE` logging level:

```
<property name="eclipselink.logging.level"
  value="FINE" />
```

Other Settings

The rest is some Data Definition Language settings for EclipseLink. During development, when we use generated example data, we want EclipseLink to drop tables before trying to create them. In production environments, you should use `create-tables`.

```
<property name="eclipselink.ddl-generation"
  value="drop-and-create-tables" />
```

And there is no need to generate SQL files, just execute them directly to the database.

```
<property name="eclipselink.ddl-generation.output-mode"
  value="database"/>
</properties>
</persistence-unit>
</persistence>
```

20.2.6. Troubleshooting

Below are some typical errors that you might get when using JPA. These are not specific to JPAContainer.

javax.persistence.PersistenceException: No Persistence provider for EntityManager
The most typical cases for this error are that the persistence unit name is wrong in the source code or in the `persistence.xml` file, or that the `persistence.xml` is at a wrong place or has some other problem. Make sure that the persistence unit name matches and the `persistence.xml` is in `WEB-INF/classes/META-INF` folder in the deployment.

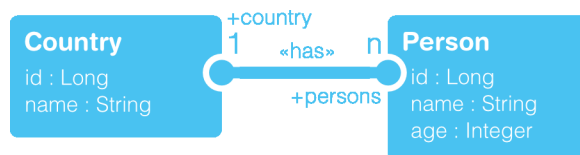
java.lang.IllegalArgumentException: The class is not an entity
 The class is missing from the set of persistent entities. If the `persistence.xml` does not have `exclude-unlisted-classes` defined as `false`, the persistent entity classes should be listed with `<class>` elements.

20.3. Defining a Domain Model

Developing a persistent application begins with defining a domain model. A domain model consists of a number of entities (classes) and relationships between them.

Figure 20.4, “A Domain Model” illustrates a simple domain model as a UML class diagram. It has two entities: **Country** and **Person**. They have a “country has persons” relationship. This is a *one-to-many relationship* with one country having many persons, each of which belongs to just one country.

Figure 20.4. A Domain Model



Realized in Java, the classes are as follows:

```

public class Country {
    private Long id;
    private String name;
    private Set<Person> persons;

    ... setters and getters ...
}

public class Person {
    private Long id;
    private String name;
    private Integer age;
    private Country country;

    ... setters and getters ...
}
  
```

You should make the classes proper beans by defining a default constructor and implementing the `Serializable` interface. A default constructor is required by the JPA entity manager for instantiating entities. Having the classes serializable is not required but often useful for other reasons.

After you have a basic domain model, you need to define the entity relationship metadata by annotating the classes.

20.3.1. Persistence Metadata

The entity relationships are defined with metadata. The metadata can be defined in an XML metadata file or with Java annotations defined in the `javax.persistence` package. With Vaadin JPAContainer, you need to provide the metadata as annotations.

For example, if we look at the Person class in the JPAContainer AddressBook Demo, we define various database-related metadata for the member variables of a class:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;
    private Integer age;

    @ManyToOne
    private Country country;
```

The JPA implementation uses reflection to read the annotations and defines a database model automatically from the class definitions.

Let us look at some of the basic JPA metadata annotations. The annotations are defined in the `javax.persistence` package. Please refer to JPA reference documentation for the complete list of possible annotations.

Annotation: `@Entity`

Each class that is enabled as a persistent entity must have the `@Entity` annotation.

```
@Entity
public class Country {
```

Annotation: `@Id`

Entities must have an identifier that is used as the primary key for the table. It is used for various purposes in database queries, most commonly for joining tables.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

The identifier is generated automatically in the database. The strategy for generating the identifier is defined with the `@GeneratedValue` annotation. Any generation type should work.

Annotation: `@OneToOne`

The `@OneToOne` annotation describes a one-to-one relationship where each entity of one type is associated with exactly one entity of another type. For example, the postal address of a person could be given as such.

```
@OneToOne
private Address address;
```

When using the JPAContainer **FieldFactory** to automatically create fields for a form, the `@OneToOne` relationship generates a nested **Form** to edit the data. See Section 20.8, “Automatic Form Generation” for more details.

Annotation: @Embedded

Just as with the @OneToOne annotation, @Embedded describes a one-to-one relationship, but says that the referenced entity should be stored as columns in the same table as the referencing entity.

```
@Embedded
private Address address;
```

The referenced entity class must have @Embeddable annotation.

The JPAContainer **FieldFactory** generates a nested **Form** for @Embedded, just as with @OneToOne.

Annotation: @OneToMany

The **Country** entity in the domain model has a *one-to-many* relationship with the **Person** entity ("country has persons"). This relationship is represented with the @OneToMany annotation. The *mappedBy* parameter names the corresponding back-reference in the **Person** entity.

```
@OneToMany(mappedBy = "country")
private Set<Person> persons;
```

When using the JPAContainer **FieldFactory** to automatically create fields for a form, the @OneToMany relationship generates a **MasterDetailEditor** for editing the items. See Section 20.8, "Automatic Form Generation" for more details.

Annotation: @ElementCollection

The @ElementCollection annotation can be used for one-to-many relationships to a collection of basic values such as **String** or **Integer**, or to entities annotated as @Embeddable. The referenced entities are stored in a separate table defined with a @CollectionTable annotation.

```
@ElementCollection
@CollectionTable(
    name="OLDPEOPLE",
    joinColumns=@JoinColumn(name="COUNTRY_ID"))
private Set<Person> persons;
```

JPAContainer **FieldFactory** generates a **MasterDetailEditor** for the @ElementCollection relationship, just as with @OneToMany.

Annotation: @ManyToOne

Many people can live in the same country. This would be represented with the @ManyToOne annotation in the **Person** class.

```
@ManyToOne
private Country country;
```

JPAContainer **FieldFactory** generates a **NativeSelect** for selecting an item from the collection. You can do so yourself as well in a custom field factory. Doing so you need to pay notice not to confuse the container between the referenced entity and its ID, which could even result in insertion of false entities in the database in some cases. You can handle conversion between an entity and the entity ID using the **SingleSelectConverter** as follows:


```
@Override
public <T extends Field> T createField(Class<?> dataType,
                                      Class<T> fieldType) {
    if (dataType == Country.class) {
        JPAContainer<Country> countries =
            JPAContainerFactory.make(Country.class, "mypunit");
        ComboBox cb = new ComboBox(null, countries);
        cb.setConverter(new SingleSelectConverter<Country>(cb));
        return (T) cb;
    }
    return super.createField(dataType, fieldType);
}
```

The JPAContainer **FieldFactory** uses the translator internally, so using it also avoids the problem.

Annotation: @Transient

JPA assumes that all entity properties are persisted. Properties that should not be persisted should be marked as transient with the `@Transient` annotation.

```
@Transient
private Boolean superDepartment;
...
@Transient
public String getHierarchicalName() {
    ...
}
```

20.4. Basic Use of JPAContainer

Vaadin JPAContainer offers a highly flexible API that makes things easy in simple cases while allowing extensive flexibility in demanding cases. To begin with, it is a **Container**, as described in Section 10.5, “Collecting Items in Containers”.

In this section, we look how to create and use **JPAContainer** instances. We assume that you have defined a domain model with JPA annotations, as described in the previous section.

20.4.1. Creating JPAContainer with JPAContainerFactory

The **JPAContainerFactory** is the easy way to create `[classname]#JPAContainer#s`. It provides a set of *make...()* factory methods for most cases that you will likely meet. Each factory method uses a different type of entity provider, which are described in Section 20.5, “Entity Providers”.

The factory methods take the class type of the entity class as the first parameter. The second parameter is either a persistence unit name (persistence context) or an **EntityManager** instance.

```
// Create a persistent person container
JPAContainer<Person> persons =
    JPAContainerFactory.make(Person.class, "book-examples");

// You can add entities to the container as well
persons.addEntity(new Person("Marie-Louise Meilleur", 117));

// Set up sorting if the natural order is not appropriate
persons.sort(new String[]{"age", "name"},
            new boolean[]{false, false});

// Bind it to a component
```

```
Table personTable = new Table("The Persistent People", persons);
personTable.setVisibleColumns("id", "name", "age");
layout.addComponent(personTable);
```

It's that easy. In fact, if you run the above code multiple times, you'll be annoyed by getting a new set of persons for each run - that's how persistent the container is. The basic `make()` uses a **CachedMutableLocalEntityProvider**, which allows modifying the container and its entities, as we do above by adding new entities.

When using just the persistence unit name, the factory creates an instance of **EntityManagerFactory** for the persistence unit and uses it to build entity managers. You can also create the entity managers yourself, as described later.

The entity providers associated with the different factory methods are as follows:

Table 20.1. JPAContainerFactory Methods

<code>make()</code>	CachingMutableLocalEntityProvider
<code>makeReadOnly()</code>	CachingLocalEntityProvider
<code>makeBatchable()</code>	BatchableLocalEntityProvider
<code>makeNonCached()</code>	MutableLocalEntityProvider
<code>makeNonCachedReadOnly()</code>	LocalEntityProvider

JPAContainerFactory holds a cache of entity manager factories for the different persistence units, making sure that any entity manager factory is created only once, as it is a heavy operation. You can access the cache to get a new entity manager with the `createEntityManagerForPersistenceUnit()` method.

```
// Get an entity manager
EntityManager em = JPAContainerFactory.
    createEntityManagerForPersistenceUnit("book-examples");

// Do a query
em.getTransaction().begin();
em.createQuery("DELETE FROM Person p").executeUpdate();
em.persist(new Person("Jeanne Calment", 122));
em.persist(new Person("Sarah Knauss", 119));
em.persist(new Person("Lucy Hannah", 117));
em.getTransaction().commit();

...
```

Notice that if you use update the persistent data with an entity manager outside a **JPAContainer** bound to the data, you need to refresh the container as described in Section 20.4.2, "Creating and Accessing Entities".

Creating JPAContainer Manually

While it is normally easiest to use a **JPAContainerFactory** to create **JPAContainer** instances, you may need to create them manually. It is necessary, for example, when you need to use a custom entity provider or extend **JPAContainer**.

First, we need to create an entity manager and then the entity provider, which we bind to a **JPAContainer**.

```
// We need a factory to create entity manager
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("book-examples");

// We need an entity manager to create entity provider
EntityManager em = emf.createEntityManager();

// We need an entity provider to create a container
CachingMutableLocalEntityProvider<Person> entityProvider =
    new CachingMutableLocalEntityProvider<Person>(Person.class,
                                                em);

// And there we have it
JPAContainer<Person> persons =
    new JPAContainer<Person>(Person.class);
persons.setEntityProvider(entityProvider);
```

You could save the first step by asking the entity manager from the **JPAContainerFactory**.

20.4.2. Creating and Accessing Entities

JPAContainer integrates with the JPA entity manager, which you would normally use to create and access entities with JPA. You can use the entity manager for any purposes you may have, and then **JPAContainer** to bind entities to user interface components such as **Table**, **Tree**, any selection components, or a **Form**.

You can add new entities to a **JPAContainer** with the `addEntity()` method. It returns the item ID of the new entity.

```
Country france = new Country("France");
Object itemId = countries.addEntity(france);
```

The item ID used by **JPAContainer** is the value of the ID property (column) defined with the `@Id` annotation. In our **Country** entity, it would have **Long** type. It is generated by the entity manager when the entity is persisted and set with the setter for the ID property.

Notice that the `addEntity()` method does *not* attach the entity instance given as the parameter. Instead, it creates a new instance. If you need to use the entity for some purpose, you need to get the actual managed entity from the container. You can get it with the item ID returned by `addEntity()`.

```
// Create a new entity and add it to a container
Country france = new Country("France");
Object itemId = countries.addEntity(france);

// Get the managed entity
france = countries.getItem(itemId).getEntity();

// Use the managed entity in entity references
persons.addEntity(new Person("Jeanne Calment", 122, france));
```

Entity Items

The `getItem()` method is defined in the normal `Vaadin Container` interface. It returns an **EntityItem**, which is a wrapper over the actual entity object. You can get the entity object with `getEntity()`.

An **EntityItem** can have a number of states: persistent, modified, dirty, and deleted. The dirty and deleted states are meaningful when using *container buffering*, while the modified state is meaningful when using *item buffering*. Both levels of buffering can be used together - user input is first written to the item buffer, then to the entity instance, and finally to the database.

The `isPersistent()` method tells if the item is actually persistent, that is, fetched from a persistent storage, or if it is just a transient entity created and buffered by the container.

The `isModified()` method checks whether the **EntityItem** has changes that are not yet committed to the entity instance. It is only relevant if the item buffering is enabled with `setBuffered(true)` for the item.

The `isDirty()` method checks whether the entity object has been modified after it was fetched from the entity provider. The dirty state is possible only when buffering is enabled for the container.

The `isDeleted()` method checks whether the item has been marked for deletion with `removeItem()` in a buffered container.

Refreshing JPAContainer

In cases where you change **JPAContainer** items outside the container, for example by through an `EntityManager`, or when they change in the database, you need to refresh the container.

The `EntityContainer` interface implemented by **JPAContainer** provides two methods to refresh a container. The `refresh()` discards all container caches and buffers and refreshes all loaded items in the container. All changes made to items provided by the container are discarded. The `refreshItem()` refreshes a single item.

20.4.3. Nested Properties

If you have a one-to-one or many-to-one relationship, you can define the properties of the referenced entity as *nested* in a **JPAContainer**. This way, you can access the properties directly through the container of the first entity type as if they were its properties. The interface is the same as with **BeanContainer** described in Section 10.5.4, “**BeanContainer**”. You just need to add each nested property with `addNestedContainerProperty()` using dot-separated path to the property.

```
// Have a persistent container
JPAContainer<Person> persons =
    JPAContainerFactory.make(Person.class, "book-examples");

// Add a nested property to a many-to-one property
persons.addNestedContainerProperty("country.name");

// Show the persons in a table, except the "country" column,
// which is an object - show the nested property instead
Table personTable = new Table("The Persistent People", persons);
personTable.setVisibleColumns("name", "age", "country.name");

// Have a nicer caption for the country.name column
personTable.setColumnHeader("country.name", "Nationality");
```

The result is shown in Figure 20.5, “Nested Properties”. Notice that the `country` property in the container remains after adding the nested property, so we had to make that column invisible.

Alternatively, we could have redefined the `toString()` method in the country object to show the name instead of an object reference.

Figure 20.5. Nested Properties

The Persistent People

NAME	AGE	NATIONALITY	
Jeanne Calment	122	France	^
Sarah Knauss	119	United States	
Marie-Louise Meilleur	117	Canada	
Lucy Hannah	117	United States	
Tane Ikai	116	Japan	^

You can use the `*` wildcard to add all properties in a nested item, for example, `"country.*"`.

20.4.4. Hierarchical Container

JPAContainer implements the `Container.Hierarchical` interface and can be bound to hierarchical components such as a **Tree** or **TreeTable**. The feature requires that the hierarchy is represented with a *parent* property that refers to the parent item. At database level, this would be a column with IDs.

The representation would be as follows:

```
@Entity
public class CelestialBody implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long    id;

    private String  name;

    @ManyToOne
    private CelestialBody parent;
    ...
} ...

// Create some entities
CelestialBody sun      = new CelestialBody("The Sun", null);
CelestialBody mercury = new CelestialBody("Mercury", sun);
CelestialBody venus    = new CelestialBody("Venus", sun);
CelestialBody earth    = new CelestialBody("Earth", sun);
CelestialBody moon     = new CelestialBody("The Moon", earth);
CelestialBody mars     = new CelestialBody("Mars", sun);
...
```

You set up a **JPAContainer** to have hierarchy by calling `setParentProperty()` with the name of the property that refers to the parent. Coincidentally, it is named `"parent"` in the example:

```
// Create the container
JPAContainer<CelestialBody> bodies =
    JPAContainerFactory.make(CelestialBody.class, "my-unit");

// Set it up for hierarchical representation
```

```
bodies.setParentProperty("parent");

// Bind it to a hierarchical component
Tree tree = new Tree("Celestial Bodies", bodies);
tree.setItemCaptionMode(Tree.ITEM_CAPTION_MODE_PROPERTY);
tree.setItemCaptionPropertyId("name");
```

You can use the `rootItemIds()` to acquire the item IDs of the root elements with no parent.

```
// Expand the tree
for (Object rootId: bodies.rootItemIds())
    tree.expandItemsRecursively(rootId);
```

Unsupported Hierarchical Features

Using `setParent()` in the container to define parenthood is not supported.

Also, the current implementation does not support `setChildrenAllowed()`, which controls whether the user can expand a node by clicking a toggle. The toggle is by default visible for all nodes, even if they have no children. The method is not supported because it would require storing the information outside the entities. You can override `areChildrenAllowed()` to implement the functionality using a custom logic.

```
// Customize JPAContainer to define the logic for
// displaying the node expansion indicator
JPAContainer<CelestialBody> bodies =
    new JPAContainer<CelestialBody>(CelestialBody.class) {
        @Override
        public boolean areChildrenAllowed(Object itemId) {
            // Some simple logic
            return getChildren(itemId).size() > 0;
        }
    };
bodies.setEntityProvider(
    new CachingLocalEntityProvider<CelestialBody>(
        CelestialBody.class, em));
```

20.5. Entity Providers

Entity providers provide access to entities persisted in a data store. They are essentially wrappers over a JPA entity manager, adding optimizations and other features important when binding persistent data to a user interface.

The choice and use of entity providers is largely invisible if you create your **JPAContainer** instances with the **JPAContainerFactory**, which hides such details.

JPAContainer entity providers can be customized, which is necessary for some purposes. Entity providers can be Enterprise JavaBeans (EJBs), which is useful when you use them in a Java EE application server.

20.5.1. Built-In Entity Providers

JPAContainer includes various kinds of built-in entity providers: caching and non-caching, read-write and read-only, and batchable.

Caching is useful for performance, but takes some memory for the cache and makes the provider stateful. *Batching*, that is, running updates in larger batches, can also enhance performance

and be used together with caching. It is stateless, but doing updates is a bit more complex than otherwise.

Using a *read-only* container is preferable if read-write capability is not needed.

All built-in providers are *local* in the sense that they provide access to entities using a local JPA entity manager.

The **CachingMutableLocalEntityProvider** is usually recommended as the first choice for read-write access and **CachingLocalEntityProvider** for read-only access.

LocalEntityProvider

A read-only, lazy loading entity provider that does not perform caching and reads its data directly from an entity manager.

You can create the provider with `makeNonCachedReadOnly()` method in **JPAContainerFactory**.

MutableLocalEntityProvider

Extends **LocalEntityProvider** with write support. All changes are directly sent to the entity manager.

Transactions can be handled either internally by the provider, which is the default, or by the container. In the latter case, you can extend the class and annotate it, for example, as described in Section 20.5.1, “Built-In Entity Providers”.

The provider can notify about updates to entities through the `EntityProviderChangeListener` interface.

BatchableLocalEntityProvider

A simple non-caching implementation of the `BatchableEntityProvider` interface. It extends **MutableLocalEntityProvider** and simply passes itself to the `batchUpdate()` callback method. This will work properly if the entities do not contain any references to other entities that are managed by the same container.

CachingLocalEntityProvider

A read-only, lazy loading entity provider that caches both entities and query results for different filter/sortBy combinations. When the cache gets full, the oldest entries in the cache are removed. The maximum number of entities and entity IDs to cache for each filter/sortBy combination can be configured in the provider. The cache can also be manually flushed. When the cache grows full, the oldest items are removed.

You can create the provider with `makeReadOnly()` method in **JPAContainerFactory**.

CachingMutableLocalEntityProvider

Just like **CachingLocalEntityProvider**, but with read-write access. For read access, caching works just like in the read-only provider. When an entity is added or updated, the cache is flushed in order to make sure the added or updated entity shows up correctly when using filters and/or sorting. When an entity is removed, only the filter/sortBy-caches that actually contain the item are flushed.

This is perhaps the most commonly entity provider that you should consider using for most tasks. You can create it with the `make()` method in **JPAContainerFactory**.

CachingBatchableLocalEntityProvider

This provider supports making updates in *batches*. You need to implement a `BatchUpdateCallback` that does all the updates and execute the batch by calling `batchUpdate()` on the provider.

The provider is an extension of the **CachingMutableLocalEntityProvider** that implements the `BatchableEntityProvider` interface. This will work properly if the entities do not contain any references to other entities that are managed by the same container.

You can create the provider with `makeBatchable()` method in **JPAContainerFactory**.

20.5.2. Using JNDI Entity Providers in JEE6 Environment

JPAContainer 2.0 introduced a new set of entity providers specifically for working in a JEE6 environment. In a JEE environment, you should use an entity manager provided by the application server and, usually, JTA transactions instead of transactions provided by JPA. Entity providers in `com.vaadin.addon.jpaccontainer.provider.jndijta` package work mostly the same way as the normal providers discussed earlier, but use JNDI lookups to get reference to an `EntityManager` and to a JTA transaction.

The JNDI providers work with almost no special configuration at all. The **JPAContainerFactory** has factory methods for creating various JNDI provider types. The only thing that you commonly need to do is to expose the `EntityManager` to a JNDI address. By default, the JNDI providers look for the `EntityManager` from `"java:comp/env/persistence/em"`. This can be done with the following snippet in `web.xml` or with similar configuration with annotations.

```
<persistence-context-ref>
  <persistence-context-ref-name>
    persistence/em
  </persistence-context-ref-name>
  <persistence-unit-name>MYPU</persistence-unit-name>
</persistence-context-ref>
```

The `"MYPU"` is the identifier of your persistence unit defined in your `persistence.xml` file.

If you choose to annotate your servlets (instead of using the `web.xml` file as described above), you can simply add the following annotation to your servlet.

```
@PersistenceContext(name="persistence/em", unitName="MYPU")
```

If you wish to use another address for the persistence context, you can define them with the `setJndiAddresses()` method. You can also define the location for the JTA **UserTransaction**, but that should be always accessible from `"java:comp/UserTransaction"` by the JEE6 specification.

20.5.3. Entity Providers as Enterprise Beans

Entity providers can be Enterprise JavaBeans (EJB). This may be useful if you use JPAContainer in a Java EE application server. In such case, you need to implement a custom entity provider that allows the server to inject the entity manager.

For example, if you need to use Java Transaction API (JTA) for JPA transactions, you can implement such entity provider as follows. Just extend a built-in entity provider of your choice and

annotate the entity manager member as `@PersistenceContext`. Entity providers can be either stateless or stateful session beans. If you extend a caching entity provider, it has to be stateful.

```
@Stateless
@TransactionManagement
public class MyEntityProviderBean extends
    MutableLocalEntityProvider<MyEntity> {

    @PersistenceContext
    private EntityManager em;

    protected LocalEntityProviderBean() {
        super(MyEntity.class);
        setTransactionsHandledByProvider(false);
    }

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    protected void runInTransaction(Runnable operation) {
        super.runInTransaction(operation);
    }

    @PostConstruct
    public void init() {
        setEntityManager(em);
        /*
         * The entity manager is transaction-scoped, which means
         * that the entities will be automatically detached when
         * the transaction is closed. Therefore, we do not need
         * to explicitly detach them.
         */
        setEntitiesDetached(false);
    }
}
```

If you have more than one EJB provider, you might want to create an abstract super class of the above and only define the entity type in implementations. You can implement an entity provider as a managed bean in Spring Framework the same way.

20.6. Filtering JPAContainer

Normally, a **JPAContainer** contains all instances of a particular entity type in the persistence context. Hence, it is equivalent to a database table or query. Just like with database queries, you often want to narrow the results down. **JPAContainer** implements the `Filterable` interface in Vaadin containers, described in Section 10.5.7, “**Filterable** Containers”. All filtering is done at the database level with queries, not in the container.

For example, let us filter all the people older than 117:

```
Filter filter = new Compare.Greater("age", 117);
persons.addContainerFilter(filter);
```

This would create a JPQL query somewhat as follows:

```
SELECT id FROM Person WHERE (AGE > 117)
```

The filtering implementation uses the JPA 2.0 Criteria API transparently. As the filtering is done at the database-level, custom filters that use the `Filterable` API do not work.

When using Hibernate, note that it does not support implicit joins. See Section 20.9.2, “Joins in Hibernate vs EclipseLink” for more details.

20.7. Querying with the Criteria API

When the `Filterable` API is not enough and you need to have more control, you can make queries directly with the JPA Criteria API. You may also need to customize sorting or joins, or otherwise modify the query in some way. To do so, you need to implement a `QueryModifierDelegate` that the JPAContainer entity provider calls when making a query. The easiest way to do this is to extend **`DefaultQueryModifierDelegate`**, which has empty implementations of all the methods so that you can only override the ones you need.

The entity provider calls specific `QueryModifierDelegate` methods at different stages while making a query. The stages are:

1. Start building a query
2. Add " ORDER BY" expression
3. Add " WHERE" expression (filter)
4. Finish building a query

Methods where you can modify the query are called before and after each stage as listed in the following table:

Table 20.2. QueryModifierDelegate Methods

<code>queryWillBeBuilt()</code>
<code>orderByWillBeAdded()</code>
<code>orderByWasAdded()</code>
<code>filtersWillBeAdded()</code>
<code>filtersWereAdded()</code>
<code>queryHasBeenBuilt()</code>

All the methods get two parameters. The `CriteriaBuilder` is a builder that you can use to build queries. The `CriteriaQuery` is the query being built.

You can use the `getRoots().iterator().next()` in `CriteriaQuery` to get the "root" that is queried, for example, the `PERSON` table, etc.

20.7.1. Filtering the Query

Let us consider a case where we modify the query for a **Person** container so that it includes only people over 116. This trivial example is identical to the one given earlier using the **`Filterable`** interface.

```
persons.getEntityProvider().setQueryModifierDelegate(
    new DefaultQueryModifierDelegate() {
        @Override
        public void filtersWillBeAdded(
            CriteriaBuilder criteriaBuilder,
            CriteriaQuery<?> query,
```

```

        List<Predicate> predicates) {
    Root<?> fromPerson = query.getRoots().iterator().next();

    // Add a "WHERE age > 116" expression
    Path<Integer> age = fromPerson.<Integer>get("age");
    predicates.add(criteriaBuilder.gt(age, 116));
    }
});

```

20.7.2. Compatibility

When building queries, you should consider the capabilities of the different JPA implementations. Regarding Hibernate, see Section 20.9.2, “Joins in Hibernate vs EclipseLink”.

20.8. Automatic Form Generation

The JPAContainer **FieldFactory** is an implementation of the `FormFieldFactory` and `TableFieldFactory` interfaces that can generate fields based on JPA annotations in a POJO. It goes further than the **DefaultFieldFactory**, which only creates simple fields for the basic data types. This way, you can easily create forms to input entities or enable editing in tables.

The generated defaults are as follows:

Annotation	Class Mapping
@ManyToOne	NativeSelect
@OneToOne, @Embedded	Nested Form
@OneToMany, @ElementCollection	MasterDetailEditor (see below)
@ManyToMany	Selectable Table

The field factory is recursive, so that you can edit a complex object tree with one form.

20.8.1. Configuring the Field Factory

The **FieldFactory** is highly configurable with various configuration settings and by extending.

The `setMultiSelectType()` and `setSingleSelectType()` allow you to specify a selection component that is used instead of the default for a field with `@ManyToMany` and `@ManyToOne` annotation, respectively. The first parameter is the class type of the field, and the second parameter is the class type of a selection component. It must be a sub-class of **AbstractSelect**.

The `setVisibleProperties()` controls which properties (fields) are visible in generated forms, subforms, and tables. The first parameter is the class type for which the setting should be made, followed by the IDs of the visible properties.

The configuration should be done before binding the form to a data source as that is when the field generation is done.

Further configuration must be done by extending the many protected methods. Please see the API documentation for the complete list.

20.8.2. Using the Field Factory

The most basic use case for the JPAContainer **FieldFactory** is with a **Form** bound to a container item:

```
// Have a persistent container
final JPAContainer<Country> countries =
    JPAContainerFactory.make(Country.class, "book-examples");

// For selecting an item to edit
final ComboBox countrySelect =
    new ComboBox("Select a Country", countries);
countrySelect.setItemCaptionMode(Select.ITEM_CAPTION_MODE_PROPERTY);
countrySelect.setItemCaptionPropertyId("name");

// Country Editor
final Form countryForm = new Form();
countryForm.setCaption("Country Editor");
countryForm.addStyleName("bordered"); // Custom style
countryForm.setWidth("420px");
countryForm.setBuffered(true);
countryForm.setEnabled(false);

// When an item is selected from the list...
countrySelect.addValueChangeListener(new ValueChangeListener() {
    @Override
    public void valueChange(ValueChangeEvent event) {
        // Get the item to edit in the form
        Item countryItem =
            countries.getItem(event.getProperty().getValue());

        // Use a JPAContainer field factory
        // - no configuration is needed here
        final FieldFactory fieldFactory = new FieldFactory();
        countryForm.setFormFieldFactory(fieldFactory);

        // Edit the item in the form
        countryForm.setItemDataSource(countryItem);
        countryForm.setEnabled(true);

        // Handle saves on the form
        final Button save = new Button("Save");
        countryForm.getFooter().removeAllComponents();
        countryForm.getFooter().addComponent(save);
        save.addClickListener(new ClickListener() {
            @Override
            public void buttonClick(ClickEvent event) {
                try {
                    countryForm.commit();
                    countryForm.setEnabled(false);
                } catch (InvalidValueException e) {
                }
            }
        });
    }
});
countrySelect.setImmediate(true);
countrySelect.setNullSelectionAllowed(false);
```

This would create a form shown in Figure 20.6, “Using FieldFactory with One-to-Many Relationship”.

Figure 20.6. Using FieldFactory with One-to-Many Relationship

If you use Hibernate, you also need to pass an **EntityManagerPerRequestHelper**, either for the constructor or with `setEntityManagerPerRequestHelper()`.

20.8.3. Master-Detail Editor

The **MasterDetailEditor** is a field component that allows editing an item property that has one-to-many relationship. The item can be a row in a table or bound to a form. It displays the referenced collection as an editable **Table** and allows adding and removing items in it.

You can use the **MasterDetailEditor** manually, or perhaps more commonly use a **JPAContainer FieldFactory** to create it automatically. As shown in the example in Figure 20.6, “Using FieldFactory with One-to-Many Relationship”, the factory creates a **MasterDetailEditor** for all properties with a `@OneToMany` or an `@ElementCollection` annotation.

20.9. Using JPAContainer with Hibernate

Hibernate needs special handling in some cases.

20.9.1. Lazy loading

In order for lazy loading to work automatically, an entity must be attached to an entity manager. Unfortunately, Hibernate can not keep entity managers for long without problems. To work around the problem, you need to use a special lazy loading delegate for Hibernate.

JPAContainer entity providers handle lazy loading in delegates defined by the `LazyLoadingDelegate` interface. The default implementation for Hibernate is defined in **HibernateLazyLoadingDelegate**. You can instantiate one and use it in an entity provider with `setLazyLoadingDelegate()`.

The default implementation works so that whenever a lazy property is accessed through the Vaadin Property interface, the value is retrieved with a separate (JPA Criteria API) query using the currently active entity manager. The value is then manually attached to the entity instance, which is detached from the entity manager. If this default implementation is not good enough, you may need to make your own implementation.

20.9.2. Joins in Hibernate vs EclipseLink

EclipseLink supports implicit joins, while Hibernate requires explicit joins. In SQL terms, an explicit join is a "FROM a INNER JOIN b ON a.bid = b.id" expression, while an implicit join is done in a WHERE clause, such as: "FROM a,b WHERE a.bid = b.id".

In a JPAContainer filter with EclipseLink, an implicit join would have form:

```
new Equal("skills.skill", s)
```

In Hibernate you would need to use **JoinFilter** for the explicit join:

```
new JoinFilter("skills", new Equal("skill", s))
```

Chapter 21

Mobile Applications with TouchKit

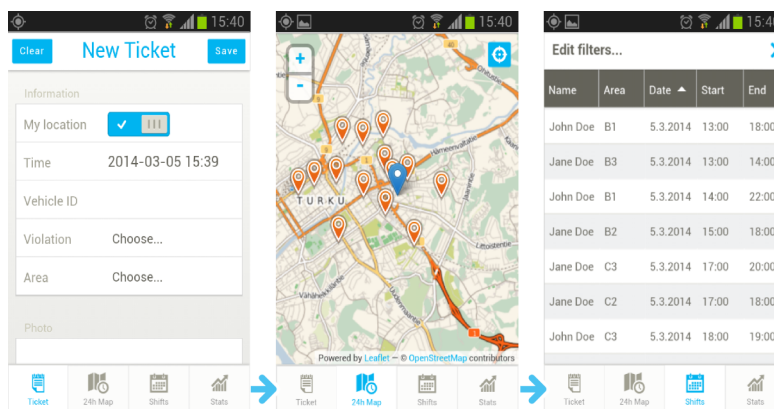
21.1. Overview	578
21.2. Considerations Regarding Mobile Browsing	580
21.3. Installing Vaadin TouchKit	582
21.4. Importing the Parking Demo	582
21.5. Creating a New TouchKit Project	583
21.6. Elements of a TouchKit Application	585
21.7. Mobile User Interface Components	591
21.8. Advanced Mobile Features	608
21.9. Offline Mode	610
21.10. Building an Optimized Widget Set	613
21.11. Testing and Debugging on Mobile Devices	613

This chapter describes how to create mobile applications using the Vaadin TouchKit.

21.1. Overview

Web browsing is becoming ever increasingly mobile and web applications need to satisfy users with both desktop computers and mobile devices, such as phones and tablets. While the mobile browsers can show the pages just like in regular browsers, the screen size, finger accuracy, and mobile browser features need to be considered to make the experience more pleasant. Vaadin TouchKit gives the power of Vaadin for creating mobile user interfaces that complement the regular web user interfaces of your applications. Just like the purpose of the Vaadin Framework is to make desktop-like web applications, the purpose of TouchKit is to allow creation of web applications that give the look and feel of native mobile applications.

Figure 21.1. The Parking Demo for Vaadin TouchKit



Creating a mobile UI is much like creating a regular Vaadin UI. You can use all the regular Vaadin components and add-ons available from Vaadin Directory, but most importantly, you can use the special TouchKit components that are optimized for mobile devices.

```
@Theme("mobiletheme")
@Widgetset("com.example.myapp.MyAppWidgetSet")
@Title("My Mobile App")
public class SimplePhoneUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Define a view
        class MyView extends NavigationView {
            public MyView() {
                super("Planet Details");

                CssLayout content = new CssLayout();
                setContent(content);

                VerticalComponentGroup group =
                    new VerticalComponentGroup();
                content.addComponent(group);

                group.addComponent(new TextField("Planet"));
                group.addComponent(new NumberField("Found"));
                group.addComponent(new Switch("Probed"));

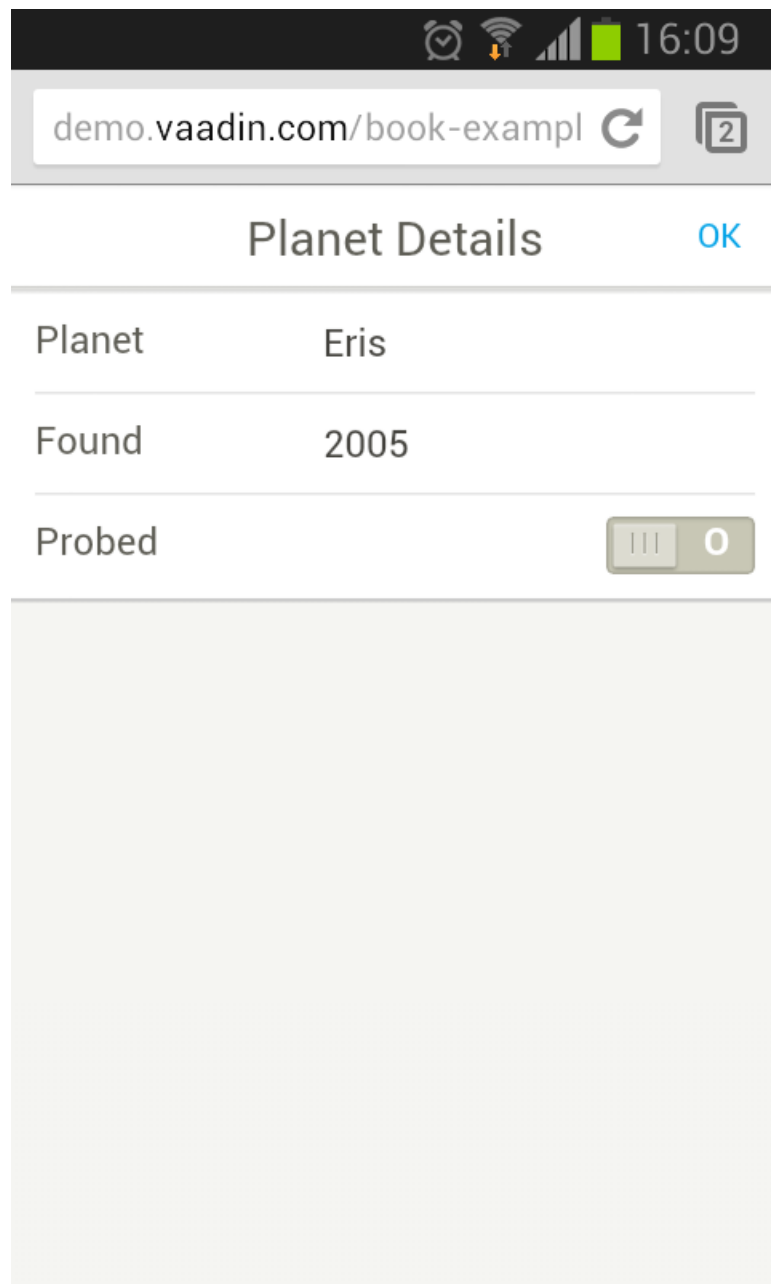
                setRightComponent(new Button("OK"));
            }
        }
    }
}
```



```
        // Use it as the content root
        setContent(new MyView());
    }
    ...
}
```

The above example omits the definition of the servlet class, does not have any UI logic yet, and you would normally implement some views, etc. The resulting UI is shown in Figure 21.2, “Simple TouchKit UI”.

Figure 21.2. Simple TouchKit UI



TouchKit supports many special mobile browser features, such as geolocation, context-specific input fields, and home screen launching. On iOS, special features such as splash screen and web app mode are supported.

In addition to developing regular server-side UIs, TouchKit allows a special *offline mode*, which is a client-side Vaadin UI that is stored in the browser cache and switched to automatically when the network connection is not available, either when starting the application or while using it. For more information, see Section 21.9, “Offline Mode”.

In this chapter, we first consider some special aspects of mobile browsing. Then, we look how to create a project that uses TouchKit. TouchKit offers a number of specialized mobile components, which are described in a dedicated section. We treat phone and tablet applications separately, and discuss testing briefly.

21.1.1. TouchKit Demos

The Parking Demo showcases the most important TouchKit features for a mobile location-based business application. The app itself is for helping parking enforcement officers write parking tickets on the streets. It uses geolocation, image acquisition from the camera of the mobile device, map navigation, data visualization with Vaadin Charts, and dynamic UIs with responsive layouts. You can try it out at <http://demo.vaadin.com/parking>. See Section 21.4, “Importing the Parking Demo” for instructions for importing the project in Eclipse.

Mobile Mail is another demo application, which shows how to implement browsing of deep category trees and make forms. You can try it out at <http://demo.vaadin.com/mobilemail>.

Some of the examples given in this chapter can be seen in action at demo.vaadin.com/touchkit-sampler.

21.1.2. Licensing

Vaadin TouchKit is a commercial product licensed under a dual-licensing scheme. The AGPL license allows open-source development, while the CVAL license needs to be purchased for closed-source use, including web deployments and internal use. Commercial licenses can be purchased from Vaadin Directory, where you can also find the license details and download Vaadin TouchKit.

21.2. Considerations Regarding Mobile Browsing

When developing web applications that support mobile browsing, you need to consider various issues that are different from non-mobile use. TouchKit is designed to help with these issues.

21.2.1. Mobile Human Interface

Mobile devices use very different human interfaces than regular computers. For example, the screen can be rotated easily to switch between portrait and landscape views. This does not just change the dimensions of the display, but also affects how to arrange components for the best user experience. In addition to TouchKit, responsive layouts help in allowing flexible layouts, as described in Section 9.10, “Responsive Themes”.

The user interface is used with a finger instead of a mouse, so there are no features such as “right-finger-button”. When using a mouse you can click double-click or right-click, but on a touch device, you are using interactions such as tap and “long tap”. Finger gestures also play

a large role, such as using a vertical swipe gesture for scrolling instead of a scroll bar. Some browsers also allow using two- or multiple-finger gestures.

There is normally no physical keyboard, but an on-screen keyboard, which can change depending on the context. You also need to ensure that it does not hide the input field to which the user is trying to enter data when it pops up. This should be handled by the browser, but is among the issues that requires special testing.

21.2.2. Bandwidth and Performance

Mobile Internet connections are often significantly slower than with fixed lines. With a low-end mobile connection, such as 384 kbps, just loading the Vaadin client-side engine can take several seconds. This can be helped by compiling a widget set that includes only the widgets for the used components, as described in Section 21.10, “Building an Optimized Widget Set”, by compiling the theme into the widget set, and so forth.

Even with mobile broadband, the latency can be significant factor, especially with highly interactive rich applications. The latency is usually almost unnoticeable in fixed lines, typically less than 100 ms, while mobile Edge connections typically have latency around 500 ms, and sometimes much higher during hiccups. You may need to limit the use of the immediate mode, text change events, and polling. The latency compensation in some components, such as **NavigationManager**, allows view change animations to occur while the server request to display the result is being made.

Further, the choice of components affects performance. TouchKit components are designed to be light-weight. Of the other Vaadin components, some are more light-weight than others. Especially, most other layout components have a more deeper DOM structure and are slower to render than the light-weight **CssLayout**. TouchKit also includes special styling for **CssLayout**.

21.2.3. Mobile Features

Phones and tablets have many integrated features that are often available in the browser interface as well. Location-awareness is one of the most recent features. And of course, you can also make phone calls.

21.2.4. Compatibility

The mobile browsing field is currently evolving at a quick pace and the special conventions introduced by leading manufacturers may, in the next few years, stabilize as new web standards. The browser support in TouchKit originally concentrated on WebKit, which appears to be emerging as the leading mobile browser core. In addition to Apple's products, also the default browser in Android uses WebKit as the layout engine. Yet they have differences, as the Android's JavaScript engine, which is highly relevant for Vaadin, is the Google Chrome's V8 engine. As of TouchKit 4, Internet Explorer on Windows Phone is also supported.

For the list of devices supported by the latest TouchKit version, see the TouchKit product page at the Vaadin site.

Vaadin TouchKit aims to follow the quickly evolving APIs of these major platforms, with the assumption that other browsers will follow their lead in standardization. Other platforms will be supported if they rise in popularity.

Back Button

Some mobile devices, especially Android and Windows Phone devices, have a dedicated back button, while iOS devices in particular do not. TouchKit does not provide any particular support for the button, but as it is a regular browser back button, you can handle it with URI fragments, as described in Section 12.11, “Managing URI Fragments”. For iOS, the browser back button is hidden if the user adds the application to the home screen, in which case you need to implement application-specific logic for the back-navigation.

21.3. Installing Vaadin TouchKit

You can download and install TouchKit from Vaadin Directory at vaadin.com/addon/vaadin-touchkit as an installation package, or get it with Maven or Ivy. If your project is not compatible with the AGPL license, you can purchase CVAL licenses from Vaadin Directory or subscribe to the Pro Tools package at vaadin.com/pro.

Add-on installation is described in detail in Chapter 18, *Using Vaadin Add-ons*. The add-on includes a widget set, so you need to compile the widget set for your project.

21.4. Importing the Parking Demo

The Parking Demo, illustrated in Figure 21.1, “The Parking Demo for Vaadin TouchKit” in the overview, showcases most of the functionality in Vaadin TouchKit. You can try out the demo online with a TouchKit-compatible browser at demo.vaadin.com/parking.

You can browse the sources on-line or, more conveniently, import the project in Eclipse (or other IDE). As the project is Maven-based, Eclipse users need to install the m2e plugin to be able to import Maven projects, as well as EGit to be able to import Git repositories. Once they are installed, you should be able to import Parking Demo as follows.

1. Select **File** → **Import**
2. Select **Maven** → **Check out Maven Project from SCM**, and click **Next**.
3. You may need to install the EGit SCM connector if you have not done so previously. If Git is not available in the SCM list, click **m2e marketplace**, select the EGit connector, and click **Finish**. You need to restart Eclipse and redo the earlier steps above.

Instead of using m2e EGit connector, you can also check out the project with another Git tool and then import it in Eclipse as a Maven project.

4. In **SCM URL**, select **git** and enter the repository URL <https://github.com/vaadin/parking-demo>.
5. Click **Finish**.
6. Compile the widget set either by clicking **Compile Widgetset** in the Eclipse toolbar or by running the `vaadin:compile` goal with Maven.
7. Deploy the application to a server. See Section 3.4.5, “Setting Up and Starting the Web Server” for instructions for deploying in Eclipse.
8. Open the URL <http://localhost:8080/parking> with a mobile device or a WebKit-compatible browser, such as Safari or Chrome, to run the Parking Demo.

21.5. Creating a New TouchKit Project

The easiest ways to create a new TouchKit application project are to either use the Maven archetype or create the project as a regular Vaadin project with the Vaadin Plugin for Eclipse and then modify it for TouchKit.

21.5.1. Using the Maven Archetype

You can create a new TouchKit application project using the Maven *vaadin-archetype-touchkit* archetype. Creating Vaadin projects with Maven is described in more detail in Section 3.5, “Creating a Project with Maven”.

For example, to create a project from the command-line, you could do:

```
$ mvn archetype:generate \
  -DarchetypeGroupId=com.vaadin \
  -DarchetypeArtifactId=vaadin-archetype-touchkit \
  -DarchetypeVersion=4.0.0 \
  -DgroupId=example.com -DartifactId=myproject \
  -Dversion=0.1.0 \
  -DapplicationName=My -Dpackaging=war
```

The *ApplicationName* parameter for the archetype is used as a prefix for the various stub class names. For example, the above "My" name results in classes such as **MyTouchKitUI**.

The generated project has the following source files:

MyTouchKitUI.java

The mobile UI for the TouchKit application. See Section 21.6.4, “The UI” for the basics of a TouchKit UI. The example UI uses **TabBarView** as the content. The first tab features a **MenuView** (see below), a navigation view stub defined in the project.

MyFallbackUI.java

A fallback UI for browsers unsupported by TouchKit, such as regular desktop browsers. See Section 21.8.1, “Providing a Fallback UI” for more information about fallback UIs.

MyServlet.java

The servlet class for the UI, defined using the `@WebServlet` annotation in Servlet API 3.0. The generated servlet customizes TouchKit to define the **MyUIProvider**, which sets the fallback UI. See Section 21.6.1, “The Servlet Class” for more details about defining a custom servlet to customize TouchKit.

MyUIProvider.java

Creates either the **MyTouchKitUI** for supported mobile browsers or **MyFallbackUI** for unsupported browsers. See Section 21.8.1, “Providing a Fallback UI” for more information about fallback UIs.

MenuView.java

Presents a stub for a menu view. The menu is made of **NavigationButtons** laid out in a **VerticalComponentGroup**. Clicking a button navigates to another view; in the stub to a **FormView** (see below).

FormView.java

Presents a stub for a data input form.

`gwt/AppWidgetSet.gwt.xml`

Widget set descriptor for the project. When compiled, it is automatically updated to include possible other add-on widget sets in the project.

`gwt/client/MyOfflineDataService.java`

A data service stub for storing data in the offline mode. See Section 21.9, “Offline Mode”.

`gwt/client/MyPersistToServerRpc.java`

Client-to-Server RPC stub to persist offline data to the server-side.

If you import the project to Eclipse or other IDE, you at least need to compile the widget set to be able to deploy the project. You can do that with Maven integration in the IDE, or from command-line with:

```
$ mvn vaadin:compile
```

See Section 3.5, “Creating a Project with Maven”. At least in Eclipse, you should now be able to import and deploy the project to a development server. You can also compile the project and launch it in a Jetty web server (port 8080) from command-line as follows:

```
$ mvn package
```

```
$ mvn jetty:run
```

Note that a project generated by the archetype defines the servlet with the `@WebServlet` annotation defined in Servlet API 3.0. The application server must support Servlet 3.0. For example, if you use Tomcat, you need at least Tomcat 7.

21.5.2. Starting from a New Eclipse Project

You can create a new TouchKit project from a regular Vaadin project created with the Vaadin Plugin for Eclipse (see Section 3.4, “Creating and Running a Project in Eclipse”).

After creating the project, you need to do the following tasks:

1. Install the TouchKit library in the project by including it in the `ivy.xml` and compile the widget set.
2. Extend **TouchkitServlet** instead of **VaadinServlet** in the servlet class, as described in Section 21.6.1, “The Servlet Class”. It is recommended that you extract the static inner class created by the wizard to a regular class, as you most probably need to do additional configuration in it.

```
@WebServlet(value = "/*",
            asyncSupported = true)
@VaadinServletConfiguration(
    productionMode = false,
    ui = MyMobileUI.class)
public class MyProjectServlet extends TouchKitServlet {
}
```

3. If you intend to define a fallback UI later, as described in Section 21.8.1, “Providing a Fallback UI”, you may want to copy the original UI class stub to use it as a fallback UI class.

4. To get started quickly, disable the use of custom theme by using `@Theme("touch-kit")` in the UI class. To create a custom mobile theme later, see Section 21.6.6, “Mobile Theme”.

```
@Theme("touchkit")
public class MyMobileUI extends UI {
```

5. Build the mobile UI preferring TouchKit components instead of the core Vaadin components, as described in Section 21.6.4, “The UI”.

We cover these and various other tasks in more detail in Section 21.6, “Elements of a TouchKit Application”.

21.6. Elements of a TouchKit Application

At minimum, a TouchKit application requires a UI class, which is defined in a deployment descriptor, as usual for Vaadin applications. You usually define a servlet class, where you can also do some TouchKit-specific configuration. You may also need to have a custom theme. These and other tasks are described in the following subsections.

21.6.1. The Servlet Class

When using a Servlet 3.0 compatible application server, you usually define the UI and make basic configuration with a servlet class with the `@WebServlet` annotation. Vaadin Plugin for Eclipse creates the servlet class as a static inner class of the UI class, while the Maven archetype creates it as a separate class, which is usually the preferred way.

The servlet class must define the UI class as usual. Additionally, you can configure the following TouchKit features in the servlet class:

- Customize bookmark or home screen icon
- Customize splash screen image
- Customize status bar in iOS
- Use special web app mode in iOS
- Provide a fallback UI (Section 21.8.1, “Providing a Fallback UI”)
- Enable offline mode

A custom servlet should normally extend the **TouchKitServlet**. You should place your code in `servletInitialized()` and call the super method in the beginning.

```
public class MyServlet extends TouchKitServlet {
    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();

        ... customization ...
    }
}
```

If you need to rather extend some other servlet, possibly in another add-on, it should be trivial to reimplement the functionality of **TouchKitServlet**, which is just to manage the TouchKit settings object.

If using `web.xml` deployment descriptor instead of the **@WebServlet**, you only need to implement custom servlet class if you need to do any of the above configuration, which you typically need to do.

21.6.2. Defining Servlet and UI with `web.xml` Deployment Descriptor

If using an old style `web.xml` deployment descriptor, you need to define the special **TouchKitServlet** class instead of the regular **VaadinServlet** in the `web.xml` deployment descriptor. Often you need to make some configuration or add special logic in a custom servlet, as described in the previous section, in which case you need to define your servlet in the deployment descriptor.

```
<servlet>
  <servlet-name>Vaadin UI Servlet</servlet-name>
  <servlet-class>
    com.vaadin.addon.touchkit.server.TouchKitServlet
  </servlet-class>
  <init-param>
    <description>Vaadin UI class to start</description>
    <param-name>ui</param-name>
    <param-value>com.example.myapp.MyMobileUI</param-value>
  </init-param>
</servlet>
```

21.6.3. TouchKit Settings

TouchKit has a number of settings that you can customize for your needs. The **TouchKitSettings** configuration object is managed by **TouchKitServlet**, so if you make any modifications to it, you need to implement a custom servlet, as described earlier.

```
public class MyServlet extends TouchKitServlet {
    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();

        TouchKitSettings s = getTouchKitSettings();
        ...
    }
}
```

The settings include special settings for iOS devices, which are contained in a separate **IosWebAppSettings** object, available from the TouchKit settings with `getIosWebAppSettings()`.

Application Icons

The location bar, bookmarks, and other places can display an icon for the web application. You can set the icon, or more exactly icons, in an **ApplicationIcons** object, which manages icons for different resolutions. The most properly sized icon for the context is used. iOS devices prefer icons with 57×57, 72×72, and 144×144 pixels, and Android devices 36×36, 48×48, 72×72, and 96×96 pixels.

You can add an icon to the application icons collection with `addApplicationIcon()`. You can acquire the base URL for your application from the servlet context, as shown in the following example.

```
TouchKitSettings s = getTouchKitSettings();
String contextPath = getServletConfig()
    .getServletContext().getContextPath();
s.getApplicationIcons().addApplicationIcon(
    contextPath + "VAADIN/themes/mytheme/icon.png");
```

The basic method just takes the icon name, while the other one lets you define its size. It also has a *preComposed* parameter, which when true, instructs Safari from adding effects to the icon in iOS.

Viewport Settings

The **ViewPortSettings** object, which you can get from the TouchKit settings with `getViewPortSettings()`, manages settings related to the display, most importantly the scaling limitations.

```
TouchKitSettings s = getTouchKitSettings();
ViewPortSettings vp = s.getViewPortSettings();
vp.setViewportUserScalable(true);
...
```

See the Safari Development Library at the Apple developer's site for more details regarding the functionality in the iOS browser.

Startup Image for iOS

iOS browser supports a startup (splash) image that is shown while the application is loading. You can set it in the **IosWebAppSettings** object with `setStartupImage()`. You can acquire the base URL for your application from the servlet context, as shown in the following example.

```
TouchKitSettings s = getTouchKitSettings();
String contextPath = getServletConfig().getServletContext()
    .getContextPath();
s.getIosWebAppSettings().setStartupImage(
    contextPath + "VAADIN/themes/mytheme/startup.png");
```

Web App Capability for iOS

iOS supports a special web app mode for bookmarks added and started from the home screen. With the mode enabled, the client may, among other things, hide the browser's own UI to give more space for the web application. The mode is enabled by a header that tells the browser whether the application is designed to be used as a web application rather than a web page.

```
TouchKitSettings s = getTouchKitSettings();
s.getIosWebAppSettings().setWebAppCapable(true);
```

See the Safari Development Library at the Apple developer's site for more details regarding the functionality in the iOS browser.

Cache Manifest

The **ApplicationCacheSettings** object manages the cache manifest, which is used to configure how the browser caches the page and other resources for the web app. See Section 21.9, “Offline Mode” for more details about its use.

21.6.4. The UI

Mobile UIs extend the **UI** class as usual and construct the user interface from components.

```
@Theme("mobiletheme")
@Widgetset("com.example.myapp.MyAppWidgetSet")
@Title("My Simple App")
public class SimplePhoneUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Create the content root layout for the UI
        TabBarView mainView = new TabBarView();
        setContent(mainView);

        ...
    }
}
```

As TouchKit comes with a custom widget set, you need to use a combining widget set for your project, defined with the `@Widgetset` annotation for the UI. The combining widget set descriptor is automatically generated by the Vaadin Plugin for Eclipse and in Maven when you install or define the TouchKit add-on.

Most commonly, you will use a combination of the major three TouchKit components as the basis of the UI: **TabBarView**, **NavigationView**, or **NavigationManager**.

If a offline UI is provided, it needs to be enabled in the initialization of the UI, as described in Section 21.9, “Offline Mode”. This code is included in the project stub created by the Maven archetype.

21.6.5. Mobile Widget Set

TouchKit includes a widget set and therefore requires compiling a project widget set that includes it, as described in Chapter 18, *Using Vaadin Add-ons*. The project widget set descriptor is automatically generated during the compilation process, whether you use Maven or the Eclipse plugin.

Note that if you have a TouchKit UI in the same project as a non-TouchKit UI, you probably do not want to compile the TouchKit widget set into its widget set. As the automatic generation of the descriptor includes all the widget sets that it finds from the class path, the result can be unwanted, and you need to edit the widget set descriptor manually.

21.6.6. Mobile Theme

You can use both Sass and CSS themes for TouchKit applications, although they are defined a bit differently from regular Vaadin themes. To optimize how a theme is loaded, you can build it into a GWT client bundle.

Defining a Regular Theme

Using plain CSS is often the easiest way to define a simple theme for a mobile application, as using Sass would not yield all the same benefits as in a regular Vaadin application. TouchKit includes its own base theme in its widget set, so you do not need to `@import` it explicitly.

A CSS theme is defined in a file located at `VAADIN/themes/mymobiletheme/styles.css`. As importing the base does not need to (and should not) be done, it could simply be as follows:

```
.stylishlabel {
  color: red;
  font-style: italic;
}
```

You need to set the theme with the `@Theme("mymobiletheme")` annotation for your UI class, as usual.

You can also use Sass by creating a `styles.scss` and then compiling it to CSS with the Vaadin theme compiler. However, as above, you should not include a base theme. The rules do not need to be wrapped in a selector with the theme name, as is recommended for regular Vaadin themes.

Responsive Mobile Themes

The responsive extension is especially useful for mobile layouts, as it makes it easy to adapt a layout for phones and tablets and for changing the screen orientation. With the extension, changing the UI layout according to screen orientation is handled entirely on the client-side by the add-on, using special CSS selectors in the theme. See Section 9.10, “Responsive Themes” for details.

The Parking Demo uses the extension. From its source code, which is available at Github, you can learn how the conditional selectors are used in the CSS defined in a GWT client bundle.

For example, the CSS for the **Stats** tab in the Parking demo defines a responsive selector as follows, to allow fitting two charts side-by-side if there is enough room horizontally:

```
.stats .statschart {
  margin-bottom: 30px;
  float: left;
  width: 100%;
}

.v-ui[width-range~="801px-"] .stats .statschart {
  width: 48% !important;
  margin: 0 1%;
}
```

Normally, if there's 800 pixels or less space horizontally, each chart takes 100% of the screen width, causing the second one to wrap to the next line in the containing **CssLayout**. If there is more space, the two charts are shown in 48% width, so that both can fit in the same line.

Defining a Theme in a GWT Client Bundle

Using a GWT theme instead of a regular Vaadin theme offers several performance benefits on mobile devices by reducing the number of resources loaded separately. All the resources, such as images and stylesheets, can be loaded with the widget set. Images can be handled as sprites tiled in bundle images.

The GWT CSS classes have their own special format, a bit similar to Sass themes. See GWT Developer's Guide for detailed information about client bundles and how to define image, CSS, and other resources.

To use a GWT client bundle in a TouchKit application, you need to define a *theme loader* that extends the TouchKit **ThemeLoader** and implements the `load()` method to inject the bundle. The theme loader and the client bundle are a client-side classes that are compiled into the widget set, and must therefore be defined under the `client` directory.

For example, in the Parking Demo we have as follows:

```
public class ParkingThemeLoader extends ThemeLoader {
    @Override
    public final void load() {
        // First load the default TouchKit theme...
        super.load();

        // ... and add Parking Demo CSS from its own bundle
        ParkingBundle.INSTANCE.fontsCss().ensureInjected();
        ParkingBundle.INSTANCE.css().ensureInjected();
        ParkingBundle.INSTANCE.ticketsCss().ensureInjected();
        ParkingBundle.INSTANCE.statsCss().ensureInjected();
        ParkingBundle.INSTANCE.shiftsCss().ensureInjected();
        ParkingBundle.INSTANCE.mapCss().ensureInjected();
    }
}
```

You can call `super.load()` to load the default TouchKit theme, but you can omit the call if you do not want to use it. In such case, your GWT theme should import the Vaadin base theme explicitly.

The theme loader must be defined in the `.gwt.xml` widget set descriptor as follows:

```
<replace-with
    class="com.vaadin.demo.parking.widgetset.client.theme.ParkingThemeLoader">

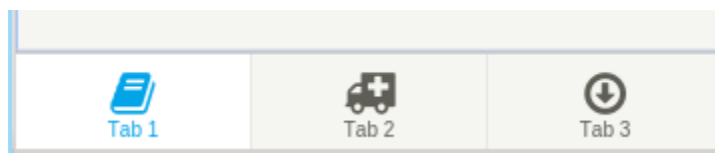
    <when-type-is
        class="com.vaadin.addon.touchkit.gwt.client.ThemeLoader" />
</replace-with>
```

See the Parking Demo sources for a complete example of defining a GWT theme.

21.6.7. Using Font Icons

You can use font icons, as described in Section 9.8, “Font Icons”, also with most TouchKit components.

Figure 21.3. Font Icons in TabBarView



For example, as is done in the UI stub of a TouchKit project created from the Maven archetype:

```
// Have a tab bar with multiple tab views
TabBarView tabBarView = new TabBarView();

// Have a tab
... create view1 ...
Tab tab1 = tabBarView.addTab(view1);

// Use the "book" icon for the tab
tab1.setIcon(FontAwesome.BOOK);
```

21.7. Mobile User Interface Components

TouchKit introduces a number of components special to mobile user interfaces to give better user interaction and to utilize the special features in mobile devices.

NavigationView

A view with a navigation bar (**NavigationBar** for navigating back and forth in a **NavigationManager**.

Toolbar

A horizontal layout especially for buttons. A sub-component of **TabBarView** or **NavigationView**.

NavigationManager

A component container that enables slide animations between the components while the server request is being made for the purpose of latency compensation. The components are typically **NavigationViews** or **SwipeViews**.

NavigationButton

A special button for initiating view change in a **NavigationManager** on the client-side, for the purpose of latency compensation.

Popover

A floating pop-up frame that can be positioned relative to a component.

SwipeView

A view for navigating back and forth in a **NavigationManager** using horizontal swipe gestures.

Switch

A sliding on/off toggle for boolean values.

VerticalComponentGroup

A vertical layout for grouping components.

HorizontalButtonGroup

A horizontal layout for grouping especially buttons.

TabBarView

A tabbed view with a content area on the top and a **Toolbar** for navigating between sub-views on the bottom.

EmailField, **NumberField**, and *[classname]UrlField*

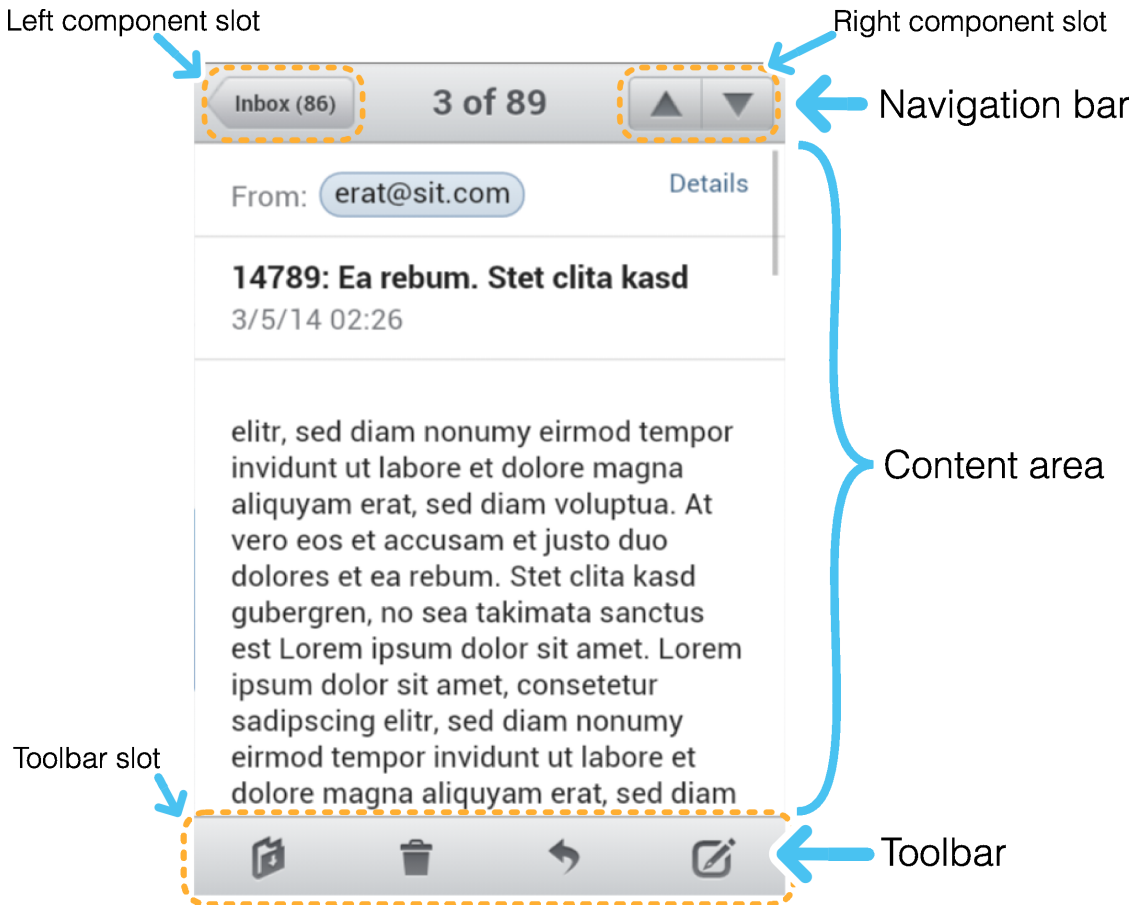
Text fields for inputting specifically email addresses, numbers, and URLs, respectively, with a specific virtual keyboard.

The components are detailed in the following subsections.

21.7.1. NavigationView

The **NavigationView** is a layout component that consists of a navigation bar and a content area. The content area is scrollable, so there is no need to use an inner panel component. In addition, there can be an optional toolbar component at the bottom of the view. A **NavigationView** is often used inside a **NavigationManager** to get view change animations.

Figure 21.4. Layout of the NavigationView



NavigationView has a full size by default. The content area is expanding, so that it takes all the space left over from the navigation bar and toolbar.

Navigation Bar

The navigation bar at the top of **NavigationView** is a separate **NavigationBar** component. It has two component slots, with one on the left and one on the right. The caption is displayed in the middle. The **NavigationBar** component can be used independently as well.

When the **NavigationBar** is used for navigation and you set the previous component with `setPreviousComponent()`, the left slot is automatically filled with a **Back** button. This is done automatically if you use the **NavigationView** inside a **NavigationManager**.

You can get access to the navigation bar component with `getNavigationBar()` to use its manipulator methods directly, but **NavigationView** also offers some shorthand methods: `setLeftComponent()`, `setRightComponent()`, and a setter and a getter for the caption.

Toolbar

A slot for an optional toolbar is located at the bottom of the **NavigationView**. The toolbar can be any component, but a **Toolbar** component made for this purpose is included in TouchKit. It is described in Section 21.7.2, “**Toolbar**”. You could also use a **HorizontalLayout** or **CssLayout**.

You usually fill the tool bar with **Button** components with an icon and no textual caption. You set the toolbar with `setToolbar()`.

21.7.2. Toolbar

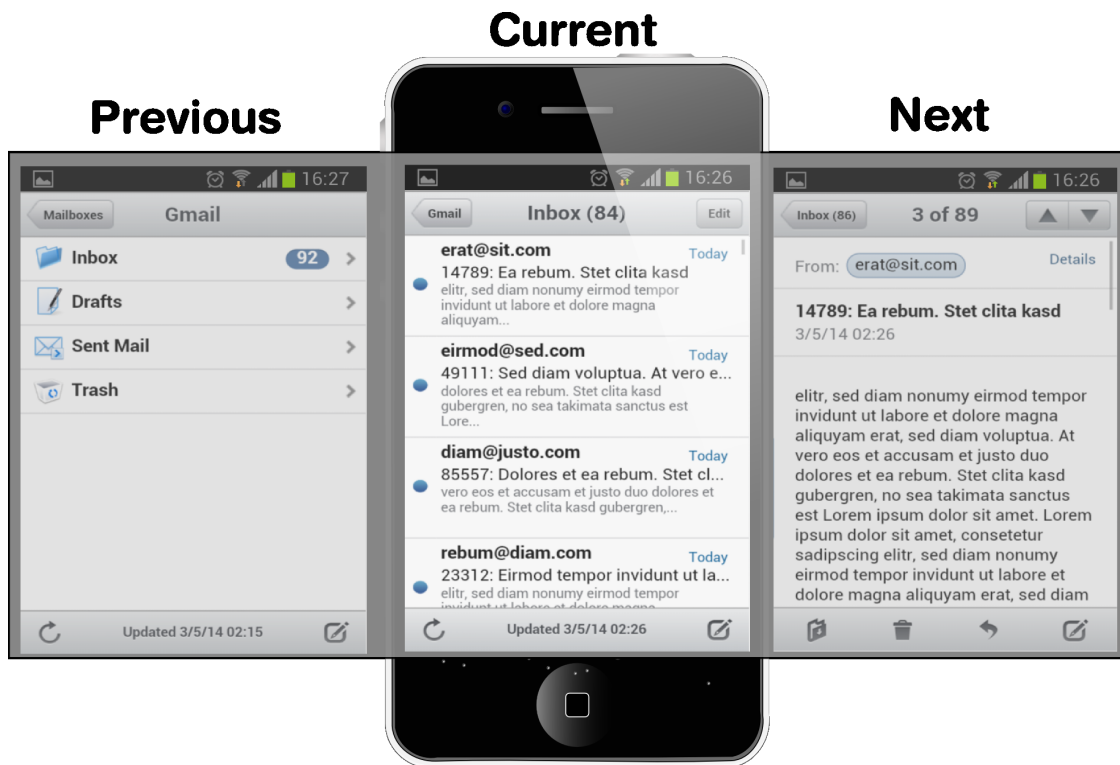
The **Toolbar** is a horizontal layout component intended for containing **Button** components. The toolbar has by default 100% horizontal width and a fixed height. The components are spread evenly in the horizontal direction. **Toolbar** is used in a **TabBarView**, as described in Section 21.7.10, “**TabBarView**”.

For a description of the inherited features, please refer to Section 7.3, “**VerticalLayout** and **HorizontalLayout**”.

21.7.3. NavigationManager

The **NavigationManager** is a visual effect component that gives sliding animation when switching between views. You can register three components: the currently displayed component, the previous one on the left, and the next component on the right. You can set these components with `setCurrentComponent()`, `setPreviousComponent()`, and `setNextComponent()`, respectively.

The **NavigationManager** component is illustrated in Figure 21.5, “**NavigationManager** with Three **NavigationViews**”.

Figure 21.5. NavigationManager with Three NavigationViews

The navigation manager is important for responsiveness, because the previous and next components are cached and the slide animation started before server is contacted to load the new next or previous views.

You give the initial view as a parameter for the constructor. Typically, you use a navigation manager as the UI content or inside a **TabBarView**.

```
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        NavigationManager manager =
            new NavigationManager(new MainView());
        setContent(manager);
    }
}
```

Changing Views

Switching between the views (components) is normally done with predefined navigation targets to enhance responsiveness. Clicking a **NavigationButton** or a button in a navigation bar starts navigation automatically without a server roundtrip. Swipe gestures are supported with the **SwipeView** component.

Navigation can also be done programmatically with the `navigateTo()` method. If breadcrumbs are enabled, the current view is also pushed to the breadcrumb stack. To navigate back, you can call `navigateBack()`, which is also called implicitly if a **Back** button is clicked in a **NavigationView**. Also, if navigation is done to the "previous" component, `navigateBack()` is done implicitly.

When navigation occurs, the current component is moved as the previous or next component, according to the direction of the navigation.

Handling View Changes

While you can put any components in the manager, some special features are enabled when using the **NavigationView**. When a view becomes visible, the `onBecomingVisible()` method in the view is called. You can override it, just remember to call the superclass method.

```
@Override
protected void onBecomingVisible() {
    super.onBecomingVisible();

    ...
}
```

Otherwise, you can handle navigation changes in the manager with a `NavigationListener`. The *direction* property tells whether the navigation was done forward or backward in the breadcrumb stack, that is, whether navigation was done with `navigateTo()` or `navigateBack()`. The current component, accessible with `getCurrentComponent()`, refers to the navigation target component.

```
manager.addNavigationListener(new NavigationListener() {
    @Override
    public void navigate(NavigationEvent event) {
        if (event.getDirection() ==
            NavigationEvent.Direction.BACK) {
            // Do something
            Notification.show("You came back to " +
                manager.getCurrentComponent().getCaption());
        }
    }
});
```

Tracking Breadcrumbs

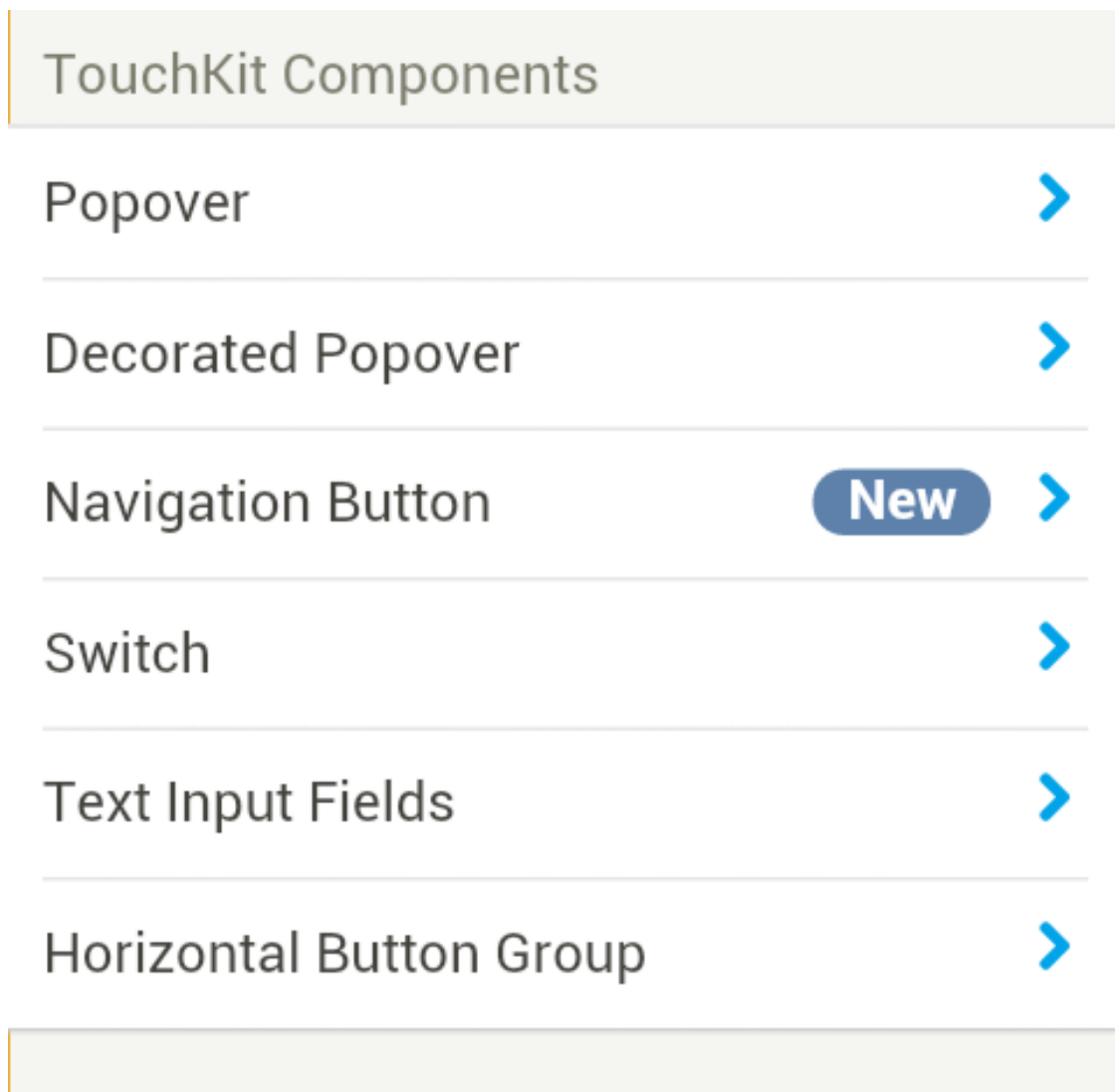
NavigationManager also handles *breadcrumb* tracking. The `navigateTo()` pushes the current view on the top of the breadcrumb stack and `navigateBack()` can be called to return to the previous breadcrumb level.

Notice that calling `navigateTo()` with the "previous" component is equivalent to calling `navigateBack()`.

21.7.4. NavigationButton

The **NavigationButton** is a special version of the regular **Button** component, designed for navigation inside a **NavigationManager** (see Section 21.7.3, "**NavigationManager**"). Clicking a navigation button will automatically navigate to the defined target view. The view change animation does not need to make a server request first, but starts immediately after clicking the button. If you leave the target view empty, an empty placeholder view is shown in the animation. The view is filled after it gets the content from the server.

A navigation button does not have a particular border by default, because multiple navigation buttons are typically used inside a **VerticalComponentGroup** to create menus, as illustrated in Figure 21.6, "**NavigationButtons Inside a Vertical Component Group**".

Figure 21.6. NavigationButtons Inside a Vertical Component Group

A navigation button has a caption and can have a description and an icon. If not given explicitly, the caption is taken from the caption of the navigation view if it is initialized before the button. The icon is positioned left of the caption, and the description is aligned on the right side of the button.

You can give the target view either in the constructor or with `setTargetView()`, or create it later by handling the button click.

```
// Button caption comes from the view caption
box.addComponent(new NavigationButton(new PopoverView()));

// Give button caption explicitly
box.addComponent(new NavigationButton("Decorated Popover",
    new DecoratedPopoverView()));
```

If the target view is not created or initialized before the button is clicked, it does not yet have a caption during the animation. The default is to use the button caption as a temporary target view caption, but you can set it explicitly with `setTargetViewCaption()`. The temporary caption

is shown during the slide animation and until the content for the view has been received from the server. It is then replaced with the proper caption of the view, and you normally want to have it the same. The temporary caption is also used as the caption of button if it is not given explicitly.

```
final NavigationButton navButton = new NavigationButton();
navButton.setTargetViewCaption("Text Input Fields");
navButton.addClickListener(
    new NavigationButtonClickListener() {

        @Override
        public void buttonClick(NavigationButtonClickEvent event) {
            navButton.getNavigationManager()
                .navigateTo(new FieldView());
        }
    });
box.addComponent(navButton);
```

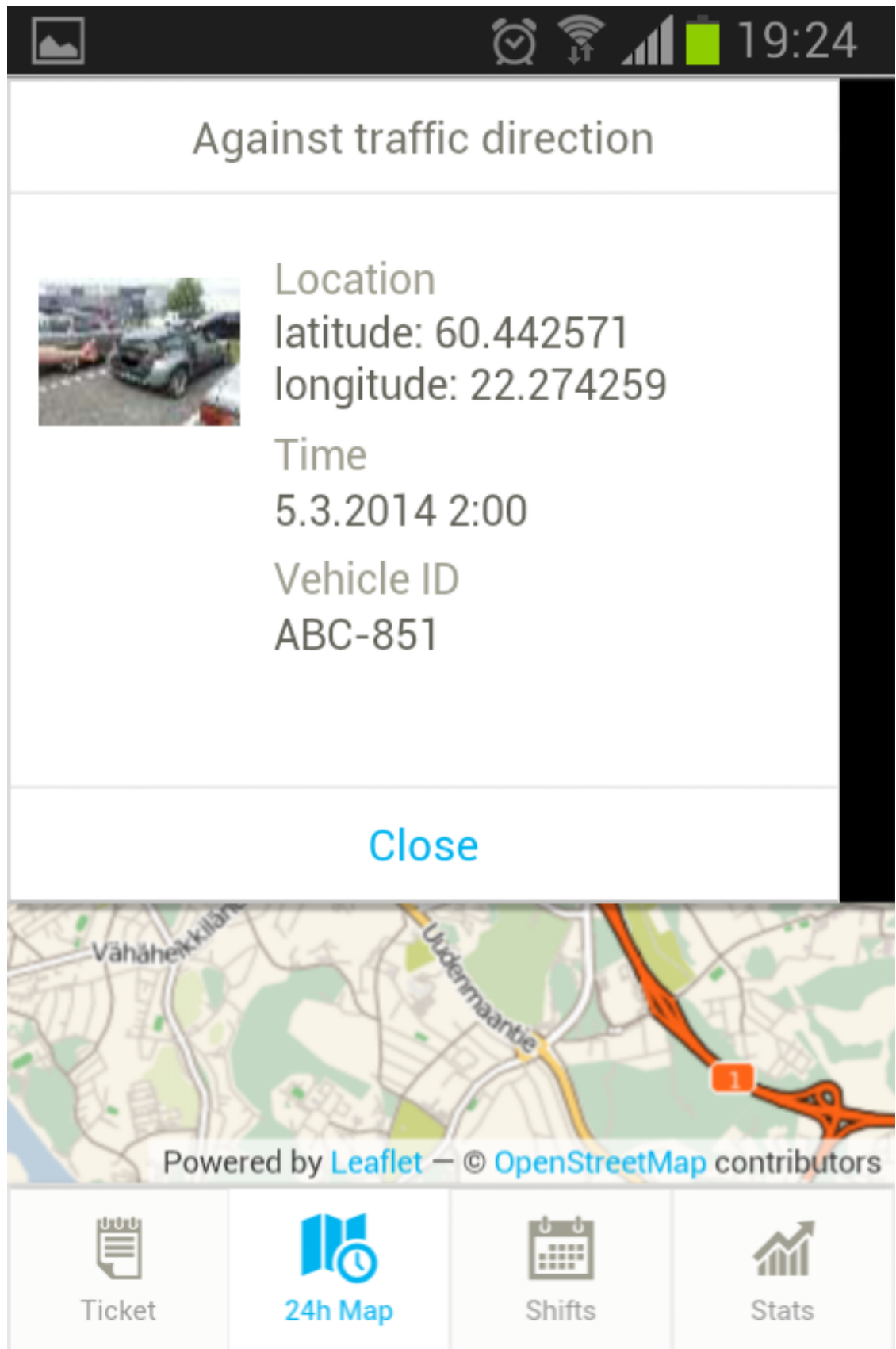
Creating views dynamically this way is recommended to reduce the memory footprint.

Notice that the automatic navigation will only work if the button is inside a **NavigationManager** (in a view managed by it). If you just want to use the button as a visual element, you can use it like a regular **Button** and handle the click events with a **ClickListener**.

21.7.5. Popover

Popover is much like a regular Vaadin sub-window, useful for quickly displaying some options or a small form related to an action. Unlike regular sub-windows, it does not support dragging or resizing by the user. As sub-windows usually require a rather large screen size, the **Popover** is most useful for tablet devices. When used on smaller devices, such as phones, the **Popover** automatically fills the entire screen.

Figure 21.7. Popover in a Phone



It is customary to use a **NavigationView** to have border decorations and caption. In the following, we subclass **Popover** to create the content.

```
class DetailsPopover extends Popover {
    public DetailsPopover() {
        setWidth("350px");
        setHeight("65%");

        // Have some details to display
        VerticalLayout layout = new VerticalLayout();
        ...

        NavigationView c = new NavigationView(layout);
        c.setCaption("Details");
        setContent(c);
    }
}
```

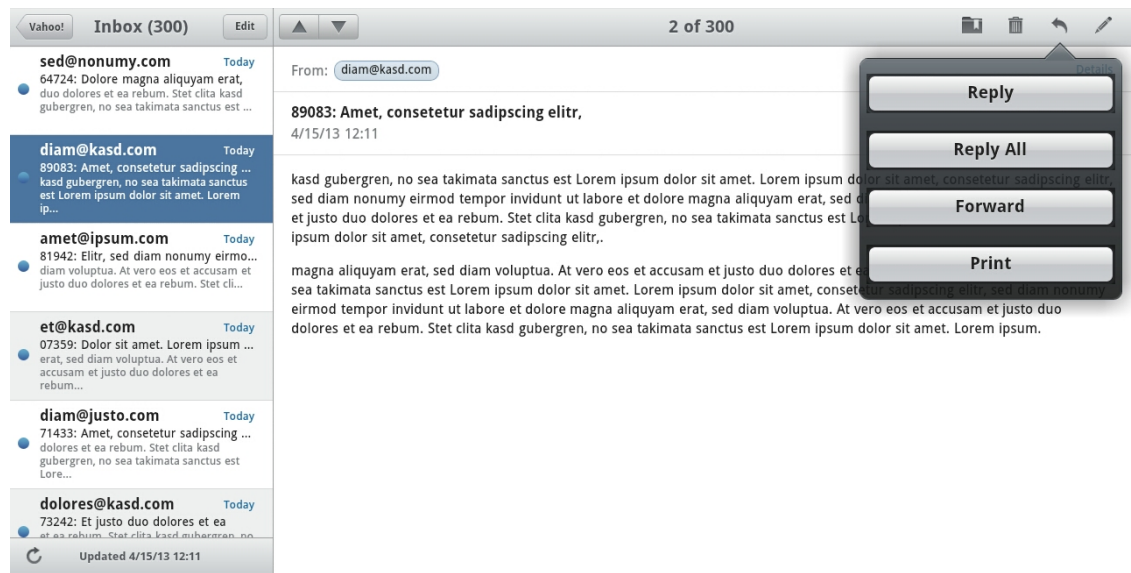
A **Popover** can be opened relative to a component by calling `showRelativeTo()`. In the following example, we open the popover when a table item is clicked.

```
Table table = new Table("Planets", planetData());
table.addItemClickListener(new ItemClickListener() {
    @Override
    public void itemClick(ItemClickEvent event) {
        DetailsPopover popover = new DetailsPopover();

        // Show it relative to the navigation bar of
        // the current NavigationView.
        popover.showRelativeTo(view.getNavigationBar());
    }
});
```

You can also add the **Popover** to the **UI** with `addWindow()`.

A popover is shown in a tablet device as illustrated Figure 21.8, “**Popover** in a Tablet Device”. In this example, we have a **CssLayout** with some buttons as the popover content.

Figure 21.8. Popover in a Tablet Device

21.7.6. SwipeView

The **SwipeView** is a wrapper that allows navigating between views by swiping them horizontally left or right. The component works together with a **NavigationManager** (see Section 21.7.6, “**SwipeView**”) to change between the views when swiped, and to animate the change. A **SwipeView** should be an immediate child of the **NavigationManager**, but can contain a **NavigationView** to provide button navigation as well.

Let us have a selection of photographs to browse. We extend **NavigationManager** that creates the slide effect and create actual image views dynamically. In the constructor, we create the two first ones.

```
class SlideShow extends NavigationManager
    implements NavigationListener {
    String imageNames[] = {"Mercury.jpg", "Venus.jpg",
        "Earth.jpg", "Mars.jpg", "Jupiter.jpg",
        "Saturn.jpg", "Uranus.jpg", "Neptune.jpg"};
    int pos = 0;

    public SlideShow() {
        // Set up the initial views
        navigateTo(createView(pos));
        setNextComponent(createView(pos+1));

        addNavigationListener(this);
    }
}
```

The individual views have a **SwipeView** and the top.

```
SwipeView createView(int pos) {
    SwipeView view = new SwipeView();
    view.setSizeFull();

    // Use an inner layout to center the image
    VerticalLayout layout = new VerticalLayout();
```

```

        layout.setSizeFull();

        Image image = new Image(null, new ThemeResource(
            "planets/" + imageNames[pos]));
        layout.addComponent(image);
        layout.setComponentAlignment(image,
            Alignment.MIDDLE_CENTER);

        view.setContent(layout);
        return view;
    }

```

When the view is swiped to either direction, we need to set the next image in that direction dynamically in the **NavigationManager**.

```

@Override
public void navigate(NavigationEvent event) {
    switch (event.getDirection()) {
        case FORWARD:
            if (++pos < imageNames.length-1)
                setNextComponent(createView(pos+1));
            break;
        case BACK:
            if (--pos > 0)
                setPreviousComponent(createView(pos-1));
    }
}

```

21.7.7. Switch

The **Switch** component is a two-state selector that can be toggled either by tapping or sliding and looks like the switch button in Apple iOS. It extends **CheckBox** and has therefore **Boolean** value type. The caption is managed by the containing layout.

```

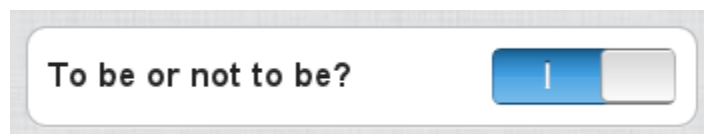
VerticalComponentGroup group =
    new VerticalComponentGroup();
Switch myswitch = new Switch("To be or not to be?");
myswitch.setValue(true);
group.addComponent(myswitch);

```

As with other field components, you can handle value changes with a `ValueChangeListener`. Use `setImmediate(true)` to get them immediately when toggled.

The result is shown in Figure 21.9, “**Switch**”.

Figure 21.9. Switch



21.7.8. VerticalComponentGroup

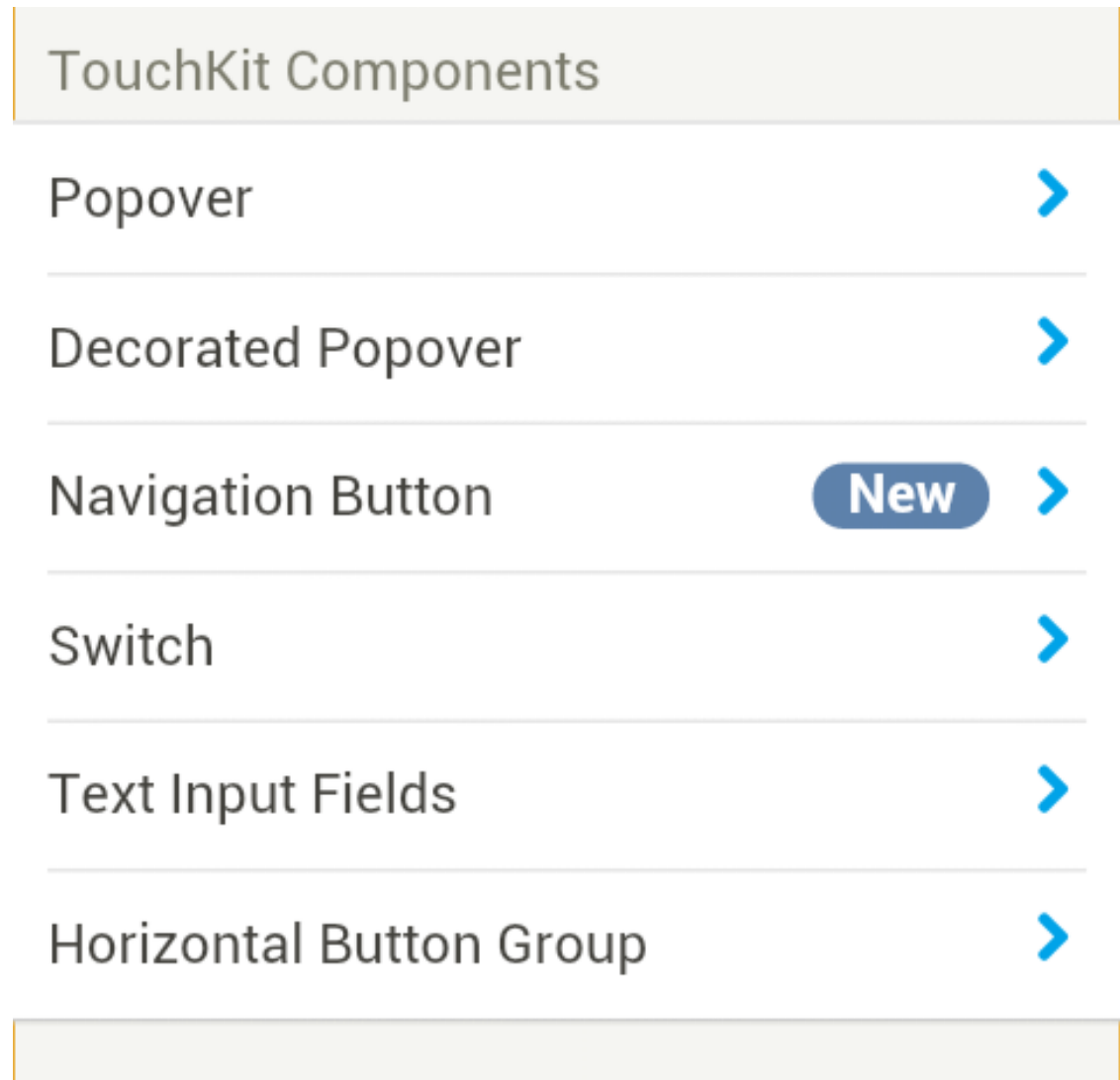
The **VerticalComponentGroup** is a layout component for grouping components in a vertical stack with a border. Component captions are placed left of the components, and the components

are aligned right. The component group is typically used for forms or with **NavigationButton** to create navigation menus.

```
VerticalComponentGroup group =  
    new VerticalComponentGroup("TouchKit Components");  
group.setWidth("100%");  
  
// Navigation to sub-views  
group.addComponent(new NavigationButton(  
    new PopoverView()));  
group.addComponent(new NavigationButton(  
    new DecoratedPopoverView()));  
  
layout.addComponent(box);
```

The result is shown in Figure 21.10, “**VerticalComponentGroup**”.

Figure 21.10. VerticalComponentGroup



21.7.9. HorizontalButtonGroup

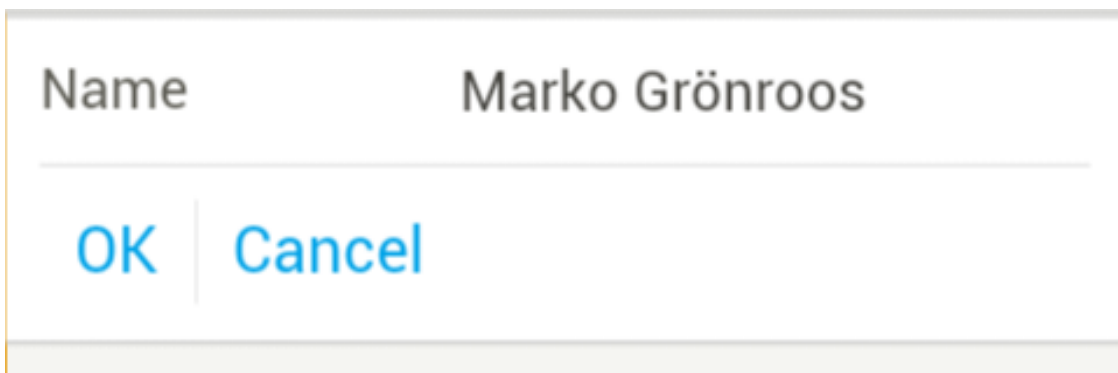
The **HorizontalButtonGroup** is intended for grouping buttons inside the slots of a **VerticalComponentGroup** with a special button group style.

```
VerticalComponentGroup vertical =
    new VerticalComponentGroup();
vertical.addComponent(new TextField("Name"));

HorizontalButtonGroup buttons =
    new HorizontalButtonGroup();
buttons.addComponent(new Button("OK"));
buttons.addComponent(new Button("Cancel"));
vertical.addComponent(buttons);
```

The result is shown in Figure 21.11, “**HorizontalButtonGroup**”

Figure 21.11. HorizontalButtonGroup

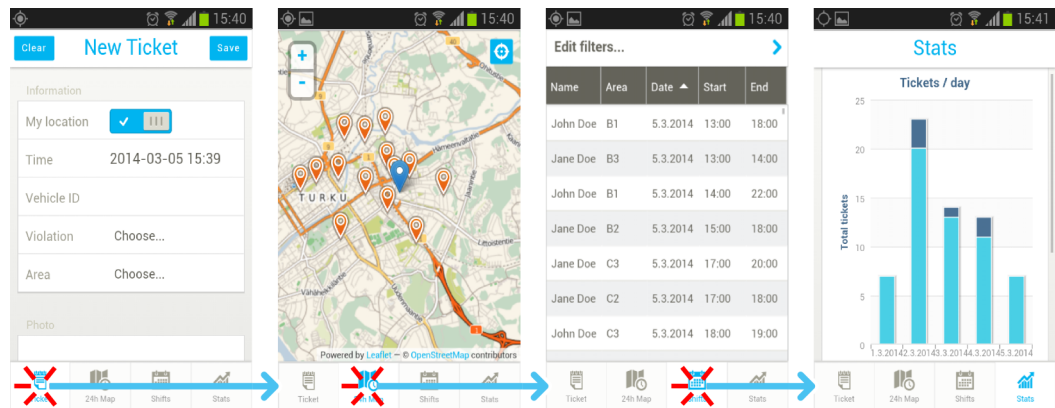


You can also make single buttons prettier by wrapping them in the component. Also the **Upload** component has a button, and you can give it the `v-button` style to make it look like a button would in the group

Despite the name, and the fact that the button group is intended for buttons, you can, in fact, put any components inside it. Whether the result is meaningful, depends on the component.

21.7.10. TabBarView

The **TabBarView** is a layout component that consist of a tab bar at the bottom of the screen and a content area. Each tab has a content component which is displayed when the tab is selected.

Figure 21.12. TabBar with Four NavigationViews

TabBarView implements `ComponentContainer`, but uses its own specialized API for manipulating tabs. To add a new tab, you need to call `addTab()` with the content component. It creates the tab and returns a **Tab** object for managing it. You should set at least the caption and icon for a tab.

```
TabBarView bar = new TabBarView();

// Create some Vaadin component to use as content
Label content = new Label("Really simple content");

// Create a tab for it
Tab tab = bar.addTab(label);

// Set tab name and/or icon
tab.setCaption("tab name");
tab.setIcon(new ThemeResource(...));
```

A tab can be removed with `removeTab()`. Note that the `ComponentContainer` methods `addComponent()` and `removeComponent()` will throw an **UnsupportedOperationException** if used.

21.7.11. EmailField

The **EmailField** is just like the regular **TextField**, except that it has automatic capitalization and correction turned off. Mobile devices also recognize the field as an email field and can offer a virtual keyboard for the purpose, so that it includes the at (`@`) and period (`.`) characters, and possibly a shorthand for `.com`.

Figure 21.13. EmailField while editing

21.7.12. NumberField

The **NumberField** is just like the regular **TextField**, except that it is marked as a numeric input field for mobile devices, so that they will show a numeric virtual keyboard rather than the default alphanumeric.

Figure 21.14. NumberField while editing

The screenshot displays a mobile application interface with three input fields stacked vertically. The first field is labeled 'Email' and contains the text 'magi@vaadin.com'. The second field is labeled 'Number!' (with the exclamation mark in red) and contains the text '42'. A blue horizontal line is positioned below the 'Number!' field. The third field is labeled 'URL' and is currently empty. A virtual numeric keypad is overlaid on the bottom half of the screen, featuring buttons for digits 1-9, 0, a backspace key (arrow and 'x'), a 'Next' button, a '123 Sym' button, and a microphone icon. A blue 'OK' button is visible above the keypad.

21.7.13. UrlField

The **UrlField** is just like the regular **TextField**, except that it is marked as a URL input field for mobile devices, so that they will show a URL input virtual keyboard rather than the default alpha-numeric. It has convenience methods `getUrl()` and `setUrl(URL url)` for converting input value from and to `java.net.URL`.

21.8. Advanced Mobile Features

21.8.1. Providing a Fallback UI

You may need to use the same URL and hence the same servlet for both the mobile TouchKit UI and for regular browsers. In this case, you need to recognize the mobile browsers compatible with Vaadin TouchKit and provide a fallback UI for any other browsers. The fallback UI can be a regular Vaadin UI, a "Sorry!" message, or a redirection to an alternate user interface.

You can handle the fallback logic in a custom **UIProvider** that creates the UIs in the servlet. As TouchKit supports only WebKit and Windows Phone browsers, you can do the recognition by checking if the *user-agent* string contains the sub-strings "webkit" or "windows phone" as follows:

```
public class MyUIProvider extends UIProvider {
    @Override
    public Class<? extends UI>
        getUIClass(UIClassSelectionEvent event) {
        String ua = event.getRequest()
            .getHeader("user-agent").toLowerCase();
        if ( ua.toLowerCase().contains("webkit")
            || ua.toLowerCase().contains("windows phone 8")
            || ua.toLowerCase().contains("windows phone 9")) {
            return MyUI.class;
        } else {
            return MyFallbackUI.class;
        }
    }
}
```

The custom UI provider has to be added in a custom servlet class, which you need to define in the `web.xml`, as described in Section 21.6.3, "TouchKit Settings". For example, as follows:

```
public class MyServlet extends TouchKitServlet {
    private MyUIProvider uiProvider = new MyUIProvider();

    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();

        getService().addSessionInitListener(
            new SessionInitListener() {
                @Override
                public void sessionInit(SessionInitEvent event)
                    throws ServiceException {
                    event.getSession().addUIProvider(uiProvider);
                }
            });

        ... other custom servlet settings ...
    }
}
```

See the Parking Demo for a working example.

21.8.2. Geolocation

The geolocation feature in TouchKit allows receiving the geographical location from the mobile device. The browser will ask the user to confirm that the web site is allowed to get the location information. Tapping **Share Location** gives the permission. The browser will give the position acquired by GPS, cellular positioning, or Wi-Fi positioning, as enabled in the device.

Geolocation is requested by calling the static `detect()` method in **Geolocator**. You need to provide a **PositionCallback** handler that is called when the device has an answer for your request. If the geolocation request succeeds, `onSuccess()` is called. Otherwise, for example, if the user did not allow sharing of his location, `onFailure()` is called. The geolocation data is provided in a **Position** object.

```
Geolocator.detect(new PositionCallback() {  
    public void onSuccess(Position position) {  
        double latitude = position.getLatitude();  
        double longitude = position.getLongitude();  
        double accuracy = position.getAccuracy();  
  
        ...  
    }  
  
    public void onFailure(int errorCode) {  
        ...  
    }  
});
```

The position is given as degrees with fractions in the WGS84 coordinate system used by GPS. The longitude is positive to East and negative to West of the prime meridian of WGS84. The accuracy is given in meters. In addition to the above data, the following are also provided:

- Altitude
- Altitude accuracy
- Heading
- Speed

If any of the position data is unavailable, its value will be zero.

The `onFailure()` is called if the positioning fails for some reason. The `errorCode` explains the reason. Error 1 is returned if the permission was denied, 2 if the position is unavailable, 3 on positioning timeout, and 0 on an unknown error.

Notice that geolocation can take significant time, depending on the location method used by the device. With Wi-Fi and cellular positioning, the time is usually less than 30 seconds. With unassisted GPS, it can reach 15 minutes on a fresh device and even longer if the reception is bad. However, once a location fix has been made, updates to the location will be faster. If you are making navigation software, you need to update the position data fairly frequently by calling the `detect()` method in **Geolocator** multiple times.

21.8.3. Storing Data in the Local Storage

The **LocalStorage** UI extension allows storing data in the HTML5 local storage from the server-side application. The storage is a singleton, which you can get with `LocalStorage.get()`.

```
final LocalStorage storage = LocalStorage.get();
```

Storing Data

You can store data in the local storage as key-value pairs with the `put()` method. Both the key and value must be strings. Storing the data requires a client round-trip, so the success or failure of the store operation can be handled with an optional `LocalStorageCallback`.

```
// Editor for the value to be stored
final TextField valueEditor = new TextField("Value");
valueEditor.setNullRepresentation("");
layout.addComponent(valueEditor);

Button store = new Button("Store", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        storage.put("value", valueEditor.getValue(),
            new LocalStorageCallback() {
                @Override
                public void onSuccess(String value) {
                    Notification.show("Stored");
                }

                @Override
                public void onFailure(FailureEvent error) {
                    Notification.show("Storing Failed");
                }
            });
    }
});
layout.addComponent(store);
```

Retrieving Data from the Storage

You can retrieve data from the local storage with the `get()` method. It takes the key and a `LocalStorageCallback` to receive the retrieved value, or a failure. Retrieving the value to the server-side requires a client round-trip and another server request is made to send the value with the callback.

```
storage.get("value", new LocalStorageCallback() {
    @Override
    public void onSuccess(String value) {
        valueEditor.setValue(value);
        Notification.show("Value Retrieved");
    }

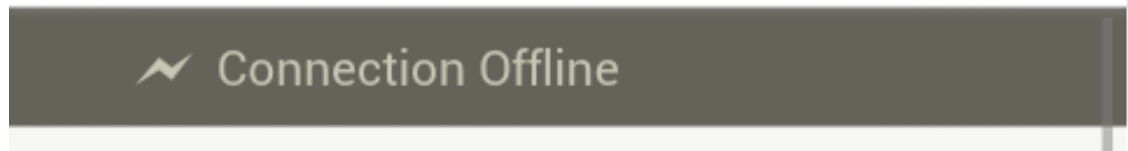
    @Override
    public void onFailure(FailureEvent error) {
        Notification.show("Failed because: " +
            error.getMessage());
    }
});
```

21.9. Offline Mode

While regular Vaadin TouchKit applications are server-side applications, TouchKit allows a special *offline mode*, which is a client-side Vaadin UI that is switched to automatically when the network connection is not available. The offline UI is included in the widget set of the regular

server-side UI and stored in the browser cache. By providing a special cache manifest, the browser caches the page so strongly that it persists even after browser restart, effectively making it an offline application.

Figure 21.15. Offline Mode in the Parking Demo



During offline operation, the offline UI can store data in the HTML5 local storage of the mobile browser and then passed to the server-side application when the connection is again available.

The offline mode is enabled in the project stub created by the Maven archetype (see Section 21.5.1, “Using the Maven Archetype”), with stubs for the offline data storage and server RPC classes.

See the Parking demo and its source code for complete examples of the offline mode.

21.9.1. Enabling the Cache Manifest

HTML5 supports a *cache manifest*, which makes offline web applications possible. It controls how different resources are cached. The manifest is generated by TouchKit, but you need to enable it in the TouchKit settings. To do so, you need to define a custom servlet, as described in Section 21.6.1, “The Servlet Class”, and call `setCacheManifestEnabled(true)` for the cache settings, as follows:

```
TouchKitSettings s = getTouchKitSettings();
...
s.getApplicationCacheSettings()
    .setCacheManifestEnabled(true);
```

You also need to define a MIME type for the manifest in the `web.xml` deployment descriptor as follows:

```
<mime-mapping>
  <extension>manifest</extension>
  <mime-type>text/cache-manifest</mime-type>
</mime-mapping>
```

Offline Mode is enabled by default.

21.9.2. Disabling Offline Mode

Offline Mode can be disabled adding the `@OfflineModeEnabled(false)` annotation to the UI class. When offline mode is disabled Vaadin Framework default reconnect dialog will be used if connection is lost.

21.9.3. Configuring Offline Mode

Additional configuration can be defined adding the **OfflineMode** extension to the UI.

```
public class OfflineTest extends UI {
```

```
@Override
protected void init(VaadinRequest request) {
    OfflineMode offlineMode = new OfflineMode();
    // Maintain the session when the browser app closes
    offlineMode.setPersistentSessionCookie(true);

    // Define the timeout in secs to wait when a server
    // request is sent before falling back to offline mode
    offlineMode.setOfflineModeTimeout(15);
    offlineMode.extend(this);

    buildUi();
}

...
}
```

You can extend the **OfflineMode** extension to transfer data conveniently from the offline UI to the server-side, as described in Section 21.9.5, “Sending Data to Server”.

21.9.4. The Offline User Interface

An offline mode is built like any other client-side module, as described in Chapter 14, *Client-Side Vaadin Development*. You can use any GWT, Vaadin, add-on, and also TouchKit widgets in the offline user interface.

Most typically, a client-side application builds a simplified UI for data browsing and entry. It stores the data in the HTML5 local storage. It watches if the server connection is restored, and if it is, it sends any collected data to the server and suggests to return to the online mode.

The Parking Demo provides an example implementation of an offline mode user interface. The Ticket view is implemented as a fat client-side widget, where the server-side view only communicates the state to the widget.

21.9.5. Sending Data to Server

Once the connection is available, the offline UI can send any collected data to the server-side. You can send the data from the offline UI, for example, by making a server RPC call to a server-side UI extension, as described in Section 17.6, “RPC Calls Between Client- and Server-Side”.

21.9.6. The Offline Theme

Normally, client-side modules have their own stylesheets in the `public` folder that is compiled into the client-side target, as described in Section 17.8, “Styling a Widget” and Section 14.3.1, “Specifying a Stylesheet”. However, you may want to have the offline mode have the same visual style as the online mode. To use the same theme as the server-side application, you need to define the theme path in the widget set definition file as follows.

```
<set-configuration-property
    name='touchkit.manifestlinker.additionalCacheRoot'
    value='src/main/webapp/VAADIN/themes/mytheme:../../VAADIN/themes/mytheme'
/>
```

You need to follow a CSS style structure required by the Vaadin theme in your offline application. If you use any Vaadin widgets, as described in Section 16.3, “Vaadin Widgets”, they will use the Vaadin theme.

21.10. Building an Optimized Widget Set

Mobile networks are generally somewhat slower than DSL Internet connections. When starting a Vaadin application, the widget set is the biggest resource that needs to be loaded in the browser. As most of the Vaadin components are not used by most applications, especially mobile ones, it is beneficial to create an optimized version of the widget set.

Vaadin supports lazy loading of individual widget implementations when they are needed. The **TouchKitWidgetSet** used in TouchKit applications optimizes the widgetset to only download the most essential widgets first and then load other widget implementation lazily. This is a good compromise for most TouchKit applications. Nevertheless, because of the high latency of most mobile networks, loading the widget set in small pieces might not be the best solution for every case. With custom optimization, you can create a monolithic widget set stripped off all unnecessary widgets. Together with proper GZip compression, it should be quite light-weight for mobile browsers.

However, if the application has big components which are rarely used or not on the initial views, it may be best to load those widgets eagerly or lazily.

You can find a working example of widget set optimization in the `ParkingWidgetset.gwt.xml` and `WidgetLoaderFactory.java` in the Parking Demo sources.

21.11. Testing and Debugging on Mobile Devices

Testing places special challenges for mobile devices. The mobile browsers may not have much debugging features and you may not be able to install third-party debugging add-ons, such as the Chrome Developer Tools.

21.11.1. Debugging

The debug window, as described in Section 12.3, “Debug Mode and Window”, works on mobile browsers as well, even if it is a bit harder to use.

The lack of in-browser analysis tools can be helped with simple client-side coding. For example, you can dump the HTML content of the page with the `innerHTML` property in the HTML DOM. To do so, you need to execute a JavaScript call from the server-side and handle its response with a call-back method, as described in Section 12.14.2, “Handling JavaScript Function Call-backs”.

Desktop Debugging

TouchKit supports especially WebKit-based browsers, which are used in iOS and Android devices. You can therefore reach a decent compatibility by using a desktop browser based on WebKit, such as Google Chrome. Features such as geolocation are also supported by desktop browsers. If you make your phone/tablet-detection and orientation detection using screen size, you can easily emulate the modes by resizing the browser. Also, the browsers have special development settings to emulate some features in touch devices.

Remote Debugging

Both Safari and Chrome support remote debugging, which allows you to debug the supported mobile browsers remotely from a desktop browser.

Chapter 22

Vaadin Spreadsheet

22.1. Overview	615
22.2. Installing Vaadin Spreadsheet	619
22.3. Basic Use	620
22.4. Spreadsheet Configuration	621
22.5. Cell Content and Formatting	622
22.6. Context Menus	624
22.7. Tables Within Spreadsheets	625

This chapter describes how to use Vaadin Spreadsheet, a Vaadin add-on component for displaying and editing Excel spreadsheets within any Vaadin application.

22.1. Overview

Spreadsheet applications have been the sonic screwdriver of business computation and data collection for decades. In recent years, spreadsheet web services have become popular with cloud-based services that offer better collaboration, require no installation, and some are even free to use. However, both desktop and third-party cloud-based services are difficult to integrate well with web applications. Being a Vaadin UI component, Vaadin Spreadsheet allows complete integration with Vaadin applications and further with the overall system. The ability to work on Excel spreadsheets allows desktop interoperability and integration with document management.

By eliminating the dependency on third-party cloud-based services, Vaadin Spreadsheet also gives control over the privacy of documents. Growing security concerns over cloud-based information storage have increased privacy requirements, with lowering trust in global third-party providers. Vaadin applications can run on private application servers, and also in a cloud if necessary, allowing you to prioritize between privacy and local and global availability.

Figure 22.1. Demo for Vaadin Spreadsheet

	A	B	C	D
1	Loan calculator			
2				
3				Edit these!
4			Loan amount	\$10 000,00
5			Interest rate	5,00%
6			Period (years)	2
7			Start date	1/1/15
8				
9				
10			Monthly payment	\$438,71
11			Number of payments	24
12			Total interest	\$529,13
13			Last payment	12/1/16
14			Total price payed	\$10 529,13
15				

Vaadin Spreadsheet is a UI component that you use much like any other component. It has full size by default, to use all the available space in the containing layout. You can directly modify the cell data in the active worksheet by entering textual and numerical values, as well as using Excel formulas for spreadsheet calculations.

```
Spreadsheet sheet = new Spreadsheet();
sheet.setWidth("400px"); // Full size by default
sheet.setHeight("250px");
```

```
// Put customary greeting in a cell
sheet.createCell(0, 0, "Hello, world");
```

```
// Have some numerical data
sheet.createCell(1, 0, 6);
sheet.createCell(1, 1, 7);
```

```
// Perform a spreadsheet calculation
```

```

sheet.createCell(1, 2, ""); // Set a dummy value
sheet.getCell(1, 2).setCellFormula("A2*B2");

// Resize a column to fit the cell data
sheet.autofitColumn(0);

layout.addComponent(sheet);
layout.setSizeFull(); // Typically

```

The result is shown in Figure 22.2, “Simple Spreadsheet”.

Figure 22.2. Simple Spreadsheet

	A	B	C	D	E
1	Hello, world				
2	6	7	42		
3					
4					
5					
6					
7					
8					

Cell values and formulas can be set, read, and styled through the server-side API, so you can easily implement custom editing features through menus or toolbars.

Full integration with a Vaadin application is reached through the server-side access to the spreadsheet data as well as visual styling. Changes in cell values can be handled in the Vaadin application and you can use almost any Vaadin components within a spreadsheet. Field components can be bound to cell data.

Vaadin Spreadsheet uses Apache POI to work on Microsoft Excel documents. You can access the Apache POI data model to perform low-level tasks, although you should note that if you make modifications to the data model, you have the responsibility to notify the spreadsheet to update itself.

22.1.1. Features

The basic features of Vaadin Spreadsheet are as follows:

- Support for touch devices

- Excel XLSX files, limited support for XLS files
- Support for Excel formulas
- Excel-like editing with keyboard support
- Lazy loading of cell data from server to browser in large spreadsheets
- Protected cells and sheets

The following features support integration with Vaadin Framework and add-ons:

- Handle changes in cell data
- Vaadin components in spreadsheet cells
- Support for Vaadin declarative format
- Vaadin TestBench element API for UI testing

22.1.2. Spreadsheet Demo

The Vaadin Spreadsheet Demo showcases the most important Spreadsheet features. You can try it out at <http://demo.vaadin.com/spreadsheet>.

22.1.3. Requirements

- Vaadin 7.4 or later
- Same browser requirements as Vaadin Framework, except Internet Explorer 9 or later is required

22.1.4. Limitations

Vaadin Spreadsheet 1.0 has the following limitations:

- No provided toolbars, menus, or other controls for formatting cells
- Vaadin Charts component can not be used inside Spreadsheet
- Limited support for the older XSL format
- Limitations of Apache POI

22.1.5. Licensing

Vaadin Spreadsheet is a commercial product licensed under the CVAL license (Commercial Vaadin Add-On License). Development licenses need to be purchased for each developer working with Vaadin Spreadsheet, regardless of whether the resulting applications using it are deployed publicly or privately in an intranet.

A Vaadin Pro Tools subscription includes a subscription license for Vaadin Spreadsheet. Perpetual licenses can be purchased from the Vaadin Spreadsheet product page, where you can also find the licensing details. For evaluation purposes, a free trial key allows using Vaadin Spreadsheet for a 30-day evaluation period.

22.2. Installing Vaadin Spreadsheet

You can download and install Spreadsheet from Vaadin Directory at vaadin.com/addon/vaadin-spreadsheet as an installation package, or get it with Maven or Ivy. You can purchase the required CVAL license or get a free trial key from Vaadin Directory or subscribe to the Pro Tools at vaadin.com/pro.

Add-on installation is described in detail in Chapter 18, *Using Vaadin Add-ons*. The add-on includes both a widget set and a theme, so you need to compile the widget sets and themes in your project.

22.2.1. Installing License Key

You need to install a license key before compiling the widget set. The license key is checked during widget set compilation, so you do not need it when deploying the application.

You can purchase Vaadin Spreadsheet or obtain a free trial key from the Vaadin Spreadsheet download page in Vaadin Directory. You need to register in Vaadin Directory to obtain the key.

See Section 18.5, “Installing Commercial Vaadin Add-on Licence” for detailed instructions on obtaining and installing the license key.

22.2.2. Compiling Widget Set

Compile the widget set as instructed in Section 18.4.2, “Compiling the Project Widget Set”. Widget set compilation should automatically update your project widget set to include the Spreadsheet widget set:

```
<inherits name="com.vaadin.addon.spreadsheet.Widgetset"/>
```

If you have set the widget set to be manually edited, you need to add the element yourself.

22.2.3. Compiling Theme

Compile the theme as instructed in Section 9.4, “Compiling Sass Themes”. If you compile in Eclipse or with Maven, the `addons.scss` file in your theme should be automatically updated to include the Spreadsheet theme:

```
@import "../../../VAADIN/addons/spreadsheet/spreadsheet.scss";
@mixin addons {
  @include spreadsheet;
}
```

If you are compiling the theme otherwise, or the theme addons are not automatically updated for some reason, you need to add the statements yourself.

22.2.4. Importing the Demo

The Demo, illustrated in Figure 22.1, “Demo for Vaadin Spreadsheet” in the overview, showcases most of the functionality in Vaadin Spreadsheet. You can try out the demo online at demo.vaadin.com/spreadsheet.

You can browse the sources on-line or, more conveniently, import the project in Eclipse (or other IDE). As the project is Maven-based, Eclipse users need to install the m2e plugin to be

able to import Maven projects, as well as EGit to be able to import Git repositories. Once they are installed, you should be able to import demo as follows.

1. Select **File** → **Import**
2. Select **Maven** → **Check out Maven Project from SCM**, and click **Next**.
3. You may need to install the EGit SCM connector if you have not done so previously. If Git is not available in the SCM list, click **m2e marketplace**, select the EGit connector, and click **Finish**. You need to restart Eclipse and redo the earlier steps above.

Instead of using m2e EGit connector, you can also check out the project with another Git tool and then import it in Eclipse as a Maven project.

4. In **SCM URL**, select **git** and enter the repository URL `https://github.com/vaadin/spread-sheet-demo`.
5. Click **Finish**.
6. Compile the widget set either by clicking **Compile Widgetset** in the Eclipse toolbar or by running the `vaadin:compile` goal with Maven.
7. Deploy the application to a server. See Section 3.4.5, “Setting Up and Starting the Web Server” for instructions for deploying in Eclipse.
8. Open the URL `http://localhost:8080/spreadsheet` with a browser.

22.3. Basic Use

Spreadsheet is a Vaadin component, which you can use as you would any component. You can create it, or load from an Excel file, create cells and new sheets, define formulas, and so forth. In the following, we go through these basic steps.

22.3.1. Creating a Spreadsheet

The default constructor for **Spreadsheet** creates an empty spreadsheet with a default sheet. The spreadsheet component has full size by default, so the containing layout must have defined size in both dimensions; a spreadsheet may not have undefined size.

In the following example, we create an empty spreadsheet with a fixed size and add it to a layout.

```
Spreadsheet sheet = new Spreadsheet();
sheet.setWidth("400px");
sheet.setHeight("300px");

layout.addComponent(sheet);
```

An empty spreadsheet automatically gets an initial worksheet with some default size and settings.

Loading an Excel Spreadsheet

You can load an Excel file from the local filesystem with a **File** reference or from memory or other source with an **InputStream**.

Working with an Apache POI Workbook

If you have an Apache POI workbook created otherwise, you can wrap it to **Spreadsheet** with the respective constructor.

You can access the underlying workbook with `getWorkbook()`. However, if you make modifications to the workbook, you must allow the spreadsheet update itself by calling appropriate update methods for the modified element or elements.

22.3.2. Working with Sheets

A "spreadsheet" in reality works on a *workbook*, which contains one or more *worksheets*. You can create new sheets and delete existing ones with `createNewSheet()` and `deleteSheet()`, respectively.

```
// Recreate the initial sheet to configure it
Spreadsheet sheet = new Spreadsheet();
sheet.createNewSheet("New Sheet", 10, 5);
sheet.deleteSheet(0);
```

When a sheet is deleted, the index of the sheets with a higher index is decremented by one. When the active worksheet is deleted, the next one by index is set as active, or if there are none, the previous one.

All operations on the spreadsheet content are done through the currently active worksheet. You can set an existing sheet as active with `setActiveSheetIndex()`.

22.4. Spreadsheet Configuration

Spreadsheet has a number of configuration parameters and sections that affect the overall appearance and function of the entire spreadsheet. The most important ones are mentioned in the following and further configuration is provided through the Apache POI API.

22.4.1. Spreadsheet Elements

The **Spreadsheet** object provides the following configuration of various UI elements:

Grid lines

Cells are by default separated by grid lines. You can control their visibility with `setGridlinesVisible()`.

Column and row headings

Headings for rows and columns display the row and column indexes, and allow selecting and resizing the rows and columns. You can control their visibility with `setRowColHeadingsVisible()`.

Top bar

The top bar displays the address of the currently selected cell and an editor for cell content. You can control its visibility with `setFunctionBarVisible()`.

Bottom bar

The bottom bar displays the address of the currently selected cell and an editor for cell content. You can control its visibility with `setSheetSelectionBarVisible()`.

Report mode

In the report mode, both the top and bottom bars are hidden. You can enable the mode with `setReportStyle()`.

22.4.2. Frozen Row and Column Panes

You can define panes of rows and columns that are frozen in respect to scrolling. You can create the pane for the current worksheet with `createFreezePane()`, which takes the number of frozen rows and columns as parameters.

```
sheet.createFreezePane(1, 2);
```

The result is shown in Figure 22.3, “Frozen Row and Column Panes”.

Figure 22.3. Frozen Row and Column Panes

	A	B	C	D	E
1	First Name	Last Name	Born		
2	Nicolaus	Copernicus	1999-03-19		
3	Galileo	Galilei	1964-03-15		
4					
5					
6					

22.5. Cell Content and Formatting

In the following, we go through various user interface features in the table cells.

22.5.1. Cell Formatting

Formatting cell values can be accomplished by using cell styles. A cell style must be created in the workbook by using `createCellStyle()`. Cell data format is set for the style with `setDataFormat()`.

```
// Define a cell style for dates
CellStyle dateStyle = sheet.getWorkbook().createCellStyle();
DataFormat format = sheet.getWorkbook().createDataFormat();
dateStyle.setDataFormat(format.getFormat("yyyy-mm-dd"));

// Add some data rows
sheet.createCell(1, 0, "Nicolaus");
sheet.createCell(1, 1, "Copernicus");
sheet.createCell(1, 2,
    new GregorianCalendar(1999, 2, 19).getTime());
```

```
// Style the date cell
sheet.getCell(1,2).setCellStyle(dateStyle);
```

22.5.2. Cell Font Style

Cells can be styled by different fonts. A font definition not only includes a particular typeface, but also weight (bold or normal), emphasis, underlining, and other such font attributes.

A font definition is managed by **Font** class in the Apache POI API. A new font can be created with `createFont()` in the workbook.

For example, in the following we make a header row in a spreadsheet bold:

```
// Create some column captions in the first row
sheet.createCell(0, 0, "First Name");
sheet.createCell(0, 1, "Last Name");
sheet.createCell(0, 2, "Born");

// Create a bold font
Font bold = sheet.getWorkbook().createFont();
bold.setBold(true);

// Set the cells in the first row as bold
for (int col=0; col <= 2; col++)
    sheet.getCell(0, col).getCellStyle().setFont(bold);
```

22.5.3. Cell Comments

Cells may have comments that are indicated by ticks in the corner of the cells, and the comment is shown when mouse is hovered on the cells. The **SpreadsheetDefaultActionHandler** described in Section 22.6.1, “Default Context Menu” enables adding comments from the context menu.

A new comment can be added through the POI API of a cell, with `addComment()`. For a detailed example for managing cell comments, we refer to the **InsertDeleteCellCommentAction** and **EditCellCommentAction** classes employed by the default action handler.

22.5.4. Merging Cells

You can merge spreadsheet cells with any variant of the `addMergedRegion()` method in **Spreadsheet**.

The **SpreadsheetDefaultActionHandler** described in Section 22.6.1, “Default Context Menu” enables merging selected cells from the context menu.

Previously merged cells can be unmerged with `removeMergedRegion()`. The method takes as its parameter a region index. You can search for a particular region through the POI **Sheet** API for the active sheet, which you can obtain with `getActiveSheet()`. The `getMergedRegion()` returns a merged region by index and you can iterate through them by knowing the number of regions, which you can find out with `getNumMergedRegions()`.

22.5.5. Components in Cells

You can have Vaadin components in spreadsheet cells and bind field components to the cell data. The components can be shown all the time or work as editors that appear when a cell is activated for editing.

Components in a spreadsheet must be generated by a `SpreadsheetComponentFactory`, which you need to implement.

22.5.6. Hyperlinks

Hyperlinks in cells can point to other worksheets in the current workbook, or to external URLs. Links must be added through the POI API. Spreadsheet provides default handling for hyperlink clicks, which can be overridden with a custom `HyperlinkCellClickHandler`, which you assign with `setHyperlinkCellClickHandler()`.

22.5.7. Popup Buttons in Cells

You can add a popup button in a cell, clicking which opens a drop-down popup overlay, which can contain any Vaadin components. You can add a popup button with any of the `setPopup()` methods for different cell addressing forms. A popup button is a **PopupButton** object, which is a regular Vaadin component container, so you can add any components to it by `addComponent()`.

You can create popup buttons for a row of cells in a cell range by defining a *table*, as described in Section 22.7, “Tables Within Spreadsheets”.

22.6. Context Menus

Grid has a context menu, which can be used for various editing functions. The context menu uses the standard Vaadin action handler mechanism.

22.6.1. Default Context Menu

The **SpreadsheetDefaultActionHandler** provides a ready set of common spreadsheet editing operations.

- Add, delete, hide, and show rows and columns
- Insert, edit, and delete cell comments
- Merge and unmerge cells
- Add filter table to selected range

The default context menu is not enabled by default; to enable it, you need to create and set it as the action handler explicitly.

```
sheet.addActionHandler(new SpreadsheetDefaultActionHandler());
```

22.6.2. Custom Context Menus

You can implement custom context menus either by implementing an `ActionHandler` or extending the **`SpreadsheetDefaultActionHandler`**. The source code of the default handler should serve as a good example of implementing context menu items for different purposes.

22.7. Tables Within Spreadsheets

A cell range in a worksheet can be configured as a *table*, which adds popup menu buttons in the header row of the range. The popup menus contain Vaadin components, which you can use to implement various functionalities in the table, such as sorting or filtering. Vaadin Spreadsheet does not include any implementations of such features, merely the UI elements to enable them.

Such a table is defined by a **`SpreadsheetTable`** or a **`SpreadsheetFilterTable`** added to the spreadsheet.

22.7.1. Creating a Table

For example, let us assume that you have a sheet with some data in a cell region. The first row of the region should contain column captions for the region.

```
Spreadsheet sheet = new Spreadsheet();

// Have a header row in a region of the sheet
sheet.createCell(1, 1, "First Name");
sheet.createCell(1, 2, "Last Name");
sheet.createCell(1, 3, "Born");
sheet.createCell(1, 4, "Died");

... insert the data ...

// Define the range
CellRangeAddress range =
    new CellRangeAddress(1, 7, 1, 4);
```

A table is created for a cell range and then registered in the spreadsheet with `registerTable()`; you can unregister it with `unregisterTable()`. The first row of the cell range should contain the table captions.

```
// Create a table in the range
SpreadsheetTable table = new SpreadsheetTable(sheet, range);
sheet.registerTable(table);

// Enable hiding each column in the popup
for (int col = range.getFirstColumn();
     col <= range.getLastColumn(); col++) {
    final int c = col; // For access in the lambda
    table.getPopupButton(col).addComponent(
        new Button("Hide Column", e -> { // Java 8
            sheet.setColumnHidden(c, true);
            table.getPopupButton(c).closePopup();
        }));
}
```

The popup buttons are typically used for performing operations on columns, such as sorting.

22.7.2. Filtering With a Table

SpreadsheetFilterTable is a spreadsheet table that allows filtering the rows in the table in the popup menus. The menu is filled with checkboxes for each unique value in the column. Deselecting the items causes hiding the respective rows in the spreadsheet.

Chapter 23

Vaadin TestBench

23.1. Overview	627
23.2. Quick Start	632
23.3. Installing Vaadin TestBench	635
23.4. Developing JUnit Tests	640
23.5. Creating a Test Case	644
23.6. Querying Elements	647
23.7. Element Selectors	649
23.8. Special Testing Topics	650
23.9. Creating Maintainable Tests	654
23.10. Taking and Comparing Screenshots	658
23.11. Running Tests	662
23.12. Running Tests in a Distributed Environment	665
23.13. Parallel Execution of Tests	669
23.14. Headless Testing	671
23.15. Behaviour-Driven Development	672
23.16. Integration Testing with Maven	673
23.17. Known Issues	676

This chapter describes the installation and use of the Vaadin TestBench.

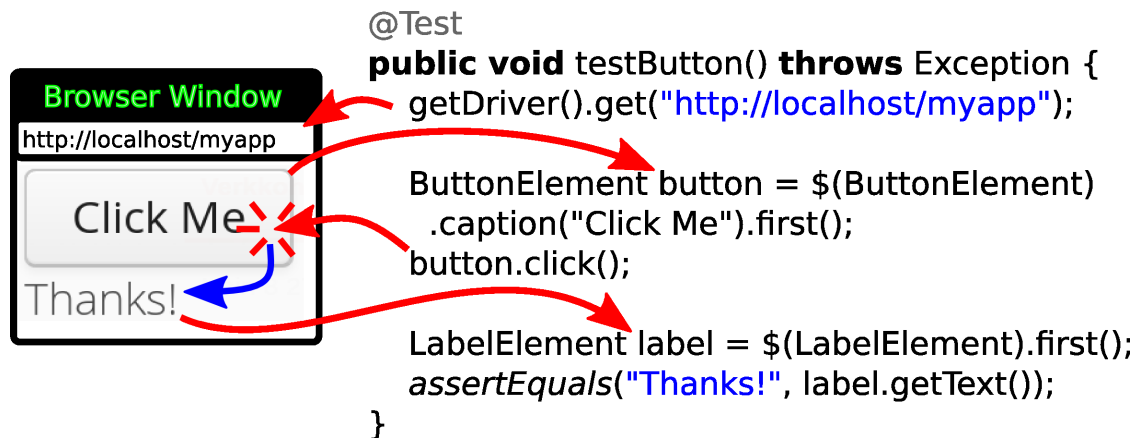
23.1. Overview

Testing is one of the cornerstones of modern software development. Extending throughout the development process, testing is the thread that binds the product to the requirements. In agile and other iterative development processes, with ever shorter release cycles and continuous

integration, the automation of integration, regression, endurance, and acceptance testing is paramount. Further, UI automation may be needed for integration purposes, such as for assistive technologies. The special nature of web applications creates many unique requirements for both testing and UI automation.

Vaadin TestBench allows controlling the browser from Java code, as illustrated in Figure 23.1, “Controlling the Browser with Testbench”. It can open a new browser window to start the application, interact with the UI components, for example, by clicking them, and then get the HTML element values.

Figure 23.1. Controlling the Browser with Testbench



Before going further into feature details, you may want to try out Vaadin TestBench yourself. You just need to create a new Vaadin project either with the Eclipse plugin or the Maven archetype. Both create a simple application stub that includes TestBench test cases for testing the UI. You also need to install an evaluation license. For instructions, jump to Section 23.2, “Quick Start” and, after trying it out, come back.

23.1.1. Vaadin TestBench in Software Development

Vaadin TestBench can work as the centerpiece of the software development process, for testing the application at all modular levels and in all the phases of the development cycle:

- Automated acceptance tests
- Unit tests
- End-to-end integration tests
- Regression tests

Let us look at each of these topics separately.

Any methodological software development, agile or not, is preceded by specification of requirements, which define what the software should actually do. *Acceptance tests* ensure that the product conforms to the requirements. In agile development, their automation allows continuous tracking of progress towards iteration goals, as well as detecting regressions. The importance of requirements is emphasized in *test-driven development* (TDD), where tests are written before actual code. In Section 23.15, “Behaviour-Driven Development”, we show how to use Vaadin

TestBench for *behaviour-driven development* (BDD), a form of TDD that concentrates on the formal behavioural specification of requirements.

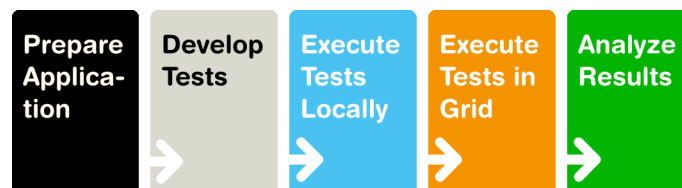
Unit testing is applied to the smallest scale of software components; in Vaadin applications these are typically individual UI components or view classes. You may also want to generate many different kinds of inputs for the application and check that they produce the desired outputs. For complex composites, such as views, you can use the Page Object Pattern described in Section 23.9.2, “The Page Object Pattern”. The pattern simplifies and modularizes testing by separating low-level details from the more abstract UI logic. In addition to serving the purpose of unit tests, it creates an abstraction layer for higher-level tests, such as acceptance and end-to-end tests.

Integration tests ensure that software units work together at different levels of modularization. At the broadest level, *end-to-end tests* extend through the entire application lifecycle from start to finish, going through many or all user stories. The aim is not just to verify the functional requirements for user interaction, but also that data integrity is maintained. For example, in a messaging application, a user would log in, both send and receive messages, and finally log out. Such test workflows could include configuration, registration, interaction between users, administrative tasks, deletion of user accounts, and so forth.

In *regression testing*, you want to ensure that only intended changes occur in the behaviour after modifying the code. There are two lines of defence against such regressions. The primary source of regression tests are the acceptance, unit, and integration tests that validate that the displayed values in the UI’s HTML representation are logically correct. Yet, they are not sufficient for detecting visual regressions, for example, because of invalid UI rendering or theme problems. Comparing screenshots to reference images forms a more sensitive layer to detect regressions, at the expense of losing robustness for changes in layout and themeing. The costs of the tradeoff can be minimized by careful application of screenshot comparison only at critical points and by making the analysis of such regressions as easy as possible. As described in Section 23.10, “Taking and Comparing Screenshots”, Vaadin TestBench automatically highlights differences in screenshots and allows masking irrelevant areas from image comparison.

You can develop such tests along with your application code, for example with JUnit, which is a widely used Java unit testing framework. You can run the tests as many times as you want in your workstation or in a distributed grid setup.

Figure 23.2. TestBench Workflow



23.1.2. Features

The main features of Vaadin TestBench are:

- Control a browser from Java
- Generate component selectors in debug window
- Validate UI state by assertions and screen capture comparison

- Screen capture comparison with difference highlighting
- Distributed test grid for running tests
- Integration with unit testing
- Test with browsers on mobile devices

Execution of tests can be distributed over a grid of test nodes, which speeds up testing. The grid nodes can run different operating systems and have different browsers installed. In a minimal setup, such as for developing the tests, you can use Vaadin TestBench on just a single computer.

23.1.3. Based on Selenium

Vaadin TestBench is based on the Selenium web browser automation library, especially Selenium WebDriver, which allows you to control browsers straight from Java code.

Selenium is augmented with Vaadin-specific extensions, such as:

- Proper handling of Ajax-based communications of Vaadin
- A high-level, statically typed element query API for Vaadin components
- Performance testing of Vaadin applications
- Screen capture comparison
- Finding HTML elements by a Vaadin selector

23.1.4. TestBench Components

The TestBench library includes WebDriver, which provides API to control a browser like a user would. This API can be used to build tests, for example, with JUnit. It also includes the grid hub and node servers, which you can use to run tests in a grid configuration.

Vaadin TestBench Library provides the central control logic for:

- Executing tests with the WebDriver
- Additional support for testing Vaadin-based applications
- Comparing screen captures with reference images
- Distributed testing with grid node and hub services

23.1.5. Requirements

Requirements for developing and running tests are:

- Java JDK 1.6 or newer
- Browsers installed on test nodes as supported by Selenium WebDriver
 - Google Chrome

- Internet Explorer
- Mozilla Firefox (ESR version recommended)
- Opera
- Mobile browsers: Android, iPhone
- A build system, such as Ant or Maven, to automate execution of tests during build process (recommended)

Note that running tests on an Extended Support Release (ESR) version of Firefox is recommended because of the frequent release cycle of Firefox, which often cause tests to fail. Download an ESR release of Firefox from <http://www.mozilla.org/en-US/firefox/organizations/all.html>. Install it alongside your normal Firefox install (do not overwrite).

For Mac OS X, note the issue mentioned in Section 23.17.1, “Running Firefox Tests on Mac OS X”.

23.1.6. Continuous Integration Compatibility

Continuous integration means automatic compilation and testing of applications frequently, typically at least daily, but ideally every time when code changes are committed to the source repository. This practice allows catching integration problems early and finding the changes that first caused them to occur.

You can make unit tests with Vaadin TestBench just like you would do any other Java unit tests, so they work seamlessly with continuous integration systems. Vaadin TestBench is tested to work with at least TeamCity and Hudson/Jenkins build management and continuous integration servers, which all have special support for the JUnit unit testing framework.

Figure 23.3. Continuous Integration Workflow

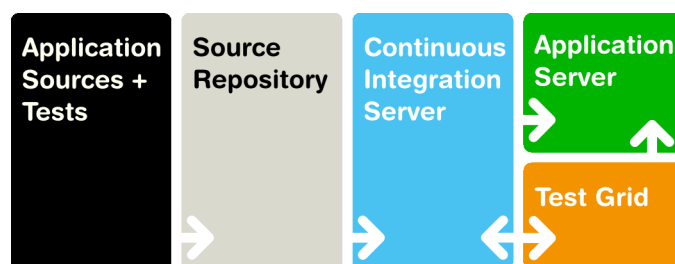


Figure 23.3, “Continuous Integration Workflow” illustrates a typical development setup. Both changes to application and test sources are checked in into a source repository, from where the CIS server checks them out, compiles, and deploys the web application to a server. Then, it runs the tests and collects the results.

23.1.7. Licensing and Trial Period

You can download Vaadin TestBench from Vaadin Directory and try it out for a free 30-day trial period, after which you are required to acquire the needed licenses. You can purchase licenses from the Directory. A license for Vaadin TestBench is also included in the Vaadin Pro Account subscription.

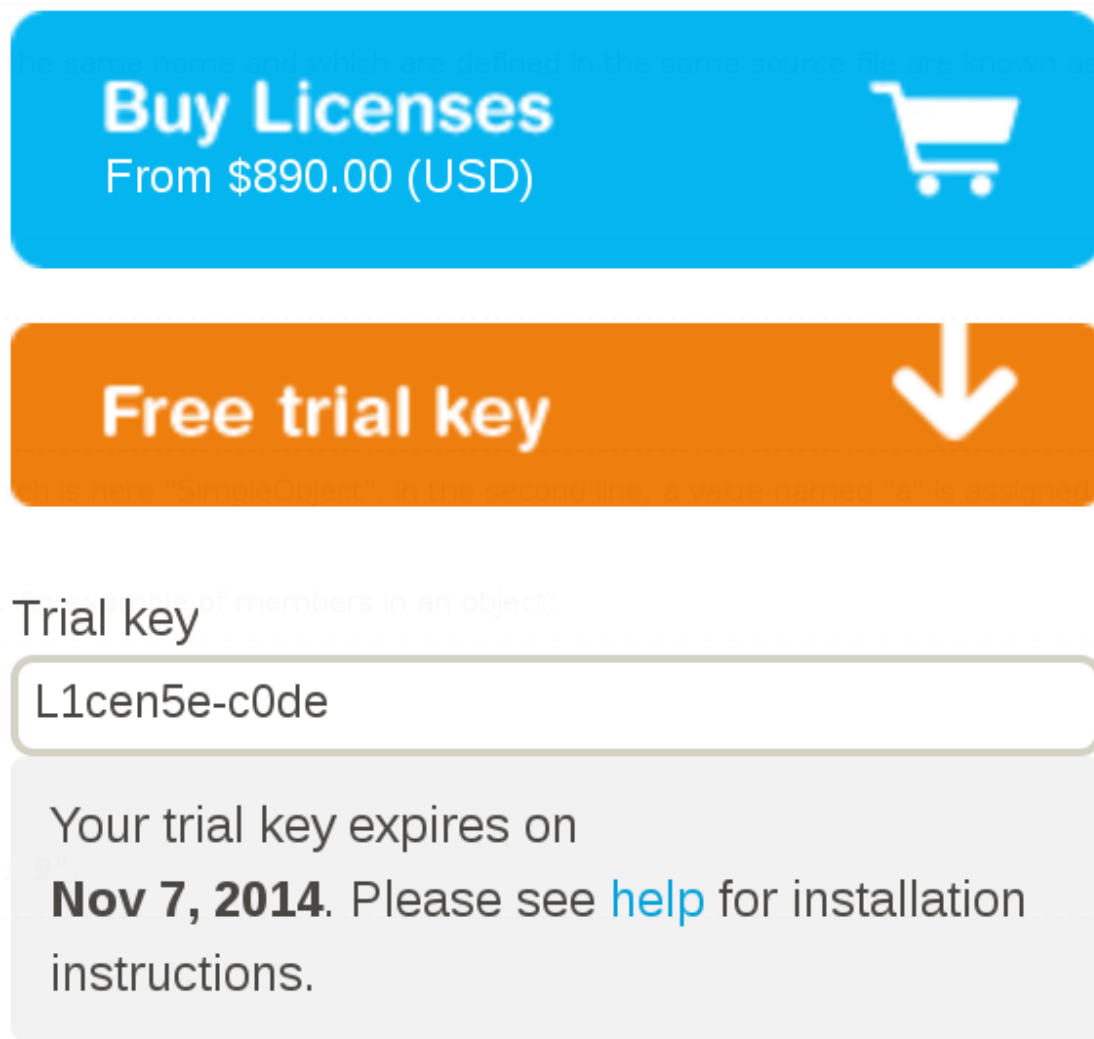
23.2. Quick Start

In the following, we give instructions for getting Vaadin TestBench running in minutes. You can create either a new Eclipse project or a Maven project. Both project types require installing a license key, so we cover that first.

23.2.1. Installing License Key

Before running tests, you need to install a license key. You can purchase Vaadin TestBench or obtain a free trial key from the Vaadin TestBench download page in Vaadin Directory. You need to register in Vaadin Directory to obtain the key.

Figure 23.4. Obtaining License Key from Vaadin Directory



To install the license key on a development workstation, you can copy and paste it verbatim to a `.vaadin.testbench.developer.license` file in your home directory. For example, in Linux and OS X:

```
$ echo "L1cen5e-c0de" > ~/.vaadin.testbench.developer.license
```

You can also pass the key as a system property to the Java application running the tests, usually with a `-D` option on the command-line:

```
$ java -Dvaadin.testbench.developer.license=L1cen5e-c0de ...
```

How you actually pass the parameter to your test runner depends on the actual test execution environment. Below are listed a few typical environments:

Eclipse IDE

To install the license key for all projects, select **Window** → **Preferences** and navigate to the **Java** → **Installed JREs** section. Select the JRE version that you use for the application and click **Edit**. In the **Default VM arguments**, give the `-D` expression as shown above.

For a single project, create a new JUnit launch configuration in **Run** → **Run configurations**. Select **JUnit** and click **New launch configuration**. If you have already ran JUnit in the project, the launch configuration already exists. Select JUnit 4 if not selected automatically. Go to **Arguments** tab and give the `-D` expression in the **VM arguments** field. Click **Run** to run the tests immediately or **Close** to just save the settings.

Apache Ant

If running tests with the `<junit>` task in Apache Ant, as described in Section 23.11.1, “Running Tests with Ant”, you can pass the key as follows:

```
<sysproperty key="vaadin.testbench.developer.license"
             value="L1cen5e-c0de"/>
```

However, you should never store license keys in a source repository, so if the Ant script is stored in a source repository, you should pass the license key to Ant as a property that you then use in the script for the value argument of the `<sysproperty>` as follows:

```
<sysproperty key="vaadin.testbench.developer.license"
             value="${vaadin.testbench.developer.license}"/>
```

When invoking Ant from the command-line, you can pass the property with a `-D` parameter to Ant.

Apache Maven

If running tests with Apache Maven, you can pass the license key with a `-D` parameter to Maven:

```
$ mvn -Dvaadin.testbench.developer.license=L1cen5e-c0de verify
```

TeamCity

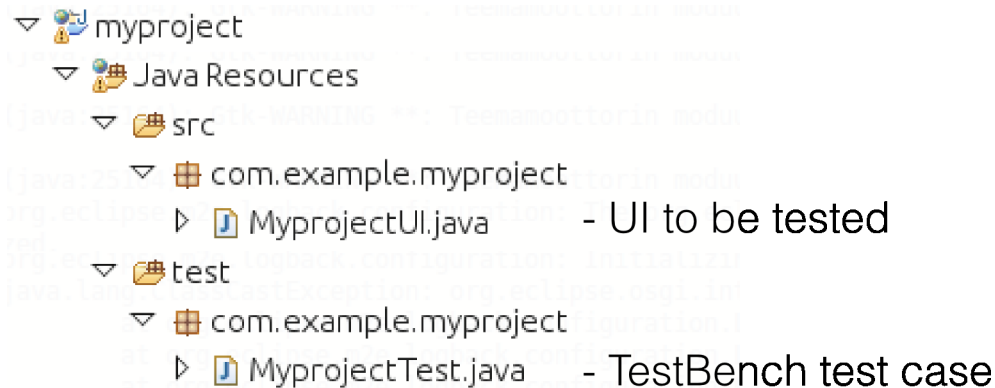
In TeamCity, you can pass the license key to build runners as a system property in the build configuration. However, this only passes it to a runner. As described above, Maven passes the parameter as is to JUnit, but Ant does not do so implicitly, so you need to forward it explicitly as described earlier.

23.2.2. Quick Start with Eclipse

Once you have installed the Vaadin Plugin for Eclipse, you can use it to create a new Vaadin 7 project with the TestBench test enabled, as described in Section 3.4.1, “Creating a Maven Project”. In the project settings, you need to have the **Create TestBench test** setting enabled.

The test case stub is created under `test` source folder, so that it will not be deployed with the application. The project and source folders are illustrated in Figure 23.5, “Eclipse Project with a Test Case”.

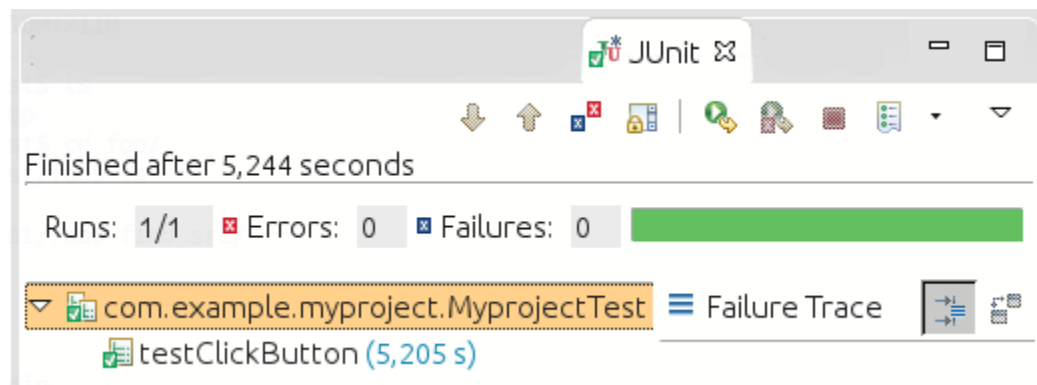
Figure 23.5. Eclipse Project with a Test Case



You can observe that the UI and the test case are much like in the illustration in Figure 23.1, “Controlling the Browser with Testbench”. The resulting test case stub is described in detail in Section 23.4.1, “Basic Test Case Structure”.

To run the test, open the `MyprojectTest.java` file in the editor and press `ShiftAltXT`. The browser should open with the application UI and TestBench run the tests. The results are displayed in the **JUnit** view in Eclipse, as shown in Figure 23.6, “JUnit Test Results in Eclipse”.

Figure 23.6. JUnit Test Results in Eclipse



23.2.3. Quick Start with Maven

With Maven, you need to create a new Vaadin project with the `vaadin-archetype-application` archetype, as described in Section 3.5, “Creating a Project with Maven”.

The `src` folder under the project contains both the sources for the application and the tests. The test case stub in the `src/test` folder is described in detail in Section 23.4.1, “Basic Test Case Structure”.

The license needs to be installed or given as parameter for the following command, as mentioned earlier. Build the project with the `verify` or a later phase in the build lifecycle. For example, from the command-line:


```
$ mvn verify
```

This will execute all required lifecycle phases, including compilation and packaging the application, launch Jetty web server to host the application, and run the TestBench tests. The TestBench tests are run in the `integration-test` phase, but the web server is stopped in the `post-integration-test` phase, hence we call the `verify` phase. The results are reported on the console. A Maven GUI, such as the one in Eclipse, will provide more visual results.

For more details on the Maven POM configuration for Vaadin TestBench, see Section 23.16, “Integration Testing with Maven”.

23.3. Installing Vaadin TestBench

As with most Vaadin add-ons, you can install Vaadin TestBench as a Maven or Ivy dependency in your project, or from an installation package. The installation package contains some extra material, such as documentation, as well as the standalone library, which you use for testing in a grid.

The component element classes are Vaadin version specific and they are packaged in a `vaadin-testbench-api` library JAR, separately from the `vaadin-testbench-core` runtime library, which is needed for executing the tests.

Additionally, you may need to install drivers for the browsers you are using.

23.3.1. Test Development Setup

In a typical test development setup, you develop tests in a Java project and run them on the development workstation. You can run the same tests in a dedicated test server, such as a continuous integration system.

In a test development setup, you do not need a grid hub or nodes. However, if you develop tests for a grid, you can run the tests, the grid hub, and one node all in your development workstation. A distributed setup is described later.

Maven Dependency

The Maven dependency for Vaadin TestBench goes by the TestBench API, which matches the Vaadin Framework version that you use. The dependency pulls the actual TestBench libraries.

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-testbench-api</artifactId>
  <version>7.x.x</version>
  <scope>test</scope>
</dependency>
```



TestBench 4.0.2 has a problem with Firefox

To make it work with latest Firefox versions, you need to use a newer version of Selenium. For example:

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>2.45.0</version>
```

```
<scope>test</scope>
</dependency>
```

You also need to define the Vaadin add-ons repository if not already defined:

```
<repository>
  <id>vaadin-addons</id>
  <url>http://maven.vaadin.com/vaadin-addons</url>
</repository>
```

The `vaadin-archetype-application` archetype, as mentioned in Section 23.2.3, “Quick Start with Maven”, includes the declarations.

Ivy Dependency

The Ivy dependency, to be defined in `ivy.xml`, would be as follows:

```
<dependency org="com.vaadin" name="vaadin-testbench-api"
  rev="latest.release" conf="nodeploy->default"/>
```

The optional `nodeploy->default` configuration mapping requires a `nodeploy` configuration in the Ivy module; it is automatically created for new Vaadin projects.

A new Vaadin project created with the Vaadin Plugin for Eclipse, as described in Section 23.2.2, “Quick Start with Eclipse”, includes the dependency.

Code Organization

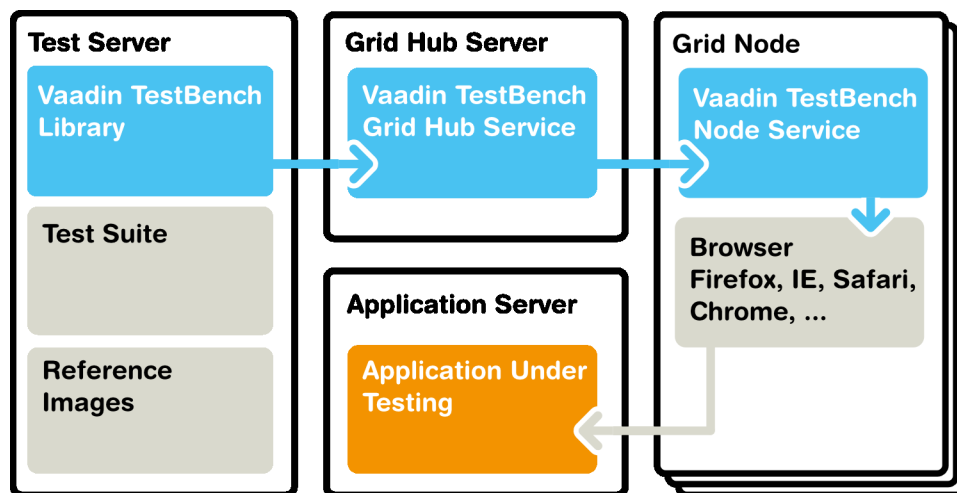
We generally recommend developing tests in a project or module separate from the web application to be tested to avoid library problems. If the tests are part of the same project, you should at least arrange the source code and dependencies so that the test classes, the TestBench library, and their dependencies would not be deployed unnecessarily with the web application.

23.3.2. A Distributed Testing Environment

Vaadin TestBench supports distributed execution of tests in a grid. A test grid consists of the following categories of hosts:

- One or more test servers executing the tests
- A grid hub
- Grid nodes

The components of a grid setup are illustrated in Figure 23.7, “Vaadin TestBench Grid Setup”.

Figure 23.7. Vaadin TestBench Grid Setup

The grid hub is a service that handles communication between the JUnit test runner and the nodes. The nodes are services that perform the actual execution of test commands in the browser.

The hub requires very little resources, so you would typically run it either in the test server or on one of the nodes. You can run the tests, the hub, and one node all in one host, but in a fully distributed setup, you install the Vaadin TestBench components on separate hosts.

Controlling browsers over a distributed setup requires using a remote WebDriver. Grid development and use of the hub and nodes is described in Section 23.12, “Running Tests in a Distributed Environment”.

23.3.3. Installation Package Contents

The installation package contains the following:

`documentation`

The documentation folder contains release notes, a PDF excerpt of this chapter of Book of Vaadin, and the license.

`maven`

The Maven folder contains the Vaadin TestBench library JARs (you can use them in non-Maven projects as well). The folder contains a POM file, so that you can install it in your local Maven repository. Please follow the instructions in Section 23.11.2, “Running Tests with Maven”.

`vaadin-testbench-standalone-4.x.x.jar`

This is a standalone version of the Vaadin TestBench library that is mainly used for running the grid hub and node services, as described in Section 23.12, “Running Tests in a Distributed Environment”.

23.3.4. TestBench Demo

A TestBench demo is available at <https://github.com/vaadin/testbench-demo>. You can browse the sources at the website and clone the repository with a Git client; from command line with:

```
$ git clone https://github.com/vaadin/testbench-demo
```

The tests can be run from the command line by issuing the following command:

```
$ mvn verify
```

The source code for the application to be tested, a desktop calculator application, is given in the `src/main/java` subfolder.

The TestBench tests for the application are located under the `src/test/java` subfolder, in the `com/vaadin/testbenchexample` package subfolder. They are as follows:

`SimpleCalculatorITCase.java`

Demonstrates the basic use of `WebDriver`. Interacts with the buttons in the user interface by clicking them and checks the resulting value. Uses the `ElementQuery` API to access the elements.

`LoopingCalculatorITCase.java`

Otherwise as the simple example, but shows how to use looping to produce programmatic repetition to create a complex use case.

`ScreenshotITCase.java`

Shows how to compare screenshots, as described in Section 23.10.3, “Taking Screenshots for Comparison”. Some of the test cases include random input, so they require masked screenshot comparison to mask the random areas out.

The example is ignored by default with an `@Ignore` annotation, because the included images were taken with a specific browser on a specific platform, so if you use another environment, they will fail. If you enable the test, you will need to run the tests, copy the error images to the reference screenshot folder, and mask out the areas with the alpha channel. Please see the `example/Screenshot_Comparison_Tests.pdf` for details about how to enable the example and how to create the masked reference images.

`SelectorExamplesITCase.java`

This example shows how to find elements in different ways; by using the high-level `ElementQuery` API as well as low-level `By.xpath()` selectors.

`VerifyExecutionTimeITCase.java`

Shows how to time the execution of a test case and how to report it.

`AdvancedCommandsITCase.java`

Demonstrates how to test context menus (see Section 23.8.5, “Testing Context Menus”) and tooltips (see Section 23.8.2, “Testing Tooltips”). Also shows how to send keypresses to a component and how to read values of table cells.

`pageobjectexample/PageObjectExampleITCase.java`

Shows how to create maintainable tests using the *Page Object Pattern* that separates the low-level page structure from the business logic, as described in Section 23.9, “Creating Maintainable Tests”. The page object classes that handle low-level interaction with the application views are in the `pageobjects` subpackage.

`bdd/CalculatorSteps.java`, `bdd/SimpleCalculation.java`

Shows how to develop tests following the *behaviour-driven development* (BDD) model, by using the `JBehave` framework. `SimpleCalculation.java` defines a JUnit-based user story with one scenario, which is defined in `Calculator-`

`Steps.java`. The scenario reuses the page objects defined in the page object example (see above) for low-level application view access and control. The example is described in Section 23.15, “Behaviour-Driven Development”.

23.3.5. Installing Browser Drivers

Whether developing tests with the WebDriver in the workstation or running tests in a grid, using some browsers requires that a browser driver is installed.

1. Download the latest browser driver

- Internet Explorer (Windows only) - install `IEDriverServer.exe` from under the latest Selenium release:

<http://selenium-release.storage.googleapis.com/index.html>

- Chrome - install ChromeDriver (a part of the Chromium project) for your platform from under the latest release at:

<http://chromedriver.storage.googleapis.com/index.html>

2. Add the driver executable to user PATH. In a distributed testing environment, give it as a command-line parameter to the grid node service, as described in Section 23.12.4, “Starting a Grid Node”.

23.3.6. Test Node Configuration

If you are running the tests in a grid environment, you need to make some configuration to the test nodes to get more stable results.

Further configuration is provided in command-line parameters when starting the node services, as described in Section 23.12.4, “Starting a Grid Node”.

Operating system settings

Make any operating system settings that might interfere with the browser and how it is opened or closed. Typical problems include crash handler dialogs.

On Windows, disable error reporting in case a browser crashes as follows:

1. Open **Control Panel** → **System**
2. Select the **Advanced** tab
3. Select **Error reporting**
4. Check that **Disable error reporting** is selected
5. Check that **But notify me when critical errors occur** is not selected

Settings for Screenshots

The screenshot comparison feature requires that the user interface of the browser stays constant. The exact features that interfere with testing depend on the browser and the operating system.

In general:

- Disable blinking cursor
- Use identical operating system themeing on every host
- Turn off any software that may suddenly pop up a new window
- Turn off screen saver

If using Windows and Internet Explorer, you should give also the following setting:

- Turn on **Allow active content to run in files on My Computer** under **Security settings**

23.4. Developing JUnit Tests

JUnit is a popular unit testing framework for Java development. Most Java IDEs, build systems, and continuous integration systems provide support for JUnit. However, while we concentrate on the development of JUnit tests in this chapter, Vaadin TestBench and the WebDriver are in no way specific to JUnit and you can use any test execution framework, or just regular Java applications, to develop TestBench tests.

You may want to keep your test classes in a separate source tree in your application project, or in an altogether separate project, so that you do not have to include them in the web application WAR. Having them in the same project may be nicer for version control purposes.

23.4.1. Basic Test Case Structure

A JUnit test case is defined with annotations for methods in a test case class. With TestBench, the test case class should extend the **TestBenchTestCase** class, which provides the WebDriver and ElementQuery APIs.

```
public class MyTestcase extends TestBenchTestCase {
```

The basic JUnit annotations used in TestBench testing are the following:

@Rule

You can define certain TestBench parameters and other JUnit rules with the @Rule annotation.

For example, to enable taking screenshots on test failures, as described in Section 23.10.2, “Taking Screenshots on Failure”, you would define:

```
@Rule
public ScreenshotOnFailureRule screenshotOnFailureRule =
    new ScreenshotOnFailureRule(this, true);
```

Note that if you use this rule, you must *not* call `driver.quit()` in your @After method, as the method is executed before the screenshot is taken, but the driver must be open to take it.

@Before

The annotated method is executed before each test (annotated with @Test). Normally, you create and set the driver here.

```
@Before
public void setUp() throws Exception {
```

```
        setDriver(new FirefoxDriver());
    }
```

The driver class should be one of **FirefoxDriver**, **ChromeDriver**, **InternetExplorerDriver**, **SafariDriver**, or **PhantomJSDriver**. Please check **RemoteWebDriver** from API documentation for the current list of implementations. Notice that some of the drivers require installing a browser driver, as described in Section 23.3.5, “Installing Browser Drivers”.

The driver instance is stored in the `driver` property in the test case. While you can access the property directly by the member variable, you should set it only with the setter.

@Test

Annotates a test method. You normally first open the page and then execute commands and make assertions on the content.

```
@Test
public void testClickButton() throws Exception {
    getDriver().get("http://localhost:8080/myproject");

    // Click the button
    ButtonElement button = $(ButtonElement.class)
        .caption("Click Me").first();
    button.click();

    // Check that the label text is correct
    LabelElement label = $(LabelElement.class).first();
    assertEquals("Thanks!", label.getText());
}
```

Normally, you would define the URL with a variable that is common for all tests, and possibly concatenate it with a URI fragment to get to an application state.

@After

After each test is finished, you normally need to quit the driver to close the browser.

```
@After
public void tearDown() throws Exception {
    driver.quit();
}
```

However, if you enable grabbing screenshots on failure with the **ScreenshotOnFailureRule**, as described in Section 23.10.2, “Taking Screenshots on Failure”, the rules are executed after `@After`, but the driver needs to be open when the rule to take the screenshot is executed. Therefore, you should not quit the driver in that case. The rule quits the driver implicitly.

You can use any other JUnit features. Notice, however, that using TestBench requires that the driver has been created and is still open.

A complete test case could be as follows:

```
import com.vaadin.testbench.ScreenshotOnFailureRule;
import com.vaadin.testbench.TestBenchTestCase;
import com.vaadin.testbench.elements.ButtonElement;
import com.vaadin.testbench.elements.LabelElement;
```

```
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.openqa.selenium.firefox.FirefoxDriver;

import java.util.List;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;

public class MyprojectTest extends TestBenchTestCase {
    @Rule
    public ScreenshotOnFailureRule screenshotOnFailureRule =
        new ScreenshotOnFailureRule(this, true);

    @Before
    public void setUp() throws Exception {
        setDriver(new FirefoxDriver()); // Firefox
    }

    /**
     * Opens the URL where the application is deployed.
     */
    private void openTestUrl() {
        getDriver().get("http://localhost:8080/myproject");
    }

    @Test
    public void testClickButton() throws Exception {
        openTestUrl();

        // At first there should be no labels
        assertFalse($(LabelElement.class).exists());

        // Click the button
        ButtonElement clickMeButton = $(ButtonElement.class)
            .caption("Click Me").first();
        clickMeButton.click();

        // There should now be one label
        assertEquals(1, $(LabelElement.class).all().size());

        // ... with the specified text
        assertEquals("Thank you for clicking",
            $(LabelElement.class).first().getText());

        // Click the button again
        clickMeButton.click();

        // There should now be two labels
        List<LabelElement> allLabels =
            $(LabelElement.class).all();
        assertEquals(2, allLabels.size());

        // ... and the last label should have the correct text
        LabelElement lastLabel = allLabels.get(1);
        assertEquals("Thank you for clicking",
            lastLabel.getText());
    }
}
```


This test case stub is created by the Vaadin project wizard in Eclipse and by the Maven archetype, as described in Section 23.2, “Quick Start”.

23.4.2. Running JUnit Tests in Eclipse

The Eclipse IDE integrates JUnit with nice control features, such as running the tests in the current test source file. The test results are reported visually in the JUnit view in Eclipse.

New Vaadin projects created with the Vaadin Plugin for Eclipse contain the TestBench API dependency, as described in Section 23.2, “Quick Start”, so you can run TestBench tests right away.

To configure an existing project for TestBench testing, you need to do the following:

1. Include the TestBench API dependency in the project.
 - a. If using a project created with the Vaadin Plugin for Eclipse, add the TestBench API library dependency in `ivy.xml`. It should be as follows:

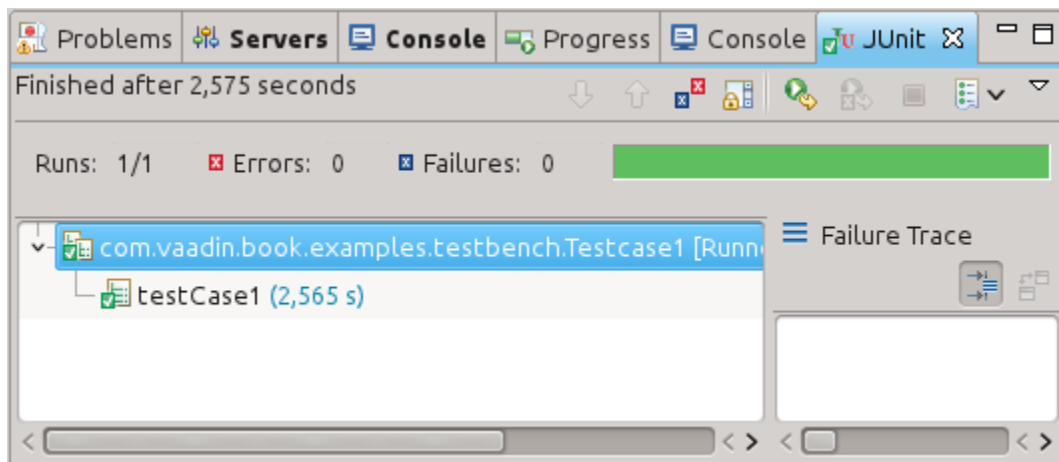
```
<dependency org="com.vaadin"
            name="vaadin-testbench-api"
            rev="latest.release"
            conf=nodeploy- → default/>
```

The TestBench API library provides element classes for Vaadin components, so its revision number follows the earliest supported Vaadin release. For old Vaadin versions, you can try using the `latest.release` as given above.

The project should contain the `nodeploy` configuration, as created for new Vaadin projects. See Section 18.3, “Installing Add-ons in Eclipse with Ivy” for more details.

- a. Otherwise, add the `vaadin-testbench-api` and `vaadin-testbench-core` JARs from the installation package to a library folder in the project, such as `lib`. You should not put the library in `WEB-INF/lib` as it is not used by the deployed Vaadin web application. Refresh the project by selecting it and pressing F5.
2. Right-click the project in Project Explorer and select **Properties**, and open the **Java Build Path** and the **Libraries** tab. Click **Add JARs**, navigate to the library folder, select the library, and click **OK**.
3. Switch to the **Order and Export** tab in the project properties. Make sure that the TestBench JAR is above the `gwt-dev.jar` (it may contain an old `httpClient` package), by selecting it and moving it with the **Up** and **Down** buttons.
4. Click **OK** to exit the project properties.
5. Right-click a test source file and select **Run As → JUnit Test**.

A JUnit view should appear, and it should open the Firefox browser, launch the application, run the test, and then close the browser window. If all goes well, you have a passed test case, which is reported in the JUnit view area in Eclipse, as illustrated in Figure 23.8, “Running JUnit Tests in Eclipse”.

Figure 23.8. Running JUnit Tests in Eclipse

If you are using some other IDE, it might support JUnit tests as well. If not, you can run the tests using Ant or Maven.

23.5. Creating a Test Case

23.5.1. Test Setup

Test configuration is done in a method annotated with `@Before`. The method is executed before each test case.

The basic configuration tasks are:

- Set TestBench parameters
- Create the web driver
- Do any other initialization

TestBench Parameters

TestBench parameters are defined with static methods in the **`com.vaadin.testbench.Parameters`** class. The parameters are mainly for screenshots and documented in Section 23.10, “Taking and Comparing Screenshots”.

23.5.2. Basic Test Case Structure

A typical test case does the following:

1. Open the URL
2. Navigate to desired state .. Find a HTML element (**`WebElement`**) for interaction
 - a. Use `click()` and other commands to interact with the element
 - b. Repeat with different elements until desired state is reached
3. Find a HTML element (**`WebElement`**) to check

4. Get and assert the value of the HTML element
5. Get a screenshot

The **WebDriver** allows finding HTML elements in a page in various ways, for example, with XPath expressions. The access methods are defined statically in the **By** class.

These tasks are realized in the following test code:

```
@Test
public void basic() throws Exception {
    getDriver().get("http://localhost:8080/tobetested");

    // Find an element to interact upon
    ButtonElement button =
        $(ButtonElement.class).id("mybutton");

    // Click the button
    button.click();

    // Check that the label text is correct
    LabelElement label = $(LabelElement.class).first();
    assertEquals("Thanks!", label.getText());
}
```

You can also use URI fragments in the URL to open the application at a specific state.

You should use the JUnit assertion commands. They are static methods defined in the `org.junit.Assert` class, which you can import (for example) with:

```
import static org.junit.Assert.assertEquals;
```

Please see the Selenium API documentation for a complete reference of the element search methods in the **WebDriver** and **By** classes and for the interaction commands in the **WebElement** class.

TestBench has a collection of its own commands, defined in the `TestBenchCommands` interface. You can get a command object that you can use by calling `testBench(driver)` in a test case.

While you can develop tests simply with test cases as described above, for the sake of maintainability it is often best to modularize the test code further, such as to separate testing at the levels of business logic and the page layout. See Section 23.9, “Creating Maintainable Tests” for information about using page objects for this purpose.

23.5.3. Creating and Closing a Web Driver

Vaadin TestBench uses Selenium WebDriver to execute tests in a browser. The **WebDriver** instance is created with the static `createDriver()` method in the **TestBench** class. It takes the driver as the parameter and returns it after registering it. The test cases must extend the **TestBenchTestCase** class, which manages the TestBench-specific features. You need to store the driver in the test case with `setDriver()`.

The basic way is to create the driver in a method annotated with the JUnit `@Before` annotation and close it in a method annotated with `@After`.

```
public class AdvancedTest extends TestBenchTestCase {
    @Before
    public void setUp() throws Exception {
        ...
        setDriver(TestBench.createDriver(new FirefoxDriver()));
    }
    ...
    @After
    public void tearDown() throws Exception {
        driver.quit();
    }
}
```

This creates the driver for each test you have in the test class, causing a new browser instance to be opened and closed. If you want to keep the browser open between the tests, you can use `@BeforeClass` and `@AfterClass` methods to create and quit the driver. In that case, the methods as well as the driver instance have to be static and you need to set the driver in a `@Before` method.

```
public class AdvancedTest extends TestBenchTestCase {
    static private WebDriver driver;

    @BeforeClass
    static public void createDriver() throws Exception {
        driver = TestBench.createDriver(new FirefoxDriver());
    }

    @Before
    public void setUp() throws Exception {
        setDriver(driver);
    }
    ...
    @AfterClass
    static public void tearDown() throws Exception {
        driver.quit();
    }
}
```

Browser Drivers

Please see the API documentation of the `WebDriver` interface for a complete list of supported drivers, that is, classes implementing the interface.

Both the Internet Explorer and Chrome require a special driver, as was noted in Section 23.3.5, “Installing Browser Drivers”. The driver executable must be included in the operating system `PATH`, be given with a driver-specific system Java property:

- Chrome: `webdriver.chrome.driver`
- IE: `webdriver.ie.driver`

You can set the property in Java with `System.setProperty(prop, key)` or pass it as a command-line parameter to the Java executable with `-Dwebdriver.chrome.driver=/path/to/driver`.

If you use an ESR version of Firefox, which is recommended for test stability, you need to the binary when creating the driver as follows:

```
FirefoxBinary binary =  
    new FirefoxBinary(new File("/path/to/firefox_ESR_10"));  
driver = TestBench.createDriver(  
    new FirefoxDriver(binary, new FirefoxProfile()));
```

23.6. Querying Elements

The high-level `ElementQuery` API allows querying Vaadin components in the browser according to their component class type, hierarchy, caption, and other properties. Once one or more components are found, they can be interacted upon. The query API forms an domain-specific language (DSL), embedded in the **`TestBenchTestCase`** class.

The basic idea of element queries match elements and return queries, which can again be queried upon, until terminated by a terminal query that returns one or more elements.

Consider the following query:

```
List<ButtonElement> buttons = $(ButtonElement.class).all();
```

The query returns a list of HTML elements of all the **Button** components in the UI. Every Vaadin component has its corresponding element class, which has methods to interact with the particular component type. We could control the buttons found by the query, for example, by clicking them as follows:

```
for (ButtonElement b: buttons)  
    b.click();
```

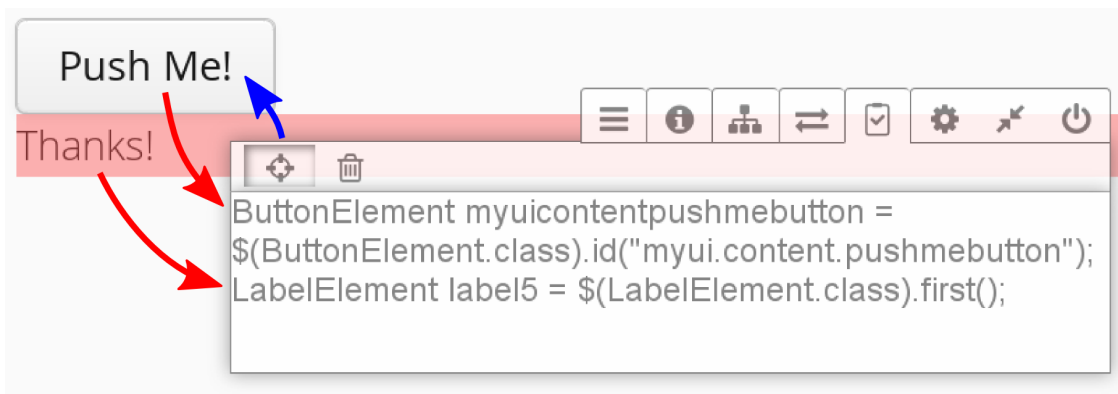
In the following sub-sections, we look into the details of element queries.

23.6.1. Generating Queries with Debug Window

You can use the debug window to easily generate the element query code to select a particular element in the UI. This should be especially useful when starting to use TestBench, to get the idea what the queries should be like.

First, enable the debug window with the `&debug` parameter for the application, as described in more detail in Section 12.3, “Debug Mode and Window”. You can interact with the UI in any way you like before generating the query code, but we recommend that you proceed by following the sequence in which the user would use the UI in each use case, making the queries at each step.

Switch to the TestBench tab in the debug window, and enable the pick mode by clicking the small button. Now, when you hover the mouse pointer on elements, it highlights them, and when you click one, it generates the TestBench element query to find the element. Use of the debug window is illustrated in Figure 23.9, “Using Debug Window to Generate Element Queries”.

Figure 23.9. Using Debug Window to Generate Element Queries

You can select and copy and paste the code from the debug window to your editor. To exit the pick mode, click the pick button again.

The debug window feature is available in Vaadin 7.2 and later.

23.6.2. Querying Elements by Component Type (\$)

The `$` method creates an **ElementQuery** that looks for the given element class. The method is available both in **TestBenchTestcase** and **ElementQuery**, which defines the context. The search is done recursively in the context.

```
// Find the first OK button in the UI
ButtonElement button = $(ButtonElement.class)
    .caption("OK").first();

// A nested query where the context of the latter
// component type query is the matching elements
// - matches the first Label inside the "content" layout.
LabelElement label = $(VerticalLayoutElement.class)
    .id("content").$(LabelElement.class).first();
```

23.6.3. Non-Recursive Component Queries (\$\$)

The `$$` method creates a non-recursive **ElementQuery**. It is a shorthand for first creating a recursive query with `$` and then calling `recursive(false)` for the query.

23.6.4. Element Classes

Each Vaadin component has a corresponding element class in TestBench, which contains methods for interacting with the particular component. The element classes extend **TestBenchElement**. It implements Selenium `WebElement`, so the Selenium element API can be used directly. The element classes are distributed in a Vaadin library rather than with TestBench, as they must correspond with the Vaadin version used in the application.

In addition to components, other Vaadin UI elements such as notifications (see Section 23.8.4, “Testing Notifications”) can have their corresponding element class. Add-on libraries may also define their custom element classes.

TestBenchElement is a TestBench command executor, so you can always use an element to create query in the sub-tree of the element. For example, in the following we first find a layout element by its ID and then do a sub-query to find the first label in it:

```
VerticalLayoutElement layout =  
    $(VerticalLayoutElement.class).id("content");  
LabelElement label = layout.$(LabelElement.class).first();
```

23.6.5. ElementQuery Objects

You can use an **ElementQuery** object to either make sub-queries to refine the query, or use a query terminator to finalize the query and get one or more matching elements.

23.6.6. Query Terminators

A query is finalized by a sub-query that returns an element or a collection of elements.

```
first()  
    Returns the first found element.  
  
get()  
    Returns the element by index in the collection of matching elements.  
  
all()  
    Returns a List of elements of the query type.  
  
id()  
    Returns the unique element having the given ID. Element IDs must always be unique  
    in the web page. It is therefore meaningless to make a complex query to match the  
    ID, just matching the element class is enough.
```

Web Elements

A query returns one or more elements extending Selenium **WebElement**. The particular element-specific class offers methods to manipulate the associated Vaadin component, while you can also use the lower-level general-purpose methods defined in **WebElement**.

23.7. Element Selectors

In addition to the high-level ElementQuery API described in the previous section, Vaadin TestBench includes the lower-level Selenium WebDriver API, with Vaadin extensions. You can find elements also by a plain XPath expression, an element ID, CSS style class, and so on. You can use such selectors together with the element queries. Like the ElementQuery API, it can be considered a domain-specific language (DSL) that is embedded in the **TestBenchTestCase** class.

The available selectors are defined as static methods in the **com.vaadin.testbench.By** class. They create and return a **By** instance, which you can use for the `findElement()` method in **WebDriver**.

The ID, CSS class, and Vaadin selectors are described below. For others, we refer to the Selenium WebDriver API documentation.

Some selectors are not applicable to all elements, for example if an element does not have an ID or is outside the Vaadin application.

23.7.1. Finding by ID

Selecting elements by their HTML element `id` attribute is a robust way to select elements, as noted in Section 23.9.1, “Increasing Selector Robustness”. It requires that you component IDs for the UI components with `setId()`.

```
Button button = new Button("Push Me!");
button.setId("pushmebutton");
```

The button would be rendered as a HTML element: `<div id="pushmebutton" ...>...</div>`. The element would then be accessible with a low-level WebDriver call:

```
findElement(By.id("pushmebutton")).click();
```

The selector is equivalent to the statically typed element query `$(ButtonElement.class).id("pushmebutton")`.

23.7.2. Finding by CSS Class

An element with a particular CSS style class name can be selected with the `By.className()` method. CSS selectors are useful for elements which have no ID, nor can be found easily from the component hierarchy, but do have a particular unique CSS style. Tooltips are one example, as they are floating `div` elements under the root element of the application. Their `v-tooltip` style makes it possible to select them as follows:

```
// Verify that the tooltip contains the expected text
String tooltipText = findElement(
    By.className("v-tooltip")).getText();
```

For a complete example, see the `AdvancedCommandsITCase.java` file in the TestBench demo described in Section 23.3.4, “TestBench Demo”.

23.8. Special Testing Topics

In the following, we go through a number of TestBench features for handling special cases, such as tooltips, scrolling, notifications, context menus, and profiling responsiveness. Finally, we look into the Page Object pattern.

23.8.1. Waiting for Vaadin

Selenium, on which Vaadin TestBench is based, is originally intended for regular web applications that load a page that is immediately rendered by the browser. In such applications, you can test the page elements immediately after the page is loaded. In Vaadin and other AJAX applications, rendering is done by JavaScript code asynchronously, so you need to wait until the server has given its response to an AJAX request and the JavaScript code finishes rendering the UI. Selenium supports AJAX applications by having special wait methods to poll the UI until the rendering is finished. In pure Selenium, you need to use the wait methods explicitly, and know what to use and when. Vaadin TestBench works together with the client-side engine of Vaadin framework to immediately detect when the rendering is finished. Waiting is implicit, so you do not normally need to insert any wait commands yourself.

Waiting is automatically enabled, but it may be necessary to disable it in some cases. You can do that by calling `disableWaitForVaadin()` in the `TestBenchCommands` interface. You can call it in a test case as follows:


```
testBench(driver).disableWaitForVaadin();
```

When disabled, you can wait for the rendering to finish by calling `waitForVaadin()` explicitly.

```
testBench(driver).waitForVaadin();
```

You can re-enable the waiting with `enableWaitForVaadin()` in the same interface.

23.8.2. Testing Tooltips

Component tooltips show when you hover the mouse over a component. Showing them require special command. Handling them is also special, as the tooltips are floating overlay element, which are not part of the normal component hierarchy.

Let us assume that you have set the tooltip as follows:

```
// Create a button with a component ID
Button button = new Button("Push Me!");
button.setId("main.button");

// Set the tooltip
button.setDescription("This is a tip");
```

The tooltip of a component is displayed with the `showTooltip()` method in the **TestBenchElementCommands** interface. You should wait a little to make sure it comes up. The floating tooltip element is not under the element of the component, but you can find it by `//div[@class='v-tooltip']` XPath expression.

```
@Test
public void testTooltip() throws Exception {
    driver.get(appUrl);

    ButtonElement button =
        $(ButtonElement.class).id("main.button");

    button.showTooltip();

    WebElement ttip = findElement(By.className("v-tooltip"));
    assertEquals(ttip.getText(), "This is a tip");
}
```

23.8.3. Scrolling

Some Vaadin components, such as **Table** and **Panel** have a scrollbar. Normally, when you interact with an element within such a scrolling region, TestBench implicitly tries to scroll to the element to make it visible. In some cases, you may wish to scroll a scrollbar explicitly. You can accomplish that with the `scroll()` (vertical) and `scrollLeft()` (horizontal) methods in the respective element classes for the scrollable components. The scroll position is given in pixels.

```
// Scroll to a vertical position
PanelElement panel = $(PanelElement.class)
    .caption("Scrolling Panel").first();
panel.scroll(123);
```

23.8.4. Testing Notifications

You can find notification elements by the **NotificationElement** class in the element query API. The element class supports getting the caption with `getCaption()`, description with `getDescription()`, and notification type with `getType()`.

Let us assume that you pop the notification up as follows:

```
Button button = new Button("Pop It Up", e -> // Java 8
    Notification.show("The caption", "The description",
        Notification.Type.WARNING_MESSAGE));
```

You could then check for the notification as follows:

```
// Click the button to open the notification
ButtonElement button =
    $(ButtonElement.class).caption("Pop It Up").first();
button.click();

// Verify the notification
NotificationElement notification =
    $(NotificationElement.class).first();
assertEquals("The caption", notification.getCaption());
assertEquals("The description", notification.getDescription());
assertEquals("warning", notification.getType());
notification.close();
```

You need to close the notification box with `close()` to move forward.

23.8.5. Testing Context Menus

Opening context menus requires special handling. First, to open a menu, you need to "context-click" on a specific sub-element in a component that supports context menus. You can do that with a `contextClick()` action in a **Actions** object.

A context menu is displayed as a floating element, which is under a special overlays element in the HTML page, not under the component from which it pops up. You can find it from the page by its CSS class `v-contextmenu`. The menu items are represented as text, and you can find the text with an XPath expression as shown in the example below.

In the following example, we open a context menu in a **Table** component, find an item by its caption text, and click it.

```
// Get a table cell to work on
TableElement table = inExample(TableElement.class).first();
WebElement cell = table.getCell(3, 0); // A cell in the row

// Perform context click action to open the context menu
new Actions(getDriver()).contextClick(cell).perform();

// Find the opened menu
WebElement menu = findElement(By.className("v-contextmenu"));

// Find a specific menu item
WebElement menuitem = menu.findElement(
    By.xpath("//*[text() = 'Add Comment']"));
```

```
// Select the menu item  
menuItem.click();
```

23.8.6. Profiling Test Execution Time

It is not just that it works, but also how long it takes. Profiling test execution times consistently is not trivial, as a test environment can have different kinds of latency and interference. For example in a distributed setup, timings taken on the test server would include the latencies between the test server, the grid hub, a grid node running the browser, and the web server running the application. In such a setup, you could also expect interference between multiple test nodes, which all might make requests to a shared application server and possibly also share virtual machine resources.

Furthermore, in Vaadin applications, there are two sides which need to be profiled: the server-side, on which the application logic is executed, and the client-side, where it is rendered in the browser. Vaadin TestBench includes methods for measuring execution time both on the server-side and the client-side.

The `TestBenchCommands` interface offers the following methods for profiling test execution time:

`totalTimeSpentServicingRequests()`

Returns the total time (in milliseconds) spent servicing requests in the application on the server-side. The timer starts when you first navigate to the application and hence start a new session. The time passes only when servicing requests for the particular session. The timer is shared in the servlet session, so if you have, for example, multiple portlets in the same application (session), their execution times will be included in the same total. *//TODO Vaadin 7: windows to roots*

Notice that if you are also interested in the client-side performance for the last request, you must call the `timeSpentRenderingLastRequest()` before calling this method. This is due to the fact that this method makes an extra server request, which will cause an empty response to be rendered.

`timeSpentServicingLastRequest()`

Returns the time (in milliseconds) spent servicing the last request in the application on the server-side. Notice that not all user interaction through the `WebDriver` cause server requests.

As with the total above, if you are also interested in the client-side performance for the last request, you must call the `timeSpentRenderingLastRequest()` before calling this method.

`totalTimeSpentRendering()`

Returns the total time (in milliseconds) spent rendering the user interface of the application on the client-side, that is, in the browser. This time only passes when the browser is rendering after interacting with it through the `WebDriver`. The timer is shared in the servlet session, so if you have, for example, multiple portlets in the same application (session), their execution times will be included in the same total.

`timeSpentRenderingLastRequest()`

Returns the time (in milliseconds) spent rendering user interface of the application after the last server request. Notice that not all user interaction through the `WebDriver` cause server requests.

If you also call the `timeSpentServicingLastRequest()` or `totalTimeSpentServicingRequests()`, you should do so before calling this method. The methods cause a server request, which will zero the rendering time measured by this method.

Generally, only interaction with fields in the *immediate* mode cause server requests. This includes button clicks. Some components, such as **Table**, also cause requests otherwise, such as when loading data while scrolling. Some interaction could cause multiple requests, such as when images are loaded from the server as the result of user interaction.

The following example is given in the `VerifyExecutionTimeITCase.java` file in the TestBench demo.

```
@Test
public void verifyServerExecutionTime() throws Exception {
    // Get start time on the server-side
    long currentSessionTime = testBench(getDriver())
        .totalTimeSpentServicingRequests();

    // Interact with the application
    calculateOnePlusTwo();

    // Calculate the passed processing time on the serve-side
    long timeSpentByServerForSimpleCalculation =
        testBench().totalTimeSpentServicingRequests() -
        currentSessionTime;

    // Report the timing
    System.out.println("Calculating 1+2 took about "
        + timeSpentByServerForSimpleCalculation
        + "ms in servlets service method.");

    // Fail if the processing time was critically long
    if (timeSpentByServerForSimpleCalculation > 30) {
        fail("Simple calculation shouldn't take " +
            timeSpentByServerForSimpleCalculation + "ms!");
    }

    // Do the same with rendering time
    long totalTimeSpentRendering =
        testBench().totalTimeSpentRendering();
    System.out.println("Rendering UI took "
        + totalTimeSpentRendering + "ms");
    if (totalTimeSpentRendering > 400) {
        fail("Rendering UI shouldn't take "
            + totalTimeSpentRendering + "ms!");
    }

    // A normal assertion on the UI state
    assertEquals("3.0",
        $(TextFieldElement.class).first()
            .getAttribute("value"));
}
```

23.9. Creating Maintainable Tests

The first important rule in developing tests is to keep them readable and maintainable. Otherwise, when the test fail, such as after refactoring the application code, the developers get impatient

in trying to understand them to fix them, and easily disable them. Readability and maintainability can be improved with the Page Object Pattern described below.

The second rule is to run the tests often. It is best to use a continuous integration server to run them at least once a day, or preferably on every commit.

23.9.1. Increasing Selector Robustness

Robustness of tests is important for avoiding failures because of irrelevant changes in the HTML DOM tree. Different selectors have differences in their robustness and it depends on how they are used.

The ElementQuery API uses the logical widget hierarchy to find the HTML elements, instead of the exact HTML DOM structure. This makes them somewhat robust, although still vulnerable to irrelevant changes in the exact component hierarchy of the UI. Also, if you internationalize the application, selecting components by their caption is not viable.

The low-level XPath selector can be highly vulnerable to changes in the DOM path, especially if the path is given down from the body element of the page. The selector is, however, very flexible, and can be used in robust ways, for example, by selecting by HTML element and a CSS class name or an attribute value. You can likewise use a CSS selector to select specific components by CSS class in a robust way.

Using Component IDs to Increase Robustness

To make UIs more robust for testing, you can set a unique *component ID* for specific components with `setId()`, as described in more detail in Section 23.7.1, “Finding by ID”.

Let us consider the following application, in which we set the IDs using a hierarchical notation to ensure that they are unique; in a more modular case you could consider a different strategy.

```
public class UIToBeTested extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setId("myui");

        final VerticalLayout content = new VerticalLayout();
        content.setMargin(true);
        content.setId("myui.content");
        setContent(content);

        // Create a button
        Button button = new Button("Push Me!");

        // Optional: give the button a unique ID
        button.setId("myui.content.pushmebutton");

        content.addComponent(button);
    }
}
```

After preparing the application this way, you can find the element by the component ID with the `id()` query terminator.

```
// Click the button
ButtonElement button =
```

```
$ (ButtonElement.class).id("myui.content.pushmebutton");  
button.click();
```

The IDs are HTML element `id` attributes and must be unique in the UI, as well as in the page in which the UI is running, in case the page has other content than the particular UI instance. In case there could be multiple UIs, you can include a UI part in the ID, as we did in the example above.

Using CSS Class Names to Increase Robustness

As a similar method to using component IDs, you can add CSS class names to components with `addStyleName()`. This enables matching them with the `findElement(By.className())` selector, as described in Section 23.7.2, “Finding by CSS Class”. You can use the selector in element queries. Unlike IDs, CSS class names do not need to be unique, so an HTML page can have many elements with the same CSS class.

You can use CSS class names also in XPath selectors.

23.9.2. The Page Object Pattern

The Page Object Pattern aims to simplify and modularize testing application views. The pattern follows the design principle of separation of concerns, to handle different concerns in separate modules, while hiding information irrelevant to other tests by encapsulation.

Defining a Page Object

A *page object* has methods to interact with a view or a sub-view, and to retrieve values in the view. You also need a method to open the page and navigate to the proper view.

For example:

```
public class CalculatorPageObject  
    extends TestBenchTestCase {  
    @FindBy(id = "button_=")  
    private WebElement equals;  
    ...  
  
    /**  
     * Opens the URL where the calculator resides.  
     */  
    public void open() {  
        getDriver().get(  
            "http://localhost:8080/?restartApplication");  
    }  
  
    /**  
     * Pushes buttons on the calculator  
     *  
     * @param buttons the buttons to push: "123+2", etc.  
     * @return The same instance for method chaining.  
     */  
    public CalculatorPageObject enter(String buttons) {  
        for (char numberChar : buttons.toCharArray()) {  
            pushButton(numberChar);  
        }  
        return this;  
    }  
}
```

```
/**
 * Pushes the specified button.
 *
 * @param button The character of the button to push.
 */
private void pushButton(char button) {
    getDriver().findElement(
        By.id("button_" + button)).click();
}

/**
 * Pushes the equals button and returns the contents
 * of the calculator "display".
 *
 * @return The string (number) shown in the "display"
 */
public String getResult() {
    equals.click();
    return display.getText();
}

...
}
```

Finding Member Elements By ID

If you have **WebElement** members annotated with **@FindBy**, they can be automatically filled with the HTML element matching the given component ID, as if done with `driver.findElement(By.id(fieldname))`. To do so, you need to create the page object with **PageFactory** as is done in the following test setup:

```
public class PageObjectExampleITCase {
    private CalculatorPageObject calculator;

    @Before
    public void setUp() throws Exception {
        driver = TestBench.createDriver(new FirefoxDriver());

        // Use PageFactory to automatically initialize fields
        calculator = PageFactory.initElements(driver,
            CalculatorPageObject.class);
    }
    ...
}
```

The members must be typed dynamically as **WebElement**, but you can wrap them to a typed element class with the `wrap()` method:

```
ButtonElement equals = equalsElement.wrap(ButtonElement.class);
```

Using a Page Object

Test cases can use the page object methods at business logic level, without knowing about the exact structure of the views.

For example:

```
@Test
public void testAddCommentRowToLog() throws Exception {
```

```
calculator.open();

// Just do some math first
calculator.enter("1+2");

// Verify the result of the calculation
assertEquals("3.0", calculator.getResult());

...
}
```

The Page Object Example

You can find the complete example of the Page Object Pattern in the `src/test/java/com/vaadin/testbenchexample/pageobjectexample/` folder in the TestBench Demo. The `PageObjectExampleITCase.java` runs tests on the Calc UI (also included in the example sources), using the page objects to interact with the different parts of the UI and to check the results.

The page objects included in the `pageobjects` subfolder are as follows:

- The **CalculatorPageObject** (as outlined in the example code above) has methods to click the buttons in the calculator and the retrieve the result shown in the "display".
- The **LogPageObject** can retrieve the content of the log entries in the log table, and right-click them to open the comment sub-window.
- The **AddComment** can enter a comment string in the comment editor sub-window and submit it (click the **Add** button).

23.10. Taking and Comparing Screenshots

You can take and compare screenshots with reference screenshots taken earlier. If there are differences, you can fail the test case.

23.10.1. Screenshot Parameters

The screenshot configuration parameters are defined with static methods in the **com.vaadin.testbench.Parameters** class.

```
screenshotErrorDirectory(default:[literal]null)
    Defines the directory where screenshots for failed tests or comparisons are stored.

screenshotReferenceDirectory(default:[literal]null)
    Defines the directory where the reference images for screenshot comparison are
    stored.

screenshotComparisonTolerance(default:[literal]0.01)
    Screen comparison is usually not done with exact pixel values, because rendering
    in browser often has some tiny inconsistencies. Also image compression may cause
    small artifacts.

screenshotComparisonCursorDetection(default:[literal>false)
    Some field component get a blinking cursor when they have the focus. The cursor
    can cause unnecessary failures depending on whether the blink happens to make
```


the cursor visible or invisible when taking a screenshot. This parameter enables cursor detection that tries to minimize these failures.

`maxScreenshotRetries`(default: 2)

Sometimes a screenshot comparison may fail because the screen rendering has not yet finished, or there is a blinking cursor that is different from the reference screenshot. For these reasons, Vaadin TestBench retries the screenshot comparison for a number of times defined with this parameter.

`screenshotRetryDelay`(default:[literal]500)

Delay in milliseconds for making a screenshot retry when a comparison fails.

For example:

```
@Before
public void setUp() throws Exception {
    Parameters.setScreenshotErrorDirectory(
        "screenshots/errors");
    Parameters.setScreenshotReferenceDirectory(
        "screenshots/reference");
    Parameters.setMaxScreenshotRetries(2);
    Parameters.setScreenshotComparisonTolerance(1.0);
    Parameters.setScreenshotRetryDelay(10);
    Parameters.setScreenshotComparisonCursorDetection(true);
    Parameters.setCaptureScreenshotOnFailure(true);
}
```

23.10.2. Taking Screenshots on Failure

Vaadin TestBench can take screenshots automatically when a test fails. To enable the feature, you need to include the **ScreenshotOnFailureRule** JUnit rule with a member variable annotated with **@Rule** in the test case as follows:

```
@Rule
public ScreenshotOnFailureRule screenshotOnFailureRule =
    new ScreenshotOnFailureRule(this, true);
```

Notice that you must not call `quit()` for the driver in the `@After` method, as that would close the driver before the rule takes the screenshot.

The screenshots are written to the error directory defined with the `screenshotErrorDirectory` parameter. You can configure it in the test case setup as follows:

```
@Before
public void setUp() throws Exception {
    Parameters.setScreenshotErrorDirectory("screenshots/errors");
    ...
}
```

23.10.3. Taking Screenshots for Comparison

Vaadin TestBench allows taking screenshots of the web browser window with the `compareScreen()` command in the **TestBenchCommands** interface. The method has a number of variants.

The `compareScreen([classname]#File)#` takes a **File** object pointing to the reference image. In this case, a possible error image is written to the error directory with the same file

name. You can get a file object to a reference image with the static `ImageFileUtil.getReferenceScreenshotFile()` helper method.

```
assertTrue("Screenshots differ",
    testBench(driver).compareScreen(
        ImageFileUtil.getReferenceScreenshotFile(
            "myshot.png")));
```

The `compareScreen([classname]#String)#` takes a base name of the screenshot. It is appended with browser identifier and the file extension.

```
assertTrue(testBench(driver).compareScreen("tooltip"));
```

The `compareScreen([classname]#BufferedImage,String)#` allows keeping the reference image in memory. An error image is written to a file with a name determined from the base name given as the second parameter.

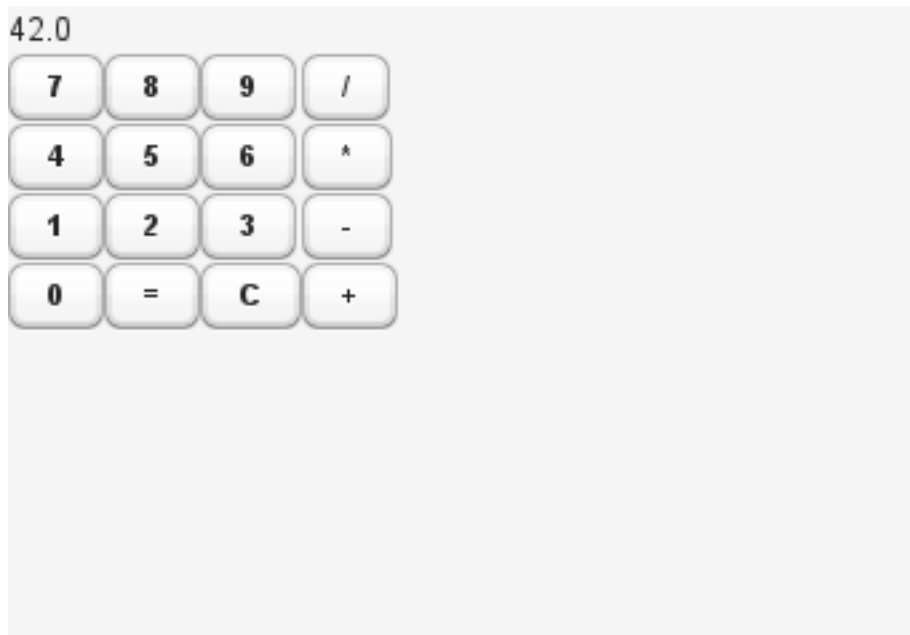
Screenshots taken with the `compareScreen()` method are compared to a reference image stored in the reference image folder. If differences are found (or the reference image is missing), the comparison method returns `false` and stores the screenshot in the error folder. It also generates an HTML file that highlights the differing regions.

Screenshot Comparison Error Images

Screenshots with errors are written to the error folder, which is defined with the `screenshotErrorDirectory` parameter described in Section 23.10.1, “Screenshot Parameters”.

For example, the error caused by a missing reference image could be written to `screenshot/errors/tooltip_firefox_12.0.png`. The image is shown in Figure 23.10, “A screenshot taken by a test run”.

Figure 23.10. A screenshot taken by a test run



Screenshots cover the visible page area in the browser. The size of the browser is therefore relevant for screenshot comparison. The browser is normally sized with a predefined default size. You can set the size of the browser window in a couple of ways. You can set the size of the browser window with, for example, `driver.manage().window().setSize(new Dimension(1024, 768))`; in the `@Before` method. The size includes any browser chrome, so the actual screenshot size will be smaller. To set the actual view area, you can use `TestBenchCommands.resizeViewportTo(1024, 768)`.

Reference Images

Reference images are expected to be found in the reference image folder, as defined with the `screenshotReferenceDirectory` parameter described in Section 23.10.1, "Screenshot Parameters". To create a reference image, just copy a screenshot from the `errors/` directory to the `reference/` directory.

For example:

```
$ cp screenshot/errors/tooltip_firefox_12.0.png screenshot/reference/
```

Now, when the proper reference image exists, rerunning the test outputs success:

```
$ java ...
JUnit version 4.5
.
Time: 18.222

OK (1 test)
```

Masking Screenshots

You can make masked screenshot comparison with reference images that have non-opaque regions. Non-opaque pixels in the reference image, that is, ones with less than 1.0 value in the alpha channel, are ignored in the screenshot comparison.

Please see the `ScreenshotITCase.java` example in the TestBench Demo for an example of using masked screenshots. The `example/Screenshot_Comparison_Tests.pdf` document describes how to enable the example and how to create the screenshot masks in an image editor.

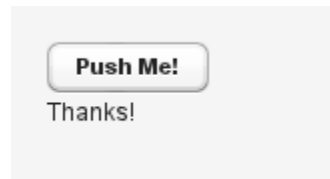
Visualization of Differences in Screenshots with Highlighting

Vaadin TestBench supports advanced difference visualization between a captured screenshot and the reference image. A difference report is written to a HTML file that has the same name as the failed screenshot, but with `.html` suffix. The reports are written to the same `errors/` folder as the screenshots from the failed tests.

The differences in the images are highlighted with blue rectangles. Moving the mouse pointer over a square shows the difference area as it appears in the reference image. Clicking the image switches the entire view to the reference image and back. Text "**Image for this run**" is displayed in the top-left corner of the screenshot to distinguish it from the reference image.

Figure 23.11, "The reference image and a highlighted error image" shows a difference report with one difference between the visualized screenshot (bottom) and the reference image (top).

Figure 23.11. The reference image and a highlighted error image



23.10.4. Practices for Handling Screenshots

Access to the screenshot reference image directory should be arranged so that a developer who can view the results can copy the valid images to the reference directory. One possibility is to store the reference images in a version control system and check-out them to the `reference/` directory.

A build system or a continuous integration system can be configured to automatically collect and store the screenshots as build artifacts.

23.10.5. Known Compatibility Problems

Screenshots when running Internet Explorer 9 in Compatibility Mode

Internet Explorer prior to version 9 adds a two-pixel border around the content area. Version 9 no longer does this and as a result screenshots taken using Internet Explorer 9 running in compatibility mode (IE7/IE8) will include the two pixel border, contrary to what the older versions of Internet Explorer do.

23.11. Running Tests

During test development, you usually run the tests from your IDE. After that, you want to have them run by a build system, possibly under a continuous integration system. In the following, we describe how to run tests by Ant and Maven.

23.11.1. Running Tests with Ant

Apache Ant has built-in support for executing JUnit tests; you can use the `<junit>` task in an Ant script to execute JUnit tests. Note that in earlier versions, you need to enable the support, you need to have the JUnit library `junit.jar` and its Ant integration library `ant-junit.jar` in the Ant classpath, as described in the Ant documentation.

The following Ant script allows testing a Vaadin application created with the Vaadin Plugin for Eclipse. It assumes that the test source files are located under a `test` directory under the current directory and compiles them to the `classes` directory. The the class path is defined with the `classpath` reference ID and should contain TestBench and other necessary libraries.

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="run-tests">
  <path id="classpath">
    <fileset dir="lib">
      <include name="vaadin-testbench-*.jar"/>
      <include name="junit-*.jar"/>
    </fileset>
  </path>
```

```
<!-- This target compiles the JUnit tests. -->
<target name="compile-tests">
  <mkdir dir="classes" />
  <javac srcdir="test" destdir="classes"
    debug="on" encoding="utf-8"
    includeantruntime="false">
    <classpath>
      <path refid="classpath" />
    </classpath>
  </javac>
</target>

<!-- This target calls JUnit -->
<target name="run-tests" depends="compile-tests">
  <junit fork="yes">
    <classpath>
      <path refid="classpath" />
      <pathelement path="classes" />
    </classpath>

    <formatter type="brief" usefile="false" />

    <batchtest>
      <fileset dir="test">
        <include name="**/*.java" />
      </fileset>
    </batchtest>
  </junit>
</target>
</project>
```

You also need to deploy the application to test, and possibly launch a dedicated server for it.

Retrieving TestBench with Ivy

To retrieve TestBench and its dependencies with Ivy in the Ant script, first install Ivy to your Ant installation, if necessary. In the build script, you need to enable Ivy with the namespace declaration and include a target for retrieving the libraries, as follows:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
  default="run-tests">
  ...
  <!-- Retrieve dependencies with Ivy -->
  <target name="resolve">
    <ivy:retrieve conf="testing" type="jar,bundle"
      pattern="lib/[artifact]-[revision].[ext]" />
  </target>

  <!-- This target compiles the JUnit tests. -->
  <target name="compile-tests" depends="resolve">
    ...
```

This requires that you have a "testing" configuration in your `ivy.xml` and that the TestBench dependency are enabled in the configuration.

```
<ivy-module>
  ...
  <configurations>
    ...
```

```
<conf name="testing" />
</configurations>

<dependencies>
...
<!-- TestBench 4 -->
<dependency org="com.vaadin"
    name="vaadin-testbench-api"
    rev="latest.release"
    conf="nodeploy,testing -> default" />
...
```

You also need to build and deploy the application to be tested to the server and install the TestBench license key.

23.11.2. Running Tests with Maven

Executing JUnit tests with Vaadin TestBench under Maven requires defining it as a dependency in any POM that needs to execute TestBench tests.

A complete example of a Maven test setup is given in the TestBench demo project available at github.com/vaadin/testbench-demo. See the `README` for further instructions.

Defining TestBench as a Dependency

You need to define the TestBench library as a dependency in the Maven POM of your project as follows:

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-testbench</artifactId>
  <version>4.x.x</version>
</dependency>
```

For instructions on how to create a new Vaadin project with Maven, please see Section 3.5, “Creating a Project with Maven”.

Running the Tests

To compile and run the tests, simply execute the `test` lifecycle phase with Maven as follows:

```
$ mvn test
...
-----
T E S T S
-----
Running TestBenchExample
Tests run: 6, Failures: 1, Errors: 0, Skipped: 1, Time elapsed: 36.736 sec <<<
FAILURE!

Results :

Failed tests:
  testDemo(TestBenchExample):
    expected:<[5/17/]12> but was:<[17.6.20]12>

Tests run: 6, Failures: 1, Errors: 0, Skipped: 1
...
```

The example configuration starts Jetty to run the application that is tested.

If you have screenshot tests enabled, as mentioned in Section 23.3.4, “TestBench Demo”, you will get failures from screenshot comparison. The failed screenshots are written to the `target/testbench/errors` folder. To enable comparing them to “expected” screenshots, you need to copy the screenshots to the `src/test/resources/screenshots/reference/` folder. See Section 23.10, “Taking and Comparing Screenshots” for more information regarding screenshots.

23.12. Running Tests in a Distributed Environment

A distributed test environment consists of a grid hub and a number of test nodes. The hub listens to calls from test runners and delegates them to the grid nodes. Different nodes can run on different operating system platforms and have different browsers installed.

A basic distributed installation was covered in Section 23.3.2, “A Distributed Testing Environment”.

23.12.1. Running Tests Remotely

Remote tests are just like locally executed tests, except instead of using a browser driver, you use a remote web driver that can connect to the hub. The hub delegates the connection to a grid node with the desired capabilities, that is, which browsers are installed in the node.

Instead of creating and handling the remote driver explicitly, as described in the following, you can use the **ParallelTest** framework presented in Section 23.13, “Parallel Execution of Tests”.

An example of remote execution of tests is given in the TestBench demo described in Section 23.3.4, “TestBench Demo”. See the `README.md` file for further instructions.

In the following example, we create and use a remote driver that runs tests in a Selenium cloud at `testingbot.com`. The desired capabilities of a test node are described with a **DesiredCapabilities** object.

```
public class UsingHubITCase extends TestBenchTestCase {

    private String baseUrl;
    private String clientKey = "INSERT-YOUR-CLIENT-KEY-HERE";
    private String clientSecret = "INSERT-YOUR-CLIENT-KEY-HERE";

    @Before
    public void setUp() throws Exception {
        // Create a RemoteDriver against the hub.
        // In you local setup you don't need key and secret,
        // but if you use service like testingbot.com, they
        // can be used for authentication
        URL testingbotdotcom = new URL("http://" +
            clientKey + ":" + clientSecret +
            "@hub.testingbot.com:4444/wd/hub");
        setDriver(new RemoteWebDriver(testingbotdotcom,
            DesiredCapabilities.iphone()));
        baseUrl = "http://demo.vaadin.com/Calc/";
    }

    @Test
    @Ignore("Requires testingbot.com credentials")
    public void testOnePlusTwo() throws Exception {
```

```
// run the test just as with "local bots"
openCalculator();
$(ButtonElement.class).caption("1").first().click();
$(ButtonElement.class).caption("+").first().click();
$(ButtonElement.class).caption("2").first().click();
$(ButtonElement.class).caption("=").first().click();
assertEquals("3.0", $(TextFieldElement.class)
    .first().getAttribute("value"));

// Thats it. Services may provide also some other goodies
// like the video replay of your test in testingbot.com
}

private void openCalculator() {
    getDriver().get(baseUrl);
}

@After
public void tearDown() throws Exception {
    getDriver().quit();
}
}
```

Please see the API documentation of the **DesiredCapabilities** class for a complete list of supported capabilities.

Running the example requires that the hub service and the nodes are running. Starting them is described in the subsequent sections. Please refer to Selenium documentation for more detailed information.

23.12.2. Starting the Hub

The TestBench grid hub listens to calls from test runners and delegates them to the grid nodes. The grid hub service is included in the Vaadin TestBench JAR and you can start it with the following command:

```
$ java -jar vaadin-testbench-standalone-4.x.x.jar \
    -role hub
```

You can open the control interface of the hub also with a web browser. Using the default port, just open URL <http://localhost:4444/>. Once you have started one or more grid nodes, as instructed in the next section, the "console" page displays a list of the grid nodes with their browser capabilities.

23.12.3. Node Service Configuration

Test nodes can be configured with command-line options, as described later, or in a configuration file in JSON format. If no configuration file is provided, a default configuration is used.

A node configuration file is specified with the `-nodeConfig` parameter to the node service, for example as follows:

```
$ java -jar vaadin-testbench-standalone-4.x.x.jar \
    -role node -nodeConfig nodeConfig.json
```

See Section 23.12.4, "Starting a Grid Node" for further details on starting the node service.

Configuration File Format

The test node configuration file follows the JSON format. It contains nested associative maps and lists to define parameters. An associative map is defined as a block enclosed in curly braces (`{ }`). A mapping is a key-value pair separated with a colon (`:`). A key is a string literal quoted with double quotes (`key`). The value can be a string literal, list, or a nested associative map. A list is a comma-separated sequence enclosed within square brackets (`[ ]`).

The top-level associative map should have two associations: *capabilities* (to a list of associative maps) and *configuration* (to a nested associative map).

```
{
  "capabilities":
  [
    {
      "browserName": "firefox",
      ...
    },
    ...
  ],
  "configuration":
  {
    "port": 5555,
    ...
  }
}
```

A complete example is given later.

Browser Capabilities

The browser capabilities are defined as a list of associative maps as the value of the *capabilities* key. The capabilities can also be given from command-line using the *-browser* parameter, as described in Section 23.12.4, “Starting a Grid Node”.

The keys in the map are the following:

platform

The operating system platform of the test node: WINDOWS, XP, VISTA, LINUX, or MAC.

browserName

A browser identifier, any of: android, chrome, firefox, htmlunit, internet explorer, iphone, opera, or phantomjs (as of TestBench 3.1).

maxInstances

The maximum number of browser instances of this type open at the same time for parallel testing.

version

The major version number of the browser.

seleniumProtocol

This should be `WebDriver` for `WebDriver` use.

firefox_binary

Full path and file name of the Firefox executable. This is typically needed if you have Firefox ESR installed in a location that is not in the system path.

Server Configuration

The node service configuration is defined as a nested associative map as the value of the `configuration` key. The configuration parameters can also be given as command-line parameters to the node service, as described in Section 23.12.4, “Starting a Grid Node”.

See the following example for a typical server configuration.

Example Configuration

```
{
  "capabilities":
  [
    {
      "browserName": "firefox",
      "maxInstances": 5,
      "seleniumProtocol": "WebDriver",
      "version": "10",
      "firefox_binary": "/path/to/firefox10"
    },
    {
      "browserName": "firefox",
      "maxInstances": 5,
      "version": "16",
      "firefox_binary": "/path/to/firefox16"
    },
    {
      "browserName": "chrome",
      "maxInstances": 5,
      "seleniumProtocol": "WebDriver"
    },
    {
      "platform": "WINDOWS",
      "browserName": "internet explorer",
      "maxInstances": 1,
      "seleniumProtocol": "WebDriver"
    }
  ],
  "configuration":
  {
    "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
    "maxSession": 5,
    "port": 5555,
    "host": ip,
    "register": true,
    "registerCycle": 5000,
    "hubPort": 4444
  }
}
```

23.12.4. Starting a Grid Node

A TestBench grid node listens to calls from the hub and is capable of opening a browser. The grid node service is included in the Vaadin TestBench JAR and you can start it with the following command:

```
$ java -jar \
    vaadin-testbench-standalone-4.x.x.jar \
```

```
-role node \  
-hub http://localhost:4444/grid/register
```

The node registers itself in the grid hub. You need to give the address of the hub either with the `-hub` parameter or in the node configuration file as described in Section 23.12.3, “Node Service Configuration”.

You can run one grid node in the same host as the hub, as is done in the example above with the localhost address.

Browser Capabilities

The browsers installed in the node can be defined either with command-line parameters or with a configuration file in JSON format, as described in Section 23.12.3, “Node Service Configuration”.

On command-line, you can issue one or more `-browser` options to define the browser capabilities. It must be followed by a comma-separated list of property-value definitions, such as the following:

```
-browser "browserName=firefox,version=10,firefox_binary=/path/to/firefox10" \  
-browser "browserName=firefox,version=16,firefox_binary=/path/to/firefox16" \  
-browser "browserName=chrome,maxInstances=5" \  
-browser "browserName=internet explorer,maxInstances=1,platform=WINDOWS"
```

The configuration properties are described in Section 23.12.3, “Node Service Configuration”.

Browser Driver Parameters

If you use Chrome or Internet Explorer, their remote driver executables must be in the system path (in the `PATH` environment variable) or be given with a command-line parameter to the node service:

```
Internet Explorer  
-Dwebdriver.ie.driver=C:\path\to\IEDriverServer.exe  
  
Google Chrome  
-Dwebdriver.chrome.driver=/path/to/ChromeDriver
```

23.12.5. Mobile Testing

Vaadin TestBench includes an iPhone and an Android driver, with which you can test on mobile devices. The tests can be run either in a device or in an emulator/simulator.

The actual testing is just like with any WebDriver, using either the **IPhoneDriver** or the **AndroidDriver**. The Android driver assumes that the hub (`android-server`) is installed in the emulator and forwarded to port 8080 in localhost, while the iPhone driver assumes port 3001. You can also use the **RemoteWebDriver** with either the `iphone()` or the `android()` capability, and specify the hub URI explicitly.

The mobile testing setup is covered in detail in the Selenium documentation for both the iOS driver and the AndroidDriver.

23.13. Parallel Execution of Tests

The **ParallelTest** class provides an easy way to run tests in parallel locally, as well as remotely in a test grid.

23.13.1. Local Parallel Execution

To enable parallel execution of tests, usually during test development, you need to extend the **ParallelTest** instead of **TestBenchTestCase** and annotate the test case class with `@RunLocally`.

```
@RunLocally
public class MyTest extends ParallelTest {
    @Test
    ...
}
```

When you run the tests, TestBench launches multiple browser windows to run each test in parallel.

Parallel execution defaults to Firefox. You can give another browser as a parameter for the annotation, as enumerated in the **Browser** enumeration:

```
@RunLocally(Browser.CHROME)
```

For Chrome and IE, you need to have the browser driver installed, as described in Section 23.3.5, “Installing Browser Drivers”.

23.13.2. Multi-Browser Execution in a Grid

To run tests in multiple different browsers or remotely, you first need to set up and launch a grid hub and one or more grid nodes, as described in Section 23.12, “Running Tests in a Distributed Environment”. This enables remote execution in a test grid, although you can run the hub and a test node also in your development workstation.

To run a test case class in a grid, you simply need to annotate the test case classes with the `@RunOnHub` annotation. It takes the host address of the hub as parameter, with `localhost` as the default host. You need to define the desired browser capabilities in a method annotated with `@BrowserConfiguration`. It must return a list of **DesiredCapabilities**.

```
@RunOnHub("hub.testgrid.mydomain.com")
public class MyTest extends ParallelTest {
    @Test
    ...

    @BrowserConfiguration
    public List<DesiredCapabilities> getBrowserConfiguration() {
        List<DesiredCapabilities> browsers =
            new ArrayList<DesiredCapabilities>();

        // Add all the browsers you want to test
        browsers.add(BrowserUtil.firefox());
        browsers.add(BrowserUtil.chrome());
        browsers.add(BrowserUtil.ie11());

        return browsers;
    }
}
```

The actual browsers tested depends on the browser capabilities of the test node or nodes.

If you have more test classes, you can put the configuration in a common base class that extends **ParallelTest**.

23.14. Headless Testing

TestBench (3.1 and later) supports fully-featured headless testing with PhantomJS (<http://phantomjs.org>), a headless browser based on WebKit. It has fast native support for various web standards: JavaScript, DOM handling, CSS selector, JSON, Canvas, and SVG.

Headless testing using PhantomJS allows for around 15% faster test execution without having to start a graphical web browser, even when performing screenshot-based testing! This also makes it possible to run full-scale functional tests on the front-end directly on a build server, without the need to install any web browsers.

It is usually best to use a graphical browser to develop the test cases, as it is possible to see interactively what happens while the tests are being executed. Once the tests are working correctly in a graphical browser, you can migrate them to run on the PhantomJS headless browser.

23.14.1. Basic Setup for Running Headless Tests

The only set up required is to install the PhantomJS binary. Follow the instructions for your operating system at PhantomJS download page, and place the binary in the system path.

The PhantomJSDriver dependency is already included in Vaadin TestBench.

Creating a Headless WebDriver Instance

Creating an instance of the **PhantomJSDriver** is just as easy as creating an instance of **FirefoxDriver**.

```
setDriver(TestBench.createDriver(  
    new PhantomJSDriver()));
```

Some tests may fail because of the small default window size in PhantomJS. Such tests are, for example, tests containing elements that pop up and might go off-screen when the window is small. To make them work better, specify a size for the window:

```
getDriver().manage().window().setSize(  
    new Dimension(1024, 768));
```

Nothing else is needed to run tests headlessly.

23.14.2. Running Headless Tests in a Distributed Environment

Running PhantomJS in a distributed grid is equally easy. First, install PhantomJS in the nodes by following the instructions in Section 23.14.1, “Basic Setup for Running Headless Tests”. Then, start PhantomJS using the following command:

```
phantomjs --webdriver=8080 \  
--webdriver-selenium-grid-hub=http://127.0.0.1:4444
```

The above will start PhantomJS in the WebDriver mode and register it with a grid hub running at 127.0.0.1:4444. After this, running tests in the grid is as easy as passing `DesiredCapabilities.phantomjs()` to the `RemoteWebDriver` constructor.

```
setDriver(new RemoteWebDriver(  
    DesiredCapabilities.phantomjs()));
```

23.15. Behaviour-Driven Development

Behaviour-driven development (BDD) is a development methodology based on test-driven development, where development starts from writing tests for the software-to-be. BDD involves using a *ubiquitous language* to communicate between business goals - the desired behaviour - and tests to ensure that the software fulfills those goals.

The BDD process starts by collection of business requirements expressed as *user stories*, as is typical in agile methodologies. A user with a *role* requests a *feature* to gain a *benefit*.

Stories can be expressed as number of *scenarios* that describe different cases of the desired behaviour. Such a scenario can be formalized with the following three phases:

- *Given* that I have calculator open
- *When* I push calculator buttons
- *Then* the display should show the result

This kind of formalization is realized in the JBehave BDD framework for Java. The TestBench Demo includes a JBehave example, where the above scenario is written as the following test class:

```
public class CalculatorSteps extends TestBenchTestCase {  
    private WebDriver driver;  
    private CalculatorPageObject calculator;  
  
    @BeforeScenario  
    public void setUpWebDriver() {  
        driver = TestBench.createDriver(new FirefoxDriver());  
        calculator = PageFactory.initElements(driver,  
            CalculatorPageObject.class);  
    }  
  
    @AfterScenario  
    public void tearDownWebDriver() {  
        driver.quit();  
    }  
  
    @Given("I have the calculator open")  
    public void theCalculatorIsOpen() {  
        calculator.open();  
    }  
  
    @When("I push $buttons")  
    public void enter(String buttons) {  
        calculator.enter(buttons);  
    }  
  
    @Then("the display should show $result")  
    public void displayShows(String result) {  
        assertEquals(result, calculator.getResult());  
    }  
}
```

The demo employs the page object defined for the application UI, as described in Section 23.9.2, “The Page Object Pattern”.

Such scenarios are included in one or more stories, which need to be configured in a class extending **JUnitStory** or **JUnitStories**. In the example, this is done in the <https://github.com/vaadin/testbench-demo/blob/master/src/test/java/com/vaadin/testbenchexample/bdd/SimpleCalculation.java> class. It defines how story classes can be found dynamically by the class loader and how stories are reported.

For further documentation, please see JBehave website at jbehave.org.

23.16. Integration Testing with Maven

TestBench is often used in Maven projects, where you want to run tests as part of the build lifecycle. While ordinary Java unit tests are usually executed in the `test` phase, TestBench tests are usually executed in the `integration-test` phase (to run the phase properly you need to invoke `verify` phase as explained later).

Section 23.2.3, “Quick Start with Maven” describes how to use the Vaadin application archetype for Maven provides a TestBench setup. In this section, we largely go through that and also describe how to make such a setup also in Vaadin add-on projects.

23.16.1. Project Structure

In Vaadin Applications

In a typical Vaadin application, such as in the one created by the archetype, you only have one module and you run tests there. The application sources would normally be in `src/main` and test code, together with TestBench tests, in `src/test`.

In Libraries and Add-ons

In multi-module projects, you may have libraries in other modules, and the actual web application in another. Here you could do library unit tests in the library modules, and integration tests in the web application module.

The multi-module project structure is recommended for Vaadin add-ons, where you have the add-on library in one module and a demo in another.

23.16.2. Overview of Lifecycle

The Maven lifecycle phases relevant to TestBench execution are as follows:

1. `compile`
 - Compile server-side
 - Compile widget set
 - Compile theme
2. `test-compile`
 - Compile test classes (both unit and integration tests)

3. `pre-integration-test`
 - Start web server (Jetty or other)
4. `integration-test`
 - Execute TestBench tests
5. `post-integration-test`
 - Stop web server
6. `verify`
 - Verify the results of the integration tests

23.16.3. Overview of Configuration

The Maven configuration in the `pom.xml` should include the following parts, with our reference toolchain given in parentheses:

- Integration test runner (Maven Failsafe Plugin)
- Web server (Jetty)
- Web server runner (Jetty Maven Plugin)
- Vaadin compilation and deployment (Vaadin Maven Plugin)

23.16.4. Vaadin Plugin Configuration

The Vaadin Maven Plugin compiles the widget set with the Vaadin Client Compiler and the theme with the Vaadin Sass compiler in the `compile` phase. Its configuration should be as is normal for Vaadin projects. The default configuration is created by the Vaadin project archetype, as described in Section 3.5, “Creating a Project with Maven”.

23.16.5. Configuring Integration Testing

Our reference toolchain uses the Maven Failsafe Plugin to execute integration tests with TestBench. The plugin executes tests (JUnit or other) defined in the test files included in the plugin configuration.

To run TestBench tests made with JUnit, you need that dependency in the `<dependencies>` section:

```
<!-- Needed for running TestBench tests -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

Surefire requires the following plugin configuration (under `<plugins>`):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
```



```
<artifactId>maven-failsafe-plugin</artifactId>
<version>2.17</version>
<configuration>
  <includes>
    <include>/**/*.Tests.java</include>
  </includes>
  <excludes>
    <!-- Here list files that might match to naming conventions unintention-
ally. We can ignore them from testing. -->
  </excludes>
</configuration>
<executions>
  <execution>
    <id>failsafe-integration-tests</id>
    <phase>integration-test</phase>
    <goals>
      <goal>integration-test</goal>
    </goals>
  </execution>
  <execution>
    <id>failsafe-verify</id>
    <phase>verify</phase>
    <goals>
      <goal>verify</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

Set the include and exclude patterns according to your naming convention. See Failsafe documentation for more details about the configuration, for example, if you want to use other test provider than JUnit.

23.16.6. Configuring Test Server

We use Jetty as our reference test server, as it is a light-weight server that is easy to configure with a Maven plugin.

The dependency for Jetty goes as follows:

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-webapp</artifactId>
  <version>9.2.10.v20150310</version>
  <scope>test</scope>
</dependency>
```

The plugin configuration for running Jetty goes as follows:

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>8.1.16.v20140903</version>

  <configuration>
    <webApp>
      <contextPath>/myapp</contextPath>
    </webApp>
    <stopKey>STOP</stopKey>
```

```
<stopPort>8005</stopPort>
</configuration>

<executions>
  <execution>
    <id>start-jetty</id>
    <phase>pre-integration-test</phase>
    <goals>
      <goal>start</goal>
    </goals>
    <configuration>
      <daemon>true</daemon>
      <scanIntervalSeconds>0</scanIntervalSeconds>
    </configuration>
  </execution>
  <execution>
    <id>stop-jetty</id>
    <phase>post-integration-test</phase>
    <goals>
      <goal>stop</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

Here you only need to configure the `contextPath` parameter, which is the context path to your web application.

23.17. Known Issues

This section provides information and instructions on a few features that are known to be difficult to use or need modification to work.

23.17.1. Running Firefox Tests on Mac OS X

Firefox needs to have focus in the main window for any focus events to be triggered. This sometimes causes problems if something interferes with the focus. For example, a **TextField** that has an input prompt relies on the JavaScript `onFocus()` event to clear the prompt when the field is focused.

The problem occurs when OS X considers the Java process of an application using TestBench (or the node service) to have a native user interface capability, as with AWT or Swing, even when they are not used. This causes the focus to switch from Firefox to the process using TestBench, causing tests requiring focus to fail. To remedy this problem, you need to start the JVM in which the tests are running with the `-Djava.awt.headless=true` parameter to disable the user interface capability of the Java process.

Note that the same problem is present also when debugging tests with Firefox. We therefore recommend using Chrome for debugging tests, unless Firefox is necessary.

Index

Symbols

@ApplicationScoped, 415
@CDIUI, 413-414
@Connect, 468
@PreserveOnRefresh, 106
@SessionScoped, 415
@SpringUI, 420
@UIScoped, 414

A

AbstractComponent, 119, 121-122
AbstractComponentContainer, 119
AbstractComponentState, 468
AbstractField, 119
add-ons
 creating, 483-485
addContainerFilter(), 341
addNestedContainerProperty(), 338
AJAX, 30, 75
Alignment, 259-261
And (filter), 342

B

BrowserWindowOpener, 105

C

caption, 122
caption property, 122
CDI, 411-417
 scopes, 414-415
Client-Side Engine, 73, 77
close(), 107
 UI, 106
closeIdleSessions, 107, 114
closing, 106, 107
compatibility, 296
component, 73
Component, 119
Component interface, 121
 caption, 122
 description, 124
 enabled, 125
 icon, 126
 locale, 127
 read-only, 128
 style name, 129
 visible, 129
connector, 467
Container, 333-343

 Filterable, 180, 341-343
 context menus, 652
Contexts and Dependency Injection, 411-417
CSS, 73, 75, 287-317
 compatibility, 296
 introduction, 289-296
CSS selections, 316
CSS3, 75
custom, 106

D

Data Binding, 74
Data Model, 74
DefaultUIProvider, 105
description, 124
description property, 124
DOM, 74, 75
Drag and Drop, 390-399
 Accept Criteria, 394-397

E

Eclipse
 widget development, 469-472
enabled, 125
enabled property, 125
Equal (filter), 342
events, 73
execute(), 375
expiration, 106, 107, 108
extension, 316

F

field, 73
Field, 132-138
Filter (in Container), 341-343
Filterable, 180

G

Google Web Toolkit, 26, 30, 73, 74, 75, 222
 themeing, 290
 widgets, 465-490
Greater (filter), 342
GreaterOrEqual (filter), 342
Grid, 189

H

heartbeat, 108
HorizontalSplitPanel, 246-248
HTML, 74
HTML 5, 75
HTML templates, 73
HTTP, 73

I

- icon, 126
- icon property, 126
- in Component, 127
- IndexedContainer, 341
- init(), 105
- interfaces, 120
- IPC add-on, 440-445
- IsNull (filter), 342
- IT Mill Toolkit, 30

J

- JavaDoc, 119
- JavaScript, 26, 74, 75, 469
 - execute(), 375
 - print(), 375-376
- JavaScript integration, 486-490
- JPAContainer, 553-576

L

- layout, 119
- Layout, 119
- Less (filter), 342
- LessOrEqual (filter), 342
- Liferay
 - display descriptor, 438
 - plugin properties, 438-439
 - portlet descriptor, 437-438
- liferay-display.xml, 438
- liferay-plugin-package.xml, 438-439
- loading, 105
- locale, 127
- locale property
 - in Component, 127
- Log4j, 399
- logout, 107

M

- Maven
 - compiling, 60
 - creating a project, 58-60
 - using add-ons, 60, 494-497
- memory leak, 400
- MethodProperty, 339

N

- nested bean properties, 337-339, 340
- NestedMethodProperty, 339
- Not (filter), 342
- Notification
 - testing, 652
- Null representation, 152

O

- Or (filter), 342
- Out of Sync, 113
- overflow, 245
- overflow CSS property, 223, 242

P

- Page
 - setLocation(), 107
- Paintable, 121
- PDF, 376
- PermGen, 400
- popup, 358
- popup windows, 358
- portal integration, 425-445
- preserving on refresh, 106
- print(), 375-376
- printing, 375-377
- push, 108

R

- read-only, 128
- read-only property, 128
- redirection, 107, 108
- responsive extension, 316-317

S

- Sampler, 119
- Sass, 73, 75
- scroll bars, 223, 241-243, 245
- Scrollable, 243, 245
- scrolling, 651
- SCSS, 75
- Serializable, 121
- server push, 73, 108
- servletInitialized(), 105
- session, 104
 - closing, 107
 - expiration, 107, 108
 - timeout, 107
- session-timeout, 113
- SessionDestroyListener, 105, 107
- SessionInitListener, 105
- setComponentAlignment(), 259-261
- setLocation(), 107
- setNullRepresentation(), 152
- setNullSettingAllowed(), 152
- setVisibleColumns(), 338
- SimpleStringFilter, 342
- Sizeable interface, 130
- SLF4J, 399
- Spreadsheet, 615-626

- Spring, 417-423
 - scopes, 421
- SQL, 74
- state object, 468
- static, 400
- style name, 129
- style name property, 129
- system messages, 108

T

- Table, 173-186, 341
- TestBenchElement, 648
- testing, 652
- Text change events, 152-154
- TextChangeEventMode, 153
- TextChangeListener, 152
- TextField, 150-154
- theme, 73, 287-317
- themeing, 290
- themes, 75
- ThreadLocal pattern, 404-405
- timeout, 107
- tooltips, 124
- TouchKit, 577-614

U

- UI, 106
 - closing, 106
 - expiration, 106
 - heartbeat, 108
 - loading, 105
 - preserving on refresh, 106
- UIProvider, 105
 - custom, 106

V

- Vaadin 6 Migration
 - add-ons, 485-486
- Vaadin CDI Add-on, 411-417
- Vaadin Data Model, 319-343
- Vaadin Designer, 267-285
- Vaadin Spring, 417-423
- VaadinRequest, 105
- VaadinService, 105
- VaadinServlet, 73, 105
- VariableOwner, 121
- VerticalSplitPanel, 246-248
- visible, 129
- visible property, 129

W

- widget, 73

- widget, definition, 448
- widgets, 465
- Window, 119
- windows
 - popup, 358

X

- XML, 75
- XMLHttpRequest, 75
