Community Experience Distilled

TYPO3 Templates

Create and modify templates with TypoScript and TemplaVoila

Jeremy Greenawalt

[PACKT] PUBLISHING open source* community experience distilled

TYPO3 Templates

Create and modify templates with TypoScript and TemplaVoila

Jeremy Greenawalt

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

TYPO3 Templates

First published: November 2010

Production Reference: 1111110

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847198-40-2

www.packtpub.com

Cover Image by Gavin Doremus (gdoremus24@gmail.com)

Credits

Author
Jeremy Greenawalt

Reviewers
Christine Gerpheide
Heike Raudenkolb
Ingo Schmitt

Acquisition Editor
Sarah Cullington

Development Editor
Maitreya Bhakal

Technical Editor
Kavita Iyer

Indexer
Hemangini Bari

Editorial Team Leader
Akshara Aware

Project Team Leader
Lata Basantani

Project Coordinator
Leena Purkait

Proofreaders
Aaron Nash
Lynda Sliwoski

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Jeremy Greenawalt is a full-time developer and part-time writer with close to ten years professional experience in website and application creation. His first love was writing, but programming quickly followed.

He is a co-founder of Vintage 56 where he helps develop websites, online shopping carts, web apps, iPhone/iOS apps, and anything else his friends can think up. Jeremy is also the Web Director of a large ministry, Generals International.

Jeremy lives near Dallas, Texas with his wife, Rebekah, and their ever-youthful puppy, Aingeal. He loves spending time at home reading, playing around on the piano, or just relaxing on the couch with his family.

You can read more from Jeremy at `pocketrevolutionary.com`, and you can follow him on Twitter at `@jgreenawalt`.

Acknowledgment

This book is the product of so many editors, coworkers, and encouragers that I'm surprised my name still gets to be on the cover. I wish I had space to thank everyone individually.

I want to thank the entire Packt team that made this book possible. Thank you, Sarah and Leena, for helping a new author through an intimidating process. A special thanks to the editors and technical editors who helped turn some rambling paragraphs and run-on sentences into a complete book.

Thanks to all of the smart people I work with. Thank you, Mike and Cindy, for always encouraging me. Thank you, Neil, for telling me to write the book when I thought I couldn't. Thank you, Kevin and Bethany, for being my test subjects sometimes. Thanks to all of my coworkers who put up with my pre-deadline breakdowns and always let me vent.

I need to thank Kasper Skårhøj for giving us TYPO3. More importantly, thanks to all the developers, well-known and anonymous, who have made TYPO3 what it is today.

Thank you, Michael Brennen, for giving me my first real programming job. I still blame you for all my Linux or PHP knowledge.

Thanks to Jeff Segars and Ron Hall from the Dallas TYPO3 Users Group for convincing me to use TYPO3 in the first place. Thanks, Ron, for letting me write my final chapter on your framework and being the first to ask for a copy of the book.

Thanks again to my wonderful wife, Rebekah. You will always be my princess.

About the Reviewers

Christine Gerpheide works as a developer and project lead at customedialabs Interactive Media Agency in Larissa, Greece. At customedialabs she helps create enterprise applications and websites, specializing in TYPO3 and mobile application development. She has also presented on TYPO3 at open source conferences in Greece and at the North American TYPO3 Conference. In her free time, Christine likes to experiment with upcoming web technologies, go backpacking, promote renewable energy, and cook Greek food. Christine was born in Washington, D.C. and has a bachelor's degree in Mathematics and Political Science from Grinnell College.

Heike Raudenkolb has been working with websites for more than ten years now. Originally started as a hobby while studying to become a certified translator in the late 1990s, web design and website production soon became her full-time day job during which she has since been building many static as well as dynamic web pages and playing with various CMS and webshop systems.

Today Heike works as a web designer and TYPO3 integrator for an Internet service provider. Her main tasks there are behind-the-scenes TYPO3 integration, customization, and templating—actually, not very different from how it is described here in this book—but she's also doing end user support and customer workshops for editors wanting to work with TYPO3.

Ingo Schmitt, born 1974, studied Electrical Engineering at the "Universität Gesamthochschule Duisburg", Germany, learned Pascal, C++, discovered the Web with Netscape 1.0, and started working for Marketing Factory Consulting GmbH, Düsseldorf in 1996. Working with PHP he developed Web-based applications, releasing his first extranet online shop in 1998. As CTO of Marketing Factory Consulting GmbH he started working with TYPO3 in 2002, including developing his own extensions. Ingo Schmitt is founder and main developer of TYPO3 Commerce, TYPO3 Certified Engineer and "IHK Prüfer für Fachinformatiker". At Marketing Factory he and his team are responsible for the complete development process for web applications and for ongoing maintenance including hosting of the applications. You can follow Ingo Schmitt on twitter at `http://twitter.com/Ischmitt`.

Marketing Factory Consulting GmbH is one of the top TYPO3 agencies in Germany, developing and hosting international multi-language web and portal applications for clients such as Henkel, Ecolab, Ista, Metabo, Wrigley, and Westfalia.

Marketing Factory also runs its own websites such as `heimwerker.de` (the biggest DIY site in the German language), `ratgeber.de`, and online shops like `blumenbutler.de`.

This book is dedicated to my parents, Loren and Kathleen, and my beautiful wife, Rebekah. To my parents, for buying me my first computer and being my most loyal readers; I miss you both. To Rebekah, for loving me and encouraging me every day before and even after the moment I decided to write this; I'll make it up to you somehow.

Table of Contents

Preface

The template systems in TYPO3 make it one of the most powerful content management systems available today, but they seem too complex for many users. Site developers, who are able to learn how to use them efficiently, can build more extensible sites quicker and more customized for their users.

This book is a step-by-step guide for building and customizing templates in TYPO3 using the best solutions available. It takes the readers through one complete example to create a fully functional demonstration site using TypoScript, TemplaVoila, and other core TYPO3 technologies.

This book starts with the basics of creating an example TYPO3 site before showing you how to add your own stylesheets and enhanced JavaScript to the template. You learn about the different types of menus and navigation, and you can try out each one with practical examples in the book. The book shows how to create multiple templates for sections or individual pages in TYPO3 and how you can make a new template completely from scratch for a newsletter. Just as importantly, you learn how to update the editing experience and impress your clients with a custom backend. Finally, you will learn how to specialize for browsers and internationalize your TYPO3 site with simple template updates.

What this book covers

Chapter 1, *Getting Started* provides an introduction to TypoScript and the overall methods of building templates in TYPO3 from the beginning up to the present. The chapter introduces a new extension for building templates, TemplaVoila!, and walks readers through building their first template from HTML to TYPO3 integration.

Chapter 2, *Enhancing Your Template with CSS* covers the different ways of integrating stylesheets into a TYPO3 site and the advantages or disadvantages of each method. The chapter concludes by looking at the built-in classes in TYPO3 that can be used to style individual elements like even and odd rows in a table or a list.

Chapter 3, *Adding Custom Template Fields* gives an introduction to data structures in TemplaVoila templates and walks readers through an example of adding new fields for a banner and a datestamp to the existing template.

Chapter 4, *Creating Flexible Menus* covers the different types of navigation that can be created in TYPO3 templates. The chapter explains the concepts of building menus with TypoScript and walks through complete examples of text-based menus, graphic menus, and special "breadcrumb" menus.

Chapter 5, *Creating Multiple Templates* goes through different examples of templates that can be built based on the existing TemplaVoila template data like templates with sidebars for additional content. The chapter concludes by walking through the process of creating special print-friendly page templates that can be loaded automatically when a visitor wants to print content from a TYPO3 site.

Chapter 6, *Creating a Template from Scratch* walks readers through building a complete template from scratch from the HTML and data structure to a finished template with custom data fields. Up until now, the book has been extending the template that was automatically generated in the first chapter, but this chapter shows how to build a custom template from scratch without the assistance of any wizards or starter templates.

Chapter 7, *Customizing the Backend Editing* covers the different ways to make the TYPO3 backend easier to use for editors by customizing the options available and making the layout and styles match the frontend more closely. It starts by showing how to customize the text editor to match the design and branding of a TYPO3 site more by removing unnecessary options and adding styles that match the overall design. After that, the chapter walks readers through modifying the backend page layouts with columns and headings to mimic the frontend design of a site.

Chapter 8, *Working with Flexible Content Elements* introduces one of the most powerful and useful technologies in TemplaVoila, flexible content elements. Flexible content elements allow developers to create new content types utilizing the power of TemplaVoila for specialized purposes such as displaying contact information or product ads consistently across a site. The chapter explains the main concepts and walks readers through multiple examples.

Chapter 9, *Creating a Mobile Website* covers everything that you need to know to create a mobile version of a TYPO3 site. The chapter talks about detecting mobile browsers and devices using TypoScript, customizing the TypoScript elements for mobile devices, and creating a separate mobile template. The chapter ends by showing how to use TypoScript to automatically redirect to a special mobile site when necessary.

Chapter 10, Going International goes through all the steps to make a TYPO3 site run with multiple languages and localizations and shows why TYPO3 is so ideally suited to international websites. The chapter covers translating content, creating a menu of language options, and building special TemplaVoila templates for different languages.

Chapter 11, Building Websites with the TemplaVoila Framework covers the new TemplaVoila Framework and looks at how it can help developers build sites faster and with less effort by using a repeatable workflow and common tools. Frameworks, cleaner coding, and rapid development are all themes that TYPO3 is moving towards as it develops for the future, and this chapter gives developers a head start by demonstrating the TemplaVoila Framework.

What you need for this book

In order to get the most out of this book, the reader will need some basic web development experience and a text editor for HTML and CSS. None of the examples use difficult HTML or CSS, but a basic understanding of the core concepts is necessary to build proper TYPO3 templates. In addition, readers will need a new TYPO3 installation based on the requirements in *Chapter 1*, *Getting Started* to run the examples in this book.

Who this book is for

If you are a developer, designer, or a site builder who wants to get the most out of TYPO3, whether you are building multiple websites for clients or optimizing their company's site, then this book is for you. It is written for new or experienced users at all levels, but some basic experience with TYPO3 editing and installation is expected.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The first function is `includeCSS`, and this is what it looks like in our TypoScript template setup with multiple CSS files:"

A block of code is set as follows:

```
## Main Menu [Begin]
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO {
   allWrap = <li class="menu-item">|</li>
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO {
   allWrap = <li class="menu-item">|</li>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "You can see the results if we are on the **Products** page ".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started

I was introduced to TYPO3 in 2006, when I started working at a ministry that was using static pages exported out of Dreamweaver. As a lazy programmer who strives for automation, I knew we needed a real content management system. We needed a full solution that was in Open Source, could grow with us, and allowed quick and easy template modifications as our goals and design ideas changed. We looked at a lot of options, but finally settled on TYPO3 after talking to some other developers I respected. They convinced me it was worth the initial effort of learning a whole new CMS (Content Management System) to have the customizability and extensibility that only TYPO3 could offer. I'm happy to say that we rebuilt that site in TYPO3 without ever looking back, but I'm a little embarrassed to admit that it took us years to learn all the possible customizations and extensions that we could use to change how templates work for our frontend users and backend editors.

My goal in this book is to show just how easy it is to build powerful, extensible websites and get you started on that next major website that you're just waiting to build. We're going to step through creating templates, building menus, updating the backend, making mobile sites, going international, and more.

Because TYPO3 has become so powerful and modular, there are often more than a few templating methods that can yield the same results. There are alternative (and possibly better) methods to many of the templating examples in this book, but I have tried to choose the easiest or most powerful techniques in each case to get a site up and running as quickly as possible. If you want to dive deeper into any particular topic after finishing this book though, I recommend continuing to learn through the online documentation at `http://typo3.org/documentation/` and the other TYPO3 books from Packt Publishing `(http://www.packtpub.com/books?keys=typo3&x=0&y=0)`.

This first chapter is going to walk you through setting up a test site to experiment and play with for the rest of the book. Even if you have a site currently running on TYPO3, I recommend setting up a fresh test site as described. With a test site, we can all start with a blank template to build on, and I really don't want anyone breaking a live site when we test different menu systems or internationalization later on.

In this chapter you will:

- Learn a little about the history of creating templates in TYPO3
- Install a powerful TYPO3 extension for templating called **TemplaVoila!**
- Create a working TYPO3 template from ten lines of HTML
- Set up a working page tree with example content and functioning menus

Basic requirements

In order to follow the tutorials in the book, there are a few requirements:

- **Basic HTML/CSS knowledge:** None of the examples use very complex HTML or CSS, but a basic understanding is necessary to build proper TYPO3 templates.
- **Text editor:** You won't require any special development software for this book, but a good text editor is handy for creating HTML templates and writing code. I use TextMate on a Mac, but e-TextEditor (Windows), Notepad++ (Windows), BBedit (Mac), or jEdit (Java) all have extensions for HTML, CSS, JavaScript, and TYPO3's own configuration language, TypoScript.
- **Test server:** We need a place to run our examples during the tutorial. If you already have a hosted server that supports MySQL and PHP 5.2 (with ImageMagick or GraphicsMagick and GDLib/FreeType), you can use that. This may sound like a lot, but most hosting providers offer this by default. Otherwise, you can run TYPO3 on your own computer using either XAMPP (Windows, Linux, or Mac) or MAMP (Mac only). Both of these packages allow you to run a full test server locally on your machine, and they already include a web server (Apache), MySQL, and PHP 5.2 (with the necessary graphics libraries).

- **TYPO3 4.4 or higher with the dummy package installed:** All of the examples in the tutorial have been tested on 4.4.2, but they should work equally well on future versions of TYPO3. You can download the newest version of the TYPO3 source with a dummy package at `http://www.typo3.org/download/`. If you have not installed TYPO3 before, detailed instructions are available in the TYPO3 documentation library (`http://typo3.org/documentation/document-library/extension-manuals/doc_inst_upgr/current/`).

How templates were created

Originally, many TYPO3 templates were built using just TYPO3's own configuration language, **TypoScript**. TypoScript is not a traditional programming language; it is actually a declarative language to configure templates and extensions in TYPO3. For example, the following TypoScript code will output **HELLO WORLD!** through a TYPO3 page. I've added TypoScript comments marked with `##` to explain what is happening:

```
## Create a default page object
page = PAGE
page.typeNum = 0

## Create a text object inside the page
page.10 = TEXT

## Assign some text to the text object
page.10.value = HELLO WORLD!
```

You could add HTML to change the layout of the page, but it wasn't very intuitive. The biggest problem, though, was there was no easy way to integrate a whole HTML template. Designers created HTML files, and then a developer had to translate it all into TypoScript for the final templates.

Next, we started integrating external HTML files directly into the template process. Of course, the first step in creating templates was to build an HTML file to define the basic structure of our pages, and a designer could do this. For example, if we designed a basic template to show a headline and some content, we start with an HTML file like this:

```
<!DOCTYPE HTML>
<html>
   <head>
      <title>Page Title</title>
   </head>
```

```
      <body>
         <h1>
            Header
         </h1>
         <div>
            Content of our new page.
         </div>
      </body>
   </html>
```

Next, we needed to add markers into the HTML to define sections of dynamic content in our HTML. The markers were created as HTML comments within the static HTML document, and we wrapped the sections we wanted to replace with these identifying markers. The designers or the TYPO3 developers do this step. In the following example , we identified the document body, main heading, and a content area:

```
<!DOCTYPE HTML>
<html>
   <head>
      <title>Page Title</title>
   </head>
   <body>
      <!-- ###DOCUMENT_BODY### -->
      <h1>
         <!-- ###HEADING### -->
         Header
         <!-- ###HEADING### -->
      </h1>
      <div>
         <!-- ###CONTENT### -->
         Content of our new page.
         <!-- ###CONTENT### -->
      </div>
      <!-- ###DOCUMENT_BODY### -->
   </body>
</html>
```

Now that the HTML file is ready, we saved it on the `fileadmin/` folder. For this example, we saved the HTML file in a subdirectory, `templates/`, as `basic_template.html`. That means the path to our file would be `fileadmin/templates/basic_template.html`.

The next step was creating a template in TypoScript to map content into our HTML template. We will look more at TypoScript templates when we start building our own example template later in this chapter, so we aren't going to step through the whole process right now. As an example, though, the following TypoScript code would be used to call the HTML file. You can find the sections that we wrapped with markers, and replace the static HTML content.

First, we created a template object and assigned our HTML file to it:

```
temp.mainTemplate = TEMPLATE
temp.mainTemplate {
template = FILE
template.file = fileadmin/templates/basic_template.html
}
```

Next, we created a page object and assigned the main template to it:

```
page = PAGE
page.typeNum = 0
page.10 < temp.mainTemplate
```

After we associated our page with our template, we start working on the HTML between the `DOCUMENT_BODY` markers in our HTML file:

```
page.10.workOnSubpart = DOCUMENT_BODY
```

Now that we were working inside the `DOCUMENT_BODY` markers, we created text objects to go between the `HEADING` and `CONTENT` markers and assign values to them:

```
page.10.marks.HEADING = TEXT
page.10.marks.HEADING.value = This is a heading
page.10.marks.CONTENT = TEXT
page.10.marks.CONTENT.value = This is our content
```

Of course, this isn't actually very dynamic as the content is being coded into the TypoScript template. The next step in the process was to map dynamic content elements in TYPO3 to the HTML through the TypoScript template. This process was more complicated and cannot be covered here, but you can learn more about it in the online tutorial, Modern Template Building (`http://typo3.org/documentation/document-library/tutorials/doc_tut_templselect/current/`).

Introducing TemplaVoila!

As you can see, this made very large HTML templates that had to be manually edited for TYPO3. The markers were one more step for designers to learn if they were moving into TYPO3 development. The TypoScript process also added more training and complexity for new developers who were trying to pick up TYPO3. Luckily for us, Kasper Skårhøj and Robert Lemke went on to create a new template extension called **TemplaVoila!.** This added more flexible page elements to replace the earlier "columns" that defined sections in TYPO3 pages. It also introduced a more intuitive backend editing experience and, more importantly, **Flexible Content Elements**. Flexible Content Elements allow us to reuse custom blocks of HTML, and are a key technology in building websites with TYPO3.

In TemplaVoila, the designers or developers create basic HTML templates without any extra comments or markers. TemplaVoila maps the dynamic content fields, or Content Elements, we want use in our pages to elements in our HTML template using a TemplaVoila Template Object. The first time we set up a site with TemplaVoila, we can run the TemplaVoila Wizard to help us automatically create the new content fields and map them to our HTML template.

Although not required to build all templates in TYPO3, TemplaVoila is highly recommended for its flexibility and more advanced templating techniques such as Flexible Content Elements, backend layout customization, and custom template fields. For that reason, we will be using TemplaVoila for the tutorial examples throughout this book.

Installing TemplaVoila

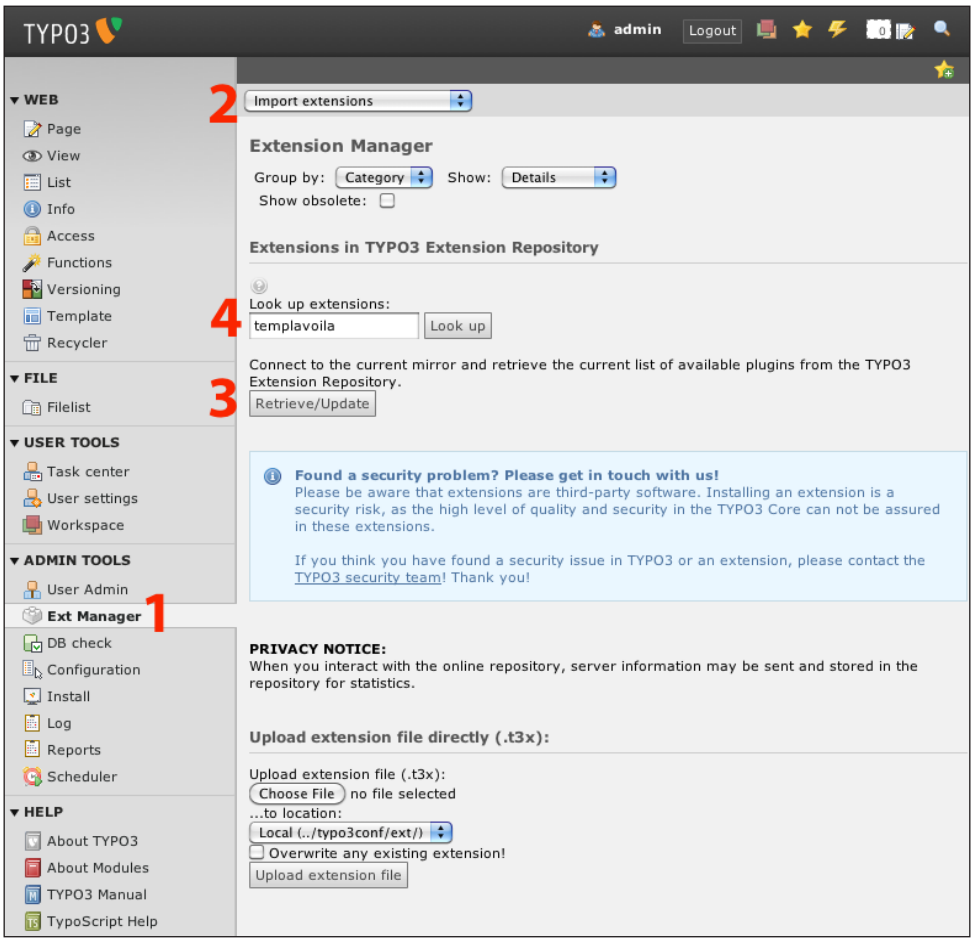
TemplaVoila is a TYPO3 extension, so we can install it through the Extension Manager in the TYPO3 backend.

If you just installed TYPO3 with the dummy package and finished the setup wizard, you can get to the backend of TYPO3 at `http://<your domain>/typo3/` (where `<your domain>` represents your own testing domain). If you are using a server running on your machine with XAMPP or MAMP, then you can probably get to your site at `http://localhost/typo3/`.

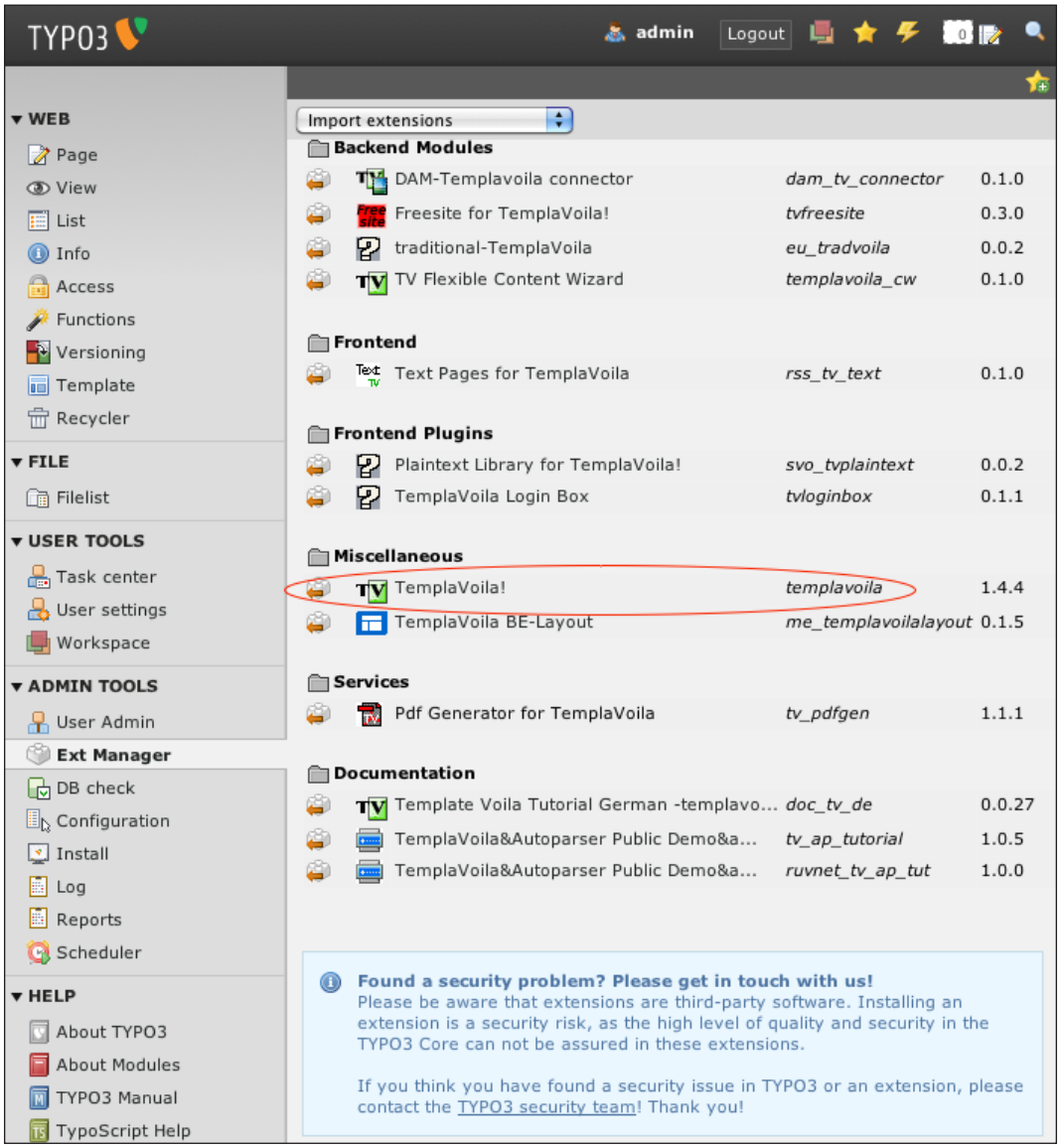
After we've logged in, we can follow these steps to install TemplaVoila:

1. We are going to use the Extension Manager, so go ahead and click on **Ext Manager** in the left frame (marked in the following screenshot with a red 1).

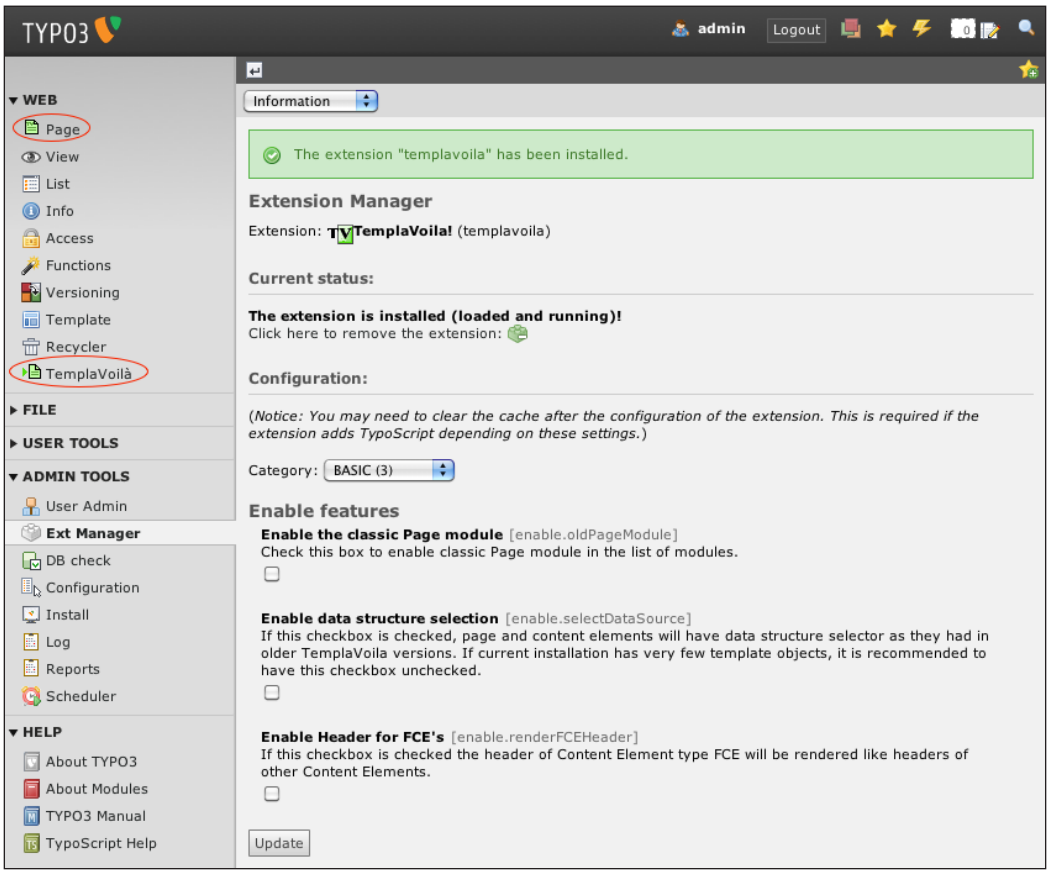
2. Select **Import Extensions** from the drop-down at the top of the main **Extension Manager** frame. After the frame refreshes, we will be able to search for new extensions in the **TYPO3 Extension Repository** (**TER**). The TER is a repository for all of the public extensions or plugins that developers have created for TYPO3.
3. Before we can search for the TemplaVoila plugin, we need to download the most current list of available extensions from the TER. Click on the **Retrieve/ Update** button to start downloading the list. Depending on your Internet connection, it may take a few minutes to download the updates. When it is done downloading, the frame will automatically refresh.
4. Now that we have an updated list, we can search for the TemplaVoila extension. Type **templavoila** into the search box and click on the **Look up** button.



5. By searching for the word "templavoila" in the TER, we have found all of the extensions related to TemplaVoila as well as the official TemplaVoila extension. Scroll down until you find the official TemplaVoila extension (circled in the following screenshot) and click on the **Import** button to the left of the name.



6. Once we click on the **Import button**, the extension is downloaded and imported into our list of available extensions. To install it, click on the **Install extension** button at the bottom of the frame.
7. The Extension Manager will show you a list of the database changes that it needs to make to install TemplaVoila. Click on the **Make updates** button on the bottom of the page.
8. TemplaVoila is now installed. The final page will show us a list of features that we can enable, but none of them are necessary for our example site. Now that TemplaVoila is installed, we can see an updated **Page** icon and a new **TemplaVoila** button in the left frame as shown in the following screenshot:



Creating a basic HTML template

Now that we have TemplaVoila installed, we are almost ready to go through the wizard and create our first TYPO3 pages. First, we need to create a basic HTML template. In other content management systems, you don't often start by creating a template from scratch; normally you would just find an existing one that you liked enough, and then you would edit the CSS and HTML to make it match your vision. TYPO3 allows us much more flexibility and freedom by making it easy to create our own templates from scratch or use the HTML/CSS files from our designers.

Now we're going to create a basic HTML template to get started. As we'll be doing all of our design work in CSS and TYPO3, we just need a "barebones" template that gives us areas to map our content and menus. Normally, we would get the HTML and CSS files at the same time from our web designers, but we are going to wait until the next chapter to integrate the CSS design. Here is the code for our HTML template:

```
<html>
   <head>
      <meta charset="utf-8" />
   </head>
   <body>
      <ul id="menu-area">
         <li class="menu-item"><a href="">Menu Item #1</a></li>
      </ul>
      <ul id="submenu-area">
         <li class="submenu-item"><a href="">Submenu Item #1</a></li>
      </ul>
      <div id="content">This is our content</div>
   </body>
</html>
```

We've already said that TemplaVoila doesn't need any extra markup or comments, but we do have some requirements. As TemplaVoila is mapping TYPO3 content to our HTML elements, all of the HTML code must be completely valid with beginning and end tags. In addition, there are certain kinds of tags that we need:

```
<html>
    <head>
        <meta charset="utf-8" />
    </head>
    <body>
        <ul id="menu-area">
            <li class="menu-item"><a href="">Menu Item #1</a></li>
        </ul>
        <ul id="submenu-area">
            <li class="submenu-item"><a href="">Submenu Item #1</a></li>
        </ul>
        <div id="content">This is our content</div>
    </body>
</html>
```

The root tag

All TemplaVoila template objects must have a "root" tag as an overall container for mapping. This can be any kind of enclosing tag; if we only wanted to use one section of a larger template, for example, we could use a `div` section. For this example, though, we are using the entire body of the HTML template, so we need `body` tags.

The menu area

We also need a container for the menu and the submenu sections, so we have added list tags for both menus with unique ID attributes or `menu-area` and `submenu-area`. These will serve as the containers for our dynamic menus, which will be filled in later through TypoScript.

The content section

Finally, we have our content section that will be used to display the actual TYPO3 page content that we care about. For more complex templates, we would probably have more than one content section, but one container is a good place to start, and is all we need for the wizard.

Now that we have created our basic HTML template, we need to save it where TemplaVoila will find it. In order for TemplaVoila to find our template without problems, we must create a directory named `templates` under the `fileadmin/` directory of your new TYPO3 installation. Once you have created the directory, save a copy of our new HTML template in it as `template.html`. The name is not required to be `template.html`, but throughout this tutorial it will be referred to as such.

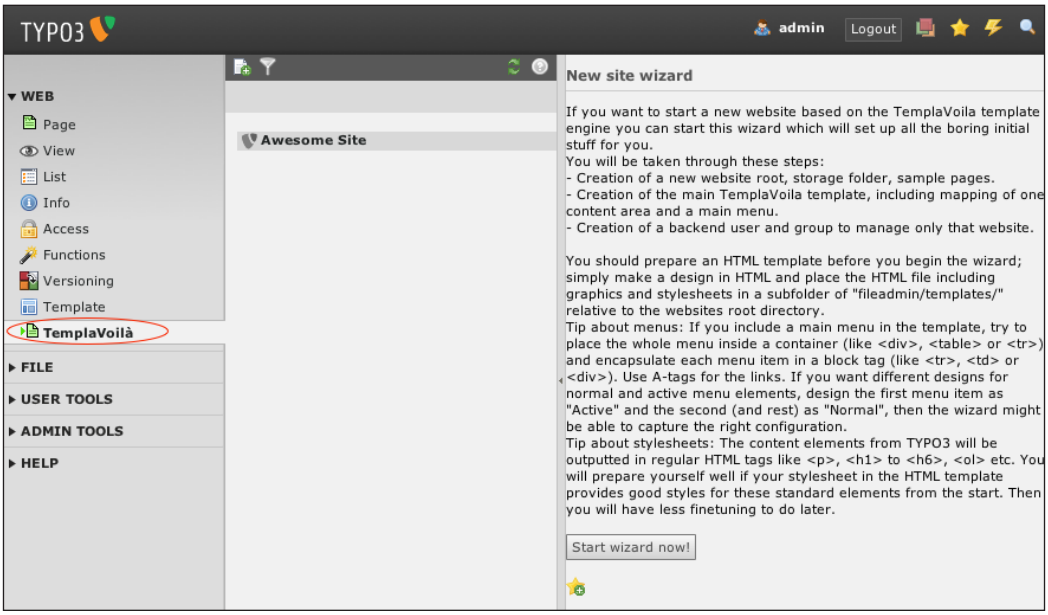
Creating your first template with the TemplaVoila Wizard

Now that TYPO3 is setup, TemplaVoila is installed, and we have an HTML template to start with, we can run the magical TemplaVoila Wizard to create a whole site in just a couple of mouse clicks. I know this section has a lot of pages for a couple of mouse-clicks, but it's important to understand what we're clicking on this for the first time.

If you log in to the backend of TYPO3 right now, you will see we have a website with absolutely no pages. When we go through the wizard, it will perform three tasks:

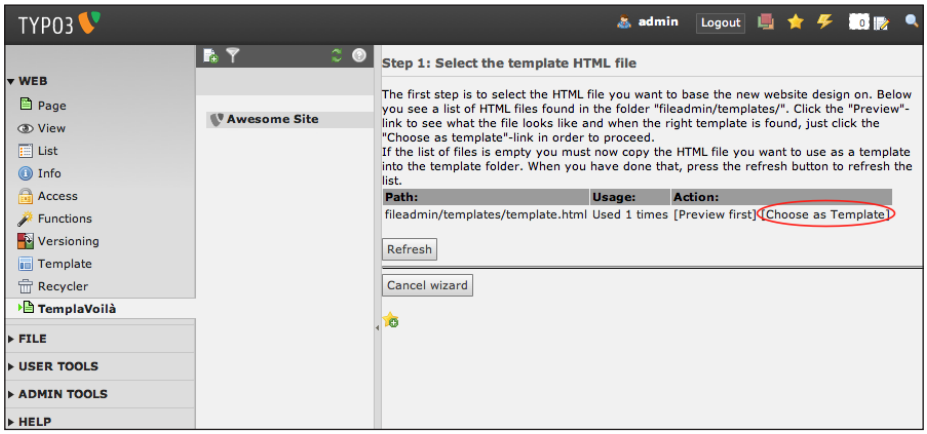
- Map dynamic content areas to our HTML template
- Generate some basic TypoScript for our menus
- Create sample content to test our new template

We are ready to start, so go ahead and choose the TemplaVoila module in the left frame to enter the **New site wizard** as shown in the following screenshot:



Selecting the HTML template

The first thing we need to do is select our HTML template. As we only have one HTML file in the `fileadmin/templates/` directory, the only choice will be our new file, `template.html`. Click on **[Choose as Template]**.



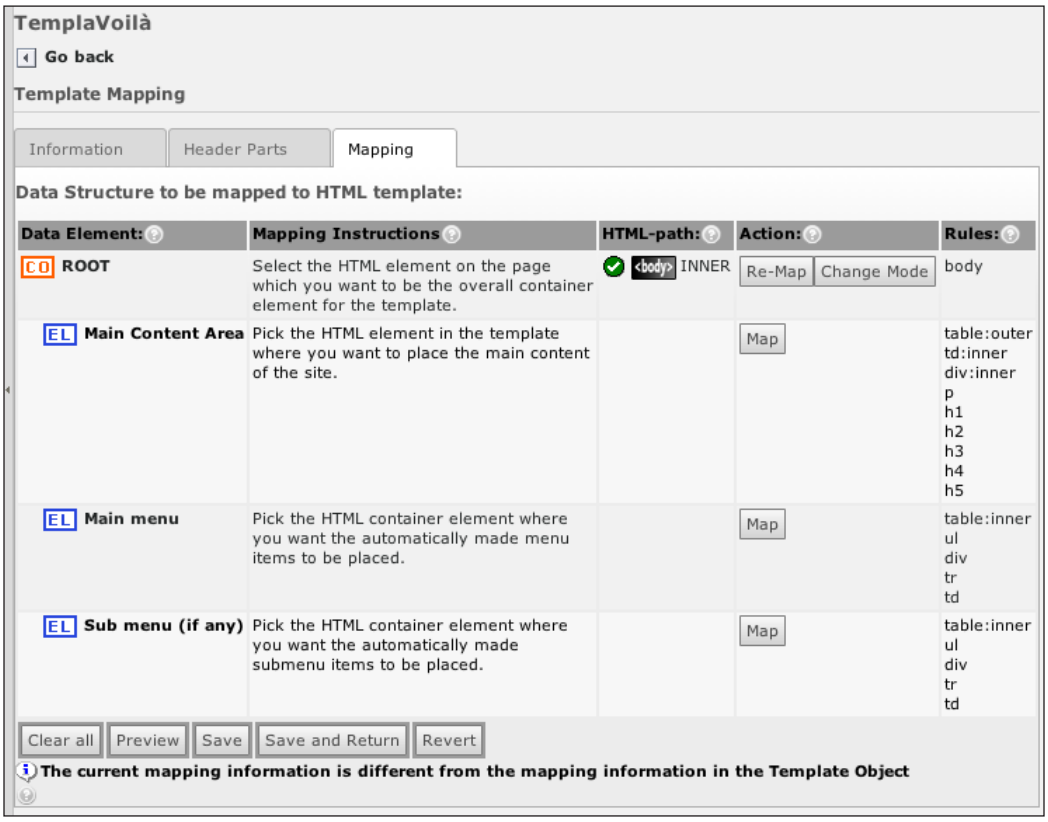
Configuring the new site

Once we click on **Start wizard now!** we are looking at a screen where we can set basic information for our website. The name of our website is required to establish a unique name for our website, but the URL of the website is not required. Running the wizard will create a new editor in the backend, so the username field is required. In the example below, I have named this user **editor**. The values that we fill in here are used to start adding content to the database, set our URL in TypoScript, and create a new user, but all of these settings can be easily changed later if necessary. Once you've filled in required information, click on **Create new site**.

Mapping the template

The screenshot that we just saw is just an introduction screen for the wizard with an animation to explain the mapping process. Click on the **Start the mapping process** button near the bottom of the page.

Now that we are starting the mapping process, we are presented with the default TemplaVoila editing pages. Make sure you are on the **Mapping** tab.



This page is the heart of TemplaVoila mapping. You'll notice that this page has five columns, and understanding them now will introduce you to some of the TemplaVoila concepts, and will save you from frustration later on.

Data elements

Each template field is referred to as a **Data Element** in the mapping section of TemplaVoila. Basically, every template section that will be replaced with TYPO3 data (menus, content elements, and so on) is considered a data element in our template. Each template in TemplaVoila uses a special Data Structure object to define all of the data elements such as **Main Content Area** or **Main menu** that will be available for mapping. When we are creating TemplaVoila templates from scratch, we must define data elements by creating a data structure. We are going to be editing data structures to add our own data elements in *Chapter 3*. This first time, though, we are going to use the data structure from the wizard because it already includes all the data elements we need.

Mapping instructions

The next column, **Mapping Instructions**, is a simple text area for notes in the data structure. The mapping instructions are not binding, but they are used as plain instructions from the creators of the data structure (the TemplaVoila creators, in this case) to the template object mappers (that is you and me, the users of the wizard). We can see, for example, that the default Data Structure includes instructions to map the **Main menu** element to **…the HTML container element where you want the automatically made menu items to be placed.**

HTML-path

Once we have mapped an element, the mapped tag will show up in the **HTML-path** column along with the mode, as we saw with the root data element discussed earlier. The "mode" of a data element refers to inner and outer mapping. If a data element is mapped with **INNER** selected, then the HTML tag from the template will remain untouched, and the TYPO3 content will go inside it. In **OUTER** mode, the tag will be replaced by the TYPO3 content. We don't need to worry about changing the mode yet, but it can be handy in special circumstances when we want TYPO3 to add in its own tag and class information.

Action

The action column simply holds the buttons for mapping, re-mapping, and changing the mode of a data element.

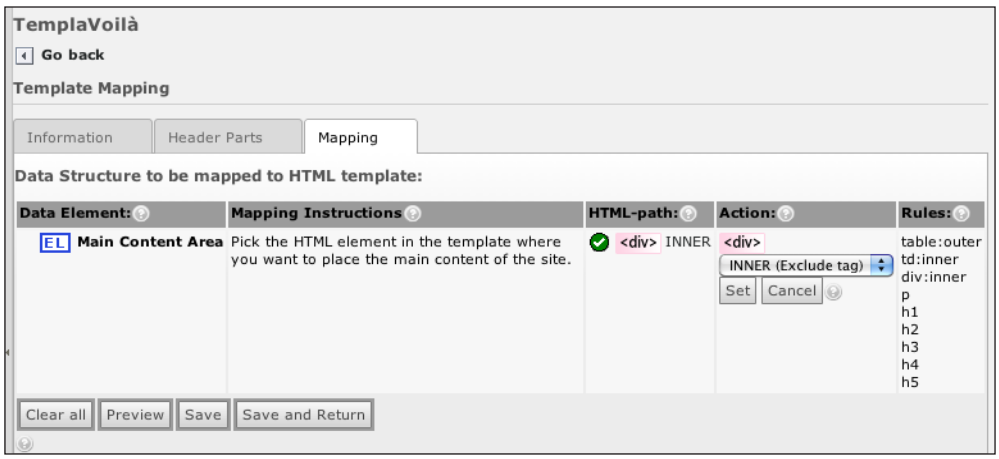
Mapping rules

The rules, as the name implies, are the technical restrictions or allowances that the data template creators have written into the structure. In this first template, the rules will only allow the root data element to be mapped to a body tag, but they allow any of the other data elements to be mapped to almost any container tag. The main content area can even be mapped to paragraph and header tags, if we so desire. As an example, TemplaVoila will not allow us to choose a span tag in the mapping windows for the main content area, because it is not allowed in the set of rules. As developers, we will normally allow mapping to most HTML containers like the example discussed earlier, but it can be helpful to be more restrictive to avoid unintended mappings.

Starting to map

Now that we understand what's possible on the mapping page, we can start mapping the elements. We're going to map the main content area first, so go ahead and click on the **Map** button in the main content area row. This will take us to the mapping screen, and we can choose an editing mode from the **Mapping Window** drop-down. **Mode: Exploded Visual** will show the rendered view of the template with clickable areas for mapping, or we can choose **Mode: HTML Source** to navigate through the HTML source code. For our purposes, we can map in the exploded visual mode. To map the correct content tag, we'll just click on the div button with the **content ID** (circled in the following screenshot). If we hover over the **div** button, we can see the class, ID, and full path so we can identify it better:

After we have chosen the mapping target, the next page prompts us to specify whether we are using **INNER** or **OUTER** mapping before we set the changes in the database. We will choose the default **INNER** mapping from drop-down in the action column because we do not want the div to be replaced with TYPO3 content; we want our TYPO3 content to go inside of the div tag. If we chose **OUTER** mapping, then the generated TYPO3 content would replace the div tags instead of just the content inside the tags. Now we can click on **Set** to save our mapping to the database, and the wizard will take us back to the main mapping section.



We've now successfully mapped our first element! I'll give you a minute to celebrate, because you deserve it. You now officially have all the skills you need to complete the rest of this section, and I'll just be here with a little bit of guidance.

Mapping the rest of our elements

Our next step is mapping the main menu, and it will work exactly as you'd expect. Go ahead and click on the **Map** button in the **Main Menu** row of the mapping page. We are going to choose the tag with the ID **menu-area** for the main menu, and choose the inner mapping mode. Remember to click on **Set** at the end to save your changes. For the **sub menu**, go ahead and repeat the same steps with the submenu element and the list tag with the ID **submenu-area**.

You'll notice that we have five buttons on the bottom of our mapping page that we can use while we are mapping our template:

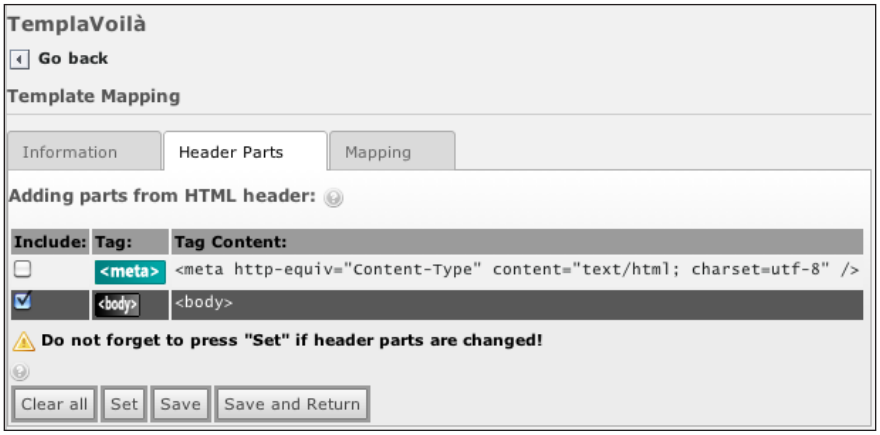
- **Clear all** will clear all of the mappings in the current template. This is helpful if we just need to start over.
- **Preview** will show us a preview of our template as it is currently mapped with unique example content to show the data elements.

- **Save** will save our current mapping.
- **Save and Return** will save our current mapping and return us to the list of templates. If we are running the TemplaVoila Wizard, it will take us to the next step in the wizard process.
- **Revert** will revert all of our changes since the last save.

Now that we have mapped our basic elements, go ahead and click on the **Save and Return** button at the bottom of the page.

Header parts

Once you click on **Save and Return**, we are taken to the next step in the wizard where we are told about mapping the HTML header parts for our template. TemplaVoila allows us to select portions of our HTML template header area to be included in the final TYPO3 output. This is helpful if we want to map CSS or JavaScript directly in the HTML. Any part of the header that we do not choose during this process will be ignored by TYPO3 when it renders our page. Right now, we don't have any special information in our header, so we are going to uncheck all of the boxes and click on **Set**. Once we've clicked on **Set**, TYPO3 will remember our choice, and we can click on **Save and Return** without losing information. If we do not click on **Set**, then TYPO3 will not remember what we checked and unchecked, so it will revert to the default, or what was last set.



Creating the main menu

After we save, the wizard is ready to start creating our menus. TemplaVoila analyzes the HTML template we have mapped to give us good default TypoScript code for our main menu. Just to make sure it is correct, it will show us what it plans to create on the next screen, as shown in the following code snippet. If the wizard has evaluated our template correctly, the default TypoScript should look like this:

```
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO {
   allWrap = <li class="menu-item">|</li>
}
```

If you have worked with TypoScript before, this code will probably be familiar to you as the basic code for all menus when you are first learning. If you don't understand the menu code yet, don't worry. We will learn how to create menus with TypoScript in *Chapter 4*:

```
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO {
   allWrap = <li class="menu-item">|</li>
}
```

Go ahead and click on the **Write main menu TypoScript code** button.

Creating the submenu

Next will be to create the submenus for our site. The TypoScript will be almost exactly the same, except we will be using an object called `subMenu` and we are going to set the entry level to `1`. Here is what the default code should look like:

```
lib.subMenu = HMENU
lib.subMenu.entryLevel = 1
lib.subMenu.wrap = <ul id="submenu-area">|</ul>
lib.subMenu.1 = TMENU
lib.subMenu.1.NO {
  allWrap = <li class="submenu-item">|</li>
}
```

Click on the **Write sub menu TypoScript code** button at the bottom of the page. Once you have chosen to write the code, the next page will only have one final button to click that says **Finish Wizard!**

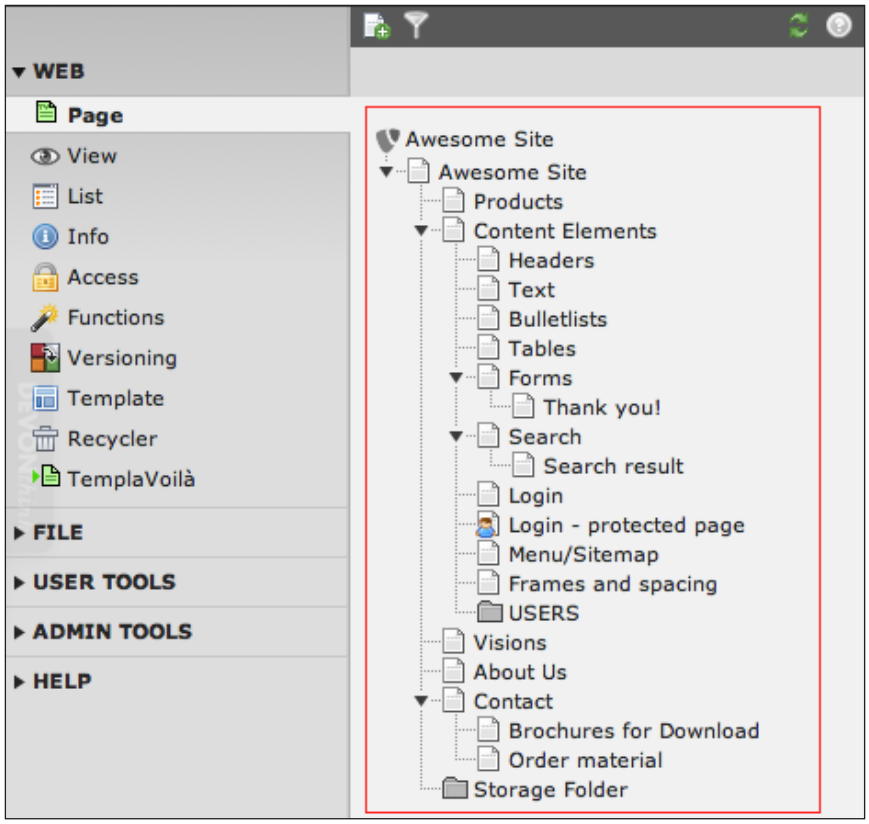
Testing the finished template

After we click that button, it will save all of our changes and load the new TYPO3 site that was created when we started the TempaVoila Wizard. We're done, and we should all be looking at a page that at least somewhat resembles the example shown in the following screenshot:

The page tree

The TemplaVoila Wizard helped us create our first template, but it also generated a TYPO3 page tree full of content. All of the websites and pages in a TYPO3 installation are organized into a single page tree that we can navigate in the middle frame of the backend. Each website has a single **root** page where the main template is created; the root page of our new site is called **Awesome Site**. The rest of the pages in our website are organized beneath the root page. All of the subpages directly

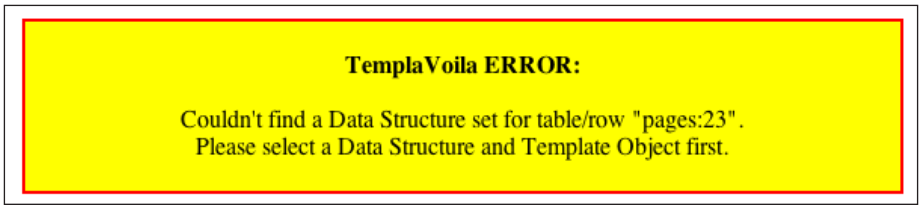
beneath our root page (**Products**, **Content Elements**, **Visions**, and so on) are at level 0 of our page tree, and they will show up in the main menu. As we go deeper in the page tree, the level number increases. The pages under **Content Elements** (**Headers**, **Text**, and so on) are on level 1, and they will show up in our submenu when we are on **Content Elements** or any of its subpages. Any subpages to those pages on level 1 are on level 2, and it can keep branching off as much as we want. This is our page tree:



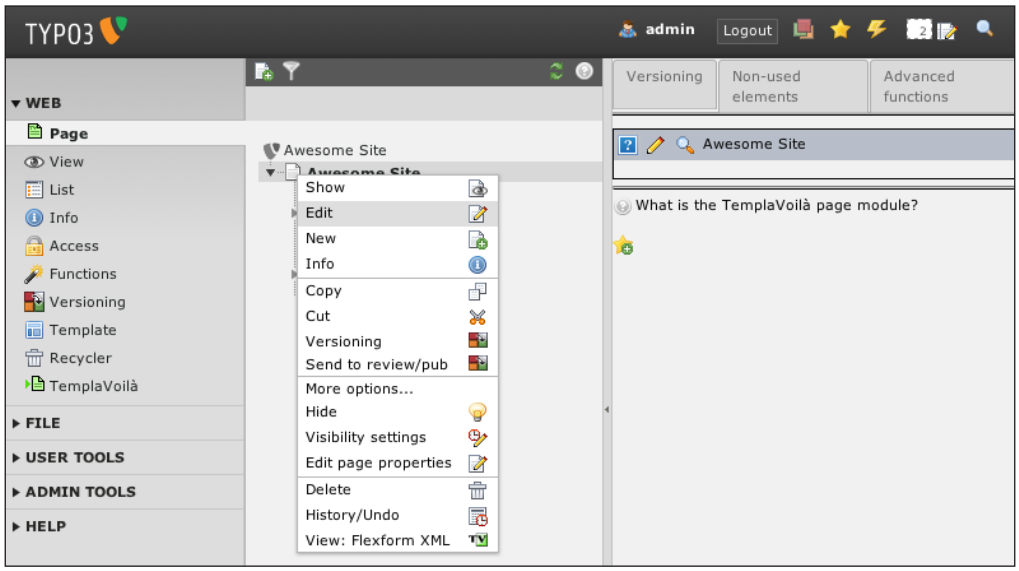
If something didn't work right

Of course, I ran through this example multiple times before writing this, and oddly kept running into a broken page at the end of the wizard. If this happens to you too, don't worry. We are about to fix any problems that the wizard caused right now.

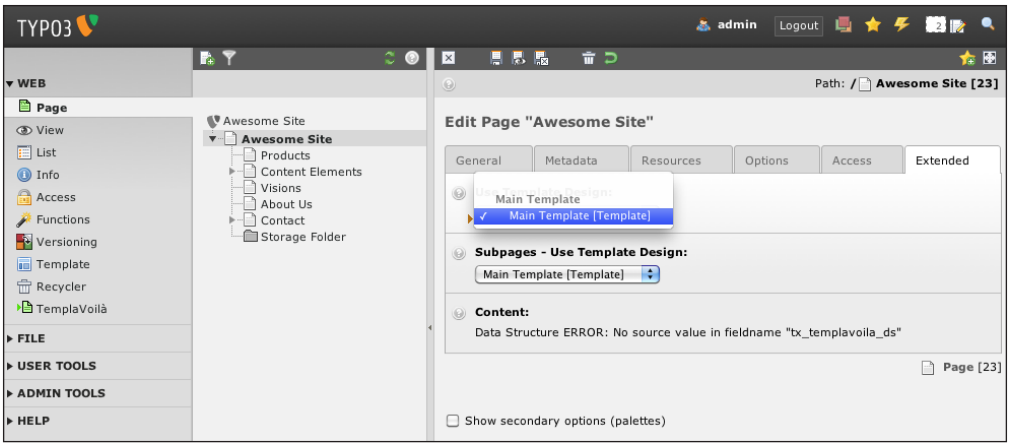
If the wizard did not assign our new template to the main page correctly, you may be confronted with this:



If this happens, it's still an easy problem to fix. This output on the frontend means that the wizard didn't assign our template to the main page. So, in the backend, select **Web | Page** in the far-left menu bar and then, in the page-tree, right-click on our main page and choose **Edit** from the pop-up menu, as shown in the following screenshot:



In the edit form just open up the last tab at the top that is labeled **Extended**. The top section is labeled **Use Template Design:** and we are going to choose **Main Template**, which is the template we just mapped, from the drop-down list. Once you choose the template, click on the disk icon to save your update.

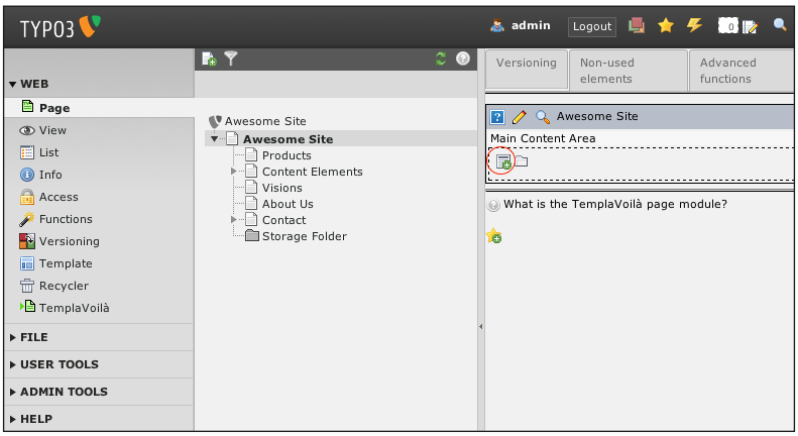


Adding content to our front page

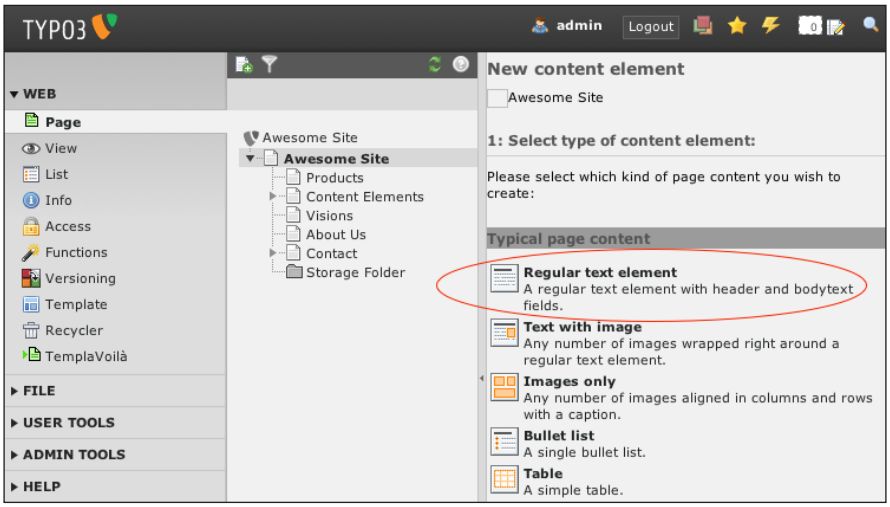
Now that we have our template, we can add some content to our front page. If the wizard already added example content, then you can skip the rest of this section.

As editors, we will use the TemplaVoila Page view to add elements that are mapped in the current template. TypoScript was used to create the main menu and sub menus, so they are not visible in the **Page** view. The only mapped data element that we can currently add content to in our template is the Main Content Area, but we will add more content areas in later chapters that will show up in the Page view.

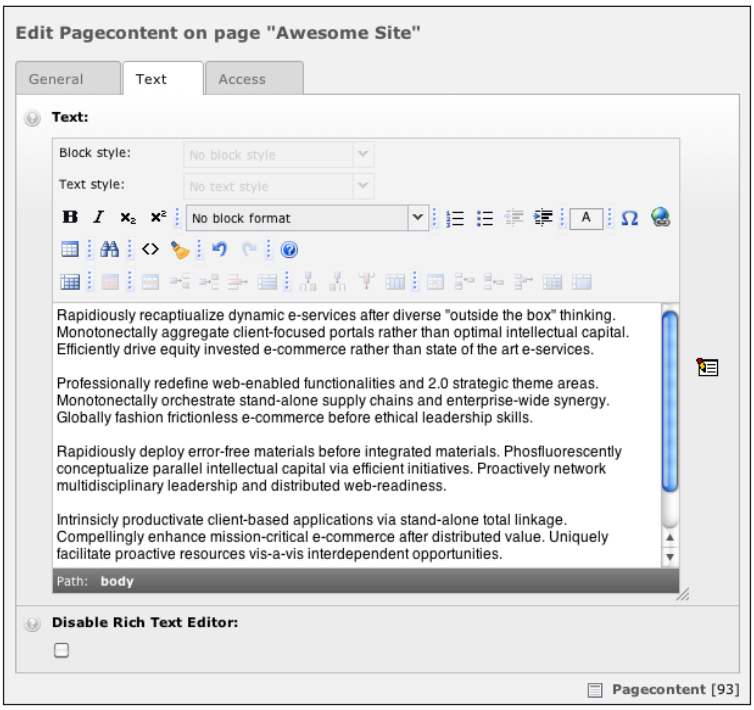
1. To add a content element, select **Web | Page** button in the far-left menu bar and then select (that is click on the page title of) the main page for our site in the page tree. You will see an edit form similar to the following screenshot.
2. Click on the Create new element button in the **Main Content Area**.



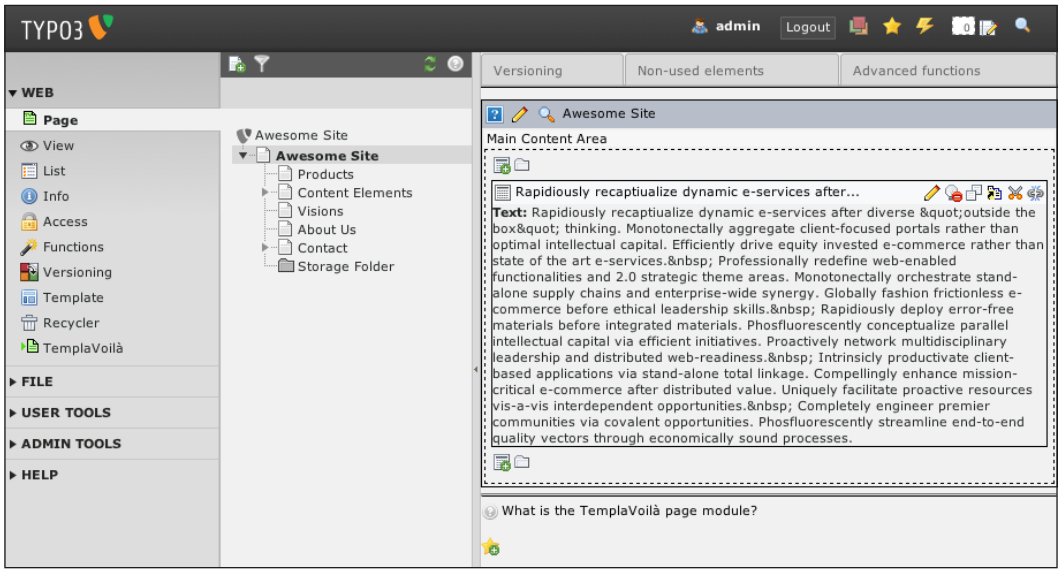
3. Next, select **Regular text element** as the type of content that we want to create. If you would like to include an image, select **Text with image**.



4. Use the edit form to add a header and some content to your new content element and save your changes.



After you've saved your new content element, you can see it in the TemplaVoila Page view and the frontend of your TYPO3 site:



Summary

Congratulations, you have a site! Okay, I know it's ugly right now, but we'll make it pretty soon enough. Right now, you just need to appreciate how far along you are. After learning about the history of TYPO3 templates, we have already created our first template in TemplaVoila and started adding content to our pages.

Our site has nothing to show our clients or bosses yet, but we can finally move on to the reason you bought this book: modifying templates. So, go ahead and get some coffee and congratulate yourself a little more, but hurry back. In the next chapter, we're going to start using CSS to update the look of our templates. We'll look at how we can dynamically call our stylesheets using TypoScript. Then, we can look at ways we can use our template and TYPO3's built-in markup to style our menus and specific sections of content.

2
Enhancing your Template with CSS

Okay, now that we have a basic site, we can lean back, look at our screens, and admit that we don't have anything to brag to our clients about yet. You paid for a book about customizing your templates, and so far we've just created an ugly dummy site with no customizations. Now that we've moved through the initial setup and have a running site, we can integrate our cascading stylesheets (CSS) into our website. More importantly, we are going to cover some of the CSS techniques that are specific to TYPO3. We're going to learn how to add our stylesheets in a number of different ways and look at the default styles in TYPO3 we can customize.

In this chapter you will:

- Learn all four ways to add a stylesheet in TYPO3 and their advantages and disadvantages
- Learn about the extra markup TYPO3 creates to help CSS developers and how to remove it

Creating a basic stylesheet

Most of the time we will be given the HTML and CSS files for a website at the same time, so we can just place them both in the `fileadmin/templates/` directory before we run the TemplaVoila Wizard. To avoid information overload, we are handling the HTML template and CSS as two different steps. We've already added the HTML template to TYPO3, so now we just need to add our CSS files.

We are going to be adding more HTML template files to our site later in this tutorial, so we will create a `css/` directory inside the `fileadmin/templates/` directory just to keep our files organized. If you have CSS files ready, copy them to the `fileadmin/templates/css/` directory now. You can use any filenames that you like, but for the rest of this tutorial the main stylesheet will be referred to as `style.css`. If you don't have a CSS file ready, you can look at my file below which includes some browser reset rules from Eric Meyer (`http://meyerweb.com/eric/tools/css/reset/`), basic styling, and code for cleaner menus.

The first thing that I did was reset the default browser styling using some of Eric Meyer's CSS:

```
/* @group Reset Styling */

html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, font, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td {
   margin: 0;
   padding: 0;
   border: 0;
   outline: 0;
   font-size: 100%;
   vertical-align: baseline;
   background: transparent;
}

blockquote, q {
   quotes: none;
}

blockquote:before, blockquote:after,
q:before, q:after {
   content: '';
   content: none;
}

:focus {
   outline: 0;
```

```
}

ins {
   text-decoration: none;
}

del {
   text-decoration: line-through;
}

table {
   border-collapse: collapse;
   border-spacing: 0;
}

/* @end Reset Styling */
```

Next, we need to include some base styling for the main HTML tags:

```
/* @group Base Styling */

body {
font-family: "Helvetica Neue", Arial, Helvetica, Geneva, sans-serif;
font-size: 16px;
line-height: 18px;
margin: 20px;
}
p, ul, div {
color: #333;
font-size: 16px;
   line-height: 18px;
}
h1 {
   font-size: 30px;
   line-height: 36px;
   margin-bottom: 18px;
   font-weight: 200;
   font-variant: small-caps;
}
h2 {
   margin-bottom: 24px;
   line-height: 30px;
   font-size: 18px;
}
h3 {
```

```
    font-size: 20px;
    line-height: 24px;
}
h4, h5, h6 {
    font-size: 18px;
    line-height: 24px;
}
ul, ol {
    margin: 0px 0px 18px 18px;
}
ul {
    list-style-type: circle;
}
ol {
    list-style: decimal;
}
td {
    padding: 5px;
}
:link, :visited {
    font-weight: bold;
    text-decoration: none;
    color: #036;
}

/* @end Base Styling */
```

Finally, my CSS file has some basic menu styling to clean up the navigation at the top of our page:

```
/* @group Menu Styling */

ul#menu-area li, ul#submenu-area li {
  list-style-type: none;
  display: inline;
   margin-right: 20px;
}

ul#menu-area {
    border-bottom: 2px solid #666;
    margin-bottom: 2px;
}

ul#submenu-area {
    margin: 0px 0px 20px 50px;
```

```
}

li.menu-item a {
   font-size: 24px;
   line-height: 24px
}

li.menu-item a, li.submenu-item a {
   color: #666;
   font-weight: normal;
   font-variant: small-caps;
}

/* @end Menu Styling */
```

Including stylesheets in TYPO3

There are almost half a dozen radically different methods to do anything in TYPO3, and CSS is no exception. TYPO3 has exactly four documented methods for including stylesheet links in templates:

- Adding headers from the HTML template during the TemplaVoila Wizard
- `page.stylesheet`
- `page.includeCSS`
- `Page.headerData`

Now, before you get too worried, let me ask you to go ahead and set your mind at ease. We are only going to end up using one method for our main template. In fact, I will cut the suspense and tell you right now that, like all good step-by-step books, we're going to use the fourth and final option. However, it's still important that you are aware of how the others work for a few reasons.

First, we'll hear all about these methods in TYPO3 documentation, forums, and other discussions. If we don't have at least a basic understanding of all four functions, each one will seem like a brand new technique when somebody mentions them online or at a conference. We won't know why we're using `page.headerData` in our code, and we'll have to spend time trying each one out as it comes up.

Second, learning the stylesheet methods is a step-by-step process. Each of these methods is successively more powerful or cleaner, and we'll appreciate `headerData` so much more after we see how all of the methods work.

We have different needs for all of our templates and websites. Just because including the CSS in our TemplaVoila headers is not the most dynamic approach for our main template, doesn't mean that we don't want to use it for a subtemplate or specialized section of one of our websites.

Finally, some of this is subjective. Just because we're going to use the headerData function for the rest of this book doesn't mean you won't decide later on that I was wrong. If you decide to use `page.includeCSS` for all of your sites one day, this will still be a good reference.

What you need for your main stylesheet

Before we look at the four options for including CSS, we need to define what we need when we include the main stylesheet for our website:

- **Multiple CSS files**: We need to be able to include multiple CSS files in the same page. At the very least, we will need the ability to include our main stylesheet and a print stylesheet.
- **Conditional CSS:** We will need to be able to target browsers or mobile devices with specific CSS. If you have ever made a complex design work in multiple browsers, you know that we sometimes need to create a special stylesheet for Internet Explorer or Opera.
- **Order:** When we are using multiple CSS files, it is very important for us to be able to define the order in which they are included because it is common that one file may override some styles in another file. Managing these overrides is an essential part of CSS development.
- **Extensibility:** We need the ability to include a different stylesheet on a page or section of pages without remapping our template or creating a new HTML file. If we want to change the look of one page, we don't have to go through all the steps of creating a new TemplaVoila template.
- **Stability:** We need a solution that will be stable for the rest of our site. This means that it obviously can't be fragile enough to break on its own, but it also means that it shouldn't add any extra steps to the normal maintenance of our site that makes it more prone to breaking.

Adding CSS with the TemplaVoila Wizard

When we mapped our template in TemplaVoila, we were given the option to include any headers that we required while including stylesheet and JavaScript declarations. Including the CSS in our TemplaVoila mapping can be a quick way to include our CSS by using the headers from our HTML template, but it does not fulfill our needs.

We can include multiple CSS files and set their order in the HTML header, but here are some disadvantages:

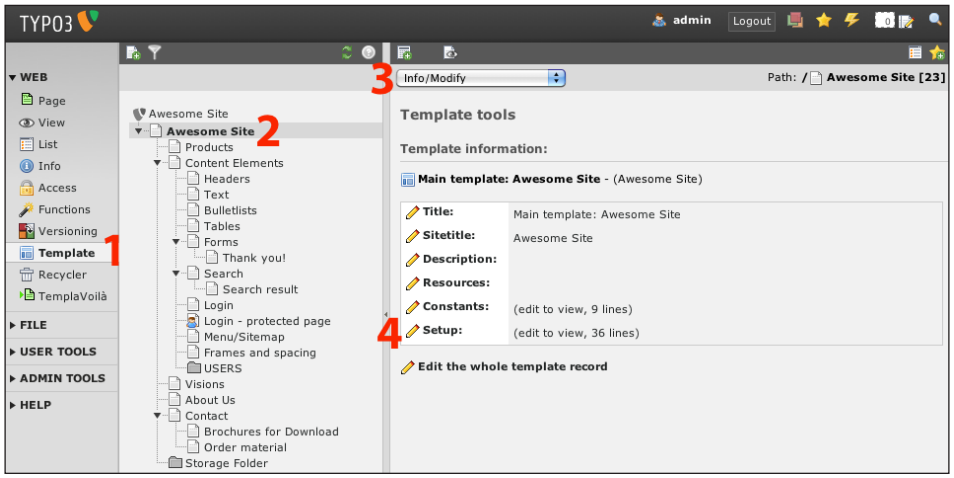
- We don't have very powerful options to conditionally include CSS for specific browsers.
- It's not extensible because we can't include a different CSS file on one page without mapping a new TemplaVoila template.
- Finally, it is our least stable option because we have added a new step to the mapping process anytime that a TemplaVoila template is created or edited. We will need to update the mapping of our TemplaVoila template at some point, and we don't want to lose all of our styling because we forgot to check one box in the wizard. I did this when we were first using TYPO3 on a live site, and trying to figure out why half my site was completely broken was one of the most panicked moments I've had as developer.

Including CSS with page.stylesheet

The second method is using the TypoScript object `page.stylesheet` to dynamically include one CSS file in our TypoScript template. Our TypoScript template can use the basic conditional statements and logic of TypoScript to give us more flexible and dynamic templating functions. We were introduced to the TypoScript template in the first chapter when we looked at the TypoScript code that the TemplaVoila Wizard was creating for our menus, but now we're going to start editing it ourselves without a wizard. Every site that you create in TYPO3 has a TypoScript template, and it is always on the root page of your page tree. In the last chapter we looked at the page tree a little, so you'll remember that the page **Awesome Site** is our current root page. Just like our TemplaVoila template defines the layout of HTML and content elements on our page, our TypoScript template will configure the dynamic output including menus, other languages, and more for the root page and all pages underneath it. Right now, we're going to use the TypoScript template to include our CSS file:

1. To update the TypoScript template, click the **Web | Template** button in the far-left menu bar.
2. Select the root page of our site, **Awesome Site**, in our page tree.
3. Select **Info/Modify** from the drop-down list at the top of the **Template tools** frame.

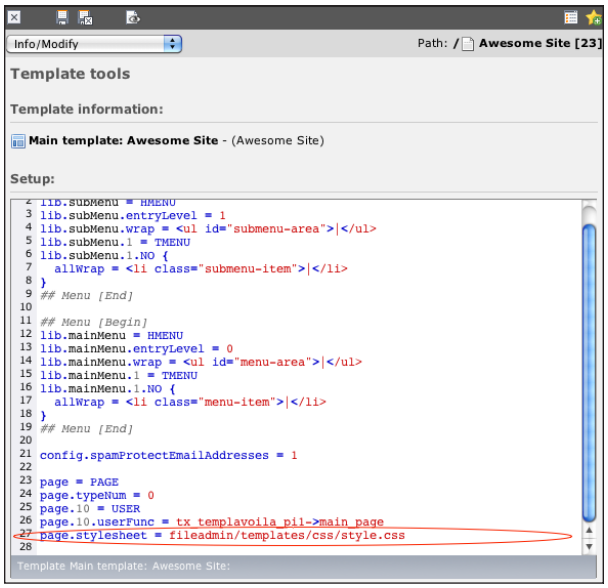
4. We are going to include `page.stylesheet` in the TypoScript template setup, so click on the edit icon next to the **Setup** label.



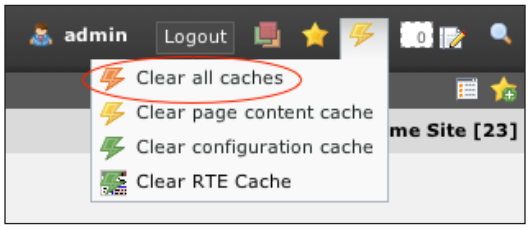
5. Scroll down in our TypoScript editor past the code that was generated by the TemplaVoila Wizard to where the page is defined, and add the following line to our TypoScript page definition to include our CSS:

```
page.stylesheet = fileadmin/templates/css/style.css
```

6. Once we've added our TypoScript line as shown in the following screenshot, click on the save icon at the top of the frame:



To see our changes on the frontend, we need to clear the TYPO3 cache. By default, TYPO3 uses intelligent caching to save our template information so it doesn't have to be rebuilt for every visitor. If we make a change to the template that might have been cached, we will have to clear the cache and let TYPO3 cache the updated version of our template. To clear the cache, click on the **Clear cache** menu in the top-right corner of the backend window, and select **Clear all caches** as shown in the following screenshot:



This is more dynamic than mapping our CSS as part of the TemplaVoila Wizard, but it has a drawback: it only works for one stylesheet. This code can be used to declare exactly one stylesheet for the TypoScript template; we can overwrite it later with another stylesheet, but we can't call two consecutive stylesheets with this method. We already decided that we need the ability to include multiple CSS files for print and browser-compatibility stylesheets, so we can't use `page.stylesheet`.

Including CSS with page.includeCSS

Now that we've discussed and eliminated the methods that are objectively not correct choices for our main template, we can look at two equivalent methods that are both powerful enough to be used to include our stylesheets in the main template. The first function is `includeCSS`, and this is what it looks like in our TypoScript template setup with multiple CSS files:

```
page.includeCSS {
    file1 = fileadmin/templates/css/style.css
    file1.media = screen
    file2 = fileadmin/templates/css/print.css
    file2.media = print
}
```

As you can see, this allows us to declare multiple stylesheets in our TypoScript, and they are included in the exact order that they are listed. Additionally, we can update the values or add a CSS file easily on a specific page or section of pages by using a **TypoScript extension template**.

A TypoScript extension template can be created on any page, and it will include all of the values of the TypoScript templates above it in the page tree followed by any specific configuration in the extension template. For example, we could create a TypoScript extension template on a page with the following lines in the setup to include another CSS file to that page and the pages below it in the page tree:

```
page.includeCSS {
    file3 = fileadmin/templates/css/style_special.css
    file3.media = screen
}
```

Unfortunately, `includeCSS` does not fulfill one of our requirements for a good solution: conditional CSS. Because we don't have direct control over the HTML headers that are being generated, we cannot use conditional comments like `<!--[if IE]>` to target specific browsers.

An advantage of the `includeCSS` function is that it implements the `link` `rel` and `type` attributes completely in TypoScript, and this may be easier for developers who haven't already learned how to declare stylesheets in normal HTML headers. Readers of this book are smart developers, so we're comfortable trading a little convenience of not typing full HTML headers for the control conditional CSS.

Including CSS with page.headerData

The final method we are going to look at is the `headerData` function in TypoScript, and this is what it looks like when we include multiple CSS files:

```
page {
    headerData.10 = TEXT
    headerData.10.value = <link rel="stylesheet" type="text/css"
                           href="fileadmin/templates/css/style.css" />
    headerData.20 = TEXT
    headerData.20.value = <link rel="stylesheet" type="text/css"
                      href="fileadmin/templates/css/more_style.css" />
}
```

The `headerData` function in TypoScript adds arbitrary headers directly to the template when it is output in the TYPO3 frontend. This works almost exactly like the `includeCSS` function, but it has two advantages: flexibility and conditional CSS. Because the data we assign is completely arbitrary, we can use the same function to declare stylesheet links, JavaScript file links, and even hardcoded CSS and JavaScript. This is like learning a suite of functions for the price of one! We can also take advantage of this flexibility to use standard header logic from an advanced template without translating it into TypoScript:

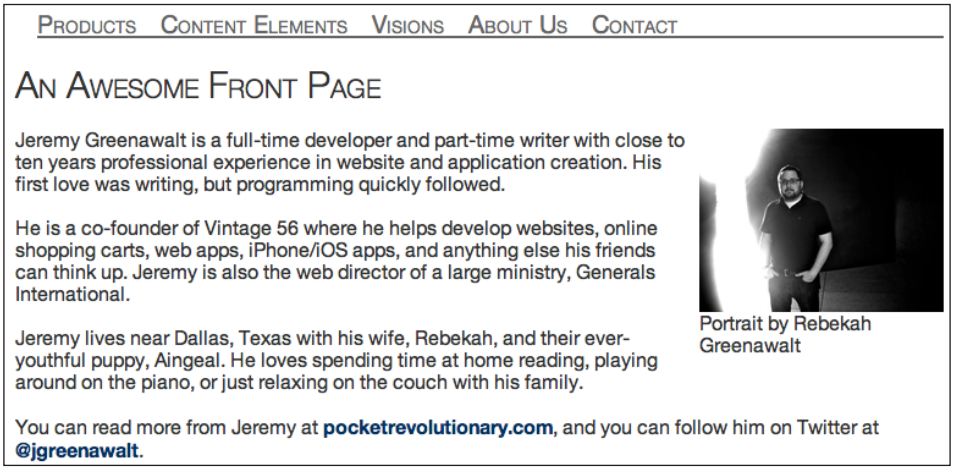
```
//Include IE Hacks CSS
page.headerData.30 =TEXT
page.headerData.30.value (
    <!--[if IE]>
          <style type="text/css" media="screen">
                @import "fileadmin/templates/css/common_ie.css";
          </style>
    <![endif]-->
)
```

Personally, I also prefer the way that `headerData` handles the ordering of the CSS files. By using the numeric values `10` and `20` in the declaration (`headerData.10`, `headerData.20`, and so on), we can guarantee the order that the links will appear in the header in the main TypoScript template and any extension templates; TYPO3 will always add them to the template in ascending order.

Go ahead and copy this code into the main TypoScript template setup of our example site and replace any of the other functions we tried:

```
page {
    headerData.10 = TEXT
    headerData.10.value = <link rel="stylesheet" type="text/css"
   href="fileadmin/templates/css/style.css" />
}
```

If you need help editing the TypoScript template, you can look at the steps at the beginning of this section again. We will be editing the TypoScript template setup a lot, so you will be very comfortable getting in and out of the Template tools pages soon enough. Remember to clear the cache after you've made the changes. Our frontend site should now look something like this:



Using default markup in TYPO3

By default, TYPO3 includes extra markup in all of the content elements it outputs including extra `div` wrappers, classes, and HTML identifiers. A standard content element from our website looks like this:

```
<!--  CONTENT ELEMENT, uid:93/textpic [begin] -->
      <div id="c93" class="csc-default" >
      <!--  Header: [begin] -->
         <div class="csc-header csc-header-n1"><h1 class="csc-
firstHeader">An Awesome Front Page</h1></div>
      <!--  Header: [end] -->

      <!--  Image block: [begin] -->
         <div class="csc-textpic csc-textpic-intext-right"><div
class="csc-textpic-imagewrap"><dl class="csc-textpic-image csc-
textpic-firstcol csc-textpic-lastcol" style="width:200px;"><dt><img
src="uploads/pics/portrait.png" width="200" height="150" border="0"
alt="" /></dt><dd class="csc-textpic-caption">Portrait by Rebekah
Greenawalt</dd></dl></div><div class="csc-textpic-text">
      <!--  Text: [begin] -->
         <p class="bodytext">Jeremy Greenawalt is a full-time
developer and part-time writer with close to ten years professional
experience in website and application creation. His first love was
writing, but programming quickly followed.
</p>
      <!--  Text: [end] -->
         </div></div><div class="csc-textpic-clear"><!-- --></div>
      <!--  Image block: [end] -->
         </div>
   <!--  CONTENT ELEMENT, uid:93/textpic [end] -->
```

In this example straight from our front page, there is more markup than actual content. This looks messy, confusing, and is one of the first things that new developers criticize about TYPO3. If we actually look at the code, we start to see how helpful this markup can be.

Look in the code above, and you'll start to notice that TYPO3 has generated classes and `div` tags according to the type of backend content element and the different layout options that were chosen in the backend. We can see the first header tag on our page is given the `.csc-firstHeader` class. We used a text /image content element with the image inset on the right, so the entire content element is wrapped with a new `.csc-textpic-intext-right` `div` tag. We included a caption for our image as well, and the caption has the class `.csc-textpic-caption`. Using this automatically-generated markup, we can override and style specific kinds of content using only the CSS stylesheet and TYPO3's own rendering workflow.

TYPO3 assigns the `.bodytext` class to all paragraph tags in text elements. If we wanted to change the default class for all paragraphs, though, we could add the following code to the bottom of our main template to change the default class to `.awesome-class`:

```
lib.parseFunc_RTE.nonTypoTagStdWrap.encapsLines.
addAttributes.P.class = awesome-class
```

As the default class is assigned to all paragraphs within the scope of our template, we can use this code to assign different classes in certain sections (with extension templates) or differentiate between regular paragraph styles and the output of extensions or non-TYPO3 pages.

Headers

TYPO3 wraps headers in a `div` tag with the `.csc-header` and a subclass, `.csc-header-nx`, that increments `x` for every header of any type, h1-h6, in a dynamic content area:

```
<div class="csc-header csc-header-n1">
   <h1 class="csc-firstHeader">Main header on page</h1>
</div>
<div class="csc-header csc-header-n2"><h2>Subheader</h2></div>
<div class="csc-header csc-header-n3"><h2>Another
                                          Subheader</h2>
</div>
```

We can also see in the example we just saw that TYPO3 adds the class `.csc-firstHeader` to the first header in any dynamic area. We can use the `.csc-firstHeader` class to emphasize the first header of any type, h1-h6, in our main content area like a headline in a newspaper by styling `div#content .csc-firstHeader` in our CSS.

Image with text areas

TYPO3 controls the layout of our image and text areas using multiple classes and subclasses:

```
<div class="csc-textpic csc-textpic-left csc-textpic-above">
   <div class="csc-textpic-imagewrap">
      <dl class="csc-textpic-image csc-textpic-firstcol csc-textpic-
                lastcol" style="width:260px;">
         <dt><img src="uploads/pics/a4_08.jpg" width="260"
               height="260" border="0" alt=""></dt>
         <dd class="csc-textpic-caption">Rapidiously leverage other's
```

```
excellent technology and cutting-edge ROI. Dramatically e-enable
functionalized applications for out-of-the-box markets.</dd>
      </dl>
   </div>
</div>
```

The CSS Styled Content TYPO3 extension, which is enabled by default, includes a default stylesheet for all of these classes, but we can still override them in our own CSS. For images that are above or below the text (like the code above), TYPO3 uses two basic classes for positioning. The first class declared assigns the horizontal positioning (`.csc-textpic-center`, `.csc-textpic-right`, or `.csc-textpic-left`), and the second class is for the image's vertical positioning related to the text (`.csc-textpic-above` or `.csc-textpic-below`). If the image is supposed to be in the text like the code below, then the horizontal and vertical positioning classes are replaced by one of four classes for layout: `.csc-textpic-intext-right`, `.csc-textpic-intext-left`, `.csc-textpic-intext-right-nowrap`, or `.csc-textpic-intext-left-nowrap`. This is an example of the TYPO3 output for an image on the right side with text wrapped around it:

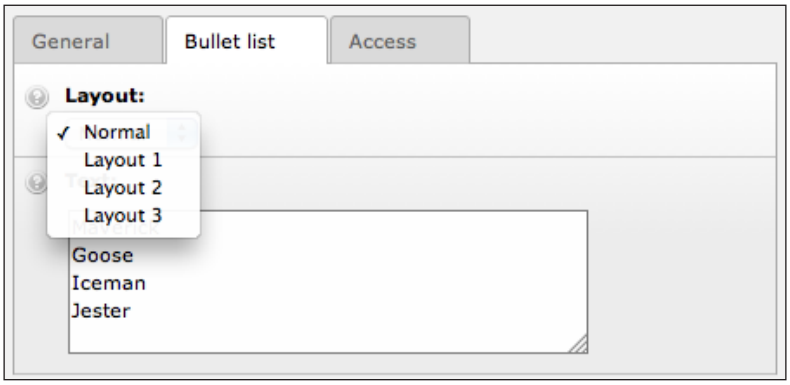
```
<div class="csc-textpic csc-textpic-intext-right">
   <div class="csc-textpic-imagewrap">
      <dl class="csc-textpic-image csc-textpic-firstcol csc-textpic-
lastcol" style="width:260px;">
         <dt><img src="uploads/pics/a4_08.jpg" width="260"
height="260" border="0" alt="" /></dt>
         <dd class="csc-textpic-caption">Rapidiously leverage other's
excellent technology and cutting-edge ROI. Dramatically e-enable
functionalized applications for out-of-the-box markets.</dd>
      </dl>
   </div>
</div>
```

Another class that we see in the examples above is the `.csc-textpic-caption` class. All text captions for images are wrapped in this class, so we can easily address it in our CSS. If we wanted to make caption text smaller, for example, we could add this to our stylesheet:

```
dd.csc-textpic-caption {
   font-size: 70%;
}
```

Bullet lists

TYPO3 allows editors to choose from four different layouts for any ordered or unordered bullet list in the backend:



Depending on the layout chosen in the backend, TYPO3 will add the CSS class `.csc-bulletlist-0`, `.csc-bulletlist-1`, `.csc-bulletlist-2`, or `.csc-bulletlist-3` to the HTML output:

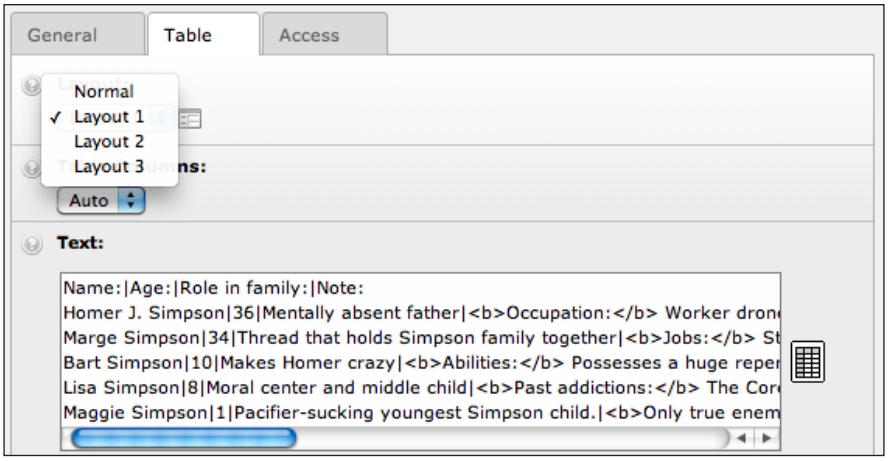
```
<ul class="csc-bulletlist csc-bulletlist-1">
   <li class="odd">Maverick</li>
   <li class="even">Goose</li>
   <li class="odd">Iceman</li>
   <li class="even">Jester</li>
</ul>
```

There is no CSS to style these classes in the default TYPO3 stylesheets, but we can use CSS to target each of these classes with our own styling.

As you can see in the code above, TYPO3 also automatically adds the classes `.even` and `.odd` to the bullet list items so we can use alternating styles to keep long lists readable or even create a two-column list.

Tables

Like lists, tables have four different default layouts that editors can choose from:



Like the lists, the layouts are associated with classes `.contenttable-0`, `.contenttable-1`, `.contenttable-2`, or `.contenttable-3` which we can use in our own CSS to style them. The rows are assigned a class of `.tr-even` or `.tr-odd`. For large sets of data, this makes alternating row colors as simple as one line CSS affecting the `.tr-even` class. Finally, all rows and cells are assigned a class `.tr-x` or `.td-x`, where `x` is a number based on the current row or cell number. Here is an example of a full table from our example site:

```
<!--  Table: [begin] -->
<table class="contenttable contenttable-1"><tbody>
   <tr class="tr-even tr-0">
      <td class="td-0">Name:</td>
      <td class="td-1">Age:</td>
      <td class="td-2">Role in family:</td>
      <td class="td-last td-3">Note:</td>
   </tr>
   <tr class="tr-odd tr-1">
      <td class="td-0">Homer J. Simpson</td>
      <td class="td-1">36</td>
      <td class="td-2">Mentally absent father</td>
      <td class="td-last td-3"><b>Occupation:</b> Worker drone/safety
inspector, Sector 7G, Springfield Nuclear Power Plant. Holds plant
record for most years worked at an entry-level position.</td>
   </tr>
   <tr class="tr-even tr-2">
      <td class="td-0">Marge Simpson</td>
```

```
      <td class="td-1">34</td>
      <td class="td-2">Thread that holds Simpson family together</td>
      <td class="td-last td-3"><b>Jobs:</b> Strikebreaking teacher
at Springfield Elementary, worker at nuclear power plant, carshop
waitress, Springfield police officer, pretzel franchisee</td>
   </tr>
</tbody></table>
<!--  Table: [end] -->
```

Removing default markup in TYPO3

We have seen how we can use the extra default markup for styling, but sometimes we want TYPO3 to only output "clean" code without extra div tags or classes for a few reasons:

- We might want to remove legacy styling from TYPO3 to make it easier to match the carefully built HTML and CSS templates we got from a designer.
- If we are generating a lot of non-cached content, we will want to generate smaller pages to keep our bandwidth bill smaller; even ten kilobytes of total markup can add up to a lot if we have a million hits a month.
- Removing some of the default markup will make our source code look better and more custom when potential clients look at our previous sites.

No matter what our motivation is, we can add TypoScript to our template to take all the TYPO3 markup down to the bare necessities. Adding the following to the bottom of our TypoScript template setup will remove the paragraph classes, extra wrapper tags, and header markup:

```
lib.parseFunc_RTE.nonTypoTagStdWrap.encapsLines.addAttributes.P.class
>
lib.stdheader.stdWrap.dataWrap >
lib.stdheader.3.headerClass >
```

This will remove most of the extra markup, but not all of it. For the brave, there are more ambitious techniques that will come up every once in a while on the forums to remove all markup including the table and list markup. We're not going to look at those in this book, though, because many of them are known to break extensions or need specific patches applied to the TYPO3 core. The good news is that the core development team is working on ways to give even greater control over the HTML to developers so it will be easier to remove all markup we don't need or is actually hampering our CSS and keep the generated classes that we want.

Summary

Now that we have declared our own stylesheet to restyle our template, our example site is looking better already with our own CSS included. In this chapter, we have learned about four different ways to declare a CSS stylesheet in the TYPO3 template and opted to use `headerData`. After setting up our basic stylesheet, we looked at all of the default HTML markup TYPO3 provides that we can use in our own CSS. We also learned how to remove or change the default markup in TYPO3 to make our frontend HTML cleaner and more readable.

Now that we have added some of our own style, we have something to show our boss. Go ahead and show off our work real quick so we can justify the time spent reading a book at our desks and get back to learning.

After we all refill our coffee mugs, we'll jump into the next chapter and start digging deep into the TypoScript to add JavaScript and extra header markup to our template. The real fun starts when we dig into the TemplaVoila structures. Once we start modifying the data structure, we'll add dynamic logos, banners, and timestamps to our websites.

3
Adding Custom Template Fields

We have a good-looking site now. It is pretty and standards-compliant, but not particularly customized. Now we can jump in and start playing with the TypoScript and TemplaVoila templates. We need to update the metadata, but we also need to start adding key design elements like banners and a logo. Like almost everything in TYPO3, there are a lot of ways to add some of these elements. We are going to use TypoScript, to make them more flexible and easier to maintain. We want to make sure that our new elements are easy to update when we want to make changes, but we don't want to have to set the logo on each page.

In this chapter you will be:

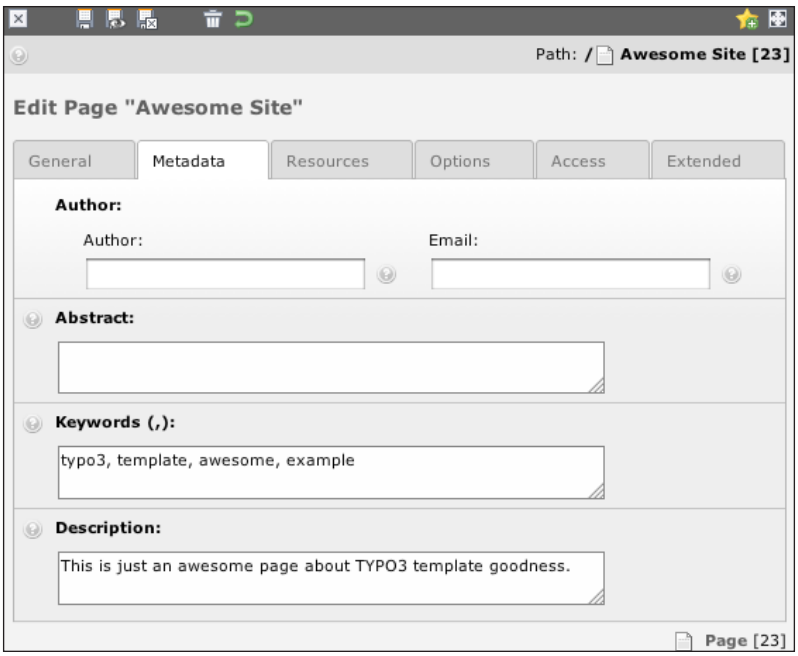
- Adding important keywords and description metadata through page properties
- Defining new elements in our TemplaVoila template
- Adding a banner to our page properties
- Adding a current date and timestamp to our template
- Adding dynamic logos to our template

Modifying the page metadata

Now that we are getting comfortable working with updating the TypoScript template, we are ready to start diving more into the HTML output generated from the TypoScript and integrating more information from the individual page properties into the template. For example, we can tie the keywords and descriptions section of the page properties to the TypoScript setup so that we output the keywords into the metadata in the head section of our HTML when it is viewed.

Proper metadata such as keywords and description on individual pages can be an important part of our overall strategy for search engine optimization (SEO). By including keywords in our individual pages, we can generate sitemaps for search engines automatically and help new visitors find our website easier.

First, we need to add some keywords and a description in the page properties of our front page. Remember that we looked at the page properties at the end of *Chapter 1* when we assigned the TemplaVoila template to our main page. This time we just need to edit the **Metadata** tab. Add some keywords and a description in the **Metadata** text areas as shown in the following screenshot:



To see this work right now, we can go ahead and add the following code to the bottom of our TypoScript template setup.

```
page.meta.keywords.field = keywords
page.meta.description.field = description
```

The nice thing about TypoScript is that it's fairly easy to read even if you're not comfortable writing it from scratch, yet. Anytime that we write `page.<something>`, we are modifying the `page` object that TYPO3 is using to build the rendered page. In this case, we are modifying `page.meta.keyword` and `page.meta.description`, and TYPO3 already knows where to place those in the rendered HTML output. Finally, the `keywords` and `description` values on the right side of the `=` are the names of the

specific fields in the page properties. When the page is generated for output, TYPO3 will pull the `keywords` and `description` fields from our page properties, assign them to the `page.meta` fields, and render the HTML with the new meta tags in the head.

If we clear the cache, reload the front page, and view the source code, we see our keywords in the HTML head:

```
<head>
   <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
   <!--
      This website is powered by TYPO3 - inspiring people to share!
      TYPO3 is a free open source Content Management Framework
initially created by Kasper Skaarhoj and licensed under GNU/GPL.
      TYPO3 is copyright 1998-2009 of Kasper Skaarhoj. Extensions are
copyright of their respective owners.
      Information and contribution at http://typo3.com/ and http://
typo3.org/
   -->
   <title>Awesome Site: Awesome Site</title>
   <meta name="generator" content="TYPO3 4.3 CMS" />
   <meta name="keywords" content="typo3, template, awesome, example"
/>
   <meta name="description" content="This is just an awesome page
about TYPO3 template goodness." />
   <link rel="stylesheet" type="text/css" href="typo3temp/
stylesheet_42a7d7391a.css" media="screen" />
   <script src="typo3temp/javascript_0b12553063.js" type="text/
javascript"></script>
   <link rel="stylesheet" type="text/css" href="fileadmin/templates/
reset.css" /><link rel="stylesheet" type="text/css" href="fileadmin/
templates/style.css" />
</head>
```

Using this technique, we've managed to move up our search rankings in about five minutes. That is impressive, but the most important thing we've done is seeing how we can link the dynamically controlled information from the page setup area into our TypoScript template.

Adding a banner

Okay, we're ready to start adding new sections for dynamic content to our template now. In order to add new content areas, we will have to start modifying the TemplaVoila data structure and template objects. For now, this is not much different from using the TemplaVoila Wizard that we used in *Chapter 1*. We are going to add new areas to the HTML file for mapping and use the backend interface to add elements to TemplaVoila, so that we can update fields with page properties or TypoScript in the future. For our first new element, we are going to add a banner to our example template.

Adding space for the banner to our HTML file

Before we modify the TemplaVoila data structure, we're going to add a `div` enclosure for the banner to our original HTML file. Like before, all that's important is that we create a set of tags with a unique identifier. We are going to map our new `div` tags in TemplaVoila, which will tell TemplaVoila to replace any content inside of them with some generated content that we will define in the data structure. If we add a `div` tag between our menu and the main content, the body of our HTML template will look like the example below.

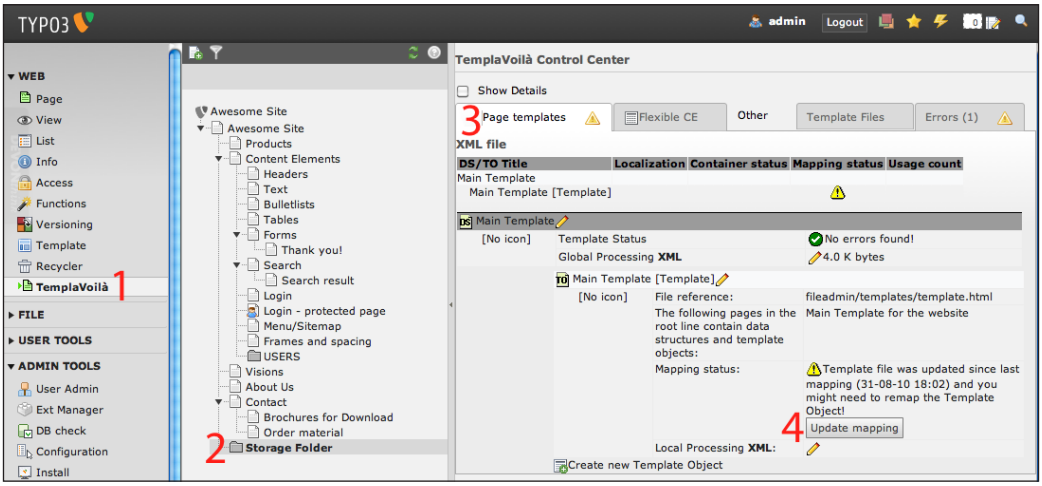
```
<!DOCTYPE HTML>
<html>
   <head>
      <meta charset="utf-8" />
   </head>
   <body>
      <ul id="menu-area"><li class="menu-item"><a href="">Menu Item
#1</a></li></ul>
      <ul id="submenu-area"><li class="submenu-item"><a
href="">Submenu Item #1</a></li></ul>
      <div id="banner_image"></div>
      <div id="content">This is our content</div>
   </body>
</html>
```

Adding the banner element to TemplaVoila

After we have saved the update to our HTML template, we need to add the element to the TemplaVoila data structure. TemplaVoila uses **data structures** to define the content elements for each template in a custom XML format that is stored in the database. In TemplaVoila, **template objects** are then used to store the mapping information between the original HTML file and the content elements in the data structure. So, our next step is to update the data structure with our new content element and then we can update the mapping in our template object.

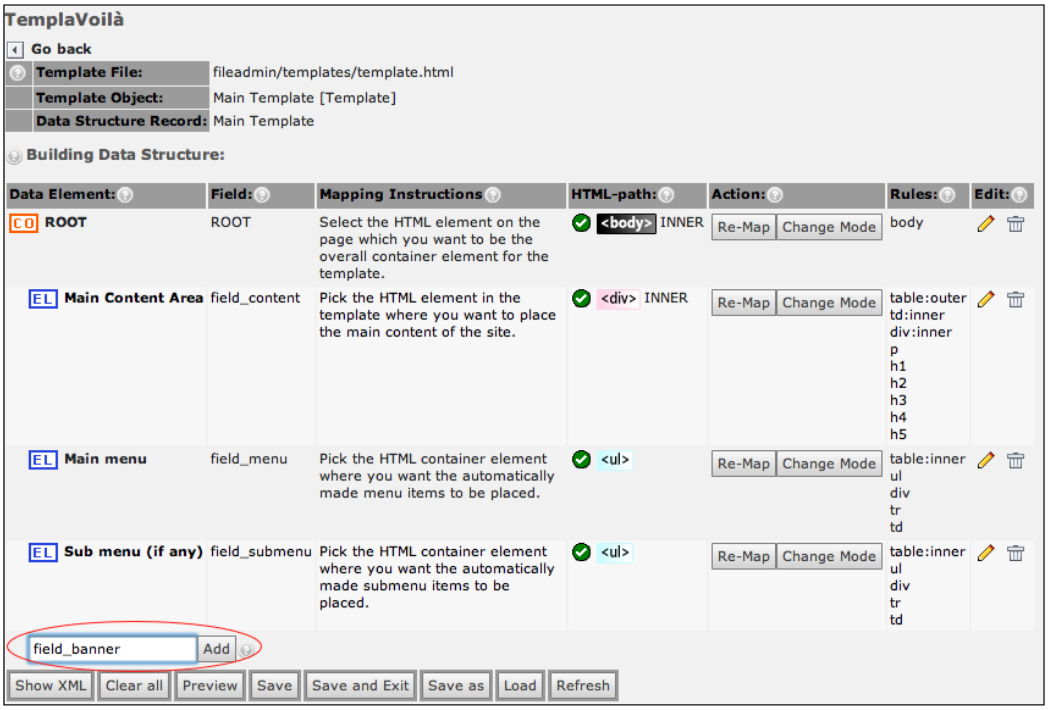
We can define the type of element and some parameters in a wizard for the data structure, and TemplaVoila will regenerate the data structure XML in the database according to what we set up. First, we need to go into the editing area for our data structure for the first time:

1. Choose **TemplaVoila** in the far-left menu to edit our TemplaVoila templates.
2. Choose **Storage Folder** from the page tree; this is where our TemplaVoila template objects and data structures are stored.
3. Make sure you are on the **Page templates** tab to see our current templates.
4. Under the **Main Template**, click on **Update mapping** to access the data structure and mapping areas.



5. On the **Template Mapping** page, click on the **Modify DS / TO** (Data Structure/Template Object) button in the **Information** tab.
6. When we click on the button to modify the data structure, TYPO3 will give us a pop-up warning about overwriting our XML. If we had manually edited the XML of our data structure at any point, then we would want to be careful here because editing the data structure with the backend graphic interface tools will overwrite the XML from scratch. As we have not edited the data structure by hand, we can go ahead and click on **OK** on the warning pop-up.

7. On the bottom of the data structure page, we can create a new element for the template. Go ahead and fill in the name of our new field, **field_banner**, and click on the **Add** button.



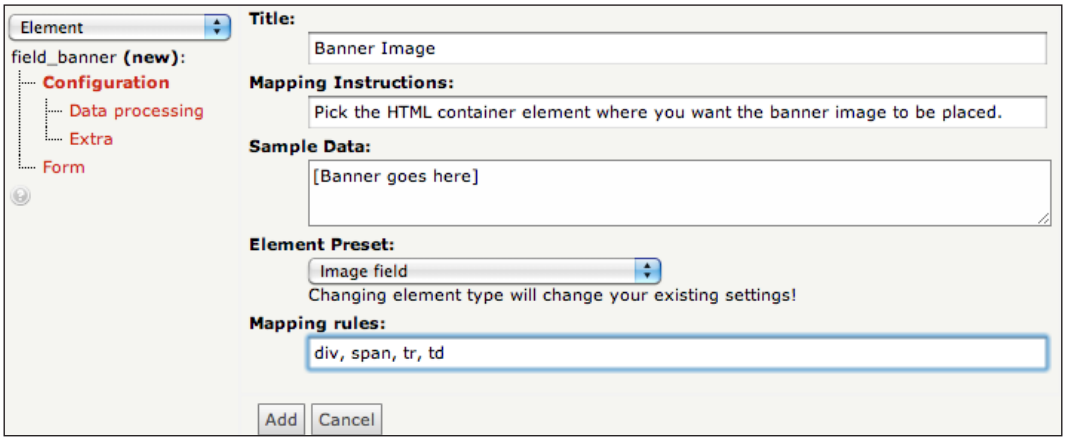
Configuring a data element

When the frame refreshes, we can fill in the fields to define our new data element. TemplaVoila used to require editing the data structure XML directly to define most of the fields for a data element, but the newest versions of TemplaVoila allow us to edit almost every field that we need completely within the graphical interface. For now, we can go ahead and fill out the fields similar to the default element fields.

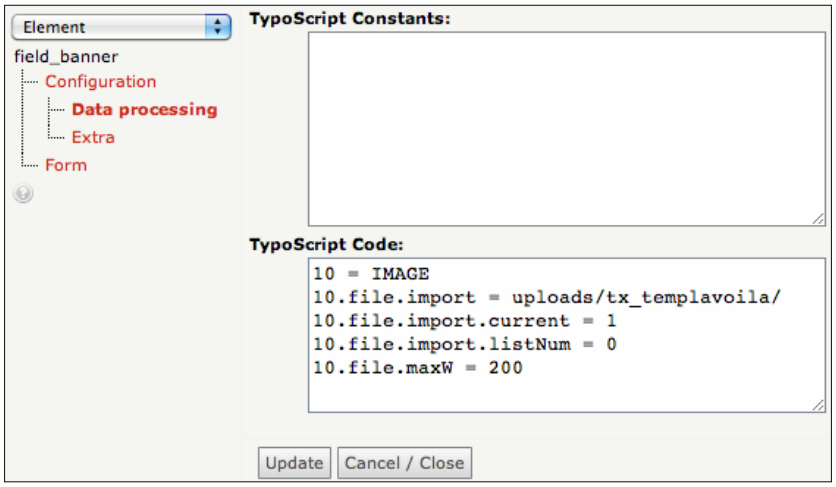
- **Title**: The title of our new data element will be displayed in the TYPO3 backend. To keep things simple, we will name our element Banner Image.
- **Mapping Instructions**: The mapping instruction will be seen in the backend on the mapping pages, and we are going to follow the same style as the default mapping instructions with our instructions: **Pick the HTML container element where you want the banner image to be placed.**

- **Sample Data**: Sample data is used when generating previews in the backend mapping area. Let's go ahead and follow the format of the default elements by leaving our **Sample Data** with the words **[Banner goes here]**, which will of course be filled in later by TYPO3.
- **Element Preset**: The **Element Preset** determines the type of element that we will be creating in the template. We can modify this information later, but by choosing an appropriate element preset at the beginning we allow TemplaVoila to automatically generate the most fitting XML and TypoScript data. This will make more sense in a moment when we start writing the TypoScript. As we are going to fill this data element with an image we will go ahead and set our element preset to **Image field**.
- **Mapping rules**: Template creators use mapping rules to restrict how other editors can map this data element in the future. We are going to tie this directly to an image, so we don't want users to accidentally map this to a header tag in the future. Let's go ahead and allow only a few container elements by setting our mapping rules as **div, span, tr, td**.

If we fill everything out correctly, this is what our screen should look like before we save it:



Once we are satisfied that we have filled out the initial configuration correctly, we can click on the **Add** button. Adding the field with the button saves it temporarily, but it is still not saved permanently to the template. Clicking on the **Add** button updates the configuration of our element based on the **Element Preset**, and TemplaVoila will generate some basic TypoScript to define our image element. Click on **field_banner | Configuration | Data processing** to see the preset TypoScript code:



As we are using this as a banner, we want to change the maximum width of our file from the default that TemplaVoila generated, `200`. Change the last line to `10.file.maxW = 800` and click on **Update**.

Now we can save our new data structure to the database. In order to permanently save our changes and overwrite the old data structure, we need to click on the **Save** button and clear the cache.

Viewing the data structure XML

Now that we have created a new element and saved the changes to our template, we can check the XML that the graphical interface created for us. Click on **Show XML** and we will see that it has added our banner image field to the XML with all of the configuration options we chose. Obviously, we only filled in a couple of fields, but it filled in the rest based on the defaults for our installation. If we want finer control over our data elements that the wizard can't give us, we can always edit the XML directly.

As you can see, most of the data structure XML is simple and easy to read. Once we edit the XML by hand, we can no longer use the wizards to modify the data structure or it will write over our changes.

The new field in our XML should look like the following code snippet. Go ahead and look through the code real quick to see the basic XML tags and TypoScript that our wizard created:

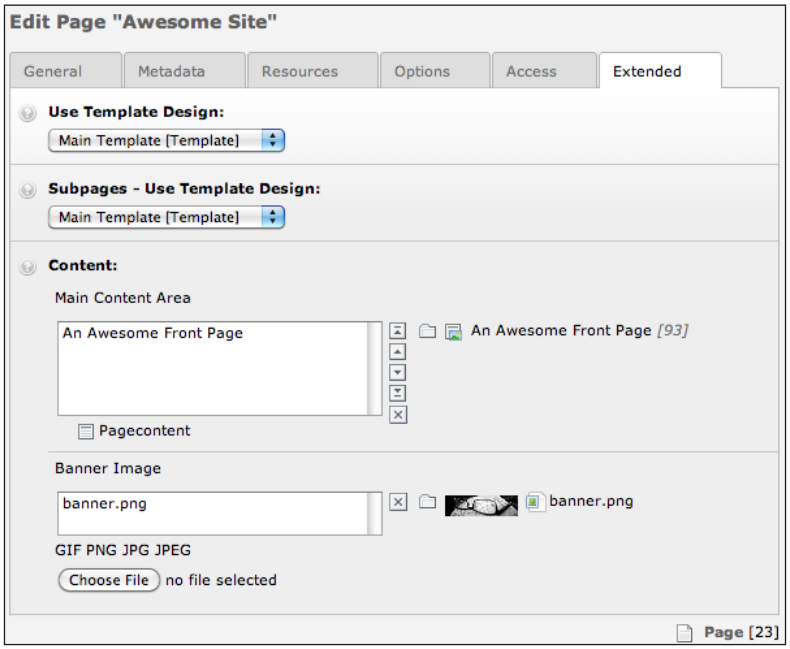
```
<field_banner type="array">
   <tx_templavoila type="array">
      <title>Banner Image</title>
      <description>Pick the HTML container element where you want the
banner image to be placed.</description>
      <sample_data type="array">
         <numIndex index="0">[Banner goes here]</numIndex>
      </sample_data>
      <eType>image</eType>
      <tags>div,span,tr,td</tags>
      <TypoScript>
         10 = IMAGE
         10.file.import = uploads/tx_templavoila/
         10.file.import.current = 1
         10.file.import.listNum = 0
         10.file.maxW = 800
      </TypoScript>
      <proc type=»array»>
         <stdWrap></stdWrap>
      </proc>
   </tx_templavoila>
   <TCEforms type=»array»>
      <label>Banner Image</label>
      <config type=»array»>
         <type>group</type>
         <internal_type>file</internal_type>
         <allowed>gif,png,jpg,jpeg</allowed>
         <max_size>1000</max_size>
         <uploadfolder>uploads/tx_templavoila</uploadfolder>
         <show_thumbs>1</show_thumbs>
         <size>1</size>
         <maxitems>1</maxitems>
         <minitems>0</minitems>
      </config>
   </TCEforms>
</field_banner>
```

Look at how the XML is storing the information that we just filled in like the sample data mapping rules. You can also see how the wizard used the element preset of `Image field` to generate some basic TypoScript and initial parameters to define our new image object.

Using our new data element

We have successfully added our own data element and configured it, and we even looked at the XML for fun. Next, we need to map our new element to its appropriate location in our original HTML file. Let's go ahead and click on the **Map** button in the backend and map our new data element to the `div` tag in our updated HTML template with the identifier `banner_image`.

Now that we have mapped the banner area, we can add a banner to our front page. This is where the editor experience is affected because we chose the **Image** preset when we created our new data element because different types of data elements are edited in different parts of the page or TypoScript template. If we had chosen **Page Content Elements**, then we would have another column visible in the **Web | Page** view. We chose the Image preset, so we are going to update and use this field by adding an image through our page properties. Go ahead and click on the page you would like to modify and right-click the page and choose **Edit page properties**. If we go to the **Extended** tab, we can see that **Banner Image** has been added as a new optional content area in the backend:



I chose a random image that was roughly the right size just to test our banner and show the example output. We haven't modified the flexible width layout in CSS or redesigned the page to work with one fixed size of banners, but we could do that later if we needed to. What we have done, though, is create our first new template element from scratch, and it wasn't even a standard element for generic content; we created a very specific banner element that could be used on a live site with just a little bit of CSS tweaking, but it doesn't look too bad right now:

Adding the date to our template

We have added our own element to the template, but we're trying to add a few more automated pieces to the template. The new banner is a nice touch, but it's just one more thing we would have to manually update in all of our current pages. Our boss really wants to see some new elements, and we want to add some features that can update themselves without extra work. Luckily for us, our boss is still impressed with the idea that modern websites can show the date programmatically and, bolstered by our newfound experience with data structures, we can add a date stamp to our example site before the next coffee break.

Adding space for the date to our HTML file

Just like the last modification, we're going to add a `div` container to our HTML template before we have a chance to forget. Let's go ahead and give our new `div` tag the identifier `timestamp`, `float` it to the `right` for a better layout, and add it directly above our menu like this:

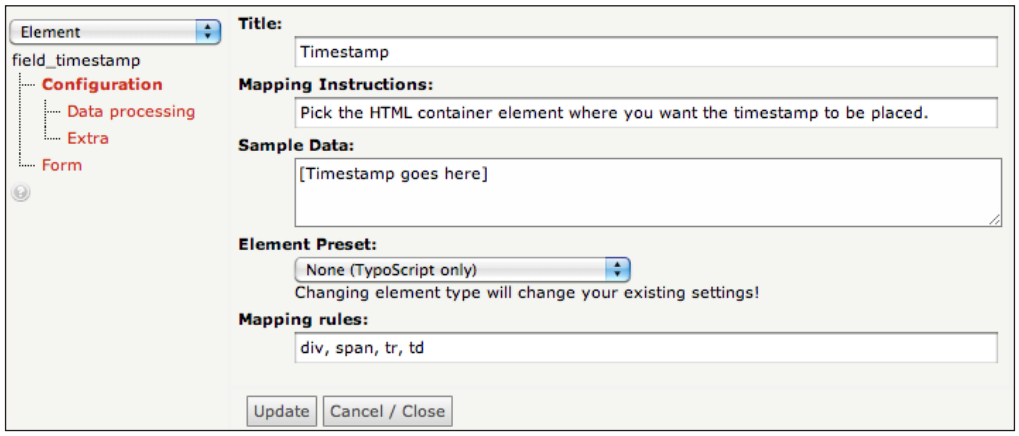
```
<body>
   <div id="timestamp" style="float: right;"></div>
   <ul id="menu-area"><li class="menu-item"><a href="">Menu Item #1</
a></li></ul>
   <ul id="submenu-area"><li class="submenu-item"><a href="">Submenu
Item #1</a></li></ul>
   <div id="banner_image"></div>
   <div id="content">This is our content</div>
</body>
```

Creating a data element

Once again, we're going to modify the data structure in the TemplaVoila backend just like last time. Let's go ahead and create a new field with the name **field_timestamp.** TemplaVoila does not allow spaces in the field name, so we use underscores instead. After you type in the field name and click on **Add**, fill in the following fields for the basic configuration:

- **Title**: We'll keep it simple again and make our title **Timestamp**. Because our new element will not contain elements for TYPO3 editors to work with, our title will only be seen on the mapping page this time.
- **Mapping Instructions**: We'll use the same format as before and say **Pick the HTML container element where you want the timestamp to be placed.**
- **Sample Data**: We can use the words **[Timestamp goes here]** as our sample data.
- **Element Preset**: This time our element is going to be using straight TypoScript without any visibility to editors so we're going to choose **None (TypoScript only)** from the preset drop-down list. As we are using **None** as our preset, we will do all of our setup and modifications in the data structure wizard itself.
- **Mapping rules**: Just like the last element, we are only going to restrict the mapping to prevent obvious mistakes by setting the mapping rules to **div, span, tr, td**.

Our new data element configuration before we save our changes looks like this:



After we've updated our configuration, we need to add our TypoScript to the element. To get to the TypoScript section of our configuration like last time, we need to click on **field_timestamp | Configuration | Data processing**. In the data processing section of the configuration, we can put the following code into the `TypoScript Code` text area:

```
10 = TEXT
10.data = date:U
10.strftime = %B %e, %Y
```

We are not doing anything extremely complicated in our TypoScript code. We are creating a text object, assigning it a `date` data type, and assigning the current time to it formatted as a standard date. For more information on TypoScript and date formatting, see the TSref.

Viewing the updated XML

Now that we have done our entire configuration, we can click on the **Add** button and save everything with the **Save** button. Like the banner element, we are going to overwrite the old template by saving our new version over it. If we choose to **Show XML** now, this is the new element we will see:

```
<field_timestamp type="array">
   <tx_templavoila type="array">
      <title>Timestamp</title>
      <description>Pick the HTML container element where you want the
timestamp to be placed.</description>
      <sample_data type="array">
```

```
            <numIndex index="0">[Timestamp goes here]</numIndex>
         </sample_data>
         <eType>none</eType>
         <tags>div,span,tr,td</tags>
         <TypoScript>10 = TEXT
            10.data = date:U
            10.strftime = %B %e, %Y</TypoScript>
         <proc type="array">
            <stdWrap></stdWrap>
            <HSC>1</HSC>
         </proc>
      </tx_templavoila>
   </field_timestamp>
```

Showing our new banner

Now that we have a new element, we can just map it to the `timestamp` `div` tag we already added to the HTML template. If we look at any of our pages on the frontend, we can now see the current date printed in the upper right corner alongside our menu:

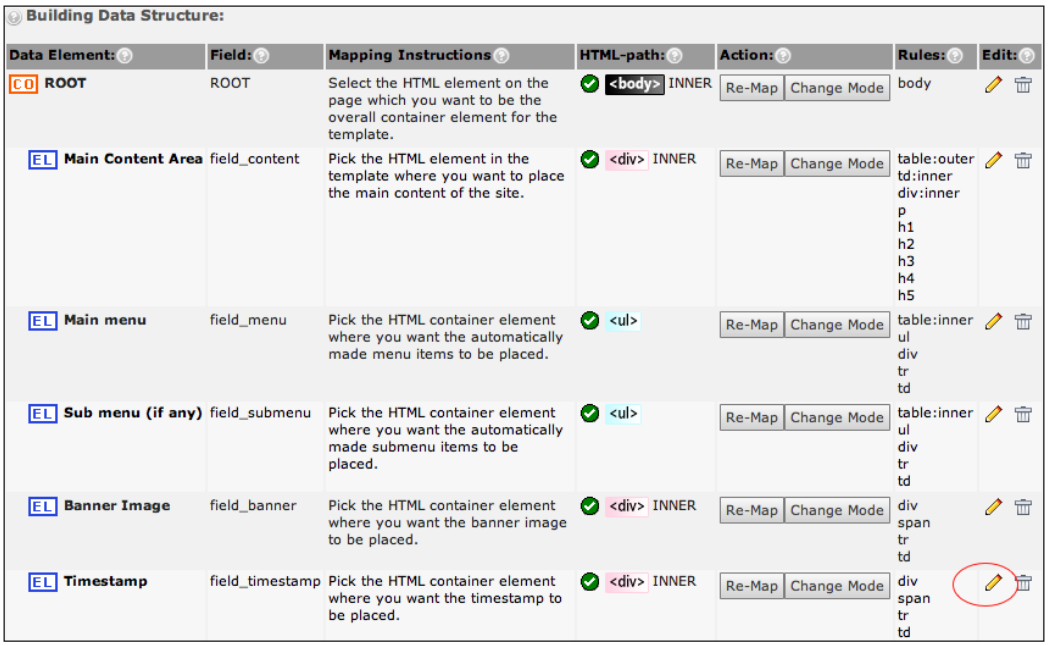
Loading the date and time from the TypoScript template

We've successfully added the date to our page with the template, and we are definitely feeling good about ourselves right now. Unfortunately, the boss points out that they want to show the entire time stamp instead of just the date. We can argue the finer points of the formatting, but what we've realized is that the exact formatting of our date stamp will need to be tweaked sometimes in the future; putting the formatting directly into the template data structure will probably cause problems in the future with this particular element. Luckily for us, TYPO3 gives us many ways of

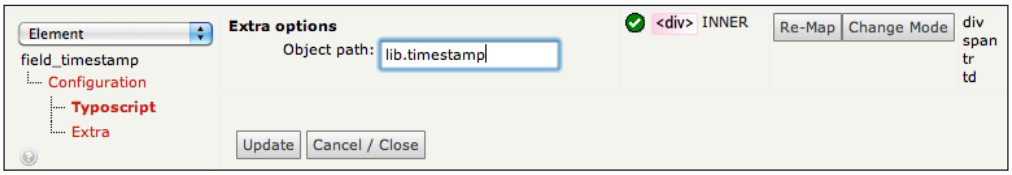
doing things, and we can just use a different method to create the same final output. Our next step is to move the actual time stamp generation and formatting rules into the template setup where they can be updated without requiring us to change the main data structure all the time. We're going to change the data structure from using integrated TypoScript to using a TypoScript object. If we use a TypoScript object in our data structure, we can assign values and configure the TypoScript object in the TypoScript template setup.

Changing our timestamp element in the data structure

Our first step is to modify the data structure to change our data element so that we can use a TypoScript object to define in the TypoScript template setup, instead of defining all of our TypoScript in the XML. Let's go ahead and get into editing mode on our data structure again like we have in the past couple sections. We can choose to edit the data element by selecting the pencil in the **timestamp** section like shown in the following screenshot:



We will change the **Element Preset** from **None (TypoScript only)** to **TypoScript Object Path** and click on the **Update** button. When the frame reloads, we can choose **Typoscript** from the tree on the left under the title **field_timestamp** to enter a new TypoScript object path. We can give our new object any name that we want starting with `lib`, but it is helpful to make sure we use a specific and clear name that makes sense when we define it in the TypoScript template setup. This time, we can call our new object `lib.timestamp`:



When we click on the **Update** button again, we will use the **Save** button to overwrite the old data structure with our new structure. If we check the XML, we can see the immediate changes:

```
<field_timestamp type="array">
   <tx_templavoila type="array">
      <title>Timestamp</title>
      <description>Pick the HTML container element where you want the
timestamp to be placed.</description>
      <sample_data type="array">
         <numIndex index="0">[Timestamp goes here]</numIndex>
      </sample_data>
      <eType>TypoScriptObject</eType>
      <tags>div,span,tr,td</tags>
      <proc type="array">
         <int>0</int>
         <HSC>0</HSC>
         <stdWrap></stdWrap>
      </proc>
      <preview></preview>
   <TypoScriptObjPath>lib.timestamp</TypoScriptObjPath>
   </tx_templavoila>
</field_timestamp>
```

Adding the timestamp object to the TypoScript template

The next thing we need to do is define the timestamp object in the TypoScript template setup, just so that we can clearly see the change. We are going to change the formatting to show the current time as well as date (for time formatting information, see the TSref). Right now, we are going to add the following code to the bottom of our TypoScript template setup on the root page:

```
lib.timestamp = COA
lib.timestamp {
   10 = TEXT
   10.data = date:U
   10.strftime = %B %e, %Y %T
}
```

If we clear the cache and look at the front page now, we can see a complete time stamp with date and time in the upper right corner. Now, when somebody needs to change the format of our time stamp, they can simply tweak the TypoScript template setup instead of diving into the XML of our carefully constructed data structure.

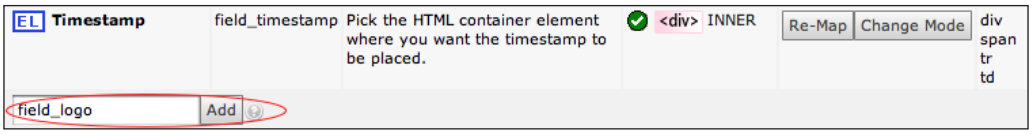
Adding a dynamic logo

Our final change will be to add a logo to the top right corner of our website. We could, of course, add this in the HTML template, but we want our logo to be a TypoScript object like the time stamp. We know that we may need to change the logo at certain times of the year, and we don't like the idea of just overwriting a single image file for every change. We also want the ability to overwrite the logo in subsections of our site or create different logos for international versions of our site in the future. So, we have decided that it should be a TypoScript object.

Our first step, like always, is to add the space for a logo to our HTML. We know that we want it at the top of our site, and we're not going to use any CSS positioning right now; we can go ahead and add it right below the body tag in our HTML:

```
<body>
   <div id="logo"></div>
   <div id="timestamp" style="float: right;"></div>
   <ul id="menu-area"><li class="menu-item"><a href="">Menu Item #1</
a></li></ul>
   <ul id="submenu-area"><li class="submenu-item"><a href="">Submenu
Item #1</a></li></ul>
   <div id="banner_image"></div>
   <div id="content">This is our content</div>
</body>
```

Now that we're getting into the flow of editing the template, we know that our next step will be adding it to the TemplaVoila data structure as a new data element. We can go ahead and open up the data structure for editing again and add a new field at the bottom of the page called **field_logo**:



Now that we know how this works, we're going to just fill in the form with the new configuration information:

- **Title**: We'll keep it simple and call this **Logo** in the backend.
- **Mapping Instructions**: Just like the other elements, we'll use the same basic kind of instructions: **Pick the HTML container element where you want the logo to be placed.**
- **Sample Data**: Our sample data can be **[Logo goes here]**.
- **Element Preset**: We know we're going to use an object this time, so we can go ahead and choose **TypoScript Object Path** from the drop-down menu.
- **Mapping rules**: Like the other elements we've added, we just want some simple rules: **div**, **span**, **tr**, **td**.
- **Object path**: We'll set this under the TypoScript section again. This time we can call it **lib.logo** so our editors know exactly what we are creating.

Now we can click on the **Add** button and use **Save** to overwrite the old template data structure with the new one. If we choose to **Show XML**, this should be the new field that the wizard created from our configuration:

```
<field_logo type="array">
   <tx_templavoila type="array">
      <title>Logo</title>
      <description>Pick the HTML container element where you want the
logo to be placed.</description>
      <sample_data type="array">
         <numIndex index="0">[Logo goes here]</numIndex>
      </sample_data>
      <eType>TypoScriptObject</eType>
      <tags>div,span,tr,td</tags>
      <proc type="array">
         <stdWrap></stdWrap>
      </proc>
      <TypoScriptObjPath>lib.logo</TypoScriptObjPath>
   </tx_templavoila>
</field_logo>
```

If everything looks okay, we can map it to the new `div` tag with the `logo` identifier and save our changes.

After we've saved our mapping, we need to add our new object to the TypoScript template. I created a quick image with the filename `logo.png` and saved it into the `fileadmin/templates/` directory. It's also helpful to make sure that our logo links back to the homepage, so we're going to add a standard wrapper to the image with a link to our example site for right now. We could have created an image element like we did for the banner, but we wouldn't be able to set the image and wrap it in a link for the entire site using TypoScript. Because we are using the TypoScript template, we can add this code to the bottom of our TypoScript template setup to define our image and the link together:

```
lib.logo = IMAGE
lib.logo.file = fileadmin/templates/logo.png
lib.logo.stdWrap.wrap = <a href="http://www.example.com/">|</a>
```

My fancy new logo simply says **Example.com** in the example below, but our site is still looking much more professional:

Summary

Now that we have started using the page properties and TypoScript, we have a whole new world of options available to us. We've already used our knowledge to enhance the search engine optimization (SEO) and add banners, a timestamp, and a logo.

Now that we are modifying the data structure, we can add any elements we want to a template without fear. We can use TypoScript and TemplaVoila together to give us the results that others are expecting from a powerful engine like TYPO3, and we deserve to be proud for a moment. It's the end of another chapter, so we can take our traditional coffee break and show our co-workers the cool site we're building. As soon as we get back, we're going to tackle one of the most important pieces of successful site building, navigation, in the next chapter. We're going to look at updating our text menu, generating graphic menus, building toolbars with icons, and even adding breadcrumb navigation to our site.

4
Creating Flexible Menus

We just got done adding flexible banners and logos to our site, and we're ready to attack one of the biggest problems in template building: navigation. If you've flipped through, you noticed that this chapter has more tables and example code than everything before it combined. Don't worry, that's a good thing. The hierarchical menu in TYPO3 is one of the most powerful concepts available for us as TYPO3 developers. It includes the ability to create flexible menus with text, graphics, JavaScript, or DHTML completely through TypoScript without spending time in Photoshop or learning a new language.

Most importantly, this chapter is packed with code and examples because navigation is possibly the most important thing to get right in web design. All of the content and soon-to-be bestselling products in the world don't mean anything if users can't find them. Our menu is more than just the text at the top of the page; it's our first marketing pitch on how great a company we are to work with because we can help you find what you're looking for in the clearest way possible. So, it's time to get excited because there's no better way to show off our skills and please our boss (maybe get a bonus) then making a beautiful menu.

In this chapter you will:

- Learn about the base hierarchical menu object in TYPO3: `HMENU`
- Get introduced to the text-based menu object
- Rebuild the main menu and submenu from scratch with fancy text-based menu options
- Create your first graphical menu with a custom font, drop-shadows, and rollover actions
- Create a breadcrumb navigation menu that helps users see where they are in the website

Page tree concepts

We are about to dive into all of the little details, but there are a few basic concepts that we need to review first. We've already talked about the page tree and levels a little in the *Chapter 1*, but we are going to be using these concepts a lot in this chapter. So, we're going to make sure we have a more complete definition that avoids any confusion:

- **Page tree**: Our TYPO3 page tree is all of the pages and folders that we work with. This includes the home page, about us, subpages, and even non-public items such as the storage folder in our example site. If we have a very simple website it could look like this:
    - **Home**
    - **About Us**
        - **Staff**
- **Level**: Our page tree will almost always have pages, subpages, and pages under those. In TYPO3, these are considered levels, and they increase as you go deeper into the page tree. For example, in our extremely simple website from the example above both **Home** and **About Us** are at the base (or root) of our page tree, so they are on level 0. The staff page is underneath the **About Us** page in the hierarchy, so it is on level 1. If we added a page for a photo gallery of our last staff lunch as a subpage to the staff page, then it would be at level 2:
    - **Home** (Level 0)
    - **About Us** (Level 0)
        - **Staff** (Level 1)
            - **Staff Lunch Gallery** (Level 2)
- **Rootline**: TYPO3 documentation actually has a few different uses for the term "rootline", but for the menu objects it is the list of pages from your current page or level moving up to the root page. In our example above, the current rootline from the **Staff Lunch Gallery** is **Staff Lunch Gallery | Staff | About Us**.

Introducing HMENU

Before we look at all the different kinds of menus in TYPO3 and all their little differences, we need to explore the base TypoScript object for all of them: **HMENU**. `HMENU` generates hierarchical menus, and everything related to menus in TYPO3 is controlled by it. As the base object, `HMENU` is the one thing that every type of menu

is guaranteed to have in common. If we understand how HMENU is creating its hierarchical menu, then everything else is just styling.

We can already see an example of HMENU being used in our own TypoScript template setup by looking at the menus that the TemplaVoila Wizard generated for us:

```
## Main Menu [Begin]
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO {
   allWrap = <li class="menu-item">|</li>
}
## Main Menu [End]

## Submenu [Begin]
lib.subMenu = HMENU
lib.subMenu.entryLevel = 1
lib.subMenu.wrap = <ul id="submenu-area">|</ul>
lib.subMenu.1 = TMENU
lib.subMenu.1.NO {
   allWrap = <li class="submenu-item">|</li>
}
## Submenu [End]
```

We can see that the wizard created two new HMENU objects, `lib.mainMenu` and `lib.subMenu`, and assigned properties for the entry level and HTML tags associated with each menu. We're about to learn what those specific properties mean, but we can already use the code from the wizard as an example of how HMENU is created and how properties are defined for it.

Types of menu objects

The HMENU class does not output anything directly. To generate our menus, we must define a menu object and assign properties to it. In our current menus, the TemplaVoila Wizard generated a menu object for each HMENU in the following highlighted lines:

```
## Main Menu [Begin]
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO {
   allWrap = <li class="menu-item">|</li>
}
```

```
## Main Menu [End]

## Submenu [Begin]
lib.subMenu = HMENU
lib.subMenu.entryLevel = 1
lib.subMenu.wrap = <ul id="submenu-area">|</ul>
lib.subMenu.1 = TMENU
lib.subMenu.1.NO {
   allWrap = <li class="submenu-item">|</li>
}
## Submenu [End]
```

There are a handful of classes for menu objects that can be used by `HMENU` to generate menus in TYPO3, but we are going to be concentrating on the two most powerful and flexible options: `TMENU` and `GMENU`.

The TemplaVoila Wizard used `TMENU` in our current menu, and it is used to generate text-based menus. Menus built with `TMENU` output the title of each page in the menu as a text link, and then we can use HTML and CSS to add styling and layout options.

Menus created with the `GMENU` class are considered graphic menus. We can use `GMENU` to dynamically generate images from our page titles so that we can use fancy fonts and effects like drop-shadow and emboss that are not supported in CSS by all browsers equally.

Menu item states

The menu system in TYPO3 allows us to define states for different menu options. For example, using the state definitions, we can customize the behavior of menu items when they are active or rolled over. The normal state (`NO`) is available and set by default, but all of the menu item states must be enabled in TYPO3 by adding code to our template like this: `lib.mainMenu.1.ACT = 1`. All menu objects share a common set of menu item states from the table below:

State	Description
`NO`	This is the normal state for menu items.
`ACT`	This state is active if the menu item is in the rootline for the current page. Remember, this means that the menu item is in the path from the current page to the highest level of our page tree.
`CUR`	This is active when the menu item is for the current page.
`IFSUB`	This state applies to any page with subpages.
`RO`	This state can be used for rollover or mouse-over actions.

State	Description
USR	This state applies to pages that restrict access by frontend user groups. This can be used to show pages like members-only sections differently in the menu.
ACTIFSUB	TYPO3 includes support for special combination states that rely on two or more conditions being met. ACTIFSUB, for example, shows the active state for a menu item if you are on a subpage of that page that is if the menu item is in your current rootline.

HMENU properties

Because HMENU is the root of all of our other menu objects, any of the properties that we learn for HMENU will be applicable to all of our menu options that we might use on future websites. I've included a list of the TypoScript properties that we are most likely to use in the TypoScript template setup, but you can see the complete list in the TSref (http://typo3.org/documentation/document-library/references/doc_core_tsref/current).

If you haven't used TypoScript much, and this is too much information all at once, don't worry. It will make more sense in a few pages when we start experimenting on our own site. Then, this will serve as a great reference.

Property	Description

entryLevel	This defines at which level in the current page tree we want the menu to start. The default, 0, is the base of the page tree. In our example site, the entryLevel for our main menu is 0:
	`lib.mainMenu.entryLevel = 0`
	In our current site, we define the entryLevel of our submenu as 1 because we want to show the subpages on level 1 underneath the top page in our current rootline:
	`lib.subMenu.entryLevel = 1`

maxItems	We can also set the maximum number of items in the menu with maxItems. After the menu reaches maxItems, it will ignore any remaining items. This affects all of the submenus as well. If we want to set it specifically for each menu associated with an HMENU object, we can to set it in the menu objects. Like in programming languages, this is called **inheritance** because the submenus are inheriting this value from the main menu

Property	Description
`begin`	This sets the first item in the menu. For example, if you wanted to skip the first three pages in your page tree, you could set the value of `begin` to 4: `lib.mainMenu.begin = 4` Like `maxItems` above, being has inheritance. If we need to control this on each menu object, we can to set it on the menu object itself.
`excludeUidList`	The `excludeUidList` property excludes the pages associated with the ID's listed. You may list as many items as you want separated by commas, and you can add the keyword `current` to the list to exclude the current page: `lib.mainMenu.excludeUidList = 4,8,15,16,23,42,current` As a caution, I wouldn't recommend actually hard-coding pages you don't want in the menu. It's better practice to just flag the pages in their setup to not show in menus so you don't have to modify the template code if pages are moved around, but this property can be good for excluding your current page from a menu when it makes more sense for the user experience.

`special`	This property allows us to create `special` menus to show updated pages, keyword pages, and others. We can also use this property with as a list if we only want to show a few specific pages like a special navigation menu:
	`lib.mainMenu.special = list`
	`lib.mainMenu.special.value = 3,12,19,80`
	We are mainly talking about normal page tree menus and submenus in this chapter, but we will talk about **breadcrumb** menus later in this chapter.

As we've already witnessed in the main menu, TYPO3 sorts our menu by the order in the page tree by default. We can use this property to list fields for TYPO3 to use in the database query. For example, if we wanted to list the main menu items in reverse alphabetical order, we could call the `alternativeSortingField` in our template:

```
lib.mainMenu.1 = TMENU
lib.mainMenu.1.alternativeSortingField = title desc
```

Common menu item properties

We've looked at the menu as a whole, but a lot of our TypoScript configuration revolves around the menu items themselves. Editing all of the menu items in TypoScript is not hard, but it's not completely straightforward. When we are adding TypoScript configuration like HTML tags to all of our menu items, we will always start by assigning them to the normal (NO) state. For example, we haven't talked about the specific properties available yet, but the following TypoScript code will assign a class to all of the menu links:

```
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO.ATagParams = class="menu-links"
```

Here are some of the most common menu item properties that we might use in our TypoScript templates:

Property	Description
addParams	Using addParams, we can add GET parameters to the link URLs in the menu. The value must be encoded for raw URL processing: `lib.mainMenu.1.NO.addParams = "&your_variable=your%20value"`
ATagParams	We can add additional attributes such as class and style to the <a> tag with ATagParams. Example: `lib.mainMenu.1.NO.ATagParams = class="menu-links"`
ATagTitle	This will specify the title attribute of the <a> tag for the menu items affected. For example, we can use the abstract or description: `lib.mainMenu.1.NO.ATagTitle.field = abstract // description`

allWrap	We can wrap the entire menu entry including the link in our own arbitrary HTML output with allWrap. We can use this in our own site to wrap each item in our main menu with a new span tag with the class menu-item by adding this to our TypoScript setup:
	`lib.mainMenu.1.NO.allWrap = <span class="menu-item">|</span>`
	All of the wrap properties in TypoScript have the same format where | represents the object. In the code above, the | is representing the entire menu entry. For example, this is the original menu entry output:
	`<a href="index.php?id=1">Home</a>`
	The allWrap property would place span tags on either side so it looks like this:
	`<span class="menu-item"><a href="index.php?id=1">Home</a></span>`

Property	Description
`wrap`	We can wrap just the menu entry (without the link) in our own arbitrary HTML output with `wrap`. We can use this to add special formatting around our menu items inside of the link.
`linkWrap`	This works like `allWrap`, but it wraps just the `<a>` tag of our menu entries. This can result in different output from `allWrap` if we are inserting images with `beforeImg` or `afterImg` (discussed below).
`before`	Like `allWrap` and and `linkWrap`, this is used to insert arbitrary HTML output near our menu item. In this case, `before` determines what will be displayed before the menu entry. If we do not need to wrap an entry with output before and after it, we can simply use the `before` property instead of `allWrap`.
`after`	This property works like `before`, but it defines what should be displayed only after the menu entry.

Introducing text-based menus

The simplest type of menu is the text-based menu, `TMENU`, which displays the page titles of menu items as text links. It's the fastest, drop-in solution for any menu, and it's what TemplaVoila created by default. The big advantages for `TMENU` are speed and flexibility. It's the fastest to implement, but it's also the fastest menu object to render because it's not downloading any special graphics or complex layouts during the loading of our page.

It also means that your design is not tied to the TypoScript template, so you can do all your customization through CSS or even JavaScript libraries. If you're designing for CSS3-compatible browsers and you really know what you're doing, then you can build a very fast menu with custom fonts and effects that degrade gracefully across unsupported browsers without TypoScript. We are going to look at ways to integrate non-web fonts and complex graphic effects using TypoScript in this chapter because, for those who aren't CSS or JavaScript geniuses already, TypoScript is easier and more flexible. Almost anything is possible, with `TMENU`, CSS, and JavaScript.

What are the disadvantages? Obviously, the main disadvantage can be appearance. If your designer has handed you a folder of mockups with fancy graphical menus, then they are not going to be impressed with our text-based navigation. If you want fancy buttons, but you're still designing for legacy browsers and graceful degradation (displaying functional but less advanced designs to older browsers) is not an option, you'll need to use `GMENU` with `GIFBUILDER`.

If you build enough websites, you will end up using various menus at different times, but `TMENU` is both a great starting point for new projects and a nice alternative when we need the basic functionality of our navigation.

TMENU Properties

As a child of HMENU, TMENU shares some common properties with the other menu objects, but there are some specific TMENUITEM properties that can be modified for the items in a text-based menu:

Property	Description
beforeImg	We can specify an image to be displayed before the menu entry with beforeImg. This property is an imgResource, so we can simply declare the file we want to display as a link, or we can expand and use the more advanced properties of an image resource discussed in the TSref (width, height, and so on). We will use some of the advanced properties in our examples.
beforeROImg	This declares an image to be displayed on rollover in conjunction with beforeImg. To use this, we must enable rollovers on our menu.
beforeImgLink	The beforeImgLink property will link the beforeImg image with the same information as the text link in the menu item if this is set.
afterWrap, afterImg, afterROImg, afterImgLink	These properties function similar to beforeWrap, beforeImg, and so on. to add images and wrapped elements after the text menu item.

Adding separators to menu items

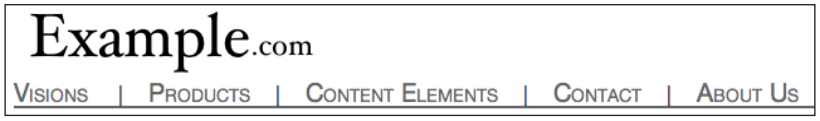
Okay, now that we've seen the properties we can use, we can start using them and improving the basic menu we currently have. We remember that our boss complained early on that our multiple word menu titles sometimes looked like separate items and wondered if we could put separators in-between the items. Of course, we could try to just add a | after every menu item by saying lib.mainMenu.1.NO.after = |, but then the menu will still have one annoying | at the end. Luckily for us, we can use the optionSplit function in TypoScript. With the optionSplit function, we can define different wrappings for the first, middle, and last elements. The optionSplit function can be used with many different menu properties including our different wrapping properties, and the basic syntax looks like this:

```
wrap = [First element] |*| [All middle elements] |*| [Last element]
```

As we're already using the `allWrap` property, we don't really want to retype the `li` three times. We can go ahead and use the `before` and `after` properties. We only actually need the vertical separators after our menu items, so we can quickly skip the `before` property and just add a special `after` to our main menu using `splitOption` like this:

```
## Main Menu [Begin]
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1.NO {
   allWrap = <li class="menu-item">|</li>
   after =       | |*|
     | |*|  
}
## Main Menu [End]
```

Now, our main menu with CSS spacing and vertical separators can look like this:



Redesigning the text-based menus

Okay, so we've made some tasteful changes to update our navigation and make our boss happier, but we need to see the full power of the basic text menu in TYPO3 before we move on to graphical menus. We can't just go back to our bosses and show them that we added little lines to the menu; it's not that impressive. We need to really experiment with this, and that means we have to make more than small tasteful changes. We need to break stuff.

1. There is a limited amount of space in our submenu area, so we'll set a maximum limit of items:

   ```
   lib.subMenu.1.maxItems = 8
   ```

2. We can skip over the first page in the main page tree (**Products**) for now, so we'll just start on the second item:

   ```
   lib.mainMenu.1.begin = 2
   ```

3. The spacing between the text items and the separators seems a little imprecise. We can replace the ` ` spaces with spacer images:

   ```
   lib.mainMenu.1.NO.after =  <img src="clear.gif" width="15"> | |*|
   <img src="clear.gif" width="15">  |  |*|   
   ```

4. If we add a class to the <a> tags, it will make it even easier to style in CSS:

```
lib.mainMenu.1.NO.ATagParams = class="menu-links"
```

5. Right now, we don't have a title field in the <a> tags, but a title helps usability for everyone and especially those using screen readers or other assistive devices. We're all about web standards right now. We can add the description or page title from the page properties to the link's title field to clarify the contents of the page to which we are linking. To insure that we don't have blank titles, we are going to use the // operator in TypoScript so that TYPO3 will look for the description field first and use the title field if the description is empty:

```
lib.mainMenu.1.NO.ATagTitle.field = description // title
```

6. It might end up looking tacky, but we just want to see what it looks like to place an image in front of our text titles in the menu. Just to be consistent, we want to make sure that the image links to the same page as the text:

```
lib.mainMenu.1 {
   NO {
      beforeImg = fileadmin/templates/bullet.png
      beforeImgLink = 1
   }
}
```

7. While we're at it, let's just add a rollover image. Remember we have to enable the rollover state at the end for it to work:

```
lib.mainMenu.1.NO {
   RO = 1
   beforeROImg = fileadmin/templates/bullet_rollover.png
}
```

8. Finally, we decide that we want to show the pages in the current rootline (with the ACT state) differently to set them apart. Our first though might be to just worry about the current page state (CUR), but it might be nice if the main menu item displayed differently even if we're on a subpage. We can use the < operator TypoScript (see TSref for more information) to copy the properties of the normal state over as a starting point. To copy all of the properties, we will write ACT < .NO (not ACT < NO). We'll change the class on the active links and remove the image in front of them. Once again, we will have to enable the ACT state to use it:

```
lib.mainMenu.1 {
   ACT < .NO
   ACT = 1
   ACT {
      ATagParams = class="active-link"
```

```
            beforeImg >
        }
    }
```

9. Finally, we need to add a class to the stylesheet to differentiate the active links according to TYPO3. We've just told the template to change the class to `active-link`, so we can add the following code to the bottom of our CSS file to change the color of the link to a light grey:

```
li.menu-item a.active-link {
   color: #ddd;
}
```

After all of our changes, our menu may not be as simple and clean as it once was, but it definitely shows what we can do with some lines of TypoScript. Here is a screenshot with from the **Content Elements** page. With the mouse hovering over the **Visions** menu item we can see the rollover image and the link title text that is being pulled from the description field of the page properties:

Final code

If you stepped through all of those changes with us, you probably noticed that we added a lot of redundant lines because we wanted all of our modifications to stand on their own. Of course, in a production environment, those changes should look cleaned up:

```
## Main Menu [Begin]
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = TMENU
lib.mainMenu.1 {
   begin = 2
   NO {
      allWrap = <li class="menu-item">|</li>
      after =  <img src="clear.gif" width="15"> | |*| <img src="clear.
gif" width="15">  |  |*|  
      ATagParams = class="menu-links"
      ATagTitle.field = description // title
      beforeImg = fileadmin/templates/bullet.png
```

```
        beforeImgLink = 1
        RO = 1
        beforeROImg = fileadmin/templates/bullet_rollover.png
    }
    ACT < .NO
    ACT = 1
    ACT {
        ATagParams = class="active-link"
        beforeImg >
    }
}
## Main Menu [End]

## Submenu [Begin]
lib.subMenu = HMENU
lib.subMenu.entryLevel = 1
lib.subMenu.wrap = <ul id="submenu-area">|</ul>
lib.subMenu.1 = TMENU
lib.subMenu.1.maxItems = 8
lib.subMenu.1.NO {
    allWrap = <li class="submenu-item">|</li>
}
## Submenu [End]
```

Introducing graphic menus

Now that we've modified our text menu, we are definitely feeling a little more confident with the hierarchical menu. We might want to show this off to the bosses, or we can make our titles jump off the page a little more before we start bragging. Now we can try out a simple graphical menu object referred to as `GMENU` in TYPO3. `GMENU` is our next step above `TMENU` in power and complexity, but it's still a pretty easy transition if we take a few minutes to understand the concepts.

Our graphical menu, `GMENU`, has the ability to use a TYPO3 class, `GIFBUILDER`, to create images on the fly. We are going to look at `GIFBUILDER` in depth in just a moment, but the basic idea is that `GIFBUILDER` can help us build complex graphics for our menus dynamically so that we can use fonts that are not be supported by all browsers, build button-like graphics, and use drop-shadows and embossing for effect. This means that we don't need to update Photoshop every time that we want to change the menu titles or replace the font, and we also don't have to learn or use Flash or JavaScript just to create flexible menu titles.

We gain a lot of freedom and power with GMENU, but it does have some disadvantages compared to TMENU. Depending on our needs, we will be generating an image for every state (normal, rollover, active, and so on), for each menu item anytime we make a change. The nice thing is that TYPO3 will cache our images, so they are only generated after we make a change and reset the cache. Overall, we will use the server a little more heavily anytime we reset the cache, and our users will need to download images for each menu item. The last disadvantage is just that it is more complex than TMENU. If our frontend developers or designers have already provided us with some awesome CSS/JavaScript or Photoshop images, then we may only need a text-based menu. In return for those possible disadvantages, we will get freedom and cross-browser compatibility, so let's see what we can do before we make any decisions.

Introducing GIFBUILDER

GIFBUILDER is a universal object in TypoScript that uses some basic TypoScript code to generate images using the ImageMagick (or GraphicsMagick) library in PHP. Generating images with ImageMagick is normally very complex, and we would have to learn a fair amount of PHP to make anything. Using GIFBUILDER, makes it relatively easy to make these same dynamic images without learning PHP or opening Photoshop. GIFBUILDER can actually be used for any images that we would want to create in TypoScript, but we are going to be using it specifically for GMENU in this book to turn our text fields into typographic images complete with layers and effects.

We are going to learn about three main objects in GIFBUILDER that will help us create our menu items:

- Boxes: We can layer simple boxes to make borders or button effects.
- Images: We can use uploaded or external image files as backgrounds or just display them as menu items.
- Text: Most importantly, we can use text objects to show our page titles in non-web fonts with drop-shadow or emboss effects.

A complete list of the properties available to GIFBUILDER is beyond the scope of this chapter, and not really necessary to build most menus. If you want to learn more about GIFBUILDER, I recommend *Mastering TypoScript* by Daniel Koch. If you have any problems with GIFBUILDER, you may need to check your ImageMagick configuration in the TYPO3 Install Tool.

The BOX object

The BOX object is one of the key TypoScript objects in GIFBUILDER. The BOX object is, like it sounds, just a simple graphical box defined by its size and color. By itself, it's not that helpful, but we can add boxes as layers to generate borders and backgrounds that will be flattened into our final generated images. We are only going to use two properties for our boxes:

- BOX.dimensions defines the dimensions of the box in the format x, y, w, h where x, y is the offset for the top right-corner and w, h is the width and height of the box.
- BOX.color defines the color of the inside of the box.

Here is an example of a gray box, 400 pixels wide, 20 pixels tall, and offset 3 pixels down and to the right:

```
lib.subMenu.1.NO {
   5 = BOX
   5.color = #aaaaaa
   5.dimensions = 3,3,400,20
}
```

The IMAGE object

The next object we can use, IMAGE, will bring in an image for normal display or basic tiling and masking. The IMAGE object can be used for complex displays, but we are only looking at menu applications and will just look at a few options:

Property	Description

file	We can declare point to a filename or use a use a complete imageFile declaration like the rollover images we used in the TMENU section:
	`10.file = fileadmin/templates/image.png`
	`10.file {`
	`  import = uploads/media/`
	`  import.field = media`
	`  import.listNum = 0`
	`  width = 15`
	`  height = 10`
	`}`

offset	This is the x, y offset from the layer beneath it.
tile	We can tile the image x, y times inside the GIFBUILDER object. The maximum is 20 in each direction, so you may need a bigger image if you need to cover more space.

The TEXT object

Finally, we're going to look at the options for a `TEXT` object in `GIFBUILDER`. `TEXT` objects are used to display any text we want in `GIFBUILDER`, but will be mainly using them to show the title of each page as a menu item. This list of properties is much more exhaustive because this would obviously be one of the most important objects to customize when we're creating a menu using graphical text:

Property	Description
`text.field`	This is the field from the menu item we want to use for the text. We are going to use `title` in our current site, but we could use the `description` or `abstract` if that was necessary.
`fontFile`	As we are rendering this text as a graphic image, we don't need to rely on browser-safe fonts. We can point this to any font file that we are licensed to use to generate our images. This is one of the most important reason to look at using `GMENU` because it means we don't have to create in images in Photoshop or Flash files for every change or rely on JavaScript/CSS solutions that may not work with the font we want to use.
`fontSize`	This is the font size of our text.
`fontColor`	This is the color of our text.
`offset`	Like the other objects, we can define an offset from the layer before this.
`align`	We can align the text to the right, left, or center.
`angle`	We can set the angle to display the text at. For a 45 degree angle, we could say `10.angle = 45`.
`niceText`	If set, this will render small text cleaner by generating the text at twice its font size scaling it down to the correct size using ImageMagick on the server. Unfortunately, this can be CPU-intensive so we should only use it in production if we've tested it with and without and decide whether we need the functionality.
`shadow.offset`	We can render the text with a shadow and define its x, y offset.
`shadow.color`	This defines the color of the shadow.
`shadow.blur`	We can change the blur of the shadow with values 1-99.
`shadow.opacity`	If blur has a value, we can set the opacity of the shadow from 0% (transparent) to 100% (opaque). Example of 80% opacity: `10.shadow.opacity = 80`
`shadow.intensity`	We can set the intensity of the shadow from 0 to 100.

Property	Description
`emboss.offset`, `emboss.blur`, `emboss.opacity`, `emboss.intensity`	We can also use an emboss effect on our text using properties similar to the shadow properties.
`emboss.highColor`	This sets the upper border color of the emboss effect.
`emboss.lowColor`	This sets the lower border color of the emboss effect.

GIFBUILDER layers

We work with `GIFBUILDER` by creating new objects for the `GIFBUILDER`, designing them with properties, and layering them by number values. Each layer is stacked in ascending order (larger numbers on top), and then TYPO3 generates a final image by flattening all of the layers into one image. It sounds a little complex, but look at this example:

```
lib.subMenu.1.NO {
   5 = BOX
   5.color = #aaaaaa
   5.dimensions = 3,3,400,20
   10 = TEXT
   10.text.field = title
   10.fontSize = 12
}
```

In the example that we just saw, `lib.subMenu.1.NO` is our `GIFBUILDER` object. Although the numbers used to identify objects (`5` and `10` in the example) are sometimes arbitrary in TYPO3, they are very important for `GIFBUILDER` because they define the ordering in layers. `GIFBUILDER` stacks it's subobjects from lowest number to highest. So, in the example that we just saw, TYPO3 is generating an image in a logical sequence:

1. A gray box is defined.
2. The dimensions of the gray box are defined to make it 400 pixels wide and 20 pixels.
3. A text object is created on top of the gray box to show the title field of the page from the menu item.
4. The size of the text for the title is set to 12 pixels.
5. TYPO3 generates a flattened image of our menu item title in a gray box.

Using this system, we can stack very simple objects on top of each other to draw basic buttons.

GIFBUILDER properties

The GIFBUILDER object will apply itself to all items in a GMENU menu. For the basic GIFBUILDER object, we are only going to look at two properties:

- XY defines the size of the image (width and height)
- backColor defines the background color for the entire image

The interesting trick for XY (and some of the other dimension properties) is that it can be based on hard-coded numbers and TypoScript constants, or it can be a calculation based on the size of another item. In the following code, the size of the GIFBUILDER object is tied directly to the size of the TEXT object declared below it:

```
10 = TEXT
10.text.field = title
XY = [10.w]+10,[10.h]+10
```

The references [10.w] and [10.h] read the current width and height of the object associated with 10. Then, we add 10 pixels onto each one to give ourselves a little bit of room for spacing. We'll use this technique in GMENU to make sure that our boxes and graphic objects always line up with our titles.

GMENU properties

The first thing that you'll probably notice going from TMENU to GMENU is that we are about to multiply the number of properties at our disposal dramatically. We've already covered the common menu properties that we won't talk about again, but I could easily finish out a new chapter just discussing all of the options we have with GIFBUILDER objects. Instead, we're just going to cover the most basic or necessary properties in the following tables to get an idea of what we can do. If it looks intimidating, don't worry. Most of the properties are created logically and build upon our earlier knowledge. Most importantly, there's no requirement to learn all of GIFBUILDER before we start playing around.

The menu object itself has just a few key properties that we need to look at:

Property	Description
min	This sets the minimum dimensions for the entire menu in the standard TypoScript width and height format (x, y). Example: `lib.mainMenu.1.min = 50,20`
max	This sets the maximum dimensions for the menu in the same format as min.

Property	Description
`useLargestItemX`	If this is set, the width of all menu items will be equal to the widest item.
`useLargestItemY`	If this is set, the height of all menu items will be equal to the tallest item.

Creating our first graphic menu

The first change we can accomplish is updating our main menu with a custom font and some rollover functionality. We can do that with minimum fuss, and it'll update our whole look nicely. I chose a freeware font from Larabie Fonts (`http://www.larabiefonts.com`) called **Deftone** because it'll show off `GIFBUILDER`, and my boss loves it. You can use any TrueType font file you would like, though. Some fonts seem to work better with ImageMagick then others, so you may need to experiment. In any case, let's start updating our menu:

1. We need to change `lib.mainMenu.1 = TMENU` to `lib.mainMenu.1 = GMENU` to use the `GMENU` objects.
2. We want a consistent height for our entire menu, so we'll enable `useLargestItemY` in our template:

   ```
   lib.mainMenu.1.useLargestItemY = 1
   ```

3. Let's update the normal menu state first. We won't be using the `div` tags around our menu items, so want to add a class to our images:

   ```
   lib.mainMenu.1.NO.ATagParams = class="menu-link"
   ```

4. We can set the background color and dimensions of our menu items. We are going to use `10` for our text object, so we can go ahead and use that as part of the size calculation to make our items exactly the same width and 5 pixels taller than the text:

   ```
   lib.mainMenu.1.NO {
      backColor = #ffffff
      XY = [10.w],[10.h]+5
   }
   ```

5. Now we can create the `TEXT` object. This is our main menu, so we're just going to use the title as our text content. We're also going to use the **Deftone** font at a size of `36` in a classic black:

   ```
   lib.mainMenu.1.NO {
      10 = TEXT
      10.text.field = title
      10.fontFile = fileadmin/templates/deftone.ttf
   ```

```
    10.fontSize = 36
    10.fontColor = #000000
    10.align = left
}
```

6. The main menu is already looking better, but we can add some flair by tilting the text up with the angle property. Because of the angle changing the dimensions, we'll push the text down a little more by adding a `50` pixel offset to the height:

```
lib.mainMenu.1.NO {
    10.offset = 0,50
    10.angle = 3
}
```

After all of our modifications, this is what our menu should look like:

Modifying based on menu states

We can't deny that the menu is looking better, but we lost that rollover functionality that we were showing off in the `TMENU`. Luckily, we can add that back with just a couple lines of code. We can use the `<` operator to copy all of the `NO` properties over, then we can change the text to red and add a shadow whenever the mouse rolls over it with the `RO` state:

```
lib.mainMenu.1 {
    RO < .NO
    RO {
        10.fontColor = #990000
        10.shadow.offset = 1,1
    }
    RO = 1
}
```

If we want to differentiate the pages in the current rootline by graying them out, it's almost the same process:

```
lib.mainMenu.1 {
    ACT < .NO
    ACT {
        10.fontColor = #999999
        10.shadow.offset = 1,1
    }
    ACT = 1
}
```

In the following screenshot, you can see the results if we are on the **Products** page (so it's active) and roll over the **Content Elements** menu item with our mouse. We could almost show this off right now.

Main menu code

After all of the tweaking, our code might be getting ugly. If we take away the redundancies and use curly braces, this is what our code can look like:

```
## Main Menu [Begin]
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = GMENU
lib.mainMenu.1.useLargestItemY = 1
lib.mainMenu.1 {
   NO {
      allWrap = <li class="menu-item">|</li>
      ATagParams = class="menu-link"
      backColor = #ffffff
      XY = [10.w],[10.h]+5
      10 = TEXT
      10 {
         text.field = title
         fontFile = fileadmin/templates/deftone.ttf
         fontSize = 36
         fontColor = #000000
         align = left
         offset = 0,50
         angle = 3
      }
   }
   RO < .NO
   RO = 1
   RO {
      10.fontColor = #990000
      10.shadow.offset = 1,1
   }
   ACT < .NO
   ACT = 1
   ACT {
      10.fontColor = #999999
      10.shadow.offset = 1,1
   }
}
## Main Menu [End]
```

Creating a graphic menu with boxes

Now that we've created a basic `GMENU`, we can get a little fancier with the submenu. This time, let's actually create buttons with embossed text. Because the emboss effect works better with certain fonts, we don't want to use the **Deftone** font again. I tried plenty of combinations early on, but I love the look of an emboss on a simple, serif typeface. I've downloaded an open source font from **The League of Movable Type** (`http://www.theleagueofmovabletype.com`) called **Goudy Bookletter 1911**, but you can use any font with clean, straight lines to get the same effect.

1. We added a hefty margin to the submenu in CSS, but we don't really want to stop and modify our stylesheets while we're playing around. Instead, we can wrap our submenu in a `ul` with a reset margin to offset the styling. This wouldn't be a permanent solution, but it's handy while we're experimenting:

   ```
   lib.subMenu.wrap = <id="submenu-area" style="margin-left:
                              0px">|</ul>
   ```

2. Just like the main menu, we need to enable GMENU by changing `lib.subMenu.1 = TMENU` to `lib.subMenu.1 = GMENU`.

3. This time, we won't worry about the height of the menu (it'll be consistent, anyway), but we do want all of the buttons to be the same width. We can set the menu to use the widest item as a base:

   ```
   lib.subMenu.1.useLargestItemX = 1
   ```

4. Like the main menu, we need to add a class to the images for future reference:

   ```
   lib.subMenu.1.NO.ATagParams = class="submenu-link"
   ```

5. This time, we'll define the background as a light gray. We are going to use `10` for TEXT object, so we'll set the width of each menu item to be `10` pixels wider than the text by using `[10.w] + 10` for our width. We will set the height of the menu as a whole to `26` pixels so we have space for boxes and text inside. The background that is visible between the boxes will be a visual border around the entire area:

   ```
   lib.subMenu.1.NO {
      backColor = #bbbbbb
      XY = [10.w]+12,26
   }
   ```

6. We're going to define a darker gray box for each menu item as our button:

   ```
   lib.subMenu.1.NO {
      5 = BOX
      5.color = #aaaaaa
      5.dimensions = 3,3,400,20
   }
   ```

7. Next, we'll add a text field on top of the box. Remember, the numbering of our objects defines their layer, so we're going to use `10` to place the `TEXT` object on top of the `BOX` object:

```
lib.subMenu.1.NO {
   10 = TEXT
   10.text.field = title
   10.fontFile = fileadmin/templates/goudy_bookletter_1911-
webfont.ttf
   10.fontSize = 12
   10.fontColor = #555555
}
```

8. This time, we're going to set the offset and alignment to center the text horizontally and vertically in the boxes:

```
lib.subMenu.1.NO {
   10.offset = 0,19
   10.align = center
}
```

9. Finally, we'll add the emboss effect using shades of gray to make the text appear sunken into the boxes:

```
lib.subMenu.1.NO {
   10.emboss.offset = 1,1
   10.emboss.highColor = #cccccc
   10.emboss.lowColor = #bbbbbb
}
```

Now that we've updated our submenu, it should look like this:

Submenu code

Once again, step-by-step modification doesn't lend itself to clean code. If we optimize our code and remove all the redundant lines, this should be our final submenu code:

```
## Submenu [Begin]
lib.subMenu = HMENU
lib.subMenu.entryLevel = 1
lib.subMenu.wrap = <ul id="submenu-area" style="margin-left: 0px">|</
ul>
lib.subMenu.1 = GMENU
lib.subMenu.1.maxItems = 8
```

```
lib.subMenu.1.useLargestItemX = 1
lib.subMenu.1.NO {
   allWrap = <li class="submenu-item" style="margin-right: 0px">|</li>
   ATagParams = class="submenu-link"
   backColor = #bbbbbb
   XY = [10.w]+12,26
   5 = BOX
   5 {
      color = #aaaaaa
      dimensions = 3,3,400,20
   }
   10 = TEXT
   10 {
      text.field = title
      fontFile = fileadmin/templates/goudy_bookletter_1911-webfont.ttf
      fontSize = 12
      fontColor = #555555
      offset = 0,19
      align = center
      emboss.offset = 1,1
      emboss.highColor = #cccccc
      emboss.lowColor = #bbbbbb
   }
}
## Submenu [End]
```

Using external images for menus

Now that we've seen how we can generate images, we should look at using external images as well. It's going to happen quite often in the real world that a designer is going to already have icons or images designed for our menu that can't possibly be replicated dynamically with `GIFBUILDER`. In our case, we'll imagine that our boss wants to use some special graphics for each main menu item. We want the editors to be able to update these images in the future without breaking our template or bother us, and it's probably a good idea to make sure that we fall back to a `GIFBUILDER` image if somebody forgets to assign an image to a menu item in the page properties.

The first thing we can do is adding an alternate image property to our menu items with the `altImgResource` property. The `altImgResource` property takes some of the same file parameters as the `beforeImg` functions we saw earlier:

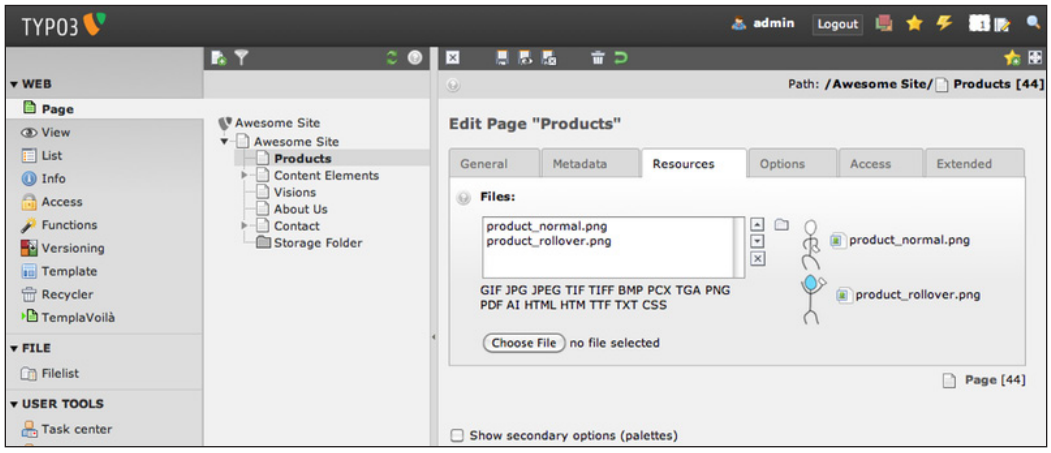
```
altImgResource.import = uploads/media/
altImgResource.import.field = media
altImgResource.import.listNum = 0
```

Using these parameters, our menu can import images associated with our pages through the resources tab in page setup. By assigning the `listNum` to 0, we will pull the first image from the list. We can also assign higher numbers in the list to use different images for rollover and active states as well. It's easier to understand if we see this in action, so we can add the `altImgResource` code highlighted below to our main menu code. Like earlier, we have comments added for information on each function:

```
## Main Menu [Begin]
lib.mainMenu = HMENU
lib.mainMenu.entryLevel = 0
lib.mainMenu.wrap = <ul id="menu-area">|</ul>
lib.mainMenu.1 = GMENU
lib.mainMenu.1.useLargestItemY = 1
lib.mainMenu.1 {
   NO {
      allWrap = <li class="menu-item">|</li>
      ATagParams = class="menu-link"
      ## Add an image resource to load up if available
      altImgResource.import = uploads/media/
      ## Use the media field of the page setup for images
      altImgResource.import.field = media
      ## Import the first image listed in the page setup
      altImgResource.import.listNum = 0
      ## If no images in the page properties, create
      ## GIFBUILDER image
      backColor = #ffffff
      XY = [10.w],[10.h]+5
      10 = TEXT
      10 {
         text.field = title
         fontFile = fileadmin/templates/deftone.ttf
         fontSize = 36
         fontColor = #000000
         align = left
         offset = 0,50
         angle = 3
      }
   }
   RO < .NO
   RO = 1
   RO {
      ## For rollover, import the second image listed in
      ## the page properties
```

```
            altImgResource.import.listNum = 1
            10.fontColor = #990000
            10.shadow.offset = 1,1
        }
        ACT < .NO
        ACT = 1
        ACT {
            ## For now, the active state will use the same
            ## altImgResource as the normal state
            10.fontColor = #999999
            10.shadow.offset = 1,1
        }
    }
    ## Main Menu [End]
```

The new menus will use the resources from the page setup to show items. After we've updated the menu, we'll notice that nothing may have changed initially because we haven't added files to the page resources. This is okay, because it just means that the menu is falling back to `GIFBUILDER` correctly. To show our images, we can add files to the resources tab of the page properties. Remember, we can access the page properties by right-clicking on our page in the page tree and selecting **Edit**:



According to our code, the menu should try to load the first image for a normal state, and it will display the second file in the list for when the mouse rolls over it. If we add these resources to our **Products** page, we will see an image replace the text for that menu item:

On rollover, the **Products** menu item will show the second image in the list:

As you can see, the `altImgResource` can be a very helpful and powerful way to use more complex images designed in Photoshop or Illustrator inside our own templates. It even keeps the HTML tags that we wrapped around our menu items and the `link` tag parameters. If we use `altImgResource` and an image returns from the page properties, than it will simply override any `GIFBUILDER` configuration. This is why we can use `GIFBUILDER` as a seamless backup menu procedure for when an image is not set in the page properties. So, if it's overriding the `GIFBUILDER` configuration, why is it considered an alternate image resource? I don't know, but according to the TSref "this is how it irreversibly is and has been for a long time."

Other types of menus

We have covered the two main types of menus in TYPO3 that we will need to know for template building, `TMENU` and `GMENU`. TYPO3 offers other menu options that we do not have time to cover in this chapter. Because so many of the other menus are extensions of `TMENU` or `GMENU`, it was more important that we spend time learning `TMENU` and `GMENU` in-depth then try to gloss over all six menu options. Now that we know the big two, we can extend our menus to some of the following options:

- `GMENU_LAYERS` and `TMENU_LAYERS` extend `GMENU` or `TMENU` to create a multi-level DHTML (Dynamic HTML) menu.
- `GMENU_FOLDOUT` extends `GMENU` to create a multi-level menu with JavaScript actions.
- `IMGMENU` creates a large image map with `GIFBUILDER` objects.
- `JSMENU` creates drop-downs that dynamically load using JavaScript.

In most situations, you won't need to go very deep into any of these menu options that date back to solving problems with Netscape 4 and predate modern JavaScript libraries, but the TSref has detailed information for all of them.

Breadcrumb navigation

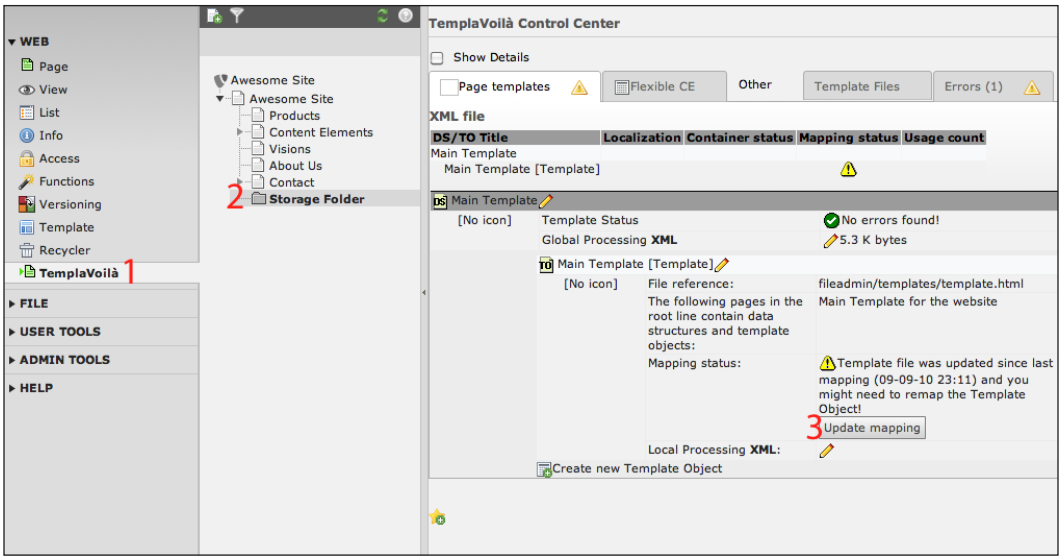
Along with standard page tree navigation, TYPO3 menu objects can be used for specific navigation using the `special` properties. It is possible to list recently updated pages, pages related by keywords, and even a complete directory listing. One of the best options, though, is the `rootline` menu that shows the path along the rootline from the top of the menu to the current page. In web development, this is called **breadcrumb** navigation because it leaves a trail for users to take back to the beginning, and it is used in e-commerce sites or websites with deep navigation trees.

To create our own breadcrumb navigation, we need to add a new element to our HTML file and TemplaVoila template. First, we need to add the space for our breadcrumb navigation to our HTML. We can place it directly below the submenu in our HTML file:

```
   <body>
      <div id="logo"></div>
      <div id="timestamp" style="float: right;"></div>
      <ul id="menu-area"><li class="menu-item"><a href="">Menu Item
#1</a></li></ul>
      <ul id="submenu-area"><li class="submenu-item"><a
href="">Submenu Item #1</a></li></ul>
      <div id="breadcrumb">Top Menu \ Next Page</div>
      <div id="banner_image"></div>
      <div id="content">This is our content</div>
   </body>
```

Once we've updated our HTML, we need to add a new field to our main TemplaVoila data structure. We will follow the same steps from the last chapter:

1. Click on **TemplaVoila** in the far-left menu.
2. Choose the **Storage Folder** from the page tree.
3. Click **Update mapping**.



4. Click **Modify DS / TO** (Data Structure / Template Object) on the mapping page.
5. At the bottom of the data structure page, add a new field called `field_breadcrumb`:

6. Finally, fill in the form for the new field with the following values:
   - **Title**: Our title can be **Breadcrumb Navigation**.
   - **Mapping Instructions**: Just like the original elements, we'll add some basic instructions: **Pick the HTML container element where you want the breadcrumb navigation to be placed**.
   - **Sample Data**: Our sample data can be **[Breadcrumb navigation goes here]**.
   - **Element Preset**: We are going to use TypoScript object to define the menu, so we can go ahead and choose **TypoScript Object Path** from the drop-down menu.

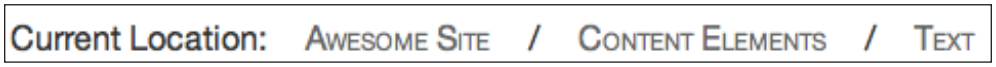
- **Mapping rules**: Like the other elements we've added, we just want some simple rules: **div**, **span**, **tr**, **td**.
- **Object path**: We need a name for our object that will be easy to remember, so we can call it **lib.breadcrumb** so our editors know exactly what we are creating.

7. Save the changes to our template and map the new field to the new breadcrumb `div` tag in our HTML file.

Now that we have updated our TemplaVoila template, we can try out the breadcrumb navigation by defining it at the end of our TypoScript template setup with the following code:

```
## Declare a new HMENU object
lib.breadcrumb = HMENU
## Define the new object as a rootline
lib.breadcrumb.special = rootline
## Sets the range from the base of the page tree (0) to the
## current level (-1)
lib.breadcrumb.special.range = 0|-1
## Adds basic wrapping and titles to menu
lib.breadcrumb.wrap = Current Location:
lib.breadcrumb.1 = TMENU
lib.breadcrumb.1
     {
   target
      = _top
  NO.before
      =  
  NO.after
      =   / |*|  / |*|  
 }
```

Using this code exactly as it stands (and some CSS like the submenu styling), we can see this output:



Pulling it all together

Now that we've seen the different menu options we can take our pick of what we want to keep for our example site. We've seen the most basic text, text with icons, graphical text, buttons, and designer image resources. Our boss will probably love the graphical main menu, but we might have been experimenting a bit too much when we converted the submenu to little embossed buttons. The alternate images were also great, but stick figures may not be the most helpful navigation aids. So, before we show this off, we can just grab the best bits of our graphical main menu and our text-based submenu:

Summary

We can be proud because in this chapter, we have learned everything that we really need to know to build great menus in TYPO3. We can now customize the navigation to our boss's content, and it's definitely time for another coffee break. In fact, we should probably show this to a few co-workers just so they know why we just spent the past hour staring at a book. As soon as we get back, we'll start creating some new templates in the next chapter. We'll be creating printable versions of our pages, special layouts for different sections, and adding them to the list for editors to choose from when they create new pages. So, take a break and hurry back with your coffee and a text editor handy.

5
Creating Multiple Templates

We've done pretty well with just a single template, but we will need multiple templates to build a complete website in TYPO3. Bosses and content editors are not going to be that impressed that we've effectively given them only one editable content area in the entire template; we are going to need some diversity in our internal pages for multiple columns or sidebars if we want to make a professional website. It's our job to make sure that the editors have options for the layout when they are adding new pages or sections or updating current ones, so we're going to start adding new layouts into the mix.

In this chapter you will:

- Create multiple templates for different page layouts
- Add a sidebar to the list of possible content areas in the backend
- Assign specific templates to pages and subpages
- Create icons for our templates to make it easier to choose the right layout for a page
- Update the icon and timestamp fields for just one section of the website
- Create a printable template with CSS
- Create a new printable subtemplate for the main template
- Create a dynamic link to a print-friendly template on each page

Creating new templates with sidebars

We know that we need more templates, but we can't spend too much time on each one or make updating more confusing for editors. Of course, we could go through all the work of creating multiple new data structures and template objects just to add some sidebars, but we would end up repeating a lot of work needlessly. Instead of creating a set of complete templates with new data structures, we can just create a handful of template objects in TemplaVoila that can share one data structure and one new HTML file. We already have one main data structure that we are using, and TemplaVoila allows multiple template objects to be mapped to a single data structure. Essentially, each template object is just a unique mapping of the fields in the data structure to an HTML file. As long as we are using most of the same data fields, we will just keep creating new template objects for the existing data structure. We can even add a few fields like sidebars to the main structure; we don't have to map them in template objects that don't need them.

Our boss really loves sidebars, so we can impress everyone with a few choices of templates with sidebars. We could create just one, but we can create three new templates with almost the same amount of work. We'll create one with a sidebar on the right, one with a sidebar on the left, and one with sidebars on both sides.

Creating the HTML and CSS

Our mapping will help us define the final layout of each template object, but we don't need to create a new HTML file for each object. We can re-use the same HTML file for multiple templates by creating a new HTML file with `div` areas for left and right sidebars. We'll just map them or ignore them in each template object depending on what we need. Technically, this template will be exactly like the current HTML template except for the sidebars, so we can even copy `template.html` to start with. We could actually modify the main HTML template, but we're still experimenting. There's no point in breaking the current template by accident. For now, we can just create a file named `template_sidebar.html` in the templates directory (`fileadmin/templates/`) that looks just like the main template with two div tags added for sidebars:

```
<!DOCTYPE HTML>
<html>
   <head>
      <meta charset="utf-8" />
   </head>
   <body>
      <div id="logo"></div>
      <div id="timestamp" style="float: right;"></div>
      <ul id="menu-area"><li class="menu-item"><a href="">Menu Item
#1</a></li></ul>
```

```
      <ul id="submenu-area"><li class="submenu-item"><a
href="">Submenu Item #1</a></li></ul>
      <div id="banner_image"></div>
      <div id="left_sidebar" class="sidebar"></div>
      <div id="right_sidebar" class="sidebar"></div>
      <div id="content">This is our content</div>
   </body>
</html>
```

With the new HTML in place, we can update our stylesheet to arrange our templates correctly. We're going to keep this pretty simple. For our site, all sidebars can be exactly 200 pixels wide, and we'll put a 30 pixel margin between the sidebar content and the main content area next to it. We can add the following code to our main stylesheet (`fileadmin/templates/style.css`) to set the width, margins, and appropriate floats:

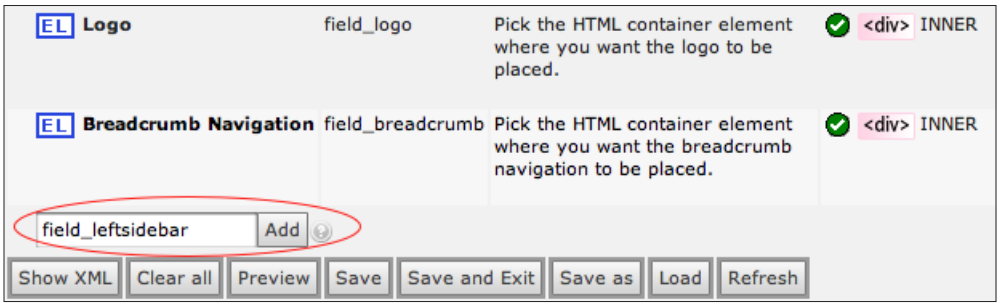
```
.sidebar {
   width: 200px;
}
#left_sidebar {
   float: left;
   margin-right: 30px;
}
#right_sidebar {
   float: right;
   margin-left: 30px;
}
```

Adding columns to the data structure

Before we create the new template objects, we need to make sure that we have data fields available for mapping. Just like when we added the banner, logo, and other fields in *Chapter 3*, *Adding Custom Template Fields,* we are going to modify the data structure.

Just to refresh, we get into the data structure modification page through the TemplaVoila view in the backend. Next to the main template, just click on the **Update mapping** button. In the **Information tab**, we can get access to add columns to the data structure by clicking on the button labeled **Modify DS/TO**. We have already done this a few times, so we can confidently click on **OK** on the warning pop up and proceed into the editing page for the data structure.

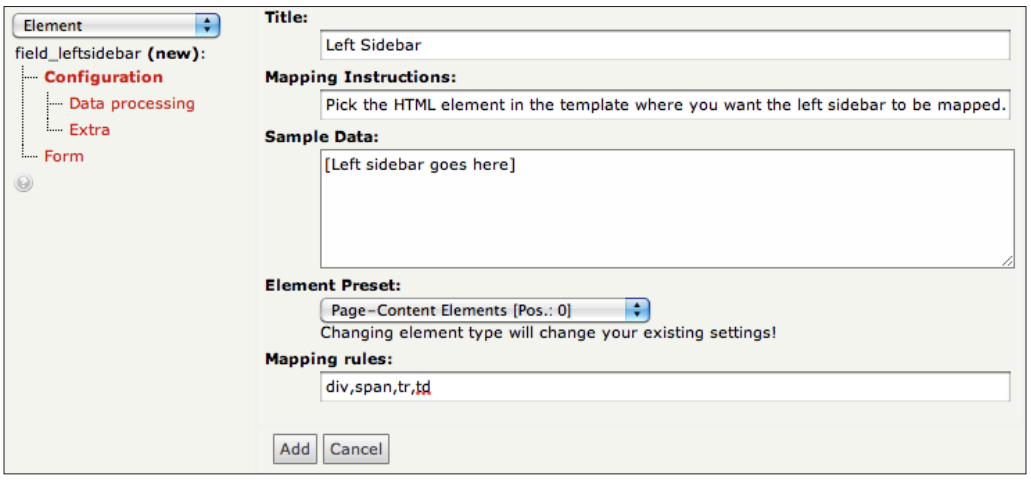
At the bottom of the page, we can add a new field called `field_leftsidebar`:



We are going to use the **Page-Content Elements** element preset so that our editors will see our new sidebar field in their TemplaVoila page view. Other than that, it's a basic element like the ones we've been creating, so we can configure it with the following settings:

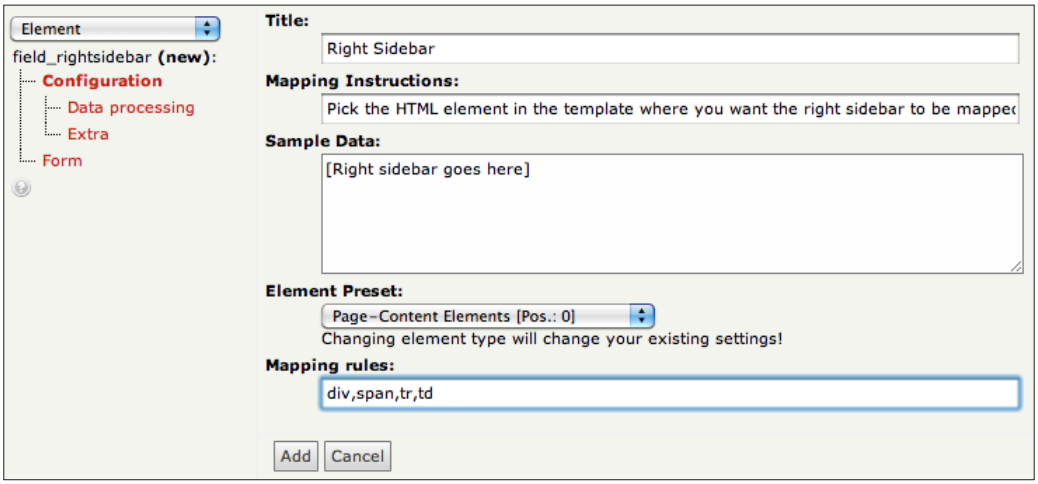
- **Field**: `field_leftsidebar`
- **Title**: `Left Sidebar`
- **Mapping Instructions**: `Pick the HTML element in the template where you want the left sidebar to be mapped.`
- **Sample Data**: `[Left sidebar goes here]`
- **Element Preset**: `Page-Content Elements`
- **Mapping rules**: `div,span,tr,td`

Our screen should look like this:



We can save and update the left sidebar and create a new right sidebar with a similar configuration:

- **Field**: `field_rightsidebar`
- **Title**: `Right Sidebar`
- **Mapping Instructions**: `Pick the HTML element in the template where you want the right sidebar to be mapped.`
- **Sample Data**: `[Right sidebar goes here]`
- **Element Preset**: `Page-Content Elements`
- **Mapping rules**: `div,span,tr,td`



Once we've created both fields, we can save our changes by clicking on **Save and Exit**.

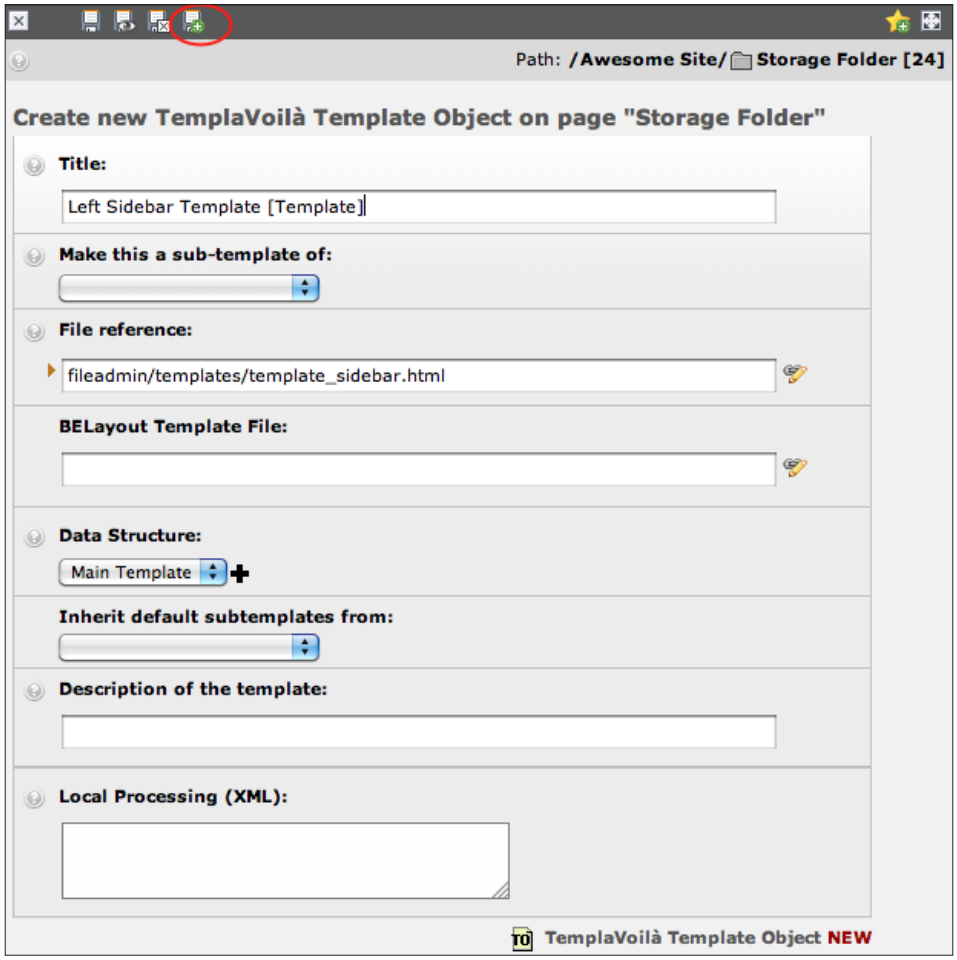
Creating new TemplaVoila template objects

Now that we have a new HTML file, updated CSS, and new fields available to us in the data structure, we can create our new template objects. There are three steps we need to go through to create each new template:

1. At the bottom of the main TemplaVoila page, click on the link labeled **Create new Template Object**.

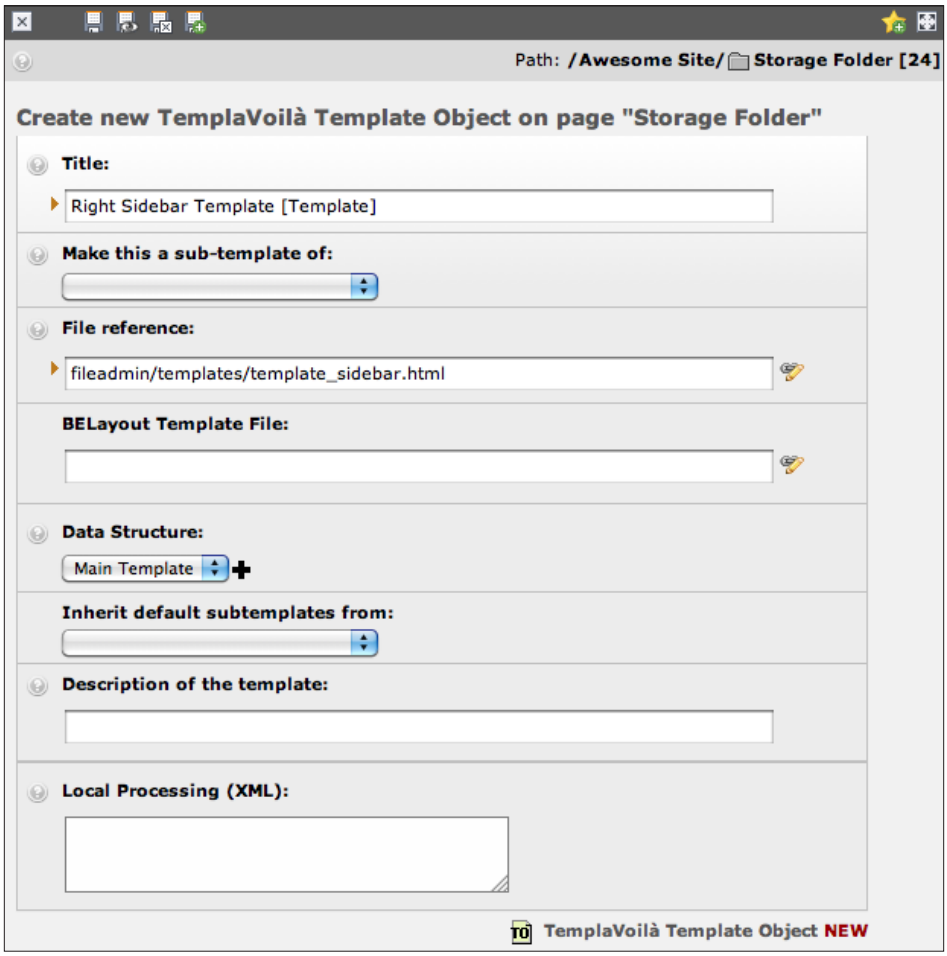
2. This brings up a configuration page for the new template. We are not dealing with subtemplates or special local processing yet, so we only need to fill out the title, choose a file reference (our HTML file) and select our data structure from the drop-down. Here are the values we will use for our example site:
    - **Title:** `Left Sidebar Template [Template]`
    - **File reference:** `fileadmin/templates/template_sidebar.html`
    - **Data Structure:** `Main Template`

3. If our configuration looks like the following screenshot, we can save it. As we need to create two more templates, we'll click on the "Save and create new icon" (circled in the following screenshot) to save our current template object and create a new one.

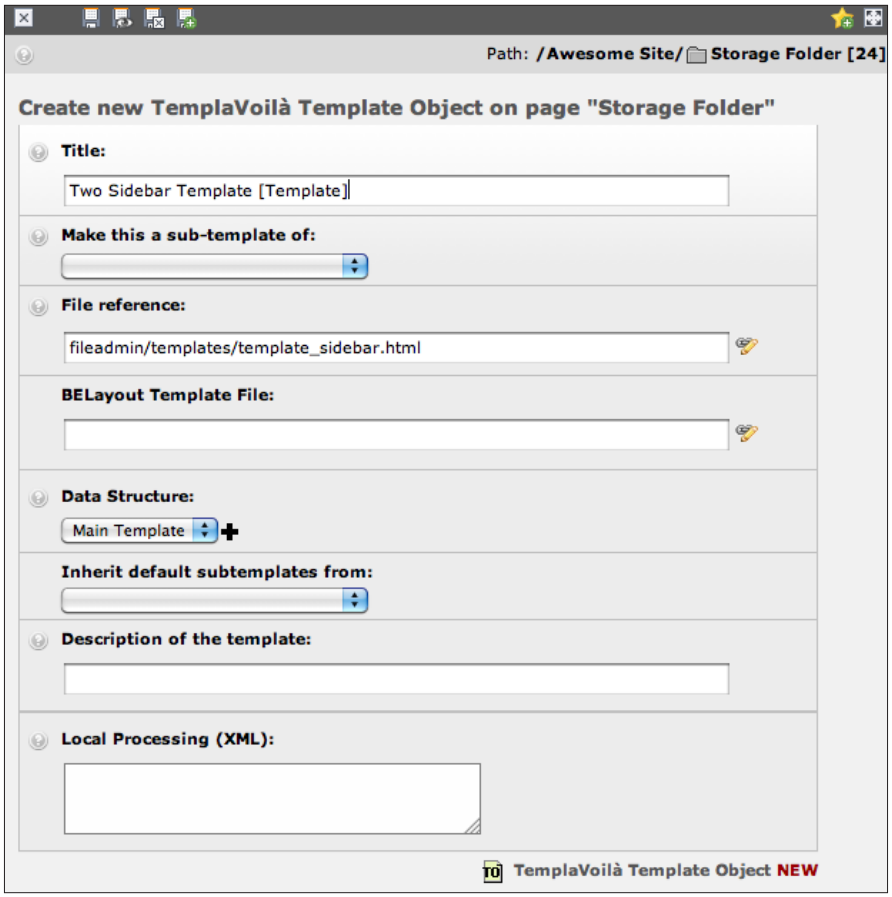


After we choose to save and create a new object, we can immediately start configuring our new one to show a sidebar on the right. Our settings will be exactly the same as **Left Sidebar Template [Template]**, but we will name it **Right Sidebar Template [Template]**. As an aside, it is important that we name our templates clearly for any site we are working on. Except for the possible icons that we can add later, this name will be the only thing an editor sees when they are trying to choose the right template for a page.

With the new title, this should be our configuration screen for the new template object:



If our configuration looks correct, we choose to save and create another document again. This time, we'll create our final template for now to show two sidebars. Once again, we will choose the same file reference and data structure; our only change will be naming our template **Two Sidebar Template [Template]**:



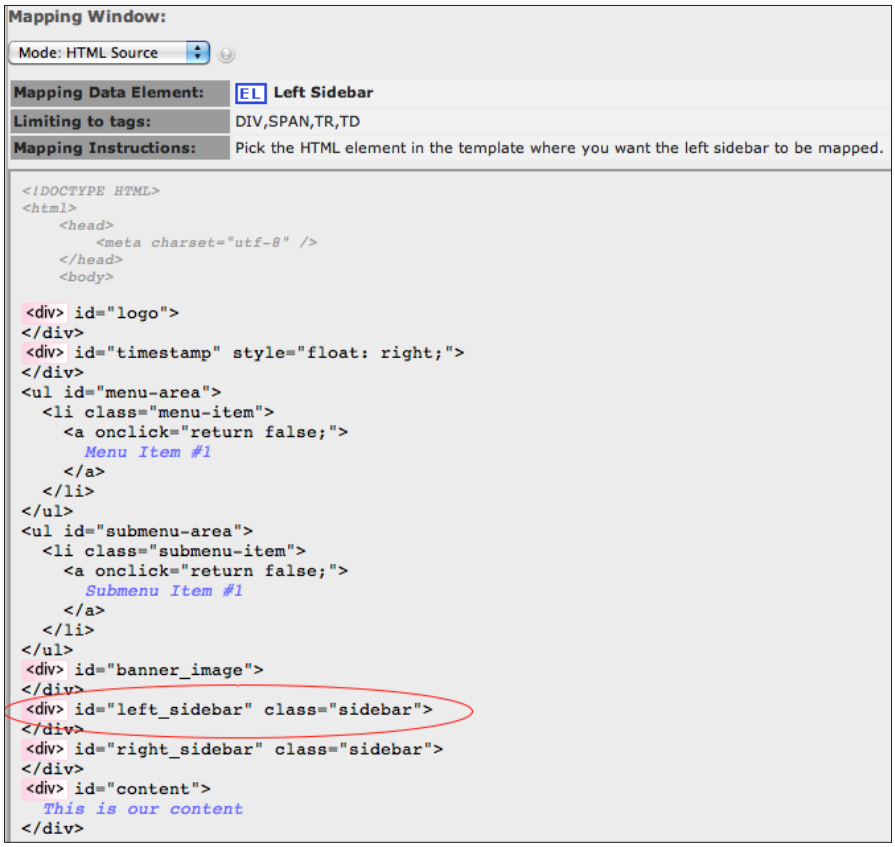
We do not have any more templates to create right now, so we can choose to save and close the document to be returned to the main TemplaVoila Control Center.

Mapping new template objects

Mapping the new template objects will be just like mapping our main template, so we can click on the **Remap** button on the main TemplaVoila page next to the new template object **Left Sidebar Template [Template]**. On the mapping page, we will map all of the elements to our new object as we have done for the main template:

1. Map **Root** to the `body` tag.
2. Map **Main Content Area** to the `div` with the ID `content`.
3. Map **Main Menu** to the `list` tag with the ID `menu-area`.
4. Map **Sub menu** to the `list` tag with the ID `submenu-area`.
5. Map **Banner Image** to the `div` with the ID `banner_image`.
6. Map **Timestamp** to the `div` with the ID `timestamp`.
7. Map **Logo** to the `div` with the ID `logo`.

The only additional mapping will be the new field labeled **Left Sidebar**, which we can map to the `div` with the identifier `left_sidebar` that we have circled in the following screenshot:



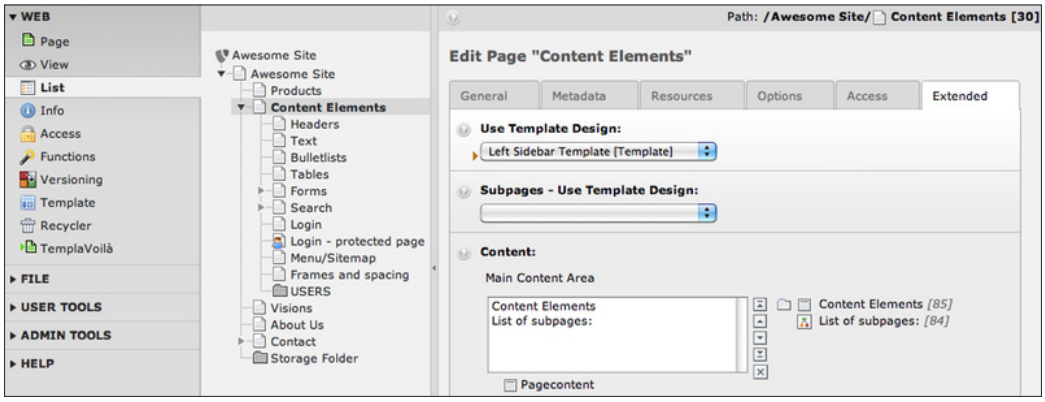
With our updated mapping, we can click on the **Save and Return** button to save all of our changes to the database and go back to the TemplaVoila Control Center to map the next templates.

From the main TemplaVoila page, we will click on the button to update the mapping of the **Right Sidebar Template [Template]**. Once again, we'll map everything like the earlier template objects with two obvious changes: we do not map the **Left Sidebar** element, and we map the **Right Sidebar** element to the `div` with the identifier `right_sidebar`. Once we have mapped everything, we save our changes and return to the main TemplaVoila page again.

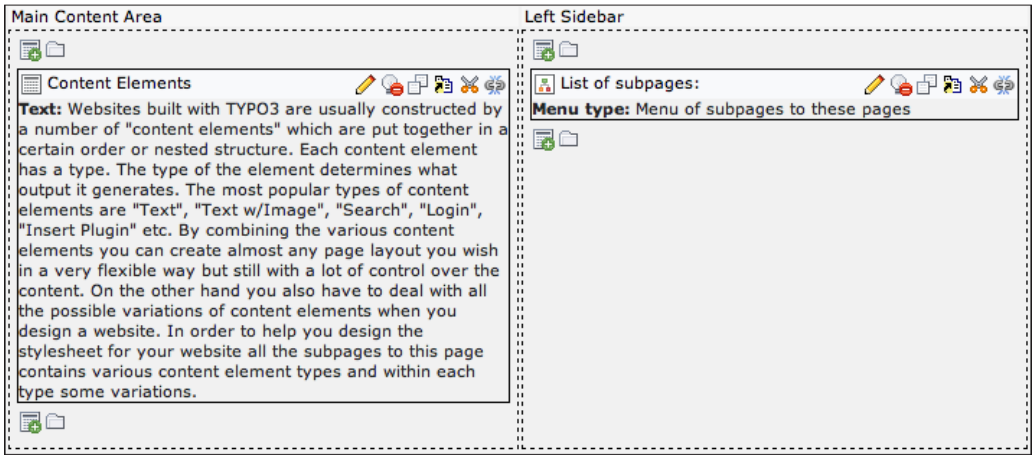
Now we can map our last template object, **Two Sidebar Template [Template]**. Again, we choose to update the mapping of our template object and map all of our elements almost exactly like our other template objects. The only difference this time is that we want to map both the **Left Sidebar** and **Right Sidebar** elements in this template object. Once we have clicked on **Save and Return** we have officially created and mapped three new templates for our editors to use. Of course, we'll need to tests them first.

Assigning a new template to our pages

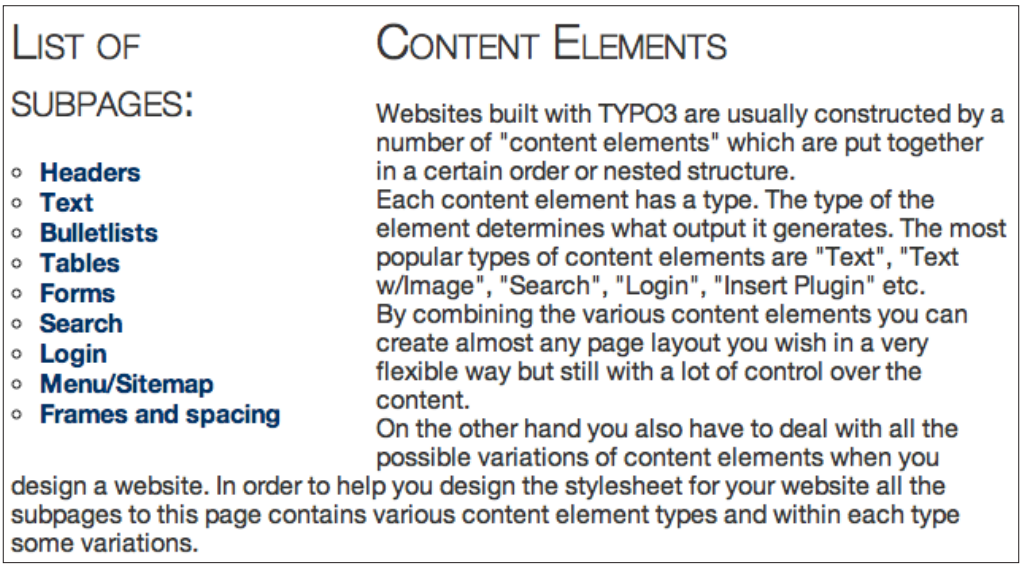
Now that we have some variety in our options, we can update some of our pages. The first thing we should do is updating the **Content Elements** page from our example site to use a sidebar. It's already got a content element to list the subpages that really should be in a sidebar on the left, and it has plenty of pages below it that we can play with. To change the template for the page, we can click on the pencil icon in the **Page** view or choose **Edit page properties** from the drop-down on the **Content Elements** page. We have already done this before when we set banner images or added resources for the menus, so should be comfortable jumping into the page properties. This time, though, we'll go to the **Extended** tab (like shown in the following screenshot). In the section labeled **Use Template Design:** we can choose **Left Sidebar Template [Template]** from the drop-down menu:



If we save and close the page properties, we can see that TYPO3 has added a new content area to the page view with the label from one of our new data structure elements, **Left Sidebar**. If we are running TYPO3 4.3 or higher (and if our browsers can handle AJAX), we can drag the **List of subpages:** content element over to the new column. If we can't use the updated AJAX backend for some reason, we can still cut and paste the element into the new section. In either case, we can move the element over and see that both columns have content in the backend page view:



When we refresh the page on the frontend of our site, we can see that the list is now floating on the left as we expected:



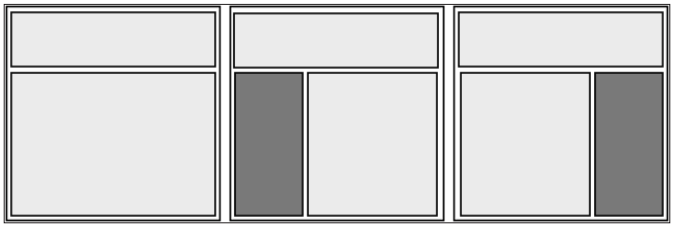
Creating icons for templates

Now that we've created our first set of templates, we should make it just a little bit easier for editors to pick the right one in the page properties. TYPO3 includes all available TemplaVoila templates in the appropriate drop-down menus, but we can create preview icons for our templates that will show up in the page properties. Preview icons are actually helpful for a couple of reasons.

Of course, preview icons can be created to show what the template will actually look like. Like I said before, naming our templates is important, but sometimes a picture works even better. A preview icon is whatever we create, but there are basically two options that are normally used: screenshots or wireframes. If we want, we can take screenshots of an example page for each template and shrink it down to illustrate the template. If that sounds too busy or possibly confusing for a thumbnail, another idea is just to create wireframes like the examples in this section. With a basic wireframe icon, we can still show how many columns are being used and basic layout, but we are not worried about colors or if a sidebar is distinguishable in a small image.

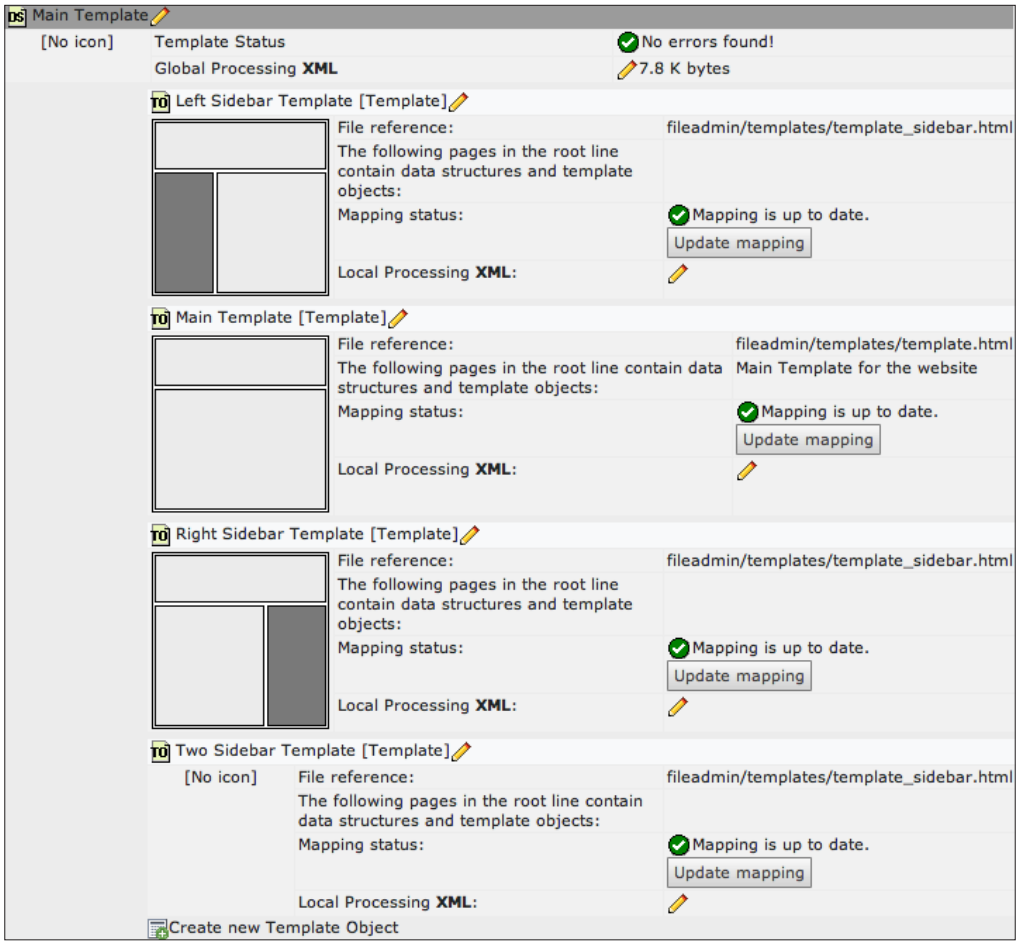
The other reason to use preview icons, though, is that they are displayed separate from the drop-down menu in the backend. If you have a large site with multiple sections, you could end up with dozens of specialized templates that your editors don't really need to see. Just because there is a special e-mail invitation template available, doesn't mean that you want to confuse your editors by putting it at the same level in the drop-down menu as the half-dozen templates that they may actually use every day. With preview icons, it's easy to highlight the common templates. If we only make previews for the handful of templates that editors need to use most often, they will rarely have to open the drop-down and see all the template objects that are used once a year or in special situations. In addition, you don't have to worry as much that they will accidentally assign the **2005 Christmas Donation Newsletter** template to the front page. It will be easier for you and the editors to have this separation between common templates and the rest.

Our first step is obviously to create a few icons in a graphics editor. I chose to create wireframe previews as you can do that in even the simplest editor with a couple of box shapes. We need to create icons for the main template and both single sidebar templates, but we'll ignore the template with two sidebars; our editors should rarely, if ever, have to use it for our simple website. We can make simple square preview icons like these for our purposes:

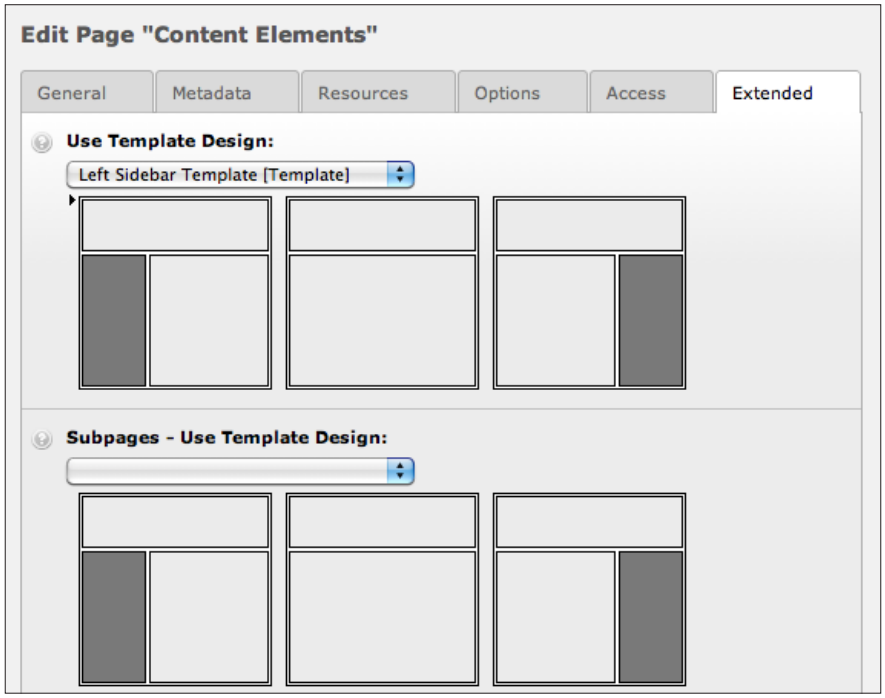


On the main TemplaVoila page in the backend, let's add an icon to the **Left Sidebar Template [Template]** by clicking on the edit icon. When we open up the editing view, we can see an area labeled **Attach preview icon (gif or png in 1:1 size):** with a blank box. We can upload a new icon with the **Choose File** button or choose one from our current directory structure by clicking on the folder icon to the right of the box. If our preview icon is named `left_sidebar_icon.png`, then it will look like the following screenshot after we have chosen it.

After we have saved the changes to **Left Sidebar Template [Template]**, we can add icons to **Right Sidebar Template [Template]** and **Main Template [Template]**. After saving our changes to all three template objects, the main TemplaVoila screen will show preview icons in the list of templates like this:



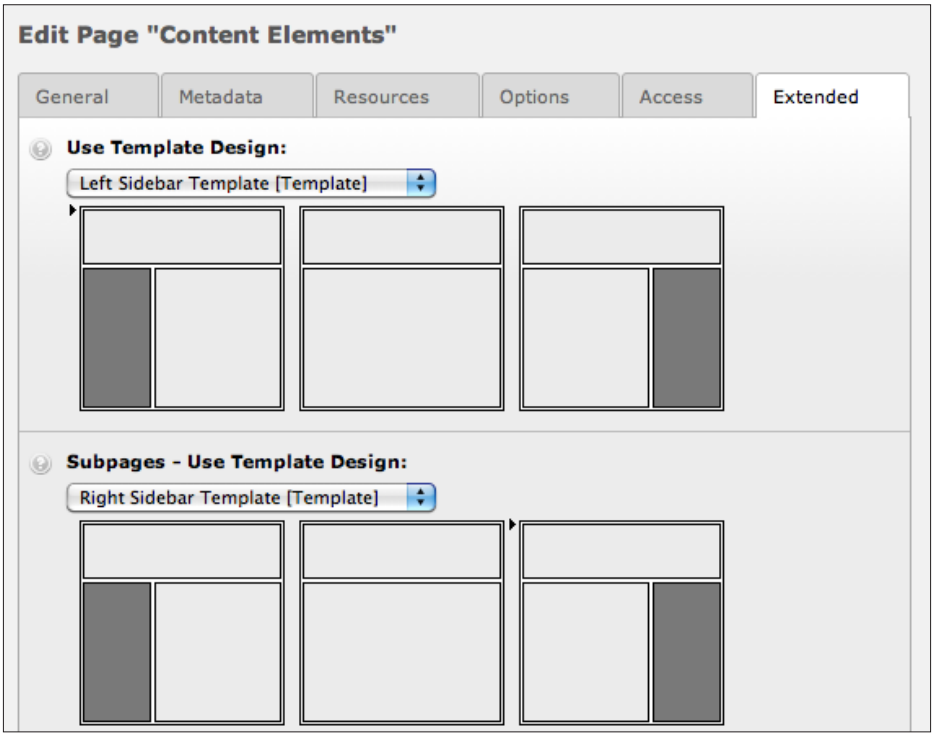
If we go to edit the page properties of the **Content Elements** page again, we can see that the preview icons are being used now. TemplaVoila is even indicating which template is currently being used with a small black arrow:



Assigning templates to subpages

If you look at the back end page view for any of the pages underneath the **Content Elements** page in the tree, you will see that they also have the **Left Sidebar** column added. Like TypoScript templates, TemplaVoila templates are always inherited from pages higher in the page tree unless they are specifically set. In our case, every subpage below **Content Elements** is now automatically using the **Left Sidebar Template [Template]** design.

If we want all of the subpages below **Content Elements** to use a different template by default, we can set a template for sub-pages in the **Subpages - Use Template Design:** section of the **Content Elements** page properties:

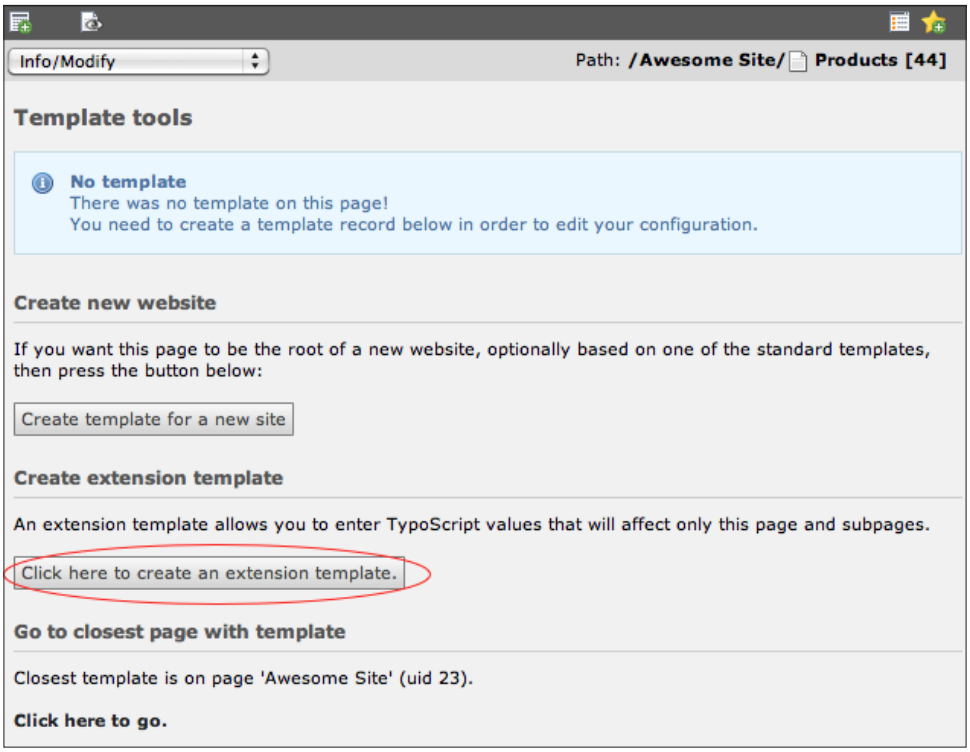


Of course, we can still override the template on any individual subpages that we want, but this means that they will default to **Right Sidebar Template [Template]** if nothing else is chosen. This is commonly done to set the internal pages of a website to a different default template than the home page.

Creating an extension template

Another way that we can change the look of individual pages beyond the basic layout is by overriding our TypoScript template settings with another kind of template concept. As we've seen, each website uses one main TypoScript template, and we've used this to define the style of our menus and assign values to our own TypoScript objects like the logo and the timestamp. We can change these values by assigning an extension template to any page, and it will affect that page and any pages below it with our TypoScript changes. Let's make some special modifications to the products page to make it more of a distinct store section under the main website.

1. In the backend, click on the **Template** link on the main sidebar to jump to the template editing view.
2. We need to edit the **Products** page, so we should choose it from the page tree after we are in the template view.
3. There is currently no TypoScript template assigned to the **Products** page besides the main site template, so we will see a screen like the one shown in the following screenshot. This page warns us that there is no current template and even points us to the closest page in the page tree with a template. We have the choice of creating a new website that will give us a brand new template and ignore all values above it in the page tree. Alternatively, we can create an extension template that will add or change values from the current templates for this page and the pages below it. We want to extend our current template without replacing it completely, so we need to click on the **Click here to create an extension template** button.



4. Once we choose to create an extension template, we are now presented with a new template to edit. The template is currently completely blank, but it is implied that all properties or values are already inherited from the templates before it. All we need to do is adjust the setup, so we'll click on the pencil icon next to the **Setup:** label to start editing the TypoScript setup values.

5. We are going to remove the timestamp from the **Products** page because our boss heard a consultant say that it was unnecessary in the catalog section of our site. Currently, the `lib.timestamp` object is assigned a value in the TypoScript of our main page. All we need to do is set the object to null in our TypoScript setup:

```
lib.timestamp >
```

6. We also want to replace the logo in the upper-left with an alternative version telling users they are in the store section of our website. We'll use the logo our designer handed us that looks suspiciously like he just added the word "Store" in grey Helvetica. We could eventually have a lot of pages under here, so we're also going to make sure that users who get too deep and click on the logo are brought straight back to the store page (instead of the main home page). We don't have redefine the entire `lib.logo` TypoScript object because it is already completely defined in the parent template; we just need to adjust two of the values in our setup with some TypoScript:

```
lib.logo.file = fileadmin/templates/logo_store.png
lib.logo.stdWrap.wrap = <a href="http://www.example.com/
products">|</a>
```

After our changes, we can refresh the **Products** page in the frontend and see a new logo (and missing timestamp):

Creating a printable template

Now that we have more options for our layouts and our design is looking better on the screen, we need to make our pages look a little better when they're printed. Our current site is pretty simple, so it doesn't break as bad as most when you try to print it. We can clean it up a little, though. We don't need menus on the printed page, for one thing, and the timestamp is already put in place by the browser when you print. We'll just remove those and maybe move some elements around to look better on paper.

Creating a print-only stylesheet

We're about to create our first subtemplate just for the printable version, but I'd be remiss if I didn't mention the most obvious tool we have to fix layouts for printing: CSS. By setting the `media` value to `print` in our CSS link, we can call a specific CSS file just for printing.

First, let's create a new stylesheet called `print.css` in the `fileadmin/templates` directory along with our normal stylesheets. We don't have to put in a lot of styling because we'll call it after the main stylesheet; we just need to hide the menus and timestamp. For good measure, we'll also align the logo on the right side of the page so that it looks cleaner on paper. The new file, `print.css`, is very basic:

```
/* Hide unnecessary elements */
ul#menu-area, ul#submenu-area, div#timestamp {
   display: none;
}
/* Move the logo to the right */
div#logo {
   float: right;
}
/* Don't let the content wrap around the logo */
div#content {
   clear: right;
}
```

Now, we can add the CSS file to our main template setup:

```
page.headerData.20 = TEXT
page.headerData.20.value = <link rel="stylesheet" type="text/css"
media="print" href="fileadmin/templates/css/print.css" />
```

As you can see, this looks just like our other stylesheet link, but we have changed the value to `20` and we have set the media attribute to `print` in the link. This means that this will load after the default CSS files and it will only load when the browser requests a `print` version of the document. If we try to print a page on the frontend now, we can see the changes:

Creating a subtemplate

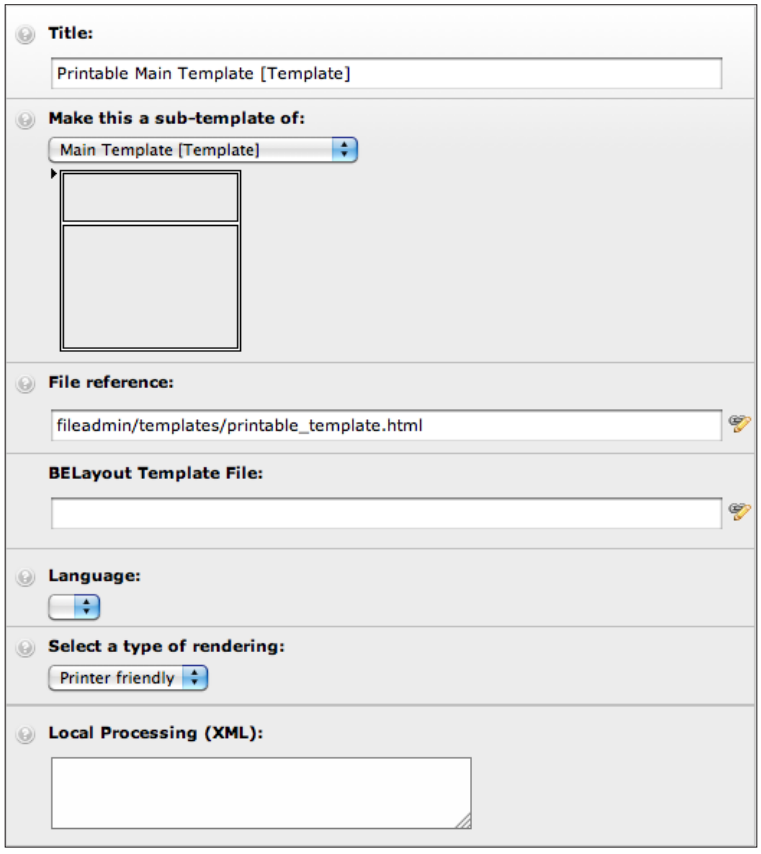
Of course, things are rarely this easy in the real world. For whatever reason, it is common to need a printer-friendly version of many templates. We can use special print templates to remove sidebars, fix ad spaces, or make a graphics-based layout friendlier to basic text. In our case, we are going to make an alternate version of our main template that uses tables instead of `div` tags. As annoying as tables can be, we have clients using Netscape Navigator 6.0 to print articles from our site, and our stylesheets are just not translating that well across the browsers.

First, we need to create a new HTML template in our `fileadmin/templates` directory named `printable_template.html` that has most of the same sections as our main template (minus the menus and timestamp) in a table format:

```
<!DOCTYPE HTML>
<html>
<head>
   <meta charset="utf-8" />
   <title></title>
</head>

<body>
   <table>
      <tr>
         <td id="logo"></td>
      </tr>
      <tr>
         <td id="banner_image"></td>
      </tr>
      <tr>
         <td id="content"></td>
      </tr>
      <tr>
         <td id="print_link"></td>
      </tr>
   </table>
</body>
</html>
```

You probably noticed the `print_link` cell above; we're going to use that in just a minute to show a dynamic link to a printable version of the current page in the frontend. For now, we need to finish creating our new template. Unlike the previous template objects we created, we want this one to be a subtemplate to the main template object. This means that, instead of standing on its own as a template object, it will be automatically available to all pages using the **Main Template [Template]** layout and will be called through special parameters. Like the other templates, we will click on the **Create new Template Object** link on the bottom of the main TemplaVoila Control Center page. In the configuration screen, we will set the normal title and file reference as well as setting it as a subtemplate of the main template and choosing a printer friendly rendering with the following options:

1. **Title**: `Printable Main Template [Template]`
2. **Make this a sub-template of**: `Main Template [Template]`
3. **File reference**: `fileadmin/templates/printable_template.html`
4. **Select a type of rendering:** `Printer friendly`



After we have filled out the configuration like the screenshot that we just saw, we can save our changes and exit to the main TemplaVoila page. We need to map the subtemplate before we forget, so we'll just click on the **Remap** button next to the printable template. This is basically the same as our main template (without a menu and timestamps), so we map the new sub-template to the current data structure with most of the same rules:

1. Map **Root** to the body tag
2. Map **Main Content Area** to the table cell with the ID `content`
3. Map **Banner** to the table cell with the ID `banner_image`
4. Map **Logo** to the table cell with the ID `logo`

Once we've mapped the elements, we can click **Save and Return** and test our new template. We can see the new subtemplate on any current page using **Main Template [Template]** on the frontend by appending the parameter `&print=1` to the URL in our address bar (for example, `http://example.com/index.php?id=73&print=1`).

Creating a printable link

A printable version of our pages doesn't do much good if nobody knows about it. Of course we know the syntax and we could spend time adding a printable link to the bottom of every page that we want it on, but that obviously doesn't make sense when we're using a powerful CMS like TYPO3. Using TypoScript, we can dynamically create a printable link at the bottom of every page that has a printable template.

Adding a printable link section to the templates

We have already added a section to map to on the `printable_template.html`, but we need to add a `div` to the main template HTML file. Go ahead and update `template.html` with a new `div` tag like this:

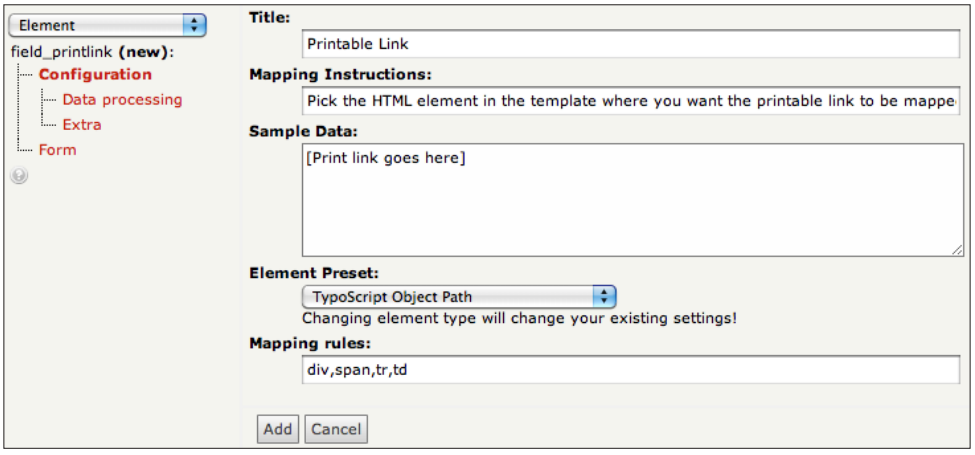
```
<!DOCTYPE HTML>
<html>
   <head>
      <meta charset="utf-8" />
   </head>
   <body>
      <div id="logo"></div>
      <div id="timestamp" style="float: right;"></div>
      <ul id="menu-area"><li class="menu-item"><a href="">Menu Item
#1</a></li></ul>
      <ul id="submenu-area"><li class="submenu-item"><a
```

```
href="">Submenu Item #1</a></li></ul>
      <div id="breadcrumb">Top Menu \ Next Page</div>
      <div id="banner_image"></div>
      <div id="content">This is our content</div>
      <div id="print_link"></div>
   </body>
</html>
```

Adding the printable link field to the data structure

This will have to be mapped to the templates in order to be usable, so we need to add another field to the data structure:

1. Go through the TemplaVoila section of the backend to the data structure editing page that we used to add the sidebar fields earlier.
2. Add a field named `field_printlink` to the bottom of the main data structure.
3. Like the menus and logo, this will be a TypoScript Object Path, so we can configure it with the following settings:
    - **Field**: `field_printlink`
    - **Title**: `Printable Link`
    - **Mapping Instructions**: `Pick the HTML element in the template where you want the printable link to be mapped.`
    - **Sample Data**: `[Print link goes here]`
    - **Element Preset**: `TypoScript Object Path`
    - **Mapping rules**: `div,span,tr,td`



4. Click on the **Add** button.
5. Choose **TypoScript** from the tree on the left.
6. Set **Object path** to `lib.printLink.`
7. Click on the **Update** button and then on **Save** to save the data structure changes to the database.

After we have saved our changes to the data structure, we just need to map the new field to the tags in the HTML files. For **Main Template [Template]**, go ahead and update the mapping to map the **Printable Link** field to the `div` with the ID `print_link`. For **Printable Main Template [Template]**, map **Printable Link** to the table cell with the ID `print_link`. When you're done, make sure you save all of the changes.

Generating a printable link with TypoScript

In the earlier TYPO3 tutorial Futuristic Template Building, they talked briefly about how to generate a printable link, but they didn't use the `typolink` object to make it as powerful as it could be. Like that tutorial, we are going to update the TypoScript template setup, but we are going to use a `typolink` solution that will work better with RealURL and multiple URL parameters.

To use the `typolink` object, we just need to setup the URL parameters and assign it to our `printLink` TypoScript object. `Typolink` will use our configuration to generate a complete URL. First, we can use the `page:uid` value to call the current page ID and assign it to the first parameter:

```
lib.printLink.typolink.parameter.data = page:uid
```

Next, we can start adding parameters to the link in a query string. We will need to use `addQueryString.exclude` to make sure we don't post the page ID twice in the parameters. Then, we can add the parameter `&print=1` to the link URL to call our printable template when the link is used:

```
lib.printLink.typolink {
   addQueryString = 1
   addQueryString.exclude = id
   additionalParams = &print=1
}
```

Finally, we can create a link to the non-print template that will remove the `print` parameters and only be displayed when the print parameter is already set:

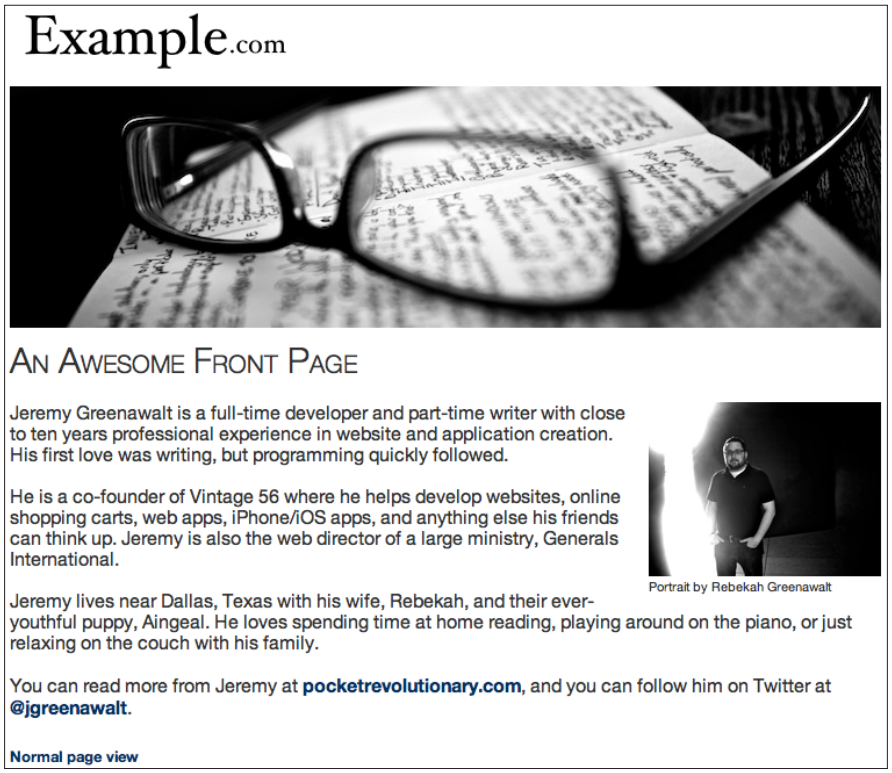
```
[globalVar = GP:print>0]
lib.printLink {
   value = Normal page view
```

```
      typolink {
         additionalParams >
         addQueryString.exclude  = print
      }
   }

   ## Return to global processing
   [global]
```

If we put all of the new TypoScript pieces together and eliminate some duplication by using curly braces, we can add this code to the bottom of our main TypoScript template setup:

```
   ## Print View Link
   lib.printLink = TEXT
   lib.printLink {
      value = Printable page view
      typolink {
         parameter.data = page:uid
         addQueryString = 1
         addQueryString.exclude = id
         additionalParams = &print=1
      }
   }

   ## Normal View Link (to run when print is greater than 0)
   [globalVar = GP:print>0]
   lib.printLink {
      value = Normal page view
      typolink {
         additionalParams >
         addQueryString.exclude  = print
      }
   }

   ## Return to global processing
   [global]
```

Now we can test this out on a frontend page. On any page using the main template, we can see a link labeled **Printable page view** on the bottom of the page. If we click on it, we see a printable version of the same page with a link at the bottom back to the normal view:



Summary

We've now started making some new templates that are more useful for internal content pages; more importantly, we know how to make any template objects we need to in the future that use existing data structures. We'll learn more about creating new data structures in the next chapter, but we've already quadrupled the possible layouts for our pages and added a printable version. On top of that, we learned how to add preview icons to make our editors' jobs easier when they take over updating the site layout themselves.

Using just these tools, we've come as far as many TYPO3 site creators who are successfully running websites, but we're only halfway done with this website. In the next chapter, we're going to create a whole new template (data structure and layout design) from scratch to create newsletter. As crazy as we can go with just the default data structure, there are times that we need to start with a whole new data structure; a newsletter is a great example. Before we do that, though, you should take a minute to show your boss the printable version links on our website. So few people actually make printable versions of pages that I'm sure your boss will be impressed you did it all dynamically.

6
Creating a Template from Scratch

We've gotten pretty good at modifying our existing templates and data structures to do whatever we want, but every once in a while we have to start over from scratch. We will sometimes need to create a new data structure because we are creating a section of our website that doesn't share the same data types as the rest of our website. As an example, we are going to create a new template for a newsletter. While the rest of our website may be centered around main content area, menus, and sidebars, our newsletter needs schedules, news articles, contact info, and other special elements. Of course, we could update our main data structure again like we did in the last chapter, but there's so little shared data that we would just make life more confusing for ourselves and anybody else who has to use our templates. We've been concentrating on the boss up to this point, but now we can start making life better for the editors. With the idea of helping the editors, we'll learn how to create a new template data structure from scratch. We will review some of the techniques from the last chapters, but we will be concentrating on the entire process of creating a data structure and some advanced TemplaVoila data elements known as "section" and "container" elements.

In this chapter you will:

- Lay out and design a new data structure from scratch
- Create your first data structure without the TemplaVoila "new site" wizard
- Learn about and create special section and container elements in our data structure for advanced template control
- Set up a new system folder in the TYPO3 page tree with a template extension and an example page

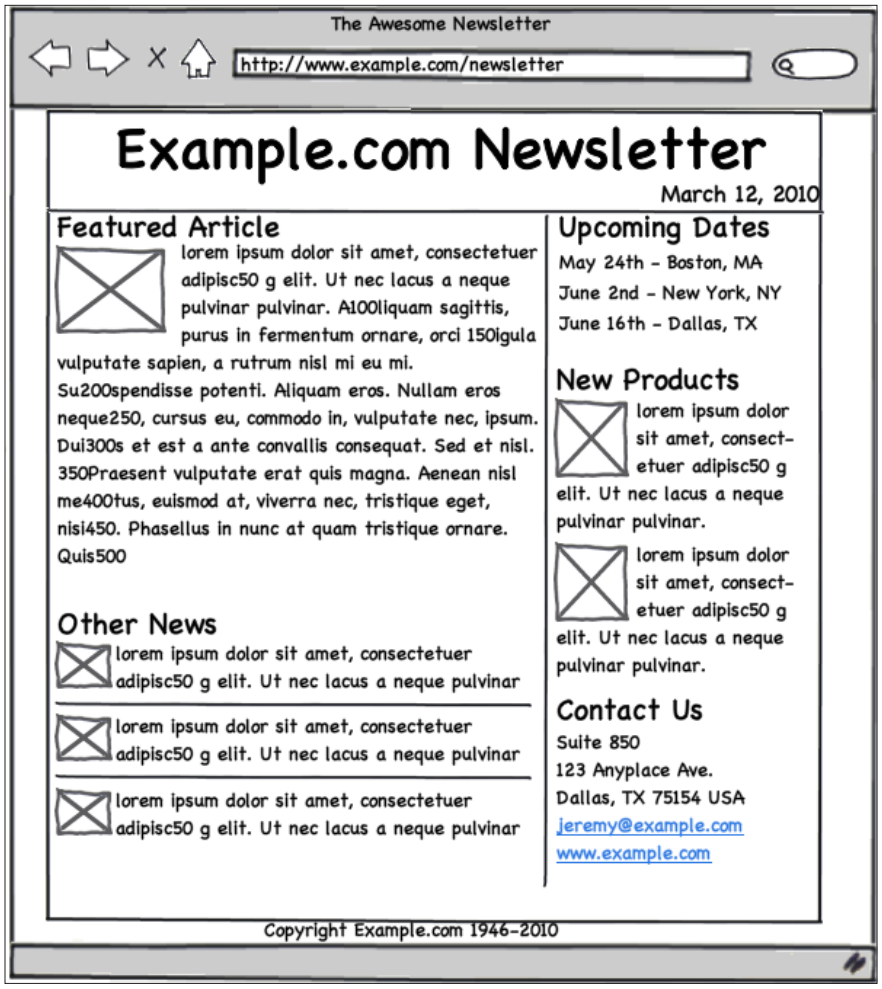
Designing the template

In this chapter, we are going to go through the entire creation of a new template including some of the processes and best practices that are necessary to build long-lasting, extensible templates. With that in mind, our first step to create a new template is to sit down with a piece of paper and decide what information we must communicate:

- We need a space for a main article.
- Along with a main article, we need a place for snippets of other, less important, news.
- Of course, every newsletter should have our letterhead banner and the date of the newsletter.
- We need some basic newsletter-like elements including a list of upcoming events, some advertisements, our contact information, and a footer.

Creating a wireframe

Now that we have a small list of requirements, we can create a basic "wireframe" mockup of our newsletter. A wireframe mockup is just a rough sketch that shows the content areas and functionality, but it does not try to convey the design elements like color, fonts, or graphics. It is a quick way of looking at the content without design distractions, and we can make wireframes with a specific tool such as Balsamiq Mockups (`http://www.balsamiq.com/`). We can also use diagramming programs such as Microsoft Visio (`http://office.microsoft.com/en-us/visio/`) or OmniGraffle (`http://www.omnigroup.com/products/omnigraffle/`) on the Mac or iPad. We can also use Photoshop or even paper. The tool we use doesn't really matter, but it is important to write down a basic sketch of where everything is going to be so we can make sure that all our required data is included; it also helps us to get an idea of what containers and tags will need to exist in our HTML template. Here is a wireframe generated by Balsamiq Mockups:



As you notice, we haven't done anything especially fancy with our template design; we've just laid out the data that we know needs to be represented in a logical way with a main section and a sidebar area.

Creating the HTML template

Now that we have our data elements designed and a basic wireframe, building the HTML template is quick and easy. We're really just creating code to match what we've already designed on paper. The only thing we have to do now is create container tags for any elements that we need to map and write HTML that can be used for an e-mail newsletter.

Our example is an HTML newsletter that could be sent to a mailing list (through a third-party service or a TYPO3 extension), so we want to make our HTML as simple and clear as possible for all of the different e-mail clients that our subscribers might be using with a few rules:

- No CSS layouts. Floating and block layouts are either completely unsupported or non-standard in many e-mail clients, so we'll have to use tables instead.
- We'll use inline CSS when possible. If we need a border for a table or cell, we'll use `style="border: 1px solid #000;"` in the HTML tag.
- Anything that may not work in an inline CSS declaration such as fonts will be declared in the head of our HTML, and we are going to include those values in the TemplaVoila mapping later.

There are more guidelines for designing an e-mail template than we can cover here. For more information, I recommend the articles on Campaign Monitor (`http://www.campaignmonitor.com/resources/`) or MailChimp (`http://www.mailchimp.com/resources`).

To create our HTML template, we only need to create a file named `template_newsletter.html` in the `fileadmin/templates/` directory and start creating our HTML based on our wireframe and e-mail requirements. We are going to go through all of the HTML right here, but you can download the complete HTML file from the Packt site (`https://www.packtpub.com/support`).

First, we can add the header information with some basic CSS for our fonts and margins:

```
<!DOCTYPE html lang="en">
<head>
   <meta charset="utf-8" />
   <title></title>
   <style type="text/css">
      * {
         font-family: "Helvetica Neue", Arial, Helvetica, Geneva,
sans-serif;
         font-size: 12px;
         line-height: 18px;
      }
      p {
         margin: 0px;
      }
      h2, h3 {
```

```
            margin: 0px;
         }
         h2 {
            font-size: 24px;
            line-height: 36px;
         }
         h3 {
            font-size: 18px;
            line-height: 24px;
         }
      </style>
   </head>
```

Next, we need to start adding the body our HTML. We'll use one main table for the entire template. We need rows for our banner and the date at the top, and we'll open up a new cell for the main content. Notice that we are using inline CSS to create our border under the date and setting the width of our main content section in the HTML tag:

```
   <body>
      <table align="center" width="550" style="border: 1px solid #000;">
         <tr>
            <td id="banner_image" colspan="2" align="center"></td>
         </tr>
         <tr>
            <td id="date" colspan="2" align="right"  style="border-
   bottom: 1px solid #000;"></td>
         </tr>
         <tr>
            <td id="main_content" width="350" valign="top">
```

For the main content section, we are going to create a nested table for the main article and news snippets:

```
   <table>
      <tr>
         <td id="main_article" style="padding-bottom: 15px;">
         </td>
      </tr>
      <tr>
         <td id="news" style="padding-bottom: 15px;">
            <h2 id="news_title"></h2>
            <div class="news_list"></div>
         </td>
      </tr>
   </table>
```

Now we can close the `main_content` cell, and create our cell for the sidebar. We will create another nested table for the sidebar:

```
</td>
<td id="sidebar" width="200" valign="top">
   <table style="border-left: 1px solid #000; padding-left: 10px;">
```

Now we will create our itinerary section to list upcoming events. We are adding HTML for each element of our itinerary (date and city) so that we can map them to fields in the page properties easier:

```
<tr>
   <td style="padding-bottom: 15px;">
      <h3 id="event_list_title"></h3>
      <div id="event_list">
         <div class="event">
            <span class="event_date" style="font-weight: bold;"></
span> - <span class="event_city"></span>
         </div>
      </div>
   </td>
</tr>
```

We will add sections for our products and contact information:

```
<tr>
   <td id="product_list" style="padding-bottom: 15px;">
      <h3 id="product_title"></h3>
      <div class="product"></div>
   </td>
</tr>
<tr>
   <td id="contact_info_section" style="padding-bottom: 15px;">
      <h3 id="contact_info_title"></h3>
      <div id="contact_info"></div>
   </td>
</tr>
```

Finally, we can close all of our tables and add a footer section at the bottom of the page:

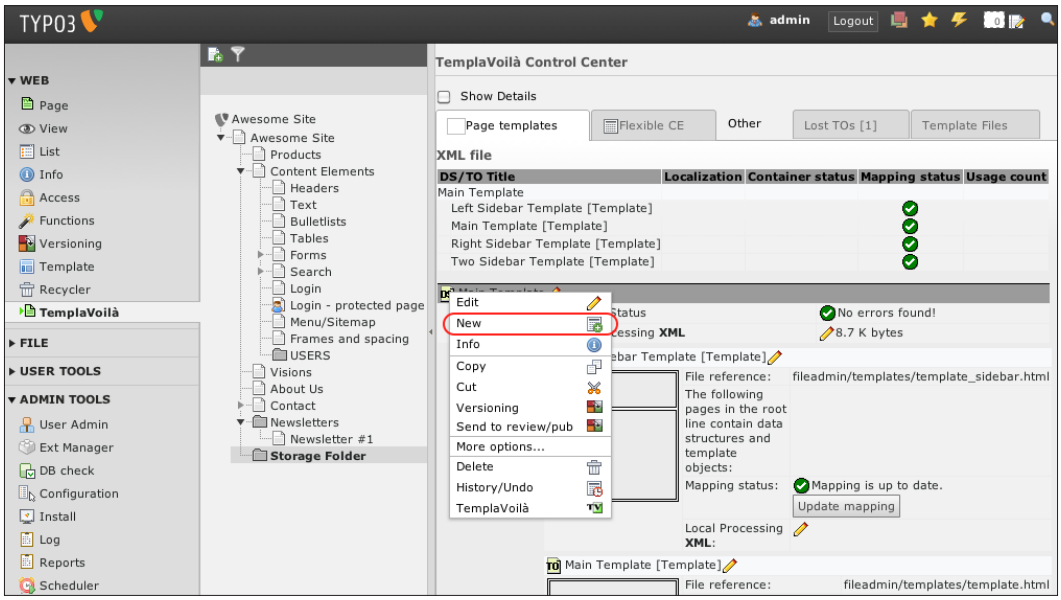
```
</table>
</td>
</tr>
</table>
<div id="footer" align="center"></div>
</body>
</html>
```

As you can see the HTML is just like we designed in the wireframe. The entire template is based on tables, and we have used HTML attributes where possible to set the padding and border of the different cells. Like we talked about, we have also included some very basic CSS in the head section to set the font sizes and reset the margins to 0 for paragraph tags. We have also created a `cell`, `div`, or `span` container with a unique identifier or class attribute for each section that will need to be mapped to a data element. Finally, we have created `h2` and `h3` header tags with unique identifier attributes for the titles that we can map in TemplaVoila.

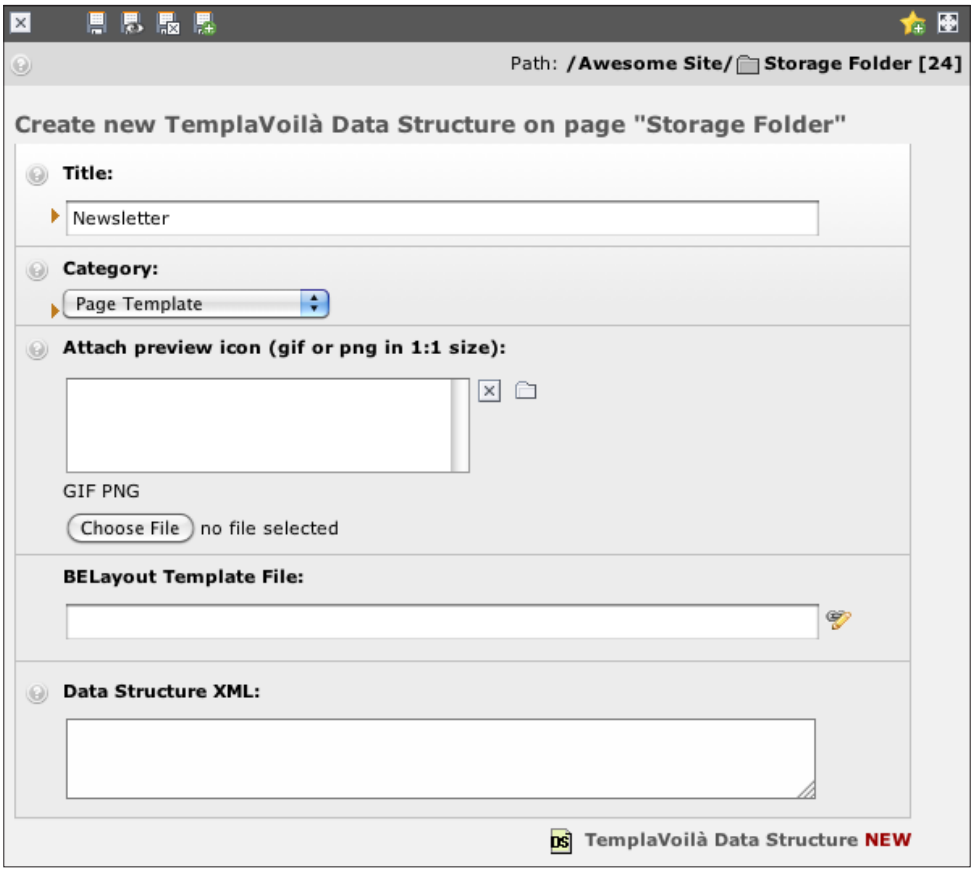
Creating the data structure

Now that we have everything planned out, we can create a data structure in TemplaVoila without the worry that we're going to have to update it a few times before we can really use it. We know about all of the main data elements that will need to be created, we know basically how they will work from the wireframe, and we have a copy of the HTML template that they will need to be mapped to as a reference guide.

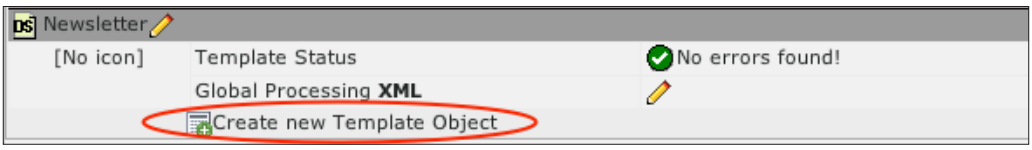
Of course, our first step is to create a blank data structure. In the TemplaVoila module in the TYPO3 backend, click on the **DS** icon for the **Main Template** to get a contextual menu. From the contextual menu, click on **New**:



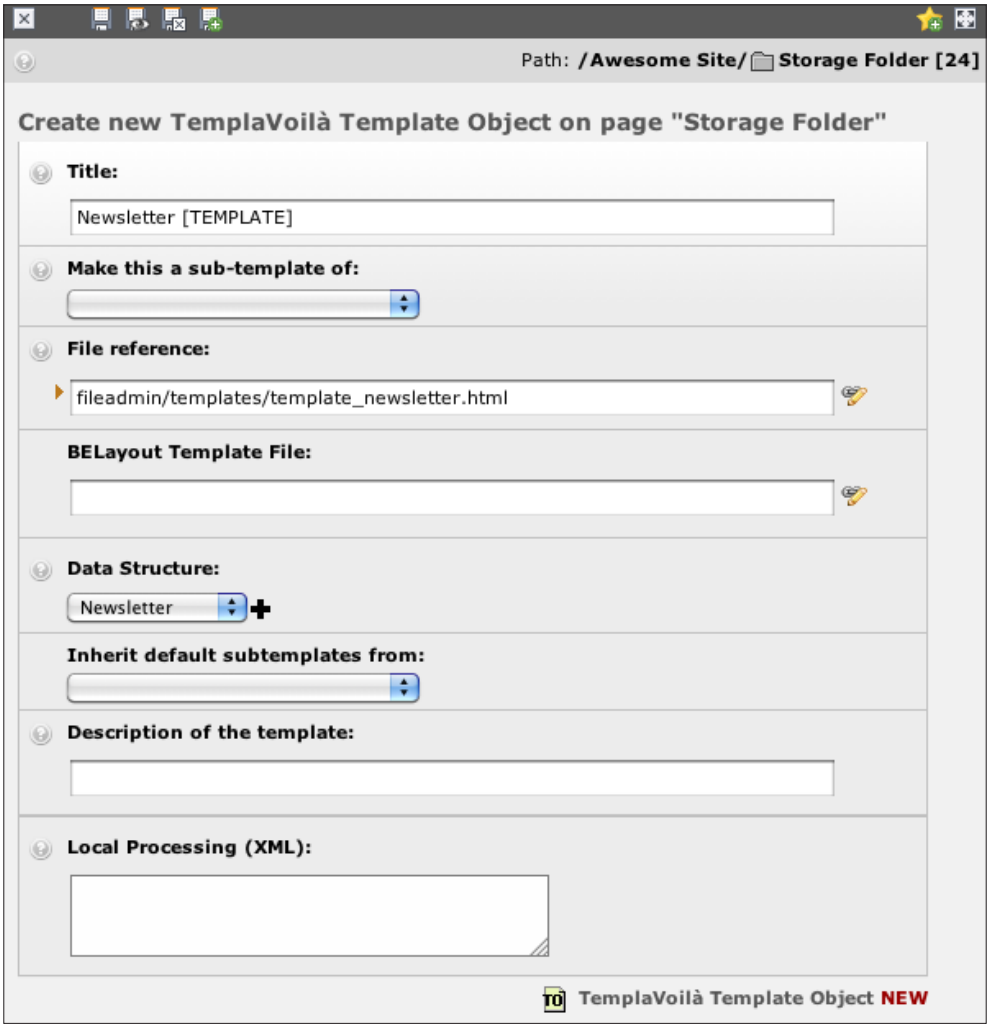
We will see a basic configuration screen for the new data structure immediately. All we need to fill in is the title and category for our new template. Our **Title** will be `Newsletter`, and the **Category** will be **Page Template** from the drop-down. We can leave the icon and XML sections blank for now and save our changes:



After we save our new template, we are ready to create the template object to map to. Under the **Newsletter** heading on the page, click on the link titled **Create new Template Object** just like we did in the last chapter to create our new template object for mapping:



Like the data structure, we can fill out a minimum of fields to save our new page. The **Title** should be `Newsletter [TEMPLATE]`, you can use the file tool to set the **File Reference** to `fileadmin/templates/template_newsletter.html`, and the **Data Structure** will be **Newsletter** for this template. Once you have filled in the fields like shown in the following screenshot, we can save and close the template object page:



In order to create the data elements for the first time, we will follow some of the same steps we took to add elements to the main data structure. This time, we will click on the **Remap** link under the new **Newsletter [Template]** object to go to the editing screen and click on the **Modify DS / TO** button on the **Information** tab. The only element that we should see already created is the **ROOT** container that will hold all of our other data elements:

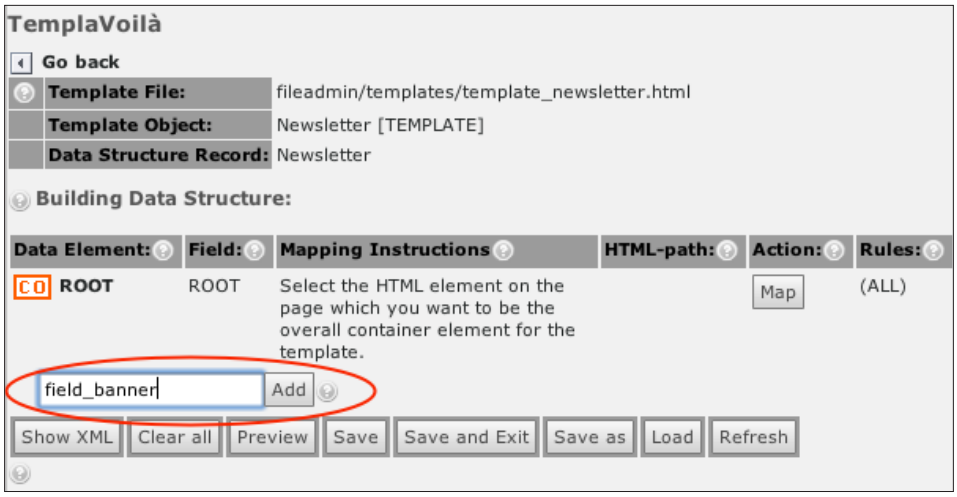
Creating data structure elements

Going down the list from our requirements, wireframe, and HTML, we can start creating new elements for this template. A lot of this is review, so we won't spend too much time on techniques we've already learned. For the first field, though, we will review the process of creating a data element one time.

The banner field

We'll start at the top with the banner image. We will make the banner image a TypoScript object so that it can be set in the TypoScript template setup. Like we've done before, this is helpful so that editors do not have to set it for each newsletter, but it can still be changed through the TypoScript template if necessary. We are calling it an image, but it will be a plain TypoScript object so we can always assign text or HTML values to it as well in the future.

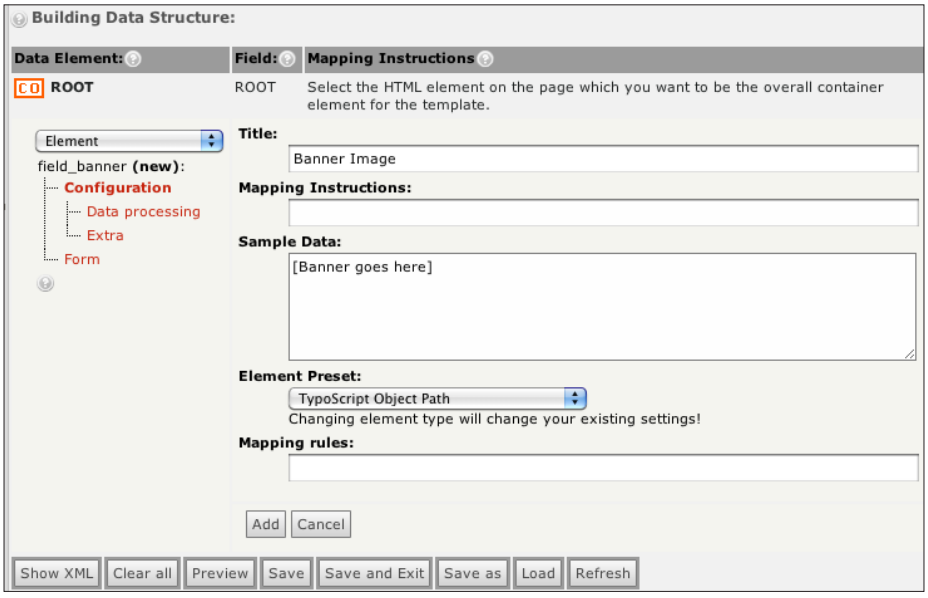
Remember, we create a new element by filling in a field name at the bottom of our editing page and clicking on **Add**:



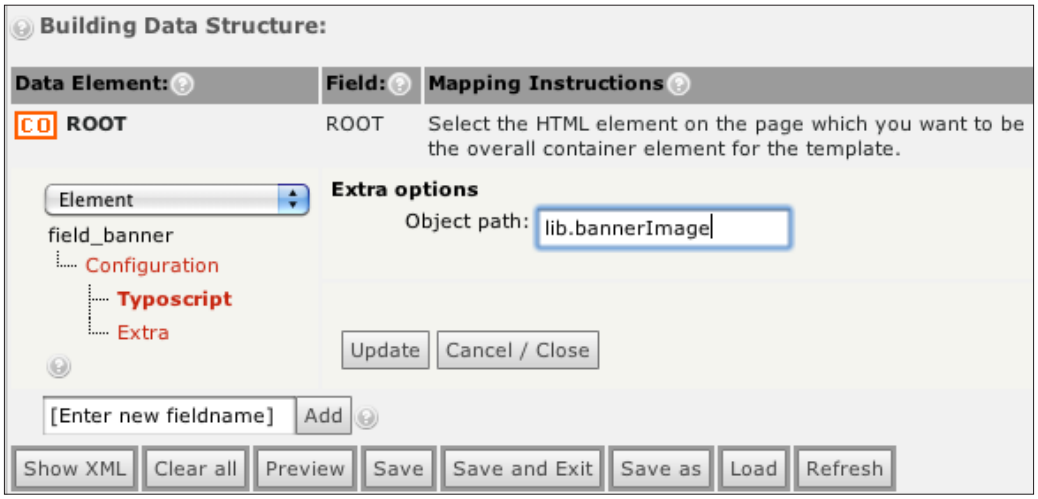
Go ahead and fill out the form with the following settings:

- **Field**: `field_banner`
- **Title**: `Banner Image`
- **Sample Data**: `[Banner goes here]`
- **Element Preset**: `TypoScript Object Path`

If your screen looks like the following screenshot, click on Add to create the element:



When the page refreshes, click on **Typoscript** in the left menu and set the **Object path** to `lib.bannerImage`:



Click on **Update** to save our changes.

The date field

Next, we can create the date field. The date needs to set by the editors for every newsletter, but it does not need to be a dynamic container for content elements like most of the dynamic elements we have made before. We can set the date to be a **Plain input field**, which means that editors will only need to input plain text through the page properties, and the template will take care of any styling. Our goal in this template is to standardize as much as possible, so using a basic input field makes the editors' jobs easier and cuts down on styling mistakes compared to using a rich text editor. We can create the date element with the following settings:

- **Field**: `field_date`
- **Title**: `Date`
- **Sample Data**: `[Date goes here]`
- **Element Preset**: `Plain input field`

The main article field

Next, we can create the **Main Article** element. Like the **Main Content** area in the main template, this element will display content elements of any type and be edited through the **Page** view. This element is the mostly like what we have been using this whole time, so we can go ahead and create it in the backend:

- **Field**: `field_main_article`
- **Title**: `Main Article`
- **Sample Data**: `[Article goes here]`
- **Element Preset**: `Page-Content Elements`

The news fields

Next, we can create the news elements. We could leave the news title hardcoded as part of the template, but let's make it editable through TypoScript instead so we can change it easier in the future or translate it for different languages someday. Go ahead create this new element as a TypoScript object before we create the news section:

- **Field**: `field_news_title`
- **Title**: `News Title`
- **Sample Data**: `[News title goes here]`
- **Element Preset**: `TypoScript Object Path`
- **Object path**: `lib.newsTitle`

Now we can create the news section. Like the main article section for the newsletter, we'll just allow content elements to be added to it in the **Page** view. Later on, we can restrict what the editors add a little more, but we'll just create a basic content element section for now:

- **Field**: `field_news`
- **Title**: `News`
- **Sample Data**: `[News goes here]`
- **Element Preset**: `Page-Content Elements`

The upcoming events title field

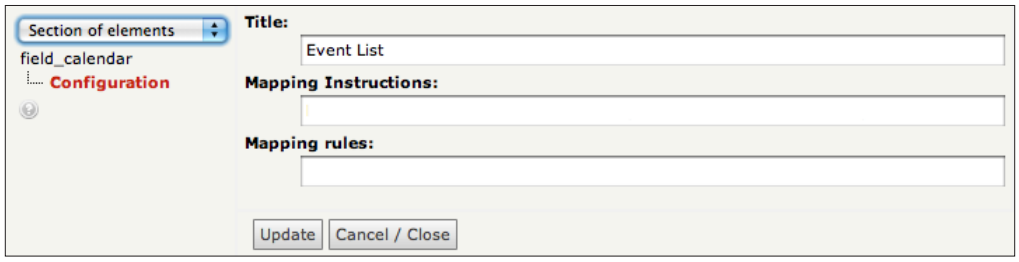
Go ahead create a new element as a TypoScript object for the title of the upcoming events section:

- **Field**: `field_event_list_title`
- **Title**: `Event List Title`
- **Sample Data**: `[Event list title goes here]`
- **Element Preset**: `TypoScript Object Path`
- **Object path**: `lib.eventListTitle`

The upcoming events list

Our next element, the list of upcoming events, will be different than what we've done before. Because we need to create a section for repeating elements (agenda items, in this case) we are going to introduce the **Section of elements** and **Container of elements** DS (Data Structure) elements. A section element is used to map a series of repeating elements inside of our data structure. For our example, we need to have one agenda section with an unknown number of events. By using a section element, we do not have to set the number of events while we are creating the template. To add the new section, we can create a new element with the fieldname `field_event_list`, and change the drop-down from **Element** to **Section of elements** (shown in the following screenshot):

- **Field:** `field_event_list`
- **Section of elements**
- **Title:** `Event List`



After we've added the section element, you can see that we can now add elements inside of it as well as to the **ROOT** container.

The event container field

Now we are going to add a container element inside of the **Event List** section. The container element, like our **ROOT** container, can hold different data elements inside of it. In our case, we are going to create a container element for the events in the upcoming events list that includes the date and city as separate elements. The reason we're going to build it like this is so that we can actually create a single container for each event that can then be repeated multiple times inside of the calendar. If we just added the event date and city elements to the **Event List** section without enclosing them in a container, they would not be grouped accurately and we would just have a series of dates and unconnected cities. Now that we understand the purpose of the container, we can go ahead and build it to see it in action. Under the **Event List** section, go ahead and create a new element:

- **Field:** `field_event`
- **Container for elements**
- **Title:** `Event`
- **Mapping Instructions:** `Pick the HTML container element where you want the event to be placed.`

The event date and city fields

When we save the new container, we can then add elements inside of it. Go ahead and create the event date element inside of the new **Event** container:

- **Field**: `field_event_date`
- **Title**: `Event Date`
- **Mapping Instructions**: `Pick the HTML container element where you want an event date to be placed.`
- **Sample Data**: `[Event date goes here]`
- **Element Preset**: `Plain input field`

After we've saved the event date field, we can create the event city field right below it:

- **Field**: `field_event_city`
- **Title**: `Event City`
- **Mapping Instructions**: `Pick the HTML container element where you want an event city to be placed`
- **Sample Data**: `[Event city goes here]`
- **Element Preset**: `Plain input field`

The product fields

Now that we've created the section and the content elements, the hardest part is behind us. We just need to create the rest of the data elements for our new template. Like the **News Title** element, we'll create a **Products Title** element so that we can map the title to a TypoScript object. We know how to do this pretty well now, so we'll just create a new element under the **ROOT** container with the following settings:

- **Field:** `field_product_title`
- **Title:** `Product Title`
- **Mapping Instructions:** `Pick the HTML container element where you want the product title to be placed`
- **Sample Data:** `[Product title goes here]`
- **Element Preset:** `TypoScript Object Path`
- **Object path:** `lib.productTitle`

For now, we'll allow the editors to add any content elements they need to into the products section of our newsletter, so we'll just make a basic content element section like the news section:

- **Field:** `field_products`
- **Title:** `Products`
- **Mapping Instructions:** `Pick the HTML container element where you want the products to be placed`
- **Sample Data:** `[Products go here]`
- **Element Preset:** `Page-Content Elements`

The contact information fields

We need to create an element for the contact information section. We need to create an element for the contact information section. We could combine the title and contact information into one TypoScript object, but that might make it harder for the editors if they need to customize the contact information (like preferred email address) for different types of newsletters in the future. We're already planning on having a few different newsletters for different departments, and we'll need to list different contact e-mails and phone numbers on the different ones. First, we'll make the title element:

- **Field:** `field_contact_title`
- **Title:** `Contact Info Title`
- **Mapping Instructions:** `Pick the HTML container element where you want the contact info title to be placed`

- **Sample Data:** `[Contact info title goes here]`
- **Element Preset:** `TypoScript Object Path`
- **Object path:** `lib.contactTitle`

Next, we'll create a TypoScript object element for the contact information:

- **Field:** `field_contact_info`
- **Title:** `Contact Info`
- **Mapping Instructions:** `Pick the HTML container element where you want the contact info to be placed`
- **Sample Data:** `[Contact info goes here]`
- **Element Preset:** `TypoScript Object Path`
- **Object path:** `lib.contactInfo`

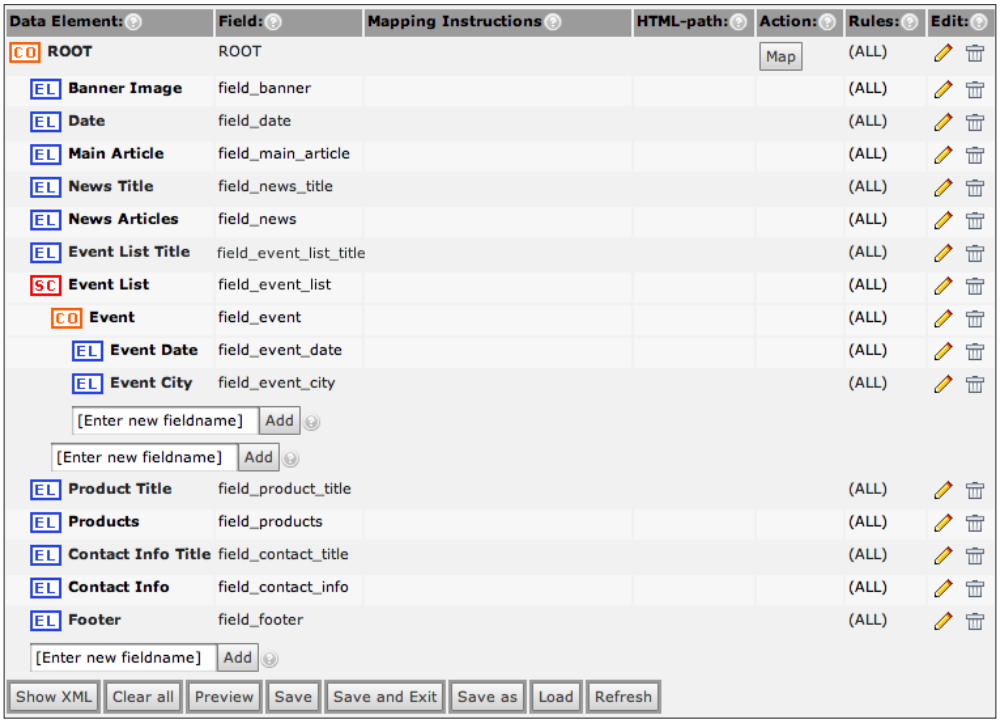
The footer field

Finally, we'll create a data element for the footer section of our page. This will also stay the same across groups of newsletters, but will need to be changed in the TypoScript for some folders. We'll create a TypoScript object for the footer as well:

- **Field:** `field_footer`
- **Title:** `Footer`
- **Mapping Instructions:** `Pick the HTML container element where you want the footer to be placed`
- **Sample Data:** `[Footer goes here]`
- **Element Preset:** `TypoScript Object Path`
- **Object path:** `lib.footer`

That may seem like a lot of data elements, but the point of this template is to spend the extra time upfront to save editors' time and prevent layout inconsistency in the future. As we broke everything down to small sections and used a lot of TypoScript options, the editors will only have three blank sections to edit in the **Page** view: **Main Article**, **News**, and **Products**.

If we've added everything correctly, our data structure should look like the following screenshot and we can use the **Save as** button to save our changes to the **Newsletter [Template]** template object and data structure.



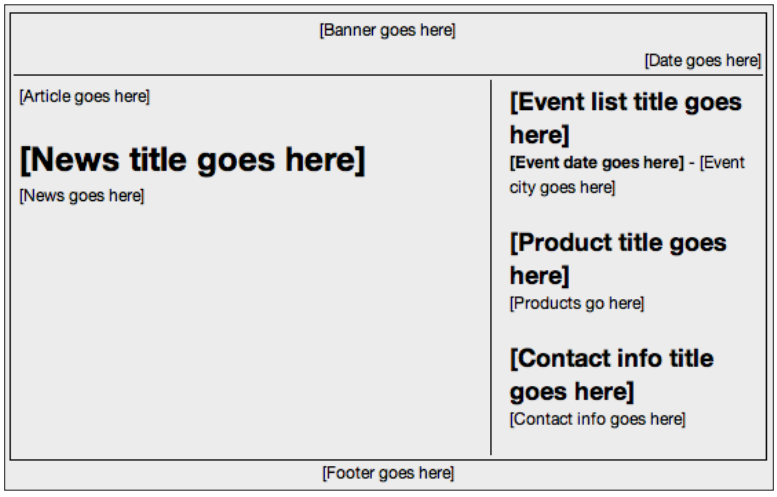
Mapping the template object

Once we have saved our new data structure, we can map all of our elements to the **Newsletter [Template]** template object. We've already mapped quite a few objects, so we'll just go through the mapping list to make sure that everything is mapped correctly.

1. Before we map the individual elements, we need to make sure that our in-line CSS is included in the header of the template. In the **Header Parts** tab, go ahead and check **<style>** tag checkbox and click on the **Set** button to save your changes:

2. Map the **ROOT** container to the `body` tag in **INNER** mode.
3. Map the **Banner Image** element to the table cell with the identifier **banner_image**.
4. Map the **Date** element to the table cell with the **date** identifier.
5. Map the **Main Article** element to the table cell with **main_article** identifier attribute.
6. Map the **News Title** element to the `h2` tag with the identifier **news_title**. Instead of using a `td` or `div`, we are mapping directly to the `h2` tag to keep the correct styling.
7. Map the **News** element to the `div` with the **news_item** class.
8. Map the **Event List Title** element to the `h3` with the **event_list_title** identifier.

9. For the **Event List** section, go ahead and map it to the `div` with the identifier or **event_list**. We need to make sure to map this to an HTML container that completely encloses whatever we need to map for the data elements inside of the **Event List** data section.
10. Map the **Event** container to the `div` with the class **event** in the **OUTER** mode. This is class in HTML because it will likely be repeating, and we just need to make sure again that we are mapping our container to an HTML tag that wraps around any data elements we will need to map. We also need to use the **OUTER** mode in TemplaVoila to ensure that it shows correctly when being repeated inside of the **Event List** section.
11. Map the **Event Date** element to the span with the **event_date** class in our template. You should notice that TYPO3 only allows you to map tags inside of the **Event** container.
12. Map the **Event City** to the span with the **event_city** class as an attribute.
13. Map the **Product Title** to the `h3` tag with the **product_title** identifier.
14. Map the **Products** element to the `div` with the class **product**.
15. Map the **Contact Info Title** to the `h3` tag using the identifier **contact_info_title**.
16. Map the **Contact Info** to the **contact_info** `div` tag.
17. Map the **Footer** to the **footer** `div` at the bottom of the HTML template.
18. Click on the **Preview** button, and choose the **Exploded Visual** mode from the drop-down. If everything is mapped correctly, you will see a preview that closely resembles our wireframe:



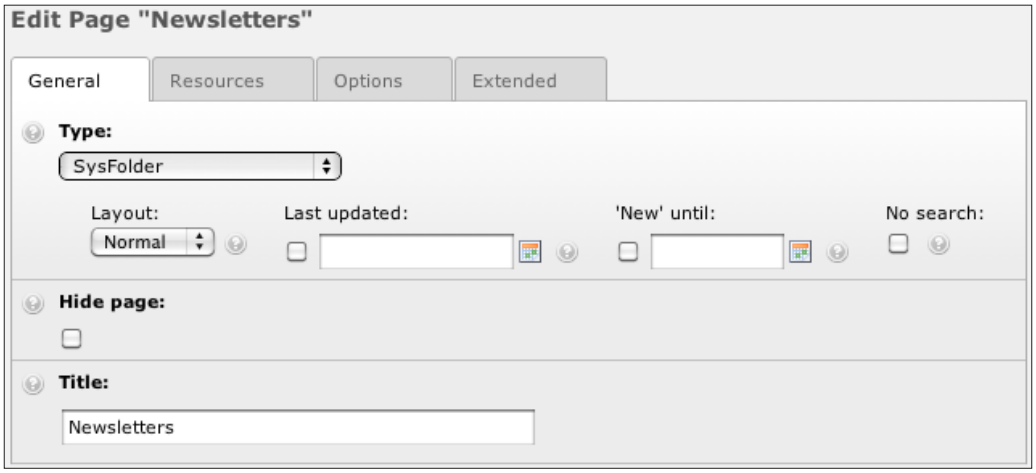
19. Click **Save and Return** to save our mapping.

Creating a folder in the page tree

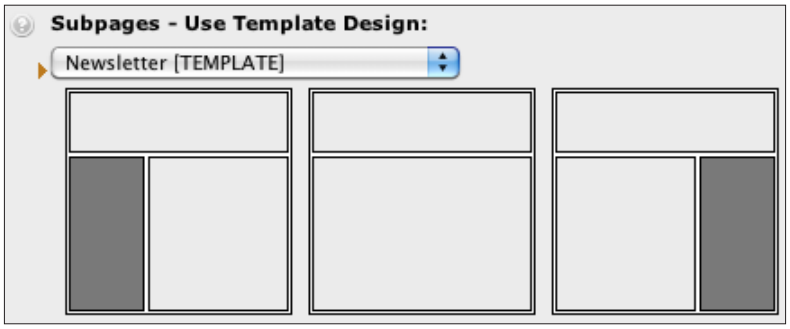
With a finished template, we need to start working on the page tree like an editor and make an area to hold our newsletters. In the page tree in the backend, click on the icon for the page right above the **Storage Folder** (should be **Contact**), and choose **New** from the contextual menu. On the next page, we just need to click on the **Page (after)** link to create our new page.

Instead of creating a page to be shown on the frontend, we want to create a system folder (`SysFolder`) to hold our future newsletters as normal pages. Unlike pages, system folders in the page tree do not appear in frontend menus, and they are not editable in the TemplaVoila page module. Instead, they are good places to hold pages or special content elements that will be accessed in other ways. In our case, we just want a place to store newsletters, and our editors will be using the generated HTML from the newsletter pages to send e-mails through a mailing list service.

If we choose `SysFolder` as the page type and set the title to **Newsletters,** our page will look like this:



Before we save and close our page settings, we can choose **Newsletter [Template]** as the default template for all subpages from the **Extended** tab:



Once the template has been chosen, we can save and close the page settings view.

Setting the TypoScript values

We need to create an extension template for the newsletters to clear out unnecessary headers and set the values for our TypoScript objects. As we move away from being the site builders, we might need to show this setup to our co-workers in the future, but for now go ahead and choose the **Template** icon from the far-right sidebar and select the **Newsletters** folder so we can make an example. Just like last time, TYPO3 will need to know if we are creating a template for a new site or simply an extension. Click on the button labeled **Click here to create an extension template** to create our new extension. This is a review of what we've already learned, we are just going to quickly add the following lines to the Setup section to clear our external CSS calls, eliminate our JavaScript, and set the TypoScript objects that we created in our template data structure:

```
## Clear our CSS and JavaScript calls
page {
  headerData.10 >
  headerData.20 >
  headerData.30 >
  jsFooterInline.10 >
}

## Set the banner image
lib.bannerImage = IMAGE
lib.bannerImage.file = fileadmin/templates/newsletter_banner.png

## Set the news title
```

```
lib.newsTitle = TEXT
lib.newsTitle.value = Other News

## Set the calendar title
lib.eventListTitle = TEXT
lib.eventListTitle.value = Upcoming Dates

## Set the products area title
lib.productTitle = TEXT
lib.productTitle.value = New Products

## Set the contact info title
lib.contactTitle = TEXT
lib.contactTitle.value = Contact Us

## Set the contact information with HTML formatting
lib.contactInfo = HTML
lib.contactInfo.value (
<p>Suite 850</p><p>1214 Rebekah Ave.</p><p>Dallas, TX 75154 USA</
p><p><a href="mailto:jeremy@example.com">jeremy@example.com</a></
p><p><a href="http://www.example.com">www.example.com</a></p>
)

## Set the footer as plaintext with a dynamic year. See the TSref for
more formatting
lib.footer = TEXT
lib.footer.data = date:U
lib.footer.strftime = %Y
lib.footer.wrap =  Copyright | Example.com
```

These are all the values that we need to set in the TypoScript, so we can save our changes and close the **Template** view.

Creating an example page

With the TypoScript changes saved we can create an example page inside of the **Newsletters** folder from the contextual menus in the page tree to test our template. Because we set the default template as **Newsletter [Template]** for all pages inside the **Newsletter** folder, we will already have the newsletter template applied, and we can see our new sections for content elements in the **Page** view:

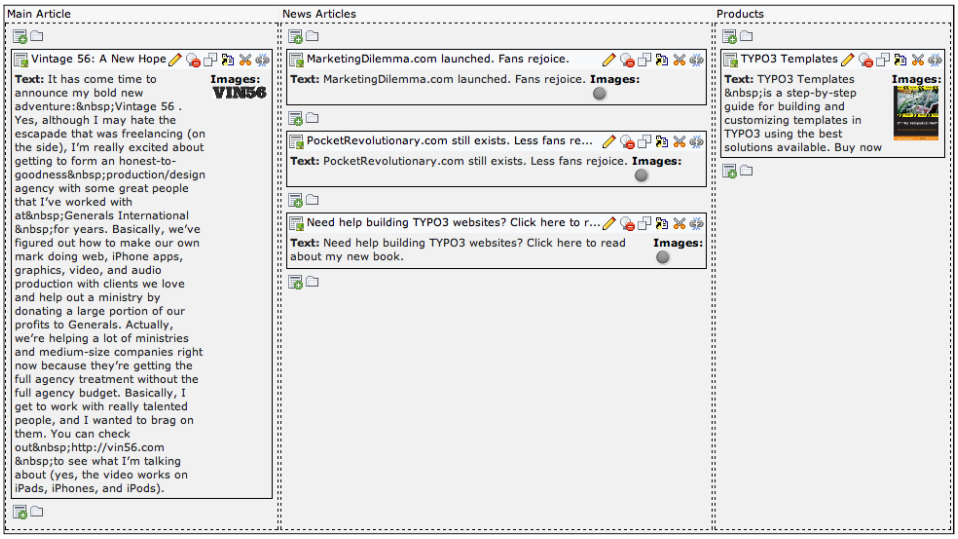
If we choose the **View** button from the sidebar, we can see that the template has already filled in a lot of our information from the TypoScript objects and HTML that we already mapped before we even start adding our own content:

When we see this much information already included in the template that means that we designed a bulletproof template. All of the output we see here is information that our editors do not have to fill in by hand for every newsletter from now on. We're already making their lives easier, and we can go ahead and finish creating an example page for them.

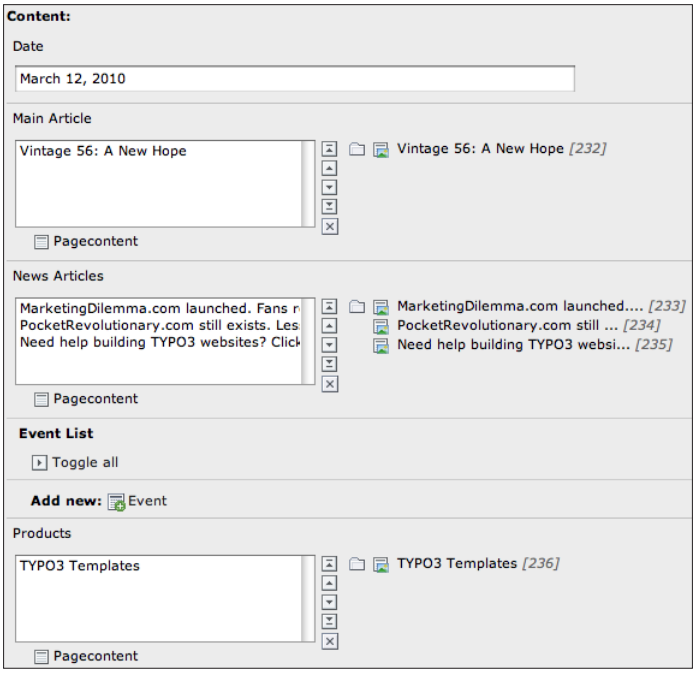
Adding test content

Now that we have a blank example page, we can add some content. We want to show the editors an example of the formatting that they should use, so we'll just go through and fill in every dynamic field with some information.

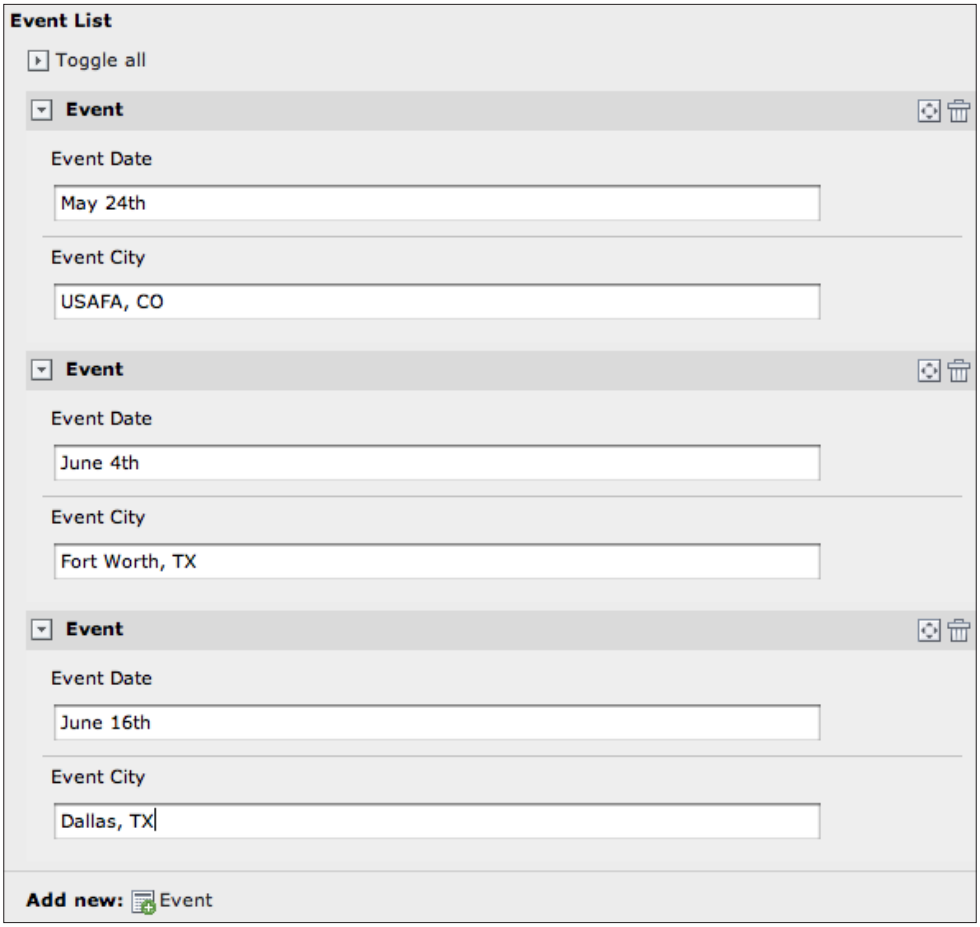
First, we can add some content elements to each column in the TemplaVoila page module. We have already worked with page content elements, so we are not going to go through this step-by-step, but you can add some basic elements with graphics to your example newsletter now. This is what our page module looks like after we have added some content:



Next, we can add the date and event information to our newsletter. Our new fields are available in the **Extended** tab of the page properties. We can see the new columns that we created in the data structure in the **Content:** section; the **Main Article**, **News**, and **Products** will show the elements that we just created in the first three steps. Now, we can fill out the date of our newsletter:



Finally, we can add events to the calendar by clicking on the **Add new: Event** button in the **Event List** section of the page properties. We can add as many events as we need by clicking on the **Event** button for each new event, and we can use the arrows and trash can icons to move our events around in the list or delete them entirely. We could do something like this with flexible-content elements, which we will learn about in a couple chapters, but doing it this way can help structure our template to make sure that editors never forget to include the upcoming events in our pages and is faster than creating multiple content elements. Let's go ahead and add a few events as shown in the following screenshot and save our changes.



Summary

After we save our content updates to the example newsletter, we're done! We should be at least a little proud because we just created a whole new template from scratch including the HTML and data structure, and we can see the results in our new newsletter:

Example.com Newsletter

March 12, 2010

Vintage 56: A New Hope

VIN56 It has come time to announce my bold new adventure: **Vintage 56**. Yes, although I may hate the escapade that was freelancing (on the side), I'm really excited about getting to form an honest-to-goodness production/design agency with some great people that I've worked with at **Generals International** for years. Basically, we've figured out how to make our own mark doing web, iPhone apps, graphics, video, and audio production with clients we love and help out a ministry by donating a large portion of our profits to Generals. Actually, we're helping a lot of ministries and medium-size companies right now because they're getting the full agency treatment without the full agency budget. Basically, I get to work with really talented people, and I wanted to brag on them. You can check out **http://vin56.com** to see what I'm talking about (yes, the video works on iPads, iPhones, and iPods).

Other News

- **MarketingDilemma.com** launched. Fans rejoice.
- **PocketRevolutionary.com** still exists. Less fans rejoice.
- Need help building TYPO3 websites? **Click here to read about my new book.**

Upcoming Dates

May 24th - USAFA, CO
June 4th - Fort Worth, TX
June 16th - Dallas, TX

New Products

TYPO3 Templates is a step-by-step guide for building and customizing templates in TYPO3 using the best solutions available.
Buy now

Contact Us

Suite 850
1214 Rebekah Ave.
Dallas, TX 75154 USA
jeremy@example.com
www.example.com

Copyright 2010 Example.com

Once again, we probably need to talk about why we created this template from scratch. You might be thinking that this example was a little contrived just so that we could experiment with creating data structures with different elements, sections, and containers and using them in the real world; you're only partially correct. We created this template from scratch because this was a one-off template for a special purpose (email) that didn't have any of the requirements or uses of the default data structure that the TemplaVoila wizard created for us. We've learned something that will come in handy when we create specialized sections for our websites like newsletters and online stores, and we also learned how to create our template without the wizard if we feel the urge to skip its processing entirely in the future. Most importantly, we've seen that we can create templates from scratch whenever we need to, and it's no longer a complex process to be feared.

We've come a long way with our template, and we've definitely made the back end editing easier by creating very defined data elements to be filled in, but we can still do more. In the next chapter we're going make the text areas easier to edit and modify our TemplaVoila data structures to customize the look of the TemplaVoila page view. So go ahead and show one of your coworkers how nice and structured our new template is, grab some coffee, and get comfortable. We're about to go very, very deep into the data structure specifications, but we'll come out with shiny, user-friendly editing.

7
Customizing the Backend Editing

In the last chapter we started making life easier for the editors with a custom template, and in this chapter we're going to continue to help our editors. We have two real goals in both of these chapters: customizing the workflow for editors and making sure that the frontend of our website is as consistent as possible. A moment ago we worked on both of those goals by making a template that enforced the original design and made filling the content as easy as fill-in-the-blank. This time we're going to work on the two main areas that editors spend their time on: the text editor and the Page module. We are going to edit the text editor to match the frontend styling in our website and add or remove functions to suit the website. We can add classes and functions that we think others might need, or we can take away functions (like tables or bright yellow text) that we don't want others in our organization to use. This is nice on small websites, but it's absolutely necessary on enterprise-level websites where many people are working on the content and consistent branding is essential. At the same time, we're going to edit the layout of the Page module in the backend to match what is shown on the frontend and even guide our users by making some sections more prominent or easier to read. Basically, we're going to spend one more chapter making sure that our site is as easy as possible to hand off and let others start to edit.

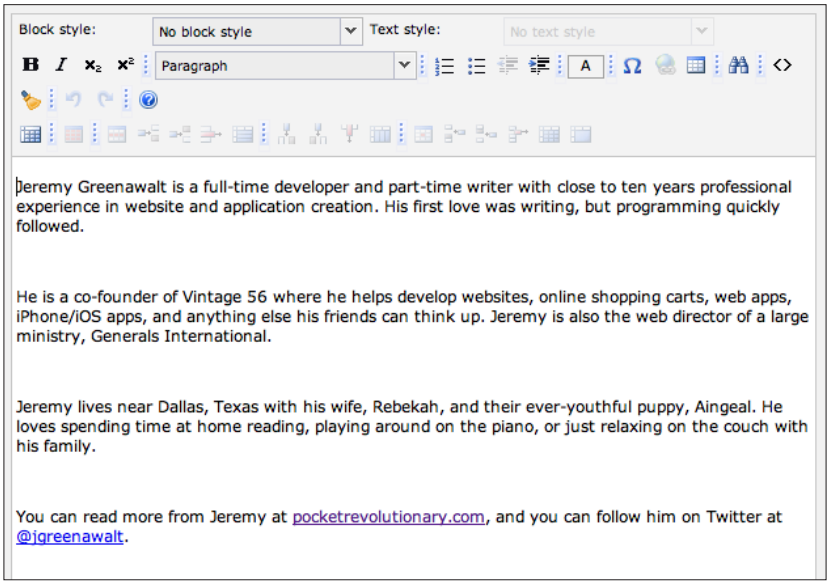
In this chapter you will:

- Learn about the default rich text editor in TYPO3
- Learn how to configure the rich text for a single page or a whole page tree
- Update the CSS in the rich text editor to match what users will see on the frontend
- Remove unwanted classes and tools from the text editor
- Add support for extra HTML tags and YouTube video to the text editor

- Learn how the configure the Page module layout of our templates with backend layouts
- Learn how to create a backend layout for data structures with multiple template objects
- Learn about static data structures and the future of TemplaVoila templates

Updating the rich text editor

While we are making life easier for the editors, we can work on one of the areas they have to spend the most time in: the Rich Text Editor (RTE). The rich text editor allows anybody editing text or content in the TYPO3 backend to add formatting and styling such as bold or italics without using special syntax or HTML, and see the results in the text area immediately. This is also known as a WYSIWYG (What You See Is What You Get) editor and we've used it in the previous chapters to edit our own content:



Out of the box, TYPO3 comes with the htmlArea RTE (extension key: `rtehtmlarea`), and it really is a good editor to work with. The problem is that it's configured by default to fit everybody, so it doesn't really fit anybody particularly well without a little bit of modification: it has it's own classes that we probably don't actually use, toolbars that may give too many options, and it blocks some handy tags such as the `embed` and `object` tags that we need for embedded video. Luckily, the htmlArea RTE is very configurable like everything in TYPO3 and allows us to override or add to almost all of its rules.

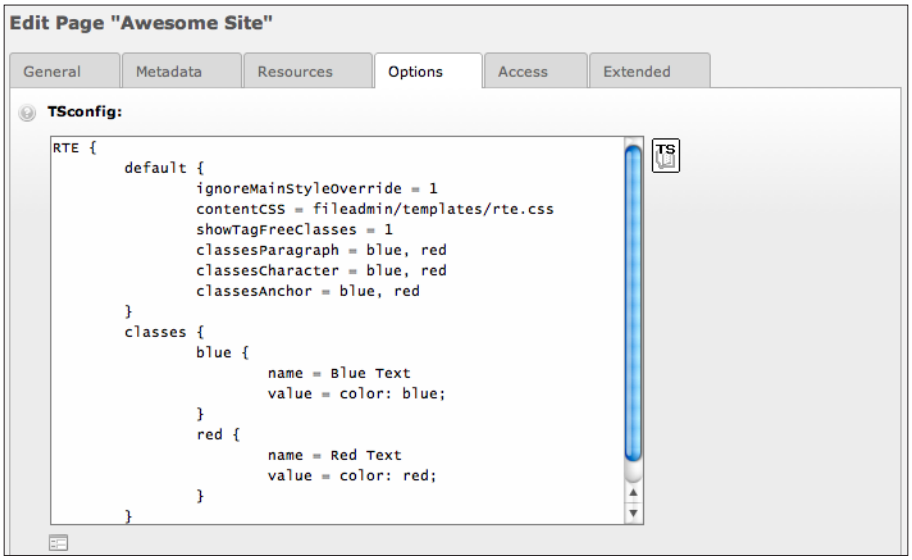
There's an entire TSref document (`http://typo3.org/documentation/document-library/extension-manuals/rtehtmlarea/current`), so we're not going to try to cover everything that is possible. The TSref is being updated with the new releases, so I recommend checking it out if you have any questions or want to get more information on configurations that we have only glossed over or skipped.

As a final note, some of the configuration properties in this section will work on other RTE options, but many are special for the htmlArea RTE. We are going to talk about the htmlArea RTE because it is already installed by default and has a lot of powerful options, but you may choose to install a different editor as an extension in the future. TYPO3 allows us to replace the RTE in the backend with TYPO3 extensions, and we might want to replace the default htmlArea RTE if we need to support older browsers or use special features like plugins for editing images. If you are using a different editor, you may need to look at its documentation for any differences in configuration.

Editing the TSconfig

Unlike the previous chapters, configuration of the RTE is done completely in the TSconfig for the user or the page. We've used the TypoScript template for frontend configuration and layout, but the TSconfig is mainly used for configuring backend modules in TYPO3 like the RTE. The rich text editor is a backend module, so we need to use the TSconfig to configure it for our editors.

The TSconfig can be modified through the **Page properties** view on any page in the **Options** tab (shown in the following screenshot):



Above you can see an example of the TSconfig with some of the modifications that we are going to look at for the RTE. Of course, you can edit the TSconfig on any page in the page tree that you would like, but we are going to be working on the Root page. Like templates, the TSconfig is inherited from the parent pages in the page tree, so we can modify all of the instances of the RTE in the entire site by modifying the Root page.

CSS properties

The first thing that we want to update in our editor is the CSS. Without special configuration, the htmlArea RTE uses its own default CSS to decide how the different options such as headings, bold, italics, and so on, look in the text area preview. We can update the TSconfig, to load our own external stylesheet for the RTE that more closely resembles our working frontend site; the editors will see the same basic styling for paragraphs and headings as the visitors and have a better idea of what their content will look like in the frontend.

According to the TSref on htmlArea, the following stylesheets are applied to the contents of the editing area by default in ascending order (we'll talk about each one in a moment):

- The `htmlarea-edited-content.css` file from the current backend TYPO3 skin (contains selectors for use in the editor but not intended to be applied in the frontend)
- A CSS file generated from the `mainStyleOverride` and `inlineStyle` assignments
- Any CSS file specified by `contentCSS` property in the page TSConfig

We don't need to worry about the first file, `htmlarea-edited-content.css`. The TYPO3 skin that styles everything in the backend controls it. By default, there is an extension named **TYPO3 skin**, and we can simply override any of the styles by using the `contentCSS` property that we will see in a minute.

The `mainStyleOverride` and `inlineStyle` properties are controlled by the htmlArea RTE, and TYPO3 generates a CSS file based on their settings in the extension. The `mainStyleOverride` contains all of the default text settings including sizes and font choices. If we are using our own CSS, we will want to ignore the original styles, so we are going to use the `ignoreMainStyleOverride` property in our TSconfig. To ignore set this property for our RTE, we can add the following line to the TSconfig on our main page:

```
RTE.default.ignoreMainStyleOverride = 1
```

The `inlineStyle` assignments in TYPO3 create default classes for use in the RTE. You've probably already noticed some of them as options in the style drop-downs in the editor (**Frame with yellow background**, **Justify Center**, **Justify Right**, and so on). If we override the `inlineStyle` assignments, the default classes are removed from the RTE. Of course, we don't need to eliminate these classes just to use our own, but we can do this if we are trying to clean up the RTE for other editors or want to make our own alignment styles.

Finally, we can use the `contentCSS` property to load our own external stylesheet into the RTE. To use our own stylesheet, we will use the following code in the TSconfig to use a new stylesheet named `rte.css` in our editor:

```
RTE.default.contentCSS = fileadmin/templates/css/rte.css
```

At this point, we're all pretty used to the syntax of TypoScript, but a review might be in order. In the previous line of TypoScript, we are updating the `contentCSS` property in the default instances of the RTE object with the new value `fileadmin/templates/css/rte.css`. If you understand that line, then everything else we're about to cover will be easy to pick up.

As we want to use our own CSS file and override the defaults at once, this will be the TypoScript for our TSconfig:

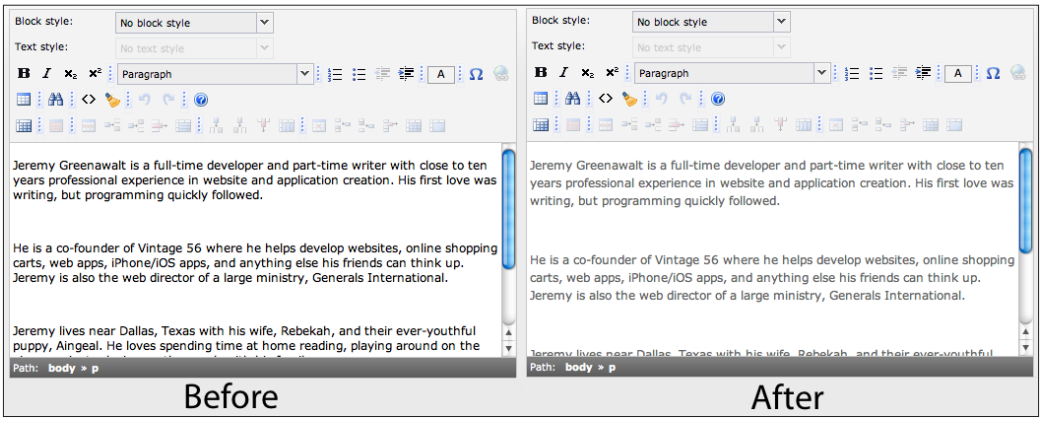
```
RTE.default {
   contentCSS = fileadmin/templates/css/rte.css
   ignoreMainStyleOverride = 1
}
```

Of course, the next thing we need to do is create our new stylesheet. We don't want to use the complete stylesheet that we are using for the frontend because it could be thousands of lines and will have a lot of formatting and page layout rules that we don't need for simple text editing. Instead, we can just copy the most important text and paragraph styles from our `style.css` file in the `fileadmin/templates/css/` directory into a new file called `rte.css`. In addition, we're going to add two new classes to `style.css` and copy them over; we'll create a class named `blue` to set the font color to pure blue and a class named `red` to set the font color to pure red. This should be the content of our new file, `rte.css`:

```
p, ul, div {
   color: #666;
   font-size: 12px;
   line-height: 18px;
}
h1 {
   font-size: 24px;
   line-height: 36px;
```

```
    margin-bottom: 18px;
    font-weight: 200;
    font-variant: small-caps;
}
h2 {
    margin-bottom: 18px;
    line-height: 18px;
    font-size: 18px;
}
h3 {
    font-size: 15px;
    line-height: 18px;
}
h4, h5, h6 {
    font-size: 12px;
    line-height: 18px;
}
ul, ol {
    margin: 0px 0px 18px 18px;
}
ul {
    list-style-type: circle;
}
ol {
    list-style: decimal;
}
td {
    padding: 5px;
}
:link, :visited {
    font-weight: bold;
    text-decoration: none;
    color: #036;
}
.blue {
    color: #0000ff;
}
.red {
    color: #ff0000;
}
```

With our new CSS we will notice at least a subtle difference in our RTE. In the following screenshot, we can see the original RTE on the left and the updated RTE on the right. As you can see, our text is now spaced out better and lightened to match the frontend:



As we have overridden the default classes without creating any new ones, the **Block style** and **Text style** drop-downs are both empty now. In the RTE, block styles are used for the "blocks" of our content such as paragraphs and headings while the text styles are applied directly to words or pieces of text within a larger block. Put simply, if we wanted a whole paragraph to be blue, we would use the **Block style** drop-down, and we would use the **Text style** drop-down if we only wanted a single word to be blue. Both of these drop-downs use our CSS classes for styling, so we'll go ahead and create our new classes in the TSconfig.

Classes properties

All of the classes in our external CSS file (`rte.css`) are made available as soon as we declared the file with the `contentCSS` property. If we want to use them, we just need to associate them with a style type (paragraph, character, image, and so on) for the default RTE. If we wanted to associate the `blue` and `red` classes with text styles, for example, we would add the following to our page's TSconfig:

```
RTE.default.classesCharacter = blue, red
```

Before we assign the classes to a type, we should declare them in the TSconfig so they show up properly in the RTE. By declaring the classes, we can set the titles we want to show in the RTE and how we want the titles to be styled in the drop-downs. Without declarations, the styles will still show up, but the titles will just be CSS class names. We want to declare them in TSconfig with specific title to make our editors' lives easier. We can add the following code to the TSconfig to declare the classes in our RTE and set more descriptive titles that will be shown in blue or red in the drop-down menus:

```
RTE.classes {
   blue {
      name = Blue Text
```

```
         value = color: blue;
      }
      red {
         name = Red Text
         value = color: red;
      }
   }
```

We can use CSS classes for more than just the text style, of course. The table below shows all of the main ways that we can associate and use classes in the htmlArea RTE, but you can read more in the htmlArea TSref. Go ahead and try some of them out with the example TypoScript lines that are shown.

RTE class properties

Properties	Description
`showTagFreeClasses`	If this value is set, classes in the `contentCSS` stylesheet without an associated tag can be used in the defined class lists. I recommend setting this at least in the example site now so that you do not have to declare `p.blue`, `div.blue`, `h1.blue`, and so on in your stylesheets. We can set it now with like this: `RTE.default.showTagFreeClasses = 1`
`classesParagraph`	The classes available through the **Block style:** drop-down box are defined here. We can make the classes `blue` and `red` from the example above available for block styling: `RTE.default.classesParagraph = blue, red`
`classesCharacter`	This works the same as `classesParagraph` for setting the classes that are available through the **Text style:** selector. Unlike the block style selector, the text style selector is used to format an individual string of text instead of the entire paragraph. Example: `RTE.default.classesCharacter = blue, red`
`classesImage`	This property defines the classes that are available for images. Example: `RTE.default.classesImage = red-border, no-border`
`classesAnchor`	This property defines the classes that are available in the **Insert Web Link** dialog box from the classes defined by the `RTE.classesAnchor` property. Example: `RTE.default.classesAnchor = blue, red`

Properties	Description

`classesTable`, `classesTD`	Like above, we can set the classes for tables or cells:
	`RTE.default.classesTable = no-headers, headers-on-top`
	`RTE.default.classesTD = blue, red`

Toolbar properties

Along with what shows up inside the content of the RTE, we can modify the toolbar itself to control what editors are able to see and use. By controlling the toolbar, we can make the RTE easier for editors and make sure that the branding and styling is consistent. The table below shows some of the most useful properties we can use to alter the toolbar available to editors. We can actually go much further in editing the individual buttons of the toolbar, but this is changing enough between major releases that I recommend using the TSref as a reference for your version of the htmlArea RTE.

Properties	Description

`showButtons`	You can set the buttons that are shown in the RTE using the `showButtons` property. The following buttons are available in htmlArea:
	`blockstylelabel`, `blockstyle`, `textstylelabel`, `textstyle`, `fontstyle`, `fontsize`, `formatblock`, `bold`, `italic`, `underline`, `strikethrough`, `subscript`, `superscript`, `lefttoright`, `righttoleft`, `left`, `center`, `right`, `justifyfull`, `orderedlist`, `unorderedlist`, `outdent`, `indent`, `textcolor`, `bgcolor`, `textindicator`, `emoticon`, `insertcharacter`, `line`, `link`, `image`, `table`, `user`, `acronym`, `findreplace`, `spellcheck`, `chMode`, `inserttag`, `removeformat`, `copy`, `cut`, `paste`, `undo`, `redo`, `showhelp`, `about`, `toggleborders`, `tableproperties`, `rowproperties`, `rowinsertabove`, `rowinsertunder`, `rowdelete`, `rowsplit`, `columninsertbefore`, `columninsertafter`, `columndelete`, `columnsplit`, `cellproperties`, `cellinsertbefore`, `cellinsertafter`, `celldelete`, `cellsplit`, `cellmerge`
	This property will override the default buttons shown, so only the buttons that we list will be shown. If we wanted to show only the two style selectors, we could set the property like this:
	`RTE.default.showButtons = blockstylelabel, blockstyle, textstylelabel, textstyle`

Properties	Description
`hideButtons`	This works like `showButtons`, but obviously it is used to hide specific buttons. If we have decided that editors should no longer be styling text with underlines, strike-throughs, subscripting or superscripting except through our CSS, we could hide them in the RTE: `RTE.default.hideButtons = underline, strikethrough, subscript, superscript`
`toolbarOrder`	We can also just change the grouping and order of the buttons that our editors see using the `toolbarOrder` property. In addition to the available buttons, we can add spaces, separators, and line breaks using the keywords space, separator, and linebreak. This is the default order of RTE buttons for reference: `blockstylelabel, blockstyle, space, textstylelabel, textstyle, bar, linebreak, fontstyle, space, fontsize, space, formatblock, bar, bold, italic, underline, bar, strikethrough, subscript, superscript, bar, lefttoright, righttoleft, bar, left, center, right, justifyfull, bar, orderedlist, unorderedlist, outdent, indent, bar, textcolor, bgcolor, textindicator, bar, emoticon, insertcharacter, line, link, image, table, user, acronym, bar, findreplace, spellcheck, bar, chMode, inserttag, removeformat, bar, copy, cut, paste, bar, undo, redo, bar, showhelp, about, linebreak, toggleborders, bar, tableproperties, bar, rowproperties, rowinsertabove, rowinsertunder, rowdelete, rowsplit, bar, columninsertbefore, columninsertafter, columndelete, columnsplit, bar, cellproperties, cellinsertbefore, cellinsertafter, celldelete, cellsplit, cellmerge`

Properties	Description
keepButtonGroup Together	As we saw above, a line breaker or a separator can delimit button groups. All buttons in a group are displayed on the same line if this property is set. Example: `RTE.default.keepButtonGroupTogether = 1`
hideTableOperations InToolbar	This can be a handy property to hide all table operations in the RTE toolbar. Example: `RTE.default.hideTableOperationsInToolbar = 1` Of course, all of these toolbar properties only control which buttons are visible or hidden, so an editor could still add a table in HTML mode.

disableContextMenu, disableRightClick	Either one of these properties can be set to disable the right-click context menu in the main editing section of the RTE. Example:
	`RTE.default.disableContextMenu = 1`
	
	or
	`RTE.default.disableRightClick = 1`

showStatusBar	This property is used to specify whether or not the status bar is displayed on the bottom of the main editor to show the current tags surrounding the text being edited. It is set by default but can be turned off: `RTE.default.showStatusBar = 0`

HTML editor properties

Finally, we can change the way that the RTE works with HTML. For example, the RTE uses paragraph tags (`<p></p>`) for all blocks in our text area, but we can replace this behavior with break tags (`<br />`) if we want. The RTE also strips certain tags, and we can change that with RTE properties as well. The following table shows the most common properties that we can use in the Page TSconfig to modify the HTML in the htmlArea RTE. Even if we don't need anything else in this section; using `allowTags` to allow embedded video can make our editor'' lives easier if we want to embed YouTube or Vimeo players in our website. All of this information is also available in the TSref for the htmlArea if you need an updated reference.

RTE editor properties	Description
`disableEnterParagraphs`	This property can be set to disable the creation of new paragraphs (using the `<p></p>` tags) when the *Return* key is hit in the editing window. If the property is set, break tags (` `) will be inserted instead. Example: `RTE.default.disableEnterParagraphs = 1`
`removeTrailingBR`	We can set this property to remove the trailing ` ` tags from block elements. Example: `RTE.default.removeTrailingBR = 1`
`removeComments`	If the `removeComments` property is set HTML comments will be removed when the content element is saved or the editor toggles to HTML source mode. Example: `RTE.default.removeComments = 1`
`removeTags`	If this property is set, tags listed will be removed when the editor saves or toggles the HTML source mode. If we wanted to remove the table tags from HTML editing, we could set it with this property: `RTE.default.removeTags = table, tbody, tr, td`
`removeTagsAndContents`	Like the `removeTags` property, `removeTagsAndContents` will remove the tags and the contents of the tags. Example: `RTE.default.removeTagsAndContents = table`

RTE editor properties	Description

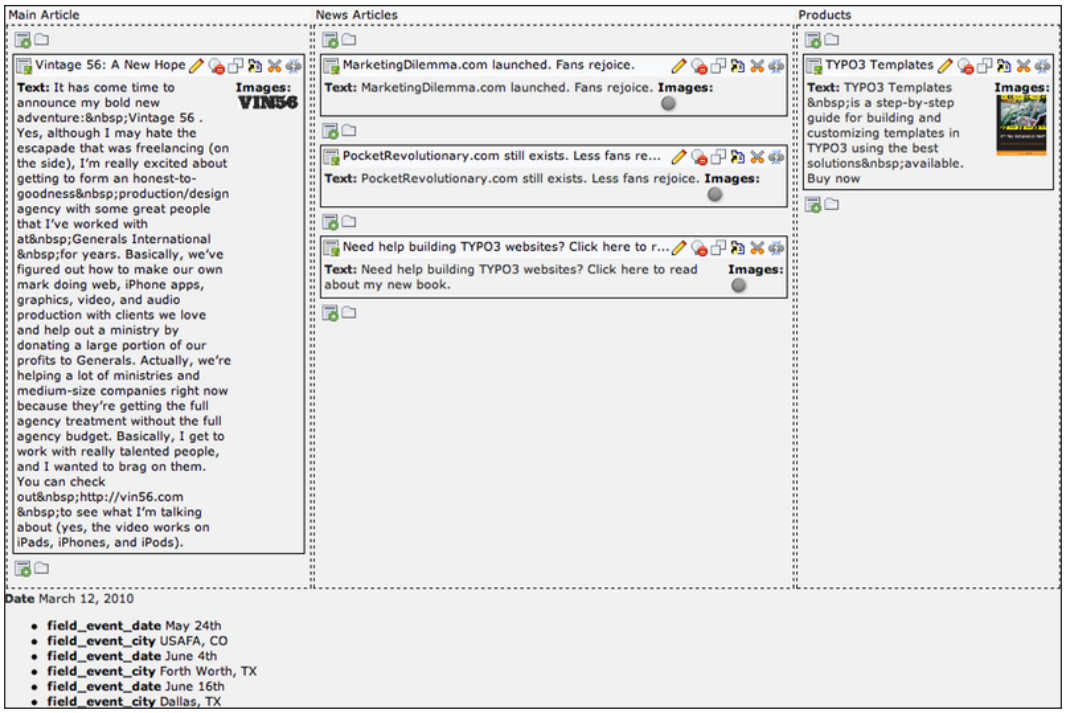
`allowTags`, `allowTagsOutside`	We can allow extra tags in the HTML editor using `allowTags` and `allowTagsOutside`. We don't want to break our current list of allowed tags, so we can use the `addToList()` function with the `:=` operator to add our tags to the current list without overwriting the preexisting values. If we wanted to add support for embedding YouTube or Vimeo videos, we can add `object`, `param`, and `embed` tags to the allowed list:
	`RTE.default.proc {`
	`  allowTags := addToList(object, param, embed)`
	`  allowTagsOutside := addToList(object, embed)`
	`  entryHTMLparser_db.allowTags < RTE.default.proc.allowTags`
	`}`
	
	Of course, that code will allow you to use the tags you need in the RTE, but you can test it now and you'll notice that it only works in the backend. In order to allow the new tags to be sent to the frontend of our website, we need to add the following code to our template's TypoScript setup:
	`lib.parseFunc_RTE.allowTags := addToList(object,param,embed)`

`allowedClasses`	Like allowTags, we can add custom classes to the allowed list for the HTML editor in the RTE. Example: `RTE.default.proc.allowedClasses := addToList(red, blue, specialClass)`

Customizing the Page module

Now that we are done customizing the RTE, we can work on the main module that editors will be looking at day after day, the Page module. The look of the Page module in TemplaVoila has improved, but it does not give our editors a good idea of how pages will look in the frontend. It's also based entirely on columns, so it can get too wide for easy editing. Like all things in TYPO3, we can adjust it and customize it to our needs. We can use an HTML file like we use for TemplaVoila template objects to make the backend look more like the frontend with headers, footers, and sidebars.

We're going to update the backend layout (commonly called BELayout in TYPO3) for the newsletter template by creating a new HTML file and assigning it to the template. As you can see, our default backend layout for the newsletter shows three columns, and there is no real indication of where things will appear in the frontend view:



Creating the HTML layout

Now we can create our own backend layout to replace the default for any pages with the newsletter template:

- We are going to make the backend layout look as much like the frontend template as possible, so we can start by copying the current newsletter HTML template, `template_newsletter.html`, to a new file, `belayout_newsletter.html`, inside the `fileadmin/templates/` directory.
- After you have created the new HTML file, open it up in a text editor so we can make some changes to adapt it to the backend better.
- Remove the style section in the head of the HTML document. We will add CSS later, but the CSS in the head section will not affect what we want right now.

- Let's change the main table tag to work better with the backend. We don't need to use tables in backend layouts, but we are using a table here to match what is seen on the frontend. We want our width to be fluid, so we'll remove the `width` property. We also don't need another border in addition to the one the Page module already provides, so we'll remove the border and replace it with a plain white background color:

```
<table align="center" style="background-color: #ffffff;">
```

- We want the banner to show up. As this is loading in the backend, TypoScript objects will not be loaded. Instead, we'll add the image as an HTML tag in the layout file:

```
<td id="banner_image" colspan="2" align="center"><img src="/
fileadmin/templates/newsletter_banner.png" alt=""></td>
```

- As we are not using pixels to constrain the width of our table, we need to change the main content column:

```
<td id="main_content" width="64%" valign="top">
```

- Inside our HTML, we can use markers to load content fields. The markers are simply the name of the field from the data structure (`field_main_article`, for example) surrounded by hash tags: `###field_main_article###`. TemplaVoila will replace this marker with an area for our different content elements such as the **Main Article** column in the default layout. So, we can use these markers in our HTML file to add the content fields to our main content column:

```
<table>
   <tr>
      <td id="main_article" style="padding-bottom:
15px;">###field_main_article###</td>
   </tr>
   <tr>
      <td id="news" style="padding-bottom: 15px;">
         <div class="news_item">###field_news###</div>
      </td>
   </tr>
</table>
```

- Next, we can update the width of the sidebar:

```
<td id="sidebar" width="36%" valign="top">
```

- TemplaVoila only lets us use markers for the fields of type "Content Elements", which are the fields that normally show up as columns in the Page module. Any of the other fields like TypoScript objects or fields that are set in the page properties cannot be part of the layout. The fields from our page properties, like our event fields in the newsletter, will still be automatically shown at the bottom of the page. So, we can't add the events fields to our layout, but we can add a placeholder section with a note for the editors so they still have a better preview of the frontend page:

```
<td style="padding-bottom: 15px;">
   <h3 id="event_list_title">Upcoming Dates</h3>
   <p>(Edit events in page properties)</p>
</td>
We can use markers for the products section, though:
<td id="product_list" style="padding-bottom: 15px;">
   <div class="product">###field_products###</div>
</td>
```

- We can use example data for the contact section. As editors will not be editing our TypoScript values, we don't need to tell them where to go to change the values. We can just show them some sample data based on the current TypoScript values so that they remember the contact information needs to fit into the column:

```
<td id="contact_info_section" style="padding-bottom: 15px;">
   <h3 id="contact_info_title">Contact Us</h3>
   <div id="contact_info">
      <p>Suite 850</p>
      <p>1214 Rebekah Ave.</p>
      <p>Dallas, TX 75154 USA</p>
      <p><a href="mailto:jeremy@example.com">jeremy@example.com</
a></p>
      <p><a href="http://www.example.com">www.example.com</a></p>
   </div>
</td>
```

- Finally, we will add some example data for the footer as well:

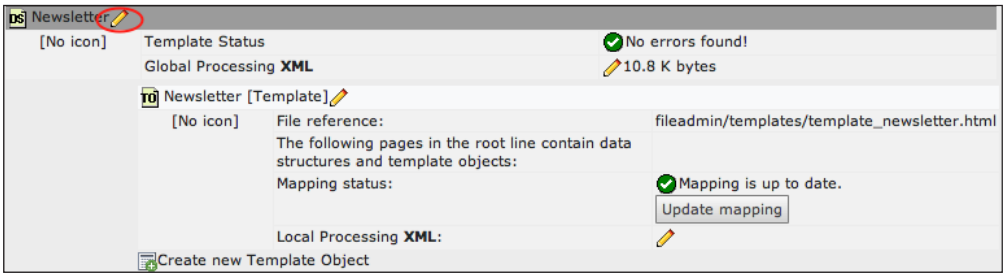
```
<div id="footer" align="center">Copyright Example.com 2010</div>
```

- After we have saved our changes to `belayout_newsletter.html`, we can open it in a browser to see that it structurally matches the newsletter template we are showing to visitors:

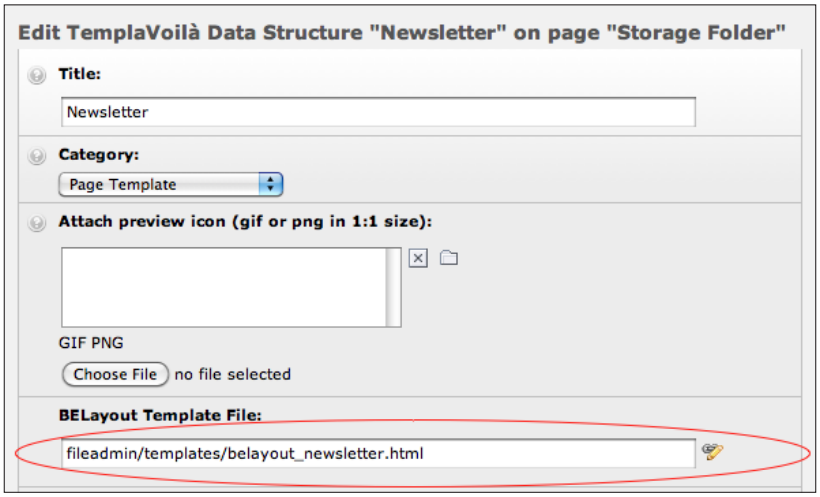
Assigning the backend layout

Now that we have an HTML file, we can assign it to the newsletter template data structure in the TemplaVoila module:

1. Go to the TemplaVoila module, and click on the pencil icon next to the **Newsletter** data structure:



2. Select our new HTML file, `fileadmin/templates/belayout_newsletter.html`, in the **BELayout Template File** field using the link icon on the right of the text field.



3. Refresh the cache, and go to our newsletter page in the backend Page module. You will see a page that looks much more like the frontend view, has a cleaner layout overall, and will be easier for our editors to use:

Adding some CSS styling

Our new backend layout is almost perfect, but it would be nice if the titles of each content section were a little clearer. We are allowed to use stylesheet links in our HTML head, so we'll take advantage of that right now. Let's add a link to our HTML file right now:

```
<head>
   <meta charset="utf-8" />
   <link rel="stylesheet" type="text/css" href="/fileadmin/templates/
css/belayout.css">

</head>
```

Now, we just need to create a minimal stylesheet for our newsletter template's Page module with some updated font sizes that might stand out a little better for our editors. Lets create the blank file `belayout.css` in our templates CSS directory (`/fileadmin/templates/css/`) right now. We'll update the size of the content element titles and previews using the built-in classes. TemplaVoila uses a class called `tpm-preview` around each content element in the backend, so we can update the font sizes to make it easier to read. The Page module also uses a class called `tpm-title-cell` for the title of each section like **Main Article** or **Products**, so we can use that class to make our titles clearer:

```
.tpm-title-cell {
   font-size: 14px;
   color: #555;
   font-weight: bold;
}
.tpm-preview {
  font-size: 12px;
  line-height: 16px;
}
```

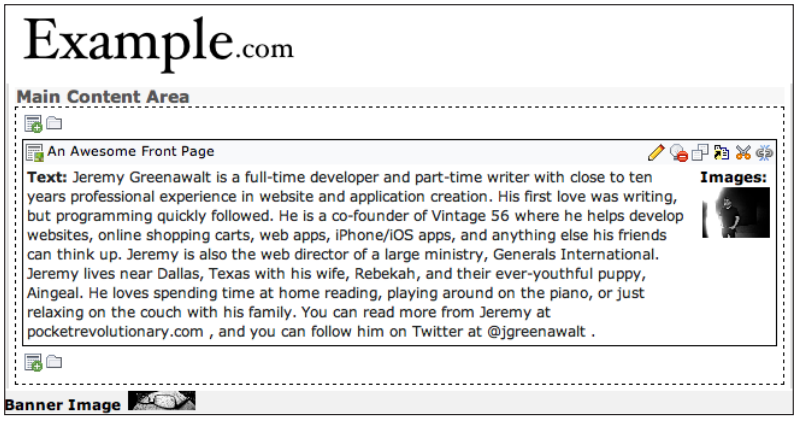
If you would like to know what classes are used for the rest of the elements in the Page module, the easiest way to find them is by viewing the source code when you are looking at the Page module in your browser. If you want an easy way to scan through the entire HTML output that TYPO3 is using, I recommend Firebug (`http://getfirebug.com/`) for the Firefox browser or the Web Inspector in Safari (`http://www.apple.com/safari/features.html#developer`).

Setting a backend layout for a data structure with multiple template objects

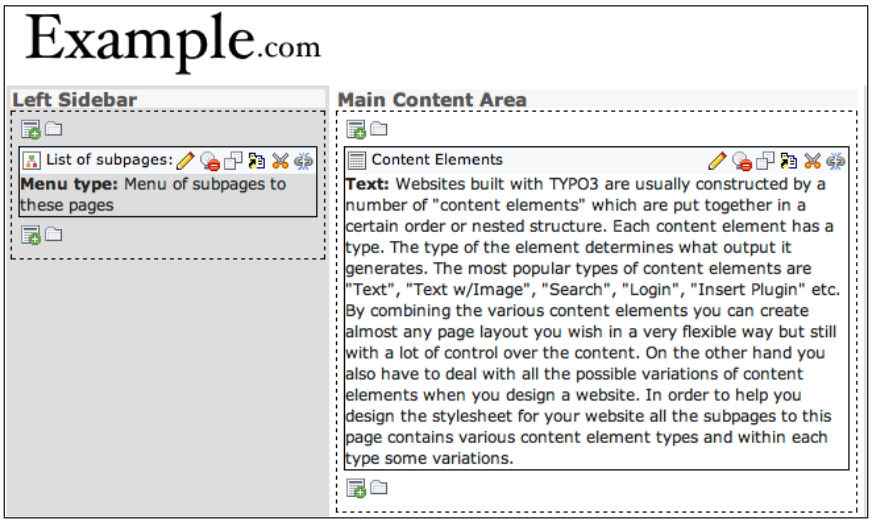
The new layout worked great for the newsletter, but we really need to update the main template. We could make a generic template, but how will it handle the multiple template objects? Luckily, the TemplaVoila Page view is already prepared to ignore any markers for content elements that are not active in the current template object. This means that we can create a generic HTML layout that includes the main content area and both the left and right sidebars without worrying about confusing our editors. Let's go ahead and create a new file named `belayout_main.html` in our templates directory (`/fileadmin/templates/`) right now with a universal layout:

```
<!DOCTYPE HTML>
<html>
   <head>
      <meta charset="utf-8" />
      <link rel="stylesheet" type="text/css" href="/fileadmin/
templates/css/belayout.css">
   </head>
   <body>
      <table style="background-color: #fff;">
         <tr>
            <td id="logo" colspan="3">
               <img src="/fileadmin/templates/logo.png">
            </td>
         </tr>
         <tr>
            <td id="left_sidebar" style="background-color: #ddd;">
               ###field_leftsidebar###
            </td>
            <td id="content">
               ###field_content###
            </td>
            <td id="right_sidebar" style="background-color: #ddd;">
               ###field_rightsidebar###
            </td>
         </tr>
      </table>
   </body>
</html>
```

Now, we can assign this as the backend layout file for the main template data structure in the TemplaVoila module just like we did for the newsletter data structure. If we load up a few pages in the backend, we can see that the inactive fields are being omitted without showing our markers at all. Without any content elements to be filled, the table cells are simply collapsing. As an example, this template does not use a left sidebar:



This is a template that uses a left sidebar:



Of course, if we had any other information in the cell besides the markers with hash tags, then they would show up. If we want to make it more specific we'll need to set a layout for the individual Template Objects.

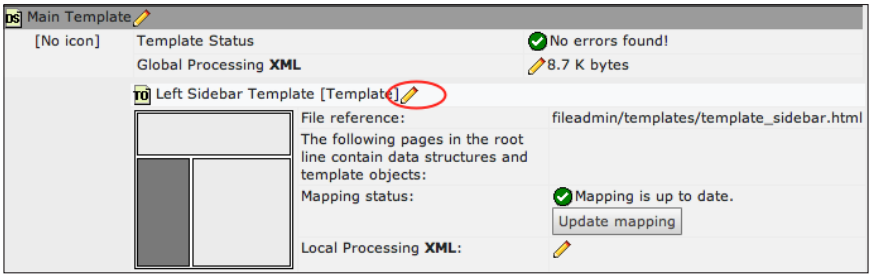
Using backend layout files for template objects

Assigning a new layout to a template's data structure is very useful, but we might not want all objects to have to same layout for some reason. Specifically, we might want to assign different layouts to the templates with sidebars without worrying about our modifications showing up for all templates under the **Main Template** data structure. The good news is that we're not limited to assigning external layouts to Data Structure objects, but we can also assign them to Template Object objects easily. If we want to create a specific layout only for the **Left Sidebar Template [Template]**, we can follow almost the same steps as we did for the Newsletter template:

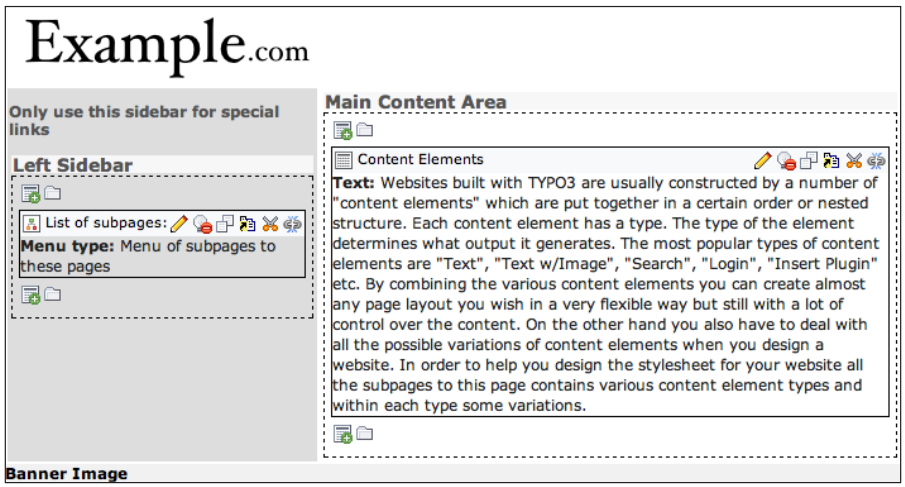
1. Create a new HTML file named `belayout_left_sidebar.html` in the templates directory (`/fileadmin/templates/`) with a basic HTML layout. We're going to add special instructions for our editors directly into the layout for this particular template. If we put instructions into the left sidebar area in `belayout_main.html`, they would show up on every page. This is a good reason to assign a special backend layout to just the **Left Sidebar Template [Template]** object:

```
<!DOCTYPE HTML>
<html>
   <head>
      <meta charset="utf-8" />
      <link rel="stylesheet" type="text/css" href="/fileadmin/
templates/css/belayout.css">
   </head>
   <body>
      <table style="background-color: #fff;">
         <tr>
            <td id="logo" colspan="3">
               <img src="/fileadmin/templates/logo.png">
            </td>
         </tr>
         <tr>
            <td id="left_sidebar" style="background-color: #ddd;">
               <h3>Only use this sidebar for special links</h3>
               ###field_leftsidebar###
            </td>
            <td id="content">
               ###field_content###
            </td>
         </tr>
      </table>
   </body>
</html>
```

2. Click the edit icon next to the **Left Sidebar Template** in the TemplaVoila module:



3. Select our new HTML file, `fileadmin/templates/belayout_left_sidebar.html`, in the **BELayout Template File** field using the link icon on the right of the text field.
4. Now we can save our changes at look at the new Page module in the backend:



Using static data structures in TemplaVoila 1.4.2

While we are talking about editing the data structures and backend layout options, we need to look at where TemplaVoila is going in the future. TemplaVoila 1.4.2 introduced static data structures as a new way of working the DS XML files, and the current workflow that we have been using for this chapter may be deprecated at some point. As of this writing, this feature is still experimental.

In fact, it is the experimental aspect of static data structures that made me almost not talk about it. It is well tested in several environments, though; it's just not going to be turned on by default for a few more versions of TemplaVoila.

For simple websites, you can skip this section. If you build enterprise websites or build a lot of websites with the same template elements, static data structures can be very helpful. I am including it in this chapter because it will change the way we build backend layout files, but it is not necessary for many simple websites.

What are static data structures

You may not realize this yet, but the current way to create data structures and backend layouts is unwieldy, prone to errors, and hard to backup. Traditionally, the XML of a DS object is stored in the database, and the only way to edit it was through the TemplaVoila module like we have been doing up to now. The major problems with this is that you had to edit only through the control center's text areas, and any custom XML could be wiped out by clicking **Modify DS / TO** and saving edits through the wizards. Many of us have gotten around this by creating XML documents in a text editor, saving the file as a backup, and copying and pasting the content into the **TemplaVoila Control Center** text areas. Static data structures save us extra steps and standardize the process.

The concept of static data structures is simple: each Data Structure object is an XML file organized by filename and directory structure. Directories in the `fileadmin` folder organize static data structures. The main data structure directory is called `ds`, and it includes a subdirectory for Flexible-Content Elements (`fce`), which we will talk about in the next chapter, and Pages (`page`). Inside the directory, each data structure can have an XML data structure file and an optional back end layout HTML file and preview icon using the DS object name to tie them together and extension to assign their purpose:

- **Data Structure**: `Template Name (page).xml`
- **Backend layout**: `Template Name (page).html`
- **Preview icon**: `Template Name (page).gif`

An example data structure for our current website would look like this:

```
ds/
   page/
      Main Template (page).xml
      Main Template (page).html
      Newsletter (page).xml
      Newsletter (page).html
   fce/
```

You'll notice that the scope of the template (`page`) must also be included in the filename. This may change in future versions of TemplaVoila, but it is a requirement right now. This is one disadvantage to static data structures right now for those who don't want spaces or parentheses in filenames on their server, but the new system has a lot of advantages for future template building, such as:

- Static files are easier to backup and restore on the server. You need to have consistent backups of your database, but actually restoring a broken data structure inside the database can be a monumental task. You can't rollback an entire database on a live site with users, and the only other option is to restore one single field from a large backup file. Static data structures can be backed up with the rest of your site's template files and restored using your default backup software.
- Similar to backing up, static data structures can be stored in a version control system such as Subversion or Git. This gives us the power to test, deploy, or rollback data structures without performing major operations.
- You can deploy new templates easier with static data structures. Have a new website that needs a template you built somewhere else? We can copy the data structure files with the Template Object HTML file into the new website, and the only operation left to do in the backend is map the two together.
- We can work on the XML in the editor of our choice. Many editors such as TextMate can already support TemplaVoila's Data Source XML syntax, and working on a large data structure inside a text area has never been a good option. Just our current main template is almost two hundred lines at this point, so just the ability to have powerful find and replace can save lots of time. If you use snippets like the TextMate bundle, you can save even more time and prevent the frustration of tracking down a simple typo.

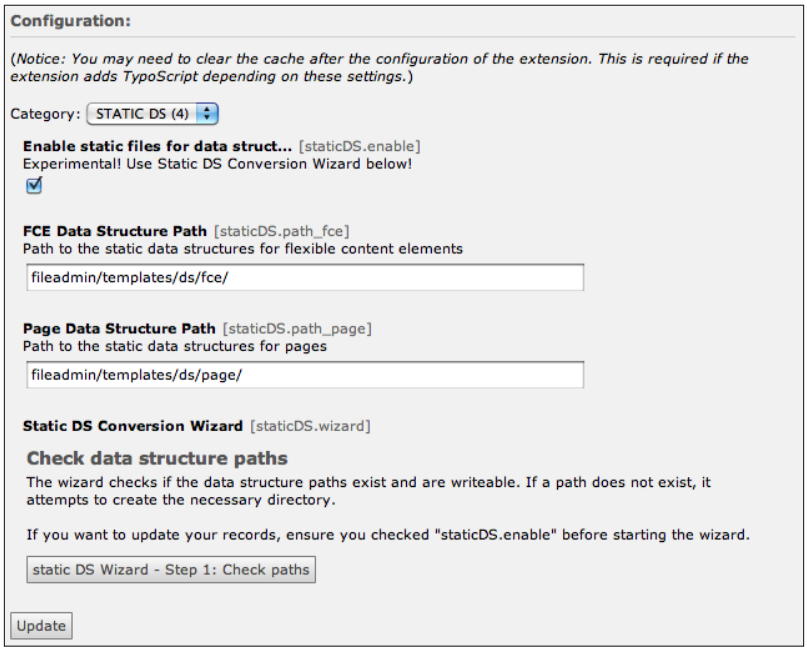
Setting up static data structures

To help you understand the static data structures, we will go through an example in our test environment. The TemplaVoila developers are still working on the implementation, so some of these steps may have changed or become more automated through wizards. Remember to check the online documentation (`http://typo3.org/documentation/document-library/extension-manuals/templavoila/current/`) if you need more updated references. Once we have enabled static data structures, TYPO3 will no longer check the database for our XML, so we need to make sure that we follow of the steps for conversion carefully.

1. First things first, you need to back up your current database. We are about to change the fundamental way that templates work, so you need to the ability to restore the current data structures in the database. The more current your backup, the easier it is to restore.

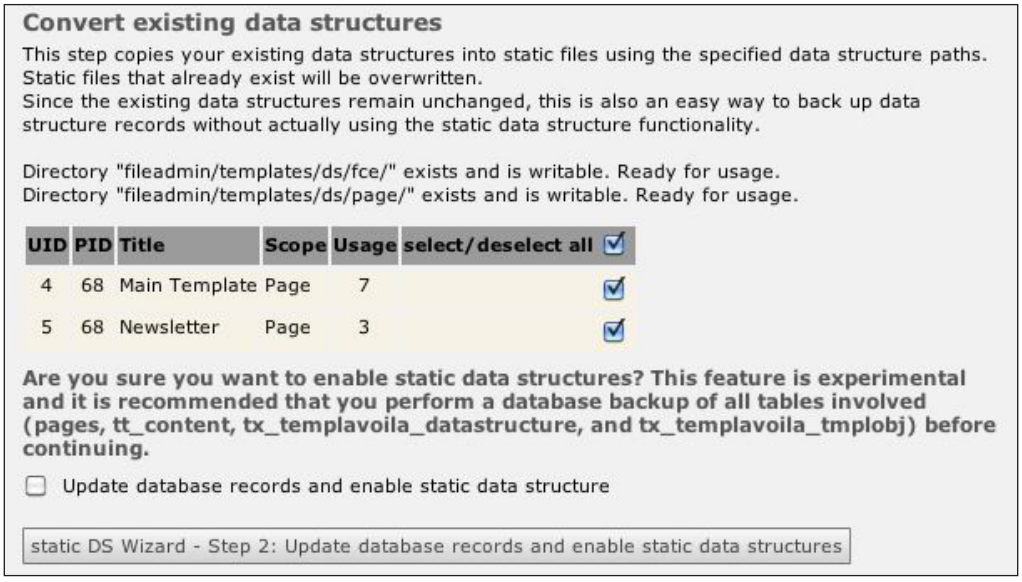
If you are not sure how to backup your database, MySQL has instructions on their website (`http://dev.mysql.com/doc/refman/5.1/en/backup-and-recovery.html`).

2. Now we can create new directory structure for our static templates. The TemplaVoila extension wizard should be able to do this, but some versions do not currently set the permissions correctly so it's easier to create them by hand. Create a directory named `ds` inside the `fileadmin` directory. Inside the `ds` directory, create two subfolders: `fce` and `page`. Again, make sure that they are readable and writable by the web server.
3. In the backend of TYPO3, go to the **Extension Manager** (labeled **Ext Manager**) under the **Admin Tools** section of the left frame. Make sure that **Loaded extensions** is selected in the main drop-down.
4. Choose the **TemplaVoila!** extension in the **Miscellaneous** folder to edit the extension installation and enable static data structures.
5. In the configuration section, check the checkbox labeled **Enable static files for data struct...** to enable static data structures in this installation and click on the **Update** button to save our changes.



6. Next, we can start the DS Wizard by clicking on the button labeled **static DS Wizard - Step 1: Check paths**. This will allow the wizard to check that our new directories are created and have the correct permissions for the static data structure files.

7. After the wizard checks our directories and permissions, it will try to convert our existing data structures by creating XML files in our `ds/page/` directory. If we had not enabled static data structures, we could still use this part of the wizard to save static copies of our XML as static data structure files for backup. We have enabled static data structures, so we need to make sure that we convert all of our data structures to prevent data loss. Once again, TYPO3 will no longer check the database for our XML when static data structures are enabled. With that in mind, we need to go ahead and select all of our data structures for the conversion step.



8. We need to confirm that we are enabling static data structures and allow the wizard to update our database by checking the box labeled **Update database records and enable static data structure**. Now is a great chance to double-check that we have a current backup of the database including the `pages`, `tt_content`, `tx_templavoila_datastructure`, and `tx_templavoila_tmplobj` tables. Once we have verified our backup, we can click on the large button labeled **static DS Wizard - Step 2: Update database records and enable static data structures**.

9. After the wizard is done processing, we can see two new files in the `/fileadmin/ds/page/` directory named `Main Template (page).xml` and `Newsletter (page).xml`. In order to move over the backend layout files, we just need to copy them from the `fileadmin` directory and rename them to match our new data structures. Copy `/fileadmin/belayout_main.html` to `/fileadmin/ds/page/Main Template (page).html` and `/fileadmin/belayout_newsletter.html` to `/fileadmin/ds/page/Newsletter (page).html`. To create backend layout files in the future, all we have to do is create an HTML file with a filename that matches the data structure we are modifying. TYPO3 will automatically detect the new layout.
10. Next, we could add a preview icon to either data structure by adding a GIF file with the name of the DS object. For example, a preview icon for the main template just has to have the filename `Main Template (page).gif` in the `/fileadmin/ds/page` folder. Currently, it detects the preview icon by name and extension, so only GIF formatted images are allowed. You may have to convert from PNG if that is what you were using before.
11. Clear the cache for TYPO3. We have made a lot of changes, so the cache will be inconsistent until we clear it. If any Template Object records lost their association with the data structures during the update, we may need to update them in the **TemplaVoila Control Center** by selecting them in the **Lost TOs** tab and updating their **Data Structure** value.
12. Finally, we can re-enable our backend layouts by copying the HTML files into the `ds/page/` directory. For example, we can use our backend layout file for the newsletter template by copying `fileadmin/templates/belayout_newsletter.html` to `fileadmin/templates/ds/page/Newsletter (page).html`. Go ahead and copy the `belayout_main.html` to `ds/page/Main Template (page).html` as well. The backend layout that we created for the Left Sidebar Template need not to be changed because the changes only affected data structures. From now on, we can create new backend layouts without ever touching the TemplaVoila module in the TYPO3 backend; we just need to make HTML files inside the `ds/page/` directory.

Modifying static data structures

After the cache is cleared and any lost templates are updated, we'll be using the new static data structure. Aside from the obvious changes we just made, a lot of our workflow will still be the same. We can still edit the data structures through the **Modify DS / TO** wizards in the control center, but now we can make a copy of our files before we make any large changes. Of course, we can also now edit the XML directly in an editor if we want.

If we want to create a new data structure, we just need to either make a blank XML file in the `/fileadmin/ds/page` directory with the name we want and the `(page)` scope or copy and duplicate one of our existing static data structure files.

Summary

Okay, now that we've wrapped up our little experiment with static data structures, we can look back at everything that we've done in this short chapter. At the beginning of this chapter, the entire editing experience for our editors was exactly like any other out-of-the-box TYPO3 installation. That would be okay, but we've seen that the developers of TYPO3 and TemplaVoila gave us the ability to tailor our backend experience to our editors' needs and the branding and layout that we have built for the frontend.

By configuring the text editor, we were able to show the editors what paragraphs and headings would really look like. In addition, we could add some classes to their toolbox we wanted to use on the frontend and remove some unwanted classes and toolbar buttons that we didn't want them to touch. We even added access to the HTML tags that they need to include video from YouTube or Vimeo directly in a text content element.

After we got the text editor looking better, we went after the Page view layout. We also saw a couple ways of including backend layouts through inline tags in the XML and external HTML files. We looked at ways to build better looking backend layouts that could apply to a whole data structure or just individual templates.

Finally, we looked at where TemplaVoila templates are going with static data structures. We looked at some of the advantages and even tried it out on our own example site. There may be some changes in implementation over time, but we'll already be a step ahead of the crowd now that we've started using the next generation of data structure workflows.

Looking at the new editing experience in the backend, we've made a lot of improvements. We also know our way around the TemplaVoila Data Structure system like experts now, so we're going to start looking at the other side of TemplaVoila in the next chapter: Flexible Content Elements. Flexible Content Elements are kind of like mini-templates for TYPO3, and we're going to use them to create ads, multi-column layouts, and a few more helpful objects. I know it seems like we've exhausted every layout trick available, but FCEs are the other half of TemplaVoila in many ways. Many developers have switched to TYPO3 just for the FCEs.

Okay, grab some more coffee, show your favorite website editor how nice the backend looks, and come back here ready to play with some Flexible Content Elements in the next chapter.

8
Working with Flexible Content Elements

Now that we know our way around TemplaVoila data structures and template objects, we're ready to introduce **flexible content elements** (or **FCEs**). As a core feature of the TemplaVoila extension, FCEs can almost be thought of as mini-templates for groups or chunks of content. Site developers use them in TYPO3 installations worldwide to create specialized content layouts for small groups of content like an advertisement where you might want a title, description, and image grouped together. There are actually so many places that you can use them that flexible content elements are used in many situations instead of custom extension development. By the end of this chapter, we'll all be creating custom FCEs to fill our special requirements.

In this chapter, you will:

- Learn about flexible content elements
- Create your first flexible content element to display contact information
- Create an FCE to wrap a group of content elements in a `div` tag for custom styling
- Use an FCE to create multi-column layouts inside any TemplaVoila content area
- Create a flexible content element for product displays

Introducing flexible content elements

I just said flexible content elements are like mini-templates, but they are actually a little more sophisticated than that. TYPO3 templates, traditionally, are just used to design the pages of your website. Well, flexible content elements are there as a way to create our own specialized types of content elements. Flexible content elements give us most of the power that we've had in the main TemplaVoila templates; the workflow and structure is almost exactly the same. We still have a data structure and a mapped object for each template, and we can still create backend layouts and preview images to help our editors. So, we already have all the raw skills we need to start creating them, but we just need a few examples of where we would want them.

Creating a flexible content element really is just like creating a new content type for our sites utilizing the power of TemplaVoila. Once created, they can be embedded into any templates or even other FCEs. We can use references to link them across separate pages. We can copy them around between TYPO3 installations easily. We can even update the main template object or data structure of an FCE and watch our changes reach to every instance of that content element in our page tree. The best examples of this are going to be some of the ones we're about to build: a contact section, a `div` to wrap around other content elements, a multi-column element, and a custom product layout.

Creating our first flexible content element

The first FCE we're going to create is going to be simple enough to show us the basic workflow, but it's also a pretty handy trick so that we can add our contact information onto multiple pages using consistent formatting across our site. We're going to create a new flexible content element for our contact information using microformats (`http://microformats.org/`) so that Google and some browsers can read our address, phone number, and email address easier. Normally, this would require a lot of extra work to place it on multiple pages, but we can create an FCE that our editors can add to any page just like a normal content element.

Building the content element

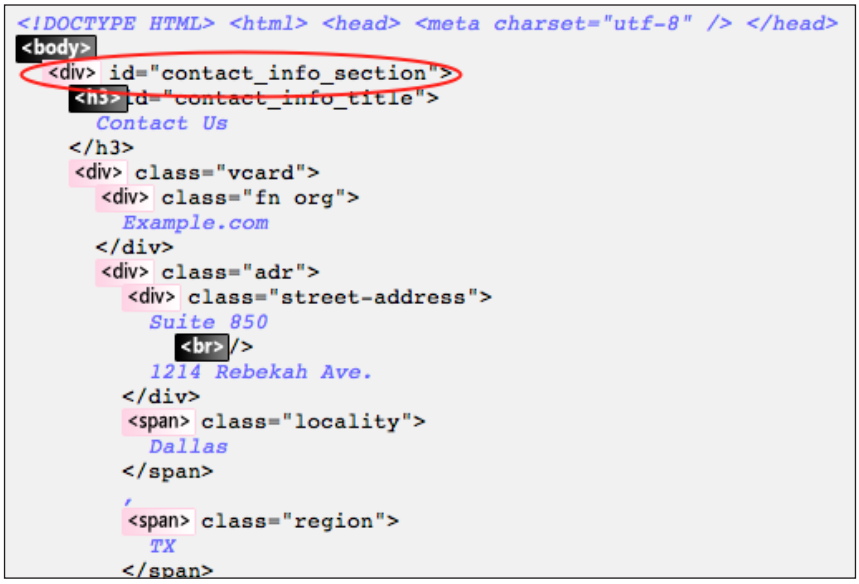
1. Of course, the first thing that we should do is create an HTML file that we can use for our mapping. FCEs are normally mapped to a part of a complete HTML template, while page templates are mapped to the whole file. For our example, we will create one universal HTML file to hold all of our flexible content element HTML snippets called `template_fce_snippets.html` in the `fileadmin/templates/` directory.

2. At this point, we can create our HTML that the FCE will be based on. Like we said before, we are going to use microformats so that our contact information can be read by software more easily. To use microformats, we are just going to add some specific class names to the `span` and `div` tags around our information. If you would like to see more information about microformats, I recommend going to `http://microformats.org/`. For now, we can add this code into our `template_fce_snippets.html` file:

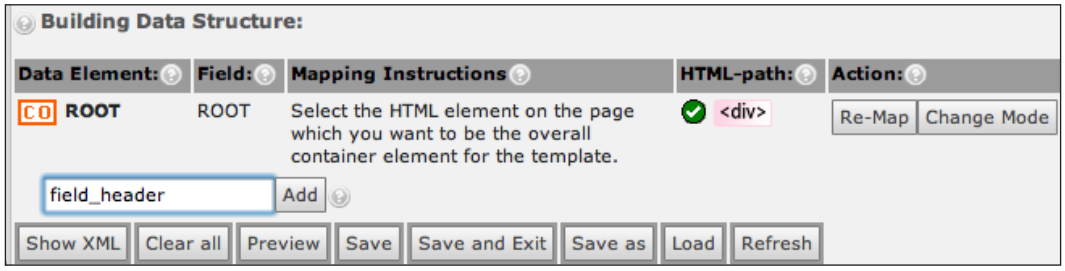
```
<!DOCTYPE HTML>
<html>
<head>
   <meta charset="utf-8" />
</head>
<body>
<div id="contact_info_section">
   <h3 id="contact_info_title">Contact Us</h3>
   <div class="vcard">
      <div class="fn org">Example.com</div>
      <div class="adr">
         <div class="street-address">1214 Rebekah Ave.
                </div>
         <span class="locality">Dallas</span>,
         <span class="region">TX</span>
         <span class="postal-code">75154</span>
         <span class="country-name">USA</span>
       </div>
      <div class="tel">(212) 555-1212</div>
      <a class="email"
           href="mailto:jeremy@example.com">jeremy@example.com</a>
   </div>
</div>
</body>
</html>
```

3. We are ready to start creating the TemplaVoila structures now, so we can go to the TemplaVoila Control Center in the backend of our installation now. We can create a new data structure by going to the **Template Files** tab in the control center. To create a new template object and data structure simultaneously based on our new HTML file, we need to click on the **Create...** link for our template file, `fileadmin/templates/template_fce_snippets.html`.

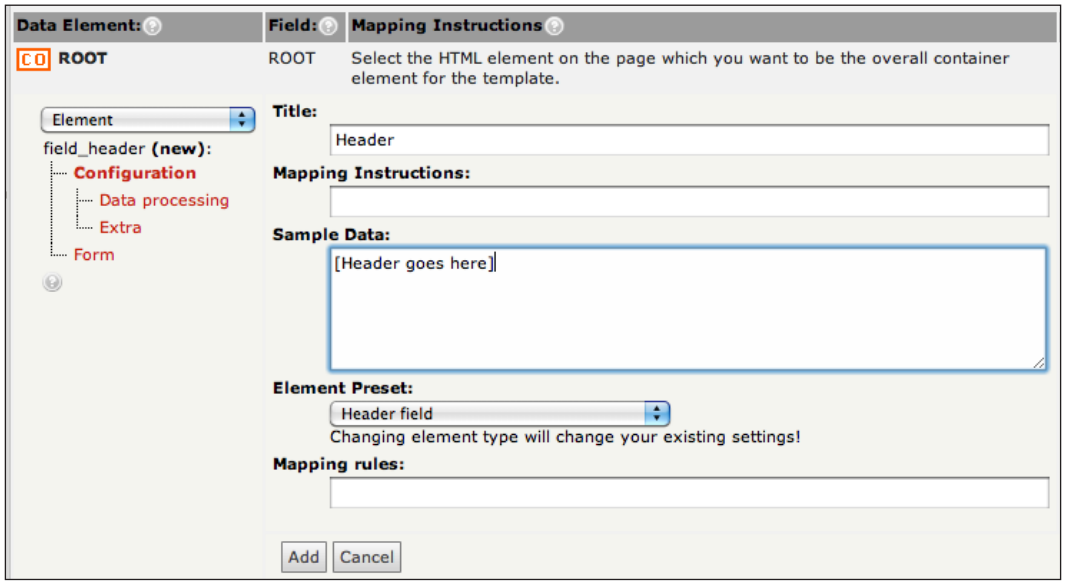
4. Now we need to choose the root data element. Go ahead and choose the main `div` tag (circled in the following screenshot). We need to make sure we set the mapping mode to **OUTER (Include tag)**. Due to an oddity in TemplaVoila, outer mapping is the only way to make sure that we actually keep the `contact_info_section` class after mapping. It may be counter-intuitive, but TemplaVoila treats root elements exactly the opposite of all other elements in its implementation of outer and inner mapping modes. Click on **Set** to start creating our data structure.



5. Now that we have set the root, we can add our own fields to the FCE data structure. All of our contact information can be static, so we will just create a field for the header. Like the page templates, we will create a new field by filling the in name, `field_header`, at the bottom of the page as shown in following screenshot, and click on **Add**.

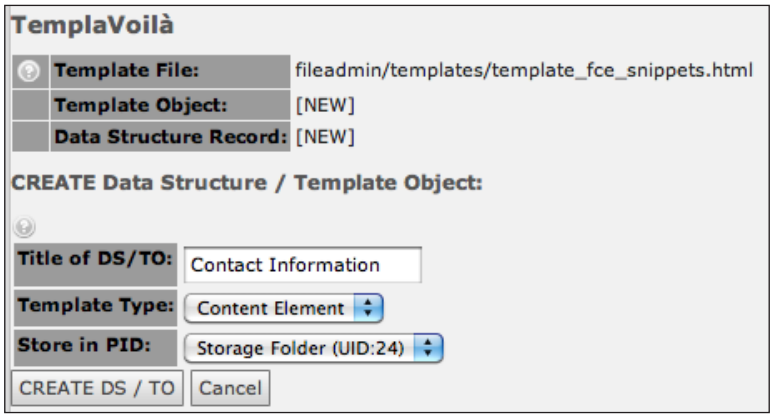


6. Now we can fill in the form for our new field. We will set the **Title** to `Header`, and we can set the **Sample Data** as `[Header goes here]`. As we are using this as a header, we can choose `Header field` as our **Element Preset**. After we have filled out the form as shown in the following screenshot, we can click on **Add** to save our settings.



7. Map the header field to the `h3` tag in our HTML template and click on **Set** to save the mapping.
8. We've finished creating our small content element, so we can click on the **Save as** button in our builder screen and save our progress. We are creating a new data structure, so we will need to fill out the **CREATE Data Structure / Template Object** portion of the TemplaVoila save screen. We will give our new element an easy title, `Contact Information`. We also need to make sure we choose **Content Element** from the **Template Type** drop down because we are creating an FCE instead of a page template this time.

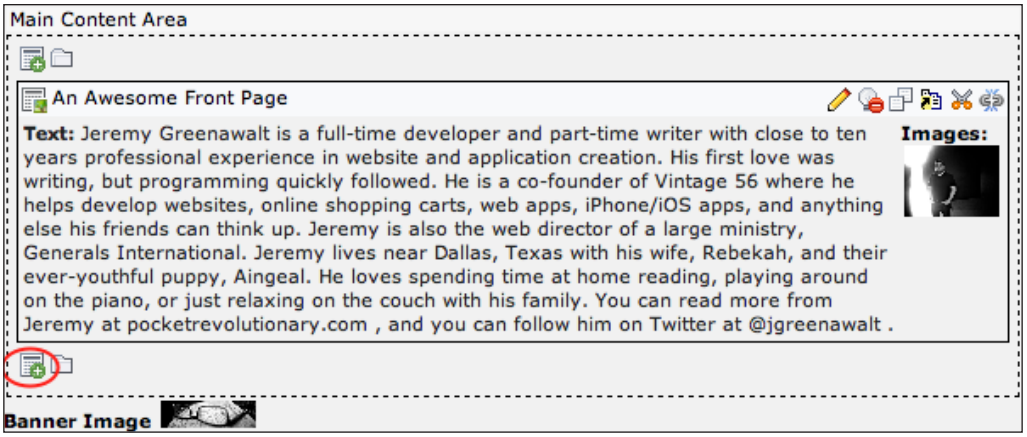
9. Our screen should look something like shown in the following screenshot before we click on the **CREATE DS / TO** button:



Testing our new content element

We can add our element to any page the same way we've been adding text or graphic elements in the past through the **Page** view.

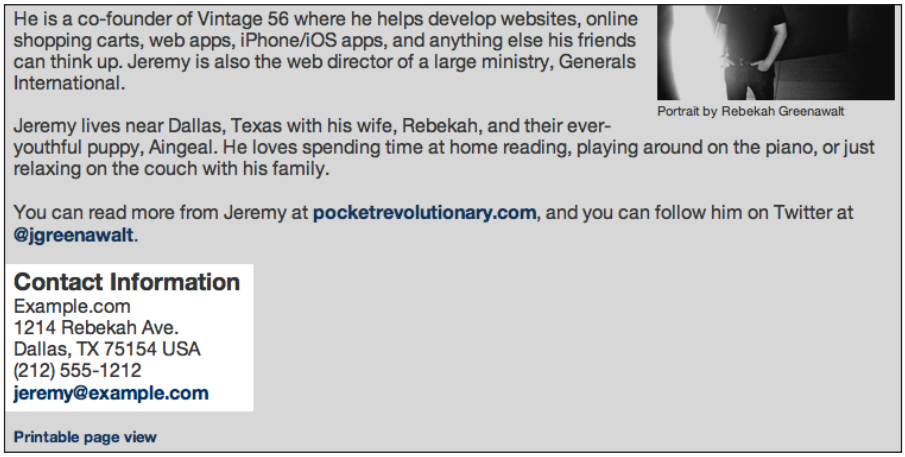
1. Go to the main page in the backend of TYPO3 and click on the new element button (circled in the following screenshot).



2. Added to the choices of standard or extension-based elements, we can see our own flexible content element, **Contact Information [Template]**, listed. Go ahead and choose it to add it to the page.

3. The next screen we see is the content editing screen where we can fill in the header for our new element:

4. Finally, we can save our new content element and see the output on the frontend (highlighted in the following screenshot):



Creating a flexible HTML wrapper

As a website grows, we sometimes run into times where we would like to assign a special style to a content element or group of content elements, but there is no easy way to do this in TYPO3 without creating a new page template. All we really want to do is wrap a `div` tag around the group of elements we are styling with a CSS class to give them any style we need from our own stylesheets. For example, we might want to highlight a group of content elements with a color background or border.

We will create a flexible content element to output a `div` tag with a blank `class` attribute that can contain normal page content elements. The FCE will have a field for the class, so our editors can fill in whatever class they need to use later.

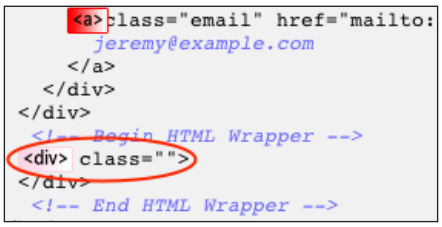
We're also keeping control over the options that editors have. They are still restricted to using CSS classes, as opposed to arbitrary style attributes, so we have not given them too much freedom.

Building the content element

1. First, we can create our HTML to which the FCE will be mapped. All we need is a `div` tag with a blank `class` attribute, so we can just add our snippet to the bottom of `/fileadmin/templates/template_fce_snippets.html`. We will also add some HTML comments around our new snippet so that we can always identify it in the HTML file:

```
<!-- BEGIN HTML Wrapper -->
<div class=""></div>
<!-- END HTML Wrapper -->
```

2. Now, we go back to the TemplaVoila module in the backend. From the **Template Files** tab in the **TemplaVoila Control Center**, click on **Create...** next to the file `fileadmin/templates/template_fce_snippets.html` label.
3. Go ahead and choose our new `div` tag between the HTML comments (circled in the following screenshot) and click on **Set** to start creating our data structure.



4. Again, choose **OUTER (Include tag)** as our mapping mode.
5. The first field we need to create is the wrapper field for the content element. We have already set the `div` tag as the root element, but we still need to create a separate field to handle content elements or we won't be able to add content into our new FCE in the **Page** view. Like before, we can create a field by filling in the new field text area with a new name, `field_wrapper`, and clicking on the **Add** button. Now we can create the field with the following values just like we did when we added fields to our main templates. Like our page templates, we are going to use the `Page-Content Elements` preset because it allows us to place other content elements inside our new field:
    - **Field:** `field_wrapper`
    - **Element**

- **Title:** `Wrapper`
- **Sample Data:** `[Content goes here]`
- **Element Preset**: **Page-Content Elements**

6. Once we have created and saved our new field, we can map it to the `div` tag by clicking on the **Map** button. We can use inner mapping this time because we want to keep the tag and this is not the **ROOT** field.
7. The next field we need to create is the class field so that we can edit the class from the page module. Instead of an element, we are creating an attribute. To create the new field, fill in the name, `field_class`, at the bottom of our page and click on **Add**. Choose **Attribute** from the drop down on the left side and fill out the field values:
   - **Title:** `Class`
   - **Sample Data:** `[Class field]`
   - **Element Preset**: **Plain input field**
8. After we have created the new class attribute and saved it, we can map it to the `class` attribute in our `div` tag. If we click on the **Map** button for the class field, we see that we can only choose the `div` tag to map to; this is okay. If the `div` tag is grayed out or disabled, we probably need to check that the root element was set with **OUTER** mapping. After we click on the `div` tag, we are presented with a slightly different mapping screen than we have seen before. Up until now, we have been mapping tags instead of attributes, so our choice has been **INNER** or **OUTER** mode. When mapping attributes, this drop down will show any blank attributes that exist within the HTML template for that tag. If we wanted to set a relation attribute, for example, the HTML just needs to have `rel=""` present in the tag with or without a value. For now, we can choose **ATTRIBUTE "class" (=)** from the drop down and click on the **Set** button to continue.

9. We've created all of the fields we need for this small content element, so we can click on the **Save as** button to save our progress. We will give our new element an easy title, `HTML Wrapper`. We also need to make sure we choose **Content Element** from the **Template Type** drop down again.

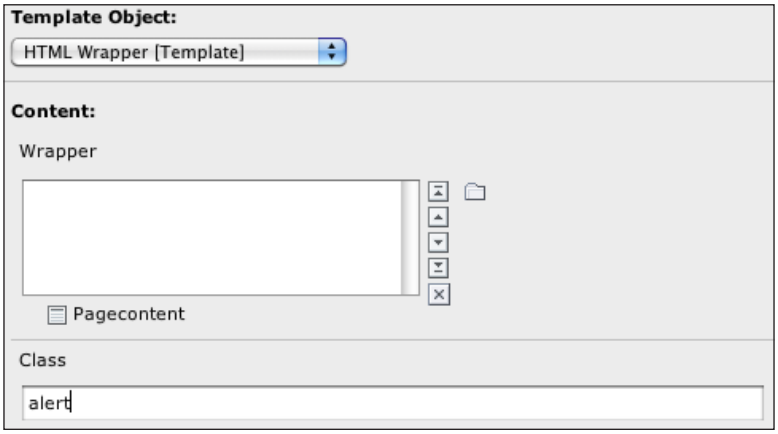
Testing our new content element

We now have a data structure and template object created as a flexible content element and mapped, so we are ready to test. We can test with almost any class from our stylesheet, but we'll make it easy by adding a new class style to the bottom of our `style.css` file with a color background, rounded corners, and a slight shadow to highlight content:

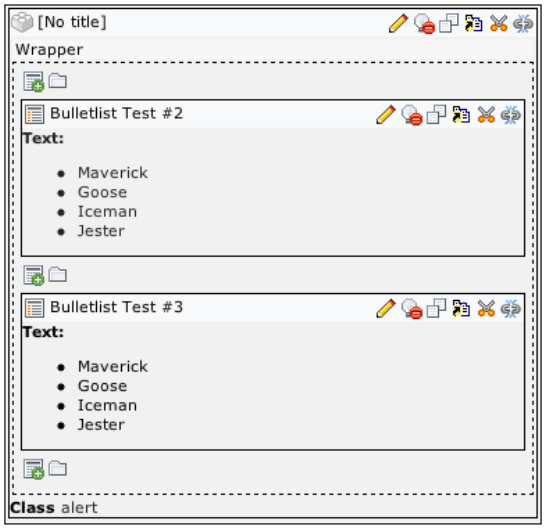
```
.alert {
  background-color: #BBCCDD;
  padding: 10px;
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
  border-radius: 10px;
  box-shadow: 2px 2px 5px #888;
  -webkit-box-shadow: 2px 2px 5px #888;
  -moz-box-shadow: 2px 2px 5px #888;
}
```

As an example, we can highlight a couple of bullet lists on the **Bulletlist** page that the TemplaVoila wizard created.

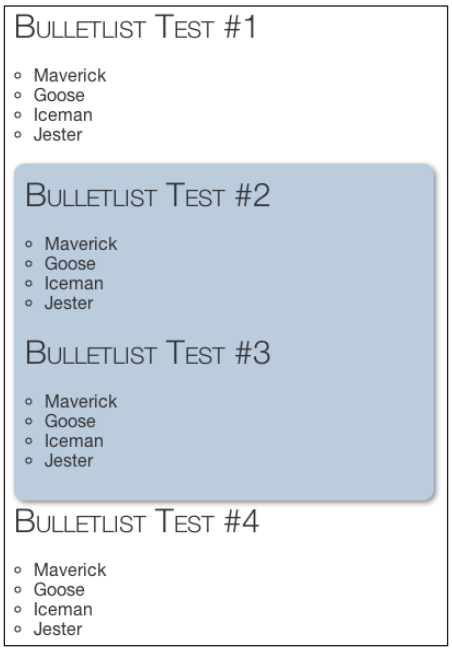
1. Go to the **Bulletlist** page in the backend of TYPO3 and choose to add a new element like we did for the **Contact Information** FCE.
2. This time, choose **HTML Wrapper [Template]** for our new element.
3. The next screen we see is the editing screen, and we can see that the titles we gave our data structure fields are showing up along with the different form elements we just declared. We can add elements to the wrapper here, but it's easier in the page module. Instead, we'll just set the **Class** field to `alert` to match our stylesheet, and save our new element.



4. Finally, in the page module, we can drag elements into our new content element, and our new `div` tag with a class we have set in the settings will wrap around them. We can drag two of our bullet lists into the FCE:



If we look at our edited page on the frontend, we can see the new CSS styling applied immediately:



Creating a multi-column layout element

As helpful as a wrapping `div` tag can be, we should start getting a little bigger with our goals. One thing that we run into all the time in the real world of site development is the need to have multi-column elements. With the rise of grid-based design and content-heavy sites, it's getting more popular to start adding two columns into the main content area under a single column article or something similar. Unfortunately, there are a lot of variations on this idea of mixing and nesting multi-column layouts, and it's not always possible or smart to create a template for every possible variation in a limited amount of time. You can easily waste all your time creating templates with a two-column element over a three-column element over a two-column element over a single-column element. I know that sounded confusing, and that's the problem. Instead, we can create a handful of useful multi-column flexible content elements that our editors can use anywhere they need to and in any order they need to. They can even nest them inside of each other if we do this right.

Right now, we're going to make a quick FCE with two columns that take up roughly half of the current content area. We're just going to start by adding some basic styling to our main stylesheet, `fileadmin/templates/style.css`:

```
.multi_column_element {
   display: inline-block;
   width: 100%;
}
#nested_column_1 {
   float: left;
   clear: left;
}
#nested_column_2 {
   float: right;
   clear: right;
}
.half {
   width: 49%;
}
```

As you can see above, we are using `inline-block` as the display setting for the entire element. If we don't set that, then the elements below it can creep up when we start using floats. For more information on CSS values like `inline-block`, I recommend the tutorials from w3schools.com (`http://www.w3schools.com/css/`).

In addition, our style floats the first column, `nested_column_1`, to the left and clears anything to its left. The second column, `nested_column_2`, floats to the right and clears anything to the right of it. If we assign the class half to both columns, then they will both take up a little under 50% of the total width with a little whitespace in the middle.

After we've modified the CSS, we need to update our HTML file. Once again, we'll add our new HTML code with identifying comments into our HTML template, `/fileadmin/templates/template_fce_snippets.html`. Go ahead and add some basic code to the main FCE HTML file to create two `divs` for columns:

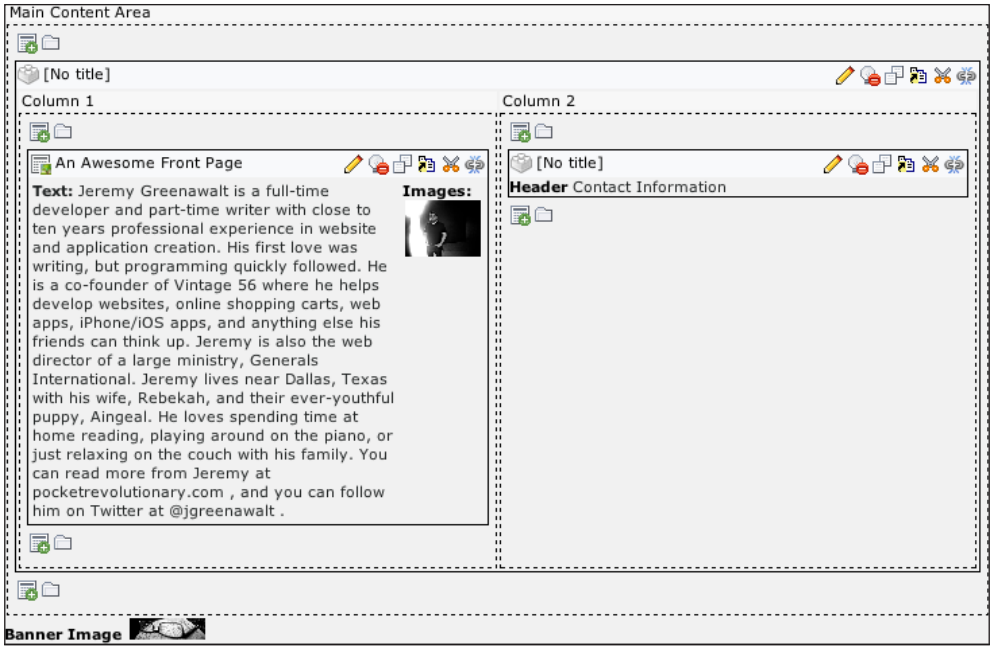
```
<!-- BEGIN 1/2 + 1/2 Element --><div class="multi_column_element">
   <div class="nested_column half" id="nested_column_1">Column 1</div>
   <div class="nested_column half" id="nested_column_2">Column 2</div>
</div>
<!-- END 1/2 + 1/2 Element -->
```

Now we're going to follow most of the same steps from the previous examples starting with the creation of a new data structure:

1. From the **Template Files** tab in the **TemplaVoila Control Center**, click on **Create...** next to the file `fileadmin/templates/template_fce_snippets.html` label.
2. Choose the main `div` tag that wraps around the entire HTML template as the root field. Again, we need to make sure we set the mapping mode to **OUTER (Include tag)**.
3. Create a new field for the first column named `field_column_1`. As the float is set completely in CSS, we will not refer to the columns as left or right columns here. This means we could swap the columns in CSS or assign different identifiers in the HTML without breaking our data structure. Go ahead and create our new field with these values:
    - **Field**: `field_column_1`
    - **Element**
    - **Title**: `Column 1`
    - **Sample Data**: `[Column #1 goes here]`
    - **Element Preset**: **Page-Content Elements**
4. Save the first field and map `field_column_1` to the `div` tag with the ID `nested_column_1`. Make sure that you select inner mapping so that the classes and identifiers are left in the `div` tag.
5. Create a new field for the second column with almost the same values as the first column:
    - **Field**: `field_column_2`
    - **Element**
    - **Title**: `Column 2`
    - **Sample Data**: `[Column #2 goes here]`
    - **Element Preset**: **Page-Content Elements**

6. Save the second column field and map it to the `div` tag with the ID `nested_column_2` in the HTML.
7. Click on the **Save as** button to save our new data structure and template object. Set the title as something memorable, `Two-Column Element`, before choosing **Content Element** as the **Template Type** and clicking **CREATE DS / TO**.

As easy as that, we've just created another FCE. We can test this one on the main page of our test site by creating a new content element with our new FCE, **Two-Column Element**, and dragging our current blocks into either side:



With two even columns, our front page should look something like this:

Extending the multi-column layout element

As you can see from our example above, two even columns are not always the best answer for a good layout. Thankfully, we can create multiple flexible content elements using the same data structure. In fact, the process is, like all of the FCE processes, almost exactly like the page template process. Just like when we created multiple template objects from the main data structure, we can create multiple FCE template objects from the multi-column data structure.

We're going to add two new template objects. Each will have a column using 33% of the total width and a column using 65% of the total width. Like the even columns, we're just using percentages as a quick way of getting up and running, and we're allowing a little less than 100% width so there will be some space between the columns. One template will have the large column on the right, and the other one will have the large column on the left. We'll just be building on the data structure that we've already built, so we can go through the process without too much explanation on each step.

1. We need to add classes for the two new possible column widths to the stylesheet. Let's go ahead and add these to the bottom of our `style.css` file:

```
.one_third {
   width: 33%;
}
.two_thirds {
   width: 65%;
}
```

2. We are going to create two new HTML snippets almost exactly like the previous one. The only difference will be in the classes of the columns. Before we set them both to half, now we will be using `one_third` or `two_thirds`. First, we'll add the HTML code with 33% width on the left with HTML comments that identify it as having 1/3 on the left and 2/3 on the right to `template_fce_snippets.html`:

```
<!-- BEGIN 1/3 + 2/3 Element -->
<div class="multi_column_element">
   <div class="nested_column one_third" id="nested_
column_1">Column 1</div>
   <div class="nested_column two_thirds" id="nested_
column_2">Column 2</div>
</div>
<!-- END 1/3 + 2/3 Element -->
```

3. Next, we'll create the HTML with the smaller column on the right. The only difference in the HTML is that we have switched the `one_third` and `two_thirds` classes between the columns. We could make this editable like the HTML wrapper, but that would just add work for the editors. With this code, there's no possibility of putting two `two_thirds` columns next to each other:

```
<!-- BEGIN 2/3 + 1/3 Element -->
<div class="multi_column_element">
   <div class="nested_column two_thirds" id="nested_
column_1">Column 1</div>
   <div class="nested_column one_third" id="nested_
column_2">Column 2</div>
</div>
<!-- END 2/3 + 1/3 Element -->
```

4. In the backend TemplaVoila module, we can go to the **Flexible CE** tab to see our current flexible content elements. To avoid confusion, we can rename the **Two-Column Element [Template]** template object. Go ahead and click on the pencil to edit it, and change the title to something more unique like `1/2 + 1/2 Element [Template]`. Save the changes.
5. Back in the TemplaVoila module, we will click on the **Create new Template Object** link under our template object labeled **1/2 + 1/2 Element [Template]**.
6. When the editing screen comes up, we can fill out the fields for our new template:
    - **Title**: `1/3 +2/3 Element [Template]`
    - **File reference**: `fileadmin/templates/template_fce_snippets.html`
    - **Data Structure**: **Two-Column Element** (or **Static**: **Two-Column Element (fce)** if you are using static data structures)
7. After you have saved your changes, repeat the process to create another template object with similar settings:
    - **Title**: `2/3 + 1/3 Element [Template]`
    - **File reference**: `fileadmin/templates/template_fce_snippets.html`
    - **Data Structure**: **Two-Column Element** (or **Static**: **Two-Column Element (fce)** if you are using static data structures)

8. Next, we can click on the **Remap** button for the **1/3 + 2/3 Element [Template]** to start mapping it. Map this new template object exactly like the template object with two even columns before.
    - Map **ROOT** to the main div with the class `multi_column_element`. Select **OUTER** for the mode.
    - Map **Column 1** to the div with the id `nested_column_1`.
    - Map **Column 2** to the div with the id `nested_column_2`.
9. Click the **Save and Return** button.
10. Repeat the mapping and saving process for the **2/3 + 1/3 Element [Template]** template object.

Now we can return to the front page and update our layout. Just click on the pencil icon for our two-column element to edit it and change the **Template Object** drop-down to **2/3 + 1/3 Element [Template]**. If we refresh our page in the frontend, we should see something that works at least a little better:

Creating a product display element

We've seen how we can use flexible content elements to give us better control over our layouts and give us multi-column layouts, but we haven't actually built a whole new kind of content element like we would expect from an extension or specialized installation. Well, it just happens that we need to work on integrating consistent product displays into our website so we have the perfect situation to learn more about the FCE system. We could use the old **Image w/ Text** content type everywhere or we could create our own new content element that would be on the same level as a customized **Image w/ Text** content element. What are the advantages of using an FCE for this little task, though?

- **Consistency**: All of the product displays will inherently look the same because the image size, link structure, and even product name placement can be controlled through the FCE.
- **Flexibility**: If we decide that we want all of the product images to be to the right of the descriptions one day, we can change the FCE, flush the cache, and see all product displays immediately change when we refresh our browsers. We can't do this with simple content elements styled in the RTE.
- **Speed**: If we spend a few extra minutes laying out a nice flexible content element, our editors will save at least a few minutes on every single product display they create; that translates to plenty of saved time and money pretty quickly.

So, our final FCE will be a new element to display products. We want to be able to show the name, description, price, and a link to our product in the store with our own consistent layouts, and we need the ability to have a few different styles of products based on CSS classes. We also want to make sure that it's easy to buy the product, so the name of the product and the image will always link to our product in the store.

Creating the HTML and CSS

Just like before, our first step is to create a good HTML template to build from. Obviously, there are a lot of ways that we could build a good product display, but here is some example HTML code that will put us all immediately on the same page:

```
<!-- BEGIN Product Element -->
<div id="product_ad" class="">
   <img id="product_image" />
   <div class="product_text">
      <p id="product_name"></p>
      <p id="product_description"></p>
```

```
        <p id="product_price"></p>
        <p id="product_link"></p>
    </div>
</div>
<!-- END Product Element -->
```

As you can see above, our template once again consists mostly of empty HTML tags with identifiers or classes that we will use in the CSS and TemplaVoila mapping. Right now, we can add our new HTML snippet to the `template_fce_snippets.html` file.

As part of our overall HTML/CSS template creation, we can go ahead and setup our stylesheet changes now. This is just for our example site, so there's no reason to go crazy on CSS customization, but we do want to make a few changes to make sure that the product graphic floats nicely to the left of the description. We're going to restrict the overall width of a product display to 450 pixels and the width of the text to 290 pixels. The image will float to the left, and its size will be set by the TypoScript in the data structure so that the server will handle any resizing that is necessary. Go ahead and add this small chunk of CSS to the bottom of your `style.css` file to make these updates:

```
#product_ad {
    width: 450px;
}
#product_ad div.product_text {
    width: 290px;
}
#product_description {
    margin-bottom: 5px;
}
#product_price {
    font-weight: bold;
}
#product_ad img {
    float: left;
    margin-right: 10px;
}
```

Creating a customized data structure

Now we're ready to create the actual data structure. The process is going to be basically the same as the other content elements, but we are going to spend more time tweaking each field to work exactly the way we want.

Our first step is always to create a new data structure again by clicking **Create...** next to the `fileadmin/templates/template_fce_snippets.html` label. Go ahead and map the root field to the main `div` tag with the ID of `product_ad`. Like before, we need to set the mapping mode to **OUTER (Include tag)** so that the `product_ad` ID attribute stays in the HTML tag.

Product name

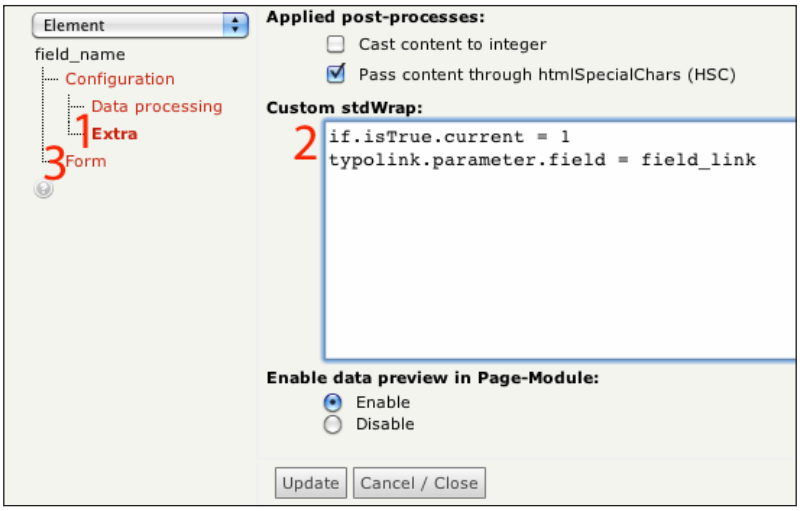
The first field we are going to create is the product name field, so go ahead and create a new field named `field_name` with some basic settings:

- **Field:** `field_name`
- **Element**
- **Title:** `Product Name`
- **Sample Data:** `[Product name]`
- **Element Preset**: **Plain input field**

1. Go ahead and save the initial configuration by clicking on the **Add** button. After you have saved the original settings, go ahead and click on the **Extra** link under the **Configuration** heading (pictured in the following screenshot). We want to wrap a link to the product page around a few of our fields including the name field, so we are going to add some TypoScript to the **Custom stdWrap** parameter for `field_name`. We are going to use two lines of TypoScript for this:

```
if.isTrue.current = 1
typolink.parameter.field = field_link
```

2. The first line checks to see that the `field_name` currently exists. We do this this so that TYPO3 only creates a `link` tag when the product name exists. The second line creates a `typolink` wrap around the `field_name` output. In addition, it gets the link parameters, or destination and target of the link, from a field named `field_link`. We will be creating this data structure field later on as a link to the product page, but we can go ahead and start using it to add dynamic links to specific data structure elements. Before we click on **Update**, our screen should look like this:



3. We are almost done with the product name field, but we are going to tweak the form display itself before we're done. Go ahead and click on the **Form** link on the left side of screen (below the **Extra** link we just clicked on). You'll see that it shows the field label and some code in an XML format called FlexForm that TYPO3 uses for backend form configurations. We want to limit how large product names are, so we are going to change the size from `48` to `20`. In addition, we want to add a checkbox to the field, so that it's easy to quickly unset the field. Some of our editors have probably already gotten used to using this for other fields, so we'll make their job easier by adding a checkbox to the text input. We will also leave `trim` in the `eval` field, which will trim the whitespace from before and after the name. After those two minor changes, here is our new form configuration code:

```
<type>input</type>
<size>20</size>
<checkbox></checkbox>
<eval>trim</eval>
```

4. Now we can save our changes to the name field by clicking on the **Update** button. Click on the **Map** button to map the product name field to the paragraph tag with the ID of `product_name`.

Product class

We are going to create a class field so that we can change the style for different product types through CSS. The easiest way to modify the styling of our entire product display is to modify the class attribute on the main `div` tag, and we can use a drop-down to make sure that our editors only have to deal with a list of known categories or product types. They'll never know that they are dynamically altering the class attribute for the main display; they'll just know that sometimes the product image is on the other side or the product name is a different color. Like before, let's go ahead and create an initial field before we start modifying it with the following settings and make sure we set it as an **Attribute**:

- **Field:** `field_class`
- **Attribute**
- **Title**: `Class`
- **Element Preset**: **Selector box**

As we have a drop-down, or selector box, we are going to need to modify the form configuration using FlexForm XML to make it work completely. We will only be using basic FlexForm tags, but you can see more information about FlexForms in the TYPO3 Core API (`http://typo3.org/documentation/document-library/core-documentation/doc_core_api/current/`).

1. First, go click on the **Form** link on the left side like when we updated the product name field.
2. Second, we can set the form label to `Category` in the text area labeled **Form label:** so that the backend form, at least, will show this as a category selection for the editors instead of a class selection.
3. Next, we are going to create the drop-down values to replace the default code in the wizard. We are going to insert one blank option in the drop-down so it's not required, and then each option after that is numbered with an index as part of the select array (see the following code snippet). For each option, there are two pieces: the label and the value. We are just setting up a normal HTML form select input, so these are the same values that we would normally be adding by hand. In the first example below, the label `Book` will be shown in the drop-down, and the value `product_book` will be assigned to the class of the main `div` if it is chosen. Here is the FlexForm code for our new drop-down:

```
<type>select</type>
<items type="array">
   <numIndex index="0" type="array">
      <numIndex index="0"></numIndex>
```

```
            <numIndex index="1"></numIndex>
        </numIndex>
        <numIndex index="1" type="array">
            <numIndex index="0">Book</numIndex>
            <numIndex index="1">product_book</numIndex>
        </numIndex>
        <numIndex index="2" type="array">
            <numIndex index="0">CD</numIndex>
            <numIndex index="1">product_cd</numIndex>
        </numIndex>
        <numIndex index="3" type="array">
            <numIndex index="0">Pet</numIndex>
            <numIndex index="1">product_pet</numIndex>
        </numIndex>
        <numIndex index="4" type="array">
            <numIndex index="0">Widget</numIndex>
            <numIndex index="1">product_widget</numIndex>
        </numIndex>
    </items>
    <default>0</default>
```

4. Now we can save the changes to our class field and map it to the main `div` with the ID `product_ad`. Choose **ATTRIBUTE "class" (=)** in the drop down before clicking on **Set**.

Product image

Let's go ahead and create a thumbnail graphic area for the product now. We are going to restrict the size of the graphic to fit our design in the TypoScript this time, and we are going to wrap it in the product link if the image is present. We'll start with basic settings again:

- **Field:** `field_image`
- **Element**
- **Title**: `Product Graphic`
- **Element Preset**: **Image field, fixed W+H**

We just set the element preset to be an image with fixed width and height dimensions, so the wizard will fill in some preliminary TypoScript for us if we click on **Add**. We do want to update the default values a little to set maximum width to 150 pixels and the maximum height to 200 pixels. We can click on the **Data processing** link on the sidebar to edit the TypoScript code for our field. The TemplaVoila data structure can actually handle all kinds of arbitrary TypoScript code and constants, but we don't always have a reason to play with this. We are just going to update the **TypoScript Code** text area with our new dimensions to optimize for portrait layouts using some basic TypoScript (see the TSref at `http://typo3.org/documentation/document-library/core-documentation/doc_core_tsref/current/` for more information):

```
10 = IMAGE
10.file.XY = 150,200
10.file.import = uploads/tx_templavoila/
10.file.import.current = 1
10.file.import.listNum = 0
10.file.maxW = 150
10.file.minW = 150
10.file.maxH = 200
10.file.minH = 200
```

Again, we are going to wrap the product page link around our image by adding our two lines of TypoScript to the **Custom stdWrap** text area:

```
if.isTrue.current = 1
typolink.parameter.field = field_link
```

After we've saved our changes, we can map our product graphic to the HTML `image` tag in our template.

Product price

Next, we can add a special field for the price with an automatic currency symbol. We are going to use the US dollar symbol, but we could just as easily use TypoScript to pull the local value for our currency symbol. For now, we need to create the field:

- **Field**: `field_price`
- **Element**
- **Title**: `Price`
- **Sample Data**: `[Price]`
- **Element Preset: Plain input field**

We are going to update the **Custom stdWrap** configuration again, but we are going to use it to prepend a currency symbol. Once again, we need to use a TypoScript conditional to make sure that the price exists with data because we never want a missing price to result in a currency symbol sitting by itself on a line. We can test that our price exists and prepend the currency symbol by adding the following to our **Custom stdWrap** area:

```
if.isTrue.current = 1
prepend=TEXT
prepend.value=$
```

A price should be a relatively short field, so we can limit its size in the form. In addition, we can add a checkbox for easy unsetting of the field by updating the form configuration just like the product name that we created earlier:

```
<type>input</type>
<size>10</size>
<checkbox></checkbox>
<eval>trim</eval>
```

Finally, we can save our price field and map it to the paragraph tag with the identifier `product_price` in our HTML template.

Product description

For our product description, we can use a default configuration for the field. Our only customization will be selecting **Text area for bodytext** as the element preset so that our editors will be inputting multi-line plain text without formatting a rich-text editor of any kind. We can create our new description field with basic settings:

- **Field**: `field_description`
- **Element**
- **Title**: `Product Description`
- **Sample Data**: `[Product description]`
- **Element Preset**: **Text area for bodytext**

Go ahead and map the description field to the `product_description` paragraph tag in the HTML and set the changes.

Text for product link

We are almost done, but we need to have text for the link at the bottom of the product display so the editors can display "Buy now!" or "Click here to learn more". We can create a basic text field, and our only modifications will be wrapping the text in the product display link and limiting the input field to 20 characters in the backend form with the following settings:

- **Field**: `field_linktext`
- **Element**
- **Title**: `Link Text`
- **Sample Data**: `[Link text]`
- **Element Preset**: **Plain input field**
- **Custom stdWrap**:

```
if.isTrue.current = 1
typolink.parameter.field = field_link
```

- **Form Configuration**:

```
<type>input</type>
<size>20</size>
<eval>trim</eval>
```

Map the link text field to the paragraph tag on the bottom of our template with the `product_link` ID.

Product link

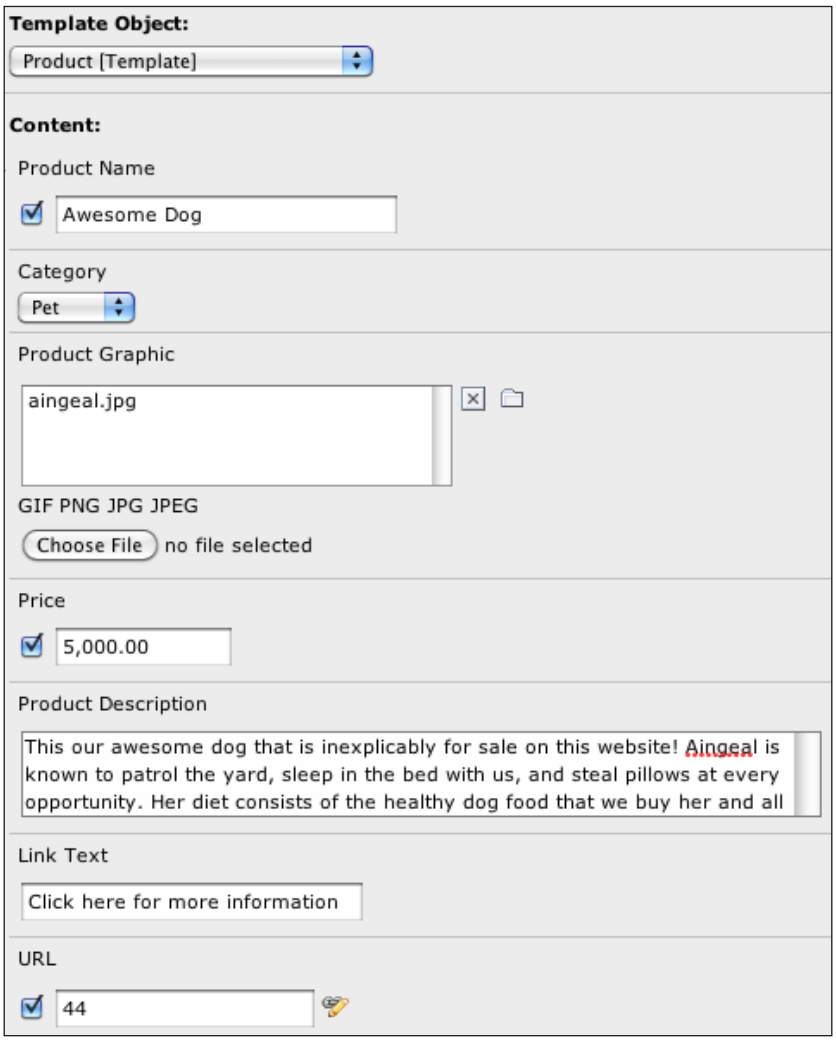
Finally, we are ready to create our link field for the main URL that we will be wrapping around our product name, graphic, and link text at the bottom of the display. First, we'll create the initial field to save our settings. Instead of **Attribute** or **Element**, we are going to choose **Not mapped** in the drop-down on the side. This means that we will not have to map the element to our HTML template, and it makes sense because we're using TypoScript in the data structure to use this element anywhere that we need it. We can create our final field with this configuration:

- **Field**: `field_link`
- **Not mapped**
- **Title**: `URL`
- **Element Preset**: **Link field**

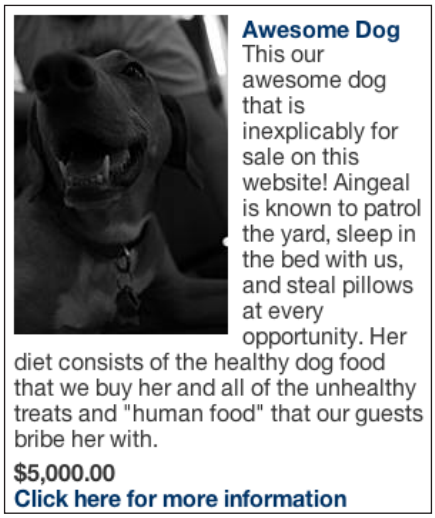
The link field does not have to be mapped, so we're done. Click on **Save as** and create a new content element named **Product** before saving all of our changes with the **CREATE DS / TO** button.

Viewing our results

Now, we can go to the **Page** view in any page in the backend to test out our new flexible content element. Like before, we can just create a new element, choose the **Product [Template]** content element, and we'll see our own very specialized form based on the data structure we just created:



Now, all of the layout and design is already taken care of; we can just fill in some basic text fields and select a picture of any size to get a uniform product display:



Summary

Okay, that's enough flexible content element information for this book, but there's still more to play around with and learn about FCEs. Like templates, FCEs can even have preview icons and layout files. For now, though, we've learned what flexible content elements area, when to use them, and how to build them. We created contact information FCE, an HTML wrapper, multi-column elements, and a nice product display element for our example site.

We also took some time to dive into the TemplaVoila data structure wizard more. It wouldn't have been smart to try to learn all of the complex ways we could modify the data structure in the earlier chapters while we were still learning how to create fields. Now that we have more experience, we got to learn how to add wraps and TypoScript directly into the data structure. We even learned how to modify the forms to control what our editors see in our content elements. We're becoming experts now.

Now that we've spent so much time learning TemplaVoila and becoming the local geniuses in flexible content elements, we're ready to take a step back and get back to our TypoScript roots. In the next chapter, we're going to learn some advanced TypoScript tools to help us tailor our site for different browsers and, most importantly, create a mobile version of our website Like always, take a break, grab some coffee, and come back ready to get into the little details.

9
Creating a Mobile Website

Okay, we've built a handful of templates, updated the backend look, and even learned how to create flexible content elements for our website. Now we need to start making sure that our website works everywhere. That means that we need it to work across all the desktop browsers that we are targeting and even mobile devices. This is one of the most important chapters, come to think of it. Almost anybody can create a website these days, but there's a surprisingly smaller amount of websites that adapt well to browser and operating system changes. There are a handful of reasons that all sites don't handle this better, but TYPO3 gives us a few advantages when websites work on mobile devices:

- **You don't have to create your own checks.** In static HTML and even many CMSs, there's no built-in way to check the browser on the fly. You might be able to use some CSS hacks, but those are fragile and can be broken by any browser updates in the future. Even if you do check for the browser, you'll have to create your own routines to handle the different environments as you run into them. TYPO3 already has browser-checking functions as part of its core and we'll be able to access it using basic TypoScript.
- **You don't need a separate mobile site.** It's common to create a whole "mobile experience" through what basically amounts to a new website. You might be able to share the content in an ideal world, but you'll still probably need to create a whole new set of templates just for the mobile devices in other CMSs. Sometimes, you'll even have to direct everyone to a mobile URL (like `http://m.example.com`) to clearly show the separation of your mobile website and your full website. We're going to use our TypoScript only to slightly tweak the normal experience for the Web; no new templates and no special URL are required, but we still use them if we want to.

This is the chapter that will show our boss that we're putting in the extra effort, and it shows that we're solving the same problems as the big companies. Here's the secret: it's really easy. Like we just saw in the list, TYPO3 is doing most of the work for us. We're going to let TYPO3 detect the user's environments and run TypoScript code specific to any browser, device, operating system, and so on that they are using. We just have to tell it what TypoScript code it should run.

In this chapter, you will:

- Learn how to use conditions in TypoScript
- Learn about all of the different conditions TypoScript can check
- Learn how to send a specific stylesheet to a browser (for example, Microsoft Internet Explorer)
- Learn how to detect a mobile device
- Create a mobile stylesheet
- Customize TypoScript elements for mobile devices
- Add a non-mobile link to the mobile version of our website
- Create a mobile subtemplate in TemplaVoila
- Learn how to redirect to an external mobile site when necessary

Introducing conditions

Before we create the mobile version of our website, we need to learn how to detect the browsers and devices our users are using to access our site. TypoScript includes built-in conditions to check user information such as browser, operating system, and language that we can use to specify which TypoScript code can run in a template. We are going to look at some of the most common conditions, but you can see the full, updated list in the TSref (`http://typo3.org/documentation/document-library/core-documentation/doc_core_tsref/current/`). These checks are similar to the PHP methods that programmers use to read server variables, but TypoScript makes them easier to use without an understanding of programming. In fact, we've already used conditions in our own template code. We were first introduced to TypoScript conditions when we added a link to our printable template in *Chapter 5* with the following code:

```
## Normal View Link (to run when print is greater than 0)
[globalVar = GP:print>0]
lib.printLink {
   value = Normal page view
   typolink {
      additionalParams >
```

```
        addQueryString.exclude  = print
    }
}

## Return to global processing
[global]
```

We used the condition above, `globalVar`, to check that the print variable was passed along in the URL with the syntax `&print=1`. If the condition is true, then the `printLink` object is modified. At the end, we used the `[global]` line to end the conditional section and return to "global" mode. The actual syntax after the equal sign in the condition, `GP:print>0`, is less important right now then the understanding that TYPO3 was able to check the condition against the current environment and run or skip the enclosed TypoScript based on the results of that test.

The universal syntax for conditions in TypoScript is pretty basic. All conditions are declared with square brackets (for example, `[browser = msie]`), and they can be combined using `AND` (`&&`) and `OR` (`||`) operators. Using an `AND` operator means that both values must be true for the condition to be valid, and an `OR` operator means that the condition is valid if either value is true. For example, we can use the condition statement below to deliver different TypoScript values to only Netscape browsers running on Windows:

```
[browser = netscape] && [system=win]
```

The `AND` operator is always read before `OR` operators, so we can also combine conditions to check to for Internet Explorer on any platform or Netscape on Windows like this:

```
[browser = msie] || [browser = netscape] && [system = win]
```

We are going to look at the following conditions in this chapter:

- `browser`
- `version`
- `system`
- `useragent`
- `language`
- `loginUser`
- `usergroup`
- `globalVar`
- `globalString`
- `userFunc`

Browsers

Syntax:

```
[browser = browser1, browser2,...]
```

One of the most common conditions to check for in TYPO3 is browsers. The syntax for the `browser` condition is `[browser = browser1, browser2,...]`. Each value in the condition is compared to the browser name and browser version concatenated together (for example, `msie6.5`). The condition is only checking to see if the value exists anywhere in the browser version from the user, so `ie` and `ms` will both match all Internet Explorer browsers. If we type `msie6`, then all Internet Explorer 6.x browsers will match. We can list as many browser values as we want after the equal sign, of course, and any match will cause the condition to be true (for example, `[browser = msie6.5, msie7, msie8]`). The table below lists the browser names that are available up through TYPO3 4.3.x.

Browser values	Browsers
`msie`	Microsoft Internet Explorer
`netscape`	Netscape Communicator
`lynx`	Lynx
`opera`	Opera
`ibrowse`	IBrowse (Amiga browser)

You'll probably notice that many of the options above are outdated, obscure, or even dead browsers and none of them are mobile browsers. Luckily, some TYPO3 developers noticed the same thing and overhauled the entire system for version 4.4 and later updates. If you are running TYPO3 4.4 or later, you can use any of the browser keys from the following updated table. These don't include mobile browsers, yet, but that should come in an update soon, and we can still use these values for fixing other cross-browser development problems:

TYPO3 4.4	Browser values
`msie`	Microsoft Internet Explorer
`firefox`	Firefox
`webkit`	WebKit
`opera`	Opera
`netscape`	Netscape
`konqueror`	Konqueror
`gecko`	Gecko
`chrome`	Chrome

TYPO3 4.4	Browser values
`safari`	Safari
`seamonkey`	SeaMonkey
`lynx`	Lynx
`amaya`	Amaya
`omniweb`	OmniWeb
`camino`	Camino
`flock`	Flock
`aol`	AOL Browser

Versions

Syntax:

```
[version = value1]
```

We can also check for the version of the browser using the `version` condition. Unlike the `browser` condition, the `version` condition checks for pure mathematical value using `=`, `>`, and `<` operators. All of the following examples would be true for Microsoft Internet Explorer 6.5:

```
[version = 6]
[version = =6.5]
[version = >5]
[version = <7]
```

Version operators	
[nothing]	The value must be part of the beginning of the version as a regular string (for example "6" or "6.5" matches "6.5").
`=`	The value must match exactly (for example, "6" does not match "6.5").
`>`	The version number must be greater than the value (for example, "6.5" is greater than "5").
`<`	The version number must be less than the value (for example, "6.5" is less than "7").

Operating systems

Syntax:

```
[system = system1, system2, ...]
```

We can also check for the user's operating system. The following table is directly from the TSref, and you'll notice that many of the specific operating systems may be outdated. Like the browser condition, TypoScript is only looking for a match for the value somewhere in the key. We can simply use `[system = win]` to match all Windows operating systems.

System values	
`linux`	Linux
`unix_sgi`	SGI / IRIX
`unix_sun`	SunOS
`unix_hp`	HP-UX
`mac`	Macintosh
`win311`	Windows 3.11
`winNT`	Windows NT
`win95`	Windows 95
`win98`	Windows 98
`amiga`	Amiga

User agents

Syntax:

```
[useragent= agent]
```

The `useragent` condition can be used as a direct match on the user agent string that the server passes along during the session. Every browser sends a user agent string showing all of the information about the operating system and browser version when it accesses a website. For example, the iPhone 4 sends out a user agent string like this:

```
Mozilla/5.0 (iPhone; U; CPU iPhone OS 4_0 like Mac OS X; en-us)
AppleWebKit/532.9 (KHTML, like Gecko) Version/4.0.5 Mobile/8A293
Safari/6531.22.7
```

In our other condition variables, TYPO3 has parsed this user agent itself to provide us with easy system and version selections. Unfortunately, the other solutions don't have mobile devices yet. If you are trying to work with mobile phones or other devices with very specific user agent, this is the fastest and easiest way to target them. User agent information is also the easiest identification information to find online by searching for something like "`iPhone user agent`". As user agents tend to be long and overly-specific, we can use a `*` at the beginning and/or the end as a wildcard:

- `[useragent = *iPhone*]`

We will use this variable in a moment to check for mobile operating systems, but we're going to look at some other helpful conditions first.

Language

Syntax:

```
[language = lang1, lang2, ...]
```

The user's browser will also tell us what language the user prefers or will accept, and we can check against this with the `language` variable. The values must either be a straight match, or we can wrap our values in wildcards like the `useragent` variable (for example, `*en-us*`).

Logged in users

Syntax:

```
[loginUser = fe_users-uid, fe_users-uid, ...]
[usergroup = group1-uid, group2-uid, ...]
```

If a user is logged in, we can evaluate against their user group or identification number as shown in the following examples:

```
[loginUser = 7,13,19]
[usergroup = 5,12,92]
```

Of course, if a user is not logged in, then they will have no user identification or current group associated with them; only a wildcard (`*`) would evaluate as true for `loginUser` or `usergroup` in that case.

Global variables and strings

Syntax:

```
[globalVar = var1=value, var2<value2, var3>value3, ...]
[globalString = var1=value, var2= *value2, var3= *value3*,...]
```

Like we saw with the `print` variable, TypoScript can handle global variables or strings that are passed in either through the session information or the URL. From our printable example before, the following condition will match a URL that includes `&print=1`:

```
[globalVar = GP:print > 0]
```

User function

Syntax:

```
[userFunc = function(parameter)]
```

Finally, we can create our own user functions for conditions with some PHP. We create the function, we decide what it checks, and we decide when it returns a true or false value. For example, we could add code similar to the following to our `localconf.php` to start checking mobile devices:

```
function mobile_check($cmd){
   switch($cmd){
      case "appleDevices":
         if(strstr($_SERVER['HTTP_USER_AGENT'],'iPhone') || strstr($_
SERVER['HTTP_USER_AGENT'],'iPod') || strstr($_SERVER['HTTP_USER_
AGENT'],'iPad')) {
            return true;
         }
      break;
      case "androidDevices":
      // ....
      break;
   }
}
```

Then, we could check for Apple mobile devices with the following condition:

```
[userFunc = mobile_check(appleDevices)]
```

This relies on PHP knowledge, so we aren't going to be using it in this chapter to detect special circumstances like mobile browsers. If you are comfortable with some PHP coding, I recommend building your own user function to test for the most common mobile devices as an exercise. The `userFunc` condition can be very powerful.

Testing browser compatibility

Using our new information about conditions, we can target browsers through our TypoScript more precisely. This is probably the most common use for conditions, after all. It's great to complain about how much a certain browser doesn't support the cool trick we just learned, but our boss and our users really don't care. At the end of the day, our site has to be usable on the visitor's machine, and that may mean that we have to detect the browser. The good news is that this is a very elegant solution to CSS or image problems. Instead of using some of the usual CSS hacks like special

CSS commenting structure that won't we be read correctly by Internet Explorer, we can actually detect the browser at the beginning and only send them the corrected stylesheet.

We can add browser-specific stylesheets using a browser condition and our `headerData` properties. We can create one stylesheet, `common_ie.css`, with all of our Internet Explorer fixes that is only served to Microsoft Internet Explorer browsers. At the same time we can create a final stylesheet, `common.css`, with code that could potentially break Internet Explorer (like `text-shadow: gray 3px 3px 5px;`) that is served to everyone else. To send different stylesheets to the different browsers, we can add this to our TypoScript template:

```
page.headerData.30 = TEXT
[browser = msie]
page.headerData.30.value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/common_ie.css" />
[else]
page.headerData.30.value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/common.css" />
[end]
```

We can also use this same structure to modify our TypoScript objects. So, if we wanted to use a great PNG logo with transparency on our website, we don't have to worry about older versions of Microsoft Internet Explorer anymore. We can use a conditional to send a GIF image to Internet Explorer 5.5 and 6, and still send our original PNG to the rest of our users:

```
lib.logo = IMAGE
lib.logo.stdWrap.wrap = <a href="http://www.example.com/">|</a>
[browser = msie]&&[version= <7]
lib.logo.file = fileadmin/templates/logo_for_ie.gif
[else]
lib.logo.file = fileadmin/templates/logo.png
[end]
```

Using conditional statements means that we no longer have to use undocumented CSS to get around or be worried about "quirks mode" in a browser breaking our carefully crafted template. Instead of sending data that we don't expect the troubling browser to parse, we're only sending our browser-specific data to the browsers that can handle it correctly.

Creating a mobile version of your website

Now we can get a little more complex and create something that your boss has probably been asking about for a year already: a mobile version of our site. Ideally, of course, our site might already look okay in a modern mobile device, but that's not always the case. In our example site, we are loading some unnecessary graphics, our two column systems do not scale well for readability, and our default fluid layout breaks pretty easily. You can see what our example looks like on an iPhone as shown in the following screenshot:

Our website is still functional, but we can make it truly optimized with just a few extra steps.

Detecting a mobile device

The first thing that we need to do to create our mobile site is decide how we are going to detect mobile users. Currently, there are no conditions for this in TYPO3 because the mobile condition that was built into TypoScript is outdated. There are developers working on updated solutions right now, and they should be in a future version of TYPO3. We could create a custom user function, but we'd have to know enough PHP to pull it off skillfully without slowing down or breaking our current site. For now, our best option is using the `useragent` condition in TypoScript. The good thing about the `useragent` condition is that it is pulled directly from the client's information, so it can never be outdated. On top of that, we can use `*` as a wildcard before and after the key that we are looking for, so we do not have to be overly explicit with our syntax. We can detect any iPhone (no matter what the hardware or software version) with one simple line:

```
[useragent = *iPhone*]
```

Now we just need to find out what all of the user agents that we need are. The available mobile devices are changing so rapidly now that there's no point in listing them all in a book. Instead, I've gathered together some of the most common user agents for our example. If we were going live with this site today, we could use an analytics tools such as Google Analytics (`http://www.google.com/analytics/`) to find out what is most popular on our site and search for the user agents through search engines and forums. For our example, we'll go with the most popular keywords:

- iPhones have the word `iPhone` in the user agent.
- iPod Touches have the word `iPod` in the user agent.
- Mobile devices using the Android operating system have the word `Android` in the user agent.
- Mobile devices that use the mobile version of the Opera browser have the words `Opera Mini` in the user agent.
- BlackBerry phones have the word `BlackBerry` in the user agent.

TypoScript doesn't allow us to combine all of these keywords into one condition check, but we can list each user agent check on the same line:

```
[useragent = *iPhone*]||[useragent = *iPod*]||[useragent =
*Android*]||[useragent = *Opera Mini*]||[useragent = *BlackBerry*]
```

Creating a mobile stylesheet

Okay, we've figured out how to detect our users' mobile devices, so now we need to start working with them. The first thing we can do is serving up a mobile stylesheet. Of course, there are CSS tricks to serving up a mobile stylesheet, but we don't need to use them because we've already done the hard part of detection. We just need to add a `headerData` object into our template after our mobile browser condition:

```
page.headerData.30 = TEXT
page.headerData.30.value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/mobile.css" />
```

Now we just need to create that stylesheet for mobile devices in our templates directory. Go ahead and create a blank file called `mobile.css` inside of the `fileadmin/templates/` directory right now, and open it up in a text editor so we can start adding some new rules. Remember, we're adding this into the header after the main stylesheet, so we only need to update the values we care about. We're just going to use our mobile stylesheet to tweak a few things:

- We need to resize our text; nobody wants to read 10px or 12px text on a tiny screen. We don't need to make it huge, but 24px text should be nice and easy to read from an arm's length distance.
- We are going to switch to a text-based menu as well, so we need to update the font sizes and margins for the menus.
- We are going to use the print link at the bottom as a link back to the full version of the website. We need to make sure it is centered and easy to find.
- Finally, we really don't need to pull off our two-column designs on a little screen. So, we are going to turn off the floats and resize all of our columns to 100%. We need to reset the product advertisements at the same time, to keep everything clean and legible across mobile devices.

After taking all of that into account, we can update our `mobile.css` stylesheet with the following CSS code:

```
/* Resize all fonts for mobile devices */
p, ul, div, h2, h3, h4, h5, h6 {
   font-size: 24px;
   line-height: 30px;
}
/* Fix menu-area padding for text-based menu */
ul#menu-area {
   padding-bottom: 20px;
}
/* Resize menu items for mobile devices */
```

```
li.menu-item a {
   font-size: 30px;
   line-height: 36px;
   margin-right: 10px;
}
/* Resize footer link for mobile devices */
#print_link {
   padding-top: 20px;
   font-size: 30px;
   line-height: 36px;
   text-align: center;
}
/* Reset all columns and product ads to 100% width and single-column
on mobile devices */
#nested_column_1, #nested_column_2, #product_ad, #product_ad div.
product_text {
   float: none;
   width: 100%;
}
```

Customizing our TypoScript objects

Finally, we can update some of our TypoScript objects for mobile devices. The first thing we need to do is getting rid of unnecessary data, so we'll go after the timestamp. We definitely don't need to waste precious space in the browser restating the time on a mobile device that is already showing it in the status bar. All we need to do is wipe out the object with a simple line of TypoScript:

```
lib.timestamp >
```

The next area that we can clean up is the menu. The graphic menu items we created earlier are not translating well to the small screen, and we need to make our page load as fast as possible over slower data networks. We are going to use text-based menus to solve both problems. We can replace our graphic menu items with a text-based system simply by re-declaring the main menu object inside of our conditional statement:

```
lib.mainMenu = HMENU
lib.mainMenu {
  entryLevel = 0
  wrap = <ul id="menu-area">|</ul>
  1 = TMENU
  1.NO.allWrap = <li class="menu-item">|</li>
}
```

Finally, our last step will be to replace the current logo with a smaller graphic:

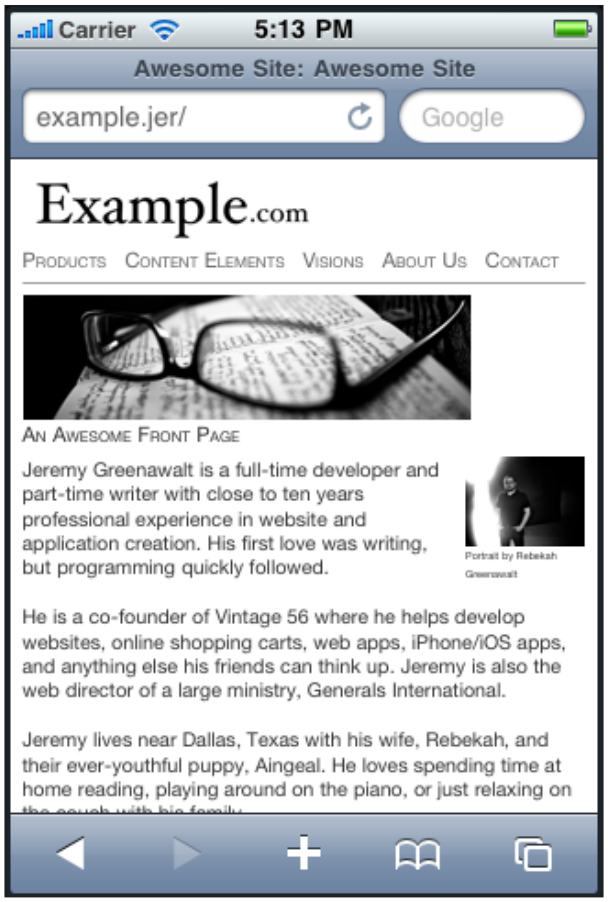
```
lib.logo.file = fileadmin/templates/logo_mobile.png
```

Bringing it all together

If we bring together all of our TypoScript modifications, we can add them to the main template setup just like this:

```
[useragent = *iPhone*]||[useragent = *iPod*]||[useragent =
*Android*]||[useragent = *Opera Mini*]||[useragent = *BlackBerry*]
page.headerData.30 = TEXT
page.headerData.30.value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/css/mobile.css" />

lib.timestamp >

lib.mainMenu >
lib.mainMenu = HMENU
lib.mainMenu {
  entryLevel = 0
  wrap = <ul id="menu-area">|</ul>
  1 = TMENU
  1.NO.allWrap = <li class="menu-item">|</li>
}

lib.logo.file = fileadmin/templates/logo_mobile.png

[end]
```

We need to to place `[end]` or `[global]` at the bottom of our conditional statements so TYPO3 knows when the condition is done and the rest of the template can be parsed. With all of our changes now, we have a faster, more functional mobile website:



Adding a non-mobile link

Creating a mobile website is great, but we should always offer an alternative link to the full (non-mobile) version of our website. This is most important if we've excluded features from the mobile version, but sometimes mobile users just want to see the full version of the website that they're used to from their laptop or desktop browsers. We can do this like when we handled the printable version of our template by using a global variable. For the purpose of our example site, we can use a global variable named `full`. If `full` is set to `1` as a `GET` or `POST` variable, then we won't show the mobile version changes to the website. To make sure that the variable is appended consistently to the link after it is set, we will declare it using `config.linkVars`:

```
config.linkVars = full
```

Unfortunately, TypoScript does not recognize nested conditions or nested conditional statements. We can't wrap parentheses around all of our `useragent` conditions, and `AND` always takes precedence over `OR` in TypoScript. We also can't nest the mobile conditional section inside of a condition for our new variable, and the entire condition check must be entered without line breaks. We'll need to add a global variable check to each `useragent` condition:

```
[useragent = *iPhone*] && [globalVar = GP:full<1] ||
[useragent = *iPod*] && [globalVar = GP:full<1] ||
[useragent = *Android*] && [globalVar = GP:full<1] ||
[useragent = *Opera Mini*] && [globalVar = GP:full<1] ||
[useragent = *BlackBerry*] && [globalVar = GP:full<1]
```

If we were using a programming language that used parentheses for `AND/OR` precedence, this is what our line would represent:

```
([useragent = *iPhone*] || [useragent = *iPod*] ||
[useragent = *Android*] || [useragent = *Opera Mini*] ||
[useragent = *BlackBerry*]) && [globalVar = GP:full<1]
```

This may seem a little complex at first, but remember that we are stretching a scripting non-programming language with our logical conditions at this point. If we needed to do this repeatedly, we might need to learn enough PHP to start creating our own extensions or create a special user function in `localconf.php`. Luckily for us, there are developers working on updates to TYPO3's mobile detection and creating extensions right now. Until those changes make it into the core, we can still do everything we need with one ugly line that we don't have to look at more than once. The trade-off is we don't have to learn PHP today or wait for the next TYPO3 version.

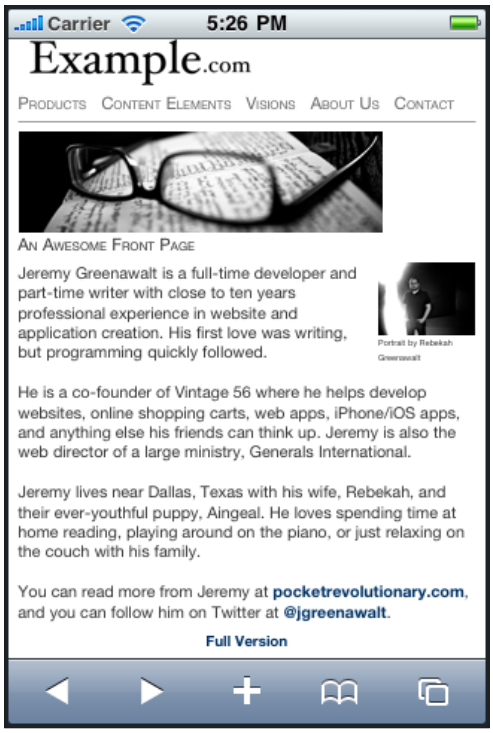
Now that we've updated our logic to check for this new variable, we need to create a link for our mobile users at the bottom of the page. Thankfully, we've already done this once for the printable templates and we can reuse our code with a little bit for tweaking:

```
[globalVar = GP:full<1]
   lib.printLink {
      value = Full Version
      typolink {
         parameter.data = page:uid
         addQueryString = 1
         addQueryString.exclude = id
         additionalParams = &full=1
      }
   }
[global]
```

You'll notice that we are still using the same TypoScript object, `lib.printLink`, as we used for the printable link. This is intentional so that we can actually replace the printable version link with the non-mobile version link on mobile devices. The `lib.printLink` object is already mapped in our templates, so we will automatically get our new link at the bottom of every page when somebody is viewing the mobile version. Of course, this means that they will not have a printable version link in the mobile view, but that is really not an issue for mobile users. To finish up, we can combine our different TypoScript declarations into one complete mobile section in our template setup:

```
config.linkVars = full
[useragent = *iPhone*] && [globalVar = GP:full<1] || [useragent =
*iPod*] && [globalVar = GP:full<1] || [useragent = *Android*] &&
[globalVar = GP:full<1] || [useragent = *Opera Mini*] && [globalVar =
GP:full<1] || [useragent = *BlackBerry*] && [globalVar = GP:full<1]
   page.headerData.30 = TEXT
   page.headerData.30.value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/css/mobile.css" />

   lib.timestamp >

   lib.mainMenu >
   lib.mainMenu = HMENU
   lib.mainMenu {
      entryLevel = 0
      wrap = <ul id="menu-area">|</ul>
      1 = TMENU
      1.NO.allWrap = <li class="menu-item">|</li>
   }

   lib.logo.file = fileadmin/templates/logo_mobile.png

   lib.printLink {
      value = Full Version
      typolink {
         parameter.data = page:uid
         addQueryString = 1
         addQueryString.exclude = id
         additionalParams = &full=1
      }
   }
[end]
```

Now we have a nice link back to the full version of our website on mobile devices:



Creating a mobile subtemplate

We've updated the CSS and TypoScript objects for our mobile version, but sometimes we might need to change the actual HTML for our mobile website. Using TemplaVoila, we will create a new template object for mobile devices just like we created a subtemplate for the printable version of our main template.

Adding a new option to our subtemplate pages

First, we need to setup a "rendering type" for our mobile template objects to show up in the TemplaVoila editing pages for subtemplates. In *Chapter 5*, we saw that TemplaVoila let us choose "Printer friendly" as a type of rendering when we created our printable template. Out of the box, that is the only type of rendering that TemplaVoila lets us choose, but we can update our TYPO3 site to show any rendering types that we want to create. To add a mobile type, we need to update the TSconfig on our main page.

In *Chapter 7*, we updated our rich text editor in the TSconfig, and we're going to add a line of TypoScript for our new mobile rendering in the same place. In the page tree, right-click on the icon for the root page and select **Edit** just like we did to get to the page properties. Click on the **Options** tab to see the TSconfig for our main page. Add this line to the bottom of the TSconfig:

```
TCEFORM.tx_templavoila_tmplobj.rendertype.addItems.mobile = Mobile
version
```

That line of TypoScript will add a new item, `mobile`, to the render types available for TemplaVoila subtemplates and show it in the drop down as `Mobile version`. After you have added that to your TSconfig as shown in the following screenshot, clear the cache in TYPO3:

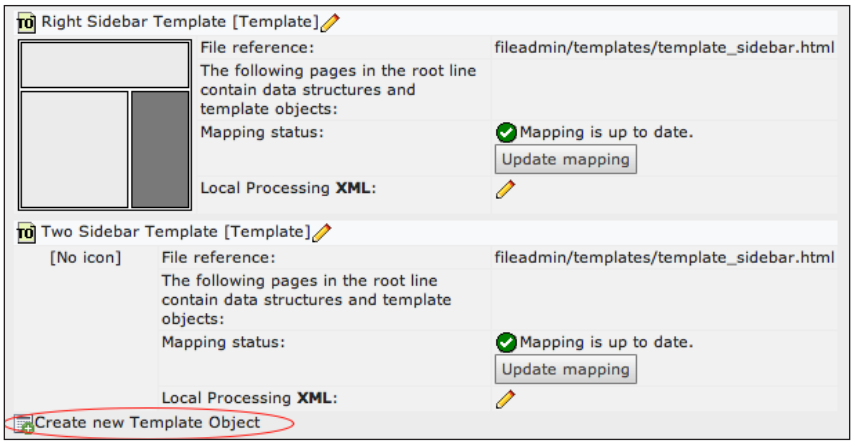
Creating a new TemplaVoila template for mobile devices

Now we can create a new template object for our mobile version. Just like the printable template in *Chapter 5*, we're going to create this as a subtemplate to the main template.

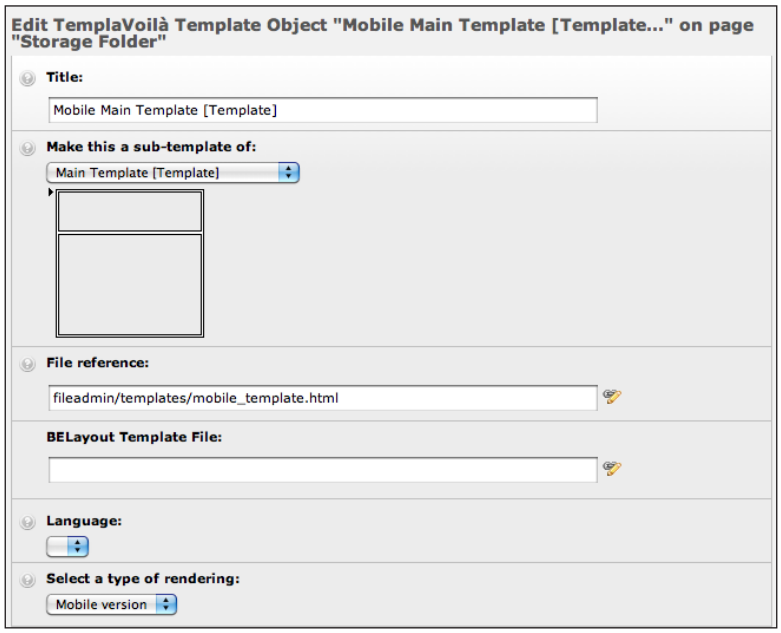
1. Create a new HTML file for the mobile versions. We can copy our original HTML file, `template.html`, in the `fileadmin/templates/` directory and just make a few modifications. First, name the new file `mobile_template.html` so we can identify it quickly on the server. Next, remove the `div` tags for the timestamp and the banner. We don't need either of these for our mobile version. Your mobile HTML template should look like this:

```
<!DOCTYPE HTML>
<html>
   <head>
      <meta charset="utf-8" />
   </head>
   <body>
      <div id="logo"></div>
      <ul id="menu-area"><li class="menu-item"><a href="">Menu
Item #1</a></li></ul>
      <ul id="submenu-area"><li class="submenu-item"><a
href="">Submenu Item #1</a></li></ul>
      <div id="content">This is our content</div>
      <div id="print_link"></div>
   </body>
</html>
```

2. Next, we need to create a new template object in the TemplaVoila module. Under the **Main Template** in the TemplaVoila module, click **Create new Template Object**:



3. Now we just need to fill in the form for our new template object. The title of our new template is **Mobile Main Template [Template]**, and we need choose **Main Template [Template]** from the drop-down that is labeled **Make this a sub-template of:**. Choose the HTML file we just created, `fileadmin/templates/mobile_template.html`, as the file reference. Finally, we can select **Mobile version** as our rendering type. If your form looks like the following screenshot, save your changes and return to the main TemplaVoila screen.



4. Click **Update mapping** next to our new template object, and map our new template. Go ahead and map the root, main content area, both menus, logo, and printable link fields to your HTML.

Adding our subtemplate to the TypoScript template setup

Finally, we can add the subtemplate to our TypoScript template. The printable version that we created in *Chapter 5* was automatically chosen, but we need to add one line of TypoScript to our template setup to choose this subtemplate inside our condition for mobile devices:

```
[useragent = *iPhone*] && [globalVar = GP:full<1] || [useragent =
*iPod*] && [globalVar = GP:full<1] || [useragent = *Android*] &&
[globalVar = GP:full<1] || [useragent = *Opera Mini*] && [globalVar =
GP:full<1] || [useragent = *BlackBerry*] && [globalVar = GP:full<1]

   page.10.childTemplate = mobile

   page.headerData.30 = TEXT
   page.headerData.30.value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/css/mobile.css" />

   ## The rest of our mobile code from above goes here
```

Remember that the TemplaVoila wizard added the following lines to our TypoScript setup automatically in *Chapter 1* when we setup our site:

```
page.10 = USER
page.10.userFunc = tx_templavoila_pi1->main_page
```

As we can see from those two lines, `page.10` represents the user function that pulls in our TemplaVoila templates and their mappings. Now, we are assigning the mobile template as a child template (also known as a subtemplate) to render our pages in the frontend with the `childTemplate` property if the mobile conditions are met. This means that for any template object we use, it will look for a subtemplate with the `mobile` rendering type. If it cannot find a subtemplate with the `mobile` rendering type (like our newsletter), then TYPO3 will simply show the normal template with our CSS and TypoScript changes.

If we refresh the cache in TYPO3, we can see our front page is using the simpler TemplaVoila template with no top banner:

Redirecting to an external mobile site

Our TypoScript, TemplaVoila, and CSS modifications will work for most websites, but sometimes we need to work with a special mobile site outside of the main TYPO3 installation. Often, companies will have a special website with a unique URL (for example, `http://m.example.com`) built in one of the newest mobile frameworks like Sencha Touch (`http://www.sencha.com/products/touch/`) or jQuery Mobile (`http://jquerymobile.com/`) that is optimized for Apple and Android devices. Luckily, we can make use of TypoScript to redirect visitors to a different URL and use the same condition statements; we just used to make sure everybody is covered.

We can use the `config.additionalHeaders` property in TypoScript to add information into the HTTP headers of our website. The HTTP headers can be used for different operations like controlling client-side caching and languages, but all we care about is the `Location` field in the headers. If we add header information with a new location (URL), then visitors will be automatically redirected to the new URL. This is the same technique that many websites use for missing or blocked pages, but we are going to use it to send people to `http://m.example.com`:

```
config.additionalHeaders = Location: http://m.example.com
```

Now, we still have a pretty good mobile device for many devices such as BlackBerry phones and Opera Mini, so we don't need to redirect everybody. We can setup a special condition for iPhones, iPods, and Android devices to go to the mobile website that our developers built with jQuery Mobile, and we can send other mobile devices to our current mobile version of the TYPO3 site:

```
[useragent = *iPhone*] && [globalVar = GP:full<1] || [useragent =
*iPod*] && [globalVar = GP:full<1] || [useragent = *Android*] &&
[globalVar = GP:full<1]

   config.additionalHeaders = Location: http://m.example.com

[end]

[useragent = *Opera Mini*] && [globalVar = GP:full<1] || [useragent =
*BlackBerry*] && [globalVar = GP:full<1]

   page.10.childTemplate = mobile

   page.headerData.30 = TEXT
   page.headerData.30.value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/css/mobile.css" />

   ## The rest of our mobile code from above goes here

[end]
```

Summary

Congratulations, we've learned what we need to do to make a better mobile version of our website, but, as importantly, we've learned how to use conditions in TypoScript to optimize our template for the users' needs. We covered the big two issues, browser dependency and mobile websites, but there is obviously a lot more we could do with these conditionals. As we saw in the first section, TypoScript has conditions for everything from the user's operating system to the day of the week on the server. This means that we can go even further with our customizations if we wanted and post special code every Friday, or we could send a different template to users who are coming from an internal IP address. On top of that, we can always use global variables and user functions to add our own special conditions. How do we do use this going forward?

First, we can send browser-specific TypoScript code with just a basic condition statement. We're not relying on CSS hacks, undocumented JavaScript, or "quirks mode" to make sure that our site renders correctly. We only have to send the stylesheets or TypoScript values that we want to send to each browser. If we have a specific stylesheet for Microsoft Internet Explorer, we can send it using a browser check. If our fancy menu breaks in Firefox 2.6, we can send a different menu to that browser. Most importantly, we don't have to know PHP or complex HTTP information to make conditional statements; it's just TypoScript.

Second, we can create a mobile site without building a whole new website. Everything we did to create a mobile site added up to twenty four lines of TypoScript, and half of that was updating the menu and logo objects. Of course, we also saw how we can create custom templates and redirect to mobile websites as well. Many site creators' aren't creating custom mobile versions because it will be too hard. By using TYPO3, that's not a problem for us. We can rely on the hard work already invested by the core developers to make our jobs easier.

Finally, we can think up our own ways to use conditions in our websites. Most of the time, it's too complex to know simple things about the operating environment unless we're used to hand coding PHP. Using conditional statements, we already have a whole toolbox ready to go. If we're building an internal website with logins and user groups, we can swap stylesheets and change the look and feel any time a user in our "Senior Executives" group logs in. Our boss will be impressed enough that we have a mobile website, but he'll/she'll be more than impressed when he/she logs in to the frontend site and sees a template specially tweaked for the executives' needs.

In the next chapter, we're going to learn about the next jump we can make to take our website experience worldwide: internationalization and localization. Those are some of the biggest buzzwords in web development along with "browser-specific" and "mobile", and we'll learn all about them in a minute. Just like the mobile website, we're going to see that TYPO3 has done most of the job for us already. All we have to do is use the tools its already provided. So, go show the mobile website to your boss real quick on his/her iPhone, grab some coffee, and come back ready to take our little example site multilingual.

10
Going International

We've optimized our templates, and we've made our site accessible to mobile devices all over the world. How big can we really grow, if only English-speaking users can use our site? We've opened ourselves up to close to a billion mobile users, but we need to start speaking their language if we want our website to be truly international. As more groups worldwide start using the Web regularly, internationalization and localization are no longer just nice features to have. We all know that our boss will not be happy in a few years if we say that we'll have to rebuild the entire website to enter a new country. Luckily, internationalization and localization are two ideas that are built into the core of TYPO3 from the beginning, and TemplaVoila only makes them easier to use. In fact, the ease of localization is one of the great strengths of TYPO3 when compared to other CMSs, and we're going to see how simple it really is in this chapter.

In this chapter, you will:

- Learn about the difference between internationalization and localization
- Add a new language to the website and start translating individual pages
- Learn how to hide pages that you don't want to translate
- Learn how to translate content elements
- Create a frontend menu for visitors to choose their own language
- Learn how to localize dynamic elements like the logo
- Create a special TemplaVoila template for a new language

Introduction to internationalization and localization

The first thing we need to clarify is our definitions of internationalization and localization. Not everybody agrees on the proper usage of the terms, and some developers still incorrectly use them interchangeably. For the purpose of TYPO3 development, though, they do have very specific and subtly different meanings.

Localization is the adaptation of an application or website to a specific audience or locale. It is sometimes used as a synonym for translation, but localization can be more complex than just changing the language. Full localization often involves adapting date and time formats, currency, graphics, legal requirements, and any other content that may be unique to a specific market. TYPO3 gives us built-in ways to localize our content with different languages, currencies, date formats, and more.

In documentation and forums, localization is often shortened to l10n because there are 10 letters between the l and the n.

Internationalization is the high-level design of an application or website to allow easy localization. Typically this involves a lot of work for the developers and site creators if they aren't using TYPO3. The entire system must be able to handle Unicode characters so that it can handle Latin characters along with Chinese, Japanese, Cyrillic, and so on. Content such as article text, titles, menus, and graphics need to be separated from the actual structure of the website. The display of currencies, time formats, and other localized contents needs to be handled separately through libraries or modules. The layout has to be flexible enough to adapt to different lengths of words. There are even more considerations, and they usually have to be designed into the application from the very beginning. Luckily for us, internationalization has already been built into TYPO3 and is still being improved. TYPO3 handles the internationalization, and we are responsible for the localization.

Internationalization can also be called globalization or written as i18n in documentation and online forums.

Adding localization to a website

Now that we understand the overall concepts of internationalization and localization, it's time to start making our website ready for the rest of the world. The good news is that internationalization is already built into TYPO3 for the most part. All we need to do is start the localization process by adding new locales to our website. We're going to go through each step of this process, but the overall workflow in TYPO3 is pretty basic:

1. Add an alternative language to the website.
2. Add our new language to the pages we want to translate.
3. Add translations to the content already existing in our default language.
4. Add a language menu so that frontend visitors can select their own language.
5. Localize any logos or graphics.

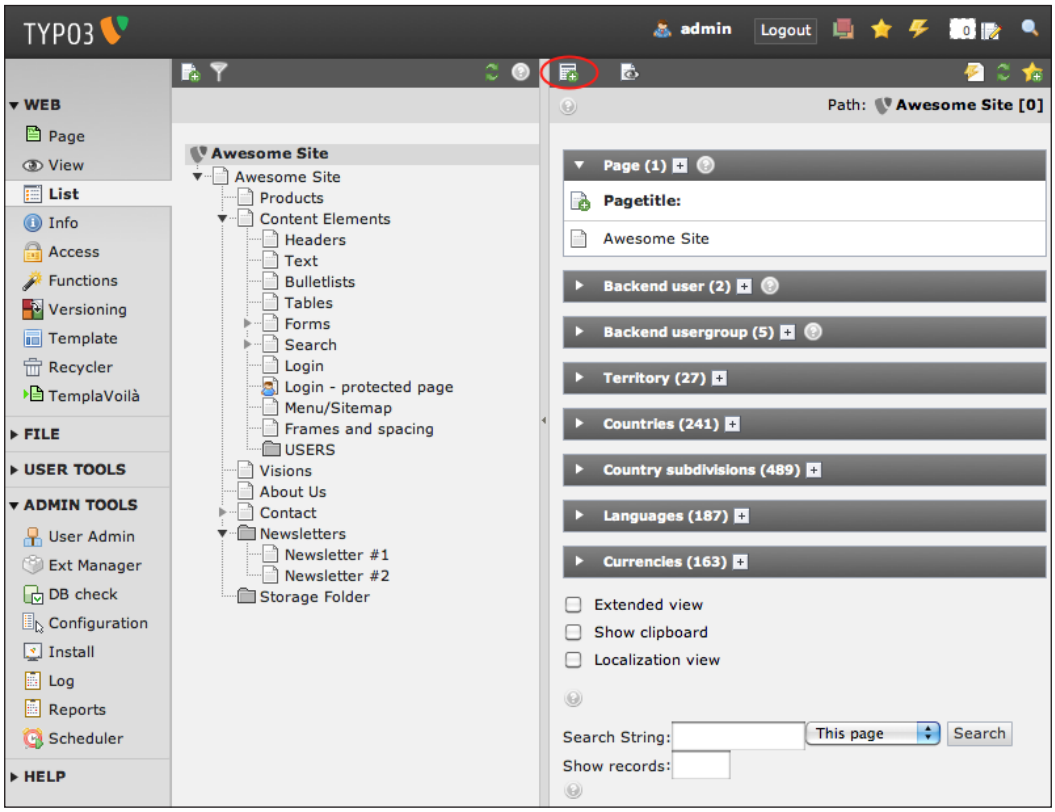
Of course, there are more things that we can tweak and play with, but that is the basic workflow that can be used to add languages and localization features to any modern TYPO3 site. We're going to start by adding the two languages of my ancestors: Irish and German. I'm not really fluent in either language, and I don't expect you to read them; we'll be using filler text to make the examples easier to follow.

We are going to use one page tree to hold multiple translations, but another method of TYPO3 localization is called the "two-tree" concept where a completely separate page tree is created for each locale. This is helpful if the pages or content are completely different, and you can read more about it at `http://typo3.org/documentation/tips-tricks/multi-language-sites-in-typo3/`.

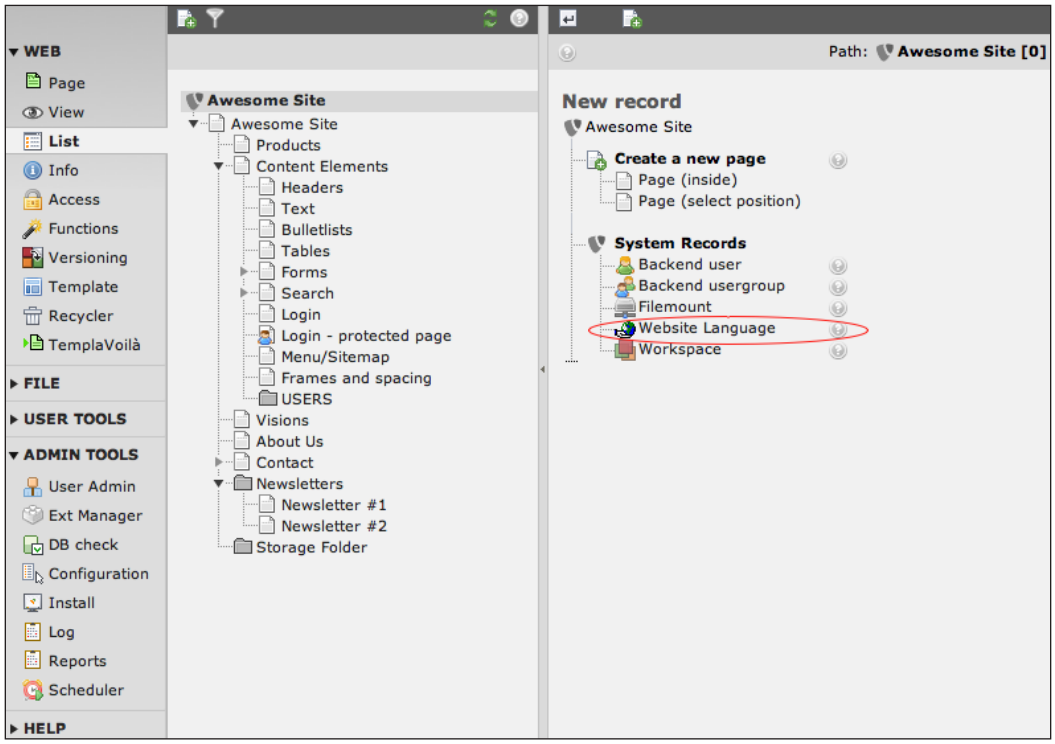
Adding a website language

Our first step in any translation project is to add a new alternative language to our website. TYPO3 is already aware of almost any language we want to use through the `static_info_tables` extension (which is installed by default), so it will already have flags and ISO 3166 country codes available (see `http://www.iso.org/iso/country_codes.htm` for more information on country codes). We just need to tell it we want to start using one of the optional languages:

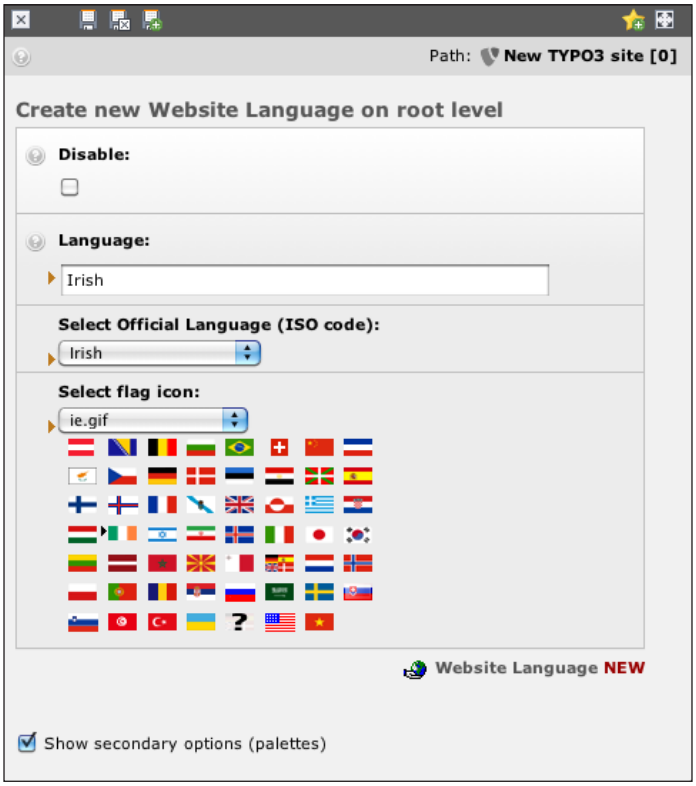
1. In the backend of TYPO3, choose the website object on the sidebar in **List** view and select the root in the page tree. Then, click on the **Create new record** button (circled in the following screenshot).



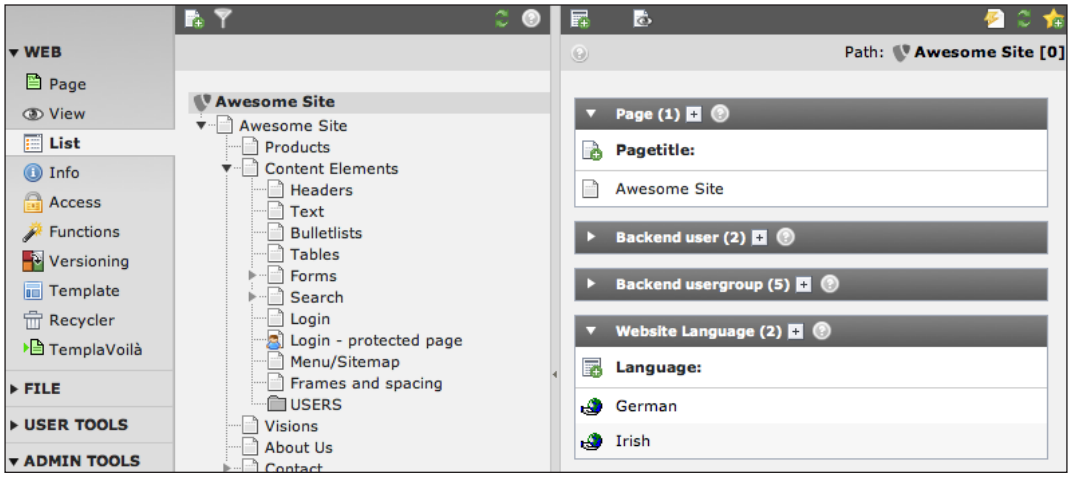
2. Under **System Records**, click on the **Website Language** button (circled in the following screenshot) to create a new TYPO3 language record.



3. We only need to fill out a few fields before we save our new **Website Language** record to the database. Obviously, we don't want to disable the language, but we could edit this record later to disable the language for the entire website using the **Disable** checkbox if we no longer offered our website in a given language. The first record we're going to create will be for the Irish language, so we will fill in the **Language** field appropriately. This is actually just a title for the backend editors to see, so we can give it any value we want. We want to make it easy, by calling it simply `Irish`. The next drop down, **Select Official Language (ISO code),** is more important. This is the ISO code that will be used for TypoScript, core localization, and extension translations. Many extensions already have translations for the most common languages, so this ISO code will be used to grab the translated labels, descriptions, and titles automatically. We will choose `Irish` from the drop-down menu. Finally, we can select a flag icon for our **Website Language** record. This icon is used to identify translated pages or content elements in the backend. Go ahead and choose `ie.gif` from the menu.



4. After we have filled out the fields, we can save our new record and repeat our steps for German. After creating both **Website Language** records, we can see them in the **List** view on the website root:



Adding your languages to TypoScript

Now that we have added some languages to our website we have to update our main TypoScript template. The nice thing is that we don't have to actually break anything that is currently working in the template, but we do have to add the ability to track which language the frontend user wants to see. The easiest and most reliable way to figure out which language should be displayed for a given page in TYPO3 is through a URL variable.

TYPO3 is already using URL variables by default to decide which page to display. When you see the URL `http://www.example.com/index.php?id=108` in a TYPO3 site, you already know that it is going to be displaying a page with an `id` of `108`. All we are going to do is add a new parameter, `L`, to the URL by saying `config.linkVars = L`. That means we can go to `http://www.example.com/index.php?id=108&L=1` in the future and see the same page in Irish. In fact, the `L` parameter is a standard convention in TYPO3 that is used specifically for languages. We will use TypoScript to associate different values for `L` with the languages records we have already created. We've already used a URL variable for the printable version of pages, so this really just a refresher. We're going to add a condition like this to change our TypoScript for each value of `L`:

```
[globalVar = GP:L=1]
```

We are also going to set two configuration properties for the entire website to help us: `config.sys_language_mode` and `config.sys_language_overlay`. We can set `config.sys_language_mode` to the property `content_fallback`. This means that we do not have to set all of the fields including (but not limited to) title, text, and image for each content element. If we decide not to set a translation for one element, like the image, TYPO3 will fall back to the value in the default language. We can still set a translated version, of course, but this means that our content elements will not break if a field inside the content element is not translated. To use content fallback, we will add this line to our TypoScript template:

```
config.sys_language_mode = content_fallback
```

Next, we are going to set `config.sys_language_overlay` to the value `hideNonTranslated` for the default language. This will hide any content elements that we do not translate. The alternative, without this property set, would be to show the default language version of the content element for any non-translated elements. If you have five elements on a page, and you only translated three, then the two non-translated elements would simply show up in the default language. As we don't want to mix German and English or Irish and English text on the same page, we can set the template to hide our non-translated elements completely. Of course, like almost all TypoScript template values, these could be reset in an extension template for a specific section if that was necessary. As a note, this is only for hiding the content elements, but we will talk about hiding entire non-translated pages in just a moment. For now, we will have to hide our non-translated content elements with this line in our TypoScript template:

```
config.sys_language_overlay = hideNonTranslated
```

For each language, we are going to associate the language UID with the ID of the **Website Language** that we created. We will also use `config.locale_all` to configure how TYPO3 natively outputs time and date values and set the language code by adding these lines to our TypoScript for each language (`L=1` and `L=2`):

```
[globalVar = GP:L=1]
config {
   ## Use the Website Language record with a UID of 1
   sys_language_uid = 1
   ## Set the locale
   locale_all = ga_IE
   ## Set language code to ie
   language = ie
}
```

Finally, we are going to add a little bit of TypoScript to change the format of our date and timestamp in the corner by updating the `strftime` format (for more information on `strftime` formatting, see the TSref at `http://typo3.org/documentation/document-library/core-documentation/doc_core_tsref/current/`). The same date is shown in a different way for each area that we are targeting:

- United States: August 7, 2010
- Ireland: 07-08-2010
- Germany: 07.08.2010

For Irish, we can reformat our time stamp like this:

```
## Change timestamp to Irish format (DD-MM-YYYY)
lib.timestamp.10.strftime = %d-%m-%Y %T
```

For German, we can use this TypoScript code to update our timestamp:

```
## Change timestamp to German format (DD.MM.YYYY)
lib.timestamp.10.strftime = %d.%m.%Y %T
```

Remember, localization goes beyond just translating the text sometimes. Our goal is to look native to our target users, and we have to be aware of how we display date, time, and currency. Of course, currency normally needs to involve conversion rates and payment systems that can handle international payments, so we need to make sure we're not just changing symbols without thinking ahead.

Now that we have an understanding of what we're changing, let's go ahead and update our main TypoScript template. The following code can be added to the bottom of our main TypoScript template setup:

```
## Configure the default language (English)
config {
   sys_language_mode = content_fallback
   sys_language_overlay = hideNonTranslated
   locale_all = en_US
   linkVars = L
}

## Irish localization
[globalVar = GP:L=1]
config {
   sys_language_uid = 1
   locale_all = ga_IE
   language = ie
}
```

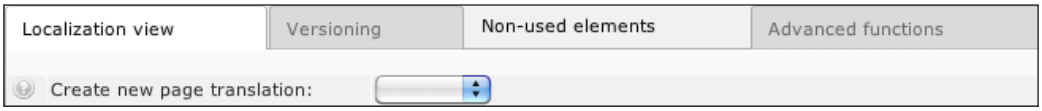
```
lib.timestamp.10.strftime = %d-%m-%Y %T

## German localization
[globalVar = GP:L=2]
config {
   sys_language_uid = 2
   locale_all = de_DE
   language = de
}
lib.timestamp.10.strftime = %d.%m.%Y %T

## Return to global processing
[global]
```

Adding localization to pages

Okay, we've successfully added a few languages to our website and updated our template to handle the new languages, but you've probably noticed that nothing has really changed. In fact, if you go to one of our pages in the frontend and add `&L=1` or `&L=2` to the URL, the only change you can see is the date format. Everything is still in English, and it's not even hiding the non-translated content. That's okay, because it just means that we have not added translations to our individual pages, yet. The TypoScript we just added will hide non-translated content elements on a translated page, but the default TYPO3 response to a non-translated page is to show the default language. We can change that in a moment if we want to hide the default language for non-translated pages. For now, let's just translate our main page.

We add a language to pages in TYPO3 through an Alternative Page Language record, and it's easier than it sounds. By adding an Alternative Page Language record to any page in TYPO3, we are saying that we want to start showing an alternative version of our page to visitors using the Website Language record and we need to start seeing translation buttons in the backend **Page** and **List** views. This is set for each page as part of the developer's design, and it makes sense once you're used to it. In the real world of international websites, it is common to translate only parts of a site. Even the largest companies will often tailor which pages need to be seen in certain regions. This happens because some pages are not applicable to all areas, local working groups may manage some sections, and sometimes it's just not feasible to translate something like a company blog into multiple languages. TYPO3 gives us control over which pages and which content elements are available in each locale that we are targeting.

We can create an Alternative Page Language record for any page through the **Page** view by clicking on the **Localization view** tab that is now available. Just choose a language from the drop-down menu, and we're ready to create our record:



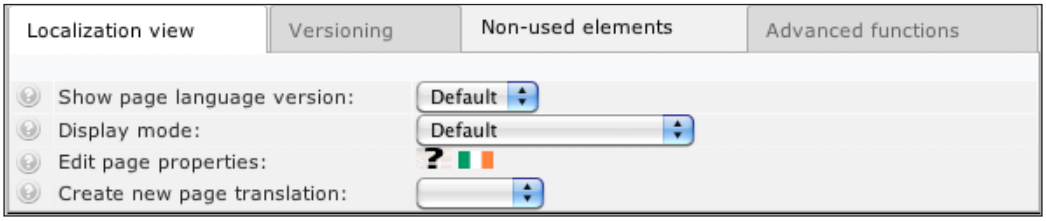
Once we choose a language from the drop down, TYPO3 will create an Alternative Page Language record that we can also see in the **List** view. This can be helpful for debugging problems with translations if we think that the Alternative Page Language record has been accidentally deleted.

By choosing a new language from the drop down, we can start creating a new record and go to an editing screen like the one shown in the following screenshot. In that screen, we can set a translated title for our page. As TYPO3 uses our page titles for menu items, this will automatically translate our menu as well. We can also use this opportunity to translate keywords, description, or anything else that is stored in the page properties. We can even modify the access rules or available resources if we want. You can see an example of the Alternative Page Language record page when it is filled out below:

Once we are happy with our changes, we can save our Alternative Page Language. If we reload the same page in the frontend with the URL variable `L=1`, then we will see that the content elements are now missing. We have set our website to hide the non-translated content elements, so we will need to translate the content elements before they become visible again.

Using the localization tab in the Page view

Now that we have added an alternative language to our current page, the **Localization view** tab in the Page module becomes a more important part for our workflow:



The drop downs can be used to change the way we view the page for editing so that we can concentrate on editing the default language or translating content. Here is a list of the options that are available and how they affect our Page view:

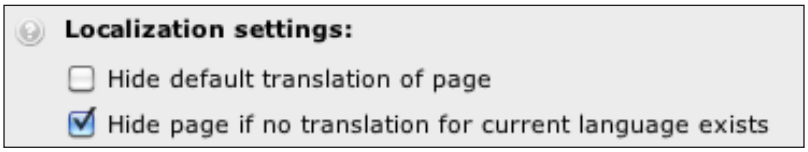
- **Show page language version:** Displays a list of the current languages available to work with.
- **Display mode:** This can be set to `Default`, `Selected language`, or `Only Localized Elements`. If it is set to `Selected language`, any content elements that have been translated into the selected language are visible along with the default language versions. If it is set to `Only localized elements`, then the current language version of all translated content elements will be shown without the original default language versions. When we start translating content elements in a moment, this will make more sense.
- **Edit page properties:** This shows flags for the alternative languages, and we can use these to access page properties such as titles and metadata for translated pages.
- **Create new page translation:** We saw this just a minute ago as a quick way of creating a new Alternative Page Language record.

Hiding non-translated pages

You probably noticed that all of our pages are still available in the menu no matter what language we are in, and most of them aren't translated. TYPO3 allows hiding certain pages if they are not translated so that they will be accessible or appear in the menu, or we can change our configuration to hide all non-translated pages by default.

We can set individual pages to stay hidden if they don't have an Alternative Page Language record for the current language through the page properties. In the **Options** tab, we can set our localization settings to hide the non-translated page by checking the option labeled **Hide page if no translation for current language exists**.

If we only want our page to be available in a non-default language (if we only want the page in Irish, not English) we can hide the default translation of a page by checking the option labeled **Hide default translation of page**:



If we want to change the default setting for our entire TYPO3 installation, we can change that as well. As this affects the entire TYPO3 installation, instead of just the page tree, we have to make our change in the TYPO3 configuration file instead of the TypoScript template. To hide all non-translated pages (pages without an Alternative Page Language record) by default, we can add the following line to our `typo3conf/localconf.php` file:

```
$TYPO3_CONF_VARS['FE']['hidePagesIfNotTranslatedByDefault'] = '1';
```

This only changes the default setting to hide pages. We can still choose to show pages without a translation through the page properties. If TYPO3 is set to hide non-translated pages by default, the option in page properties changes to read **Show page even if no translation exists**. You can still choose to show pages that may not need a native translation like image galleries or links to online stores that have their own translation.

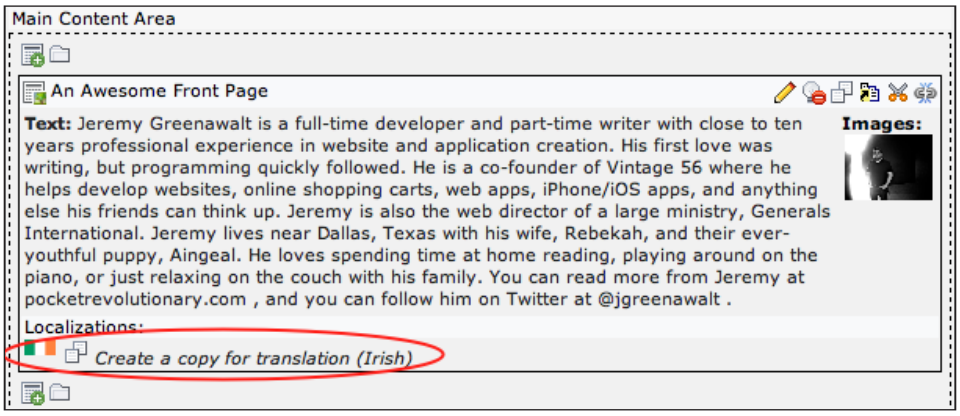
Translating content

We've created an Alternative Page Language record on our target page, so now we can start translating the content elements. Translating a content element is almost the same as editing a content element in the default language, so this is pretty easy to learn.

1. We need to open the page we are translating in the **Page** view. You'll notice that we have localization flags added to our built-in content elements for translating.

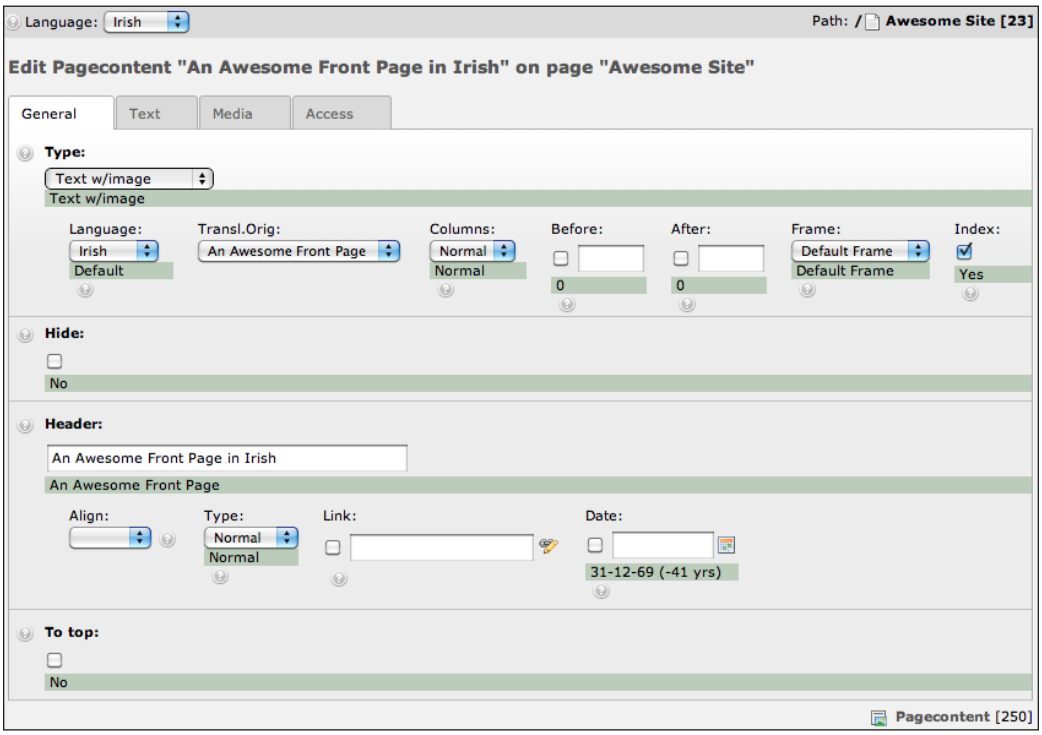
Unfortunately, some content elements, like FCEs, do not use localization flags. The easiest way to translate those elements is to copy the content element and change the **Language** property for the translated element in the content element editing module, which we will see on the next page.

2. Click on the link labeled **Create a copy for translation (Irish)**. It should look like the following screenshot, but if it doesn't we just need to check to make sure that the Alternative Page Language was created correctly.



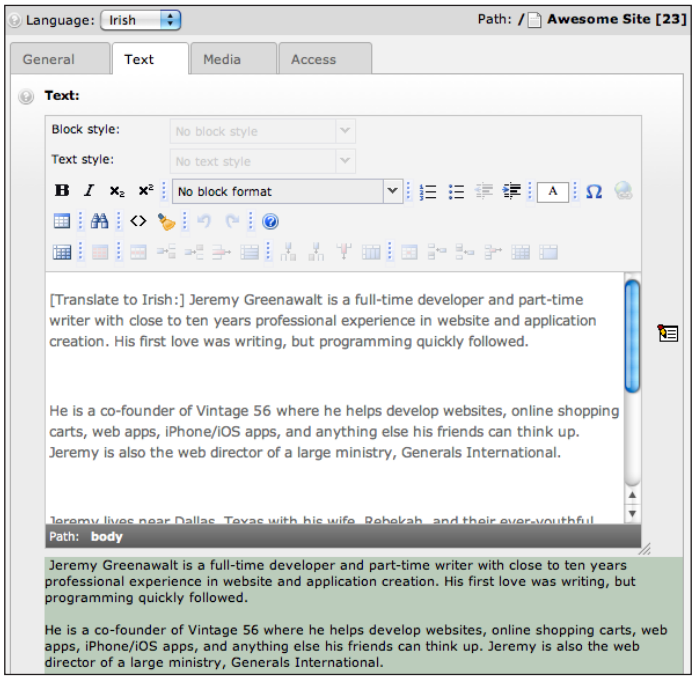
3. After we have clicked on the link to create a copy, TYPO3 will return us to the Page view. We need to click on the flag icon next to the new copy of our content element to start editing the translation.

4. Our editing page looks similar to what we're used to with just a few additions. You can see in the following screenshot that we have a few new drop downs available to choose the current language. If we change the language here, it will simply move this copy of the content element over to that language. The drop down labeled **Transl.Orig** is also helpful for showing us what element we are translating. The most helpful information we can see on the screen is probably the text highlighted under each value. These are the values from the default language, and they will only change when the default language version of the content element is updated. This is essential when we are doing a lot of translation.



5. We've seen our way around the new editing page, so we can go ahead and update the **Header** for the new Irish version of the content element with a new title, `An Awesome Front Page in Irish`.

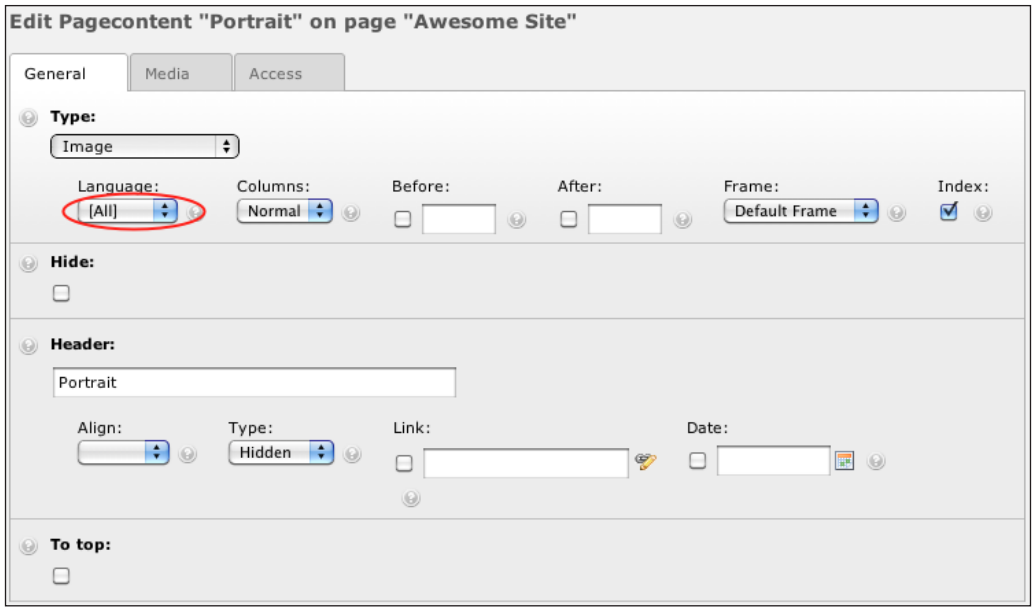
6. In the **Text** tab, we can translate the main text of our content area. Notice again that the entire default language version of the text is highlighted directly below the RTE. We can see that TYPO3 has automatically added the text `[Translate to Irish:]` to the beginning of our text area as a reminder when we are working on the site. For now, we can just leave the text as it is with the translation marker left at the beginning to let us know when we're showing the Irish version of our content.



7. Finally, we can change the image for this content element if we need to. As we have content fallback turned on, we can leave it as it is. If we don't change anything, then it will automatically get the image from the default language.

Creating universal elements

What if we have a content element that doesn't need translation but that we want to show in every language? The most common reason to do this is for non-text images as there is no need to create multiple translated versions of portraits or product pictures. For any content element, though, we can set the **Language** field to `[All]` (circled in the following screenshot). This sets the content element to appear localized for all alternative languages, and it will show a multi-flag icon next to it in the localized **Page** view.



Adding content without a default language

Creating a content element that shows in all languages is pretty straightforward, but sometimes we may need to create content elements that only exist for a non-default language. We may need to add some legal language, or we could just be advertising a special that only applies to our group in Ireland. How do we create a content element without a default language version?

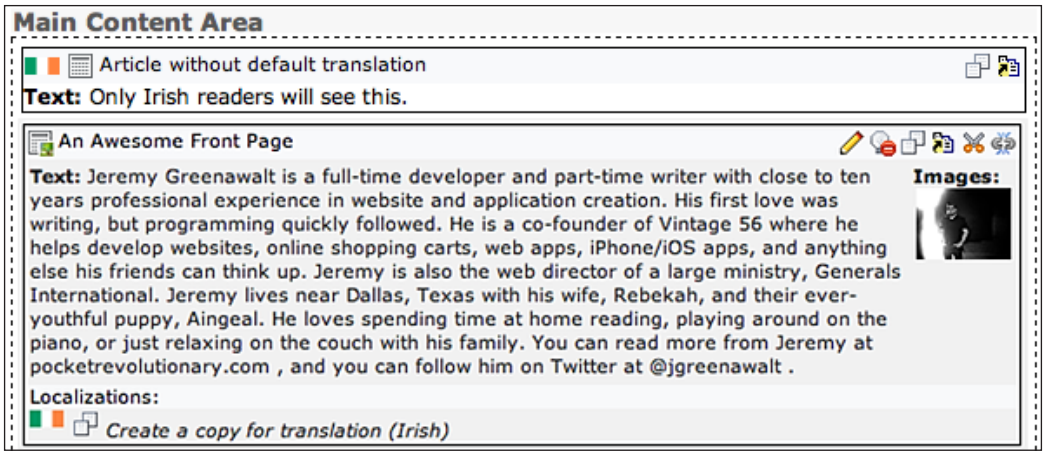
1. Create the new content element in the backend **Page** view just like any other element that we have created until now.
2. Change the **Language** drop down to our target language, `Irish`.
3. Go ahead and fill out the rest of the context fields in Irish.
4. Test it in the frontend. As we have set up the default language to hide non-translated elements, our new element will simply not appear in anything but the Irish frontend.

These non-default content elements normally work great, but we do have to be careful when we create elements that don't exist in the default language. As editors will probably be doing the layout and design of any page in the default language, we need to use extra care in testing to make sure that our new element still works correctly in the target language after any major layout changes to the page. This sounds obvious, but it can be an easy mistake if we are dealing with multiple columns and complex layouts in the default language.

TemplaVoila translator workflow

Once we start dealing with translating all of our content, we probably need to start thinking about how we can set this up for translators. We may be able to translate some of the content, but we know that we'll need to hand it over to somebody else eventually. Of course, TYPO3 can be complex, and we don't need full-fledged editors for each translator position. In fact, we might not want to give them access to change the layout or change page properties for the default language at any time. Luckily, the TYPO3 developers already considered this, and we can restrict backend users or groups to specific languages. Now that we have added alternative languages to our site, we can see a new section in the User Admin module labeled **Limit to languages** (shown in the following screenshot).

If we turn off the default language, then the **Page** view will be specifically tailored to translating. We can see in the translator's screen in the following screenshot that there are no buttons to create new content elements. They are given only flags to translate the content elements that already exist. This makes sure that translators are not creating new content on our main site, but it also makes it easier for them to find what needs to be translated and do their job quickly. They don't need to become TYPO3 editors or worry about breaking the main site when they are translating.



Adding a basic language menu

Now that we have some pages translated we need to make it easier to access. The only way to see the localized pages right now is by adding `&L=1` or `&L=2` to the address of any page, and that's not going to work for our visitors. We need to create a basic language menu for them. Adding the language menu is like adding a link to a printable version of our pages. We just need to provide a link to the localized version of the page instead. We're going to go through the same steps of adding a spot to the template in HTML and TemplaVoila, and then we can assign the languages to it through TypoScript.

1. Our first step is to add a container for the new menu to our HTML templates. Go ahead and open `template.html` and `template_sidebar.html` in your editor of choice and add a new `div` tag above the closing `body` tag:

```
    <div id="print_link"></div>
    <ul id="language-menu"></ul>
</body>
```

2. We need to add the menu to the main TemplaVoila templates. We've done this a few times by now, so we'll go through it quickly this time. To remember how to add a field to the data structure, you can look at *Chapter 3*. In the TemplaVoila module, click on **Update mapping** under the **Main Template** heading.
3. Click on **Modify DS / TO** to update the data structure.

4. We can add a field to the bottom of the data structure with the following values:
    - **Field**: `field_langmenu`
    - **Element**
    - **Title**: `Language Menu`
    - **Sample Data**: `[Language menu goes here]`
    - **Element Preset**: `TypoScript Object Path`
    - **Object path**: `lib.langMenu`
5. After we have created the new field, we need to make sure that we have saved our changes and the new field can be mapped.
6. Map the new field, `field_langmenu`, to the `ul` container that we added to the HTML files with the ID of `language-menu`. Make sure to save the new mapping to all of our main template files so that the menu doesn't suddenly disappear when our users are browsing the new site.

Adding the language menu to our TypoScript template

Finally, we can update our main TypoScript template setup with values for our new language menu. We are going to use the same `HMENU` class that we used for the main menu and submenu, but this time we are going to use the special property like we used to add breadcrumb navigation in *Chapter 4*. We will use a `TMENU` to display our new menu, but we will set the special property to language in order to take advantage of some functions specific to languages.

1. First, we will create a menu object and assign the special property:

```
lib.langMenu = HMENU
lib.langMenu {
```

2. Next, we can assign the `special` property. The values that we assign to the special property will correlate with the language parameter values. The first link in our menu will be `0` (default), the second link will be `1` (Irish), and the third link will be `2` (German).

```
special = language
special.value = 0,1,2
```

3. We need to create the TMENU object and add formatting to the menu items:

```
1 = TMENU
1 {
   NO = 1
   NO.allWrap = <li style="display: inline; margin-right:
10px;">|</li>
```

4. Now, we can create a menu title for each language. If we let TYPO3 automatically detect our titles, it will use the translated page title of the current page. Instead, we want to use the names of our languages:

```
NO.stdWrap.setCurrent = English || Irish || German
NO.stdWrap.current = 1
```

5. We want the current language to stand out, so we will wrap it in bold tags. We can also deactivate the link for the current language so that visitors do not unnecessarily try to click on it and reload the page.

```
ACT < .NO
ACT.linkWrap = <b>|</b>
ACT.doNotLinkIt = 1
```

6. Finally, we can deactivate links to languages that are not available on the current page. This way, we don't link to German if this particular page does not have an Alternate Page Language for German. We are going to use the USERDEF1 item state that is defined in the TSref (http://typo3.org/documentation/document-library/references/doc_core_tsref/). Like the NO or ACT states, this is a Boolean value that tells us something about the current state of the page. The USERDEF1 is normally reserved for user-defined states. As we assigned the special property to language, it will automatically assign it for all menu items that link to a language that is not available on the current page:

```
      USERDEF1 < .NO
      USERDEF1.doNotLinkIt = 1
   }
}
```

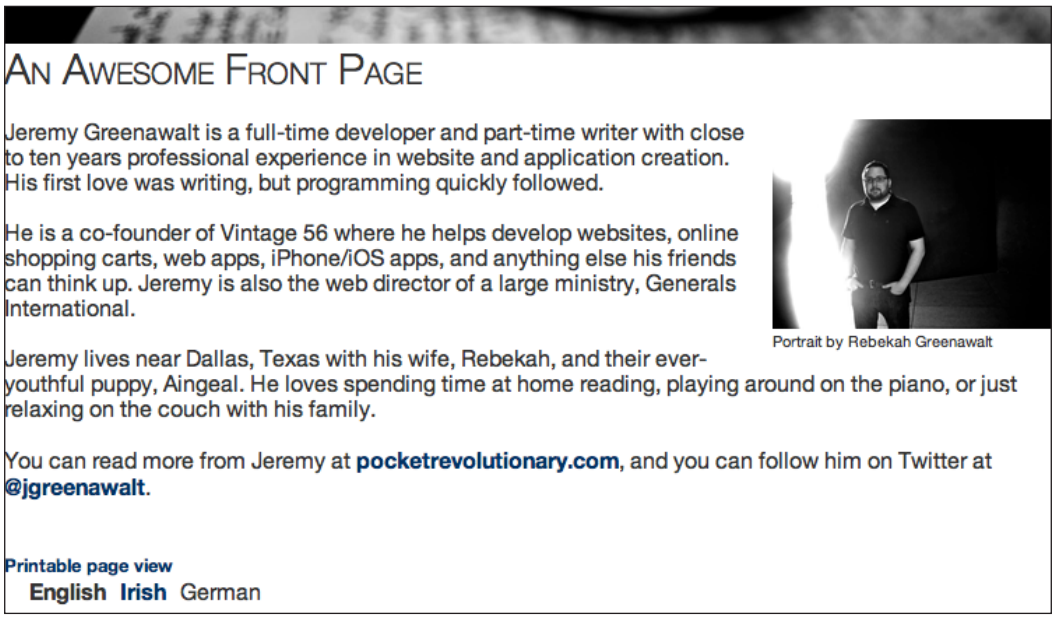
Without the comments, this is our final code for the TypoScript template setup:

```
lib.langMenu = HMENU
lib.langMenu {
  special = language
  special.value = 0,1,2
  1 = TMENU
  1 {
    NO = 1
```

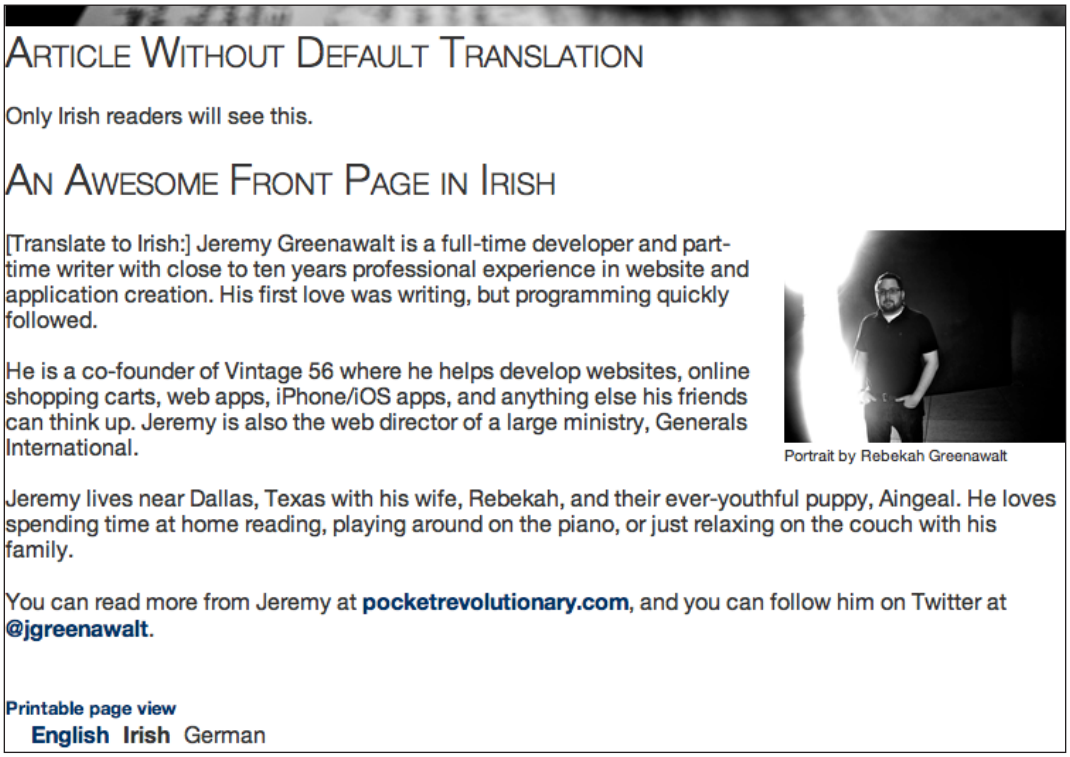
```
    NO.allWrap = <li style="display: inline; margin-right:
10px;">|</li>
    NO.stdWrap.setCurrent = English || Irish || German
    NO.stdWrap.current = 1
    ACT < .NO
    ACT.linkWrap = <b>|</b>
    ACT.doNotLinkIt = 1
    USERDEF1 < .NO
    USERDEF1.doNotLinkIt = 1
  }
}
```

Viewing our changes on the frontend

After we've created our new menu, we can go back to our front page and see the differences between the languages. I've included two screenshots for reference, but we can see the same results on our example site right now in front of us. The first screenshot shows the default language, and everything should look the same as before except for the new language menu at the bottom. We can notice that the menu item for English is bold with no link because it's the current language. The Irish link is active because we have created an Irish translation of this page. The German link is not active because we do not have a German translation of this page.



Now we can click on the Irish link to see our translated page. We can see that our main article is showing the translated version, and our menu has updated to reactivate the link English. We can also notice that an article we added only for the Irish language shows up for the first time; it was not rendered at all when we saw the page in its default language.



Adding flags for language selection

We can do one better than having a text menu for the languages, because we can create nice flag buttons. We can even make it so that flags for languages that are not available will appear obviously inactive.

1. Of course, the first thing we need to do is get some flags for our buttons. If we're lucky, we can go down the hall and ask the graphic designer nicely to throw something together for the new site we're building. If we don't have access to a designer and we can't make them ourselves, we can just search Google for `"free flag icons"`; we're sure to find some that are at least usable for testing. We only need three flags right now.

2. We want to make our new flag icons easy to identify in our TypoScript, so we will name them `usa.gif`, `ireland.gif`, and `germany.gif` and place them in the template images folder, `fileadmin/templates/images/`.
3. Next, we can update our TypoScript template to replace the `TMENU` object with a `GMENU` object. We'll recognize the `GMENU` properties from *Chapter 4*. First, we'll replace the `TMENU` object with a `GMENU` object, wrap our menu items in list tags, and set the title parameter of each link to the page title in the correct language:

```
1 = GMENU
1 {
   NO = 1
   NO {
      allWrap = <li style="display: inline; margin-right:
10px;">|</li>
      ATagTitle.field = title
```

4. Next, we can set the dimensions for our menu items and assign our flag icons to the menu:

```
   XY = [5.w]+10, [5.h]
   5 = IMAGE
   5.file = fileadmin/templates/images/usa.gif || fileadmin/
templates/images/ireland.gif || fileadmin/templates/images/
germany.gif
   5.offset = 0,0
}
```

5. Again, we will deactivate links for the current language:

```
ACT < .NO
ACT.noLink = 1
```

6. Finally, we will remove the links from the unavailable languages and make their flag icons grayscale so they will appear inactive:

```
   USERDEF1 < .NO
   USERDEF1 {
      noLink = 1
      20 = EFFECT
      20.value = gray
   }
}
```

If all languages are available on a page, we can see the active version of all flags like the menu screenshot as shown in the following screenshot:

If only German and English are available, we can see that the Irish flag is monochrome:

You'll notice that we removed the link from the current language flag, but we did not make it appear inactive. We could change the TypoScript to show the inactive graphics for all buttons that do not link, but we should be careful before we gray out too many buttons. If there is only a default language on any page, all of the buttons will be gray and it will look like our language menu is just broken.

Adding a localized logo

While we are localizing our website, we should go ahead and make sure that our logo is translated for all of the languages we are targeting. We thought ahead enough to make our logo a TypoScript object when we added it to our website, so all we need to do is change the values for the object within the realm of each language. Our original logo image is located in the `fileadmin/templates` directory, so we can make our own localized versions named `logo_ie.png` and `logo_de.png` in the same directory. As we want to keep our visitors in the same language when they click on the logo, we can also update the link to connect to our home page with the correct language variable in the URL. To set our changes for the languages, we just need to set the new values between the `globalVar` conditional statements like shown in the following code (new lines are highlighted):

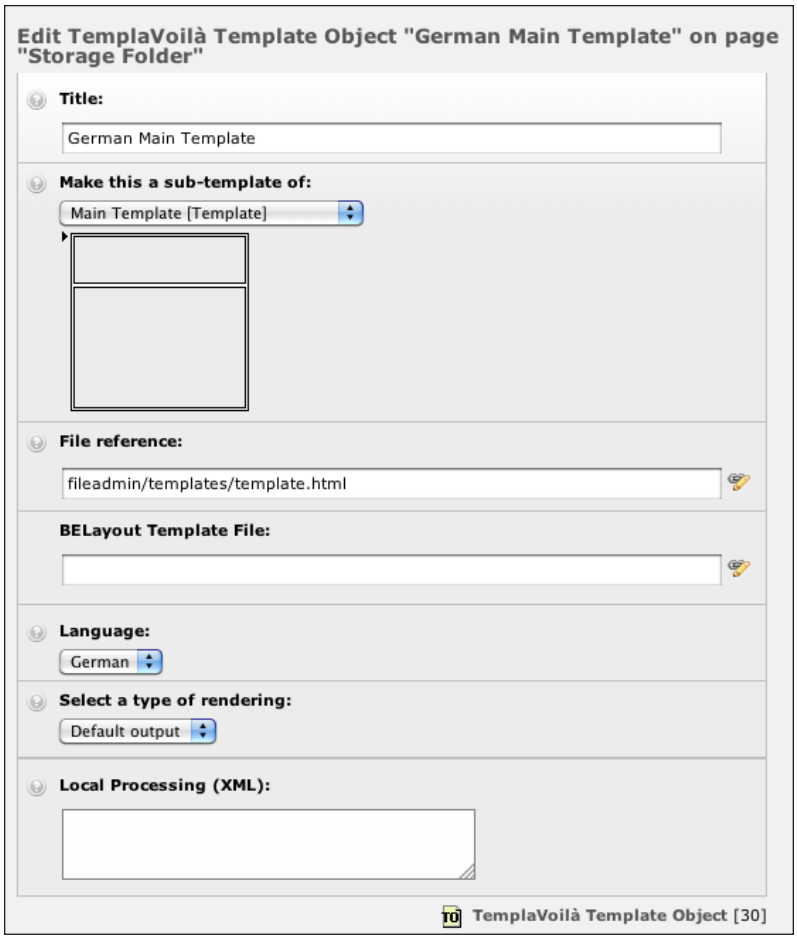
```
#Setting up Irish language:
[globalVar = GP:L=1]
config {
   sys_language_uid = 1
   language = ie
}
lib.timestamp.10.strftime = %d-%m-%Y %T
lib.logo.file = fileadmin/templates/logo_ie.png
lib.logo.stdWrap.wrap = <a href="http://www.example.com/index.
php?L=1">|</a>
```

```
#Setting up German language:
[globalVar = GP:L=2]
config {
   sys_language_uid = 2
   language = de
}
lib.timestamp.10.strftime = %d.%m.%Y %T
lib.logo.file = fileadmin/templates/logo_de.png
lib.logo.stdWrap.wrap = <a href="http://www.example.com/index.
php?L=2">|</a>

[global]
```

Creating localized TemplaVoila templates

Along with tailoring our content and TypoScript for a specific language, we can also create separate TemplaVoila templates for each language. That means we can use a different HTML file, change our mapping, or even map fields that aren't mapped in the default language's template. Instead of having to set up our own logic to switch the template for each language, it is handled by TemplaVoila completely in the background using the same subtemplate logic we used for a printable template. This gives us more flexibility with our localization, and we can add it as easy as creating any other template object:

1. The first step is obviously to create a new template object in the TemplaVoila module. Don't worry about the language settings yet; just create a blank template object named `German Main Template`, and set the file reference to `fileadmine/templates/template.html`.
2. Next, we can assign the new template object as a subtemplate of `Main Template [Template]`. This automatically links the new object so it can inherit the data structure values from the main template like when we created our printable template in *Chapter 5*.
3. We want to use this for the German localization of the website, so choose German in the drop down labeled **Language**. As we declared this as a subtemplate of the main template, any page using the main template will display this template when the current language is German. Our editing screen should look like the following screenshot before we save:



4. Finally, we can update the mapping for the template to make our new template different from the main template. If you want to see that the template is automatically called correctly, leave the banner image unmapped as a test.

After we save our new mapping settings, we can view the front page in German. Notice that the banner image automatically disappears when we switch to German without us ever having to update the TypoScript or content of the page. This is just a simple example, but we can easily find uses for this automatic template replacement once we start launching sites in the real world. For example, if we already know that we have less content translated to show in German than English, we can change all two column layouts to use a single column when the active language is German. There's no reason our site needs to look any worse just because we are trying to target a new area.

Summary

Congratulations! Our site is now capable of going international, and we can definitely pat ourselves on the back. The ideas of internationalization and localization sound pretty daunting at first, but TYPO3 has really done most of the work for us. We just had to learn how to tweak the templates a little to give us the most flexibility. With a few changes, though, we figured out that we could go all the way from a simple translated site that looks basically the same in all languages to a fully-configured website that can change it's design and functionality for each locale that we're targeting. We even learned how to customize our logos and templates for each locale. It seems easy now, but our mobile-friendly, internationalized website just jumped beyond most of our competition in only a few chapters.

In the final chapter, we are going to look at where TYPO3 template development is going with a new TemplaVoila framework. As we've seen TemplaVoila and TypoScript can work together to make a very powerful template system. There are places that it can be made easier to use or faster to customize, and that is why we're going to be looking at the TemplaVoila Framework in our final chapter. It's not a replacement for anything we've learned so far, but it is both a final set of enhancements to what we've been working on and a glimpse of where TYPO3 is going in the future. So, you know the routine by now: show off our new international website to one of your co-workers, grab some coffee, and come back ready to learn just how powerful a little framework can really be.

11
Building Websites with the TemplaVoila Framework

We've officially covered everything you must know to modify and create templates from beginning to end, and we've even learned how to handle special cases like mobile websites and internationalization. Now we are going to take a step back and look at the overall workflow for bringing everything together and building websites. In the real world, we will need to go from designer comps and mockups to launched websites. As we've already covered mapping, content elements, and everything else we need to know to create a great website from mockups with a traditional workflow, we're going to look at how we can reverse our thinking a little and build our TYPO3 sites more effectively. We're going to look at the new TemplaVoila Framework that has just recently been introduced by Ron Hall of Busy Noggin, Inc. and Web-Empowered Church (WEC) that can help us build sites faster and with less unnecessary effort.

This chapter gives a good overview of this new framework and shows some real-world examples, but it doesn't cover everything and we won't go through building a whole new example site from scratch. Instead, we are going to look at how the framework functions and how you can start your next project with it. For more information, downloads, and the latest news, go to `http://templavoilaframework.sitebuilderlab.com`.

In this chapter, you will:

- Learn about the TemplaVoila Framework workflow and approach to template building
- Learn how frontend designs are built in the framework using "skins"
- Create a new skin and learn how to modify it

- Learn how to use the page templates included in the TemplaVoila Framework
- Learn how to use the content elements included in the framework to make multiple layouts

What is the TemplaVoila Framework?

The TemplaVoila Framework is a lean framework with a common code base, a prescribed workflow, and a set of best practices for template development. Like other software frameworks such as Ruby on Rails or FLOW3, the TemplaVoila Framework provides a basic structure and gives us some common tools (such as multi-column flexible content elements) to help us build sites faster. At the same time, the framework helps us define a repeatable workflow by changing the way we think about building templates and helps us concentrate on the unique tasks we want to accomplish instead of making the same templates over and over for different sites.

If this framework is meant to reverse our thinking, why is it at the end of the book? The TemplaVoila Framework was designed to be used and understood best by those of us who already have experience developing websites with the core technologies of TemplaVoila, TypoScript, and frontend coding. Just like FLOW3 doesn't replace a developer's knowledge of PHP, the TemplaVoila Framework doesn't replace our knowledge of TemplaVoila and TypoScript. For one thing, we can only get so far even in a framework if we don't know how to build menus or other basic operations. TYPO3 knowledge is still needed to build good templates. Secondly, one of the reasons that the TemplaVoila Framework is powerful is because we have all the power of TemplaVoila and TypoScript to extend and customize it with our own templates and FCEs.

Benefits of the TemplaVoila Framework

So, if the TemplaVoila Framework doesn't replace TemplaVoila and TypoScript, why do we need to learn it?

- We can build sites faster. With a standardized workflow and set of tools, we can start working quicker and avoid miscommunication. Using the built-in page templates and FCEs, we can also skip the initial process of creating new HTML files from hand, mapping them, and creating common flexible content elements.

- The TemplaVoila Framework includes common FCEs and page templates. Multiple column layouts, HTML wrappers, and other flexible content elements and page templates are included in the core of the framework so we can spend our time building custom FCEs or templates after we've exhausted the built-in options.
- The TemplaVoila Framework uses "skins" for the look and feel of our websites. As simple collections of CSS, HTML, and TypoScript files, the framework skins (which are not related to TYPO3 backend skins) can be moved around easily as packages or directories. We can build a skin completely on a development server or our own computers and move it up to the production site only when we're done. We can even develop the skin on our own computer while others on our team are creating content on a staging server with a basic wireframe skin. When the skin is ready, we can just upload our skin to the staging server as a single directory. Additionally, since skins are just collections of files in a directory, so we can use version control software like Git or Subversion to store them safely and work with multi-developer teams.
- The TemplaVoila Framework skins are easy to reuse. Skins can be downloaded from developers like Web-Empowered Church (WEC) through the TYPO3 Extension Repository, or we can reuse skins that we have built before. We can build our own wireframe skins to use at the beginning of each site. We can use a skin from one TYPO3 site as a starting point for a different website with a similar design.
- Less remapping. All of the templates in the TemplaVoila Framework are already mapped, so we only need to map specialized templates that we build outside of the framework. In addition, the TemplaVoila Framework maps blocks instead of individual elements. Instead of mapping the title, banner, search area, and menu at the top of the page, it just maps the header as an entire block. This means less remapping when the search area changes and no remapping when a different skin is applied with a new element in the header.
- The TemplaVoila Framework's default backend is easier for editors. Instead of going through and making our own modifications to the backend layout files, we can rely on the default, tab-driven design of the TemplaVoila Framework. This layout is consistent between framework-driven sites, so it makes for easier training and documentation when we work with clients. Of course, we can still make our own backend layouts if we want to, but it is not as necessary.

- Frameworks, cleaner coding, and rapid development are all themes that TYPO3 is moving towards with other projects like FLOW3 and TYPO3 v.5 as it develops for the future. These concepts are the future of TYPO3 development, and using the TemplaVoila Framework gives us a head start.

The TemplaVoila Framework workflow

The traditional TemplaVoila workflow is to build static template files in HTML, CSS, and JavaScript, test them, and then create and map content elements in TemplaVoila. In the TemplaVoila Framework, however, the built-in templates are already mapped to working content elements. Compared to a traditional workflow, we work backward to update the look and feel of our site by using TypoScript to adjust the HTML layout and writing CSS for styling. That means that we can use one set of templates, and create completely different looks and structures just by changing the skin.

This workflow also means that different people can be working on different parts of the site creation process at the same time. After we install the framework, we can start using the built-in skins to design how the page tree and our content will be laid out (which is called the information architecture) using a "wireframe" skin that only shows the basic layout of our pages without design elements. As it's a live site, we can even click through the links and see how our IA (information architecture) design will actually function. Once a basic IA has been agreed upon, the designers can start working on the visual design mockups. Instead of waiting for the designers to finish completely, the client or editors can start adding content to the basic site and creating pages while we work on special functionality such as internationalization or custom FCEs. After the visual design mockups are done, we can just integrate the visual design into a content-filled, functional site by creating a template skin.

Installing the TemplaVoila Framework

The TemplaVoila Framework is an extension, but it can be used to create a new site in two different ways. It can be installed as an extension to a TYPO3 dummy package installation, or we can install a bundle called the QuickSite package that includes TYPO3, the TemplaVoila Framework extension, a database schema for the framework, and some helpful extensions.

Unless you need to start with a default TYPO3 dummy package for some reason, I recommend the QuickSite package as the most common and easy way to get a site started with the Framework. Even if you are converting an existing site to a new design and the TemplaVoila Framework, it is often easiest to start with a fresh

QuickSite installation in a staging area and import chunks of data from the live site. It's a good opportunity to reverse our thinking and try to do things the TemplaVoila Framework way instead of just transferring over bad habits from before.

We are not going to create a whole site in this chapter, but you do want to have the TemplaVoila Framework available to see some of the examples. We are going to use the QuickSite package for this chapter, so go ahead and download the QuickSite package and install it right now using the detailed instructions for your platform at `http://templavoilaframework.sitebuilderlab.com`. After you have downloaded and installed QuickSite, the next section will go through the setup process.

Setting up QuickSite for the first time

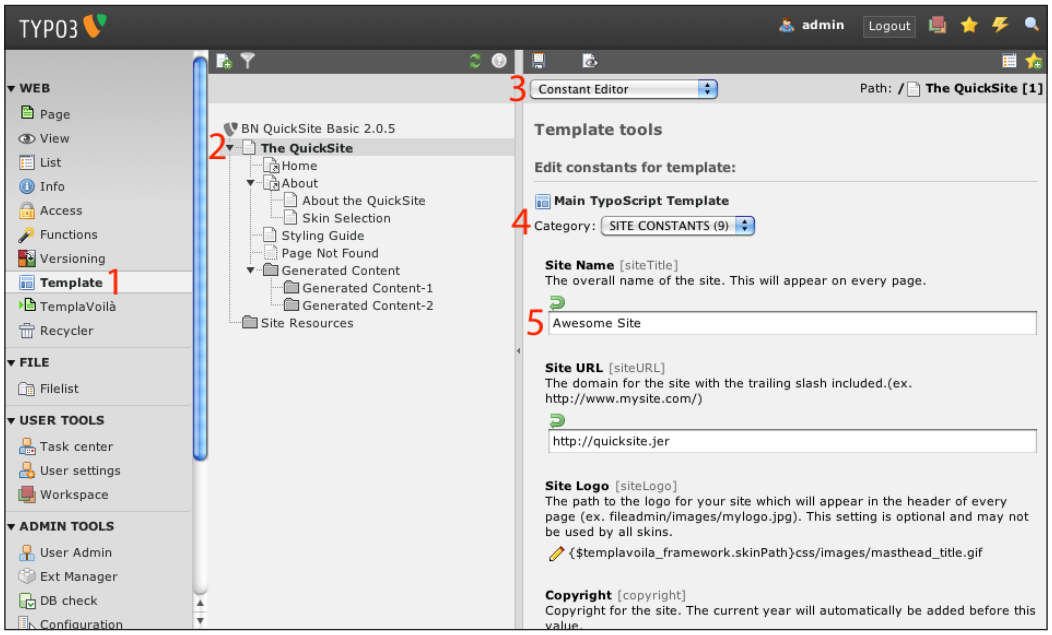
QuickSite starts with the wireframe skin as a default and some example content. We can log in to the backend with the username of `admin` and the password of `password`. After we log in the first time, TYPO3 will ask us to change our password. Go ahead and change it to something more secure right now.

Assigning a site URL

Next, we need to edit the TypoScript constants for the main template to assign a **Site URL**:

1. First, click on the **Template** module on the far left sidebar.
2. Click on the root page for our page tree, which should be labeled **The QuickSite** by default.
3. Next, choose **Constant Editor** from the drop-down at the top of the Template editing pane. TYPO3 installations and the TemplaVoila Framework in particular, use TypoScript constants for configuration options that the TypoScript template needs access to.
4. Select **SITE CONSTANTS** from the **Category** drop down.

5. Finally, fill in the fields labeled **Site Name** and **Site URL** with appropriate values like shown in the following screenshot and click on the save icon.

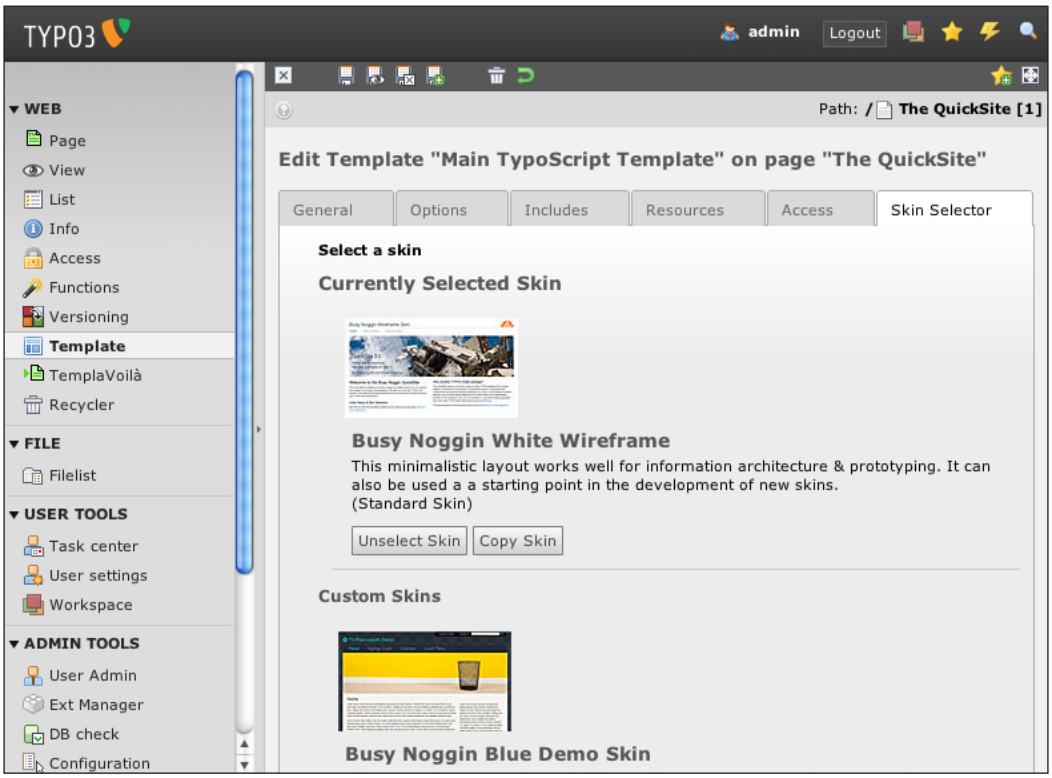


Selecting a skin

Now we can select a new skin. By default, QuickSite uses a plain white skin for wireframing, but we can change it to one of the other built-in options right now:

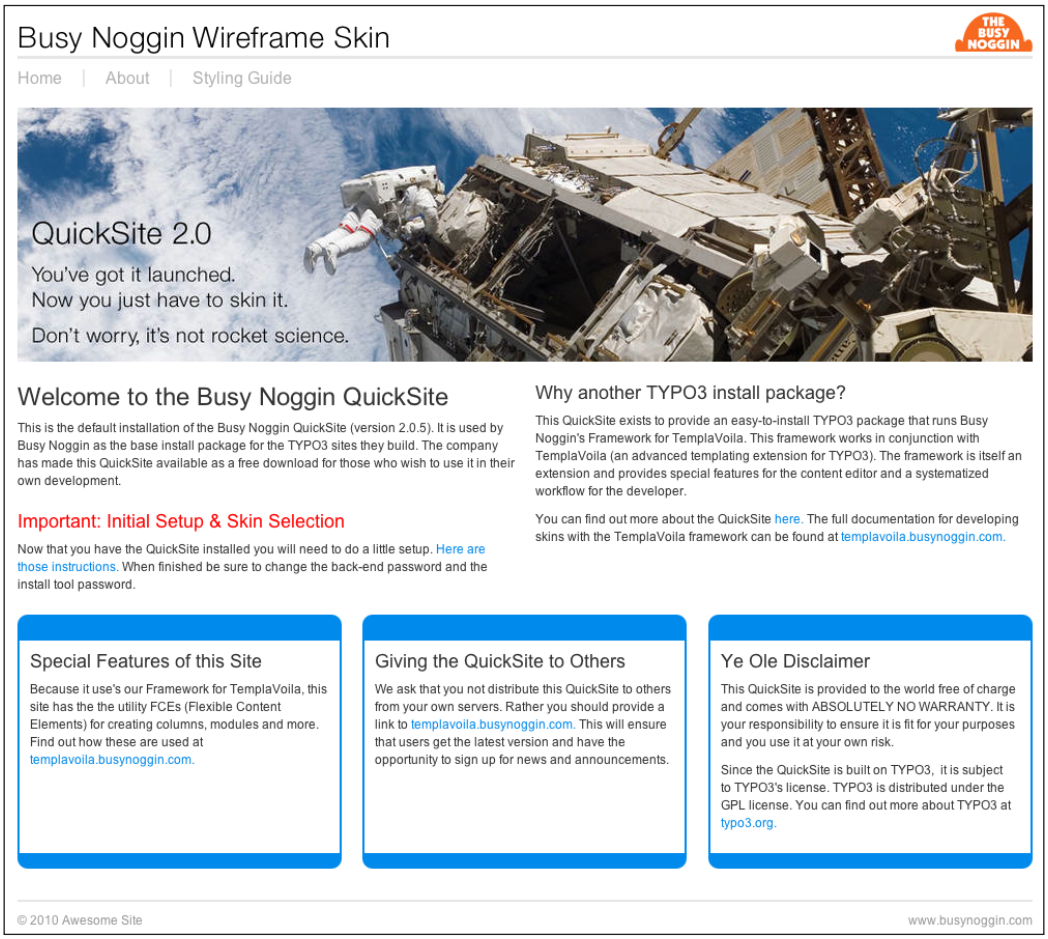
1. Choose **Info/Modify** from the drop down at the top of the editing pane.
2. Click on the link labeled **Edit the whole template record**.
3. Now, we can go to the **Skin Selector** tab to choose a new skin for our template (shown in the following screenshot).

4. The **Busy Noggin White Wireframe** skin is chosen by default, but we can select a new skin by clicking on the **Select Skin** button under any of the available skins.



Viewing our QuickSite frontend

Finally, we can clear the TYPO3 cache to see our changes in the frontend. After setup is complete and the cache is cleared, our front page will look like this:



Planning with the wireframe skin

Now that we've installed QuickSite on a test domain, we can use the built-in Busy Noggin White Wireframe skin to start planning the information architecture (IA) with our team. By wireframing in a live site, we can immediately see how things will work in the final product. We can use the TYPO3 backend to create basic pages and example content and move them around in the page tree. We can even start using the links and menus that we will use in the final version. Finally, we can start

training the clients or editors on the TYPO3 backend they will be using; they can start entering real content instead of just placeholder text and stock photos that have to be replaced at the end.

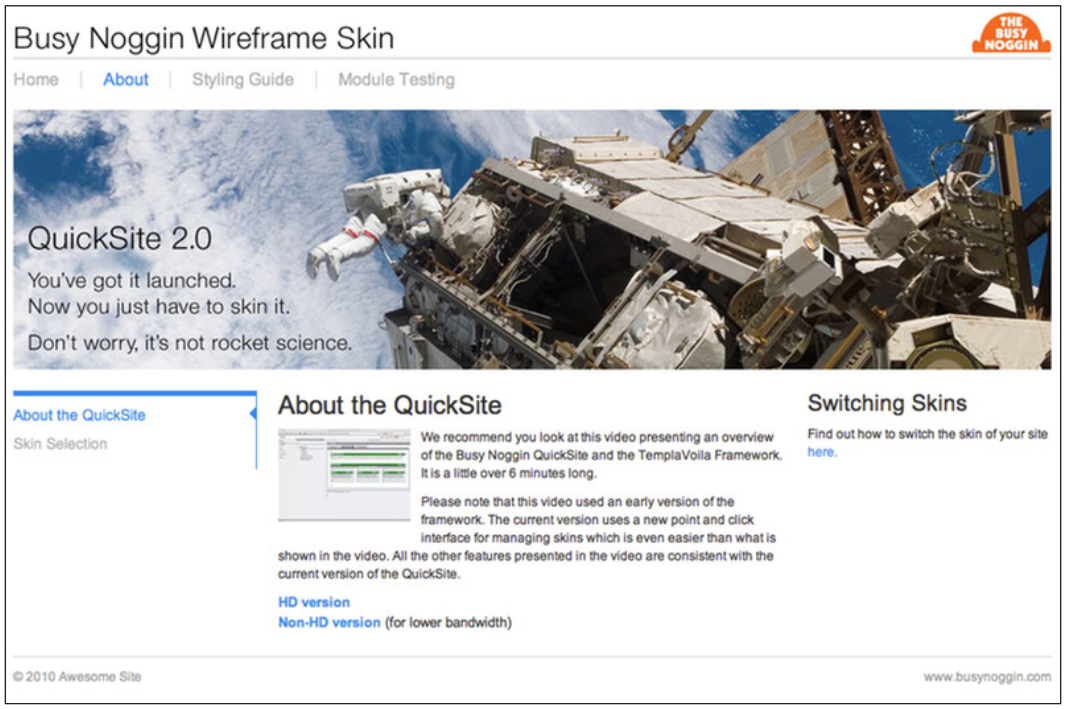
Because we are creating real pages to test the flow of our site, we can even do some basic usability testing to make sure that our IA is well-designed for the target audience. This is a great benefit of using the wireframe skin compared to planning everything on paper or in software; regular users can actually test our information architecture to let us know if content was where they expected it to be. For more information on testing, I recommend Jakob Nielsen's website, `http://useit.com`.

The included Busy Noggin White Wireframe skin is just a basic skin to get us started, but we can easily create our own custom skin for wireframing sites. If we are doing work for an agency or just want to show our own brand internally, we can create a copy of the wireframe skin and modify it easily. We can replace the title on the skin, the logo in the upper-right corner, and any other small tweaks that would be more appropriate for our clients. Then, we can save a copy of that skin for all future products, and we will always have a professional, branded skin available to use for wireframing new sites.

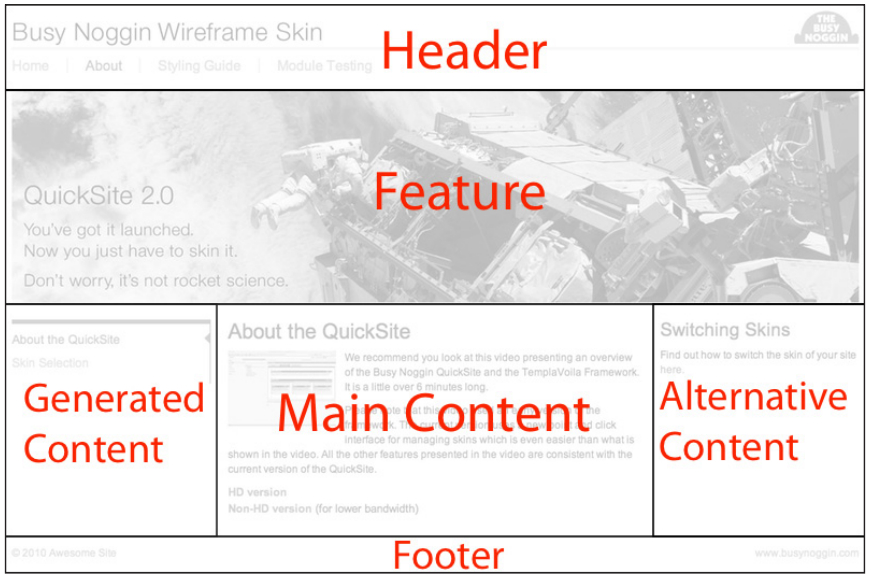
Designing the page layouts

Once we have a page structure and the rough design mockups from the designers, we can start working on the overall HTML structure for our pages. Normally, we will end up having 2-3 different page layouts for any given site, but not too many more. When we look at the design mockup, our goal is to visualize the large blocks in the design and categorize them. Once again, we are not mapping each element like traditional TemplaVoila design, but we are defining the large chunks of content that we need. By defining the large blocks, we can decide which page templates and FCEs we are going to need to make the design work. This also means that we can change the skin later on without altering the HTML structure.

If this is our design mockup:



We want to divide it into blocks like this:



We are going to look at the different templates and FCEs in a moment, but first we need to decide the types of content areas in our structure. All of the blocks in our design can be categorized as one of these types:

- **Header**: This is, of course, at the top of the page and is easily recognizable in almost all designs. Every template in the TemplaVoila Framework includes a header, and this is typically a good place for the logo, main menu, and any universal tools for logging in, searching, or choosing a language.
- **Footer**: This block is at the bottom of every page template in the TemplaVoila Framework, and it is normally reserved for the copyright information and important links like support, FAQs, and contact information.
- **Main content block**: This is the central block of our page where all of the main content for this page is shown. Every page must have a main content block, so it is also included in every built-in (or "core") page template.
- **Feature**: This area is for content that falls right below the header of our page template. It is normally used for banners that are unique to a page. We don't need to use this block on every page, but it is available in every core page template. It will only appear, if it has content.
- **Additional content blocks**: These blocks are optionally used for secondary content on a page. The right and left sidebars that we used in earlier templates would be considered additional content blocks. The core page templates are broken up into series by what additional content blocks they support.
- **Generated content blocks**: These blocks can be used for content that will be automatically generated using TypoScript or plugins and displayed on multiple pages. Local menus, login areas, and other generated content are placed in here. Generated content blocks are only included in some of the core page templates.

Now, using those types of blocks, we can start looking at the HTML structure that we need to fulfill the client's needs and match the designer's mockups.

Page Templates

The TemplaVoila Framework comes with 15 different mapped page templates in the core that are broken into three main series: F1, F2, and F3. All of the page templates are similar in their main structure, but they include different content areas and different body tag IDs for styling them in CSS or calling them in JavaScript. We can see all of the available page templates in the **TemplaVoila module** in the backend, but it helps to see what each series has to offer:

F1 Series:

- Header
- Footer
- Featured Content
- Main Content Area

F2 Series:

- Header
- Footer
- Feature Content
- Main Content Area
- Content Area 2

F3 Series:

- Header
- Footer
- Feature Content
- Main Content Area
- Content Area 2
- Content Area 3

Of course, we can use page templates from any of the series in our own website; for example we can have a page template from the F1 series, the front page and a page template from the F2 series on internal pages. Once we know which page templates we want the editors to use, we can move the unneeded page templates into a `SysFolder` labeled **Unused Templates** so they will not show up in the normal page editing screens.

Remember, these are just the core page templates that are included in the TemplaVoila Framework. We can still create special page templates for newsletters or print later on if we need to.

Utility FCEs

In addition to page templates, the TemplaVoila Framework includes a few built-in FCES, or what it calls **Utility FCEs,** for laying out our content inside content blocks. Using the Utility FCEs, we can add multiple columns, custom HTML tags, and other necessary elements to our page design. If there are any FCEs that we are not going to need for our website, we can move them to the **Unused Templates** folder with our unnecessary page templates to keep them out of the way without permanently deleting them.

There are five Utility FCEs that we will look at:

- Column groups
- Module groups
- HTML wrapper
- Plain image
- Module feature image

Column groups

Column groups are available to display multiple columns inside a page template content block. The TemplaVoila Framework includes column groups of 2, 3, and 4 columns for us to use in our design. Compared to the basic column groups we built ourselves earlier in the book, the column groups in the TemplaVoila Framework have a few key advantages in how they adjust to the content area in which they are placed.

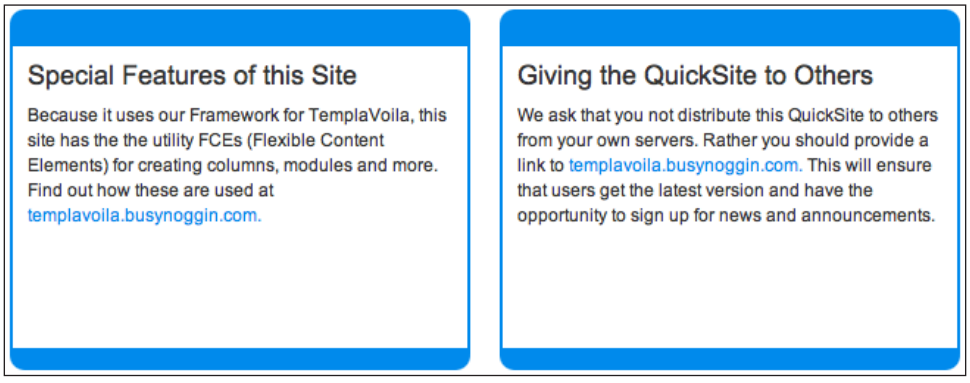
- All column groups are self-adjusting to the content they are placed in, so they correctly fill up the content area using pixel calculations for their maximum width and gutters (the visual space between columns).
- In addition, column groups can be nested one-level deep; we can nest a three-column group inside of a two-column group if we needed to match a particular design. This can be especially handy if we are emulating grid-based newspaper layouts within our designs.

- The gutter width is set in pixels as part of the TypoScript file for the skin for consistency and flexibility and then output to the browser as part of the `style` attribute for each column. By using TypoScript for this, the framework will be able to use calculations to output fixed-width columns that still have the flexibility of being moved around and changed in the backend.
- Column groups have options to define the column distribution (Half / Half, Third / Two Third, and so on) and whether there should be a separating line at the bottom of the column group when they are created on each page. This makes it easy to adjust the look and feel of a column group without remapping the template or making major content changes.

Of course, if we want our own multi-column element with different sizes of columns than the framework has available, we can look back at *Chapter 8* to make our own FCEs as well.

Module groups

Module groups can be used to show distinct "modules" of content on our pages with up to four modules in a row and customizable borders. You can see an example in the following screenshot of a module group from our TemplaVoila Framework installation, but it's important to remember that the look of our modules can be changed a lot through our skin and by which options we choose:



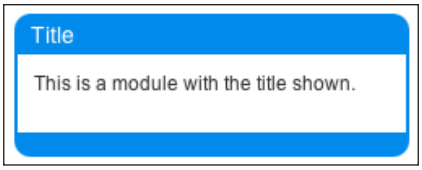
Like column groups, module groups come in groups of 1, 2, 3, or 4 modules and are self-adjusting to fill the content block they are in perfectly. Columns can be nested one-level deep inside of modules, but modules cannot be nested inside each other. With one level of nesting though, we can create a module with two columns of content, for example, easily without worrying about defining widths or redefining our gutters. Module groups also use options to define the module distribution (Half / Half, Third / Two Third, and so on).

Module options

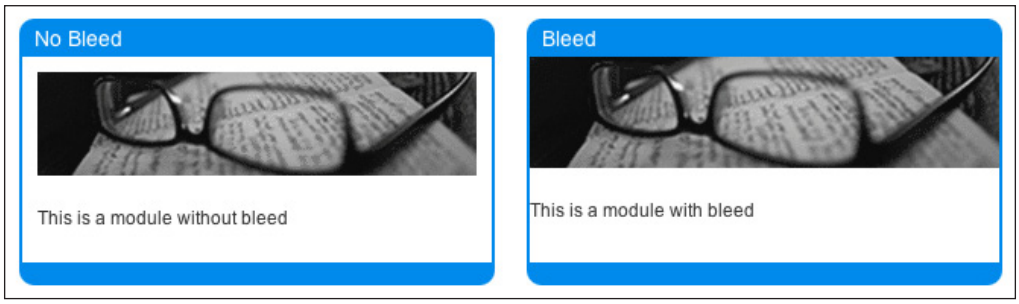
Module groups also include options that can be set per module in the backend editing screens like shown in the following screenshot for each group:

Using these options, we can change the look of each module in a group:

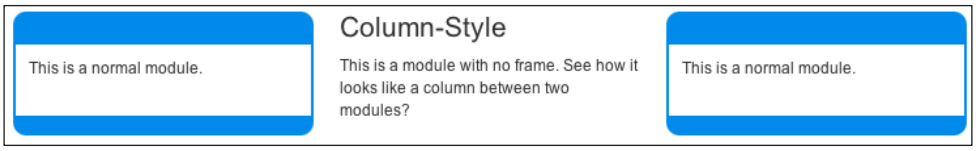
- **Show Title**: We can choose whether or not to show the module title for each module. This is what a module looks like with the title shown:



- **Bleed**: We can set each module to bleed to the edges or keep its margins with the **Bleed** option. In the example below, the module on the left does not have bleed turned on, so we can see the margin of white space around the image and the text. The module on the right has bleed turned on, so there is no margin anymore:



- **Column Style (unframed)**: We can also set individual modules to appear unframed so they look like columns. This is helpful if we want a content block to look like it has a column between two modules like shown in the following screenshot.



You may notice that the modules have a lot of HTML markup. This is done so that they can be styled in as many ways as possible (rounded corners, stylized border treatments, drop shadows, and so on) completely in CSS. In order to allow all of these designs, the HTML markup is written to be able to use the **CSS sliding doors technique** (`http://www.alistapart.com/articles/slidingdoors/`).

HTML wrapper

Like the HTML wrapper FCE that we built in *Chapter 8*, this Utility FCE allows us to wrap a content element or set of content elements in custom HTML tags. Unlike the one we built before, which was hardcoded to be a `div` tag, this FCE allows us to use any HTML we want before and after the content elements inside of it.

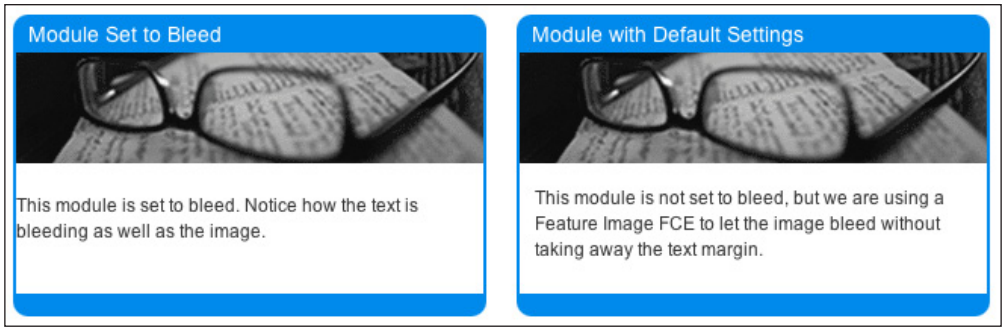
Plain image

Normally, we can display images using TYPO3's built-in content elements Text with Image or Image without any tweaking or special programming. Occasionally, we might need more control over the display of our images. The Plain Image FCE in the TemplaVoila Framework core lets us manually set any of these options:

- **Link**
- **Alternate image text**
- **Image width**
- **Margin-Top**
- **Margin-Bottom**
- **Margin-Right**
- **Margin-Left**
- **Z-Index** (to determine stacking order)
- **Display** (inline, float left, float right, display block)

Module Feature Image

We've seen that we can set the module to bleed when we want edge-to-edge coverage by the image and the text. In many circumstances, we may only want the image to bleed while the text keeps its margins inside the module. In these situations, we can use the Module Feature Image FCE. We can set the module itself not to bleed (so the words have spacing between them and the sides), and then we can add the Module Feature Image FCE into the module with an image. The image defined in the Module Feature Image will automatically stretch edge-to-edge inside the module without affecting the rest of the content. In the example (as shown in the following screenshot), the module on the left has bleed turned on so the text has no space around it; the module on the right has bleed disabled so the text has space around it, but it is using a Module Feature Image FCE at the top for the banner.



Creating a custom skin

The TemplaVoila Framework has two types of skins: standard skins that are packaged with the extension and stored in the database and custom skins in a directory that we can define. By default, custom skins are stored in `fileadmin/templates/skins/`, but this can be changed in the **Extension Manager**. To create a new skin, we can start by copying a standard skin, so we don't have to manually create all of the files and directories we will need:

1. Click on the **Template** view on the left sidebar and choose the main template in the root page of our site.
2. Just like when we chose a skin to use, choose **Info/Modify** from the drop-down at the top of the editing pane and click on the link labeled **Edit the whole template record**.
3. Go to the **Skin Selector** tab where we chose a skin before.
4. Under the skin that we want to copy click on the **Copy Skin** button. For now, let's copy the **Busy Noggin White Wireframe** skin.

5. As soon as we choose to copy the skin, the page will refresh and we can see that there are two identical skins. More importantly, the framework has created a new directory in our custom skin folder using the name of the original skin. In our case, we will have a new directory named `skin_bn_white_wireframe` in the `fileadmin/templates/skin/` directory. To make our new skin unique, we need to rename that directory. Let's go ahead and change the name to `skin_awesome_book`.
6. Next, we need to update the title and description for the skin. Inside the `fileadmin/templates/skin/skin_awesome_book/` directory, we can open a file named `info.txt` in a text editor. The `info.txt` file is a plaintext file with the current title and description. Go ahead and update the `info.txt` file to look like this:

   ```
   Title: Awesome Book Wireframe
   Description: This is our first custom skin. It may not be perfect,
   but it's ours.
   ```

7. After you save the changes to your skin, reload the **Skin Selector** tab and click on the **Select Skin** button next to our new skin labeled **Awesome Book Wireframe**.

Editing a skin

A custom skin in the Busy Noggin TemplaVoila Framework is really just a directory that holds the unique CSS, JavaScript, TypoScript, and other files that we need to set the look and feel of our frontend. That means that editing an entire skin can be done just by editing the files inside of a text editor or, for CSS and JavaScript, possibly a development environment or dedicated CSS editor like CSSEdit. Instead of working through tools and wizards in the back end of TYPO3, we are going to look at the individual files that we can modify in a text editor to completely change the look of our new skin. So, let's go into our skin directory at `fileadmin/templates/skin/skin_awesome_book` to start editing the files.

Editing TypoScript for the HTML structure

The HTML structure of our skin is defined using TypoScript in the `skin_typoscript.ts` file inside of the `typoscript/` directory of our skin folder, and we can customize it inside a normal text editor. You'll notice that both files in the `typoscript/` folder have a unique file extension, `ts`, to identify them as TypoScript files.

Remember that our templates are mapped using large blocks for content, so we use the TypoScript in this file to define the layout and functionality inside the header, footer, and other content blocks. There are four main types of TypoScript objects

inside the file that we can edit to change the structure of our templates: `header`, `footer`, `preCode`, and `postCode` objects (which are used to customize all of the content blocks).

The header object is simply identified as `header`, and this is where we can define all of the code for the top of our page. We can use TypoScript to define a main menu, submenu, search box, login area, or any other information that we need at the top of every page. Because this is part of the skin, it can be changed here without any remapping and we can copy it over to other framework-based TYPO3 sites easier. This is some of the code from the original White Wireframe skin, and you can see that it defines the logo and the main menu in HTML and TypoScript:

```
header = COA
header {
    wrap = <div id="header"> | </div>
    10 = IMG_RESOURCE
    10 {
           file = {$siteLogo}
           stdWrap.dataWrap = <h3 id="masthead" style="width:{TSFE:last
ImgResourceInfo|0}px;height:{TSFE:lastImgResourceInfo|1}px;background:
url(|) no-repeat;"><a href="{$siteURL}">{$siteTitle}</a></h3>
    }
    20 = HMENU
    20 {
           entryLevel = 0
           wrap = <ul id="globalMenu">|</ul><div
class="clearOnly"> </div><!-- end #globalMenu  -->
           1 = TMENU
           1 {
                  noBlur = 1
                  NO {
                         subst_elementUid = 1
                         before = <li id="globalMenuItem-
{elementUid}">|*|<li id="globalMenuItem-{elementUid}">|*|<li
id="globalMenuItem-{elementUid}" class="last">
                         after = </li>
                         stdWrap.htmlSpecialChars = 1
                  }
           }
    }
}
```

The footer object is also identified simply as `footer` in the TypoScript file, and it can be found near the bottom of the code. We can use standard TypoScript to enter copyright information, contact details, and anything else that we need at the bottom of the screen just like we would in the TypoScript template setup of our website. In the code from the White Wireframe skin, we can see that Busy Noggin has included the TypoScript constant copyright and a link to his website as part of the footer object:

```
footer = TEXT
footer {
   data = date:U
   strftime = %Y
   dataWrap = <div class="clearOnly"> </div><div
id="footer" class="clear"><p id="footerCopyright">&copy; 
; |  {$copyright}</p><a id="footerHomeLink" href="http://www.
busynoggin.com/">www.busynoggin.com</a><div class="clearOnly"> </
div></div><!-- end #footer  -->
}
```

The TypoScript code includes many objects with names starting with `preCode` and `postCode` to wrap around different parts of the template. For example, `preCodeFeature` and `postCodeFeature` will place code directly before and after the feature content block, but only if the feature content area is used on a page. In the same way, there are `preCode` and `postCode` objects for almost all of the content blocks available in our templates, but they will only show up when that content block is used. We can wrap the secondary content block in all of our templates with two unique content areas with the objects `preCodeContentBlock-2` and `postCodeContentBlock-2`. There are a lot of `preCode` and `postCode` objects available, but most of them are set to null in the TypoScript code. The easiest way to start modifying them is to find the definition for the block we want to edit in a working example skin like the wireframe and add our own TypoScript to replace the null values.

Another helpful object, `additionalDocHeadCode`, can be found at the bottom of `skin_typoscript.ts`. We can use `additionalDocHeadCode` to add extra CSS or JavaScript to the head of our HTML structure right after the standard stylesheet and JavaScript includes. For example, the wireframe skin adds a special Internet Explorer 6 stylesheet after the rest of our CSS files by default:

```
additionalDocHeadCode = HTML
additionalDocHeadCode.value (
   <!--[if IE 6]>
          <link rel="stylesheet" type="text/css" href="{$templavoila_
framework.skinPath}css/ie6.css" />
   <![endif]-->
)
```

This is an advantage over the traditional way because it means that we can upload our skin to a staging server or copy it for another site, and we don't have to worry about editing the TypoScript template setup to load additional CSS files.

Finally, we can also target individual page templates with our TypoScript objects. Remember that all of the templates have a unique identifier like `F3b` or `F1c`. We can add the unique identifier to the beginning of our object path to target these specific page templates. For example, in TypoScript, we can set `f3b.header` to show a different menu configuration. All of the page templates except `F3b` will pull the code from the header object, but any pages using the `F3b` template will pull their header code from the `f3b.header` object. This can be useful if we want to create a specialized look for certain page templates, and it can be used very effectively with page template groups like `F1a`, `F1b`, and `F1c` where the only structural difference is the template identifier. For example, all three templates could be identical in everything except for the header.

Editing CSS

All of the CSS for our skin is stored in, of course, the `css/` directory. The TemplaVoila Framework will automatically call two default skins, `mainstyle.css` and `rte.css`, but we can also use the directory to store any additional CSS files that we call through the `additionalDocHeadCode` object in our `skin_typoscript.ts` file. In addition to the actual CSS files, we can store all of the images used by our stylesheets in the `css/images/` directory. By storing the images in a subfolder with our CSS, we can test our styles easier with tools like CSSEdit and insure that no images of ours are lost when we move our CSS to another framework installation or even another skin.

The `rte.css` file contains all of the styling like typography and paragraph or header styles that we want to include both in the rich text editor and the frontend. We've already seen this before in earlier chapters, but we want to give the editors a better backend experience by using the same fonts, line heights, headers (`h1-h6`), and other styles in the RTE as what is used in the frontend. Like before, we can also add unique classes to this stylesheet like `redText` or `blueText`, and we can configure the TSconfig to show them in the RTE drop-down menus.

The main stylesheet, `mainstyles.css`, is used for the frontend layout and contains all of the CSS code except what we have already placed in `rte.css`. The naming conventions, ID names, and class names in the `mainstyles.css` included with the wireframe skin provide a blueprint to building our own stylesheets. Each template has its own unique ID for the body tag, so they be can styled using declarations like `#f1d` and `#f2a`. In addition, the individual content blocks from the core templates have unique identifiers that we can see in the wireframe CSS like `#feature`, `#generatedContent-1`, and `#contentBlock-2`. By using the wireframe skin as an example, we can create our own styling from scratch or adapt CSS files from other developers to style our skin.

Editing TypoScript constants

After we've worked on some CSS, we will probably need to update the TypoScript constants for our skin. The TypoScript constants for our skin are defined in the `skin_constants.ts` file inside the `typoscript/` directory, and they are available to define the default settings for many of the framework's features like bleed and padding:

```
featureBleedDefault = 0
featureLeftPadding = 0
featureRightPadding = 0
```

These constants can be overridden in the TypoScript templates through the **Constants Editor**, but we can set defaults here to provide a consistent baseline for anywhere our skin is installed. In fact, on our own sites we should rarely have to set any of these constants in the **Constants Editor** because they are so easy to update with a text editor in the skin files.

More importantly than setting defaults, the TypoScript constants file defines the widths of the frontend containers for every core page template. The widths do not take into account padding, margin, or border sizes, but they are used to calculate the relative widths of columns, modules, and the maximum image sizes. If we change the size of our containers in the CSS, we need to adjust the values in this file. When the values are set correctly, the TemplaVoila Framework can automatically set the frontend size of columns or modules in pixels by taking into account the container width that we set here, gutters, and the distribution (Half / Half, Third / Two Third, and so on).

Adding JavaScript

All of our JavaScript files for the framework skin go in the `js/` directory, and a default JavaScript file, `skin.js`, is provided for us that will be automatically loaded by the templates. If we have additional JavaScript files that we want to use, they can be placed in the `js/` directory as well and included through the `additionalDocHeadCode` object in the TypoScript file for our skin.

The TemplaVoila Framework automatically loads jQuery to make the modules within a group equal in height using the built-in `core.js` file. If you do not want to automatically include the jQuery library, you can disable it by adding this constant declaration to the `skin_constants.ts` file:

```
enableJQuery = 0
```

Of course, if you choose to set this constant to zero neither jQuery nor the `core.js` file will be loaded. If you want to include jQuery some other way or have equal height modules, you will need to add your own JavaScript in the `js/` directory.

The `skin.js` file and any files that we add will be included after the jQuery library, so jQuery plugins can be used without a problem. In order to prevent conflicts with other JavaScript libraries, the TemplaVoila Framework loads jQuery in `noConflict` mode. This means that we must use the word `jQuery` instead of a `$` to load jQuery methods and objects:

```
// This is correct
jQuery('.myClass').hide();
//This is not correct
$('.myClass').hide();
```

Additional resources

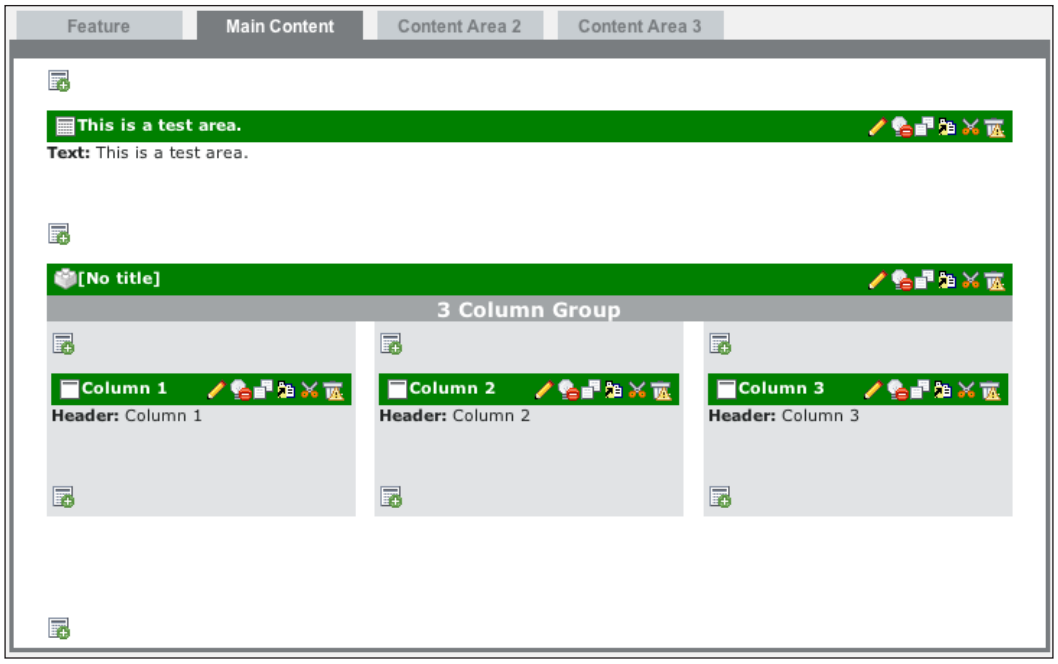
In addition to the required files and directories in our skin directory, we can create other directories to keep additional resources with our skin. For example, we can create a `fonts/` directory to store any fonts we load through TypoScript or CSS. Finally, we can replace the `screenshot.jpg` with an updated representation of our skin for the editors or other site builders to see in the **Skin Selector**.

Adding special functionality

Now that we've created a custom skin, we can still put more of our TemplaVoila and TypoScript skills to work. We might still need a lean, table-based page template for our e-mail newsletter, and we still might want to add another language to our site. Luckily, the TemplaVoila Framework is just a framework, and we still have all of the power of TemplaVoila and TypoScript at our disposal. We can create our own template objects in TemplaVoila to handle the newsletter, but this time we can use field names from the core data structure XML files so we don't need to remap anything. We can still add our own data structures fields as well. We can also plug in any of the TypoScript that we really need in the main template or create custom FCEs for advertising or product display. The workflow of our special functionality can all stay the same, or we can try to build on some of the standard features that the framework makes available. In either case, we're probably going to want to do some custom TypoScript and TemplaVoila coding using what we've learned, and the framework will get out of our way for that kind of development.

Adding content

As the TemplaVoila Framework is a framework for TypoScript and TemplaVoila, adding content and editing pages is almost exactly the same as it has always been. The only difference is a uniform new look to the backend using a tab for each content block, and you can see how clean it looks in the following screenshot. For normal editing, we can choose any templates that we didn't move to the **Unused Templates** folder in the **Page Properties** and start adding content. Any Utility FCEs that we did not move to the **Unused Templates** folder, like columns or modules, can be added to a page like the flexible contents that we used before. Here is an example of a test page with a three-column group in the main content area with the new backend layout:

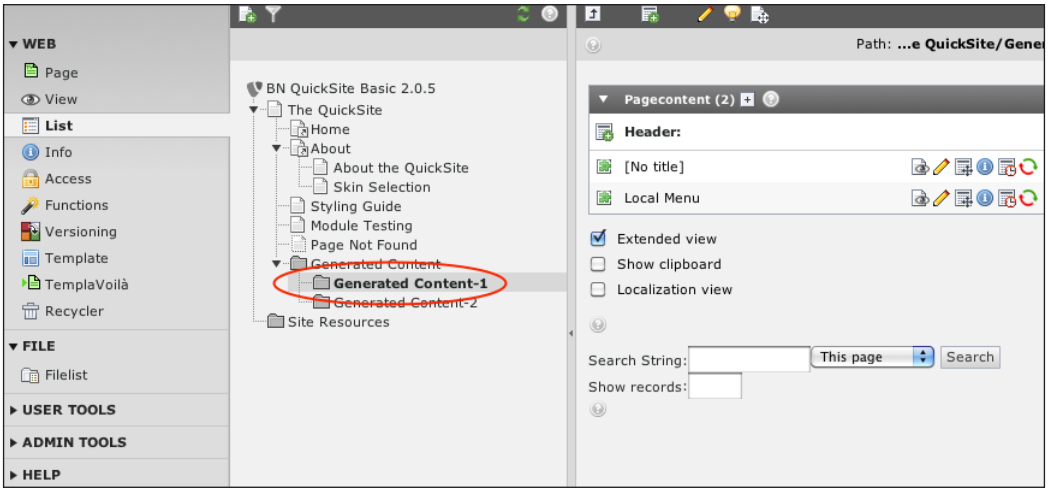


Feature content

The Feature block is specially designed for banners or featured content at the top of our pages. It is available in all of the core page templates, but is not shown on the frontend until we add content to it. In addition to only being created when it has content, the Feature block is special because we can set it to bleed (display edge-to-edge without margins or padding) in the **Page Properties**. If we don't set it in the **Page Properties**, it will automatically use the default setting in the skin's TypoScript constants file (`skin_constants.ts`).

Generated content

We can add generated content to our pages by using a core template with a generated content block and placing dynamic content elements into one of our generated content `SysFolders`:

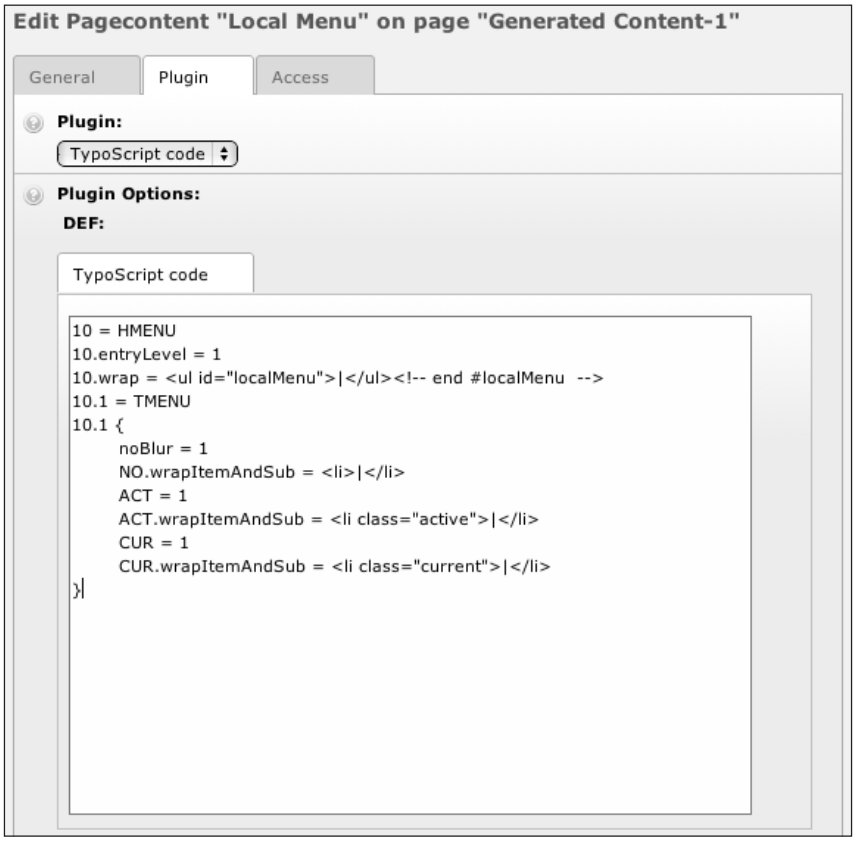


For example, a template with a content block mapped to `Generated Content-1` will automatically show all of the content elements in the `Generated Content-1` system folder. This is helpful for including local menus or generated ads in some of our pages.

If we want to create local menus, we normally need access to the template to use the power of TypoScript for this:

```
10 = HMENU
10.entryLevel = 1
10.wrap = <ul id="localMenu">|</ul><!-- end #localMenu  -->
10.1 = TMENU
10.1 {
    noBlur = 1
    NO.wrapItemAndSub = <li>|</li>
    ACT = 1
    ACT.wrapItemAndSub = <li class="active">|</li>
    CUR = 1
    CUR.wrapItemAndSub = <li class="current">|</li>
}
```

Luckily, QuickSite includes a **TypoScript Code** extension, so we can add our own arbitrary TypoScript by creating a new content element, choosing **Insert plugin** as the content type, and choosing the **TypoScript code** plugin from the **Plugin** drop-down menu. Then we can add our own TypoScript into the content element like this:



Summary

We just learned about a new framework that can help us build sites faster and more consistently with a standard code base and a core set of page templates and FCEs. Of course, we were really only able to scratch the surface of creating sites with the Busy Noggin's TemplaVoila Framework. The only way to really understand the Framework or TYPO3 templates at all is to start coding. Now that we know how it works as a framework, we can start integrating our knowledge of TypoScript menus to make our header really stand out with custom menus or build a skin that works with mobile browsers.

Even if we decide not to use the TemplaVoila Framework on a project today because we're working with an existing site that we're afraid of breaking, it's good that we took some time to learn the TemplaVoila Framework. It is a great chance to look at some best practices in action and see some of the future trends of TYPO3 (cleaner coding and rapid development); we might think about ways that we can reduce the amount of remapping necessary between our templates or make editing easier with tab-based back end layouts. For further reading about the TemplaVoila Framework, you can go to `http://templavoilaframework.sitebuilderlab.com`.

TYPO3 Templates summary

Congratulations, you made it! We have successfully learned how to create our own templates from scratch, add dynamic elements, create awesome menus, and make our own flexible content elements. In the end, we even learned about three emerging technologies that many other TYPO3 site creators are just starting to experiment with: mobile websites, international websites, and the TemplaVoila Framework.

Now is the most important part. You can relax for five minutes, drink your coffee, and feel accomplished that you finished this rather large book. After that, open a browser window to your test site, fire up your text editor, and start creating. You should still be at least a little excited about what you've learned, and I'm sure something in here has given you an idea that you want to work on. So, start building it. Play. Experiment. Make stuff.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Thank you for buying

TYPO3 Templates

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

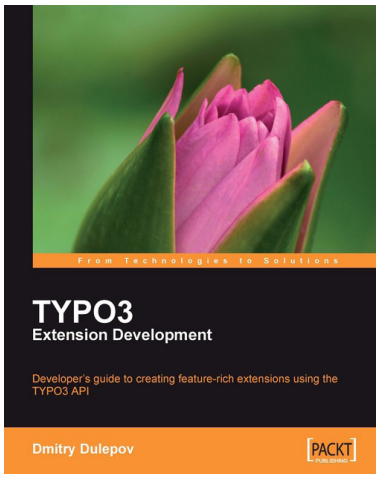
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



TYPO3 Extension Development

ISBN: 978-1-847192-12-7 Paperback: 232 pages

Developer's guide to creating feature rich extensions using the TYPO3 API

1. Covers the complete extension development process from planning and extension generation through development to writing documentation
2. Includes both front-end and back-end development
3. Describes TYPO3 areas not covered in the official documentation (such as using AJAX and eID)

Mastering TypoScript: TYPO3 Website, Template, and Extension Development

ISBN: 978-1-904811-97-8 Paperback: 400 pages

Build powerful web applications, quickly and cleanly, with the Django application framework

1. Powerful control and customization using TypoScript
2. Covers templates, extensions, admin, interface, menus, and database control
3. You don't need to be an experienced PHP developer to use the power of TypoScript