Ияну Аделекан

Kotlin

ПРОГРАММИРОВАНИЕ НА ПРИМЕРАХ

Создавайте реальный мир Android и веб-приложений вместе с Kotlin

bhv®

Packt›

Kotlin Programming By Example

Build real-world Android and web applications the Kotlin way

Iyanu Adelekan

Packt>

BIRMINGHAM - MUMBAI

Ияну Аделекан

Kotlin

ПРОГРАММИРОВАНИЕ НА ПРИМЕРАХ

Санкт-Петербург
«БХВ-Петербург»
2020

УДК 004.438 Kotlin
ББК 32.973.26-018.1
А29

Аделекан И.

А29 Kotlin: программирование на примерах: Пер. с англ. — СПб.: БХВ-Петербург, 2020. — 432 с.: ил.

ISBN 978-5-9775-6673-5

Книга посвящена разработке мобильных приложений для Android на языке Kotlin. Рассматриваются основные элементы языка, такие как функции и классы, приемы объектно-ориентированного программирования. Рассказывается о разработке микросервисов RESTful для приложений Android, о методах реализации шаблона архитектуры MVC. Описаны способы централизации, преобразования и хранения данных с применением Logstash, защиты приложений с использованием Spring Security. Изучается управление зависимостями с помощью Kotlin. Уделено внимание развертыванию микросервисов Kotlin для AWS и приложений Android в Play Store.

Для программистов

УДК 004.438 Kotlin
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Екатерина Сависте*
Перевод с английского	*Александра Сергеева*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 05.06.20.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 34,83.
Тираж 1000 экз. Заказ № 5777.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано в ОАО «Можайский полиграфический комбинат».
143200, Россия, г. Можайск, ул. Мира, 93.
www.oaompk.ru, тел.: (495) 745-84-28, (49638) 20-685

ISBN 978-1-78847-454-2 (англ.)
ISBN 978-5-9775-6673-5 (рус.)

Оглавление

Моим маме и папе

за их непоколебимую веру в меня и за безграничную любовь ко мне.
Никакие слова не смогут передать всю степень моей благодарности им

mapt.io

Mapt — это электронная онлайн-библиотека, предоставляющая своим посетителям полный доступ к более чем пяти тысячам книг и видеокурсов, а также к современным средствам обучения, которые помогут вам повышать свое профессиональное мастерство и строить карьеру. Для получения дополнительной информации, пожалуйста, посетите наш веб-сайт.

Для чего нужна подписка на Mapt?

- Получив практические советы и рекомендации из электронных книг и видеокурсов, созданных более чем четырьмя тысячами профессионалов в вашей ообласти интересов, вы станете тратить меньше времени на обучение и больше — на программирование.
- Вы усовершенствуете процесс своего обучения с помощью планов развития навыков, созданных специально для вас.
- Каждый месяц вы будете получать бесплатную электронную книгу или видеокурс.
- Библиотека Mapt содержит все необходимые средства для поиска нужных вам материалов.
- Копируйте наши материалы и вставляйте в свои разработки, распечатывайте их и добавляйте контент в закладки.

PacktPub.com

Знаете ли вы, что издательство Packt предлагает электронные версии всех опубликованных им книг в формате PDF и ePub? Каждый, кто приобрел печатную книгу издательства, может на сайте **www.PacktPub.com** со скидкой приобрести доработанную и исправленную копию ее электронной версии. Свяжитесь с нами по адресу **service@packtpub.com** для получения более подробной информации.

На сайте **www.PacktPub.com** вы также можете получить бесплатный доступ к подборке технических статей, подписаться на ряд бесплатных новостных рассылок и получить эксклюзивные скидки и предложения как на печатные книги Packt, так и на их электронные версии.

Об авторе

Ияну Аделекан (Iyanu Adelekan) — инженер-программист, увлеченный решением проблем, связанных с веб-приложениями и приложениями для Android. Ияну предпочитает работать над проектами с открытым исходным кодом, является автором и ведущим разработчиком Kanary — веб-фреймворка Kotlin, предназначенного для создания интерфейсов прикладного программирования RESTful. Помимо разработки программного обеспечения, использует любую возможность для популяризации знаний и алгоритмов программирования. В свободное время любит читать фантастику и играть в шахматы.

Я благодарен Богу за силы, которые он мне дал для начала и успешного завершения этой книги.

Я также благодарю своих братьев и сестер, родителей и друзей за их любовь и поддержку моей работы над созданием этого курса. Благослови вас Бог.

О рецензенте

Егор Андреевичи (Egor Andreevici) с 2010 года занимается разработкой приложений для Android. Недавно он переехал в Канаду, чтобы присоединиться к команде Square в качестве разработчика приложений Android. Егор увлекается IT-технологиями, чистым кодом, разработкой средств тестирования приложений и архитектурой программного обеспечения. Он открыл для себя Kotlin пару лет назад и с тех пор просто влюбился в лаконичность и выразительность этого языка. В свободное время Егор любит анализировать приложения с открытым исходным кодом, читать и путешествовать.

Packt ищет авторов

Если вы заинтересованы в том, чтобы стать автором Packt, сегодня же посетите сайт **authors.packtpub.com** и подайте заявку. Мы сотрудничаем с тысячами разработчиков и технических специалистов, помогая им поделиться с мировым техническим сообществом своим видением проблем. Вы можете прислать общую заявку, или подать заявку по конкретной актуальной теме, для которой мы ищем авторов, или представить свою собственную идею.

Предисловие

После того, как язык Kotlin был объявлен официально поддерживаемым языком для Android, его популярность значительно выросла. И эта популярность во многом оправданна, поскольку Kotlin — современный и глубоко продуманный язык, имеющий множество областей применения, включая интернет-приложения, мобильные устройства, собственные приложения и многое другое. На протяжении последних лет аудитория поклонников Kotlin растет быстрыми темпами.

Для кого предназначена эта книга?

Эта книга будет полезна читателям всех возрастов независимо от того, имеют ли они опыт программирования. Главное, чтобы они хотели изучать язык Kotlin.

При подготовке книги особое внимание уделялось подходу, основанному на соображении, что новичкам важно прежде всего ознакомиться с основными темами и концепциями изучаемого материала. Поэтому главы этой книги представлены в порядке возрастания его сложности. Даже если вы являетесь новичком, то всё равно сможете легко и быстро изучить язык Kotlin и писать программы, допуская минимальное количество ошибок.

Предполагается также, что опытные читатели быстрее, чем новички, освоят материал книги. Если вы имеете некоторый опыт в программировании и разработке приложений, просмотрите приведенные в книге примеры кода. Благодаря этим примерам вы сможете лучше представить круг рассматриваемых тем и перечень решаемых задач. Разработчикам Java можно пропустить начальные главы и сразу перейти к основному содержанию книги.

Независимо от того, к какой категории пользователей вы себя относите, будьте уверены в том, что именно ваши запросы учитывались при подготовке книги!

Структура книги

- *Глава 1. Основы* — продемонстрировано применение языка Kotlin для написания простых программ, показано, каким образом выполняется настройка нового проекта Android, а также представлены основные понятия, необходимые для разработки приложений Android, взаимодействующих с веб-серверами.

- *Глава 2. Создание Android-приложения Tetris* — поможет быстро перейти к разработке приложений, предназначенных для Android, поскольку в ней подробно описано создание классической игры тетрис.
- *Глава 3. Реализация логики и функциональности игры Tetris* — объяснит, каким образом создаются представления, как с помощью моделей реализуется логика приложения, а также как выполняется представление данных для просмотра. Кроме того, в ней приведены сведения об обработке событий пользовательского интерфейса.
- *Глава 4. Разработка и реализация серверной части интернет-мессенджера с помощью фреймворка Spring Boot 2.0* — объясняет, каким образом можно спроектировать и внедрить серверную часть приложения (бэкэнд), предоставляющую веб-ресурсы для клиентских приложений.
- *Глава 5. Создание Android-приложения Messenger: часть I* — посвящена созданию современных Android-приложений с использованием шаблона MVP (модель-представление-презентатор). В ней будет показано создание фронтэнда — клиентской части приложения Messenger, взаимодействующего с бэкэндом Messenger.
- *Глава 6. Создание Android-приложения Messenger: часть II* — служит продолжением предыдущей главы. В ней разработка приложения Messenger будет завершена.
- *Глава 7. Средства хранения данных* — рассказывает о различных методах хранения данных, которые предоставляются платформой приложений Android. В ней также показано, как использовать эти методы для создания приложений, обеспечивающих хранение и выборку полезной информации из различных хранилищ.
- *Глава 8. Защита и развертывание приложений Android* — содержит пошаговое описание процесса развертывания приложения Android. Кроме того, в главе даются ответы на важные вопросы безопасности применительно к Android-приложениям.
- Глава 9. *Создание серверной части веб-приложения Place Reviewer на основе платформы Spring* — содержит более подробное описание процесса разработки серверной части веб-приложения на примере создания в среде Spring Framework приложения Place Reviewer (Описание местоположений) и реализации его с помощью Spring MVC.
- *Глава 10. Реализация веб-интерфейса приложения Place Reviewer* — поясняет, каким образом создается интерфейс веб-приложения, описывающего какое-либо местоположение и задействующего возможности API Google Places. Из этой главы вы также узнаете, каким образом создаются тесты для веб-приложений.

Как получить максимум пользы от книги?

Если вы новичок, то при чтении этой книги от вас потребуется лишь настойчивое желание научиться программировать. Сначала задача по изучению нового языка программирования может показаться весьма сложной, но при определенном упорстве вы быстро станете профессионалом. Читайте каждую главу в том порядке, в котором она представлена, — так вам будет легче понять материал всей книги. Внимательно изучайте каждый фрагмент кода, пытаясь полностью осознать, что в нем выполняется. Самостоятельно реализуйте и запускайте каждую программу из этой книги.

Загрузите файлы примеров кода

Файлы примеров кода для этой книги можно загрузить из аккаунта, созданного на сайте **www.packtpub.com**. Независимо от того, где вы купили книгу, посетите сайт **www.packtpub.com/support** и создайте свой аккаунт, после чего вы сможете получить файлы кода по электронной почте.

Чтобы самостоятельно загрузить файлы кода:

1. Зарегистрируйтесь или авторизуйтесь на сайте **www.packtpub.com**.
2. Выберите вкладку **SUPPORT** (Поддержка).
3. Щелкните на кнопке **Code Downloads & Errata** (Загрузить файлы кода и опечатки).
4. Введите название книги в поле **Search** (Поиск) и следуйте дальнейшим инструкциям, которые отображаются на экране. После загрузки файла убедитесь, что распаковали или извлекли папку, применяя последнюю версию программ архивации:
   - WinRAR/7-Zip для Windows;
   - Zipeg/iZip/UnRarX для Mac;
   - 7-Zip/PeaZip для Linux.

Пакет кода для книги можно также найти на GitHub по адресу: **https://github.com/PacktPublishing/Kotlin-Programming-By-Example**. В случае обновления кода материал по этому адресу также будет обновлен.

На сайте **https://github.com/PacktPublishing** доступны и другие пакеты кода, относящиеся к достаточно большому каталогу книг и видеокурсов. Просмотрите их!

Файловый архив

Читатели русского издания книги могут бесплатно скачать файловый архив исходных кодов всех программ книги и ряд вспомогательных материалов с FTP-сервера издательства «БХВ-Петербург» по адресу: **ftp://ftp.bhv.ru/9785977566735.zip**, а также со страницы книги на сайте **www.bhv.ru**.

Загрузите цветные изображения

Мы также предоставляем PDF-файл с цветными изображениями, используемыми в этой книге. Этот файл можно загрузить по следующему адресу:

https://www.packtpub.com/sites/default/files/downloads/KotlinProgrammingByExample_ColorImages.pdf.

Используемые условные обозначения

В книге используется ряд текстовых выделений:

- `код в тексте` — моноширинным шрифтом `Courier` выделяются кодовые слова в тексте, имена таблиц базы данных, элементы, вводимые пользователем, и маркеры Твиттер.

  Пример: «В Java объект `HttpsURLConnection` может использоваться для безопасной передачи данных по сети»;

- myfile.jks — шрифтом Arial выделяются имена папок, имена файлов, расширений файлов и путей;

- блок кода выделяется следующим образом:

```
release {
   storeFile file("../my-release-key.jks")
   storePassword "password"
   keyAlias "my-alias"
   keyPassword "password"
}
```

- если нужно выделить определенную часть блока кода, соответствующие строки или элементы выделяются полужирным шрифтом:

```
release {
   storeFile file("../my-release-key.jks")
   storePassword "password"
   keyAlias "my-alias"
   keyPassword "password"
}
```

- ввод или вывод в командной строке записывается следующим образом:

```
./gradlew assembleRelease
```

- **полужирным** шрифтом выделяются слова, представленные на экране, опции меню и диалоговых окон программ, а также интернет-адреса (URL).

  Пример: «При необходимости введите остальные данные своей учетной записи и щелкните на кнопке **COMPLETE REGISTRATION** для завершения процесса регистрации учетной записи»;

- *курсивом* выделяются новые термины и важные слова.

  Пример: «Kotlin — это строго типизированный объектно-ориентированный язык, функционирующий на основе *виртуальной машины Java* (*JVM*)»;

-  таким образом выделяются Предупреждения или важные заметки;
-  таким значком обозначены полезные советы и различные пояснения.

Обратная связь

Мы приветствуем отзывы читателей книги:

- *общая обратная связь* — посылайте электронные письма по адресу: **feedback@packtpub.com** и упомяните при этом название книги в теме вашего сообщения. Если у вас имеются вопросы по любому аспекту этой книги, пожалуйста, пишите по адресу: **questions@packtpub.com**;
- *опечатки* — хотя мы заботились о точности изложения материала, ошибки случаются. Если вы нашли в книге ошибку, мы будем благодарны, когда вы сообщите нам об этом. Пожалуйста, посетите сайт **www.packtpub.com/submit-errata**, выберите книгу, щелкните на ссылке **Errata Submission Form** (Форма подачи заявки об опечатке) и введите свое сообщение;
- *борьба с пиратством* — если вы обнаружите незаконные копии наших работ в любой форме в Интернете, мы будем благодарны, если вы предоставите нам адрес местонахождения или название веб-сайта. Пожалуйста, свяжитесь с нами по адресу **copyright@packtpub.com** и укажите ссылку на незаконно опубликованный материал;
- *если вы хотите стать автором* — если вы хотите поделиться своими знаниями и опытом работы и заинтересованы в написании или публикации своей книги, посетите сайт: **authors.packtpub.com** и оставьте заявку.

Отзывы

Прочитав и освоив материал этой книги, пожалуйста, оставьте отзыв на сайте, где вы ее приобрели. Тогда потенциальные ее читатели смогут ознакомиться с ним и учесть ваше мнение при принятии решения о покупке книги, сотрудники Packt смогут узнать, что вы думаете о наших продуктах, а автор сможет ознакомиться с вашими отзывами, посвященными этой книге. Заранее благодарны! Для получения дополнительной информации о Packt, пожалуйста, обратитесь на сайт адресу: **packtpub.com**.

Читателям русского издания

Читатели русского издания книги могут посылать свои отзывы на адрес издательства «БХВ-Петербург» **mail@bhv.ru**, а также оставлять их на странице книги на сайте издательства **www.bhv.ru**.

1

Основы

Изучение языков программирования отпугивает многих людей, и они этого обычно избегают. Но поскольку вы все же обратились к данной книге, я предполагаю, что вы заинтересованы в освоении языка программирования Kotlin, и уверен в том, что со временем вы станете экспертом в этом вопросе. Позвольте же мне поздравить вас со смелым выбором по изучению языка Kotlin.

Независимо от проблемной области, для которой вы, возможно, будете создавать решения, будь то разработка приложений, работа в сети или формирование распределенных систем, язык Kotlin — это отличный выбор, позволяющий разрабатывать системы, обеспечивающие требуемые результаты. Другими словами, вы не ошибетесь, если приложите усилия для изучения Kotlin. Именно поэтому вам понадобится подходящее введение в этот язык программирования.

Итак, Kotlin — это строго типизированный объектно-ориентированный язык, функционирующий на основе *виртуальной машины Java* (JVM), которая может применяться при разработке приложений во многих проблемных областях. В дополнение к работе с JVM Kotlin может компилироваться и для JavaScript и вследствие этого представляет собой серьезный вариант для выбора при разработке клиентских веб-приложений. Kotlin также может компилироваться с помощью технологии Kotlin/Native непосредственно в собственные двоичные файлы, которые функционируют в системах при отсутствии виртуальной машины. Язык программирования Kotlin первоначально был разработан компанией JetBrains, которая базируется в Санкт-Петербурге (Россия). Разработчики из JetBrains и являются специалистами, поддерживающими функционирование языка. Язык Kotlin назван в честь острова Котлин, находящегося в море недалеко от Санкт-Петербурга.

Язык Kotlin предназначен для разработок промышленного программного обеспечения во многих областях, но в большинстве случаев используется для экосистемы Android. На момент подготовки книги Kotlin — это один из трех языков, объявлен-

ных Google в качестве официальных языков для Android. Синтаксически Kotlin подобен языку Java, и целью создания языка Kotlin фактически стало получение альтернативы Java. Вследствие этого у него имеется множество важных преимуществ, выявляющихся при использовании Kotlin вместо Java при разработке программного обеспечения.

В этой главе рассматриваются следующие темы:

- установка Kotlin;
- основы программирования на языке Kotlin;
- инсталляция и установка Android Studio;
- плагин Gradle;
- основы работы в Интернете.

Начало работы с Kotlin

Для разработки программ на языке Kotlin сначала нужно установить на вашем компьютере *среду выполнения Java* (Java Runtime Environment, JRE). Среда JRE загружается в одном архивном файле с *набором разработки Java* (Java Development Kit, JDK). Так что мы здесь воспользуемся JDK.

Наиболее простой способ установить JDK на компьютере — обратиться к одному из установщиков JDK, предоставляемых фирмой Oracle, которая и владеет Java. На ее сайте по адресу: **http://www.oracle.com/technetwork/java/javase/downloads/index.html** доступны различные установщики, предназначенные для основных операционных систем (рис. 1.1).

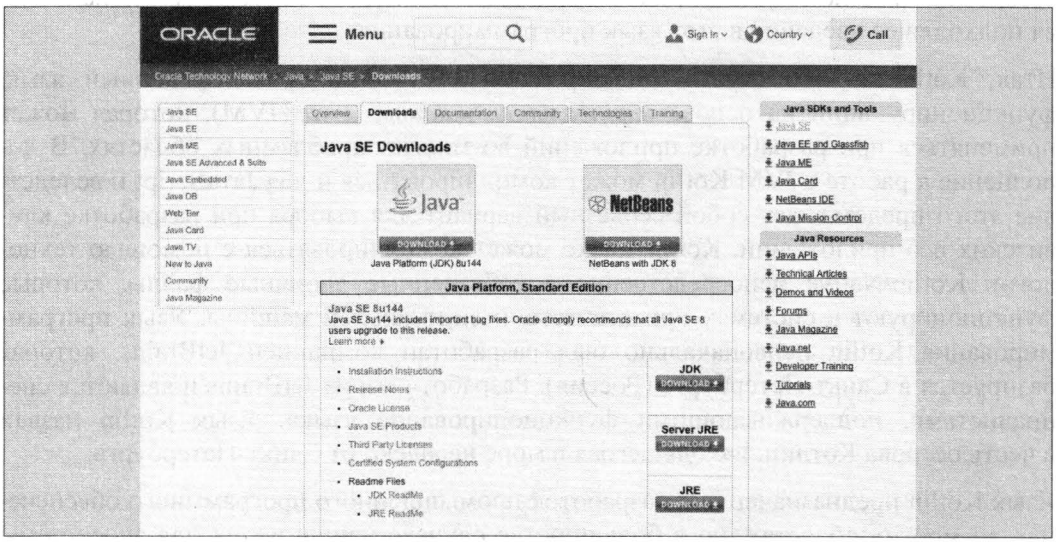


Рис. 1.1. Веб-страница Java SE

Щелкнув на кнопке загрузки JDK, мы попадем на веб-страницу, откуда можно загрузить JDK, соответствующий вашей операционной системе и архитектуре процессора (рис. 1.2). Загрузите подходящий для вашего компьютера JDK и перейдите к следующему разделу.

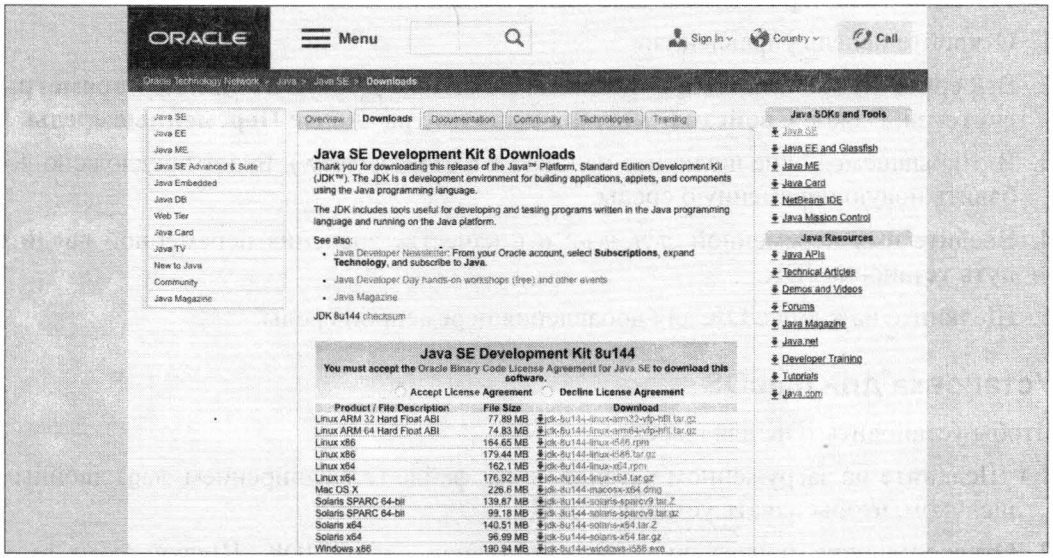


Рис. 1.2. Страница загрузки JDK

Установка JDK

Порядок установки JDK на компьютере зависит от используемой операционной системы. Далее рассмотрены соответствующие ей варианты установки.

Установка для Windows

Чтобы установить JDK для Windows:

1. Щелкните на загруженном установочном файле (с расширением exe) двойным щелчком, чтобы начать установку JDK.
2. В окне приветствия щелкните на кнопке **Next** (Далее). После этого откроется окно, в котором можно выбрать компоненты для установки. Оставьте заданный по умолчанию вариант выбора и щелкните на кнопке **Next**.
3. В следующем окне отобразится предложение по выбору папки, в которой будут находиться файлы после установки. Пока оставьте выбранной папку по умолчанию (но запомните расположение этой папки, поскольку она понадобится вам позже). Щелкните на кнопке **Next**.
4. Следуйте инструкциям, представленным в следующих окнах и по мере надобности щелкайте на кнопке **Next**. У вас могут запросить пароль администратора,

при необходимости введите его. После этого Java будет установлена на вашем компьютере.

После завершения установки JDK необходимо настроить значение переменной среды JAVA_HOME. Для этого в Windows 7 выполните следующие действия (в других версиях Windows порядок действий аналогичен):

1. Откройте панель управления.
2. Выберите опцию **Система** и щелкните на ссылке **Дополнительные параметры системы**. В окне **Свойства системы** щелкните на кнопке **Переменные среды**.
3. В открывшемся окне щелкните на кнопке **Создать** (New). Будет предложено добавить новую переменную среды.
4. Введите имя переменной JAVA_HOME и в качестве значения переменной введите путь установки JDK.
5. Щелкните на кнопке **OK** для добавления переменной среды.

Установка для macOS

Чтобы установить JDK для macOS:

1. Щелкните на загруженном установочном файле (с расширением dmg) двойным щелчком, чтобы начать установку JDK.
2. Откроется окно поиска, включающее значок пакета JDK. Щелкните на этом значке двойным щелчком для запуска программы установки.
3. В начальном окне щелкните на кнопке **Continue** (Продолжить).
4. В открывшемся окне установки щелкните на кнопке **Install** (Установить).
5. При необходимости введите логин и пароль администратора и щелкните на кнопке **Install Software** (Установить программу).

После этого завершится установка JDK и появится окно подтверждения.

Установка для Linux

Установить JDK для Linux легко и просто с помощью команды `apt-get`:

1. Обновите индекс пакетов вашего компьютера. Для этого в окне терминала выполните следующую команду:

```
sudo apt-get update
```

2. Проверьте, установлена ли Java, выполнив следующую команду:

```
java -version
```

3. Если Java установлена, будет выведена информация о версии установленной Java в вашей системе. Если версия Java не установлена, выполните команду:

```
sudo apt-get install default-jdk
```

На этом все! Установка JDK завершена.

Компиляция программ Kotlin

Теперь, когда JDK настроен и подготовлен к работе, нужно установить средство для фактической компиляции и запуска ваших программ на языке Kotlin.

Программы Kotlin могут компилироваться непосредственно с помощью компилятора командной строки Kotlin или собираться и запускаться в интегрированной среде разработки (IDE, Integrated Development Environment).

Компилятор командной строки может устанавливаться в macOS с помощью менеджеров пакетов Homebrew и MacPorts, в Linux — с помощью менеджера пакетов SDKMAN!, и в Windows — с помощью специального программного обеспечения (см. далее). Также компилятор командной строки можно установить и вручную.

Установка компилятора командной строки в macOS

Компилятор командной строки Kotlin может быть установлен на платформе macOS различными способами. Два наиболее распространенных способа его установки на macOS — с помощью менеджеров пакетов Homebrew и MacPorts.

Homebrew

Homebrew — менеджер пакетов для платформы macOS. Он широко используется для установки пакетов, необходимых для создания программных проектов. Чтобы установить Homebrew, откройте свой терминал macOS и запустите Homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Придется подождать несколько секунд, пока он загрузится и установится. После установки проверьте, правильно ли Homebrew работает, выполнив следующую команду в окне терминала:

```
brew -v
```

Если название текущей версии Homebrew выводится в терминале, значит, Homebrew на вашем компьютере уже установлен.

После завершения установки Homebrew найдите свой терминал и выполните следующую команду:

```
brew install kotlin
```

Дождитесь окончания установки, после чего вы будете готовы компилировать программы Kotlin с помощью компилятора командной строки.

MacPorts

Как и HomeBrew, MacPorts является менеджером пакетов для macOS. Чтобы установить MacPorts, выполните следующие действия:

1. Установите Xcode и инструменты командной строки Xcode.
2. Получите лицензию Xcode. Это можно сделать в терминале, выполнив следующую команду: `xcodebuild -license`.
3. Установите необходимую версию MacPorts.

Версии MacPort можно загрузить на сайте по адресу: **https://www.macports.org/install.php**.

После загрузки запустите терминал и выполните с правами суперпользователя команду:

```
sudo port install kotlin
```

Установка компилятора командной строки в Linux

Пользователи Linux могут легко установить компилятор командной строки для Kotlin с помощью менеджера пакетов SDKMAN!

Этот подход часто используется для установки пакетов в системах на основе UNIX, таких как Linux и его различные дистрибутивы, например Fedora и Solaris. Чтобы установить SDKMAN!, выполните три несложных действия.

1. Загрузите программное обеспечение в систему с помощью команды `curl`, для чего в окне терминала выполните следующую команду:

   ```
   curl -s "https://get.sdkman.io" | bash
   ```

2. После выполнения предыдущей команды в окне терминала отобразится набор инструкций. Следуйте этим инструкциям для завершения установки. После выполнения инструкций запустите следующую команду:

   ```
   source "$HOME/.sdkman/bin/sdkman-init.sh"
   ```

3. Выполните следующую команду:

   ```
   sdk version
   ```

Если номер версии установленного приложения SDKMAN! отображается в окне терминала, значит, установка прошла успешно.

Успешно установив в системе SDKMAN!, можно установить компилятор командной строки, выполнив команду:

```
sdk install kotlin
```

Установка компилятора командной строки в Windows

Для работы с компилятором командной строки Kotlin в Windows выполните следующие действия:

1. Загрузите версию программного обеспечения на сайте GitHub по следующему адресу: **https://github.com/JetBrains/kotlin/releases/tag/v1.2.30**.
2. Найдите и разархивируйте загруженный файл.

3. Откройте извлеченную папку kotlinc\bin.
4. Запустите командную строку с путем к папке.

Теперь можно и в Windows использовать компилятор Kotlin из командной строки.

Запуск вашей первой программы Kotlin

Установив компилятор командной строки, опробуем его с помощью простой программы Kotlin. Перейдите в домашний каталог и создайте новый файл с именем Hello.kt (все файлы Kotlin имеют расширение kt).

Откройте только что созданный файл в окне текстового редактора по вашему выбору и введите в него следующий текст:

```
// Вывод приветствия Hello world в стандартной системе
output.
fun main (args: Array<String>) {
   println("Hello world!")
}
```

Сохраните изменения, внесенные в файл программы. После сохранения изменений откройте окно терминала и введите следующую команду:

```
kotlinc hello.kt -include-runtime -d hello.jar
```

Эта команда компилирует вашу программу в исполняемый файл hello.jar.

Флаг `-include- runtime` указывает, что скомпилированный файл JAR должен стать автономным, — при наличии в команде этого флага библиотека времени выполнения (`runtime`) Kotlin будет включена в JAR-файл. Флаг `-d` указывает, что в нашем случае нужно, чтобы вызывался вывод компилятора.

Завершив компиляцию первой программы Kotlin, запустите ее — в конце концов, написание программ не доставляет удовольствия, если их нельзя запускать. Откройте окно терминала, если оно еще не открыто, и перейдите в каталог, где сохранен JAR-файл (в нашем случае — домашний каталог). Для запуска скомпилированного JAR-файла выполните следующую команду:

```
java -jar hello.jar
```

После выполнения этой команды вы должны увидеть приветствие **Hello world!** Если это так, поздравляем — вы только что написали первую программу Kotlin!

Написание сценариев с помощью Kotlin

Одна из возможностей Kotlin заключается в том, что его можно использовать для написания сценариев. *Сценарии* — это программы, написанные для конкретных сред исполнения с общей целью автоматизации выполнения задач. Сценарии Kotlin имеют добавленное к имени файла расширение kts.

Написание сценария Kotlin напоминает создание программы Kotlin. На самом деле сценарий Kotlin идентичен обычной программе Kotlin, и единственное важное от-

личие сценария Kotlin от обычной программы Kotlin — это отсутствие основной функции.

Создайте файл в выбранном вами каталоге и назовите его NumberSum.kts. Откройте файл и введите в него следующий текст:

```
val x: Int = 1
val y: Int = 2
val z: Int = x + y
println(z)
```

Как вы наверняка догадались, этот сценарий выводит сумму `1` и `2` в стандартном окне вывода системы. Сохраните изменения в файле и запустите сценарий:

```
kotlinc -script NumberSum.kts
```

Важно отметить, что сценарий Kotlin не нужно компилировать.

Применение REPL

Аббревиатура REPL образована от слов *Read–Eval–Print Loop* (цикл чтение-вычисление-печать). REPL — это интерактивная оболочка, в которой программы могут выполняться с немедленным получением результатов. Интерактивная оболочка вызывается с помощью команды `kotlinc` без каких-либо аргументов.

Для запуска Kotlin REPL выполните команду `kotlinc` в окне терминала.

Если REPL успешно запущена, в окне терминала отобразится приветственное сообщение с символами >>>, предупреждающее тем самым, что REPL ожидает ввода. Вы можете ввести код в терминале, как в любом текстовом редакторе, и получить немедленную обратную связь с REPL (рис. 1.3).

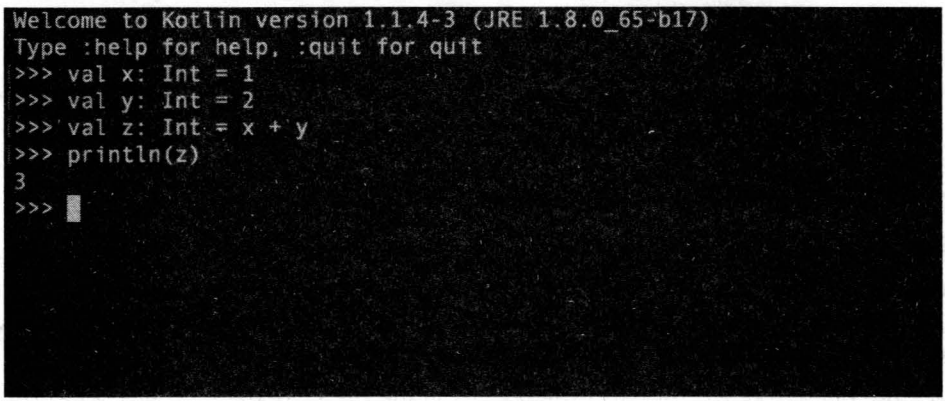


Рис. 1.3. Kotlin REPL

Как показано на рис. 1.3, целые значения `1` и `2` присваиваются, соответственно, переменным `x` и `y`. Сумма `x` и `y` сохраняется в новой переменной `z`, а переменная и ее значение `z` выводятся в окне вывода с помощью функции `println()`.

Работа в IDE

Создание программ с помощью командной строки вполне возможно, хотя обычно разработчики для создания приложений применяют специальное программное обеспечение. Этот подход целесообразен при работе над крупными проектами.

Интегрированная среда разработки (Integrated Development Environment, IDE) представляет собой компьютерное приложение, включающее набор инструментов и утилит, предназначенных для программистов, которые разрабатывают приложения. Существует ряд IDE, применяемых для разработок на языке Kotlin. Из подобных IDE именно IntelliJ IDEA обладает наиболее полным набором функций, используемых для разработки приложений Kotlin. Поскольку IntelliJ IDEA создавалась разработчиками Kotlin, ее применение обеспечивает ряд преимуществ по сравнению с другими IDE — например, она обладает обширным набором функций для написания программ Kotlin и наличием своевременных обновлений, учитывающих самые последние усовершенствования и дополнения к языку программирования Kotlin.

Установка IntelliJ IDEA

Среду IntelliJ IDEA для Windows, macOS и Linux можно загрузить непосредственно с веб-сайта JetBrains по адресу: **https://www.jetbrains.com/idea/download**. Здесь доступны для загрузки две версии: платная версия — **Ultimate** и бесплатная версия — **Community** (рис. 1.4). Версии Community вполне достаточно, если вы будете

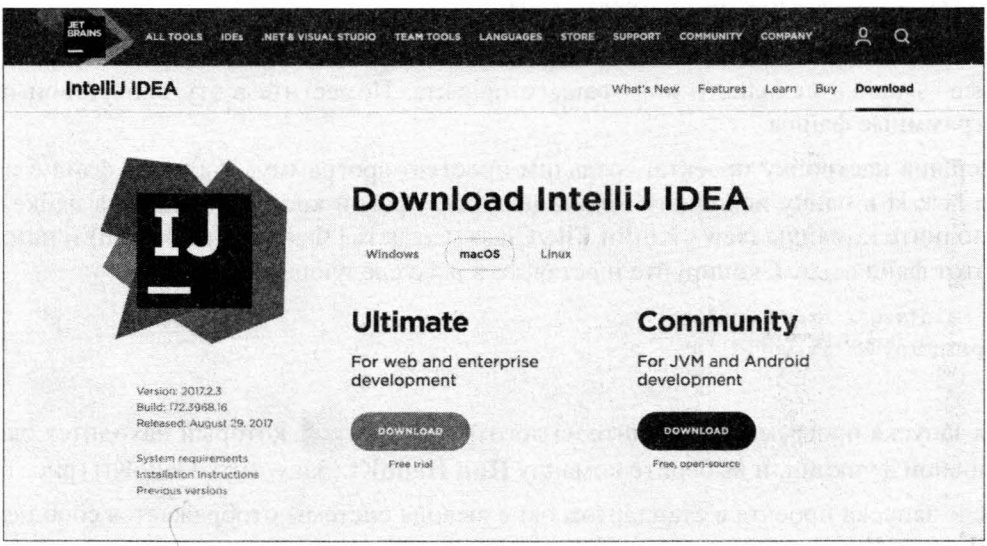


Рис. 1.4. Страница загрузки IntelliJ IDEA

запускать программы лишь из этой главы. Выберите требуемую версию IDE, загрузите ее, а после завершения загрузки щелкните на загруженном файле двойным щелчком и установите среду в операционной системе, как и любую другую программу.

Настройка проекта Kotlin с помощью IntelliJ

Процесс настройки проекта Kotlin с помощью IntelliJ достаточно прост:

1. Запустите приложение IntelliJ IDE.
2. Щелкните на кнопке **Create New Project** (Создать новый проект).
3. Выберите **Java** среди доступных опций проекта в левой части вновь открытого окна.
4. Добавьте Kotlin/JVM в качестве дополнительной библиотеки проекта.
5. Выберите SDK проекта из раскрывающегося списка в окне.
6. Щелкните на кнопке **Next** (Далее).
7. Выберите шаблон, если необходимо его использовать, затем перейдите к следующей странице.
8. Укажите название проекта в соответствующем поле ввода — назовите проект `HelloWorld`.
9. Установите местоположение проекта в поле ввода.
10. Щелкните на кнопке **Finish** (Готово).

После этого ваш проект будет создан, и вы увидите окно IDE (рис. 1.5).

В левой части окна отобразится представление проекта, отражающее логическую структуру его файлов. Здесь имеются две папки:

- .idea — ключает специфичные для проекта файлы настроек IntelliJ;
- src — папка исходного кода вашего проекта. Поместите в эту папку свои программные файлы.

Завершив настройку проекта, создадим простую программу. Добавьте файл с именем hello.kt в папку исходного кода: щелкните правой кнопкой мыши на папке src, выполните команды **New | Kotlin File/Class** (Создать | Файл/Класс Kotlin) и назовите этот файл `hello`. Скопируйте и вставьте в него следующий код:

```
fun main(args: Array<String>) {
   println("Hello world!")
}
```

Для запуска программы щелкните на логотипе Kotlin  , который находится около основной функции, и выберите команду **Run HelloKt** (Запустить HelloKt) (рис. 1.6).

После запуска проекта в стандартом окне вывода системы отображается сообщение **Hello world!**.

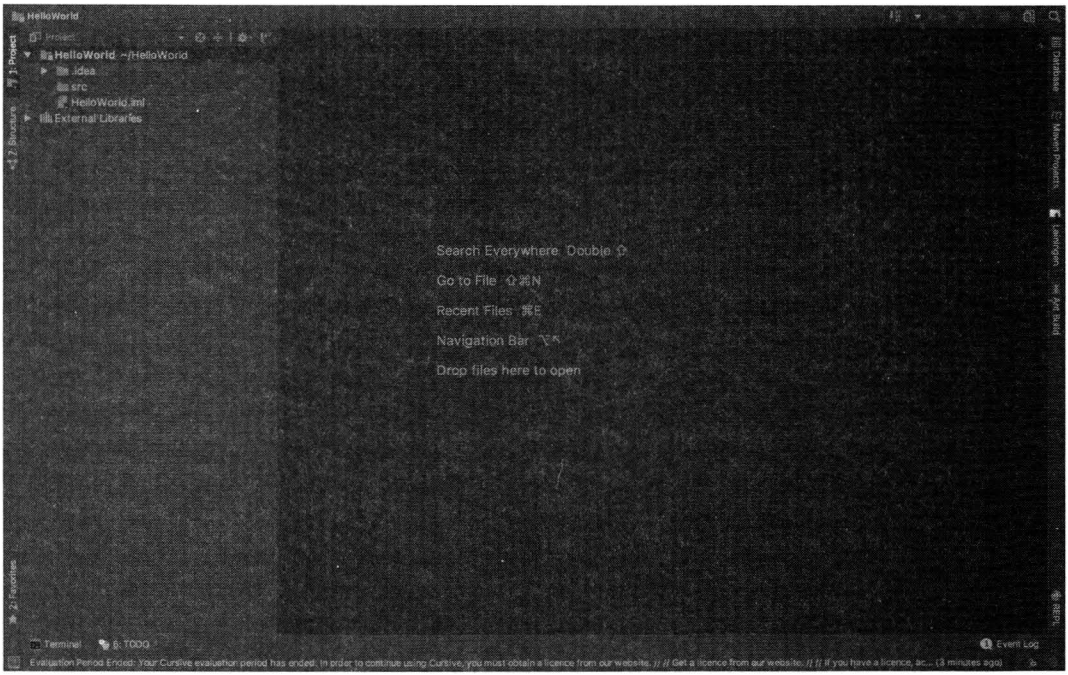


Рис. 1.5. Окно проекта в среде IntelliJ IDEA

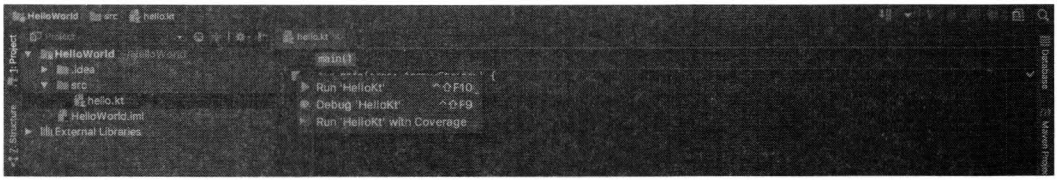


Рис. 1.6. Запуск проекта в среде IntelliJ IDEA

Основы языка программирования Kotlin

Теперь, после настройки среды разработки и среды IDE, перейдем к изучению собственно языка Kotlin. Начнем со знакомства с его основами и затем перейдем к более сложным темам, таким как *объектно-ориентированное программирование* (*ООП*).

Базовые понятия

В этом разделе мы рассмотрим базовые понятия языка Kotlin — так сказать, строительные блоки языка. И начнем с переменных.

Переменные

Переменная — это идентификатор ячейки памяти, в которой хранится значение. Более просто описывать переменную с помощью идентификатора, содержащего значение. Рассмотрим следующую программу:

```
fun main(args: Array<String>) {
   var x: Int = 1
}
```

Перед x указано, что это — переменная (var), ее значение равно 1. Точнее говоря, x является *целой* переменной. Переменная x является целой переменной, поскольку x определена как переменная типа Int. Соответственно, переменная x может содержать только целочисленные значения. Можно также сказать, что x является *экземпляром класса* Int. Пока еще вам может быть не совсем понятно, что в этом контексте означают слова *экземпляр* и *класс*. В дальнейшем это станет ясно, а пока что внимательнее рассмотрим переменные.

При определении переменной в Kotlin мы используем ключевое слово var. Это ключевое слово показывает, что переменная (от *англ.* variable) изменчива по своей природе — ее значение может меняться. *Тип данных* объявленной переменной следует после двоеточия, идущего за идентификатором переменной. Надо отметить, что тип данных переменной не должен определяться явно. Причина в том, что язык Kotlin поддерживает *выведение типов* — автоматическое определение типа переменной на основании данных, которыми переменная инициализируется. Определение переменной x могло бы быть следующим:

```
var x = 1
```

В результате мы получим переменную того же типа Int.

Точку с запятой можно добавлять в конец строки определения переменной, но так же как в языках, подобных JavaScript, она не требуется:

```
var x = 1 // Переменная, идентифицированная x, содержащая 1
var y = 2 // Переменная, идентифицированная y, содержащая 2
var z: Int = x + y // Переменная, идентифицированная z, содержащая 3
```

Если не нужно, чтобы значения переменных изменялись в ходе выполнения программы, можно сделать их неизменяемыми. Неизменяемые переменные определяются с помощью ключевого слова val следующим образом:

```
val x = 200
```

Область действия переменной

Область действия переменной — это область программы, в которой переменная имеет значение. Другими словами, область действия переменной — область программы, где может применяться переменная. Переменные языка Kotlin имеют блочную область видимости. Следовательно, переменные могут применяться во всех областях, для которых они были определены:

```
fun main(args: Array<String>) {
   // начало блока A
   var a = 10
   var i = 1

   while (i < 10) {
      // начало блока B
      val b = a / i
      print(b)
      i++
   }
   print(b) // ошибка: переменная b за пределами области видимости
}
```

В этой программе можно непосредственно наблюдать влияние области действия переменных на примере двух блоков. При определении функции открывается новый блок. Этот блок помечен в нашем примере как `A`. Внутри блока `A` объявлены переменные `a` и `i`. Таким образом, область действия переменных `a` и `i` находится внутри блока `A`.

Цикл `while` сформирован внутри блока `A`, и открыт новый блок `B` (объявления циклов отмечают начало новых блоков). В пределах блока `B` объявлено значение `b`. То есть значение `b` находится в области блока `B` и не может использоваться вне этой области. Таким образом, если попытаться вывести значение, хранящееся в `b`, вне блока `B`, появится ошибка.

Следует отметить, что переменные `a` и `i` в блоке `B` использоваться могут. Дело в том, что блок `B` существует в области действия блока `A`.

Локальные переменные

Так называются переменные, локальные для области. Переменные `a`, `i` и `b` в предыдущем примере являются локальными переменными.

Операнды и операторы

Оператор — это часть инструкции, определяющая значение, с которым необходимо работать. Оператор также выполняет определенную операцию над своими *операндами*. Вот примеры операторов: `+`, `-`, `*`, `/` и `%`. Операторы могут классифицироваться в зависимости от типа исполняемых операций и числа операндов, которые находятся под воздействием оператора.

Основываясь на типе операций, которые выполняются оператором, можно привести следующую классификацию операторов в Kotlin (табл. 1.1):

- реляционные операторы;
- операторы присваивания;
- логические операторы;

- арифметические операторы;
- поразрядные операторы.

Таблица 1.1. *Классификация операторов с примерами*

Тип оператора	Примеры
Реляционные операторы	>, <, >=, <=, ==
Операторы присваивания	+=, -=, *=, /=, =
Логические операторы	&&, ||, !
Арифметические операторы	+, -, *, /
Поразрядные операторы	and(bits), or(bits), xor(bits), inv(), shl(bits), shr(bits), ushr(bits)

Исходя из количества операндов, мы можем определить два основных типа операторов в Kotlin (табл. 1.2):

- унарные операторы;
- бинарные операторы.

Таблица 1.2. *Основные типы операторов с примерами*

Тип оператора	Описание	Примеры
Унарный оператор	Требуется только один операнд	!, ++, --
Бинарный оператор	Требуются два операнда	+, -, *, /, %, &&, ||

Типы переменной

Тип переменной с учетом ее области значений представляет набор возможных значений этой переменной. Во многих случаях имеет смысл явно указывать тип значения, которое необходимо сохранить в объявленной переменной. Этого можно добиться при обращении к типу данных. Наиболее важными типами языка Kotlin являются следующие:

- `Int`;
- `Float`;
- `Double`;
- `Boolean`;
- `String`;
- `Char`;
- `Array`.

Int

Этот тип представлен 32-разрядным целым числом со знаком. Если переменная объявляется с этим типом, пространство значений переменной представлено набором целых чисел, и переменная может иметь только целочисленные значения. Тип `Int` уже встречался нам несколько раз в рассмотренных ранее примерах. К типу `Int` относятся целые значения, находящиеся в диапазоне от –2 147 483 648 до 2 147 483 647.

Float

Этот тип представлен 32-битным числом с плавающей запятой одинарной точности. При объявлении переменной с типом `Float` он указывает, что переменная может включать только значения с плавающей запятой. Диапазон составляет приблизительно ±3,40282347E+38F (6–7 значащих десятичных цифр).

Пример: `var pi: Float = 3.142`

Double

Этот тип представляет 64-битное число с плавающей запятой двойной точности. Подобно типу `Float`, тип `Double` указывает, что объявленная переменная может включать значения с плавающей точкой. Важное различие между типами `Double` и `Float` в том, что `Double` сохраняет без переполнения числа, находящиеся в гораздо большем диапазоне. Этот диапазон составляет примерно ±1,79769313486231570E+308 (15 значащих десятичных цифр).

Пример: `var d: Double = 3.142`

Boolean

Истинные и ложные логические значения истинности представлены типом `Boolean`:

```
var t: Boolean = true
var f: Boolean = false
```

Логическими операторами здесь являются `&&`, `||`, и `!` (табл. 1.3).

***Таблица 1.3.** Логические операторы с примерами*

Название оператора	Оператор	Описание	Тип оператора
Конъюнкция	`&&`	Значение `true`, если два его операнда имеют значение `true`, в противном случае — значение `false`	Бинарный
Дизъюнкция	`||`	Значение `true`, если хотя бы один операнд имеет значение `true`, в противном случае — значение `false`	Бинарный
Отрицание	`!`	Инвертирует значение булева операнда	Унарный

String (строка)

Строка — последовательность символов. В Kotlin строки представлены классом `string`. Строки можно писать, вводя последовательность символов и заключая ее в двойные кавычки. Пример:

```
val martinLutherQuote: String = "Free at last, Free at last, Thank God
almighty we are free at last."
```

Char

Этот тип используется для представления *символов*. Символ — это единица информации, которая приблизительно соответствует графеме, или графемоподобной единице или символу. В Kotlin символы относятся к типу `Char`, причем символы в одинарных кавычках, такие как `$`, `%`, и `&`, служат примерами символов:

```
val c: Char = 'i' // Пример символа
```

Напомним, как упоминалось ранее, что строка является последовательностью символов:

```
var c: Char
val sentence: String = "I am made up of characters."
for (character in sentence) {
   c = character // Значение символа, присвоенное c, без ошибки
   println(c)
}
```

Array (массив)

Массив — это структура данных, состоящая из набора элементов или значений, причем каждый элемент имеет хотя бы один индекс, или ключ. Массивы удобно использовать для хранения наборов элементов, которые позднее задействуются в программе.

В языке Kotlin массивы создаются с помощью библиотечного метода `arrayOf()`. Значения, которые необходимо сохранить в массиве, передаются в виде последовательности, элементы которой разделены запятыми:

```
val names = arrayOf("Tobi", "Tonia", "Timi")
```

Каждое значение массива имеет уникальный индекс, определяющий его положение в массиве, он применяется в дальнейшем для выборки этого значения. Набор индексов в массиве начинается с индекса 0 и увеличивается с шагом 1.

Значение, находящееся в любой заданной позиции индекса массива, можно получить, вызывая метод `Array#get()` или же применяя операцию `[]`:

```
val numbers = arrayOf(1, 2, 3, 4)
println(numbers[0]) // выводит 1
println(numbers.get(1)) // выводит 2
```

В каждый момент времени можно изменить значение позиции для массива:

```
val numbers = arrayOf(1, 2, 3, 4)
println(numbers[0]) // выводит 1
numbers[0] = 23
println(numbers[0]) // выводит 23
```

Размер массива можно проверять в любой момент с помощью свойства `length`:

```
val numbers = arrayOf(1, 2, 3, 4)
println(numbers.length) // выводит 4
```

Функции

Функция — это блок кода, определяемый однократно и применяемый произвольное число раз. При написании программ рекомендуется разбивать сложные программные процессы на более мелкие блоки, которые выполняют конкретные задачи. Выполнить это необходимо по нескольким причинам:

- *улучшение читаемости кода* — гораздо проще читать программы, которые разбиты на функциональные блоки. Объем воспринимаемого в тот или иной момент кода значительно сокращается при использовании функций. На написание или настройку разделов большой базы кода программист тратит слишком много времени. А при использовании функций контекст программы, с которым необходимо ознакомиться при совершенствовании логики программы, ограничивается лишь телом функции, в котором находится логика;
- *более удобное сопровождение базы кода* — использование функций в базе кода облегчает поддержку программы. Если нужно вносить изменения в конкретное свойство программы, это столь же просто, как и настройка функции, где создано это свойство.

Объявление функций

Функции объявляются с помощью ключевого слова `fun`. Вот пример простого определения функции:

```
fun printSum(a: Int, b: Int) {
   print(a + b)
}
```

Эта функция просто выводит сумму двух значений, которые переданы ей в качестве аргументов. Определение функций можно разбить на следующие компоненты:

- *идентификатор функции* — присвоенное ей имя. Идентификатор необходимо применять при ссылке на функцию, если позднее ее следует вызвать в программе. В показанном примере объявления функции идентификатором функции является фрагмент `printSum`;
- *пара круглых скобок, содержащих разделенный запятыми список аргументов, которые передаются в качестве значений функции.* Значения, передаваемые

в функцию, называются *аргументами функции*. Все передаваемые в функцию аргументы должны иметь тип. Определение типа аргумента следует за двоеточием, помещенным после имени аргумента;

- *спецификация типа возврата* — типы возвращаемых функций указываются так же, как и типы переменных и свойств. Спецификация возвращаемого типа следует за последней круглой скобкой и указывает тип после двоеточия.
- *блок, содержащий тело функции.* При рассмотрении показанного ранее примера объявления функции может сначала показаться, что в ней отсутствует тип возвращаемого значения. Но это не так — функция имеет тип возвращаемого значения `Unit` (модуль). То есть функцию эту можно объявить и так:

```
fun printSum(a: Int, b: Int): Unit {
   print(a + b)
}
```

Впрочем, тип возвращаемого значения `Unit` может и не указываться явным образом.

Идентификатор для функции необязателен. Функции, не имеющие идентификатора, называются *анонимными*. Анонимные функции присутствуют в языке Kotlin в форме *лямбда-выражений*.

Вызов функций

Функции не выполняются сразу же после определения. Для выполнения кода функции нужно его вызвать. Функции можно вызывать как функции, как методы и непрямым образом (косвенно) — с помощью методов `invoke()` и `call()`. Вот пример прямого функционального вызова с использованием самой функции:

```
fun repeat(word: String, times: Int) {
   var i = 0

   while (i < times) {
      println(word)
      i++
   }
}

fun main(args: Array<String>) {
   repeat("Hello!", 5)
}
```

Скомпилируйте и запустите этот код. В результате слово **Hello** пять раз выводится на экране. Значение `Hello` передано как первое значение в функции, а число `5` — как второе значение. Аргументы `word` и `times` используются в функции `repeat` для хранения значений `Hello` и `5`. Цикл `while` запускается и выводит слово, пока `i` меньше указанного количества итераций. Счетчик `i++` применяется для увеличения значе-

ния переменной i на 1 — значение i увеличивается на единицу при каждой итерации цикла. Цикл прекращает работу, когда значение i равно 5. Поэтому слово Hello и будет напечатано пять раз. Компиляция и запуск программы приведут к выводу, показанному на рис. 1.7.

Другие методы вызова функций будут рассмотрены в книге далее.

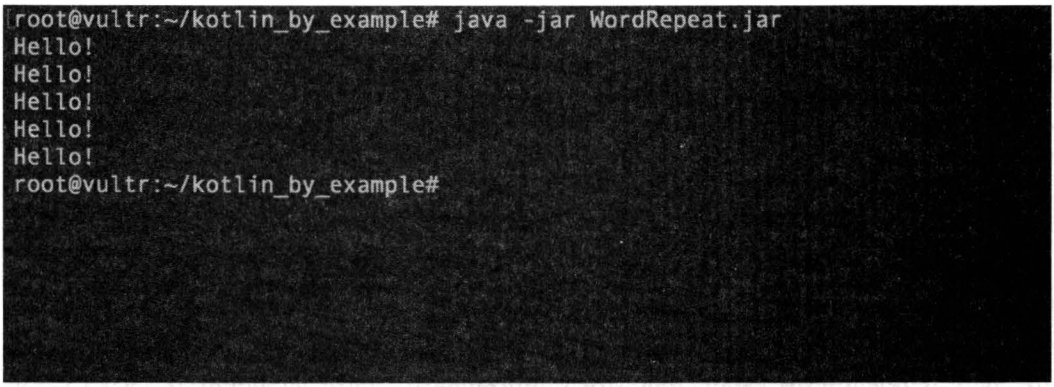


Рис. 1.7. Вывод примера прямого функционального вызова функции

Возвращаемые значения

Возвращаемое значение, как следует из его названия, — это значение, возвращаемое функцией. Обратите внимание, что функции в Kotlin могут возвращать значения при исполнении. Тип возвращаемого функцией значения определяется типом возврата функции. Это продемонстрировано в следующем фрагменте кода:

```
fun returnFullName(firstName: String, surname: String): String {
    return "${firstName} ${surname}"
}

fun main(args: Array<String>) {
    val fullName: String = returnFullName("James", "Cameron")
    println(fullName) // вывод: James Cameron
}
```

В этом коде функция returnFullName в качестве входных параметров принимает две разные строки и возвращает при вызове строковое значение. Тип возврата определен в заголовке функции. Возвращаемая строка создается с помощью шаблонов строк:

```
"${firstName} ${surname}"
```

Значения имени и фамилии интерполируются в строку символов.

Соглашение об именовании функций

Соглашения об именовании функций в языке Kotlin аналогичны тем, что применяются в языке Java. То есть при именовании методов используется «подход верблюда» — имена пишутся так, что каждое слово в имени начинается с заглавной буквы, без пробелов и знаков препинания:

```
// Хорошее имя функции
fun sayHello() {
   println("Hello")
}

// Плохое имя функции
fun say_hello() {
   println("Hello")
}
```

Комментарии

При написании кода иногда необходимо записывать важную информацию, относящуюся к написанному коду. Для этого применяются *комментарии*. В языке Kotlin имеется три типа комментариев:

- однострочные комментарии;
- многострочные комментарии;
- комментарии Doc.

Однострочные комментарии

Как следует из названия, эти комментарии занимают одну строку. Однострочные комментарии начинаются с двух обратных слешей: //. После компиляции программы все идущие после этих слешей символы игнорируются. Рассмотрим следующий код:

```
val b: Int = 957 // однострочный комментарий
// println(b)
```

Значение, сохраняющееся в переменной b, не выводится на консоль, поскольку функция, выполняющая операцию печати, закомментирована.

Многострочные комментарии

Многострочные комментарии занимают несколько строк. Они начинаются с обратного слеша, за которым следует звездочка: /*, и заканчиваются звездочкой, за которой следует обратный слеш: */:

```
/*
* Многострочный комментарий.
* И это тоже комментарий.
*/
```

Комментарии Doc

Этот тип комментария похож на многострочный комментарий. Основное различие их в том, что применяется он для документирования кода в программе. Комментарий doc (комментарий документа) начинается с обратного слеша, за которым следуют два символа звездочки: `/**`, и заканчивается звездочкой, за которой следует обратный слеш: `*/`:

```
/**
   * Adds an [item] to the queue.
   * @return the new size of the queue.
*/
fun enqueue(item: Object): Int { ... }
```

Управление потоком выполнения программы

При написании программ часто возникает необходимость управлять их выполнением. Так случается, если создаются программы, в которых решения принимаются в зависимости от условий и состояния данных. Язык Kotlin располагает для этого набором структур, известных пользователям, которые уже работали ранее с языками программирования, — например, конструкциями `if`, `while` и `for`. Имеются и другие конструкции, которые могут быть вам еще не знакомы. В этом разделе мы рассмотрим имеющиеся в нашем распоряжении структуры для управления потоком выполнения программы.

Условные выражения

Условные выражения используются для ветвления программы. Они выполняют или пропускают программные операторы на основе результатов условного теста. Условные выражения — это точки принятия решений в программе.

Язык Kotlin располагает для обработки ветвления двумя основными структурами: выражение `if` и выражение `when`.

Выражение *if*

Выражение `if` используется для принятия логического решения, основанного на выполнении условия. Для записи выражений `if` служит ключевое слово `if`:

```
val a = 1

if (a == 1) {
   print("a is one")
}
```

Здесь выражение `if` проверяет, выполняется ли условие: `a == 1` (читай: `a` равно `1`). Если условие истинно (`true`), строка `a` выводится на экран в виде отдельной строки, иначе — ничего не печатается.

Выражения `if` часто содержат одно или несколько сопутствующих слов `else` или `else if`. Эти сопутствующие ключевые слова могут применяться в дальнейшем для управления потоком выполнения программы. Рассмотрим, например, следующее выражение `if`:

```
val a = 4
if (a == 1) {
   print("a is equal to one.")
} else if (a == 2) {
   print("a is equal to two.")
} else {
   print("a is neither one nor two.")
}
```

Здесь сначала проверяется, равно ли `a` значению `1`. Этот тест оценивается как `false` (ложь), поэтому проверяется следующее условие. Конечно, `a` не равно и `2`. Следовательно, второе условие также оценивается как ложное. В результате того, что все предыдущие условия оцениваются как `false` (ложь), и выполняется последний оператор. Следовательно, на экран выводится сообщение: **a is neither one nor two.**

Выражение *when*

Выражение `when` служит еще одним средством управления потоком выполнения программ. Рассмотрим простой пример:

```
fun printEvenNumbers(numbers: Array<Int>) {
   numbers.forEach {
      when (it % 2) {
         0 -> println(it)
      }
   }
}

fun main (args: Array<String>) {
   val numberList: Array<Int> = arrayOf(1, 2, 3, 4, 5, 6)
   printEvenNumbers(numberList)
}
```

Здесь функция `printEvenNumbers` в качестве единственного аргумента принимает целочисленный массив. Массивы мы рассмотрим в этой главе позже, а пока представим их в виде последовательного набора значений из некого их пространства. В нашем случае переданный массив включает значения из пространства целых чисел. Каждый элемент массива входит в итерацию, обеспечиваемую методом `forEach`, и проверяется с помощью выражения `when`.

В нашем случае переменная `it` представляет собой текущее значение элемента массива, по отношению к которому выполняется итерация с помощью метода `forEach`. Оператор `%` является бинарным оператором, выполняющим операцию деления пер-

вого операнда (it) на значение второго операнда (2) с возвращением остатка от деления. Таким образом, выражение when на каждой итерации берет текущее значение первого операнда (it), делит его на 2, получает остаток от этого деления и проверяет, равно ли оно 0. Если это так, значение является четным, и, следовательно, выводится на печать.

Чтобы просмотреть, как работает программа, скопируйте ее код и вставьте в файл, затем скомпилируйте и запустите программу.

Оператор *Elvis*

Оператор Elvis — это сокращенная структура, имеющаяся в языке Kotlin. Она принимает следующую форму:

```
(выражение) ?: значение2
```

Применение этой структуры в программе Kotlin демонстрируется в следующем блоке кода:

```
val nullName: String? = null
val firstName = nullName ?: "John"
```

Если значение переменной nullName не нуль, оператор Elvis возвращает его, в противном случае возвращается строка "John". Таким образом, переменной firstName присваивается значение, возвращаемое оператором Elvis.

Циклы

Операторы цикла применяются с тем, чтобы набор операторов в блоке кода, помещенном в цикл, несколько раз при исполнении повторялся. Язык Kotlin предлагает следующие циклические конструкции — циклы for, while и do...while.

Цикл *for*

Цикл for в Kotlin выполняет итерацию (перебор) любых предоставленных ему объектов. Это похоже на выполнение цикла for..in в языке Ruby. Цикл for имеет следующий синтаксис:

```
for (объекты коллекции) { ... }
```

Блок в цикле for не нужен, если в цикле задействован только один оператор. *Коллекция* — это тип структуры, подлежащей перебору.

Рассмотрим следующую программу:

```
val numSet = arrayOf(1, 563, 23)

for (number in numSet) {
   println(number)
}
```

Каждое значение в массиве numSet перебирается в цикле и присваивается переменной number. После этого значение переменной number выводится в стандартное средство вывода системы.

Каждый элемент массива имеет *индекс* — позицию элемента в массиве. Множество индексов массива в языке Kotlin начинается с нуля.

Если вместо вывода на печать собственно числовых значений перебираемых в цикле чисел нужно вывести на печать индексы этих чисел, сделать это можно следующим образом:

```
for (index in numSet.indices) {
   println(index)
}
```

Можно также указать и тип переменной итератора:

```
for (number: Int in numSet) {
   println(number)
}
```

Цикл *while*

Цикл while выполняет инструкции внутри блока до тех пор, пока справедливо его условие. Создается он с помощью ключевого слова while. Цикл while имеет следующий синтаксис:

```
while (условие) { ... }
```

Как и в случае цикла for, наличие блока не является обязательным, если в пределах цикла находится только одно выражение. В цикле while операторы в блоке выполняются многократно, пока сохраняется указанное условие.

Рассмотрим следующий код:

```
val names = arrayOf("Jeffrey", "William", "Golding", "Segun", "Bob")
var i = 0

while (!names[i].equals("Segun")) {
   println("I am not Segun.")
   i++
}
```

Здесь блок кода в пределах цикла while выполняется до тех пор, пока не встречается имя Segun. При этом на печать выводится выражение: **I am not Segun.** Если встретится имя Segun, цикл завершится и ничего больше выведено не будет (рис. 1.8).

```
root@vultr:~/kotlin_by_example# kotlinc -script Printer.kts
I am not Segun.
I am not Segun.
I am not Segun.
root@vultr:~/kotlin_by_example#
```

Рис. 1.8. Вывод примера кода с циклом `while`

Ключевые слова *break* и *continue*

При объявлении циклов часто возникает необходимость либо выйти из цикла — если выполняется условие, — либо запустить внутри цикла в какой-нибудь момент времени следующую итерацию. Это можно сделать с помощью ключевых слов `break` и `continue`. Рассмотрим пример, иллюстрирующий этот подход. Откройте новый файл сценария Kotlin и скопируйте в него следующий код:

```
data class Student(val name: String, val age: Int, val school: String)

val prospectiveStudents: ArrayList<Student> = ArrayList()
val admittedStudents: ArrayList<Student> = ArrayList()

prospectiveStudents.add(Student("Daniel Martinez", 12, "Hogwarts"))
prospectiveStudents.add(Student("Jane Systrom", 22, "Harvard"))
prospectiveStudents.add(Student("Matthew Johnson", 22, "University of Maryland"))
prospectiveStudents.add(Student("Jide Sowade", 18, "University of Ibadan"))
prospectiveStudents.add(Student("Tom Hanks", 25, "Howard University"))

for (student in prospectiveStudents) {
   if (student.age < 16) {
      continue
   }
   admittedStudents.add(student)

   if (admittedStudents.size >= 3) {
      break
   }
}
println(admittedStudents)
```

Эта программа представляет собой упрощенный вариант утилиты для отбора из списка абитуриентов студентов, принятых на обучение. В начале кода создается класс данных для моделирования данных на каждого абитуриента, после чего создаются два списка массивов: один из них содержит информацию об абитуриентах — тех, кто подал заявку на поступление, а другой — информацию о студентах.

Следующие пять строк кода добавляют абитуриентов в список потенциальных студентов. Затем объявляется цикл, в котором выполняется перебор имен всех претендентов, присутствующих в списке абитуриентов. Если возраст претендента меньше 16 лет, цикл переходит к следующей итерации. Тем самым моделируется сценарий, при котором слишком молодых претендентов принять на обучение нельзя (т. е. нельзя внести в список принятых студентов). Если же претенденту уже исполнилось 16 лет, он добавляется в список допущенных к рассмотрению абитуриентов.

Затем выражение `if` используется для проверки, больше или равно трем число принятых студентов. Если условие равно истинно (`true`), программа выходит из цикла и дальнейшие итерации не выполняются. Последняя строка программы распечатывает имена абитуриентов, попавших в список студентов.

Запустите программу и посмотрите на полученный результат (рис. 1.9).

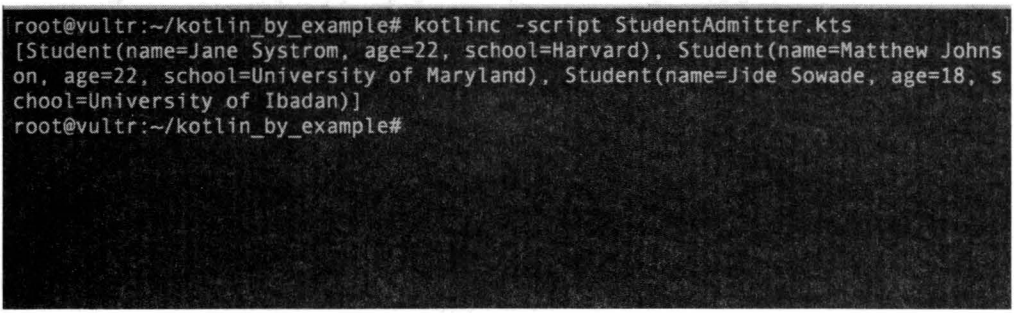


Рис. 1.9. Вывод примера кода с ключевыми словами `break` и `continue`

Цикл *do...while*

Цикл `do...while` аналогичен циклу `while`, за исключением того, что проверка условия цикла выполняется *после* первой итерации. Цикл `do...while` имеет следующий синтаксис:

```
do {
...
} while (условие)
```

Операторы внутри блока выполняются, если выполнено (`true`) проверочное условие:

```
var i = 0

do {
   println("I'm in here!")
   i++
} while (i < 10)

println("I'm out here!")
```

Обнуляемые значения

NullPointerException — с этим объектом наверняка сталкивались те, кто писал код на Java. Система типов языка Kotlin располагает защитой от нулевых значений — исключает появление в коде нулевых ссылок. То есть в языке Kotlin присутствуют обнуляемые типы и ненулевые типы — типы, которые могут иметь нулевое значение, и типы, которые не могут его иметь.

Для пояснения действия NullPointerException рассмотрим следующую программу Java:

```
class NullPointerExample {
   public static void main(String[] args) {
      String name = "James Gates";
      System.out.println(name.length()); // выводит 11
      name = null; // присваивает null переменной name
      System.out.println(name.length()); // вызывает NullPointerException
   }
}
```

Эта программа выполняет простую задачу вывода в стандартное системное устройство вывода длины строковой переменной. Но в ней присутствует одна проблема — если ее скомпилировать и запустить, будет сгенерировано исключение нулевого указателя и выполнение завершится на полпути (рис. 1.10).

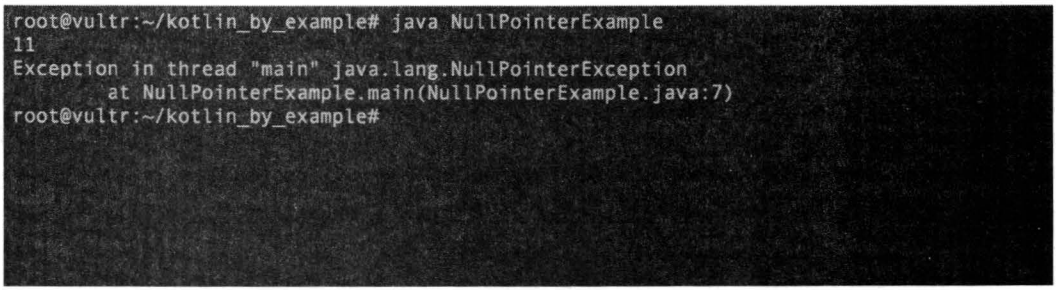


Рис. 1.10. Выполнение программы завершено на полпути

Можете ли вы определить причину появления NullPointerException (исключения нулевого указателя)? Исключение возникает, если метод String#length используется при нулевой ссылке. При этом программа прекращает выполнение и генерирует исключение. Понятно, что нежелательно, чтобы это в программах происходило.

При работе с языком Kotlin такой исход можно предотвратить, устранив возможность присвоения нулевого значения объекту name:

```
var name: String = "James Gates"
println(name.length)

name = null // не допускается присваивание значения null
println(name.length)
```

Как показано на рис. 1.11, система типов Kotlin приходит к выводу, что нулевое значение неправильно назначено переменной name, и предупреждает программиста об ошибке, которую необходимо исправить.

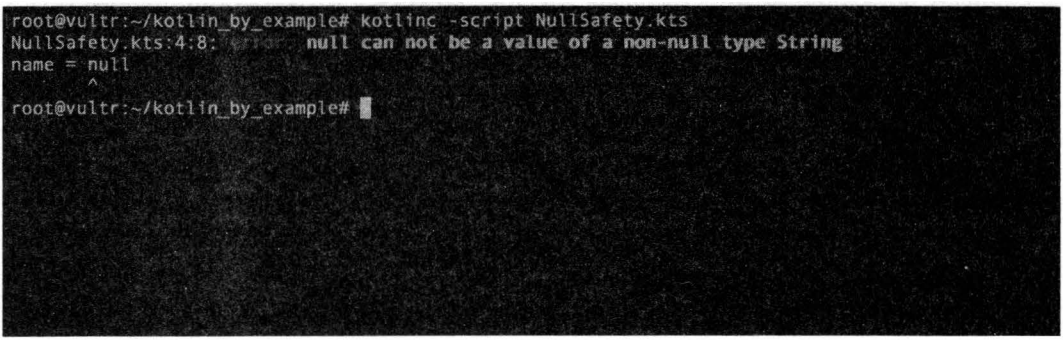


Рис. 1.11. Система типов Kotlin предупреждает программиста об ошибке, которую необходимо исправить

Любопытно, что произойдет, если в сценарии программист вынужден разрешить передачу нулевых значений? В этом случае программист просто объявляет переменную как *обнуляемую* путем добавления символа ?:

```
var name: String? = "James"
println(name.length)

name = null // разрешено присваивание значения null
println(name.length)
```

Независимо от того, что переменная объявлена обнуляемой, при запуске и этой программы мы также получим ошибку. Дело в том, что и к свойству length такой переменной следует обращаться безопасным способом. Это также можно сделать с помощью символа ?:

```
var name: String? = "James"
println(name?.length)

name = null // разрешено присваивание значения null
println(name?.length)
```

Здесь применен безопасный оператор ?., и программа теперь выполняется надлежащим образом. Вместо генерирования исключения нулевого указателя система типов распознает, что на нулевой указатель имеется ссылка, и предотвращает вызов метода length() для нулевого объекта. На рис. 1.12 показан безопасный для типа вывод.

Альтернативой применению безопасного оператора ?. служит использование оператора !! — он позволяет программе продолжать исполнение, и после вызова функции по нулевой ссылке генерирует исключение KotlinNullPointerException.

Эффект от замены ?. на !!. в приведенной ранее программе показан на рис. 1.13.

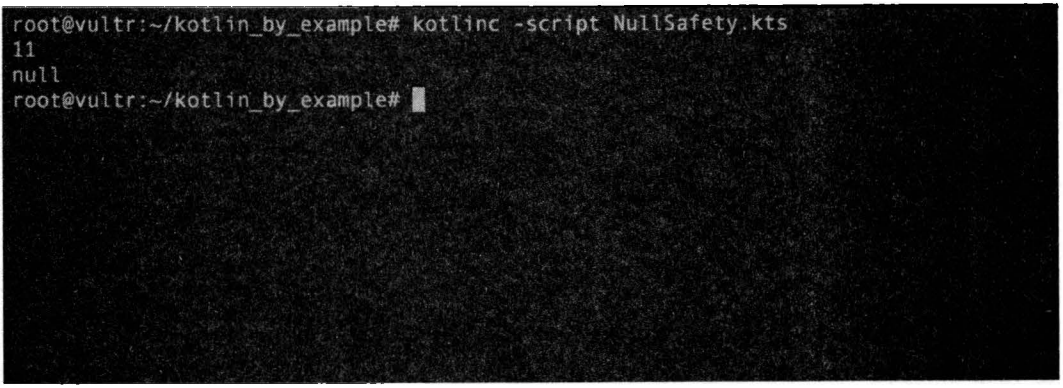


Рис. 1.12. Результат применения безопасного оператора `?.`

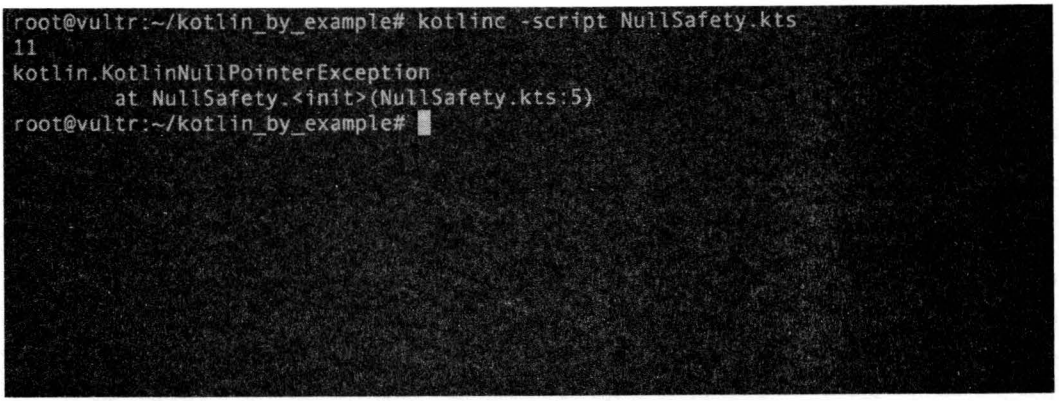


Рис. 1.13. В результате применения оператора `!!` генерируется исключение `KotlinNullPointerException`

Пакеты

Пакет представляет собой логическую группу связанных классов, интерфейсов, перечислений, аннотаций и функций. По мере увеличения количества исходных файлов их, вследствие различных причин, необходимо группировать в содержательные отдельные наборы — например, для удобства при обслуживании приложений, для предотвращения конфликтов имен и реализации контроля доступа.

Пакет создается с помощью ключевого слова `package`, за которым следует имя пакета:

```
package foo
```

В программном файле может быть только одна инструкция для пакета. Если пакет для файла программы не указан, содержимое файла помещается в пакет по умолчанию.

Ключевое слово *import*

Классы и типы часто используют другие классы и типы, находящиеся вне пакета, в котором они объявлены. Этого можно добиться путем импорта ресурсов пакета. Если два класса принадлежат одному пакету, импорт не требуется:

```
package animals
data class Buffalo(val mass: Int, val maxSpeed: Int, var isDead:
   Boolean = false)
```

В следующем фрагменте кода класс `Buffalo` не нужно импортировать в программу, поскольку он существует в том же пакете (animals), что и класс `Lion`:

```
package animals
class Lion(val mass: Int, val maxSpeed: Int) {
   fun kill(animal: Buffalo) { // для импорта используется тип Buffalo
      if (!animal.isDead) {
         println("Lion attacking animal.")
         animal.isDead = true
         println("Lion kill successful.")
      }
   }
}
```

Для импорта классов, функций, интерфейсов и типов в отдельных пакетах применяется ключевое слово `import`, за которым следует имя пакета. Например, следующая функция `main` существует в пакете по умолчанию. Таким образом, если необходимо в функции `main` использовать классы `Lion` и `Buffalo`, необходимо импортировать их с помощью ключевого слова `import`:

```
import animals.Buffalo
import animals.Lion

fun main(args: Array<String>) {
   val lion = Lion(190, 80)
   val buffalo = Buffalo(620, 60)
   println("Buffalo is dead: ${buffalo.isDead}")
   lion.kill(buffalo)
   println("Buffalo is dead: ${buffalo.isDead}")
}
```

Концепция объектно-ориентированного программирования

До сих пор классы использовались в ряде примеров, но их концепция не рассматривалась подробно. Этот раздел знакомит вас с основами классов, а также и с другими объектно-ориентированными конструкциями, доступными в языке Kotlin.

Базовые понятия

В начале формирования языков программирования высокого уровня программы создавались в виде наборов процедур. И доступные языки программирования носили, в основном, процедурный характер. *Процедурный язык программирования* — это язык, где для составления программ применяется серия структурированных, четко определенных шагов.

По мере развития индустрии программного обеспечения программы становились все крупнее, и возникла необходимость в разработке более совершенного подхода к их разработке. Это привело к появлению *объектно-ориентированных языков программирования*.

Модель объектно-ориентированного программирования — это модель, сосредоточенная на объектах и данных, а не на действиях и последовательной логике. В объектно-ориентированном программировании объекты, классы и интерфейсы составляются, расширяются и наследуются с целью создания программного обеспечения промышленного уровня.

Класс — это модифицируемый и расширяемый шаблон программы для создания объектов и поддержания ее состояния посредством использования переменных, констант и свойств. Класс обладает характеристиками и поведением: *характеристики* представлены как переменные, а *поведение* реализовано в форме методов.

Методы — это функции, специфичные для класса или набора классов. Классы имеют возможность наследовать характеристики и поведение от других классов. Эта способность называется *наследованием*.

Язык Kotlin является объектно-ориентированным языком программирования и, следовательно, поддерживает все функции объектно-ориентированного программирования. В Kotlin, подобно Java и Ruby, разрешено только *единственное наследование*. Некоторые языки, такие как C++, поддерживают множественное наследование. Недостаток множественного наследования в том, что оно приводит к проблемам управления, — таким как одноименные коллизии (коллизии одинаковых имен).

Класс, наследуемый от другого класса, называется *подклассом*, а класс, от которого он наследуется, является *суперклассом*.

Интерфейс — это структура, обеспечивающая определенные характеристики и поведение в классах. Поведенческое принуждение через интерфейсы может быть внедрено путем реализации интерфейса в классе. Подобно классам, интерфейс может расширять другой интерфейс.

Объект — является экземпляром класса, который может обладать собственным уникальным состоянием.

Работа с классами

Класс объявляется с помощью ключевого слова `class`, за которым следует имя класса:

```
class Person
```

Как показано в приведенном примере, класс в Kotlin не должен иметь тела. Хотя это и прекрасно, однако достаточно часто бывает необходимо, чтобы у класса имелись характеристики и поведение, находящиеся внутри тела. Этого можно добиться, вводя открывающие и закрывающие скобки:

```
class HelloPrinter {
   fun printHello() {
      println("Hello!")
   }
}
```

В приведенном фрагменте кода имеется класс `HelloPrinter` с объявленной в нем единственной функцией. Функция, объявленная в классе, называется *методом*. Методы также могут называться *поведением*. Как только метод объявлен, он может использоваться всеми экземплярами класса.

Создание объектов

Объявление экземпляра класса или объекта аналогично объявлению переменной. Мы можем создать экземпляр класса `HelloPrinter`, как показано в следующем коде:

```
val printer = HelloPrinter()
```

Здесь `printer` является экземпляром класса `HelloPrinter`. Открывающие и закрывающие скобки после имени класса `HelloPrinter` служат для вызова основного конструктора класса `HelloPrinter`. *Конструктор* похож на функцию — это и есть специальная функция, которая инициализирует объект типа.

Функция, объявленная в классе `HelloPrinter`, может вызываться непосредственно объектом `printer` в любой момент времени:

```
printer.printHello() // выводит hello
```

Иногда требуется, чтобы функция вызывалась напрямую классом без необходимости создания объекта. Это может быть сделано с использованием сопутствующего объекта.

Сопутствующие объекты

Сопутствующие объекты объявляются в классе с использованием ключевых слов `companion` и `object`. Внутри сопутствующего объекта можно использовать статические функции:

```
class Printer {
   companion object DocumentPrinter {
      fun printDocument() = println("Document printing successful.")
   }
}
fun main(args: Array<String>) {
   Printer.printDocument() // printDocument() вызывается через
                           // сопутствующий объект
   Printer.Companion.printDocument() // также printDocument()
                                     // вызывает сопутствующий объект
}
```

Иногда необходимо указывать идентификатор сопуствующего объекта. Для этого имя помещается после ключевого слова `object`:

```
class Printer {
   companion object DocumentPrinter { // Сопутствующий объект
                                      // идентифицируется DocumentPrinter
   fun printDocument() = println("Document printing successful.")
   }
}
fun main(args: Array<String>) {
   Printer.DocumentPrinter.printDocument() // printDocument() вызывается
                                   // именованным сопутствующим объектом
}
```

Свойства класса

Классы могут иметь *свойства*, которые объявляются с помощью ключевых слов `var` и `val`. Например, в следующем фрагменте кода класс `Person` имеет три свойства: `age`, `firstName` и `surname`:

```
class Person { var age = 0
   var firstName = ""
   var surname = ""
}
```

К свойствам можно получить доступ с помощью экземпляра класса, в котором сохраняется свойство. Для этого к идентификатору экземпляра добавляется один символ точки (.), за которым следует имя свойства. Например, в следующем фрагменте кода создан экземпляр с именем `person` класса `Person`, а свойства `firstName`, `surname` и `age` назначаются путем доступа к свойствам:

```
val person = Person()
person.firstName = "Raven"
person.surname = "Spacey"
person.age = 35
```

Преимущества языка Kotlin

Как отмечалось ранее, язык Kotlin разработан для улучшения языка Java и поэтому имеет в сравнении с ним ряд преимуществ:

- *нулевая безопасность* — в программах Java часто генерируется исключение `NullPointerException`. Kotlin решает эту проблему, предоставляя нуль-безопасную систему типов;
- *наличие функций расширения* — функции легко добавляются в определенные в программных файлах классы для расширения их функциональности. В Kotlin это можно сделать с помощью функций расширения;
- *синглетоны* — в программах Kotlin легко реализовать шаблон синглетона. Реализация синглетона в Java требует значительно бóльших усилий, чем в Kotlin;
- *классы данных*. При написании программ обычно создают класс с единственной целью — сохранить данные в переменных. Множество строк кода занимает выполнение столь обыденной задачи. Классы данных в Kotlin позволяют легко создавать подобные классы, включающие данные в одной строке кода;
- *типы функций* — в отличие от языка Java, Kotlin имеет типы функций. Это позволяет функциям принимать в качестве параметров другие функции и определять функции, которые возвращают функции.

Разработка с помощью Kotlin приложений для Android

Итак, к настоящему моменту вкратце рассмотрены некоторые функции, предоставленные в наше распоряжение языком Kotlin для разработки универсальных приложений. Далее в этой книге мы разберемся, каким образом эти функции могут применяться при разработке приложений для Android, — именно в этой области язык Kotlin имеет непререкаемый авторитет.

Но прежде чем продолжить, необходимо настроить систему для выполнения поставленной задачи. Основным требованием для разработки приложений для Android является наличие подходящей интегрированной среды разработки (IDE) — это даже не требование, а путь для упрощения процесса разработки. Из большого числа вариантов IDE наиболее популярными среди разработчиков Android считаются:

- Android Studio;
- Eclipse;
- IntelliJ IDE.

Среда Android Studio на сегодня является наиболее универсальной из доступных сред разработки для Android. Как следствие, именно на эту IDE мы и будем ориентироваться во всех главах книги, связанных с Android.

Установка Android Studio

На момент подготовки этой книги версией Android Studio, поставляемой в комплекте с полной поддержкой Kotlin, была Android Studio 3.0. Ее «канарская» версия (canary version) может быть загружена на сайте по адресу: **https://developer.android.com/studio/preview/index.html**. После загрузки откройте загруженный пакет или исполняемый файл и следуйте инструкциям по установке. Воспользуйтесь мастером настройки, который проведет вас через процедуру настройки IDE (рис. 1.14).

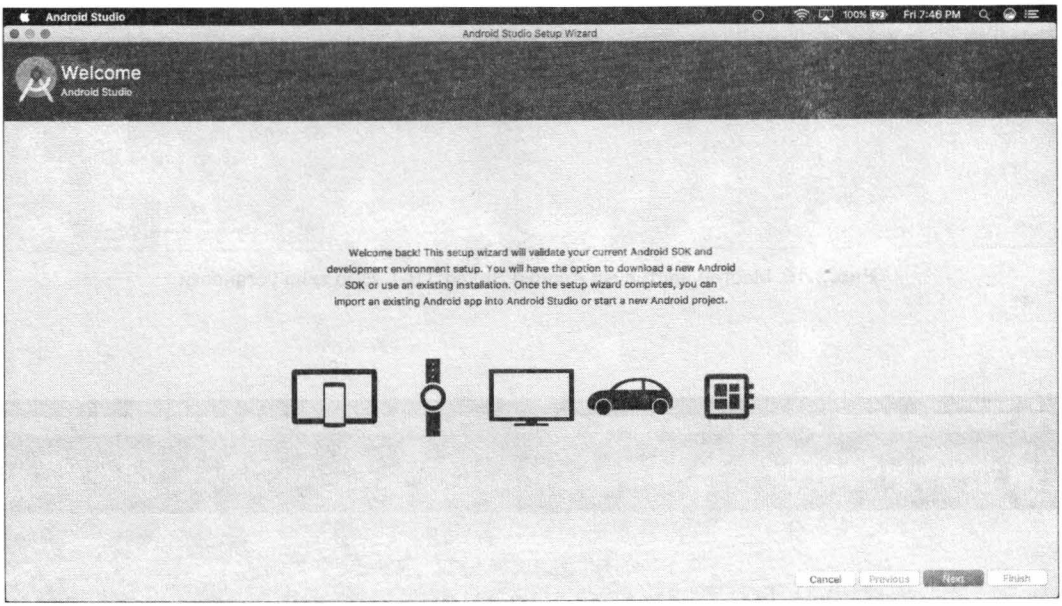


Рис. 1.14. Мастер настройки Android Studio 3.0: шаг первый

При переходе к следующему окну настройки вам будет предложено выбрать необходимый тип установки Android Studio (рис. 1.15).

Выберите установку **Standard** (Стандартная), нажмите на кнопку **Next** (Дальше), и вы перейдете к следующему окну. В окне **Verify Settings** (Проверить настройки) щелкните на кнопке **Finish** (Готово) — Android Studio загрузит необходимые компоненты. Вам нужно подождать несколько минут, пока загрузка не будет завершена (рис. 1.16).

Щелкните на кнопке **Finish** после завершения загрузки компонентов — перед вами откроется стартовое окно Android Studio (рис. 1.17). Теперь вы готовы работать с этой IDE.

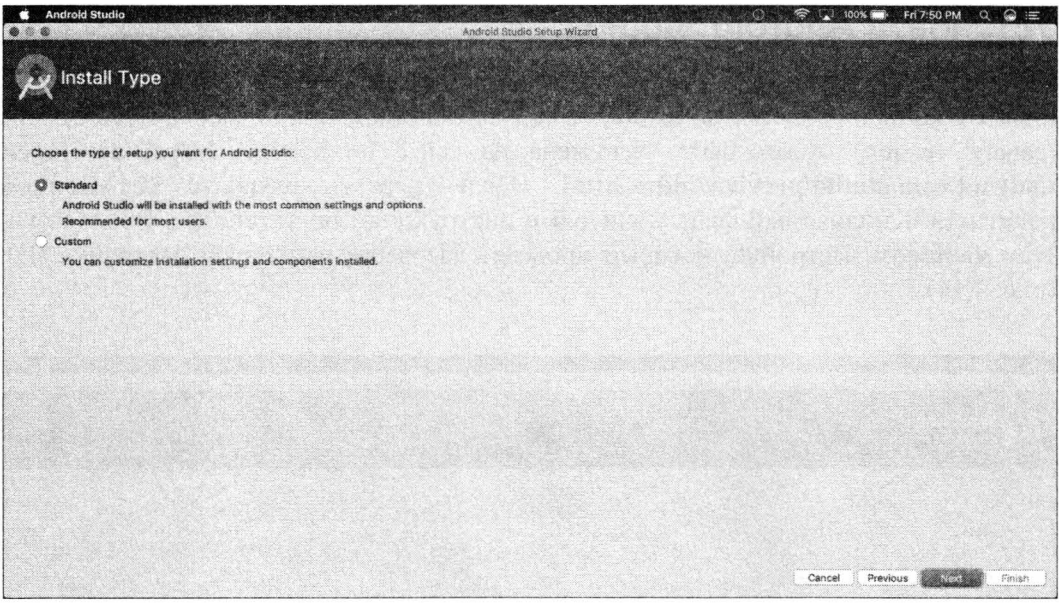


Рис. 1.15. Мастер настройки Android Studio 3.0: выбор типа установки

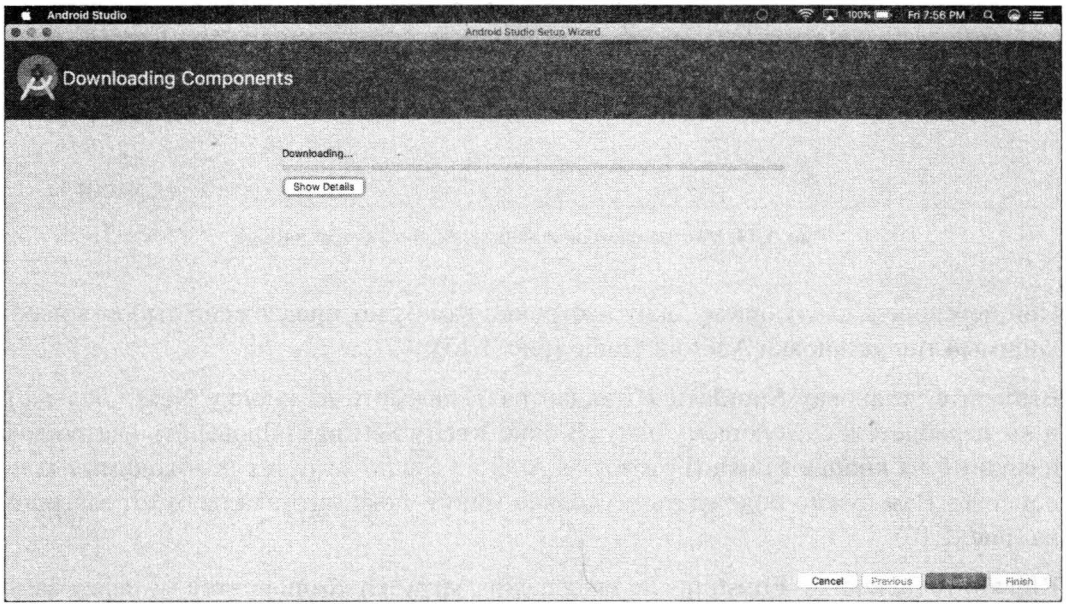


Рис. 1.16. Мастер настройки Android Studio 3.0: загрузка необходимых компонентов

Рис. 1.17. Android Studio 3.0: стартовое окно

Создание первого приложения для Android

Создадим с помощью Android Studio простое приложение для Android. Назовем его `HelloApp`. Это приложение по нажатию кнопки будет отображать на экране приветствие: **Hello world!**.

В стартовом окне Android Studio щелкните на кнопке **Start a new Android Studio project** (Начать новый проект Android Studio). Откроется окно, где нужно указать некоторые детали, касающиеся будущего приложения, — например название приложения, домен вашей компании и местоположение проекта.

Введите `HelloApp` в качестве имени приложения и укажите домен компании. Если доменное имя компании отсутствует, укажите в поле ввода домена компании любое допустимое доменное имя. Поскольку это тренировочный проект, указывать реальное доменное имя не обязательно. Укажите местоположение, где необходимо сохранить этот проект, установите флажок для включения поддержки Kotlin.

После заполнения необходимых параметров нажмите на кнопку **Next**, и вы перейдете к следующему окну (рис. 1.18).

Здесь следует указать целевые устройства. Поскольку приложение создается для запуска на смартфонах, установите флажок **Phone and Tablet** (Телефон и планшет), если он еще не установлен. Рядом с каждым предусмотренным на рис. 1.18 устройством имеются раскрывающиеся списки целевых уровней API для создаваемых проектов. *Уровень API* — это целое число, которое однозначно идентифицирует

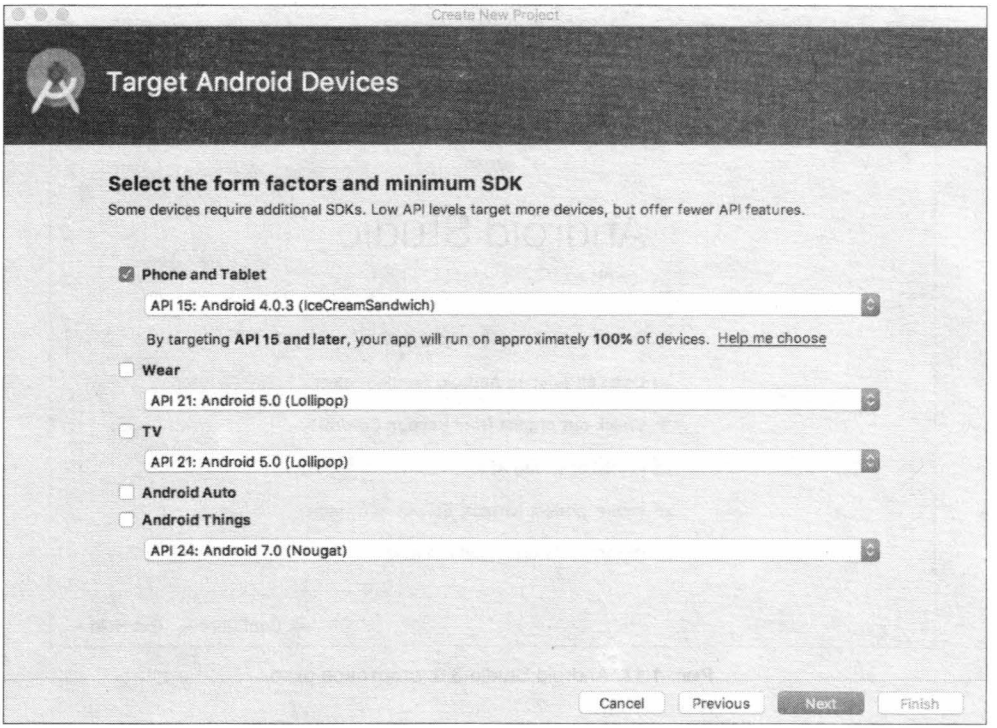


Рис. 1.18. Android Studio 3.0: окно указания целевых устройств и целевого уровня API для создаваемого проекта

раздел API, предлагаемый версией платформы Android. Выберите для устройства **Phone and Tablet** уровень **API 15**, если он еще не выбран, нажмите на кнопку **Next**, и вы перейдете к следующему окну (рис. 1.19).

Здесь следует выбрать действие, которое необходимо добавить в наше приложение. *Действие* — это отдельный экран с уникальным пользовательским интерфейсом, похожим на окно. Мы обсудим действия более подробно в *главе 2*, посвященной созданию приложения Tetris для Android. Сейчас же выберите **Empty Activity** (Пустое действие), нажмите на кнопку **Next**, и вы перейдете к следующему окну (рис. 1.20).

Здесь необходимо сконфигурировать только что указанное нами действие, чтобы оно было создано. Назовите действие `HelloActivity` и убедитесь, что установлены флажки **Generate Layout File** (Генерировать структуру файла) и **Backwards Compatibility** (Обратная совместимость). Затем щелкните на кнопке **Finish** — Android Studio настроит ваш проект через несколько минут.

После завершения установки вы увидите окно IDE, содержащее файлы вашего проекта.

Ошибки, связанные с отсутствием необходимых компонентов проекта, могут возникать на любом этапе разработки проекта. Недостающие компоненты можно загрузить с помощью менеджера SDK.

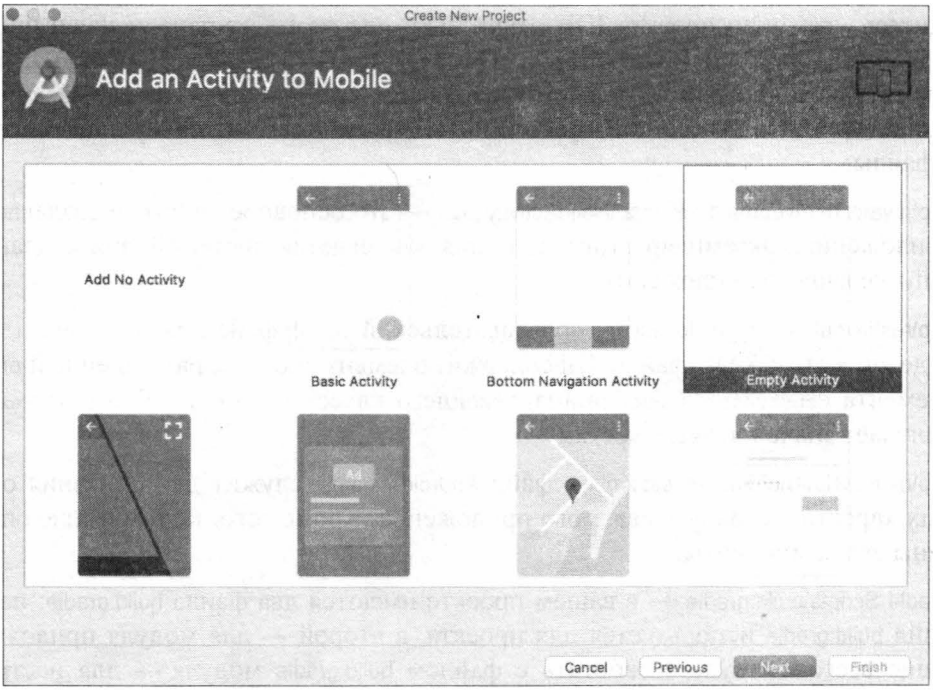


Рис. 1.19. Android Studio 3.0: окно выбора действий для создаваемого проекта

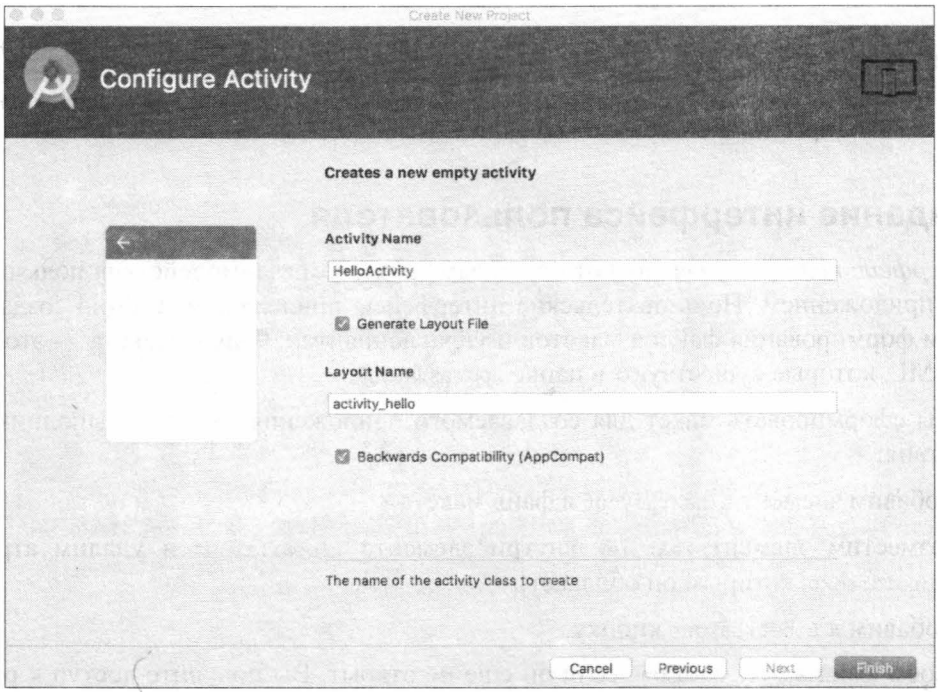


Рис. 1.20. Android Studio 3.0: окно конфигурирования действия для создаваемого проекта

Убедитесь, что окно проекта IDE открыто — для этого на панели навигации нажмите **View | Tool Windows | Project** (Вид | Окна инструментов | Проект), а также, что представление Android в настоящее время выбирается из раскрывающегося списка в верхней части окна **Project**. В левой части этого окна вы увидите следующие файлы:

- app\java\com.mydomain.helloapp\HelloActivity.java — это основное действие создаваемого приложения. Экземпляр этого действия запускается системой при создании и запуске вашего приложения;
- app\res\layout\activity_hello.xml — пользовательский интерфейс для `HelloActivity` определен в этом XML-файле. Он содержит элемент `TextView`, размещенный внутри элемента `ConstraintLayout`, принадлежащего классу `ViewGroup`. Текст из `TextView` получает значение `Hello World!`;
- app\manifests\AndroidManifest.xml — файл AndroidManifest служит для описания основных характеристик создаваемого приложения. Кроме того, в этом файле определены его компоненты;
- Gradle Scripts\build.gradle — в вашем проекте имеются два файла build.gradle: первый файл build.gradle используется для проекта, а второй — для модуля приложения. Чаще всего вы будете работать с файлом build.gradle модуля — для настройки процедуры компиляции инструментов Gradle и формирования вашего приложения.

Gradle — это система автоматизации сборки с открытым исходным кодом, используемая для объявления конфигураций проекта. В Android Gradle применяется в качестве инструмента для формирования приложений, для создания пакетов и управления зависимостями приложений.

Создание интерфейса пользователя

Интерфейс пользователя служит основным средством взаимодействия пользователя с приложением. Пользовательские интерфейсы приложений Android создаются путем формирования файлов макетов и управления ими. Файлы макета — это файлы XML, которые существуют в папке app\res\layout\.

Чтобы сформировать макет для создаваемого приложения `HelloApp`, выполним три действия:

1. Добавим элемент `LinearLayout` в файл макета.
2. Разместим элемент `TextView` внутри элемента `LinearLayout` и удалим атрибут `android:text`, которым он обладает.
3. Добавим к `LinearLayout` кнопку.

Откроем файл activity_hello.xml, если он еще не открыт. Вы получите доступ к редактору макета. Если редактор находится в представлении **Design** (Дизайн), измените

его на представление **Text** (Текст), переключая соответствующие кнопки в нижней части окна редактора макета. Редактор макетов должен иметь примерно такой вид, как на рис. 1.21.

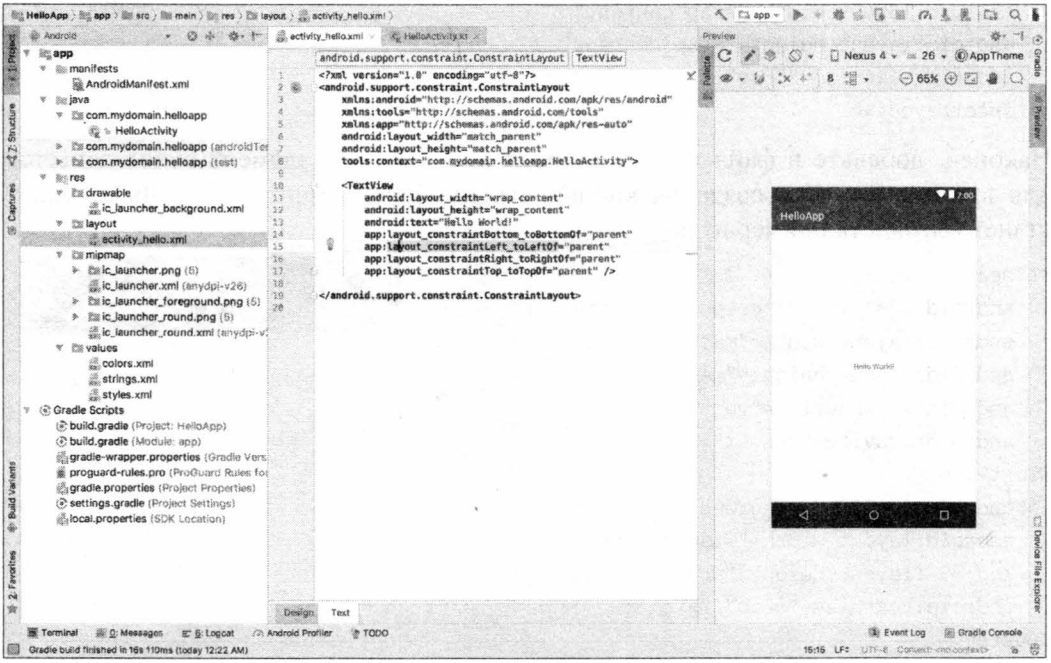


Рис. 1.21. Редактор макетов

Элемент линейной разметки `LinearLayout` представляет собой объект класса `ViewGroup` и размещает дочерние представления в горизонтальном или вертикальном порядке в пределах одного столбца. Скопируйте фрагмент кода, необходимого `LinearLayout`, из приведенного далее блока и вставьте его в элемент `ConstraintLayout`, который предшествует элементу `TextView`:

```
<LinearLayout
   android:id="@+id/ll_component_container"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:gravity="center">
</LinearLayout>
```

Теперь скопируйте и вставьте элемент `TextView`, представленный в файле **activity_hello.xml**, в тело элемента `LinearLayout` и удалите атрибут `android:text`:

```
<LinearLayout
   android:id="@+id/ll_component_container"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
```

```
   android:orientation="vertical"
   android:gravity="center">
<TextView
   android:id="@+id/tv_greeting"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:textSize="50sp" />
</LinearLayout>
```

Наконец, добавьте в файл макета элемент кнопки. Этот элемент будет дочерним для `LinearLayout`. Для создания кнопки используйте элемент `Button`. Файл макета станет выглядеть примерно так:

```
<LinearLayout
   android:id="@+id/ll_component_container"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:gravity="center">
<TextView
   android:id="@+id/tv_greeting"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:textSize="50sp" />
<Button
   android:id="@+id/btn_click_me"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginTop="16dp"
   android:text="Click me!"/>
</LinearLayout>
```

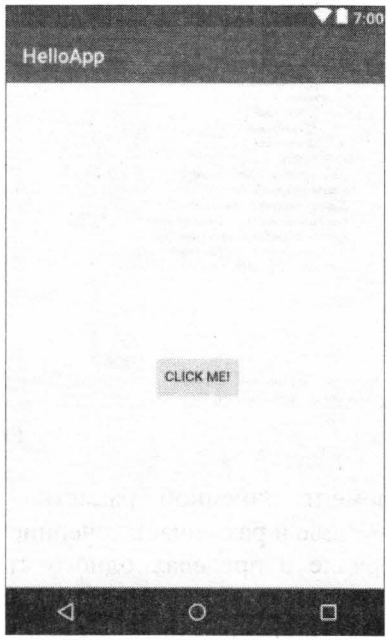


Рис. 1.22. Пользовательский интерфейс создаваемого приложения

Переключитесь в редакторе макетов на представление **Design**, чтобы увидеть, каким образом внесенные изменения отражены при визуализации в пользовательском интерфейсе (рис. 1.22).

Макет мы создали, но и получили проблемы — кнопка **CLICK ME!** при нажатии не активируется. Исправим это, добавляя в кнопку слушатель событий щелчка на кнопке. Найдите и откройте файл HelloActivity.java, отредактируйте функцию, добавляющую логику для события щелчка на кнопке **CLICK ME!**, а также импортирующую необходимый пакет. Все это выполняет следующий код:

```
package com.mydomain.helloapp
import android.support.v7.app.AppCompatActivity
```

```
import android.os.Bundle
import android.text.TextUtils
import android.widget.Button
import android.widget.TextView
import android.widget.Toast
class HelloActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_hello)
      val tvGreeting = findViewById<TextView>(R.id.tv_greeting)
      val btnClickMe = findViewById<Button>(R.id.btn_click_me)
      btnClickMe.setOnClickListener {
         if (TextUtils.isEmpty(tvGreeting.text)) {
            tvGreeting.text = "Hello World!"
         } else {
            Toast.makeText(this, "I have been clicked!",
               Toast.LENGTH_LONG).show()
         }
      }
   }
}
```

В этом фрагменте кода с помощью функции `findViewById` добавлены ссылки на элементы `TextView` и `Button`, имеющиеся в файле макета `activity_hello`. Функция `findViewById` может применяться для получения ссылок на элементы макета, которые находятся в текущем заданном представлении контента. Во второй строке функции `onCreate` для представления содержимого `HelloActivity` задан макет `activity_hello.xml`.

Рядом с идентификатором функции `findViewById` имеется запись `TextView`, заключенная в угловые скобки. Подобный подход называется *универсальной функцией*. Он используется для обеспечения того, чтобы идентификатор ресурса, передаваемый в `findViewById`, принадлежал элементу `TextView`.

После добавления ссылочных объектов устанавливается значение `onClickListener` равным `btnClickMe`. Слушатели используются для прослушивания событий в приложении. Чтобы выполнить действие при щелчке на элементе, передадим в метод `setOnClickListener` лямбда-выражение, содержащее действие, которое необходимо выполнить.

После щелчка на `btnClickMe` проверяется элемент `tvGreeting`, что позволяет видеть, был ли он установлен для включения какого-либо текста. Если для `TextView` текст не установлен, то соответствующий текст установлен как `Hello World!`, в противном случае отображается текст `I have been clicked!`.

Запуск приложения

Для запуска приложения щелкните на кнопке **Run 'app'** (Выполнить приложение) в правой верхней части окна IDE (рис. 1.23) или нажмите комбинацию клавиш <Ctrl>+<R> и выберите цель развертывания. Приложение `HelloApp` будет сформировано, установлено и запущено для цели развертывания.

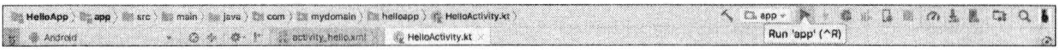


Рис. 1.23. Кнопка **Run 'app'**

Для использования в качестве цели развертывания можно применить одно из доступных в системе виртуальных устройств или создать собственное виртуальное устройство. Можно также подключить к компьютеру через USB физическое устройство Android и выбрать его в качестве цели. Окончательный выбор остается за вами. Выбрав устройство для развертывания, нажмите кнопку **OK** для создания и запуска приложения.

После запуска приложения на экране устройства отобразится созданный макет (рис. 1.24).

А по щелчку на кнопке **CLICK ME!** на экране устройства появляется приветствие: **Hello World!** (рис. 1.25).

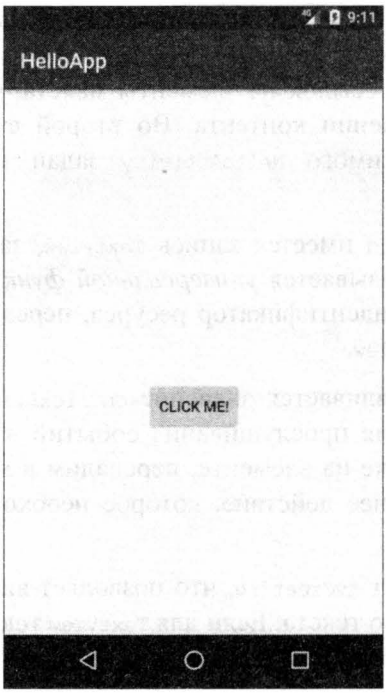


Рис. 1.24. Созданный макет отображен на экране устройства

Рис. 1.25. По щелчку на кнопке **CLICK ME!** на экране устройства появляется приветствие: **Hello World!**

Рис. 1.26. При последующих щелчках на кнопке **CLICK ME!** на экране устройства отображается сообщение: **I have been clicked!**

При последующих щелчках на кнопке **CLICK ME!** на экране устройства отображается сообщение: **I have been clicked!** (Я нажал!) (рис. 1.26).

Основы работы в Интернете

Большинство приложений, установленных на устройствах, взаимодействуют с серверами. Крайне важно, чтобы вы вникли в ряд связанных с сетью концепций, а уже затем продолжили чтение книги. В этом разделе все понятия излагаются вкратце.

Что такое сеть?

Сеть представляет собой сложную систему взаимосвязанных систем, обладающих способностью связываться с другими системами из общей сети по одному или нескольким протоколам. *Протокол* — это четко определенная официальная система правил, регулирующих обмен информацией между устройствами.

Протокол передачи гипертекста

Все коммуникации через Интернет осуществляются в соответствии с протоколами. Особенно важным протоколом для обеспечения связи между системами является *протокол передачи гипертекста* (Hypertext Transfer Protocol, HTTP).

Через Интернет ежедневно передаются миллиарды файлов изображений, видео, текстовых сообщений, документов и т. п. Все эти файлы передаются с помощью протокола HTTP. Это прикладной протокол, предназначенный для распределенных и гипермедийных информационных систем. Можно сказать, что этот протокол служит основополагающим компонентом для общения через Интернет. Основным преимуществом применения HTTP для передачи данных по системам служит его высокая стабильность. Это гарантируется за счет использования надежных протоколов — таких как *протокол управления передачами* (Transmission Control Protocol, TCP) и *протокол интернета* (Internet Protocol, IP).

Клиенты и серверы

Веб-клиентом является любое приложение, которое взаимодействует с веб-сервером, использующим HTTP. *Веб-сервер* — это компьютер, предоставляющий или обслуживающий веб-ресурсы для веб-клиентов. *Веб-ресурс* — это всё, что предоставляет веб-контент (содержимое). Веб-ресурсом может быть медиафайл, HTML-документ, шлюз и т. п. Клиентам нужен веб-контент для различных целей, таких как рендеринг информации и манипулирование данными.

Клиенты и серверы общаются друг с другом через протокол HTTP. Одна из основных причин использования HTTP состоит в том, что он чрезвычайно надежен при передаче данных. Использование HTTP гарантирует, что потери данных в цикле запрос/ответ не произойдет.

Запросы и отклики HTTP

HTTP-запрос, как следует из названия, — это запрос веб-ресурса, отправляемый веб-клиентом на сервер через HTTP. Отклик HTTP — это ответ, отправленный сервером, на запрос в HTTP-транзакции (рис. 1.27).)

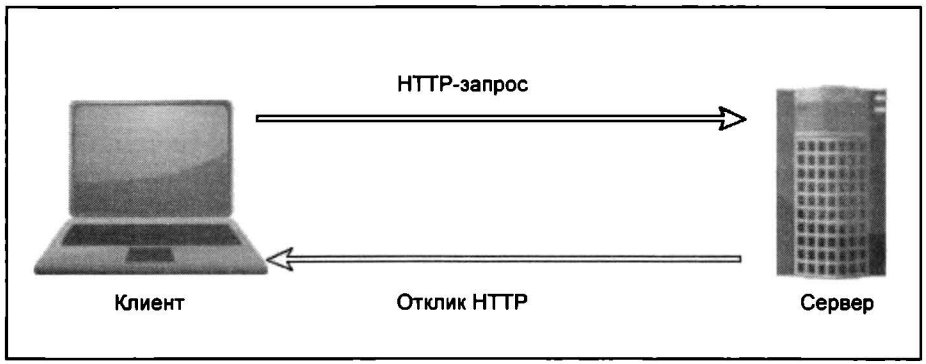


Рис. 1.27. Запросы и отклики HTTP

Методы HTTP

В HTTP поддерживаются несколько методов запроса. Эти методы также называются *командами*. Методы HTTP указывают тип действия, которое должно выполняться сервером. Некоторые распространенные методы HTTP представлены в табл. 1.4.

***Таблица 1.4.** Некоторые распространенные методы HTTP*

HTTP-метод	Описание
GET	Получает именованный ресурс, присутствующий на клиенте
POST	Отправляет данные с клиента на сервер
DELETE	Удаляет находящийся на сервере именованный ресурс
PUT	Сохраняет собранные клиентом данные в именованном ресурсе, находящемся на сервере
OPTIONS	Возвращает методы HTTP, которые поддерживает сервер
HEAD	Получает заголовки HTTP без содержимого

Подведем итоги

В этой главе вы познакомились с языком Kotlin и изучили его основы. Вы узнали, как установить язык Kotlin на компьютер, что такое IDE, какая IDE используется для написания программ на Kotlin, как создавать и запускать сценарии Kotlin и как задействовать REPL. Кроме того, мы рассмотрели работу с IntelliJ IDEA и Android Studio, после чего реализовали простое приложение для Android. Наконец, вам были представлены фундаментальные концепции, связанные с сетью.

В следующей главе в процессе создания приложения для Android мы познакомимся с написанием программ на Kotlin. Рассмотрим архитектуру приложения Android и важные компоненты приложения Android, а также и другие темы.

2

Создание Android-приложения Tetris

В предыдущей главе были кратко рассмотрены важные темы, относящиеся к ядру языка Kotlin. Мы познакомились с основами этого языка, а также с универсальным подходом к разработке программного обеспечения, который обеспечивает объектно-ориентированное программирование. А сейчас мы применим полученные знания, непосредственно приступив к разработке приложения для Android.

В этой главе рассматриваются следующие темы:

- компоненты приложения Android;
- представления;
- группы представлений;
- ограничения макетов;
- реализация макетов с помощью XML;
- строковые и размерные ресурсы;
- обработка событий ввода данных.

Знакомство с указанными темами мы организуем на основе практического подхода — в процессе реализации макетов и компонентов классической игры Tetris. Поскольку игра разрабатывается как приложение для Android, сначала необходимо привести краткий обзор этой операционной системы.

Знакомство с Android

Android — это мобильная операционная система на основе Linux, которая разработана и поддерживается Google. Эта система предназначена в основном для поддержки интеллектуальных мобильных устройств, таких как смартфоны и планшеты. Взаимодействие пользователей с операционной системой Android реализуется

на основе *графического интерфейса пользователя* (Graphic User Interface, GUI). Пользователи устройств на базе Android взаимодействуют с операционной системой, прежде всего, через визуальный сенсорный интерфейс, выполняя такие жесты, как касания и смахивания.

Программы устанавливаются на устройства с ОС Android в виде *приложений*, которые выполняются в среде операционной системы и решают одну или несколько задач для достижения цели или ряда целей. После установки приложений на мобильные устройства огромные возможности получают как пользователи, так и разработчики приложений. Пользователи обращаются к предоставляемым приложениями возможностям для решения своих повседневных задач, а разработчики на волне спроса на такие программы создают приложения, которые соответствуют потребностям пользователей и, возможно, приносят прибыль.

Разработчики приложений для Android используют широкий спектр инструментов и утилит для создания высокопроизводительных приложений, относящихся к самым разным сферам деятельности, таким как отдых, бизнес и электронная коммерция. Существуют также и игровые приложения.

В этой главе мы более подробно рассмотрим ряд инструментов и утилит, предоставляемых платформой приложений Android.

Компоненты приложения Android

Платформа приложений Android предоставляет разработчикам ряд компонентов, которые можно задействовать для создания пользовательского интерфейса приложений в целом и приложения Tetris в частности. *Компонент Android* — это многократно используемый шаблон программы или объекта, с помощью которого можно определить те или иные возможности приложения.

Некоторые важные компоненты, предоставляемые платформой приложений Android, приведены в следующем списке:

- действия (Activity);
- намерения (Intents);
- фильтры намерений (Intent filters);
- фрагменты (Fragments);
- службы (Services);
- загрузчики (Loaders);
- провайдеры контента (Content providers).

Действия

Действие — это компонент Android, который играет центральную роль в реализации потока приложения и взаимодействия между его компонентами. Действия реа-

лизуются в форме классов. Экземпляр действия используется системой Android для инициации кода.

Действие важно при создании пользовательских интерфейсов приложений. Оно предоставляет окно, в котором рисуются элементы пользовательского интерфейса. Проще говоря, с помощью действий создаются экраны приложений.

Намерения

Намерение облегчает взаимосвязь между действиями. По сути, намерения играют в приложении Android роль мессенджеров. Они служат объектами обмена сообщениями, посылаемыми для запроса действий от компонентов приложения. Намерения могут использоваться для выполнения таких действий, как запрос начала действия и передача в среде Android широковещательных сообщений.

Имеется два следующих типа намерений:

- *неявные намерения* — это объекты-мессенджеры, которые конкретно не определяют компонент приложения для выполнения действия, но указывают действие, которое должно выполняться, и позволяют компоненту, который может находиться в другом приложении, выполнить действие. Компоненты, которые могут неявно обрабатывать запрашиваемое действие, определяются системой Android;
- *явные намерения* — эти намерения явно указывают компонент приложения, который должен выполнить действие. Их можно применять для выполнения действий в пользовательском приложении — таких, например, как запуск действия (рис. 2.1).

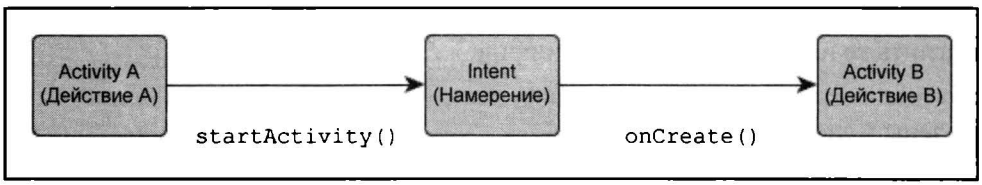


Рис. 2.1. Схема применения явного намерения

Фильтры намерений

Фильтр намерений — это объявление в файле манифеста приложения, где указывается тип намерения, которое хотел бы получить компонент. Это полезно для ряда случаев — таких, например, как выполняемый сценарий, когда желательно, чтобы действие в одном приложении выполняло определенное действие, запрошенное компонентами другого приложения. Тогда фильтр намерений может быть объявлен в манифесте приложения для действия, которое желательно обработать с помощью внешнего запроса. Если вы не желаете, чтобы действие обрабатывало неявные намерения, то не объявляете для него фильтр намерений.

Фрагменты

Фрагмент — это прикладной компонент, представляющий часть пользовательского интерфейса, существующего в действии. Подобно действию, фрагмент имеет макет, который можно изменять и который отображается в окне действия.

Службы

В отличие от большинства других компонентов, *служба* не предоставляет пользовательского интерфейса. Службы применяются для выполнения в приложении фоновых процессов. Службе не требуется приложение, которое ее создало, чтобы быть на переднем плане для запуска на выполнение.

Загрузчики

Загрузчик — это компонент, который позволяет загружать данные для последующего отображения их в действии или во фрагменте из источника данных, такого как провайдер контента.

Провайдеры контента

Эти компоненты помогают приложению контролировать доступ к ресурсам данных, сохраняющимся в том или ином приложении. Кроме того, *провайдер контента* облегчает обмен данными с другим приложением через открытый интерфейс программирования приложений (рис. 2.2).

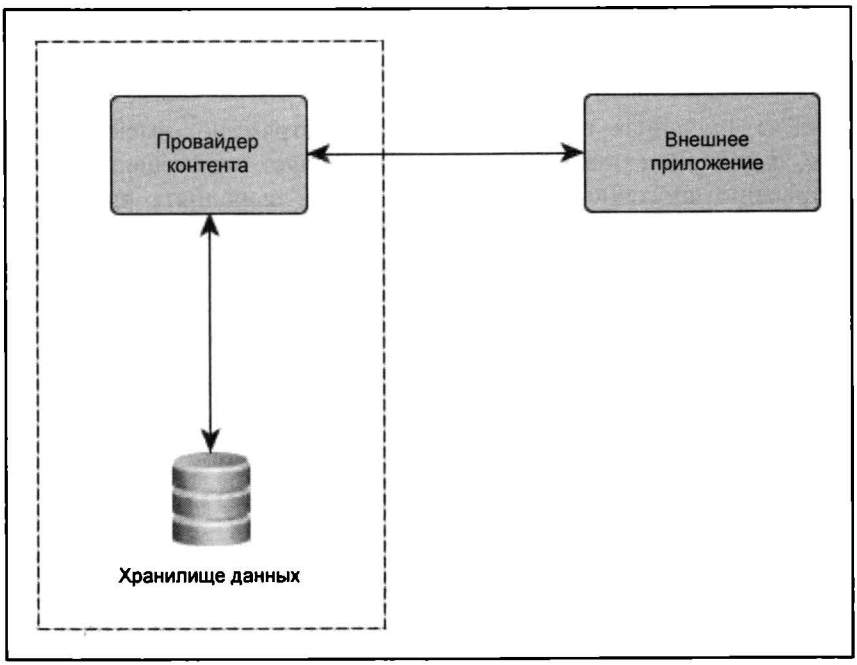


Рис. 2.2. Схема взаимодействия провайдера контента с внешним приложением

Tetris — правила игры

Прежде чем приступить к разработке приложения Tetris для Android, познакомимся с правилами этой игры.

Tetris («тетрис») — это игра-головоломка, основными компонентами которой являются цветные квадратные блоки-плитки. Название «тетрис» происходит от слов «тетра» — греческий числовой префикс для четырех предметов — и «теннис». Плитки в тетрисе формируют так называемые *тетрамино́*, представляющие собой геометрические фигуры, состоящие из четырех соединенных сторонами плиток (рис. 2.3).

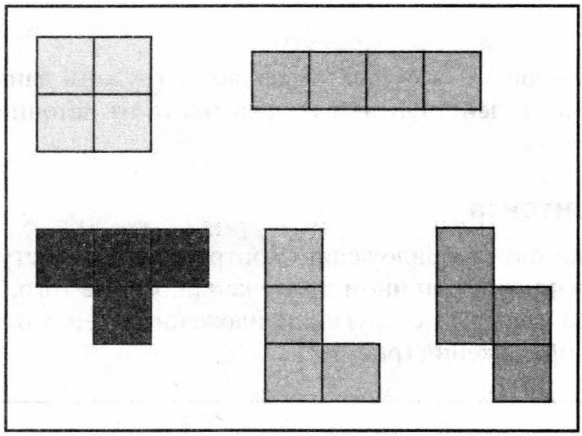


Рис. 2.3. Варианты тетрамино игры Tetris

Во время игры случайная последовательность тетрамино, имеющих различную ориентацию, падает на игровое поле. Игрок управляет перемещениями тетрамино. Каждое тетрамино способно по указаниям игрока выполнять ряд движений: их можно перемещать влево, вправо и вращать. Кроме того, скорость падения каждого тетрамино может быть увеличена. Цель игры — сформировать из опускающихся тетрамино непрерывную горизонтальную линию, состоящую из десяти квадратных плиток. Если подобная линия сформирована, она убирается с экрана.

Теперь, имея представление о функционировании игры Tetris, разберемся со спецификой формирования пользовательского интерфейса приложения.

Создание интерфейса пользователя

Как упоминалось ранее, пользовательский интерфейс служит средством, с помощью которого пользователь взаимодействует с приложением. Важность пользовательского интерфейса трудно переоценить. Прежде чем приступить к процессу программирования пользовательского интерфейса, полезно создать графическое представление о реализуемом пользовательском интерфейсе. Это может быть сде-

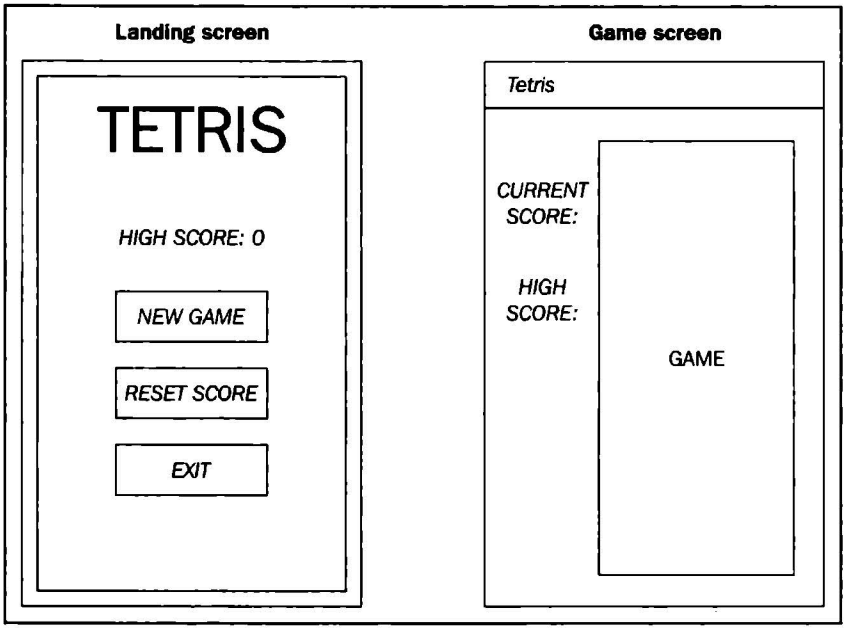


Рис. 2.4. Эскиз реализуемого пользовательского интерфейса

лано с помощью различных инструментов вплоть до Photoshop, но достаточно и простого эскиза (рис. 2.4).

На эскизе показано, что для этого приложении понадобятся два разных экрана: стартовый (главный) экран (**Landing screen**) и экран игры (**Game screen**), на котором протекает фактический игровой процесс. Поскольку экранов два, то и отдельных действий тоже должно быть два, — назовем их соответственно `MainActivity` и `GameActivity`.

Действие `MainActivity` служит отправной точкой создаваемого приложения. Оно содержит пользовательский интерфейс и относящуюся к стартовому экрану логику. Как показано в эскизе, пользовательский интерфейс стартового экрана включает заголовок приложения (**TETRIS**); представление, демонстрирующее пользователю его текущие высшие достижения (**HIGH SCORE**); и три кнопки для выполнения различных действий: кнопка **NEW GAME** (Новая игра), как следует из ее названия, запускает игру; кнопка **RESET SCORE** (Сбросить счет) приведет к обнулению очков пользователя до нуля; а кнопка **EXIT** (Выход) закроет приложение.

Действие `GameActivity` служит программным шаблоном игрового экрана. Для этого действия мы создадим представления и логические взаимодействия между пользователем и игрой. Пользовательский интерфейс этого действия включает панель с отображенным на нем названием приложения (**Tetris**); два текстовых представления, отображающих текущее количество (**CURRENT SCORE**) и рекордное количество очков (**HIGH SCORE**) пользователя; а также элемент макета, в котором происходит игровой процесс собственно тетриса **GAME** (Игра).

Реализация макета

Теперь, когда мы определили действия, необходимые в этом приложении, и когда получили общее представление о виде пользовательского интерфейса при просмотре его пользователем, приступим к реализации этого пользовательского интерфейса.

Создайте новый проект Android в Android Studio и назовите его `Tetris no activity`. После открытия окна IDE вы увидите, что структура проекта аналогична структуре, описанной в *главе 1*.

Сначала нужно добавить в проект действие `MainActivity`. Желательно, чтобы `MainActivity` было пустым действием. Для добавления действия `MainActivity` в проект щелкните правой кнопкой мыши на исходном пакете и в контекстном меню выберите команды **New | Activity | Empty Activity** (Создать | Действие | Пустое действие) (рис. 2.5)

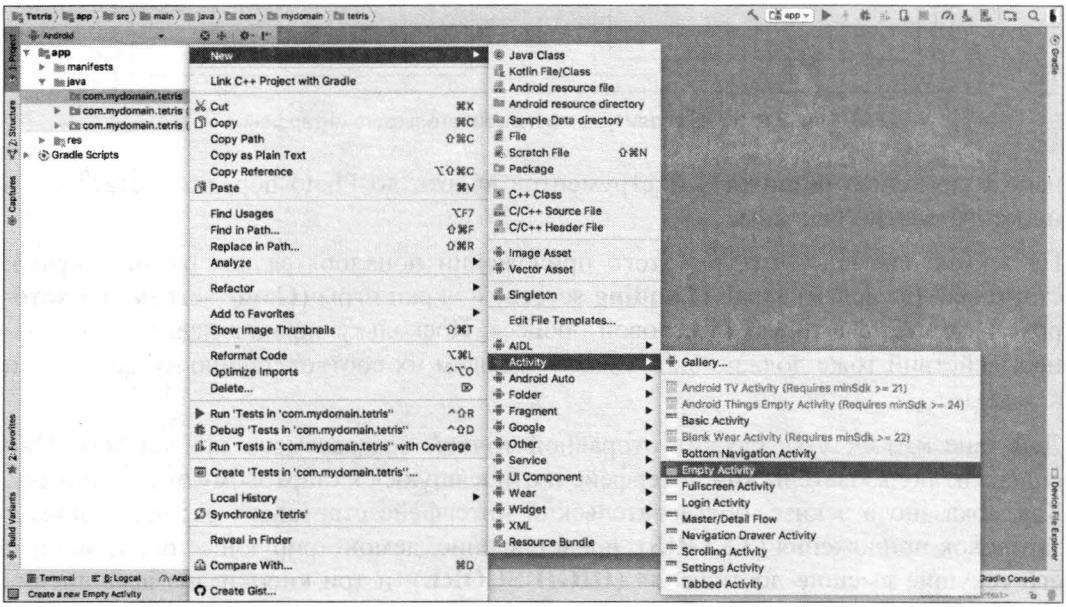


Рис. 2.5. Добавление в проект действия `MainActivity`

Назовите это действие `MainActivity` и убедитесь в том, что установлены флажки **Generate Layout File** (Генерировать файл макета), **Launcher Activity** (Действие загрузчика) и **Backwards Compatibility (AppCompat)** (Обратная совместимость (на уровне приложений)).

После добавления действия в проект перейдите к его файлу ресурсов макета. В этом файле должен содержаться примерно такой код:

```
<?xml version="1.0"
encoding="utf-8"?><android.support.constraint.ConstraintLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.mydomain.tetris.MainActivity">
</android.support.constraint.ConstraintLayout>
```

Первая строка файла ресурсов указывает используемую в файле версию XML, а также применяемую кодировку символов. В нашем случае применяется кодировка `utf-8`. Аббревиатура UTF расшифровывается как Unicode Transformation Format (Формат преобразования Unicode). Этот формат кодирования может быть столь же компактным, как и код ASCII (Американский стандартный код для обмена информацией, American Standard Code for Information Interchange) — наиболее распространенный символьный формат текстовых файлов, который может включать любой символ Unicode. Следующие восемь строк кода определяют тип группы представлений `ConstraintLayout`, который отображается в пользовательском интерфейсе `MainActivity`.

Прежде чем двигаться дальше, подробнее рассмотрим элемент `ConstraintLayout`.

Элемент *ConstraintLayout*

`ConstraintLayout` — это тип группы представлений, который позволяет выполнять гибкое позиционирование и изменение размера виджетов приложения. При работе с `ConstraintLayout` могут использоваться различные типы ограничений. Познакомимся с некоторыми из них.

Поля (Margins)

Поле — это пространство между двумя элементами макета. Если для элемента установлено боковое поле, оно добавляется к соответствующим границам макета, т. е. поле является как бы промежутком между границей макета и стороной элемента, к которому добавлено поле (рис. 2.6).

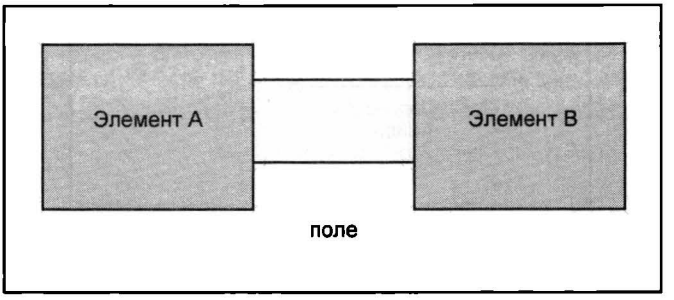


Рис. 2.6. Схема установки полей

Цепочки (Chains)

Цепочки — это ограничения, обеспечивающие групповое поведение вдоль одной оси. Ось может быть как горизонтальной, так и вертикальной (рис. 2.7).

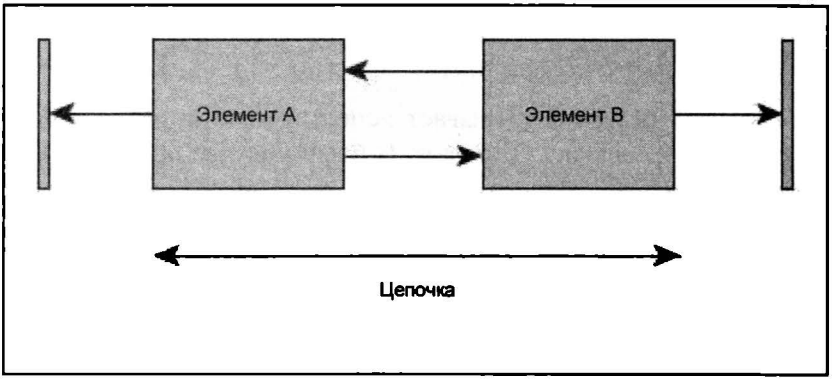


Рис. 2.7. Схема организации цепочек

Коллекция элементов представляет собой цепочку, если элементы связаны в двух направлениях.

Ограничения размерностей (Dimension constraints)

Эти ограничения касаются размерностей, относящихся к макету виджетов (рис. 2.8). Ограничения размерностей могут устанавливаться для виджетов и с помощью `ConstraintLayout`.

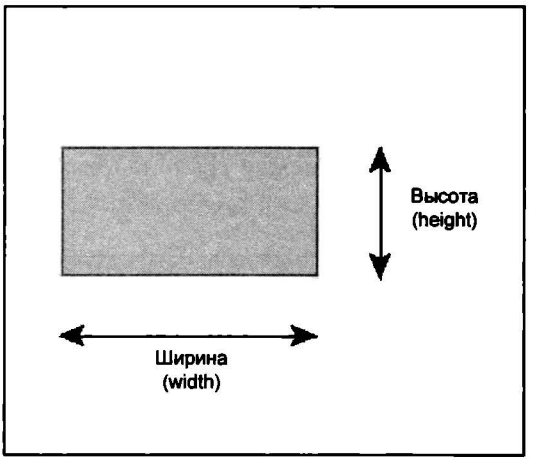


Рис. 2.8. Ограничения размерности

Размерности виджета можно указать с помощью атрибутов `android:layout_width` и `android:layout_height`:

```
<TextView
android:layout_height="16dp"
android:layout_width="32dp"/>
```

В ряде случаев можно пожелать, чтобы виджет имел те же размерности, что и его родительская группа представлений. Это можно реализовать, назначая атрибуту размерности значение `match_parent`:

```
<LinearLayout
   android:layout_width="120dp"
   android:layout_height="100dp">
<TextView
   android:layout_width="match_parent"
   android:layout_height="match_parent"/>
</LinearLayout>
```

В качестве альтернативы, если нужно, чтобы размерности виджета не фиксировались, а выполнялось обертывание содержащихся в них элементов, атрибуту размерности следует присвоить значение `wrap_content`:

```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="I wrap around the content within me"
  android:textSize="15sp"/>
```

Теперь, когда вы более четко представляете себе задачи `ConstraintLayout`, а также ограничения виджета, взглянем более внимательно на файл activity_main.xml:

```
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
    tools:context="com.mydomain.tetris.MainActivity">
</android.support.constraint.ConstraintLayout>
```

Рассматривая элемент `ConstraintLayout`, заметим, что размерности, относящиеся к ширине и длине, установлены равными `match_parent`. Это значит, что размерности `ConstraintLayout` соответствуют размерностям текущего окна. Для определения пространств имен XML используются атрибуты, имеющие префикс `xmlns:`. Значения, установленные для всех атрибутов пространств имен XML, используют пространство имен URI. Аббревиатура URI является сокращением от слов Uniform Resource Identifier (унифицированный идентификатор ресурса) и, как следует из этого названия, идентифицирует ресурс, необходимый для пространства имен.

Атрибут `tools:context` обычно устанавливается соответствующим корневому элементу в файле макета XML и указывает действие, с которым связан макет, — в нашем случае речь идет о действии `MainActivity`.

Теперь, когда мы получили представление о том, что сосредоточено в макете, описанном в файле activity_main.xml, добавим к нему некоторые элементы макета. Из эскиза видно, что все элементы макета размещены в вертикальном положении. Реализовать это можно с использованием элемента `LinearLayout`:

```
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context="com.mydomain.tetris.MainActivity">
   <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintLeft_toLeftOf="parent"
      app:layout_constraintRight_toRightOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      android:layout_marginVertical="16dp"
      android:orientation="vertical">
   </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Поскольку необходимо, чтобы элемент `LinearLayout` имел те же размерности, что и его предок, присвоим атрибутам `android:layout_width` и `android:layout_height` значение `match_parent`. Затем укажем ограничения для краев `LinearLayout` с помощью атрибутов:

- `app:layout_constraintBottom_toBottomOf` — он выравнивает нижний край одного элемента по нижнему краю другого элемента;
- `app:layout_constraintLeft_toLeftOf` — он выравнивает левый край элемента по левому краю другого элемента;
- `app:layout_constraintRight_toRightOff` — он выравнивает правый край элемента по правому краю другого элемента;
- `app:layout_constraintTop_toTopOf` — он выравнивает верхнюю границу элемента по верхней границе другого элемента.

В нашем случае все края `LinearLayout` выровнены по краю его предка — `ConstraintLayout`. А атрибут `android:layout_marginVertical` добавляет поле размером `16dp` вверху и внизу элемента.

Определение ресурсов измерений

Обычно в файле макета может находиться множество элементов, задающих одинаковые значения ограничений для атрибутов. Подобные значения должны добавляться в файл ресурсов измерений. Перейдем к созданию такого файла. В представлении проекта приложения перейдите в каталог res\values и создайте в нем новый файл ресурсов измерений с именем dimens (рис. 2.9).

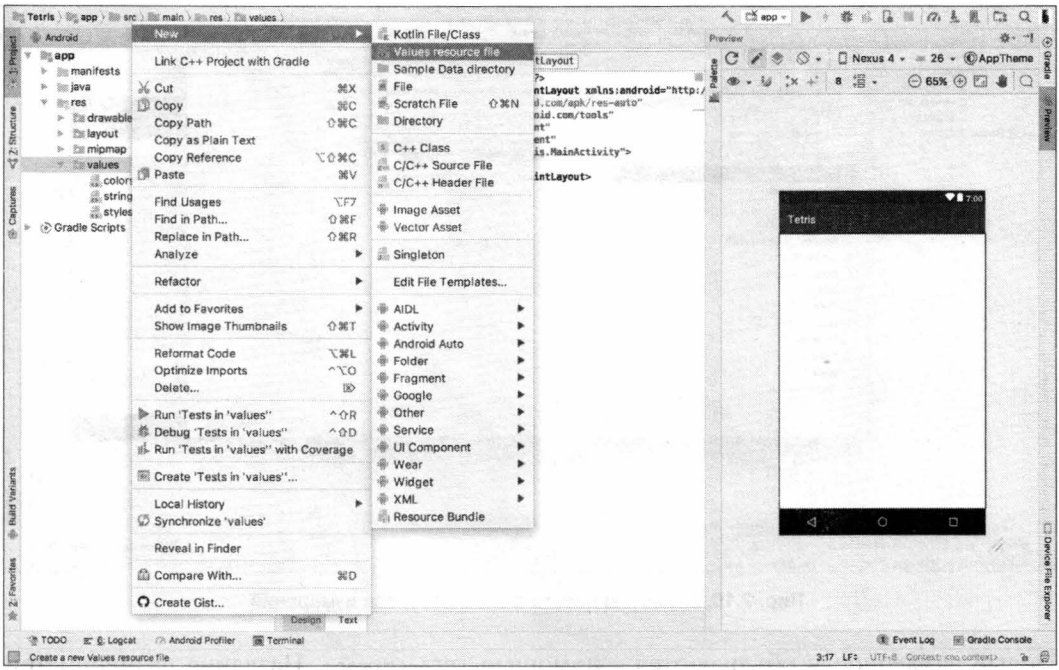


Рис. 2.9. Создание нового файла ресурсов измерений

Оставьте все атрибуты файла с указанием заданных по умолчанию значений (рис. 2.10).

Создав файл, откройте его. Содержание его похоже на следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<resources></resources>
```

Первая строка файла dimens.xml объявляет версию XML и кодировку используемых в файле символов. Вторая строка включает тег ресурсов `<resources>`. Этот тег служит для объявления размерностей. Добавьте в этот файл несколько значений измерений, как показано в следующем коде:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <dimen name="layout_margin_top">16dp</dimen>
   <dimen name="layout_margin_bottom">16dp</dimen>
```

```
    <dimen name="layout_margin_start">16dp</dimen>
    <dimen name="layout_margin_end">16dp</dimen>
    <dimen name="layout_margin_vertical">16dp</dimen>
</resources>
```

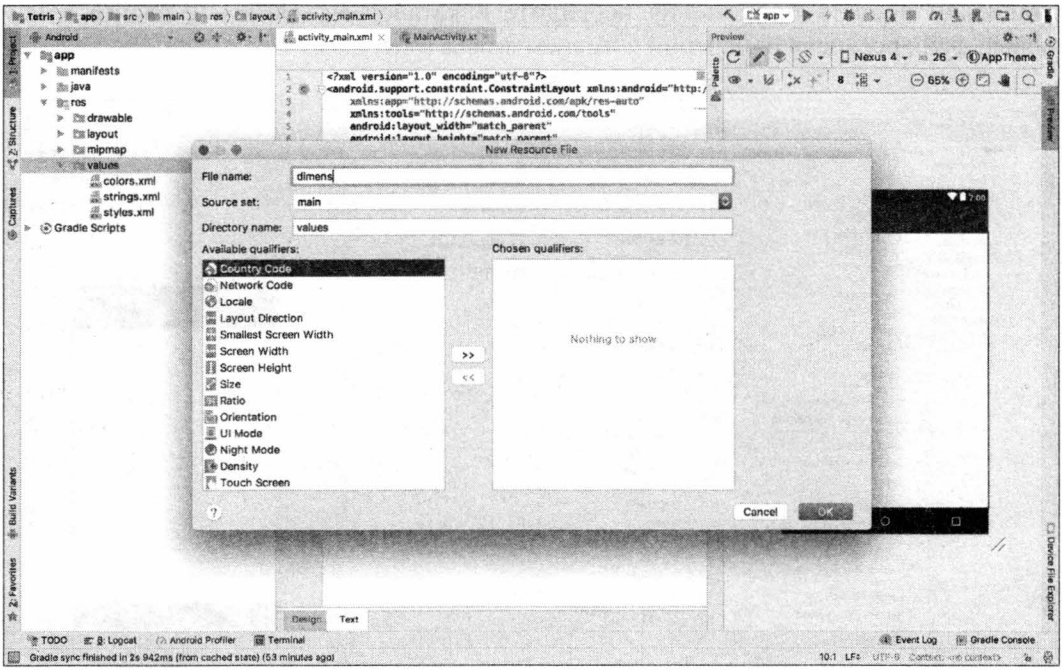


Рис. 2.10. Атрибуты нового файла ресурсов измерений

Новые размерности объявляются с помощью тега `<dimen>`. Названия размерностей, как правило, пишутся с учетом «правила змеи». А значение для размерности добавляется с помощью открывающего тега `<dimen>` и закрывающего тега `</dimens>`.

После добавления нескольких размерностей можно использовать их в линейном макете:

```
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.mydomain.tetris.MainActivity">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
```

```
      app:layout_constraintRight_toRightOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      android:layout_marginTop="@dimen/layout_margin_top"
<!- layout_margin_top dimension reference ->
      android:layout_marginBottom="@dimen/layout_margin_bottom"
<!- layout_margin_top dimension reference ->
      android:orientation="vertical"
      android:gravity="center_horizontal">
</LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Итак, мы получили настройку группы представлений `LinearLayout`, и теперь нужно добавить к ней необходимые представления макета. Но прежде чем это сделать, надо разобраться с концепциями представлений и групп представлений.

Представления (Views)

Представление — элемент макета, который занимает заданную область экрана и отвечает за отрисовку и обработку событий. Представление является базовым классом для элементов пользовательского интерфейса, или виджетов, таких как текстовые поля, поля ввода и кнопки. Все представления расширяют класс `View`.

Представления могут формироваться в макете XML в исходном файле:

```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Roll the dice!"/>
```

Помимо создания представлений непосредственно в файле макета, также можно создавать их программно в файлах программы. Например, текстовое представление можно сформировать путем создания экземпляра класса `TextView` и передачи контекста его конструктору. Этот подход продемонстрирован в следующем фрагменте кода:

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      val textView: TextView = TextView(this)
   }
}
```

Группы представлений (View groups)

Группа представлений — это особое представление, которое может включать в себя представления. Группу представлений, содержащую одно или несколько представлений, обычно называют *родительским представлением*, и представления

включаются в нее как ее дочерние представления. Группа представлений является родительским классом для некоторых других контейнеров представлений. Примерами групп представлений могут служить: `LinearLayout`, `CoordinatorLayout`, `ConstriantLayout`, `RelativeLayout`, `AbsoluteLayout`, `GridLayout` и `FrameLayout`.

Группы представлений могут формироваться в макете XML в исходном файле:

```
<LinearLayout
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginTop="16dp"
android:layout_marginBottom="16dp"/>
```

Подобно представлениям, группы представлений могут также формироваться программно в классах компонентов. В следующем фрагменте кода создается линейный макет путем формирования экземпляра класса `LinearLayout` и передачи `context` из `MainActivity` к его конструктору:

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      val linearLayout: LinearLayout = LinearLayout(this)
   }
}
```

Разобравшись с концепциями представлений и групп представлений, можно добавить к макету еще несколько представлений. Текстовые представления добавляются в макет с помощью элемента `<TextView>`, а кнопки — с помощью элемента `<Button>`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      tools:context="com.mydomain.tetris.MainActivity">
   <LinearLayout
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         app:layout_constraintBottom_toBottomOf="parent"
         app:layout_constraintLeft_toLeftOf="parent"
         app:layout_constraintRight_toRightOf="parent"
         app:layout_constraintTop_toTopOf="parent"
         android:layout_marginTop="@dimen/layout_margin_top"
         android:layout_marginBottom="@dimen/layout_margin_bottom"
         android:orientation="vertical">
```

```
    <TextView
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="TETRIS"
          android:textSize="80sp"/>
    <TextView
          android:id="@+id/tv_high_score"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="High score: 0"
          android:textSize="20sp"
          android:layout_marginTop="@dimen/layout_margin_top"/>
    <LinearLayout
          android:layout_width="match_parent"
          android:layout_height="0dp"
          android:layout_weight="1"
          android:orientation="vertical">
       <Button
          android:id="@+id/btn_new_game"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="New game"/>
       <Button
          android:id="@+id/btn_reset_score"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Reset score"/>
       <Button
          android:id="@+id/btn_exit"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="exit"/>
       </LinearLayout>
    </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Как можно видеть, здесь добавлены два текстовых представления для сохранения заголовка приложения и рекордного количества очков, а также три кнопки, предназначенные для выполнения необходимых действий. При этом задействовано два новых атрибута: `android:id` и `android:layout_weight`. Атрибут `android:id` используется для установки в макете уникального идентификатора элемента, поскольку два элемента в одном макете не могут иметь одинаковые идентификаторы. Атрибут `android:layout_weight` служит для указания значения приоритета, а именно размера пространства для представления в родительском контейнере:

```
<LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:orientation="vertical">
   <Button
      android:layout_width="70dp"
      android:layout_height="40dp"
      android:text="Click me"/>
   <View
      android:layout_width="70dp"
      android:layout_height="0dp"
      android:layout_weight="1"/>
</LinearLayout>
```

В этом фрагменте кода два дочерних представления содержатся в линейном макете. Кнопка явно устанавливает оба ограничения размерностей так, что они будут соответствовать значениям `70dp` и `40dp` соответственно. С другой стороны, ширина представления равна `70dp`, а высота — `0dp`. Вследствие наличия атрибута `android:layout_weight`, принимающего значение `1`, высота представления устанавливается таким образом, чтобы покрывать в родительском представлении всё оставшееся пространство.

Теперь, когда понятно, как в этом макете все устроено, взглянем на полученный нами его дизайн (рис. 2.11).

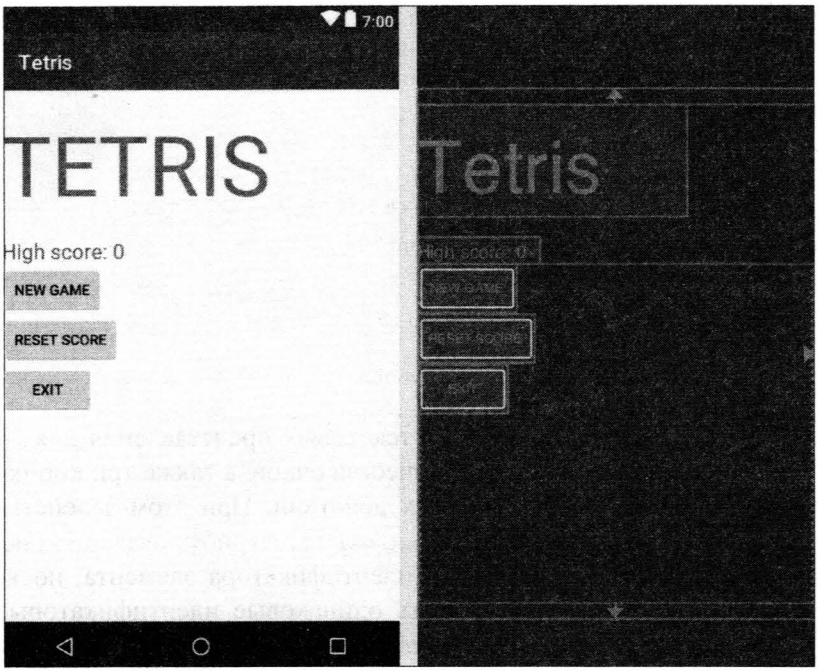


Рис. 2.11. Предварительный просмотр дизайна макета приложения

Как видите, пока что мы не достигли поставленной цели. В отличие от эскиза, элементы макета не центрированы, а выровнены влево. Эту проблему можно решить применением в группах линейных представлений атрибута `android:gravity`. В следующем фрагменте кода атрибут `android:gravity` применен для центрирования виджетов макетов в обоих линейных макетах:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      tools:context="com.mydomain.tetris.MainActivity">
   <LinearLayout
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         app:layout_constraintBottom_toBottomOf="parent"
         app:layout_constraintLeft_toLeftOf="parent"
         app:layout_constraintRight_toRightOf="parent"
         app:layout_constraintTop_toTopOf="parent"
         android:layout_marginTop="@dimen/layout_margin_top"
         android:layout_marginBottom="@dimen/layout_margin_bottom"
         android:orientation="vertical"
         android:gravity="center">
      <!-- Выравнивание дочерних элементов по центру группы представлений -->
      <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="TETRIS"
         android:textSize="80sp"/>
      <TextView
         android:id="@+id/tv_high_score"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="High score: 0"
         android:textSize="20sp"
         android:layout_marginTop="@dimen/layout_margin_top"/>
      <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:orientation="vertical"
            android:gravity="center">
      <!-- Выравнивание дочерних элементов по центру группы представлений -->
      <Button
            android:id="@+id/btn_new_game"
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="New game"/>
        <Button
            android:id="@+id/btn_reset_score"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Reset score"/>
        <Button
            android:id="@+id/btn_exit"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="exit"/>
        </LinearLayout>
    </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

В результате выполнения этого кода атрибуту `android:gravity` присваивается значение `center`, и виджеты выравниваются надлежащим образом. Эффекты применения групп представлений `android:gravity` к группам макетов представлений показаны на рис. 2.12.

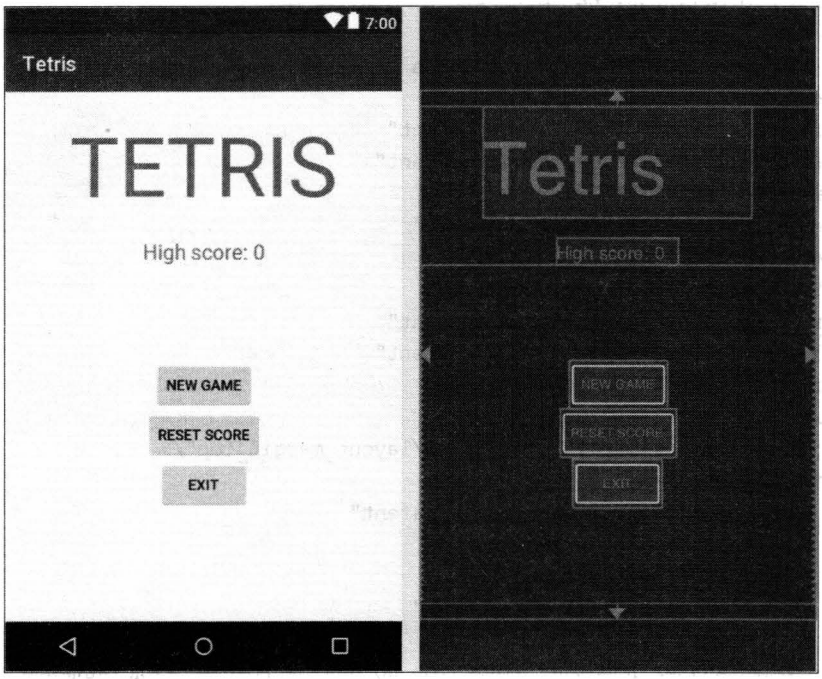


Рис. 2.12. Виджеты в макете выровнены по центру

Определение ресурсов строк

До сих пор в качестве значений атрибутов элемента, которые служат для установки текста, передавались жестко закодированные строки. Однако это не лучший вариант и его, как правило, следует избегать. Вместо этого строковые значения должны добавляться в файл ресурсов строк.

Заданный по умолчанию файл для ресурсов строк — strings.xml — можно найти в каталоге res\values (рис. 2.13).

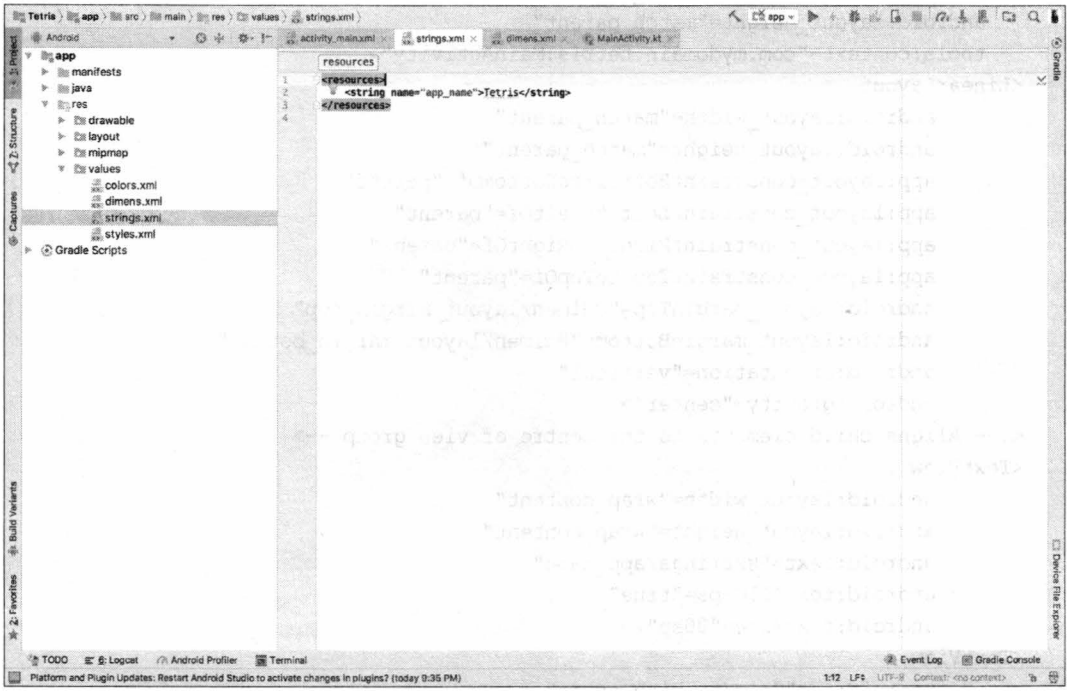


Рис. 2.13. Путь к файлу для ресурсов строк

Строковые значения добавляются как ресурсы строк с помощью XML-тега `<string>`. Нам нужно добавить строковые ресурсы для всех строковых значений, которые использовались до сих пор. Поэтому добавьте в файл ресурсов строк следующий код:

```
<resources>
   <string name="app_name">Tetris</string>
   <string name="high_score_default">High score: 0</string>
   <string name="new_game">New game</string>
   <string name="reset_score">Reset score</string>
   <string name="exit">exit</string>
</resources>
```

Теперь необходимо изменить файл макета MainActivity с тем, чтобы он мог использовать созданный файл ресурсов строк. На ресурс строк можно ссылаться с помощью @strings/ с префиксом имени файла ресурса строк:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      tools:context="com.mydomain.tetris.MainActivity">
   <LinearLayout
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         app:layout_constraintBottom_toBottomOf="parent"
         app:layout_constraintLeft_toLeftOf="parent"
         app:layout_constraintRight_toRightOf="parent"
         app:layout_constraintTop_toTopOf="parent"
         android:layout_marginTop="@dimen/layout_margin_top"
         android:layout_marginBottom="@dimen/layout_margin_bottom"
         android:orientation="vertical"
         android:gravity="center">
   <!-- Aligns child elements to the centre of view group -->
   <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@strings/app_name"
         android:textAllCaps="true"
         android:textSize="80sp"/>
   <TextView
         android:id="@+id/tv_high_score"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@strings/high_score_default"
         android:textSize="20sp"
         android:layout_marginTop="@dimen/layout_margin_top"/>
   <LinearLayout
         android:layout_width="match_parent"
         android:layout_height="0dp"
         android:layout_weight="1"
         android:orientation="vertical"
         android:gravity="center">
   <!-- Выравнивание дочерних элеметов по центру группы представлений -->
   <Button
        android:id="@+id/btn_new_game"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="@strings/new_game"/>
    <Button
        android:id="@+id/btn_reset_score"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@strings/reset_score"/>
    <Button
        android:id="@+id/btn_exit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@strings/exit"/>
      </LinearLayout>
    </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Обработка событий ввода данных

В цикле взаимодействия пользователя с приложением имеется средство, поддерживающее для него некоторую форму ввода данных. Этим средством является взаимодействие с виджетом. Ввод данных может перехватываться с помощью событий. В приложениях Android события перехватываются из определенного объекта представления, с которым взаимодействует пользователь. Необходимые для обработки входных событий структуры и процедуры предоставляются классом `View`.

Слушатели событий

Слушатель событий — это процедура в приложении, которая ожидает события пользовательского интерфейса. Типов событий, которые могут генерироваться в приложении, достаточно много. Вот примеры некоторых общих событий: события щелчка, события касания, события длинного щелчка и события изменения текста.

Для захвата события виджета и выполнения действия при его возникновении слушатель событий должен быть в представлении установлен. Это может достигаться путем вызова набора представлений. Для такого вызова применяется метод `Listener()` с передачей либо лямбда-выражения, либо ссылки на функцию.

В следующем примере демонстрируется захват события щелчка на кнопке. Лямбда-выражение при этом передается методу `setOnClickListener` класса представления:

```
val button: Button = findViewById<Button>(R.id.btn_send)
button.setOnClickListener {
    // Действия, выполняемые по событию щелчка
}
```

Вместо лямбда-выражения может быть передана и ссылка на функцию:

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      val btnExit: Button = findViewById<Button>(R.id.btn_exit)
      btnExit.setOnClickListener(this::handleExitEvent)
   }
   fun handleExitEvent(view: View) {
      finish()
   }
}
```

В классе представления доступны многие методы установки слушателя. Рассмотрим некоторые примеры:

- `setOnClickListener()` — устанавливает функцию, которая вызывается при щелчке на представлении;
- `setOnContextClickListener()` — устанавливает функцию, которая вызывается после щелчка на контексте представления;
- `setOnCreateContextMenuListener()` — устанавливает функцию, которая вызывается при создании контекстного меню представления;
- `setOnDragListener()` — устанавливает функцию, которая будет вызываться при возникновении события перетаскивания в представлении;
- `setOnFocusChangeListener()` — устанавливает функцию, вызываемую при изменении фокуса представления;
- `setOnHoverChangeListener()` — устанавливает функцию, вызываемую при возникновении события наведения на представление;
- `setOnLongClickListener()` — устанавливает функцию, которая вызывается при возникновении события длинного щелчка в представлении;
- `setOnScrollChangeListener()` — устанавливает функцию, вызываемую при изменении положений прокрутки (X или Y) для представления.

Слушатель событий — процедура в прикладной программе, которая ожидает события пользовательского интерфейса.

Имея четкое понимание, каким образом обрабатываются события ввода данных, перейдем к реализации соответствующей логики в `MainActivity`.

Основной экран действия содержит панель приложения. Необходимо скрыть этот элемент макета, поскольку наше представление его не требует (рис. 2.14).

Панель приложения называется также *панелью действий*. Панели действий служат экземплярами класса `ActionBar`. Экземпляр виджета панели действий в макете мож-

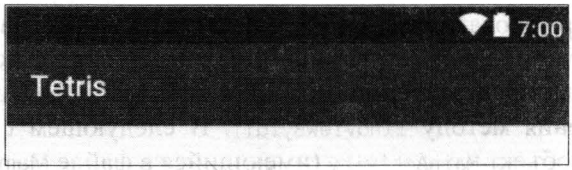


Рис. 2.14. Панель приложения

но получить с помощью переменной доступа `supportActionBar`. При выполнении следующего кода панель действий извлекается и скрывается, если не возвращается нулевая ссылка:

```
package com.mydomain.tetris
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.support.v7.app.ActionBar
import android.view.View
import android.widget.Button
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      val appBar: ActionBar? = supportActionBar
      if (appBar != null) {
         appBar.hide()
      }
   }
}
```

И хотя приведенный здесь код выполняет необходимые действия, он может быть сокращен вследствие использования системы безопасности типов Kotlin, как показано здесь:

```
package com.mydomain.tetris
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.Button
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      supportActionBar?.hide()
   }
}
```

Если ссылка `supportActionBar` не является пустой ссылкой на объект, вызывается метод `hide()`, если не предусмотрено что-либо другое. Это предотвратит исключение нулевого указателя.

Настало время сформировать ссылки на объекты для виджетов, которые имеются в макетах. Это нужно по многим причинам, например для регистрация слушателя. Ссылки на объекты представления можно получить, передавая идентификатор ресурса представления методу `findViewById()`. В следующем фрагменте кода мы добавим ссылки на объект `MainActivity` (имеющийся в файле MainActivity.kt):

```
package com.mydomain.tetris
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.TextView

class MainActivity : AppCompatActivity() {

   var tvHighScore: TextView? = null

   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      supportActionBar?.hide()
      val btnNewGame = findViewById<Button>(R.id.btn_new_game)
      val btnResetScore = findViewById<Button>(R.id.btn_reset_score)
      val btnExit = findViewById<Button>(R.id.btn_exit)
      tvHighScore = findViewById<TextView>(R.id.tv_high_score)
   }
}
```

Теперь, при наличии ссылок на объекты для элементов пользовательского интерфейса, нужно обработать некоторые из событий. Установим слушатели событий щелчков для всех кнопок макета (не имеет смысла кнопка, которая не сработает в случае щелчка на ней).

Как уже упоминалось ранее (см. *разд. «Создание интерфейса пользователя»*), кнопка **NEW GAME** (Новая игра) предназначена только для перехода пользователя к игровому действию, в котором и происходит собственно игра (для этого нам понадобится явное намерение). Добавьте частную функцию, содержащую логику, которая выполняется после щелчка на кнопке **NEW GAME** (Новая игра) для `MainActivity` (в файле MainActivity.kt), и установите ссылку на функцию с помощью вызова `setOnClickListener()`:

```
package com.mydomain.tetris
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.TextView
```

```
class MainActivity : AppCompatActivity() {
   var tvHighScore: TextView? = null

   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      supportActionBar?.hide()
      val btnNewGame = findViewById<Button>(R.id.btn_new_game)
      val btnResetScore = findViewById<Button>(R.id.btn_reset_score)
      val btnExit = findViewById<Button>(R.id.btn_exit)
      tvHighScore = findViewById<TextView>(R.id.tv_high_score)

      btnNewGame.setOnClickListener(this::onBtnNewGameClick)
   }

   private fun onBtnNewGameClick(view: View) { }
}
```

Создадим новое пустое действие и назовем его `GameActivity`. После создания действия можно применить намерение для запуска этого действия при щелчке на кнопке **NEW GAME** (Новая игра):

```
private fun onBtnNewGameClick(view: View) {
   val intent = Intent(this, GameActivity::class.java)
   startActivity(intent)
}
```

Первая строка тела функции создает новый экземпляр класса `Intent` и передает конструктору текущий контекст и требуемый класс действия. Обратите внимание, что ключевое слово `this` передается конструктору в качестве первого аргумента. Ключевое слово `this` используется для ссылки на текущий экземпляр, в котором вызывается. Следовательно, в качестве первого аргумента конструктору фактически передается текущее действие (`MainActivity`). Может возникнуть вопрос, почему действие передается в качестве первого аргумента конструктора `Intent`, если для первого аргумента необходим контекст? Дело в том, что все действия являются расширениями абстрактного класса `Context`. Следовательно, все действия находятся в их собственных правовых контекстах.

Метод `startActivity()` вызывается для запуска действия, от которого не ожидается каких-либо результатов. Если намерение передается в качестве единственного аргумента, оно начинает действие, от которого не ожидается результата. Запустите приложение и понаблюдайте за эффектом щелчка на кнопке.

Класс `Context` служит абстрактным классом в структуре приложения Android. Реализация контекста обеспечивается системой Android. Класс `Context` позволяет получить доступ к ресурсам конкретного приложения. Он также предоставляет доступ к вызовам для операций на уровне приложений, таких как запуск, отправка широковещательных сообщений и получение намерений.

Теперь реализуем следующие функции для щелчков на кнопках **EXIT** (Выход) и **RESET SCORE** (Сбросить счет):

```
package com.mydomain.tetris
import android.content.Intent
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.TextView

class MainActivity : AppCompatActivity() {

   var tvHighScore: TextView? = null

   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      supportActionBar?.hide()

      val btnNewGame = findViewById<Button>(R.id.btn_new_game)
      val btnResetScore = findViewById<Button>(R.id.btn_reset_score)
      val btnExit = findViewById<Button>(R.id.btn_exit)
      tvHighScore = findViewById<TextView>(R.id.tv_high_score)
      btnNewGame.setOnClickListener(this::onBtnNewGameClick)
      btnResetScore.setOnClickListener(this::onBtnResetScoreClick)
      btnExit.setOnClickListener(this::onBtnExitClick)
   }

   private fun onBtnNewGameClick(view: View) {
      val intent = Intent(this, GameActivity::class.java)
      startActivity(intent)
   }

   private fun onBtnResetScoreClick(view: View) {}

   private fun onBtnExitClick(view: View) {
      System.exit(0)
   }
}
```

Обращение к методу `System.exit()` в функции `onBtnExitClick` прекращает дальнейшее выполнение программы и завершает ее, если в качестве аргумента передается целое число `0`. Последнее, что необходимо выполнить для обработки событий щелчка, — реализовать логику для сброса наибольшего количество очков. Для этого нужно сначала реализовать некоторую логику для хранения данных, что

позволит сохранить наибольшее количество очков. Для выполнения этой задачи воспользуемся интерфейсом `SharedPreferences`.

Использование интерфейса *SharedPreferences*

Интерфейс `SharedPreferences` обеспечивает хранение, организацию доступа и модификацию данных, а также сохранение данных в наборах пар ключ-значение.

Настроим простую утилиту для обработки потребности в хранении данных для нашего приложения, применив интерфейс `SharedPreferences`. Создайте в исходном каталоге проекта пакет с именем storage — щелкните правой кнопкой мыши на исходном каталоге и выберите команды **New | Package** (Создать | Пакет) (рис. 2.15).

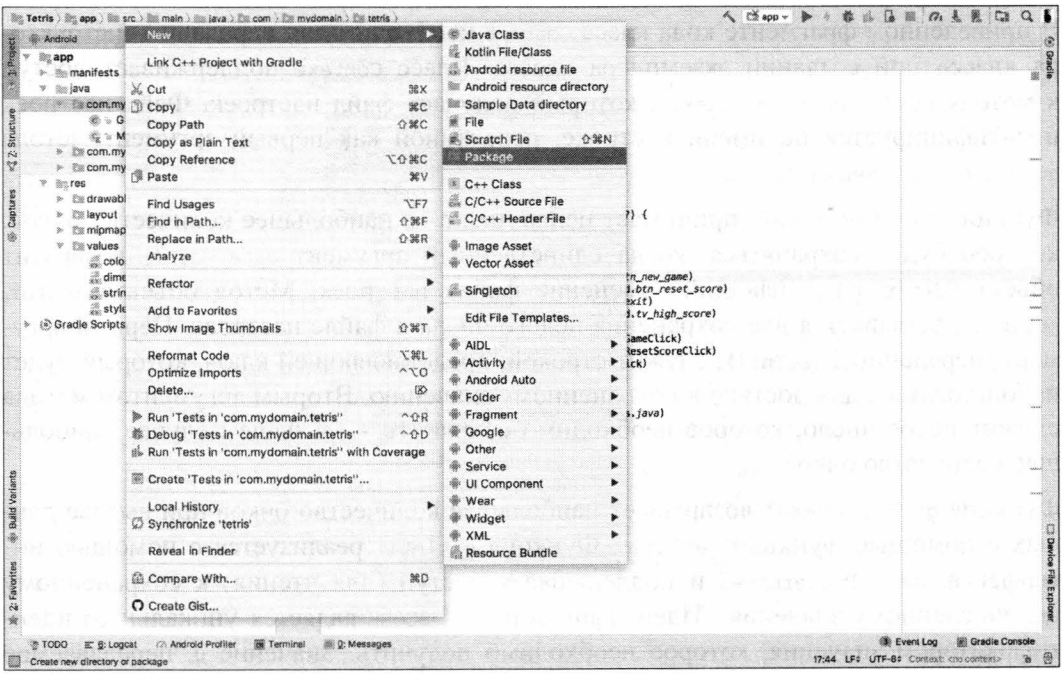


Рис. 2.15. Создание пакета

Затем внутри пакета storage создайте новый класс Kotlin под названием `AppPreferences`. Введите в файл класса следующий код:

```
package com.mydomain.tetris.storage
import android.content.Context
import android.content.SharedPreferences

class AppPreferences(ctx: Context) {

   var data: SharedPreferences = ctx.getSharedPreferences
   ("APP_PREFERENCES", Context.MODE_PRIVATE)
```

```
    fun saveHighScore(highScore: Int) {
        data.edit().putInt("HIGH_SCORE", highScore).apply()
    }

    fun getHighScore(): Int {
        return data.getInt("HIGH_SCORE", 0)
    }

    fun clearHighScore() {
        data.edit().putInt("HIGH_SCORE", 0).apply()
    }
}
```

В приведенном фрагменте кода класс `Context` необходим для передачи конструктора класса при создании экземпляра класса. Класс `Context` поддерживает доступ к методу `getSharedPreferences()`, который получает файл настроек. Файл настроек идентифицируется по имени в строке, переданной как первый аргумент метода `getSharedPreferences()`.

Функция `saveHighScore()` принимает целое число — наибольшее количество очков, которое будет сохраняться, когда единственный аргумент `data.edit()` возвратит объект `Editor`, разрешающий изменение файла настроек. Метод объекта `Editor`, `putInt()`, вызывается для сохранения целого числа в файле настроек. Первый аргумент, переданный `putInt()`, служит строкой, представляющей ключ, который будет использоваться для доступа к сохраненному значению. Вторым аргументом метода служит целое число, которое необходимо сохранить, — в нашем случае наибольшее количество очков.

Функция `getHighScore()` возвращает наибольшее количество очков при вызове данных с помощью функции `getInt()`. Функция `getInt()` реализуется с помощью интерфейса `SharedPreferences` и поддерживает доступ (для чтения) к сохраненному целочисленному значению. Идентификатор `HIGH_SCORE` является уникальным идентификатором значения, которое необходимо получить. Значение `0`, передаваемое второму аргументу функции, указывает заданное по умолчанию значение, которое должно возвращаться сценарием, если в сценарии отсутствует соответствующее указанному ключу значение.

Функция `clearHighScore()` сбрасывает значение наибольшего количества очков до нуля, просто перезаписывая нулем значение, соответствующее ключу `HIGH_SCORE`.

Создав класс утилит `AppPreferences`, можно завершить в действии `MainActivity` создание функции `onBtnResetScoreClick()`:

```
private fun onBtnResetScoreClick(view: View) {
    val preferences = AppPreferences(this)
    preferences.clearHighScore()
}
```

Теперь после щелчка на кнопке **RESET SCORE** (Сбросить счет) имеющееся значение наибольшего количества очков сбрасывается к нулю. Но нам необходимо также при этом дать пользователю возможность своего рода обратной связи. Создадим обратную связь с помощью класса `Snackbar`.

Чтобы получить возможность применения в приложении Android класса `Snackbar`, необходимо добавить в процесс сборки сценария Gradle на уровне модуля зависимость библиотеки поддержки дизайна Android. Для этого внесите следующую строку кода под строкой закрытия зависимостей `build.gradle`:

```
implementation 'com.android.support:design:26.1.0'
```

После добавления этой строки сценарий на уровне модуля `build.gradle` будет иметь вид, аналогичный следующему:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
   compileSdkVersion 26
   buildToolsVersion "26.0.1"
   defaultConfig {
      applicationId "com.mydomain.tetris"
      minSdkVersion 15
      targetSdkVersion 26
      versionCode 1
      versionName "1.0"
      testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"

   }
   buildTypes {
      release {
         minifyEnabled false
         proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
         }
      }
   }
   dependencies {
      implementation fileTree(dir: 'libs', include: ['*.jar'])
      implementation "org.jetbrains.kotlin:
      kotlin-stdlib-jre7:$kotlin_version"
      implementation 'com.android.support:appcompat-v7:26.1.0'
      implementation 'com.android.support.constraint:
      constraint-layout:1.0.2'
      testImplementation 'junit:junit:4.12'
```

```
        androidTestImplementation 'com.android.support.test:runner:1.0.1'
        androidTestImplementation 'com.android.support.test.espresso:espressocore:
                                                                        3.0.1'
        implementation 'com.android.support:design:26.1.0'
    // добавление библиотеки поддержка дизайна android
}
```

Добавив указанные изменения, синхронизируйте свой проект, щелкнув на ссылке **Sync Now** (Синхронизировать) всплывающего в окне редактора сообщения (рис. 2.16).

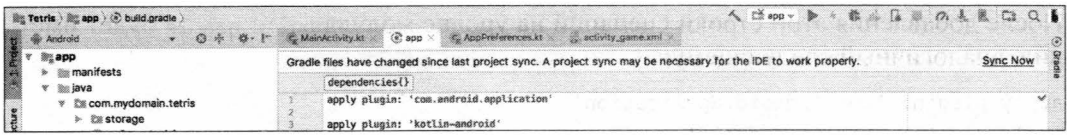


Рис. 2.16. Ссылка **Sync Now**

Без лишних слов изменим теперь функцию `onBtnResetClick()`, обеспечивающую в форме `Snackbar` поддержку обратной связи с пользователем после сброса наибольшего числа очков:

```
private fun onBtnResetScoreClick(view: View) {
    val preferences = AppPreferences(this)
    preferences.clearHighScore()
    Snackbar.make(view, "Score successfully reset",
    Snackbar.LENGTH_SHORT).show()
}
```

После щелчка на кнопке **RESET SCORE** наибольшее количество очков игрока успешно сбрасывается (рис. 2.17).

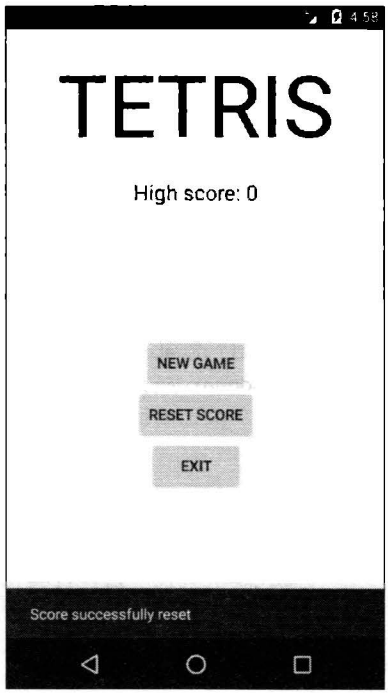


Рис. 2.17. После щелчка на кнопке **RESET SCORE** наибольшее количество очков игрока успешно сбрасывается

Прежде чем идти дальше, необходимо обновить текст для макета MainActivity, выводимый в текстовом представлении наибольшего количества очков, что позволит отобразить сам факт сброса наибольшего количества очков. Для этого следующим образом изменим текст, содержащийся в текстовом представлении:

```
private fun onBtnResetScoreClick(view: View) {
   val preferences = AppPreferences(this)
   preferences.clearHighScore()
   Snackbar.make(view, "Score successfully reset",
      Snackbar.LENGTH_SHORT).show()
   tvHighScore?.text = "High score: ${preferences.getHighScore()}"
}
```

Реализация макета игрового действия

К этому моменту мы успешно создали макет основного действия. Но прежде чем завершить эту главу, необходимо также создать макет для игрового действия — GameActivity. Для этого откройте файл activity_game.xml и добавьте в него следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      tools:context="com.mydomain.tetris.GameActivity">
   <LinearLayout
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         android:orientation="horizontal"
         android:weightSum="10"
         android:background="#e8e8e8">
      <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:orientation="vertical"
            android:gravity="center"
            android:paddingTop="32dp"
            android:paddingBottom="32dp"
            android:layout_weight="1">
      <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="0dp"
            android:layout_weight="1"
```

```
            android:orientation="vertical"
            android:gravity="center">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/current_score"
            android:textAllCaps="true"
            android:textStyle="bold"
            android:textSize="14sp"/>
        <TextView
            android:id="@+id/tv_current_score"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="18sp"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="@dimen/layout_margin_top"
            android:text="@string/high_score"
            android:textAllCaps="true"
            android:textStyle="bold"
            android:textSize="14sp"/>
        <TextView
            android:id="@+id/tv_high_score"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="18sp"/>
        </LinearLayout>
        <Button
            android:id="@+id/btn_restart"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/btn_restart"/>
        </LinearLayout>
        <View
            android:layout_width="1dp"
            android:layout_height="match_parent"
            android:background="#000"/>
        <LinearLayout
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="9">
        </LinearLayout>
    </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Большинство задействованных в этом макете атрибутов представлений уже использовались ранее и поэтому не нуждаются в дополнительном пояснении. Единственными исключениями являются атрибуты `android:background` и `android:layout_weightSum`.

Атрибут `android:background` служит для установки цвета фона представления или группы представлений. Значения `#e8e8e8` и `#000` переданы в качестве значений двух экземпляров атрибута `android:background`, использованного в макете. Значение `#e8e8e8` представляет собой шестнадцатеричный код цвета для серого цвета, а `#000` — для черного цвета.

Атрибут `android:layout_weightSum` определяет максимальную весовую сумму в группе представлений и рассчитывается как сумма значений `layout_weight` для всех дочерних представлений. Первый линейный макет в файле activity_game.xml объявляет весовую сумму для всех дочерних представлений равной `10`. Таким образом, непосредственные потомки линейного макета имеют весá макета, равные `1` и `9` соответственно.

Здесь также использованы три ресурса строк, которые ранее в файл ресурсов строк не добавлялись. Так что вставим их в файл strings.xml прямо сейчас:

```
<string name="high_score">High score</string>
<string name="current_score">Current score</string>
<string name="btn_restart">Restart</string>
```

Наконец, добавим к игровому действию некоторую простую логику для заполнения текстовых представлений наибольшего количества очков и текущего числа очков:

```
package com.mydomain.tetris

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import android.widget.Button
import android.widget.TextView
import com.mydomain.tetris.storage.AppPreferences

class GameActivity: AppCompatActivity() {

   var tvHighScore: TextView? = null
   var tvCurrentScore: TextView? = null
   var appPreferences: AppPreferences? = null

   public override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_game)
      appPreferences = AppPreferences(this)

      val btnRestart = findViewById<Button>(R.id.btn_restart)
      tvHighScore = findViewById<TextView>(R.id.tv_high_score)
      tvCurrentScore = findViewById<TextView>(R.id.tv_current_score)
```

```
        updateHighScore()
        updateCurrentScore()
    }

    private fun updateHighScore() {
        tvHighScore?.text = "${appPreferences?.getHighScore()}"
    }

    private fun updateCurrentScore() {
        tvCurrentScore?.text = "0"
    }
}
```

В приведенном фрагменте кода создаются ссылки на объекты для элементов представлений макета. Кроме того, здесь объявлены функции `updateHighScore()` и `updateCurrentScore()`. Эти две функции вызываются при создании представления. Они устанавливают заданное по умолчанию количество очков, отображаемое в текущий момент, и текстовые представления максимального количества очков, которые объявляются в файле макета.

Сохраните внесенные в проект изменения, скомпилируйте и запустите приложение. Щелкните на кнопке **New Game**, чтобы приложение открыло только что созданный макет (рис. 2.18).

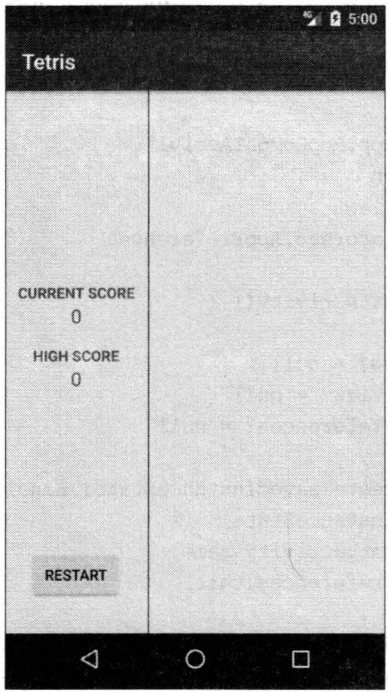


Рис. 2.18. Вид макета приложения по состоянию на текущий момент

Правая сторона макета, которая пока не показывает никакого содержания, и станет той областью, где будет происходить игра в тетрис. Мы реализуем эту область в *главе 3*. А пока рассмотрим манифест приложения.

Манифест приложения

Манифест приложения — это XML-файл, который присутствует в каждом приложении Android. Он находится в корневой папке манифестов приложения. Файл манифеста содержит важную информацию, относящуюся к приложению, действующему в операционной системе Android. Информация, содержащаяся в файле androidManifest.xml приложения, должна быть воспринята системой Android прежде, чем запустится само приложение. Вот информация, которая должна регистрироваться в манифесте приложения:

- имя пакета Java для приложения;
- действия, имеющиеся в приложении;
- службы, применяемые в приложении;
- фильтры намерений, которые направляют неявные намерения действию;
- описания используемых в приложении приемников широковещательных сообщений;
- данные, относящиеся к присутствующим в приложении провайдерам контента;
- классы, реализующие различные компоненты приложения;
- разрешения, необходимые для приложения.

Структура файла манифеста приложения

Общая структура файла androidManifest.xml показана в следующем фрагменте кода. Этот фрагмент кода содержит все возможные элементы и объявления, которые могут включаться в файл манифеста:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
   <uses-permission />
   <permission />
   <permission-tree />
   <permission-group />
   <instrumentation />
   <uses-sdk />
   <uses-configuration />
   <uses-feature />
   <supports-screens />
   <compatible-screens />
   <supports-gl-texture />
```

```
   <application>
      <activity>
         <intent-filter>
            <action />
            <category />
            <data />
         </intent-filter>
         <meta-data />
      </activity>
      <activity-alias>
         <intent-filter>
         . . .
         </intent-filter>
         <meta-data />
      </activity-alias>
      <service>
         <intent-filter>
         . . .
         </intent-filter>
         <meta-data/>
      </service>
      <receiver>
         <intent-filter>
         . . .
         </intent-filter>
      <meta-data />
      </receiver>
      <provider>
         <grant-uri-permission />
         <meta-data />
         <path-permission />
      </provider>
      <uses-library />
   </application>
</manifest>
```

Как видно из приведенного фрагмента кода, в файле манифеста может содержаться огромное число элементов, и многие из них рассматриваются в этой книге. Некоторые элементы этого файла уже применялись и в приложении Tetris. Откройте файл androidManifest.xml для приложения Tetris. Содержимое этого файла должно походить на следующий фрагмента кода:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.mydomain.tetris">
   <application
      android:allowBackup="true"
```

```
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:supportsRtl="true"
      android:theme="@style/AppTheme">
      <activity android:name=".MainActivity">
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
   </activity>
   <activity android:name=".GameActivity" />
   </application>
</manifest>
```

Далее в алфавитном порядке рассмотрены элементы, которые использовались в этом файле манифеста:

- <action>;
- <activity>;
- <application>;
- <category>;
- <intent-filter>;
- <manifest>.

Элемент <*action*>

Применяется для добавления действия к фильтру намерений и всегда является дочерним элементом для элемента <intent-filter>. Фильтр намерений должен включать один или несколько этих элементов. Если для фильтра намерений не объявлен элемент действия, фильтр не принимает объекты намерения. Синтаксис элемента имеет следующий вид:

```
<action name=""/>
```

Атрибут name здесь указывает название обрабатываемого действия.

Элемент <*activity*>

Этот элемент объявляет имеющуюся в приложении активность. Чтобы действия восприняла система Android, все они должны объявляться в манифесте приложения. Элемент <activity> всегда размещается внутри родительского элемента <application>. В следующем фрагменте кода показано объявление действия в файле манифеста с использованием элемента <activity>:

```
<activity android:name=".GameActivity" />
```

Атрибут `name` здесь указывает имя класса, который реализует декларируемое действие.

Элемент <application>

Этот элемент служит объявлением приложения. Он включает подэлементы, объявляющие имеющиеся в приложении компоненты. Следующий код демонстрирует применение элемента `<application>`:

```
<application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:supportsRtl="true"
      android:theme="@style/AppTheme">
   <activity android:name=".MainActivity">
      <intent-filter>
         <action android:name="android.intent.action.MAIN" />
         <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
   </activity>
   <activity android:name=".GameActivity" />
</application>
```

Элемент `<application>` в приведенном фрагменте кода задействует шесть атрибутов:

- `android:allowBackup` — служит для указания, разрешено ли приложению принимать участие в инфраструктуре резервного копирования и восстановления. Если установлено значение `true`, приложение может копироваться системой Android. Иначе, если для этого атрибута задано значение `false`, система Android не будет создавать резервную копию приложения;
- `android:icon` — указывает ресурс значков для приложения. Также можно применять его для указания ресурсов значков для компонентов приложения;
- `android:label` — определяет заданную по умолчанию метку для приложения в целом. Также можно применять его для указания заданных по умолчанию меток для компонентов приложения;
- `android:roundIcon` — определяет ресурс круглых значков, который будет применяться при необходимости. Если средство запуска запрашивает значок приложения, платформа Android возвращает либо `android:icon`, либо `android:roundIcon`. Возвращаемое значение зависит от конфигурации компиляции устройства. Поскольку возвращается любой значок, важно указать ресурс для обоих атрибутов;
- `android:supportsRtl` — указывает, готово ли приложение поддерживать макеты right-to-left (RTL), т. е. направление *справа налево*. Приложение настроено на его поддержку, если для этого атрибута установлено значение `true`. В противном случае приложение не поддерживает макеты RTL.

- `android:theme` — определяет ресурс стиля, задающего для всех действий в приложении установленную по умолчанию тему.

Элемент <*category*>

Этот элемент является дочерним элементом для `<intent-filter>`. Применяется для указания имени категории в родительском компоненте фильтра намерений элемента.

Элемент <*intent-filter*>

Этот элемент указывает тип намерения, на которое могут реагировать компоненты действия, службы и приемник вещания. Фильтр намерений всегда объявляется в родительском компоненте с помощью элемента `<intent-filter>`.

Элемент <*manifest*>

Корневой элемент файла манифеста приложения. Включает один элемент `<application>` и указывает атрибуты `xmlns:android` и `package`.

Подведем итоги

В этой главе подробно рассмотрена структура приложения Android. По ходу дела мы познакомились с семью основными компонентами приложения Android: действиями, намерениями, фильтрами намерений, фрагментами, службами, загрузчиками и провайдерами (поставщиками) контента.

Кроме того, мы разобрались с процессом создания макета, особенностями макета ограничений, типами имеющихся ограничений макета, ресурсами строк, ресурсами размерностей, представлениями, группами представлений и работой с интерфейсом `SharedPreferences`.

В следующей главе мы углубимся в мир тетриса, реализуем игровой процесс, а также решающую логику приложения.

3

Реализация логики и функциональности игры Tetris

В предыдущей главе мы начали разработку классической игры Tetris. Были определены требования к макету приложения и реализованы предварительно заданные элементы макета, созданы два действия для приложения: `MainActivity` и `GameActivity`, а также определены основные характеристики и поведение представлений, но пока не сделано ничего, что относилось бы к основному игровому процессу приложения. Так что в этой главе речь пойдет о реализации этого игрового процесса. Мы рассмотрим также и следующие темы:

- обработка исключений;
- шаблон «модель-представление-презентатор» (Model-View-Presenter).

Реализация игрового процесса Tetris

Поскольку нам необходимо реализовать игровой процесс, мы здесь сосредоточимся на дальнейшей разработке действия `GameActivity`. На рис. 3.1 показано, как в конечном итоге должна выглядеть наша игра.

В *разд. «Tetris — правила игры» главы 2* мы договорились, что Tetris — это игра-головоломка, которая основана на упорядочении падающих фигур — *тетрамино́*, каждая из которых состоит из четырех соединенных различными сторонами цветных блоков-плиток.

Моделирование тетрамино

Поскольку поведение тетрамино очень важно для игрового процесса Tetris, нужно правильно программно смоделировать эти элементы. Чтобы добиться этого, представим каждую часть тетрамино в виде некоего «строительного блока». Эти блоки содержат характерный присущий только им набор функций, подразделяемых на характеристики и поведение.

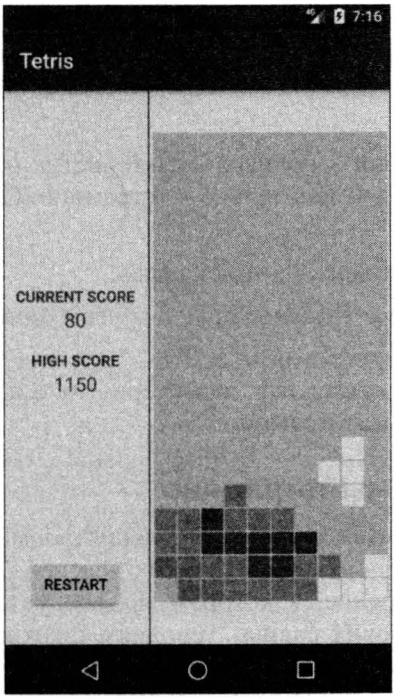


Рис. 3.1. Так должна выглядеть полностью разработанная игра

Характеристики блока

Рассмотрим некоторые характеристики, присущие блокам, составляющим тетрамино:

- *форма* — блок имеет фиксированную форму, которую изменять нельзя;
- *размерности* — блок обладает характеристиками размерности, за которые мы примем их высоту и ширину;
- *цвет* — блок всегда окрашен. Цвет блока фиксирован и поддерживается все время его существования;
- *пространственная характеристика* — блок занимает фиксированное пространство;
- *позиционная характеристика* — в любой момент времени блок располагает позицией, которая существует вдоль двух осей: X и Y.

Поведения блока

Основным поведением блока является его способность испытывать различные движения: поступательное и вращательное. *Поступательное движение* — это тип движения, когда тело перемещается из одной точки пространства в другую. В Tetris блок может характеризоваться поступательными движениями влево, вправо и вниз.

Вращательное движение — это присущий твердым телам тип движения, которое выполняется по криволинейной траектории. Другими словами, вращательное движение предполагает вращение объекта в свободном пространстве. Вращать в Tetris можно все блоки.

Теперь, после представления основных характеристик и вариантов поведения блока, необходимо учесть их применительно к тетрамино. Следует добавить лишь следующие два момента:

- тетрамино состоят из четырех плиток каждая;
- все плитки в тетрамино соединены друг с другом одной или несколькими сторонами.

Далее мы приступим к переводу этих характеристик в программные модели и начнем с моделирования формы тетрамино.

Моделирование формы тетрамино

Подход, который применяется для моделирования формы, варьируется в зависимости от многочисленных переменных, таких как вид рассматриваемой формы и пространственное измерение, в котором моделируется форма. Моделирование трехмерных форм, при прочих равных условиях, сложнее моделирования двумерных. К счастью, тетрамино представляют собой двумерные фигуры. Но прежде чем моделировать те или иные формы программными методами, важно точно представлять себе, как они должны выглядеть на экране. Пусть в разрабатываемой нами игре Tetris имеется семь основных блоков тетрамино: O, I, T, L, J, S и Z (рис. 3.2).

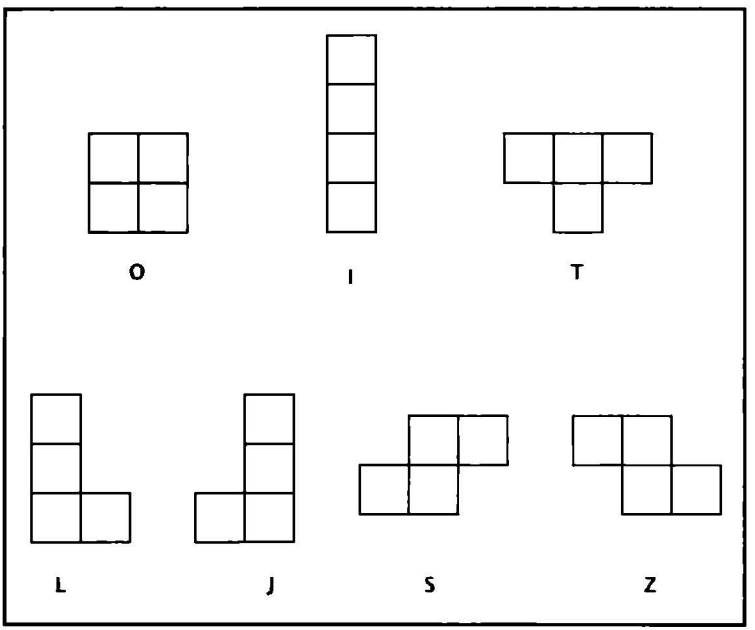


Рис. 3.2. Семь основных блоков тетрамино

Все показанные на рис. 3.2 формы блоков тетрамино занимают пространство в пределах своих границ. Область пространства, занятая формой, может отображаться как контур, или фрейм. Это схоже с размещением изображения в кадре. Следует смоделировать этот фрейм, который и будет представлять собой отдельную форму. Поскольку отдельные плитки, составляющие тетрамино и входящие во фрейм, представляют собой двумерные объекты, для хранения информации, специфичной для фрейма, применим двумерный байтовый массив. *Байт* — это цифровая единица информации, состоящая из восьми битов. *Бит* — это двоичная цифра, наименьшая единица данных в компьютере, которая принимает значение 1 или 0.

Идея состоит в моделировании формы фрейма с помощью двумерного массива путем представления занятых плитками областей во фрейме значением байта, равном 1, а тех, что не заняты плитками, — значением 0. Рассмотрим, например, следующий фрейм (рис. 3.3, *а*).

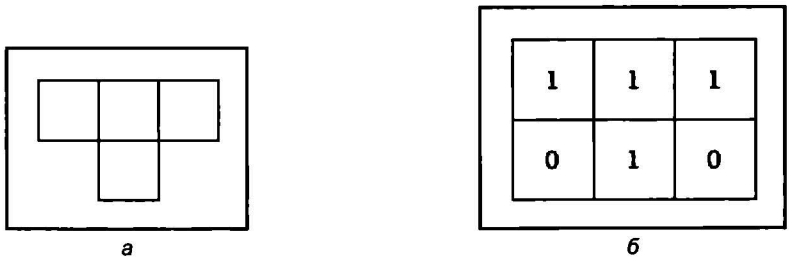


Рис. 3.3. Пример фрейма (*а*) и визуализация его в виде двумерного массива байтов (*б*)

Вместо визуализации его как целой формы можно визуализировать его как двумерный массив байтов, охватывающий две строки и три столбца (рис. 3.3, *б*).

Значение байта, равное 1, присваивается ячейкам массива, образующим форму фрейма. Значение байта, равное 0, присваивается ячейкам, не являющимся частью формы фрейма. Нетрудно смоделировать это с помощью класса. Во-первых, необходимо воспользоваться функцией, генерирующей необходимую структуру массива байтов, которая будет применяться для хранения байтов фрейма. Создайте в исходном пакете новый пакет и назовите его именем, включающим слово `helper`. В этом пакете создайте файл HelperFunctions.kt. В него мы включим все используемые в ходе разработки этого приложения вспомогательные функции. Откройте файл HelperFunctions.kt и введите в него следующий код:

```
package com.mydomain.tetris.helpers

fun array2dOfByte(sizeOuter: Int, sizeInner: Int): Array<ByteArray>
            = Array(sizeOuter) { ByteArray(sizeInner) }
```

Указанный код определяет функцию `array2dOfByte()` с двумя аргументами. Первый аргумент — это номер строки создаваемого массива, а второй — номер столбца сгенерированного массива байтов. Метод `array2dOFByte()` генерирует и возвращает новый массив с указанными свойствами. Теперь, настроив вспомогательную функ-

цию генерации байтового массива, пойдем дальше и создадим класс Frame. Организуйте в исходном пакете новый пакет и назовите его models. Все объектные модели будут включаться в этот новый пакет. И уже внутри пакета models в файле Frame.kt создайте класс Frame и введите в этот файл следующий код:

```
package com.mydomain.tetris.models

import com.mydomain.tetris.helpers.array2dOfByte

class Frame(private val width: Int) {
   val data: ArrayList<ByteArray> = ArrayList()

   fun addRow(byteStr: String): Frame {
      val row = ByteArray(byteStr.length)

      for (index in byteStr.indices) {
         row[index] = "${byteStr[index]}".toByte()
      }
      data.add(row)
      return this
   }

   fun as2dByteArray(): Array<ByteArray> {
      val bytes = array2dOfByte(data.size, width)
      return data.toArray(bytes)
   }
}
```

Класс Frame содержит два свойства: width и data. Свойство width является целочисленным свойством, которое задает необходимую ширину генерируемого фрейма (число столбцов в байтовом массиве фрейма). Свойство data содержит список элементов массива в пространстве значений ByteArray. Нужно объявить две различные функции: addRow() и get(). Функция addRow() обрабатывает строку, преобразуя каждый отдельный символ строки в байтовое представление, и добавляет байтовое представление в байтовый массив, после чего байтовый массив добавляется в список данных. Функция get() преобразует список массива данных в байтовый массив, который затем возвращает.

Разобравшись с общим принципом моделирования фреймов для сохранения блоков тетрамино, можно смоделировать формы различных вариантов тетрамино. Для этого применим класс enum. Перед выполнением дальнейшей обработки создадим в пакете models файл Shape.kt. И начнем с моделирования следующей простой формы тетрамино (рис. 3.4).

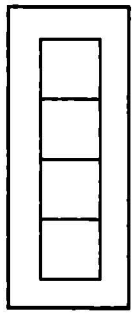


Рис. 3.4. Простая форма тетрамино для моделирования

Применяя концепцию представления фреймов в виде двумерного массива байтов, можно представить фрейм этой формы в виде двумерного массива байтов с четырьмя строками и одним столбцом, каждая ячейка которого заполнена байтовым значением, равным 1. Учитывая этот момент, смоделируем форму. В файле Shape.kt создадим класс enum Shape, как показано в следующем коде:

```
enum class Shape(val frameCount: Int, val startPosition: Int) {
   Tetromino(2, 2) {
      override fun getFrame(frameNumber: Int): Frame {
         return when (frameNumber) {
            0 -> Frame(4).addRow("1111")
            1 -> Frame(1)
                        .addRow("1")
                        .addRow("1")
                        .addRow("1")
                        .addRow("1")
            else -> throw IllegalArgumentException("$frameNumber
                                    is an invalid frame number.")
         }
      }
   };
   abstract fun getFrame(frameNumber: Int): Frame
}
```

Класс enum объявляется путем размещения ключевого слова enum перед ключевым словом class. Основной конструктор показанного в этом фрагменте кода класса enum Shape включает два аргумента. Первым аргументом служит frameCount — целочисленная переменная, указывающая число возможных фреймов, в которых может находиться форма. Второй аргумент startPosition указывает предполагаемую начальную позицию формы вдоль оси X в поле игрового процесса. Далее в классе enum объявляется функция getFrame(). Имеется существенное различие между этой функцией и функциями, которые объявлялись раннее. Функция getFrame() здесь объявлена с помощью ключевого слова abstract. Абстрактная функция не имеет реализации (следовательно, не имеет тела) и применяется для абстрагирования поведения, которое должно быть реализовано расширяющим классом. Рассмотрим следующие строки кода в классе enum:

```
Tetromino(2, 2) {
   override fun getFrame(frameNumber: Int): Frame {
      return when (frameNumber) {
         0 -> Frame(4).addRow("1111")
         1 -> Frame(1)
                     .addRow("1")
                     .addRow("1")
                     .addRow("1")
                     .addRow("1")
```

```
            else -> throw IllegalArgumentException("$frameNumber
                                    is an invalid frame number.")
        }
    }
};
```

В приведенном блоке кода создается экземпляр класса `enum`, который обеспечивает реализацию объявленной абстрактной функции. Идентификатором экземпляра является `Tetromino`. Целочисленное значение, равное `2`, передается в качестве аргумента для обоих свойств: `frameCount` и `startPosition` — конструктора для объекта `Tetromino`. Кроме того, `Tetromino` поддерживает реализацию для функции `getFrame()` в соответствующем блоке путем переопределения функции `getFrame()`, объявленной в `Shape`. Функции переопределяются с помощью ключевого слова `override`. Реализация функции `getFrame()` в `Tetromino` использует целочисленную переменную `frameNumber`. Эта переменная определяет фрейм `Tetromino`, который будет возвращаться. Может возникнуть вопрос, почему `Tetromino` располагает более, чем одним фреймом. Это результат возможности вращения тетрамино. Тетрамино в одну колонку, представленное на рис. 3.4, можно повернуть влево или вправо, что позволит получить форму, показанную на рис. 3.5.

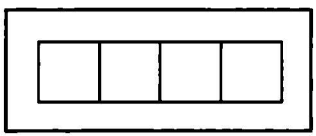


Рис. 3.5. Форма тетрамино, представленная на рис. 3.4, повернута влево или вправо

Если переменная `frameNumber` передает функции `getFrame()` значение `0`, функция `getFrame()` возвращает объект `Frame`, моделирующий фрейм для `Tetromino` в горизонтальном положении (см. рис. 3.5). Если значением `frameNumber` является `1`, она возвращает объект фрейма, моделирующий форму в вертикальном положении (см. рис. 3.4).

Если значением `frameNumber` не является ни значение `0`, ни значение `1`, `IllegalArgumentException` отбрасывается функцией.

Важно отметить, что `Tetromino` является одновременно и объектом, и константой. Как правило, классы `enum` используются для создания констант. Класс `enum` является идеальным выбором для моделирования форм тетрамино, поскольку в этом случае может реализовываться фиксированный набор форм.

Уяснив, как работает класс `enum Shape`, можно смоделировать остальные возможные формы тетрамино:

```
enum class Shape(val frameCount: Int, val startPosition: Int) {
```

Создадим форму тетрамино с одним фреймом и начальной позицией `1`. Моделируемое здесь тетрамино имеет квадратную форму (см. рис. 3.2, вариант O):

```
Tetromino1(1, 1) {
   override fun getFrame(frameNumber: Int): Frame {
      return Frame(2)
             .addRow("11")
             .addRow("11")
   }
},
```

Создадим форму тетрамино с двумя фреймами и начальной позицией 1. Моделируемое здесь тетрамино имеет Z-образную форму (см. рис. 3.2, вариант Z):

```
Tetromino2(2, 1) {
   override fun getFrame(frameNumber: Int): Frame {
      return when (frameNumber) {
         0 -> Frame(3)
                    .addRow("110")
                    .addRow("011")
         1 -> Frame(2)
                    .addRow("01")
                    .addRow("11")
                    .addRow("10")
         else -> throw IllegalArgumentException("$frameNumber
                                    is an invalid frame number.")
      }
   }
},
```

Создадим форму тетрамино с двумя фреймами и начальной позицией 1. Моделируемое здесь тетрамино имеет S-образную форму (см. рис. 3.2, вариант S):

```
Tetromino3(2, 1) {
   override fun getFrame(frameNumber: Int): Frame {
      return when (frameNumber) {
         0 -> Frame(3)
                    .addRow("011")
                    .addRow("110")
         1 -> Frame(2)
                    .addRow("10")
                    .addRow("11")
                    .addRow("01")
         else -> throw IllegalArgumentException("$frameNumber is
                                    an invalid frame number.")
      }
   }
},
```

Создадим форму тетрамино с двумя фреймами и начальной позицией 2. Моделируемое здесь тетрамино имеет I-образную форму (см. рис. 3.2, вариант I):

```
Tetromino4(2, 2) {
   override fun getFrame(frameNumber: Int): Frame {
      return when (frameNumber) {
         0 -> Frame(4).addRow("1111")
         1 -> Frame(1)
                    .addRow("1")
                    .addRow("1")
                    .addRow("1")
                    .addRow("1")
         else -> throw IllegalArgumentException("$frameNumber is an
                                                invalid frame number.")
      }
   }
},
```

Создадим форму тетрамино с четырьмя фреймами и начальной позицией 1. Моделируемое здесь тетрамино имеет Т-образную форму (см. рис. 3.2, вариант Т):

```
Tetromino5(4, 1) {
   override fun getFrame(frameNumber: Int): Frame {
      return when (frameNumber) {
         0 -> Frame(3)
                    .addRow("010")
                    .addRow("111")
         1 -> Frame(2)
                    .addRow("10")
                    .addRow("11")
                    .addRow("10")
         2 -> Frame(3)
                    .addRow("111")
                    .addRow("010")
         3 -> Frame(2)
                    .addRow("01")
                    .addRow("11")
                    .addRow("01")
         else -> throw IllegalArgumentException("$frameNumber is an
                                                invalid frame number.")
      }
   }
},
```

Создадим форму тетрамино с четырьмя фреймами и начальной позицией 1. Моделируемое здесь тетрамино имеет J-образную форму (см. рис. 3.2, вариант J).

```
Tetromino6(4, 1) {
   override fun getFrame(frameNumber: Int): Frame {
      return when (frameNumber) {
```

```
            0 -> Frame(3)
                        .addRow("100")
                        .addRow("111")
            1 -> Frame(2)
                        .addRow("11")
                        .addRow("10")
                        .addRow("10")
            2 -> Frame(3)
                        .addRow("111")
                        .addRow("001")
            3 -> Frame(2)
                        .addRow("01")
                        .addRow("01")
                        .addRow("11")
            else -> throw IllegalArgumentException("$frameNumber is
                                                an invalid frame number.")
        }
    }
},
```

Создадим форму тетрамино с четырьмя фреймами и начальной позицией 1. Моделируемое здесь тетрамино имеет L-образную форму (см. рис. 3.2, вариант L):

```
Tetromino7(4, 1) {
    override fun getFrame(frameNumber: Int): Frame {
        return when (frameNumber) {
            0 -> Frame(3)
                        .addRow("001")
                        .addRow("111")
            1 -> Frame(2)
                        .addRow("10")
                        .addRow("10")
                        .addRow("11")
            2 -> Frame(3)
                        .addRow("111")
                        .addRow("100")
            3 -> Frame(2)
                        .addRow("11")
                        .addRow("01")
                        .addRow("01")
            else -> throw IllegalArgumentException("$frameNumber is
                                                an invalid frame number.")
            }
        }
    };
    abstract fun getFrame(frameNumber: Int): Frame
}
```

Смоделировав фрейм блока и форму, необходимо программно смоделировать сам блок. Рассмотрим эту задачу как возможность демонстрации совместимости языка Kotlin с языком Java путем реализации модели с помощью Java. Создайте новый класс Java в каталоге models (**models | New | Java Class**) под именем `Block`. Приступим к процессу моделирования, добавляя переменные экземпляра, представляющие характеристики блока:

```
package com.mydomain.tetris.models;
import android.graphics.Color;
import android.graphics.Point;

public class Block {
   private int shapeIndex;
   private int frameNumber;
   private BlockColor color;
   private Point position;
   public enum BlockColor {
      PINK(Color.rgb(255, 105, 180), (byte) 2),
      GREEN(Color.rgb(0, 128, 0), (byte) 3),
      ORANGE(Color.rgb(255, 140, 0), (byte) 4),
      YELLOW(Color.rgb(255, 255, 0), (byte) 5),
      CYAN(Color.rgb(0, 255, 255), (byte) 6);
      BlockColor(int rgbValue, byte value) {
         this.rgbValue = rgbValue;
         this.byteValue = value;
      }

      private final int rgbValue;
      private final byte byteValue;
   }
}
```

В указанном блоке кода добавлены четыре переменные экземпляра: `shapeIndex`, `frameNumber`, `color` и `position`. Переменная `shapeIndex` включает индекс формы блока; переменная `frameNumber` отслеживает количество фреймов, относящихся к форме блока; переменная `color` содержит цветовые характеристики блока; а переменная `position` используется для отслеживания текущей пространственной позиции блока в игровом поле.

Шаблон `enum` — `BlockColor` — добавляется внутри класса `Block`. Этот шаблон `enum` формирует постоянный набор экземпляров `BlockColor`, каждый из которых обладает свойствами `rgbValue` и `byteValue`. Свойство `rgbValue` является целым числом, которое однозначно определяет цвет RGB, указанный с помощью метода `Color.rgb()`. Ну а `Color` служит классом, предоставляемым платформой приложения Android, где `rgb()` — это метод класса, определенный внутри класса `Color`. Пять вызовов `Color.rgb()` определяют соответственно цвета: розовый, зеленый, оранжевый, желтый и голубой.

В классе `Block` мы пользуемся ключевыми словами `private` и `public`. Они совершенно необходимы, и каждое имеет смысл. Эти два ключевых слова вместе с ключевым словом `protected` называют *модификаторами доступа*.

Модификаторы доступа — это ключевые слова, используемые для указания ограничений доступа к классам, методам, функциям, переменным и структурам. В языке Java имеются три модификатора доступа: `private`, `public` и `protected`. В языке Kotlin модификаторы доступа называются *модификаторами видимости*. Доступными модификаторами видимости в Kotlin являются `public`, `protected`, `private` и `internal`.

Модификатор частного доступа: *private*

Методы, переменные, конструкторы и структуры, объявленные частными, могут быть доступны только в пределах декларирующего их класса. Это правило работает во всех случаях, за исключением частных функций и свойств верхнего уровня, которые видны всем членам одного файла. Частные переменные внутри класса могут быть доступны вне класса путем объявления методов геттеров и сеттеров, разрешающих доступ. Определение методов *сеттеров* и *геттеров* (установки и получения) в Java показано в следующем коде.

```
public class Person {
   Person(String fullName, int age) {
      this.fullName = fullName;
      this.age = age;
   }

   private String fullName;
   private int age;
   public String getFullName() {
      return fullName;
   }

   public int getAge() {
      return age;
   }
}
```

В языке Kotlin создание методов сеттеров и геттеров выполняется следующим образом:

```
public class Person(private var fullName: String) {
   var name: String
   get() = fullName
   set(value) {
      fullName = value
   }
}
```

Использование модификатора `private` служит основным методом для сокрытия информации в программах. Сокрытие информации также известно как *процедура инкапсуляции*.

Модификатор общего доступа: *public*

Методы, переменные, конструкторы и структуры, объявленные общедоступными, находятся в свободном доступе и за пределами декларирующего их класса. Общедоступный (публичный) класс, находящийся в другом, относительно класса доступа, пакете, должен быть импортирован перед возможным его применением. В следующем классе применяется модификатор доступа `public`:

```
public class Person { .. }
```

Модификатор защищенного доступа: *protected*

Переменные, методы, функции и структуры, объявленные защищенными, доступны только классам в том же пакете, что и определяющий их класс, или же подклассам для определяющего их класса, которые находятся в отдельном пакете:

```
public class Person(private var fullName: String) {
   protected name: String
   get() = fullName
   set(value) {
      fullName = value
   }
}
```

Модификатор внутренней видимости: *internal*

Модификатор внутренней видимости используется для объявления элемента, видимого в том же модуле. Модуль представляет набор скомпилированных вместе файлов Kotlin. Модуль может быть проектом Maven, исходным набором Gradle, модулем IntelliJ IDEA или набором файлов, скомпилированных с помощью вызова для задачи Ant. Применение модификатора внутренней видимости происходит следующим образом:

```
internal class Person { }
```

* * *

Четко представляя функции модификаторов доступа и видимости, можно продолжить реализацию класса `Block`. Нам необходимо создать конструктор для класса, который инициализирует созданные нами переменные экземпляра, указывая их начальные состояния. Определения конструктора в Java синтаксически отличаются от определений конструктора в Kotlin:

```
public class Block {
   private int shapeIndex;
   private int frameNumber;
```

```
   private BlockColor color;
   private Point position;
```

Рассмотрим определение конструктора:

```
private Block(int shapeIndex, BlockColor blockColor) {
   this.frameNumber = 0;
   this.shapeIndex = shapeIndex;
   this.color = blockColor;
   this.position = new Point(AppModel.FieldConstants
                             .COLUMN_COUNT.getValue()/2, 0);
}

public enum BlockColor {
   PINK(Color.rgb(255, 105, 180), (byte) 2),
   GREEN(Color.rgb(0, 128, 0), (byte) 3),
   ORANGE(Color.rgb(255, 140, 0), (byte) 4),
   YELLOW(Color.rgb(255, 255, 0), (byte) 5),
   CYAN(Color.rgb(0, 255, 255), (byte) 6);
   BlockColor(int rgbValue, byte value) {
      this.rgbValue = rgbValue;
      this.byteValue = value;
   }
   private final int rgbValue;
   private final byte byteValue;
   }
}
```

Обратите внимание, что для предыдущего определения конструктора предоставлен частный доступ. С нашей точки зрения, нежелательно, чтобы этот конструктор был доступен вне класса `Block`. Поскольку нужно, чтобы другие классы имели возможности для создания экземпляра блока, следует определить статический метод, который это разрешает. Назовем этот метод `createBlock`:

```
public class Block {
   private int shapeIndex;
   private int frameNumber;
   private BlockColor color;
   private Point position;
```

Рассмотрим определение конструктора класса:

```
private Block(int shapeIndex, BlockColor blockColor) {
   this.frameNumber = 0;
   this.shapeIndex = shapeIndex;
</span> this.color = blockColor;
   this.position = new Point( FieldConstants.COLUMN_COUNT.getValue()/2, 0);
}
```

```
public static Block createBlock() {
   Random random = new Random();
   int shapeIndex = random.nextInt(Shape.values().length);
   BlockColor blockColor = BlockColor.values()
        [random.nextInt(BlockColor.values().length)];

   Block block = new Block(shapeIndex, blockColor);
   block.position.x = block.position.x - Shape.values()
         [shapeIndex].getStartPosition();
   return block;
}

public enum BlockColor {
   PINK(Color.rgb(255, 105, 180), (byte) 2),
   GREEN(Color.rgb(0, 128, 0), (byte) 3),
   ORANGE(Color.rgb(255, 140, 0), (byte) 4),
   YELLOW(Color.rgb(255, 255, 0), (byte) 5),
   CYAN(Color.rgb(0, 255, 255), (byte) 6);
   BlockColor(int rgbValue, byte value) {
      this.rgbValue = rgbValue;
      this.byteValue = value;
   }

   private final int rgbValue;
   private final byte byteValue;
   }
}
```

Метод `createBlock()` случайным образом выбирает индекс для формы тетрамино в классе `enum Shape` и `BlockColor`, а затем присваивает два случайно выбранных значения элементам `shapeIndex` и `blockColor`. Новый экземпляр `Block` создается с двумя переданными ему в качестве аргументов значениями, и позиция блока устанавливается вдоль оси X. Наконец, метод `createBlock()` возвращает созданный и инициализированный блок.

Необходимо добавить в класс `Block` несколько методов геттеров и сеттеров (получения и установки). Эти методы позволят получать доступ к важнейшим свойствам экземпляров блока. Добавьте следующие методы в класс `Block`:

```
public static int getColor(byte value) {
   for (BlockColor colour : BlockColor.values()) {
      if (value == colour.byteValue) {
         return colour.rgbValue;
      }
   }
   return -1;
}
```

```
public final void setState(int frame, Point position) {
   this.frameNumber = frame;
   this.position = position;
}

@NonNull
   public final byte[][] getShape(int frameNumber) {
      return Shape.values()[shapeIndex].getFrame(
         frameNumber).as2dByteArray();
   }

public Point getPosition() {
   return this.position;
}

public final int getFrameCount() {
   return Shape.values()[shapeIndex].getFrameCount();
}

public int getFrameNumber() {
   return frameNumber;
}

public int getColor() {
   return color.rgbValue;
}

public byte getStaticValue() {
   return color.byteValue;
}
```

Аннотация `@NonNull` предоставляется платформой приложения Android и обозначает, что возвращаемое поле, параметр или метод не могут иметь значение `null`. В приведенном фрагменте кода эта аннотация используется в строке перед определением метода `getShape()`, позволяя указать на тот факт, что метод не может возвращать нулевое значение.

В Java *аннотация* — это форма метаданных, которая может добавляться в исходный код Java. Аннотации могут применяться для классов, методов, переменных, параметров и пакетов. Аннотации также могут объявляться и применяться в Kotlin.

Аннотация `@NotNull` существует в составе пакета android.support.annotation. Добавьте импорт этого пакета в область импорта пакетов в верхней части файла Block.java:

```
import android.support.annotation.NonNull;
```

Имеется еще один момент, о котором необходимо позаботиться в классе `Block`. В последней строке конструктора `Block` положение переменной экземпляра позиции для текущего экземпляра блока задается следующим образом:

```
this.position = new Point(FieldConstants.COLUMN_COUNT.getValue()/2, 0);
```

Дело в том, что количество столбцов поля, где станут генерироваться тетрамино, определяется значением, равным `10`. Это постоянное значение будет в коде нашего приложения использоваться несколько раз, и поэтому удобно объявить его как константу. Создайте в исходном пакете базового приложения пакет с именем constants и добавьте в него новый файл Kotlin с именем FieldConstants. Затем добавьте в него константы для числа столбцов и строк игрового поля. Это поле имеет десять столбцов и двадцать строк:

```
enum class FieldConstants(val value: Int) {
   COLUMN_COUNT(10), ROW_COUNT(20);
}
```

Импортируйте этот пакет в файл Block.java с помощью enum-класса `FieldConstants` и замените целое `10` постоянным значением `COLUMN_COUNT`:

```
this.position = new Point(FieldConstants.COLUMN_COUNT.getValue()/2, 0);
```

Вот и все! На этом программное моделирование класса `Block` завершено.

Создание модели приложения

До сих пор речь шла о моделировании конкретных компонентов, составляющих блоки тетрамино. Теперь займемся определением логики приложения. Создадим модель приложения, реализующую необходимую логику игрового поля Tetris, а также обслуживающую промежуточный интерфейс между представлениями и созданными компонентами блока (рис. 3.6).

Рис. 3.6. Взаимодействие представления и модели приложения

Представление отправит модели приложения запрос на выполнение, модель выполнит действие, если оно допустимо, и отправит отзыв представлению. Подобно созданным до сих пор моделям, для модели приложения необходим отдельный файл класса. Поэтому создадим новый файл Kotlin с именем AppModel.kt и добавим

к файлу класс под названием AppModel с импортом элементов Point, FieldConstants, array2dOfByte и AppPreferences:

```
package com.mydomain.tetris.models

import android.graphics.Point
import com.mydomain.tetris.constants.FieldConstants
import com.mydomain.tetris.helpers.array2dOfByte
import com.mydomain.tetris.storage.AppPreferences

class AppModel
```

Некоторые функции класса AppModel отслеживают текущий счет, состояние игрового поля приложения Tetris, текущий блок, текущее состояние игры, текущий статус игры и движений, которые выполняет текущий блок. AppModel также должен иметь прямой доступ к значениям, хранящимся в файле SharedPreferences приложения посредством созданного класса AppPreferences. Удовлетворение столь разных требований немного настораживает, но все не так сложно, как кажется на первый взгляд.

Сначала следует добавить константы, которые будут использоваться классом AppModel, — понадобятся константы для возможных игровых статусов и возможных движений, выполняемых в процессе игры. Эти константы легко создаются с помощью классов enum:

```
class AppModel {
   enum class Statuses {
      AWAITING_START, ACTIVE, INACTIVE, OVER
   }

   enum class Motions {
      LEFT, RIGHT, DOWN, ROTATE
   }
}
```

Мы здесь создали четыре константы состояния: константа AWAITING_START представляет состояние игры до ее запуска, константы ACTIVE и INACTIVE представляют состояние игрового процесса (если этот процесс выполняется или нет), константа OVER — статус, принимаемый игрой на момент ее завершения.

Ранее в этой главе указано, что блок может участвовать в четырех различных движениях — их можно перемещать вправо, влево, вверх/вниз и выполнять вращение. Константы LEFT, RIGHT, UP, DOWN и ROTATE определены в enum-классе Motions для представления этих различных движений.

Добавление необходимых констант можно продолжить, включая необходимые свойства класса для AppModel, а именно:

```
package com.mydomain.tetris.models
import android.graphics.Point
```

```
import com.mydomain.tetris.constants.FieldConstants
import com.mydomain.tetris.helpers.array2dOfByte
import com.mydomain.tetris.storage.AppPreferences

class AppModel {
   var score: Int = 0
   private var preferences: AppPreferences? = null

   var currentBlock: Block? = null
   var currentState: String = Statuses.AWAITING_START.name

   private var field: Array<ByteArray> = array2dOfByte(
      FieldConstants.ROW_COUNT.value,
      FieldConstants.COLUMN_COUNT.value
   )

   enum class Statuses {
      AWAITING_START, ACTIVE, INACTIVE, OVER
   }

   enum class Motions {
      LEFT, RIGHT, DOWN, ROTATE
   }
}
```

Константа `score` относится к целочисленному свойству, применяемому для сохранения текущего счета игрока в игровом сеансе. Константа `preferences` — к частному свойству, которое будет содержать объект `AppPreferences` для обеспечения непосредственного доступа к файлу `SharedPreferences` приложения. Константа `currentBlock` — к свойству, которое будет содержать текущий блок трансляции через игровое поле. Константа `currentState` содержит состояние игры, а константа `Statuses.AWAITING_START.name` возвращает имя `Statuses.AWAITING_START` в форме строки `AWAITING_START`. Текущее состояние игры немедленно инициализируется значением `AWAITING_START`, поскольку это первое состояние, в которое должно перейти действие `GameActivity` после запуска. И наконец, константа `field` представляет двумерный массив, служащий в качестве игрового поля.

Далее следует добавить несколько методов сеттеров и геттеров: `setPreferences()`, `setCellStatus()` и `getCellStatus()`. Добавьте к модели `AppModel` следующие методы:

```
fun setPreferences(preferences: AppPreferences?) {
   this.preferences = preferences
}

fun getCellStatus(row: Int, column: Int): Byte? {
   return field[row][column]
}
```

```
private fun setCellStatus(row: Int, column: Int, status: Byte?) {
   if (status != null) {
      field[row][column] = status
   }
}
```

Метод `setPreferences()` устанавливает свойство предпочтений для `AppModel`, что позволит передать экземпляр класса `AppPreferences` в виде аргумента этой функции. Метод `getCellStatus()` возвращает состояние ячейки, имеющейся в указанной позиции столбца строки в двумерном массиве поля. Метод `setCellStatus()` устанавливает состояние имеющейся в поле ячейки равным указанному байту. Функции для проверки состояния в модели также необходимы — они служат средой для подтверждения состояния, в котором находится игра в текущий момент.

Поскольку имеются три возможных статуса игры, которым соответствуют три игровых состояния, то для каждого из них необходимо наличие трех функций. В качестве этих трех функций используются методы `isGameAwaitingStart()`, `isGameActive()` и `isGameOver()`:

```
class AppModel {

   var score: Int = 0
   private var preferences: AppPreferences? = null

   var currentBlock: Block? = null
   var currentState: String = Statuses.AWAITING_START.name

   private var field: Array<ByteArray> = array2dOfByte(
      FieldConstants.ROW_COUNT.value,
      FieldConstants.COLUMN_COUNT.value
)

fun setPreferences(preferences: AppPreferences?) {
   this.preferences = preferences
}

fun getCellStatus(row: Int, column: Int): Byte? {
   return field[row][column]
}

private fun setCellStatus(row: Int, column: Int, status: Byte?) {
   if (status != null) {
      field[row][column] = status
   }
}
```

```
fun isGameOver(): Boolean {
    return currentState == Statuses.OVER.name
}

fun isGameActive(): Boolean {
    return currentState == Statuses.ACTIVE.name
}

fun isGameAwaitingStart(): Boolean {
    return currentState == Statuses.AWAITING_START.name
}

enum class Statuses {
    AWAITING_START, ACTIVE, INACTIVE, OVER
}

enum class Motions {
    LEFT, RIGHT, DOWN, ROTATE
}
}
```

Все три метода возвращают логические значения `true` или `false` в зависимости от соответствующего состояния, в котором находится игра. До сих пор в `AppModel` не применялось константа `score`. Добавим функцию, которую можно использовать для увеличения значения очков, находящегося в некотором диапазоне. Назовем эту функцию `boostScore()`:

```
private fun boostScore() {
    score += 10
    if (score > preferences?.getHighScore() as Int)
    preferences?.saveHighScore(score)
}
```

При вызове функция `boostScore()` увеличивает текущий счет игрока на 10 очков, после чего проверяется, не превышает ли текущий счет игрока уже установленный рекорд, записанный в файле настроек. Если текущая оценка больше сохраненного рекорда, рекорд переписывается с учетом текущего значения очков.

Начиная работу с основными функциями и полями, можно перейти к созданию более сложных функций. Первая из этих функций — `generateNextBlock()`:

```
private fun generateNextBlock() {
    currentBlock = Block.createBlock()
}
```

Функция `generateNextBlock()` создает новый экземпляр блока и устанавливает значение `currentBlock` равным вновь созданному экземпляру.

Прежде чем перейти к определениям методов, создадим еще один класс enum для хранения постоянных значений ячеек. В пакете констант создайте файл CellConstants.kt и добавьте в него следующий исходный код:

```
package com.mydomain.tetris.constants

enum class CellConstants(val value: Byte) {
   EMPTY(0), EPHEMERAL(1)
}
```

Вам может быть интересно, для чего нужны эти константы. Вспомните, при создании класса Frame при моделировании фрейма блоков определялась функция addRow(), которая в качестве аргумента обрабатывает строку, состоящую из 1 и 0. Значение, равное 1, представляет составляющие фрейм ячейки, а значение, равное 0, представляет ячейки, исключенные из фрейма, а затем преобразует данные значения, равные 1 и 0, в байтовые представления. Этими байтами мы и будем манипулировать в функциях, и для них необходимо располагать соответствующими константами.

Импортируйте вновь созданный класс enum в AppModel. Применим его в следующей функции:

```
private fun validTranslation(position: Point, shape: Array<ByteArray>):
Boolean {
   return if (position.y < 0 || position.x < 0) {
      false
   } else if (position.y + shape.size > FieldConstants.ROW_COUNT.value) {
      false
   } else if (position.x + shape[0].size > FieldConstants
            .COLUMN_COUNT.value) {
      false
   } else {
      for (i in 0 until shape.size) {
         for (j in 0 until shape[i].size) {
            val y = position.y + i
            val x = position.x + j
            if (CellConstants.EMPTY.value != shape[i][j] &&
               CellConstants.EMPTY.value != field[y][x]) {
               return false
            }
         }
      }
      true
   }
}
```

Добавьте в модель AppModel приведенный здесь метод validTranslation(). Как следует из названия, эта функция служит для проверки допустимости поступательного

движения тетрамино в игровом поле на основе набора условий. Метод возвращает логическое значение `true`, если трансляция корректна, и `false` — в противном случае. Первые три условия проверяют, находится ли в поле позиция, в которую переводится тетрамино. Блок `else` проверяет, свободны ли клетки, в которые пытается перейти тетрамино. Если это не так, возвращается значение `false`.

Для использования метода `validTranslation()` нужна функция вызова. С этой целью объявим функцию `moveValid()`. Добавим к модели `AppModel` следующую функцию:

```
private fun moveValid(position: Point, frameNumber: Int?): Boolean {
   val shape: Array<ByteArray>? = currentBlock?
            .getShape(frameNumber as Int)
   return validTranslation(position, shape as Array<ByteArray>)
}
```

Функция `moveValid()` применяет функцию `validTranslation()`, которая проверяет, разрешен ли выполненный игроком ход. Если перемещение разрешено, возвращается значение `true`, в противном случае — значение `false`. Также нужно создать еще несколько важных методов, а именно методы: `generateField()`, `resetField()`, `persistCellData()`, `assessField()`, `translateBlock()`, `blockAdditionPossible()`, `shiftRows()`, `startGame()`, `restartGame()`, `endGame()` и `resetModel()`.

Сначала поработаем с методом `generateField()`. Добавьте к модели `AppModel` следующий код:

```
fun generateField(action: String) {
   if (isGameActive()) {
      resetField()
      var frameNumber: Int? = currentBlock?.frameNumber
      val coordinate: Point? = Point()
      coordinate?.x = currentBlock?.position?.x
      coordinate?.y = currentBlock?.position?.y

      when (action) {
         Motions.LEFT.name -> {
            coordinate?.x = currentBlock?.position?.x?.minus(1)
         }
         Motions.RIGHT.name -> {
            coordinate?.x = currentBlock?.position?.x?.plus(1)
         }

         Motions.DOWN.name -> {
            coordinate?.y = currentBlock?.position?.y?.plus(1)
         }

         Motions.ROTATE.name -> {
            frameNumber = frameNumber?.plus(1)
```

```
            if (frameNumber != null) {
               if (frameNumber >= currentBlock?.frameCount as Int) {
                  frameNumber = 0
               }
            }
         }
      }

      if (!moveValid(coordinate as Point, frameNumber)) {
         translateBlock(currentBlock?.position as Point,
                        currentBlock?.frameNumber as Int)
         if (Motions.DOWN.name == action) {
            boostScore()
            persistCellData()
            assessField()
            generateNextBlock()
            if (!blockAdditionPossible()) {
               currentState = Statuses.OVER.name;
               currentBlock = null;
               resetField(false);
            }
         }
      } else {
         if (frameNumber != null) {
            translateBlock(coordinate, frameNumber)
            currentBlock?.setState(frameNumber, coordinate)
         }
      }
   }
}
```

Метод `generateField()` генерирует обновление поля. Это обновление поля определяется действием, которое передается в качестве аргумента `generateField()`.

Сначала метод `generateField()` проверяет, находится ли игра при вызове в активном состоянии. Если игра активна, извлекаются номер фрейма и координаты блока. Затем с помощью выражения `when` определяется запрашиваемое действие. После идентификации запрошенного действия (движение влево, вправо или вниз) координаты блока соответствующим образом изменяются. Если запрашивается вращательное движение, значение `frameNumber` изменяется с учетом соответствующего номера фрейма, который представляет вращающееся тетрамино.

Затем метод `generateField()` посредством метода `moveValid()` проверяет, является ли запрошенное движение действительным. Если это движение не является действительным, текущий блок фиксируется в поле в его текущей позиции с помощью метода `translateBlock()`.

Далее приводятся методы `resetField()`, `persistCellData()` и `assessField()`, которые вызываются с помощью метода `generateField()`. Добавим их к модели `AppModel`:

```
private fun resetField(ephemeralCellsOnly: Boolean = true) {
   for (i in 0 until FieldConstants.ROW_COUNT.value) {
      (0 until FieldConstants.COLUMN_COUNT.value)
      .filter { !ephemeralCellsOnly || field[i][it] ==
               CellConstants.EPHEMERAL.value }
      .forEach { field[i][it] = CellConstants.EMPTY.value }
   }
}

private fun persistCellData() {
   for (i in 0 until field.size) {
      for (j in 0 until field[i].size) {
         var status = getCellStatus(i, j)
         if (status == CellConstants.EPHEMERAL.value) {
            status = currentBlock?.staticValue
            setCellStatus(i, j, status)
         }
      }
   }
}

private fun assessField() {
   for (i in 0 until field.size) {
      var emptyCells = 0;
      for (j in 0 until field[i].size) {
         val status = getCellStatus(i, j)
         val isEmpty = CellConstants.EMPTY.value == status
         if (isEmpty)
            emptyCells++
         }
         if (emptyCells == 0)
            shiftRows(i)
      }
}
```

Как можно заметить, метод `translateBlock()` не реализуется. Он добавляется к модели `AppModel` вместе с методами `blockAdditionPossible()`, `shiftRows()`, `startGame()`, `restartGame()`, `endGame()` и `resetModel()` следующим образом:

```
private fun translateBlock(position: Point, frameNumber: Int) {
   synchronized(field) {
      val shape: Array<ByteArray>? = currentBlock?.getShape(frameNumber)
      if (shape != null) {
```

```
            for (i in shape.indices) {
                for (j in 0 until shape[i].size) {
                    val y = position.y + i
                    val x = position.x + j
                    if (CellConstants.EMPTY.value != shape[i][j]) {
                        field[y][x] = shape[i][j]
                    }
                }
            }
        }
    }
}

private fun blockAdditionPossible(): Boolean {
    if (!moveValid(currentBlock?.position as Point,
        currentBlock?.frameNumber)) {
            return false
        }
        return true
    }

    private fun shiftRows(nToRow: Int) {
        if (nToRow > 0) {
            for (j in nToRow - 1 downTo 0) {
                for (m in 0 until field[j].size) {
                    setCellStatus(j + 1, m, getCellStatus(j, m))
                }
            }
        }

    for (j in 0 until field[0].size) {
        setCellStatus(0, j, CellConstants.EMPTY.value)
    }
}
fun startGame() {
    if (!isGameActive()) {
        currentState = Statuses.ACTIVE.name
        generateNextBlock()
    }
}

fun restartGame() {
    resetModel()
    startGame()
}
```

```
fun endGame() {
   score = 0
   currentState = AppModel.Statuses.OVER.name
}

private fun resetModel() {
   resetField(false)
   currentState = Statuses.AWAITING_START.name
   score = 0
}
```

В сценарии, когда запрошенное перемещение является движением вниз, а перемещение недопустимо, это означает, что блок достиг нижней границы поля. В этом случае счет игрока возрастает благодаря применению метода `boostScore()`, и состояния всех ячеек поля сохраняются посредством метода `persistCellData()`. Затем вызывается метод `assessField()` для построчного сканирования строк поля и проверки заполняемости находящихся в строках ячеек:

```
private fun assessField() {
   for (i in 0 until field.size) {
      var emptyCells = 0;
      for (j in 0 until field[i].size) {
         val status = getCellStatus(i, j)
         val isEmpty = CellConstants.EMPTY.value == status
         if (isEmpty)
            emptyCells++
         }
         if (emptyCells == 0)
         shiftRows(i)
   }
}
```

В том случае, если в строке заполнены все ячейки, строка очищается и сдвигается на величину, определенную с помощью метода `shiftRow()`. После завершения оценки поля создается новый блок — с помощью метода `generateNextBlock()`:

```
private fun generateNextBlock() {
   currentBlock = Block.createBlock()
}
```

Прежде чем вновь сгенерированный блок передается полю, модель `AppModel` должна с помощью метода `blockAdditionPossible()` убедиться в том, что поле еще не заполнено, а блок может перемещаться в поле:

```
private fun blockAdditionPossible(): Boolean {
   if (!moveValid(currentBlock?.position as Point,
       currentBlock?.frameNumber)) {
         return false
   }
```

```
    return true
}
```

Если добавление блока невозможно, это значит, что все блоки размещены по верхнему краю поля. В таком случае игра завершается. Тогда текущее состояние игры устанавливается как `Statuses.OVER`, а `currentBlock` — к значению `null`. И наконец, поле очищается.

С другой стороны, если перемещение было действительным с самого начала, блок транслируется с учетом его новых координат посредством метода `translateBlock()`, а состояние текущего блока устанавливается с учетом новых его координат и значения `frameNumber`.

Благодаря этим дополнениям, успешно создана модель приложения для логики игрового процесса. Теперь создадим представление, которое использует модель `AppModel`.

Создание представления *TetrisView*

Итак, приступим. У нас успешно реализованы классы для моделирования блоков, фреймы и формы для различных тетрамино, которые используются в приложении, а также класс `AppModel` для координации всевозможных взаимодействий между представлениями и этими созданными программными компонентами. Но без представления `TetrisView` пользователь не сможет взаимодействовать с `AppModel`. А если пользователь не сможет взаимодействовать с игрой, она потеряет всякий смысл. Так что в этом разделе мы реализуем представление `TetrisView` — пользовательский интерфейс, с помощью которого пользователь играет в Tetris.

Создайте пакет с именем view в вашем исходном пакете и добавьте в него файл TetrisView.kt. Поскольку необходимо, чтобы `TestrisView` принадлежал классу `View`, необходимо объявить его как расширение класса `View`. Для этого добавьте в файл TetrisView.kt следующий код:

```
package com.mydomain.tetris.views

import android.content.Context
import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import android.graphics.RectF
import android.os.Handler
import android.os.Message
import android.util.AttributeSet
import android.view.View
import android.widget.Toast
import com.mydomain.tetris.constants.CellConstants
import com.mydomain.tetris.GameActivity
```

```
import com.mydomain.tetris.constants.FieldConstants
import com.mydomain.tetris.models.AppModel
import com.mydomain.tetris.models.Block

class TetrisView : View {

   private val paint = Paint()
   private var lastMove: Long = 0
   private var model: AppModel? = null
   private var activity: GameActivity? = null
   private val viewHandler = ViewHandler(this)
   private var cellSize: Dimension = Dimension(0, 0)
   private var frameOffset: Dimension = Dimension(0, 0)

   constructor(context: Context, attrs: AttributeSet) :
               super(context, attrs)

   constructor(context: Context, attrs: AttributeSet, defStyle: Int) :
               super(context, attrs, defStyle)

   companion object {
      private val DELAY = 500
      private val BLOCK_OFFSET = 2
      private val FRAME_OFFSET_BASE = 10
   }
}
```

Класс `TetrisView` объявлен здесь как расширение класса `View`, поскольку `View` является классом, который расширяется всеми элементами представления в приложении. Так как тип `View` имеет конструктор, который должен инициализироваться, мы объявили здесь для `TetrisView` два вторичных конструктора, которые инициализируют два различных конструктора класса представления, — в зависимости от того, какой вторичный конструктор вызывается.

Свойство `paint` является экземпляром `android.graphics.Paint`. Класс `Paint` включает информацию о стиле и цвете, относящуюся к рисованию текстов, растровых изображений и геометрических построений. Объект `lastMove` используется для отслеживания промежутка времени в миллисекундах, в течение которого выполняется перемещение. Экземпляр `model` служит для хранения экземпляра `AppModel`, с которым представление `TetrisView` будет взаимодействовать при управлении игровым процессом. `Activity` — это экземпляр созданного класса `GameActivity`. `CellSize` и `frameOffset` — свойства, содержащие размеры ячеек в игре и смещения фрейма соответственно.

Оба класса: `ViewHandler` и `Dimension` — не поддерживаются фреймворком для Android-приложений. И нам необходимо реализовать оба эти класса.

Реализация класса *ViewHandler*

Поскольку блоки перемещаются вдоль полей в интервалах с постоянной задержкой по времени, необходимо переводить поток, обрабатывающий движение блоков, в спящий режим и потом пробуждать этот поток, что позволит через некоторое время приступить к перемещению блоков. Хороший способ выполнить это требование — воспользоваться обработчиком, обрабатывающим запросы для задержки сообщений и продолжающим обработку сообщений после завершения задержки. Более точно, согласно документации Android, *обработчик позволяет пересылать и обрабатывать объекты Message, связанные с MessageQueue потока.* Каждый экземпляр обработчика связан с потоком и очередью сообщений потока.

`ViewHandler` — это пользовательский обработчик, реализуемый для `TetrisView` и отвечающий потребностям представления при отправке и обработке сообщений. Поскольку `ViewHandler` является подклассом `Handler`, необходимо расширить `Handler` и добавить в класс `ViewHandler` необходимое поведение.

Добавьте следующий класс `ViewHandler` в качестве частного (`private`) класса `TetrisView`:

```
private class ViewHandler(private val owner: TetrisView) : Handler() {
   override fun handleMessage(message: Message) {
      if (message.what == 0) {
         if (owner.model != null) {
            if (owner.model!!.isGameOver()) {
               owner.model?.endGame()
               Toast.makeText(owner.activity, "Game over",
                  Toast.LENGTH_LONG).show();
            }
            if (owner.model!!.isGameActive()) {
               owner.setGameCommandWithDelay(AppModel.Motions.DOWN)
            }
         }
      }
   }

   fun sleep(delay: Long) {
      this.removeMessages(0)
      sendMessageDelayed(obtainMessage(0), delay)
   }
}
```

Класс `ViewHandler` воспринимает экземпляр для представления `TetrisView` как аргумент соответствующего конструктора. Этот класс отменяет имеющуюся в соответствующем суперклассе класса функцию `handleMessage()`. Метод `handleMessage()` проверяет отправку сообщения. Целое значение `what` означает, что сообщение отправлено. По окончании игры вызывается функция `endGame()` модели `AppModel` и

отображается всплывающее окно, предупреждающее игрока о завершении игры. Если игра находится в активном состоянии, начинается движение вниз.

Метод `sleep()` просто удаляет любое отправленное сообщение и направляет новое сообщение с задержкой, величина которой указана в аргументе `delay`.

Реализация класса *Dimension*

Класс `Dimension` необходим для включения двух свойств: `width` (ширина) и `height` (высота). Именно этот класс служит идеальным кандидатом для применения класса данных. Добавьте следующий частный класс к классу `TetrisView`:

```
private data class Dimension(val width: Int, val height: Int)
```

Приведенная строка включает эти свойства, а также сеттеры и геттеры, которые для них необходимы.

Реализация класса *TetrisView*

Как можно догадаться, представление `TetrisView` еще далеко от завершения. Сейчас нам необходимо реализовать несколько методов сеттеров для свойств `model` и `activity` представления:

```
fun setModel(model: AppModel) {
   this.model = model
   }

fun setActivity(gameActivity: GameActivity) {
   this.activity = gameActivity
}
```

Удостоверьтесь, что они добавлены в класс `TetrisView`.

Методы `setModel()` и `setActivity()` представляют сеттер функции для свойств `model` и `activity` экземпляра. Как следует из названия, метод `setModel()` устанавливает текущую модель, используемую представлением, а `setActivity()` устанавливает применение действия. Теперь добавим три дополнительных метода: `setGameCommand()`, `setGameCommandWithDelay()` и `updateScores()`:

```
fun setGameCommand(move: AppModel.Motions) {
   if (null != model && (model?.currentState ==
      AppModel.Statuses.ACTIVE.name)) {
      if (AppModel.Motions.DOWN == move) {
         model?.generateField(move.name)
         invalidate()
         return
      }
      setGameCommandWithDelay(move)
   }
}
```

```
fun setGameCommandWithDelay(move: AppModel.Motions) {
   val now = System.currentTimeMillis()
   if (now - lastMove > DELAY) {
      model?.generateField(move.name)invalidate()
      lastMove = now
   }
   updateScores()
   viewHandler.sleep(DELAY.toLong())
}

private fun updateScores() {
   activity?.tvCurrentScore?.text = "${model?.score}"
   activity?.tvHighScore?.text =
      "${activity?.appPreferences?.getHighScore()}"
}
```

Метод `setGameCommand()` устанавливает исполняемую игрой текущую команду перемещения. Если выполняется команда перемещения `DOWN` (Вниз), модель приложения генерирует поле для блока, выполняющего перемещение вниз. Метод `invalidate()`, вызываемый в `setGameCommand()`, может приниматься как запрос на внесение изменений на экране. Метод `invalidate()` в конечном итоге приводит к вызову метод `onDraw()`.

Метод `onDraw()` унаследован от класса `View`. Метод вызывается, если представление отображает свое содержимое. Необходимо предоставить специальную реализацию для этого представления, поэтому добавьте в класс `TetrisView` следующий код:

```
override fun onDraw(canvas: Canvas) {
   super.onDraw(canvas)
   drawFrame(canvas)
   if (model != null) {
      for (i in 0 until FieldConstants.ROW_COUNT.value) {
         for (j in 0 until FieldConstants.COLUMN_COUNT.value) {
            drawCell(canvas, i, j)
         }
      }
   }
}

private fun drawFrame(canvas: Canvas) {
   paint.color = Color.LTGRAY
   canvas.drawRect(frameOffset.width.toFloat(),
      frameOffset.height.toFloat(), width -
      frameOffset.width.toFloat(),
      height - frameOffset.height.toFloat(), paint)
   }
```

```
private fun drawCell(canvas: Canvas, row: Int, col: Int) {
   val cellStatus = model?.getCellStatus(row, col)
   if (CellConstants.EMPTY.value != cellStatus) {
      val color = if (CellConstants.EPHEMERAL.value == cellStatus) {
         model?.currentBlock?.color
    } else {
       Block.getColor(cellStatus as Byte)
    }
    drawCell(canvas, col, row, color as Int)
   }
}

private fun drawCell(canvas: Canvas, x: Int, y: Int, rgbColor: Int) {
   paint.color = rgbColor
   val top: Float = (frameOffset.height + y * cellSize.height +
                                                    BLOCK_OFFSET).toFloat()
   val left: Float = (frameOffset.width + x * cellSize.width +
                                                    BLOCK_OFFSET).toFloat()
   val bottom: Float = (frameOffset.height + (y + 1) * cellSize.height -
                                                    BLOCK_OFFSET).toFloat()
   val right: Float = (frameOffset.width + (x + 1) * cellSize.width -
                                                    BLOCK_OFFSET).toFloat()
   val rectangle = RectF(left, top, right, bottom)
   canvas.drawRoundRect(rectangle, 4F, 4F, paint)
}

override fun onSizeChanged(width: Int, height: Int, previousWidth: Int,
   previousHeight: Int) {
   super.onSizeChanged(width, height, previousWidth, previousHeight)
   val cellWidth = (width - 2 * FRAME_OFFSET_BASE) /
                                          FieldConstants.COLUMN_COUNT.value
   val cellHeight = (height - 2 * FRAME_OFFSET_BASE) /
                                            FieldConstants.ROW_COUNT.value
   val n = Math.min(cellWidth, cellHeight)
   this.cellSize = Dimension(n, n)
   val offsetX = (width - FieldConstants.COLUMN_COUNT.value * n) / 2
   val offsetY = (height - FieldConstants.ROW_COUNT.value * n) / 2
   this.frameOffset = Dimension(offsetX, offsetY)
}
```

Метод `onDraw()` в `TetrisView` переопределяет функцию `onDraw()` в ее суперклассе, т. е. он принимает объект `canvas` в качестве единственного аргумента и должен вызвать функцию `onDraw()` суперкласса. Это делается путем вызова метода `super.onDraw()` и передачи ему экземпляра объекта `canvas` в качестве аргумента.

После вызова методов `super.onDraw()` и `onDraw()` в представлении `TetrisView` вызывается метод `drawFrame()`, который рисует фрейм для представления `TetrisView`. Затем

отдельные его ячейки рисуются в пределах объекта `canvas` путем применения созданных функций `drawCell()`.

Метод `setGameCommandWithDelay()` функционирует аналогично методу `setGameCommand()`, но кроме того, он обновляет счет игры и переводит `viewHandler` в спящий режим после выполнения игровой команды. Функция `updateScore()` служит для обновления в игровом действии текстовых представлений текущего счета и рекорда.

Функция `onSizeChanged()` вызывается при изменении размера представления. Эта функция обеспечивает доступ к текущим значениям ширины и высоты представления, а также к прежним значениям ширины и высоты. Как и с другими примененными переопределенными функциями, мы вызываем в его суперклассе функцию-аналог и используем предоставленные аргументы ширины и высоты для вычисления и задания размеров каждой ячейки — `cellSize`. Наконец, в `onSizeChanged()` вычисляются `offsetX` и `offsetY` и применяются для установки `frameOffset`.

Завершение разработки действия *GameActivity*

К этому моменту нами успешно реализованы представления, обработчики, вспомогательные функции, классы и модели, необходимые для объединения игры Tetris в одно целое. Завершим начатую работу, соединяя все указанные элементы в действии `GameActivity`. Прежде всего необходимо добавить вновь созданное представление `tetris` в макет игровой активности. Это можно легко сделать, добавив представление `TetrisView` в качестве дочернего элемента в любое место файла макета, используя тег макета `<com.mydomain.tetris.views.TetrisView>`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      tools:context="com.mydomain.tetris.GameActivity">
   <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="horizontal"
            android:weightSum="10"
            android:background="#e8e8e8">
      <LinearLayout
               android:layout_width="wrap_content"
               android:layout_height="match_parent"
               android:orientation="vertical"
               android:gravity="center"
               android:paddingTop="32dp"
```

```
            android:paddingBottom="32dp"
            android:layout_weight="1">
        <LinearLayout
                android:layout_width="wrap_content"
                android:layout_height="0dp"
                android:layout_weight="1"
                android:orientation="vertical"
                android:gravity="center">
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/current_score"
                android:textAllCaps="true"
                android:textStyle="bold"
                android:textSize="14sp"/>
            <TextView
                android:id="@+id/tv_current_score"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="18sp"/>
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginTop="@dimen/layout_margin_top"
                android:text="@string/high_score"
                android:textAllCaps="true"
                android:textStyle="bold"
                android:textSize="14sp"/>
            <TextView
                android:id="@+id/tv_high_score"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="18sp"/>
        </LinearLayout>
        <Button
            android:id="@+id/btn_restart"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/btn_restart"/>
    </LinearLayout>
    <View
        android:layout_width="1dp"
        android:layout_height="match_parent"
        android:background="#000"/>
```

```
      <LinearLayout
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="9">
            <!-- Добавление TetrisView -->
            <com.mydomain.tetris.views.TetrisView
            android:id="@+id/view_tetris"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
         </LinearLayout>
      </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Добавив представление `tetris` в файл activity_game.xml, откройте класс `GameActivity` и внесите в него следующие изменения:

```
package com.mydomain.tetris

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import android.view.MotionEvent
import android.view.View
import android.widget.Button
import android.widget.TextView
import com.mydomain.tetris.models.AppModel
import com.mydomain.tetris.storage.AppPreferences
import com.mydomain.tetris.views.TetrisView

class GameActivity: AppCompatActivity() {
   var tvHighScore: TextView? = null
   var tvCurrentScore: TextView? = null

   private lateinit var tetrisView: TetrisView
   var appPreferences: AppPreferences? = null
   private val appModel: AppModel = AppModel()

   public override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_game)
      appPreferences = AppPreferences(this)
      appModel.setPreferences(appPreferences)

      val btnRestart = findViewById<Button>(R.id.btn_restart)
      tvHighScore = findViewById<TextView>(R.id.tv_high_score)
      tvCurrentScore = findViewById<TextView>(R.id.tv_current_score)
      tetrisView = findViewById<TetrisView>(R.id.view_tetris)
      tetrisView.setActivity(this)
```

```
        tetrisView.setModel(appModel)
        tetrisView.setOnTouchListener(this::onTetrisViewTouch)
        btnRestart.setOnClickListener(this::btnRestartClick)
        updateHighScore()
        updateCurrentScore()
    }

private fun btnRestartClick(view: View) {
    appModel.restartGame()
    }

        private fun onTetrisViewTouch(view: View, event: MotionEvent):
        Boolean {
        if (appModel.isGameOver() || appModel.isGameAwaitingStart()) {
            appModel.startGame()
            tetrisView.setGameCommandWithDelay(AppModel.Motions.DOWN)
        } else if(appModel.isGameActive()) {
            when (resolveTouchDirection(view, event)) {
                0 -> moveTetromino(AppModel.Motions.LEFT)
                1 -> moveTetromino(AppModel.Motions.ROTATE)
                2 -> moveTetromino(AppModel.Motions.DOWN)
                3 -> moveTetromino(AppModel.Motions.RIGHT)
            }
        }
        return true
    }

    private fun resolveTouchDirection(view: View, event: MotionEvent):
    Int {
        val x = event.x / view.width
        val y = event.y / view.height
        val direction: Int
        direction = if (y > x) {
            if (x > 1 - y) 2 else 0
            }
            else {
              if (x > 1 - y) 3 else 1
              }
            return direction
        }

        private fun moveTetromino(motion: AppModel.Motions) {
            if (appModel.isGameActive()) {
                tetrisView.setGameCommand(motion)
            }
        }
```

```
    private fun updateHighScore() {
        tvHighScore?.text = "${appPreferences?.getHighScore()}"
    }

    private fun updateCurrentScore() {
        tvCurrentScore?.text = "0"
    }
}
```

Здесь добавлена объектная ссылка на элемент макета представления `tetris` в файле activity_game.xml в форме свойства `tetrisView`, а также создан экземпляр `AppModel`, который будет применяться в `GameActivity`. При выполнении метода `oncreate()` устанавливается действие для использования представлением `tetrisView` в текущем экземпляре `GameActivity`, а также модель, предусматривающая применение `tetrisView` к `appModel` (созданное свойство экземпляра `AppModel`). Кроме того, для функции `onTetrisViewTouch()` установлен слушатель «в одно касание» для представления `tetrisView`.

Если представление `tetrisView` с помощью касания активизировано, когда игра находится в состояниях `AWAITING_START` или `OVER`, начинается новая игра. Если же представление `tetrisView` с помощью касания активизировано, когда игра находится в состоянии `ACTIVE`, направление, в котором произошло касание, определяется для `tetrisView` с помощью `resolveTouchDirection()`, и блок тетрамино перемещается на основе действия, переданного ему с помощью метода `moveTetromino()`. Если касание выполнено влево, метод `moveTetromino()` вызывается с помощью `AppModel.Motions.LEFT`, который устанавливается в качестве аргумента. Это приводит к перемещению тетрамино на игровом поле влево. Касания вправо, вниз и вверх приводят к тому, что представление `tetrisView` соответственно смещается в направлении вправо, вниз и с использованием вращения.

Выполнив все изменения, соберите и запустите проект. После запуска проекта на использованном устройстве перейдите к игровой активности и коснитесь представления `tetris` в правой части экрана. Игра начнется (рис. 3.7).

Не стесняйтесь поиграть в созданную вами игру. Вы это действительно заслужили!

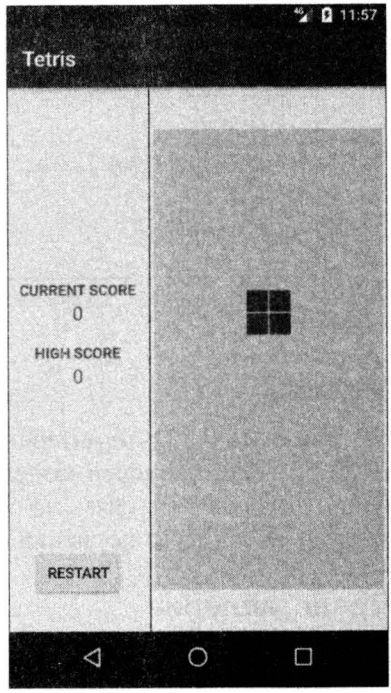


Рис. 3.7. Игра началась!

Введение в шаблон Model-View-Presenter (MVP)

В ходе разработки приложения Tetris мы создали в основном его коде структуру, в соответствии с которой произвели разделение программных файлов на различные пакеты в зависимости от выполняемых ими задач. При этом нами была предпринята попытка абстрагировать логику приложения в классе `AppModel`, тогда как взаимодействие с пользователем, связанное с игровым процессом, обрабатывается классом представления `TetrisView`. Такой подход, безусловно, позволил внести некоторый порядок в наш базовый код — в отличие, скажем, от размещения всей логики в одном большом файле класса.

Надо отметить, что имеются и более интересные способы разделения проблем в приложении Android. Один из подобных способов представлен шаблоном MVP.

Что такое MVP?

Распространенный в Android шаблон MVP (Model-View-Presenter, модель-представление-презентатор) получен на основе шаблона MVC (Model-View-Controller, модель-представление-контроллер). Шаблон MVP (рис. 3.8) пытается предупредить возникновение проблем, связанных с логикой приложения. Существует множество причин, из-за которых требуется это делать, и в том числе:

- повышение удобства при сопровождении кода программ;
- увеличение степени надежности приложений.

Познакомимся с действующими лицами шаблона MVP.

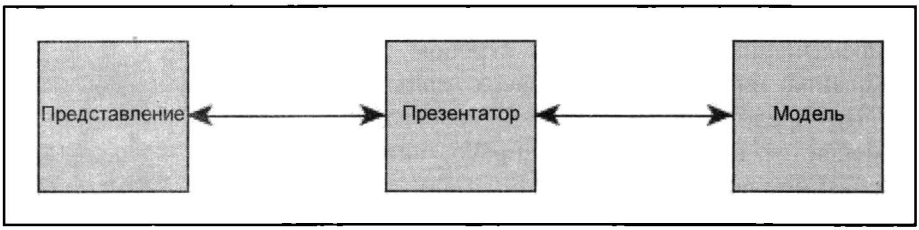


Рис. 3.8. Шаблон MVP

Модель

Модели в MVP — это интерфейсы, задача которых состоит в управлении данными. В зону ответственности моделей входит взаимодействие с базами данных, выполнение вызовов API, связь по сетям и координация объектов и других программных компонентов для выполнения конкретных задач.

Представление

Представления — это объекты приложения, отображающие контент для пользователей и служащие интерфейсом для ввода пользователем данных. Представление

может быть действием, фрагментом или виджетом Android. Представление отвечает за визуализацию данных выбранным презентатором способом.

Презентатор

Презентатор — это слой, выступающий посредником между представлением и моделью. Основная обязанность презентатора — запросить модель и обновить представление. Иными словами, логика представления включена в состав презентатора. Важно понимать, что презентатор имеет прямое отношение к представлению.

Различные реализации MVP

Шаблон MVP располагает различными средствами, с помощью которых он реализуется на практике. Например, некоторые реализации MVP применяют *контракт* для описания взаимодействия между представлением и презентатором.

Кроме того, имеются реализации MVP, применяющие в презентаторе обратные вызовы жизненного цикла, например `onCreate()`. При этом предпринимается попытка отразить обратные вызовы, имеющиеся в жизненном цикле действия. Другие реализации полностью отказываются от реализации этих обратных вызовов.

В действительности в приложениях Android отсутствует единая истинная реализация MVP, но имеются лучшие подходы, которым можно следовать при реализации этого шаблона. Об этих лучших подходах вы узнаете в *главе 5*, где также получите практический опыт разработки приложения MVP с помощью Kotlin.

Подведем итоги

В этой главе знакомство с возможностями языка Kotlin проходило в процессе реализации классической игры Tetris. Мы рассмотрели целый комплекс вопросов, связанных с моделированием логических компонентов приложения с помощью классов, с модификаторами доступа и видимости, созданием представлений и обработчиков в приложениях Android, использованием классов данных для удобства при создании моделей данных, а также узнали о существовании шаблонов MVP.

В следующей главе мы применим наши знания языка Kotlin в области, связанной с Интернетом, — путем реализации серверной части приложения для обмена интернет-сообщениями.

4

Разработка и реализация серверной части интернет-мессенджера с помощью фреймворка Spring Boot 2.0

Первые две главы этой книги на примере практической реализации классической игры Tetris помогли вам получить четкое представление об основах языка программирования Kotlin. В третьей главе мы завершили разработку этой игры: реализовали логику игрового приложения и с помощью класса модели приложения создали программные модели для блоков, фигур, фреймов и приложения в целом. Кроме того, в процессе реализации представления `tetrisView` — представления, с которым в процессе игры взаимодействует пользователь приложения, — мы разобрались, как создавать собственные представления.

Продолжим дальнейшее совершенствование наших навыков программирования на языке Kotlin на примере простого приложения для обмена интернет-сообщениями с использованием платформы Android (приложение Messenger). В процессе реализации этого Android-приложения мы сначала разработаем интерфейс RESTful API, который как бы «из-за кулис» предоставляет приложению веб-контент. Интерфейс прикладного программирования мы сформируем с помощью фреймворка Spring Boot 2.0. Разработав интерфейс программирования приложения, мы развернем его на удаленном сервере.

При изучении этой главы вы познакомитесь со следующими темами:

- базовый дизайн системы;
- моделирование поведения системы с помощью диаграмм состояний;
- основы проектирования баз данных;

- моделирование базы данных с помощью диаграмм отношений сущностей (E-R);
- создание внутренних микросервисов с помощью фреймворка Spring Boot 2.0;
- работа с СУБД PostgreSQL;
- управление зависимостями с помощью фреймворка Maven;
- службы Amazon Web Services (AWS).

Итак, не тратя лишних слов, перейдем к разработке интерфейса программирования Android-приложения Messenger.

Разработка API Messenger

Для создания полностью функционального интерфейса прикладного программирования RESTful нашего Android-приложения для обмена интернет-сообщениями Messenger (далее в тексте API Messenger) следует четко представлять себе концепцию интерфейсов прикладного программирования REST (Representational State Transfer, передача репрезентативного состояния) и служб RESTful.

Интерфейсы прикладного программирования

Интерфейс прикладного программирования — это набор функций, подпрограмм, процедур, протоколов и ресурсов, которые применяются для создания программного обеспечения. Другими словами, интерфейс прикладного программирования (API, Application Programming Interface) — это набор четко определенных и надлежащим образом структурированных методов, или каналов связи между программными компонентами.

Интерфейсы прикладного программирования могут разрабатываться для использования в различных областях. Некоторые общие области приложений, для которых разрабатываются API-интерфейсы, включают разработку веб-систем, операционных систем и компьютерного оборудования, а также взаимодействие со встроенными системами.

REST

Протокол RESTful state transfer (передача состояния покоя), также обозначаемый аббревиатурой REST, — это способ облегчения функциональной работы и взаимодействий между двумя или большим числом различных систем (или подсистем) через Интернет. Веб-сервисы, которые поддерживают REST, позволяют взаимодействующим системам получать доступ к веб-контенту. Они также выполняют авторизованные операции с веб-контентом, к которому имеется доступ. Эти межсистемные коммуникации осуществляются с помощью строго определенного набора операций без сохранения состояния. Веб-сервис RESTful относится к REST и предоставляет веб-контент системам связи посредством заранее определенных операций без сохранения состояния.

В настоящее время многочисленные системы, взаимодействующие с веб-сервисами, поддерживают REST. Системы, поддерживающие REST, основаны на архитектуре клиент-сервер. API, который мы начинаем разрабатывать, базируется на REST и, соответственно, будет использовать передачу репрезентативного состояния.

Разработка системы API Messenger

Что ж, приступим к проектированию системы API Messenger. Но прежде всего нам надо разобраться в том, что же представляет собой проектирование системы и какие действия необходимо выполнить.

Проектирование системы — это процесс уточнения архитектуры, модулей, интерфейсов и данных для системы с целью удовлетворения определенных требований, которые рассматриваются на предварительном этапе системного анализа. Проектирование системы включает большое число процессов и охватывает различные направления дизайна. Кроме того, основательное проектирование систем требует понимания многочисленных тем, среди которых, например, присутствуют такие, как сопряжение и связность, рассмотрение которых выходит за рамки этой книги. Учитывая это обстоятельство, приведем базовые определения используемых в нашей системе взаимодействий и данных. И будем формировать нашу систему постепенно.

Инкрементная разработка

Инкрементная разработка — это подход, который используется для разработки систем. Реализация этого подхода предполагает *модель поэтапной сборки* — метод разработки программного обеспечения, когда продукт разрабатывается, внедряется и тестируется поэтапно. Именно таким образом мы поступим при разработке API Messenger. Приступая к созданию программного кода, мы не станем указывать все, что нужно для API Messenger, — определим сначала набор спецификаций, который позволит приступить к разработке, затем сформируем некоторые функциональные возможности, после чего — повторим процесс.

Чтобы применять методологию инкрементной разработки, следует использовать программное обеспечение, в которое можно вносить изменения в процессе его разработки, — например, когда требуется изменить тип обслуживаемых системой данных. Фреймворк Spring Boot является идеальным кандидатом для постепенной разработки систем, поскольку позволяет вносить изменения в систему быстро и легко.

До этого момента мы пару раз упоминали о Spring Boot, но не обсуждали этот программный продукт. Рассмотрим его подробнее.

Spring Boot

Spring Boot — это фреймворк для веб-приложений, который спроектирован и разработан для загрузки и разработки приложений Spring. В свою очередь, Spring — это платформа веб-приложений, упрощающая разработку веб-приложений для

платформы Java. Таким образом, Spring Boot облегчает создание серьезных приложений, основанных на Spring.

В этой главе далее мы узнаем, как создавать веб-приложения с помощью Spring Boot, но начнем мы с рассмотрения других вопросов. Еще до разработки приложения следует представить себе, что́ именно оно делает (нельзя ничего создать, если неведомо, как оно будет функционировать).

Задачи системы Messenger

Определим начальные требования к системе обмена сообщениями и перечень действий, которые могут в системе выполняться, и обозначим высокоуровневые варианты применения приложения Messenger.

Варианты использования

Вариант использования — это описание того, каким образом сущность применяет систему. Под *сущностью* понимается взаимодействующий с системой тип пользователя или компонента. Для определенных вариантов использования сущности также могут называться *а́кторами*.

Определим акторов в системе обмена сообщениями. Понятно, что очевидным актором является пользователь приложения — человек, использующий приложение для удовлетворения потребностей в обмене сообщениями. Другой актор, который должен в идеале присутствовать, — это администратор. Однако, для целей нашего простого приложения, которое служит для обмена сообщениями, речь пойдет об акторе-пользователе. Рассмотрим варианты использования для этого актора:

- для отправки и получения сообщений пользователь применяет платформу обмена сообщениями Messenger;
- пользователь использует платформу Messenger, чтобы увидеть в приложении обмена сообщениями Messenger других пользователей;
- пользователь применяет платформу Messenger для задания и обновления своего статуса;
- пользователь может зарегистрироваться на платформе Messenger;
- пользователь может войти в платформу Messenger.

Приведенных вариантов использования достаточно, чтобы мы начали действовать. Если в какой-то момент в ходе разработки системы мы столкнемся с новым вариантом использования, его можно будет легко добавить в систему. Теперь, после определения вариантов использования системы, необходимо должным образом описать поведение системы для этих случаев.

Поведение системы

Определим *поведение системы*, чтобы получить точное представление о ее функциональных особенностях, а также для четкого описания взаимодействия между компонентами системы. Поскольку наше приложение весьма несложно, можно описать его поведение с помощью так называемой *диаграммы состояний*.

Диаграммы состояний

Диаграммы состояний применяются для описания поведения систем, способных находиться в различных возможных состояниях. На диаграммах состояний указывается конечное число возможных состояний, в которых может находиться система.

На рис. 4.1 приведена диаграмма состояний для нашей системы с учетом определенных вариантов ее использования.

Кружки на диаграмме представляют состояния системы в определенные моменты времени. Каждая стрелка указывает на действие, которое пользователь может запросить в системе. При первоначальном запуске API-интерфейс ожидает запросов от клиентских приложений. Это поведение отображается в состоянии **Ожидание действия** (Waiting for action). Если API-интерфейс получает запрос действия от клиентского приложения, система выходит из состояния ожидания действия и обрабатывает отправленный соответствующим процессом запрос.

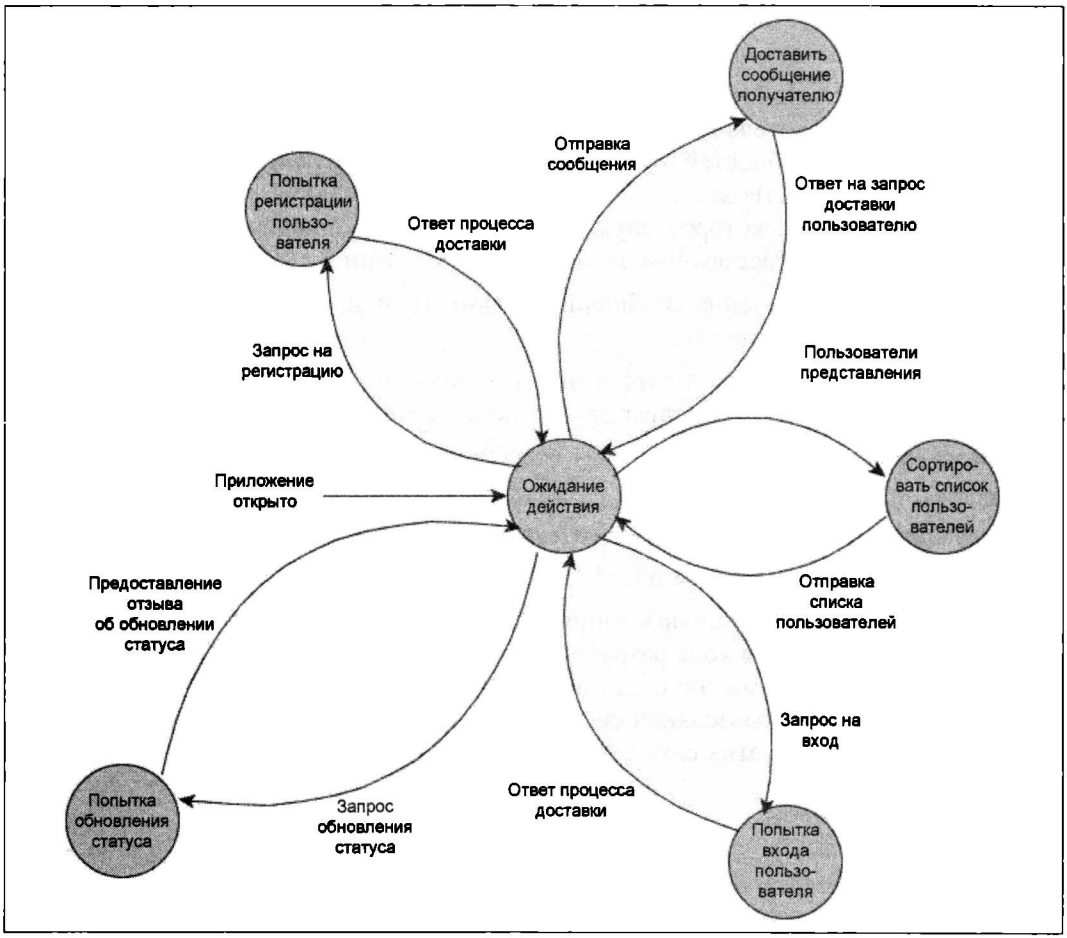


Рис. 4.1. Диаграмма состояний проектируемой системы

Например, пользователь запрашивает обновление состояния из приложения Android — тогда сервер выходит из состояния **Ожидание действия** (Waiting for action) и выполняет процесс обновления состояния **Попытка обновления статуса** (Attempt status update), после чего возвращается в состояние **Ожидание действия** (Waiting for action).

Идентификация данных

Перед созланием системы важно иметь представление о типе данных, которые ей будут необходимы. Можно легко идентифицировать эти данные из приведенных ранее определений вариантов использования. Анализ вариантов использования позволяет сделать вывод, что необходимы два основных типа данных. Речь идет о пользовательских данных и данных сообщения. Как следует из их наименований, *пользовательские данные* — это данные, характеризующие каждого пользователя, а *данные сообщения* — это данные, относящиеся к отправленному сообщению. Не углубляясь пока в рассмотрение таких объектов, как схемы, сущности или диаграммы отношений сущностей, получим представление о данных, необходимых для системы.

Поскольку наше приложение служит для обмена сообщениями, пользователю нужно знать имя пользователя, номер телефона, пароль и содержание сообщения о состоянии. Также полезно отслеживать состояние учетных записей, чтобы знать, активирована ли учетная запись определенного пользователя или по какой-либо причине деактивирована. В общем, для отправки сообщений требуется не так уж и много информации — нужно отслеживать отправителя сообщения и предполагаемого получателя сообщения.

Пожалуй, это все, что относится к необходимым данным. Далее мы определим еще некоторые данные, требуемые для разработки приложения, но сейчас займемся программированием.

Реализация серверной части приложения Messenger

Имея уже некоторое представление о направлениях использования системы сообщений в целом, о необходимых системе данных и о поведении системы, можно приступить к разработке ее серверной части (*бэкэнда* — от англ. backend). Как уже упоминалось, для разработки API Messenger мы используем фреймворк Spring Boot, поскольку это идеальный вариант для поэтапной разработки приложений. Кроме того, известно, что Kotlin и Spring Boot отлично «ладят» друг с другом.

Поскольку в API Messenger осуществляется обработка данных, для хранения данных, необходимых системе обмена сообщениями, понадобится подходящая база данных. В качестве базы данных мы задействуем PostgreSQL. Вкратце опишем ее возможности.

PostgreSQL

PostgreSQL — это объектно-реляционная система управления базами данных (СУБД), особое внимание в которой уделяется возможности расширения и поддержанию соответствия стандартам. PostgreSQL также известна как Postgres. Обычно она используется в качестве сервера базы данных. При этом ее основными задачами являются безопасное хранение данных и возврат данных, которые сохранены по запросу программных приложений.

Использование PostgreSQL в качестве хранилища данных обеспечивает массу преимуществ. Вот лишь некоторые из них:

- *расширяемость* — возможность PostgreSQL легко расширяться пользователями, поскольку исходный код этого программного продукта находится в свободном доступе;
- *переносимость* — эта база данных доступна на всех основных платформах, в том числе и UNIX-подобных. Совместимость с Windows достигается благодаря фреймворку Cygwin;
- *интегрируемость* — готовность к использованию инструментов на основе графического интерфейса, облегчающих взаимодействие с PostgreSQL.

Установка PostgreSQL

Установка PostgreSQL несложна для всех платформ. Здесь мы рассмотрим процесс ее установки в Windows, macOS и Linux.

Установка в среде Windows

Для установки PostgreSQL в Windows:

1. Загрузите и запустите соответствующую версию программы интерактивной установки Windows PostgreSQL. Ее можно загрузить с сайта по адресу:

   https://www.enterprisedb.com/downloads/postgres-postgresql-downloads#windows.

2. Установите PostgreSQL в качестве службы Windows. Обязательно запомните имя и пароль учетной записи службы PostgreSQL для Windows — это понадобится вам позднее в процессе установки.
3. Выберите процедурный язык PL/pgsql для установки, когда программа установки предложит вам сделать это.
4. Когда появится окно **Installation options** (Параметры установки), можно установить и графический клиент pgAdmin. Если вы устанавливаете pgAdmin, подключите также по запросу программы установки и модуль Adminpack contrib.

Если все действия были выполнены верно, PostgreSQL успешно установится в системе.

Установка в среде macOS

PostgreSQL может легко устанавливаться на macOS с помощью менеджера пакетов Homebrew. Если он в системе не установлен, обратитесь к *главе 1* за инструкциями по его установке. После успешной установки Homebrew в вашей системе откройте окно терминала и выполните следующую команду:

```
brew search postgres
```

Далее следуйте инструкциям по установке при появлении запросов в окне терминала. В процессе установки вас могут попросить ввести пароль администратора вашей системы. Введите пароль и дождитесь сообщения об окончании процесса установки.

Установка в среде Linux

PostgreSQL можно легко установить в Linux с помощью программы установки PostgreSQL Linux.

1. Перейдите на страницу загрузки программы установки PostgreSQL на сайте по адресу: **https://www.enterprisedb.com/downloads/postgres-postgresqldownloads**.
2. Выберите версию PostgreSQL, которую необходимо установить.
3. Выберите соответствующую программу Linux для установки PostgreSQL.
4. Щелкните на кнопке загрузки, чтобы загрузить программу установки.
5. После завершения загрузки программы установки запустите ее и следуйте соответствующим инструкциям по установке.
6. При предоставлении информации, необходимой для установки, PostgreSQL будет установлен в вашей системе.

Теперь, после настройки PostgreSQL в системе, можно приступить к созданию Messenger API.

Создание нового приложения Spring Boot

Первоначальное создание приложения Spring Boot можно легко выполнить при помощи IntelliJ IDE и инициализатора Spring. Откройте IntelliJ IDE и создайте новый проект. Для этого щелкните на кнопке **Create New Project** (Создать новый проект) и выберите опцию **Spring Initializr** (Инициализатор Spring) на левой боковой панели экрана **New Project** (Новый проект) (рис. 4.2), после чего щелкните на кнопке **Next** (Далее) для перехода к следующему шагу.

На этом шаге перед отображением следующего окна среда IDE извлекает Spring Initializer (Инициализатор Spring).

Инициализатор Spring поставляется с плагином Spring, на момент подготовки этой книги доступным только в среде IntelliJ IDEA Ultimate Edition, для пользования которой нужна платная лицензия. Если у вас установлена среда IntelliJ IDEA Community Edition, чтобы продолжить работу над приложением, создайте проект с помощью утилиты инициализатора Spring, доступной на сайте **https://start.spring.io**, после чего импортируйте этот проект в среду IntelliJ IDEA.

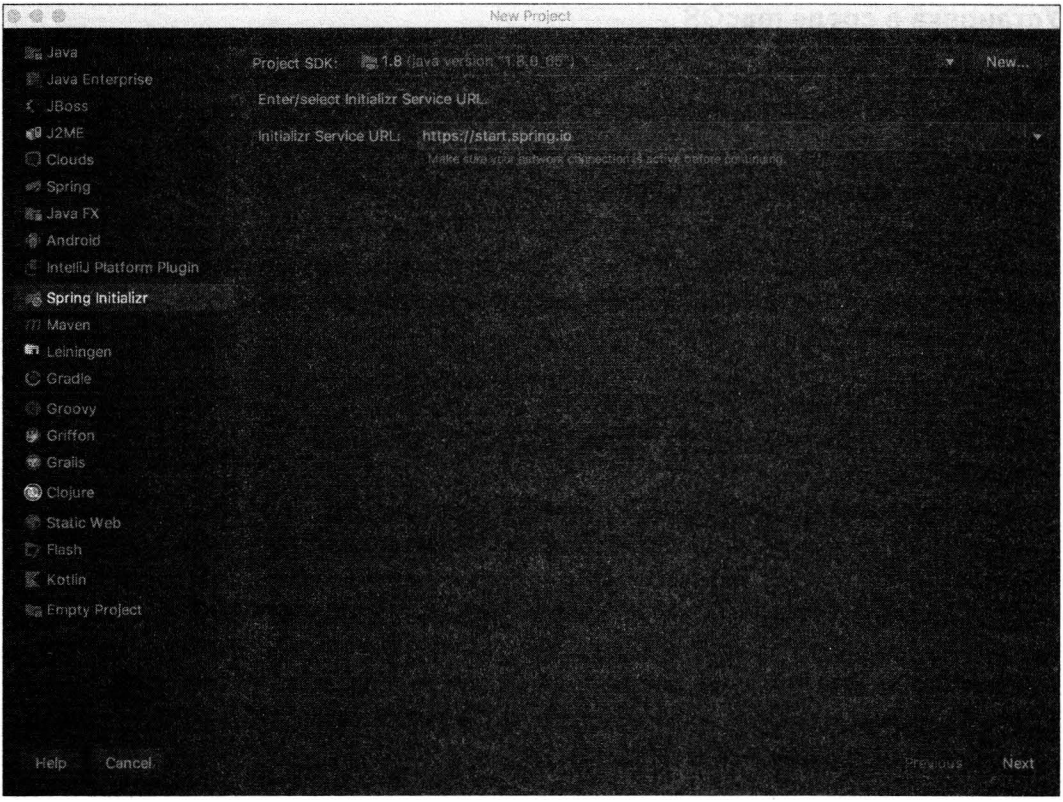


Рис. 4.2. Создание нового проекта с помощью **Spring Initializr**

После извлечения инициализатора Spring вам предложат предоставить соответствующую информацию о создаваемом проекте (рис. 4.3). Вы можете ввести сведения, использованные для разработки приложения в этой книге, или же собственные данные. Если вы решили использовать данные из книги, выполните следующие действия:

1. Введите `com.example` в поле **Group** (ID группы).
2. Введите `messenger-api` в поле **Artifact** (ID приложения).
3. Выберите **Maven Project** в поле **Type** (Тип проекта), если еще не сделан такой выбор.
4. Оставьте вариант упаковки (поле **Packaging**) и версию Java (поле **Java Version**) без изменений.
5. В качестве языка программирования (поле **Language**) выберите **Kotlin**. Это важно, поскольку наша книга посвящена языку Kotlin.
6. Оставьте в поле **Version** значение **SNAPSHOT** без изменений.
7. Введите в поле **Description** описание вашего варианта выбора.

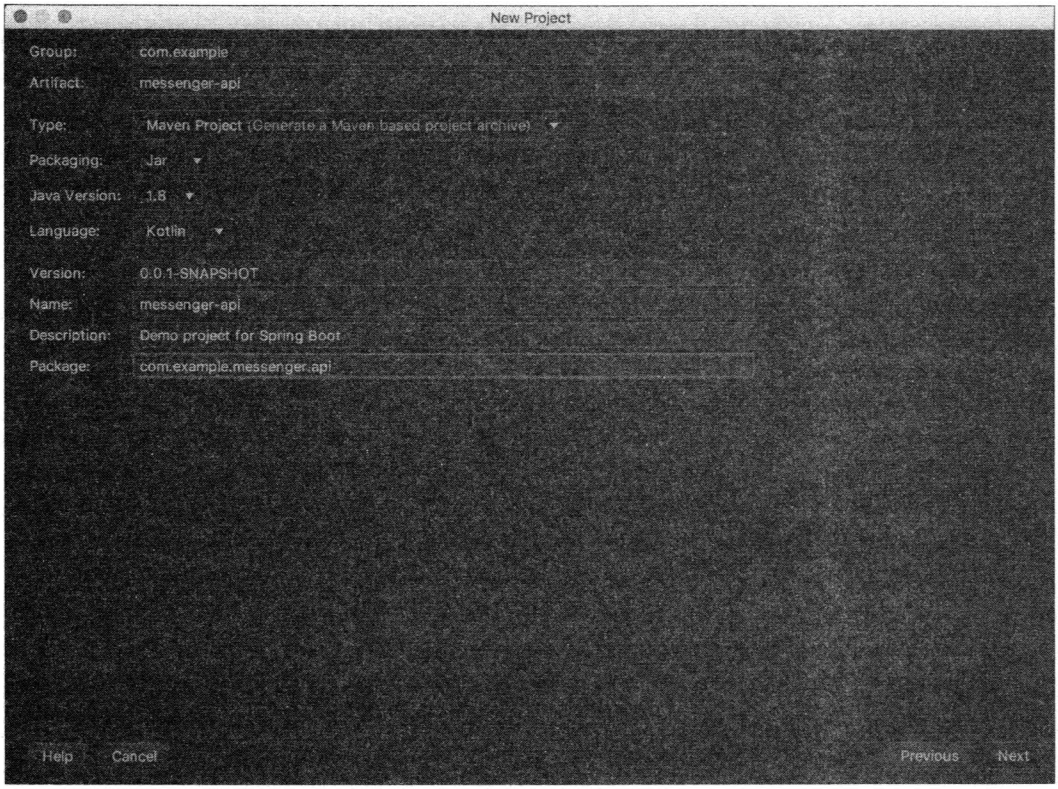


Рис. 4.3. Окно ввода информации о создаваемом проекте

8. Введите `messenger-api` в поле **Name** (Имя проекта).
9. В качестве названия пакета введите `com.example.messenger.api` в поле **Package** (Название пакета).

Завершив ввод этой информации о проекте, перейдите к следующему окну, щелкнув на кнопке **Next** (Далее).

В следующем окне (рис. 4.4) необходимо выбрать зависимости проекта. Прежде всего следует выбрать зависимости **Security** (Безопасность) — в разделе **Core**, **Web** (Веб) — в разделе **Web**, **JPA** и **PostgreSQL** — в разделе **SQL**. Кроме того, в раскрывающемся меню **Spring Boot** (Версии Spring Boot), находящемся в верхней части окна, выберите в качестве версии: **2.0.0 M5**.

Завершив выбор требуемых зависимостей, щелкните на кнопке **Next** (Далее) для перехода к окну окончательной настройки. Вам будет предложено указать название и местоположение проекта. Введите `messenger-api` в качестве имени проекта и выберите место, где будет храниться проект на вашем компьютере. Щелкните на кнопке **Finish** (Готово) и подождите завершения настройки проекта. На экране появится новое окно IDE, включающее исходные файлы проекта.

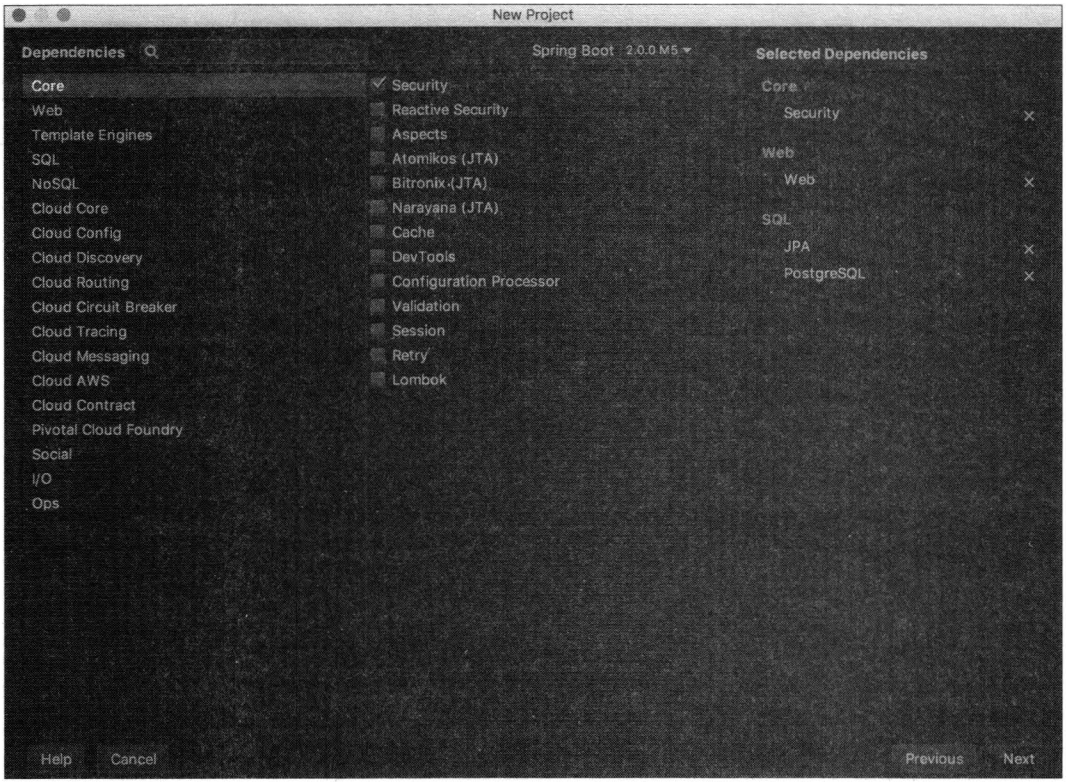


Рис. 4.4. Выбор зависимостей проекта

Знакомство со Spring Boot

Рассмотрим структуру исходных файлов программы для нашего приложения Spring Boot (рис. 4.5).

Все файлы исходного кода приложения находятся в каталоге src. Этот каталог содержит основные программные файлы приложения, а также написанные для приложения тестовые программы. Основные файлы прикладных программ должны быть помещены в каталог src/main, а тестовые программы — в каталог src/test. Основной каталог содержит два подкаталога: kotlin и resources. При работе над материалом этой главы все пакеты и основные файлы исходного кода помещаются в пакет com.example.messenger.api каталога kotlin. Рассмотрим включенный в этот пакет файл MessengerApiApplication.kt (рис. 4.6).

Этот файл содержит основную функцию. Она служит точкой входа в каждое приложение Spring Boot и вызывается при запуске приложения. После вызова функция запускает приложение Spring, вызывая функцию `SpringApplication.run()`, имеющую два аргумента: первый аргумент служит ссылкой на класс, а второй — на аргументы, передаваемые приложению при запуске.

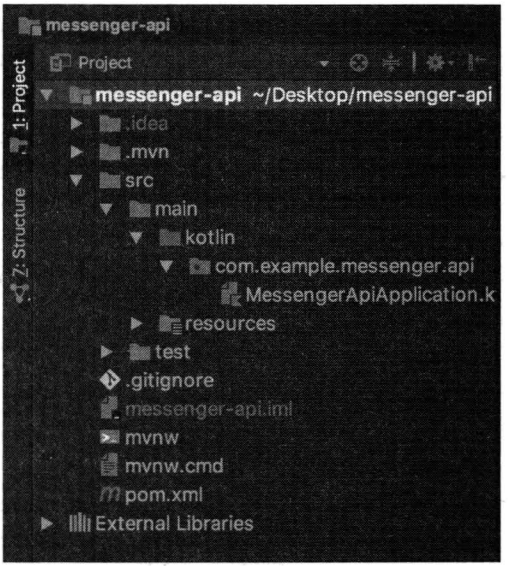


Рис. 4.5. Структура исходных файлов программы для нашего приложения Spring Boot

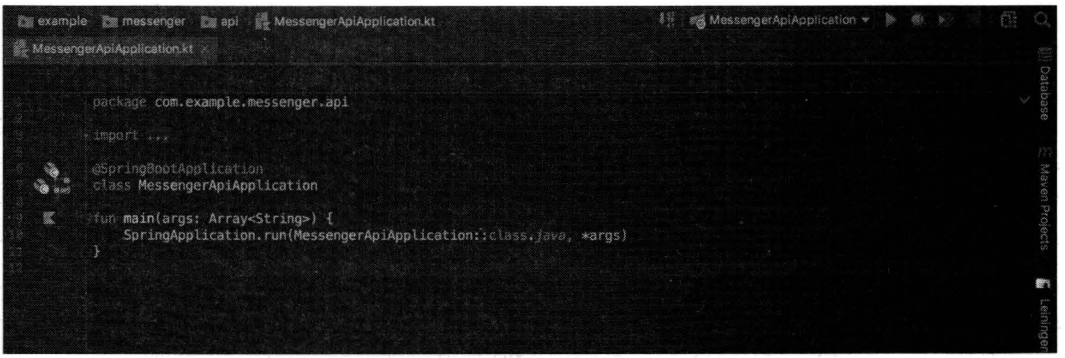


Рис. 4.6. Содержимое файла MessengerApiApplication.kt

В том же файле имеется класс `MessengerApiApplication`. Этот класс аннотируется с помощью аннотации `@SpringBootApplication`. Применение этой аннотации эквивалентно комбинированному использованию аннотаций `@Configuration`, `@EnableAutoConfiguration` и `@ComponentScan`. Классы, аннотируемые с помощью `@Configuration` служат источниками определений *бина* (bean).

Бин (bean) — это объект, который создается и собирается контейнером Spring IoC.

Атрибут `@EnableAutoConfiguration` указывает Spring Boot на тот факт, что приложение Spring автоматически настроено на основе предоставленных вами зависимостей jar. Аннотация `@ComponentScan` настраивает каталоги сканирования компонентов для использования совместно с классами `@Configuration`.

При разработке приложений Spring Boot по разным причинам необходимо использовать несколько аннотаций. Поначалу использование этих аннотаций вносит небольшую путаницу, но со временем они станут для вас привычными.

Помимо файла MessengerApiApplication.kt, другим важным файлом является файл application.properties, расположенный в каталоге src/main/resources. Этот файл применяется для настройки свойств приложений Spring Boot. Открыв этот файл, вы обнаружите, что в нем отсутствует содержимое. Дело в том, что еще не определены конфигурации или свойства приложения. Продолжим работу и добавим в файл application.properties пару конфигураций:

```
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
```

Свойство `spring.jpa.generate-ddl` определяет, должна ли схема базы данных генерироваться при запуске приложения. Если для этого свойства установлено значение `true`, схема создается при запуске приложения, в противном случае схема не создается. Свойство `spring.jpa.hibernate.ddl-auto` используется для указания режима DDL. Параметр `create-drop` устанавливает, чтобы схема создавалась при запуске приложения и уничтожалась при его завершении.

Свойства использовались для определения схемы базы данных, а теперь нам предстоит создать для Messenger API фактическую базу данных. Если вместе с PostgreSQL установлен и графический клиент pgAdmin, можно легко создать базу данных с его помощью. Если pgAdmin не установлен, все равно можно создать базу данных посредством команды PostgreSQL `createdb`. Перейдите в окно терминала и введите следующую команду:

```
createdb -h localhost --username=<username> --password messenger-api
```

Флаг `-h` используется для указания имени хоста компьютера, с которым работает сервер базы данных. Флаг `--username` задает имя пользователя для поддержки соединения с сервером (вместо *<username>* впишите здесь имя пользователя сервера). Флаг `--password` вызывает запрос ввода пароля; `messenger-api` — это имя, которое назначается создаваемой базе данных. После ввода команды нажмите клавишу ввода для запуска команды. Введите ваш пароль после отображения соответствующего запроса. В PostgreSQL будет создана база данных с именем `messenger-api`.

Теперь, после настройки базы данных, нужно подключить к базе данных приложение Spring Boot. Добавьте в файл application.properties следующие конфигурации:

```
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.url=jdbc:postgresql://localhost:5432/messenger-api
spring.datasource.username=<username>
spring.datasource.password=<password>
```

Здесь фигурируют свойства: `spring.datasource.url`, `spring.datasource.username` и `spring.datasource.password`.

Свойство `spring.datasource.url` указывает ссылку URL JDBC, посредством которой Spring Boot будет подключаться к базе данных.

Свойства `spring.datasource.username` и `spring.datasource.password` служат для указания имени пользователя сервера и пароля, который соответствует данному имени пользователя. Замените *<username>* и *<password>* вашими именем пользователя и паролем.

После настройки этих свойств вы будете готовы запустить приложение Spring Boot. Наше приложение Spring Boot с именем `messenger-api` запускается с помощью щелчка на логотипе Kotlin рядом с основной функцией в файле MessengerApi Application.kt и выбора параметра **Run** (Выполнить), как показано на рис. 4.7.

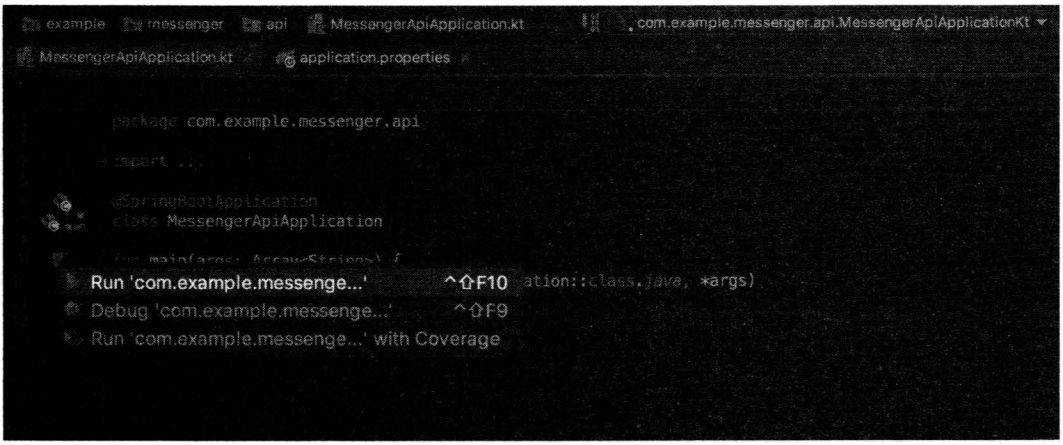


Рис. 4.7. Запуск приложения Spring Boot

Подождите немного, пока завершится создание проекта. После завершения процесса сборки проекта приложение будет запущено на сервере Tomcat.

Давайте продолжим изучение файлов проекта. Найдите в корневом каталоге проекта файл pom.xml. Аббревиатура POM расшифровывается как *Project Object Model* (Объектная модель проекта). На сайте Apache Maven о POM приведена следующая информация: *Project Object Model, или POM, является основной единицей работы в Maven*. Это — XML-файл, который содержит информацию о проекте и сведения о конфигурации, используемые Maven для создания проекта. Обнаружив этот файл, откройте его. Несложно, верно? Все это хорошо, но не помешает познакомиться с кратким описанием Maven.

Maven

Apache Maven — это инструмент для управления программным проектом, основанный на концепции POM. Maven может применяться для нескольких целей — в том числе таких, как управление сборкой проекта и составление документации.

Разобравшись немного с файлами проекта, продолжим разработку системы, добавляя некоторые модели для обработки определенных ранее данных.

Создание моделей

Переходим к моделированию определенных ранее данных в подходящие классы сущностей, которые могут быть проанализированы Spring Boot для построения схемы базы данных. Первая модель, которой мы займемся, — это модель пользователя. Создайте под пакетом `com.example.messenger.api` пакет с именем `models`, а в нем —файл User.kt и введите в него следующий код:

```
package com.example.messenger.api.models

import org.hibernate.validator.constraints.Length
import org.springframework.format.annotation.DateTimeFormat
import java.time.Instant
import java.util.*
import javax.persistence.*
import javax.validation.constraints.Pattern
import javax.validation.constraints.Size

@Entity
@Table(name = "`user`")
@EntityListeners(UserListener::class)
class User(
   @Column(unique = true)
   @Size(min = 2)
   var username: String = "",
   @Size(min = 11)
   @Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$")
   var phoneNumber: String = "",
   @Size(min = 60, max = 60)
   var password: String = "",
   var status: String = "",
   @Pattern(regexp = "\\A(activated|deactivated)\\z")
   var accountStatus: String = "activated"
)
```

В приведенном блоке кода использовалось большое число аннотаций. Рассмотрим, что́ выполняет каждая из них в том порядке, как они появляются. Прежде всего, имеется аннотация `@Entity`, указывающая, что класс является сущностью Java Persistence API (JPA). Аннотация `@Table` указывает имя таблицы для представляемой классом сущности. Подобный подход полезен при генерации схемы. Если же аннотация `@Table` не используется, именем сгенерированной таблицы становится имя класса. В базе данных PostgreSQL создается таблица базы данных с именем пользователя. Аннотация `@EntityListeners`, как следует из названия, определяет слушателя сущности для класса сущности. В настоящее время класс `UserListener` еще не создан, мы сделаем это немного позже.

Рассмотрим свойства класса `User`. Сейчас в него добавлено семь свойств класса. Первые пять свойств: `username`, `phoneNumber`, `password`, `status` и `accountStatus` пред-

ставляют тип данных, которые характеризуют пользователя, как ранее определено в *разд. «Идентификация данных» этой главы*. Теперь, когда создан пользовательский объект, продолжим работу. Но у нас имеется одна проблема. Необходимо располагать способом уникальной идентификации каждого созданного пользователя. Кроме того, для дальнейшего применения важно отслеживать, когда именно новые пользователи добавляются к платформе обмена сообщениями. После тщательного рассмотрения мы поймем, насколько важно иметь в нашей сущности свойства `id` и `createdAt`. Может возникнуть вопрос о том, зачем добавлять в пользовательскую сущность свойства `id` и `createdAt`? В конце концов, нигде раньше не было указано, что они нам нужны. Это правда. Но поскольку разработка серверной части происходит постепенно, то разрешается по мере необходимости вносить в нее изменения и дополнения. Продолжим работу и добавим эти два свойства:

```
@Entity
@Table(name = "`user`")
@EntityListeners(UserListener::class)
class User(
   @Column(unique = true)
   @Size(min = 2)
   var username: String = "",
   @Size(min = 8, max = 15)
   @Column(unique = true)
   @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$")
   var phoneNumber: String = "",
   @Size(min = 60, max = 60)
   var password: String = "",
   var status: String = "available",
   @Pattern(regexp = "\\A(activated|deactivated)\\z")
   var accountStatus: String = "activated",
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   var id: Long = 0,
   @DateTimeFormat
   var createdAt: Date = Date.from(Instant.now())
)
```

Отлично. Теперь следует понять, что делает каждая аннотация:

- аннотация `@Column` используется для указания свойства, представляющего столбец таблицы. На практике все свойства сущностей представляют столбец в таблице. В приведенном коде используется фрагмент кода `@Column(unique = true)` специально для введения ограничения на уникальность свойств. Это полезно в тех случаях, когда нежелательно, чтобы определенное значение атрибута имело более чем одну запись;
- аннотация `@Size` служит для указания присутствующего в таблице размера атрибута;

- аннотация `@Pattern` указывает шаблон, которому должен соответствовать атрибут таблицы, чтобы он стал корректным;
- аннотация `@Id` определяет свойство, однозначно идентифицирующее сущность (в нашем случае свойство `id`);
- аннотация `@GeneratedValue(strategy = GenerationType.AUTO)` определяет, что желательно, чтобы значение `id` генерировалось автоматически;
- аннотация `@DateTimeFormat` накладывает ограничение на временные метки для значений, которые хранятся в столбце `create_at` пользовательской таблицы.

Пришло время создать класс `UserListener`. Создадим новый пакет под названием `listeners`. Добавим к пакету следующий класс `UserListener`:

```
package com.example.messenger.api.listeners

import com.example.messenger.api.models.User
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
import javax.persistence.PrePersist
import javax.persistence.PreUpdate

class UserListener {
   @PrePersist
   @PreUpdate
   fun hashPassword(user: User) {
      user.password = BCryptPasswordEncoder().encode(user.password)
   }
}
```

Пароли пользователей не следует сохранять в базе данных в виде простого текста. По соображениям безопасности они должны быть соответствующим образом хешированы перед сохранением. Функция `hashPassword()` выполняет процедуру хеширования, заменяя строковое значение, хранящееся в свойстве `password` пользовательского объекта, на его хешированный эквивалент с помощью `BCrypt`. Аннотации `@PrePersist` и `@PreUpdate` указывают, что эту функцию следует вызывать до сохранения или обновления записи пользователя в базе данных.

Теперь создадим объект для сообщений — добавим в пакет `models` класс `Message`, а в этот класс — следующий код:

```
package com.example.messenger.api.models

import org.springframework.format.annotation.DateTimeFormat
import java.time.Instant
import java.util.*
import javax.persistence.*
```

```
@Entity
class Message(
   @ManyToOne(optional = false)
   @JoinColumn(name = "user_id", referencedColumnName = "id")
   var sender: User? = null,
   @ManyToOne(optional = false)
   @JoinColumn(name = "recipient_id", referencedColumnName = "id")
   var recipient: User? = null,
   var body: String? = "",
   @ManyToOne(optional = false)
   @JoinColumn(name="conversation_id", referencedColumnName = "id")
   var conversation: Conversation? = null,
   @Id @GeneratedValue(strategy = GenerationType.AUTO) var id: Long = 0,
   @DateTimeFormat
   var createdAt: Date = Date.from(Instant.now())
)
```

Мы задействовали здесь несколько знакомых аннотаций, а также две новые. Как уже упоминалось, каждое сообщение имеет отправителя и получателя. Как отправители, так и получатели сообщений являются пользователями на платформе обмена сообщениями, поэтому объект сообщения имеет свойства отправителя и получателя типа `User` (Пользователь). Пользователь может быть отправителем многих сообщений, а также получателем многих сообщений. Это отношения, которые необходимо реализовать. Применим аннотацию `@ManyToOne` для реализации этих отношений. Отношения many-to-one (многие-к-одному) не являются обязательными, поэтому используем конструкцию `@ManyToOne(optional = false).@JoinColumn`, которая указывает столбец для присоединения к ассоциации сущностей или коллекции элементов:

```
@JoinColumn(name = "user_id", referencedColumnName = "id")
var sender: User? = null
```

В этом фрагменте кода добавляется атрибут `user_id`, который ссылается на `id` пользователя в таблице сообщений.

При внимательном рассмотрении можно заметить, что свойство `conversation` (беседа) использовалось в классе `Message`. Дело в том, что отправляемые между пользователями сообщения находятся в *потоках бесед* (conversation threads). Проще говоря, каждое сообщение относится к определенной теме. Поэтому необходимо в пакет `models` добавить класс `Conversation`, представляя сущность `conversation`:

```
package com.example.messenger.api.models

import org.springframework.format.annotation.DateTimeFormat
import java.time.Instant
import java.util.*
import javax.persistence.*
```

```
@Entity
class Conversation(
    @ManyToOne(optional = false)
    @JoinColumn(name = "sender_id", referencedColumnName = "id")
    var sender: User? = null,
    @ManyToOne(optional = false)
    @JoinColumn(name = "recipient_id", referencedColumnName = "id")
    var recipient: User? = null,
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = 0,
    @DateTimeFormat
    val createdAt: Date = Date.from(Instant.now())
) {
    @OneToMany(mappedBy = "conversation", targetEntity = Message::class)
    private var messages: Collection<Message>? = null
}
```

Свойству `conversation` **принадлежит большое число сообщений, поэтому в теле класса** `Conversation` **имеется коллекция сообщений.**

Итак, мы практически завершили работу по созданию моделей сущностей. Необходимо только добавить соответствующие коллекции для отправленных и полученных сообщений пользователя:

```
package com.example.messenger.api.models

import com.example.messenger.api.listeners.UserListener
import org.springframework.format.annotation.DateTimeFormat
import java.time.Instant
import java.util.*
import javax.persistence.*
import javax.validation.constraints.Pattern
import javax.validation.constraints.Size

@Entity
@Table(name = "`user`")
@EntityListeners(UserListener::class)
class User(
    @Column(unique = true)
    @Size(min = 2)
    var username: String = "",
    @Size(min = 8, max = 15)
    @Column(unique = true)
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$")
    var phoneNumber: String = "",
    @Size(min = 60, max = 60)
```

```
    var password: String = "",
    var status: String = "available",
    @Pattern(regexp = "\\A(activated|deactivated)\\z")
    var accountStatus: String = "activated",
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = 0,
    @DateTimeFormat
    var createdAt: Date = Date.from(Instant.now())
) {
    // коллекция отправленных сообщений
    @OneToMany(mappedBy = "sender", targetEntity = Message::class)
    private var sentMessages: Collection<Message>? = null
    // коллекция полученых сообщений
    @OneToMany(mappedBy = "recipient", targetEntity = Message::class)
    private var receivedMessages: Collection<Message>? = null
}
```

Вот и все! Объекты созданы. Для того чтобы понять созданные сущности, а также их отношения, рассмотрим диаграмму отношений сущностей (Entity Relationship, E-R), где отображены созданные сущности и отношения между ними (рис. 4.8).

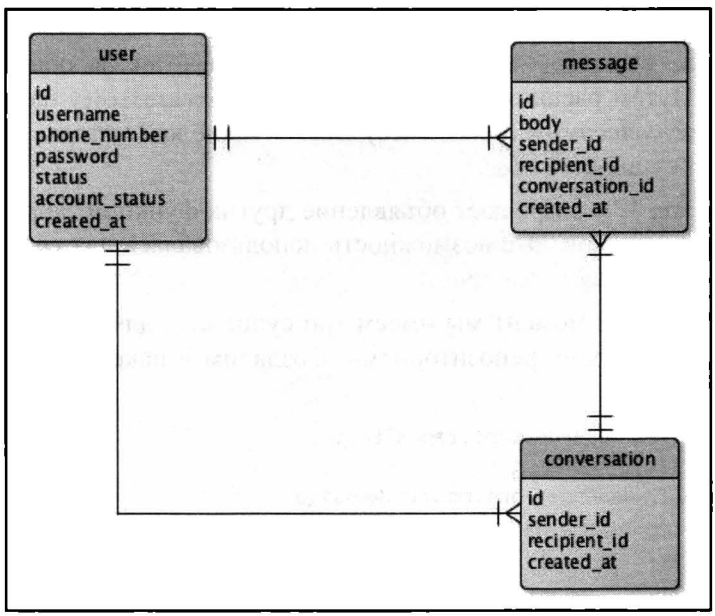


Рис. 4.8. Диаграмма отношений сущностей проектируемой системы

Согласно приведенной на рис. 4.8 диаграмме, у пользователя имеется большое число сообщений, среди которых присутствуют сообщения, принадлежащие пользователю, и сообщения, относящиеся собственно к беседе, причем беседа располагает

значительным числом сообщений. Кроме того, пользователь участвует в большом количестве бесед.

Процесс создания необходимых моделей затрудняет одна проблема — отсутствует возможность доступа к хранящимся в этих сущностях данным. Для ее устранения необходимо сформировать *репозитории*.

Создание репозиториев

Благодаря Spring Data JPA можно автоматически создавать реализации репозитория на основе интерфейса репозитория. Рассмотрим, как это работает, создавая репозиторий для доступа к сущностям `User`. Создайте пакет `repositaries` и включите в него файл UserRepository.kt:

```
package com.example.messenger.api.repositories

import com.example.messenger.api.models.User
import org.springframework.data.repository.CrudRepository
interface UserRepository : CrudRepository<User, Long> {

   fun findByUsername(username: String): User?
   fun findByPhoneNumber(phoneNumber: String): User?
}
```

Параметр `UserRepository` расширяет интерфейс `CrudRepository`. Тип сущности и тип идентификатора, с которыми он работает, указываются в общих параметрах `CrudRepository`. Путем расширения `CrudRepository`, `UserRepository` наследует методы для работы с *постоянством* (persistence) `User` — такие как методы сохранения, поиска и удаления сущностей `User`.

Кроме того, Spring JPA разрешает объявление других функций запроса с использованием сигнатур методов. Эта возможность использовалась для создания функций `findByUsername()` и `findByPhoneNumber()`.

Поскольку на текущий момент мы имеем три сущности, для запросов к ним необходимо располагать тремя репозиториями. Создадим в пакете `repositories` интерфейс `MessageRepository`:

```
package com.example.messenger.api.repositories

import com.example.messenger.api.models.Message
import org.springframework.data.repository.CrudRepository

interface MessageRepository : CrudRepository<Message, Long> {
   fun findByConversationId(conversationId: Long): List<Message>
}
```

Заметим, что в предыдущем методе сигнатур указан `List<Message>` как тип возврата. Spring JPA автоматически распознает его и при вызове `findByConversationId()` возвращает перечень элементов `Message`.

И наконец, реализуем интерфейс `ConversationRepository`:

```
package com.example.messenger.api.repositories

import com.example.messenger.api.models.Conversation
import org.springframework.data.repository.CrudRepository

interface ConversationRepository : CrudRepository<Conversation, Long> {
   fun findBySenderId(id: Long): List<Conversation>

   fun findByRecipientId(id: Long): List<Conversation>

   fun findBySenderIdAndRecipientId(senderId: Long,
      recipientId: Long): Conversation?
}
```

Теперь, после установления сущностей и необходимых репозиториев для запроса этих сущностей, можно начать работу по реализации бизнес-логики серверной части Messenger. Но для этого необходимо кое-что узнать о сервисах и особенностях их реализации.

Сервисы и реализации сервисов

Реализация сервиса — это бин spring, который аннотируется с помощью `@Service`. Бизнес-логика для spring-приложений чаще всего используется в реализации сервиса. *Сервис*, с другой стороны, является интерфейсом с сигнатурами функций для поведения приложения, которые должны быть реализованы при помощи реализации классов. Простой способ уяснить разницу между ними состоит в понимании, что сервис — это интерфейс, а реализация сервиса — это класс, реализующий сервис.

Перейдем к созданию некоторых сервисов и реализации сервисов. Создайте сервисный пакет. В нашем случае в него будут внесены как сервисы, так и реализации сервисов. Сформируйте в этом пакете интерфейс `UserService` со следующими кодами:

```
package com.example.messenger.api.services

import com.example.messenger.api.models.User

interface UserService {
   fun attemptRegistration(userDetails: User): User

   fun listUsers(currentUser): List<User>

   fun retrieveUserData(username: String): User?

   fun retrieveUserData(id: Long): User?

   fun usernameExists(username: String): Boolean
}
```

В представленном интерфейсе `UserService` определены функции, которые следует объявить классами, реализующими `UserService`. Это означает, что сервис `UserService` уже готов к реализации, и мы ее сейчас создадим, добавив к сервисному пакету класс `UserServiceImpl`, в котором нам необходимо иметь переопределяющие функции для `attemptRegistration()`, `listUsers()`, `retrieveUserData()` и `usernameExists()`:

```
package com.example.messenger.api.services

import com.example.messenger.api.exceptions.InvalidUserIdException
import com.example.messenger.api.exceptions.UserStatusEmptyException
import com.example.messenger.api.exceptions.UsernameUnavailableException
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.UserRepository
import org.springframework.stereotype.Service

@Service
class UserServiceImpl(val repository: UserRepository) : UserService {

   @Throws(UsernameUnavailableException::class)
   override fun attemptRegistration(userDetails: User): User {
      if (!usernameExists(userDetails.username)) {
         val user = User()
         user.username = userDetails.username
         user.phoneNumber = userDetails.phoneNumber
         user.password = userDetails.password
         repository.save(user)
         obscurePassword(user)
         return user
      }
      throw UsernameUnavailableException("The username
                  ${userDetails.username} is unavailable.")
   }

   @Throws(UserStatusEmptyException::class)
   fun updateUserStatus(currentUser: User, updateDetails: User): User {
      if (!updateDetails.status.isEmpty()) {
         currentUser.status = updateDetails.status
         repository.save(currentUser)
         return currentUser
      }
      throw UserStatusEmptyException()
   }

   override fun listUsers(currentUser: User): List<User> {
      return repository.findAll().mapTo(ArrayList(), { it })
                     .filter{ it != currentUser }
   }
```

```
    override fun retrieveUserData(username: String): User? {
        val user = repository.findByUsername(username)
        obscurePassword(user)
        return user
    }

    @Throws(InvalidUserIdException::class)
    override fun retrieveUserData(id: Long): User {
        val userOptional = repository.findById(id)
        if (userOptional.isPresent) {
            val user = userOptional.get()
            obscurePassword(user)
            return user
        }
        throw InvalidUserIdException("A user with an id of '$id'
                                     does not exist.")
    }

    override fun usernameExists(username: String): Boolean {
        return repository.findByUsername(username) != null
    }

    private fun obscurePassword(user: User?) {
        user?.password = "XXX XXXX XXX"
    }
}
```

В определении основного конструктора для `UserServiceImpl` экземпляр `UserRepository` указан как требуемый аргумент. Не следует беспокоиться о самостоятельной передаче подобного аргумента. Spring распознает, что `UserServiceImpl` нуждается в экземпляре `UserRepository`, и предоставляет его классу посредством внедрения зависимости. В дополнение к реализованным функциям объявлена функция `obscurePassword()`, которая просто хеширует пароли внутри сущности `User` с помощью `XXX XXXX XXX`.

По-прежнему, в духе создания и реализации сервиса, продолжим работу и добавим немного дополнительной информации для сообщений и бесед. Вставим в сервис интерфейс `MessageService`:

```
package com.example.messenger.api.services

import com.example.messenger.api.models.Message
import com.example.messenger.api.models.User

interface MessageService {

    fun sendMessage(sender: User, recipientId: Long,
                    messageText: String): Message
}
```

Для интерфейса `sendMessage()` добавлен единственный метод сигнатуры, который должен быть переопределен с помощью `MessageServiceImpl`. Далее приводится реализация сервиса сообщений:

```
package com.example.messenger.api.services

import com.example.messenger.api.exceptions.MessageEmptyException
import
com.example.messenger.api.exceptions.MessageRecipientInvalidException
import com.example.messenger.api.models.Conversation
import com.example.messenger.api.models.Message
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.ConversationRepository
import com.example.messenger.api.repositories.MessageRepository
import com.example.messenger.api.repositories.UserRepository
import org.springframework.stereotype.Service

@Service
class MessageServiceImpl(val repository: MessageRepository,
             val conversationRepository: ConversationRepository,
             val conversationService: ConversationService,
             val serRepository: UserRepository) : MessageService {

   @Throws(MessageEmptyException::class,
          MessageRecipientInvalidException::class)
   override fun sendMessage(sender: User, recipientId: Long,
                            messageText: String): Message {
      val optional = userRepository.findById(recipientId)

         if (optional.isPresent) {
            val recipient = optional.get()

            if (!messageText.isEmpty()) {
               val conversation: Conversation = if (conversationService
                              .conversationExists(sender, recipient)) {

               conversationService.getConversation(sender, recipient)
                                  as Conversation
            } else {
               conversationService.createConversation(sender, recipient)
            }
            conversationRepository.save(conversation)

            val message = Message(sender, recipient, messageText, conversation)
            repository.save(message)
            return message
         }
```

```
        } else {
            throw MessageRecipientInvalidException("The recipient id
                                '$recipientId' is invalid.")
        }
        throw MessageEmptyException()
    }
}
```

В приведенной реализации метода `sendMessage()` сначала проверяется, не пусто ли содержимое сообщения. Если это не так, функция проверяет, имеется ли активный диалог (`Conversation`) между отправителем и получателем. Если такой диалог имеется, он извлекается и сохраняется в переменной `conversation`, в противном случае между двумя пользователями создается новый активный диалог (`Conversation`) и тоже сохраняется в переменной `conversation`. Когда переменная `conversation` сохранена, создается и сохраняется сообщение.

Теперь можно реализовать интерфейсы `ConversationService` и `ConversationServiceImpl`. Создайте в `services` интерфейс `ConversationService` и добавьте следующий код:

```
package com.example.messenger.api.services

import com.example.messenger.api.models.Conversation
import com.example.messenger.api.models.User

interface ConversationService {

    fun createConversation(userA: User, userB: User): Conversation
    fun conversationExists(userA: User, userB: User): Boolean
    fun getConversation(userA: User, userB: User): Conversation?
    fun retrieveThread(conversationId: Long): Conversation
    fun listUserConversations(userId: Long): List<Conversation>
    fun nameSecondParty(conversation: Conversation, userId: Long): String
}
```

Теперь у нас имеется шесть сигнатур функций, а именно: `createConversation()`, `conversationExists()`, `getConversation()`, `retrieveThread()`, `listUserConversations()` и `nameSecondParty()`. Теперь добавим к сервисам `ConversationServiceImpl` и реализуем первые три метода: `createConversation()`, `conversationExists()` и `getConversation()`. Их реализация показана в следующем фрагменте кода:

```
package com.example.messenger.api.services

import com.example.messenger.api.exceptions.ConversationIdInvalidException
import com.example.messenger.api.models.Conversation
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.ConversationRepository
import org.springframework.stereotype.Service
```

```
@Service
class ConversationServiceImpl(val repository: ConversationRepository) :
      ConversationService {

   override fun createConversation(userA: User, userB: User):
               Conversation {
      val conversation = Conversation(userA, userB)
      repository.save(conversation)
      return conversation
   }

   override fun conversationExists(userA: User, userB: User): Boolean {
      return if (repository.findBySenderIdAndRecipientId
                (userA.id, userB.id) != null)
      true
      else repository.findBySenderIdAndRecipientId
                (userB.id, userA.id) != null
   }

  override fun getConversation(userA: User, userB: User): Conversation? {
     return when {
       repository.findBySenderIdAndRecipientId(userA.id,
                              userB.id) != null ->
       repository.findBySenderIdAndRecipientId(userA.id, userB.id)
       repository.findBySenderIdAndRecipientId(userB.id,
                              userA.id) != null ->
       repository.findBySenderIdAndRecipientId(userB.id, userA.id)
      else -> null
      }
   }
}
```

Добавив первые три метода, включите оставшиеся три метода: `retrieveThread()`, `listUserConversations()` **и** `nameSecondParty()` **— под** `ConversationServiceImpl`:

```
override fun retrieveThread(conversationId: Long): Conversation {
   val conversation = repository.findById(conversationId)

   if (conversation.isPresent) {
      return conversation.get()
   }
   throw ConversationIdInvalidException("Invalid conversation id '$conversationId'")
}

override fun listUserConversations(userId: Long): ArrayList<Conversation> {
   val conversationList: ArrayList<Conversation> = ArrayList()
```

```
    conversationList.addAll(repository.findBySenderId(userId))
    conversationList.addAll(repository.findByRecipientId(userId))

    return conversationList
}

override fun nameSecondParty(conversation: Conversation, userId: Long): String {
        return if (conversation.sender?.id == userId) {
            conversation.recipient?.username as String
        } else {
            conversation.sender?.username as String
    }
}
```

Возможно, вы заметили, что в классах реализации сервиса пропущены исключения различных типов. Поскольку эти исключения еще не созданы, самое время это сделать. Кроме того, для каждого из этих исключений нужно создать обработчик `ExceptionHandler`. Обработчики исключений будут отправлять клиентам соответствующие сообщения об ошибках в сценариях, где генерируются исключения.

Создайте пакет `exceptions` и добавьте в него файл **AppExceptions.kt**. Включите в этот файл следующий код:

```
package com.example.messenger.api.exceptions

class UsernameUnavailableException(override val message: String) :
RuntimeException()

class InvalidUserIdException(override val message: String)
RuntimeException()

class MessageEmptyException(override val message: String = "A message
cannot be empty.") : RuntimeException()

class MessageRecipientInvalidException(override val message: String) :
RuntimeException()

class ConversationIdInvalidException(override val message: String) :
RuntimeException()

class UserDeactivatedException(override val message: String) :
RuntimeException()

class UserStatusEmptyException(override val message: String = "A user's
status cannot be empty") : RuntimeException()
```

Каждое исключение расширяет `RuntimeException`, если эти исключения случаются во время выполнения сервера. Все исключения также обладают свойством `message`. Как

следует из названия, это сообщение является сообщением об исключении. Теперь, когда исключения добавлены, нужно создать классы рекомендаций для контроллеров. Классы `ControllerAdvice` используются для обработки возникающих в приложении Spring ошибок. Они созданы с использованием аннотации `@ControllerAdvice`. Кроме того, рекомендация контроллера является типом компонента Spring. Создадим класс рекомендации контроллера для обработки некоторых из представленных исключений.

Рассмотрим исключения `UsernameUnavailableException`, `InvalidUserIdException` и `UserStatusEmptyException`. Заметим, что все эти три исключения относятся к пользователю. Таким образом, назовем рекомендацию контроллера, который обслуживает эти исключения: `UserControllerAdvice`. Создадим пакет `components` и добавим к нему следующий класс `UserControllerAdvice`:

```
package com.example.messenger.api.components

import com.example.messenger.api.constants.ErrorResponse
import com.example.messenger.api.constants.ResponseConstants
import com.example.messenger.api.exceptions.InvalidUserIdException
import com.example.messenger.api.exceptions.UserStatusEmptyException
import com.example.messenger.api.exceptions.UsernameUnavailableException
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.ControllerAdvice
import org.springframework.web.bind.annotation.ExceptionHandler

@ControllerAdvice
class UserControllerAdvice {

   @ExceptionHandler(UsernameUnavailableException::class)
   fun usernameUnavailable(usernameUnavailableException:
                           UsernameUnavailableException):

         ResponseEntity<ErrorResponse> {
      val res = ErrorResponse(ResponseConstants.USERNAME_UNAVAILABLE
                        .value, usernameUnavailableException.message)
      return ResponseEntity.unprocessableEntity().body(res)
   }

   @ExceptionHandler(InvalidUserIdException::class)
   fun invalidId(invalidUserIdException: InvalidUserIdException):
         ResponseEntity<ErrorResponse> {
      val res = ErrorResponse(ResponseConstants.INVALID_USER_ID.value,
                              invalidUserIdException.message)
      return ResponseEntity.badRequest().body(res)
   }
```

```
    @ExceptionHandler(UserStatusEmptyException::class)
    fun statusEmpty(userStatusEmptyException: UserStatusEmptyException):
    ResponseEntity<ErrorResponse> {
        val res = ErrorResponse(ResponseConstants.EMPTY_STATUS.value,
        userStatusEmptyException.message)
        return ResponseEntity.unprocessableEntity().body(res)
    }
}
```

Сейчас у нас определена функция для обслуживания каждого из трех исключений, которые могут быть сгенерированы. Аннотируем каждую из этих функций с помощью аннотации `@ExceptionHanlder()`. Аннотация `@ExceptionHanlder()` принимает ссылку на класс для исключения, обрабатываемого функцией. Каждая функция принимает один аргумент, который является экземпляром сгенерированного исключения. Кроме того, все определенные функции возвращают экземпляр `ResponseEntity<ErrorResponse>`. `ResponseEntity` (сущность ответа) представляет собой полностью отправленный клиенту HTTP-ответ (`ErrorResponse`). Но его у нас еще нет. Так что создайте пакет `constants` и добавьте к нему следующий класс `ErrorResponse`:

```
package com.example.messenger.api.constants
class ErrorResponse(val errorCode: String, val errorMessage: String)
```

`ErrorResponse` является простым классом с двумя свойствами: `errorCode` и `errorMessage`. Перед продолжением работы к пакету `constants` добавьте следующий enum-класс `ResponseConstants`:

```
package com.example.messenger.api.constants
enum class ResponseConstants(val value: String) {
    SUCCESS("success"), ERROR("error"),
    USERNAME_UNAVAILABLE("USR_0001"),
    INVALID_USER_ID("USR_002"),
    EMPTY_STATUS("USR_003"),
    MESSAGE_EMPTY("MES_001"),
    MESSAGE_RECIPIENT_INVALID("MES_002"),
    ACCOUNT_DEACTIVATED("GLO_001")
}
```

Теперь создадим еще три класса рекомендаций контроллеров. Этими классами являются: `MessageControllerAdvice`, `ConversationControllerAdvice` и `RestControllerAdvice`. Класс `RestControllerAdvice` определит обработчики исключений для ошибок, которые могут случиться в любом месте на сервере в течение всего времени выполнения.

Далее приводится класс `MessageControllerAdvice`:

```
package com.example.messenger.api.components

import com.example.messenger.api.constants.ErrorResponse
import com.example.messenger.api.constants.ResponseConstants
```

```
import com.example.messenger.api.exceptions.MessageEmptyException
import com.example.messenger.api.exceptions.MessageRecipientInvalidException
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.ControllerAdvice
import org.springframework.web.bind.annotation.ExceptionHandler

@ControllerAdvice
class MessageControllerAdvice {
   @ExceptionHandler(MessageEmptyException::class)
   fun messageEmpty(messageEmptyException: MessageEmptyException):
      ResponseEntity<ErrorResponse> {
   //ErrorResponse object creation
   val res = ErrorResponse(ResponseConstants.MESSAGE_EMPTY.value,
                           messageEmptyException.message)

   // Возврат ResponseEntity, содержащего соответствующий ErrorResponse
   return ResponseEntity.unprocessableEntity().body(res)
}

@ExceptionHandler(MessageRecipientInvalidException::class)
    fun messageRecipientInvalid(messageRecipientInvalidException:
                           MessageRecipientInvalidException):
      ResponseEntity<ErrorResponse> {
      val res = ErrorResponse(ResponseConstants.MESSAGE_RECIPIENT_INVALID
                        .value, messageRecipientInvalidException.message)
      return ResponseEntity.unprocessableEntity().body(res)
   }
}
```

Затем добавим класс `ConversationControllerAdvice`, **который выглядит следующим образом:**

```
package com.example.messenger.api.components

import com.example.messenger.api.constants.ErrorResponse
import com.example.messenger.api.exceptions.ConversationIdInvalidException
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.ControllerAdvice
import org.springframework.web.bind.annotation.ExceptionHandler

@ControllerAdvice
class ConversationControllerAdvice {
   @ExceptionHandler
   fun conversationIdInvalidException(conversationIdInvalidException:
         ConversationIdInvalidException): ResponseEntity<ErrorResponse> {
```

```
        val res = ErrorResponse("", conversationIdInvalidException.message)
        return ResponseEntity.unprocessableEntity().body(res)
    }
}
```

И наконец, добавим класс `RestControllerAdvice`:

```
package com.example.messenger.api.components

import com.example.messenger.api.constants.ErrorResponse
import com.example.messenger.api.constants.ResponseConstants
import com.example.messenger.api.exceptions.UserDeactivatedException
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.ControllerAdvice
import org.springframework.web.bind.annotation.ExceptionHandler

@ControllerAdvice
class RestControllerAdvice {

    @ExceptionHandler(UserDeactivatedException::class)
    fun userDeactivated(userDeactivatedException:
                        UserDeactivatedException):
            ResponseEntity<ErrorResponse> {
        val res = ErrorResponse(ResponseConstants.ACCOUNT_DEACTIVATED
                        .value, userDeactivatedException.message)
        // Возврат ответа ошибка HTTP 403 неудачной авторизации
        return ResponseEntity(res, HttpStatus.UNAUTHORIZED)
    }
}
```

К настоящему моменту нами реализована бизнес-логика, и теперь почти все готово для облегченного ввода в наш API HTTP-запросов через конечные точки REST. Прежде чем выполнить это, защитим наш API.

Ограничение доступа к API

С точки зрения поддержки мер безопасности, не следует позволять любому пользователю иметь доступ к ресурсам API RESTful. Необходимо разработать способ для ограничения доступа к нашему серверу только кру́гом зарегистрированных пользователей. Реализуем это, применяя Spring Security и JSON Web Tokens (JWT).

Spring Security

Spring Security представляет собой настраиваемую структуру управления доступом для приложений Spring. Это общепринятый стандарт для защиты приложений, созданных с помощью Spring. Поскольку он выбран для добавления зависимости

Security в начале создания этого проекта, не нужно добавлять зависимость Spring Security в файл pom.xml, поскольку она уже включена.

JSON Web Tokens

Согласно информации, размещенной на веб-сайте JWT (**https://tools.ietf.org/html/rfc7519**): *JSON Web Tokens является открытым, стандартным методом для безопасного представления требований (representing claims), возникающих между двумя сторонами.* Для реализации аутентификации в приложениях токены JWT позволяют выполнять декодирование, проверку и генерацию токенов JWT. Токены JWT можно использовать вместе со Spring Boot. В следующих разделах показано, как применять комбинацию JWT и Spring Security для защиты серверной части Messenger.

Первое, что нужно сделать в приложении Spring для начала работы с JWT, — добавить его зависимость в файл проекта pom.xml:

```
<dependencies>
   ...
   <dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt</artifactId>
      <version>0.7.0</version>
   </dependency>
</dependencies>
```

При включении новой зависимости Maven в файл pom.xml, IntelliJ попросит вас импортировать новые зависимости (рис. 4.9).

Рис. 4.9. Предложение импортировать новые зависимости

Получив такое предложение, щелкните на ссылке **Import Changes** (Импорт изменений), и зависимость JWT будет импортирована в проект.

Конфигурирование веб-безопасности

Первое, что нужно сделать, — это создать пользовательскую конфигурацию веб-безопасности. Добавьте в пакет com.example.messenger.api пакет конфигурации, а в него — класс `WebSecurityConfig` и введите следующий код:

```
package com.example.messenger.api.config

import com.example.messenger.api.filters.JWTAuthenticationFilter
import com.example.messenger.api.filters.JWTLoginFilter
import com.example.messenger.api.services.AppUserDetailsService
import org.springframework.context.annotation.Configuration
import org.springframework.http.HttpMethod
import org.springframework.security.config.annotation.authentication.
                                                builders.AuthenticationManagerBuilder
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import org.springframework.security.config.annotation.web.configuration.
                                                                    EnableWebSecurity
import org.springframework.security.config.annotation.web.configuration.
                                                            WebSecurityConfigurerAdapter
import org.springframework.security.core.userdetails.UserDetailsService
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
import org.springframework.security.web.authentication.
                                                UsernamePasswordAuthenticationFilter

@Configuration
@EnableWebSecurity
class WebSecurityConfig(val userDetailsService: AppUserDetailsService)
        : WebSecurityConfigurerAdapter() {

  @Throws(Exception::class)
  override fun configure(http: HttpSecurity) {
     http.csrf().disable().authorizeRequests()
                .antMatchers(HttpMethod.POST, "/users/registrations")
                .permitAll()
                .antMatchers(HttpMethod.POST, "/login").permitAll()
                .anyRequest().authenticated()
               .and()
```

Позволим применить `Filter` к запросам `/login`:

```
               .addFilterBefore(JWTLoginFilter("/login",
                  authenticationManager()),
                  UsernamePasswordAuthenticationFilter::class.java)
```

Позволим фильтровать другие запросы для проверки наличия JWT в заголовке:

```
               .addFilterBefore(JWTAuthenticationFilter(),
                     UsernamePasswordAuthenticationFilter::class.java)
     }

     @Throws(Exception::class)
     override fun configure(auth: AuthenticationManagerBuilder) {
        auth.userDetailsService<UserDetailsService>(userDetailsService)
```

```
            .passwordEncoder(BCryptPasswordEncoder())
    }
}
```

Для аннотирования `WebSecurityConfig` применяется аннотация `@EnableWebSecurity`. Это включает поддержку веб-безопасности Spring Security. Кроме того, `WebSecurityConfig` расширяет `WebSecurityConfigurerAdapter` и переопределяет некоторые из его методов `configure()` для добавления ряда настроек в конфигурацию веб-безопасности.

Метод `configure(HttpSecurity)` определяет, какие URL-пути должны быть защищены, а какие — нет. В `WebSecurityConfig` разрешены все запросы `POST` к путям `/users/registrations` и `/login`. Эти две конечные точки не должны быть защищены, поскольку пользователь не может пройти аутентификацию до входа в систему или до регистрации на платформе. Кроме того, для запросов добавлены фильтры. Запросы к `/login` будут отфильтрованы с помощью `JWTLoginFilter` (еще предстоит это реализовать), а все запросы, которые не прошли проверку подлинности и не были разрешены, будут отфильтрованы `JWTAuthenticationFilter` (это также предстоит еще реализовать).

Параметр `configure(AuthenticationManagerBuilder)` устанавливает `UserDetailsService` и указывает применяемый кодировщик пароля.

Итак, у нас имеется ряд использованных, но еще не реализованных классов. Начнем с реализации класса `JWTLoginFilter`. Создадим новый пакет под названием filters и добавим в него следующий класс `JWTLoginFilter`:

```
package com.example.messenger.api.filters

import com.example.messenger.api.security.AccountCredentials
import com.example.messenger.api.services.TokenAuthenticationService
import com.fasterxml.jackson.databind.ObjectMapper
import org.springframework.security.authentication.AuthenticationManager
import org.springframework.security.authentication.
                                                UsernamePasswordAuthenticationToken
import org.springframework.security.core.Authentication
import org.springframework.security.core.AuthenticationException
import org.springframework.security.web.authentication.
                                              AbstractAuthenticationProcessingFilter
import org.springframework.security.web.util.matcher.AntPathRequestMatcher
import javax.servlet.FilterChain
import javax.servlet.ServletException
import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse
import java.io.IOException

class JWTLoginFilter(url: String, authManager: AuthenticationManager) :
AbstractAuthenticationProcessingFilter(AntPathRequestMatcher(url)){
```

```
    init {
        authenticationManager = authManager
    }

    @Throws(AuthenticationException::class, IOException::class,
                                                    ServletException::class)
    override fun attemptAuthentication( req: HttpServletRequest,
                                        res: HttpServletResponse): Authentication{
    val credentials = ObjectMapper()
            .readValue(req.inputStream, AccountCredentials::class.java)
    return authenticationManager.authenticate(
        UsernamePasswordAuthenticationToken(
        credentials.username,
        credentials.password,
        emptyList()
        )
    )
}

@Throws(IOException::class, ServletException::class)
override fun successfulAuthentication(
    req: HttpServletRequest,
    res: HttpServletResponse, chain: FilterChain,
    auth: Authentication) {
        TokenAuthenticationService.addAuthentication(res, auth.name)
    }
}
```

Класс `JWTLoginFilter` принимает строку URL и экземпляр `AuthenticationManager` в качестве аргументов для своего первичного конструктора. Также можно видеть, как расширяется `AbstractAuthenticationProcessingFilter`. Этот фильтр перехватывает входящие HTTP-запросы к серверу и пытается их аутентифицировать. Метод `attemptAuthentication()` выполняет фактический процесс аутентификации. Экземпляр `ObjectMapper()` задействуется для чтения учетных данных, присутствующих в запросе через HTTP, после чего `authenticationManager` используется для аутентификации запроса.

`AccountCredentials` является еще одним классом, который предстоит реализовать. Создайте новый пакет под названием security и добавьте к нему файл AccountCredentials.kt:

```
package com.example.messenger.api.security
class AccountCredentials {
    lateinit var username: String
    lateinit var password: String
}
```

Здесь мы имеем переменные для `username` и `password`, поскольку они будут применяться для аутентификации пользователя.

Метод `SuccessfulAuthentication()` вызывается при успешной аутентификации пользователя. Единственная выполняемая функцией задача — добавление в заголовок авторизации HTTP-ответа токенов аутентификации. Фактическое добавление этого заголовка выполняется с помощью `TokenAuthenticationService.addAuthentication()`. Добавим этот сервис в наш пакет services:

```
package com.example.messenger.api.services

import io.jsonwebtoken.Jwts
import io.jsonwebtoken.SignatureAlgorithm
import org.springframework.security.authentication.
                                                UsernamePasswordAuthenticationToken
import org.springframework.security.core.Authentication
import org.springframework.security.core.GrantedAuthority
import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse
import java.util.Date
import java.util.Collections.emptyList
internal object TokenAuthenticationService {
   private val TOKEN_EXPIRY: Long = 864000000
   private val SECRET = "$78gr43g7g8feb8we"
   private val TOKEN_PREFIX = "Bearer"
   private val AUTHORIZATION_HEADER_KEY = "Authorization"

   fun addAuthentication(res: HttpServletResponse, username: String) {
      val JWT = Jwts.builder()
                    .setSubject(username)
                    .setExpiration(Date(System.currentTimeMillis() +
                                                          TOKEN_EXPIRY))
                    .signWith(SignatureAlgorithm.HS512, SECRET)
                    .compact()
   res.addHeader(AUTHORIZATION_HEADER_KEY, "$TOKEN_PREFIX $JWT")
}

fun getAuthentication(request: HttpServletRequest): Authentication? {
   val token = request.getHeader(AUTHORIZATION_HEADER_KEY)
   if (token != null) {
```

Выполним синтаксический разбор токена:

```
      val user = Jwts.parser().setSigningKey(SECRET)
                     .parseClaimsJws(token.replace(TOKEN_PREFIX, ""))
                     .body.subject
         if (user != null)
            return UsernamePasswordAuthenticationToken(user, null,
               emptyList<GrantedAuthority>())
         }
```

```
        return null
    }
}
```

Как следует из названий, `addAuthentication()` добавляет токен аутентификации к заголовку `Authorization` для отклика HTTP, и `getAuthentication()` аутентифицирует этого пользователя.

Добавим следующий класс `JWTAuthenticationFilter` к пакету filters:

```
package com.example.messenger.api.filters

import com.example.messenger.api.services.TokenAuthenticationService
import org.springframework.security.core.context.SecurityContextHolder
import org.springframework.web.filter.GenericFilterBean
import javax.servlet.FilterChain
import javax.servlet.ServletException
import javax.servlet.ServletRequest
import javax.servlet.ServletResponse
import javax.servlet.http.HttpServletRequest
import java.io.IOException
class JWTAuthenticationFilter : GenericFilterBean() {
    @Throws(IOException::class, ServletException::class)
    override fun doFilter(request: ServletRequest,
                          response: ServletResponse,
                          filterChain: FilterChain) {
        val authentication = TokenAuthenticationService
                    .getAuthentication(request as HttpServletRequest)
        SecurityContextHolder.getContext().authentication = authentication
        filterChain.doFilter(request, response)
    }
}
```

Функция `doFilter()` из `JWTAuthenticationFilter` вызывается контейнером всякий раз, когда пара запрос/ответ проходит через ряд фильтров в результате клиентского запроса на ресурс. Экземпляр `FilterChain`, перешедший в `doFilter()`, позволяет фильтру передавать запрос и ответ следующей сущности из ряда фильтров.

Наконец, необходимо, как обычно, реализовать класс `AppUserDetailsService`, помещая его в пакет services нашего проекта:

```
package com.example.messenger.api.services

import com.example.messenger.api.repositories.UserRepository
import org.springframework.security.core.GrantedAuthority
import org.springframework.security.core.authority.SimpleGrantedAuthority
import org.springframework.security.core.userdetails.User
import org.springframework.security.core.userdetails.UserDetails
import org.springframework.security.core.userdetails.UserDetailsService
```

```
import org.springframework.security.core.userdetails.UsernameNotFoundException
import org.springframework.stereotype.Component
import java.util.ArrayList

@Component
class AppUserDetailsService(val userRepository: UserRepository)
UserDetailsService {

   @Throws(UsernameNotFoundException::class)
   override fun loadUserByUsername(username: String): UserDetails {
      val user = userRepository.findByUsername(username) ?:
                 throw UsernameNotFoundException("A user with the
                                                  username $username doesn't exist")
      return User(user.username, user.password,
                  ArrayList<GrantedAuthority>())
   }
}
```

Функция `loadUsername(String)` предпринимает попытку загрузить `UserDetails` пользователя, соответствующий имени пользователя, переданному в функцию. Если пользователь, имеющий указанное имя, не может быть найден, генерируется исключение `UsernameNotFoundException`.

На этом Spring Security успешно настроен. Теперь с помощью контроллеров можно предоставить некоторые функциональные возможности API через конечные точки RESTful.

Доступ к ресурсам сервера через конечные точки RESTful

Пока что речь шла о создании моделей, компонентов, сервисов и реализации сервисов, а также об интегрировании Spring Security в приложение Messenger. Но мы до сих пор не касались одного аспекта, суть которого заключается в создании средств, с помощью которых внешние клиенты могут взаимодействовать с Messenger API. Выполним это путем создания классов контроллеров, обрабатывающих запросы, поступающие из различных путей HTTP-запросов. Как всегда, первое, что следует сделать, — это создать пакет, содержащий контроллеры, которые необходимо сформировать. Создадим пакет controllers.

Первым реализуемым контроллером станет `UserController`. Этот контроллер сопоставляет относящиеся к пользовательскому ресурсу HTTP-запросы с действиями в классе, которые обрабатывают и отвечают на HTTP-запрос. Прежде всего для облегчения регистрации новых пользователей нужна конечная точка. Вызовем действие, которое обрабатывает подобный запрос на регистрацию действия `create`. Далее приводится код `UserController` с действием `create`:

```
package com.example.messenger.api.controllers

import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.UserRepository
import com.example.messenger.api.services.UserServiceImpl
import org.springframework.http.ResponseEntity
import org.springframework.validation.annotation.Validated
import org.springframework.web.bind.annotation.*
import javax.servlet.http.HttpServletRequest

@RestController
@RequestMapping("/users")
class UserController(val userService: UserServiceImpl,
                     val userRepository: UserRepository) {

   @PostMapping
   @RequestMapping("/registrations")

   fun create(@Validated @RequestBody userDetails: User):
              ResponseEntity<User> {
      val user = userService.attemptRegistration(userDetails)
      return ResponseEntity.ok(user)
   }
}
```

Класс контроллера аннотируется с помощью аннотаций `@RestController` и `@RequestMapping`. Аннотация `@RestController` определяет, что классом является REST-контроллер. Аннотация `@RequestMapping` в том виде, в котором она применялась ранее с классом `UserController`, сопоставляет все запросы с путями, начинающимися с `/users` и с `UserController`.

Функция `create` аннотируется с помощью аннотаций `@PostMapping` и `@RequestMapping("/registrations")`. Комбинация этих двух аннотаций сопоставляет все запросы `POST` с путем `/users/registrations` и функцией `create`. Экземпляр `User`, аннотированный с помощью аннотаций `@Validated` и `@RequestBody` передается функции `create`. Аннотация `@RequestBody` привязывает значения JSON, отправленные в теле запроса `POST`, к `userDetails`. Аннотация `@Validated` гарантирует, что JSON-параметры действительны. Теперь, при наличии готовой конечной точки, проверим ее. Запустите приложение и перейдите к окну терминала. Отправьте запрос к Messenger API, применяя `CURL`, следующим образом:

```
curl -H "Content-Type: application/json" -X POST -d
'{"username":"kevin.stacey",
   "phoneNumber":"5472457893",
   "password":"Hello123"}'
http://localhost:8080/users/registrations
```

Сервер создаст пользователя и направит ответ, подобный следующему:

```
{
   "username":"kevin.stacey",
   "phoneNumber":"5472457893",
   "password":"XXX XXXX XXX",
   "status":"available",
   "accountStatus":"activated",
   "id":6,"createdAt":1508579448634
}
```

Все отлично, но можно заметить, что в HTTP-ответе имеется ряд нежелательных значений, таких как параметры отклика `password` и `accountStatus`. В дополнение к этому желательно, чтобы в `createAt` содержалась читаемая дата. Сделаем это, применяя ассемблер и объект значения.

Во-первых, сформируем объект значения. Создаваемый объект значения будет включать в соответствующей форме данные пользователя, которые необходимо направить клиенту, и ничего более. Создадим пакет helpers.objects с файлом ValueObjects.kt в нем:

```
package com.example.messenger.api.helpers.objects

data class UserVO(
   val id: Long,
   val username: String,
   val phoneNumber: String,
   val status: String,
   val createdAt: String
)
```

Как можно видеть, `UserVO` является классом данных, моделирующим информацию, которую необходимо направить пользователю, и ничем более. Пока это выполняется, добавим объекты-значения для некоторых других ответов, которые рассмотрим позже, чтобы затем не возвращаться к этому файлу:

```
package com.example.messenger.api.helpers.objects

data class UserVO(
   val id: Long,
   val username: String,
   val phoneNumber: String,
   val status: String,
   val createdAt: String
)

data class UserListVO(
   val users: List<UserVO>
)
```

```
data class MessageVO(
   val id: Long,
   val senderId: Long?,
   val recipientId: Long?,
   val conversationId: Long?,
   val body: String?,
   val createdAt: String
)

data class ConversationVO(
   val conversationId: Long,
   val secondPartyUsername: String,
   val messages: ArrayList<MessageVO>
)

data class ConversationListVO(
   val conversations: List<ConversationVO>
)
```

Теперь, при наличии необходимых объектов значений, создадим для `UserVO` ассемблер. *Ассемблер* — это компонент, который собирает (ассемблирует) необходимое значение объекта. Ассемблер, который мы создаем, назовем `UserAssembler`. Поскольку это компонент, он принадлежит пакету components:

```
package com.example.messenger.api.components

import com.example.messenger.api.helpers.objects.UserListVO
import com.example.messenger.api.helpers.objects.UserVO
import com.example.messenger.api.models.User
import org.springframework.stereotype.Component

@Component
class UserAssembler {

   fun toUserVO(user: User): UserVO {
      return UserVO(user.id, user.username, user.phoneNumber,
                    user.status, user.createdAt.toString())
   }

   fun toUserListVO(users: List<User>): UserListVO {
      val userVOList = users.map { toUserVO(it) }
      return UserListVO(userVOList)
   }
}
```

Ассемблер имеет единственную функцию `toUserVO()`, которая воспринимает `User` в качестве аргумента и возвращает соответствующий `UserVO`. Метод `toUserListVO()`

принимает список экземпляров `User` и возвращает соответствующий список `UserListVO`.

Отредактируем конечную точку `create` для применения `UserAssembler` и `UserVO`:

```
package com.example.messenger.api.controllers

import com.example.messenger.api.components.UserAssembler
import com.example.messenger.api.helpers.objects.UserVO
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.UserRepository
import com.example.messenger.api.services.UserServiceImpl
import org.springframework.http.ResponseEntity
import org.springframework.validation.annotation.Validated
import org.springframework.web.bind.annotation.*
import javax.servlet.http.HttpServletRequest

@RestController
@RequestMapping("/users")
class UserController(val userService: UserServiceImpl,
                     val userAssembler: UserAssembler,
                     val userRepository: UserRepository) {
   @PostMapping
   @RequestMapping("/registrations")
   fun create(@Validated @RequestBody userDetails: User):
              ResponseEntity<UserVO> {
      val user = userService.attemptRegistration(userDetails)
      return ResponseEntity.ok(userAssembler.toUserVO(user))
   }
}
```

Запустим заново сервер и перешлем новый запрос на регистрацию `User`. Получим ответ, который от API гораздо более уместен:

```
{
   "id":6,
   "username":"kevin.stacey",
   "phoneNumber":"5472457893",
   "status":"available",
   "createdAt":"Sat Oct 21 11:11:36 WAT 2017"
}
```

Завершим процесс создания конечной точки, обеспечив все необходимые конечные точки для Android-приложения Messenger. Во-первых, добавим конечные точки для отображения сведений об `User`, для списка всех пользователей, для получения сведений о текущем пользователе и обновления статуса от `User` к `UserController`:

```
package com.example.messenger.api.controllers

import com.example.messenger.api.components.UserAssembler
import com.example.messenger.api.helpers.objects.UserListVO
```

```
import com.example.messenger.api.helpers.objects.UserVO
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.UserRepository
import com.example.messenger.api.services.UserServiceImpl
import org.springframework.http.ResponseEntity
import org.springframework.validation.annotation.Validated
import org.springframework.web.bind.annotation.*
import javax.servlet.http.HttpServletRequest

@RestController
@RequestMapping("/users")
class UserController(val userService: UserServiceImpl,
                     val userAssembler: UserAssembler,
                     val userRepository: UserRepository) {

@PostMapping
@RequestMapping("/registrations")
fun create(@Validated @RequestBody userDetails: User):
              ResponseEntity<UserVO> {
   val user = userService.attemptRegistration(userDetails)
   return ResponseEntity.ok(userAssembler.toUserVO(user))
}

@GetMapping
@RequestMapping("/{user_id}")
fun show(@PathVariable("user_id") userId: Long):
ResponseEntity<UserVO> {
   val user = userService.retrieveUserData(userId)
   return ResponseEntity.ok(userAssembler.toUserVO(user))
}

@GetMapping
@RequestMapping("/details")
fun echoDetails(request: HttpServletRequest): ResponseEntity<UserVO>{
   val user = userRepository.findByUsername
              (request.userPrincipal.name) as User
   return ResponseEntity.ok(userAssembler.toUserVO(user))
}

@GetMapping
fun index(request: HttpServletRequest): ResponseEntity<UserListVO> {
   val user = userRepository.findByUsername
              (request.userPrincipal.name) as User
   val users = userService.listUsers(user)

   return ResponseEntity.ok(userAssembler.toUserListVO(users))
}
```

```
@PutMapping
fun update(@RequestBody updateDetails: User,
     request: HttpServletRequest): ResponseEntity<UserVO> {
     val currentUser = userRepository.findByUsername
               (request.userPrincipal.name)
      userService.updateUserStatus(currentUser as User, updateDetails)
      return ResponseEntity.ok(userAssembler.toUserVO(currentUser))
   }
}
```

Создадим контроллеры для обработки ресурсов сообщений и ресурсов бесед — `MessageController` и `ConversationController` соответственно. Но до создания контроллеров напишем ассемблеры, которые используются для сборки объектов значений из сущностей JPA. Далее приводится код ассемблера `MessageAssembler`:

```
package com.example.messenger.api.components

import com.example.messenger.api.helpers.objects.MessageVO
import com.example.messenger.api.models.Message
import org.springframework.stereotype.Component

@Component
   class MessageAssembler {
      fun toMessageVO(message: Message): MessageVO {
         return MessageVO(message.id, message.sender?.id,
                        message.recipient?.id, message.conversation?.id,
                          message.body, message.createdAt.toString())
   }
}
```

А теперь напишем ассемблер `ConversationAssembler`:

```
package com.example.messenger.api.components

import com.example.messenger.api.helpers.objects.ConversationListVO
import com.example.messenger.api.helpers.objects.ConversationVO
import com.example.messenger.api.helpers.objects.MessageVO
import com.example.messenger.api.models.Conversation
import com.example.messenger.api.services.ConversationServiceImpl
import org.springframework.stereotype.Component

@Component
class ConversationAssembler(val conversationService:
                            ConversationServiceImpl,
                            val messageAssembler: MessageAssembler) {
   fun toConversationVO(conversation: Conversation, userId: Long):
ConversationVO {
   val conversationMessages: ArrayList<MessageVO> = ArrayList()
```

```
    conversation.messages.mapTo(conversationMessages) {
        messageAssembler.toMessageVO(it)
}
return ConversationVO(conversation.id, conversationService
                        .nameSecondParty(conversation, userId),
                        conversationMessages)
    }
    fun toConversationListVO(conversations: ArrayList<Conversation>,
        userId: Long): ConversationListVO {
            val conversationVOList = conversations.map {
                oConversationVO(it,
                userId) }
        return ConversationListVO(conversationVOList)
    }
}
```

Теперь все подготовлено для создания непосредственно контроллеров `MessageController` и `ConversationController`. Для столь несложного приложения, как наш Messenger, необходимо лишь действие по созданию сообщения для `MessageController`. Далее приводится код `MessageController` с действием `create` по созданию сообщения:

```
package com.example.messenger.api.controllers

import com.example.messenger.api.components.MessageAssembler
import com.example.messenger.api.helpers.objects.MessageVO
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.UserRepository
import com.example.messenger.api.services.MessageServiceImpl
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.*
import javax.servlet.http.HttpServletRequest

@RestController
@RequestMapping("/messages")
class MessageController(val messageService: MessageServiceImpl,
                        val userRepository: UserRepository,
                        val messageAssembler: MessageAssembler) {
    @PostMapping
    fun create(@RequestBody messageDetails: MessageRequest,
               request: HttpServletRequest): ResponseEntity<MessageVO> {
        val principal = request.userPrincipal
        val sender = userRepository.findByUsername(principal.name) as User
        val message = messageService.sendMessage(sender,
                    messageDetails.recipientId, messageDetails.message)
```

```
        return ResponseEntity.ok(messageAssembler.toMessageVO(message))
    }
    data class MessageRequest(val recipientId: Long, val message: String)
}
```

И наконец, необходимо создать контроллер `ConversationController`. **Нам нужно иметь лишь две конечные точки: одну — для перечисления всех активных бесед пользователя, и другую — для получения сообщений, имеющихся в цепочке бесед. Эти конечные точки обслуживаются действиями** `list()` **и** `show()` **соответственно. Далее приводится код класса** `ConversationController`:

```
package com.example.messenger.api.controllers

import com.example.messenger.api.components.ConversationAssembler
import com.example.messenger.api.helpers.objects.ConversationListVO
import com.example.messenger.api.helpers.objects.ConversationVO
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.UserRepository
import com.example.messenger.api.services.ConversationServiceImpl
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.*
import javax.servlet.http.HttpServletRequest

@RestController
@RequestMapping("/conversations")
class ConversationController(

    val conversationService: ConversationServiceImpl,
    val conversationAssembler: ConversationAssembler,
    val userRepository: UserRepository
) {

    @GetMapping
    fun list(request: HttpServletRequest):
        responseEntity<ConversationListVO>
{
        val user = userRepository.findByUsername(request
                        .userPrincipal.name) as User
        val conversations = conversationService.listUserConversations
                            (user.id)
        return ResponseEntity.ok(conversationAssembler
                        .toConversationListVO(conversations, user.id))
    }

    @GetMapping
    @RequestMapping("/{conversation_id}")
```

```
    fun show(@PathVariable(name = "conversation_id") conversationId: Long,
          request: HttpServletRequest): ResponseEntity<ConversationVO> {
    val user = userRepository.findByUsername(request
    .userPrincipal.name) as User
    val conversationThread = conversationService.retrieveThread
    (conversationId)
    return ResponseEntity.ok(conversationAssembler
    .toConversationVO(conversationThread, user.id))
    }
}
```

Все отлично! Осталась только одна небольшая проблема. У пользователя имеется статус учетной записи, и, возможно, эта учетная запись может быть деактивирована, не так ли? В таком случае мы, как создатели API, не захотим, чтобы деактивированный пользователь использовал нашу платформу. То есть нужно придумать способ предотвращения взаимодействия подобного пользователя с нашим API. Для этого имеется несколько способов, и в нашем случае мы используем перехватчик. Перехватчик перехватывает HTTP-запрос и выполняет над ним одну или несколько операций до того, как продолжит обработку потока (цепочки) запросов. Как и ассемблеры, перехватчик является компонентом. Итак, вызовем перехватчик, который и проверит действительность аккаунта `AccountValidityInterceptor`. Далее приводится класс перехватчика (не забывайте, он входит в пакет components):

```
package com.example.messenger.api.components

import com.example.messenger.api.exceptions.UserDeactivatedException
import com.example.messenger.api.models.User
import com.example.messenger.api.repositories.UserRepository
import org.springframework.stereotype.Component
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter
import java.security.Principal
import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse

@Component
class AccountValidityInterceptor(val userRepository: UserRepository) :
HandlerInterceptorAdapter() {

    @Throws(UserDeactivatedException::class)
    override fun preHandle(request: HttpServletRequest,
    response: HttpServletResponse, handler: Any?): Boolean {

        val principal: Principal? = request.userPrincipal

    if (principal != null) {
        val user = userRepository.findByUsername(principal.name)
                as User
```

```
        if (user.accountStatus == "deactivated") {
            throw UserDeactivatedException("The account of this user has
            been deactivated.")
        }
    }
    return super.preHandle(request, response, handler)
  }
}
```

Класс `AccountValidityInterceptor` переопределяет функцию `preHandle()` своего суперкласса. Эта функция вызывается для выполнения некоторых операций еще до направления запроса к необходимому для него действию контроллера. После создания перехватчика его следует зарегистрировать в приложении Spring. Эту настройку можно выполнить с помощью `WebMvcConfigurer`. Добавьте файл **AppConfig** в пакет **config** проекта. В файл введите следующий код:

```
package com.example.messenger.api.config

import com.example.messenger.api.components.AccountValidityInterceptor
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.InterceptorRegistry
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration
class AppConfig : WebMvcConfigurer {

    @Autowired
    lateinit var accountValidityInterceptor: AccountValidityInterceptor

    override fun addInterceptors(registry: InterceptorRegistry) {
        registry.addInterceptor(accountValidityInterceptor)
        super.addInterceptors(registry)
    }
}
```

Класс `AppConfig` является подклассом класса `WebMvcConfigurer` и переопределяет функцию `addInterceptor(InterceptorRegistry)` в ее суперклассе. Перехватчик `accountValidityInterceptor` добавлен в реестр перехватчиков с помощью функции `registry.addInterceptor()`.

На этом мы завершили работу с кодом, необходимым для поддержки веб-ресурсов Android-приложения Messenger. Теперь нужно развернуть этот код на удаленном сервере.

Развертывание API Messenger на Amazon Web Services

Развертывание приложения Spring Boot на Amazon Web Services (AWS) — это процесс несложный. Процедура развертывания может быть выполнена в течение 10 минут. В этом разделе вы узнаете, как развертывать на AWS приложения, созданные на основе Spring. Перед развертыванием приложения необходимо настроить на AWS базу данных PostgreSQL, к которой и будет подключено приложение.

Установка PostgreSQL на AWS

Первое, что вам необходимо сделать, — это создать учетную запись AWS. Создайте ее, перейдя по следующей ссылке: **https://portal.aws.amazon.com/billing/signup#/start** и зарегистрировавшись. Затем войдите в консоль AWS и перейдите к Amazon Relational Database Service (RDS) — из панели навигации выберите команды **Services | Database | RDS** (Сервисы | База данных | RDS), после чего в панели управления RDS (рис. 4.10) щелкните на кнопке **Get Started Now** (Запустить).

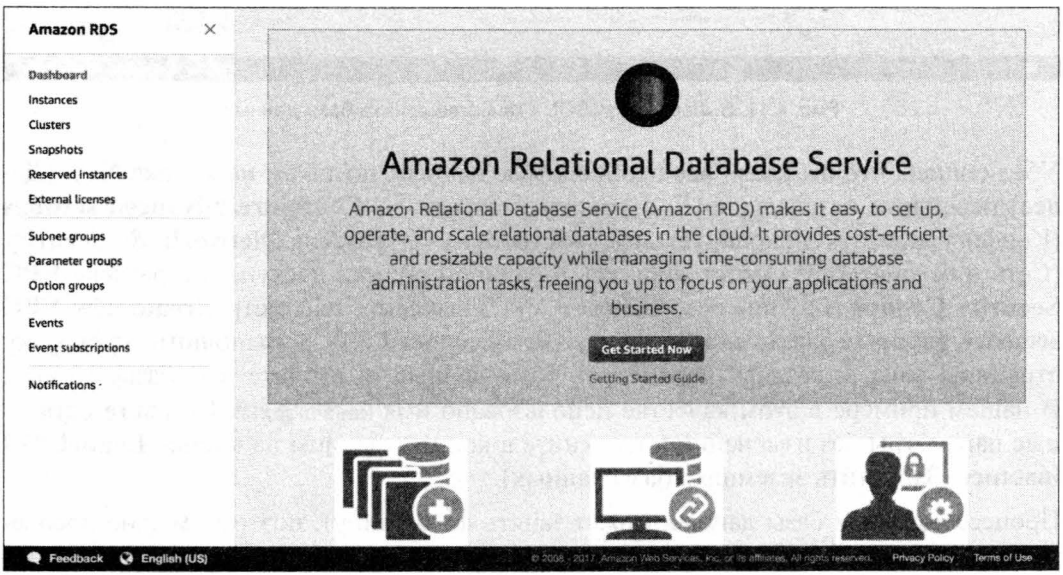


Рис. 4.10. Панель управления RDS

После этого вы перейдете на веб-страницу запуска экземпляра базы данных (рис. 4.11) — выберите здесь PostgreSQL в качестве движка базы данных и убедитесь, что установлен флажок **Only enable options eligible for RDS Free Usage Tier** (Активизировать параметры, допустимые для уровня свободного использования RDS).

Щелчком на кнопке **Next** (Далее) перейдите к следующему набору процедур настройки. Спецификации экземпляров оставьте без изменений и введите необходи-

мые параметры базы данных: имя экземпляра базы данных, главное имя пользователя и мастер-пароль. В качестве имени экземпляра базы данных в нашем примере использовалось имя `messenger-api`, но можно выбрать и другое имя.

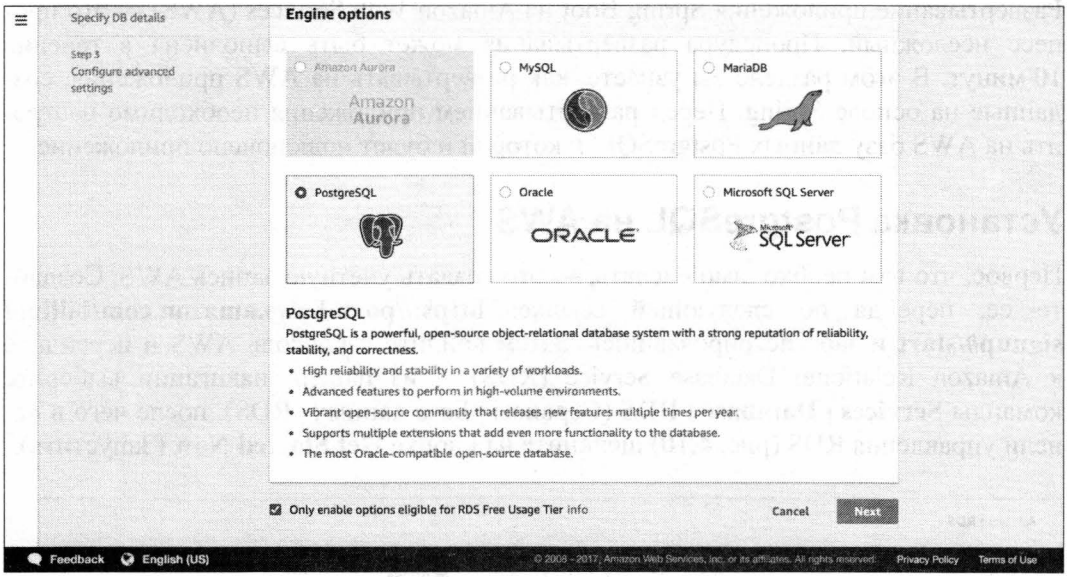


Рис. 4.11. Выбор PostgreSQL в качестве движка базы данных

Убедившись, что вы ввели все необходимые данные, щелчком на кнопке **Next** (Далее) перейдите на следующую страницу настроек — **Configure advanced settings** (Конфигурировать дополнительные настройки). В разделе **Network & Security** (Сеть и безопасность) удостоверьтесь в наличии общего доступа и в разделе **VPC Security Groups** (Группы безопасности VPC) выберите параметр **Create new VPC security group** (Создать новую группу безопасности VPC). Выполните прокрутку страницы вниз, к разделу параметров базы данных, и введите имя базы данных. В нашем примере в этом качестве использовано имя `MessengerDB`. Оставьте остальные параметры без изменений и щелкните в конце страницы на кнопке **Launch DB instance** (Запустить экземпляр базы данных).

Процесс создания базы данных может занять до 10 минут, поэтому можно посоветовать вам сделать перерыв на кофе, в конце которого ваш экземпляр базы данных будет в AWS создан (рис. 4.12).

Щелкните в этом окне на кнопке **View DB instance details** (Просмотреть подробности экземпляра базы данных) — откроется страница, где будет отображена подробная информация о только что развернутом экземпляре базы данных. Выполните на странице прокрутку до раздела **Connect** (Подключить), что позволит просмотреть сведения о подключении экземпляра базы данных (рис. 4.13).

Эти данные необходимы для успешного подключения к `MessengerDB` на этом экземпляре базы данных PostgreSQL. Чтобы подключить `messenger-api` для связи

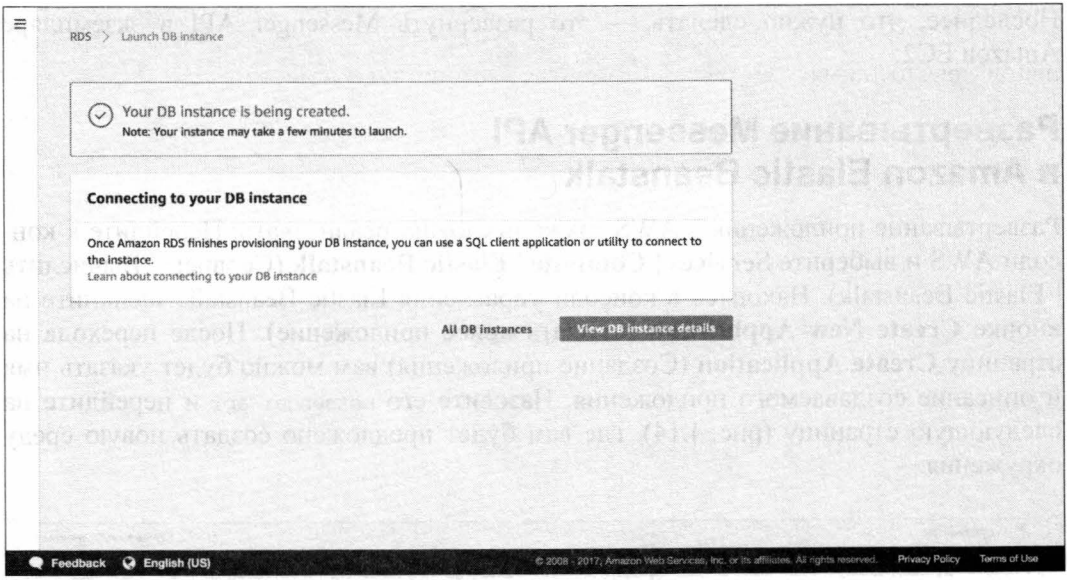


Рис. 4.12. Экземпляр базы данных создан в AWS

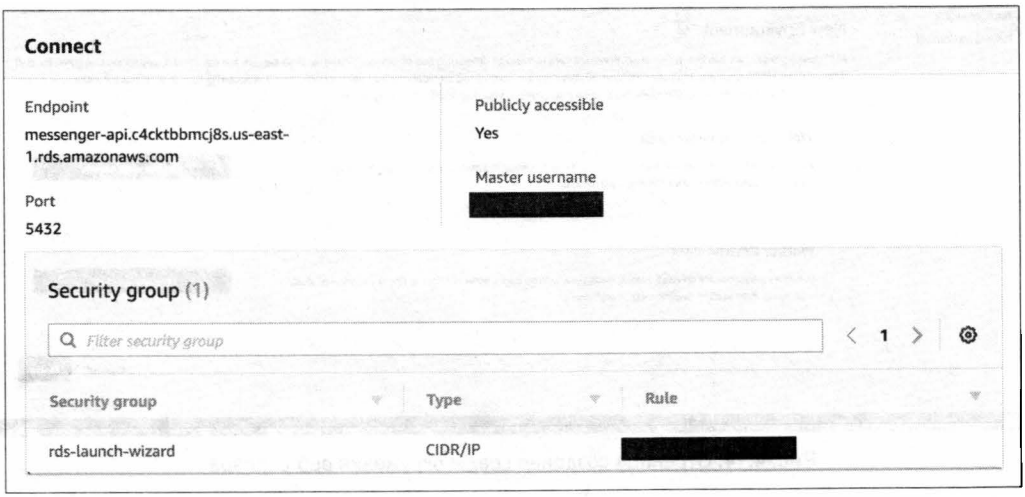


Рис. 4.13. Сведения о подключении экземпляра базы данных

с `MessengerDB`, следует изменить свойства `spring.datasource.url`, `spring.datasource.username` и `spring.datasource.password` в файле application.properties. После выполнения этих действий файл application.properties должен приобрести следующий вид:

```
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.url=jdbc:postgresql://<endpoint>/MessengerDB
spring.datasource.username=<master_username>
spring.datasource.password=<password>
```

Последнее, что нужно сделать, — это развернуть Messenger API в экземпляре Amazon EC2.

Развертывание Messenger API в Amazon Elastic Beanstalk

Развертывание приложения в AWS также несложно реализовать. Перейдите к консоли AWS и выберите **Services | Compute | Elastic Beanstalk** (Сервисы | Вычислить | Elastic Beanstalk). Находясь в консоли управления Elastic Beanstalk, щелкните на кнопке **Create New Application** (Создать новое приложение). После перехода на страницу **Create Application** (Создание приложения) вам можно будет указать имя и описание создаваемого приложения. Назовите его `messenger-api` и перейдите на следующую страницу (рис. 4.14), где вам будет предложено создать новую среду окружения.

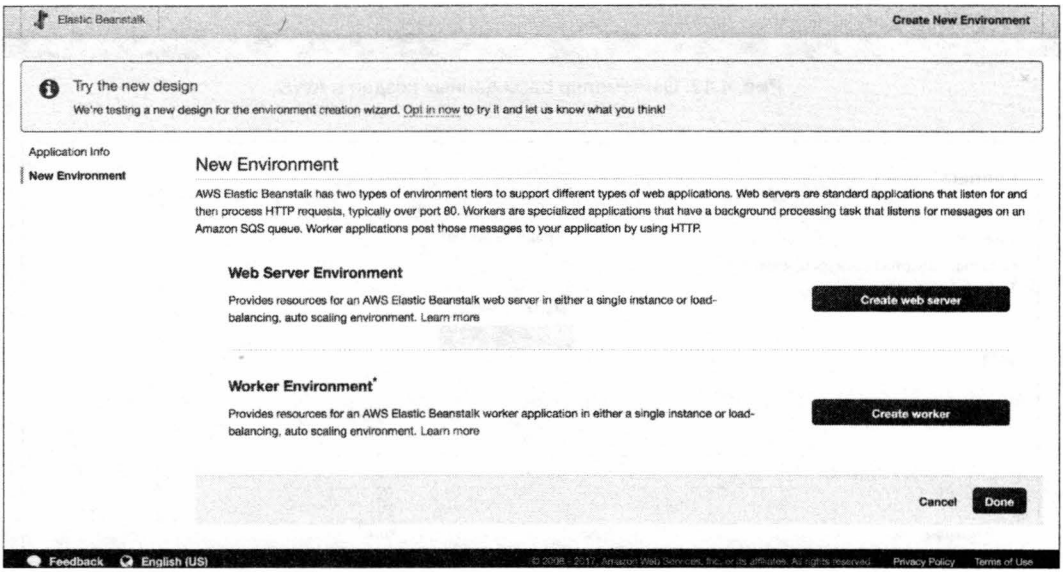


Рис. 4.14. Страница создания среды окружения веб-сервера

Создайте среду окружения веб-сервера (**Web Server Environment**), нажав на кнопки **Create web server** и **Done** (Выполнить). На следующей странице (рис. 4.15) нужно настроить тип среды — выберите предварительно определенную конфигурацию **Tomcat** и измените тип среды на **Single instance** (Единственный экземпляр).

Щелчком на кнопке **Next** (Далее) перейдите на следующую страницу (рис. 4.16). Здесь вам нужно выбрать источник данных для приложения. Укажите **Upload your own** (Загрузить собственные данные).

Теперь надо создать подходящий файл проекта для загрузки. Можно упаковать Messenger API в архив jar с помощью Maven.

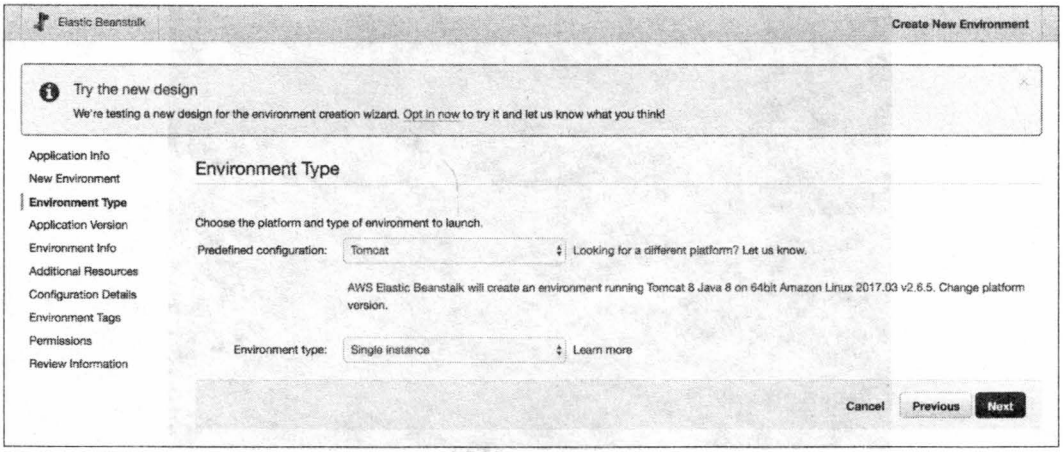


Рис. 4.15. Настройка типа среды окружения

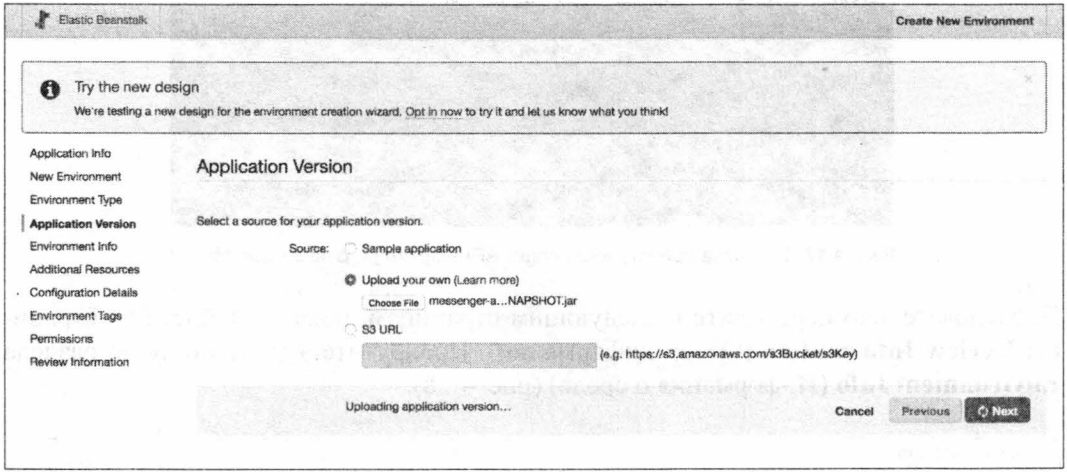


Рис. 4.16. Выбор источника данных для приложения

Откройте в правой части окна IDE вашего проекта (рис. 4.17) вкладку **Maven Projects** (Проекты Maven) и выберите **messenger-api | Lifecycle | package** (messenger-api | Жизненный цикл | пакет) — JAR-архив проекта будет создан и сохранен в его целевом каталоге.

Вернитесь в AWS и выберите этот файл jar в качестве исходного файла для загрузки. Оставьте другие свойства без изменений и щелкните на кнопке **Next** (Далее). Возможно, придется подождать несколько минут, пока файл в формате архива jar загрузится. После завершения загрузки отобразится новая страница с информацией о вашей среде окружения. Пройдите, щелкая на кнопке **Next**, несколько следующих страниц, пока не появится страница **Configuration Details** (Подробности конфигурации). Измените тип экземпляра на `t2.micro`.

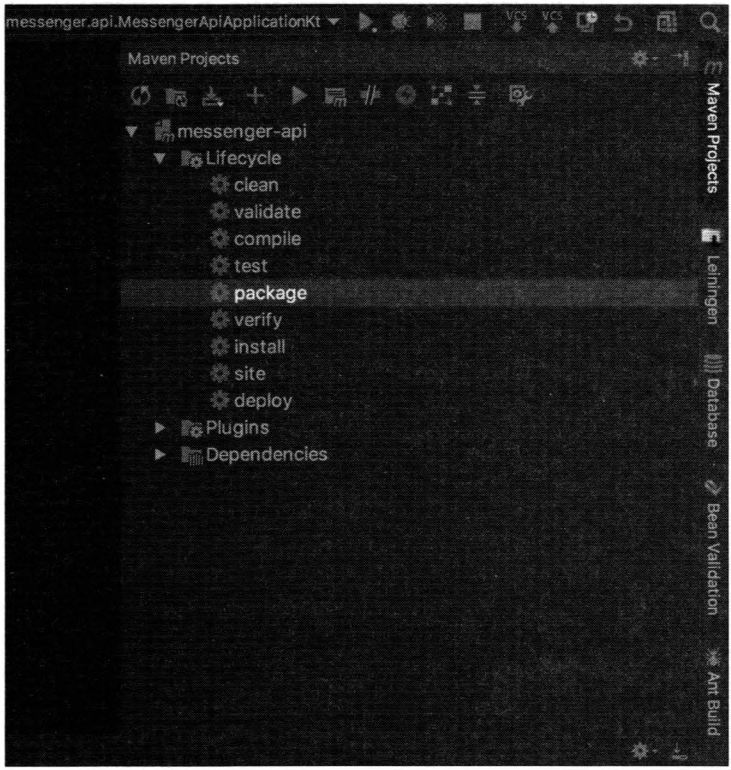


Рис. 4.17. Упаковка проекта Messenger API в архив jar с помощью Maven

Последовательно переходите к следующим страницам, пока не дойдете до страницы **Review Information** (Обзор информации). Прокрутите эту страницу до раздела **Environment Info** (Информация о среде) (рис. 4.18).

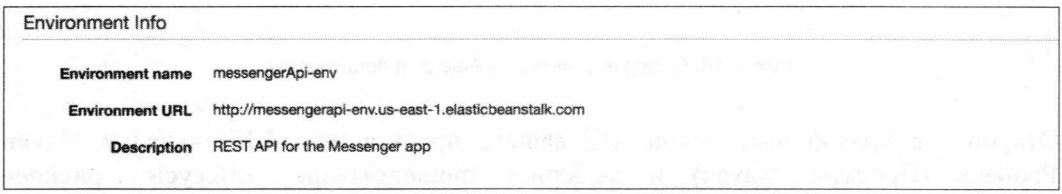


Рис. 4.18. Страница **Review Information**: раздел **Environment Info**

Приведенный в этом разделе интернет-адрес (URL-ссылка) будет отличаться от адреса (URL) вашего проекта. Запомните этот адрес, поскольку он понадобится вам позже. Прокрутите страницу вниз и щелкните на кнопке **Launch** (Запуск). После этого Elastic Beanstalk запустит вашу новую среду.

По завершении запуска настройка вашей среды может считаться завершенной — вы успешно развернули `messenger-api` в AWS.

Подведем итоги

В этой главе рассказано, как применять язык Kotlin для создания интерфейса прикладного программирования Spring Boot REST. В процессе изучения вы освоили основы проектирования систем. Поведение системы API Messenger выражено с помощью диаграммы состояний и показано, каким образом правильно интерпретировать представленную на этой диаграмме информацию. Затем на основе диаграммы отношений сущностей создано подробное схематическое представление объектов системы и их взаимосвязей.

Мы также разобрались, как на локальном компьютере настроить СУБД PostgreSQL и создать новую базу данных PostgreSQL. Показано, как можно создать микросервис с помощью Spring Boot 2.0, выполнить подключение микросервиса к базе данных и с помощью Spring Data поддерживать взаимодействие с данными, находящимися в базе данных.

Мы научились правильно защищать веб-приложение RESTful Spring Boot с помощью Spring Security и JSON Web Tokens (JWT). Для облегчения аутентификации пользователей с помощью JWT созданы пользовательские конфигурации Spring Security, а также пользовательские фильтры, применяемые в процессе аутентификации. И наконец было рассмотрено, каким образом развертывать приложение Spring Boot для AWS.

В следующей главе на примере создания Android-приложения для обмена сообщениями вы углубите знакомство с Android-приложениями Kotlin.

5

Создание Android-приложения Messenger: часть I

В предыдущей главе мы приступили к созданию приложения Messenger — выполнили разработку и внедрение интерфейса программирования приложений REST, с которым взаимодействует клиентское приложение Messenger. В ходе реализации серверного интерфейса API были рассмотрены следующие вопросы: работа с Spring Boot, интерфейсы прикладного программирования RESTful и его работа, создание баз данных с помощью PostgreSQL и развертывание веб-приложений Spring Boot в AWS.

В этой главе мы сделаем следующий шаг на пути создания приложения для обмена интернет-сообщениями, разработав клиентское приложение Messenger (фронтэнд) и интегрировав его с RESTful API, созданным в *главе 4*. В процессе разработки приложения Messenger для Android рассматривается большое число новых тем, среди которых:

- создание Android-приложений на основе шаблона MVP;
- связь с сервером через HTTP;
- работа с Retrofit;
- реактивное программирование;
- использование аутентификации на основе токенов в приложении Android.

Прочитав эту главу, вы почувствуете, насколько универсален язык Kotlin в области разработки приложений для Android. Что ж, приступим.

Разработка Android-приложения Messenger

И начнем мы с создания для приложения нового проекта Android Studi — оргагнизуйте новый проект Android Studio под названием `Messenger` и с именем пакета com.example.messenger. Обратитесь к *главе 1*, чтобы вспомнить этапы создания про-

екта Android. В процессе настройки проекта, когда поступит запрос на создание нового действия запуска, назовите действие `LoginActivity` и сделайте это действие пустым.

Включение зависимостей проекта

В ходе этой главы задействован ряд внешних зависимостей приложений, и важно включить их в проект. Откройте файл build.gradle уровня модуля и добавьте в него следующие зависимости:

```
dependencies {
   implementation fileTree(dir: 'libs', include: ['*.jar'])
   implementation "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
   implementation 'com.android.support:appcompat-v7:26.1.0'
   implementation 'com.android.support.constraint:constraint-layout:1.0.2'
   implementation 'com.android.support:recyclerview-v7:26.1.0'
   implementation 'com.android.support:design:26.1.0'

   implementation "android.arch.persistence.room:runtime:1.0.0-alpha9-1"
   implementation "android.arch.persistence.room:rxjava2:1.0.0-alpha9-1"
   implementation 'com.android.support:support-v4:26.1.0'
   implementation 'com.android.support:support-vector-drawable:26.1.0'
   annotationProcessor "android.arch.persistence.room:compiler:1.0.0-alpha9-1"

   implementation "com.squareup.retrofit2:retrofit:2.3.0"
   implementation "com.squareup.retrofit2:adapter-rxjava2:2.3.0"
   implementation "com.squareup.retrofit2:converter-gson:2.3.0"
   implementation "io.reactivex.rxjava2:rxandroid:2.0.1"

   implementation 'com.github.stfalcon:chatkit:0.2.2'

   testImplementation 'junit:junit:4.12'
   androidTestImplementation 'com.android.support.test:runner:1.0.1'
   androidTestImplementation 'com.android.support.test.espresso
                              :espresso-core:3.0.1'
}
```

Убедитесь, что в файле build.gradle отсутствуют конфликтующие версии библиотек поддержки Android. Теперь измените файл проекта build.gradle для включения в него, наряду с зависимостями инструментов сборки Android, также и репозиториев `jcenter` и `Google`.

Далее приводится код файла сборки верхнего уровня, куда можно добавить параметры конфигурации, общие для всех подпроектов/модулей:

```
buildscript {
   ext.kotlin_version = '1.1.4-3'
```

```
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0-alpha9'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Не беспокойтесь, если пока не знаете, какие зависимости сейчас добавлены, поскольку в ходе чтения этой главы вы получите соответствующие пояснения.

Разработка интерфейса входа в систему (Login UI)

Создав проект, сформируйте новый пакет с именем ui в исходном пакете приложения com.example.messenger. Этот пакет будет содержать все связанные с пользовательским интерфейсом классы и логику приложения Android. Создайте пакет входа в систему login, включив его в ui. Как можно догадаться, этот пакет будет содержать классы и логику, относящиеся именно к процессу входа пользователя в систему. Затем переместите действие `LoginActivity` в пакет login. После перемещения действия `LoginActivity` необходимо сформировать для действия регистрации в системе подходящий макет.

Создайте файл источника макета activity_login.xml и внесите в него следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.login.LoginActivity"
    android:orientation="vertical"
    android:paddingTop="32dp"
    android:paddingBottom="@dimen/default_margin"
```

```
    android:paddingStart="@dimen/default_padding"
    android:paddingEnd="@dimen/default_padding"
    android:gravity="center_horizontal">
    <EditText
        android:id="@+id/et_username"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:hint="@string/username"/>
    <EditText
        android:id="@+id/et_password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/default_margin"
        android:inputType="textPassword"
        android:hint="@string/password"/>
    <Button
        android:id="@+id/btn_login"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/default_margin"
        android:text="@string/login"/>
    <Button
        android:id="@+id/btn_sign_up"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/default_margin"
        android:background="@android:color/transparent"
        android:text="@string/sign_up_solicitation"/>
    <ProgressBar
        android:id="@+id/progress_bar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:visibility="gone"/>
</LinearLayout>
```

Здесь использованы строковые и размерные ресурсы, которые еще не созданы в соответствующих файлах ресурсов XML, так что эти ресурсы следует добавить. Мы также включим ресурсы, которые нам потребуются позднее, на этапе разработки приложения, что исключит необходимость перехода вперед-назад между программными файлами и файлами ресурсов. Откройте файл строковых ресурсов проекта strings.xml и убедитесь, что в него добавлены следующие ресурсы:

```
<resources>
    <string name="app_name">Messenger</string>
    <string name="username">Username</string>
    <string name="password">Password</string>
    <string name="login">Login</string>
```

```
    <string name="sign_up_solicitation">
        Don\'t have an account? Sign up!
    </string>
    <string name="sign_up">Sign up</string>
    <string name="phone_number">Phone number</string>
    <string name="action_settings">settings</string>
    <string name="hint_enter_a_message">Type a message...</string>
        <!-- Настройки учетной записи -->
    <string name="title_activity_settings">Settings</string>
    <string name="pref_header_account">Account</string>
    <string name="action_logout">logout</string>
</resources>
```

Теперь создайте файл ресурсов размерностей dimens.xml и добавьте к нему следующие ресурсы размерностей:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="default_margin">16dp</dimen>
    <dimen name="default_padding">16dp</dimen>
</resources>
```

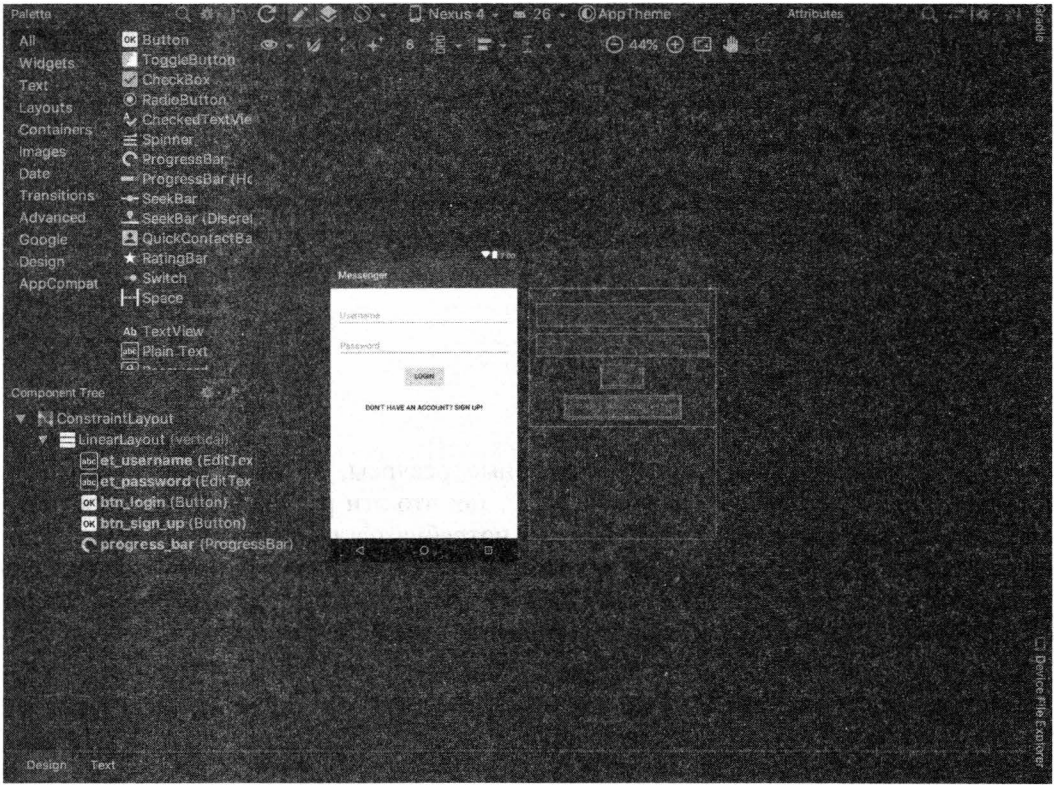


Рис. 5.1. Предварительный просмотр дизайна созданного макета

Добавив необходимые ресурсы проекта, вернитесь к файлу activity_login.xml и переключите редактор в представление **Design** (Дизайн) для просмотра созданного макета (рис. 5.1).

Макет простой, но функциональный и идеально подходит для создаваемого нами приложения Messenger.

Создание представления входа в систему

Приступим к работе с действием `LoginActivity`. Поскольку наше приложение создается с помощью шаблона MVP (шаблон Model-View-Presenter), `LoginActivity` фактически является представлением (View). Очевидно, `LoginActivity` резко отличается от любого общего представления, поскольку описывает процедуру входа в систему. Можно определить набор необходимых поведений, которыми должно обладать представление, определяющее интерфейс входа пользователя в систему:

- отображать для пользователя, когда выполняется его вход в систему, индикатор процесса выполнения;
- иметь возможность сокрытия индикатора выполнения в случае необходимости;
- показывать пользователям их ошибки ввода в поля формы в случае их обнаружения;
- иметь возможность направлять пользователя на его домашнюю страницу;
- иметь возможность перенаправлять незарегистрированного пользователя на страницу регистрации.

Определяя указанные поведения, следует убедиться, что `LoginActivity`, как представление регистрации, всем этим условиям удовлетворяет. Идеальный способ сделать это — использовать интерфейс. Создайте интерфейс `LoginView` в пакете login, содержащий следующий код:

```
package com.example.messenger.ui.login

interface LoginView {
    fun showProgress()
    fun hideProgress()
    fun setUsernameError()
    fun setPasswordError()
    fun navigateToSignUp()
    fun navigateToHome()
}
```

Пока всё с нашим интерфейсом `LoginView` хорошо, но ряд проблем существует. Представление `LoginView` должно располагать возможностью привязки своих представлений компоновки к соответствующим представлениям объектов. Кроме того, представление `LoginView` должно обеспечивать пользователю обратную связь при появлении ошибки аутентификации. Полагаете, что не только эти два поведения

должны обеспечиваться LoginView? Вы правы. Все представления должны иметь возможность привязки элементов макета к программным объектам. Кроме того, само представление регистрации также должно обеспечивать пользователю некоторую обратную связь, если во время аутентификации возникла проблема.

Создадим два отдельных интерфейса для реализации этого поведения. Назовем первый интерфейс BaseView. Создайте в пакете com.example.messenger.ui пакет base и добавьте в него интерфейс под именем BaseView со следующим содержимым:

```
package com.example.messenger.ui.base

import android.content.Context

interface BaseView {
   fun bindViews()
   fun getContext(): Context
}
```

Интерфейс BaseView обеспечивает объявление реализующим классом функций bindViews() и getContext() для привязки вида и поиска контекста соответственно.

Затем создайте в пакете com.example.messenger.ui пакет аутентификации auth и добавьте в него интерфейс под именем AuthView со следующим содержимым:

```
package com.example.messenger.ui.auth

interface AuthView {
   fun showAuthError()
}
```

Отличная работа! Вернитесь к интерфейсу LoginView и удостоверьтесь, что он расширяется интерфейсами BaseView и AuthView следующим образом:

```
package com.example.messenger.ui.login

import com.example.messenger.ui.auth.AuthView
import com.example.messenger.ui.base.BaseView

interface LoginView : BaseView, AuthView {
   fun showProgress()
   fun hideProgress()
   fun setUsernameError()
   fun setPasswordError()
   fun navigateToSignUp()
   fun navigateToHome()
}
```

При объявлении интерфейса LoginView с расширениями BaseView и AuthView мы гарантируем, что каждый класс, который реализует LoginView, должен объявлять функции bindViews(), getContext() и showAuthError() в дополнение к тем, что объяв-

лены в LoginView. Важно отметить: любой класс, реализующий LoginView, эффективнее LoginView, BaseView и AuthView. Характеристика обладающего многими типами класса известна как *полиморфизм*.

Настроив LoginView, можно пойти дальше и поработать с LoginActivity. Во-первых, мы имплементируем в LoginActivity методы, объявленные в BaseView и AuthView, затем добавим методы, специфичные для LoginView. Представление LoginActivity показано в следующем коде:

```
package com.example.messenger.ui.login

import android.content.Context
import android.content.Intent
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.EditText
import android.widget.ProgressBar
import android.widget.Toast
import com.example.messenger.R

class LoginActivity : AppCompatActivity(), LoginView,
   View.OnClickListener
{
   private lateinit var etUsername: EditText
   private lateinit var etPassword: EditText
   private lateinit var btnLogin: Button
   private lateinit var btnSignUp: Button
   private lateinit var progressBar: ProgressBar

override fun onCreate(savedInstanceState: Bundle?) {

   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_login)

   bindViews()
}
```

Привязка ссылки на объект вида макета для просмотра элементов при вызове выглядит следующим образом:

```
override fun bindViews() {
   etUsername = findViewById(R.id.et_username)
   etPassword = findViewById(R.id.et_password)
   btnLogin = findViewById(R.id.btn_login)
   btnSignUp = findViewById(R.id.btn_sign_up)
```

```
    progressBar = findViewById(R.id.progress_bar)
    btnLogin.setOnClickListener(this)
    btnSignUp.setOnClickListener(this)
}

/**
* Отображение соответствующего сообщения об ошибке аутентификации при вызове
*/
override fun showAuthError() {
  Toast.makeText(this, "Invalid username and password combination.",
    Toast.LENGTH_LONG).show()
}
override fun onClick(view: View) {
    }
    override fun getContext(): Context {
        return this
    }
}
```

Итак, пока полет нормальный. Успешно выполнена реализация методов `BaseView` и `AuthView` в `LoginActivity`. Обратимся к методам, специфичным для `LoginView`, а именно: к `showProgress()`, `hideProgress()`, `setUsernameError()`, `setPasswordError()`, `navigateToSignUp()` и `navigateToHome()`. Далее приведена необходимая реализация этих методов. Пойдем дальше и добавим их в действие `LoginActivity`:

```
override fun hideProgress() {
    progressBar.visibility = View.GONE
}

override fun showProgress() {
    progressBar.visibility = View.VISIBLE
}

override fun setUsernameError() {
    etUsername.error = "Username field cannot be empty"
}

override fun setPasswordError() {
    etPassword.error = "Password field cannot be empty"
}

override fun navigateToSignUp() {
}

override fun navigateToHome() {
}
```

Добавив все определенные здесь методы, мы реализовали класс `LoginActivity` для реализации представления `LoginView` наряду с интерфейсом `View.OnClickListener`. Точно так же представление `LoginActivity` поддерживает реализации для функций, объявленных внутри этих интерфейсов. Обратите внимание, каким образом текущий экземпляр `LoginActivity` передается в качестве аргумента методу `btnLogin.setOnClickListener()`. Это можно реализовать, поскольку `LoginActivity` объявлен для реализации интерфейса `View.OnClickListener`. Аналогично `LoginActivity` является действительным экземпляром `View.OnClickListener` (прекрасный пример функционирования полиморфизма).

Теперь, после обработки представления для входа в систему, необходимо создать модель для обработки логики входа, а также необходимые сервисы и репозитории (хранилища) данных, с которыми будет взаимодействовать эта модель. Сначала рассмотрим необходимые сервисы, а затем — прежде чем создавать средства взаимодействия (интеракторы) — разработаем репозитории данных.

Создание сервиса Messenger API и репозиториев данных

Прежде чем углубляться в процесс разработки приложений, следует рассмотреть вопрос о хранении данных. При этом необходимо решить два важных вопроса: где хранятся данные и как получать к ним доступ?

Если речь идет о месте для хранения данных, то данные сохраняются как локально (на устройстве Android), так и удаленно (в Messenger API). Ответ на второй вопрос так же прост. Для получения доступа к сохраняемым данным и для облегчения поиска данных нужно создать подходящие модели, сервисы и репозитории.

Хранение данных локально с помощью хранилища *SharedPreferences*

Позаботимся в первую очередь о локальном хранении данных. Поскольку наше приложение не является сложным, нам не придется сохранять локально большие объемы данных. Единственный род данных, который нужно сохранить на устройстве, — это токены доступа и данные пользователя. Для этого мы задействуем хранилище `SharedPreferences`.

Сначала создайте в исходном пакете приложения пакет данных data. Ранее мы определили, что будем работать с данными, которые хранятся локально и удаленно. Поэтому в пакете data создайте два дополнительных пакета и назовите первый пакет local, а второй — remote. Согласно подходу, который уже применялся для приложения Tetris, рассматриваемого нами в *главах 2* и *3*, для локального сохранения данных мы используем класс `AppPreferences`. Создайте класс `AppPreferences` в локальной среде и заполните его следующим содержимым:

```
package com.example.messenger.data.local

import android.content.Context
import android.content.SharedPreferences
import com.example.messenger.data.vo.UserVO
class AppPreferences private constructor() {
   private lateinit var preferences: SharedPreferences
   companion object {
      private val PREFERENCE_FILE_NAME = "APP_PREFERENCES"
      fun create(context: Context): AppPreferences {
          val appPreferences = AppPreferences()
          appPreferences.preferences = context
      .getSharedPreferences(PREFERENCE_FILE_NAME, 0)
       return appPreferences
   }
}

val accessToken: String?
get() = preferences.getString("ACCESS_TOKEN", null)
fun storeAccessToken(accessToken: String) {
   preferences.edit().putString("ACCESS_TOKEN", accessToken).apply()
}

val userDetails: UserVO
get(): UserVO {
```

Следующий код возвращает экземпляр `UserVO`, содержащий соответствующие данные пользователя:

```
return UserVO(
   preferences.getLong("ID", 0),
   preferences.getString("USERNAME", null),
   preferences.getString("PHONE_NUMBER", null),
   preferences.getString("STATUS", null),
   preferences.getString("CREATED_AT", null)
   )
}
```

Следующий код сохраняет данные пользователя, переданные `UserVO` экземпляру класса `SharedPreferences`:

```
fun storeUserDetails(user: UserVO) {
   val editor: SharedPreferences.Editor = preferences.edit()
   editor.putLong("ID", user.id).apply()
   editor.putString("USERNAME", user.username).apply()
   editor.putString("PHONE_NUMBER", user.phoneNumber).apply()
   editor.putString("STATUS", user.status).apply()
   editor.putString("CREATED_AT", user.createdAt).apply()
}
```

```
fun clear() {
   val editor: SharedPreferences.Editor = preferences.edit()
   editor.clear()
   editor.apply()
   }
}
```

В классе `AppPreferences` определим функции `storeAccessToken(String)`, `storeUserDetails(UserVO)` и `clear()`. Функция `storeAccessToken(String)` служит для сохранения токена доступа, полученного с удаленного сервера, в локальный файл настроек. Функция `storeUserDetails(UserVO)` принимает в качестве единственного аргумента объект пользовательского значения (объект данных, включающих информацию о пользователе) и сохраняет в файле настроек информацию, содержащуюся в объекте значения. Метод `clear()`, как следует из названия, очищает все значения, которые были сохранены в файле настроек. Экземпляр `AppPreferences` также включает свойства `accessToken` и `userDetails`, каждое из которых имеет специальные функции для получения соответствующих значений. В дополнение к функциям и свойствам, определенным в `AppPreferences`, также создан объект-компаньон, обладающий единственной функцией `create(Context)`. Метод `create()`, как следует из названия, создает и возвращает для применения новый экземпляр `AppPreferences`. Мы сформировали основной конструктор `AppPreferences` приватным, поскольку потребовали, чтобы любой класс, использующий `AppPreferences`, применял для создания экземпляров `AppPreferences` метод `create()`.

Создание объектов значений

Подобно тому, как создавалась серверная часть Messenger, для моделирования общих типов данных нужно создать объекты-значения, которые обрабатываются в приложении. Создайте в пакете data пакет vo. Объекты значений, которые мы формируем, вам уже знакомы. Они точно такие же, как те, что созданы при разработке API. Сейчас мы собираемся создавать `ConversationListVO`, `ConversationVO`, `UserListVO`, `UserVO` и `MessageVO`. Создайте файлы Kotlin для хранения каждого из этих объектов-значений в пакете vo. Прежде чем создавать какие-либо модели данных объекта списка значений, следует создать базовые модели. Этими моделями являются `UserVO`, `MessageVO` и `ConversationVO`. Создайте класс данных `UserVO` следующим образом:

```
package com.example.messenger.data.vo

data class UserVO(
   val id: Long,
   val username: String,
   val phoneNumber: String,
   val status: String,
   val createdAt: String
)
```

Поскольку объекты значения создавались ранее, предыдущий код не нуждается в подробном объяснении. Добавьте `MessageVO` к файлу **MessageVO.kt** следующим образом:

```
package com.example.messenger.data.vo

data class MessageVO(
   val id: Long,
   val senderId: Long,
   val recipientId: Long,
   val conversationId: Long,
   val body: String,
   val createdAt: String
)
```

Теперь создайте класс данных `ConversationVo` в файле **ConversationVO.kt** следующим образом:

```
package com.example.messenger.data.vo

data class ConversationVO(
   val conversationId: Long,
   val secondPartyUsername: String,
   val messages: ArrayList<MessageVO>
)
```

После создания основных объектов значений сформируем `ConversationListVO` и `UserListVO`. Класс `ConversationListVO` представлен следующим образом:

```
package com.example.messenger.data.vo

data class ConversationListVO(
   val conversations: List<ConversationVO>
)
```

Класс данных `ConversationListVO` имеет единственное свойство `conversations` типа `List`, которое может включать лишь элементы типа `ConversationVO`. Класс данных `UserListVO` аналогичен `ConversationListVO`, за исключением того, что имеет свойство пользователя, которое может включать только элементы типа `UserVO` вместо свойства `conversations`. Далее приводится класс данных `UserListVO`:

```
package com.example.messenger.data.vo

data class UserListVO(
   val users: List<UserVO>
)
```

Получение удаленных данных

Ранее определялось, что важные данные, необходимые для функционирования Android-приложения Messenger, будут храниться удаленно на сервере Messenger. Крайне важно наличие эффективного средства, с помощью которого Android-приложение может получать доступ к данным, хранящимся в его серверной части. Для этого приложение Messenger должно располагать возможностью взаимодействия с API через HTTP.

Связь с удаленным сервером

В Android предусмотрено несколько способов взаимодействия с удаленным сервером. Общие сетевые библиотеки, используемые в сообществе пользователей Android, — это `Retrofit`, `OkHttp` и `Volley`. Каждая из этих библиотек имеет свои преимущества и недостатки. В этом проекте мы воспользуемся библиотекой `Retrofit`, но ради исследовательского интереса рассмотрим, каким образом можно поддерживать взаимодействие с удаленным сервером с помощью библиотеки `OkHttp`.

Осуществление связи с серверами с помощью *OkHttp*

Клиент `OkHttp` является эффективным и удобным в использовании HTTP клиентом. Он поддерживает как синхронные, так и асинхронные сетевые вызовы. Применение клиента `OkHttp` для Android несложно. Просто добавьте его зависимость в файл уровня модуля проекта `build.gradle`:

```
implementation 'com.squareup.okhttp3:okhttp:3.9.0'
```

Отправка запросов на сервер с помощью *OkHttp*

Как утверждалось, API `OkHttp` создан для удобства пользователей. Как следствие, пересылка запросов посредством `OkHttp` выполняется легко и удобно. Далее приводится метод `post(String, String)`, который принимает URL-ссылку и тело запроса JSON в качестве аргументов и направляет запрос POST по указанной URL-ссылке с телом JSON:

```
fun post(url: String, json: String): String {
   val mediaType: MediaType = MediaType.parse("application/json;
                                              charset=utf-8")
   val client:OkHttpClient = OkHttpClient()
   val body: RequestBody = RequestBody.create(mediaType, json)
   val request: Request = Request.Builder()
                               .url(url)
                               .post(body)
                               .build()
   val response: Response = client.newCall(request).execute()
   return response.body().string()
}
```

Приведенную функцию несложно применить на практике — вызовите ее с соответствующими значениями, как и любую другую функцию:

```
val fullName: String = "John Wayne"
   val response = post("http://example.com", "{ \"full_name\": fullName")
println(response)
```

Как видите, все очень просто, не так ли? Хорошо, если вы согласны. Установление связи с удаленным сервером с помощью `OkHttp` — это несложная процедура, но применение с этой целью библиотеки `Retrofit` еще проще. Но перед тем, как приступить к работе с `Retrofit`, желательно правильно смоделировать данные, которые направляются в HTTP-запросы.

Моделирование запросов данных

Применим классы данных для моделирования данных HTTP-запроса, которые необходимо направить в API. Затем создадим пакет запроса в удаленном пакете. Имеются четыре очевидных запроса, которые содержат полезные данные, направляемые в API. Речь идет о запросах на вход в систему, сообщениях, запросах на обновление статуса и запросах, содержащих данные пользователя. Эти четыре запроса моделируются с помощью объектов `LoginRequestObject`, `MessageRequestObject`, `StatusUpdateRequestObject` и `UserRequest` соответственно.

Следующий фрагмент кода показывает класс данных `LoginRequestObject`. Добавьте его в пакет запроса и сделайте то же самое для других упомянутых объектов запроса:

```
package com.example.messenger.data.remote.request

data class LoginRequestObject(
   val username: String,
   val password: String
)
```

Класс данных `LoginRequestObject` располагает свойствами `username` и `password`, поскольку это учетные данные, которые необходимо предоставить конечной точке входа в API. Класс данных `MessageRequestObject` представлен следующим образом:

```
package com.example.messenger.data.remote.request

data class MessageRequestObject(val recipientId: Long, val message: String)
```

Класс `MessageRequestObject` также располагает двумя свойствами. Этими свойствами являются `recipientId` — ID пользователя-получателя сообщения и `message` — тело отправляемого сообщения:

```
package com.example.messenger.data.remote.request

data class StatusUpdateRequestObject(val status: String)
```

Класс данных `StatusUpdateRequestObject` имеет единственное свойство состояния. Как следует из названия, это состояние, которое пользователь хочет обновить в своем текущем сообщении:

```
package com.example.messenger.data.remote.request

data class UserRequestObject(
   val username: String,
   val password: String,
   val phoneNumber: String = ""
)
```

Объект `UserRequestObject` аналогичен объекту `LoginRequestObject`, за исключением того, что он содержит дополнительное свойство `phoneNumber`. Этот объект запроса имеет различные варианты использования — например, содержит данные о регистрации пользователя, направляемые в API.

После создания необходимых объектов запроса мы можем приступить к созданию реального сервиса Messenger API — `MessengerApiService`.

Создание сервиса Messenger API

Создадим сервис, выполняющий важную работу по связи с API Messenger, который мы разработали в *главе 4*. Для создания этого сервиса применим библиотеку `Retrofit` и RxJava-адаптер Retrofit. `Retrofit` — это безопасный для типов HTTP-клиент для Android и Java, созданный Square Inc., а RxJava — это реализация ReactiveX с открытым исходным кодом, написанная на Java и для Java (так называемое *реактивное программирование*).

Мы добавили `Retrofit` в Android-проект в начале этой главы (см. *разд. «Включение зависимостей проекта»*) с помощью следующей строки:

```
implementation "com.squareup.retrofit2:retrofit:2.3.0"
```

А также добавили зависимость адаптера RxJava для Retrofit к сценарию уровня модуля build.gradle следующим образом:

```
implementation "com.squareup.retrofit2:adapter-rxjava2:2.3.0"
```

Первым шагом в создании сервиса с помощью `Retrofit` является определение интерфейса, описывающего HTTP API. Создайте в пакете source приложения пакет service и добавьте в него интерфейс `MessengerApiService` следующим образом:

```
package com.example.messenger.service

import com.example.messenger.data.remote.request.LoginRequestObject
import com.example.messenger.data.remote.request.MessageRequestObject
import com.example.messenger.data.remote.request.StatusUpdateRequestObject
import com.example.messenger.data.remote.request.UserRequestObject
import com.example.messenger.data.vo.*
import io.reactivex.Observable
import okhttp3.ResponseBody
```

```
import retrofit2.Retrofit
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory
import retrofit2.converter.gson.GsonConverterFactory
import retrofit2.http.*

interface MessengerApiService {
   @POST("login")
   @Headers("Content-Type: application/json")
   fun login(@Body user: LoginRequestObject):
            Observable<retrofit2.Response<ResponseBody>>

   @POST("users/registrations")
   fun createUser(@Body user: UserRequestObject): Observable<UserVO>

   @GET("users")
   fun listUsers(@Header("Authorization") authorization: String):
                Observable<UserListVO>

   @PUT("users")
   fun updateUserStatus(
      @Body request: StatusUpdateRequestObject,
      @Header("Authorization") authorization: String): Observable<UserVO>

   @GET("users/{userId}")
   fun showUser(
      @Path("userId") userId: Long,
      @Header("Authorization") authorization: String): Observable<UserVO>

   @GET("users/details")
   fun echoDetails(@Header("Authorization") authorization: String):
                  Observable<UserVO>

   @POST("messages")
   fun createMessage(
      @Body messageRequestObject: MessageRequestObject,
      @Header("Authorization") authorization: String): Observable<MessageVO>

   @GET("conversations")
   fun listConversations(@Header("Authorization") authorization: String):
                        Observable<ConversationListVO>

   @GET("conversations/{conversationId}")
   fun showConversation(
      @Path("conversationId") conversationId: Long,
      @Header("Authorization") authorization: String): Observable<ConversationVO>
}
```

Как видно из приведенного фрагмента кода, для правильного описания отправленных HTTP-запросов Retrofit полагается на использование аннотаций. Посмотрите на следующий фрагмент кода, например:

```
@POST("login")
@Headers("Content-Type: application/json")
fun login(@Body user: LoginRequestObject):
Observable<retrofit2.Response<ResponseBody>>
```

Аннотация @POST указывает Retrofit, что эта функция описывает запрос HTTP POST, который сопоставлен с путем /login. Аннотация @Headers применяется для указания заголовков HTTP-запроса. В HTTP-запросе из приведенного фрагмента кода заголовок Content-Type устанавливается как application/json. Поэтому содержимым, отправляемым по данному запросу, является JSON.

Аннотация @Body определяет, что аргумент пользователя, переданный login(), содержит данные тела запроса JSON для отправки в API. Параметр user относится к типу LoginRequestObject (ранее созданный объект запроса). И наконец, объявляется, что функция возвращает объект Observable, содержащий объект retrofit2.Response.

Кроме аннотаций @POST, @Headers и @Body, используются аннотации @GET, @PUT, @Path и @Header. Аннотации @GET и @PUT служат для указания запросов GET и PUT соответственно. Аннотация @Path используется для объявления значения как аргумента пути для пересылаемого HTTP-запроса. Посмотрите, например, на функцию showUser():

```
@GET("users/{userId}")
fun showUser(
   @Path("userId") userId: Long,
   @Header("Authorization") authorization: String): Observable<UserVO>
```

Функция showUser является функцией, описывающей запрос GET с помощью пути users/{userId}. Но фрагмент {userId} не является частью пути HTTP-запроса. Retrofit заменит {userId} значением, которое передается аргументу userId для функции showUser(). Обратите внимание, каким образом userId аннотирован с помощью @Path("userId"). При этом retrofit известно, что userId содержит значение, которое следует разместить там, где {userId} размещается в URL-пути для HTTP-запроса. @Header аналогичен @Headers, за исключением того, что применяется для указания одной пары ключ-значение для заголовка в отправляемом HTTP-запросе. Аннотирующая авторизация с помощью @Header("Authorization") устанавливает заголовок Authorization для HTTP-запроса, который направляется к значению, хранящемуся в рамках авторизации.

После создания соответствующего интерфейса MessengerApiService для моделирования HTTP API, с которым связывается наше приложение, необходимо располагать возможностью получения экземпляра данного сервиса. Для этого следует создать объект-компаньон Factory, который отвечает за создание экземпляров MessengerApiService:

```
package com.example.messenger.service

import com.example.messenger.data.remote.request.LoginRequestObject
import com.example.messenger.data.remote.request.MessageRequestObject
import com.example.messenger.data.remote.request.StatusUpdateRequestObject
import com.example.messenger.data.remote.request.UserRequestObject
import com.example.messenger.data.vo.*
import io.reactivex.Observable
import okhttp3.ResponseBody
import retrofit2.Retrofit
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory
import retrofit2.converter.gson.GsonConverterFactory
import retrofit2.http.*

interface MessengerApiService {

   ...
   companion object Factory {
      private var service: MessengerApiService? = null
```

При вызове возвращается экземпляр объекта `MessengerApiService`. **Создается новый экземпляр объекта** `MessengerApiService`, **если не был удачно создан** `getInstance()`:

```
MessengerApiService {
  if (service == null) {
      val retrofit = Retrofit.Builder()
      .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
      .addConverterFactory(GsonConverterFactory.create())
      .baseUrl("{AWS_URL}")
      // Заменяет AWS_URL на URL-ссылку для AWS EC2
      // Экземпляр был развернут в предыдущей главе
      .build()
      service = retrofit.create(MessengerApiService::class.java)
   }
   return service as MessengerApiService
   }
   }
}
```

`Factory` **обладает единственной функцией** `getInstance()`, **которая при вызове** создает **и возвращает экземпляр** `MessengerApiService`. **Экземпляр** `Retrofit.Builder` **применяется для создания интерфейса. Установим** `CallAdapterFactory` **при** использовании **в** `RxJava2CallAdapterFactory`, **а также** `ConverterFactory` **при использовании** в `GsonConverterFactory` **(обрабатывает сериализацию и десериализацию** JSON). **Не забывайте заменять** `"{AWS_URL}"` **на** URL **для экземпляра** Messenger API AWS EC2, **развернутого в** *главе 4*.

Успешно созданный экземпляр `Retrofit.Builder()` применяется для создания экземпляра `MessengerApiService`:

```
service = retrofit.create(MessengerApiService::class.java)
```

И наконец, сервис возвращается для использования при помощи `getInstance()`.

Тем не менее, хотя создан сервис, пригодный для связи с API Messenger, его нельзя использовать для связи с сетью без указания необходимых разрешений в файле AndroidManifest. Поэтому откройте файл AndroidManifest проекта и добавьте следующие две строки кода в тег `<manifest></manifest>`:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Теперь, при наличии подготовленного сервиса Messenger, создадим соответствующие репозитории для работы этого сервиса.

Реализация репозиториев данных

Вы уже знакомы с репозиториями, поэтому сразу перейдем к сути задания. Создаваемые репозитории аналогичны тем, которые формировались для API Messenger в *главе 4*. Единственное отличие в том, что источником данных для создаваемых репозиториев является удаленный сервер, а не расположенная на хосте база данных.

Создайте внутри пакета remote пакет repository. Прежде всего репозиторий пользователей мы реализуем для извлечения данных, относящихся к пользователям приложений. Добавим к репозиторию интерфейс `UserRepository` следующим образом:

```
package com.example.messenger.data.remote.repository

import com.example.messenger.data.vo.UserListVO
import com.example.messenger.data.vo.UserVO
import io.reactivex.Observable
interface UserRepository {
   fun findById(id: Long): Observable<UserVO>
   fun all(): Observable<UserListVO>
   fun echoDetails(): Observable<UserVO>
}
```

Поскольку это интерфейс, нужно создать класс, реализующий функции, указанные в `UserRepository`. Назовем этот класс `UserRepositoryImpl`. Создадим внутри пакета repository новый класс `UserRepositoryImpl` следующим образом:

```
package com.example.messenger.data.remote.repository

import android.content.Context
import com.example.messenger.service.MessengerApiService
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.data.vo.UserListVO
```

```
import com.example.messenger.data.vo.UserVO
import io.reactivex.Observable

class UserRepositoryImpl(ctx: Context) : UserRepository {

   private val preferences: AppPreferences = AppPreferences.create(ctx)
   private val service: MessengerApiService = MessengerApiService.getInstance()

   override fun findById(id: Long): Observable<UserVO> {
      return service.showUser(id, preferences.accessToken as String)
   }

override fun all(): Observable<UserListVO> {
   return service.listUsers(preferences.accessToken as String)
}

override fun echoDetails(): Observable<UserVO> {
   return service.echoDetails(preferences.accessToken as String)
   }
}
```

Показанный здесь класс `UserRepositoryImpl` имеет две переменные экземпляра: `preferences` и `service`. Переменная `preferences` является экземпляром созданного ранее класса `AppPreferences`, а переменная `service` служит экземпляром `MessengerApiService`, который извлекается функцией `getInstance()`, определенной в объекте-компаньоне `Factory` в интерфейсе `MessengerApiService`.

Объект `UserRepositoryImpl` поддерживает реализации функций `findById()`, `all()` и `echoDetails()`, определенных в `UserRepository`. Эти три реализованные функции используют сервис для извлечения находящихся на сервере необходимых данных с помощью HTTP-запросов. Функция `findById()` вызывает в сервисе функцию `showUser()` для пересылки запроса к Messenger API с целью извлечения подробностей, относящихся к имени пользователя, указанного с помощью ID.

Функция `showUser()` требует в качестве второго аргумента токен авторизации вошедшего в систему текущего пользователя. Затребованный токен предоставляется посредством экземпляра `AppPreferences`, передаваемого в функцию `preferences.accessToken` в качестве второго аргумента.

Функция `all()` использует функцию `MessengerApiService#listUsers()` для получения всех пользователей, зарегистрированных в службе сообщений. Функция `echoDetails()` задействует функцию `MessengerApiService#echoDetails()` для получения подробностей, связанных с именем пользователя, находящегося в текущий момент в системе.

Создадим репозиторий для бесед, чтобы облегчить доступ к данным, относящимся к беседам. Добавьте интерфейс `ConversationRepository` к `com.example.messenger.data.remote.repository` со следующим содержимым:

```
package com.example.messenger.data.remote.repository

import com.example.messenger.data.vo.ConversationListVO
import com.example.messenger.data.vo.ConversationVO
import io.reactivex.Observable
interface ConversationRepository {
   fun findConversationById(id: Long): Observable<ConversationVO>
   fun all(): Observable<ConversationListVO>
}
```

Теперь в пакете создайте соответствующий класс `ConversationRepositoryImpl` следующим образом:

```
package com.example.messenger.data.remote.repository

import android.content.Context
import com.example.messenger.service.MessengerApiService
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.data.vo.ConversationListVO
import com.example.messenger.data.vo.ConversationVO
import io.reactivex.Observable

class ConversationRepositoryImpl(ctx: Context) : ConversationRepository {

   private val preferences: AppPreferences = AppPreferences.create(ctx)
   private val service: MessengerApiService = MessengerApiService.getInstance()
```

Этот класс производит извлечение из Messenger API информации, относящейся к беседе с запрошенным пользователем, имеющим идентификатор беседы:

```
override fun findConversationById(id: Long): Observable<ConversationVO> {
   return service.showConversation(id, preferences.accessToken as String)
}
```

И при вызове извлекает из API все активные беседы текущего пользователя:

```
   override fun all(): Observable<ConversationListVO> {
      return service.listConversations(preferences.accessToken as String)
   }
}
```

Функция `findConversationById(Long)` извлекает цепочку бесед с пользователем, имеющим соответствующий ID, который передан функции. Функция `all()` просто извлекает все активные данные текущего пользователя.

Создание интеракторов входа

Пришло время сформировать интерактор входа в систему, который будет служить моделью для взаимодействия при входе в нее. Создайте в пакете login интерфейс `LoginInteractor`, содержащий следующий код:

```
package com.example.messenger.ui.login

import com.example.messenger.data.local.AppPreferences
import com.example.messenger.ui.auth.AuthInteractor

interface LoginInteractor : AuthInteractor {
interface OnDetailsRetrievalFinishedListener {
   fun onDetailsRetrievalSuccess()
   fun onDetailsRetrievalError()
}

   fun login(username: String, password: String,
   listener: AuthInteractor.onAuthFinishedListener)
   fun retrieveDetails(preferences: AppPreferences,
   listener: OnDetailsRetrievalFinishedListener)
}
```

Можно заметить, что `LoginInteractor` расширяет `AuthInteractor`. Этот подход аналогичен тому, как `LoginView` расширяет `AuthView`. Интерфейс `AuthInteractor` объявляет поведения и характеристики, которые должны реализовываться любым интерактором (при любом входе), когда выполняется обработка логики, связанной с аутентификацией. Реализуем интерфейс `AuthInteractor`. Затем добавим интерфейс `AuthInteractor` в пакет com.exampla.messenger.auth:

```
package com.example.messenger.ui.auth

import com.example.messenger.data.local.AppPreferences
import com.example.messenger.data.remote.vo.UserVO
interface AuthInteractor {

   var userDetails: UserVO
   var accessToken: String
   var submittedUsername: String
   var submittedPassword: String
   interface onAuthFinishedListener {

      fun onAuthSuccess()
      fun onAuthError()
      fun onUsernameError()
      fun onPasswordError()
   }
   fun persistAccessToken(preferences: AppPreferences)
   fun persistUserDetails(preferences: AppPreferences)
}
```

Каждый интерактор, реализующий интерфейс `AuthInteractor`, должен иметь несколько полей: `userDetails`, `accessToken`, `submittedUsername` и `submittedPassword`. Кроме

того, интерактор, реализующий AuthInteractor, должен располагать методами persistAccessToken(AppPreferences) и persistUserDetails(AppPreferences). Как следует из названий методов, они сохраняют токены доступа и сведения о пользователе в экземпляре класса SharedPreferences приложения. Как вы уже догадались, нужно создать класс реализации для LoginInteractor. Назовем этот класс LoginInteractorImpl.

Далее приводится класс LoginInteractorImpl с реализованным методом login(). Добавьте его к пакету login в пакете ui:

```
package com.example.messenger.ui.login

import com.example.messenger.data.local.AppPreferences
import com.example.messenger.data.remote.request.LoginRequestObject
import com.example.messenger.data.vo.UserVO
import com.example.messenger.service.MessengerApiService
import com.example.messenger.ui.auth.AuthInteractor
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers
class LoginInteractorImpl : LoginInteractor {

override lateinit var userDetails: UserVO

   override lateinit var accessToken: String
   override lateinit var submittedUsername: String
   override lateinit var submittedPassword: String

   private val service: MessengerApiService = MessengerApiService.getInstance()

   override fun login(username: String, password: String,
   listener: AuthInteractor.onAuthFinishedListener) {
      when {
```

Когда в форме входа оказывается пустой параметр username, такой username признается недействительным. Если это происходит, вызывается функция слушателя onUsernameError():

```
username.isBlank() -> listener.onUsernameError()
```

Вызовем функцию слушателя onPasswordError(), если предоставлен пустой пароль:

```
   password.isBlank() -> listener.onPasswordError()
else -> {
```

Выполним инициализацию полей модели submittedUsername и submittedPassword и создадим соответствующий объект LoginRequestObject:

```
submittedUsername = username
submittedPassword = password
val requestObject = LoginRequestObject(username, password)
```

Применим `MessengerApiService` для направления в Messenger API запроса на вход.

```
service.login(requestObject)
        .subscribeOn(Schedulers.io())
           // Подписка Observable на поток Scheduler
        .observeOn(AndroidSchedulers.mainThread())
           // настройка наблюдения должна быть сделана в главном потоке
        .subscribe({ res ->
        if (res.code() != 403) {
            accessToken = res.headers()["Authorization"] as String
        listener.onAuthSuccess()
           } else {
```

Достигнута ветвь, когда сервер возвращает код состояния HTTP **403** (запрещено). Это означает, что вход в систему не выполнен, и пользователь не авторизован для доступа к серверу.

```
                    listener.onAuthError()
                }
            }, { error ->
                listener.onAuthError()
                error.printStackTrace()
            })
        }
    }
  }
}
```

Функция `login()` сначала проверяет, что предоставленные аргументы имени пользователя и пароля не пусты. Функция `onUsernameError()` из `onAuthFinishedListener` вызывается, если встречается пустое имя пользователя, а функция `onPasswordError()` — если встречается пустой пароль. Если ни имя, ни пароль не указаны, то для направления запроса на вход в мессенджер (API Messenger) используется объект `MessengerApiService`. Если запрос на вход выполнен успешно, устанавливается свойство `accessToken` для токена доступа, полученного из заголовка `Authorization` для ответа API, а затем вызывается функция слушателя `onAuthSuccess()`. В случае сбоя запроса на вход вызывается функция слушателя `onAuthError()`.

Представляя процесс входа в систему, добавьте к `LoginInteractorImpl` методы `retrieveDetails()`, `persistAccessToken()` и `persistUserDetails()`:

```
  override fun retrieveDetails(preferences: AppPreferences,
          listener: LoginInteractor.OnDetailsRetrievalFinishedListener)
{
```

Получение информации о пользователе при первом входе:

```
service.echoDetails(preferences.accessToken as String)
        .subscribeOn(Schedulers.io())
```

```
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({ res ->
            userDetails = res
            listener.onDetailsRetrievalSuccess()},
        { error ->
            listener.onDetailsRetrievalError()
            error.printStackTrace()})
}

override fun persistAccessToken(preferences: AppPreferences) {
    preferences.storeAccessToken(accessToken)
}

override fun persistUserDetails(preferences: AppPreferences) {
    preferences.storeUserDetails(userDetails)
}
```

Внимательно прочитайте комментарии, сопровождающие приведенные фрагменты кода в файлах примеров исходного кода, размещенных в сопровождающем книгу файловом архиве. Благодаря им вы сможете глубже познакомиться с работой интерактора входа `LoginInteractor`. А мы рассмотрим теперь работу с презентатором входа `LoginPresenter`.

Создание презентатора входа

Презентатор, как было продемонстрировано в *главе 3*, является посредником между представлением и моделью. Поэтому нам необходимо иметь подходящие презентаторы для представлений, что облегчит непосредственное взаимодействие между моделями и представлениями. Сделать презентатор несложно. Сначала следует организовать интерфейс, объявляющий поведения, которые будут демонстрироваться презентатором. Создайте в пакете `login` интерфейс `LoginPresenter` со следующим кодом:

```
package com.example.messenger.ui.login

interface LoginPresenter {
    fun executeLogin(username: String, password: String)
}
```

Как можно видеть из приведенного фрагмента кода, необходим класс, действующий как `LoginPresenter` для `LoginView` при использовании функции `executeLogin(String, String)`. Эта функция вызывается представлением, а затем взаимодействует с моделью, обрабатывающей логику входа в приложение. Создадим класс `LoginPresenterImpl`, реализующий презентатор `LoginPresenter`:

```
package com.example.messenger.ui.login

import com.example.messenger.data.local.AppPreferences
import com.example.messenger.ui.auth.AuthInteractor
```

```
class LoginPresenterImpl(private val view: LoginView) :
LoginPresenter, AuthInteractor.onAuthFinishedListener,
LoginInteractor.OnDetailsRetrievalFinishedListener {

   private val interactor: LoginInteractor = LoginInteractorImpl()
   private val preferences: AppPreferences =
                                          AppPreferences.create(view.getContext())

      override fun onPasswordError() {
         view.hideProgress()
         view.setPasswordError()

      }
      override fun onUsernameError() {
         view.hideProgress()
         view.setUsernameError()
      }

      override fun onAuthSuccess() {
         interactor.persistAccessToken(preferences)
         interactor.retrieveDetails(preferences, this)
      }

      override fun onAuthError() {
         view.showAuthError()
         view.hideProgress()
      }

      override fun onDetailsRetrievalSuccess() {
         interactor.persistUserDetails(preferences)
         view.hideProgress()
         view.navigateToHome()
      }

      override fun onDetailsRetrievalError() {
         interactor.retrieveDetails(preferences, this)
      }

      override fun executeLogin(username: String, password: String) {
         view.showProgress()
         interactor.login(username, password, this)
   }
}
```

Класс `LoginPresenterImpl` реализует презентатор `LoginPresenter`, интеракторы `AuthInteractor.onAuthFinishedListener` и `LoginInteractor.OnDetailsRetrievalFinished-`

Listener и все запрашиваемые интерфейсами поведения. LoginPresenterImpl также переопределяет семь функций: onPasswordError(), onUsernameError(), onAuthSuccess(), onAuthError(), onDetailsRetrievalSuccess(), onDetailsRetrievalError() и executeLogin(String, String). Взаимодействие между презентатором LoginPresenter и интерактором LoginInteractor можно увидеть в функциях onAuthSuccess() и executeLogin(String, String): когда пользователь направляет свои данные для входа, представление LoginView вызывает функцию executeLogin(String, String) в LoginPresenter. В свою очередь, LoginPresenter использует LoginInteractor для обработки действительной процедуры входа путем вызова функции login(String, String) из LoginInteractor.

Если вход пользователя выполнен успешно, вызывается функция обратной связи onAuthSuccess() из LoginPresenter для интерактора LoginInteractor. Это приводит к сохранению возвращенного сервером токена доступа и получению данных учетной записи вошедшего в систему пользователя. Если запрос на вход в систему сервером отклонен, вызывается функция onAuthError(), а пользователь получает информационное сообщение об ошибке.

Если данные учетной записи пользователя успешно извлечены интерактором, из LoginPresenter вызывается функция обратной связи onDetailsRetrievalSuccess(). Это приводит к сохранению данных учетной записи. Отображаемый пользователю во время входа в систему индикатор выполнения скрывается функцией view.hideProgress(), после чего пользователь переходит на свою домашнюю страницу с помощью функции view.navigateToHome(). Если получение пользовательских данных не выполнено, с помощью интерактора LoginInteractor вызывается функция onDetailsRetrievalError(). Затем презентатор запрашивает о другой попытке на получение данных учетной записи пользователя, еще раз вызывая interactor.retrieveDetails(preferences, this).

Завершение работы с *LoginView*

Как вы, возможно, помните, нами еще не завершена осуществленная ранее реализация LoginView. Функции типа navigateToSignUp(), navigateToHome() и onClick(view: View) еще имеют пустые тела. Более того, для LoginView отсутствует взаимодействие с LoginPresenter. Исправим сейчас эту недоработку.

Прежде всего, чтобы перейти от пользователя к экрану регистрации и домашней странице, необходимо располагать для них представлениями. Мы здесь не станем заниматься реализацией для них макетов (этот вопрос рассматривается в следующих разделах). Сейчас же необходимо, чтобы они просто существовали. Создайте под com.example.messenger.ui пакеты signup и main. Сформируйте новое пустое действие под названием SignUpActivity в пакете signup и новое пустое действие под названием MainActivity в пакете main.

Откройте файл LoginActivity.kt. Необходимо изменить в нем упомянутые здесь функции для выполнения соответствующих им задач. Но сначала нужно добавить част-

ные свойства для экземпляра LoginPresenter и экземпляра AppPreferences — добавьте их в верхней части описания класса LoginActivity:

```
private lateinit var progressBar: ProgressBar
private lateinit var presenter: LoginPresenter
private lateinit var preferences: AppPreferences
```

А вот теперь модифицируйте функции navigateToSignUp(), navigateToHome() и onClick(view: View), как показано в следующем фрагменте кода:

```
override fun navigateToSignUp() {
   startActivity(Intent(this, SignUpActivity::class.java))
}

override fun navigateToHome() {
   finish()
   startActivity(Intent(this, MainActivity::class.java))
}

override fun onClick(view: View) {
   if (view.id == R.id.btn_login) {
      presenter.executeLogin(etUsername.text.toString(),
         etPassword.text.toString())
   } else if (view.id == R.id.btn_sign_up) {
      navigateToSignUp()
   }
}
```

Функция navigateToSignUp() использует при вызове явное намерение запустить SignUpActivity. Функция navigateToHome() работает аналогично navigateToSignUp() — она запускает MainActivity. Основное различие между navigateToHome() и navigateToSignUp() — в том, что функция navigateToHome() уничтожает текущий экземпляр LoginActivity, вызывая finish() перед запуском MainActivity.

Метод onClick() применяет LoginPresenter, чтобы начать в нашем сценарии процесс входа по щелчку на кнопке входа. С другой стороны, если щелкнуть на кнопке регистрации, функция navigateToSignUp() запустит SignUpActivity.

Отлично! Созданы необходимое представление, презентатор и модель для связанной с логином логики приложения. Следует учесть, что перед входом в систему необходимо зарегистрировать пользователя на платформе. То есть нам предстоит еще реализовать нашу логику регистрации. Выполним это в следующем разделе.

Разработка интерфейса регистрации (SignUp UI)

Приступим к разработке пользовательского интерфейса процесса регистрации. Во-первых, следует реализовать необходимые представления, начиная с создания макета для SignUpActivity. Создание макета SignUpActivity не потребует значитель-

ных усилий: нам понадобятся три поля ввода для указания имени пользователя, пароля и номера телефона пользователя, который проходит регистрацию, кнопка для отправки формы регистрации, а также индикатор выполнения, отображающий процесс выполнения регистрации.

Далее приведен код этого макета из файла activity_sign_up.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".ui.signup.SignUpActivity"
   android:paddingTop="@dimen/default_padding"
   android:paddingBottom="@dimen/default_padding"
   android:paddingStart="@dimen/default_padding"
   android:paddingEnd="@dimen/default_padding"
   android:orientation="vertical"
   android:gravity="center_horizontal">
<EditText
   android:id="@+id/et_username"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:hint="@string/username"
   android:inputType="text"/>
<EditText
   android:id="@+id/et_phone"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_marginTop="@dimen/default_margin"
   android:hint="@string/phone_number"
   android:inputType="phone"/>
<EditText
   android:id="@+id/et_password"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_marginTop="@dimen/default_margin"
   android:hint="@string/password"
   android:inputType="textPassword"/>
<Button
   android:id="@+id/btn_sign_up"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginTop="@dimen/default_margin"
   android:text="@string/sign_up"/>
```

```
<ProgressBar
   android:id="@+id/progress_bar"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginTop="@dimen/default_margin"
   android:visibility="gone"/>
</android.support.constraint.ConstraintLayout>
```

Визуальная реализация этого макета XML приведена на рис. 5.2.

Рис. 5.2. Визуальная реализация макета пользовательского интерфейса процесса регистрации

Как можно видеть, разработанный макет содержит все необходимые элементы, речь о которых шла ранее.

Создание интерактора регистрации

Теперь реализуем интерактор регистрации, который служит моделью для взаимодействия с еще нереализованным презентатором регистрации. Создайте внутри пакета signup интерфейс `SignUpInteractor`:

```
package com.example.messenger.ui.signup

import com.example.messenger.ui.auth.AuthInteractor
interface SignUpInteractor : AuthInteractor {
   interface OnSignUpFinishedListener {
      fun onSuccess()
```

```
        fun onUsernameError()
        fun onPasswordError()
        fun onPhoneNumberError()
        fun onError()
    }
    fun signUp(username: String, phoneNumber: String, password: String,
        listener: OnSignUpFinishedListener)
    fun getAuthorization(listener: AuthInteractor.onAuthFinishedListener)
}
```

Можно заметить, что интерактор `SignUpInteractor` расширяет `AuthInteractor`. Аналогично интерактору `LoginInteractor`, `SignUpInteractor` нуждается в применении свойств: `userDetails`, `accessToken`, `submittedUsername` и `submittedPassword`. Кроме того, `SignUpInteractor` нуждается в сохранении токена доступа пользователя, а также и данных пользователя с помощью функций `persistAccessToken(AppPreferences)` и `persistUserDetails(AppPreferences)`, объявленных в интерфейсе `AuthInteractor`.

Здесь мы также создали внутри интерактора `SignUpInteractor` интерфейс `OnSignUpFinishedListener`, объявив обратные связи, которые следует реализовать с помощью этого интерфейса. Прослушивателем их будет служить презентатор `SignUpPresenter`, который нам еще предстоит реализовать.

При формировании `SignUpInteractorImpl` мы начали с объявления свойств и реализации его метода `login()`. Создайте `SignUpInteractorImpl`, как показано далее, и удостоверьтесь, что он добавлен в тот же пакет, что и `SignUpInteractor`:

```
package com.example.messenger.ui.signup

import android.text.TextUtils
import android.util.Log
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.data.remote.request.LoginRequestObject
import com.example.messenger.data.remote.request.UserRequestObject
import com.example.messenger.data.vo.UserVO
import com.example.messenger.service.MessengerApiService
import com.example.messenger.ui.auth.AuthInteractor
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers

class SignUpInteractorImpl : SignUpInteractor {
    override lateinit var userDetails: UserVO
    override lateinit var accessToken: String
    override lateinit var submittedUsername: String
    override lateinit var submittedPassword: String
    private val service: MessengerApiService = MessengerApiService.getInstance()

    override fun signUp(username: String,
        phoneNumber: String, password: String,
```

```
        listener: SignUpInteractor.OnSignUpFinishedListener){
            submittedUsername = username
            submittedPassword = password
         val userRequestObject = UserRequestObject(username, password,
         phoneNumber)
         when {
            TextUtils.isEmpty(username) -> listener.onUsernameError()
            TextUtils.isEmpty(phoneNumber) -> listener.onPhoneNumberError()
            TextUtils.isEmpty(password) -> listener.onPasswordError()
         else -> {
```

Регистрация нового пользователя на платформе Messenger с помощью `MessengerApiService` выполняется посредством следующего кода:

```
                service.createUser(userRequestObject)
                        .subscribeOn(Schedulers.io())
                        .observeOn(AndroidSchedulers.mainThread())
                        .subscribe({ res ->
                userDetails = res
                listener.onSuccess()
            }, { error ->
                listener.onError()
                error.printStackTrace()
                })
            }
        }
    }
}
```

Теперь добавьте методы `getAuthorization()`, `persistAccessToken()` и `persistUserDetails()` под `SignUpInteractorImpl`:

```
override fun getAuthorization(listener:
AuthInteractor.onAuthFinishedListener) {
    val userRequestObject = LoginRequestObject(submittedUsername, submittedPassword)
```

Разрешим вход для уже зарегистрированного пользователя на платформе пользователя с помощью `MessengerApiService`:

```
service.login(userRequestObject)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe( { res ->
accessToken = res.headers()["Authorization"] as String
```

Теперь пользователь успешно вошел в систему — следовательно, вызовем функцию обратного вызова `onAuthSuccess()` слушателя:

```
        listener.onAuthSuccess()
    }, { error ->
        listener.onAuthError()
```

```
        error.printStackTrace()
    })
}
override fun persistAccessToken(preferences: AppPreferences) {
    preferences.storeAccessToken(accessToken)
}
override fun persistUserDetails(preferences: AppPreferences) {
    preferences.storeUserDetails(userDetails)
}
```

Класс `SignUpInteractorImpl` служит непосредственной реализацией для интерфейса `SignUpInteractor interface`. Обратите внимание: объявление класса `SignUpInteractorImpl` включает объявления свойств для `userDetails`, `accessToken`, `submittedUsername` и `submittedPassword`, которыми должен располагать слушатель `AuthInteractor.signUp (String, String, String, SignUpInteractor.OnSignUpFinishedListener)`, содержащий регистрационную логику приложения. Если действительны все представленные пользователем значения, он регистрируется на платформе с помощью функции `createUser(UserRequestObject)` из `MessengerApiService`, которая создана посредством `Retrofit`.

Слушатель `getAuthorization(AuthInteractor.onAuthFinishedListener)` вызывается для авторизации вновь зарегистрированного пользователя платформы по обмену сообщениями.

Обязательно просматривайте комментарии к классу `SignUpInteractorImpl`, приведенные в файлах примеров исходного кода, размещенных в сопровождающем книгу файловом архиве, для получения дополнительной информации.

Следующим в нашей повестке дня является создание презентатора регистрации `SignUpPresenter`.

Создание презентатора регистрации

Как это выполнялось при создании `LoginPresenter`, нужно сформировать интерфейс `SignUpPresenter` вместе с классом `SignUpPresenterImpl`. Формируемый `SignUpPresenter` совершенно не сложен. Для нашего приложения нам необходимо располагать презентатором регистрации, имеющим свойство типа `AppPreferences`, а также выполняющую процесс регистрации функцию. Далее приводится интерфейс `SignUpPresenter`:

```
package com.example.messenger.ui.signup
import com.example.messenger.data.local.AppPreferences
interface SignUpPresenter {
    var preferences: AppPreferences
    fun executeSignUp(username: String, phoneNumber: String, password:
    String)
}
```

Далее приводится код для реализации `SignUpPresenter`:

```
package com.example.messenger.ui.signup
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.ui.auth.AuthInteractor
class SignUpPresenterImpl(private val view: SignUpView): SignUpPresenter,
SignUpInteractor.OnSignUpFinishedListener,
AuthInteractor.onAuthFinishedListener {
   private val interactor: SignUpInteractor = SignUpInteractorImpl()
   override var preferences: AppPreferences = AppPreferences
                                    .create(view.getContext())
```

Указанная далее функция обратной связи `onSuccess()` **вызывается, если пользователь успешно зарегистрировался:**

```
override fun onSuccess() {
   interactor.getAuthorization(this)
}
```

Обратный вызов формируется при возникновении ошибки во время регистрации пользователя:

```
override fun onError() {
   view.hideProgress()
   view.showSignUpError()
}

override fun onUsernameError() {
   view.hideProgress()
   view.setUsernameError()
}

override fun onPasswordError() {
   view.hideProgress()
   view.setPasswordError()
}

override fun onPhoneNumberError() {
   view.hideProgress()
   view.setPhoneNumberError()
}

override fun executeSignUp(username: String, phoneNumber: String,
   password: String) {
   view.showProgress()
   interactor.signUp(username, phoneNumber, password, this)
}
```

```
override fun onAuthSuccess() {
    interactor.persistAccessToken(preferences)
    interactor.persistUserDetails(preferences)
    view.hideProgress()
    view.navigateToHome()
}

override fun onAuthError() {
    view.hideProgress()
    view.showAuthError()
    }
}
```

Приведенный ранее класс `SignUpPresenterImpl` реализует интерфейсы `SignUpPresenter`, `SignUpInteractor.OnSignUpFinishedListener` и `AuthInteractor.onAuthFinishedListener`, которые поддерживают реализации для ряда необходимых функций. Этими функциями являются `onSuccess()`, `onError()`, `onUsernameError()`, `onPasswordError()`, `onPhoneNumberError()`, `executeSignUp(String, String, String)`, `onAuthSuccess()` и `onAuthError()`. Класс `SignUpPresenterImpl` принимает единственный аргумент как основной конструктор. Этим аргументом должен быть тип `SignUpView`.

Функция `executeSignUp(String, String, String)` вызывается `SignUpView` для начала процесса регистрации пользователя. Функция `onSuccess()` вызывается, если успешно выполнен запрос на регистрацию пользователя. Эта функция немедленно вызывает функцию интерактора `getAuthorization()` для получения токена доступа для вновь зарегистрированного пользователя. В случае, когда запрос на регистрацию отвергнут, вызывается обратная связь `onError()`. Это приводит к сокрытию отображаемого пользователю индикатора выполнения, при этом появляется соответствующее сообщение об ошибке.

Методы `onUsernameError()`, `onPasswordError()` и `onPhoneNumberError()` служат обратными связями и вызываются при возникновении ошибки в предоставленном имени пользователя, пароле или номере телефона соответственно. Функция `onAuthSuccess()` является обратной связью при успешном выполнении процедуры авторизации. С другой стороны, `onAuthError()` вызывается, если авторизация не выполнена.

Создание представления регистрации

Пришло время выполнить обработку `SignUpView`. Сначала необходимо создать интерфейс `SignUpView`, а затем и реализацию `SignUpActivity` этого интерфейса. Заметим, что в нашем приложении `SignUpView` является расширением `BaseView` и `AuthView`. Далее приводится интерфейс `SignUpView`:

```
package com.example.messenger.ui.signup

import com.example.messenger.ui.auth.AuthView
import com.example.messenger.ui.base.BaseView
```

```
interface SignUpView : BaseView, AuthView {

    fun showProgress()
    fun showSignUpError()
    fun hideProgress()
    fun setUsernameError()
    fun setPhoneNumberError()
    fun setPasswordError()
    fun navigateToHome()
}
```

Теперь приступим к модификации класса `SignUpActivity` для реализации `SignUpView` и применим `SignUpPresenter`. Внесем изменения в следующий фрагмент кода для `SignUpActivity`:

```
package com.example.messenger.ui.signup

import android.content.Context
import android.content.Intent
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.EditText
import android.widget.ProgressBar
import android.widget.Toast
import com.example.messenger.R
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.ui.main.MainActivity

class SignUpActivity : AppCompatActivity(), SignUpView,
View.OnClickListener {
    private lateinit var etUsername: EditText
    private lateinit var etPhoneNumber: EditText
    private lateinit var etPassword: EditText
    private lateinit var btnSignUp: Button
    private lateinit var progressBar: ProgressBar
    private lateinit var presenter: SignUpPresenter
    override fun onCreate(savedInstanceState: Bundle?) {

    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_sign_up)
    presenter = SignUpPresenterImpl(this)
    presenter.preferences = AppPreferences.create(this)
    bindViews()
}
```

```
override fun bindViews() {
   etUsername = findViewById(R.id.et_username)
   etPhoneNumber = findViewById(R.id.et_phone)
   etPassword = findViewById(R.id.et_password)
   btnSignUp = findViewById(R.id.btn_sign_up)
   progressBar = findViewById(R.id.progress_bar)
   btnSignUp.setOnClickListener(this)
}

override fun showProgress() {
   progressBar.visibility = View.VISIBLE
}

override fun hideProgress() {
   progressBar.visibility = View.GONE
}

override fun navigateToHome() {
   finish()
   startActivity(Intent(this, MainActivity::class.java))
}

override fun onClick(view: View) {
   if (view.id == R.id.btn_sign_up) {
      presenter.executeSignUp(etUsername.text.toString(),
      etPhoneNumber.text.toString(),
      etPassword.text.toString())
     }
     }
}
```

Теперь добавим к `SignUpActivity` **приведенные далее функции** `setUsernameError()`, `setPhoneNumberError()`, `setPasswordError()`, `showAuthError()`, `showSignUpError()` **и** `getContext()`:

```
override fun setUsernameError() {
  etUsername.error = "Username field cannot be empty"
}

override fun setPhoneNumberError() {
   etPhoneNumber.error = "Phone number field cannot be empty"
}

override fun setPasswordError() {
   etPassword.error = "Password field cannot be empty"
}
```

```
override fun showAuthError() {
   Toast.makeText(this, "An authorization error occurred.
   Please try again later.",
   Toast.LENGTH_LONG).show()
}

override fun showSignUpError() {
   Toast.makeText(this, "An unexpected error occurred.
   Please try again later.",
   Toast.LENGTH_LONG).show()
}

override fun getContext(): Context {
   return this
}
```

Отличная работа! К этому моменту мы уже на полпути к завершению разработки приложения Messenger. Ваши усилия не пропали даром. Но нам есть еще над чем поработать в части, относящейся к основному интерфейсу приложения, которую мы рассмотрим в следующей главе.

Подведем итоги

В этой главе начат процесс разработки Android-приложения Messenger. При этом мы рассмотрели широкий диапазон тем: описан шаблон «модель-представление-презентатор» (Model-View-Presenter, MVP) и подробно показано, каким образом можно создавать приложения, используя подобный современный подход к разработке.

Кроме того, здесь широко использовались RxJava и RxAndroid, в связи с чем приведены сведения о реактивном программировании. Показано также, каким образом можно взаимодействовать с удаленным сервером, используя библиотеки `OkHttp` и `Retrofit`, после чего сделан еще один шаг вперед и внедрен полнофункциональный сервис Retrofit для взаимодействия с API Messenger, который был разработан в *главе 4*.

В следующей главе мы завершим создание приложения Messenger.

6

Создание Android-приложения Messenger: часть II

В предыдущей главе мы активно занимались созданием Android-приложения Messenger. Нами был подробно рассмотрен процесс разработки Android-приложений на языке Kotlin и исследованы шаблон «модель-представление-презентатор» (MVP) и способы его применения для создания мощных и полнофункциональных Android-приложений. Мы познакомились с основами реактивного программирования и узнали, как использовать RxJava и RxAndroid при создании пользовательских приложений. Мы также рассмотрели некоторые доступные средства связи с удаленным сервером, получили информацию о библиотеках `OkHttp` и `Retrofit` и внедрили полнофункциональный сервис Retrofit для облегчения связи с API Messenger (см. *главу 4*). В итоге, путем интеграции всех сведений, полученных нами в части Android и Kotlin, мы создали пользовательский интерфейс входа и регистрации для приложения Messenger.

В этой главе мы завершим разработку приложения Messenger, рассмотрев следующие темы:

- работа с настройками приложения;
- использование ChatKit;
- тестирование Android-приложения;
- выполнение фоновых задач.

А осталось нам реализовать основной интерфейс пользователя (Main UI).

Создание основного интерфейса пользователя (Main UI)

Подобно тому как выполнялась реализация Login UI (интерфейса входа в систему) и SignUp UI (интерфейса регистрации в системе), создадим модель, представление

и презентатор для основного интерфейса пользователя. Пояснения здесь будут более краткими по сравнению с предыдущим изложением — мы объясним только новые концепции. И начнем с разработки основного представления (MainView).

Разработка основного представления *MainView*

Прежде чем мы приступим к созданию основного представления, необходимо четко понимать задачи пользовательского интерфейса, который необходимо реализовать, для чего следует однозначно сформулировать положения, описывающие функциональные возможности MainView:

- основное представление должно отображать активные беседы (conversations) текущего пользователя, вошедшего в систему при запуске;
- основное представление должно позволять вошедшему в систему пользователю создавать новую беседу;
- основное представление должно отображать контакты пользователя, вошедшего в систему в текущий момент (а для нашего приложения — выводить список всех пользователей, зарегистрированных на платформе Messenger);
- пользователь должен иметь доступ к экрану настроек непосредственно из основного представления;
- пользователь должен иметь возможность выхода из приложения непосредственно из основного представления.

Отлично! У нас есть список кратких положений, описывающих возможности MainView, и на его основе можно продолжить создание представления MainView непосредственно в кодах. Но мы пока отложим эту работу и создадим визуальный набросок MainView, что позволит нам сформировать более четкое представление о том, какой вид будет иметь это представление на самом деле (рис. 6.1).

Как видно из рис. 6.1, MainActivity может предоставлять пользователю два совершенно разных представления: **Conversations Screen** (Экран бесед) и **Contacts Screen** (Экран контактов). Идеальный способ реализовать этот вариант — развернуть в пределах MainActivity два отдельных фрагмента: фрагмент бесед и фрагмент контактов.

Имея четкое представление о содержании MainView, реализуем интерфейс для объявления поведений MainView:

```
package com.example.messenger.ui.main
import com.example.messenger.ui.base.BaseView

interface MainView : BaseView {
   fun showConversationsLoadError()
   fun showContactsLoadError()
   fun showConversationsScreen()
   fun showContactsScreen()
```

```
    fun getContactsFragment(): MainActivity.ContactsFragment
    fun getConversationsFragment(): MainActivity.ConversationsFragment
    fun showNoConversations()
    fun navigateToLogin()
    fun navigateToSettings()
}
```

Отлично! Отложим пока реализацию `MainView` для `MainActivity` для дальнейшей обработки, а сейчас займемся интерактором `MainInteractor`.

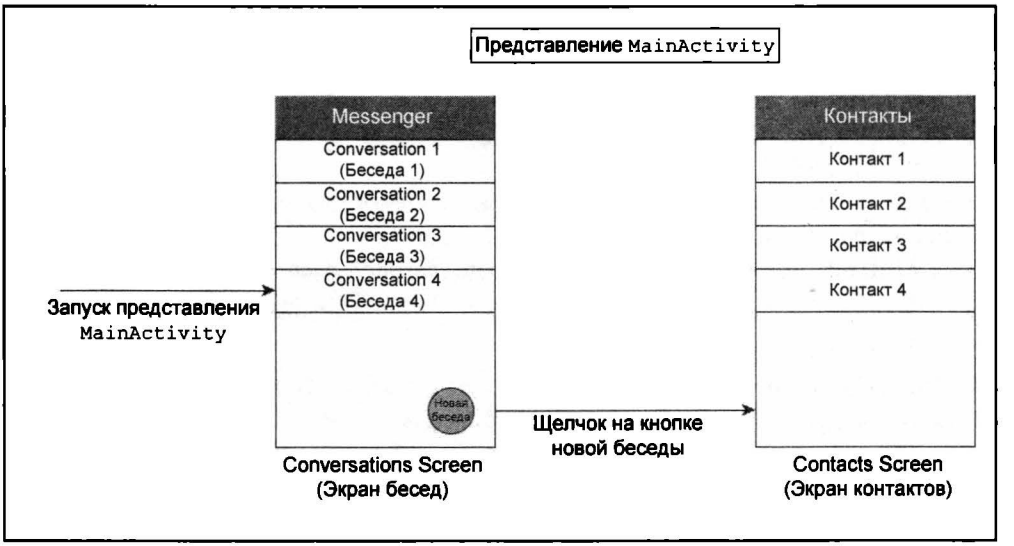


Рис. 6.1. Схема представления `MainActivity`

Создание интерактора *MainInteractor*

Желательно, чтобы пользователь просматривал контакты других пользователей на платформе Messenger, а их активные беседы — в главном представлении (на экране). Было бы неплохо также, чтобы пользователь имел возможность выходить из платформы непосредственно с главного экрана. Чтобы реализовать эти пожелания, интерфейс `MainInteractor` должен уметь выполнять загрузку контактов, бесед и обеспечивать выход из платформы пользователя. Далее приводится интерфейс `MainInteractor`. Удостоверьтесь, что этот интерфейс и все остальные файлы группы Main находятся в пакете com.example.messenger.ui.main:

```
package com.example.messenger.ui.main

import com.example.messenger.data.vo.ConversationListVO
import com.example.messenger.data.vo.UserListVO
interface MainInteractor {
```

```
interface OnConversationsLoadFinishedListener {
    fun onConversationsLoadSuccess(
       conversationsListVo: ConversationListVO)
       fun onConversationsLoadError()
    }

interface OnContactsLoadFinishedListener {
    fun onContactsLoadSuccess(userListVO: UserListVO)
    fun onContactsLoadError()
}

interface OnLogoutFinishedListener {
    fun onLogoutSuccess()
}

fun loadContacts(
   listener: MainInteractor.OnContactsLoadFinishedListener)
   fun loadConversations(
      listener: MainInteractor.OnConversationsLoadFinishedListener)
      fun logout(listener: MainInteractor.OnLogoutFinishedListener)
}
```

Здесь мы добавили к интерфейсу `MainInteractor` **интерфейсы** `OnConversations-LoadFinishedListener`, `OnContactsLoadFinishedListener` **и** `OnLogoutFinishedListener`. **Они все реализованы с помощью** `MainPresenter`. **Эти обратные вызовы необходимы для презентатора в целях выполнения соответствующих действий независимо от успешности или неуспешности при загрузке беседы, контакта или выполнении процесса выхода пользователя из системы.**

Далее представлен класс `MainInteractorImpl` **с реализованным методом** `loadContacts()`:

```
package com.example.messenger.ui.main

import android.content.Context
import android.util.Log
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.data.remote.repository.ConversationRepository
import com.example.messenger.data.remote.repository.ConversationRepositoryImpl
import com.example.messenger.data.remote.repository.UserRepository
import com.example.messenger.data.remote.repository.UserRepositoryImpl
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers
class MainInteractorImpl(val context: Context) : MainInteractor {
   private val userRepository: UserRepository = UserRepositoryImpl(context)
   private val conversationRepository: ConversationRepository =
                                                  ConversationRepositoryImpl(context)
   override fun loadContacts(listener:
      MainInteractor.OnContactsLoadFinishedListener) {
```

Загрузим всех зарегистрированных на платформе Messenger API пользователей. Эти пользователи будут контактами для текущего вошедшего в систему пользователя:

```
userRepository.all()
              .subscribeOn(Schedulers.io())
              .observeOn(AndroidSchedulers.mainThread())
              .subscribe({ res ->
```

Теперь контакты успешно загружены. Функция `onContactsLoadSuccess()` вызывается с помощью данных ответа API, переданных в качестве аргумента:

```
listener.onContactsLoadSuccess(res) },
   { error ->
```

Если загрузка контакта неуспешна, тогда вызывается функция `onContactsLoadError()`:

```
      listener.onContactsLoadError()
      error.printStackTrace()})
   }
}
```

Функция `loadContacts()` обращается к репозиторию `UserRepository` для загрузки списка всех доступных на платформе мессенджера пользователей. Если пользователи извлечены успешно, вызывается функция слушателя `onContactsLoadSuccess()` со списком загруженных пользователей, переданным в качестве аргумента. Иначе вызывается функция `onContactsLoadError()` и в области стандартного вывода системы показывается сообщение об ошибке.

До сих пор нам не приходилось применять класс `MainInteractorImpl`. К нему следует добавить функции `loadConversations()` и `logout()` — эти две обязательные функции приведены в следующем фрагменте кода. Добавьте их в класс `MainInteractorImpl`.

```
   override fun loadConversations(
   listener: MainInteractor.OnConversationsLoadFinishedListener) {
```

Здесь выполняется извлечение всех бесед вошедшего в систему текущего пользователя, при этом используется экземпляр репозитория для бесед:

```
      conversationRepository.all()
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe({ res -> listener.onConversationsLoadSuccess(res) },
      { error ->
         listener.onConversationsLoadError()
         error.printStackTrace()})
      }
      override fun logout(
         listener: MainInteractor.OnLogoutFinishedListener) {
```

При входе в систему выполняется очистка данных пользователя из файла общих настроек и производится обратный вызов `onLogoutSuccess()` слушателя:

```
    val preferences: AppPreferences = AppPreferences.create(context)
    preferences.clear()
    listener.onLogoutSuccess()
}
```

Функция `loadConversations()` работает подобно функции `loadContacts()`. Различие их в том, что `ConversationRepository` используется для извлечения вместо списка контактов активных бесед, которые в текущий момент ведет пользователь. Функция `logout()` просто очищает файл настроек, используемый приложением для удаления данных текущего пользователя, затем вызывается метод `onLogoutSuccess()` для поддерживаемого интерфейса `OnLogoutFinishedListener`.

И это все, что касается класса `MainInteractorImpl`. Следующей на повестке дня является реализация интерфейса `MainPresenter`.

Создание презентатора *MainPresenter*

Как всегда, первое, что следует сделать — это создать интерфейс презентатора, который определяет функции для их реализации классом реализации презентатора. Далее приводится код интерфейса `MainPresenter`:

```
package com.example.messenger.ui.main

interface MainPresenter {
    fun loadConversations()
    fun loadContacts()
    fun executeLogout()
}
```

Функции `loadConversations()`, `loadContacts()` и `executeLogout()` вызываются представлением `MainView` и должны реализовываться классом `MainPresenterImpl`.

Код класса `MainPresenterImpl` с определенными свойствами, а также методами `onConversationsLoadSuccess()` и `onConversationsLoadError()` приводится далее:

```
package com.iyanuadelekan.messenger.ui.main

import com.iyanuadelekan.messenger.data.vo.ConversationListVO
import com.iyanuadelekan.messenger.data.vo.UserListVO
class MainPresenterImpl(val view: MainView) : MainPresenter,
MainInteractor.OnConversationsLoadFinishedListener,
MainInteractor.OnContactsLoadFinishedListener,
MainInteractor.OnLogoutFinishedListener {
    private val interactor: MainInteractor = MainInteractorImpl
    (view.getContext())
    override fun onConversationsLoadSuccess(conversationsListVo:
    ConversationListVO) {
        Let's check if currently logged in user has active conversations:
```

```
      if (!conversationsListVo.conversations.isEmpty()) {
         val conversationsFragment = view.getConversationsFragment()
         val conversations = conversationsFragment.conversationsval adapter =
                                        conversationsFragment.conversationsAdapter
conversations.clear()
adapter.notifyDataSetChanged()
```

После получения бесед из API добавим каждую беседу в список бесед для `ConversationFragment`, причем адаптер бесед уведомляется после каждого добавленного элемента:

```
      conversationsListVo.conversations.forEach { contact ->
      conversations.add(contact)
      adapter.notifyItemInserted(conversations.size - 1)
      }
   } else {
      view.showNoConversations()
   }
}
override fun onConversationsLoadError() {
   view.showConversationsLoadError()
   }
}
```

Кроме того, добавьте приведенные далее функции `onContactsLoadSuccess()`, `onContactsLoadError()`, `onLogoutSuccess()`, `loadConversations()`, `loadContacts()` и `executeLogout()` в класс `MainPresenterImpl`:

```
override fun onContactsLoadSuccess(userListVO: UserListVO) {

   val contactsFragment = view.getContactsFragment()
   val contacts = contactsFragment.contacts
   val adapter = contactsFragment.contactsAdapter
```

Очистим ранее загруженные контакты в списке контактов и уведомим адаптер об изменении набора данных:

```
contacts.clear()
adapter.notifyDataSetChanged()
```

Теперь добавим каждый полученный из API контакт в список контактов для `ContactsFragment`, причем адаптер контактов уведомляется после каждого добавления элемента:

```
      userListVO.users.forEach { contact ->
      contacts.add(contact)
      contactsFragment.contactsAdapter.notifyItemInserted(
         contacts.size-1)
   }
}
```

```
override fun onContactsLoadError() {
   view.showContactsLoadError()
}

override fun onLogoutSuccess() {
   view.navigateToLogin()
}

override fun loadConversations() {
   interactor.loadConversations(this)
}

override fun loadContacts() {
   interactor.loadContacts(this)
}

override fun executeLogout() {
   interactor.logout(this)
}
```

Итак, мы успешно создали интерфейсы `MainInteractor` и `MainPresenter`. Пришло время завершить работу над `MainView` и его макетами.

Реализация *MainView*

Прежде всего необходимо поработать с файлом макета activity_main.xml. Измените этот файл, чтобы он включал следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".ui.main.MainActivity">
<LinearLayout
android:id="@+id/ll_container"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"/>
</android.support.design.widget.CoordinatorLayout>
```

В корневом представлении файла макета имеется единственный макет `LinearLayout`. Это представление `ViewGroup` играет роль контейнера для бесед и фрагментов контактов. Если речь идет о фрагменте бесед и фрагменте контактов, необходимо сформировать для них соответствующие макеты. Создайте файл макета fragment_conversations.xml в каталоге ресурса макета проекта со следующим содержанием:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
xmlns:app="http://schemas.android.com/apk/res-auto">
<android.support.v7.widget.RecyclerView
   android:id="@+id/rv_conversations"
   android:layout_width="match_parent"
android:layout_height="match_parent"/>
<android.support.design.widget.FloatingActionButton
   android:id="@+id/fab_contacts"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_margin="@dimen/default_margin"
   android:src="@android:drawable/ic_menu_edit"
app:layout_anchor="@id/rv_conversations"
app:layout_anchorGravity="bottom|right|end"/>
</android.support.design.widget.CoordinatorLayout>
```

Здесь внутри корневого представления `CoordinatorLayout` создаются два дочерних представления: `RecyclerView` и `FloatingActionButton`. Представление `RecyclerView` является виджетом Android, применяемым в качестве контейнера для отображения больших наборов данных, которые можно эффективно прокручивать, поддерживая ограниченное количество просмотров. Виджет `RecyclerView`, раз его зависимость добавлена на уровне build.gradle сценария модуля нашего проекта, можно использовать следующим образом:

```
implementation 'com.android.support:recyclerview-v7:26.1.0'
```

Поскольку мы применяем виджеты `RecyclerView`, нам необходимо создать соответствующие макеты держателей представлений для каждого виджета `RecyclerView`. Так что создадим внутри каталога макетов ресурсов файлы vh_contacts.xml и vh_conversations.xml.

Далее приводится содержимое макета vh_contacts.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="match_parent"
   android:id="@+id/ll_container"
   android:layout_height="wrap_content">
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="vertical"
   android:padding="@dimen/default_padding">
```

```
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="horizontal">
<TextView
   android:id="@+id/tv_username"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:textSize="18sp"
   android:textStyle="bold"/>
<LinearLayout
   android:layout_width="0dp"
   android:layout_height="wrap_content"
   android:layout_weight="1"
   android:gravity="end">
<TextView
   android:id="@+id/tv_phone"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginLeft="@dimen/default_margin"
   android:layout_marginStart="@dimen/default_margin"/>
</LinearLayout>
</LinearLayout>
<TextView
   android:id="@+id/tv_status"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"/>
</LinearLayout>
<View
   android:layout_width="match_parent"
   android:layout_height="1dp"
   android:background="#e8e8e8"/>
</LinearLayout>
```

А здесь — содержимое макета vh_conversations.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="match_parent"
   android:id="@+id/ll_container"
   android:layout_height="wrap_content">
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="vertical"
   android:padding="@dimen/default_padding">
```

```
<TextView
   android:id="@+id/tv_username"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:textStyle="bold"
   android:textSize="18sp"/>
<TextView
   android:id="@+id/tv_preview"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"/>
</LinearLayout>
<View
   android:layout_width="match_parent"
   android:layout_height="1dp"
   android:background="#e8e8e8"/>
</LinearLayout>
```

Как указано в справочном руководстве разработчиков Android: *кнопки действия Floating (плавающее действие) применяются для особого типа действия. Они отличаются значком в виде кружка, который «плавает» над пользовательским интерфейсом, имеют специальные режимы движения, связанные с морфингом, запуском и точкой привязки переноса.*

Таким образом, мы можем применить виджет `FloatingActionButton`, поскольку мы добавили зависимость библиотеки дизайна поддержки Android в сценарий проекта build.gradle:

```
implementation 'com.android.support:design:26.1.0'
```

Создайте файл макета fragment_contacts.xml внутри каталога макета ресурсов, включающий следующий XML-код:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="match_parent"
   android:layout_height="match_parent">
   <android.support.v7.widget.RecyclerView
   android:id="@+id/rv_contacts"
   android:layout_width="match_parent"
   android:layout_height="match_parent"/>
</LinearLayout>
```

Пришло время завершить создание класса `MainActivity`. Для этого придется выполнить большую работу. Прежде всего надо объявить необходимые свойства класса. Затем требуется поддержать реализации для следующих методов: `bindViews()`, `showConversationsLoadError()`, `showContactsLoadError()`, `showConversationsScreen()`, `showContactsScreen()`, `getContext()`, `getContactsFragment()`, `getConversationsFragment()`, `navigateToLogin()` и `navigateToSettings()`. Наконец, надо создать классы `ConversationsFragment` и `ContactsFragment`.

В первую очередь добавим фрагменты `ConversationsFragment` и `ContactsFragment` в представление `MainActivity`. Код `ConversationsFragment` приводится далее. Включите его в представление `MainActivity`.

```
// Класс ConversationsFragment расширяет класс Fragment
class ConversationsFragment : Fragment(), View.OnClickListener {

   private lateinit var activity: MainActivity
   private lateinit var rvConversations: RecyclerView
   private lateinit var fabContacts: FloatingActionButton
   var conversations: ArrayList<ConversationVO> = ArrayList()
   lateinit var conversationsAdapter: ConversationsAdapter
```

Если первоначально создается интерфейс пользователя `ConversationsFragment`, вызывается следующий метод:

```
override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup, savedInstanceState: Bundle?): View? {
   // Раздувание макета фрагмента
   val baseLayout =
      inflater.inflate(R.layout.fragment_conversations,
      container, false)

   // Привязки представления макета
   rvConversations = baseLayout.findViewById(R.id.rv_conversations)
   fabContacts = baseLayout.findViewById(R.id.fab_contacts)
   conversationsAdapter = ConversationsAdapter(
   getActivity(),.conversations)

   // Настройка адаптера представления переработчика бесед
   // для созданного адаптера бесед
   rvConversations.adapter = conversationsAdapter
```

Настроим диспетчер макетов бесед переработчика и разберемся, каким образом можно просматривать диспетчер линейного макета:

```
rvConversations.layoutManager =
   LinearLayoutManager(getActivity().baseContext)
   fabContacts.setOnClickListener(this)
   return baseLayout
}

   override fun onClick(view: View) {
      if (view.id == R.id.fab_contacts) {
         this.activity.showContactsScreen()
      }
   }
```

```
    fun setActivity(activity: MainActivity) {
        this.activity = activity
    }
}
```

Фрагмент `ConversationsFragment` **владеет элементом макета** `RecyclerView`. **Представления** `Recycler` **нуждаются в адаптерах для обеспечения привязки набора данных к представлениям, отображаемым в представлении** `RecyclerView`. **Просто включим в представление** `RecyclerView` **применение адаптера** `Adapter` **для поддержки данных для представлений, которые он отображает на дисплее. Добавим** `ConversationsAdapter` **в качестве внутреннего класса для** `ConversationsFragment`:

```
class ConversationsAdapter(private val context:
Context, private val dataSet: List<ConversationVO>) :
RecyclerView.Adapter<ConversationsAdapter.ViewHolder>(),
    ChatView.ChatAdapter {
        val preferences: AppPreferences =
            AppPreferences.create(context)
        override fun onBindViewHolder(holder: ViewHolder, position:
            Int) {
                val item = dataSet[position] // получение элемента в текущей позиции
                val itemLayout = holder.itemLayout // привязка макета
                // держателя представления к локальной переменной
         itemLayout.findViewById<TextView>(R.id.tv_username).text =
             item.secondPartyUsername
         itemLayout.findViewById<TextView>(R.id.tv_preview).text =
             item.messages[item.messages.size - 1].body
```

Теперь установим `View.OnClickListener` **для** `itemLayout`:

```
itemLayout.setOnClickListener {
    val message = item.messages[0]
    val recipientId: Long
    recipientId = if (message.senderId ==
        preferences.userDetails.id) {
        message.recipientId
    } else {
        message.senderId
    }
    navigateToChat(item.secondPartyUsername,
        recipientId, item.conversationId)
    }
}

override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): ViewHolder {
```

Теперь давайте создадим макет держателя:

```
    val itemLayout = LayoutInflater.from(parent.context)
    .inflate(R.layout.vh_conversations, null, false)
    .findViewById<LinearLayout>(R.id.ll_container)
     return ViewHolder(itemLayout)
}

override fun getItemCount(): Int {
    return dataSet.size
}

override fun navigateToChat(recipientName: String,
recipientId: Long, conversationId: Long?) {
    val intent = Intent(context, ChatActivity::class.java)
    intent.putExtra("CONVERSATION_ID", conversationId)
    intent.putExtra("RECIPIENT_ID", recipientId)
    intent.putExtra("RECIPIENT_NAME", recipientName)
    context.startActivity(intent)
}

class ViewHolder(val itemLayout: LinearLayout)
    RecyclerView.ViewHolder(itemLayout)
}
```

При формировании представления переработчика `Adapter` существует ряд важных методов, для которых нужно предоставить заказные реализации. Этими методами являются: `onCreateViewHolder()`, `onBindViewHolder()` и `getItemCount()`. Метод `onCreateViewHolder()` вызывается, если представлению переработчика требуется новый экземпляр держателя представления. Метод `onBindViewHolder()` вызывается представлением переработчика для отображения данных в наборе данных с указанным местоположением. Метод `getItemCount()` вызывается для получения количества пунктов в наборе данных. Представление `ViewHolder` описывает используемый элемент представления, а также метаданные о его месте в представлении `RecyclerView`.

Внутренним классом является класс, вложенный в другой класс.

Понимая, что происходит в `ConversationsFragment`, перейдем к реализации фрагмента `ContactsFragment`. Добавим в класс `MainActivity` следующий класс `ContactsFragment`:

```
class ContactsFragment : Fragment() {
    private lateinit var activity: MainActivity
    private lateinit var rvContacts: RecyclerView
    var contacts: ArrayList<UserVO> = ArrayList()
    lateinit var contactsAdapter: ContactsAdapter
```

```
    override fun onCreateView(inflater: LayoutInflater,
    container: ViewGroup, savedInstanceState: Bundle?): View? {
      val baseLayout = inflater.inflate(R.layout.fragment_contacts,
         container, false)
    rvContacts = baseLayout.findViewById(R.id.rv_contacts)
     contactsAdapter = ContactsAdapter(getActivity(), contacts)
    rvContacts.adapter = contactsAdapter
    rvContacts.layoutManager = LinearLayoutManager(getActivity().baseContext)
    return baseLayout
}
    fun setActivity(activity: MainActivity) {
       this.activity = activity
    }
}
```

Как можно заметить, аналогично `ConversationsFragment` **фрагмент** `ContactsFragment` **применяет** `RecyclerView` **для отображения пользователю приложения элементов представления контактов. Соответствующим классом адаптера для этого** `RecyclerView` **является** `ContactsAdapter`. **Он представлен в следующем фрагменте кода. Добавьте его во внутренний класс для** `ContactsFragment`:

```
class ContactsAdapter(private val context: Context,
                      private val dataSet: List<UserVO>) :
                      RecyclerView.Adapter<ContactsAdapter.ViewHolder>(),
                      ChatView.ChatAdapter {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val itemLayout = LayoutInflater.from(parent.context)
                    .inflate(R.layout.vh_contacts, parent, false)
    val llContainer = itemLayout.findViewById<LinearLayout>
                     (R.id.ll_container)
    return ViewHolder(llContainer)
}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val item = dataSet[position]
    val itemLayout = holder.itemLayout
    itemLayout.findViewById<TextView>(R.id.tv_username).text = item.username
    itemLayout.findViewById<TextView>(R.id.tv_phone).text = item.phoneNumber
    itemLayout.findViewById<TextView>(R.id.tv_status).text = item.status
       itemLayout.setOnClickListener {
       navigateToChat(item.username, item.id)
    }
}

override fun getItemCount(): Int {
    return dataSet.size
}
```

```
override fun navigateToChat(recipientName: String,recipientId: Long,
    conversationId: Long?) {
    val intent = Intent(context, ChatActivity::class.java)
    intent.putExtra("RECIPIENT_ID", recipientId)
    intent.putExtra("RECIPIENT_NAME", recipientName)
    context.startActivity(intent)
}

class ViewHolder(val itemLayout: LinearLayout) :
    RecyclerView.ViewHolder(itemLayout)
}
```

Итак, все хорошо, и после создания необходимых фрагментов можно приступить к работе над свойствами и методами класса `MainActivity`. Добавьте приведенные далее определения свойств в верхнюю часть класса `MainActivity`:

```
private lateinit var llContainer: LinearLayout
private lateinit var presenter: MainPresenter
// Создание экземпляров фрагмента
private val contactsFragment = ContactsFragment()
private val conversationsFragment = ConversationsFragment()
```

Затем модифицируйте метод `onCreate()` для отражения следующих изменений:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    presenter = MainPresenterImpl(this)
    conversationsFragment.setActivity(this)
    contactsFragment.setActivity(this)
    bindViews()
    showConversationsScreen()
}
```

Теперь добавьте методы `bindViews()`, `showConversationsLoadError()`, `showContactsLoadError()`, а также методы `showConversationsScreen()` и `showContactsScreen()` в нижнюю часть класса `MainActivity`:

```
override fun bindViews() {
    llContainer = findViewById(R.id.ll_container)
}

override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.main, menu)
    return super.onCreateOptionsMenu(menu)
}

override fun showConversationsLoadError() {
    Toast.makeText(this, "Unable to load conversations.
```

```
    Try again later.",
    Toast.LENGTH_LONG).show()
}

override fun showContactsLoadError() {
    Toast.makeText(this, "Unable to load contacts. Try again later.",
    Toast.LENGTH_LONG).show()
}
```

Давайте начнем новую транзакцию фрагмента и заменим любой фрагмент, присутствующий в контейнере фрагмента действия, на `ConversationsFragment`:

```
override fun showConversationsScreen() {
    val fragmentTransaction = fragmentManager.beginTransaction()
    fragmentTransaction.replace(R.id.ll_container, conversationsFragment)
    fragmentTransaction.commit()

    // Начало процесса загрузки беседы
    presenter.loadConversations()
    supportActionBar?.title = "Messenger"
    supportActionBar?.setDisplayHomeAsUpEnabled(false)
}

override fun showContactsScreen() {
    val fragmentTransaction = fragmentManager.beginTransaction()
    fragmentTransaction.replace(R.id.ll_container, contactsFragment)
    fragmentTransaction.commit()
    presenter.loadContacts()
    supportActionBar?.title = "Contacts"
    supportActionBar?.setDisplayHomeAsUpEnabled(true)
}
```

Наконец, добавим функции `showNoConversations()`, `onOptionsItemSelected()`, `getContext()`, `getContactsFragment()`, `getConversationsFragment()`, `navigateToLogin()` **и** `navigateToSettings()` **в нижнюю часть класса** `MainActivity`:

```
override fun showNoConversations() {
    Toast.makeText(this, "You have no active conversations.",
    Toast.LENGTH_LONG).show()
}

override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    when (item?.itemId) {
    android.R.id.home -> showConversationsScreen()
    R.id.action_settings -> navigateToSettings()
    R.id.action_logout -> presenter.executeLogout()
}
return super.onOptionsItemSelected(item)
}
```

```
override fun getContext(): Context {
   return this
}

override fun getContactsFragment(): ContactsFragment {
   return contactsFragment
}

override fun getConversationsFragment(): ConversationsFragment {
   return conversationsFragment
}

override fun navigateToLogin() {
   startActivity(Intent(this, LoginActivity::class.java))
   finish()
}

override fun navigateToSettings() {
   startActivity(Intent(this, SettingsActivity::class.java))
}
```

Благодаря комментариям, сопровождающим приведенные фрагменты кода в файлах примеров, размещенных в прилагаемом к книге файловом архиве, вы сможете лучше понять, что было сделано. Обязательно внимательно читайте эти комментарии!

Создание меню *MainActivity*

В функции `onCreateOptionsMenu(Menu)` из `MainActivity` содержится меню, которое еще не реализовано. Добавьте файл main.xml в пакет menu под каталогом ресурсов приложения. Файл main.xml должен включать следующее содержимое:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto">
<item
   android:id="@+id/action_settings"
   android:orderInCategory="100"
   android:title="@string/action_settings"
   app:showAsAction="never" />
<item
   android:id="@+id/action_logout"
   android:orderInCategory="100"
   android:title="@string/action_logout"
   app:showAsAction="never" />
</menu>
```

Прекрасно! Мы уже находимся на целый этап ближе к завершению этого проекта. Настало время выполнить обработку пользовательского чата. Функции `showConversationLoadError()` и `showMessageSendError()` представляют интерфейс, где реально проходит чат.

Создание пользовательского интерфейса чата

Пользовательский интерфейс чата, который мы собираемся создать, должен отображать поток сообщений для активной беседы, а также разрешать пользователю направлять новое сообщение тому, с кем поддерживается беседа. Начнем этот раздел с создания макета представления, которое отображается для пользователя.

Создание макета чата

Для создания макета представления чата мы воспользуемся библиотекой Android с открытым исходным кодом `ChatKit`. Эта библиотека предоставляет гибкие компоненты (виджеты) для реализации интерфейса пользовательского чата в проектах Android, а также утилиты для управления данными и пользовательским интерфейсом чата.

Библиотека `ChatKit` добавляется в проект Messenger при помощи следующей строки кода в сценарии build.gradle:

```
implementation 'com.github.stfalcon:chatkit:0.2.2'
```

Как упоминалось, библиотека `ChatKit` предоставляет ряд полезных виджетов пользовательского интерфейса, в том числе виджеты `MessagesList` и `MessageInput`. Виджет `MessagesList` предназначен для отображения и управления сообщениями (management of messages) в потоках бесед. Виджет `MessageInput` предоставляет возможность ввода текстовых сообщений. В дополнение к поддержке нескольких стилистических опций, `MessageInput` поддерживает простые процессы проверки вводимых данных.

Рассмотрим, каким образом можно применять виджеты `MessagesList` и `MessageInput` в файле макета. Создайте новый пакет chat в com.example.messenger.ui и добавьте к нему новое пустое действие под названием `ChatActivity`. Откройте файл activity_chat.xml макета активностей `ChatActivity` и включите в него следующий XML-код:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
tools:context="com.example.messenger.ui.chat.ChatActivity">
```

```
<com.stfalcon.chatkit.messages.MessagesList
   android:id="@+id/messages_list"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_above="@+id/message_input"/>
<com.stfalcon.chatkit.messages.MessageInput
   android:id="@+id/message_input"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_alignParentBottom="true"
   app:inputHint="@string/hint_enter_a_message" />
</RelativeLayout>
```

Как можно видеть, в приведенном XML-коде применяются виджеты интерфейсов `MessagesList` и `MessageInput`, основанные на `ChatKit`, так же как и любые другие Android-виджеты. Оба виджета: `MessagesList` и `MessageInput` — локализованы в пакете com.stfalcon.chatkit.messages. Откройте окно дизайна макета, чтобы увидеть этот макет.

Следующим пунктом в нашей повестке дня является создание класса `ChatView`:

```
package com.example.messenger.ui.chat

import com.example.messenger.ui.base.BaseView
import com.example.messenger.utils.message.Message
import com.stfalcon.chatkit.messages.MessagesListAdapter
interface ChatView : BaseView {
   interface ChatAdapter {
   fun navigateToChat(recipientName: String, recipientId: Long,
   conversationId: Long? = null)
}

   fun showConversationLoadError()
   fun showMessageSendError()
   fun getMessageListAdapter(): MessagesListAdapter<Message>
}
```

В представлении `ChatView` мы определили интерфейс `ChatAdapter`, объявляя единственную функцию `navigateToChat(String, Long, Long)`. Этот интерфейс должен реализовываться с помощью адаптеров, которые могут направить пользователя к представлению `ChatView`. Оба созданных ранее адаптера: `ConversationsAdapter` и `ContactsAdapter` — обеспечивают эту возможность.

Функции `showConversationLoadError()` и `showMessageSendError()` при реализации должны отображать сообщения об ошибках, если при загрузке соответственно беседы или сообщения происходят сбои.

Виджет интерфейса пользователя `MessagesList`, основанный на `ChatKit`, должен располагать `MessagesListAdapter` для управления его набором данных сообщений. Функция `getMessageListAdapter()` при реализации с помощью `ChatView` возвращает `MessagesListAdapter` для `MessagesList` в интерфейсе пользователя.

Подготовка моделей чата для интерфейса пользователя

Для добавления сообщений к `MessagesListAdapter` для `MessageList` необходимо в соответствующей модели реализовать ChatKit-интерфейс `IMessage`. Реализуем эту модель здесь. Создадим пакет com.example.messenger.utils.message и добавим в него соответствующий класс `Message`:

```
package com.example.messenger.utils.message
import com.stfalcon.chatkit.commons.models.IMessage
import com.stfalcon.chatkit.commons.models.IUser
import java.util.*
data class Message(private val authorId: Long, private val body: String,
   private val createdAt: Date) : IMessage {
   override fun getId(): String {
      return authorId.toString()
   }

   override fun getCreatedAt(): Date {
      return createdAt
   }

   override fun getUser(): IUser {
      return Author(authorId, "")
   }

   override fun getText(): String {
      return body
   }
}
```

Кроме того, следует создать класс `Author`, реализующий ChatKit-интерфейс `IUser`. Далее приводится реализация для этого класса:

```
package com.example.messenger.utils.message

import com.stfalcon.chatkit.commons.models.IUser
data class Author(val id: Long, val username: String) : IUser {

   override fun getAvatar(): String? {
      return null
   }
```

```
    override fun getName(): String {
        return username
    }

    override fun getId(): String {
        return id.toString()
    }
}
```

Класс `Author` моделирует подробные сведения об авторе сообщения — такие как имя автора, его ID, аватар (если имеется).

К этому моменту мы достаточно серьезно обработали представления и макеты. Далее приступим к реализации интерактора `ChatInteractor` и презентатора `ChatPresenter`.

Создание интерактора *ChatInteractor* и презентатора *ChatPresenter*

Понимая, в чем состоят функции презентаторов и интеракторов, перейдем непосредственно к созданию кода. Далее приводится код интерфейса `ChatInteractor`. Этот интерфейс и все остальные файлы группы Chat относятся к пакету com.example.messenger.ui.chat:

```
package com.example.messenger.ui.chat

import com.example.messenger.data.vo.ConversationVO
interface ChatInteractor {
    interface OnMessageSendFinishedListener {
        fun onSendSuccess()
        fun onSendError()
    }
interface onMessageLoadFinishedListener {
    fun onLoadSuccess(conversationVO: ConversationVO)
    fun onLoadError()
}
fun sendMessage(recipientId: Long, message: String, listener:
    OnMessageSendFinishedListener)
    fun loadMessages(conversationId: Long, listener:
    onMessageLoadFinishedListener)
}
```

Далее приводится соответствующий класс `ChatInteractorImpl` для интерфейса `ChatInteractor`:

```
package com.example.messenger.ui.chat

import android.content.Context
import com.example.messenger.data.local.AppPreferences
```

```
import com.example.messenger.data.remote.repository.ConversationRepository
import com.example.messenger.data.remote.repository.ConversationRepositoryImpl
import com.example.messenger.data.remote.request.MessageRequestObject
import com.example.messenger.service.MessengerApiService
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers

class ChatInteractorImpl(context: Context) : ChatInteractor {
private val preferences: AppPreferences = AppPreferences.create(context)
private val service: MessengerApiService = MessengerApiService.getInstance()
private val conversationsRepository: ConversationRepository =
                                                    ConversationRepositoryImpl(context)
```

Метод, приведенный далее, вызывается для загрузки сообщения из потока бесед:

```
override fun loadMessages(conversationId: Long, listener:
ChatInteractor.onMessageLoadFinishedListener) {

   conversationsRepository.findConversationById(conversationId)
   .subscribeOn(Schedulers.io())
   .observeOn(AndroidSchedulers.mainThread())
   .subscribe({ res -> listener.onLoadSuccess(res)},
{ error ->
   listener.onLoadError()
   error.printStackTrace()})
}
```

Нижеприведенный метод вызывается для пересылки пользователю сообщения:

```
   override fun sendMessage(recipientId: Long, message: String,
   listener: ChatInteractor.OnMessageSendFinishedListener) {
      service.createMessage(MessageRequestObject(
      recipientId, message), preferences.accessToken as String)
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe({ _ -> listener.onSendSuccess()},
   { error ->
      listener.onSendError()
      error.printStackTrace()})
   }
}
```

Теперь обработаем код `ChatPresenter` и `ChatPresenterImpl`. Для `ChatPresenter` необходимо создать интерфейс, передающий объявление для двух функций: `sendMessage(Long, String)` и `loadMessages(Long)`. Далее приводится код интерфейса `ChatPresenter`:

```
package com.example.messenger.ui.chat

interface ChatPresenter {
```

```
    fun sendMessage(recipientId: Long, message: String)
    fun loadMessages(conversationId: Long)

}
```

Класс реализации интерфейса `ChatPresenter` **имеет следующий вид:**

```
package com.iyanuadelekan.messenger.ui.chat

import android.widget.Toast
import com.iyanuadelekan.messenger.data.vo.ConversationVO
import com.iyanuadelekan.messenger.utils.message.Message
import java.text.SimpleDateFormat

class ChatPresenterImpl(val view: ChatView) : ChatPresenter,
        ChatInteractor.OnMessageSendFinishedListener,
        ChatInteractor.onMessageLoadFinishedListener {

    private val interactor: ChatInteractor = ChatInteractorImpl
                                             (view.getContext())

    override fun onLoadSuccess(conversationVO: ConversationVO) {
        val adapter = view.getMessageListAdapter()

        // Создание форматировщика данных для форматирования дат createdAt
        // Получено от Messenger API
        val dateFormatter = SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
```

Обратимся к загруженному из API сообщению из беседы, создадим новый объект `IMessage` **для сообщения, которое в настоящее время повторяется, и добавим** `IMessage` **в начало** `MessagesListAdapter`:

```
        conversationVO.messages.forEach { message ->
        adapter.addToStart(Message(message.senderId, message.body,
        dateFormatter.parse(message.createdAt.split(".")[0])), true)
    }
}

override fun onLoadError() {
    view.showConversationLoadError()
}

override fun onSendSuccess() {
    Toast.makeText(view.getContext(), "Message sent",
    Toast.LENGTH_LONG).show()
}

override fun onSendError() {
    view.showMessageSendError()
}
```

```
override fun sendMessage(recipientId: Long, message: String) {
   interactor.sendMessage(recipientId, message,this)
}

override fun loadMessages(conversationId: Long) {
   interactor.loadMessages(conversationId, this)
   }
}
```

Напомним, что комментарии, сопровождающие приведенные фрагменты кода в файлах примеров исходного кода, размещенных в прилагаемом к книге файловом архиве, призваны помочь вам разобраться, что здесь к чему.

И, наконец, обработаем действие `ChatActivity`. Прежде всего начнем с объявления необходимых свойств для этого действия и выполним обработку его метода жизненного цикла `onCreate()`.

Модифицируем действие `ChatActivity` для включения следующего кода:

```
package com.example.messenger.ui.chat

import android.content.Context
import android.content.Intent
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.MenuItem
import android.widget.Toast
import com.example.messenger.R
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.ui.main.MainActivity
import com.example.messenger.utils.message.Message
import com.stfalcon.chatkit.messages.MessageInput
import com.stfalcon.chatkit.messages.MessagesList
import com.stfalcon.chatkit.messages.MessagesListAdapter
import java.util.*

class ChatActivity : AppCompatActivity(), ChatView,
MessageInput.InputListener {

   private var recipientId: Long = -1
   private lateinit var messageList: MessagesList
   private lateinit var messageInput: MessageInput
   private lateinit var preferences: AppPreferences
   private lateinit var presenter: ChatPresenter
   private lateinit var messageListAdapter: MessagesListAdapter<Message>

   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_chat)
```

```
        supportActionBar?.setDisplayHomeAsUpEnabled(true)
        supportActionBar?.title = intent.getStringExtra("RECIPIENT_NAME")

        preferences = AppPreferences.create(this)
        messageListAdapter = MessagesListAdapter(
        preferences.userDetails.id.toString(), null)
        presenter = ChatPresenterImpl(this)
        bindViews()
```

Проанализируем пакет дополнений от намерения, которое запустило действие `ChatActivity`. Если отсутствует дополнение, идентифицированное ключами `CONVERSATION_ID` и `RECIPIENT_ID`, по умолчанию возвращается значение, равное `-1`:

```
val conversationId = intent.getLongExtra("CONVERSATION_ID", -1)
recipientId = intent.getLongExtra("RECIPIENT_ID", -1)
```

Если `conversationId` не равно `-1`, тогда `conversationId` действительно, поэтому загрузите сообщения в беседу:

```
        if (conversationId != -1L) {
            presenter.loadMessages(conversationId)
        }
    }
}
```

В приведенном коде мы создали свойства `recipientId`, `messageList`, `messageInput`, `preferences`, `presenter` и `messageListAdapter`, относящиеся к типам `Long`, `MessageList`, `MessageInput`, `AppPreferences`, `ChatPresenter` и `MessageListAdapter` соответственно. Свойство `messageList` служит представлением, отображающим различные представления для сообщений, предоставленных ему со стороны `messageListAdapter`. Вся логика, содержащаяся в `onCreate()`, имеет отношение к инициализации представлений в действии. Код в `onCreate()` содержит комментарии для поддержки полного представления о происходящих изменениях — внимательно изучите каждую строку кода перед выполнением процедур. Действие `ChatActivity` реализует `MessageInput.InputListener`. Классы, реализующие этот интерфейс, должны поддерживать соответствующий метод `onSubmit()`, поэтому далее мы добавим данный метод в действие `ChatActivity`.

Функция, переопределенная из `MessageInput.InputListener`, вызывается, если пользователь отправляет сообщение с помощью виджета `MessageInput`:

```
override fun onSubmit(input: CharSequence?): Boolean {
    // Создание нового объекта Message и его добавление
    // в начало MessagesListAdapter
    messageListAdapter.addToStart(Message(
        preferences.userDetails.id, input.toString(), Date()), true)
        // Начало сообщения, отсылаемого процедурой с ChatPresenter
    presenter.sendMessage(recipientId, input.toString())
    return true
}
```

Метод `onSubmit()` воспринимает `CharSequence` этого сообщения, направленного `MessageInput`, и создает для него соответствующий экземпляр `Message`. Этот экземпляр затем добавляется в начало `MessageList`, вызывая метод `messageListAdapter.addToStart()` с экземпляром `Message`, переданным в качестве аргумента. После добавления созданного `Message` к `MessageList` применяется экземпляр `ChatPresenter` для инициализации процедуры пересылки серверу.

Обработаем другие переопределения методов, которые необходимо выполнить. Добавим к `ChatActivity` показанные далее методы `showConversationLoadError()`, `showMessageSendError()`, `getContext()` и `getMessageListAdapter()`:

```
override fun showConversationLoadError() {
   Toast.makeText(this, "Unable to load thread.
   Please try again later.",
   Toast.LENGTH_LONG).show()
}

override fun showMessageSendError() {
   Toast.makeText(this, "Unable to send message.
   Please try again later.",
   Toast.LENGTH_LONG).show()
}

override fun getContext(): Context {
   return this
}

override fun getMessageListAdapter(): MessagesListAdapter<Message> {
   return messageListAdapter
}
```

И наконец, переопределим методы `bindViews()`, `onOptionsItemSelected()` и `onBackPressed()` следующим образом:

```
override fun bindViews() {
   messageList = findViewById(R.id.messages_list)
   messageInput = findViewById(R.id.message_input)
   messageList.setAdapter(messageListAdapter)
   messageInput.setInputListener(this)
}

override fun onOptionsItemSelected(item: MenuItem?): Boolean {
   if (item?.itemId == android.R.id.home) {
      onBackPressed()
   }
   return super.onOptionsItemSelected(item)
}
```

```
override fun onBackPressed() {
   super.onBackPressed()
   finish()
}
```

Все идет нормально! Мы успешно создали бóльшую часть приложения Messenger, что, несомненно, заслуживает похвалы. Единственное, что остается сделать перед завершением этой главы, — создать действие по настройке приложения, благодаря которому пользователи смогут обновлять статусы своего профиля. Сделайте заслуженный перерыв на кофе, прежде чем перейти к следующему разделу.

Создание действия по настройке приложения

Настало время разработать простое действие по настройке приложения, с помощью которого пользователь сможет обновить состояние своего профиля. Организуйте новый пакет settings в составе пакета com.example.messenger.ui. В этом пакете создайте новое действие по настройке. Назовите это действие `SettingsActivity`. Для создания действия по настройке щелкните правой кнопкой мыши на пакете settings и выберите команды **New | Activity | Settings Activity** (Создать | Действие | Действие по настройке). Введите необходимые данные для нового действия по настройке, такие как название действия и его заголовок, а затем щелкните на кнопке **Finish** (Готово).

В процессе создания нового действия `SettingsActivity` Android Studio добавит к проекту ряд дополнительных файлов. Кроме того, к проекту будет добавлен новый каталог ресурсов app/res/xml, включающий следующие файлы:

- pref_data_sync.xml;
- pref_general.xml;
- pref_headers.xml;
- pref_notification.xml.

Файлы pref_notification.xml и pref_data_sync.xml вы можете удалить — они не используются в нашем проекте. Рассмотрим файл pref_general.xml:

```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">

   <SwitchPreference
   android:defaultValue="true"
   android:key="example_switch"
   android:summary=
   "@string/pref_description_social_recommendations"
   android:title="@string/pref_title_social_recommendations" />
```

```
<!-- ПРИМЕЧАНИЕ: EditTextPreference принимает атрибуты EditText. -->
<!-- ПРИМЕЧАНИЕ Сводка EditTextPreference должна быть установлена в ее значение
с помощью кода активности. -->
<EditTextPreference
   android:capitalize="words"
   android:defaultValue="@string/pref_default_display_name"
   android:inputType="textCapWords"
   android:key="example_text"
   android:maxLines="1"
   android:selectAllOnFocus="true"
   android:singleLine="true"
   android:title="@string/pref_title_display_name" />

<!-- ПРИМЕЧАНИЕ: Скрыть кнопки, чтобы упростить интерфейс. Пользователи могут
коснуться экрана за пределами диалогового окна,
чтобы закрыть его. -->
<!-- ПРИМЕЧАНИЕ: Сводке ListPreference должно быть присвоено значение
соответствующим кодом действия. -->
<ListPreference
   android:defaultValue="-1"
   android:entries="@array/pref_example_list_titles"
   android:entryValues="@array/pref_example_list_values"
   android:key="example_list"
   android:negativeButtonText="@null"
   android:positiveButtonText="@null"
   android:title="@string/pref_title_add_friends_to_messages" />

</PreferenceScreen>
```

Корневым представлением этого XML-файла макета является `PreferenceScreen`. Ну а `PreferenceScreen` является корнем иерархии `Preference`. Класс `PreferenceScreen` находится на верхнем уровне `Preference`. Слово `Preference` использовано здесь несколько раз. Определим, что оно всё же значит. Это представление для основного пользовательского интерфейсного строительного блока `Preference`, который отображается в виде списка с помощью действия `PreferenceActivity`.

Класс `Preference` поддерживает соответствующее представление для настройки, отображаемой в `PreferenceActivity`, и ассоциируется с классом `SharedPreferences` для сохранения и поиска данных об установках. Классы `SwitchPreference`, `EditTextPreference` и `ListPreference` в предыдущем фрагменте кода представляют собой подклассы класса `DialogPreference`, который, в свою очередь, является подклассом класса `Preference`. Класс `PreferenceActivity` — это основной класс, который необходим для действия с целью отображения пользователю иерархии настроек.

Представления `SwitchPreference`, `EditTextPreference` и `ListPreference` в файле pref_general.xml не нужны. Удалите их прямо сейчас из этого XML-файла. Необхо-

дима установка, которая разрешает пользователю обновлять на платформе Messenger свое состояние. Это весьма специфичный вариант использования, и поэтому неудивительно, что отсутствует виджет настроек, предоставляющий эту возможность. Не беспокойтесь! Далее мы реализуем пользовательскую установку, пригодную для выполнения этой задачи. Назовем ее `ProfileStatusPreference`. И создадим новый класс `ProfileStatusPreference`, в пакет settings которого войдет следующий код:

```
package com.example.messenger.ui.settings

import android.content.Context
import android.preference.EditTextPreference
import android.text.TextUtils
import android.util.AttributeSet
import android.widget.Toast
import com.example.messenger.data.local.AppPreferences
import com.example.messenger.data.remote.request.
   StatusUpdateRequestObject
import com.example.messenger.service.MessengerApiService
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers

class ProfileStatusPreference(context: Context, attributeSet: AttributeSet)
: EditTextPreference(context, attributeSet) {

   private val service: MessengerApiService = MessengerApiService
                                              .getInstance()
   private val preferences: AppPreferences = AppPreferences
                                             .create(context)

   override fun onDialogClosed(positiveResult: Boolean) {
      if (positiveResult) {
```

В следующем фрагменте `EditText` из `ProfileStatusPreference` связывается с переменной `etStatus`:

```
         val etStatus = editText
         if (TextUtils.isEmpty(etStatus.text)) {
            // Отображение сообщения об ошибке, если пользователь
            // пытается передать пустой статус
            Toast.makeText(context, "Status cannot be empty.",
               Toast.LENGTH_LONG).show()
         } else {
            val requestObject =
            StatusUpdateRequestObject(etStatus.text.toString())
```

Применим MessengerApiService для обновления статуса пользователя:

```
            service.updateUserStatus(requestObject,
            preferences.accessToken as String)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({ res ->
```

Теперь, сохраним обновленные данные пользователя, если обновление статуса прошло успешно:

```
            preferences.storeUserDetails(res) },
            { error ->
            Toast.makeText(context, "Unable to update status at the " +
               "moment. Try again later.", Toast.LENGTH_LONG).show()
               error.printStackTrace()})

            }
        }
        super.onDialogClosed(positiveResult)
    }
}
```

Класс ProfileStatusPreference расширяет класс EditTextPreference, который является дочерним для Preference, и разрешает ввод строки в EditText. EditTextPreference также является дочерним для DialogPreference и представляет диалоговое окно для пользователя, включающее представление Preference при щелчке на нем. Если диалоговое окно DialogPreference закрыто, вызывается метод onDialogClosed(Boolean). Положительное значение аргумента Boolean — истина (true) — передается методом onDialogClosed(), если диалог закрывается с положительным результатом. Ложь (false) передается методом onDialogClosed(), если диалоговое окно закрывается с отрицательным результатом, например при щелчке на кнопке отмены отображения диалогового окна.

Класс ProfileStatusPreference переопределяет функцию onDialogClosed() из EditTextPreference. Если диалоговое окно закрыто с положительным результатом, проверяется действительность статуса, содержащегося в функции EditText из ProfileStatusPreference. Если сообщение о статусе действительно, статус обновляется с помощью API, в противном случае отображается сообщение об ошибке.

После создания ProfileStatusPreference вернитесь к файлу pref_general.xml и обновите его для отражения XML-кода в следующем фрагменте:

```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">
   <com.example.messenger.ui.settings.ProfileStatusPreference
      android:key="profile_status"
      android:singleLine="true"
      android:inputType="text"
```

```
        android:maxLines="1"
        android:selectAllOnFocus="true"
        android:title="Profile status"
        android:defaultValue="Available"
        android:summary="Set profile status (visible to contacts)."/>
</PreferenceScreen>
```

Как показано в приведенном коде, в фрагменте кода использовалось `ProfileStatusPreference`, как и другие настройки, связанные с фреймворком Android-приложения.

Пойдем далее и проверим файл pref_headers.xml:

```
<preference-headers
xmlns:android="http://schemas.android.com/apk/res/android">
   <!-- Эти заголовки настроек используются только на планшетах. -->
   <header
      android:fragment=
      "com.example.messenger.ui.settings.SettingsActivity
      $GeneralPreferenceFragment"
      android:icon="@drawable/ic_info_black_24dp"
      android:title="@string/pref_header_general" />
   <header
      android:fragment=
      "com.example.messenger.ui.settings.SettingsActivity
      $NotificationPreferenceFragment"
      android:icon="@drawable/ic_notifications_black_24dp"
      android:title="@string/pref_header_notifications" />
   <header
      android:fragment=
      "com.example.messenger.ui.settings.SettingsActivity
      $DataSyncPreferenceFragment"
      android:icon="@drawable/ic_sync_black_24dp"
      android:title="@string/pref_header_data_sync" />
</preference-headers>
```

Этот файл заголовка настроек определяет заголовки для различных настроек в `SettingsActivity`. Измените файл, чтобы он содержал следующий код:

```
<preference-headers
xmlns:android="http://schemas.android.com/apk/res/android">
<header
   android:fragment=
   "com.example.messenger.ui.settings.SettingsActivity
   $GeneralPreferenceFragment"
   android:icon="@drawable/ic_info_black_24dp"
   android:title="@string/pref_header_account" />
</preference-headers>
```

Прекрасно! Приступим к обработке действия `SettingsActivity`. Модифицируйте тело `SettingsActivity` для включения содержимого, показанного в следующем блоке кода:

```
package com.example.messenger.ui.settings

import android.content.Intent
import android.os.Bundle
import android.preference.PreferenceActivity
import android.preference.PreferenceFragment
import android.view.MenuItem
import android.support.v4.app.NavUtils
import com.example.messenger.R
```

Действие `PreferenceActivity` представляет набор настроек приложения:

```
class SettingsActivity : AppCompatPreferenceActivity() {

   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      supportActionBar?.setDisplayHomeAsUpEnabled(true)
   }

   override fun onMenuItemSelected(featureId: Int, item: MenuItem):
   Boolean {
      val id = item.itemId
      if (id == android.R.id.home) {
         if (!super.onMenuItemSelected(featureId, item)) {
            NavUtils.navigateUpFromSameTask(this)
         }
         return true
      }
      return super.onMenuItemSelected(featureId, item)
   }
```

Следующая функция `onBuildHeaders()` вызывается, если действию необходим список заголовков сборки:

```
override fun onBuildHeaders(target: List<PreferenceActivity.Header>)
{
   loadHeadersFromResource(R.xml.pref_headers, target)
}
```

Приведенный далее метод, который предотвращает внедрение фрагментов из вредоносных программ и всех неизвестных фрагментов, должен быть здесь запрещен:

```
override fun isValidFragment(fragmentName: String): Boolean {
   return PreferenceFragment::class.java.name == fragmentName
   || GeneralPreferenceFragment::class.java.name == fragmentName
}
```

Приведенный далее фрагмент показывает общие настройки:

```
class GeneralPreferenceFragment : PreferenceFragment() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        addPreferencesFromResource(R.xml.pref_general)
        setHasOptionsMenu(true)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val id = item.itemId
        if (id == android.R.id.home) {
            startActivity(Intent(activity, SettingsActivity::class.java))
            return true
        }
        return super.onOptionsItemSelected(item)
    }
}
}
```

Действие `SettingsActivity` расширяет `AppCompatPreferenceActivity` — действие, реализующее необходимые вызовы для использования с `AppCompat`. Действие `SettingsActivity` является действием по настройке `PreferenceActivity`, представляющем набор настроек приложения. Функция `onBuildHeaders()` из `SettingsActivity` вызывается, если действию необходимо располагать встроенным перечнем заголовков. Функция `ValidFragment()` предотвращает внедрение фрагментов вредоносными программами в `SettingsActivity`. `ValidFragment()` возвращает значение `true` (истина), если фрагмент является действительным, и `false` (ложь) — в противном случае.

В действии `SettingsActivity` определим класс `GeneralPreferenceFragment`. Класс `GeneralPreferenceFragment` расширяет фрагмент `PreferenceFragment`. Фрагмент `PreferenceFragment` является абстрактным классом, определенным в фреймворке Android-приложения, и отображает иерархию экземпляров `Preference` в виде списков.

Настройки из файла pref_general.xml добавлены в `GeneralPreferenceFragment` в методе `onCreate()` путем вызова функции `addPreferencesFromResource(R.xml.pref_general)`.

С этими изменениями, выполненными в действии `SettingsActivity`, можно полагать, что обработка настроек для приложения Messenger успешно завершена, и можно перейти к выполнению приложения Messenger.

* * *

Скомпонуйте и запустите приложение Messenger на устройстве (виртуальном или физическом). После запуска приложения вас перенаправят прямо к действию `LoginActivity` — откроется окно входа в приложение (рис. 6.2).

Первое, что следует сделать, — зарегистрировать нового пользователя на платформе Messenger. Выполним это с помощью действия `SignUpActivity`: щелкните в окне, показанном на рис. 6.2, на кнопке **DON'T HAVE AN ACCOUNT? SIGN UP!** (Не имеете учетной записи? Создайте ее!), и вас перенаправят к действию `SignUpActivity` — окну регистрации нового пользователя (рис. 6.3).

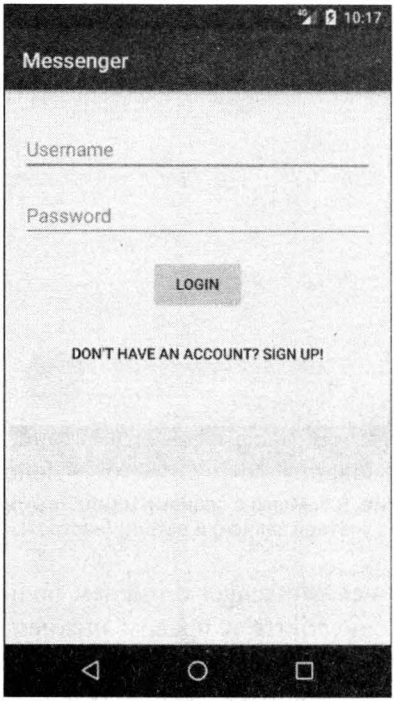


Рис. 6.2. Экран входа в приложение (действие `LoginActivity`)

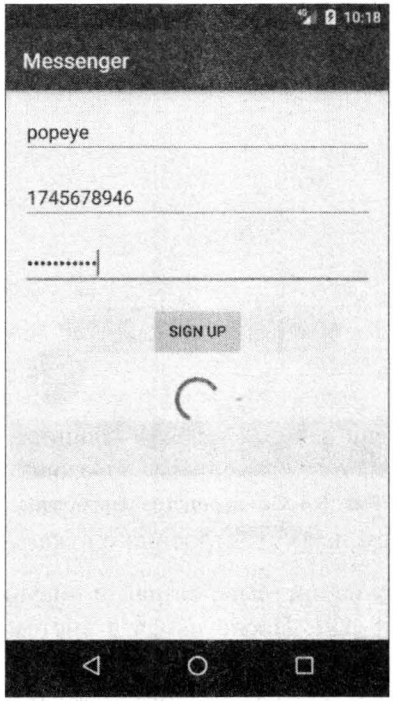


Рис. 6.3. Окно регистрации нового пользователя (действие `SignUpActivity`)

Создайте новую учетную запись пользователя в этом действии: введите `popeye` в качестве имени пользователя, а также номер телефона и пароль, затем щелкните на кнопке **SIGN UP** (Создать учетную запись). Будет создана учетная запись нового пользователя на платформе Messenger с именем пользователя `popeye`. После завершения регистрации вы будете перенаправлены к действию `MainActivity`, в котором отображается представление бесед (рис. 6.4).

Поскольку у вновь зарегистрированного пользователя нет активных бесед, в этом окне отображается соответствующее всплывающее сообщение. И это неудивительно — ведь ему не с кем беседовать, он там один... Чтобы продемонстрировать функциональность чата, необходимо создать на платформе нашего мессенджера учетную запись еще одного пользователя. Выйдите из учетной записи `popeye`, щелкнув на меню в виде трех точек в правом верхнем углу экрана и выбрав опцию **logout** (рис. 6.5).

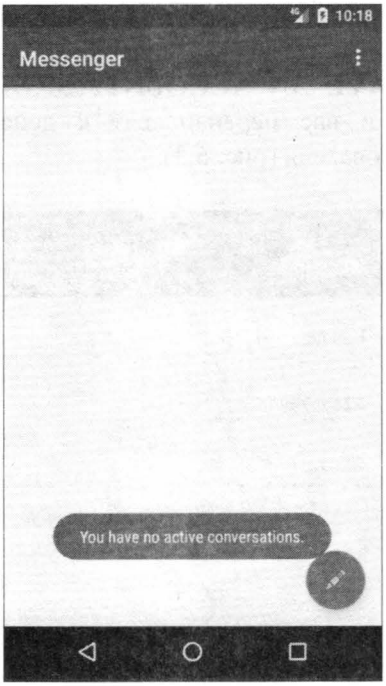


Рис. 6.4. Окно представления бесед (действие `MainActivity`)

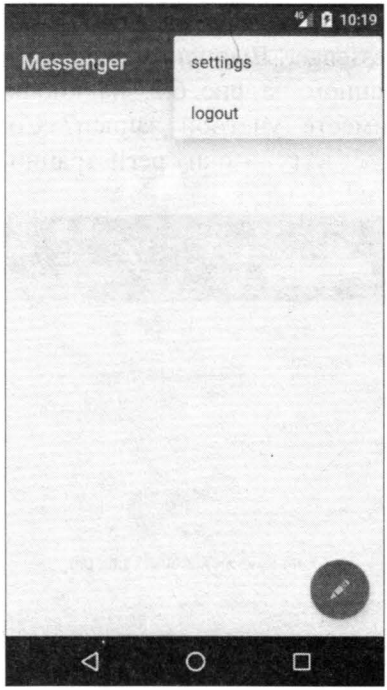


Рис. 6.5. Меню с опциями **logout** (выход из учетной записи) и **setting** (настройки)

Выйдя из системы, создайте новую учетную запись Messenger с именем пользователя `dexter`. После входа в систему как `dexter`, щелкните в правом нижнем углу представления бесед на кнопке с плавающим действием (кружок с изображением карандаша) для создания нового сообщения — отобразится экран представления контактов, в котором уже будет присутствовать контакт **popeye**.

По щелчку на контакте **popeye** откроется окно для создания сообщения (`ChatActivity`). Направим сообщение контакту **popeye** — наберем в расположенном над клавиатурой поле ввода фразу, например: `Hey, Popeye` (Эй, Попай!), — и щелкнем на кнопке отправки . Сообщение немедленно отправится пользователю **popeye** (рис. 6.6).

Вернувшись в окно представления бесед (`MainActivity`), заметим, что в нем имеется сообщение, отправленное пользователю **popeye** (рис. 6.7).

Проверим, действительно ли сообщение доставлено пользователю **popeye**. Выйдите из платформы Messenger, где вы сейчас зарегистрированы как **dexter**, и войдите как **popeye**. После входа в систему вы увидите сообщение, отправленное вам пользователем **dexter** (рис. 6.8).

Невероятно! Сообщение доставлено! Ответим пользователю **dexter**: откройте беседу и направьте **dexter** сообщение (рис. 6.9).

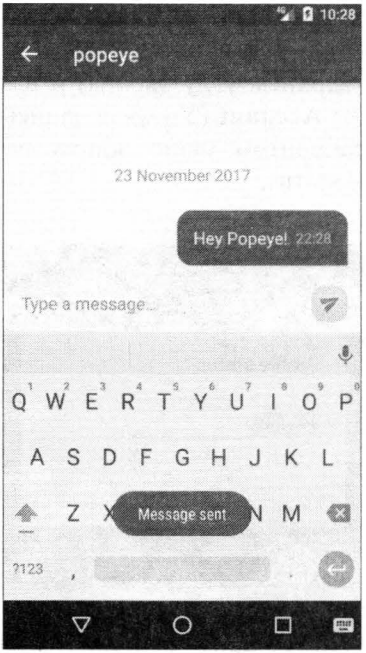


Рис. 6.6. Отправка сообщения

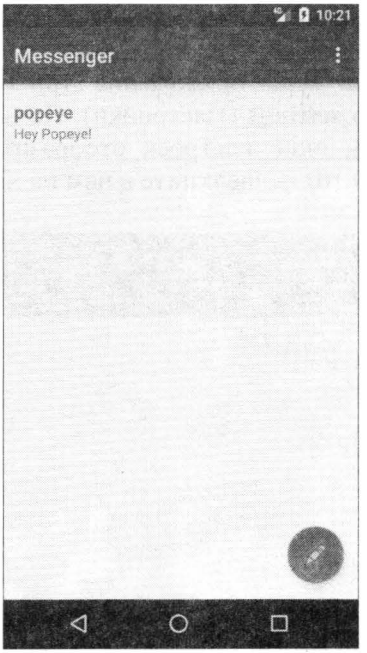


Рис. 6.7. Беседа с пользователем **popeye** в окне пользователя **dexter**

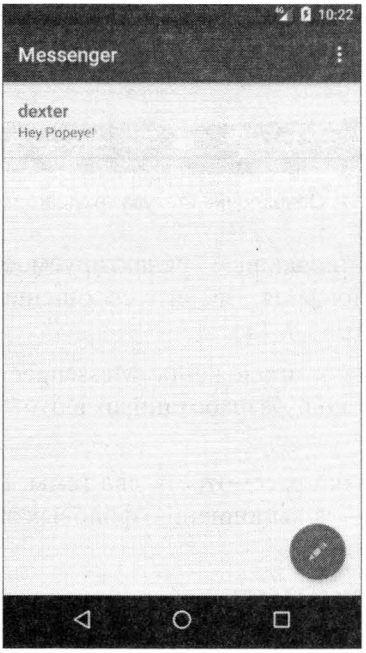


Рис. 6.8. Сообщение, отправленное пользователем **dexter**, доставлено пользователю **popeye**

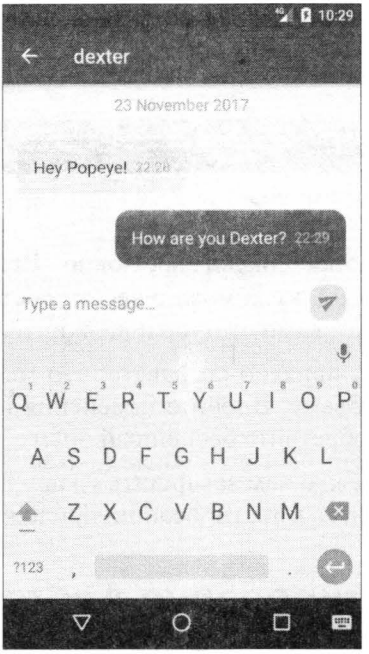


Рис. 6.9. Отвечаем пользователю **dexter** сообщением: **How are you Dexter?** (Как ты, Декстер?)

Пришло время обновить статус профиля пользователя **popeye**. Вернитесь к окну основного действия (см. рис. 6.5) и получите доступ к заданию настроек — щелкните на меню приложения (три точки в правом верхнем углу экрана) и выберите опцию **settings** (Настройки). По щелчку на кнопке **Account** (Учетная запись) в активном окне настроек отобразится экран с фрагментом меню общих настроек (рис. 6.10) — щелкните в нем на настройке **Profile status**.

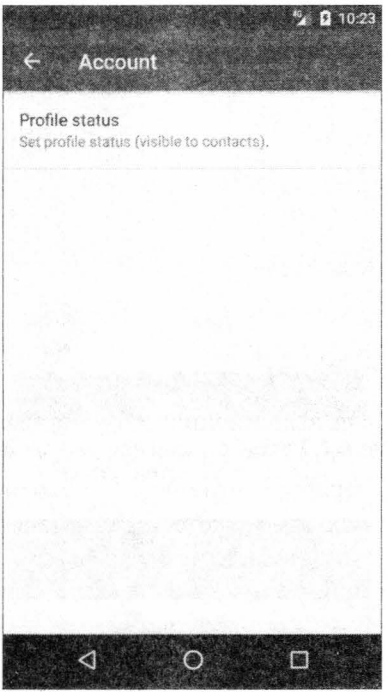


Рис. 6.10. Фрагмент меню общих настроек

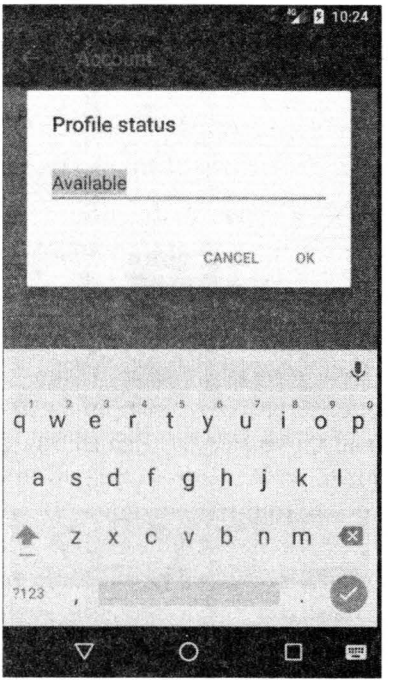


Рис. 6.11. Обновление статуса текущего профиля

Откроется диалоговое окно **Profile status**, включающее редактируемое поле (`EditText`), куда можно ввести состояние нового профиля. Введите сообщение о выбранном вами статусе и щелкните на кнопке **OK** (рис. 6.11).

Что ж, приятно сознавать, что мы успешно создали приложение Messenger в полном объеме. Вносите изменения и дополнения в код, разработанный в этой главе, и приобретайте бесценный опыт.

Но прежде чем завершить главу, необходимо кратко рассмотреть две темы. Первая относится к тестированию приложения, а вторая — к выполнению фоновых задач.

Тестирование Android-приложения

Тестирование приложения — это процесс, с помощью которого разработанное программное обеспечение проверяется с точки зрения качества его создания. На качество программного обеспечения влияет ряд факторов. К таким факторам относятся

удобство использования, функциональность, надежность и согласованность приложений. Тестирование приложения для Android может принести несколько положительных моментов, в том числе:

- обнаружение сбоев;
- повышение стабильности программного обеспечения.

Интегральных тестов приложений для Android имеется множество, однако рассмотрение их выходит за рамки этой книги. Тем не менее, далее приводится список ресурсов по тестированию Android-приложений, которые вы сможете (а вероятно, обязаны) изучить в свободное время:

- Espresso: **https://developer.android.com/training/testing/espresso/index.html**;
- Roboelectric: **http://robolectric.org**;
- Mockito: **http://site.mockito.org**;
- Calabash: **https://github.com/calabash/calabash-android**.

Выполнение фоновых операций

В процессе разработки приложения Messenger для выполнения асинхронных операций широко использовалась библиотека RxAndroid. Во многих случаях при использовании RxAndroid результаты фоновых операций наблюдались в основном потоке Android-приложения. В некоторых случаях вы можете не применять для этого стороннюю библиотеку, например ту же RxAndroid. Вместо этого можно использовать решение, встроенное в платформу приложений Android. Android предоставляет для этого ряд возможностей. Одним из таких вариантов является AsyncTask.

AsyncTask

Класс `AsyncTask` позволяет выполнять фоновые операции и публиковать результаты операций в потоке пользовательского интерфейса приложения без нагрузки на управление обработчиками и потоками. `AsyncTask` лучше всего применять в ситуациях, если необходимо выполнять короткие операции (short operations). Расчеты `AsyncTask` выполняются в фоновом потоке, их результаты публикуются в потоке пользовательского интерфейса (UI thread). Больше об `AsyncTask` можно узнать на сайте, доступном по адресу: **https://developer.android.com/reference/android/os/AsyncTask.html**.

IntentService

Сервис `IntentService` также является хорошим кандидатом для выполнения запланированных операций, выполняемых в фоновом режиме, независимо от действия. Как сказано в справочнике разработчика Android: *`IntentService` является базовым*

классом для служб, которые по требованию обрабатывают асинхронные запросы (выраженные как намерения). Клиенты направляют запросы через вызовы `startService` *(намерение); сервис запускается по мере необходимости и обрабатывает по очереди каждое намерение, используя рабочий поток, и останавливается, когда завершается его работа.* Узнать больше о `IntentService` можно на сайте по адресу: **https://developer.android.com/reference/android/app/IntentService.html**.

Подведем итоги

В этой главе завершена разработка приложения Messenger для Android. В процессе изучения показано, как использовать `ChatKit` — стороннюю библиотеку для создания красивых пользовательских интерфейсов чата. В дополнение к этому рассмотрены утилиты, предлагаемые платформой Android-приложений. Из первых рук мы получили информацию о разработке действия по настройкам в Android, которое помогло узнать о `PreferenceScreen`, `PreferenceActivity`, `DialogPreference`, `Preference` и `PreferenceFragment`. Наконец, кратко обсуждалось тестирование приложений Android и выполнение фоновых операций.

В следующей главе рассматриваются различные варианты сохранения данных, предоставляемые фреймворком Android-приложений.

7

Средства хранения данных

В предыдущей главе были затронуты такие важные темы, как использование сторонних библиотек и тестирование приложений Android, речь также шла о способах запуска на платформе Android фоновых задач. Эта глава посвящена вопросам хранения данных. До сих пор постоянные данные приложения сохранялись по мере необходимости, и для этих целей использовалось исключительно хранилище `SharedPreferences`. Однако такой способ не является единственным вариантом сохранения данных при работе с платформой Android-приложений. И в этой главе подробно рассматриваются иные доступные на Android средства хранения данных. В частности, речь пойдет:

- о внутреннем хранилище;
- о внешнем хранилище;
- о сетевом хранилище;
- о базе данных SQLite;
- о провайдерах контента.

Мы также определим, какой метод хранения лучше всего подходит для различных случаев. И начнем мы с рассмотрения внутреннего хранилища.

Внутреннее хранилище

В этом разделе мы рассмотрим доступное хранилище данных в среде Android-приложений, позволяющее разработчикам сохранять в памяти устройства частные данные. Как следует из определения *частный*, другие приложения не могут получать доступ к данным текущего приложения, хранящимся во *внутреннем хранилище*. Кроме того, при удалении этого приложения его файлы удаляются из хранилища.

Запись файлов во внутреннее хранилище

Для создания частного файла во внутреннем хранилище вызывается метод `openFileOutput()`. Эта функция использует два аргумента: первый служит именем открываемого файла (в форме `String`), а второй указывает режим работы. Заметим, что метод `openFileOutput()` вызывается в пределах экземпляра `Context`, такого как `Activity`.

Метод `openFileOutput()` возвращает поток `FileOutputStream`. Этот поток может применяться для указания записи файла с помощью метода `write()`. После завершения записи файла поток `FileOutputStream` следует закрыть, вызывая соответствующий метод `close()`. Процесс записи файла иллюстрируется следующим фрагментом кода:

```
private fun writeFile(fileName: String) {
   val content: String = "Hello world"
   val stream: FileOutputStream = openFileOutput(fileName, Context.MODE_PRIVATE)
   stream.write(content.toByteArray())
   stream.close()
}
```

С помощью объекта `MODE_PRIVATE` определяется режим работы, в котором создается файл с заданным именем (или заменяется файл с соответствующим именем), который делается частным для этого приложения.

Чтение файлов из внутреннего хранилища

Для чтения частного файла обратитесь к объекту `FileInputStream`, вызывая `openFileInput()`. Этот метод имеет единственный аргумент — имя файла для чтения. Метод `openFileInput()` следует вызывать в пределах экземпляра `Context`. После получения объекта `FileInputStream` выполняется считывание байтов из файла путем вызова соответствующей функции `read()`. После завершения чтения из файла закройте его, вызывая метод `close()`. Процесс чтения файла иллюстрируется следующим фрагментом кода:

```
private fun readFile(fileName: String) {
   val stream: FileInputStream = openFileInput(fileName)
   val data = ByteArray(1024)

   stream.read(data)
   stream.close()
}
```

Пример приложения, использующего внутреннее хранилище

Поскольку подходящий пример, как правило, помогает представить суть концепции, давайте быстро создадим использующее внутреннюю память приложение, вы-

полняющее *обновление файлов*. Работа с программой обновления файлов очень проста. Выполняется сбор текстовых данных от пользователя через поле ввода и обновление файла, который хранится во внутреннем хранилище. Затем пользователь с помощью представления в приложении может проверить текст, имеющийся в этом файле. Довольно просто, не правда ли? И это действительно так! Создайте новый проект Android и сами назовите его. Убедитесь, что присвоенное вами имя отражает цель приложения. После создания проекта сформируйте в пределах пакета проекта src два новых пакета с именами base и main.

Добавьте интерфейс `BaseView` в пакет base со следующим кодом:

```
package com.example.storageexamples.base

interface BaseView {

    fun bindViews()

    fun setupInstances()
}
```

Из предыдущей главы вы уже знакомы с определениями интерфейса представления. Если это не так, просмотрите еще раз соответствующий материал. В пакете main создайте представление `MainView`, которое следующим образом расширяет представление `BaseView`:

```
package com.example.storageexamples.main

import com.example.storageexamples.base.BaseView

interface MainView : BaseView {

    fun navigateToHome()
    fun navigateToContent()
}
```

Приложение, выполняющее обновление файлов, будет содержать два разных представления в виде фрагментов, с которыми может знакомиться пользователь. Первое представление — исходное (домашнее), с его помощью пользователь может обновить содержимое файла, а второе — представление содержимого (контентное), из которого пользователь может прочесть содержимое обновленного файла.

Создайте новое пустое действие внутри модуля main. Назовите это действие `MainActivity`. Удостоверьтесь, что для действия `MainActivity` сформировано пусковое действие. После создания `MainActivity` убедитесь, что оно перед обработкой расширяет `MainView`. Откройте файл activity_main.xml и отредактируйте его с включением следующего фрагмента кода:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context="com.example.storageexamples.main.MainActivity">

      <LinearLayout
         android:id="@+id/ll_container"
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         android:orientation="vertical"/>
      </android.support.constraint.ConstraintLayout>
```

Здесь в корневое представление файла макета добавлена группа представлений `LinearLayout`. Этот макет служит контейнером для домашнего и контентного фрагментов приложения. Теперь нам необходимо создать макеты для этих фрагментов.

Далее приведен код макета домашнего фрагмента fragment_home.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:paddingTop="@dimen/padding_default"
android:paddingBottom="@dimen/padding_default"
android:paddingStart="@dimen/padding_default"
android:paddingEnd="@dimen/padding_default"
android:gravity="center_horizontal"
android:layout_height="match_parent">
   <TextView
        android:id="@+id/tv_header"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/header_title"
        android:textSize="45sp"
        android:textStyle="bold"/>
   <EditText
        android:id="@+id/et_input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_top_large"
        android:hint="@string/hint_enter_text"/>
   <Button
        android:id="@+id/btn_submit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_default"
        android:text="@string/submit"/>
```

```
   <Button
        android:id="@+id/btn_view_file"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/view_file"
        android:background="@android:color/transparent"/>
</LinearLayout>
```

Перед открытием окна разработки для просмотра макета нужно добавить несколько ценных ресурсов. Ваш файл проекта strings.xml должен иметь вид, аналогичный следующему (за исключением строчного ресурса app_name):

```
<resources>
   <string name="app_name">Storage Examples</string>
   <string name="hint_enter_text">Enter text here...</string>
   <string name="submit">Update file</string>
   <string name="view_file">View file</string>
   <string name="header_title">FILE UPDATER</string>
</resources>
```

Кроме того, проект должен содержать файл dimens.xml, включающий следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <dimen name="padding_default">16dp</dimen>
   <dimen name="margin_default">16dp</dimen>
   <dimen name="margin_top_large">64dp</dimen>
</resources>
```

По завершении добавления приведенных ресурсов можно просмотреть окно макета домашнего фрагмента fragment_home.xml (рис. 7.1).

Теперь рассмотрим макет контентного фрагмента. Добавьте в каталог макетов ресурсов файл макета fragment_content.xml, включающий следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:padding="@dimen/padding_default"
android:layout_height="match_parent">
   <TextView
      android:id="@+id/tv_content"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:textSize="20sp"
      android:textStyle="bold"
      android:layout_marginTop="@dimen/margin_default"/>
</LinearLayout>
```

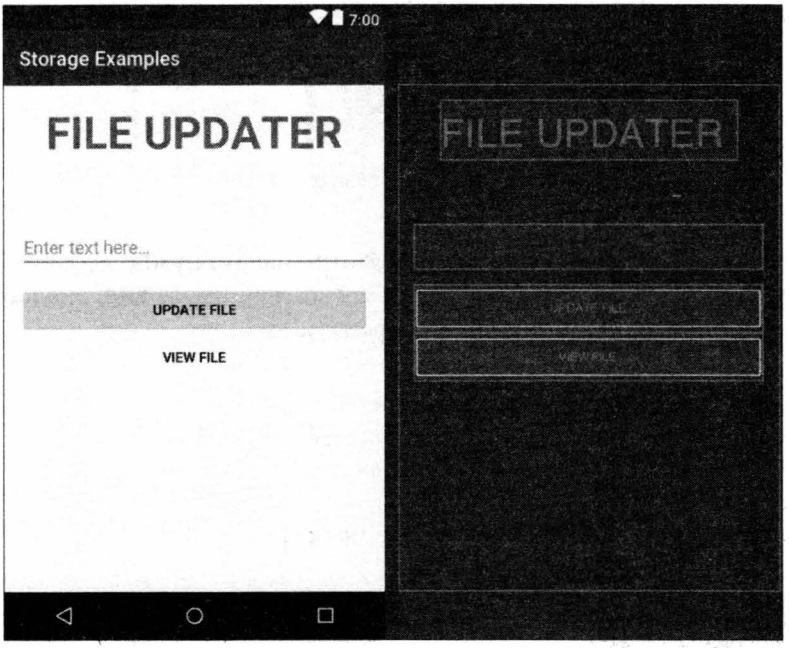


Рис. 7.1. Окно макета домашнего фрагмента приложения

Макет содержит единственное представление `TextView`, в котором находится имеющийся в файле внутреннего хранилища текст, отображаемый пользователю приложения. Можно при желании просмотреть окно разработки макета, но вряд ли вы увидите там большой объем информации.

Теперь необходимо создать соответствующие классы фрагментов, что позволит отображать только что созданные макеты фрагментов. Добавим класс `HomeFragment` к действию `MainActivity` следующим образом:

```
class HomeFragment : Fragment(), BaseView, View.OnClickListener {

   private lateinit var layout: LinearLayout
   private lateinit var tvHeader: TextView
   private lateinit var etInput: EditText
   private lateinit var btnSubmit: Button
   private lateinit var btnViewFile: Button

   private var outputStream: FileOutputStream? = null

    override fun onCreateView(inflater: LayoutInflater,
                              container: ViewGroup?,
                              savedInstanceState: Bundle?): View {

      // "Раздувание" макета fragment_home.xml
      layout = inflater.inflate(R.layout.fragment_home,
                                container, false) as LinearLayout
```

```
        setupInstances()
        bindViews()

        return layout
    }

    override fun bindViews() {
        tvHeader = layout.findViewById(R.id.tv_header)
        etInput = layout.findViewById(R.id.et_input)
        btnSubmit = layout.findViewById(R.id.btn_submit)
        btnViewFile = layout.findViewById(R.id.btn_view_file)

        btnSubmit.setOnClickListener(this)
        btnViewFile.setOnClickListener(this)
}
```

Следующий метод предназначен для создания копии свойств экземпляра:

```
override fun setupInstances() {
```

Откройте новый поток `FileOutputStream` в файле под названием content_file. Этот файл является частным — он сохраняется во внутреннем хранилище и доступен лишь для нашего приложения:

```
outputStream = activity?.openFileOutput("content_file",
    Context.MODE_PRIVATE)
}
```

Следующая функция вызывается для отображения пользователю ошибки, если заданы некорректные вводимые данные:

```
private fun showInputError() {
    etInput.error = "File input cannot be empty."
    etInput.requestFocus()
}
```

Запишите строчный контент с помощью файла в поток `FileOutputStream`:

```
private fun writeFile(content: String) {
    outputStream?.write(content.toByteArray())
    outputStream?.close()
}
```

Приведенная далее функция вызывается для очистки ввода в поле ввода:

```
private fun clearInput() {
    etInput.setText("")
}
```

Следующий фрагмент кода отображает сообщение пользователю об успехе при вызове:

```
private fun showSaveSuccess() {
   Toast.makeText(activity, "File updated successfully.",
   Toast.LENGTH_LONG).show()
}

override fun onClick(view: View?) {
   val id = view?.id
   if (id == R.id.btn_submit) {
      if (TextUtils.isEmpty(etInput.text)) {
```

Если пользователь введет пустое значение в качестве вводимых данных в файл контента, отображается сообщение об ошибке:

```
  showInputError()
  } else {
```

Запишем контент в файл, очистим ввод `EditText` и отобразим сообщение об успешном обновлении файла:

```
            writeFile(etInput.text.toString())
            clearInput()
            showSaveSuccess()
         }
      } else if (id == R.id.btn_view_file) {
         // Выборка ссылки на MainActivity
         val mainActivity = activity as MainActivity
```

Перейдем к фрагменту контента и отобразим на панели действий «домашнюю» кнопку, что позволит пользователю вернуться к предыдущему фрагменту:

```
         mainActivity.navigateToContent()
         mainActivity.showHomeNavigation()
      }
   }
}
```

Просмотрите комментарии, сопровождающие приведенные фрагменты кода в файлах примеров исходного кода, размещенных в сопровождающем книгу файловом архиве, чтобы убедиться в том, что вы понимаете суть происходящих изменений.

При добавлении фрагмента `HomeFragment` к действию `MainActivity` следует также добавить фрагмент для отображения пользователю макета fragment_content.xml. Далее приводится необходимый класс `ContentFragment`. Добавим этот класс к действию `MainActivity`:

```
class ContentFragment : Fragment(), BaseView {
   private lateinit var layout: LinearLayout
   private lateinit var tvContent: TextView
   private lateinit var inputStream: FileInputStream
```

```
    override fun onCreateView(inflater: LayoutInflater?, container: ViewGroup?,
                              savedInstanceState: Bundle?): View {
        layout = inflater?.inflate(R.layout.fragment_content,
                                   container, false) as LinearLayout
        setupInstances()
        bindViews()
        return layout
    }
    override fun onResume() {
```

Обновим контент, который отображается в `TextView` при возобновлении фрагмента:

```
updateContent()
    super.onResume()
}

private fun updateContent() {
    tvContent.text = readFile()
}

override fun bindViews() {
    tvContent = layout.findViewById(R.id.tv_content)
}

override fun setupInstances() {
    inputStream = activity.openFileInput("content_file")
}
```

Следующий код просматривает содержимое файла во внутреннем хранилище и возвращает содержимое в виде строки:

```
private fun readFile(): String {
    var c: Int
    var content = ""
    c = inputStream.read()
    while (c != -1) {
        content += Character.toString(c.toChar())
        c = inputStream.read()
    }

    inputStream.close()
    return content
    }
}
```

Экземпляр `tvContent` (представление `TextView` отображает пользователю содержимое файла) обновляется после возобновления действия `ContentFragment`. Содержимое представления `TextView` обновляется при выводе из представления `TextView` текста,

соответствующего содержимому, которое прочтено из файла с помощью метода `readFile()`. И, наконец, необходимо завершение для `MainActivity`.

Полностью завершенный класс `MainActivity` должен иметь вид, аналогичный следующему:

```
package com.example.storageexamples.main

import android.aupport.v4.app.Fragment
import android.content.Context
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.text.TextUtils
import android.view.LayoutInflater
import android.view.MenuItem
import android.view.View
import android.view.ViewGroup
import android.widget.*
import com.example.storageexamples.R
import com.example.storageexamples.base.BaseView
import java.io.FileInputStream
import java.io.FileOutputStream

class MainActivity : AppCompatActivity(), MainView {

   private lateinit var llContainer: LinearLayout
```

Установим экземпляры фрагментов:

```
private lateinit var homeFragment: HomeFragment
private lateinit var contentFragment: ContentFragment

override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)
   setupInstances()
   bindViews()
   navigateToHome()
}

override fun bindViews() {
   llContainer = findViewById(R.id.ll_container)
}

override fun setupInstances() {
   homeFragment = HomeFragment()
   contentFragment = ContentFragment()
}
```

```
private fun hideHomeNavigation() {
  supportActionBar?.setDisplayHomeAsUpEnabled(false)
}

private fun showHomeNavigation() {
   supportActionBar?.setDisplayHomeAsUpEnabled(true)
}

override fun navigateToHome() {
   val transaction = supportFragmentManager.beginTransaction()
   transaction.replace(R.id.ll_container, homeFragment)
   transaction.commit()
   supportActionBar?.title = "Home"
}

override fun navigateToContent() {
   val transaction = supportFragmentManager.beginTransaction()
   transaction.replace(R.id.ll_container, contentFragment)
   transaction.commit()
   supportActionBar?.title = "File content"
}

override fun onOptionsItemSelected(item: MenuItem?): Boolean {
   val id = item?.itemId
   if (id == android.R.id.home) {
      navigateToHome()
      hideHomeNavigation()
   }
   return super.onOptionsItemSelected(item)
}

class HomeFragment : Fragment(), BaseView, View.OnClickListener {
   private lateinit var layout: LinearLayout
   private lateinit var tvHeader: TextView
   private lateinit var etInput: EditText
   private lateinit var btnSubmit: Button
   private lateinit var btnViewFile: Button
   private lateinit var outputStream: FileOutputStream
   override fun onCreateView(inflater: LayoutInflater?,
                        container: ViewGroup?, savedInstanceState: Bundle?): View {
```

Создадим макет fragment_home.xml:

```
   layout = inflater?.inflate(R.layout.fragment_home,
                              container, false) as LinearLayout
   setupInstances()
   bindViews()
```

```
    return layout
}

override fun bindViews() {
    tvHeader = layout.findViewById(R.id.tv_header)
    etInput = layout.findViewById(R.id.et_input)
    btnSubmit = layout.findViewById(R.id.btn_submit)
    btnViewFile = layout.findViewById(R.id.btn_view_file)
    btnSubmit.setOnClickListener(this)
    btnViewFile.setOnClickListener(this)
}

// Метод создания копии экземпляра свойств
    override fun setupInstances() {
```

Откроем новый поток `FileOutputStream` для файла под именем content_file. Этот файл является частным файлом, который сохраняется во внутреннем хранилище и доступен лишь для нашего приложения:

```
outputStream = activity.openFileOutput("content_file",
    Context.MODE_PRIVATE)
}

// Вызвано для отображения ошибки пользователю при некорректных
// вводимых данных
    private fun showInputError() {
        etInput.error = "File input cannot be empty."
        etInput.requestFocus()
    }

    // Запись содержимого строки в файл с помощью [FileOutputStream]
    private fun writeFile(content: String) {
        outputStream.write(content.toByteArray())
    }

    // Вызвано для очистки ввода в поле ввода
    private fun clearInput() {
        etInput.setText("")
    }

    // Отображение сообщения об успехе пользователю при вызове
    private fun showSaveSuccess() {
        Toast.makeText(activity, "File updated successfully.",
        Toast.LENGTH_LONG).show()
    }
```

```
    override fun onClick(view: View?) {
        val id = view?.id
        if (id == R.id.btn_submit) {
```

Следующий фрагмент кода отображает сообщение об ошибке, если пользователь направляет в качестве ввода файла контента пустое значение:

```
            if (TextUtils.isEmpty(etInput.text)) {
                showInputError()
            } else {
                // Запись контента в файл, очистка ввода EditText
                // и отображение сообщения об успешном обновлении файла
                writeFile(etInput.text.toString())
                clearInput()
                showSaveSuccess()
            }
        } else if (id == R.id.btn_view_file) {
            // Выборка ссылки на MainActivity
            val mainActivity = activity as MainActivity
```

Переместим пользователя к фрагменту контента и отобразим «домашнюю» кнопку на панели действий, что разрешит пользователю вернуться к предыдущему фрагменту:

```
            mainActivity.navigateToContent()
            mainActivity.showHomeNavigation()
            }
        }
}

class ContentFragment : Fragment(), BaseView {
    private lateinit var layout: LinearLayout
    private lateinit var tvContent: TextView
    private lateinit var inputStream: FileInputStream
    override fun onCreateView(inflater: LayoutInflater?,
                              container: ViewGroup?,
                              savedInstanceState: Bundle?): View {
        layout = inflater?.inflate(R.layout.fragment_content,
                                   container, false) as LinearLayout
        setupInstances()
        bindViews()
        return layout
    }
```

Обновим контент, который отображается в представлении `TextView` при возобновлении фрагмента:

```
    override fun onResume() {
        updateContent()
```

```
        super.onResume()
    }

    private fun updateContent() {
        tvContent.text = readFile()
    }

    override fun bindViews() {
        tvContent = layout.findViewById(R.id.tv_content)
    }

    override fun setupInstances() {
         inputStream = activity.openFileInput("content_file")
    }
```

Теперь рассмотрим содержимое файла во внутреннем хранилище и возвратим содержимое в виде строки:

```
    private fun readFile(): String {
        var c: Int
        var content = ""
        c = inputStream.read()
        while (c != -1) {
            content += Character.toString(c.toChar())
            c = inputStream.read()
        }
        inputStream.close()
        return content
        }
    }
}
```

На этом мы полностью завершили действие `MainActivity`, и наше приложение готово к выполнению.

Создайте и запустите проект на выбранном вами устройстве. После запуска приложения окно домашнего фрагмента `MainActivity` отобразится на дисплее устройства. Введите в поле ввода (`EditText`) произвольный текст (рис. 7.2).

После добавления текста в поле ввода (`EditText`) щелкните на кнопке **UPDATE FILE** (Обновить файл) — файл внутреннего хранилища обновится в соответствии с предоставленным содержимым, и по завершении этого обновления вы получите соответствующее уведомление. Щелкните теперь на кнопке **VIEW FILE** (Просмотр файла) — введенный текст отображается с помощью соответствующего представления `TextView`, которое включает обновленное содержимое файла (рис. 7.3).

Несмотря на простоту, созданное приложение для обновления файлов служит хорошей иллюстрацией того, каким образом приложения Android взаимодействуют со своим внутренним хранилищем.

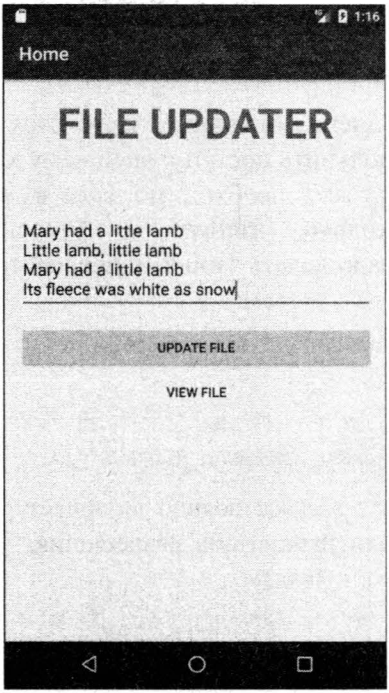


Рис. 7.2. Окно домашнего фрагмента отображено на дисплее устройства, и в поле ввода введена произвольная запись

Рис. 7.3. Обновленное содержимое файла внутреннего хранилища отображено в окне контентного фрагмента

Сохранение кэшированных файлов

Если данные нет надобности сохранять на постоянной основе, вместо записи данных в хранилище воспользуйтесь командой `cacheDir` для открытия файла, представляющего каталог (внутри внутреннего хранилища), где приложение должно сохранять временные файлы кэша.

Команда `cacheDir` возвращает класс `File`. Таким способом — с помощью класса `File` — можно использовать все методы, находящиеся в вашем распоряжении, например метод `outputStream()`, который возвращает поток `FileOutputStream`.

Внешнее хранилище

Внешнее хранилище используется для создания используемых совместно и общедоступных для чтения файлов и обеспечения доступа к ним. Совместно используемое внешнее хранилище поддерживается всеми устройствами Android. Первое, что необходимо для того, чтобы использовать внешнее хранилище, — это разрешить доступ к нему.

Получение разрешения на доступ к внешнему хранилищу

Ваше приложение должно располагать разрешениями READ_EXTERNAL_STORAGE и WRITE_EXTERNAL_STORAGE, прежде чем оно сможет получить доступ к внешнему хранилищу API. При этом разрешение READ_EXTERNAL_STORAGE необходимо, если из внешнего хранилища требуется выполнять только чтение, а разрешение WRITE_EXTERNAL_STORAGE — если для приложения надо делать запись непосредственно во внешнее хранилище.

Эти два разрешения, как было показано в предыдущих главах, можно легко добавить в файл манифеста:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Следует отметить, что разрешение WRITE_EXTERNAL_STORAGE неявно включает разрешение READ_EXTERNAL_STORAGE. Таким образом, если нужны оба разрешения, достаточно запросить только разрешение WRITE_EXTERNAL_STORAGE:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Проверка доступности носителя данных

Случается, что по какой-либо причине, например при неисправности, внешний носитель данных может оказаться недоступным. Соответственно, важно проверить доступность внешних носителей данных, прежде чем попытаться их использовать.

Для проверки доступности носителя данных должен вызваться метод getExternalStorageState(). Так, чтобы проверить, доступно ли внешнее хранилище для записи, можно применить в приложении следующий фрагмент кода:

```
private fun isExternalStorageWritable(): Boolean {

   val state = Environment.getExternalStorageState()
  return Environment.MEDIA_MOUNTED == state
}
```

Здесь мы сначала получаем текущее состояние внешнего хранилища, а затем проверяем, находится ли оно в состоянии MEDIA_MOUNTED. Если это так, с помощью метода isExternalStorageWritable() возвращается значение true (истина), и приложение может выполнять запись в это хранилище.

Проверка доступности для чтения внешнего хранилища также проста:

```
private fun isExternalStorageReadable(): Boolean {

   val state = Environment.getExternalStorageState()
   return Environment.MEDIA_MOUNTED == state ||
```

```
    Environment.MEDIA_MOUNTED_READ_ONLY == state
}
```

Таким образом, ваше приложение может считывать данные из внешнего хранилища, если оно находится либо в состоянии `MEDIA_MOUNTED`, либо в состоянии `MEDIA_MOUNTED_READ_ONLY`.

Хранение общедоступных файлов

В общедоступном каталоге должны храниться файлы, к которым пользователю или другим приложениям доступ может понадобиться позже. Примерами таких каталогов являются каталоги Pictures/ и Music/.

Для получения объекта `File`, представляющего необходимый общедоступный каталог, приложением должен вызываться метод `getExternalStoragePulicDirectory()`. Тип каталога для выборки должен передаваться ему в качестве единственного аргумента функции.

Далее приведена функция, которая создает каталог для сохранения музыки:

```
private fun getMusicStorageDir(collectionName: String): File {

    val file = File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_MUSIC), collectionName)
        if (!file.mkdir()) {
            Log.d("DIR_CREATION_STATUS", "Directory creation failed.")
            return file
        }
```

В случае возникновения ошибки при создании общего каталога в консоли выводится соответствующее сообщение.

Кэширование файлов с помощью внешнего хранилища

Иногда в сценариях необходимо получить содержимое файлов, которые кэшированы во внешнем хранилище. Открыть файл, представляющий каталог внешнего хранилища, где приложение должно сохранять кэшированные файлы, можно с помощью команды `externalCacheDir`.

Сетевое хранилище

А теперь речь пойдет о хранилище данных на удаленном сервере. В отличие от рассмотренных средств хранения, этот вариант использует для хранения и извлечения существующих на удаленном сервере данных сетевое соединение. Подобный носитель данных мы уже рассматривали при создании в предыдущих главах Android-приложения Messenger — приложение по обмену сообщениями пользовалось уда-

ленным сервером для хранения и поиска информации. Клиент-серверная архитектура обычно применяется в тех случаях, когда удаленный сервер является источником данных для клиентского приложения: клиент направляет запрос о получении необходимых данных (обычно запрос GET) к серверу через HTTP, а сервер отвечает, возвращая клиенту нужные данные, тем самым завершая цикл HTTP-транзакций.

Работа с базой данных SQLite

SQLite — это наиболее популярная *реляционная система управления базами данных* (РСУБД), которая, в отличие от многих подобных систем, не представляет собой собственно клиент-серверную базу данных. Вместо этого база данных SQLite непосредственно встроена в приложение.

ОС Android обеспечивает полную поддержку СУБД SQLite. Базы данных SQLite доступны всем классам Android-проекта. Обратите внимание, однако, что в Android база данных доступна только создавшему ее приложению.

Для работы с SQLite в Android рекомендуется применение библиотеки Room persistence. Первый шаг для работы с этой библиотекой в Android заключается во включении в сценарий проекта build.gradle необходимых зависимостей:

```
implementation "android.arch.persistence.room:runtime:1.0.0-alpha9-1"
implementation "android.arch.persistence.room:rxjava2:1.0.0-alpha9-1"
implementation "io.reactivex.rxjava2:rxandroid:2.0.1"
kapt "android.arch.persistence.room:compiler:1.0.0-alpha9-1"
```

При использовании библиотеки Room можно легко сформировать необходимые сущности. Все они должны быть аннотированы с помощью аннотации @Entity. Далее приводится простая сущность User:

```
package com.example.roomexample.data

import android.arch.persistence.room.ColumnInfo
import android.arch.persistence.room.Entity
import android.arch.persistence.room.PrimaryKey

@Entity
data class User(
   @ColumnInfo(name = "first_name")
   var firstName: String = "",
   @ColumnInfo(name = "surname")
   var surname: String = "",
   @ColumnInfo(name = "phone_number")
   var phoneNumber: String = "",
   @PrimaryKey(autoGenerate = true)
   var id: Long = 0
)
```

Здесь библиотека Room создает необходимую таблицу SQLite для определенной сущности User. Таблица имеет имя (user) и четыре атрибута (id, first_name, surname и phone_number). Атрибут id является основным для созданной пользовательской таблицы — в нашем случае для него применяется аннотация @PrimaryKey. При этом в аннотации @PrimaryKey указано, что первичные ключи каждой записи в пользовательской таблице должны генерироваться Room при помощи установки autoGenerate = true. Аннотация @ColumnInfo используется для указания дополнительной информации, относящейся к столбцу в таблице. Например, обратите внимание на следующий фрагмент кода:

```
@ColumnInfo(name = "first_name")
var firstName: String = ""
```

Приведенный код указывает, что имеется атрибут firstName, которым обладает User. Аннотация @ColumnInfo(name = "first_name") устанавливает название столбца first_name в пользовательской таблице для атрибута firstName.

Для ведения записей в базе данных и чтения из нее необходимо располагать объектом *Data Access Object* (DAO). Объект DAO позволяет выполнять операции с базой данных с использованием аннотированных методов. Далее приводится код объекта DAO для сущности User:

```
package com.example.roomexample.data

import android.arch.persistence.room.Dao
import android.arch.persistence.room.Insert
import android.arch.persistence.room.OnConflictStrategy
import android.arch.persistence.room.Query
import io.reactivex.Flowable

@Dao
interface UserDao {
   @Query("SELECT * FROM user")
   fun all(): Flowable<List<User>>
   @Query("SELECT * FROM user WHERE id = :id")
   fun findById(id: Long): Flowable<User>
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun insert(user: User)
}
```

Аннотация @Query отмечает метод в классе DAO как метод запроса. Запрос, который выполняется при вызове метода, передается в качестве значения аннотации. Естественно, запросы, переданные @Query, являются SQL-запросами. Создание SQL-запросов — слишком обширная тема для того, чтобы ее здесь охватить полностью, но неплохо потратить немного времени, чтобы хотя бы понять, как их правильно создавать.

Аннотация `@Insert` используется для вставки данных в таблицу. Другими имеющимися важными аннотациями являются `@Update` и `@Delete` — они служат соответственно для обновления и удаления данных в таблице базы данных.

Наконец, после создания необходимых объектов и DAO, необходимо определить базу данных приложения. Для этого следует создать подкласс `RoomDatabase` и аннотировать его с помощью аннотации `@Database`. Как минимум, эта аннотация должна включать коллекцию ссылок на классы сущностей и номер версии базы данных. Далее приводится примерный абстрактный класс `AppDatabase`:

```
@Database(entities = [User::class], version = 1)
public abstract class AppDatabase : RoomDatabase() {
   abstract fun userDao(): UserDao
}
class.Add
```

Создавая класс базы данных приложения, можно получить экземпляр базы данных, вызывая `databaseBuilder()`:

```
val db = Room.databaseBuilder(<context>, AppDatabase::class.java,
                              "app-database").build()
```

После получения экземпляра `RoomDatabase`, можно его использовать для извлечения объектов доступа к данным, которые, в свою очередь, могут применяться для чтения, записи, обновления, запроса и удаления данных из базы данных.

Продолжая принятую до сих пор практику, создадим простое приложение, показывающее, как можно использовать SQLite в Android с помощью библиотеки `Room`. Приложение, которое мы собираемся создать, разрешит его пользователю вручную вводить относящуюся к пользователям информацию, а также просматривать вводимую позднее информацию о пользователях.

Создайте новый проект Android с пустым действием `MainActivity`, которое установлено в качестве действия запуска. Добавьте в сценарий build.gradle приложения следующие зависимости:

```
implementation 'com.android.support:design:26.1.0'
implementation "android.arch.persistence.room:runtime:1.0.0"
implementation "android.arch.persistence.room:rxjava2:1.0.0"
implementation "io.reactivex.rxjava2:rxandroid:2.0.1"
kapt "android.arch.persistence.room:compiler:1.0.0"
```

Кроме того, примените к сценарию build.gradle отдельный плагин `kotlin-kapt`:

```
apply plugin: 'kotlin-kapt'
```

После добавления указанных зависимостей проекта создайте данные и пакет ui в пакете источника данных проекта. В пакет ui добавьте действие `MainView`, например, следующим образом:

```
package com.example.roomexample.ui

interface MainView {
   fun bindViews()
  fun setupInstances()
}
```

После добавления представления `MainView` в пакет ui также перенесите в пакет ui действие `MainActivity`. Рассмотрим базу данных приложения. Поскольку приложение хранит пользовательскую информацию, необходимо создать сущность `User`. В пакет data добавьте следующую сущность `User`:

```
package com.example.roomexample.data

import android.arch.persistence.room.ColumnInfo
import android.arch.persistence.room.Entity
import android.arch.persistence.room.PrimaryKey

@Entity
data class User(
   @ColumnInfo(name = "first_name")
   var firstName: String = "",
   @ColumnInfo(name = "surname")
   var surname: String = "",
   @ColumnInfo(name = "phone_number")
   var phoneNumber: String = "",
   @PrimaryKey(autoGenerate = true)
   var id: Long = 0
)
```

Создайте в пакете data интерфейс `UserDao`:

```
package com.example.roomexample.data

import android.arch.persistence.room.Dao
import android.arch.persistence.room.Insert
import android.arch.persistence.room.OnConflictStrategy
import android.arch.persistence.room.Query
import io.reactivex.Flowable

@Dao
interface UserDao {
   @Query("SELECT * FROM user")
   fun all(): Flowable<List<User>>
   @Query("SELECT * FROM user WHERE id = :id")
   fun findById(id: Long): Flowable<User>
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun insert(user: User)
}.
```

Интерфейс `UserDao` включает три метода: `all()`, `findById()` и `Insert()`. Метод `all()` возвращает объект `Flowable`, включающий список всех пользователей. Метод `findById()` находит пользователя `User`, чей идентификатор `id` соответствует тому, что передан методу, если таковой имеется, и возвращает пользователя `User` в объект `Flowable`. Метод `insert()` используется для вставки пользователя в виде записи в таблицу `user`.

Теперь, имея DAO и сущность, создадим класс `AppDatabase`. Добавьте в пакет data следующий код:

```
package com.example.roomexample.data

import android.arch.persistence.room.Database
import android.arch.persistence.room.Room
import android.arch.persistence.room.RoomDatabase
import android.content.Context

@Database(entities = arrayOf(User::class), version = 1, exportSchema = false)
internal abstract class AppDatabase : RoomDatabase() {

   abstract fun userDao(): UserDao
   companion object Factory {
      private var appDatabase: AppDatabase? = null
      fun create(ctx: Context): AppDatabase {
         if (appDatabase == null) {
            appDatabase = Room.databaseBuilder(ctx.applicationContext,
                                               AppDatabase::class.java,
                                                "app-database").build()
         }
         return appDatabase as AppDatabase
      }
   }
}
```

К этому моменту мы создали сопутствующий объект `Factory`, обладающий единственной функцией `create()`, которая призвана сформировать экземпляр `AppDatabase`, если таковой не создан, и вернуть этот экземпляр для применения.

Созданием `AppDatabase` завершается цикл работы с данными. Теперь необходимо создать подходящие макеты для представлений приложения. Добавим в действие `MainActivity` два фрагмента: первый будет использоваться для сбора входных данных, относящихся к создаваемому новому пользователю, а второй — отображать информацию обо всех созданных в `RecyclerView` пользователях. Сначала включим в макет activity_main.xml следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
```

```
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.roomexample.ui.MainActivity">

<LinearLayout
      android:id="@+id/ll_container"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:orientation="vertical"/>
</android.support.constraint.ConstraintLayout>
```

Здесь объект `LinearLayout` включает фрагменты из действия `MainActivity`. Добавим файл fragment_create_user.xml в каталог макетов источника данных с помощью следующего кода:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:gravity="center_horizontal"
android:padding="@dimen/padding_default">
   <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:textSize="32sp"
         android:text="@string/create_user"/>
   <EditText
         android:id="@+id/et_first_name"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:layout_marginTop="@dimen/margin_default"
         android:hint="@string/first_name"
         android:inputType="text"/>
   <EditText
         android:id="@+id/et_surname"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:layout_marginTop="@dimen/margin_default"
         android:hint="@string/surname"
         android:inputType="text"/>
   <EditText
         android:id="@+id/et_phone_number"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:layout_marginTop="@dimen/margin_default"
```

```
        android:hint="@string/phone_number"
        android:inputType="phone"/>
    <Button
        android:id="@+id/btn_submit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_default"
        android:text="@string/submit"/>
    <Button
        android:id="@+id/btn_view_users"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_default"
        android:text="@string/view_users"/>
</LinearLayout>
```

Теперь добавим исходный код макета fragment_list_users.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/rv_users"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

Файл fragment_list_users.xml включает представление `RecyclerView`, содержащее информацию о каждом пользователе, сохраненном в базе данных. Для представления `RecyclerView` надо создать элемент ресурса макета хранителя представлений. Создадим новый файл макета vh_user.xml и добавим в него следующее содержимое:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:padding="@dimen/padding_default"
android:layout_height="wrap_content">
    <TextView
        android:id="@+id/tv_first_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/tv_surname"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_default"/>
    <TextView
        android:id="@+id/tv_phone_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_default"/>
    <View
        android:layout_width="match_parent"
        android:layout_height="1dp"
        android:layout_marginTop="@dimen/margin_default"
        android:background="#e8e8e8"/>
</LinearLayout>
```

Как и следовало ожидать, необходимо добавить в проект некоторые строковые и размерные ресурсы. Откройте файл макета strings.xml приложения и внесите в него следующие строковые ресурсы:

```
<resources>

    <string name="first_name">First name</string>
    <string name="surname">Surname</string>
    <string name="phone_number">Phone number</string>
    <string name="submit">Submit</string>
    <string name="create_user">Create User</string>
    <string name="view_users">View users</string>
</resources>
```

Многоточия «...» между фрагментами кода означают наличие в местах их вставки в файлах кода дополнительного — чаще всего показанного ранее — кода.

Теперь создайте в проекте следующие ресурсы измерений вашего проекта:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="padding_default">16dp</dimen>
    <dimen name="margin_default">16dp</dimen>
</resources>
```

Теперь приступим к работе с действием `MainActivity`. Как указывалось ранее, в классе `MainActivity` применяются два различных фрагмента: первый фрагмент позволяет сохранять персональные данные в базе данных SQL, а второй — просматривать пользователю информацию о персонах, которая хранится в базе данных.

Сначала создадим фрагмент `CreateUserFragment`. Добавим в действие `MainActivity` (размещен в файле MainActivity.kt) следующий класс фрагмента:

```
class CreateUserFragment : Fragment(), MainView, View.OnClickListener {

   private lateinit var btnSubmit: Button
   private lateinit var etSurname: EditText
   private lateinit var btnViewUsers: Button
   private lateinit var layout: LinearLayout
   private lateinit var etFirstName: EditText
   private lateinit var etPhoneNumber: EditText
   private lateinit var userDao: UserDao
   private lateinit var appDatabase: AppDatabase

   override fun onCreateView(inflater: LayoutInflater,
               container: ViewGroup?, savedInstanceState: Bundle?): View
      layout = inflater.inflate(R.layout.fragment_create_user,
               container, false) as LinearLayout
      bindViews()
      setupInstances()
      return layout
   }
   override fun bindViews() {
      btnSubmit = layout.findViewById(R.id.btn_submit)
      btnViewUsers = layout.findViewById(R.id.btn_view_users)
      etSurname = layout.findViewById(R.id.et_surname)
      etFirstName = layout.findViewById(R.id.et_first_name)
      etPhoneNumber = layout.findViewById(R.id.et_phone_number)
      btnSubmit.setOnClickListener(this)
      btnViewUsers.setOnClickListener(this)
   }

   override fun setupInstances() {
   appDatabase = AppDatabase.create(activity)
   // Получение экземпляра AppDatabase
   userDao = appDatabase.userDao() // получение экземпляра UserDao
}
```

Следующий метод проверяет представленные в созданной пользователем форме входные данные:

```
private fun inputsValid(): Boolean {

   var inputValid = true
   val firstName = etFirstName.text
   val surname = etSurname.text
   val phoneNumber = etPhoneNumber.text

   if (TextUtils.isEmpty(firstName)) {
      etFirstName.error = "First name cannot be empty"
```

```
            etFirstName.requestFocus()
            inputValid = false

        } else if (TextUtils.isEmpty(surname)) {
            etSurname.error = "Surname cannot be empty"
            etSurname.requestFocus()
            inputValid = false

        } else if (TextUtils.isEmpty(phoneNumber)) {
            etPhoneNumber.error = "Phone number cannot be empty"
            etPhoneNumber.requestFocus()
            inputValid = false

        } else if (!android.util.Patterns.PHONE
            .matcher(phoneNumber).matches()) {
                etPhoneNumber.error = "Valid phone number required"
                etPhoneNumber.requestFocus()
                inputValid = false
        }
        return inputValid
}
```

Следующая функция отображает всплывающее сообщение, говорящее об успешном создании учетной записи пользователя:

```
private fun showCreationSuccess() {
    Toast.makeText(activity, "User successfully created.",
                   Toast.LENGTH_LONG).show()
    }

    override fun onClick(view: View?) {
        val id = view?.id
        if (id == R.id.btn_submit) {
            if (inputsValid()) {
                val user = User(
                    etFirstName.text.toString(),
                    etSurname.text.toString(),
                    etPhoneNumber.text.toString())
                    Observable.just(userDao)
                    .subscribeOn(Schedulers.io())
                    .subscribe( { dao ->
                   dao.insert(user) // using UserDao to save user to database.
                    activity?.runOnUiThread { showCreationSuccess() }
                }, Throwable::printStackTrace)
            }
        } else if (id == R.id.btn_view_users) {
            val mainActivity = activity as MainActivity
```

```
            mainActivity.navigateToList()
            mainActivity.showHomeButton()
        }
    }
}
```

Ранее уже неоднократно выполнялась обработка фрагментов. Поэтому сосредоточим внимание на тех частях этого фрагмента, которые функционируют с `AppDatabase`. В `setupInstances()` установим ссылки на `AppDatabase` и `UserDao`. Экземпляр `AppDatabase` мы получим, вызывая функцию `create()` из `Factory`, сопутствующего объекта для `AppDatabase`. Экземпляр `UserDao` легко извлекается путем вызова `appDatabase.userDao()`.

Перейдем к методу `onClick()` из класса фрагмента. После щелчка на кнопке отправки данных передаваемая информация о пользователе проверяется на достоверность. Соответствующее сообщение об ошибке отображается, если недопустим какой-либо пункт из входных данных. Если все входные данные действительны, формируется содержащий предоставленную информацию о пользователе новый объект `User`, который сохраняется в базе данных. Все это реализовано в следующем коде:

```
if (inputsValid()) {
    val user = User(
        etFirstName.text.toString(),
        etSurname.text.toString(),
        etPhoneNumber.text.toString())

        Observable.just(userDao)
                .subscribeOn(Schedulers.io())
                .subscribe( { dao ->
        dao.insert(user) // using UserDao to save user to database.
        activity?.runOnUiThread { showCreationSuccess() }
    }, Throwable::printStackTrace)
}
```

Добавьте теперь к действию `MainActivity` следующий фрагмент `ListUsersFragment`:

```
class ListUsersFragment : Fragment(), MainView {
    private lateinit var layout: LinearLayout
    private lateinit var rvUsers: RecyclerView
    private lateinit var appDatabase: AppDatabase
    override fun onCreateView(inflater: LayoutInflater,
    container: ViewGroup?, savedInstanceState: Bundle?): View {
        layout = inflater.inflate(R.layout.fragment_list_users,
        container, false) as LinearLayout
        bindViews()
        setupInstances()
        return layout
}
```

Выполним привязку экземпляра представления обработчика для пользователя к его элементу макета:

```
override fun bindViews() {
   rvUsers = layout.findViewById(R.id.rv_users)
   }

   override fun setupInstances() {
      appDatabase = AppDatabase.create(activity)
      rvUsers.layoutManager = LinearLayoutManager(activity)
      rvUsers.adapter = UsersAdapter(appDatabase)
   }

   private class UsersAdapter(appDatabase: AppDatabase) :
   RecyclerView.Adapter<UsersAdapter.ViewHolder>() {
      private val users: ArrayList<User> = ArrayList()
      private val userDao: UserDao = appDatabase.userDao()
      init {
      populateUsers()
   }

   override fun onCreateViewHolder(parent: ViewGroup?, viewType: Int):
   ViewHolder {
      val layout = LayoutInflater.from(parent?.context)
      .inflate(R.layout.vh_user, parent, false)
      return ViewHolder(layout)
   }

   override fun onBindViewHolder(holder: ViewHolder?, position: Int) {
      val layout = holder?.itemView
      val user = users[position]
      val tvFirstName = layout?.findViewById<TextView>(R.id.tv_first_name)
      val tvSurname = layout?.findViewById<TextView>(R.id.tv_surname)
      val tvPhoneNumber = layout?.findViewById<TextView>
      (R.id.tv_phone_number)
      tvFirstName?.text = "First name: ${user.firstName}"
      tvSurname?.text = "Surname: ${user.surname}"
      tvPhoneNumber?.text = "Phone number: ${user.phoneNumber}"
   }

   // Заполняет список пользователей ArrayList объектами User
   private fun populateUsers() {
      users.clear()
```

Сведем всех пользователей в таблицу пользователей базы данных. После успешного извлечения списка добавим все пользовательские объекты пользователя в список пользователей `ArrayList`:

```
    userDao.all()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({ res ->
        users.addAll(res)
        notifyDataSetChanged()
        }, Throwable::printStackTrace)
    }
    override fun getItemCount(): Int {
        return users.size
    }
    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)
    }
}
```

Класс `UsersAdapter` во фрагменте `ListUsersFragment` применяет экземпляр `UserDao` для заполнения его списка пользователей. Это заполнение выполняется с помощью метода `populateUsers()`. При вызове метода `populateUsers()` список всех пользователей, которые сохранены приложением, получается путем вызова `userDao.all()`. После успешного нахождения пользователей все объекты `User` добавляются к пользователям `ArrayList` из `UserAdapter`. Затем адаптер уведомляется об изменении данных в своем наборе данных с помощью вызова метода уведомления `DataSetChanged()`.

Действие `MainActivity` нуждается в некоторых незначительных дополнениях. Завершенное действие `MainActivity` должно выглядеть следующим образом:

```
package com.example.roomexample.ui

import android.app.Fragment
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.support.v7.widget.LinearLayoutManager
import android.support.v7.widget.RecyclerView
import android.text.TextUtils
import android.view.LayoutInflater
import android.view.MenuItem
import android.view.View
import android.view.ViewGroup
import android.widget.*
import com.example.roomexample.R
import com.example.roomexample.data.AppDatabase
import com.example.roomexample.data.User
import com.example.roomexample.data.UserDao
import io.reactivex.Observable
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers
```

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      navigateToForm()
   }

   private fun showHomeButton() {
      supportActionBar?.setDisplayHomeAsUpEnabled(true)
   }

   private fun hideHomeButton() {
      supportActionBar?.setDisplayHomeAsUpEnabled(false)
   }

   private fun navigateToForm() {
      val transaction = fragmentManager.beginTransaction()
   transaction.add(R.id.ll_container, CreateUserFragment())
    transaction.commit()
}
```

Следующая функция вызывается, когда пользователь щелкает на кнопке возврата и если в стеке фрагментов имеется один или несколько фрагментов. В результате менеджер фрагментов извлекает фрагмент и отображает его для пользователя:

```
override fun onBackPressed() {
   if (fragmentManager.backStackEntryCount > 0) {
      fragmentManager.popBackStack()
      hideHomeButton()
   } else {
      super.onBackPressed()
   }
}

private fun navigateToList() {
   val transaction = fragmentManager.beginTransaction()
   transaction.replace(R.id.ll_container, ListUsersFragment())
   transaction.addToBackStack(null)
   transaction.commit()
}

override fun onOptionsItemSelected(item: MenuItem?): Boolean {
   val id = item?.itemId
   if (id == android.R.id.home) {
      onBackPressed()
      hideHomeButton()
   }
```

```
return super.onOptionsItemSelected(item)
}

class CreateUserFragment : Fragment(), MainView, View.OnClickListener {
...
    }
    class ListUsersFragment : Fragment(), MainView {

    }
}
```

Запустим приложение, чтобы увидеть, хорошо ли оно работает. Создайте и запустите проект на устройстве по вашему выбору. После запуска проекта вы увидите форму создания учетной записи пользователя. Введите в форму какую-либо информацию о пользователе (рис. 7.4).

После введения в форму создания пользователя верной информации щелкните на кнопке **SUBMIT** (Передать данные) для сохранения этого пользователя в базе данных приложения SQLite. Вас уведомят, когда учетная запись пользователя будет успешно сохранена. Получив уведомление, щелкните на кнопке **VIEW USERS** (Просмотр пользователей) для просмотра информации о пользователе, учетная запись которого только что сохранена (рис. 7.5).

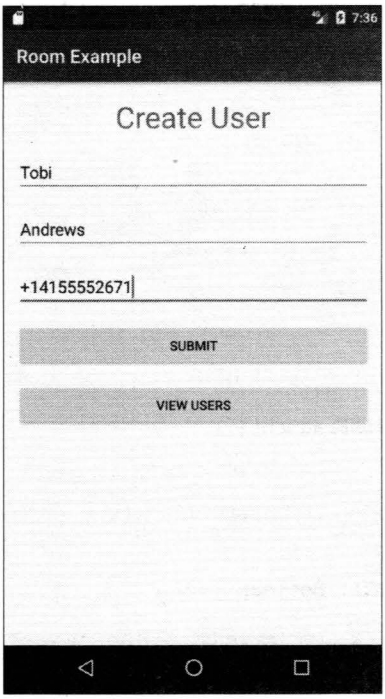


Рис. 7.4. Форма создания учетной записи пользователя

Рис. 7.5. Информация о пользователе, сохраненная в базе данных

Теперь вы можете создавать и просматривать информацию, относящуюся к любому числу пользователей. Не существует ограничения по количеству данных, входящих в базу данных!

Работа с контент-провайдерами

В *главе 2* вкратце рассматривались провайдеры контента как компоненты Android. При этом было отмечено, что провайдеры контента помогают приложению контролировать доступ к ресурсам данных, которые хранятся либо внутри приложения, либо в другом приложении. Кроме того, отмечалось, что провайдер контента облегчает обмен данными с другим приложением через открытый интерфейс прикладного программирования.

Поведение провайдера контента аналогично поведению базы данных. Провайдер контента позволяет вставлять, удалять, редактировать, обновлять и запрашивать контент. Эти возможности поддерживаются с помощью таких методов, как `insert()`, `update()`, `delete()` и `query()`. Во многих случаях контролируемые провайдером контента данные имеются в базе данных SQLite.

Провайдер контента для приложения можно создать, выполнив следующие пять простых шагов:

1. Создайте класс провайдера контента, который расширяет `ContentProvider`.
2. Определите URI-адрес контента.
3. Создайте источник данных, с которым будет взаимодействовать поставщик контента. Этот источник данных обычно представлен в форме базы данных SQLite. Если SQLite служит источником данных, необходимо создать `SQLiteOpenHelper` и переопределить его метод `onCreate()` для формирования базы данных, управляемой поставщиком контента.
4. Реализуйте необходимые методы провайдера контента.
5. Зарегистрируйте контент-провайдер в файле манифеста своего проекта.

Имеются шесть методов, которые должны реализовываться провайдером контента:

- `onCreate()` — этот метод вызывается для инициализации базы данных;
- `query()` — этот метод возвращает данные по запросу с помощью `Cursor`;
- `insert()` — этот метод вызывается для включения новых данных в провайдер контента;
- `delete()` — этот метод вызывается для удаления данных из провайдера контента;
- `update()` — этот метод вызывается для обновления данных в провайдере контента;
- `getType()` — при вызове этого метода провайдеру контента возвращается тип данных MIME.

Чтобы научиться работать с функциями провайдера контента, создадим пример проекта, где применяется провайдер контента и база данных SQLite. Создайте новый проект Android Studio под названием ContentProvider и внесите в него при создании пустое действие MainActivity. Подобно всем созданным в этой главе приложениям, этот пример несложен. Приложение разрешает пользователю вводить в текстовые поля информацию о продукте (название продукта и его производителя) и сохранять ее в базе данных SQLite. Пользователь затем может просматривать информацию о ранее сохраненных продуктах одним щелчком на кнопке. Измените файл activity_main.xml, включив в него следующий код XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:gravity="center_horizontal"
android:padding="16dp"
tools:context="com.example.contentproviderexample.MainActivity">
   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:gravity="center"
      android:text="@string/content_provider_example"
      android:textColor="@color/colorAccent"
      android:textSize="32sp"/>
   <EditText
      android:id="@+id/et_product_name"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_marginTop="16dp"
      android:hint="Product Name"/>
   <EditText
      android:id="@+id/et_product_manufacturer"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_marginTop="16dp"
      android:hint="Product Manufacturer"/>
   <Button
      android:id="@+id/btn_add_product"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_marginTop="16dp"
      android:text="Add product"/>
   <Button
      android:id="@+id/btn_show_products"
```

```
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_marginTop="16dp"
      android:text="Show products"/>
</LinearLayout>
```

После добавления указанных изменений добавьте в файл проекта strings.xml следующий строковый ресурс:

```
<string name="content_provider_example">Content Provider Example</string>
```

Теперь создайте в пакете com.example.contentproviderexample файл ProductProvider.kt и добавьте в него следующее содержимое:

```
package com.example.contentproviderexample

import android.content.*
import android.database.Cursor
import android.database.SQLException
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper
import android.database.sqlite.SQLiteQueryBuilder
import android.net.Uri
import android.text.TextUtils
internal class ProductProvider : ContentProvider() {

   companion object {
      val PROVIDER_NAME: String = "com.example.contentproviderexample
                                              .ProductProvider"
   val URL: String = "content://$PROVIDER_NAME/products"
   val CONTENT_URI: Uri = Uri.parse(URL)
   val PRODUCTS = 1
   val PRODUCT_ID = 2
   // Объявления базы данных и свойства таблицы
   val DATABASE_VERSION = 1
   val DATABASE_NAME = "Depot"
   val PRODUCTS_TABLE_NAME = "products"
   // Обьявления имени столбца таблицы 'products'
   val ID: String = "id"
   val NAME: String = "name"
   val MANUFACTURER: String = "manufacturer"
   val uriMatcher: UriMatcher = UriMatcher(UriMatcher.NO_MATCH)
   val PRODUCTS_PROJECTION_MAP: HashMap<String, String> = HashMap()
   SQLiteOpenHelper class that creates the content provider's database:
   private class DatabaseHelper(context: Context) :
   SQLiteOpenHelper(context, DATABASE_NAME, null,
   DATABASE_VERSION) {
```

```
        override fun onCreate(db: SQLiteDatabase) {
            val query = " CREATE TABLE " + PRODUCTS_TABLE_NAME +
                    " (id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                            " name VARCHAR(255) NOT NULL, " +
                        " manufacturer VARCHAR(255) NOT NULL);"
            db.execSQL(query)
        }

        override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
        newVersion: Int) {
            val query = "DROP TABLE IF EXISTS $PRODUCTS_TABLE_NAME"
            db.execSQL(query)
            onCreate(db)
        }
    }
}

private lateinit var db: SQLiteDatabase
override fun onCreate(): Boolean {
    uriMatcher.addURI(PROVIDER_NAME, "products", PRODUCTS)
    uriMatcher.addURI(PROVIDER_NAME, "products/#", PRODUCT_ID)
    val helper = DatabaseHelper(context)
```

Давайте воспользуемся `SQLiteOpenHelper` **для получения доступной для записи базы данных — новая база данных создается, если она еще не существует:**

```
    db = helper.writableDatabase
    return true
}

override fun insert(uri: Uri, values: ContentValues): Uri {
    // Вставка новой записи о продукте в таблицу products
    val rowId = db.insert(PRODUCTS_TABLE_NAME, "", values)
    // Если rowId больше 0, значит запись о продукте была
    // успешно добавлена
    if (rowId > 0) {
        val _uri = ContentUris.withAppendedId(CONTENT_URI, rowId)
        context.contentResolver.notifyChange(_uri, null)
        return _uri
    }

  // Генерирование исключения, если запись о продукте не была добавлена
    throw SQLException("Failed to add product into " + uri)
}

override fun query(uri: Uri, projection: Array<String>?,
                   selection: String?, selectionArgs: Array<String>?,
                   sortOrder: String): Cursor {
```

```
    val queryBuilder = SQLiteQueryBuilder()
    queryBuilder.tables = PRODUCTS_TABLE_NAME
    when (uriMatcher.match(uri)) {
        PRODUCTS -> queryBuilder.setProjectionMap(PRODUCTS_PROJECTION_MAP)
        PRODUCT_ID -> queryBuilder.appendWhere(
                "$ID = ${uri.pathSegments[1]}"
    )
}

val cursor: Cursor = queryBuilder.query(db, projection, selection,
                              selectionArgs, null, null, sortOrder)
cursor.setNotificationUri(context.contentResolver, uri)
return cursor
}

override fun delete(uri: Uri, selection: String,
            selectionArgs: Array<String>): Int {
    val count = when(uriMatcher.match(uri)) {
        PRODUCTS -> db.delete(PRODUCTS_TABLE_NAME, selection, selectionArgs)
        PRODUCT_ID -> {
            val id = uri.pathSegments[1]
            db.delete(PRODUCTS_TABLE_NAME, "$ID = $id " +
            if (!TextUtils.isEmpty(selection)) "AND
                ($selection)" else "", selectionArgs)
        }
        else -> throw IllegalArgumentException("Unknown URI: $uri")
    }

    context.contentResolver.notifyChange(uri, null)
    return count
}

override fun update(uri: Uri, values: ContentValues, selection: String,
                                  selectionArgs: Array<String>): Int {
    val count = when(uriMatcher.match(uri)) {
        PRODUCTS -> db.update(PRODUCTS_TABLE_NAME, values,
                                selection, selectionArgs)
        PRODUCT_ID -> {
            db.update(PRODUCTS_TABLE_NAME, values,
            "$ID = ${uri.pathSegments[1]} " +
                if (!TextUtils.isEmpty(selection)) " AND
                    ($selection)" else "", selectionArgs)
            }
            else -> throw IllegalArgumentException("Unknown URI: $uri")
        }
```

```
        context.contentResolver.notifyChange(uri, null)
    return count
    }

override fun getType(uri: Uri): String {
    // Возврат соответствующего типа MIME для записей
    return when (uriMatcher.match(uri)){
        PRODUCTS -> "vnd.android.cursor.dir/vnd.example.products"
        PRODUCT_ID -> "vnd.android.cursor.item/vnd.example.products"
        else -> throw IllegalArgumentException("Unpermitted URI: " + uri)
      }
    }
}
```

После добавления подходящего `ProductProvider` для поддержки контента, относящегося к сохраненным продуктам, в файле `AndroidManifest` необходимо зарегистрировать новый компонент. Следующий фрагмент кода добавляет провайдер в файл манифеста:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.contentproviderexample">

<application
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
android:label="@string/app_name"
android:roundIcon="@mipmap/ic_launcher_round"
android:supportsRtl="true"
android:theme="@style/AppTheme">
<activity android:name=".MainActivity">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<provider android:authorities="com.example.contentproviderexample
   .ProductProvider" android:name="ProductProvider"/>
</application>
</manifest>
```

Теперь модифицируем действие `MainActivity` для применения этого недавно зарегистрированного провайдера. Изменим файл MainActivity.kt для включения в него следующего кода:

```
package com.example.contentproviderexample

import android.content.ContentValues
import android.net.Uri
```

```
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.text.TextUtils
import android.view.View
import android.widget.Button
import android.widget.EditText
import android.widget.Toast

class MainActivity : AppCompatActivity(), View.OnClickListener {
private lateinit var etProductName: EditText
private lateinit var etProductManufacturer: EditText
private lateinit var btnAddProduct: Button
private lateinit var btnShowProduct: Button

override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)
   bindViews()
   setupInstances()
}

private fun bindViews() {
   etProductName = findViewById(R.id.et_product_name)
   etProductManufacturer = findViewById(R.id.et_product_manufacturer)
   btnAddProduct = findViewById(R.id.btn_add_product)
   btnShowProduct = findViewById(R.id.btn_show_products)
}

private fun setupInstances() {
   btnAddProduct.setOnClickListener(this)
   btnShowProduct.setOnClickListener(this)
   supportActionBar?.hide()
}

private fun inputsValid(): Boolean {
   var inputsValid = true
   if (TextUtils.isEmpty(etProductName.text)) {
      etProductName.error = "Field required."
      etProductName.requestFocus()
      inputsValid = false
   } else if (TextUtils.isEmpty(etProductManufacturer.text)) {
      etProductManufacturer.error = "Field required."
      etProductManufacturer.requestFocus()
      inputsValid = false
   }
```

```
    return inputsValid
}

private fun addProduct() {
    val contentValues = ContentValues()
    contentValues.put(ProductProvider.NAME, etProductName.text.toString())
    contentValues.put(ProductProvider.MANUFACTURER,
    etProductManufacturer.text.toString())
    contentResolver.insert(ProductProvider.CONTENT_URI, contentValues)
    showSaveSuccess()
}
```

Следующая функция вызывается для отображения находящихся в базе данных продуктов:

```
private fun showProducts() {
    val uri = Uri.parse(ProductProvider.URL)
    val cursor = managedQuery(uri, null, null, null, "name")
    if (cursor != null) {
        if (cursor.moveToFirst()) {
            do {
            val res = "ID: ${cursor.getString(cursor.getColumnIndex
            (ProductProvider.ID))}" + ",
            \nPRODUCT NAME: ${cursor.getString(cursor.getColumnIndex
            ( ProductProvider.NAME))}" + ",
            \nPRODUCT MANUFACTURER: ${cursor.getString(cursor.getColumnIndex
            (ProductProvider.MANUFACTURER))}"
            Toast.makeText(this, res, Toast.LENGTH_LONG).show()
        } while (cursor.moveToNext())
    }
} else {
    Toast.makeText(this, "Oops, something went wrong.",
                            Toast.LENGTH_LONG).show()
    }
}

private fun showSaveSuccess() {
    Toast.makeText(this, "Product successfully saved.",
                            Toast.LENGTH_LONG).show()
    }

    override fun onClick(view: View) {
        val id = view.id
        if (id == R.id.btn_add_product) {
            if (inputsValid()) {
                addProduct()
            }
```

```
        } else if (id == R.id.btn_show_products) {
            showProducts()
        }
    }
}
```

В приведенном блоке кода следует сосредоточить внимание на двух методах: `addProduct()` и `showProducts()`. Метод `addProduct()` сохраняет данные о продукте в экземпляре `contentValues`, а затем вставляет эти данные в базу данных SQLite с помощью `ProductProvider`, вызывая `contentResolver.insert(ProductProvider.CONTENT_URI, contentValues)`. Метод `showProducts()` применяет `Cursor` для отображения хранящейся в базе данных информации о продукте с помощью всплывающих сообщений.

Теперь, уже представляя, что происходит, скомпилируйте и запустите приложение, как вы это делали до сих пор, дождитесь установки и запуска. Вы попадете прямо в действие `MainActivity` и получите доступ к форме для ввода названия и производителя продукта (рис. 7.6).

После ввода корректной информации о продукте щелкните на кнопке **ADD PRODUCT** (Добавить продукт) — продукт будет добавлен в виде новой записи в таблицу продуктов базы данных SQLite приложения. Внесите еще несколько про-

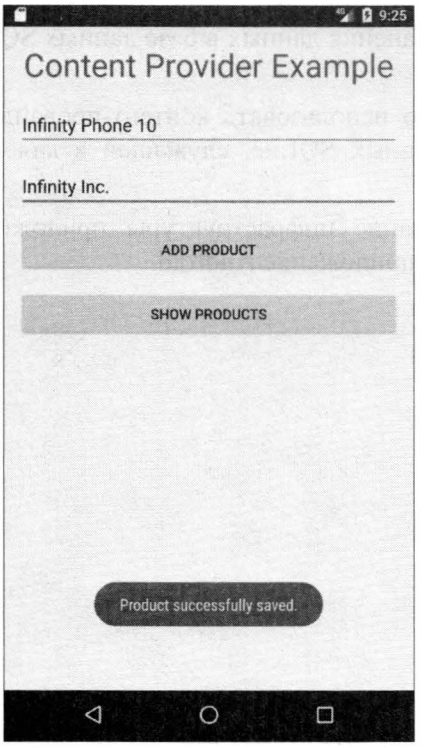


Рис. 7.6. Форма для ввода названия и производителя продукта

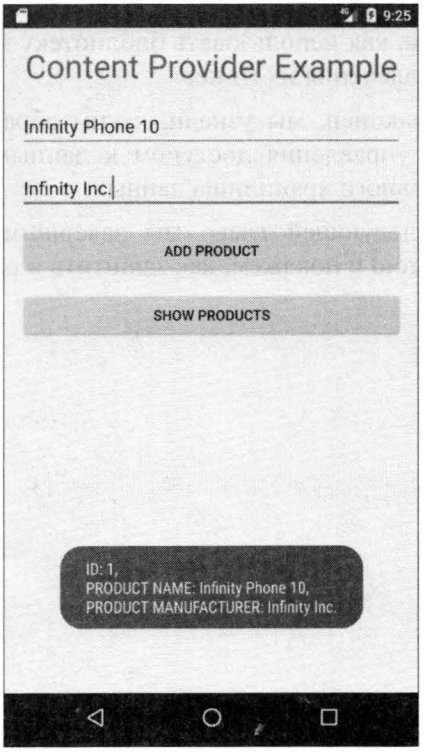


Рис. 7.7. Вывод информации о продукте из базы данных приложения

дуктов с помощью этой формы и щелкните на кнопке **SHOW PRODUCTS** (Показать продукты).

Это приведет к вызову методов `showProducts()` в `MainActivity`. Все записи о продуктах последовательно выбираются и отображаются во всплывающих сообщениях (рис. 7.7).

Как раз это и нужно было реализовать в примере приложения для того, чтобы продемонстрировать, как работают контент-провайдеры. Сделайте приложение более привлекательным, добавив ему функциональности в части обновления и удаления записей продуктов. Благодаря этому вы хорошо разберетесь в сути происходящего!

Подведем итоги

В этой главе подробно изучены различные способы сохранения данных, которые предоставляет платформа приложений Android. Рассмотрено, каким образом можно использовать внутреннее и внешнее хранилище для хранения данных в частных и общедоступных файлах. Показано, каким образом можно работать с файлами кэша с помощью как внутреннего, так и внешнего хранилища.

В этой главе также содержатся сведения о СУБД SQLite и показано, каким образом можно задействовать ее в приложениях для Android. Мы познакомились также с тем, как использовать библиотеку `Room` для хранения данных в базе данных SQLite и извлечения их из нее.

И, наконец, мы узнали, каким образом можно использовать контент-провайдеры для управления доступом к данным базы данных SQLite, служащей в качестве основного хранилища данных.

В следующей главе мы завершим исследование инфраструктуры приложений Android и покажем, как защитить и развернуть приложение Android.

8

Защита и развертывание приложений Android

В предыдущих главах рассматривалось применение языка Kotlin в области разработки мобильных приложений, при этом основное внимание уделялось платформе Android. В *главе 7* мы познакомились с различными средствами хранения данных, предоставляемыми для разработки приложений на платформе приложений Android: внутренними, внешними и сетевыми хранилищами и базой данных SQLite, а также с разработкой программ, использующих эти средства. Было рассказано и о применении библиотеки `Room` и провайдеров контента для хранения и извлечения данных из базы данных SQLite.

Завершая в этой главе исследование платформы Android, мы сосредоточим свое внимание на двух чрезвычайно важных темах:

- безопасности приложений Android;
- развертывании приложений Android.

И начнем мы с вопросов обеспечения безопасности приложений Android.

Обеспечение безопасности приложения Android

Обеспечение безопасности создаваемого программного обеспечения является весьма важным фактором. Помимо принятых в операционной системе Android мер безопасности, важно, чтобы разработчики уделяли внимание обеспечению соответствия приложений установленным стандартам безопасности. В этом разделе содержится ряд важных советов по части мер безопасности и рекомендации по их применению. Следование этим рекомендациям сделает ваши приложения менее уязвимыми для вредоносных программ, которые могут устанавливаться на клиентском устройстве.

Хранение данных

При прочих равных условиях конфиденциальность сохраняемых приложением на устройстве данных является наиболее распространенной проблемой для мер безопасности при разработке приложения для Android. При соблюдении некоторых простых правил данные вашего приложения окажутся в большей безопасности.

Использование внутреннего хранилища

Как показано в предыдущей главе, внутреннее хранилище обеспечивает хороший способ хранения частных данных на устройстве. Каждое приложение Android имеет соответствующий внутренний каталог хранения, в котором создаются и в который записываются частные файлы. Поскольку эти файлы являются частными для создаваемого приложения, они недоступны для других приложений на клиентском устройстве. Как правило, если данные должны быть доступны только вашему приложению и их можно хранить во внутреннем хранилище, обязательно там их и храните. Обратитесь к предыдущей главе, где показано, каким образом можно использовать внутреннее хранилище.

Использование внешнего хранилища

Файлы внешнего хранилища не являются частными для приложений и поэтому могут стать доступными для других приложений на том же клиентском устройстве. Поэтому следует рассмотреть возможность *шифрования данных приложения* перед его сохранением во внешнем хранилище. Имеется ряд библиотек и пакетов, которые можно использовать для шифрования данных перед их сохранением во внешнем хранилище. Так, библиотека `Conceal` в Facebook (**http://facebook.github.io/conceal/**) — хороший вариант для шифрования данных, размещаемых во внешнем хранилище.

И, кроме этого, не забывайте также о полезном практическом правиле — *не храните конфиденциальные данные во внешнем хранилище*. Важность соблюдения этого правила объясняется тем, что файлами внешнего хранилища можно свободно манипулировать. В силу этого входящие данные, извлеченные из внешнего хранилища, должны проверяться в обязательном порядке.

Использование провайдеров контента

Как известно из предыдущей главы, провайдеры контента могут либо запретить, либо открыть внешний доступ к данным приложения. Применяйте атрибут `android:exported` при регистрации провайдера контента в файле манифеста для открытия или запрета внешнего доступа к провайдеру контента. Присвойте атрибуту `android:exported` значение `true`, если необходимо, чтобы провайдер контента экспортировался, в противном случае установите для этого атрибута значение `false`.

В дополнение к этому — для предотвращения внедрения SQL (метода внедрения кода, который предполагает исполнение злоумышленником вредоносных SQL-ин-

струкций, находящихся в поле ввода) — следует использовать методы запроса провайдера контента, например: `query()`, `update()` и `delete()`.

Сетевая безопасность

Существует ряд рекомендаций, которые следует соблюдать при выполнении с помощью приложений Android сетевых транзакций. В этом разделе мы рассмотрим такие категории, как протокол Интернета (Internet Protocol, IP) и сетевую телефонию, представляющие собой лучшие способы работы с сетями.

IP-сети

При обмене данными с удаленным компьютером по IP важно гарантировать, чтобы приложение применяло *протокол HTTPS* везде, где это возможно (т. е. всюду, где он поддерживается на сервере). Одна из основных причин появления такого требования заключается в том, что устройства зачастую подключаются к незащищенным сетям, таким как общедоступные беспроводные соединения. Протокол HTTPS обеспечивает зашифрованную связь между клиентами и серверами, независимо от сети, к которой они подключены. При работе с Java для безопасной передачи данных по сети применяется `HttpsURLConnection`. Важно также отметить, что не следует доверять данным, полученным через незащищенное сетевое соединение.

Сетевая телефония

В тех случаях, когда данные должны свободно передаваться через сервер и клиентские приложения, наряду с IP-сетями — вместо таких возможностей, как протокол службы обмена короткими сообщениями (Short Messaging Service, SMS), — следует обратиться к *протоколу обмена сообщениями в облаке Firebase* (Firebase Cloud Messaging, FCM). Благодаря FCM обеспечивается многоплатформенное решение для обмена сообщениями, которое гарантирует бесперебойную и надежную передачу сообщений между приложениями.

Протокол SMS не следует использовать в качестве надежного средства передачи сообщений с данными, поскольку этот протокол:

- не зашифрован;
- не является строго аутентифицированным;
- сообщения, отправленные с помощью SMS, могут быть подделаны;
- SMS-сообщения могут быть перехвачены.

Валидация вводимых данных

Валидация вводимых данных чрезвычайно важна, поскольку позволяет избежать возможных угроз безопасности. Один из подобных рисков, как объясняется в *разд. «Использование провайдеров контента»*, — внедрение кода SQL. Внедрение вредоносного сценария SQL можно предотвратить с помощью параметризованных

запросов и обширной очистки входных данных, которые применяются в исходных запросах SQL.

В дополнение к этому извлеченные из внешнего хранилища входные данные должны быть проверены надлежащим образом, поскольку внешнее хранилище не является надежным источником данных.

Работа с учетными данными пользователя

Риск фишинга — мошеннического получения доступа к конфиденциальным данным пользователей (логинам и паролям) — можно уменьшить, отказавшись от требования по вводу учетных данных пользователя в приложении. Вместо постоянного запроса учетных данных пользователя рассмотрите возможность применения *токена авторизации*. Исключите необходимость сохранения на устройстве имен пользователей и паролей. Вместо этого применяйте обновляемый токен авторизации.

Запутывание кода

Перед публикацией приложения Android обязательно используйте *инструмент запутывания кода* — такой как ProGuard, чтобы не позволять беспрепятственный доступ к вашему исходному коду с помощью различных средств типа инструментов декомпиляции. ProGuard предварительно упакован в Android SDK, и, следовательно, включение зависимостей ему не требуется. Этот продукт автоматически включается в процесс сборки, если указать тип сборки соответствующим дате выпуска. Дополнительные сведения о ProGuard можно получить по адресу: **https://www.guardsquare.com/en/proguard**.

Защита широковещательных приемников

По умолчанию компонент широковещательного приемника экспортируется и затем может вызываться на том же устройстве другими приложениями. Контролировать доступность приложений для их получателей можно с помощью *разрешений безопасности*. Разрешения для широковещательных приемников можно установить в файле манифеста приложения с помощью элемента `<receiver>`.

Динамически загружаемый код

В сценариях, где необходима динамическая загрузка кода вашим приложением, следует убедиться, что загружаемый код поступает из надежного источника. В дополнение к этому следует любой ценой уменьшить риск фальсификации кода. Загрузка и выполнение сфальсифицированного кода представляет большую угрозу для безопасности. Если код загружается с удаленного сервера, убедитесь, что он передается по защищенной зашифрованной сети. Динамически загружаемый код

запускается с теми же разрешениями безопасности, что и ваше приложение (разрешения определяются в файле манифеста вашего приложения)[1].

Службы обеспечения безопасности

В отличие от широковещательных приемников, службы обеспечения безопасности по умолчанию не экспортируются системой Android. По умолчанию экспорт таких служб выполняется только при добавлении к объявлению сервиса в файле манифеста фильтра намерений. Для поддержки экспорта служб обеспечения безопасности должен применяться атрибут `android:exported`. Присвойте атрибуту `android:exported` значение `true`, чтобы обеспечить выполнение экспорта служб, и значение `false` — в противном случае.

Запуск и публикация Android-приложения

К настоящему моменту подробно рассмотрена система Android, разработка приложений для Android и некоторые другие важные темы, в том числе и обеспечение безопасности приложений Android. Теперь пришло время перейти к последней теме этой книги, касающейся экосистемы Android, — рассмотреть запуск и публикацию приложения Android.

Возможно, вам интересно узнать, что означают слова «запуск и публикация». *Запуск* — это деятельность, которая включает в себя представление нового продукта для общественности (конечных пользователей). *Публикация* приложения для Android — процесс предоставления пользователям приложения для Android. Для успешного запуска приложения Android необходимо выполнить ряд действий, приведенных в следующем списке:

- уточнение политик разработчиков программ для Android;
- подготовка учетной записи разработчика Android;
- планирование локализации;
- планирование одновременного выпуска;
- тестирование на соответствие руководству по качеству;
- создание подготовленного к выпуску пакета приложения (APK);
- подготовка списка ресурсов вашего приложения для его включения в Play Store (Магазин приложений);
- загрузка пакета приложения на альфа- или бета-канал;
- определение совместимости устройства;
- предварительные отчеты;

[1] Следует учитывать, что в последнее время корпорация Google запретила публиковать в каталоге Google Play приложения с динамически загружаемым кодом (*Прим. ред.*).

- ценообразование и настройка распространения приложения;
- выбор варианта распространения;
- настройка продуктов и подписок в приложении;
- определение рейтинга контента вашего приложения;
- публикация вашего приложения.

Наконец-то! Это довольно-таки длинный список. Не волнуйтесь, если какой-то пункт из списка вам не понятен. Сейчас мы рассмотрим все пункты подробнее.

Уточнение политик разработчиков программ для Android

Имеется набор политик для разработчиков программ, которые созданы исключительно с той целью, чтобы Play Store (Магазин приложений) оставался надежным источником программного обеспечения для пользователей. При нарушении этих определенных политик возникают последствия. Важно, чтобы вы внимательно изучили и полностью приняли эти политики разработчика — представили цели их и последствия, а затем продолжили процесс запуска приложения.

Подготовка учетной записи разработчика Android

Для запуска приложения в Play Store (Магазин приложений) вам необходима учетная запись разработчика Android. Убедитесь, что она создана, для чего зарегистрируйтесь как разработчик и подтвердите правильность данных своей учетной записи. Если вам придется в своем приложении для Android продавать продукты, необходимо создать учетную запись продавца.

Планирование локализации

Иногда в целях локализации вы можете создать несколько копий приложения, каждое из которых локализовано для определенного языка. Тогда необходимо заранее спланировать локализацию и следовать рекомендованному контрольному списку локализации для разработчиков Android. Этот контрольный список можно просмотреть на сайте по адресу: **https://developer.android.com/distribute/best-practices/launch/localization-checklist.html.**

Планирование одновременного выпуска

Возможно, что вы пожелаете запустить продукт на нескольких платформах. Это связано с рядом преимуществ, среди которых: расширение потенциального рынка для вашего продукта, уменьшение барьера доступа к продукту и максимизация числа потенциальных установок вашего приложения. Одновременный выпуск на многочисленных платформах, как правило, отличная идея. Если необходимо реали-

зовать это с каким-либо продуктом, удостоверьтесь, что вы планируете это заранее. Если невозможно одновременно запустить приложение на нескольких платформах, убедитесь, что предоставлены средства, с помощью которых заинтересованные потенциальные пользователи смогут предоставить контактные данные, что позволит вам с ними связаться, как только продукт станет доступным на выбранной ими платформе.

Тестирование на соответствие руководству по качеству

Рекомендации по обеспечению качества предоставляют шаблоны тестирования, которые можно применять для подтверждения того, что ваше приложение соответствует основным функциональным и нефункциональным требованиям, которые составляют предмет ожиданий пользователей Android. Перед запуском убедитесь, что вы запускаете приложение с учетом этих руководств по качеству. Доступ к руководствам по качеству приложений можно получить на сайте по адресу: **https://developer.android.com/develop/quality-guidelines/index.html**.

Создание подготовленного к выпуску пакета приложения (APK)

Подготовленный к выпуску пакет APK — это приложение Android, которое упаковано с применением оптимизации, а затем создано и подписано с помощью ключа выпуска. Создание подготовленного к выпуску APK — важный шаг при запуске приложения для Android. Обратите на этот шаг особое внимание.

Подготовка списка ресурсов вашего приложения для его включения в Play Store

Этот шаг предполагает подбор всех ресурсов, необходимых для включения вашего продукта в Play Store (Магазин приложений). Среди этих ресурсов присутствуют журнал вашего приложения, экранные снимки, описания, рекламная графика и видео, если таковые имеются, но список этим не ограничивается. Убедитесь, что в список ресурсов вашего приложения включена ссылка на политику конфиденциальности приложения. Важно также локализовать список ресурсов приложения на всех языках, которые поддерживает ваше приложение.

Загрузка пакета приложения на альфа- или бета-канал

Поскольку тестирование — эффективный и проверенный способ обнаружения дефектов в программном обеспечении, ведущий к улучшению его качества, рекомендуется загрузить пакет приложения в альфа- и бета-каналы, что облегчит выполнение альфа- и бета-тестирования этого продукта. Альфа-тестирование и бета-тестирование — это два типа приемочного тестирования приложений.

Определение совместимости устройства

Здесь имеется в виду объявление версий Android и размеров экрана, с учетом которых разработано приложение. На этом этапе важно быть максимально точным, поскольку неточное указание версий Android и размеров экрана неизбежно приведет к дополнительным проблемам для пользователей при работе с вашим приложением.

Предварительные отчеты

Предварительные отчеты применяются для выявления проблем, обнаруженных после автоматического тестирования приложения на различных устройствах Android. Отчеты перед запуском будут доставлены, если включить их при загрузке пакета приложения на альфа- или бета-канал.

Ценообразование и настройка распространения приложения

Сначала определите, каким способом вы желаете монетизировать приложение. С учетом этого настройте приложение либо на бесплатную установку, либо на платную загрузку. После указания цены приложения выберите страны, где желательно распространить приложение.

Выбор варианта распространения

Этот шаг предполагает подбор устройств и платформ, например Android TV и Android Wear, с помощью которых будет распространяться приложение. После этого команда Google Play рассмотрит ваше приложение. Если приложение будет одобрено, Google Play сделает его доступным для обнаружения.

Настройка продуктов и подписок в приложении

Если в рамках своего приложения вы желаете продавать продукты, необходимо в приложении определить эти продукты и подписки. Укажите страны, где вы сможете продавать продукты и разрешать различные финансовые вопросы, такие как налоговые отчисления. На этом этапе также настройте торговую учетную запись.

Определение рейтинга контента вашего приложения

Необходимо указать точную оценку для приложения, если вы публикуете его в Play Store (Магазин приложений). Этот шаг обязателен в рамках политики программы Android Developer по одной важной причине — он поможет соответствующей возрастной группе, на которую вы ориентируетесь, обнаружить ваше приложение.

Публикация вашего приложения

Будем считать, что вы прошли указанные предварительные шаги и теперь готовы опубликовать свое приложение в производственном канале Play Store (Магазин приложений). Осталось создать выпуск, который позволит загружать APK-файлы приложения и развертывать приложение в определенной теме. В конце процедуры создания выпуска вы сможете опубликовать приложение, щелкнув на кнопке **Confirm rollout** (Подтвердить развертывание).

Итак, это все, что нужно знать для публикации нового приложения в Play Store. В большинстве случаев нет необходимости проходить все шаги последовательно — просто выбирайте те из них, что относятся к типу приложения, которое вы собираетесь опубликовать. Как всегда, лучший способ понять последовательность шагов — рассмотреть пример. Поэтому в этом разделе мы опубликуем в магазине Play Store разработанное в предыдущих главах приложение Messenger. Точно так же можно опубликовать и любые разработанные вами приложения.

Создание учетной записи разработчика Google Play

Первое, что требуется при публикации приложения в Play Store, — это создать учетную запись разработчика Google Play. Это несложно сделать, и мы рассмотрим данный процесс чуть позже. Сейчас же выберите удобный веб-браузер и перейдите по следующему адресу: **https://play.google.com/apps/publish/signup**.

После открытия веб-страницы вы получите предложение войти в свою учетную запись Google. Войдя в учетную запись Google (рис. 8.1), необходимо принять соглашение о программе разработчика.

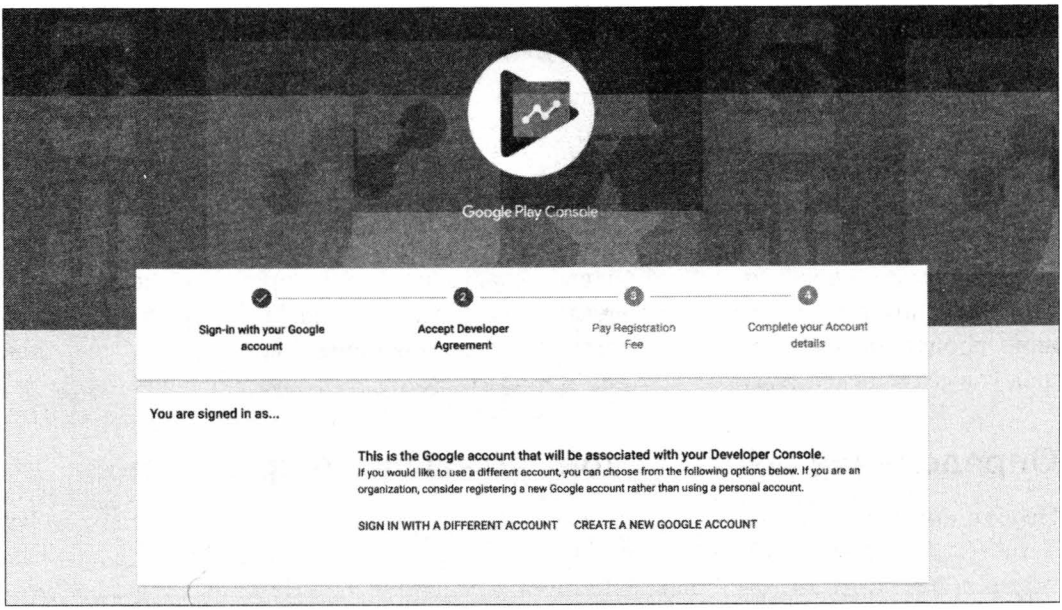


Рис. 8.1. Вошли в учетную запись Google

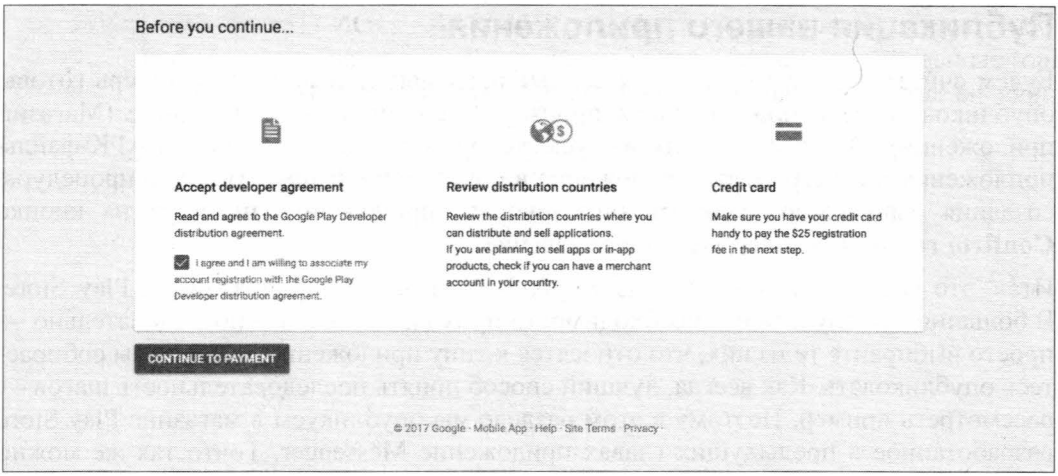


Рис. 8.2. Принятие соглашения с Google Play Developer

Примите соглашение с Google Play Developer (рис. 8.2), прокрутив страницу вниз и установив флажок **I agree and I am willing to associate my account registration with the Google Play Developer distribution agreement** (*Я* согласен, и я хочу связать регистрацию своей учетной записи с соглашением о распространении Google Play Developer).

После подписания соглашения щелкните на кнопке **CONTINUE TO PAYMENT** (Продолжить для оплаты) для продолжения процесса создания учетной записи разработчика, на следующем шаге которого вам необходимо будет заплатить единовременный регистрационный взнос в Google Play для разработчиков в размере 25 долларов США. Вы пройдете через процесс оплаты без проблем. После успешного выполнения платежа вам будет предложено продолжить регистрацию (рис. 8.3).

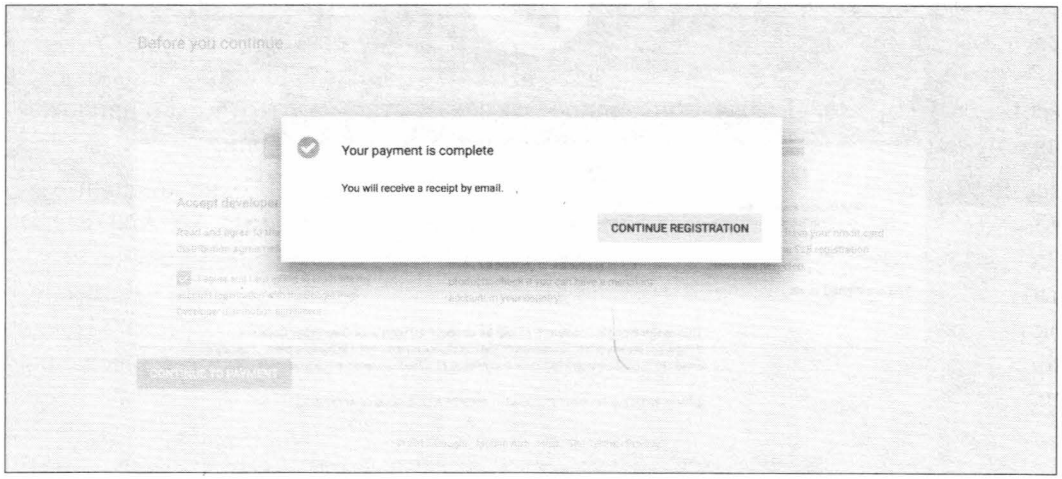


Рис. 8.3. Щелкните на кнопке **CONTINUE REGISTRATION**

После щелчка на кнопке **CONTINUE REGISTRATION** (Продолжить регистрацию) вы перейдете к последнему этапу процесса регистрации, на котором получите запрос на заполнение данных вашей учетной записи (рис. 8.4).

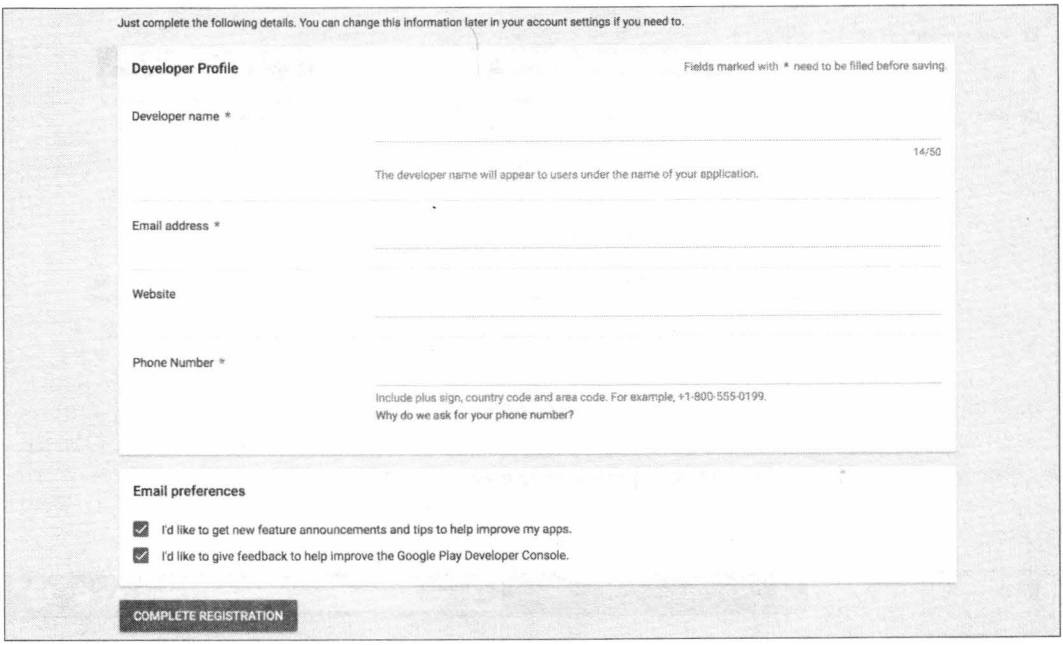


Рис. 8.4. Заполните данные для вашей учетной записи

Введите необходимые данные учетной записи и щелкните на кнопке **COMPLETE REGISTRATION** (Завершить регистрацию) для завершения процесса регистрации учетной записи.

После завершения регистрации вас перенаправят на консоль разработчика Google Play (рис. 8.5). Отсюда можно управлять своими приложениями, использовать игровые службы Google Play, контролировать заказы, загружать отчеты о приложениях, просматривать оповещения и управлять настройками консоли.

Рассмотрим теперь процесс публикации Android-приложения. В окне консоли разработчика Google Play (см. рис. 8.5) щелкните на кнопке **PUBLISH AN ANDROID APP ON GOOGLE PLAY** (Опубликовать приложение Android на Google Play). Выберите в открывшемся окне язык по умолчанию и введите `Messenger` как заголовок приложения при запросе, затем щелкните на кнопке **CREATE** (Создать). После этого на консоли **Developer** (Разработчик) создается новый проект приложения (рис. 8.6).

Но прежде чем приступить к процессу публикации приложения, необходимо подписать выпуск APK для приложения Messenger.

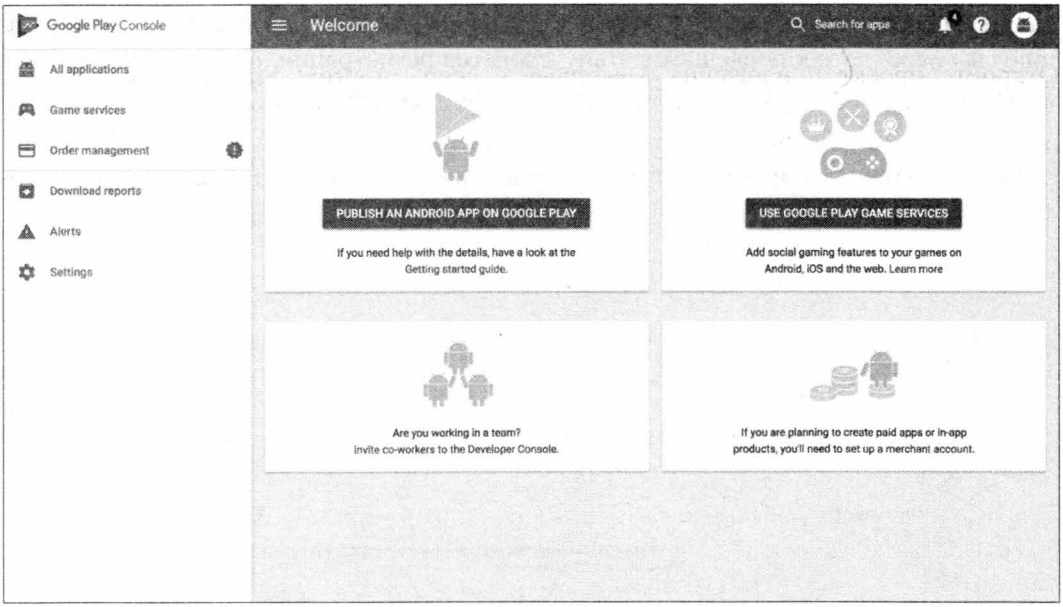


Рис. 8.5. Окно консоли разработчика Google Play

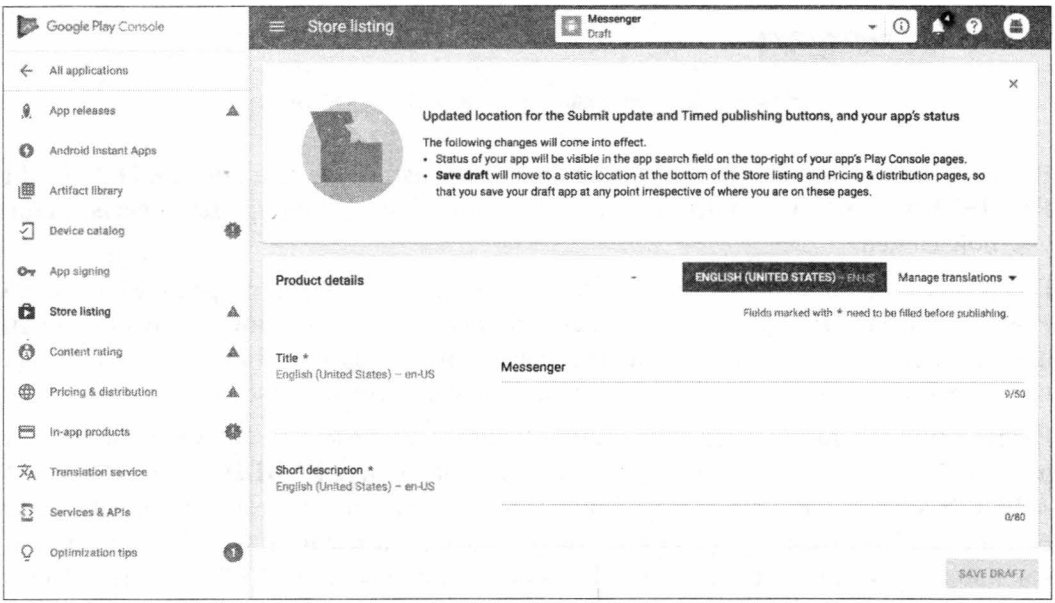


Рис. 8.6. Новый проект приложения на консоли **Developer**

Создание подписи приложения для выпуска

Откройте проект приложения Messenger в среде Android Studio. И хотя среда Android Studio не является единственным способом для подписи приложений, именно этот путь используется в этой главе для подписи приложения Messenger. Прежде всего сгенерируйте частный (закрытый) ключ для подписи, выполнив следующую команду в терминале Android Studio:

```
keytool -genkey -v -keystore my-release-key.jks -keyalg
   RSA -keysize 2048 -validity 10000 -alias my-alias
```

При выполнении этой команды вам предложат ввести пароль `keystore`, а также предоставить дополнительную информацию для вашего ключа. Затем генерируется ключ `keystore` в виде файла под именем my-release-key.jks, который и сохраняется в текущем каталоге. Ключ, содержащийся в `keystore`, действителен на протяжении 10 000 дней.

Теперь, после генерирования закрытого ключа, настроим Gradle для подписи APK. Откройте файл модульного уровня build.gradle и добавьте блок `signingConfigs {}` в блок `android {}` — с записями для `storeFile`, `storePassword`, `keyAlias` и `keyPassword`. После этого передайте этот объект свойству `signingConfig` в сборке вашего приложения. Используйте следующий фрагмент в качестве примера:

```
android {
   compileSdkVersion 26
   buildToolsVersion "26.0.2"
   defaultConfig {
      applicationId "com.example.messenger"
      minSdkVersion 16
      targetSdkVersion 26
      versionCode 1
      versionName "1.0"
      testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
      vectorDrawables.useSupportLibrary = true
   }

   signingConfigs {
      release {
         storeFile file("../my-release-key.jks")
         storePassword "password"
         keyAlias "my-alias"
         keyPassword "password"
      }
   }
   buildTypes {
      release {
         minifyEnabled false
```

```
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
    'proguard-rules.pro'
        signingConfig signingConfigs.release
      }
    }
}
```

После выполнения этих действий можно подписывать APK, но прежде чем сделать это, следует изменить текущее имя пакета. Имя пакета com.example может применяться только в Google Play, поэтому следует изменить имя пакета, прежде чем публиковать приложение в Play Store. Не беспокойтесь — с помощью Android Studio легко изменить имя корневого пакета приложения. Во-первых, убедитесь, что среда Android Studio настроена для отображения структуры каталогов проекта. Это можно реализовать, щелкнув на раскрывающемся списке в верхней левой части окна IDE и выбрав **Project** (Проект) (рис. 8.7).

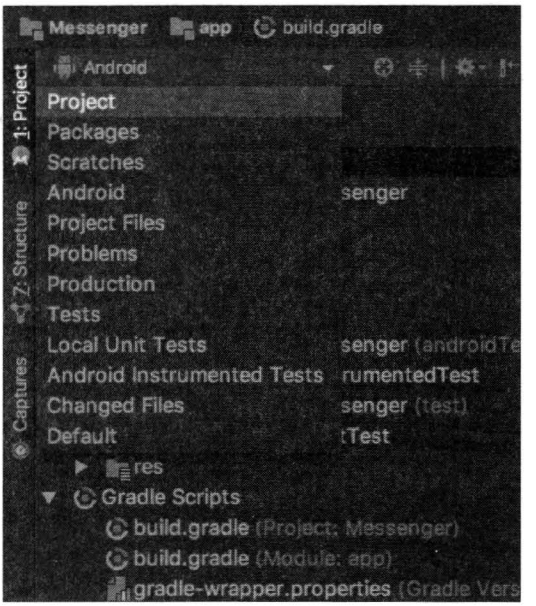


Рис. 8.7. Структура каталогов проекта

После выполнения указанных действий отмените скрытие всех пустых промежуточных пакетов в представлении структуры проекта, сняв флажок у параметра **Hide Empty Middle Packages** (Скрыть пустые промежуточные пакеты) в меню настроек структуры проекта (рис. 8.8).

После этого пустые промежуточные пакеты больше не будут сокрыты, и пакет com.example.messenger разделится на три видимых пакета: com, example и messenger. Переименуйте пакет example произвольным образом — измените, например, имя example на имя, полученное из комбинации ваших имени и фамилии. То есть если ваши имя и фамилия Kevin Fakande, имя пакета example будет переименовано на

kevinfakande. Пакет можно переименовать, щелкнув на нем правой кнопкой мыши и выбрав команды **Refactor | Rename** (Рефакторинг | Переименовать). После переименования пакета проверьте ваш манифест и файлы build.gradle, чтобы убедиться, что в них отражено изменение пакета вашего проекта. То есть где бы ни находилась строка com.example.messenger — либо в файлах build.gradle, либо в `manifest` — измените ее на `com.{full_name}.messenger`.

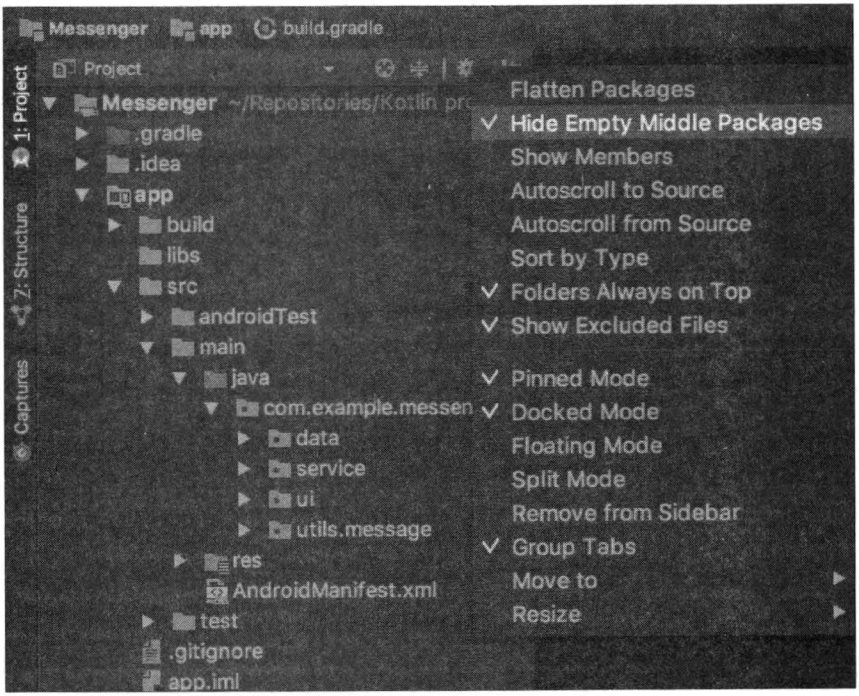


Рис. 8.8. Снимите флажок у параметра **Hide Empty Middle Packages**

Выполнив указанные изменения, можно подписать заявку. В окне терминала Android Studio введите следующую команду:

```
./gradlew assembleRelease
```

В результате выполнения этой команды будет создан выпуск APK, подписанный вашим частным (закрытым) ключом в пути <project_name>/<module_name>/build/outputs/apk/release. APK получит наименование <module_name>-release.apk. Поскольку модуль в этом проекте называется app, APK будет именоваться app-release.apk. APK, подписанные частным (закрытым) ключом, готовы к распространению. После подписи APK можно завершить публикацию приложения Messenger.

Выпуск приложения для Android

Подписав приложение Messenger, можно приступить к заполнению необходимых сведений о приложении для выпуска приложения. Во-первых, надо создать подхо-

дящий список магазинов для приложения. Откройте приложение Messenger в консоли Google Play и перейдите на страницу со списком магазинов — это можно выполнить, выбрав на боковой панели навигации опцию **Store Listing** (см. рис. 8.6).

На странице списка магазинов консоли Google Play (рис. 8.9) следует заполнить всю необходимую информацию. В нее входят сведения о продукте, такие как его название, краткое описание, полное описание, а также графические ресурсы и информация о категоризации, включая тип приложения, его категорию и рейтинг контента, контактные данные и указание на политику конфиденциальности.

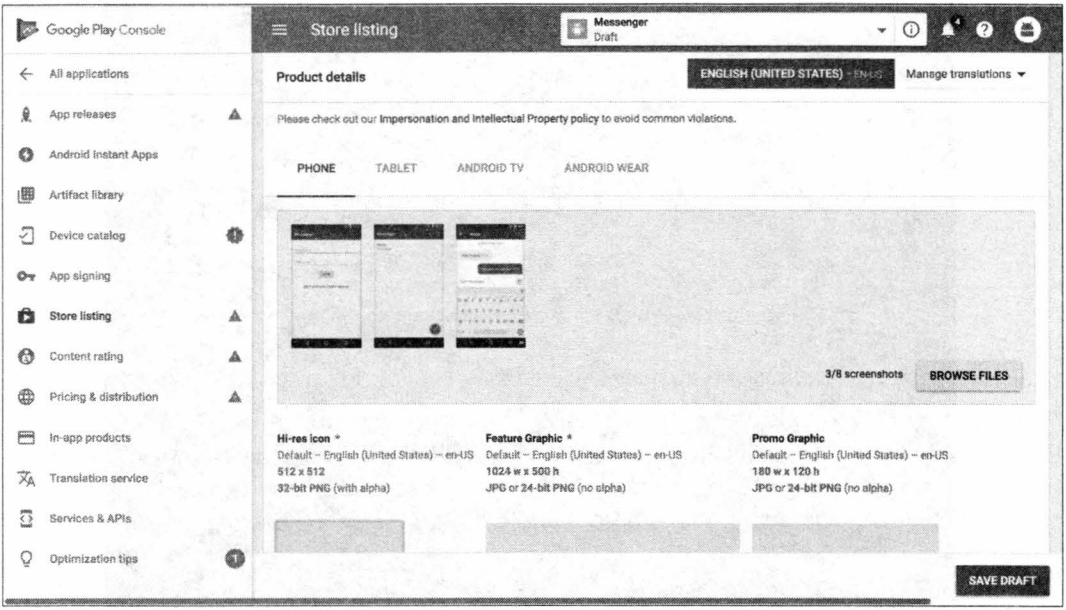


Рис. 8.9. Страница со списком магазинов консоли Google Play

После заполнения информации для списка магазинов следует ввести информацию о ценах и способах распространения приложения. Выберите на левой навигационной панели консоли Google Play опцию **Pricing & distribution** (Расценки и распределение) для раскрытия страницы выбора предпочтений (рис. 8.10). Для нашего примера мы установили цену приложения как **FREE** (Бесплатно) Также произвольным образом выбраны для распространения приложения пять стран: Нигерия, Индия, Соединенные Штаты Америки, Великобритания и Австралия.

Помимо выбора варианта цены и доступных для распространения продукта стран, необходимо предоставить дополнительную информацию о настройках распространения: о категории устройства, о пользовательской программе и информацию о согласии.

Пришло время внести подписанный APK в приложение Google Play Console. Выполните команду **App releases | MANAGE BETA | EDIT RELEASE** (Выпуски приложения | Управлять бета-версией | Изменить выпуск). На открывшейся стра-

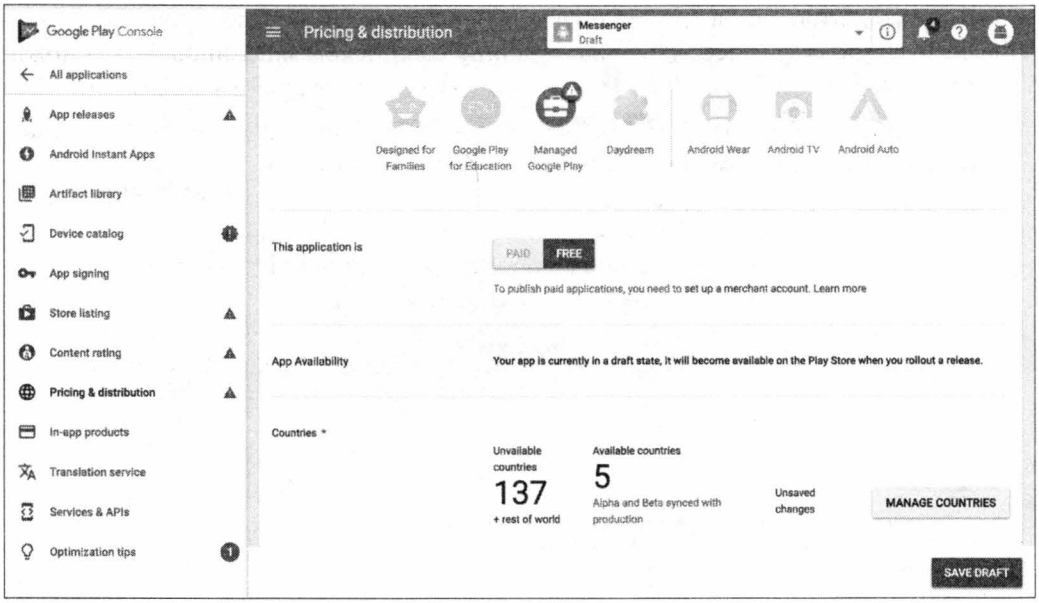


Рис. 8.10. Страница выбора предпочтений консоли Google Play

нице вас могут спросить, не желаете ли вы зарегистрироваться на подписку в приложении Google Play (рис. 8.11).

Для нашего примера нажмите кнопку **OPT-OUT** (ОПТ-выход). На открывшейся странице (рис. 8.12) вы сможете выбрать ваш APK-файл для загрузки из файловой системы компьютера, щелкнув на кнопке **BROWSE FILES** (Просмотр файлов).

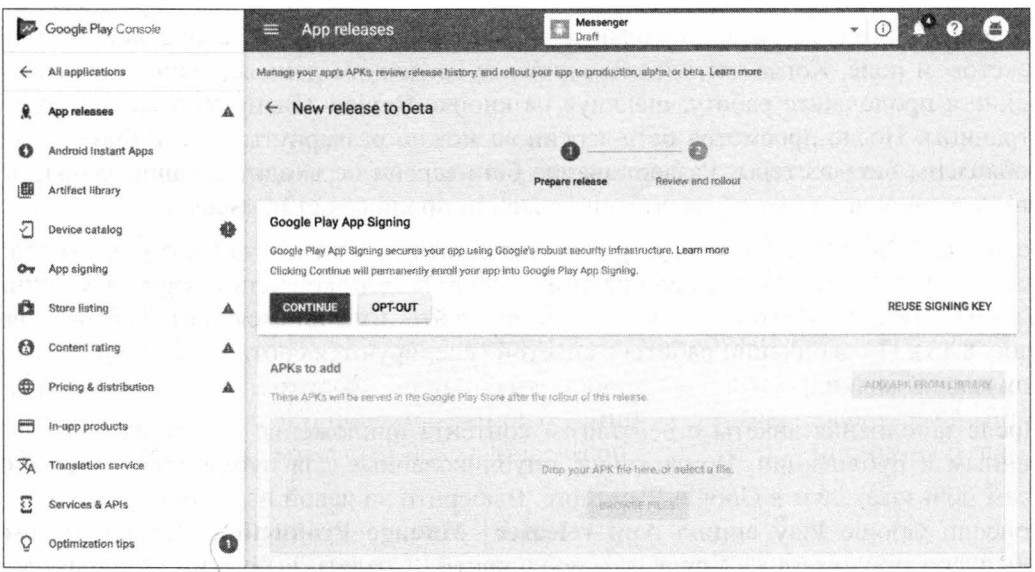


Рис. 8.11. Страница управления бета-версией приложения консоли Google Play

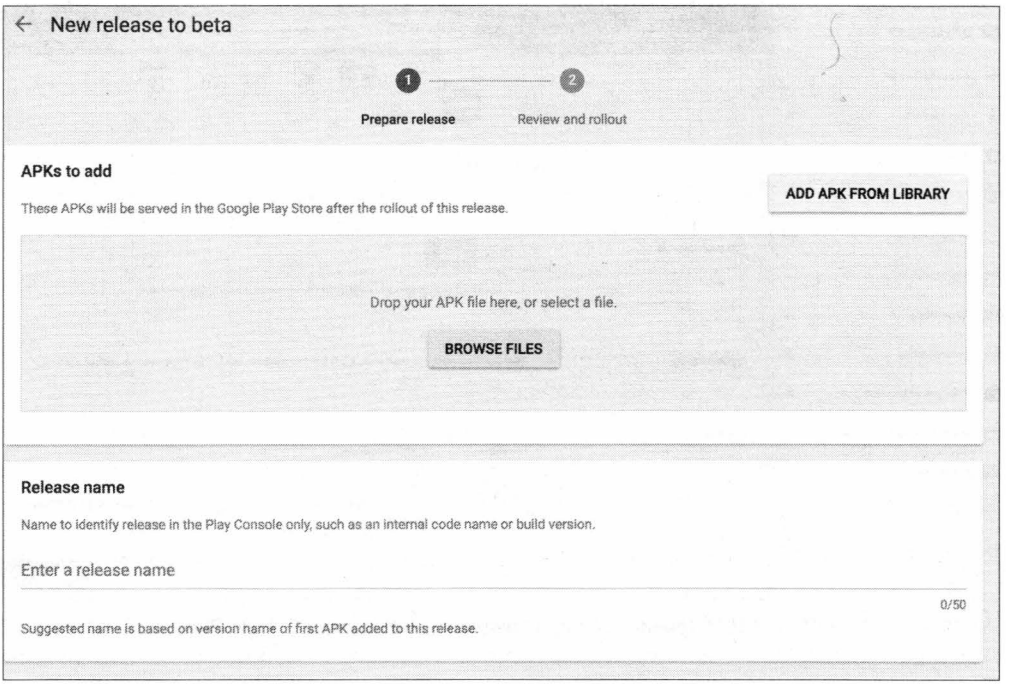


Рис. 8.12. Выбор APK-файла для загрузки из файловой системы компьютера

После выбора соответствующего APK он загружается в консоль Google Play. По завершению загрузки консоль автоматически добавит предлагаемое название выпуска для вашей бета-версии. Это название будет основано на названии версии загруженного APK. Измените название выпуска, если вас не устроит предложенное название. Затем внесите подходящее примечание к выпуску в предоставленном текстовом поле. Когда вас удовлетворят все введенные данные, выполните сохранение и продолжите работу, щелкнув на кнопке **Review** (Выпуск) в нижней части страницы. После просмотра бета-версии ее можно развернуть, если в приложение добавлены бета-тестеры. Развертывание бета-версии не входит в наши планы, поэтому вернемся к главной цели — публикации приложения Messenger.

Загрузив APK для приложения, можно заполнить обязательную анкету для оценки содержания. Выберите на левой навигационной панели консоли Google Play опцию **Content rating** (Рейтинг контента) и следуйте имеющимся там инструкциям (рис. 8.13). По окончании работы с анкетой генерируются соответствующие оценки для вашей заявки.

После заполнения анкеты с рейтингом контента приложение считается подготовленным к публикации. Приложения, опубликованные для публикации, доступны всем пользователям в Google Play Store. Выберите на левой навигационной панели консоли Google Play опцию **App releases | Manage Production | Create releases** (Выпуски приложения | Управлять продукцией | Создать выпуски). Получив подсказку о загрузке APK, щелкните в правой части страницы на кнопке **ADD APK**

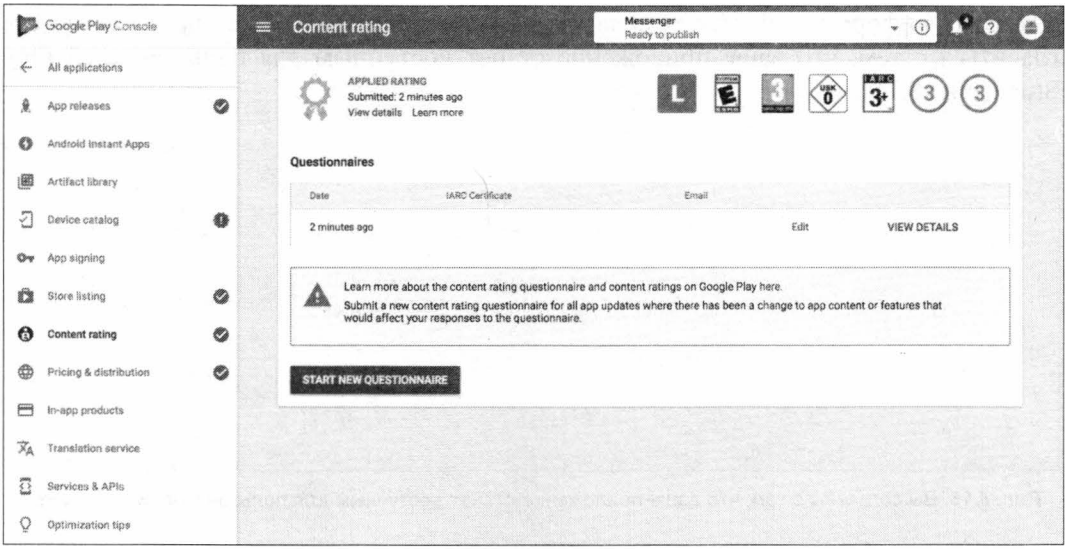


Рис. 8.13. Заполнение обязательной анкеты для оценки содержания приложения в консоли Google Play

FROM LIBRARY (Добавить APK из библиотеки), выделите предварительно загруженный APK (APK с названием версии 1.0) и заполните необходимые сведения о выпуске, аналогично тому как это делалось при создании бета-версии. Щелкните на кнопке обзора внизу страницы, когда будете готовы продолжить, — на следующей странице отобразится краткое резюме вашего приложения (рис. 8.14).

Внимательно просмотрите представленную в резюме информацию. Развертывание приложения начнется, как только вы заявите, что удовлетворены указанной

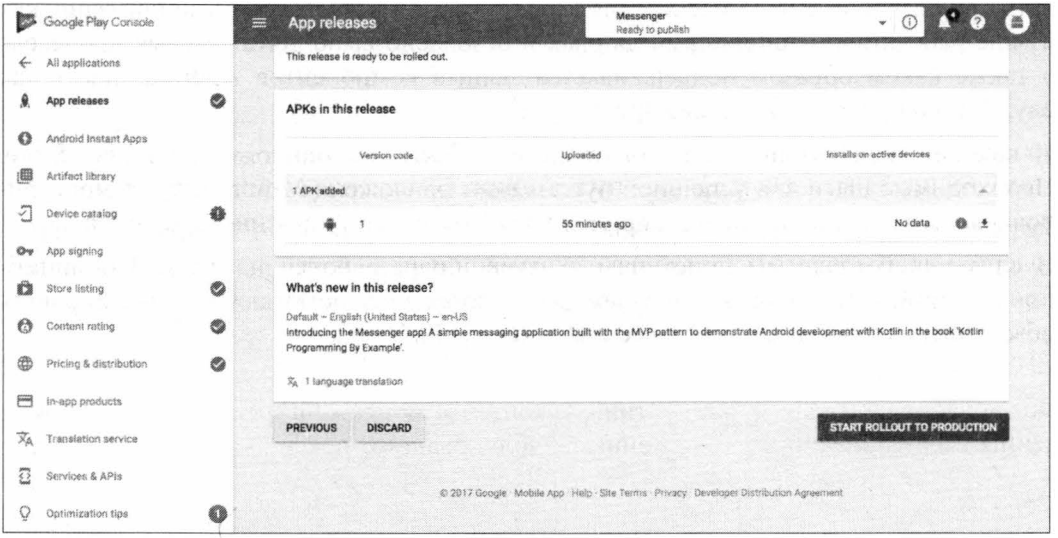


Рис. 8.14. Краткое резюме вашего приложения в консоли Google Play

в резюме информацией. Перед началом развертывания вам будет предложено согласиться с тем, что ваше приложение станет доступным для пользователей Play Store (рис. 8.15).

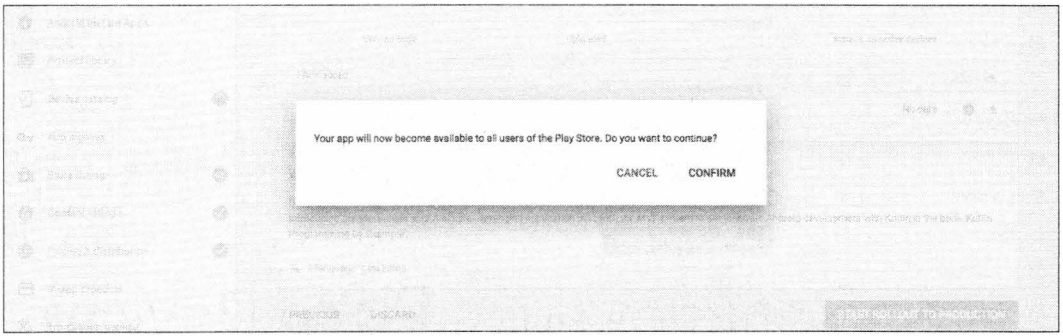


Рис. 8.15. Вы согласны с тем, что ваше приложение станет доступным для пользователей Play Store?

Щелкните на кнопке **Confirm** (Подтвердить), как только будете готовы к запуску приложения Messenger в Play Store. Поздравляем! Вы опубликовали свое первое приложение в магазине Google Play!

Подведем итоги

В этой главе мы завершили изучение среды приложений Android и узнали, каким образом можно защищать приложения Android и публиковать их в магазине Google Play. Мы обозначили угрозы безопасности для приложений Android и описали способы их устранения, а также привели рекомендации, которым необходимо следовать при разработке приложений для экосистемы Android. Нами было показано, как лучше работать с хранилищами данных и безопасно выполнять сетевые процессы, а также каким образом поддерживается защита компонентов Android, таких как службы и широковещательные приемники.

И наконец, основательно описан процесс публикации приложений в Play Store. Необходимые шаги для успешной публикации приложения Android продемонстрированы на примере публикации приложения Messenger в магазине Google Play.

В следующей главе мы приступим к изучению разработки на языке Kotlin веб-приложений, а именно: разработаем веб-приложение, помогающее сформировать обзор того или иного местоположения.

9

Создание серверной части веб-приложения Place Reviewer на основе платформы Spring

В главах с *четвертой* по *седьмую* мы сосредоточились на использовании возможностей языка Kotlin при разработке приложений для платформы Android. В *главе 8*, изучение которой мы только что завершили, подробно рассматривались различные действия по защите и развертыванию приложений Android. Были приведены рекомендации, связанные с обеспечением безопасности при работе с хранилищами данных, а также в процессе обмена данными по сети. Обсуждались меры безопасности, применяемые при обработке пользовательских данных и учетных данных пользователя, и различные методики защиты некоторых компонентов приложений Android, таких как службы и широковещательные приемники. Наконец, буквально по шагам было показано, каким образом следует разворачивать приложение Android в магазине Google Play.

В заключительных главах книги на примере разработки приложения Place Reviewer (Обозреватель местоположения) мы подробно разберемся с тем, каким образом можно использовать язык Kotlin при создании веб-приложений на основе платформы Spring. В этой главе основное внимание уделяется разработке серверной части (бэкэнда) приложения Place Reviewer, а в *главе 10* — его интерфейсу. При изучении этой главы вы познакомитесь со следующими темами:

- шаблон проектирования «модель-представление-контроллер» (Model-View-Controller);
- механизм Logstash и его применение для централизации, преобразования и сохранения данных;
- создание безопасных сайтов с помощью Spring Security.

Приступим к изучению этих вопросов и начнем с рассмотрения шаблона проектирования «модель-представление-контроллер».

Шаблон проектирования MVC

Шаблон MVC (Model-View-Controller, модель-представление-контроллер) — это шаблон проектирования приложений, который применяется главным образом для разделения задач в современных приложениях. Говоря более конкретно, этот шаблон проектирования пользовательских интерфейсов сначала разделяет приложение на три отдельных компонента: модель, представление и контроллер. Подобное разделение приложения удобно по ряду причин. И одна из них состоит в необходимости изоляция логики представления от основной бизнес-логики. Рассмотрим три компонента шаблона MVC (рис. 9.1).

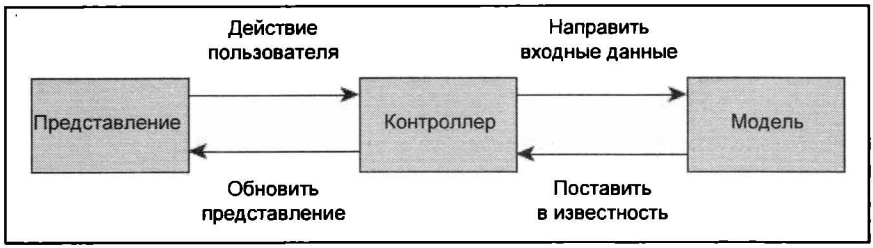


Рис. 9.1. Три компонента шаблона MVC и их взаимодействие

Модель

Модель — это компонент, который отвечает за управление данными и логикой приложения MVC. Поскольку модель является главным менеджером всех данных и бизнес-логики, можно относиться к ней как к локомотиву приложения MVC.

Представление

Это визуальное представление данных, которые существуют и генерируются приложением. Попросту говоря, *представление* — это основное поле взаимодействия пользователя с приложением.

Контроллер

Контроллер является посредником между представлением и моделью. Он отвечает за извлечение входных данных (преимущественно из представления) и передачу соответствующим образом преобразованных входных данных в модель. Контроллер также отвечает за обновление представления с помощью данных при возникновении такой необходимости.

Разработка и реализация серверной части (бэкэнда) веб-приложения Place Reviewer

Поскольку у читателя уже имеется практический опыт разработки системы, в этой главе мы не станем фокусироваться на процессах, связанных с проектированием системы Place Reviewer. Вместо этого мы в самой общей форме определим варианты использования системы, выявим объекты, необходимые для реализации базы данных системы, и более подробно рассмотрим процесс разработки собственно системы. Итак, перейдем к рассмотрению вариантов использования системы Place Reviewer.

Варианты использования

Как мы уже делали в этой книге ранее, приступим к описанию вариантов использования системы, определяя круг ее участников (акторов). Но прежде чем их назвать, уточним, каким целям служит веб-приложение Place Reviewer.

Попросту говоря, веб-приложение Place Reviewer — это интернет-приложение, облегчающее пользователям платформы формирование обзоров того или иного местоположения. После регистрации в приложении пользователь сможет применять эту платформу для формирования основанного на личном опыте обзора любого местоположения. Выбирать местоположение, которое пользователь желает просмотреть, он может с помощью карты местности.

Теперь, когда стало понятно, что делает приложение Place Reviewer, продолжим рассмотрение и определим участников в системе Place Reviewer. Как вы наверняка догадались, на момент запуска приложения Place Reviewer, которое мы собираемся создать, имеется только один его участник — пользователь этого приложения. Варианты действий пользователя таковы:

- пользователь применяет приложение Place Reviewer для формирования отзывов о местоположении;
- пользователь с помощью приложения Place Reviewer просматривает созданные другими пользователями обзоры;
- пользователь может уточнять местоположение, которое уже просмотрено другим пользователем, при помощи интерактивной карты;
- пользователь может регистрироваться на платформе Place Reviewer;
- пользователь может выйти из своей учетной записи Place Reviewer.

Итак, к этому моменту уточнено, что делает система Place Reviewer, идентифицированы участники системы и четко описаны варианты использования системы ее единственным актором — пользователем `User`. Сделаем еще один шаг, определяя необходимые для системы данные.

Необходимые данные

Из рассмотренных вариантов использования системы можно определить тип данных, необходимых для приложения Place Reviewer, и сформировать соответствующие модели. Первый тип используемой информации — данные пользователя (сущность `User`), а второй тип — данные обзора (сущность `Review`). Как следует из названия типов данных, пользовательские данные относятся к зарегистрированному на платформе пользователю, а данные обзора представляют собой собственно данные для формирования созданного на платформе обзора.

Для пользователя нужны следующие данные: адрес его электронной почты, имя пользователя, пароль и состояние учетной записи. Кроме того, для каждого пользователя платформы потребуется уникальный идентификатор пользователя, а также дата регистрации пользователя. Для обзоров данных будут нужны название обзора, текст, содержание обзора, адрес проверенного места, название проверенного места, информация о местоположении проверенного места (координаты: широта и долгота) и идентификатор местоположения, что позволит точно указать проверяемое местоположение. Кроме того, для создаваемого обзора также потребуются уникальный идентификатор и информация, относящаяся ко времени создания обзора.

На этом этапе у вас может возникнуть вопрос: зачем нужна информация, относящаяся к местоположению (название, адрес, идентификатор, а также долгота и широта), вместе с информацией, относящейся к обзору? Почему бы не отделить эту информацию и не рассматривать ее как отдельный тип данных, к которым мы будем обращаться? Если вы так подумали, то вы правы. Действительно, при наличии таблицы базы данных, включающей информацию обо всех местах, которые нам хотелось бы просматривать на платформе, это было бы удобно. К сожалению, подобная таблица пока что отсутствует.

Может возникнуть вопрос: как же предоставлять пользователю возможность просмотра местоположений, информация о которых отсутствует? Ответ прост — мы обратимся к Google Places API. Как это реализовать, мы рассмотрим в следующей главе, а пока приступим к реализации бэкэнда (серверной части) веб-приложения Place Reviewer.

Установка базы данных

Поскольку в системе необходимо сохранять информацию, нам придется настроить базу данных, которая позволит приложению сохранять и извлекать данные. Учитывая, что в предыдущем разработанном нами клиент-серверном приложении (см. *главу 4*) в качестве основного хранилища данных использовалась база данных Postgres, задействуем ее в качестве основного хранилища данных и в этом случае. Так как в *главе 4* мы уже говорили о том, как осуществляется настройка Postgres в различных системах, рассказывать об этом здесь еще раз смысла нет. Перейдем к делу и создадим базу данных. Откройте терминал и выполните следующую команду:

```
createdb -h localhost –username=<username> place-reviewer
```

В результате выполнения этой команды в системе создается база данных `place-reviewer`. Имя пользователя, вводимое вместо аргумента *<username>*, станет именем пользователя, которое будет применяться для подключения к этой базе данных.

Итак, база данных для приложения создана, и мы можем приступить к реализации ее серверной части (бэкэнда), для чего применим фреймворк Spring Framework.

Реализация серверной части веб-приложения

Уточнив варианты использования приложения и настроив базу данных для подключения к приложению, продолжим его реализацию. Откройте IntelliJ IDEA и создайте новый проект с инициализатором Spring. После щелчка на кнопке **Next** (Далее) IntelliJ извлечет инициализатор Spring, затем вы получите предложение о предоставлении сведений о приложении. Прежде чем перейти к очередному этапу установки, выполните следующие действия.

1. В качестве идентификатора группы введите `com.example`.
2. В качестве идентификатора приложения укажите `place-reviewer`.
3. Выберите **Maven Project** в качестве типа проекта, если он еще не выбран.
4. Оставьте без изменений текущий параметр упаковки и версию Java.
5. В качестве языка выберите **Kotlin**. Это важно, поскольку в этой книге изучается именно этот язык.
6. Измените атрибут версии на 1.0.0.
7. Введите описание своего приложения. Наше описание выглядит следующим образом: `A nifty web application for the creation of location reviews` (Изящное веб-приложение для создания обзоров местоположения).
8. Введите `com.example.placereviewer` в качестве имени пакета.

Инициализатор Spring поставляется с плагином Spring, на момент подготовки этой книги доступным только в среде IntelliJ IDEA Ultimate Edition, для пользования которой нужна платная лицензия. Если у вас установлена среда IntelliJ IDEA Community Edition, чтобы продолжить работу над приложением, создайте проект с помощью утилиты инициализатора Spring, доступной на сайте **https://start.spring.io**, после чего импортируйте этот проект в среду IntelliJ IDEA.

После завершения ввода необходимой информации о проекте перейдите к следующему экрану, щелкнув на кнопке **Next**, и выберите зависимости проекта: в разделе **Core** — опции **Security**, **Session** и **Cache**, в разделе **Web** — **Web**, в разделе **Template Engines** — **Thymeleaf**, в разделе **SQL** — **PostgreSQL**. Кроме того, в раскрывающемся меню выбора **Spring Boot Version** (Версия загрузки Spring), находящемся в верхней части экрана, выберите версию **2.0.0 M7**. После завершения выбора необходимых зависимостей содержимое экрана должно принять вид, схожий с показанным на рис. 9.2.

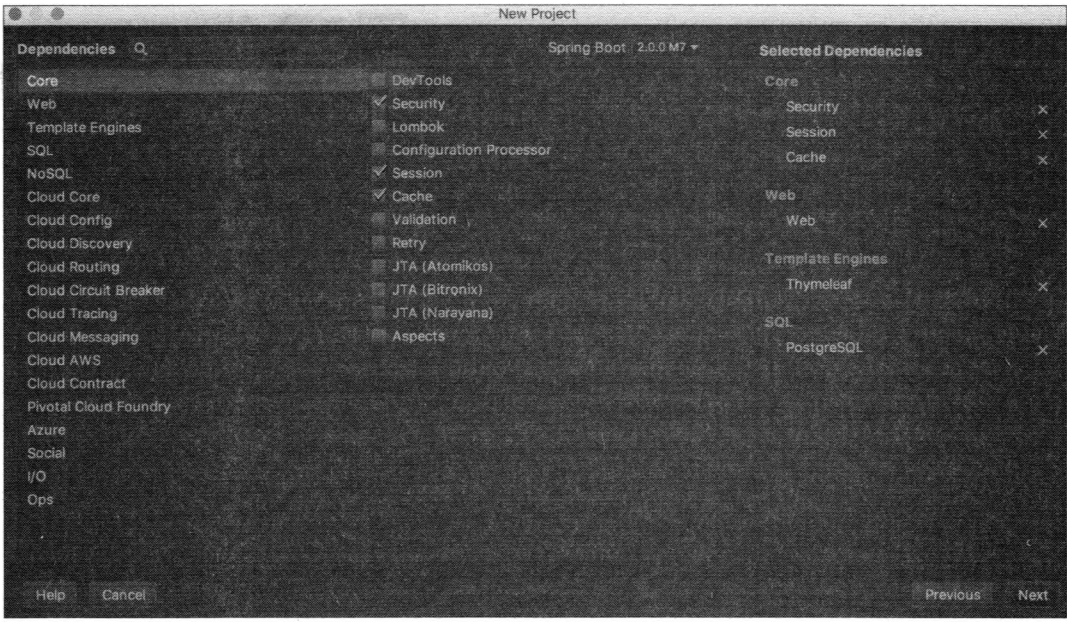


Рис. 9.2. Выбор зависимостей проекта

Убедившись, что выбраны все необходимые зависимости, щелкните на кнопке **Next** (Далее) для перехода к завершающему экрану установки. На этом экране следует указать название и местоположение файлов проекта. Введите `place-reviewer` в качестве названия проекта и выберите место на компьютере, в котором будет сохраняться проект (рис. 9.3).

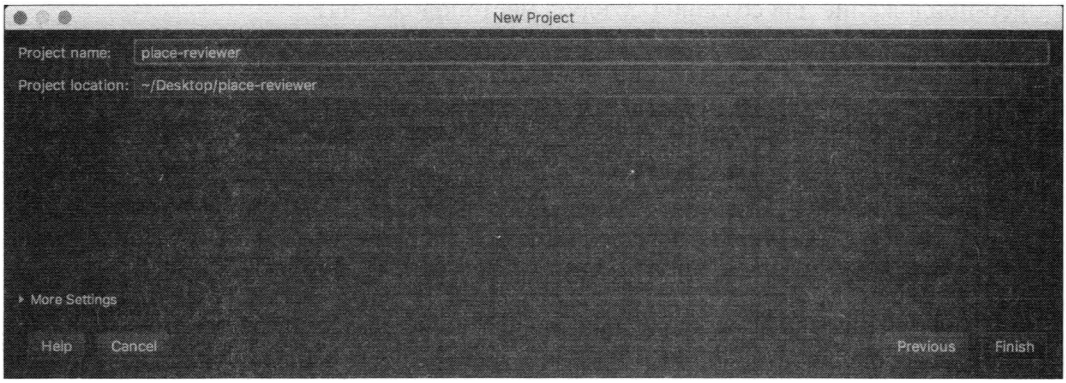


Рис. 9.3. Название проекта и место на компьютере, где будут храниться файлы проекта

После этого щелкните на кнопке **Finish** (Готово) и ожидайте завершения создания проекта — вы окажетесь в новом окне IDE, содержащем исходные файлы проекта. Нет необходимости описывать здесь сведения о структуре проекта Spring, поскольку в одной из предыдущих глав (см. *главу 4*), мы уже работали с подобным проек-

том. Но прежде чем двигаться дальше, добавьте в файл проекта pom следующие зависимости:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>org.webjars</groupId>
   <artifactId>bootstrap</artifactId>
   <version>4.0.0-beta.3</version>
</dependency>
<dependency>
   <groupId>org.webjars</groupId>
   <artifactId>jquery</artifactId>
   <version>3.2.1</version>
</dependency>
```

Теперь продолжим подключение серверной части (бэкэнда) приложения Place Reviewer к базе данных Postgres.

Подключение бэкэнда приложения Place Reviewer к базе данных Postgres

Для подключения бэкэнда приложения Place Reviewer к созданной для него базе данных PostgreSQL следует изменить файл проекта application.properties для включения в него необходимых свойств, упрощающих соединение базы данных с PostgreSQL. Откройте файл проекта application.properties и добавьте в него следующие свойства:

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.generate-ddl=true
spring.datasource.url=jdbc:postgresql://localhost:5432/place-reviewer
spring.datasource.driver.class-name=org.postgresql.Driver
spring.datasource.username=<username>
```

Вместо фрагмента *<username>* впишите здесь соответствующее имя пользователя.

После добавления этих свойств Spring Boot получит возможность при запуске приложения подключиться к указанной базе данных. Создадим теперь модели для ранее идентифицированных сущностей `User` и `Review`.

Создание моделей

Ранее мы определили два различных типа объектов, которые следует учесть в нашей системе: сущность `User` и сущность `Review`. Теперь пришло время сформировать соответствующие модели для этих сущностей. Первой сущностью, которой мы

займемся, станет User. Создайте в пакете com.example.placereviewer пакет data. Добавьте в него пакет model, а в этот пакет — файл User.kt со следующим содержимым:

```
package com.example.placereviewer.data.model

import com.example.placereviewer.listener.UserListener
import org.springframework.format.annotation.DateTimeFormat
import java.time.Instant
import java.util.*
import javax.persistence.*
import javax.validation.constraints.Pattern
import javax.validation.constraints.Size

@Entity
@Table(name = "`user`")
@EntityListeners(UserListener::class)
data class User(
   @Column(unique = true)
   @Size(min = 2)
   @Pattern(regexp = "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\\\.[A-Z] {2,6}\$")
   var email: String = "",
      @Column(unique = true)
   var username: String = "",
      @Size(min = 60, max = 60)
   var password: String = "",
      @Column(name = "account_status")
   @Pattern(regexp = "\\A(activated|deactivated)\\z")
   var accountStatus: String = "activated",
      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
   var id: Long = 0,
      @DateTimeFormat
      @Column(name = "created_at")
   var createdAt: Date = Date.from(Instant.now())
) {
   @OneToMany(mappedBy = "reviewer", targetEntity = Review::class)
   private var reviews: Collection<Review>? = null
}
```

Нет необходимости подробно объяснять, что происходит в приведенном фрагменте кода, поскольку у нас имеется опыт по созданию в Spring сущностей. В этом фрагменте кода мы определили сущность User вместе с ее email, username, password, accountStatus, Id и свойствами createdAt в качестве их атрибутов. Кроме того, указано, что User располагает большим числом сущностей Review. Мы также задали слушатель сущности с помощью аннотации @EntityListener. Но для сущности User

не создана ни сущность Review, ни сущность UserListener. До обращения к сущности Review поработаем с сущностью User и создадим слушатель этой сущности — добавим в пакет com.example.placereviewer новый пакет слушателя сущности, а в него — файл UserListener.kt, который включает следующий код:

```
package com.example.placereviewer.listener

import com.example.placereviewer.data.model.User
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
import javax.persistence.PrePersist
import javax.persistence.PreUpdate

class UserListener {
   @PrePersist
   @PreUpdate
   fun hashPassword(user: User) {
      user.password = BCryptPasswordEncoder().encode(user.password)
   }
}
```

Слушатель UserListener включает единственную функцию hashPassword, которая вызывается перед сохранением и перед обновлением сущности User. Этот метод выполняет только одну задачу — кодирования свойства пользователя password в bcrypt-эквивалент до сохранения его в базе данных.

Создав необходимые слушатели для сущности User, обратим внимание на определение сущности Review. Создайте в com.example.placereviewer.data.models файл Review.kt, включающий следующее содержимое:

```
package com.example.placereviewer.data.model

import org.springframework.format.annotation.DateTimeFormat
import java.time.Instant
import java.util.*
import javax.persistence.*
import javax.validation.constraints.Size

@Entity
@Table(name = "`review`")
data class Review(
   @ManyToOne(optional = false)
   @JoinColumn(name = "user_id", referencedColumnName = "id")
   var reviewer: User? = null,
      @Size(min = 5)
   var title: String = "",
      @Size(min = 10)
```

```
    var body: String = "",
        @Column(name = "place_address")
        @Size(min = 2)
    var placeAddress: String = "",
        @Column(name = "place_name")
    var placeName: String = "",
        @Column(name = "place_id")
    var placeId: String = "",
    var latitude: Double = 0.0,
    var longitude: Double = 0.0,
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = 0,
        @DateTimeFormat
        @Column(name = "created_at")
    var createdAt: Date = Date.from(Instant.now())
)
```

Как показано в приведенном фрагменте кода, создан класс данных `Review` со следующими свойствами: `reviewer`, `title`, `body`, `placeAddress`, `placeName`, `placeId`, `latitude`, `longitude`, `id` и `createdAt`. Свойство `reviewer` имеет тип `User`. Оно ссылается на создателя обзора. Каждый обзор должен формироваться пользователем. Кроме того, многие обзоры создаются одним пользователем. Применим аннотацию `@ManyToOne` для того, чтобы надлежащим образом объявить эту связь между сущностями `Review` и `User`.

Создание репозиториев данных

Поскольку необходимые объекты имеются и настроены, нам теперь нужны репозитории, которые используются для доступа к данным, относящимся к нашим сущностям. Создайте пакет репозиториев в пакете com.example.placereviewer. Имея две сущности, сформируем два репозитория — по одному для доступа к данным, хранящимся в каждой сущности. Первым репозиторием станет `UserRepository`, а вторым — `ReviewRepository`. Создайте в com.example.placereviewer.data.repository файл интерфейса UserRepository, который включает следующее содержимое:

```
package com.example.placereviewer.data.repository

import com.example.placereviewer.data.model.User
import org.springframework.data.repository.CrudRepository
interface UserRepository : CrudRepository<User, Long> {
    fun findByUsername(username: String): User?
}
```

Метод `findByUsername(String)` извлекает сущность `User` из базы данных, которая располагает именем пользователя, соответствующим тому, что передается в качестве аргумента функции. Далее приводится код интерфейса `ReviewRepository`:

```
package com.example.placereviewer.data.repository

import org.springframework.data.repository.CrudRepository

interface ReviewRepository : CrudRepository<Review, Long> {
   fun findByPlaceId(placeId: String)
}
```

Настроив сущности и репозитории для запроса этих сущностей, мы можем приступить к реализации основной бизнес-логики приложения Place Reviewer в форме служб и реализаций служб.

Реализация бизнес-логики Place Reviewer

Как мы отметили ранее, в приложении, которое придерживается шаблона проектирования MVC, имеются три основных компонента: модель, представление и контроллер. Компонентами, отвечающими за управление данными и выполнение бизнес-логики, в приложении Place Reviewer служат модели. И мы планируем реализовать эти модели в форме служб, которые можно использовать в серверной части (бэкэнде) приложения. Сейчас нам необходимо создать две основные службы: первая служба управляет данными, относящимися к пользователям приложения, а вторая — данными обзоров.

Сначала создадим интерфейс `UserService`, определяющий поведения, которые должны реализовываться действительным классом `UserServiceImpl`. Ранее, в описании вариантов использования приложения Place Reviewer, указывалось, что пользователь должен иметь возможность регистрации в платформе (и, разумеется, открытия в ней учетной записи). Соответственно, в нашей модели эту процедуру следует учесть. Создайте в пакете root проекта пакет service. Затем добавьте в него интерфейс `UserService`:

```
package com.example.placereviewer.service

interface UserService {

 fun register(username: String, email: String, password: String): Boolean
}
```

Здесь объявлен метод, который следует реализовать с помощью действительной службы `UserService`. Это метод `register(String, String, String)`, использующий в качестве аргументов три строки: первая строка — имя пользователя для регистрации, вторая — действительный адрес электронной почты пользователя, а третья — его пароль. При вызове с соответствующими аргументами метод `register()` пытается зарегистрировать пользователя с помощью предоставленных учетных данных и возвращает `true`, если пользователь был успешно зарегистрирован, в противном случае возвращается `false`.

Далее приведена реализация интерфейса `UserService`. Добавьте ее в пакет service:

```
package com.example.placereviewer.service

import com.example.placereviewer.data.model.User
import com.example.placereviewer.data.repository.UserRepository
import org.springframework.stereotype.Service

@Service
class UserServiceImpl(val userRepository: UserRepository) : UserService {

   override fun register(username: String, email: String,
                         password: String): Boolean {
      val user = User(email, username, password)
      userRepository.save(user)

      return true
   }
}
```

Функция `register()`, реализованная классом `UserServiceImpl`, выполняет очевидные действия: при передаче ей достоверных аргументов `username`, `email` и `password` она формирует новый объект пользователя и передает соответствующие аргументы его конструктору. После создания объекта пользователя пользователь сохраняется в базе данных с помощью следующей строки:

```
userRepository.save(user)
```

Класс `userRepository` является экземпляром созданного ранее `UserRepository`. Этот экземпляр автоматически внедряется фреймворком Spring Framework в конструктор для `UserServiceImpl`. После того как пользователь сохранен в базе данных, возвращается булево значение `true`.

Теперь реализуем интерфейс службы обзора. Эта служба облегчает формирование созданных пользователями платформы обзоров, а также их отображение. Соответственно, нам нужно реализовать методы `createReview()` и `listReview()` в интерфейсе пользователя service.

Добавим следующий интерфейс `ReviewService` в пакет service нашего проекта:

```
package com.example.placereviewer.service

import com.example.placereviewer.data.model.Review

interface ReviewService {

  fun createReview(reviewerUsername: String, reviewData: Review): Boolean

   fun listReviews(): Iterable<Review>
}
```

На следующем шаге нам надо создать для этой службы обзора класс `ReviewServiceImpl`. Добавим его, как и все службы, которые будут создаваться в этой главе далее, в пакет com.example.placereviewer.service:

```
package com.example.placereviewer.service

import com.example.placereviewer.data.model.Review
import com.example.placereviewer.data.model.User
import com.example.placereviewer.data.repository.ReviewRepository
import com.example.placereviewer.data.repository.UserRepository
import org.springframework.stereotype.Service

@Service
class ReviewServiceImpl(val reviewRepository: ReviewRepository, val
userRepository: UserRepository) : ReviewService {

   override fun listReviews(): Iterable<Review> {
      return reviewRepository.findAll()
   }

   override fun createReview(reviewerUsername: String,
                             reviewData: Review): Boolean {
    val reviewer: User? = userRepository.findByUsername(reviewerUsername)

      if (reviewer != null) {
         reviewData.reviewer = reviewer
         reviewRepository.save(reviewData)
         return true
      }
      return false
   }
}
```

Функция `listReviews()` возвращает итерацию, содержащую все данные обзора, которые сохранялись в базе данных приложения. С другой стороны, функция `createReview()` принимает строку, значением которой является имя пользователя, создавшего обзор, и экземпляр `Review`, включающий данные для созданного обзора. Функция `createReview()` сначала извлекает пользователя с определенным именем, вызывая метод `findByUsername()` из класса `UserRepository`. Пользователь, который извлечен, и является автором обзора, а значит, и его обозревателем.

Если с помощью класса `UserRepository` возвращается не нулевой объект, значит, пользователь реально существует, и тогда пользователь, который извлекался, назначается свойству обозревателя сохраняемого обзора. После этого назначения обзор записывается в базу данных, и функция возвращает `true`, тем самым показывая, что процесс прошел успешно. Если пользователь с указанным именем не найден, с помощью `createReview()` возвращается `false`.

Создав соответствующие модели в форме служб, рассмотрим меры по защите приложения Place Reviewer. Это очень важно, поскольку нежелательно, чтобы посторонние лица имели доступ к ресурсам приложения.

Обеспечение безопасности бэкэнда Place Reviewer

Подобно тому, как в *главе 4* мы предпринимали меры обеспечения безопасности API Messenger, для защиты серверной части (бэкэнда) Place Reviewer также используется Spring Security. Впрочем, здесь в способе защиты нашего приложения имеет место небольшое изменение. В *главе 4* Spring Security настраивался для явного использования веб-токенов JSON при авторизации клиентских приложений. Здесь же мы будем полагаться исключительно на возможности Spring Security. При этом другие технологии, такие как веб-токены JSON, не используются. Что ж, приступим к разработке мер безопасности для нашего бэкэнда.

Прежде всего создадим пользовательскую конфигурацию веб-безопасности для приложения. Эта пользовательская конфигурация обеспечивает реализацию класса `WebSecurityConfigurerAdapter` из Spring Framework. Создайте в com.example.placereviewer пакет config и добавьте следующий класс `WebSecurityConfig`:

```
package com.example.placereviewer.config

import com.example.placereviewer.service.AppUserDetailsService
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.HttpMethod
import org.springframework.security.authentication.AuthenticationManager
import org.springframework.security.config.BeanIds
import org.springframework.security.config.annotation
        .authentication.builders.AuthenticationManagerBuilder
import org.springframework.security.config.annotation
        .web.builders.HttpSecurity
import org.springframework.security.config.annotation
        .web.configuration.EnableWebSecurity
import org.springframework.security.config.annotation
        .web.configuration.WebSecurityConfigurerAdapter
import org.springframework.security.core.userdetails
        .UserDetailsService
import org.springframework.security.crypto.bcrypt
        .BCryptPasswordEncoder
import org.springframework.security.web
        .DefaultRedirectStrategy
import org.springframework.security.web.RedirectStrategy

@Configuration
@EnableWebSecurity
```

```
class WebSecurityConfig(val userDetailsService: AppUserDetailsService) :
WebSecurityConfigurerAdapter() {

   private val redirectStrategy: RedirectStrategy = DefaultRedirectStrategy()

   @Throws(Exception::class)
   override fun configure(http: HttpSecurity) {
      http.authorizeRequests()
         .antMatchers(HttpMethod.GET,"/register").permitAll()
         .antMatchers(HttpMethod.POST,"/users/registrations").permitAll()
         .antMatchers(HttpMethod.GET,"/css/**").permitAll()
         .antMatchers(HttpMethod.GET,"/webjars/**").permitAll()
         .anyRequest().authenticated()
         .and()
         .formLogin()
         .loginPage("/login")
         .successHandler { request, response, _ ->
            redirectStrategy.sendRedirect(request, response, "/home")
         }
         .permitAll()
         .and()
         .logout()
         .permitAll()
      }

   @Throws(Exception::class)
   override fun configure(auth: AuthenticationManagerBuilder) {
      auth.userDetailsService<UserDetailsService>(userDetailsService)
         .passwordEncoder(BCryptPasswordEncoder())
      }
   @Bean(name = [BeanIds.AUTHENTICATION_MANAGER])
   override fun authenticationManagerBean(): AuthenticationManager {
      return super.authenticationManagerBean()
   }
}
```

Как уже упоминалось в книге ранее, метод `configure(HttpSecurity)` из класса `WebSecurityConfig` служит для настройки защиты путей HTTP URL. С помощью метода `configure(HttpSecurity)` мы здесь настраиваем Spring Security для разрешения всех HTTP POST-запросов для `/users/registrations` и GET-запросов, пути которых соответствуют путям `/register`, `/css` и `/webjars/**`. Кроме этого, разрешаем все HTTP-запросы на страницу входа, доступ к которой можно получить с помощью пути `/login`.

В действие входа в систему добавлен обработчик успешного входа, который использует свойство `redirectStrategy`, определенное для класса `WebSecurityConfig` для

перенаправления клиента к пути /home при успешном входе пользователя в систему. Наконец, разрешены все запросы на выход из системы для нашего бэкэнда.

С помощью configure(AuthenticationManagerBuilder) устанавливается UserDetailsService для применения и указывается кодировщик применяемого пароля. В нашем случае используется кодировщик BcryptPasswordEncoder. Как вы уже, наверное, заметили, проекте пока еще не создана реализация для UserDetailsService. Выполним это сейчас. Добавим объект AppUserDetailsService в пакет com.example.placereviewer.service:

```
package com.example.placereviewer.service

import com.example.placereviewer.data.repository.UserRepository
import org.springframework.security.core.GrantedAuthority
import org.springframework.security.core.userdetails.User
import org.springframework.security.core.userdetails.UserDetails
import org.springframework.security.core.userdetails.UserDetailsService
import org.springframework.security.core.userdetails
        .UsernameNotFoundException
import org.springframework.stereotype.Service
import java.util.ArrayList

@Service
class AppUserDetailsService(private val userRepository: UserRepository) :
UserDetailsService {

   @Throws(UsernameNotFoundException::class)
   override fun loadUserByUsername(username: String): UserDetails {
      val user = userRepository.findByUsername(username) ?:
          throw UsernameNotFoundException("A user with the username
                                          $username doesn't exist")

      return User(user.username, user.password,
                  ArrayList<GrantedAuthority>())
   }
}
```

С помощью loadUsername(String) предпринимается попытка загрузить UserDetails для пользователя, соответствующего переданному функции имени пользователя. Если пользователь с указанным именем не обнаружен, тогда генерируется исключение UsernameNotFoundException.

К этому моменту вы вполне успешно настроили Spring Security для бэкэнда. Отлично!

Теперь, после завершения работ с объектами, репозиториями, службами, реализациями служб и настройкой Spring Security, мы можем приступить к реализации *фронтэнда* — пользовательского интерфейса и функциональности, которые работают на клиентской стороне нашего приложения (от *англ.* frontend). Однако реали-

зация веб-интерфейса приложения будет рассматриваться в следующей главе, а сейчас мы сосредоточимся на других моментах и рассмотрим процесс предоставления веб-контента клиентскому приложению с помощью Spring MVC.

Обслуживание веб-контента с помощью Spring MVC

В Spring MVC HTTP-запросы обрабатываются контроллерами. *Контроллеры* — это классы, помеченные как `@Controller`, — аналогично тому, как с помощью `@RestController` аннотировались rest-контроллеры. Лучше всего понять, как работают контроллеры, — это рассмотреть соответствующий пример. Создадим простой контроллер Spring MVC, обрабатывающий запросы HTTP GET, отправленные с помощью пути `/say/hello` при возвращении представления, которое отвечает за отображение HTML-страницы пользователю.

Создайте в пакете com.example.placereviewer пакет controller и добавьте в него следующий класс:

```
package com.example.placereviewer.controller

import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping

@Controller
@RequestMapping("/say")
class HelloController {

   @GetMapping("/hello")
   fun hello(): String {
      return "hello"
   }
}
```

Как можно видеть, создание контроллера не является сложной задачей. Аннотирование контроллера `HelloController`, реализованное с помощью `@Controller`, указывает Spring, что этот класс является контроллером Spring MVC и как таковой может обрабатывать HTTP-запросы. Кроме того, аннотирование `HelloController` с помощью `@RequestMapping("/say")` определяет, что этот контроллер обрабатывает HTTP-запросы, которые проходят/поступают посредством соответствующих базовых путей. Определим в нашем контроллере действие `hello()`. Поскольку это действие аннотировано с помощью `@GetMapping("/hello")`, обработка GET-запросов выполняется по пути `/say/hello`. Строка, возвращаемая `hello()`, служит именем ресурса представления, который предоставляется клиенту при отправке запроса по этому маршруту.

Поскольку действие `hello()` приводит к тому, что к клиенту возвращается представление под названием `hello`, следующая задача — добавить в проект подобное

представление. Представления обычно добавляются в папку шаблонов каталога ресурсов проекта Spring. Добавьте в проект файл hello.html, выполнив щелчок правой кнопкой мыши на опции **templates** (Шаблоны) и выбрав команду **New | HTML File** (Создать | Файл HTML) (рис. 9.4).

Рис. 9.4. Добавление в проект файла hello.html

Теперь нужно задать создаваемому HTML-файлу имя — в качестве имени введите `hello` и нажмите кнопку **OK** (рис. 9.5), после чего среда IntelliJ IDEA сгенерирует HTML-файл в выбранном каталоге.

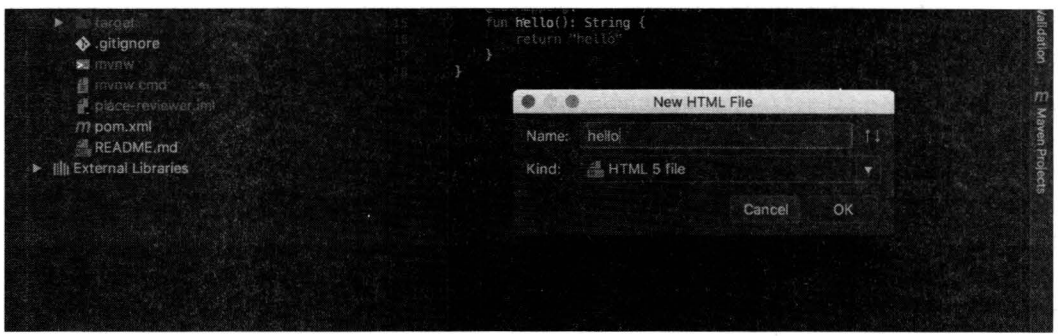


Рис. 9.5. Файлу hello.html присваивается имя `hello`

Затем измените содержимое файла для включения следующего базового кода HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Hello</title>
</head>
<body>
Hello world!
</body>
</html>
```

Теперь нужно проверить, работает ли созданный нами контроллер. В этом мы убедимся, если после обращения к нему вернется HTML-страница с сообщением **Hello world!**. Обращение реализуется путем направления GET-запроса по пути маршрута контроллера. Обратите внимание, что следует добавить GET-запросы, направленные по пути `/say/hello`, как запросы, разрешенные Spring Security без аутентификации. Выполнить это несложно — просто измените `configure(HttpSecurity)` в `WebSecurityConfig` для разрешения направления GET-запросов по пути `/say/hello`, как показано в следующем фрагменте кода:

```
@Throws(Exception::class)
override fun configure(http: HttpSecurity) {
   http.authorizeRequests()
      .antMatchers(HttpMethod.GET,"/say/hello").permitAll() // added line
       .antMatchers(HttpMethod.GET,"/register").permitAll()
       .antMatchers(HttpMethod.POST,"/users/registrations").permitAll()
       .antMatchers(HttpMethod.GET, "/css/**").permitAll()
       .antMatchers(HttpMethod.GET, "/webjars/**").permitAll()
       .anyRequest().authenticated()
       .and()
       .formLogin()
       .loginPage("/login")
       .successHandler { request, response, _ ->
          redirectStrategy.sendRedirect(request, response, "/home")
       }
       .permitAll()
       .and()
       .logout()
       .permitAll()
}
```

Создайте и запустите приложение Spring, затем откройте ваш веб-браузер и перейдите по следующему URL-адресу: **http://localhost:5000/say/hello** (рис. 9.6).

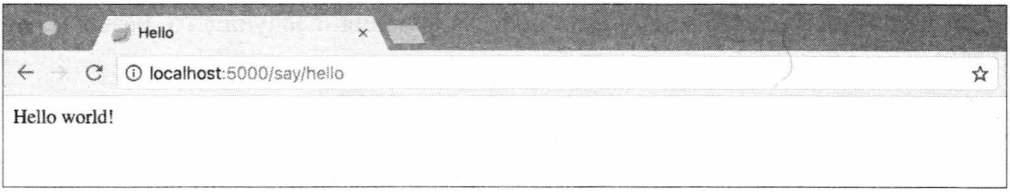


Рис. 9.6. Вас с энтузиазмом приветствует сообщение **Hello world!**

Управление журналами приложений Spring с помощью ELK

При формировании систем, предназначенных для развертывания, важно учитывать, какими именно средствами управляются файлы журнала сервера. *Журнал сервера* — файл журнала, создаваемый и поддерживаемый сервером. Файлы журнала обычно состоят из списка действий, исполненных сервером. Средством управления файлами журналов приложений, которое следует внимательно рассмотреть, и служит стек ELK (Elasticsearch, Logstash и Kibana). В этом разделе мы узнаем, как управлять файлами журналов приложений Spring с помощью стека ELK.

Создание журналов с помощью Spring

Прежде чем приступить к настройке стека ELK для управления журналами Spring, необходимо настроить Spring для создания файлов журналов. Для этого обратимся к файлу application.properties проекта Spring и настроим бэкэнд Place Reviewer для создания журналов.

Откройте файл application.properties проекта и добавьте в него следующую строку кода:

```
logging.file=application.log
```

Эта строка кода настроит Spring для генерирования и сохранения журналов сервера в файле application.log. Данный файл создается и сохраняется в корневом каталоге проекта при следующем запуске проекта. Всего, что было сделано до сих пор, вполне достаточно для настройки стека журналов сервера. И начнем мы с установки поискового движка Elasticsearch.

Установка Elasticsearch

Чтобы установить Elasticsearch, выполните следующие действия:

1. Загрузите Elasticsearch, упакованный в ZIP-файл, обратившись на сайт по адресу: **https://www.elastic.co/downloads/elasticsearch** (рис. 9.7).
2. Извлеките Elasticsearch из ZIP-файла после загрузки.
3. Запустите Elasticsearch, для чего в окне терминале выполните команду: `bin/elasticsearch` (`bin/elasticsearch.bat` — для Windows).

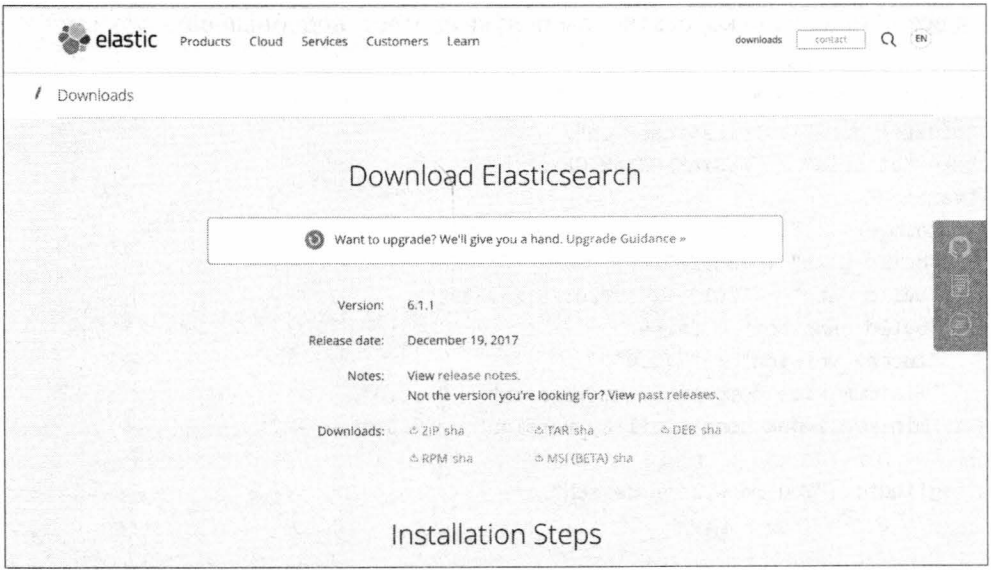


Рис. 9.7. Загрузка ZIP-файла Elasticsearch

После выполнения в окне терминала команды `bin/elasticsearch` Elasticsearch будет запущен в системе (рис. 9.8).

Запустив Elasticsearch, можно проверить, правильно ли он функционирует, выполнив в окне терминала следующую команду:

```
curl -XGET http://localhost:9200
```

```
$ elasticsearch-6.1.1/bin/elasticsearch
[2018-01-16T01:06:00,339][INFO ][o.e.n.Node               ] [] initializing ...
[2018-01-16T01:06:00,461][INFO ][o.e.e.NodeEnvironment    ] [Df8YuN2] using [1]
data paths, mounts [[/ (/dev/disk1)]], net usable_space [11.6gb], net total_spac
e [111.8gb], types [hfs]
[2018-01-16T01:06:00,461][INFO ][o.e.e.NodeEnvironment    ] [Df8YuN2] heap size
[990.7mb], compressed ordinary object pointers [true]
[2018-01-16T01:06:00,483][INFO ][o.e.n.Node               ] node name [Df8YuN2]
derived from node ID [Df8YuN2dQTa5CJH0cSq9KQ]; set [node.name] to override
[2018-01-16T01:06:00,484][INFO ][o.e.n.Node               ] version[6.1.1], pid[
19550], build[bd92e7f/2017-12-17T20:23:25.338Z], OS[Mac OS X/10.12.5/x86_64], JV
M[Oracle Corporation/Java HotSpot(TM) 64-Bit Server VM/1.8.0_65/25.65-b01]
[2018-01-16T01:06:00,484][INFO ][o.e.n.Node               ] JVM arguments [-Xms1
g, -Xmx1g, -XX:+UseConcMarkSweepGC, -XX:CMSInitiatingOccupancyFraction=75, -XX:+
UseCMSInitiatingOccupancyOnly, -XX:+AlwaysPreTouch, -Xss1m, -Djava.awt.headless=
true, -Dfile.encoding=UTF-8, -Djna.nosys=true, -XX:-OmitStackTraceInFastThrow, -
Dio.netty.noUnsafe=true, -Dio.netty.noKeySetOptimization=true, -Dio.netty.recycl
er.maxCapacityPerThread=0, -Dlog4j.shutdownHookEnabled=false, -Dlog4j2.disable.j
mx=true, -XX:+HeapDumpOnOutOfMemoryError, -Des.path.home=/Users/Iyanu/Downloads/
elasticsearch-6.1.1, -Des.path.conf=/Users/Iyanu/Downloads/elasticsearch-6.1.1/c
onfig]
[2018-01-16T01:06:01,607][INFO ][o.e.p.PluginsService     ] [Df8YuN2] loaded mod
ule [aggs-matrix-stats]
```

Рис. 9.8. Elasticsearch запущен в системе

Если все установлено корректно, вы получите ответ, подобный следующему:

```
{
   "name" : "Df8YuN2",
   "cluster_name" : "elasticsearch",
   "cluster_uuid" : "Z8SYAKLNSZaMiGkYz7ihfg",
   "version" : {
      "number" : "6.1.1",
      "build_hash" : "bd92e7f",
      "build_date" : "2017-12-17T20:23:25.338Z",
      "build_snapshot" : false,
      "lucene_version" : "7.1.0",
      "minimum_wire_compatibility_version" : "5.6.0",
      "minimum_index_compatibility_version" : "5.0.0"
   },
   "tagline" : "You Know, for Search"
}
```

Установка Kibana

Процесс установки Kibana — сервиса для визуализации данных Elasticsearch — аналогичен подобному процессу для Elasticsearch:

1. Загрузите соответствующий архив Kibana с сайта по адресу: **https://www.elastic.co/downloads/kibana**.
2. Извлеките Kibana из архива.
3. Запустите его с помощью команды: `bin/kibana`.

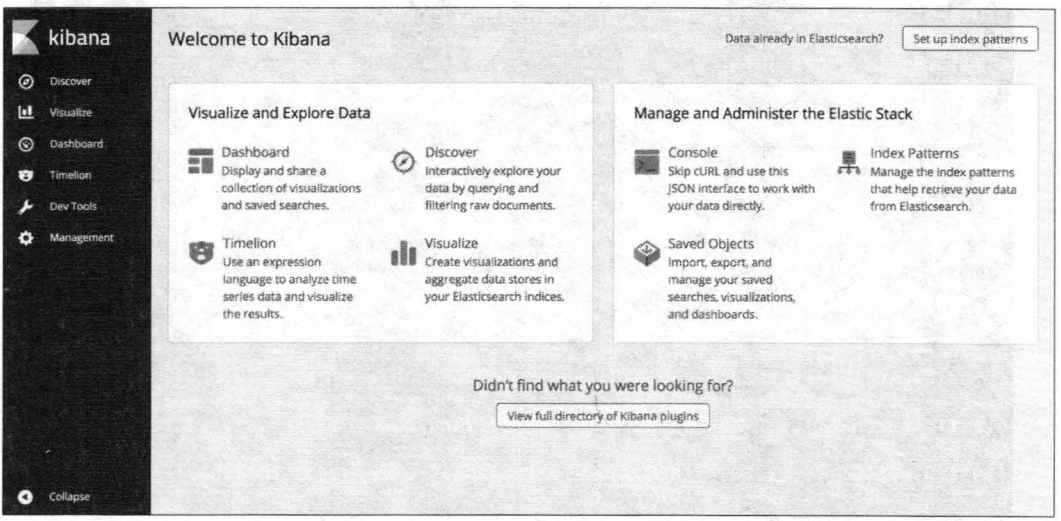


Рис. 9.9. Веб-интерфейс Kibana

После загрузки и запуска Kibana проверьте его работу в окне вашего браузера путем ввода строки адреса **http://localhost:5601/**. Если всё работает, вы познакомитесь с веб-интерфейсом Kibana (рис. 9.9).

Установка Logstash

Установка Logstash — инструмента для сбора, систематизации и анализа логов — выполняется следующим образом:

1. Загрузите соответствующий пакет (в виде ZIP-архива) с сайта, доступного по адресу: **https://www.elastic.co/downloads/logstash**.
2. Распакуйте этот пакет.

Для установки Logstash просто загрузить и запустить пакет недостаточно. Необходимо настроить его, что позволит понять структуру файла журнала Spring. Выполним это путем создания файла конфигурации Logstash. Файл конфигурации Logstash содержит три критических раздела: раздел ввода, раздел фильтра и раздел вывода. Каждый раздел устанавливает плагины, которые важны для файлов журналов. Создайте файл logstash.conf в подходящем каталоге и добавьте в него следующий код:

```
input {
   file {
      type => "java"
      path => "/<path-to-project>/place-reviewer/application.log"
      codec => multiline {
         pattern => "^%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME}.*"
         negate => "true"
         what => "previous"
      }
   }
}

filter {
  #Tag log lines containing tab character followed by 'at' as stacktrace.
   if [message] =~ "\tat" {
      grok {
         match => ["message", "^(\tat)"]
         add_tag => ["stacktrace"]
      }
   }

#Формат журнала, заданный по умолчанию, - Grok Spring Boot
grok {
   match => [ "message",
              "(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME})
```

```
            %{LOGLEVEL:level} %{NUMBER:pid} --- \[(?<thread>
            [A-Za-z0-9-]+)\][A-Za-z0-9.]*\.(?<class>
            [A-Za-z0-9#_]+)\s*:\s+(?<logmessage>.*)",
            "message",
            "(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME})
            %{LOGLEVEL:level} %{NUMBER:pid} --- .+?
            :\s+(?<logmessage>.*)"
   ]
}
# Анализ штампов времени в поле timestamp
date {
   match => [ "timestamp" , "yyyy-MM-dd HH:mm:ss.SSS" ]
   }
}
output {
   # Print each event to stdout and enable rubydebug.
   stdout {
      codec => rubydebug
   }
   # Send parsed log events to Elasticsearch
   elasticsearch {
      hosts => ["127.0.0.1"]
   }
}
```

Пояснение действий, выполняемых плагинами из приведенного фрагмента кода, выходит за рамки этой книги. Для облегчения понимания сути происходящего в него добавлены комментарии.

Измените `path` в плагине файла из раздела ввода, присвоив ему значение абсолютного пути для файла application.log приложения Place Reviewer.

Завершив работу с файлом конфигурации Logstash, запустите Logstash с помощью следующей команды:

```
/bin/logstash -f logstash.conf
```

Если все настроено правильно, Logstash должен начать сохранение событий журнала. Последнее, что следует выполнить — настроить Kibana для чтения накопленных данных.

Настройка Kibana

Kibana легко настроить для чтения журналов, которые сохранялись в индексе поиска Elasticsearch. Получите доступ к веб-интерфейсу (**http://localhost:5601/**) и перейдите на страницу управления настройками, щелкнув на кнопке **Management** (Управление) в левой части панели навигации веб-интерфейса Kibana (см. рис. 9.9). Первым шагом в настройке Kibana служит создание шаблона индекса. Щелкните на

странице управления настройками на ссылке **Index Patterns** (Шаблоны индекса) для выбора распознанных Kibana шаблонов индексов (рис. 9.10).

Поскольку вы ранее не создавали в Kibana шаблон индекса, вам поступит предложение выполнить это (рис. 9.11).

Рис. 9.10. Страница управления настройками Kibana

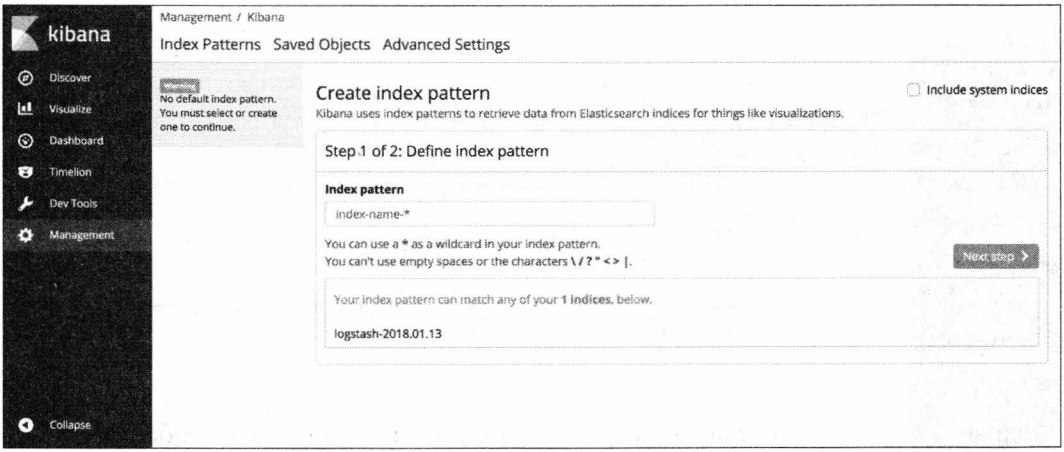


Рис. 9.11. Предложение создать в Kibana шаблон индекса

Введите в поле **Index pattern** (Шаблон индекса) имя одного из распознанных Kibana индексов (оно уже отображено в самом нижнем поле экрана) и перейдите к следующему шагу, нажав на кнопку **Next step**.

Теперь выберите в раскрывающемся списке **Time Filter field name** (Имя поля временного фильтра) `@timestamp` в качестве имени поля фильтра времени (рис. 9.12) и завершите создание шаблона индекса щелчком на кнопке **Create Index pattern** (Создать шаблон индекса). Можно в любое время управлять созданными шаблона-

ми индекса на странице управления настройками, выбирая команду **Index Patterns** (Шаблоны индекса).

Проверьте сохраненные шаблоны (рис. 9.13): если вы увидите только что созданный шаблон, тогда примите поздравления — вы успешно настроили Kibana!

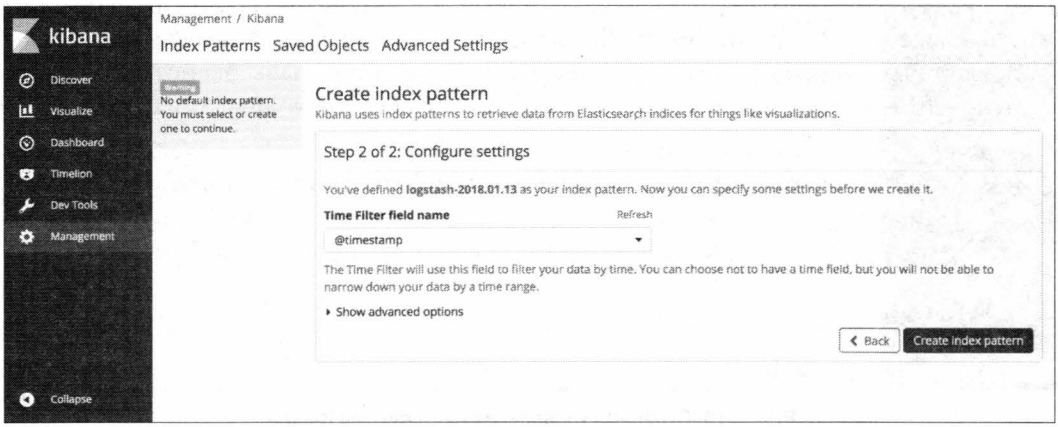


Рис. 9.12. Выбор имени временного фильтра

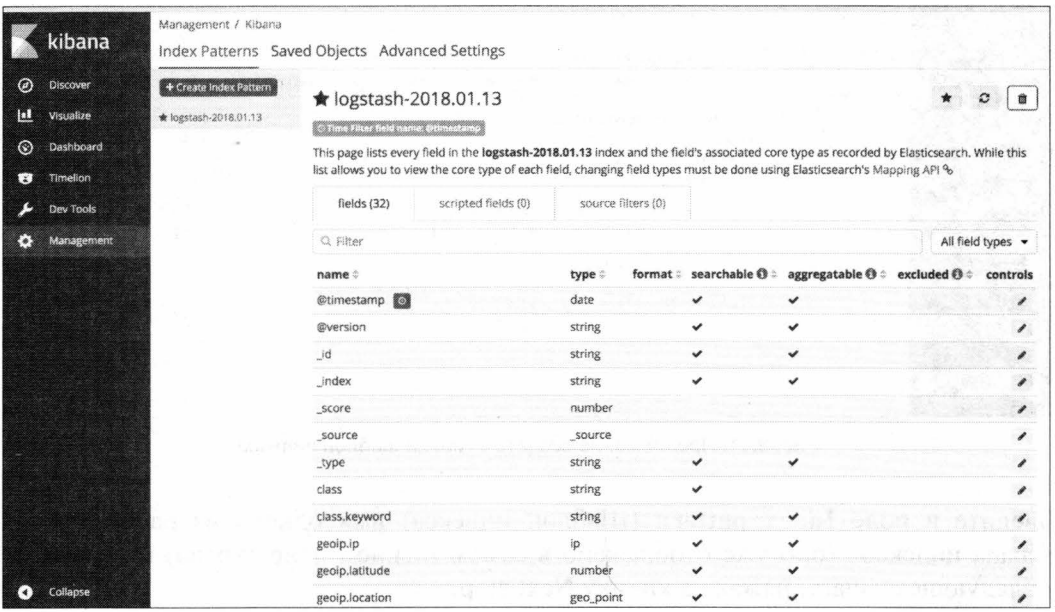


Рис. 9.13. Сервис Kibana успешно настроен

Подведем итоги

В этой главе был сделан упор на изучение языка Kotlin и его возможностей при разработке веб-платформ на примере реализации серверной части приложения Place Reviewer. Кроме того, было показано, как настроить проект Spring Framework, использующий Spring MVC для создания современных приложений, следующих шаблону проектирования Model-View-Controller (модель-представление-контроллер), а также как настроить Spring Security для предотвращения неаутентифицированного доступа к веб-приложениям Spring. В завершение главы вы познакомились со стеком ELK и узнали, как его применить для управления журналами сервера.

В следующей главе мы закончим создание приложения Place Reviewer, реализовав его интерфейс. На примере реализации веб-интерфейса приложения будет показано, как создавать многофункциональные веб-приложения с помощью API Google Places и тестировать веб-приложения, созданные с помощью Spring Framework.

10

Реализация веб-интерфейса приложения Place Reviewer

В предыдущей главе мы продолжили исследование языка Kotlin как универсального средства для создания веб-приложений, начав разработку веб-приложения Place Reviewer. В начале главы мы рассмотрели шаблон проектирования Model-View-Controller (модель-представление-контроллер) и основательно изучили используемые в приложениях MVC основные компоненты: модели, представления и контроллеры. Получив четкое представление о шаблоне проектирования MVC и его функциональных особенностях, мы приступили к разработке и реализации серверной части приложения Place Reviewer.

Прежде всего, мы описали предполагаемые варианты использования этого приложения. Затем определили необходимые для создания приложения данные, которые будут задействованы в принятых вариантах его использования. А после определения этих данных приступили к разработке бэкэнда (серверной части приложения) — создали базу данных Postgres, с которой будет взаимодействовать приложение, и реализовали необходимые для него сущности и модели.

В предыдущей главе было также показано, как защитить приложение, — с помощью аутентификации на основе Spring Security, но на этот раз без использования JWT. И в завершение главы мы остановились на проблемах создания контроллеров для приложений на основе Spring MVC и управления журналами сервера с помощью стека ELK.

В этой главе мы завершим создание приложения Place Reviewer, реализовав его интерфейс. Вы также узнаете:

- о работе с Google Places API;
- о тестировании приложения.

А теперь перейдем к материалу главы, рассмотрев процесс реализации представлений для нашего приложения.

Создание представлений с помощью библиотеки шаблонов Thymeleaf

Как уже отмечалось ранее, собственно представление (View) является представлением данных, имеющихся в приложении и генерируемых им. Представления служат основными точками взаимодействия пользователя с приложением, созданным по шаблону MVC. Уровень представления может опираться на различные технологии, обеспечивающие визуализацию пользовательской информации. В Spring поддерживается несколько вариантов представлений, также называемых *шаблонами*. Поддержка шаблонов в приложении Spring обеспечивается *механизмом шаблонов*. Проще говоря, механизм шаблонов позволяет использовать статические файлы шаблонов со слоем представления приложения. Механизм шаблонов также называется *библиотекой шаблонов*. Для совместного использования вместе со Spring доступны следующие библиотеки шаблонов:

- Thymeleaf;
- JSP/JSTL;
- Freemaker;
- Tiles;
- Velocity.

Приведенный перечень не является исчерпывающим, поскольку имеется целый ряд других библиотек шаблонов, доступных для использования со Spring. Но в нашем случае для обеспечения поддержки процессов обработки шаблонов для нашего приложения мы воспользуемся библиотекой шаблонов Thymeleaf. Если вы помните, поддержка шаблонов была включена в Thymeleaf при первоначальном создании нашего проекта. Изучение раздела зависимостей в файле pom.xml проекта покажет, как добавить Thymeleaf в проект:

```
<dependencies>
...
<dependency>
<groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
...
</dependencies>
```

На этом месте вам может потребоваться более формальное определение платформы Thymeleaf. Как отмечено на официальном сайте Thymeleaf: *Thymeleaf — это современный серверный шаблонизатор Java для веб-приложений и автономных сред. Основной целью Thymeleaf является привнесение элегантных естественных шаблонов в рабочий процесс разработки HTML-кода, который сможет корректно отображаться в браузерах, а также функционировать в качестве статических прототипов, обеспечивая более тесное сотрудничество в группах разработчиков.*

Если вам необходимо более основательное понимание возможностей и целей Thymeleaf, подробная информация о нем находится на сайте по адресу: **http:// thymeleaf. org**.

В предыдущей главе мы рассмотрели простой пример создания представления в Spring, где была показана реализация представления `hello.html` для контроллера `HelloController`. Однако созданное представление отображало сообщение `Hello world!` только для своих зрителей. В этой главе мы создадим более сложные представления и начнем с создания представления, облегчающего регистрацию пользователей на платформе.

Реализация представления регистрации пользователей

В этом разделе мы преследуем две задачи. Во-первых, создадим слой представления, облегчающий регистрацию на платформе Place Reviewer новых пользователей. Во-вторых, создадим подходящие контроллеры и действия, что позволит пользователю располагать представлением регистрации и обрабатывать представления формы регистрации. Достаточно просто, верно? Рад, что вы тоже так считаете! Создайте в проекте Place Reviewer шаблон register.html. Напомним, что все файлы шаблонов находятся в каталоге шаблонов в разделе ресурсов. Теперь добавьте в файл следующий шаблон HTML:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
   <title>Register</title>
   <link rel="stylesheet" th:href="@{/css/app.css}"/>
   <link rel="stylesheet"
   href="/webjars/bootstrap/4.0.0-beta.3/css/bootstrap.min.css"/>

   <script src="/webjars/jquery/3.2.1/jquery.min.js"></script>
   <script src="/webjars/bootstrap/4.0.0-beta.3/
   js/bootstrap.min.js"></script>
</head>
<body>
   <nav class="navbar navbar-default nav-enhanced">
      <div class="container-fluid container-nav">
         <div class="navbar-header">
            <div class="navbar-brand">
               Place Reviewer
            </div>
         </div>
         <ul class="navbar-nav" th:if="${principal != null}">
            <li>
               <form th:action="@{/logout}" method="post">
```

```
                <button class="btn btn-danger" type="submit">
                    <i class="fa fa-power-off" aria-hidden="true"></i>
                    Sign Out
                </button>
            </form>
        </li>
    </ul>
</div>
</nav>
<div class="container-fluid" style="z-index: 2; position: absolute">
    <div class="row mt-5">
        <div class="col-sm-4 col-xs-2"> </div>
        <div class="col-sm-4 col-xs-8">
            <form class="form-group col-sm-12 form-vertical form-app"
            id="form-register" method="post"
            th:action="@{/users/registrations}">
                <div class="col-sm-12 mt-2 lead text-center text-primary">
                    Create an account
                </div>
                <hr>
                <input class="form-control" type="text" name="username"
                placeholder="Username" required/>
                <input class="form-control mt-2" type="email" name="email"
                placeholder="Email" required/>
                <input class="form-control mt-2" type="password"
                    name="password"
                placeholder="Password" required/>
                <span th:if="${error != null}" class="mt-2 text-danger"
                style="font-size: 10px" th:text="${error}"></span>
                <button class="btn btn-primary form-control mt-2 mb-3"
                    type="submit">
                    Sign Up!
                </button>
            </form>
        </div>
        <div class="col-sm-4 col-xs-2"></div>
    </div>
</div>
</body>
</html>
```

В этом фрагменте кода для создания шаблона страницы регистрации пользователя применялся HTML-код. Формируемая им веб-страница несложна. Она включает панель навигации и форму, в которую пользователь вводит необходимые регистрационные данные для отправки. Поскольку мы работаем с шаблоном Thymeleaf, неудивительно, что здесь использован ряд специфичных для Thymeleaf атрибутов. Рассмотрим некоторые из них подробнее:

- `th:href` — атрибут модификатора атрибута. При обработке с помощью механизма шаблонов этот атрибут вычисляет интернет-адрес (URL) ссылки и устанавливает его в соответствующем теге, где он используется. Примерами тегов, где может применяться этот атрибут, являются `<a>` и `<link>`. Образец использования атрибута `th:href` в приведенном фрагменте кода показан далее:

```
<link rel="stylesheet" th:href="@{/css/app.css}"/>
```

- `th:action` — этот атрибут функционирует так же, как атрибут HTML-действия. Указывает, куда направлять данные формы при ее отправке. Следующий фрагмент кода показывает, что данные формы должны направляться в конечную точку, имеющую путь **/users/registrations**:

```
<form class="form-group col-sm-12 form-vertical form-app"
   id="form-register" method="post"
   th:action="@{/users/registrations}">
...
</form>
```

- `th:text` — этот атрибут используется для указания содержащегося в контейнере текста:

```
<span th:text="Hello world"></span>
```

- `th:if` — этот атрибут может применяться для указания, следует ли отображать HTML-тег на основе результата проверки условия:

```
<span th:if="${error != null}" class="mt-2 text-danger"
style="font-size: 10px" th:text="${error}"></span>
```

  В данном фрагменте кода, если существует ошибка атрибута модели и ее значение не равно нулю, тег `span` на странице HTML отображается, в противном случае — нет.

Мы также использовали атрибут `th:if` в панели навигации, чтобы указать, когда должна отображаться кнопка, позволяющая пользователю выйти из своей учетной записи:

```
<ul class="navbar-nav" th:if="${principal != null}">
   <li>
      <form th:action="@{/logout}" method="post">
         <button class="btn btn-danger" type="submit">
            <i class="fa fa-power-off" aria-hidden="true"></i> Sign Out
         </button>
      </form>
   </li>
</ul>
```

Если атрибут `principal` модели установлен в шаблоне и он не является `null`, кнопка выхода пользователя из своей учетной записи отображается. Атрибут `principal` будет `null` до тех пор, пока пользователь не войдет в свою учетную запись.

Добавление панели навигации в шаблон напрямую может, на первый взгляд, показаться удачным шагом, однако лучше бы мы сначала обдумали, правильно ли это... Достаточно часто случается так, что элемент DOM панели навигации применяется в приложении неоднократно. Поэтому нет смысла постоянно переписывать в шаблонах один и тот же фрагмент кода, создающий панель навигации. И для устранения ненужных повторов мы реализуем панель навигации как фрагмент, который можно включать в шаблон в любой момент.

Создайте в каталоге templates каталог fragments и добавьте в него файл navbar.html, содержащий следующий код:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
   <head>
       <meta charset="UTF-8">
   </head>
   <body>
   <nav class="navbar navbar-default nav-enhanced" th:fragment="navbar">
      <div class="container-fluid container-nav">
         <div class="navbar-header">
            <div class="navbar-brand">
               Place Reviewer
            </div>
         </div>
         <ul class="navbar-nav" th:if="${principal != null}">
            <li>
               <form th:action="@{/logout}" method="post">
                  <button class="btn btn-danger" type="submit">
                     <i class="fa fa-power-off"
                        aria-hidden="true"></i>Sign Out
                  </button>
               </form>
            </li>
         </ul>
      </div>
   </nav>
   </body>
</html>
```

В этом блоке кода определен фрагмент панели навигации, доступный для включения в шаблоны с помощью атрибута `th:fragment`. Данный фрагмент может быть в любой момент включен в шаблон с помощью атрибута `th:insert`. Модифицируйте HTML-код тега `<body>` в файле register.html для использования фрагмента панели навигации следующим образом:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

```
<head>
   <title>Register</title>
   <link rel="stylesheet" th:href="@{/css/app.css}"/>
   <link rel="stylesheet"
   href="/webjars/bootstrap/4.0.0-beta.3/css/bootstrap.min.css"/>
   <script src="/webjars/jquery/3.2.1/jquery.min.js"></script>
   <script src="/webjars/bootstrap/4.0.0-beta.3/
   js/bootstrap.min.js"></script>
</head>
<body>
   <div th:insert="fragments/navbar :: navbar"></div>
   <!-- Вставка фрагмента навигации -->
      <div class="container-fluid" style="z-index: 2;
         position: absolute">
         <div class="row mt-5">
            <div class="col-sm-4 col-xs-2">
            </div>
            <div class="col-sm-4 col-xs-8">
               <form class="form-group col-sm-12
                  form-vertical form-app"
                     id="form-register" method="post"
                     th:action="@{/users/registrations}">
                        <div class="col-sm-12 mt-2 lead
                           text-center text-primary">
                           Create an account
                        </div>
                        <hr>
                        <input class="form-control" type=
                           "text" name="username"
                        placeholder="Username" required/>
                        <input class="form-control mt-2" type=
                           "email" name="email"
                        placeholder="Email" required/>
                        <input class="form-control
                           mt-2" type="password"
                        name="password" placeholder=
                           "Password" required/>
                        <span th:if="${error != null}" class=
                           "mt-2 text-danger"
                        style="font-size: 10px"
                           th:text="${error}"></span>
                        <button class="btn btn-primary
                           form-control mt-2 mb-3"
                           type="submit">
                           Sign Up!
```

```
                        </button>
                    </form>
                </div>
                <div class="col-sm-4 col-xs-2"></div>
            </div>
        </div>
    </body>
</html>
```

Как можно здесь видеть, разделение панели навигации HTML приводит к более лаконичному коду и улучшает качество разработанных нами шаблонов.

Сделав необходимый шаблон для страницы регистрации пользователя, нужно сформировать контроллер, отображающий этот шаблон для посетителя сайта. Создадим контроллер приложения. Функция контроллера состоит в предоставлении пользователю по запросу веб-страниц приложения Place Reviewer.

Добавьте показанный далее класс `ApplicationController` в пакет controller:

```
package com.example.placereviewer.controller

import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.GetMapping

@Controller
class ApplicationController {

    @GetMapping("/register")
    fun register(): String {
        return "register"
    }
}
```

В этом фрагменте кода не выполняется ничего особенного. Мы создали MVC-контроллер с одним действием, которое обрабатывает HTTP-запрос `GET` для пути `/register`, отображая представление register.html для пользователя.

К этому моменту уже почти всё готово для просмотра только что созданной нами страницы регистрации. Но прежде чем сделать это, следует добавить файл app.css, который необходим для register.html. Статические ресурсы, такие как CSS-файлы, нужно добавлять к каталогу static в каталоге resource приложения. Включите каталог css в каталог static и поместите в него файл app.css, содержащий следующий код:

```
//app.css
.nav-enhanced {
    background-color: #00BFFF;
    border-color: blueviolet;
    box-shadow: 0 0 3px black;
}
```

```
.container-nav {
   height: 10%;
   width: 100%;
   margin-bottom: 0;
}

.form-app {
   background-color: white;
   box-shadow: 0 0 1px black;
   margin-top: 50px !important;
   padding: 10px 0;
}
```

Отличная работа! Теперь вы можете запустить приложение Place Reviewer. После его запуска откройте свой браузер и зайдите на веб-страницу, доступную по адресу: **http://localhost:5000/register**.

Пришло время реализовать логику, связанную с регистрацией пользователя. Для этого следует объявить действие, которое принимает данные формы, направленные регистрационной формой, и соответствующим образом обрабатывает эти данные с целью успешной регистрации пользователя на платформе. Как вы помните, мы договаривались, что данные формы должны быть направлены с помощью запроса `POST` по пути **/users/registrations**.

Следовательно, необходимо действие, обрабатывающее подобный HTTP-запрос. Добавьте класс `UserController` в пакет com.example.placereviewer.controller с помощью приведенного далее кода:

```
package com.example.placereviewer.controller

import com.example.placereviewer.component.UserValidator
import com.example.placereviewer.data.model.User
import com.example.placereviewer.service.SecurityService
import com.example.placereviewer.service.UserService
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.validation.BindingResult
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.ModelAttribute
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestMapping

@Controller
@RequestMapping("/users")
class UserController(val userValidator: UserValidator,
   val userService: UserService, val securityService: SecurityService) {
```

```
    @PostMapping("/registrations")
    fun create(@ModelAttribute form: User, bindingResult: BindingResult,
        model: Model): String {

        userValidator.validate(form, bindingResult)
        if (bindingResult.hasErrors()) {
            model.addAttribute("error", bindingResult.allErrors.first()
                .defaultMessage)
            model.addAttribute("username", form.username)
            model.addAttribute("email", form.email)
            model.addAttribute("password", form.password)
            return "register"
        }

        userService.register(form.username, form.email, form.password)
        securityService.autoLogin(form.username, form.password)
        return "redirect:/home"
    }
}
```

Здесь функция `create()` обрабатывает HTTP-запросы `POST`, направленные по пути **/users/registrations**. Она использует три аргумента. Первым из них является `form`, который служит объектом для класса `User`. Аргумент `@ModelAttribute` применяется для аннотации формы и указывает, что аргумент извлекается моделью. Атрибут модели формы заполняется данными, представленными формой для конечной точки. В регистрационной форме представлены все параметры: `username`, `email` и `password`. А все объекты типа `User` имеют свойства `username`, `email` и `password` — следовательно, указанные в форме данные присваиваются соответствующим свойствам модели.

Второй аргумент функции служит экземпляром для аргумента `BindingResult`, который является хранителем результата для `DataBinder`. В нашем случае он используется для связки результатов выполняемого `UserValidator` процесса проверки, которую мы собираемся сформировать в ближайшее время. Третьим аргументом является `Model`. Мы применяем его для добавления атрибутов в нашу модель для последующего доступа к ней со стороны слоя представления.

Прежде объяснять логику, которая реализована в действии `create()`, следует реализовать как `UserValidator`, так и `SecurityService`. Класс `UserValidator` служит единственной задаче — проверяет информацию о пользователе, которая представлена бэкэнду. Создайте пакет com.example.placereviewer.component и включите в него класс `UserValidator`:

```
package com.example.placereviewer.component

import com.example.placereviewer.data.model.User
import com.example.placereviewer.data.repository.UserRepository
import org.springframework.stereotype.Component
```

```
import org.springframework.validation.Errors
import org.springframework.validation.ValidationUtils
import org.springframework.validation.Validator

@Component
class UserValidator(private val userRepository:
   UserRepository) : Validator
{

   override fun supports(aClass: Class<*>?): Boolean {
      return User::class == aClass
   }

   override fun validate(obj: Any?, errors: Errors) {
      val user: User = obj as User
```

Далее выполняется проверка, в ходе которой определяется, будут ли не пусты отправленные пользовательские параметры. Пустой параметр отклоняется с кодом ошибки и сообщением о ней:

```
ValidationUtils.rejectIfEmptyOrWhitespace(errors, "username",
         "Empty.userForm.username", "Username cannot be empty")
ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password",
         "Empty.userForm.password", "Password cannot be empty")
ValidationUtils.rejectIfEmptyOrWhitespace(errors, "email",
         "Empty.userForm.email", "Email cannot be empty")
```

Далее выполняется проверка длины предоставленного имени пользователя. Имя, длина которого меньше 6 символов, отклоняется:

```
      if (user.username.length < 6) {
         errors.rejectValue("username", "Length.userForm.username",
                     "Username must be at least 6 characters in length")
      }
```

Теперь проводится проверка, существует ли предоставленное имя пользователя. Имя, которое уже вводилось ранее, отвергается:

```
if (userRepository.findByUsername(user.username) != null) {
   errors.rejectValue("username", "Duplicate.userForm.username",
                     "Username unavailable")
}
```

Далее выполняется проверка длины отправленного пароля. Пароли длиной менее 8 символов, отклоняются:

```
      if (user.password.length < 8) {
         errors.rejectValue("password", "Length.userForm.password",
                     "Password must be at least 8 characters in length")
      }
   }
}
```

Класс `UserValidator` реализует интерфейс `Validator`, который используется для проверки объектов. Этот метод переопределяет два метода: `supports(Class<*>?)` и `validate(Any?, Errors)`. Функция `supports()` применяется для утверждения, что валидатор может проверять предоставленный ему объект. При использовании класса `UserValidator` метод `supports()` указывает, что предоставленный объект является экземпляром класса `User`. Следовательно, все объекты типа `User` поддерживаются для подтверждения `UserValidator`.

Функция `validate()` проверяет предоставленные объекты. В тех случаях, когда происходит отклонение валидации, регистрируется ошибка с предоставлением объекта `Error`. Убедитесь, что вы ознакомились с комментариями, сопровождающими код метода `validate()` в файлах примеров исходного кода, размещенных в сопровождающем книгу файловом архиве, — это позволит вам лучше представить, что именно происходит внутри метода.

Теперь обработаем `SecurityService`. Реализация `SecurityService` выполняется для упрощения идентификации текущего пользователя и автоматического входа пользователя после его регистрации.

Добавьте интерфейс `SecurityService` в com.example.placereviewer.service:

```
package com.example.placereviewer.service

interface SecurityService {
   fun findLoggedInUser(): String?
   fun autoLogin(username: String, password: String)
}
```

Теперь добавьте класс `SecurityServiceImpl` в пакет com.example.placereviewer.service. Как следует из его имени, класс `SecurityServiceImpl` реализует интерфейс `SecurityService`:

```
package com.example.placereviewer.service

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.security.authentication.AuthenticationManager
import
org.springframework.security.authentication.
   UsernamePasswordAuthenticationToken
import org.springframework.security.core.context.SecurityContextHolder
import org.springframework.security.core.userdetails.UserDetails
import org.springframework.stereotype.Service

@Service
class SecurityServiceImpl(private val userDetailsService:
AppUserDetailsService)
      : SecurityService {

   @Autowired
   lateinit var authManager: AuthenticationManager
```

```
    override fun findLoggedInUser(): String? {
        val userDetails = SecurityContextHolder.getContext().authentication.details

        if (userDetails is UserDetails) {
            return userDetails.username
        }

        return null
    }

    override fun autoLogin(username: String, password: String) {
        val userDetails: UserDetails = userDetailsService
                                            .loadUserByUsername(username)
        val usernamePasswordAuthenticationToken =
            UsernamePasswordAuthenticationToken(userDetails, password,
            userDetails.authorities)

        authManager.authenticate(usernamePasswordAuthenticationToken)

        if (usernamePasswordAuthenticationToken.isAuthenticated) {
            SecurityContextHolder.getContext().authentication =
                usernamePasswordAuthenticationToken
        }
    }
}
```

Функция `findLoggedInUser()` возвращает имя вошедшего в систему пользователя. Выборка имени пользователя осуществляется с помощью класса `SecurityContextHolder` из Spring Framework. Экземпляр класса `UserDetails` извлекается путем доступа к авторизованным данным пользователя с помощью вызова `SecurityContextHolder.getContext().authentication.details`. Важно отметить, что этот вызов возвращает `Object`, а не экземпляр `UserDetails`. То есть следует выполнить проверку типа, и это позволит подтвердить, что полученный объект также соответствует типу `UserDetails`. Если это так, возвращается имя пользователя, который в текущий момент вошел в систему. В противном случае возвращается `null`.

Метод `autoLogin()` применяется для простой задачи аутентификации пользователя после регистрации на платформе. Представленное имя пользователя и его пароль передаются в качестве аргументов метода `autoLogin()`, после чего для зарегистрированного пользователя создается экземпляр `UsernamePasswordAuthenticationToken`. После создания этого экземпляра для аутентификации токена пользователя применяется `AuthenticationManager`. Если `UsernamePasswordAuthenticationToken` успешно аутентифицирован, к нему устанавливается свойство аутентификации текущего пользователя.

Включив необходимые дополнения в класс, вернемся к `UserController` для завершения пояснений действия по его созданию. В методе `create()` представленная форма

ввода в первую очередь проверяется в соответствии с экземпляром `UserValidator`. Ошибки, возникающие в ходе проверки данных формы, связаны с экземпляром `BindingResult`, который добавлен в контроллер с помощью Spring. Рассмотрим эти строки кода:

```
if (bindingResult.hasErrors()) {
   model.addAttribute("error", bindingResult.allErrors
                                          .first().defaultMessage)
   model.addAttribute("username", form.username)
   model.addAttribute("email", form.email)
   model.addAttribute("password", form.password)
   return "register"
}
```

Класс `bindingResult` сначала проверяется для уточнения, произошли ли ошибки во время проверки данных формы. Если имеются ошибки, извлекаем сообщение о первой обнаруженной ошибке и устанавливаем атрибут ошибки `model` таким образом, чтобы он содержал сообщение об ошибке для последующего доступа с помощью представления. Кроме того, создаем атрибуты `model` для хранения каждого ввода, представленного пользователем. Наконец, повторно отображаем представление регистрации для пользователя.

Обратите внимание: в предыдущем фрагменте кода несколько вызовов методов сделаны для одного экземпляра `Model`. Однако имеется более удобный способ для выполнения этого — с применением функции Kotlin `with`:

```
if (bindingResult.hasErrors()) {
   with (model) {
      addAttribute("error", indingResult.allErrors.first().defaultMessage)
      addAttribute("error", form.username)
      addAttribute("email", form.email)
      addAttribute("password", form.password)
   }
   return "register"
}
```

Посмотрите, как легко и удобно применять эту функцию! Теперь изменим контроллер `UserController` для использования так, как показано в предыдущем коде.

Вас может удивить наше решение по сохранению предоставленных пользователем данных в атрибутах модели? Это сделано с тем, чтобы получить возможность восстановить содержащиеся в регистрационной форме данные, а именно первоначально направленные данные до повторной визуализации представления регистрационной формы. Это поможет освободить пользователя от ввода заново всех данных формы, если только один введенный показатель формы окажется недопустимым.

Если вводимые пользователем данные недопустимы, выполняется следующий код:

```
userService.register(form.username, form.email, form.password)
 securityService.autoLogin(form.username, form.password)
return "redirect:/home"
```

Как и ожидалось, если предоставленные пользователем данные действительны, он регистрируется на платформе, автоматически входит в свою учетную запись и перенаправляется на свою домашнюю страницу. Прежде чем опробовать регистрационную форму, необходимо сделать две вещи:

- примените атрибуты модели, указанные в register.html;
- создайте шаблон home.html и контроллер для визуализации шаблона.

Всё это несложно выполнить. Сначала примените атрибуты модели, для чего измените содержащуюся в register.html форму следующим образом:

```
<form class="form-group col-sm-12 form-vertical form-app"
id="form-register" method="post" th:action="@{/users/registrations}">
<div class="col-sm-12 mt-2 lead text-center text-primary">
   Create an account
</div>
<hr>
<!-- Примененные атрибуты модели с th:value -->
<input class="form-control" type="text" name="username"
placeholder="Username" th:value="${username}" required/>
<input class="form-control mt-2" type="email" name="email"
placeholder="Email" th:value="${email}" required/>
<input class="form-control mt-2" type="password" name="password"
placeholder="Password" th:value="${password}" required/>
<span th:if="${error != null}" class="mt-2 text-danger"
style="font-size: 10px" th:text="${error}"></span>
<button class="btn btn-primary form-control mt-2 mb-3" type="submit">
   Sign Up!
</button>
</form>
```

Обнаружили ли вы внесенные изменения? Если вы отметите, что атрибут `th:value` шаблона Thymeleaf использован для предварительной установки хранящегося на входах формы значения с учетом соответствующих значений атрибута `model`, вы будете правы. Теперь создадим простой шаблон home.html. Добавьте шаблон home.html в каталог templates:

```
<html>
   <head>
      <title> Home</title>
   </head>
      <body>
        You have been successfully registered and are now in your home page.
```

```
    </body>
</html>
```

Теперь обновите `ApplicationController` для включения действия, обрабатывающего запросы `GET` в `/home`, как показано далее:

```
package com.example.placereviewer.controller

import com.example.placereviewer.service.ReviewService
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.web.bind.annotation.GetMapping
import java.security.Principal
import javax.servlet.http.HttpServletRequest

@Controller
class ApplicationController(val reviewService: ReviewService) {
   @GetMapping("/register")
   fun register(): String {
      return "register"
   }

   @GetMapping("/home")
   fun home(request: HttpServletRequest, model: Model,
            principal: Principal): String {
      val reviews = reviewService.listReviews()

      model.addAttribute("reviews", reviews)
      model.addAttribute("principal", principal)

      return "home"
   }
}
```

Действие `home` извлекает список всех представлений, которые сохраняются в базе данных. Кроме того, действие `home` устанавливает атрибут `model`, определяющий блок данных, содержащий информацию о текущем, вошедшем в систему, пользователе. Наконец, действие `home` отображает для пользователя домашнюю страницу.

Сделав необходимые приготовления, зарегистрируем пользователя на платформе Place Reviewer. Создайте и запустите приложение, откройте страницу регистрации в браузере: **http://localhost:5000/register**. Первым делом желательно убедиться, функционируют ли наши проверки формы, для чего введем и отправим неверные данные формы (рис. 10.1).

Как можно заметить, ошибка была обнаружена со стороны `UserValidator`, успешно установлена связь с `BindingResult` и выведено соответствующее сообщение об ошибке представления. Смело вводите неверные данные для других форм ввода

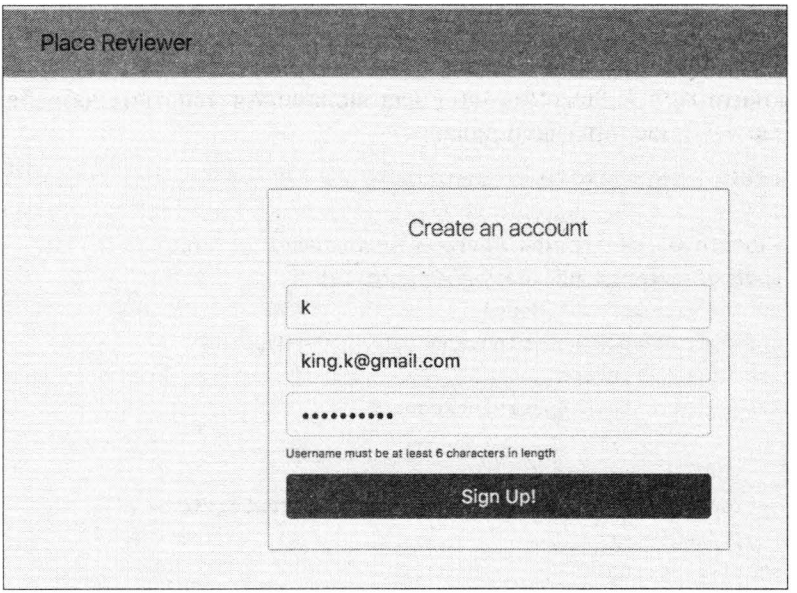


Рис. 10.1. Вводим и отправляем неверные данные формы

и проверяйте, что прочие реализованные проверки функционируют так, как и ожидалось. Теперь убедимся, функционирует ли логика регистрации. Введем `king.kevin`, `king.k@gmail.com` и `Kingsman406` в поля для имени пользователя, адреса электронной почты и пароля соответственно и щелкнем на кнопке **Sign Up!**. Будет создан новый аккаунт, и вы получите доступ к домашней странице (рис. 10.2).

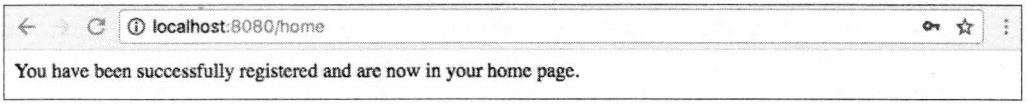


Рис. 10.2. Новый аккаунт создан

Для вас не должно стать сюрпризом, что по мере чтения этой главы в домашнюю страницу будут внесены серьезные изменения. Однако обратим пока внимание на создание подходящей страницы входа пользователя.

Реализация представления входа

Подобно тому как было реализовано представление регистрации пользователей, сначала следует разработать шаблон представления. Шаблон, необходимый для представления входа в систему, имеет форму, куда в качестве входных данных вводится имя пользователя и его пароль при входе в систему. Также необходимо сформировать кнопку, облегчающую отправку в систему формы входа, — в конце концов, какой смысл в форме, если она не может быть отправлена. Кроме того, должно иметься средство оповещения пользователя, если процесс входа

в систему будет нарушен, — например, в случае, если пользователь введет недопустимую комбинацию имени пользователя и пароля. Наконец, следует предоставить ссылку на страницу регистрации учетной записи, если у пользователя страницы входа в систему еще нет учетной записи.

Итак, определив, что́ необходимо для реализации шаблона, приступим к его созданию. Добавьте в каталог шаблонов файл login.html. Как всегда, сначала следует включить в шаблон необходимые таблицы стилей и сценарии. Это выполнено в следующем фрагменте кода:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
   <head>
      <title>Login</title>
      <link rel="stylesheet" th:href="@{/css/app.css}"/>
      <link rel="stylesheet"
      href="/webjars/bootstrap/4.0.0-beta.3/css/bootstrap.min.css"/>

      <script src="/webjars/jquery/3.2.1/jquery.min.js"></script>
      <script src="/webjars/bootstrap/4.0.0-beta.3/
      js/bootstrap.min.js"></script>
   </head>
   <body>
   </body>
</html>
```

Добавив необходимые для шаблона стиль и код JavaScript, теперь можно работать над разделом <body> шаблона. Как уже упоминалось, основной раздел HTML-шаблона содержит элементы DOM, которые отображаются пользователю при загрузке страницы. Добавьте следующий код в поле тега <body> в login.html:

```
<div th:insert="fragments/navbar :: navbar"></div>

<div class="container-fluid" style="z-index: 2; position: absolute">
   <div class="row mt-5">
      <div class="col-sm-4 col-xs-2"></div>
      <div class="col-sm-4 col-xs-8">
         <form class="form-group col-sm-12 form-vertical form-app"
         id="form-login" method="post" th:action="@{/login}">
            <div class="col-sm-12 mt-2 lead text-center text-primary">
               Login to your account
            </div>
            <hr>
            <input class="form-control" type="text" name="username"
            placeholder="Username" required/>
            <input class="form-control mt-2" type="password"
            name="password" placeholder="Password" required/>
```

```
            <span th:if="${param.error}" class="mt-2 text-danger"
               style="font-size: 10px">
               Invalid username and password combination
            </span>
            <button class="btn btn-primary form-control mt-2 mb-3"
               type="submit">
               Go!
            </button>
         </form>
         <div class="col-sm-12 text-center" style="font-size: 12px">
            Don't an account? Register <a href="/register">here</a>
         </div>
      </div>
      <div class="col-sm-4 col-xs-2">
      <div th:if="${param.logout}"
         class="col-sm-12 text-success text-right">
         You have been logged out.
      </div>
   </div>
   </div>
</div>
```

После добавления основного раздела (body) созданный HTML-код хорошо описывает структуру страницы входа. Наряду с добавлением необходимой формы, на страницу мы поместили ранее созданный фрагмент навигационной панели, что позволило не повторять здесь его код. Мы также добавили средство, с помощью которого пользователю предоставляется обратная связь, относящаяся к ошибкам, которые могут возникнуть в процессе входа в систему. Это выполнено посредством следующих строк:

```
<span th:if="${param.error}" class="mt-2 text-danger"
   style="font-size: 10px">
   Invalid username and password combination
</span>
```

Если определяется param.error, это значит, что во время входа в систему произошла ошибка, и для пользователя отображается сообщение **Invalid username and password combination** (Некорректная комбинация имени пользователя и пароля). Следует иметь в виду, что помимо страницы входа, часто являющейся первой точкой контакта пользователя с веб-приложением, подобное отображение также может быть и последней точкой контакта пользователя с приложением во время сеанса взаимодействия. Это случается при выходе пользователя из системы. После завершения взаимодействия пользователя с приложением и выхода из системы его следует перенаправить на экран входа в систему. Для реализации такой возможности внесем текст, уведомляющий пользователя о том, что он вышел из системы:

```
<div th:if="${param.logout}" class="col-sm-12 text-success text-right">
   You have been logged out.
</div>
```

Тег `<div>` отображается для пользователя после успешного его выхода из учетной записи.

Сейчас нам следует реализовать контроллер для визуализации login.html, но, как вы помните, это уже сделано при использовании настраиваемой конфигурации Spring MVC с помощью реализации класса `MvcConfig`, что и показано в следующем фрагменте кода:

```
...
override fun addViewControllers(registry: ViewControllerRegistry?) {
   registry?.addViewController("/login")?.setViewName("login")
}
```

Для добавления представления контроллера, обрабатывающего запросы к `/login`, и установки представления, которое используется для только что внедренного шаблона входа, применим экземпляр класса `ViewControllerRegistry`. Создайте и запустите приложение для просмотра реализованного сейчас нами представления (рис. 10.3). Веб-страница может быть доступна по адресу **http:localhost:5000/login**.

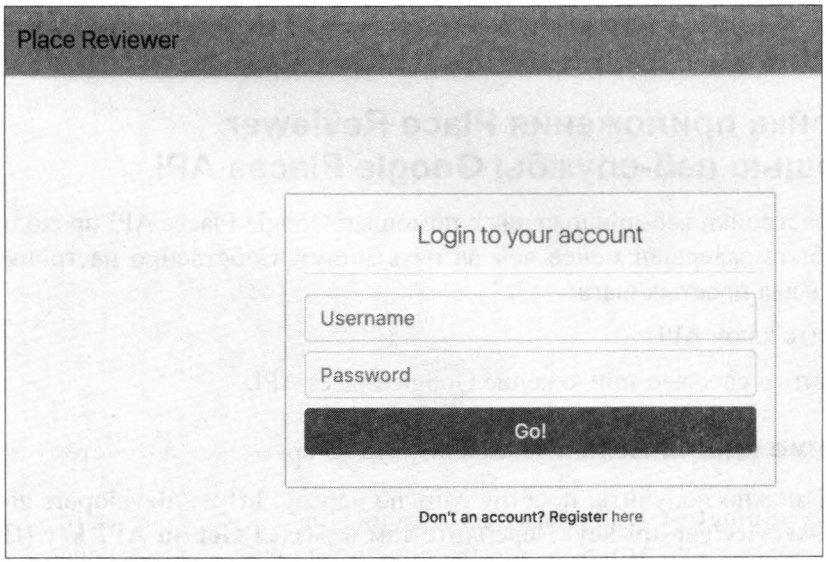


Рис. 10.3. Форма входа в приложение зарегистрированного пользователя

Попытка войти с неверными учетными данными пользователя вызовет отображение ожидаемого сообщения об ошибке (рис. 10.4).

А вход с действительными учетными данными приводит на домашнюю страницу приложения. Но прежде чем обратиться к этой домашней странице, нужно еще по-

трудиться над ее отображением. Однако с этого момента нам следует напрямую работать с Google Places API, поэтому необходимо сначала настроить приложение на работу с ним.

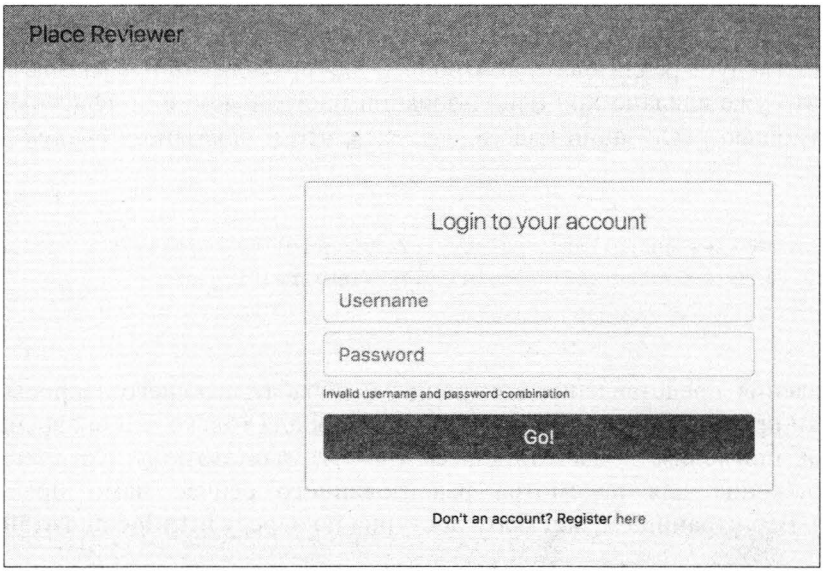


Рис. 10.4. Попытка войти в приложение с неверными учетными данными пользователя

Настройка приложения Place Reviewer с помощью веб-службы Google Places API

Процесс настройки веб-приложения с помощью Google Places API проходит быстро и может быть завершен менее чем за пять минут. Собственно настройка требует выполнить два простых шага:

1. Получить ключ API.
2. Включить в свое веб-приложение Google Places API.

Получение ключа API

Ключ API можно получить, посетив сайт по адресу: **https://developers.google.com/places/webservice/get-api-key**. Перейдите там в раздел **Get an API key** (Получение ключа API) и щелкните на кнопке **GET A KEY** (Получить ключ) (рис. 10.5).

По нажатию на эту кнопку вы получаете доступ к странице, где можно выбрать уже готовый проект или же создать новый проект для интеграции с веб-службой Google Places API. Щелкните там на раскрывающемся списке и выберите параметр **Create a new project** (Создать новый проект). Вас попросят указать название проекта. Введите в качестве названия проекта: `Place Reviewer` (рис. 10.6).

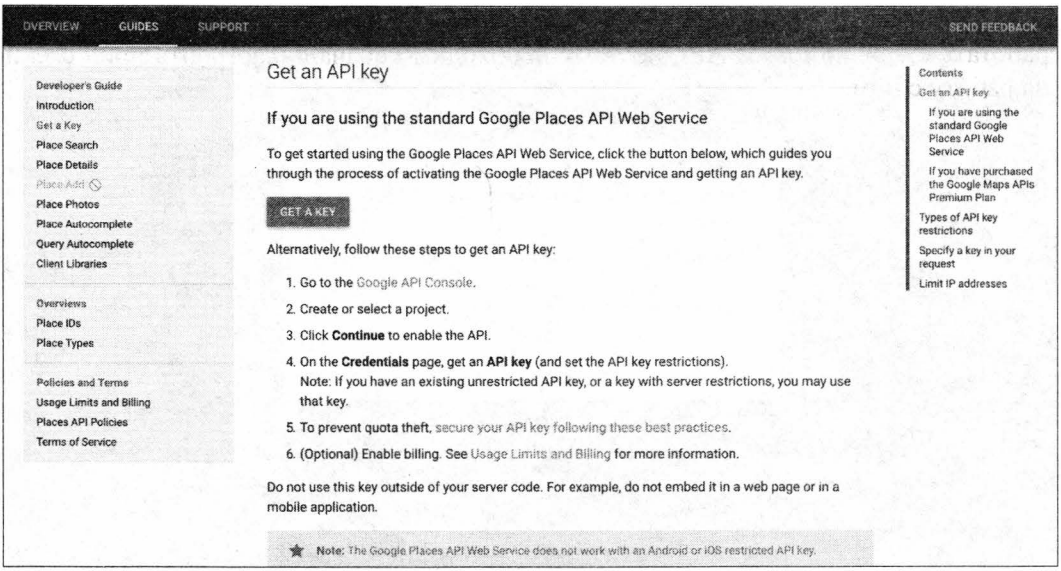


Рис. 10.5. Раздел Get an API key веб-сервиса Google

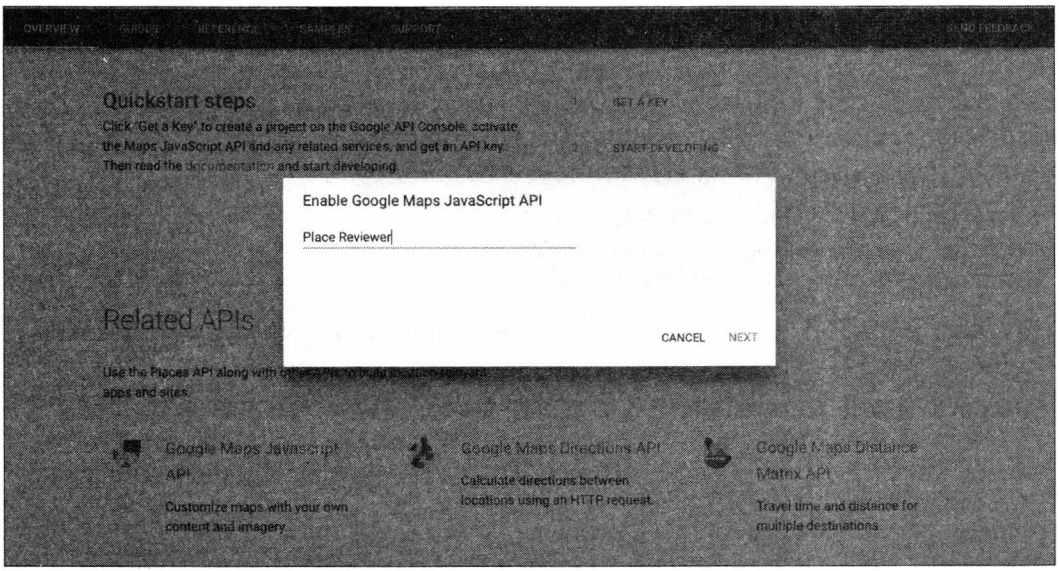


Рис. 10.6. Введите в качестве названия проекта: `Place Reviewer`

После задания имени проекта щелкните на кнопке **NEXT** (Далее) — ваш проект будет настроен для использования с API, и вы получите ключ API (рис. 10.7)!

Теперь, получив ключ API, рассмотрим, как его использовать.

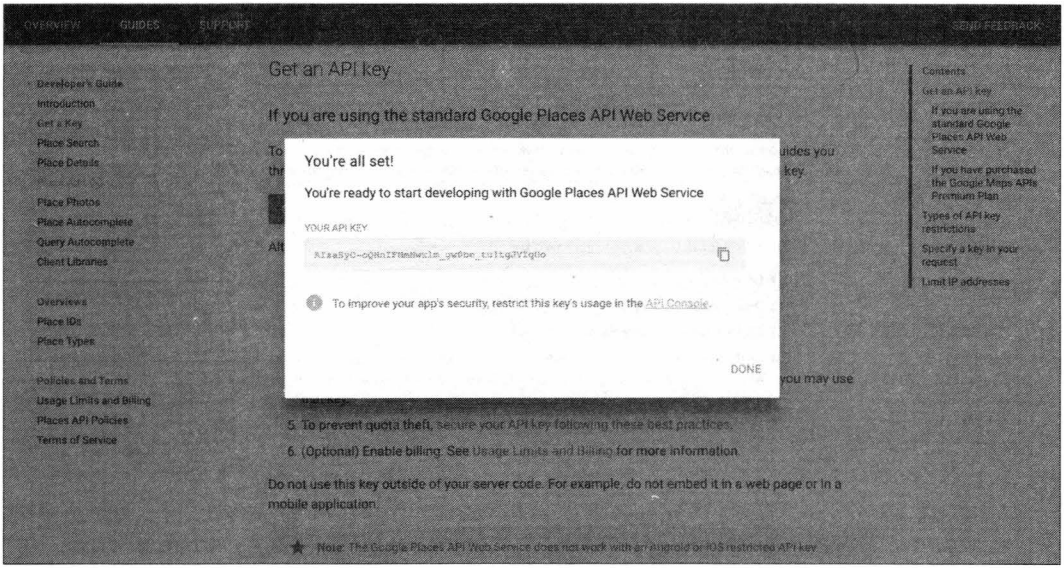


Рис. 10.7. Ключ API получен

Включение Google Places API в веб-приложение

Использовать ключ API для веб-службы Google Places API так же просто, как и его сгенерировать. Для включения в веб-приложение сгенерированного ключа API необходимо вставить в разметку страницы, которая будет использоваться веб-службой, следующую строку HTML:

```
<script type="text/javascript"
src="https://maps.googleapis.com/maps/api/js?key=
   {{API_KEY}}&libraries=places"></script>
```

Убедитесь в том, что вместо `{{API_KEY}}` вы вставили сгенерированный ключ API.

Реализация домашнего представления

Как и ожидалось, на этом этапе нам придется написать много кода. Но в соответствии с практикой, известной из предыдущих глав, следует сначала создать простой графический дизайн макета представления, а уже затем приступать к его программированию. Подобный подход в долгосрочной перспективе экономит значительное количество времени, указывая четкое направление относительно того, чего необходимо достичь.

Итак, нам необходимо, чтобы результирующая домашняя страница выполняла следующее:

1. Отображала последние обзоры, отправленные на платформу.
2. Обеспечивала непосредственный доступ к веб-странице для создания обзора.

3. Поддерживала средства, с помощью которых пользователь может выйти из своей учетной записи.
4. Позволяла пользователю просматривать точное местоположение проверенного места с помощью карты.

Учитывая эти требования, можно сформировать черновой набросок финального шаблона, который будет выглядеть примерно так, как показано на рис. 10.8.

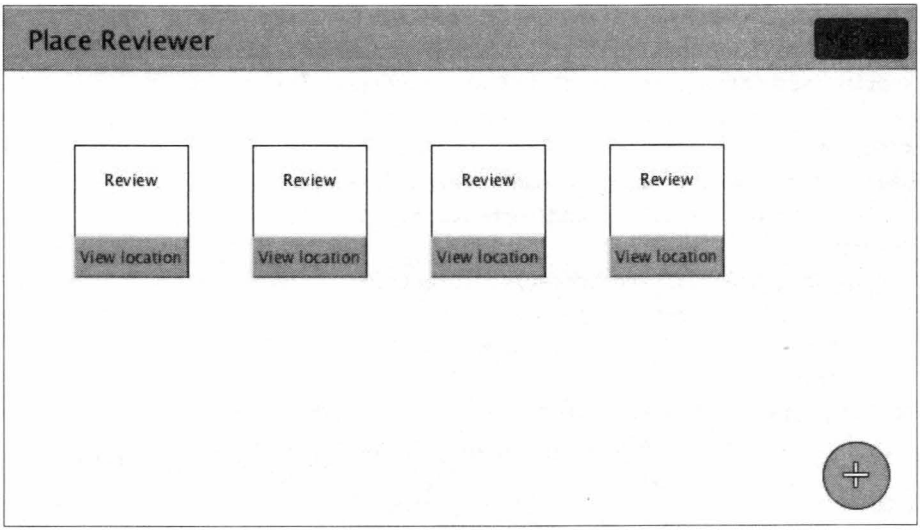


Рис. 10.8. Черновой набросок финального шаблона

Создадим на основе этого наброска функциональный макет. Как можно заметить, в нашем наброске удовлетворяются требования к макету с первого по четвертый пункт. По щелчку на кнопке **View location** (Просмотреть местоположение) пользователь получит доступ к *модулю* — странице, где будет отображена карта, показывающая точное место, которое вы просмотрели.

Получив ясное представление о создаваемом шаблоне, перейдем непосредственно к написанию кода. В первую очередь в шаблон необходимо включить внешние таблицы стилей и сценарии. Откройте файл home.html и добавьте в него следующий код:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
   <head>
      <title>Home</title>
      <!-- Доабвление внешних таблиц стилей -->
      <link rel="stylesheet" th:href="@{/css/app.css}"/>
      <link rel="stylesheet"
         href="https://cdnjs.cloudflare.com/ajax/libs/toastr.js
               /latest/toastr.min.css">
```

```
    <link rel="stylesheet"
    href="/webjars/bootstrap/4.0.0-beta.3/css/bootstrap.min.css"/>
    <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/font-awesome
          /4.7.0/css/font-awesome.min.css"/>
    <link href="https://fastcdn.org/Buttons/2.0.0/css/buttons.css"
    rel="stylesheet">

    <!-- Включение внешнего кода Javascript -->
    <script src="/webjars/jquery/3.2.1/jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs
               /toastr.js/latest/toastr.min.js">
    </script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs
           /popper.js/1.12.6/umd/popper.min.js">
    </script>
    <script src="/webjars/bootstrap/4.0.0-beta.3/
            js/bootstrap.min.js"></script>
    <script src="https://fastcdn.org/Buttons/2.0.0/js/buttons.js">
       </script>
    <script type="text/javascript"
    src="https://maps.googleapis.com/maps/api/js?key={{API_KEY}}
             &libraries=places">
  </head>
</html>
```

Отлично! В дополнение к внешним таблицам стилей в этом шаблоне мы собираемся использовать внутренние стили. Для определения в файлах HTML внутренних таблиц стилей включите в заголовок HTML тег `<style>` и введите необходимые CSS-правила. Для этого добавьте в файл home.html следующий стиль:

```
</script>

  <!-- Определение внешних стилей -->
   <style>
      #map {
         height: 400px;
   }

   .container-review {
      background-color: white;
      border-radius: 2px;
      font-family:sans-serif;
      box-shadow: 0 0 1px black;
      border-color: black;
      padding: 0;
```

```
      min-width: 250px;
      height: 230px;
   }

   .review-author {
      font-size: 15px
    }

   .review-location {
      font-size: 12px
   }

   .review-title {
      font-size: 13px;
      text-decoration-style: dotted;
      height: calc(20 / 100 * 230px);
   }

   .review-content {
      font-size: 12px;
      height: calc(40 / 100 * 230px);
   }

   .review-header {
      height: calc(20 / 100 * 230px)
   }

   hr {
      margin: 0;
   }

   .review-footer {
      height: calc(20 / 100 * 230px);
   }
</style>
```

Теперь поработаем над основным разделом (телом) страницы. Как известно, все элементы, входящие в тело HTML-шаблона, должны находиться в теге <body>. Учитывая это, соответственно доработаем файл home.html. И начнем с добавления в файл шаблона следующего кода HTML:

```
<!-- Вызов функции showNoReviewNotification(), определенной -->
<!-- во внутреннем коде Javascript для этого файла до загрузки документа. -->
<body
   th:onload="'javascript:showNoReviewNotification(
      ' + ${reviews.size() == 0} + '
   )'">
```

```
<div th:insert="fragments/navbar :: navbar"></div>
<div class="container">
   <div class="row mt-5">
      <!-- Создание контейнеров представления для каждого
         выбранного обзора -->
      <!-- Созданы разные контейнеры <div> для автора обзора, -->
      <!-- местоположения, заголовка и тела. -->
      <div th:each="review: ${reviews}"
      class="col-sm-2 container-review mt-4 mr-2">
         <div class="review-header pt-1">
            <div class="col-sm-12 review-author text-success">
               <b th:text="${review.reviewer.username}"></b>
            </div>
            <div th:text="${review.placeName}"
               class="col-sm-12 review-location">
             </div>
             </div>
             <hr>
             <b>
             <div th:text="${review.title}"
                class="col-sm-12 review-title pt-1">
             </div>
             </b>
             <hr>
             <div th:text="${review.body}"
                class="col-sm-12 review-content pt-2">
             </div>
                <div class="review-footer">
                <!-- Создание разных элементов DOM
                <button> для отображения просмотренных
                   местоположений. -->
                <!-- До щелчка на кнопке приложение отображает содержание модуля,
                   показывающего просмотренное местоположение на карте -->
                <button class="col-sm-12 button button-small
                   button-primary"
                 type="button" data-toggle="modal"
                    data-target="#mapModal"
                 style="height: inherit; border-radius: 2px;"
                 th:onclick="'javascript:showLocation(
                       ' + ${review.latitude} + ','
                       + ${review.longitude} + ',\''
                       + ${review.placeId} + '\'
                 )'">
                 <i class="fa fa-map-o" aria-hidden="true"></i>
                 View location
```

```
            </button>
          </div>
        </div>
    </div>
</div>
```

Прекрасная работа! И не беспокойтесь пока о том, что приведенный блок кода сейчас делает, — все объяснится в свое время. Идем дальше и продолжаем формировать тело файла home.html, внося в него следующие строки кода:

```
<!-- Создание модуля -->
<div class="modal fade" id="mapModal">
   <div class="modal-dialog modal-lg" role="document">
   <div class="modal-content">
      <div class="modal-header">
      <h5 class="modal-title">Reviewed location</h5>
      <button type="button" class="close"
      data-dismiss="modal" aria-label="Close">
         <span aria-hidden="true">&times;</span>
      </button>
   </div>
   <div class="modal-body">
      <div class="container-fluid">
         <div id="map"> </div>
       </div>
   </div>
   <div class="modal-footer">
      <button type="button" class="btn btn-primary"
      data-dismiss="modal">
         Done
      </button>
   </div>
</div>
</div>
</div>
```

В приведенном коде создается модуль, демонстрирующий карту с точным местоположением, по которому по запросу пользователя создан обзор. Но еще не закончен домашний шаблон, так что продолжим работу над разделом <body>, добавляя в него следующие строки кода:

```
<span style="bottom: 20px; right: 20px; position: fixed">
   <form method="get" th:action="@{/create-review}">
      <button class="button button-primary button-circle
       button-giant navbar-bottom" type="submit">
         <i class="fa fa-plus"></i>
       </button>
```

```
   </form>
</span>
```

Наконец, добавьте на HTML-страницу внутренний код JavaScript:

```
<script>
   // Отображение toast-извещения пользователю, если отсутствует обзор
   function showNoReviewNotification(show) {
      if (show) {
         toastr.info('No reviews to see');
      }
   }
```

Следующая функция инициализирует и отображает карту с указанием местоположения, про которое написан обзор:

```
function showLocation(latitude, longitude, placeId) {
   var center = new google.maps.LatLng(latitude, longitude);

   var map = new google.maps.Map(document.getElementById('map'), {
      center: center,
      zoom: 15,
      scrollwheel: false
   });
   var service = new google.maps.places.PlacesService(map);
      loadPlaceMarker(service, map, placeId);
}
```

Загруженный маркер местоположения создает маркер на карте в выбранном для обзора местоположении:

```
function loadPlaceMarker(service, map, placeId) {
   var request = {
      placeId: placeId
   };

   service.getDetails(request, function (place, status) {
      if (status === google.maps.places.PlacesServiceStatus.OK) {
         new google.maps.Marker({
            map: map,
            title: place.name,
          place: {
             placeId: place.place_id,
             location: place.geometry.location
          }
       })
   }
});
}
```

```
</script>
</body>
```

К этому моменту мы внесли в файл home.html много дополнений. Рассмотрим теперь, что же происходит в представлении, начиная с тега <head>. Таблицы стилей и сценарии, необходимые для домашней страницы, вставлены в строках от 4 до 16 тега <head>. Включенный CSS-код имеет следующий вид:

```
<link rel="stylesheet" th:href="@{/css/app.css}"/>
<link rel="stylesheet"
     href="https://cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/toastr.min.css">
<link rel="stylesheet" href="/webjars/bootstrap/4.0.0-beta.3/
                                                    css/bootstrap.min.css"/>
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
                                        font-awesome/4.7.0/css/font-awesome.min.css"/>
<link href="https://fastcdn.org/Buttons/2.0.0/css/buttons.css"
                                                    rel="stylesheet">
```

Напомню, что это внешние таблицы стилей CSS для нашего приложения. Здесь toastr — библиотека для создания всплывающих уведомлений (тост-уведомлений) JavaScript; bootstrap — универсальная библиотека для разработки веб-сайтов и веб-приложений; font-awesome — набор значков для веб-сайтов и веб-приложений; а Buttons — универсальная и настраиваемая библиотека веб-кнопок и CSS.

Сразу после CSS-включений следует ряд внешних включений JavaScript:

```
<script src="/webjars/jquery/3.2.1/jquery.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/toastr.js/latest/
                                                    toastr.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.6/umd/
                                                    popper.min.js"></script>
<script src="/webjars/bootstrap/4.0.0-beta.3/js/bootstrap.min.js"></script>
<script src="https://fastcdn.org/Buttons/2.0.0/js/buttons.js"></script>
<script type="text/javascript" src="https://maps.googleapis.com/maps/api/js?key=
                                        {{API_KEY}}&libraries=places"></script>
```

Включения сценариев в приведенном порядке предназначены для: JQuery — библиотеки JavaScript, специально разработанной для упрощения процесса написания сценариев HTML на стороне клиента; библиотеки Toastr; Popper —библиотеки, используемой для управления попперами (всплывающими подсказками) в веб-приложениях; библиотеки Bootstrap; кнопок и веб-службы Google Places API. Не забудьте также, что во фрагменте {{API_KEY}} следует заменить значение API_KEY на свой ключ API для веб-службы Google Places API.

Сразу после включения кода JavaScript в файле home.html определена внутренняя таблица стилей для веб-страницы. К сожалению, объяснение таблиц стилей и процессов их создания выходит за рамки этой книги. Тем не менее, рекомендую вам освежить знания по CSS в свободное время.

Далее в файл home.html добавлен тег <body> следующим образом:

```
<body th:onload="'javascript:showNoReviewNotification(
   ' + ${reviews.size()
== 0} + ')'">
```

Конструкция th:onload используется для указания кода JavaScript, который должен быть запущен после полной загрузки страницы. Короче говоря, она указывает на код, который должен быть выполнен после события onload. В этом случае сценарий, который необходимо запустить, является функцией JavaScript, которая определена далее по шаблону showNoReviewNotification(boolean), — она отображает всплывающее сообщение о том, что отсутствуют обзоры, доступные для просмотра, если пуст список предоставленных моделью обзоров. Функция эта объявляется в шаблоне следующим образом:

```
function showNoReviewNotification(show) {
   if (show) {
      toastr.info('No reviews to see');
   }
}
```

Функция showNoReviewNotification(boolean) содержит единственный Boolean-аргумент — show. Если show имеет значение true, пользователь получает тост-уведомление с сообщением: **No reviews to see** (Нет обзоров, которые можно показать). Отображение всплывающих тост-уведомлений для пользователя возможно благодаря использованию библиотеки Toastr.

Если доступные для просмотра пользователем обзоры имеются, для каждого элемента обзора создается контейнер следующим образом:

```
<!-- Создает контейнеры представления для каждого выбранного обзора -->
<!-- Создаются разные контейнеры <div> для автора обзора, -->
<!-- местоположения, заглавия и тела. -->
<div th:each="review: ${reviews}" class="col-sm-2 container-review mt-4 mr-2">
   <div class="review-header pt-1">
      <div class="col-sm-12 review-author text-success">
         <b th:text="${review.reviewer.username}"></b>
      </div>
      <div th:text="${review.placeName}" class="col-sm-12 review-location">
      </div>
   </div>
   <hr>
   <b>
      <div th:text="${review.title}" class="col-sm-12 review-title pt-1">
      </div>
   </b>
   <hr>
   <div th:text="${review.body}" class="col-sm-12 review-content pt-2">
   </div>
```

```
   <div class="review-footer">
      <!-- Создание разных элементов DOM <button> для отображения
      просмотренных местоположений. -->
      <!-- До щелчка на кнопке приложение выводит модуль,
      отображающий просмотренное местоположение на карте -->
      <button class="col-sm-12 button button-small button-primary"
      type="button" data-toggle="modal" data-target="#mapModal"
      style="height: inherit; border-radius: 2px;"
      th:onclick="'javascript:showLocation('
         + ${review.latitude} + ','
         + ${review.longitude} + ',\''
         + ${review.placeId} + '\
      ')'">
      <i class="fa fa-map-o" aria-hidden="true"></i>
      View location
      </button>
   </div>
</div>
```

В каждом контейнере обзора отображается имя пользователя обозревателя, название обозреваемого места, название обзора, тело обзора и кнопка, позволяющая пользователю просматривать обозреваемое местоположение. Атрибут `th:each` из Thymeleaf применяется для перебора каждого обзора в списке обзоров:

```
<div th:each="review: ${reviews}" class="col-sm-2 container-review mt-4 mr-2">
```

Хороший способ понять итерационный процесс — воспринять `th:each="review: ${reviews}"` как `For each review in reviews`. Представленный в настоящее время обзор хранится в переменной `review`. Следовательно, к данным, содержащимся в повторяемом обзоре, можно получить доступ, как и к любому другому объекту. Подобный случай представлен далее:

```
<div th:text="${review.placeName}" class="col-sm-12 review-location"></div>
```

С помощью атрибута `th:text` устанавливается текст, который вызывается в теге `<div>` с помощью значения, присвоенного `review.placeName`.

Необходимо также объяснить процесс, с помощью которого карты местоположения показываются пользователю. Внимательно посмотрите на следующие строки кода:

```
<button class="col-sm-12 button button-small button-primary" type="button"
data-toggle="modal" data-target="#mapModal" style="height: inherit;
border-radius: 2px;"
th:onclick="'javascript:showLocation('
   + ${review.latitude} + ','
   + ${review.longitude} + ',\''
   + ${review.placeId} + '\
')'">
```

```
<i class="fa fa-map-o" aria-hidden="true"></i>
View location
</button>
```

В этом блоке кода определяется кнопка, которая выполняет две функции, если происходит событие щелчка. Во-первых, пользователю отображается модуль, идентифицируемый по ID: `mapModal`. Во-вторых, инициализируется и отображается карта, демонстрирующая точное просмотренное местоположение. Визуализация карты возможна благодаря функции JavaScript `showLocation()`, определенной в файле шаблона.

Функция `showLocation()` использует три параметра в качестве аргументов. Первый параметр — это координата долготы, второй — координата широты, а третий — уникальный идентификатор рассмотренного местоположения. Идентификатор местоположения предоставляется Google Places API. Сначала `showLocation()` извлекает центральную точку местоположения, соответствующего предоставленным координатам. Это выполняется с помощью класса `google.maps.LatLng` Google Places API. Попросту говоря, `LatLng` представляет собой точку в географических координатах (с указанием долготы и широты). После извлечения центральной точки создается новая карта с использованием класса `Map` (снова с помощью Google Places API), как показано в следующем фрагменте кода:

```
var map = new google.maps.Map(document.getElementById('map'), {
   center: center,
   zoom: 15,
   scrollwheel: false
});
```

Полученная карта помещается в элемент контейнера DOM с картой ID (идентификаторов). После создания необходимой карты формируется маркер местоположения в точном месте с помощью функции `loadPlaceMarker()`, которая использует экземпляры `google.maps.places.PlacesService`, `Map` и идентификатор местоположения в качестве трех аргументов.

Класс `PlacesService` располагает методами для поиска информации о местоположении и поиска местоположений.

Экземпляры `google.maps.places.PlacesService` первоначально применяются для получения подробностей о местоположении с указанным ID местоположения (местоположения обзора). Если подробная информация о местоположении успешно получена, `status === google.maps.places.PlacesServiceStatus.OK` устанавливается как `true`, и маркер местоположения размещается на карте. Сам маркер формируется с помощью класса `google.maps.Marker`. Метод `Marker()` принимает необязательный объект параметров в качестве единственного аргумента. Если объект параметров присутствует, маркер места создается с указанными параметрами. В нашем случае в объекте параметров указана карта. Таким образом, маркер устанавливается на карту при ее создании.

Наконец, в шаблон включена форма, направляющая запрос GET по пути **/reviews/new** после его отправки, и добавляется кнопка, отправляющая форму по щелчку:

```
<form method="get" th:action="@{/reviews/new}">
   <button class="button button-primary button-circle button-giant
   navbar-bottom" type="submit"><i class="fa fa-plus"></i></button>
</form>
```

Это всё, что необходимо выполнить для создания домашней страницы, поэтому произведите ее проверку. Перестройте и запустите приложение, зарегистрируйте учетную запись и просмотрите только что сформированную домашнюю страницу (рис. 10.9).

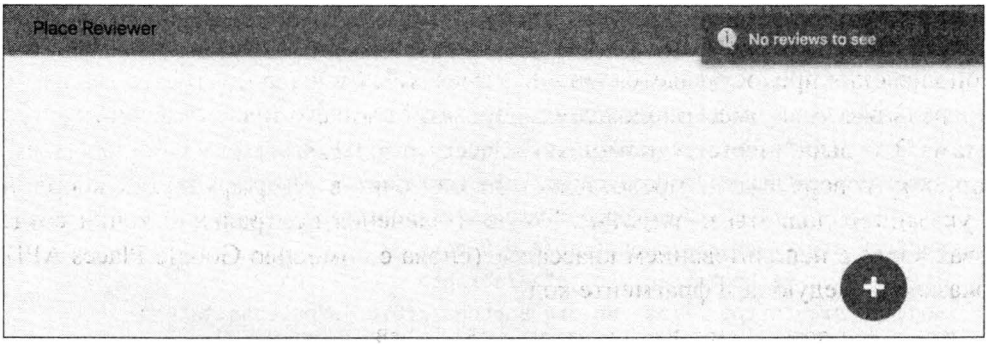


Рис. 10.9. Созданная домашняя страница

Заметьте, на платформе еще нет ни одного обзора для просмотра. Теперь необходимо создать веб-страницу, которая позволит формировать обзоры.

Реализация веб-страницы создания обзора

К настоящему времени созданы представления для регистрации пользователей и входа в систему, а также домашняя страница для зарегистрированных пользователей, позволяющая просматривать размещенные на платформе обзоры. Теперь необходимо поработать над представлением, которое облегчает создание подобных обзоров. Как всегда, прежде чем создавать представление, обдумаем действие, которое отвечает за отображение разрабатываемого для пользователя представления. Откройте класс ApplicationController:

```
@GetMapping("/create-review")
fun createReview(model: Model, principal: Principal): String {
   model.addAttribute("principal", principal)
   return "create-review"
}
```

Действие createReview() обрабатывает HTTP-запросы GET с учетом пути запроса **/create-review**, возвращая клиенту для отображения шаблон create-review.html. Добавим файл create-review.html в каталог template проекта.

Подобно тому как мы делали это ранее, добавим в файл create-review.html внешние стили и сценарии:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
   <head>
      <title>New review</title>
      <!-- Добавление внешних таблиц стилей -->
      <link rel="stylesheet" th:href="@{/css/app.css}"/>
      <link rel="stylesheet" href="/webjars/bootstrap/4.0.0-beta.3
                                    /css/bootstrap.min.css"/>
      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com
                  /font-awesome/4.7.0/css/font-awesome.min.css"/>
      <link href="https://fastcdn.org/Buttons/2.0.0/css/buttons.css"
                  rel="stylesheet">
      <!-- Включение внешего кода Javascript -->
      <script src="/webjars/jquery/3.2.1/jquery.min.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js
                  /1.12.6/umd/popper.min.js"></script>
      <script src="/webjars/bootstrap/4.0.0-beta.3/
                js/bootstrap.min.js"></script>
      <script src="https://fastcdn.org/Buttons/2.0.0/js/buttons.js">
           </script>
      <script type="text/javascript" src="https://maps.googleapis.com/
      maps/api/js?key={{API_KEY}}&libraries=places">
      </script>
```

Затем добавим необходимую внутреннюю таблицу стилей для веб-страницы:

```
<!-- Определение внутренней таблицы стилей -->
<style>
   #map {
      height: 400px;
   }

   #container-place-data {
      height: 0;
   visibility: hidden;
   }

   #container-place-info {
      font-size: 14px;
   }

   #container-selection-status {
      visibility: hidden;
   }
```

```
   </style>
</head>
```

Следующий вопрос нашей повестки дня относится к формированию необходимой формы для ввода данных обзора. Доработаем шаблон create-review.html с помощью следующего кода:

```
<body>
   <div th:insert="fragments/navbar :: navbar"> </div>
   <div class="container-fluid">
      <div class="row">
         <div class="col-sm-12 col-xs-12">
         <!-- Создание формы обзора -->
         <form class="form-group col-sm-12 form-vertical form-app"
            id="form-login" method="post" th:action="@{/reviews}">
               <div class="col-sm-12 mt-2 lead">Write your review</div>
               <div th:if="${error != null}" class="text-danger"
               th:text="${error}"> </div>
               <hr>
               <input class="form-control" type="text" name="title"
               placeholder="Title" th:value="${title}" required/>
               <textarea class="form-control mt-4" rows="13" name="body"
               placeholder="Review" th:value="${body}"
                  required></textarea>
               <div class="form-group" id="container-place-data">
                  <!-- Поля ввода для данных формы специфического
                     местоположения -->
                  <!-- Данные ввода формы для полей ниже,
                  поддерживаемых Google Places API -->
                  <input class="form-control" id="place_address"
                  th:value="${placeAddress}" type="text"
                     name="placeAddress" required/>
                  <input class="form-control" id="place_name" type="text"
                  name="placeName" th:value="${placeName}" required/>
                  <input class="form-control" id="place_id" type="text"
                  name="placeId" th:value="${placeId}" required/>
                  <input id="location-lat" type="number" name="latitude"
                  step="any" th:value="${latitude}" required/>
                  <input id="location-lng" type="number" name="longitude"
                  step="any" th:value="${longitude}" required/>
               </div>
               <div class="form-group mb-3">
               <button class="button button-pill" type="button"
               data-toggle="modal" data-target="#mapModal">
               <i class="fa fa-map-marker" aria-hidden="true"></i>
               Select Location
            </button>
```

```
            <button class="button button-pill button-primary">
               Submit Review</button>
         </div>
         <div class="text-success ml-2" id="container-selection-status">
         Location selected</div>
         </form>
   </div>
</div>
```

Теперь добавим модуль, разрешающий пользователю выбирать местоположение обзора на карте (не следует пока вникать в детали процесса выбора, мы рассмотрим его далее):

```
<!-- Модуль карты -->
<div class="modal fade" id="mapModal">
   <div class="modal-dialog modal-lg" role="document">
      <div class="modal-content">
         <div class="modal-header">
            <h5 class="modal-title">Select place to review</h5>
            <button type="button" class="close" data-dismiss="modal"
            aria-label="Close">
               <span aria-hidden="true">&times;</span>
            </button>
         </div>
         <div class="modal-body">
         <div class="container-fluid">
          <div id="map"> </div>
          <div class="row mt-2" id="container-place-info">
          <div class="col-sm-12" id="container-place-name">
          <b>Place Name:</b>
       </div>
       <div class="col-sm-12" id="container-place-address">
       <b>Place Address:</b>
       </div>
   </div>
</div>
</div>
<div class="modal-footer">
<button type="button" class="btn btn-primary"
data-dismiss="modal">Done</button>
</div>
</div>
</div>
</div>
</div>
```

И наконец, завершим шаблон, включив в него внутренний код JavaScript, как показано в приведенном далее фрагменте кода:

```
<script>
   // Создание ссылки на поле формы
   var formattedAddressField = document
                        .getElementById('place_address');
   var placeNameField = document.getElementById('place_name');
   var placeIdField = document.getElementById('place_id');
   var latitudeField = document.getElementById('location-lat');
   var longitudeField = document.getElementById('location-lng');

   // Создание ссылки на контейнер
   var containerPlaceName = document.getElementById
                           ('container-place-name');
   var containerPlaceAddress = document.getElementById
                           ('container-place-address');
   var containerSelectionStatus = document.getElementById
                           ('container-selection-status');
```

В приведенном фрагменте кода созданы ссылки на важные элементы DOM, имеющиеся на странице, т. е. ссылки для размещения на ней определенных полей ввода: для адреса, имени, идентификатора, а также для координат широты и долготы определенного местоположения. Кроме того, добавлены ссылки на контейнеры, которые отображают сведения о выбранном местоположении, а именно: название и адрес места. На этом этапе объявим несколько функций: `initialize()`, `getPlaceDetailsById()`, `updateViewData()`, `setFormValues()`, `showSelectionsStatusContainer()` и `setContainerText()`.

Начните с добавления функций `initialize()` и `getPlaceDetailsById()`:

```
// Вызвано для инициализации карты Google
function initialize() {

   navigator.geolocation.getCurrentPosition(function(location) {

      var latitude = location.coords.latitude;
      var longitude = location.coords.longitude;

      var center = new google.maps.LatLng(latitude, longitude);

      var map = new google.maps.Map(document.getElementById('map'), {
         center: center,
         zoom: 15,
         scrollwheel: false
      });
```

```
    var service = new google.maps.places.PlacesService(map);

    map.addListener('click', function(data) {
      getPlaceDetailsById(service, data.placeId);
    });
  });
}
```

Мы добавили приведенную далее функцию для получения информации о конкретном месте из вызванной Google Places API, к которому обратились для получения сведений о местоположении:

```
function getPlaceDetailsById(service, placeId) {

  var request = {
    placeId: placeId
  };

  service.getDetails(request, function (place, status) {
    if (status === google.maps.places.PlacesServiceStatus.OK) {
      updateViewData(place)
    }
  });
}
```

Теперь добавим функции `updateViewData()` и `setFormValues()`:

```
// Вызвано для обновления информации о представлении
function updateViewData(place) {
  setFormValues(
    place.formatted_address,
    place.name,
    place.place_id,
    place.geometry.location.lat(),
    place.geometry.location.lng()
  );

  setContainerText('<b>Place Name: </b>' + place.name,
    '<b>Place Address: </b>' + place.formatted_address);

  showSelectionStatusContainer();
}
```

Приведенная далее функция вызывается для обновления данных формы представления:

```
function setFormValues(formattedAddress, placeName, placeId,  latitude, longitude) {

  formattedAddressField.value = formattedAddress;
  placeNameField.value = placeName;
  placeIdField.value = placeId;
```

```
   latitudeField.value = latitude;
   longitudeField.value = longitude;
}
```

Наконец, завершите шаблон, добавив показанный здесь код:

```
      function showSelectionStatusContainer() {
         containerSelectionStatus.style.visibility = 'visible'
      }

      function setContainerText(placeNameText, placeAddressText) {
         containerPlaceName.innerHTML = placeNameText;
         containerPlaceAddress.innerHTML = placeAddressText;
      }

      // Инициализация карты при завершении загрузки окна
      google.maps.event.addDomListener(window, 'load', initialize);
            </script>
   </body>
</html>
```

Как и в предыдущем шаблоне, мы начинаем создание файла create-review.html с добавления в HTML-тег `<head>` внешнего и внутреннего кода CSS и JavaScript, которые необходимы для шаблона. Затем в шаблоне создаем форму, принимающую в качестве входных данных следующие данные:

- `title` — пользовательский заголовок создаваемого обзора;
- `body` — тело обзора (основной текст обзора);
- `placeAddress` — адрес проверяемого места;
- `placeName` — название места, которое проверяется;
- `placeId` — уникальный ID проверяемого местоположения;
- `latitude` — координата широты для рассматриваемого местоположения;
- `longitude` — координата долготы для рассматриваемого местоположения.

Пользователю не нужно вводить данные формы для `placeAddress`, `placeName`, `placeId`, `latitude` и `longitude`, поэтому родительский элемент `<div>` сокрыт для упомянутых элементов ввода. Для получения информации о местоположениях применяется Google Places API. Обратите внимание, что в шаблоне применен модуль (modal) для отображения карты при выборе местоположения. Модуль переключается кнопкой, которая добавлена в шаблон следующим образом:

```
<button class="button button-pill" type="button"
   data-toggle="modal" datatarget="#
mapModal">
   <i class="fa fa-map-marker" aria-hidden="true"></i> Select Location
</button>
```

По щелчку на кнопке для пользователя отобразится модульная карта. После отображения карты пользователь может щелкнуть на желаемом местоположении для просмотра его на карте. Выполнение щелчка вызовет событие щелчка карты, которое, в свою очередь, обрабатывается слушателем, определенным в шаблоне следующим образом:

```
map.addListener('click', function(data) {
   getPlaceDetailsById(service, data.placeId);
});
```

Функция `getPlacesDetailsById()` имеет два аргумента: экземпляр `google.maps.places.PlacesService` и ID местоположения, информация о котором отображается. Затем используется экземпляр класса `PlacesService` для получения информации о месте. После завершения поиска представление обновляется с помощью полученной информации: устанавливаются данные формы для конкретного местоположения, обновляется название местоположения и контейнер адресов в модульной карте, пользователю отображается сообщение, указывающее, что местоположение выбрано успешно. После выбора местоположения и ввода необходимых данных формы пользователь может представить свой отзыв.

Итак, уже все готово для опробования страницы по созданию обзора. Но прежде чем это сделать, необходимо создать валидатор проверки, а также действие контроллера, обрабатывающее запросы `POST`, направленные по пути **/reviews**. Начнем с `ReviewValidator` и добавим показанный здесь класс `ReviewValidator` в com.example.placereviewer.component:

```
package com.example.placereviewer.component

import com.example.placereviewer.data.model.Review
import org.springframework.stereotype.Component
import org.springframework.validation.Errors
import org.springframework.validation.ValidationUtils
import org.springframework.validation.Validator

@Component
class ReviewValidator: Validator {

   override fun supports(aClass: Class<*>?): Boolean {
      return Review::class == aClass
   }

   override fun validate(obj: Any?, errors: Errors) {
      val review = obj as Review
      ValidationUtils.rejectIfEmptyOrWhitespace(errors, "title",
         "Empty.reviewForm.title", "Title cannot be empty")
      ValidationUtils.rejectIfEmptyOrWhitespace(errors, "body",
         "Empty.reviewForm.body", "Body cannot be empty")
```

```
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "placeName",
           "Empty.reviewForm.placeName")
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "placeAddress",
           "Empty.reviewForm.placeAddress")
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "placeId",
          "Empty.reviewForm.placeId")
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "latitude",
          "Empty.reviewForm.latitude")
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "longitude",
          "Empty.reviewForm.longitude")

        if (review.title.length < 5) {
           errors.rejectValue("title", "Length.reviewForm.title",
              "Title must be at least 5 characters long")
        }

        if (review.body.length < 5) {
           errors.rejectValue("body", "Length.reviewForm.body",
              "Body must be at least 5 characters long")
        }
    }
}
```

Поскольку функционирование пользовательских валидаторов рассматривалось ранее, перейдем к реализации класса контроллера для относящихся к обзорам HTTP-запросам. Создайте класс `ReviewController` в com.example.placereviewer.controller и добавьте в него следующий код:

```
package com.example.placereviewer.controller

import com.example.placereviewer.component.ReviewValidator
import com.example.placereviewer.data.model.Review
import com.example.placereviewer.service.ReviewService
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.validation.BindingResult
import org.springframework.web.bind.annotation.ModelAttribute
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestMapping
import javax.servlet.http.HttpServletRequest

@Controller
@RequestMapping("/reviews")
class ReviewController(val reviewValidator: ReviewValidator,
                       val reviewService: ReviewService) {
```

```
    @PostMapping
    fun create(@ModelAttribute reviewForm: Review, bindingResult:
        BindingResult,
    model: Model, request: HttpServletRequest): String {
        reviewValidator.validate(reviewForm, bindingResult)
        if (!bindingResult.hasErrors()) {
            val res = reviewService.createReview(request.userPrincipal.name,
                                                                    reviewForm)
            if (res) {
                return "redirect:/home"
            }
        }
        with (model) {
            addAttribute("error", bindingResult.allErrors.first().defaultMessage)
            addAttribute("title", reviewForm.title)
            addAttribute("body", reviewForm.body)
            addAttribute("placeName", reviewForm.placeName)
            addAttribute("placeAddress", reviewForm.placeAddress)
            addAttribute("placeId", reviewForm.placeId)
            addAttribute("longitude", reviewForm.longitude)
            addAttribute("latitude", reviewForm.latitude)
        }
        return "create-review"
    }
}
```

Добавив классы `ReviewValidator` и `ReviewController`, создайте и запустите проект, войдите в систему как пользователь и перейдите на своем браузере по адресу: **http://localhost:5000/create-review**.

После загрузки страницы вы получите доступ к форме, которую можно использовать для добавления нового обзора (рис. 10.10).

Но прежде чем отправить обзор, пользователи должны выбрать местоположение для просмотра, для чего надо щелкнуть на кнопке **Select Location** (Выбрать местоположение).

По щелчку на кнопке **Select Location** пользователь получит доступ к карте, на которой можно выбрать местоположение для просмотра (рис. 10.11). Щелчок по местоположению на карте приведет к отображению на ней информационного окна, куда входят данные, относящиеся к выбранному местоположению. Кроме того, обновляется модульный контейнер для сохранения выбранного названия местоположения и адреса (рис. 10.12).

Выбрав местоположение для просмотра, щелчком на кнопке **Done** (Готово) модульное окно можно закрыть и приступить к заполнению полей названия и основной части обзора (рис. 10.13).

Обратите внимание: форма обзора отмечает, что местоположение для обзора выбрано успешно. Когда пользователь заполнит всю необходимую для обзора информацию, можно продолжить процесс отправки обзора щелчком на кнопке **Submit Review** (Отправить обзор).

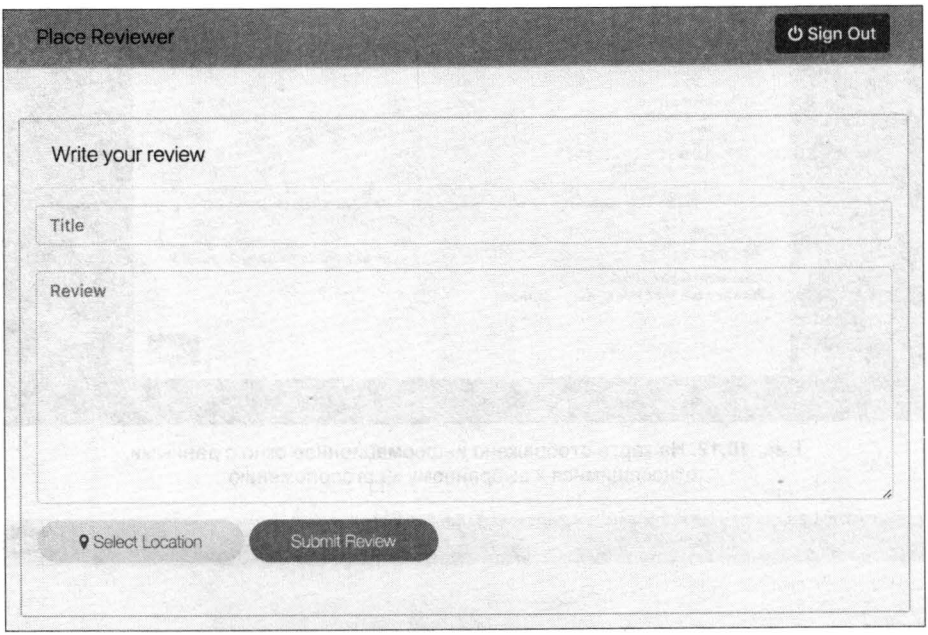


Рис. 10.10. Форма для добавления нового обзора

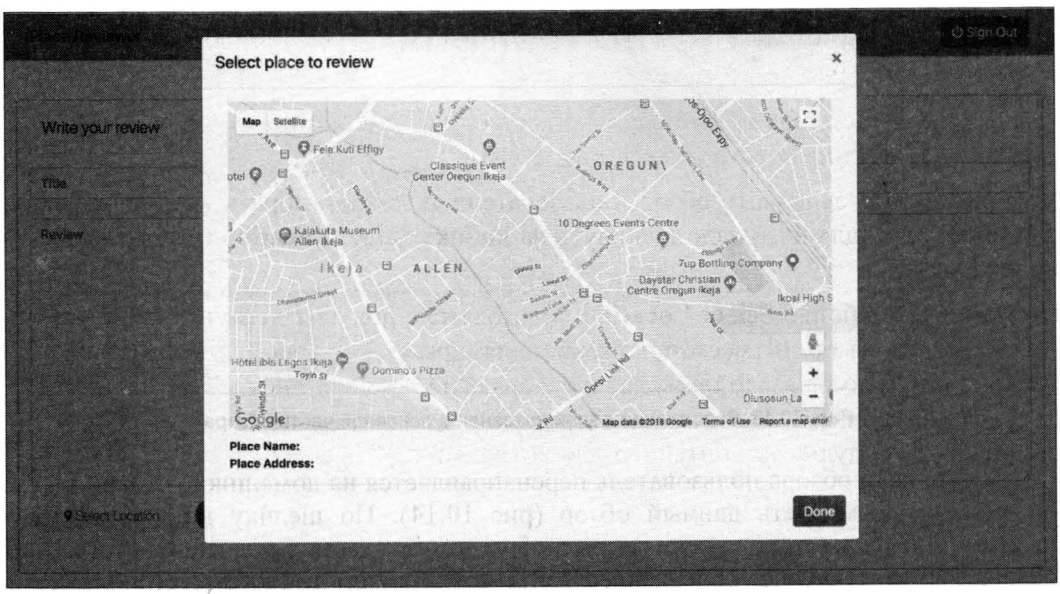


Рис. 10.11. Карта, на которой можно выбрать местоположение для просмотра

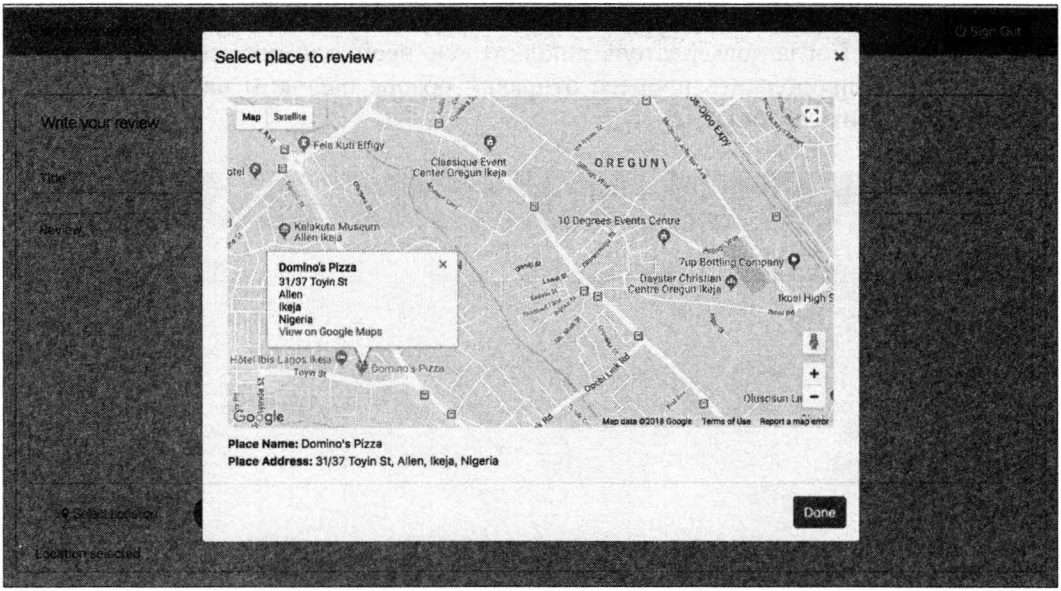


Рис. 10.12. На карте отображено информационное окно с данными, относящимися к выбранному местоположению

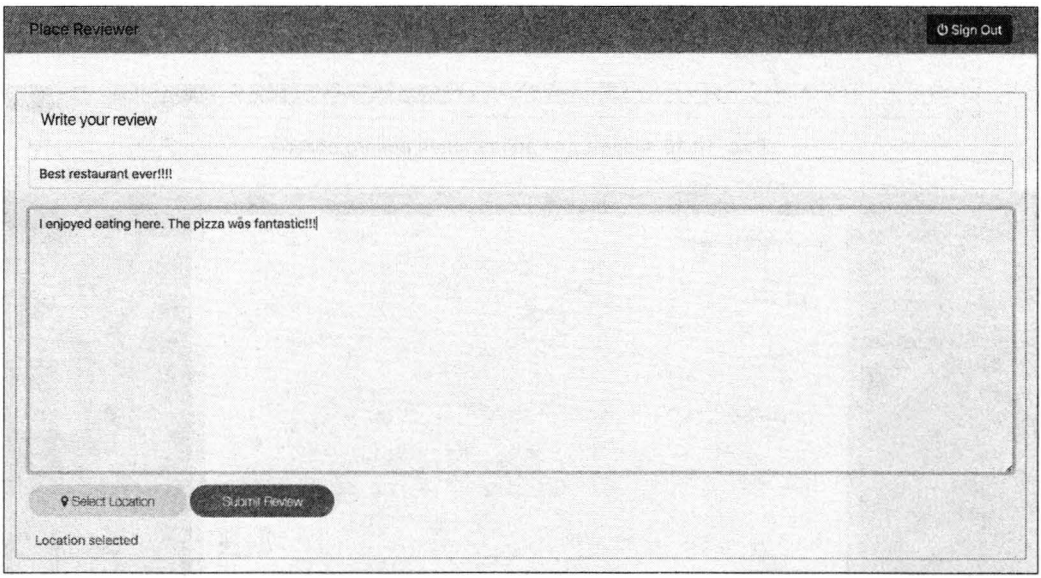


Рис. 10.13. Заполнение полей названия и основной части обзора

После отправки обзора пользователь перенаправляется на домашнюю страницу, где он может просмотреть данный обзор (рис. 10.14). По щелчку на кнопке **View location** (Обзор местоположения) для любого отображаемого на главной странице обзора выводится окно, содержащее карту с указанием точного местоположения обзора (рис. 10.15).

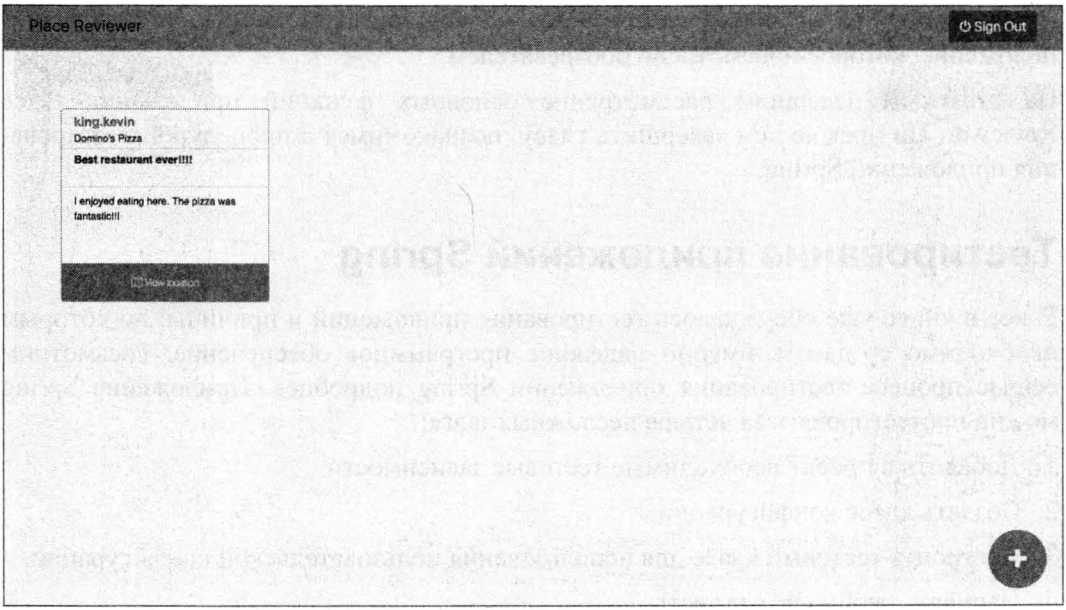


Рис. 10.14. Домашняя страница пользователя, где он может просмотреть отправленный обзор

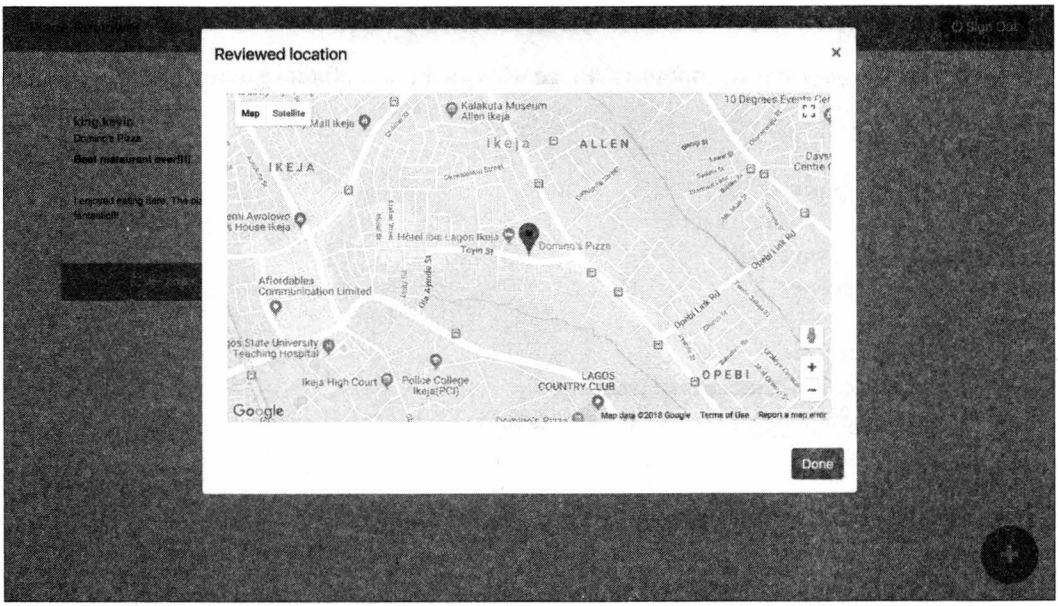


Рис. 10.15. Окно, содержащее карту с указанием точного местоположения обзора

Отображаемая пользователю карта имеет маркер, указывающий на точное местоположение, которое просмотрено обозревателем.

На этом мы завершим рассмотрение основных функций приложения Place Reviewer. Но прежде чем завершить главу, познакомимся с процедурой тестирования приложений Spring.

Тестирование приложений Spring

Ранее в книге уже обсуждалось тестирование приложений и причины, по которым необходимо создавать именно надежное программное обеспечение. Рассмотрим сейчас процесс тестирования приложений Spring подробнее. Приложение Spring можно протестировать за четыре несложных шага:

1. Добавить в проект необходимые тестовые зависимости.
2. Создать класс конфигурации.
3. Настроить тестовый класс для использования пользовательской конфигурации.
4. Написать необходимые тесты.

Рассмотрим теперь каждый из этих шагов.

Добавление в проект необходимых тестовых зависимостей

Откройте файл pom.xml из проекта Place Reviewer и добавьте следующие зависимости:

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
   <scope>test</scope>
   <exclusions>
      <exclusion>
         <groupId>org.hamcrest</groupId>
         <artifactId>hamcrest-core</artifactId>
      </exclusion>
   </exclusions>
</dependency>
<dependency>
   <groupId>org.hamcrest</groupId>
   <artifactId>hamcrest-library</artifactId>
   <version>1.3</version>
   <scope>test</scope>
</dependency>
```

В следующих разделах мы покажем, как создавать тесты с помощью JUnit — среды тестирования для языка программирования Java, а также Hamcrest — библиотеки, предоставляющей средства сопоставления (matchers), которые комбинируются для формирования значимых выражений намерений (meaningful expressions of intent).

Создание класса конфигурации

Создание класса конфигурации для теста помогает правильно проводить написанные тесты. В каталог src/test/kotlin из проекта Place Reviewer добавьте в пакет com.example.placereviewer пакет config. Включите в создаваемый проект показанный далее класс `TestConfig`:

```
package com.example.placereviewer.config

import org.springframework.context.annotation.ComponentScan
import org.springframework.context.annotation.Configuration

@Configuration
@ComponentScan(basePackages = ["com.example.placereviewer"])
class TestConfig
```

Настройка тестового класса для применения пользовательской конфигурации

Откройте тестовый класс приложения Spring и примените аннотацию `@ContextConfiguration` для указания классов конфигурации, используемых тестовым классом. Для этого откройте файл PlaceReviewerApplicationTests.kt (находится в пакете com.example.placereviewer из каталога src/test/kotlin проекта) и настройте класс конфигурации следующим образом:

```
package com.example.placereviewer

import com.example.placereviewer.config.TestConfig
import org.junit.runner.RunWith
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.junit4.SpringRunner

@RunWith(SpringRunner::class)
@SpringBootTest
@ContextConfiguration(classes = [TestConfig::class])
class PlaceReviewerApplicationTests
```

Отличная работа! Теперь вы готовы написать несколько тестов приложений.

Создание первого теста

Написание кода для тестов приложения похоже на написание кода для любой другой части приложения Spring. Можно точно так же применять компоненты и услуги. Продемонстрируем это.

Добавим следующий интерфейс `TestUserService` в com.example.placereviewer.service:

```
package com.example.placereviewer.service

import com.example.placereviewer.data.model.User
interface TestUserService {
   fun getUser(): User
}
```

Теперь добавим в пакет следующий класс `TestUserServiceImpl`:

```
package com.example.placereviewer.service

import com.example.placereviewer.data.model.User
import org.springframework.stereotype.Service

@Service
internal class TestUserServiceImpl : TestUserService {

   // Тестовая заглушка, имитирующая выборку пользователя
   override fun getUser(): User {
      return User(
         "user@gmaiil.com",
         "test.user",
         "password"
      )
   }
}
```

Вернемся к файлу PlaceReviewerApplicationTests.kt и модифицируем его для отражения этих изменений:

```
package com.example.placereviewer

import com.example.placereviewer.config.TestConfig
import com.example.placereviewer.data.model.User
import com.example.placereviewer.service.TestUserService
import org.hamcrest.Matchers.instanceOf
import org.hamcrest.MatcherAssert.assertThat
import org.junit.Test
import org.junit.runner.RunWith
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.junit4.SpringRunner
```

```
@RunWith(SpringRunner::class)
@SpringBootTest
@ContextConfiguration(classes = [TestConfig::class])
class PlaceReviewerApplicationTests {

    @Autowired
    lateinit var userService: TestUserService
    @Test

    fun testUserRetrieval() {
        val user = userService.getUser()
        assertThat(user, instanceOf(User::class.java))
    }
}
```

Метод `testUserRetrieval()` представляет собой тест, который при запуске применяет заглушку метода, определенную в `TestUserServiceImpl`, для получения доступа к пользователю и подтверждает, что возвращаемый функцией объект является экземпляром класса `User`.

Рис. 10.16. Запуск теста

Для выполнения написанного теста щелкните на кнопке **Run Test** (Выполнить тест), находящейся в IDE-окне в созданном тесте (рис. 10.16) — тест `testUserRetrieval` будет запущен, а результат его выполнения отобразится в нижней части окна IDE (рис. 10.17).

Рис. 10.17. Результат выполнения теста

В рассматриваемом случае написанный тест пройден. И это прекрасно. Однако по мере разработки более крупных и сложных приложений и написания тестов для модулей приложений вы заметите, что написанные тесты чаще всего не выполняются. Если это произойдет, не волнуйтесь — сохраняйте спокойствие и продолжайте отладку приложения. Со временем вы научитесь создавать надежное программное обеспечение.

Подведем итоги

В этой главе завершено наше путешествие в глубины языка Kotlin и создано приложение Place Reviewer. По ходу этого процесса подробно изучено создание слоев представлений для приложений на основе Spring MVC. Кроме того, показано, каким образом можно интегрировать приложение с веб-службами Google Places API для обеспечения доступности приложения.

Мы также узнали о проверке ввода формы с помощью классов `Validator` и `BindingResult`. Наконец, рассмотрена настройка тестирования и написан тест для приложения на основе Spring.

Что дальше?

Если вы дошли до этого раздела, то наверняка успешно изучили данную книгу.

Первым делом позвольте поздравить вас с несомненным успехом! Ваша преданность и стремление к изучению языка Kotlin достойны всяческих похвал. И сейчас у вас может возникнуть вопрос о дальнейшем изучении этого языка. Данный раздел специально написан, чтобы представить вам перспективы движения в этом направлении.

Прежде всего, необходимо совершенствоваться в овладении языком Kotlin. Конечно, для этого необходимо самоотверженно практиковаться, серьезно и длительно, но, в конечном итоге, это вполне выполнимо. Следующие советы помогут вам достичь высокого уровня мастерства программирования на Kotlin:

- **Ежедневно практикуйтесь в программировании на языке Kotlin.** Это важно, поскольку тогда всё, что вы узнали о Kotlin, закрепится в вашем арсенале средств. Кроме того, вы обнаружите новые конструкции, шаблоны, структуры данных и парадигмы, которые расширят ваши навыки программирования.
- **Читайте! Читайте! Читайте!** И нельзя умалить значение этого совета. Для овладения каким-либо навыком важно, чтобы вы как можно больше о нем узнали. Поэтому чтение о Kotlin и о связанных с Kotlin темах должно стать привычкой. Для начала можно обратиться к официальному справочнику по языку Kotlin, доступ к которому открыт по ссылке: **https://kotlinlang.org/docs/reference**. И кроме этого, обращайтесь также и к другим серьезным книгам о языке Kotlin.
- **Задавайте вопросы, относящиеся к языку Kotlin, и отвечайте на них.** Почти на всех этапах вашего пути по овладению Kotlin у вас будут возникать вопросы — и важные, и совсем скромные. Не оставляйте эти вопросы без ответа. Такие платформы, как Stack Overflow и Quora, специально созданы для обмена знаниями. Обретите привычку обращаться к этим платформам для получения ответов на вопросы о языке. Кроме того, отвечайте здесь на вопросы и вы в удобное для вас время.
- **Упражняйтесь самостоятельно!** Наибольший опыт при работе с инструментом можно получить только при выполнении самостоятельной работы с ним. Не уклоняйтесь от принятия предложений по созданию приложений на Kotlin.

Именно опыт необходим для ускоренного совершенствования при работе с инструментом.

Если вы поддерживаете Kotlin и следуете этим советам, то быстро овладеете всеми тонкостями этого языка. Желаем удачи в этом деле!

Другие книги, которые могут вам пригодиться

Если вам понравилась эта книга, вас могут заинтересовать другие книги от Packt:

- **Functional Kotlin** (Mario Arias, Rivu Chakraborty), ISBN: 978-1-78847-648-5
  - Изучите концепции функционального программирования с Kotlin
  - Откройте для себя сопрограммы в Kotlin
  - Раскройте для себя использование плагина funkTionale
  - Изучите монады, функтиоры и аппликативы
  - Объедините функциональное программирование с ООП и реактивным программированием
  - Раскройте использование монад с funkTionale
  - Откройте для себя потоки обработки
- **Kotlin Blueprints** (Ashish Belagali, Hardik Trivedi, Akshay Chordiya), ISBN: 978-1-78839-080-4
  - Узнайте, каким образом универсальные возможности Kotlin делают его отличным выбором для создания приложений на различных платформах и как обеспечиваются преимущества для бизнеса и технологий
  - Создавайте надежные веб-приложения с использованием Kotlin и Spring Boot
  - Без труда создавайте приложения для Android с помощью Kotlin
  - Пишите обширные настольные приложения на Kotlin
  - Узнайте, как язык Kotlin может генерировать код Javascript и каким образом использовать его как на стороне клиента, так и на стороне сервера
  - Составьте представление о написании собственных приложений с помощью Kotlin/Native
  - Изучайте практические аспекты программирования в каждом приложении

Оставьте отзыв — сообщите другим читателям свое мнение о книге

Пожалуйста, поделитесь мыслями об этой книге с другими читателями, оставив отзыв на сайте, где вы ее приобрели. Если вы приобрели книгу на сайте Amazon, пожалуйста, оставьте отзыв на странице Amazon этой книги. Это очень важно, по-

скольку тогда другие потенциальные читатели книги смогут узнать ваше непредвзятое мнение при принятии решения о ее покупке, программисты ознакомятся с тем, что клиенты думают о продуктах, а авторы смогут прочесть ваши отзывы на авантитуле, который создается Packt. Это займет всего несколько минут вашего времени, но представляет большую ценность для потенциальных клиентов, наших авторов и Packt. Спасибо!

Читателям русского издания

Читатели русского издания книги могут посылать свои отзывы на адрес издательства «БХВ-Петербург» **mail@bhv.ru**, а также оставлять их на странице книги на сайте издательства **www.bhv.ru.**

Предметный указатель

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Kotlin

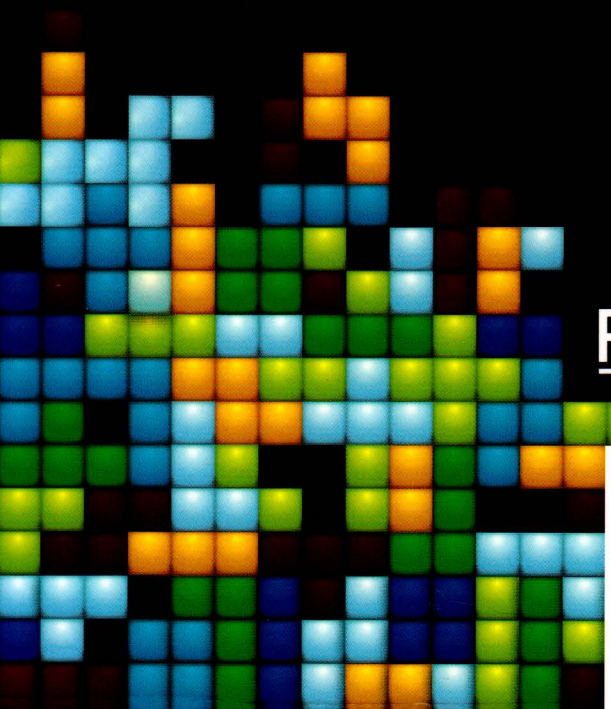
ПРОГРАММИРОВАНИЕ НА ПРИМЕРАХ

Языку программирования Kotlin присущ лаконичный исходный код. А после того как компания Google анонсировала поддержку Kotlin в качестве официального языка программирования для Android, пришло время разработки приложений Kotlin с нуля и запуска их в работу.

Книга знакомит с основными компонентами языка и шаг за шагом демонстрирует его особенности. Возможности Kotlin раскрываются в процессе разработки трех приложений. Первое из них — классическая игра Тетрис, на примере которой показаны принципы объектно-ориентированного программирования. Более сложное приложение — мессенджер — позволяет глубже изучить потенциал Kotlin. В ходе создания третьего приложения, Place Reviewer (Описание местоположений), используются Google Maps API и Place Picker.

В книге рассматриваются следующие темы:

- Компоненты языка программирования Kotlin
- Разработка мощных микросервисов RESTful для приложений Android
- Реактивное программирование эффективных приложений Android
- Реализация шаблона архитектуры MVC и управление зависимостями с помощью Kotlin
- Централизация, преобразование и хранение данных с помощью Logstash
- Защита приложений с использованием Spring Security
- Развертывание микросервисов Kotlin для AWS и приложений Android в Play Store



Packt>

bhv®

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6673-5