Инструменты разработки мобильных приложений развиваются очень быстро, и с помощью Flutter – открытого бесплатного SDK от Google – вы можете создавать приложения для Android, iOS и Google Fuchsia.

На базе примеров из книги вам предлагается создать три полноценных приложения (органайзер, мессенджер и игру), которые можно установить на мобильные устройства или доработать для реального использования. Знакомство с Flutter начинается с изучения основ, а для закрепления своих знаний вы разработаете два традиционных приложения. Затем вы научитесь создавать игры на Flutter и познакомитесь с новыми возможностями этого фреймворка. В книге показано, какие проблемы могут возникнуть при создании Flutter-приложений, рассмотрены способы их решения.

Полезные советы на каждый день облегчат вашу работу!

Прочитав книгу, вы научитесь:

- создавать проекты на Flutter и грамотно их структурировать;
- использовать готовые элементы пользовательского интерфейса во Flutter, включая виджеты, контролы и расширения;
- компоновать пользовательский интерфейс;
- использовать среду разработки Android Studio;
- создавать серверные backend-приложения и подключаться к ним из Flutter-приложений.

Эта книга предназначена для разработчиков, которые ищут возможность создавать мобильные приложения сразу для нескольких платформ на основе общей базы исходных кодов. Желательны наличие опыта разработки программного обеспечения и знание основ iOS и Android.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
e-mail: books@alians-kniga.ru

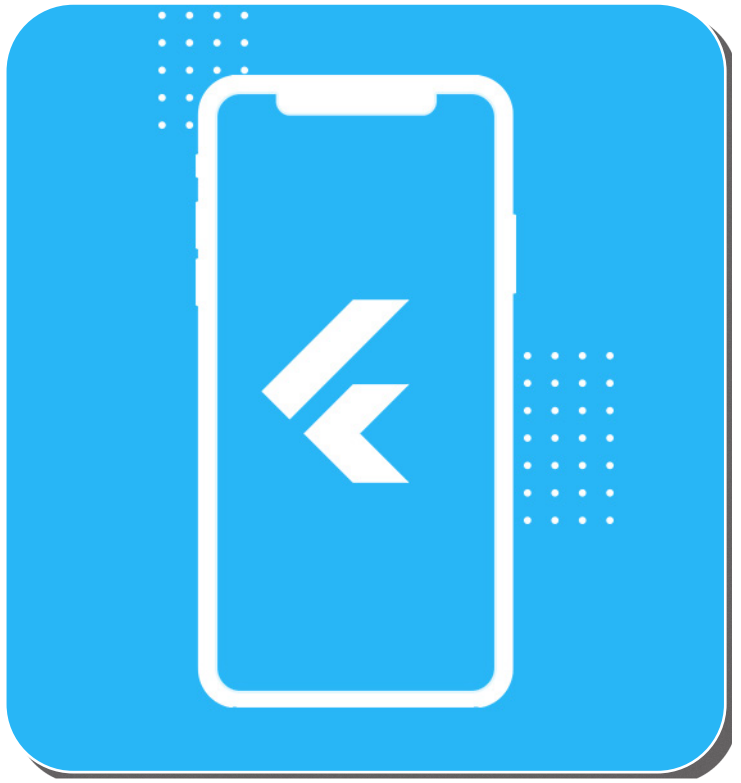
Apress
ДМК ПРЕСС издательство
www.дмк.рф

ISBN 978-5-97060-808-1

FLUTTER на практике

Фрэнк Заметти

FLUTTER
на практике



ДМК ПРЕСС издательство

Binwell University

Фрэнк Заметти

Flutter на практике

Frank Zammetti

Practical Flutter

Improve your Mobile Development with Google's Latest Open-Source SDK

Apress®

Фрэнк Заметти

Flutter на практике

Прокачиваем навыки мобильной разработки с помощью открытого фреймворка от Google

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2020

УДК 004.43
ББК 32.972
326

Заметти Ф.
326 Flutter на практике: Прокачиваем навыки мобильной разработки с помощью открытого фреймворка от Google / пер. с анг. А. С. Тищенко. – М.: ДМК Пресс, 2020. – 328 с.: ил.

ISBN 978-5-97060-808-1

Познакомьтесь с возможностями Flutter – открытого фреймворка от Google. В книге описываются история Flutter, его функционал и конкретные примеры использования. Вы узнаете, как создавать проекты на Flutter и грамотно их структурировать, компоновать пользовательский интерфейс, используя готовые элементы (виджеты, контролы, расширения), разрабатывать серверные backend-приложения и подключаться к ним из Flutter-приложений. Практическим результатом работы с книгой станет создание трех полноценных приложений – органайзера, мессенджера и игры. Впоследствии изучение материала книги позволит вам перейти к более сложным проектам.

Издание предназначено для разработчиков, желающих создавать мобильные приложения сразу для нескольких платформ на основе общей базы исходных кодов. Наличие опыта разработки программного обеспечения и знание основ iOS и Android приветствуется.

УДК 004.43
ББК 32.972

First published in English under the title Practical Flutter; Improve your Mobile Development with Google's Latest Open-Source SDK by Frank Zammetti, edition: 1.

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

ISBN 978-5-97060-808-1 (рус.)
ISBN 978-1-4842-4971-0 (анг.)

От издателя

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Я бы хотел посвятить эту книгу бабочкам, которые порхают на ветру.

Хотя постойте, это слишком просто.

Я бы хотел посвятить эту книгу игрокам, которые, как выражаются британцы, «a flutter on the horses» (перевод: «делают ставки на лошадей [во время скачек]»).

Да, вообще-то это и есть реальное использование слова flutter [прим. пер.: одно из значений flutter – «делать небольшие ставки»], но это тоже слишком просто.

Нет, я бы хотел посвятить эту книгу всему неизведанному, что человечеству только предстоит открыть или даже создать.

По своей натуре я пессимист, но я борюсь с этим каждый день, поскольку я признаю, что вселенная – это удивительнейшее место, и несмотря на все, что нам говорят в вечерних новостях, человечество способно эту великую и чудесную красоту создать.

И с моей заявленной целью – быть бессмертным, потому что смерть – это не я, это то, что было до меня, я просто хочу идти дальше и пропустить это, – я с нетерпением жду, чтобы увидеть всё на свете!

Оглавление

Об авторе

Фрэнк Заметти – автор ряда популярных технических книг, был программистом около 40 лет, 25 из которых занимался этим профессионально, надо же ему было что-то кушать. В те времена на его визитке значилось *архитектор*, хотя он продолжал писать тупой код и каждый день крутился как мог. Фрэнк – первоклассный гик: если он не заставляет свой компьютер выполнять приказы (скорее всего, дьявольские), то занят просмотром, чтением или написанием научной фантастики, моделированием рельсотрона, катушки Теслы или любой другой штуковины, способной прикончить его в любой момент; он любит без причины цитировать *«Вавилон 5»*, *«Властелин колец»*, *«Хроники Риддика»* или *«Настоящих гениев»*, а также играть в компьютерные игры. Еще Фрэнк – рок-музыкант (клавишник) и заядлый любитель пиццы и других углеводов. У него есть жена, собака и несколько детей. Если подвести итог его крутости, то он тот, кто всегда готов воскликнуть «С тобой мой меч!» (ну да, обычно гики цитируют *«Властелин колец»* без очевидной причины).

О техническом рецензенте (обозревателе)

Герман ван Росмален работает разработчиком и архитектором программного обеспечения для De Nederlandsche Bank NV, центрального банка в Нидерландах. За его плечами более 30 лет опыта разработки приложений на разных языках программирования. Герман был вовлечен в создание приложений для мейнфреймов, десктопов, серверов, веб-браузеров и смартфонов. Последние 4 года он занимается в основном разработкой на .NET/C# и Angular после 15 лет работы с Java.

Герман живет в маленьком городке Пейнаккер в Нидерландах со своей женой Лизбет и детьми Барбарой, Леони и Рамоном. Наравне с разработкой софта в свободное время Фрэнк тренирует женскую футбольную команду на протяжении последних 10 лет.

И конечно же, он болеет за Фейеноорд (футбольный клуб из города Роттердам в Нидерландах, который считается одним из ведущих клубов страны)!

О переводе

Данная книга была переведена специалистами компании Binwell, а также действующими преподавателями Binwell University. Надеемся, что наши книги и переводы позволят вам легко освоить новые технологии, а также построить успешную карьеру в сфере информационных технологий.

Над переводом книги «Flutter на практике» работали:

Артем Тищенко, переводчик – руководитель направления Mobile в компании Binwell, специалист в разработке нативных (iOS Swift/Objective C), а также кросс-платформенных (Xamarin и Flutter) мобильных приложений. Ментор Binwell University, соавтор ряда популярных статей для Microsoft Developer Blogs и Хабрахабры.

Вячеслав Черников, редактор перевода – руководитель разработки в компании Binwell, руководитель Binwell University. Работает в сфере Mobile и разработки ПО с 2005 года. Создавал приложения и игры для iOS, Android, Symbian, Windows Mobile, Meego, Linux и Windows UWP. Имеет богатый опыт разработки нативных (iOS Swift/Objective C) и кросс-платформенных (Xamarin, Qt, PhoneGap/HTML5, Unity) мобильных приложений. Автор книги «Разработка мобильных приложений на C# для iOS и Android» (ДМК Пресс, 2020), а также популярных статей для Хакера, Хабрахабры, Microsoft Developer Blogs, спикер, преподаватель и немного [безумный] ученый.

Также выражаем благодарность **Александру Рыжкову** и **Сергею Селютину** за помощь с корректировками.

Благодарности

Если вы никогда не занимались написанием книги, то я открою вам секрет: написание самой книги – это лишь малая часть большой работы над ней. Иногда, мне кажется, наименьшая часть!

Поэтому я хочу поблагодарить всех, кто усердно работал и помогал с этим проектом (не важно, лично или с редактурой), включая Нэнси Чен, Луиса Корригана, Джеймса Маркхэма, Германа ван Росмалена, Вэлмода Спара и Даниша Кумара. Если вашего имени нет в списке, хотя оно должно здесь быть, я приношу свои искренние извинения и благодарю вас.

Также я хотел бы поблагодарить Ларса Бака и Каспера Ланга за создание Dart, довольно элегантного и очень приятного в использовании языка программирования, лежащего в основе Flutter. Говорю как человек, который создал свой собственный язык и набор инструментов для него много лет назад, я очень-очень ценю то, что вы сделали, ребята. Честь вам и слава!

Работа над книгой по Flutter требует от меня благодарности почти всей команде разработчиков этого фреймворка. Я занимаюсь мобильной разработкой около 20 лет (посмотрите на etherient.com, страницу Products, а конкретно Eliminator – игра, которую я выпустил в 2001 году для платформы Pocket PC; я верю, что это было мое первое мобильное приложение, по крайней мере первая удачная попытка), и тогда, насколько я могу судить, я использовал достаточно много утилит, фреймворков и библиотек. Учитывая весь этот опыт, я могу с уверенностью сказать, что Flutter даже с первой версии был на голову выше всех.

Это поразительно, сколько команда Flutter смогла сделать за такой относительно короткий период времени, и без их тяжелой работы я бы, очевидно, не написал эту книгу! Я с нетерпением жду возможности все больше и больше использовать Flutter, а также мне очень интересно, что будет с Flutter дальше!

Введение

Создание кросс-платформенных приложений, которые выглядят и работают как нативные, – сложная задача даже после многих лет работы над ее решением. Вы можете писать отдельные программы для каждой платформы и пытаться сделать их дизайн максимально похожим. Но фактически это значит написать одно приложение несколько раз. Заказчики, как правило, не готовы за такое платить!

Может, взять HTML-страницу и использовать один и тот же код для всех платформ? Но тогда вы можете остаться в дураках с точки зрения возможностей самого устройства, не говоря уже о том, что производительность часто находится на низком уровне (существуют способы минимизирования проблем, но они никогда не исчезнут полностью).

Я занимаюсь этим уже второе десятилетие (серьезно!), поэтому я видел такое много раз. И если я замечу на горизонте образ единорога, то буду сомневаться до конца. Однако если, подойдя поближе, окажется, что единорог действительно реален?

Итак, я представляю вам единорога, который на самом деле существует, – Flutter!

Благодаря талантливым инженерам из Google Flutter – это платформа, позволяющая писать (более или менее) кросс-платформенный код, который одинаково работает на Android и iOS, при этом обеспечивая производительность, идентичную нативным приложениям. Flutter, созданный с использованием современных инструментов и методов разработки, открывает программистам мир мобильной разработки, в котором, даже не побоюсь сказать, *весело* работать!

В этой книге вы познакомитесь с Flutter, создав два реальных приложения вместо надуманных примеров, предназначенных лишь для демонстрации технологии. На этом пути вы узнаете много реальных тонкостей, включая проблемы, с которыми я столкнулся, и решения, которые я нашел. При этом вы получите практический опыт реального использования Flutter, который подготовит вас к созданию собственных приложений в будущем.

Вы также узнаете то, как создавать серверные приложения на Node.js и Web Sockets. Эти бонусы являются полезным дополнением к описанию самого фреймворка Flutter и языка Dart.

Кроме того, вы сможете создать дополнительное третье приложение, которое разительно отличается от первых двух, – игру! Да, мы вместе создадим игру на Flutter, чтобы рассмотреть такие возможности, которые редко встречаются в настоящих приложениях, но дают вам взглянуть на фреймворк с разных сторон и получить максимум опыта. Эта игра может быть не совсем «практичной», но игры всегда увлекательны, а немного веселья никому не повредит!

Дочитав до конца, вы получите представление о том, что такое Flutter, и у вас будет отличная возможность создать свое новое крутое приложение на его основе.

Если вы были компьютерным энтузиастом в 80-х годах, то наверняка помните, каково это было – набивать на своем компьютере машинный код из журнальной статьи в 20 страниц мелким шрифтом, чтобы сыграть в игру или запустить приложение для сверки ваших доходов и расходов (да, мы действительно это делали – были даже радиостанции, транслировавшие исходники, которые затем компилировались в готовое приложение с помощью специальной утилиты, подобно тому как сейчас кодируется звук для передачи по телефонной линии).

Итак, прежде чем начать, я предлагаю вам зайти на веб-сайт Apress, найти эту книгу и скачать себе исходные коды примеров. Это позволит обойтись без стирания пальцев в кровь, перепечатывая листинги из книги!

Не забывайте, лучший способ чему-то научиться – это делать, поэтому обязательно скачайте и измените примеры под себя, увидев своими глазами, на что эти изменения повлияют. После прочтения каждой главы, связанной с приложением, заходите в исходные коды и пробуйте добавить одну или две свои функции, и я даже дам вам пару советов. Думаю, вскоре вы поймете, что благодаря возможностям Flutter небольшие изменения могут существенно повлиять на то, что появляется на экране.

Итак, приготовьтесь к приятной и информативной поездке в мир Flutter!

Я надеюсь, что вам понравится эта книга и вы многому научитесь. Это моя главная цель! Так что перекусите, сядьте в кресле поудобнее, подготовьте ноутбук и приступайте. Вас ждут приключения! (Да, я прекрасно понимаю, как банально это звучит.)

Исходные коды примеров вы можете найти в репозитории на GitHub: `https://github.com/Apress/practical-flutter`.

ГЛАВА 1

FLUTTER: ПЛАВНОЕ ПОГРУЖЕНИЕ

Поехали!

Если вы спросите десятерых разных разработчиков мобильных приложений, как они их разрабатывают для Android и iOS, то, скорее всего, получите десять разных ответов. Но это ненадолго, благодаря дебютанту на этой сцене – Flutter.

В первой главе мы рассмотрим разработку для мобильных устройств, то, как Flutter вписывается в эту картину и, возможно, полностью ее меняет. Мы начнем с ним работать, получим базовое представление о языке и фреймворке, а также подготовим почву для создания реальных приложений.

Итак, давайте сразу поговорим о том, что же такое мобильная разработка.

Медитации над бездной

Разработка софта – это непростая задача!

Я не хочу утомлять вас историей, но факт в том, что я начал, так или иначе, программировать с 7 лет, а это означает, что я занимаюсь этим почти 40 лет (около 25 из них профессионально). Я много повидал и много сделал, но главное, что я понял: разработка софта – это непростая задача. Конечно, некоторые отдельные задачи и проекты могут быть простыми, но в целом это довольно сложная работа, которой мы занимаемся!

И это мы еще не говорим о мобильной разработке, которая куда сложнее!

Я начал разработку мобильных приложений примерно два десятилетия назад, еще во времена Windows CE/Pocket PC и Palm Pilot (были и другие платформы, но именно эти две были единственными настоящими игроками на рынке). Тогда все было не так уж и плохо, несмотря на ограниченный набор устройств и возможностей инструментов разработки, которыми мы располагали. Безусловно, использование этих инструментов было не таким приятным, как сегодня, но был всего один способ разработки приложений для Pocket PC, один способ разработки приложений для Palm OS. Звучит не очень, но отсутствие выбора приводит к отсутствию путаницы, что является одной из самых больших трудностей в области разработки софта на сегодняшний день.

А еще, хотя сегодня это считается непопулярным, тогда не было понятия кросс-платформенной разработки. Раньше приходилось писать код дважды, чтобы приложение работало на обеих платформах. Учитывая различия между ними, это было не так-то просто.

С тех пор индустрия мобильных устройств и приложений претерпела немало эволюционных изменений, подъемов и падений. Долгое время у нас было много платформ для поддержки: Android, iOS, webOS, Tizen, Windows Mobile и несколько других, которые я даже не помню. Все это время перенос

приложений между платформами был нормой, поскольку не было хорошего кросс-платформенного подхода, по крайней мере без существенных компромиссов. Да, со временем стало проще, потому что улучшился инструментарий для нативной разработки. Apple выпустила свой SDK для iOS в 2008 году, а Google выпустил свой Android SDK год спустя – в 2009-м. Нам приходилось разрабатывать приложения для каждой платформы, поскольку разработка iOS основана на языке Objective-C (сегодня чаще на языке Swift), в то время как разработка Android основана преимущественно на языке Java (теперь чаще на Kotlin).

В конце концов, количество платформ стало сокращаться. На сегодняшний день это гонка Android и iOS, хотя есть и другие платформы, которые чаще всего используются *только* при решении специфических задач. В связи с этим применение кросс-платформенных инструментов становится более привлекательным.

Наша прогрессивная эпоха интернета предлагает создавать приложения с помощью технологий, лежащих в его основе, и как результат получить приложение, которое выглядит и работает примерно одинаково на обеих платформах (теоретически и на других тоже). Это сопровождается компромиссами, которые со временем минимизируются, но все еще существуют. Такие вещи, как производительность и прямой доступ к возможностям «железа», все еще сложно совместить с веб-технологиями.

Однако, помимо веб-технологий, в последние несколько лет мы наблюдали рождение и других кросс-платформенных инструментов, которые позволяют нам написать приложение один раз и работать с ним примерно одинаково в разных операционных системах. Популярные варианты – Corona SDK (в первую очередь для игр, но не обязательно), Xamarin, PhoneGap (просто веб-технологии, умно завернутые в собственный компонент WebView), Titanium и Sencha Touch (опять же, на основе веб-технологий, но с хорошим слоем абстракции над ним), может, еще несколько. Так что сейчас нам доступно множество различных вариантов, каждый со своими плюсами и минусами.

А теперь внимание! На арену выходит новый конкурент, жаждущий убить остальных и показать единственный верный путь написания кросс-платформенных мобильных приложений: Flutter.

Да, это немного глупое название… но знаете, мы можем закрыть на это глаза, потому что преимуществ у него выше крыши!

Что за (глупое) название?

Flutter – это продукт Google – ну, вы знаете, корпорации, которая основательно контролирует интернет, хорошо это или плохо (в случае с Flutter я думаю, что хорошо). Изначально этот фреймворк родился под именем Sky в 2015 году на саммите разработчиков Dart (не забудьте это слово, Dart, мы к нему скоро вернемся). Сначала он работал только на операционной системе Android от са-

мого Google, но вскоре был портирован и на iOS, так что сегодня он поддерживает две ведущие мобильные операционные системы.

Первые версии Flutter были выпущены сразу после его анонса, а кульминацией стал выпуск стабильной версии «Flutter 1.0» 4 декабря 2018 года. После этого Flutter был готов к прайм-тайму, и пришло время для разработчиков запрыгивать на борт! Популярность Flutter можно было бы охарактеризовать как метеорическую, и на это есть веские причины.

Одна из них заключается в следующем: первоначально заявленная цель Flutter или, по крайней мере, одна из основных, заключалась в отрисовке пользовательского интерфейса со скоростью 120 кадров в секунду. Google осознавал, что отзывчивый интерфейс приведет пользователей в восторг, поэтому эта функциональность и легла в основу Flutter. Это благородная цель, которой достигают лишь немногие кросс-платформенные фреймворки (даже нативные инструменты – и те часто испытывают трудности со скоростью отрисовки сложного интерфейса).

Flutter также предоставляет свои готовые компоненты пользовательского интерфейса – в отличие, например, от Xamarin и ReactNative, он не использует нативные контролы. Другими словами, когда вы хотите, чтобы Flutter отобразил кнопку, он рисует ее сам, а не просит об этом операционную систему, как делают другие фреймворки. Именно это и отличает Flutter от остальных и позволяет приложениям быть одинаковыми на разных платформах. Важно то, что новые компоненты пользовательского интерфейса, или *виджеты* (это слово тоже запомните, потому что, как и Dart, оно тоже скоро встретится), могут быть добавлены во Flutter быстро и легко, не беспокоясь о том, поддерживает ли их сама операционная система.

Это также позволяет Flutter предоставлять специфические наборы виджетов в стилистиках Material и Cupertino. Первый реализует Material Design от самой Google – стиль Android по умолчанию. Последний реализует стиль iOS от Apple.

Flutter можно разделить на четыре основные части, включая Dart. Я собираюсь оставить это до следующего раздела, так что давайте перейдем ко второму компоненту – основному движку Flutter. Этот движок в основном написан на C++ и использует библиотеку Skia, так что производительность отрисовки сравнима с нативной. Skia – это небольшая графическая библиотека с открытым исходным кодом, которая также написана на C++ и имеет очень высокую производительность на всех поддерживаемых платформах.

В качестве третьего основного компонента Flutter предоставляет унифицированный доступ к возможностям поддерживаемых операционных систем. Другими словами, код для запуска камеры на iOS и Android будет единым, а для этого можно использовать готовые методы Flutter.

Последний компонент – это виджеты, но, как и Dart, они тоже заслуживают собственного раздела, так что вернемся к ним позднее.

Если коротко, то Flutter состоит из этих четырех модулей, поэтому не так уж и много нужно знать, чтобы начать на нем разрабатывать. Тем не менее я ду-

маю, что немного углубленное изучение инструментов никогда не будет лишним. Надеюсь, вы согласны!

Теперь давайте более детально разберем Dart и виджеты, о которых говорили ранее.

Dart: язык богов?

Когда Google начал работать над Flutter, им предстояло ответить на главный вопрос: какой язык программирования выбрать? Может быть, язык веб-разработки, такой как JavaScript? Или же Java, язык Android? Или ради поддержки iOS выбрать Swift (в конце концов, Swift является языком с открытым исходным кодом)? Возможно, что-то вроде Go или Ruby было бы хорошим вариантом. Как насчет «старой школы», C/C++? А может, попробовать C # от Microsoft (у него тоже открытый исходный код)?

Я уверен, что было много вариантов, но в конце концов Google решил (не без причины!) использовать язык, который они создали несколько лет назад: Dart.

Вся следующая глава посвящена Dart, поэтому сейчас я воздержусь от деталей, но приведу небольшой пример:

```
import "dart:math" as math;

class Point {
   final num x, y;
   Point(this.x, this.y);
   Point.origin() : x = 0, y = 0;
   num distanceTo(Point other) {
      var dx = x - other.x;
      var dy = y - other.y;
      return math.sqrt(dx * dx + dy * dy);
   }

   Point operator +(Point other) => Point(x + other.x, y + other.y);
}

void main() {
   var p1 = Point(10, 10);
   var p2 = Point.origin();
   var distance = p1.distanceTo(p2);
   print(distance);
}
```

Не обязательно сразу детально понимать всё, что вы здесь видите. Тем не менее если вы работали раньше с любым C-подобным языком, например Java или JavaScript, то готов поспорить, что вы без проблем во всем разберетесь. В этом и заключается главное преимущество Dart: большинство современных разработчиков смогут написать и понять подобный код довольно легко.

ПРИМЕЧАНИЕ. Интересно, что мы называем языки C-подобными, но сам C – потомок гораздо более старого языка ALGOL. Я думаю, что ALGOL никогда не получит заслуженного уважения, так что этой ремаркой я выражаю всю любовь к нему!

Не вдаваясь во все мельчайшие подробности (для этого предназначена следующая глава), я думаю, что даже небольшой справки по Dart будет достаточно. Google создал Dart еще в 2011 году, и изначально он был представлен на конференции GOTO в Орхусе, Дания. Первый релиз 1.0 состоялся в ноябре 2013 года, примерно за два года до выпуска Flutter. За Dart стоит благодарить Ларса Бака (который разработал ещё и JavaScript-движок V8, используемый в Chrome и Node.js) и Каспера Лунда.

Dart – это лаконичный язык, который быстро набирает обороты, в основном благодаря Flutter. Поскольку он считается языком общего назначения, его широко используют для создания веб-приложений, серверного кода и приложений IoT (Internet of Things, «интернет вещей»). Пока я писал эту главу, вышел опрос о том, какие языки программирования вызывают наибольший интерес разработчиков в 2019 году, опубликованный JAXenter: `https://jaxenter.com/poll-results-dart-word-2019-154779.html`. В результате два языка заметно опередили остальные: Dart и Python, Dart вырвался вперед. Dart испытал наибольший рост в 2018 году. И хотя Flutter – почти наверняка один из самых популярных вариантов его использования, но, несмотря на это, Dart развивается во всех направлениях. Так что будьте уверены, Dart не обделен вниманием.

Так что же такое Dart? Предыдущий пример кода демонстрирует главные ключевые моменты, о которых я хотел сказать:

- Dart полностью объектно-ориентирован;
- инфраструктура языка включает в себя сборщик мусора, поэтому нет необходимости следить за памятью;
- его синтаксис основан на C, который подойдет большинству разработчиков (тем не менее, как и у любого другого языка с похожим синтаксисом, у него есть ряд особенностей, которые поначалу могут сбить вас с толку);
- он поддерживает общие языковые конструкции, такие как интерфейсы, наследование, абстрактные классы, шаблонные классы (generics, или «дженерики») и статическую типизацию;
- Dart включает проверку соответствия типов. Это позволяет использовать алгоритм для контроля правильности вашего кода;
- он поддерживает многопоточность, так что одновременно может исполняться несколько отдельных процессов, обеспечивая высокую производительность;
- Dart может компилироваться в нативный код для повышения производительности. Он не только компилируется в код для процессоров ARM и x86 в режиме Ahead Of Time (при сборке приложения), но и может транслиро-

ваться в JavaScript, а также поддерживает динамическую компиляцию во время исполнения (Just In Time);

- Dart позволяет использовать большой набор репозиториев с готовыми библиотеками, которые обеспечивают дополнительную функциональность для всего, что может понадобиться разработчикам;
- поддержка популярных сред разработки, включая Visual Studio Code и IntelliJ IDEA;
- ядро Dart поддерживает создание «слепков» (snapshots), которые позволяют упаковать весь исполняемый код (не только ваш код, но и библиотеки) в единый двоичный файл, что ускоряет запуск приложения.

Dart также зарегистрирован в качестве международного стандарта ECMA-408, а его актуальную спецификацию всегда можно получить на сайте www.dartlang.org/guides/language/spec.

Как я уже отмечал ранее, вся вторая глава будет посвящена Dart, а пока мы перейдем к следующей важной теме.

Виджеты окружают!

Давайте вернемся к разговору о главном госте нашего шоу, Flutter, и концепции, которая лежит в его основе, – виджетах.

Flutter – это и есть виджет. Когда я говорю, что *он и есть* виджет, я имею в виду... ну... я имею в виду, что *почти* все в нем является виджетом (гораздо сложнее найти во Flutter то, что виджетом *не является*!).

Что же такое виджет, спросите вы? Это части вашего пользовательского интерфейса (хотя и не все виджеты явно отображаются на экране). Виджет также представляет собой фрагмент кода, например:

```
Text("Hello!")
```

и это также виджет...

```
RaisedButton(
   onPress : function() {
      // Сделай что-нибудь.
   },
   child : Text("Click me!")
)
```

и это тоже виджет...

```
ListView.builder(
   itemCount : cars.length,
   itemBuilder : (inContext, inNum) {
      return new CarDescriptionCard(card[inNum]);
   }
)
```

и наконец, это виджет:

```
Center(
   child : Container(
      child : Row(
         Text("Child 1"),
         Text("Child 2"),
         RaisedButton(
            onPress : function() {
               // Do something.
            },
            child : Text("Click me")
         )
      )
   )
)
```

Последний пример интересен тем, что на самом деле это иерархия виджетов: виджет Center, а в нем виджет Container, содержащий виджет Row, который, в свою очередь, содержит дочерние виджеты Text и кнопку RaisedButton.

Не важно, что это за виджеты (хотя названия их отлично характеризуют), *главное*, что вся иерархия виджетов, которую вы видите, *сама по себе* также считается виджетом Flutter.

Да, во Flutter повсюду виджеты! Виджеты окружают! Flutter – это Опра [Популярная ведущая ТВ-шоу в США. – *Прим. перев.*] в мире фреймворков пользовательского интерфейса: вам нужен виджет – вы получаете виджет! Да, вы получаете виджет! Вы ВСЕ получаете виииииджеты!

Как я сказал ранее, во Flutter практически все – это виджет. Есть очевидные вещи, о которых люди думают, употребляя слово виджет в контексте пользовательского интерфейса: кнопки, списки, изображения, поля текстовых форм и все такое прочее. Это все виджеты. Но во Flutter то, что вы виджетами не считаете, это все еще они. Например, рамка вокруг изображения, состояние поля текстовой формы, текст, отображаемый на экране, даже тема, которую использует приложение.

В результате мы видим, что код во Flutter – это гигантская иерархия виджетов (и эта иерархия имеет конкретное имя во Flutter: Widgets Tree, «дерево виджетов»). Видите ли, большинство виджетов являются контейнерами, это означает, что они могут иметь дочерние элементы. Некоторые виджеты могут иметь только один такой элемент, в то время как другие могут иметь много. И тогда у каждого из них может быть один или несколько дочерних элементов, и так далее, и так далее!

Все виджеты являются классами Dart, и у них есть обязательное требование: предоставить метод `build()`. Этот метод должен возвращать… подождите, подождите… *другие виджеты*! Есть очень мало исключений из этого, например низкоуровневые виджеты, такие как виджет Text, который возвращает при-

митивный тип (в нашем случае – String), но большинство возвращает один или несколько виджетов. Помимо этого требования, виджет – это старый добрый класс Dart, который не отличается от класса в любом другом объектно-ориентированном языке (за исключением синтаксиса).

Виджет во Flutter расширяет (extends) один из стандартных классов, которые он же и предоставляет, что характерно для объектно-ориентированной парадигмы. Расширенный класс (extended class) определяет, с каким виджетом мы имеем дело на фундаментальном уровне. Есть два самых базовых класса, которые вы будете использовать, вероятно, 99% времени: `StatelessWidget` и `StatefulWidget`.

Виджет, унаследованный от `StatelessWidget`, не изменяется после отображения и называется виджетом без состояния, потому что он не имеет состояния (логично). Такие виджеты, как `Icon` (отображает небольшие изображения) и `Text` (отображает строки текста), тоже называют виджетом без состояния. Примером подобного класса может быть следующее:

```
class MyTextWidget extends StatelessWidget {
   Widget build(inContext) {
      return new Text("Hello!");
   }
}
```

Да, здесь нет ничего особенного!

В отличие от `StatelessWidget`, наследники базового класса `StatefulWidget` изменяются, когда пользователь взаимодействует с ним. `CheckBox`, `Slider`, `TextField` – это все известные примеры виджетов с состоянием (и, кстати, когда вы видите, что они написаны с большой буквы, то это означает, что я имею в виду фактические имена классов Flutter, а не общие термины). Когда вы кодируете такой виджет, вам нужно создать *два* класса: сам класс виджета с состоянием (`StatefulWidget`) и класс состояния (`State`), связанный с ним. Вот пример виджета StatefulWidget и связанного с ним класса State:

```
class LikesWidget extends StatefulWidget {
   @override
   LikesWidgetState createState() => LikesWidgetState();
}

class LikesWidgetState extends State<LikesWidget> {
   int likeCount = 0;

   void like() {
      setState(() {
         likeCount += 1;
      });
   }

   @override
```

```
   Widget build(BuildContext inContext) {
      return Row(
         children : [
            RaisedButton(
               onPressed : like,
               child : Text('$likeCount')
            )
         ]
      );
   }
}
```

Опять же, я не жду, что вы полностью поймете этот код, так как расширять знания по Dart мы начнем позже. Но я всё равно считаю, что вы приблизительно понимаете, что здесь происходит. По крайней мере, то, как код виджета и его объект состояния взаимодействуют и связаны. Может, это не так уж очевидно, но не беспокойтесь, это ненадолго!

Возвращаясь к виджетам без состояния, следует отметить, что термин «виджет без состояния» не совсем точен, потому что, будучи классом Dart, который имеет свойства и инкапсулированные данные, виджеты без состояния в некотором смысле имеют состояние. Основное различие между StatelessWidget и StatefulWidget заключается в том, что виджет без состояния (stateless) не умеет *автоматически* перерисовываться при изменении его «состояния», тогда как виджет с состоянием может. Когда виджет с состоянием изменяется, независимо от того, что вызывает изменение, возникают определенные события жизненного цикла. Когда состояние виджета изменяется, то происходит вызов определенных методов, и он перерисовывается (если необходимо, то Flutter делает это автоматически).

Представьте: оба типа виджетов могут иметь состояние, но Flutter распознает и управляет только stateful-виджетом. Таким образом, только StatefulWidget может быть автоматически перерисован в ответ на внешнее событие, и это контролируется самим Flutter, а вам не нужно прописывать код вручную.

Возможно, теперь вы захотите пользоваться только виджетами с состояниями, так как это сократит вашу работу, но вскоре вы поймете, что это не совсем так. В результате вы предпочтете виджет без состояния, даже если сейчас это кажется вам нелогичным. Но давайте опустим это ненадолго.

Есть два важных аспекта, на которые вы уже наверняка обратили внимание. Во-первых, пользовательский интерфейс построен путем создания виджетов. Это приводит к дереву виджетов, о котором я упоминал ранее. В то время как код самих виджетов является обычным классом, важно учитывать вложенность и расположение виджетов на экране. Это важно потому, что большинство виджетов Flutter сами по себе довольно просты, и только через композицию вы можете создать сложный пользовательский интерфейс. Даже относительно тривиальный пользовательский интерфейс содержит в себе целую группу виджетов.

Во-вторых, пользовательский интерфейс во Flutter описывается напрямую в коде. Я знаю, что это кажется очевидным, но задумайтесь вот над чем: для пользовательского интерфейса Flutter нет отдельного языка разметки, как HTML в веб-разработке. Преимущество заключается в том, что есть только один язык для изучения, единая парадигма для восприятия. Может, сначала это и незаметно, но это важное преимущество Flutter над конкурирующими вариантами.

Пока это все, что нужно знать о виджетах. Мы изучим каталог виджетов более подробно начиная с главы 3, и, конечно, мы рассмотрим использование каждого из них в главе 4, когда будем делать приложения с ними. В конце концов, вы получите хорошие знания о наиболее распространенных виджетах Flutter, а также другие, не менее полезные базовые навыки использования и создания виджетов в целом.

Ближе к делу: плюсы и минусы Flutter

Как и с любым фреймворком, нам как хорошим разработчикам нужно оценить преимущества и подводные камни Flutter. Во Flutter есть и то, и другое, я же не буду бросаться в крайности и говорить, что это «панацея от всех бед» или же полный провал. И если кто-то говорит вам, что он идеален, то вам просто пускают пыль в глаза. Во Flutter есть свои недостатки, но я скромно предположу, что есть довольно много проектов и разработчиков, для которых он может стать отличным вариантом, если не *лучшим*.

Давайте уже обсудим все плюсы и минусы Flutter, а также сравним его с конкурентами.

- За: горячая перезагрузка (hot reload) – к ней я вернусь после того, как мы изучим настройку окружения и взглянем на первый пример приложения, – вы сами убедитесь, что это большое преимущество Flutter. ReactNative также включает возможность горячей перезагрузки, особенно если вы используете сторонний компонент Expo. Однако во Flutter эта функциональность реализована более качественно и стабильно. Немногие фреймворки, даже нативные, могут похвастаться подобными возможностями.
- Против: только для мобильных устройств. На момент написания этой книги можно было использовать Flutter только для разработки мобильных приложений iOS и Android. Если вы полюбите Flutter, то будете разочарованы тем, что не сможете использовать его для разработки всех ваших приложений. Тем не менее обратите внимание, что я начал со слов «на момент написания». Это потому, что с высокой степенью вероятности Flutter также будет доработан для поддержки приложений для веб-браузеров, Windows, macOS, Linux и других платформ. [И да! Flutter портировали на веб-браузеры и десктопные платформы, хотя процесс этот еще только начался. – *Прим. перев.*]

- За: да, он действительно кросс-платформенный – ваши приложения Flutter будут корректно работать на iOS и Android (и в конечном итоге преемнике Android – Fuchsia). Flutter поставляет два набора виджетов из коробки, один для iOS и один для Android, поэтому ваши приложения могут как выглядеть одинаково на обеих платформах, так и учитывать стилистику целевой операционной системы.
- Против: веб-разработчики, не привыкшие видеть описание логики поведения и разметки пользовательского интерфейса в одном классе, как правило, очень эмоционально реагируют на примеры Flutter. Для сравнения, ReactNative, у которого также в одном классе описывается и логика, и разметка, первое время страдал от подобных жалоб, но потом разработчики привыкли.
- За: Dart – простой и мощный, объектно-ориентированный и строго типизированный, что позволяет разработчикам быть очень продуктивными, быстрыми и делать меньше ошибок. Как только вы пройдете тернистый начальный путь обучения, вам понравится Dart, особенно в сравнении с JavaScript, Objective-C или Java.
- Против: Google – я причисляю это к недостаткам, и вы определенно можете со мной не согласиться (раньше я часто спорил об этом сам с собой). Некоторые люди чувствуют себя некомфортно от такого контроля Google над интернетом, даже если Google этим активно не пользуется. Когда вы доминируете, то, как правило, многое контролируете. Тем не менее некоторые люди опасаются, что Google наращивает обороты и на еще одном направлении – для них это может быть уже слишком. Другие, конечно, посмотрят на это и скажут, как здорово, что гигант поддерживает такую замечательную технологию. Так что этот «недостаток» можно назвать спорным. И выбрать сторону можете только вы.
- За: виджеты – Flutter предоставляет разработчикам богатый набор виджетов, и этого может быть вполне достаточно для построения любого приложения. Вы также можете создавать и свой собственный виджет (на самом деле вы *всегда* будете так делать, не важно, на каком уровне), и даже использовать многие сторонние виджеты, дабы расширить возможности приложения. Эти виджеты так же просты в использовании, как и те, что нам предлагает сам Flutter.
- Против: дерево виджетов (widgets tree) может стать недостатком, ведь иногда вы будете сталкиваться с очень глубоко вложенной иерархией, и разобраться со структурой кода станет не так-то просто. С развитием интернета мы к этому уже привыкли, потому что HTML сам по себе является древовидным, однако поскольку практически все во Flutter является виджетами, иерархия иногда может быть даже глубже HTML, а стиль кода Dart выглядит сложнее. Конечно, есть методы, позволяющие это упростить. Но о них я расскажу позже на примерах реального кода, и да, это все

еще недостаток, потому что вы должны осознавать его и уметь работать с ним самостоятельно. Ни Flutter, ни Dart не предложат вам подсказок в решении.

- За: инструменты – как вы увидите в следующем разделе, настроить среду разработки для Flutter очень легко. Тем не менее вы можете выйти за пределы этой базовой среды и использовать многие уже привычные вам инструменты.
- Против: реактивное программирование и управление состоянием – Flutter обычно считается реактивным (reactive). Метод `build()`, который вы видели ранее, принимает в качестве аргумента текущее состояние, а то, что он возвращает, – это визуальное представление этого виджета, на основе обновленного состояния. Когда обновляется состояние, виджет «реагирует» на это и обновляет себя с помощью повторного вызова метода `build()`. Всё это – стандартный механизм Flutter и типовой жизненный цикл виджета. Сравните это с «нереактивными» подходами, когда вы создаете виджет, а затем самостоятельно вызываете нужные методы для его модификации или обновления. В целом реактивная парадигма довольно удобна, хотя для Flutter она может быть и недостатком, потому что иногда это усложняет простые вещи (вы сами увидите подобные проблемы в последующих главах, а также научитесь с ними справляться). С этим связана и сложность управления состояниями, которая является недостатком Flutter в том смысле, что нет канонически правильного и неправильного способа это делать. Существует множество подходов, и у каждого есть свои плюсы и минусы, а вам нужно будет решить, что лучше соответствует вашим потребностям (и да, я буду предлагать то, что считаю хорошим подходом). Google работает над таким каноничным подходом прямо сейчас, но пока он не готов, я буду рассматривать отсутствие определенного пути как недостаток (хотя некоторые считают гибкость преимуществом!).
- За: специфические для платформы виджеты – поскольку интерфейсы Flutter пишутся с помощью кода, у вас может быть одна кодовая база, которая поддерживает как iOS, так и Android, но даже в ней есть различия, которые вам нужно учитывать. Например, вы всегда можете узнать в коде значения `Platform.isAndroid` и `Platform.isIOS`, чтобы определить, на каком устройстве работает ваше приложение, а затем добавить условие для создания разных виджетов для разных платформ. Возможно, вам нужен RaisedButton на Android и Button на iOS.
- Против: размер приложения – приложения Flutter, как правило, немного больше своих нативных аналогов, потому что они включают основной движок Flutter, библиотеки и другие ресурсы фреймворка. Размер приложения элементарного «Hello, World!» на Flutter может превышать 7 МБ. Поэтому если вам решительно важен размер приложения, то Flutter может стать не лучшим выбором.

Надеюсь, что к этому моменту у вас появилось понимание сильных и слабых сторон Flutter, поэтому предлагаю перейти к практике.

Хватит болтать, начинаем практику с Flutter!

Прежде чем мы доберемся до кода, нам следует установить Flutter и некоторые инструменты, которые могут понадобиться. Надеюсь, вы понимаете, что не так уж просто начать писать код на Flutter, не установив его!

К счастью, настроить рабочее окружение довольно легко.

Flutter SDK

Первый шаг, который вы должны сделать, – это загрузка, установка и настройка Flutter SDK. Это очень важно! Второй шаг, который технически не обязателен, но необходим для целей этой книги, – это загрузка, установка и настройка Android Studio, включая Android SDK и эмулятор.

Во-первых, зайдите на `https://flutter.io` для загрузки установочных пакетов и получения документации Flutter. Нажмите кнопку **Get Started** в верхней части. Найдите **Install** и выберите свою операционную систему (Windows, MacOS или Linux).

Обратите внимание, что мне совсем не стыдно признать, что я в первую очередь пользователь Windows. Это то, в чем я разбираюсь и что предпочитаю! Так что эта книга будет ориентирована на Windows, и если вы используете другую ОС, то вы в какой-то степени будете сами по себе. С учетом вышесказанного я буду обращать ваше внимание на особенности инструментов для разных ОС, если будут иметься существенные различия. На самом деле их не должно быть вне зависимости от того, используете вы Windows или нет. При этом Flutter сам проинструктирует вас, если возникнут проблемы.

Выберите соответствующую ссылку, и Flutter предоставит вам информацию о загрузке и установке SDK. SDK не отличается от любого другого софта, поэтому трудностей быть не должно. Хочу отметить, что в инструкции вас просят указать путь для SDK. Но вы можете пропустить этот шаг. Просто обратите внимание, что если вы пропустите его, все команды должны быть выполнены из каталога SDK или с указанием полных путей до приложений в этом каталоге. Как только мы дойдем до шага с Android Studio, вы обнаружите, что добавление SDK к пути действительно не имеет значения, Android Studio сделает это за вас. Но если вы собираетесь работать с командной строкой, то не забудьте прописать полный путь.

Первая команда, которую вы *будете* выполнять из командной строки, и первая, которую вы будете делать сразу после установки SDK в соответствии с инструкциями на сайте, – это «flutter doctor». Большинство команд, которые вы будете вводить при работе с SDK, если не все из них, начинаются с flutter, который фактически является исполняемым файлом, например doctor – это одна

из команд, которую вы можете выполнить с его помощью. Это важно, потому что команда doctor проверит, сможете ли вы начать работать, и если нет, то скажет вам, что не так.

Если вы запустите ее сейчас, то, скорее всего, обнаружите проблемы, так и должно быть на данном этапе, не волнуйтесь, следующий шаг это исправит: установка Android Studio.

Android Studio

Еще раз, инструкции на сайте Flutter сопровождают вас и имеют незначительные различия для каждой ОС, но как только вы установили Android Studio, запустите её и воспользуйтесь мастером настройки. Это загрузит Android SDK, образы эмулятора и все необходимое для его работы. Затем установите специальные плагины Dart и Flutter, в документации это подробно описано.

Если вы продолжите следовать инструкциям, запустится процесс подключения вашего Android телефона или планшета к компьютеру, но убедитесь, что `flutter doctor` его видит. Однако вы можете пропустить этот шаг! Конечно, если у вас есть устройство Android, то необходимость в эмуляторе пропадает.

Однако если вы предпочитаете iOS или если вам не нравится использовать реальное устройство при разработке кода Flutter – я не подключаю свой телефон, – тогда мое предложение состоит в том, чтобы войти в Android Studio, запустить менеджер AVD (Android Virtual Machine), который вы можете найти в меню конфигурации на экране запуска, и создать себе виртуальное устройство Android. Я предлагаю создать виртуальное устройство Pixel_2, используя уровень API 28 (убедитесь, что вы установили данную версию API), и задать ему разрешение 1080×1920 (420 dpi) с операционной системой Android 9. Затем выберите образ x86 (x86_64). Если говорить о производительности, виртуальные устройства Android долгое время имели плохую репутацию, но сейчас этот тип виртуальных устройств работает исключительно хорошо, достигая почти нативной производительности в большинстве случаев. Хотя это не имеет значения, идем дальше и настроим его SD-карту на 512 МБ. Значения по умолчанию должны соответствовать вашим желаниям, но уровень API и тип процессора – это ключевые аспекты.

Когда всё будет готово – запускаем код Flutter на эмуляторе. Или вы можете сделать все это из командной строки с помощью SDK. Мы же в данной книге будем делать это в Android Studio.

Обратите внимание, что если вы повторно запустите `flutter doctor`, он все равно будет сообщать о проблеме, а именно что он не может найти устройство Android, предполагая, что виртуальное устройство, которое вы создали, не работает. Но если flutter doctor обнаружит его, он выдаст вам справку, что все хорошо. Наконец, если вы видите сообщение о том, что эмулятор *не* запущен, при условии что реальное устройство Android подключено и это единственная проблема, о которой сообщает flutter doctor, то вы можете не обращать на нее внимания.

Если вас сейчас интересует iOS, пожалуйста, успокойтесь! Хоть мы используем Android Studio, это никоим образом не означает, что все это не применимо для iOS. Об iOS необходимо знать в первую очередь то, что если вы хотите протестировать реальное устройство iOS или создать свое приложение для продажи, вам понадобится компьютер Mac и Apple Xcode IDE. Приложения для продажи не рассматриваются в этой книге, хотя, будь то для iOS или Android, эмулятор отлично соответствует нашим задачам.

Типичное приложение «Hello, World!»

Если вы продолжите следовать инструкции на веб-сайте, то последним шагом станет создание небольшого приложения Flutter. Документация там отличная, но я предлагаю пропустить ее и позволить мне провести вас по этому пути самому.

Первым шагом необходимо позволить Android Studio (в сочетании с Flutter SDK) создать приложение для нас. Процесс довольно прост, и как только мы запустим это базовое приложение на нашем виртуальном устройстве, мы немного изменим его, чтобы вы могли видеть горячую перезагрузку наглядно.

Но сначала давайте создадим проект! При первом запуске Android Studio вы увидите окно, как показано на рис. 1-1.

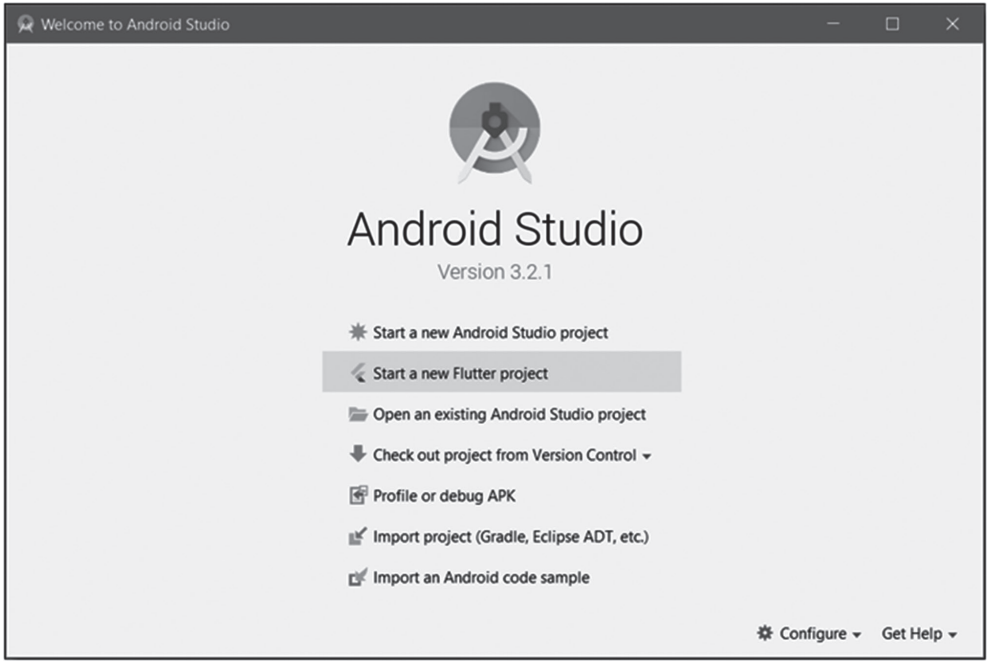


Рисунок 1-1. *Первый шаг в Android Studio*

Видите строчку *Start a new Flutter Project*? Это то, что нужно, кликайте! Вы увидите начальный экран мастера нового приложения, как показано на рис. 1-2.

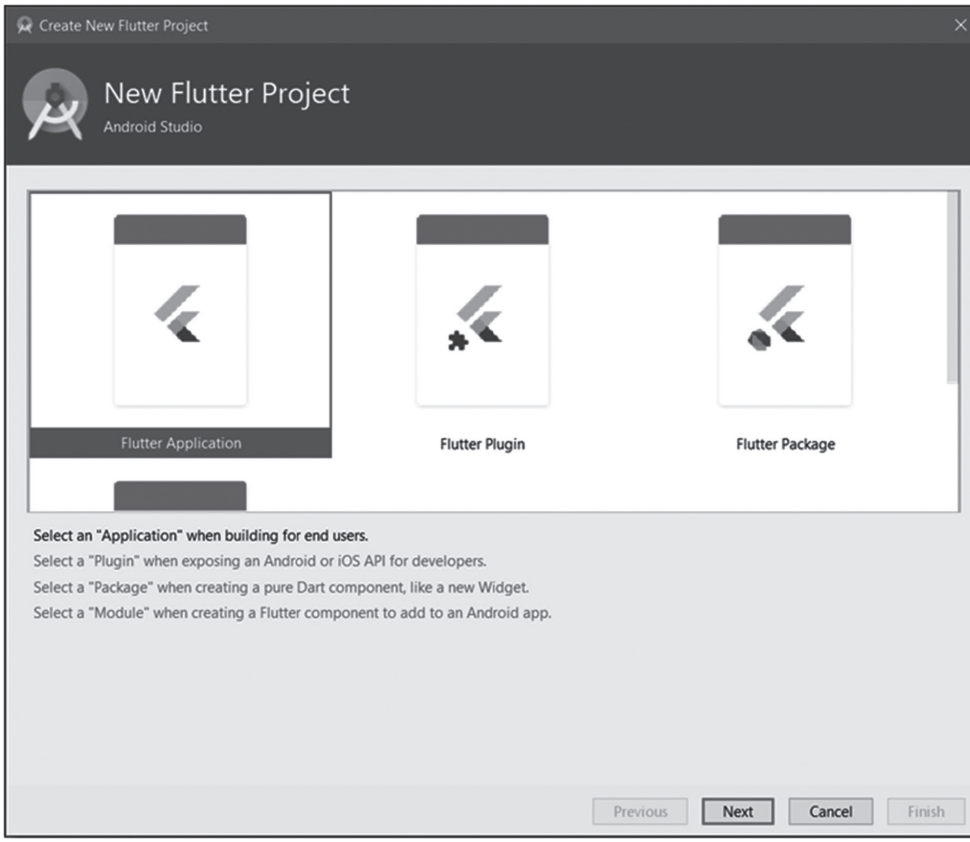


Рисунок 1-2. *Выберите тип проекта Flutter, который хотите создать*

Существует четыре типа проектов Flutter, которые вы можете создать:

- Flutter Application (его мы будем использовать в этой книге);
- Flutter Plugin (плагин позволяет использовать нативную функциональность Android или iOS для ваших приложений Flutter на основе Dart);
- Flutter Package (необходим только в том случае, если вы хотите распространять пользовательский виджет независимо от приложения);
- Flutter Module (позволяет встраивать приложение Flutter в нативное приложение Android).

Выберите **Flutter Application** и нажмите кнопку **Next** (Далее). Откроется окно, как показано на рис. 1-3.

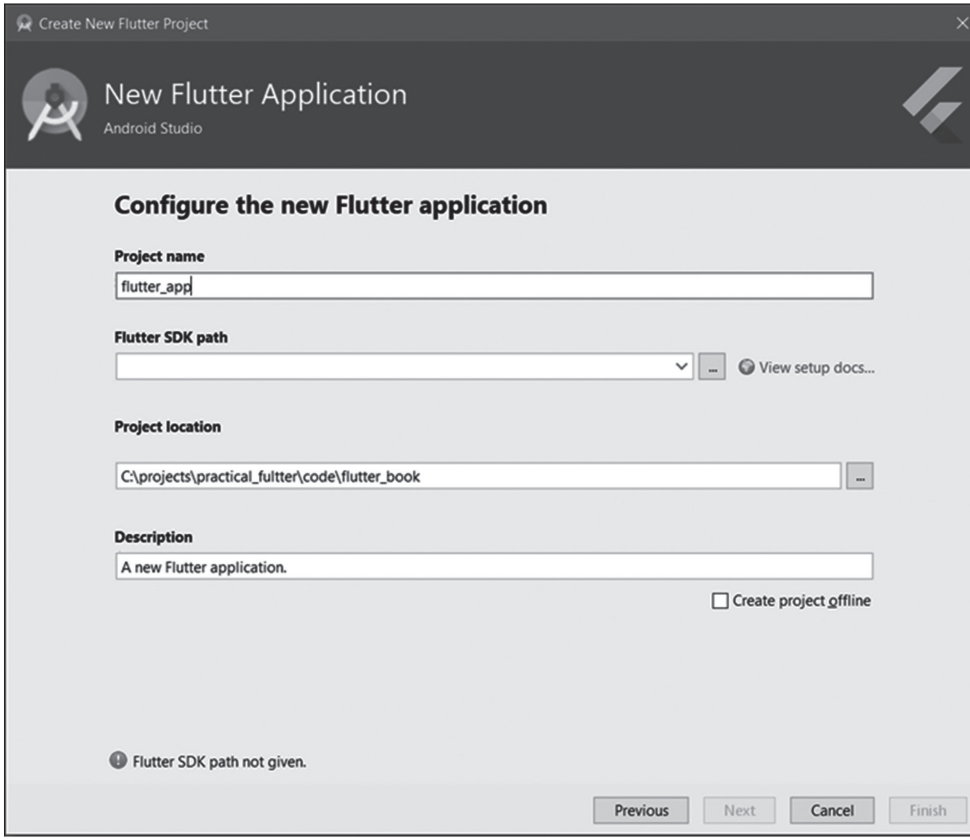


Рисунок 1-3. *Ввод необходимой информации о вашем приложении*

Здесь вы введете информацию о создаваемом приложении. Вы можете дать проекту любое название или описание, а также оставить значения по умолчанию. При необходимости обновите поле **Project location** (или просто используйте значение по умолчанию). Вы видите эту ошибку внизу? Не пугайтесь, вы уже выбрали нужный путь. Но если вы видите это, то Android Studio еще не знает, где находится Flutter SDK, и вам необходимо его указать. Просто перейдите к SDK, который вы должны были установить ранее, и убедитесь, что Android Studio этим доволен (ошибка исчезнет), и снова нажмите кнопку **Next**, чтобы перейти к экрану с рис. 1-4.

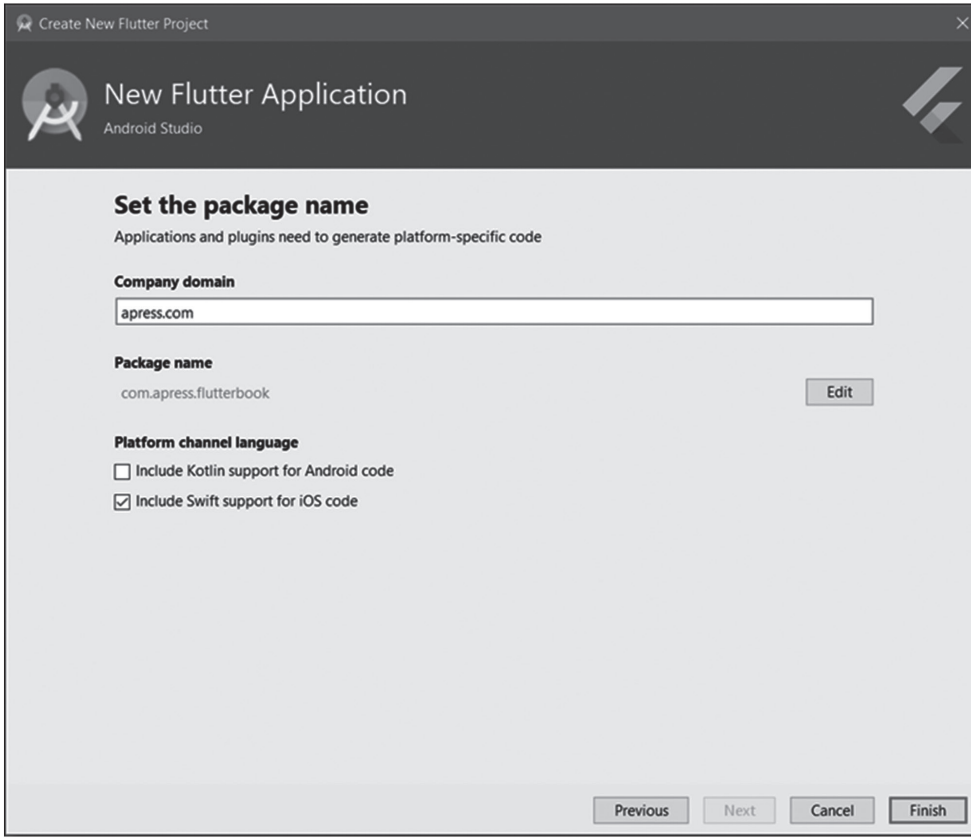


Рисунок 1-4. *Окончательная информация о проекте*

Этот экран требует немного больше информации, в первую очередь домен компании. Если у вас нет домена, то вы можете поместить любое значение, которое вам нравится. Тебя зовут Jim? Ты *можешь* ввести «Jim»! Ты можешь написать «Jim», даже если это не твое имя, хотя это было бы немного странно. Обратите внимание, что полное имя пакета состоит из названия приложения и домена вашей компании, введенного на последнем экране. Это имя пакета должно быть уникальным, если вы хотите опубликовать приложение в магазине приложений, хотя для нашего тестирования здесь это не имеет значения.

Примечание. Вы также можете увидеть поле `Sample Application`, в зависимости от версии Android Studio и установленного плагина Flutter. Это для того, чтобы мастер сгенерировал образец кода для вас, если хотите, но нам это не нужно.

Наконец, выберите язык платформы, оставляя Kotlin неотмеченным, так как Swift будет наиболее подходящим. Это относится к базовому языку платформы, используемому под обертками Flutter, и если вы не собираетесь взаи-

модействовать с нативным кодом приложения, вам, по идее, должно быть без разницы. В конечном счете это не имеет значения для целей книги.

Так что нажмите кнопку **Finish**, и Android Studio покажет вам простое приложение Flutter. Это может занять несколько минут, поэтому проверьте строку состояния внизу, чтобы убедиться, что все задачи выполнены. Когда всё будет готово, посмотрите в верхнюю часть Android Studio, на панель инструментов, и найдите выпадающий элемент, в котором перечислены подключенные устройства, как показано на рис. 1-5.

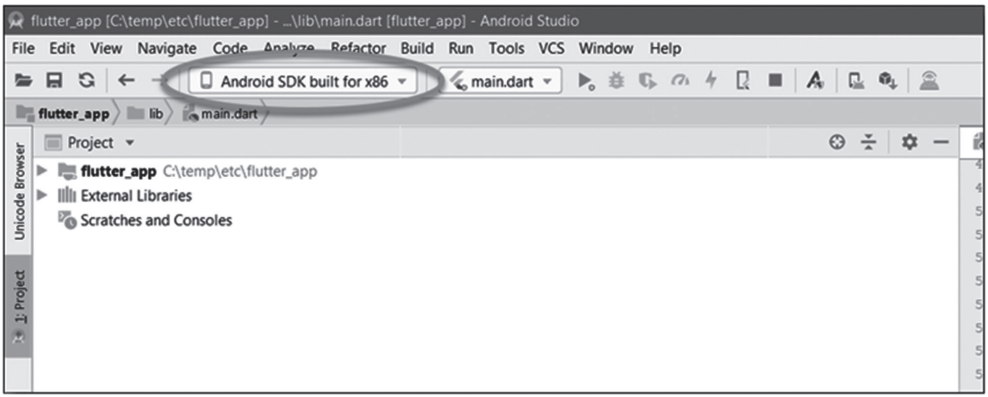


Рисунок 1-5. *Выпадающий список устройств в Android Studio*

Вы должны увидеть созданный ранее эмулятор. Выберите его, и, если он еще не запущен, он должен запуститься в ближайшее время. Как только это произойдет, нажмите на значок **Run** (зеленую стрелку рядом с выпадающим списком `main.dart`) – это точка запуска приложения. Подождите, пока приложение будет собрано, развернуто и запущено на эмуляторе (в зависимости от вашей машины это может занять до минуты, поэтому будьте терпеливы – процесс ускорится после первой сборки). Вы должны увидеть в эмуляторе что-то вроде рис. 1-6.

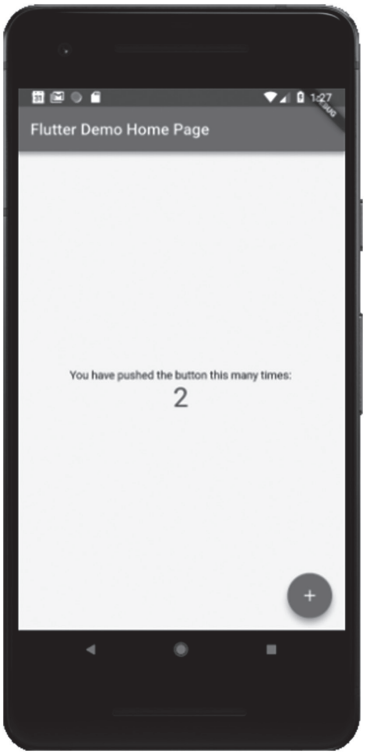


Рисунок 1-6. *Мое первое приложение Flutter!*

Это простое приложение, но оно многое показывает. Нажмите на круглую кнопку со знаком «плюс» (она называется «плавающей кнопкой действия», или Floating Action Button, FAB) и обратите внимание, что счетчик увеличивается с каждым щелчком мыши.

Получившийся код должен автоматически открываться в Android Studio (если файл `main.dart` найден в директории верхнего уровня) и быть следующим (я, конечно, удалил комментарии и отформатировал его, чтобы он лучше выглядел в печатном варианте):

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext context) {
      return MaterialApp(
         title: 'Flutter Demo',
         theme: ThemeData(
            primarySwatch: Colors.blue,
```

```
            ),
            home: MyHomePage(title: 'Flutter Demo Home Page'),
        );
    }
}

class MyHomePage extends StatefulWidget {

    MyHomePage({Key key, this.title}) : super(key: key);

    final String title;

    @override
    _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

    int _counter = 0;

    void _incrementCounter() {
        setState(() {
        _counter++;
    });
}

@override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text(widget.title),
        ),
        body: Center(
            child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                    Text(
                        'You have pushed the button this many times:',
                    ),
                    Text(
                        '$_counter',
                        style: Theme.of(context).textTheme.display1,
                    ),
                ],
            ),
        ),
        floatingActionButton: FloatingActionButton(
            onPressed: _incrementCounter,
            tooltip: 'Increment',
```

```
            child: Icon(Icons.add),
          ),
        );
    }
}
```

Хотя здесь не так много кода, зато он много делает. Думаю, пока вам недостаточно знаний, чтобы полностью разобраться в происходящем, ведь мы еще не говорили о Dart настолько подробно. Но я не хочу оставлять вас абсолютно в неведении, так что есть несколько ключевых моментов, которые я разъясню.

Во-первых, обратите внимание, что основная точка входа каждого приложения Flutter является методом `main()`. В `main()` вызывается метод `runApp()`, который возвращает виджет верхнего уровня. В начале иерархии всегда есть виджет, который содержит все остальные, здесь это экземпляр класса `MyApp`. Этот класс является виджетом без состояния, поэтому единственная его задача – предоставить метод `build()`. Виджет, возвращаемый из него (помните: `build()` – всегда возвращающий виджет, в котором могут быть дочерние элементы, а могут и не быть), – это экземпляр `MaterialApp`, который является виджетом, предоставленным Flutter (это можно увидеть в самом начале нашего кода). Мы поговорим об этом виджете в главе 3, когда будем подробнее рассматривать виджеты Flutter, но главное, что он обеспечивает базовую инфраструктуру для приложения Material. Как видите, мы задали название в одном из аргументов конструктора `MaterialApp`, это название вы увидите в строке состояния (Status Bar) вашего приложения. Также вы можете установить тему для приложения Flutter и предоставить подробную информацию о ней, например основной цвет, который она использует, у нас он синий.

Наш виджет `MaterialApp` содержит один дочерний элемент, который представлен экземпляром класса `MyHomePage`.

Класс `MyHomePage` определяет виджет с состоянием, так что нам понадобится два класса, класс «core», который наследуется от `StatefulWidget`, и класс состояния, связанный с ним, наследуется от `State`.

В любом случае, метод `build()` этого виджета снова возвращает единственный виджет, на этот раз `Scaffold`. Но не зацикливайтесь на этом, потому что в главе 3 мы разберем каждый из них. Если в двух словах, то `Scaffold` обеспечивает фундаментальный визуальный макет для приложения, включая такие элементы, как строка состояния (`AppBar` – это фактически виджет), где находится название. `Scaffold` также обеспечивает механизмы, позволяющие «зацепить» FAB, – экземпляр виджета `FloatingActionButton` передается в конструктор `Scaffold`.

Другой аргумент, переданный в конструктор `Scaffold`, – это тело, в которое мы добавляем другие виджеты в качестве дочерних. Здесь вы можете наблюдать мантру «все это виджет» в действии, потому что у нас есть центральный виджет, который является контейнерным виджетом, который – как вы уже догадались – центрирует свой дочерний элемент. В этом случае дочерним является виджет `Column`, который размещает уже своих детей в одну колонну.

`Column` содержит два дочерних виджета `Text` – один для статического текста «You have pushed the button this many times:», а другой для отображения количества нажатий кнопки.

Все станет понятнее, когда мы в течение следующих двух глав углубимся в Dart, а затем во Flutter. И хотя я опустил много деталей, мне кажется, этого объяснения вполне достаточно для достойного представления о том, что происходит в коде.

Горячая перезагрузка: вот что я люблю!

Здесь все становится невероятно крутым! Убедитесь, что у вас есть приложение, работающее в эмуляторе, а затем перейдите к Android Studio и найдите эту строку кода:

```
Text(
   'You have pushed the button this many times:',
),
```

Итак, вы можете изменить данный текст, поменяв «button» на «FAB» и зажав **Ctrl+S**, или выберите **Save All** в меню **File**. Теперь наблюдайте за эмулятором, и почти сразу вы увидите, что ваше изменение отражается на экране (это может занять несколько секунд, но всё же быстрее, чем при первом запуске).

Очень круто, не правда ли?

Горячая перезагрузка работает только в режиме отладки, в котором вы находитесь; это можно понять благодаря отладочному баннеру в правом верхнем углу приложения. В этом режиме ваше приложение фактически работает в виртуальной машине Dart (VM), а не компилируется в собственный код процессора, что происходит, когда вы создаете реальное приложение (да, ваше приложение будет работать медленнее в режиме отладки). Горячая перезагрузка работает путем обновления измененных файлов исходного кода в уже работающую виртуальную машину Dart, на которой размещается ваше приложение. Когда это происходит, VM обновляет классы, которые изменились, обновляя любые измененные поля и методы. Затем структура Flutter инициирует перестроение дерева виджетов, и ваши изменения отражаются автоматически. Вам не нужно пересобирать или перезапускать что-либо; все происходит автоматически по мере необходимости, чтобы ваши изменения были отображены на экране как можно быстрее.

Время от времени вы можете обнаружить, что изменение не приводит к перезагрузке, как ожидалось. Если это произойдет, первое, что нужно попробовать, – это нажать на значок горячей перезагрузки на панели инструментов, который на рис. 1-7 выглядит как молния (вы также можете найти опцию горячей перезагрузки в меню **Run** с соответствующей горячей клавишей **Ctrl+/**).

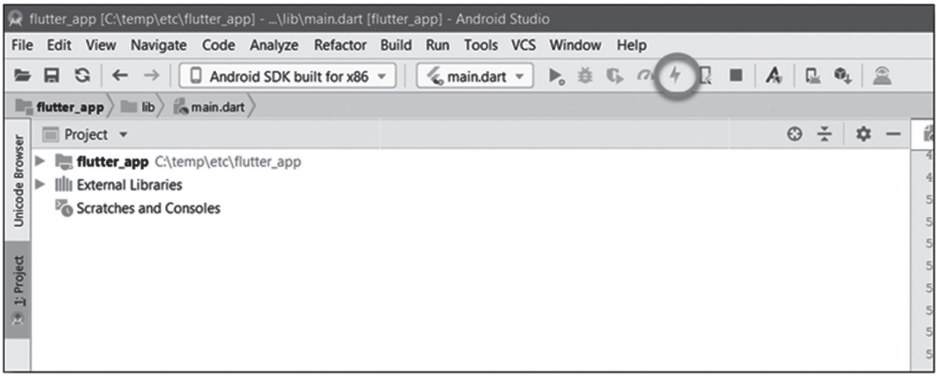


Рисунок 1-7. *Значок горячей перезагрузки в Android Studio*

Это должно помочь вам. Также обратите внимание на консоль, которая должна находиться в нижней части Android Studio, там вы увидите следующее сообщение:

```
Performing hot reload...
Reloaded 1 of 448 libraries in 2,777ms.
```

Кроме того, небольшая подсказка должна появиться рядом с консолью во время перезагрузки.

Обратите внимание, если вы нажали на **FAB** несколько раз, а затем изменили текст, текущее состояние приложения продолжит существование. Другими словами, количество повторных нажатий на кнопку сохранится после горячей перезагрузки (hot reload). Это позволяет легко изменять пользовательский интерфейс и мгновенно отображать текущее состояние, так что вы можете быстро и просто сверять вашу верстку с дизайном. Но что, если вы хотите, чтобы состояние не сохранялось? Тогда вы, вероятно, захотите выполнить горячий перезапуск (restart). Поэтому вам следует сделать это вручную (в отличие от горячей перезагрузки, которая происходит автоматически, когда вы вносите изменения в код и сохраняете его), выбрав опцию **Hot restart** (горячий перезапуск – не перезагрузка) в меню **Run** или нажав соответствующую горячую клавишу (**Ctrl+Shift+/**).

Интересно, что значка для горячего перезапуска на панели инструментов нет, но это всё же перезапустит ваше приложение, хоть и не выполнит новую сборку и очистку состояния.

Вы, естественно, можете перезапускать сборку в любое время (это происходит по команде **Run**), но каждый раз вы будете дожидаться компиляции, что ни капельки не экономит ваше время. Горячий перезапуск (restart) сработает почти так же быстро, как и горячая перезагрузка (reload), потому что работы он делает намного меньше, но достигает примерно того же эффекта.

Надеюсь, вы видите, насколько вы можете быть эффективны, используя горячую перезагрузку. Мне кажется, вы оцените это, когда познакомитесь с Flutter поближе!

Базовая структура приложения Flutter

Одна из последних тем, которой я коснусь во вступительной главе, – это общая структура приложения, которое было создано для вас. На рис. 1-8 вы видите первичную структуру каталога.

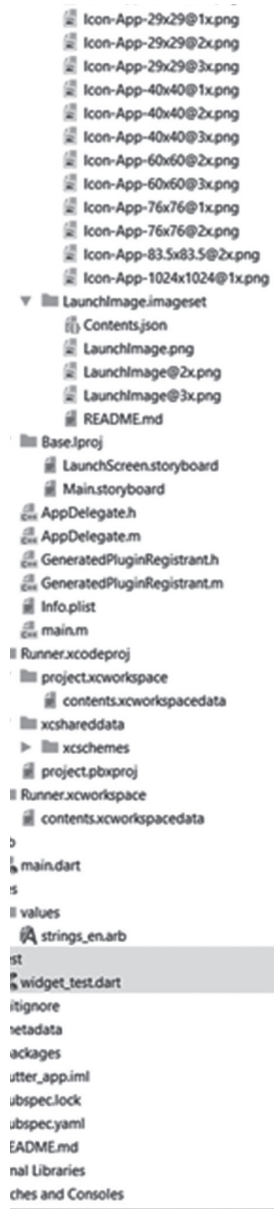


Рисунок 1-8. *Первичная структура каталога проекта*

Как видите, существует пять каталогов верхнего уровня. А теперь подробнее о каждом из них:

- `android` – содержит специфический для Android код и ресурсы, такие как значки приложений, код Java, а также конфигурацию Gradle и ресурсы (Gradle является системой сборки Android). На самом деле это фактически целый проект Android, который вы можете построить с использованием стандартных инструментов Android. По большому счету, вам нужно изменять только значки (которые находятся в каталогах `android/app/src/main/res`, где каждый подкаталог имеет разное разрешение) и, в зависимости от того, что делает ваше приложение, файл `AndroidManifest.xml` в `android/app/src/main`, где вы можете установить специальные свойства приложения для Android;

- `ios` – как и android, этот каталог содержит код проекта, специфичный для iOS. Критическим содержимым здесь является каталог `ios/Runner/Assets.xcassets`, в котором находятся значки для вашего приложения, и файл `Info.plist` в `ios/Runner`, который служит той же цели, что и файл `AndroidManifest.xml` для приложений Android;

- `lib` – хотя сначала это может показаться странным, это место, где будет жить ваш код! Вы можете относительно свободно организовывать свой код, создавая любую структуру каталогов, хотя вам понадобится один файл, который служит точкой входа, и большую часть времени им будет main.dart, созданный автоматически;

- `res` – этот каталог содержит такие ресурсы, как строки для перевода вашего приложения на иностранные языки. Но в данной книге мы не будем иметь с ними дело;

- `test` – здесь вы найдете Dart-файлы для тестирования вашего приложения. Flutter предоставляет утилиту Widget Tester, которая использует автоматические тесты, чтобы подтвердить функциональность ваших виджетов. Мы не будем с ним работать, так же как и с `res`, потому что это необязательная часть разработки Flutter, о которой отдельно можно написать целую книгу! Тестирование – это, конечно, важно, но пока вы не научитесь писать приложения Flutter, вам будет нечего тестировать, а эта книга фокусируется на первой части вашего пути.

Хоть он и скрыт по умолчанию в Android Studio, есть также каталог .idea, в котором хранится информация о конфигурации Android Studio, поэтому вы можете его игнорировать (обратите внимание, что Android Studio основана на IDE IntelliJ IDEA, отсюда и название). Существует также скрытый каталог сборки, содержащий информацию, которую используют Android Studio и Flutter SDK для создания вашего приложения. Но мы проигнорируем и это.

Помимо каталогов, вы также найдете некоторые файлы в корне проекта. Это, как правило, единственные файлы, о которых вам нужно беспокоиться за

пределами каталога lib (все остальное на скриншоте вам не нужно знать вообще), и это:

- `.gitignore` – файл управления версиями Git используется, чтобы знать, какие файлы игнорировать из управления версиями. Использование Git совершенно необязательно при написании приложений Flutter, но этот файл генерируется в любом случае. Управление версиями – это то, что не сможет охватить даже целая книга, поэтому вы можете игнорировать данный файл;
- `.metadata` – данные, которые Android Studio отслеживает в вашем проекте. Вы можете игнорировать и это, так как вы никогда не будете редактировать их самостоятельно;
- `.packages` – у Flutter есть свой собственный менеджер пакетов для управления зависимостями в вашем проекте. Этот менеджер пакетов называется Pub, и он используется для отслеживания зависимостей в вашем проекте. Вы не будете взаимодействовать с ними напрямую или даже напрямую с Pub, поэтому его тоже можно оставить без внимания;
- `*.iml` – этот файл должен быть назван в честь вашего проекта и является файлом конфигурации проекта Android Studio. Вы никогда не будете редактировать его напрямую, так что игнорируем;
- `pubspec.lock` и `pubspec.yaml` – вы когда-нибудь работали с NPM? Знакомы с package.json и файлами package-lock.json, которые он использует? Ну, это те же вещи, но для Pub! Если вы незнакомы с NPM, `pubspec.yaml` – это то, как вы описываете свой проект для Pub, включая его зависимости. Файл `pubspec.lock` – это внутренний файл Pub. Вы определенно можете редактировать `pubspec.yaml`, но не `pubspec.lock`, а pubspec.yaml мы позже подробно рассмотрим;
- `README.md` – файл readme, который вы можете использовать, как хотите. Как правило, это файл Markdown – сайты, такие как GitHub, используют для отображения информации о вашем проекте при переходе к репозиторию, где этот файл находится в корневом каталоге.

Самым важным файлом здесь является pubspec.yaml, и он один из немногих, которые вам нужно будет редактировать, поэтому, если вы все забыли, его лучше запомните! Мы доберемся до него позже, когда нам нужно будет добавить зависимости в наш проект, но на данный момент сгенерированного файла вполне достаточно для наших нужд.

Еще парочка моментов «под прикрытием»

Если вы посмотрите на некоторые из файлов в каталоге *ios*, то заметите слово «Runner». Это подсказка о том, как работают приложения Flutter при сборке и запуске установочного пакета. Как отмечалось ранее, горячая перезагруз-

ка работает, потому что в режиме отладки ваш код запускается в виртуальной машине. Однако при сборке установочного пакета это так не работает. Ваш код компилируется в собственный код ARM. Фактически компилируется в библиотеку для процессора ARM, которую в дальнейшем использует нативное приложение, это объясняет, почему ваш код находится в каталоге lib.

Библиотеки Flutter наряду с вашим кодом приложения компилируются «до запуска» (Ahead-Of-Time, AOT) с помощью LLVM (Low-Level Virtual Machine, инфраструктура компилятора, написанная на C++, которая предназначена для компиляции и оптимизации программ, написанных на различных языках программирования) на iOS. На Android используется Native Development Kit (NDK) для сборки ARM-библиотеки. Эта библиотека включена в так называемый «runner», который является просто нативным приложением, которое... подождите, подождите... запускает ваше приложение. Представьте, что это тонкая обертка вокруг вашего приложения, которая знает, как запустить приложение, и предоставляет необходимые условия. В некотором смысле runner по-прежнему представлен виртуальной машиной, хотя и очень тонкой (почти как контейнер Docker, если вы с ним знакомы).

Наконец, runner вместе со скомпилированной библиотекой упаковывается в файл .ipa для iOS или файл .apk для Android, и у вас есть полный, готовый к установке или публикации пакет! Когда приложение запускается, runner загружает библиотеку Flutter и ваш код приложения, и с этого момента вся визуализация, ввод/вывод и обработка событий делегируются скомпилированному приложению Flutter.

ПРИМЕЧАНИЕ. Это очень похоже на то, как работает большинство кросс-платформенных мобильных игровых движков. Ранее я написал книгу о Corona SDK, библиотеке, которую я очень люблю, она работает очень похожим образом, хотя там используется язык Lua вместо Dart (который, могу поспорить, команда Flutter тоже рассматривала!). Любопытно, что Google, по сути, черпал вдохновение из игровых движков, чтобы создать Flutter, потому что это доказывает то, что я всегда говорил: если вы хотите быть лучшим программистом, единственный вид проекта, в котором вам следует отточить свои навыки, – это игровой. На этот раз мир получил целую платформу приложений! И если вы еще не заглянули вперед, последние две главы этой книги посвящены созданию игры Flutter, потому что я всегда советую создавать игры!

Итого

С этой главой вы начали свое путешествие в мир Flutter! Вы узнали о том, что такое Flutter, что он предлагает и почему вам стоит его использовать (и даже некоторые причины, по которым вы не захотите его использовать). Вы узнали о важных концепциях, таких как Dart и виджеты; узнали, как настроить свою среду разработки, чтобы работать с кодом Flutter; создали свое первое очень простое приложение Flutter и запустили его в эмуляторе.

В следующей главе вы поближе познакомитесь с Dart и получите хорошую базу, чтобы приступить к созданию реальных приложений Flutter!

ГЛАВА 2

МГНОВЕННОЕ РУКОВОДСТВО ПО DART

В предыдущей главе вы получили краткое введение в Dart, язык, который Google выбрал для Flutter. Это была очень короткая справка по общим аспектам Dar, чтобы вам было проще понять базовые примеры кода.

Так как все приложения Flutter написаны на Dart, то вам необходимо усвоить информацию, предоставленную в данной главе. После ее прочтения вы познакомитесь с Dart поближе, по крайней мере достаточно близко, чтобы смело кодить в последующих главах (надеюсь, что знания из этой главы задержатся в вашем мозгу надолго). Мы углубимся в Dart, но не будем забывать и о Flutter, ведь только так сформируется полная картина.

Если честно, это не будет исчерпывающим взглядом на Dart. Я постараюсь заполнить пробелы в последующих главах, когда мы будем изучать код приложения, но некоторые темы либо очень редко используются, либо очень специализированные, и если вы их пропустите, вам это не помешает. Всё, что описано в этой книге, составляет около 95 % информации о Dart. Естественно, в онлайн-документации по Dart (на `www.dartlang.org`) вы найдете дополнительные материалы, охватывающие и подробно раскрывающие все аспекты языка, так что если вы действительно хотите погрузиться в Dart с головой, то после прочтения этой главы я рекомендую ознакомиться и с информацией на сайте.

Давайте начнем наше путешествие с основ и ключевых понятий, которые вы должны знать для лучшего понимания Dart.

Вещи, которые вы должны знать

Как и любой современный язык программирования, Dart предлагает много возможностей, хотя он и основан на тех же концепциях, что и большинство других языков, хотя Dart имеет и свои уникальные особенности.

Но прежде чем мы начнем говорить о концепциях, хотите увидеть кое-что крутое? Взгляните на рис. 2-1. Это веб-приложение известно как DartPad, и оно предоставлено веб-сайтом dartlang.org, а именно `https://dartpad.dartlang.org`.

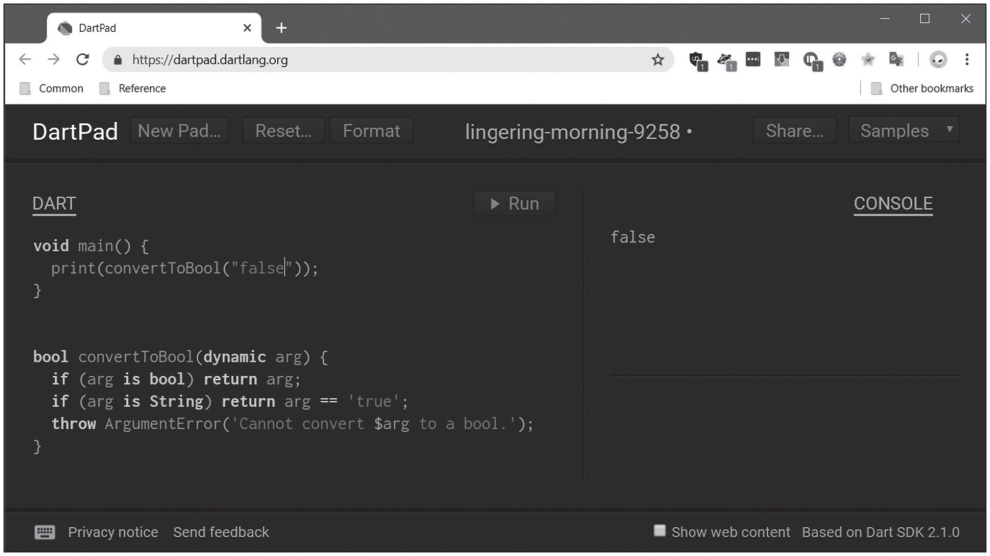


Рисунок 2-1. *FlutterPad – ваша экспериментальная площадка для кода Dart в интернете!*

Этот лаконичный инструмент позволяет проверить большинство возможностей Dart в режиме реального времени без необходимости установки каких-либо приложений! Это отличный способ быстро и легко проверить ваши навыки. Просто введите слева код, нажмите **Run** (Выполнить), и справа вы увидите результаты!

Ну что же, давайте начнем!

Все языки, включая Dart, имеют перечень ключевых слов, которые можно использовать только по прямому назначению. Давайте рассмотрим эти ключевые слова. Я попытался сгруппировать их, чтобы дать вам как можно больше контекста для их изучения.

ПРИМЕЧАНИЕ. Эта книга предполагает, что вы уже не новичок в программировании и что у вас есть опыт работы с C-подобным языком. Это особенно актуально для текущей главы, потому что многие из ключевых слов Dart не отличаются от любого другого C-подобного языка, с которым вы знакомы. Я приложу только очень краткие описания и оставлю более подробные объяснения для тех ключевых слов и понятий, которые уникальны для Dart или, если не уникальны, по крайней мере, немного неординарны.

Все о комментариях – без лишних комментариев

Я хочу начать с обсуждения, потому что чувствую, что с комментариями разработчики работали и работают недостаточно эффективно. Комментарии – это важная часть программирования, нравится вам это или нет, и Dart предлагает три формы комментариев.

Во-первых, Dart поддерживает однострочные комментарии, используя знакомую вам комбинацию символов //. Компилятор игнорирует все символы в строке, которые следуют за двумя слешами. То есть два слеша могут находиться как в начале, так и в конце строки:

```
// Определяем возраст пользователя.
int age = 25; // Возраст 25
```

Не поймите меня неправильно: я не утверждаю, что это пример хорошего или правильного комментирования! На самом деле наоборот! Я просто использую его в качестве примера, чтобы продемонстрировать эту форму комментария в Dart.

Вторая форма – многострочные комментарии, Dart здесь также типичен и использует открывающую последовательность символов /* и закрывающую */. Например:

```
/*
   Эта функция вычисляет баланс счета с использованием метода
   расчета стоимости Миллера-Хоторна.
*/
```

Все между последовательностями /* и */ игнорируется.

Последняя форма комментариев, предоставляемых Dart, называется «комментарии документации» (comments for documentation). Эти комментарии предназначены для автоматической генерации документации на основе исходных кодов. Они могут быть однострочными или многострочными за счет последовательностей /// или /** и */:

```
/// Это комментарий документации.
/**
   Это тоже
   комментарий документации.
*/
```

Как и в случае с другими форматами, все в строке, начинающейся с последовательности /// (может быть как в начале, так и в конце строки), игнорируется. Однако есть исключение: все, заключенное в таком комментарии в скобки, воспринимается как ссылка на класс, метод, поле, переменную верхнего уровня, функцию или параметр и попадает в документацию по родительскому элементу. Так, например:

```
class Pet {
   num legs;
   /// Накормите вашего питомца [Treats].
   feed(Treats treat) {
      // Покормите зверюгу!
   }
}
```

При генерации документация (вы можете сделать это с помощью инструмента dartdoc из Dart SDK) текст «[Treats]» станет ссылкой на документацию для класса Treats (при условии что dartdoc может найти Treats в области видимости класса Pet).

Совет. Пожалуйста, комментируйте свой код и комментируйте его хорошо, особенно если с ним предстоит работать другим разработчикам (поверьте мне, даже если вы думаете, что такого не произойдет, – хорошо написанный комментарий к коду, в который вы сами не заглядывали годами, будет настоящей находкой). В мире программирования существует вечная дискуссия о комментировании. Некоторые разработчики вообще не хотят писать какие-либо комментарии (это лагерь «самодокументирующегося кода»), другие просто хотят, чтобы люди писали полезные комментарии. Я уже давно принадлежу к последнему лагерю. Для меня комментарии так же важны, как и код. Я пришел к этому выводу после 25 лет профессиональной разработки софта, поскольку понял, что поддержание кода других людей и даже моего собственного – это огромная проблема. Да, попробуйте написать «самодокументируемый» код, поверьте, это очень полезно. Но потом, как только вы это сделаете, прокомментируйте его!

Конечно, если вы говорите в своем комментарии, что код «i++;» увеличивает i на единицу, то это пустая потеря времени. Поэтому пишите полезные комментарии.

Все меняется: переменные

Начнем с того, что все в Dart является объектом. Переменные в Dart, как и в любом другом языке, хранят значение или ссылку на объект. В некоторых языках существует разница между примитивами, такими как числа (int, double и т. д.) и строки (string), и объектами, которые являются экземплярами классов. Но только не в Dart! Здесь объектом является все, даже целые числа, функции и сам null – все это объекты, экземпляры классов, и все они происходят от общего класса Object.

Объявление и инициализация переменной

В Dart вы можете объявить переменную двумя способами:

```
var x;
```

или

```
<some specific type> x;
```

Обратите внимание, что здесь x имеет значение null, даже если оно имеет числовой тип. Это значение по умолчанию, если вы не определяете переменную на этапе создания:

```
var x = "Mel Brooks";
String x = "Mel Brooks";
```

Вы видите кое-что интересное: когда вы используете var x, Dart сам определяет тип переменной из присвоенного значения. Он знает, что x здесь – это ссылка на String. Но вы также можете объявить тип явно, как в String x.

Существует руководство по стилю написания программ в Dart, которое гласит, что вы должны объявлять локальные переменные, такие как var и другие, используя явное указание типа (например, String x = "Mel Brooks"), но это вопрос ваших предпочтений.

Также есть третий вариант:

```
dynamic x = "Mel Brooks";
```

Здесь тип dynamic («динамический») говорит Dart о том, что переменная x может меняться со временем. Например, если позже вы напишете

```
x = 42;
```

Dart не будет жаловаться, что x теперь указывает на числовое значение, а не на строку.

Существует, по сути, четвертый и последний вариант объявления переменной:

```
Object x = "Mel Brooks";
```

Поскольку все в Dart происходит из общего класса Object, то и подобное объявление корректно работает, так как присваиваемая строка (String) является потомком класса Object.

Константы и конечные значения

Наконец, со всем этим связаны ключевые слова const и final, которые определяют переменную как константу (const) или конечное (неизменяемое) значение (final):

```
const x = "Mel Brooks";
```

Они также работают с аннотациями типа:

```
const String x = "Mel Brooks";
```

Вместо этого вы можете использовать final:

```
final x = "Mel Brooks";
```

И это не просто вопрос предпочтений. Разница в том, что переменные const являются постоянными во время компиляции, так что их значение не может присваиваться во время работы приложения. Итак, если вы попытаетесь написать

```
const x = DateTime.now();
```

то это не сработает. Но вы можете сделать так:

```
final x = DateTime.now();
```

По сути, final означает, что вы можете установить значение переменной только один раз, зато сделать это во время выполнения вашей программы,

а const означает, что вы можете установить значение только один раз, но оно должно быть явно задано уже в исходных кодах (до компиляции).

И вот еще кое-что о const: вы можете применить его как к значениям, так и к переменной. Например (и не беспокойтесь о том, что незнакомы с типом List, мы скоро до него доберемся – но я уверен, что вы все равно все поймете!):

```
List lst = const [ 1, 2, 3];
print(lst);
lst = [ 4, 5, 6 ];
print(lst);
lst[0] = 999;
print(lst);
```

Все работает так, как ожидалось: печатается первоначальный список значений (1, 2, 3), затем появляется ссылка и печатается новый список (4, 5, 6), и, наконец, обновляется первый элемент, и список печатается снова (999, 5, 6). Однако если вы переместите lst[0] = 999 на третью строку, то вы получите исключение (exception), потому что пытаетесь изменить список, который был помечен как const. Это одна из специфических особенностей Dart (я уверен, что и другие языки сталкиваются с подобным, хотя такая ситуация и не очень распространена).

Примечание. Переменные и другие идентификаторы могут начинаться с буквы или подчеркивания, а затем сопровождаться любой комбинацией букв и цифр (и, конечно, столько комбинаций, сколько вы захотите!). Все, что начинается с подчеркивания, имеет особое значение: оно является закрытым (private) для библиотеки (или класса), в котором находится, то есть не видно внешним классам. Dart не имеет ключевых слов видимости, таких как public и private, которые можно найти в других языках, например Java. Переменные, у которых названия начинаются с подчеркивания, автоматически считаются закрытыми (private).

Ну он и тип... типы данных

Dart является строго типизированным языком программирования, но, как ни странно, вам не обязательно задавать эти типы в коде. Они необязательны, поскольку Dart определяет типы автоматически, когда вы не указываете их вручную.

Строковые значения

Для работы со строками в Dart предлагается использовать тип String, который представляет собой последовательность символов в кодировке UTF-16. Для присваивания переменной строки должны быть заключены в одиночные или двойные кавычки. Они могут включать выражения, использующие синтаксис ${expression}. Если выражение ссылается на идентификатор, то вы можете удалить фигурные скобки. Итак:

```
String s1 = "Rickety Rocket";
String s2 = "${s1} blast off!";
String s3 = '$s1 blast off!';
print (s2);
print (s3);
```

Здесь вы видите двойные и одинарные кавычки, а также обе формы выражений (иногда называемые маркерами).

Для склеивания строк из частей вы можете использовать оператор +, как и во многих других языках. Также можно использовать следующий синтаксис:

```
return "Skywalker," "Luke";
```

Все строковые значения, конечно, могут содержать ключевые слова или любые последовательности символов в кодировке UTF-16.

Числовые значения

По старой доброй традиции, обычные целочисленные значения имеют тип int. Значения int в Dart VM лежат в диапазоне от -2^{63} до $2^{63} - 1$ (вне зависимости от того, под какую платформу компилируется Dart-приложение – для чисел типа int всегда выделяется 64 бита).

Число с плавающей запятой двойной точности, как указано в стандарте IEEE 754, имеет тип double.

int и double – это подклассы num, поэтому вы можете определить переменную как `num w = 5` или `num x = 5.5`, а также `int y = 5` или `double z = 5.5`. Dart знает, что x является double на основе его значения, так же он знает и про z, тип для которого вы указали вручную.

Числовое значение может быть преобразовано в строковое с помощью метода toString() класса int и double:

```
int i = 5;
double d = 5.5;
String si = i.toString();
String sd = d.toString();
print(i);
print(d);
print(si);
print(sd);
```

А строковое может быть преобразовано в числовое с помощью метода `parse()` для классов `int` и `double`:

```
String si = "5";
String sd = "5.5"; int i = int.parse(si);
double d = double.parse(sd);
print(si);
```

```
print(sd);
print(i);
print(d);
```

Примечание. Обратите внимание, что `String` – единственный примитивный тип данных, название которого начинается с заглавной буквы. Я считаю, что это стоит пояснить. На самом деле это не совсем верно: названия классов `Map` и `List` также начинаются с заглавных букв. Тем не менее я не уверен, что они должны быть в той же категории, что и `String`, учитывая, что `String` является более «внутренним» типом данных, таким как `int` и `double`. Но мы можем обсудить это в другой раз – просто усвойте, что названия одних типов данных начинаются с заглавной буквы, а других – нет!

Логические значения

Логические переменные имеют тип `bool` и могут принимать только значения истина/ложь (ключевые слова `true` и `false` соответственно).

Обратите внимание, что безопасность типов Dart означает, что вы не можете написать код, как в языке C:

```
if (some_non_boolean_variable)
```

Вместо этого вы должны написать что-то вроде:

```
if (some_non_boolean_variable.someMethod())
```

Другими словами, логические операции Dart всегда требуют явного значения `true/false` при работе с переменными типа bool. [В языке C, например, значение `false` эквивалентно числу 0, а любая другая цифра – true. Это позволяет использовать целые числа, и не только их, вместо `true/false`. – *Прим. перев.*]

Классы List и Map

Класс `List` в Dart сродни массиву (`array`) в других языках. `List` (список) представляет собой список значений, создаваемый в синтаксисе JavaScript:

```
List lst = [ 1, 2, 3 ];
```

Примечание. Как правило, вы видите названия list (а затем set и map) с маленькой буквы при обращении к конкретному экземпляру класса, а при обращении к самим классам Map, Set или List вы должны писать их с большой буквы.

Вы, конечно, можете сделать это следующим образом:

```
var lst1 = [ 1, 2, 3 ];
Object lst2 = [ 1, 2, 3 ];
```

Списки индексируются с нуля, поэтому `list.length-1` дает вам индекс последнего элемента. Вы можете получить доступ к его элементам по индексу:

```
print (lst[1]);
```

List имеет набор доступных методов. Я не собираюсь обсуждать их все, так как эта глава не справочное руководство, тем более большинство из них можно найти практически в любом другом языке, который предлагает похожую на List конструкцию. Вы, вероятно, и сами уже знакомы с большинством из них, я же приведу несколько примеров методов List:

```
List lst = [ 8, 3, 12 ];
lst.add(4);
lst.sort((a, b) => a.compareTo(b));
lst.removeLast();
print(lst.indexOf(4));
print(lst);
```

Dart также предлагает класс Set (множество), который похож на List, но это неупорядоченный список, а значит, вы не можете получить элементы по индексу, вы должны использовать методы contains() и containsAll() вместо прямого обращения:

```
Set cookies = Set();
cookies.addAll([ "oatmeal", "chocolate", "rainbow" ]);
cookies.add("oatmeal"); // Это не вызовет ошибку
cookies.remove("chocolate");
print(cookies);
print(cookies.contains("oatmeal"));
print(cookies.containsAll([ "chocolate", "rainbow" ]));
```

В нашем примере вызов contains() возвращает true, в то время как вызов containsAll() возвращает false, так как chocolate был удален методом remove(). Обратите внимание, что добавление существующего значения в список типа Set с помощью метода add() не вызывает ошибку.

В Dart также есть класс Map (карта/отображение), который иногда называют *dictionary* (словарь) или *hash* (хеш-таблица), или *object literal* в JavaScript, экземпляр которого может быть создан несколькими способами:

```
var actors = {
   "Ryan Reynolds" : "Deadpool",
   "Hugh Jackman" : "Wolverine"
};
print(actors);
```

```
var actresses = Map();
actresses["scarlett johansson"] = "Black Widow";
actresses["Zoe Saldana"] = "Gamora";
print (actresses);
```

```
var movies = Map<String, int>();
movies["Iron Man"] = 3;
movies["Thor"] = 3;
```

```
print(movies);

print(actors["Ryan Reynolds"]);
print(actresses["Elizabeth Olsen"]);
movies.remove("Thor");
print(movies);
print(actors.keys);
print(actresses.values);

Map sequels = { };
print(sequels.isEmpty);
sequels["The Winter Soldier"] = 2;
sequels["Civil War"] = 3;
sequels.forEach((k, v) {
   print(k + " sequel #" + v.toString());
});
```

Первый вариант отображения с актерами создается с помощью фигурных скобок с данными, определенными непосредственно внутри них. Map с актрисами использует ключевое слово new для создания нового экземпляра отображения. Здесь элементы добавляются таким образом, что содержимое внутри скобок – это ключ, а то, что после знака равно, – значение. Теперь это неразрывная связка ключ–значение. Третья версия показывает, что вы сами можете определить типы для ключей и их значений. Таким образом, если вы попытаетесь сделать:

```
Movies[3] = "Iron Man";
```

то получите ошибку компиляции, потому что 3 – это int, но изначально тип ключа определен как String.

Далее мы рассмотрим часто используемые методы. Метод remove() удаляет элемент из map. Вы можете получить список ключей (keys) или значений (values) с помощью обращения к одноименным свойствам keys и values класса Map (на самом деле будут вызываться специальные методы геттеры (getter), о которых мы поговорим позже). Метод isEmpty() сообщает вам, является отображение (Map) пустым или нет (есть также метод isNotEmpty(), если вы предпочитаете его). Map также предоставляет методы contains() и containsAll(), как это делает List. Наконец, метод forEach() позволяет выполнить произвольную функцию для каждого элемента отображения (это еще не все возможности, но пока не беспокойтесь о деталях).

В классе Map, как и в List, доступно гораздо больше служебных методов, чем мы можем здесь рассмотреть, вскоре мы встретимся с некоторыми из них в исходном коде примеров.

Существует также специальный тип dynamic (динамический), который, по сути, обходит систему безопасности типов Dart. Представьте, что вы пишете:

```
Object obj = some_object;
```

Dart знает, что для любого класса можно вызвать методы toString() и hashCode(), так как эти методы описаны в Object, а любой другой класс наследуется от него. Если вы попытаетесь вызвать obj.fakeMethod(), то получите предупреждение, потому что Dart во время компиляции определяет, что fakeMethod() не является методом класса Object. Но если вы напишете

```
dynamic obj = some_object;
```

и затем вызовите obj.fakeMethod(), то не получите предупреждение во время компиляции, зато получите ошибку во время выполнения. Подумайте о dynamic как о способе сказать Dart: «Эй, я здесь главный, поверь мне, я знаю, что делаю!» Динамический тип обычно используется с такими вещами, как данные, передаваемые между процессами, поэтому вы можете нечасто с этим сталкиваться, но стоит понимать, что он принципиально отличается от типа Object.

Перечисления – если одного значения мало!

Вам нужен объект, который содержит фиксированное количество постоянных значений? Не хотите иметь кучу переменных, захламляющих все вокруг, и вам не нужен полноценный класс? Тогда enum (сокращенно от *enumeration*, перечисление) вам подойдет! Смотрите! Вот он!

```
enum SciFiShows { Babylon_5, Stargate_SG1, Star_Trek };
```

И вот что вы можете с ним сделать:

```
main() {
assert(SciFiShows.Babylon_5.index == 0);
assert(SciFiShows.Stargate_SG1.index == 1);
assert(SciFiShows.Star_Trek.index == 2);
print(SciFiShows.values);
print(SciFiShows.Stargate_SG1.index);
var show = SciFiShows.Babylon_5;
switch (show) {
      case SciFiShows.Babylon_5: print("B5"); break;
      case SciFiShows.Stargate_SG1: print("SG1"); break;
      case SciFiShows.Star_Trek: print("ST"); break;
   }
}
```

Каждое значение в enum имеет скрытый индекс (свойство index), поэтому вы всегда можете найти порядковый номер заданного элемента (вы получите ошибку компиляции, если значение в enum отсутствует). Еще вы можете получить список всех значений enum через свойство values (которое также имеет скрытый метод getter). Наконец, перечисления особенно полезны в операторах switch, и Dart выдаст вам ошибку компиляции, если у вас нет подходящего значения в enum.

А ты его точно знаешь? Ключевые слова «as» и «is»

Эти два оператора часто используются вместе: ключевое слово `is` позволяет вам определить, относится ли объект к конкретному типу; `as` приводит объект к определенному типу. Например:

```
if (shape is Circle){
   print(shape.circumference);
}
```

Метод `print()` (который записывает контент в консоль) выведет значение поля circumference, только если объект shape является Circle.

Или же попробуйте сделать следующее:

```
(shape as Circle).circumference = 20;
```

Таким образом, если переменная shape имеет значение типа `Circle` или может быть приведена к `Circle`, то у вас не возникнет никаких проблем (например, shape имеет значение типа `Oval`, который является подклассом `Circle`).

Обратите внимание, что в примере с `is` ничего не произойдет, если фигура не является `Circle`, а в примере с `as` вы получите исключение (exception), если фигура не может быть приведена к `Circle`.

Плыть по течению: управление логикой потока команд

Для выполнения логических операций Dart включает ряд выражений и конструкций, большинство из которых будут знакомы почти всем программистам.

Циклы

Цикл в Dart очень похож на циклы в других языках:

```
for (var i = 0; i < 10; i++) {
   print(i);
}
```

Существует также выражение `for-in`, если целевой класс реализует итератор, позволяющий переходить к следующему элементу списка:

```
List starfleet = [ "1701", "1234", "1017", "2610", "7410" ];
main() {
   for (var shipNum in starfleet) {
      print("NCC-" + shipNum);
   }
}
```

`List` – один из классов, который реализует итератор, поэтому этот код отлично работает. Если вы предпочитаете функциональный стиль, то можете использовать метод `forEach` следующим образом:

```
main() {
   starfleet.forEach((shipNum) => print("NCC-" + shipNum));
}
```

Примечание. Не беспокойтесь об этих функциях, особенно если синтаксис выглядит немного чуждо. Мы перейдем к функциям всего через несколько разделов, это достаточно просто, если сосредоточиться на изучении.

Циклы do и while предлагают знакомые конструкции do-while и while-do:

```
while (!isDone()) {
   // Делать что-нибудь
}
do {
   showStatus();
} while (!processDone());
```

Обратите внимание, что, как и в большинстве других языков, в Dart доступно ключевое слово continue, которое позволяет сразу перейти к следующему шагу цикла, прервав текущий шаг. Чтобы выйти из цикла досрочно, существует ключевое слово break (оно также работает и в операторе switch).

Switch

Dart предлагает использовать привычный оператор switch для конструкций множественного выбора, которые создаются с помощью ключевых слов case, break и default:

```
switch (someVariable) {
   case 1:
      // Сделайте что-нибудь
      break;
   case 2:
      // Сделайте что-нибудь еще
      break;
   default:
      // Это не первое или второе
   break;
}
```

Оператор switch в Dart может работать с целочисленными или строковыми типами, при этом сравниваемые объекты должны быть одного и того же типа (и никакие подклассы общего предка здесь недопустимы!). Также сравниваемые классы не должны переопределять оператор сравнения ==.

Оператор if

Наконец-то добрались и до оператора if, который по сути является ключевым элементом управления потоком команд в программе. Обратите внимание, что условия оператора if в Dart должны всегда принимать значения типа bool. И конечно же, вы можете использовать ключевое слово else:

```
if (mercury == true || venus == true ||
   earth == true || mars == true
){
   print ("It's an inner planet");
} else if (jupiter || saturn || uranus || neptune) {
   print ("It's an outer planet");
} else {
   print("Poor Pluto, you are NOT a planet");
}
```

Обратите внимание, что если mercury, venus, earth и mars были типами bool, то запись if (mercury || venus || earth || mars) также будет здесь работать.

Больше, чем ничто: void

В большинстве языков, если функция ничего не возвращает, вам нужно поставить перед ней ключевое слово void. В Dart, который поддерживает void, вы тоже *можете* это сделать, но это не обязательно.

Тем не менее в Dart void более… любопытен, чем в других языках.

Во-первых, если функция ничего не возвращает, вы можете полностью опустить тип возвращаемых данных; вам даже не нужно ставить void перед ней, как в большинстве языков (хотя вы можете сделать это, если хотите). В таких случаях в конец функции добавляется неявный возврат null (return null;). Это касается всех примеров кода.

Если вы поставите void перед функцией, то получите ошибку компиляции при попытке что-либо вернуть из нее. Иногда в этом есть смысл. Но если вы попытаетесь вернуть null, ошибки не будет. Вы также можете вернуть функцию void (функцию, перед которой стоит void). Здесь это действительно становится странновато:

```
void func() { }

class MyClass {
   void sayHi() {
      print("Hi");
      dynamic a = 1;
      return a;
   }
}
```

```
main() {
   MyClass mc = MyClass();
   var b = mc.sayHi();
   print(b);
}
```

Учитывая, что `sayHi()` является функцией `void`, вы ждете, что ее возврат приведет к ошибке, верно? Как бы не так! Он будет скомпилирован. Ну, он будет скомпилирован, за исключением строки `print(b)`. Это вызовет ошибку компиляции. Причина в том, что `void` не возвращает какого-либо значения, а возможность писать `var b = mc.sayHi();` нужна только для прикола.

Так что да, void в Dart – это странная штука. Советую не использовать его, если не знаете, зачем это вам нужно.

Но `void` используется не только для функций, которые не возвращают данные. Вы также можете использовать его в качестве параметра шаблонного класса (generic class) вместо `Object`:

```
main() {
   List<void> l = [ 1, 2 ];// Эквивалент List<Object> = [ 1, 2 ];
   print(l);
}
```

А вот *почему* вы можете так сделать, я расскажу в разделе об асинхронном коде.

Операторы

Dart имеет типичный набор операторов, с большинством из которых вы знакомы. Список этих операторов вы можете увидеть в табл. 2-1.

Таблица 2-1. *Операторы Dart*

Оператор	Значение
+	Сложение
-	Вычитание
-expr	Префиксный оператор «унарный минус» (он же отрицание / обратный знак выражения)
*	Умножение
/	Деление
~/	Вернуть целочисленный результат деления
%	Получить остаток целочисленного деления (по модулю)
++var	Префиксный оператор «инкрементирование» (increment, приращение), эквивалентно записи var = var + 1 (значение выражения var + 1), выполняется перед обращением к текущему значению переменной

Оператор	Значение
var++	Постфиксный оператор «инкрементирование», аналогичен ++var, но выполняется после обращения к текущему значению переменной
--var	Префиксный оператор «декрементирование» (decrement, уменьшение), эквивалентный var = var – 1 (значение выражения – это var – 1)
var--	Постфиксный оператор «декрементирование», аналогично var = var – 1 (значение выражения var – 1)
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
=	Присваивание
&	Логическое И (AND)
|	Логическое ИЛИ (OR)
^	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)
~expr	Унарное побитовое дополнение (нули становятся единицами, а единицы становятся нулями)
<<	Сдвиг влево
>>	Сдвиг вправо
a ? b : c	Тернарное условное выражение, эквивалентно if (a) b else c;
a ?? b	Двоичное условное выражение: если a не null, то возвращает a, в противном случае возвращает b
..	Каскадная нотация
()	Функция
[]	Доступ к списку
.	Доступ к членам

Примечание по оператору равенства (`==`): это проверка значений, а не проверка объектов. Когда вам нужно проверить, ссылаются ли две переменные на один и тот же объект, используйте глобальную функцию `identical()`.

При использовании оператора `==` (как в `if (a == b)`) `true` возвращается, если они оба имеют значение `null`, `false` – если только один. Когда выполняется это выражение, на самом деле вызывается метод `==()` первого операнда (да, «==» – это действительно название метода!).

Итак:

```
if (a == b)
```

эквивалентно...

```
if (a.==(b))
```

Примечание по оператору присваивания (=): существует также оператор ??=, который выполняет назначение только в том случае, если первый операнд равен null.

Еще одна заметка об операторе =: существует множество составных операторов, которые объединяют назначение и операцию, такие как

```
-= /= %= >>= A= += *= ~/= <<= &= |=
```

Примечание к оператору доступа к членам класса (.): существует также условная версия написания ?., которая позволяет вам получить доступ к методу или свойству объекта, если этот объект не null.

Рассмотрим такой пример:

```
var person = findPerson("Frank Zammetti");
```

Если person может быть null, то написание print(person?.age) позволит избежать ошибки нулевого указателя (null pointer exception). Результат операции в этом случае будет null, но без ошибки.

Примечание к оператору каскадной записи (..): он позволяет составить код следующим образом:

```
var person = findPerson("Frank Zammetti");
obj.age = 46;
obj.gender = "male";
obj.save();
```

или написать его так:

```
findPerson("Frank Zammetti")
...age = 46
..gender = "male"
..save();
```

Используйте любой стиль, на ваш вкус и цвет, Dart все равно. Классы также могут определять пользовательские операторы, но в этом нет смысла, пока мы не поговорим о классах, так что давайте сделаем это сейчас!

Коротко про ООП в Dart

Dart является объектно-ориентированным языком, так что мы имеем дело с классами и объектами. Создать класс так же просто, как

```
class Hero { }
```

Да, это все!

Экземпляры класса

Классы очень редко бывают пустыми, чаще они содержат какие-либо переменные (их также называют «члены», «поля» или «свойства»). Чтобы объявить их, вам нужно написать следующий код:

```
class Hero {
   String firstName;
   String lastName;
}
```

Любое поле экземпляра класса, которое вы не инициализируете значением, имеет по умолчанию значение `null`. Dart будет автоматически генерировать метод `getter` («получатель», возвращает текущее значение) для каждой переменной, и он также будет генерировать `setter` («установитель», устанавливает новое значение) для любых переменных, не отмеченных словом `final`.

Переменные могут быть помечены как статические, это означает, что вы можете использовать их без создания экземпляра класса:

```
class MyClass {
   static String greeting = "Hi";
}
main() {
   print(MyClass.greeting);
}
```

Этот код выведет «Hi» без создания экземпляра `MyClass`.

Методы

Классы также могут иметь функции, называемые методами:

```
class Hero {
   String firstName;
   String lastName;
   String sayName() {
      return "$lastName, $firstName";
   }
}
```

Мы рассмотрим функции более подробно в следующем разделе, но я уверен, что вы уже и так с ними знакомы. Если же нет, то эта книга, вероятно, не лучшая отправная точка для вас, так как она предполагает, что вы программист хотя бы начального уровня. А пока усвойте, что ключевое слово `return` возвращает значение из функции (или метода, если он часть класса) и завершает ее выполнение.

Теперь у нас есть метод `sayName()`, который мы могли бы вызвать так:

```
main() {
```

```
   Hero h = new Hero ();
   h.firstName = "Luke";
   h.lastName = "Skywalker";
   print(h.sayName());
}
```

Этот код показывает, что методы работы со свойствами (getter, setter) действительно были созданы за нас автоматически, поэтому `h.firstName = «Luke»;` работает.

Я кое-что пропустил: как и практически во всех объектно-ориентированных языках, ключевое слово new создает объекты заданного типа, как показано в предыдущем примере. Тем не менее в Dart ключевое слово new не является обязательным. Таким образом, в дополнение к предыдущему коду вы также можете написать

```
var h = Hero();
```

Честно говоря, это было, на мой взгляд, одной из самых странных вещей, к которым нужно привыкнуть в Dart! Я не уверен, что есть какая-то веская причина, чтобы делать именно так, поэтому просто пишите, как считаете нужным!

Методы также могут быть помечены как статические с помощью ключевого слова static:

```
class MyClass {
   static sayHi() {
      print("Hi");
   }
}
main() {
   MyClass.sayHi();
}
```

Как и в примере со статической переменной, здесь снова выводится «Hi», но на этот раз в результате вызова `sayHi()` без создания экземпляра `MyClass`.

Конструкторы

Классы зачастую содержат конструкторы, то есть специальные функции, которые выполняются при создании их экземпляра. Просто добавим один:

```
class Hero {
   String firstName;
   String lastName;
   Hero(String fn, String ln) {
      firstName = fn;
      lastName = ln;
   }
   String sayName() {
```

```
      return "$lastName, $firstName";
   }
}
```

Конструктор всегда имеет то же имя, что и класс, и не имеет return. А наш тестовый код будет выглядеть так:

```
main() {
   Hero h = new Hero("Luke", "Skywalker");
   print(h.sayName());
}
```

Однако конструктор, который просто устанавливает значения переменных экземпляра класса, используется очень часто, поэтому в Dart есть сокращенная форма для его записи:

```
class Hero {
   String firstName;
   String lastName;
   Hero(this.firstName, this.lastName);
   String sayName() {
      return "$lastName, $firstName";
   }
}
```

Ключевое слово «this»

Ключевое слово this ссылается на текущий экземпляр класса, внутри которого выполняется блок кода. Как правило, вы должны использовать this только в случае конфликта имен. Например:

```
class Account {
   int balance;
   Account(int balance) {
      this.balance = balance;
   }
}
```

Философы спорят о том, следует ли вам когда-либо употреблять this в других случаях, так как это позволяет устранить неоднозначность (мой личный опыт говорит, что вы никогда не будете этого делать, так как эта позиция достаточно спорна). А также это ключевое слово необходимо для краткой записи конструктора класса – this().

Обратите внимание, что если ваш класс не содержит конструктора, как в первых трех версиях Hero, упомянутых ранее, Dart автоматически сгенерирует конструктор без аргументов, который просто вызовет конструктор родительского класса без аргументов (в примере выше «родителем», или суперклассом, явля-

ется Object). Кроме того, обратите внимание, что подклассы не наследуют конструкторы.

Конструктор также может быть помечен ключевым словом factory. Оно используется, когда конструктор может не возвращать экземпляр своего класса. Я знаю, звучит странно, потому что это необычная возможность для большинства объектно-ориентированных языков, но такое может произойти, если, например, вы хотите вернуть существующий экземпляр класса из кеша уже построенных объектов и не создавать новый объект, как происходит по умолчанию.

Конструктор factory может также возвращать экземпляр подкласса, а не сам класс. В противном случае factory работает так же, как и любой другой конструктор, и вы можете вызывать его похожим образом, с той лишь разницей, что внутри него нет доступа к this.

Подкласс

Я упомянул подклассы минуту назад, так как же мы их определяем? Вот простой пример:

```
class Hero {
   String firstName;
   String lastName;
   Hero.build(this.firstName, this.lastName);
   String sayName() {
      return "$lastName, $firstName";
   }
}

class UltimateHero extends Hero {
   UltimateHero(fn, ln) : super.build(fn, ln);
   String sayName() {
      return "Jedi $lastName, $firstName";
   }
}
```

Ключевое слово extends, за которым следует имя класса, от которого мы наследуемся, – это все, что нужно.

Тем не менее здесь происходит немного больше интересного. Во-первых, понятие именованных конструкторов. Взгляните на класс Hero. Видите метод Hero.build()? Это тоже конструктор, который мы называем *именованным*. Причина, по которой необходим this, заключается в том, что в классе UltimateHero конструкторы не наследуются. Но, учитывая, что конструктор должен делать то же самое, что и Hero.build(), нет смысла дублировать код (принцип DRY – Don't Repeat Yourself, не повторяй себя). Так как же вызвать конструктор родительского класса? А вот как: super.build(fn, ln); эта часть следует за конструктором UltimateHero(fn, ln). Ключевое слово super позволяет вызывать

методы или свойства, доступные в родительском классе. Но способа вызвать неименованный конструктор, к сожалению, нет. Другими словами, `super(fn, ln)`, который работает в большинстве иных языков, не работает в Dart. Единственное, что мы можем сделать, – это вызвать именованный конструктор, что мы и делаем, используя синтаксис после двоеточия.

Методы Getter и Setter

Теперь я хочу вернуться к понятиям `getter` (получает) и `setter` (устанавливает). В Dart есть возможность создавать свои собственные `getter` и `setter`, помимо автоматически генерируемых, что дает возможность управлять логикой объектов на лету. Для этого Dart предлагает ключевые слова `get` и `set`:

```
class Hero {
   String firstName;
   String lastName;
   String get fullName => "$lastName, $firstName";
   set fullName(n) => firstName = n;
   Hero(String fn, String ln) {
      firstName = fn;
      lastName = ln;
   }
   String sayName() {
      return "$lastName, $firstName";
   }
}
```

Здесь у нас появляется поле `fullName`. Когда мы попытаемся получить к нему доступ, то получим такое же объединение `lastName` и `firstName`, как и в `sayName()`, но когда мы попытаемся задать его, мы перезапишем поле `firstName`. Итак, теперь мы можем это проверить:

```
main() {
   Hero h = new Hero("Luke", "Skywalker");
   print(h.sayName());
   print(h.fullName);
   h.fullName = "Anakin";
   print(h.fullName);
}
```

Вывод здесь будет таким:

```
Skywalker, Luke
Skywalker, Luke
Skywalker, Anakin
```

Надеюсь, вы понимаете, почему!

Интерфейсы

Dart, как и большинство других объектно-ориентированных языков, не различает понятия классов и интерфейсов. Вместо этого класс Dart также неявно определяет интерфейс. Следовательно, мы могли бы повторно реализовать класс `UltimateHero` следующим образом:

```
class UltimateHero implements Hero {
   @override
   String firstName;
   @override
   String lastName;
   UltimateHero(this.firstName, this.lastName);
   String sayName() {
      return "Jedi $lastName, $firstName";
   }
}
```

`@override` – это аннотация, но мы перейдем к ней позже. А пока просто запомните, что необходимо указать Dart, что мы переопределяем метод `getter` и `setter` суперкласса для двух отмеченных полей, а без этого получим ошибку. С этими доработками нам также нужно изменить конструктор, потому что теперь мы не наследуемся от класса и не имеем доступа к конструктору `Hero.build()` (поскольку конструкторы никогда не наследуются, а реализация интерфейса также означает, что у нас нет доступа к поведению класса, который обеспечивает интерфейс, мы просто говорим, что наш новый класс предоставляет те же функциональные возможности, что и контрактные обязательства интерфейса), поэтому сделаем конструктор, который повторяет то, что есть в `Hero`. Единственное новое изменение – это замена ключевого слова `extends` на `implements`, поскольку теперь мы реализуем интерфейс, определенный классом `Hero`, а не наследуемся от него.

Совет. Как правильно использовать `implements` или `extends`? Это вопрос, который часто звучит в мире объектно-ориентированного программирования. Некоторые люди считают, что композиционная модель, которая реализует `implements`, чище. Другие считают, что иерархия классов лучше подходит классическому ООП, и поэтому предпочитают `extends`. Независимо от вашего мнения, поймите один ключевой момент: это разные понятия, а в Dart, как и в Java и многих других языках ООП, вы можете наследоваться напрямую только от одного класса, а интерфейсов реализовать столько, сколько хотите. Итак, если ваша цель – создать класс, который предоставляет единый интерфейс для нескольких классов, то, скорее всего, вам нужно слово `implements`. Иначе вы можете выбрать путь `extends`.

Абстрактные классы

Давайте кратко обсудим `abstract`. Это ключевое слово обозначает абстрактный класс, например:

```
abstract class MyAbstractClass {
   someMethod();
}
```

Для класса MyAbstractClass нет возможности создать экземпляры, так как у него нет реализации. Однако этот класс может быть унаследован потомком, который реализует необходимые методы, и сам может иметь экземпляры. Методы внутри абстрактных классов могут обеспечивать реализацию или даже сами могут быть абстрактными, и в этом случае они всегда должны быть реализованы подклассом. Метод someMethod( ) считается абстрактным, потому что у него нет тела метода, однако вместо этого вы можете сделать так:

```
abstract class MyAbstractClass {
   someMethod() {
      // Сделай что-нибудь
   }
}
```

В этом случае someMethod( ) имеет реализацию по умолчанию, и поэтому подкласс не обязан реализовывать ее, если не хочет.

В дополнение к наследованию классов, реализации интерфейсов и абстрактным классам Dart также предлагает понятие подмешивания (mixin) классов, вместе с которым в игру вступает ключевое слово with:

```
class Person {}

mixin Avenger {
   bool wieldsMjolnir = false;
   bool hasArmor = false;
   bool canShrink = true;
   whichAvenger() {
      if (wieldsMjolnir) {
         print("I'm Thor");
      } else if (hasArmor) {
         print("I'm Iron Man");
      } else {
         print("I'm Ant Man");
      }
   }
}

class Superhero extends Person with Avenger { }

main() {
   Superhero s = new Superhero();
   s.whichAvenger();
}
```

Здесь у нас есть два класса, Person и Superhero, и один mixin Avenger (который мы задаем при помощи ключевого слова mixin, стоящего перед его определением). Обратите внимание, что Person и Superhero являются пустыми классами, это означает, что вызов whichAvenger() должен поступать из другого места, и это так: мы «подмешиваем Мстителя в класс Супергероя», указав with Avenger в определении класса Superhero. Теперь, что бы ни было в mixin Avenger, это будет присутствовать и в Superhero.

Видимость

В Java и многих других языках ООП вам обычно нужно указать, какой уровень видимости должен быть у членов класса, используя ключевые слова, такие как public, private и protected. Dart другой: все общедоступно, если название не начинается с подчеркивания, которое помечает его как доступное только для текущей библиотеки или класса.

Операторы

Как говорил Стив Джобс: «Еще кое-что!»

Из различных операторов, которые предоставляет Dart, специальными являются следующие (запятые и точка не являются операторами!): <, >, <=, >=, -, +, /, ~/, *, %, |, ^, &, <<, >>, [], []=, ~, ==. Почему же они специальные? Ну, только их вы можете переопределить в классе, используя ключевое слово operator:

```
class MyNumber {
   num val;
   num operator + (num n) => val * n;
   MyNumber(v) { this.val = v; }
}

main() {
   MyNumber mn = MyNumber(5);
   print(mn + 2);
}
```

Здесь класс MyNumber переопределяет оператор +. Данная реализация оператора вернет значение val * n (умножить) вместо ожидаемого val + n. Таким образом, когда main() выполнится, то в консоли вы увидите 10 (5*2) вместо 7 (5+2), чего ожидали бы по умолчанию. Это произошло из-за того, что мы переопределили оператор +.

Есть одна хитрость: если вы переопределяете оператор ==, вы также должны переопределить getter для свойства hashCode. В противном случае эквивалентность не может быть достоверно определена.

Уф, это было непросто! Ведь данный раздел, вероятно, охватывает большую часть того, что вам нужно знать о классах и объектах в Dart.

Кое-что о функциях

У функций Dart есть свой тип: `Function`. Это означает, что функции могут быть переменными, могут передаваться как параметры, а также могут быть независимыми объектами. Есть одна ключевая функция, о которой вы уже знаете, и это `main()`.

Функции в Dart не лишены синтаксического сахара. Они могут иметь именованные и необязательные параметры. Даже если вы используете именованные параметры или только позиционные (типичный стиль списка параметров), у вас могут быть и необязательные параметры, но вы не можете смешивать эти два стиля. Вы также можете задать для параметров значения по умолчанию. Изучите этот код:

```
greet(String name) {
   print("Hello, $name");
}

class MyClass {
   greetAgain({ Function f, String n = "human" }) {
      f(n);
   }
}

main() {
   MyClass mc = new MyClass();
   greet("Frank");
   mc.greetAgain( f : greet, n : "Traci" );
   mc.greetAgain( f : greet);
}
```

Здесь вы наглядно видите большую часть того, о чем я говорил. Во-первых, у нас есть функция `greet()`, а еще класс с методом `greetAgain()`. Этот метод принимает список именованных параметров, и да, один из этих параметров является функцией! Посмотрите, параметр `n` имеет значение по умолчанию. Круто, да? Затем внутри функции `greatAgain()` мы вызываем функцию, на которую ссылается `f`, передавая ей значение параметра `n`. Другими словами, любая функция может передаваться в качестве значения параметра `f`, потому что она записана как функция, и мы можем использовать переменную `f` для ее вызова.

Теперь, в функции `main()` мы сначала просто вызываем `greet()`, передавая ему имя, чтобы вывести приветствие. Затем мы вызываем метод `greetAgain()` экземпляра `MyClass mc`, и на этот раз мы передаем именованные параметры, а значение параметра `f` является ссылкой на функцию `greet()`. Я продемонстрировал это дважды, чтобы вы могли запомнить, как это работает. Итак, если вы не передадите имя, то увидите приветствие обычного человека (human).

Примечание. Во многих языках данные, которые вы передаете функциям, называются аргументами. Это термин, на котором я вырос, но документация по Dart предпочитает слово «параметр» (parameter). Честно говоря, иногда я могу смешивать эти термины, но в данном контексте они означают одно и то же.

К сожалению, DartPad на момент написания этой книги не позволит импортировать библиотеки, поэтому нам понадобится аннотация `@required`, которая должна быть перед параметром `n` в `greetAgain()`, а не параметром `f`. Итак, поскольку вы можете запустить код в DartPad и попробовать его, я опустил эту аннотацию. Также обратите внимание, что при использовании позиционных параметров вы не используете `@required`, вместо этого вы заключаете необязательные параметры в квадратные скобки.

Хотя у большинства функций есть имя, они могут быть и анонимными. В качестве примера:

```
main() {
   var bands = [ "Dream Theater", "Kamelot", "Periphery" ];
   bands.forEach((band) {
      print("${bands.indexOf(band)}: $band");
   });
}
```

Здесь есть функция, передаваемая методу `forEach()` для объектов из `List`, но у нее нет имени, и в результате она существует только на время выполнения `forEach()`.

Для функций очень важна область, которую они задают. Dart считается «лексически ограниченным» (lexically scoped) языком, а значит, видимость переменной в основном определяется структурой самого кода. Если она заключена в фигурные скобки, то находится в пределах этой области видимости, и эта область расширяется вниз, что означает, что если у вас есть вложенные функции (да, это можно сделать в Dart!), без разницы, как глубоко вы двигаетесь, вам все равно доступна переменная, определенная выше. Чтобы понять это, ознакомьтесь с примером:

```
bool topLevel = true;

main() {
   var insideMain = true;

   myFunction() {
      var insideFunction = true;
      nestedFunction() {
         var insideNestedFunction = true;
         assert(topLevel);
         assert(insideMain);
         assert(insideFunction);
         assert(insideNestedFunction);
      }
```

```
   }
}
```

nestedFunction() может использовать любую переменную вплоть до верхнего уровня.

Dart также поддерживает концепцию замыканий (closure), так что функция захватывает, или «замыкает», свою лексическую область видимости, даже если функция используется за пределами исходной области видимости. Другими словами, если функция имеет доступ к переменной, то она в некотором смысле «запомнит» эту переменную, даже если области, в которой находится переменная, больше не существует, когда функция выполняется.

В качестве примера:

```
remember(int inNumber) {
   return () => print(inNumber);
}

main() {
   var jenny = remember(8675309);
   jenny();
}
```

Здесь вызов jenny() выдает 8675309, хотя параметр не был ему передан. Это происходит потому, что jenny() включает лексическую область remember() и контекст выполнения, который содержит значение, переданное в вызов remember() при получении ссылки. Это сбивает с толку, если вы никогда не встречались с этим раньше, но хорошая новость заключается в том, что вам, вероятно, не нужно будет использовать много замыканий в Dart (по сравнению с JavaScript, где это происходит постоянно).

Dart также поддерживает стрелки, или лямбда-нотацию для определения функций. Это одно и то же:

```
talkl() { print("abc"); }
talk2() => print("abc");
```

Что такое Assertions

Ключевое слово assert, как и в большинстве других языков, используется только для тестовых сборок. Оно используется для прерывания выполнения, когда заданное условие ложно, и выдает исключение AssertionException. Например:

```
assert (firstName == null);
assert (age > 25);
```

При необходимости вы можете добавить сообщение к assert следующим образом:

```
assert (firstName != null, "First name was null");
```

Вне времени: асинхронность

Асинхронное программирование – это важная составляющая в наши дни! Оно есть везде, на всех языках, и Dart не исключение. В Dart ключевыми для асинхронности являются два класса, `Future` и `Stream`, наряду с двумя ключевыми словами, `async` и `await`. Оба класса – это объекты, которые возвращают асинхронные функции до завершения длительной операции, позволяя программе ожидать результата и продолжать при этом выполнять другие действия, а затем продолжить работу с того места, где началось ожидание.

Чтобы вызвать функцию, которая возвращает `Future`, вы используете ключевое слово `await`:

```
await someLongRunningFunction();
```

И это все! Ваш код в фоне будет ждать, пока не завершится `someLongRunningFunction()`. Программа продолжит выполнять другие процессы, а не заблокируется длительной операцией (например, если `someLongRunningFunction()` сделать синхронной, то обработчик нажатия кнопки все заблокирует). Сама асинхронная функция должна быть помечена, перед определением ее тела, ключевым словом async и возвращать `Future`:

```
Future someLongRunningFunction() async {
   // Делаем что-то, что занимает много времени
}
```

Есть еще кое-что: функция, которая вызывает `someLongRunningFunction()`, сама должна быть помечена как `async`:

```
MyFunction() async {
   await someLongRunningFunction();
}
```

Вы можете ждать сколько угодно функций в одной асинхронной функции, ожидание сработает для каждой из них.

Примечание. Существует также Future API, который позволяет вам делать то же самое, но без использования `async` и `await`. Я не рассматриваю это только потому, что большинство считает `async/await` более элегантными, и я разделяю это мнение. Не бойтесь исследовать этот API самостоятельно, если вам интересно.

Streams обрабатываются практически так же, но для чтения их данных необходимо использовать асинхронный цикл `for`:

```
await for (varOrType identifier in expression) {
   // Выполняется каждый раз, когда Stream выдает значение.
}
```

Разница между ними в том, что использование `Future` означает, что возврат из длительной функции не будет происходить до тех пор, пока эта функция не

завершится, сколь бы она ни выполнялась. А функция Stream может постепенно возвращать данные, так что ваш код, ожидающий их, будет выполняться каждый раз, когда Stream выдает значение. Вы можете прервать цикл, чтобы остановить чтение Stream, и цикл сам завершится, когда асинхронная функция закроет этот Stream. Как и прежде, использование await разрешено только внутри функции с async.

Тсс, тихо! Библиотеки (и видимость)

Библиотеки используются в Dart, чтобы сделать код модульным и разделяемым. Библиотека предоставляет внешний API для вашего кода. Она также служит методом изоляции в том смысле, что все в библиотеке, что начинается с подчеркивания, видно только внутри этой библиотеки. Интересно, что каждое приложение Dart автоматически становится библиотекой, независимо от того, делаете вы что-то для этого или нет! Библиотеки могут быть упакованы и доставлены другим пользователям с помощью инструмента публикации Dart SDK, который является менеджером пакетов и ресурсов.

Обратите внимание, я не буду рассказывать о создании библиотек, поскольку это слишком долго и не понадобится в данной книге. Так что если вы заинтересованы в распространении своих библиотек, то вам следует обратиться к документации Dart. Хочу отметить, что Dart SDK входит в состав Flutter SDK, так что он у вас уже есть.

Чтобы использовать библиотеку, в игру вступает ключевое слово import:

```
import "dart:html";
```

Некоторые библиотеки предоставляются вашим собственным кодом, а другие встроены в Dart, как эта. Для встроенных библиотек есть специальные URI, они начинаются со слова dart:, которое является частью схемы URI.

Если библиотека, которую вы импортируете, берется из какого-либо пакета (мы кратко затронули это в главе 1, а подробнее поговорим в следующей главе), тогда вместо dart: вам стоит использовать package: схему:

```
import "package:someLib.dart";
```

Если библиотека представлена частью вашего кода или файлом, который вы скопировали в свою кодовую базу, тогда URI – это относительный путь файловой системы:

```
import "../libs/myLibrary.dart";
```

Иногда вы можете импортировать две библиотеки, но в них есть конфликтующие идентификаторы. Например, у libl есть класс Account, как и у lib2, но вам нужно импортировать оба. В этом случае вступает в игру ключевое слово as:

```
import "libs/libl.dart";
```

```
import "libs/lib2.dart" as lib2;
```

Теперь, если вы хотите ссылаться на класс Account в libl, вы пишете:

```
Account a = new Account();
```

Но если вам нужен Account из lib2, вы должны написать:

```
lib2.Account = new lib2.Account();
```

При таком способе все в библиотеке будет импортировано. Однако вы можете импортировать только части библиотеки:

```
import "package:lib1.dart" show Account;
import "package:lib2.dart" hide Account;
```

Здесь будет импортирован лишь класс Account из libl, и будет *импортировано* все, кроме класса Account из lib2.

Все предыдущие способы немедленно импортируют библиотеку при старте приложения. Но вы можете и отложить загрузку, это поможет сократить время первоначального запуска:

```
import "libs/libl.dart" deferred as libl;
```

Осталось еще немного работы! Когда вы дойдете до места в коде, где вам нужна эта библиотека, загрузите ее:

```
await lib1.loadLibrary();
```

Как вы узнали в предыдущем разделе, этот код должен быть в функции, помеченной async.

Обратите внимание, что многократный вызов loadLibrary() для библиотеки – это нормально и не принесет никакого вреда. Кроме того, до тех пор, пока библиотека не загружена, константы, определенные в ней, если таковые имеются, на самом деле не являются константами – они не существуют, пока библиотека не загружена, поэтому вы не можете их использовать.

Если вы захотите провести A/B-тестирование с вашим приложением, отложенная загрузка может быть также полезна, поскольку вы сможете динамически загружать разные библиотеки, чтобы их сравнить.

Для тебя я сделаю исключение: обработка исключений

Обработка исключений (exception) в Dart проста и очень похожа на Java или JavaScript, или даже большинство других языков. В отличие от них, в Dart не обязательно объявлять, какие именно исключения должна выдавать функция, а какие вы не должны перехватывать. Другими словами, все исключения в Dart не проверяются.

Для начала вы можете сами «бросить» (throw) исключение:

```
throw FormatException("This value isn't in the right format");
```

Исключения – это объекты, поэтому вам нужно создать один, чтобы затем «бросить».

Интересно, что в Dart вам не обязательно бросать какой-либо специальный exception, это может быть даже строка. Вы можете бросить строку в качестве исключения:

```
throw "This value isn't in the right format";
```

Это exception, и это очень удобно. Однако, с учетом вышесказанного, бросать все, что выходит за пределы классов Error или Exception, считается дурным тоном, поэтому «выбрасывать всякую фигню» – это одна из возможностей в Dart, которую все намеренно игнорируют!

Идем дальше! Чтобы поймать исключение, вы пишете:

```
try {
   somethingThatMightThrowAnException();
} on FormatException catch (fe) {
   print(fe);
} on Exception catch (e) {
   Print("Some other Exception: " + e);
} catch (u) {
   print("Unknown exception");
} finally {
   print("All done!");
}
```

Здесь заслуживают отдельного внимания несколько вещей. Во-первых, вы оборачиваете код, который может генерировать исключение (запомните: лучше обрабатывать только те исключения, которые вы ожидаете), в блок try. Затем вы обрабатываете одно или несколько исключений по вашему усмотрению. Здесь функция thatMightThrowAnException() может генерировать исключение FormatException, которое мы хотим обработать. Потом мы будем обрабатывать любой другой объект, который является подклассом Exception, и отображать его сообщение. Наконец, все, что было выброшено, будет обработано как неизвестное исключение.

Далее обратите внимание на синтаксические различия: вы можете писать в <exception_type> catch или можете просто написать catch(<object_identifier>), где object_identifier – это объект, который был выброшен под любым именем, которое вы хотите обработать в блоке catch. Разница в вашей цели: если вы просто хотите обработать исключение, но не заботитесь о брошенном объекте, вы можете игнорировать его тип. Если вам не важен тип, но вы хотите получить брошенный объект, просто используйте catch. Когда вам важен и тип, и сам объект, используйте <exception_type> catch(<object_identifier>).

Вы также можете добавить finally к блоку try... catch. Такой код будет выполняться независимо от того, было выброшено какое-либо исключение

или нет. Код в `finally` будет выполняться после того, как все активные блоки `catch` закончат свою работу.

Наконец, вы можете определить свои собственные классы исключений, просто унаследовавшись от `Exception` или `Error`, и ваши исключения будут работать точно так же, как и предоставленные Dart.

У меня есть сила: генераторы

Иногда у вас есть код, который возвращает несколько значений. Возможно, этот код опирается на удаленную систему, которую ему нужно вызвать. В этом случае вы не захотите блокировать интерфейс вашего приложения, пока эти значения загружаются, для решения задачи можно сгенерировать этот список «ленивым» (lazy) способом. Или же вы можете просто не хотеть или не иметь возможности создать весь список сразу. В любом случае вам будет полезно познакомиться с генераторами (generator).

В Dart есть два типа генераторов: синхронный, который возвращает объект `Iterable`, и асинхронный, который возвращает объект `Stream`. Давайте сначала обсудим синхронный:

```
Iterable<int> countTo(int max) sync* {
   int i = 0;
   while (i < max) yield i++;
}
main() {
   Iterable it = countTo(5);
   Iterator i = it.iterator;
   while (i.moveNext()) {
      print(i.current);
   }
}
```

Первое, на что нужно обратить внимание, – это маркер `sync*` перед телом функции. Он говорит Dart о том, что это функция генератора (кстати, генератор всегда является функцией). Второй важный момент – использование ключевого слова `yield` внутри генератора. Оно добавляет значение к `Iterable`, которое создается за кулисами и возвращается из функции.

Вызов `countTo()` немедленно вернет объект перечисления (iterable). Затем ваш код может извлечь из него итератор, чтобы начать проход по списку результатов, даже если он еще не заполнен. Интересно, что `countTo()` не будет выполняться, пока вызывающий его код не извлечет этот итератор, а затем не вызовет для него `moveNext()`. Когда это произойдет, `countTo()` будет выполняться, пока точка не достигнет оператора `yield`. Выражение `i++` вычисляется и «возвращается» вызывающей стороне через «невидимый» итератор. Затем `countTo()` приостанавливается (так как он еще не выполнил условие завершения), а `moveNext()` возвращает `true`. Поскольку код, использующий `countTo()`, выполняет итерацию

за итерацией, мы можем только прочитать его текущее значение через свойство current.

Затем countTo() возобновляет выполнение и вызывает moveNext(). Когда цикл заканчивается, метод неявно вызывает return, что приводит к его завершению. В этот момент moveNext() возвращает false вызывающей стороне, и цикл while завершается.

Второй тип генератора можно продемонстрировать с помощью такого кода:

```
Stream<int> countTo(int max) async* {
   int i = 0;
   while (i < max) yield i++;
}

main() async {
   Stream s = countTo(5);
   await for (int i in s) { print(i); }
}
```

Разница заключается в использовании Stream в качестве возвращаемого типа и применении маркера async* вместо sync* перед телом функции. Другое отличие заключается в использовании метода countTo(). Поскольку это асинхронный метод, нам нужно, чтобы функция, в которую он был вызван, также была помечена как асинхронная. Поэтому добавляется await. Это форма цикла for с поддержкой многопоточности. Поскольку цикл for ожидает функцию countTo(), чтобы выполнить свою работу, эта функция фактически «выталкивает» значение в цикл for через Stream. Из примера выше может показаться неочевидным, почему нужно делать именно так, но представьте, что, вместо того чтобы просто увеличивать i, приходится делать запрос на сервер, чтобы получить следующее значение. Надеюсь, так ценность генераторов более наглядна.

Мета-Dart: метаданные

Dart также поддерживает понятие метаданных, встроенных в ваш код. В других языках их называют аннотациями, как и в Dart (документация называет это «аннотациями метаданных», metadata annotations, что, на мой взгляд, немного многословно, но более точно).

Dart предоставляет две аннотации, одну из которых вы видели ранее: @override. Она используется для указания того, что класс намеренно переопределяет член своего родительского суперкласса.

Другая аннотация Dart – это @deprecated. Она помечает поле, которое уже устарело и может быть скоро удалено. @deprecated часто используют для методов в классе, которые будут удалены в следующей версии.

Еще вы можете создавать собственные аннотации. Аннотация – просто класс, так что это может быть аннотацией:

```
class MyAnnotation {
   final String note;
   const MyAnnotation(this.note);
}
```

Здесь аннотация может принимать аргумент, поэтому мы используем его следующим образом:

```
@MyAnnotation("This is my function")
Void myFunction() {
      // Делать что-нибудь
}
```

Вы можете задать аннотации для следующих языковых элементов: библиотеки, классы, определения типов, параметры методов, конструкторы, фабрики (factory), функции, поля, объявления переменных, а также директив `import` и `export`. Метаданные, переносимые аннотациями, могут быть извлечены во время выполнения с помощью рефлексии в Dart, но я оставлю это в качестве упражнения для вас.

Пообобщаемся? Дженерики, или обобщения

Дженерики (generics) используются в Dart для создания шаблонов классов и методов. Например, если вы пишете

```
var ls = List<String>();
```

то Dart знает, что список `ls` может содержать только строки. На этапе компиляции по шаблону создается класс `List`, работающий лишь со строками (String).

Не лучше ли было назвать это *«уточнителями»*(specifics)? Вы говорите Dart, какой *конкретно* (specifically) тип данных содержит этот список. Дженерики вступают в игру, когда вы пишете что-то вроде:

```
abstract class Things<V> {
   T getByName(String name);
   void setByName(String name, V value);
}
```

Здесь мы говорим Dart, что класс `Things` можно использовать для любого типа, где `V` является его заменой (универсальные типы, подобные `V`, обозначаются одной буквой, чаще всего `E`, `K`, `S`, `T` или `V`). Теперь, когда класс `Things<V>` служит интерфейсом, вы можете реализовать много разных версий, используя разные типы (например, `Person`, `Car`, `Dog` и `Planet`, которые могут реализовывать один и тот же базовый интерфейс).

`Lists` и `Maps` могут быть определены так, как это было показано ранее, но вы можете использовать и упрощенную форму:

```
var brands = <String>[ "Ford", "Pepsi", "Disney" ];
   var movieStars = <String, String>{
      "Pitch Black : "Vin Diesel",
      "Captain American" : "Chris Evans",
      "Star Trek" : "William Shatner"
};
```

В Dart шаблонные типы *материализуются* (*reified*), это означает, что тип класса-параметра сохраняется с ними на время выполнения программы, что позволяет вам проверить тип коллекции ключевым словом `is`:

```
var veggies = List<String>();
veggies.addAll([ "Peas", "Carrots", "Cauliflower"]);
print(veggies is List<String>);
```

Как и следовало ожидать, это выведет в консоль `true` благодаря рефлексии. Хотя все кажется очевидным, но это не всегда работает в других языках. Java, например, использует затирание (erasure) вместо материализации (reification), это значит, что шаблонный тип удаляется во время выполнения. Таким образом, вы сможете проверить, что переменная типа `List`, но не сможете проверить, что это `List<String>`. Я рад, что в Dart это работает не так! Наконец, методы могут использовать дженерики так же, как и классы:

```
class C {
   E showFirst<E>(List<E> lst) {
      E item = lst[0];
      if (item is num) {
         print("It's a number");
      }
      print(item);
      return item;
   }
}

main() async {
   C c = new C();
   c.showFirst(<String>[ "Java", "Dart" ]);
   c.showFirst(<num>[ 42, 66 ]);
}
```

Как видите, мы можем передать любой тип методу `showFirst()`, а он уже определяет его с помощью ключевого слова `is` и действует соответственно. В этом одно из ключевых преимуществ дженериков: вам не нужно писать две разные версии `showFirst()`, одну для обработки строк и одну для обработки чисел. С этим прекрасно справится всего один метод.

Подведем итоги

В этой главе вы познакомились с тем, что может предложить Dart. Вы узнали об основах, таких как типы данных, операторы, комментарии, логические операторы и управление потоком, а также о понятиях среднего уровня, таких как классы, дженерики и библиотеки. Наконец, вы познакомились с более сложными темами: асинхронность функции, генераторы и аннотации метаданных. Так что у вас должна была сформироваться достаточно прочная основа из знаний о языке Dart, с которой уже можно начать изучение Flutter.

В следующей главе мы проведем обзор Flutter, сосредоточившись прежде всего на виджетах, которые он предлагает. Мы начнем использовать наши знания о Dart, одновременно изучая Flutter, что подготовит нас к созданию реальных проектов в 4-й главе!

ГЛАВА 3

СКАЖИ ПРИВЕТ МОЕМУ МАЛЕНЬКОМУ ДРУГУ FLUTTER. ЧАСТЬ I

В первой главе вы получили краткое введение во Flutter, а во второй – познакомились с Dart. Теперь пришло время снова подробнее взглянуть на Flutter.

Учитывая, что во Flutter уйма виджетов, здесь мы рассмотрим только часть из них. А как работать с API операционных систем, мы рассмотрим в следующей главе.

Эта глава (вместе со следующей), как и предыдущая, не стремится стать справочным материалом. Подробное описание более чем 100 доступных виджетов (и каждого с многочисленными опциями, методами и событиями) заняло бы сотни страниц и просто продублировало бы документацию с сайта `flutter.io`. Мы же рассмотрим и подробно обсудим те виджеты и API-интерфейсы, которые, по моему мнению, большинство разработчиков будут использовать регулярно. Я также опишу виджеты, которые демонстрируют общие концепции.

Но, безусловно, виджетов заметно больше, чем вы найдете в этой и следующей главах, также есть шанс, что к моменту выхода книги на прилавки виджетов будет еще больше, чем когда я ее писал. Однако эта и следующая главы предоставят вам отличный обзор того, что доступно мне сейчас, и подготовят вас к написанию кода приложения.

Набор виджетов

Мы начнем с рассмотрения виджетов, и, как я упоминал ранее, на момент написания этой книги их было более ста. Я попытался организовать их в логические группы, чтобы дать вам о них общее представление.

Примечание. Там, где это возможно, я пытался сопоставить Material-в виджеты (стиль Android) с подобными виджетами в стиле iOS. Некоторые из них уникальны для той или иной платформы или не имеют прямого сходства, но большинство имеют, так что вы сможете это увидеть самостоятельно. Я думаю, что такой подход поможет вам понять, как работать с кросс-платформенными проектами.

Layout (компоновка)

Виджеты компоновки (layout) помогут вам организовать пользовательский интерфейс и структурировать его различными способами. В некотором смысле они позволяют вам создать каркас вашего приложения.

MaterialApp, Scaffold, Center, Row, Column, Expanded, Align и Text

Компоновка во Flutter строится на основе сеточной структуры, которая включает строки (Row) и столбцы (Column), следовательно, мы будем работать с виджетами `Row` и `Column`. Каждый из них может иметь один или несколько дочерних элементов, и эти дочерние элементы будут расположены горизонтально (поперек экрана) в случае виджета `Row` или вертикально (сверху вниз) для виджета `Column`.

Использовать их очень просто, вы можете видеть это в листинге 3-1.

Листинг 3-1. *Основы*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext context) {
      return MaterialApp(title : "Flutter Playground",
         home : Scaffold(
            body : Center(
               child : Row(
                  children : [
                     Text("Child1"),
                     Text("Child2"),
                     Text("Child3")
                  ]
               )
            )
         )
      );
   }
}
```

На рис. 3-1 показан результат выполнения данного кода.

Рисунок 3-1. *Основы в картинках!*

Здесь используются не только виджеты `Row` и `Column`, поэтому давайте разберем пример подробнее.

Это приложение Flutter, поэтому оно начинается с обычного импорта. Библиотека Dart включает виджеты стиля `Material`. Затем идет функция `main()`, которая создает экземпляр класса `MyApp` и передает его функции `runApp()`, которую предоставляет Flutter. `MyApp` – это виджет верхнего уровня Flutter, необходимый для запуска приложения.

Класс `MyApp` – это StatelessWidget, так как для него нам не нужно никакого состояния, а метод `build()` создает виджет типа `MaterialApp`, который создает для нас некую «инфраструктуру», поэтому с него мы и начнем. Если хотите, то вы можете использовать другой виджет – `WidgetsApp`, но тогда потребуется написать немного больше кода, чтобы как минимум определить маршруты (мы разберем их в главе про экраны) вашего приложения, так что без необходимости не стоит использовать такой подход. Даже при разработке под iOS вы все равно можете использовать `MaterialApp` в качестве виджета верхнего уровня (в настоящее время в iOS нет специального виджета `CupertinoApp` или чего-то в этом роде, хотя мог бы быть).

Параметр `title` (заголовок), который вы здесь видите, является свойством этого виджета. Заголовок – это строка текста, которая используется операци-

онной системой в качестве названия приложения. Виджет MaterialApp предоставляет довольно много других свойств. Например: свойство color, которое определяет основной цвет, используемый для приложения в интерфейсе ОС, и theme, которое принимает в качестве значения виджет ThemeData и описывает цвета, используемые для приложения.

Для виджета MaterialApp также требуется свойство home, значение которого должно быть виджетом. home – это начало дерева виджетов для основного экрана вашего приложения (или, по крайней мере, экрана, с которого пользователь начинает, главный он или нет). Обычно это виджет Scaffold («строительные леса»). Существует несколько виджетов Scaffold, но все они служат одной цели – реализации базовой структуры макета для ваших страниц в приложение. Как и MaterialApp, базовый виджет Scaffold заботится об общих элементах пользовательского интерфейса, таких как **Navigation Bar** (верхняя панель навигации, где обычно располагается кнопка **Назад**), **Drawer** (выдвигающееся боковое меню) и **Bottom Sheet** (диалоговое окно, выезжающее снизу экрана). Другим классом виджетов Scaffold является CupertinoPageScaffold, он создан специально для iOS и предоставляет базовую структуру разметки страниц iOS. Здесь есть верхняя панель навигации и место для контента на странице. А еще CupertinoTabScaffold, он похож на CupertinoPageScaffold, но содержит панель навигации со вкладками внизу.

Примечание. Чтобы использовать виджеты Cupertino, вам необходимо импортировать в приложение «package:flutter/cupertino.dart»;. Затем, если хотите, вы можете поменять Scaffold на CupertinoPageScaffold, здесь вместо свойства home используется свойство child. Также обратите внимание, что нет никаких ограничений на применение виджетов Cupertino на Android-устройствах. Напомним, что Flutter рисует интерфейс сам, а не полагается на ОС, что позволит вам иметь одинаковый интерфейс на всех поддерживаемых платформах!

Виджет Scaffold предоставляет ряд свойств, включая:

- floatingActionButton, который позволяет вашему приложению поддерживать Floating Action Button, или FAB (этот виджет мы рассмотрим позже);
- drawer, позволяющий указать виджет для бокового выезжающего меню;
- bottomNavigationBar, который дает возможность вашему приложению иметь панель навигации внизу, например для отображения вкладок;
- backgroundColor, позволяющий вам задать цвет фона страницы.

Какой бы вариант Scaffold вы не взяли, ему необходим дочерний виджет, который указывается с помощью свойства body. Если хотите, чтобы все ваши виджеты были расположены вертикально по центру, используйте виджет Center. Он будет центрировать все свои дочерние элементы внутри себя. Имейте в виду, что виджет Center будет автоматически заполнять собой все пространство, которое позволит занять его родительский виджет. В этом случае роди-

тельским виджетом является Scaffold, который автоматически занимает весь размер экрана, поэтому виджет Center также будет заполнять весь экран.

Дочерний элемент внутри Center – это виджет Row, который, таким образом, будет располагаться по центру экрана. У виджета Row есть свойство children, позволяющее нам указывать массив виджетов, которые будут расположены в Row. Здесь определены три дочерних элемента: три виджета Text. Виджет Text отображает одну или несколько строк текста. Вот некоторые интересные свойства, которые поддерживает Text:

- overflow, которое сообщает Flutter, что делать, когда текст выходит за границы своего контейнера (например, TextOverflow.ellipsis приводит к добавлению многоточия в конце);
- textAlign позволяет вам определить, как текст должен быть выровнен по горизонтали;
- textScaleFactor, который масштабирует текст.

Если вы попробуете запустить пример (а вы УЖЕ попробовали, верно?!), то увидите, что все виджеты Text будут сдвинуты влево. А вдруг мы хотим, чтобы они были центрированы горизонтально? Тогда нам нужно задать MainAxis Alignment.center для поля mainAxisAlignment у Row (это просто другое свойство, сродни children).

Теперь дочерние элементы у Row должны вписываться в горизонтальное пространство, которое он заполняет. Сразу скажу, что будет ошибкой иметь дочерние элементы, которым нужно больше места, чем может предоставить Row (или другой контейнер), если он не поддерживает прокрутку. Но что, если мы хотим, чтобы второй Text заполнил все доступное пространство? Тогда мы можем сделать это с помощью нового виджета:

```
Expanded(child : Text("Child2"))
```

Виджет Expanded приводит к тому, что его дочерний элемент заполняет все доступное пространство. Теперь, после рендеринга первого и третьего виджетов Text, оставшееся пространство будет заполнено виджетом с текстом «Child2».

Стоит упомянуть здесь еще один виджет – Align. Как и виджет Center, Align обычно используется, когда у вас есть только один дочерний элемент, он выполняет те же функции, что и Center, но обладает большей гибкостью, поскольку предназначен не только для центрирования контента. Он выравнивает свой дочерний элемент внутри себя, а также может изменять размер в зависимости от размера дочернего элемента. Ключом к его использованию является свойство alignment (выравнивание). Если вы задали ему значение Alignment.center, поздравляем, вы только что создали клон виджета Center! Значением свойства alignment является экземпляр класса Alignment, но значение Alignment.center – это статический экземпляр, значения x и y которого равны 0 и 0. Значения x и y – это то, как вы определяете выравнивание, при этом 0, 0 – это центр прямоугольной области, которую занимает виджет Align. Если у вас есть значе-

ния –1 и –1, то это верхний левый угол прямоугольника, а 1, 1 – нижний правый (начинаете понимать, как это работает?)

Наконец, у нас еще есть виджет Column, который я оставил напоследок, потому что практически все, что обсуждалось для виджета Row, применимо и к Column. Очевидное отличие состоит в том, что его дочерние элементы располагаются вертикально. Почти все, что применимо для Row, также подходит Column, разве что в вертикальном направлении. Конечно, вы можете вкладывать виджеты Row в виджеты Column и, наоборот, создавать произвольные сложные макеты интерфейса, именно к этому и сводится большая часть разработки пользовательского интерфейса во Flutter!

Container, Padding, Transform

Виджет Container наряду с Row и Column (без учета виджетов уровня приложения или страниц), вероятно, один из наиболее популярных виджетов, предлагаемых Flutter для компоновки вашего пользовательского интерфейса. Он мастер на все руки в том смысле, что поддерживает большое количество возможностей, доступных в других виджетах.

Например, что вы будете делать, если захотите указать отступы вокруг второго виджета Text из предыдущего примера? Ответ прост – обернуть его в виджет Padding:

```
Padding(padding : EdgeInsets.all(20), child : Text("Child2") )
```

Мы задаем отступы в 20 пикселей вокруг текста (сверху, снизу, слева и справа, посредством EdgeInsets.all(20)). Вы можете использовать only() вместо all(), чтобы указать отступы слева, справа, сверху и снизу по отдельности, или можете использовать absolute(), чтобы указать вертикальное и горизонтальное значение, которое будет применено одновременно к «верхнему и нижнему» и/или «левому и правому» отступам.

Что, если вы хотите увеличить этот текст до 200 %? Тут в игру вступает виджет Transform:

```
Transform.scale(scale : 2, child : Text("Child2"))
```

Статический метод scale() возвращает новый виджет Transform с масштабным коэффициентом 2, это в два раза больше обычного.

Вы спросите, какое отношение это имеет к Container? Ну, это связано с тем, что Container поддерживает из коробки всю эту функциональность и многое другое! Например, мы можем имитировать виджет Center, заменив его следующим Container:

```
Container(alignment : Alignment.center, child...
```

Мы также можем масштабировать с ним Text:

```
Container(transform : Matrix4.identity().scale(2.0), child :
Text("Child2") )
```

Синтаксис немного сложнее, потому что теперь мы должны использовать матричную математику для ручного масштабирования дочернего виджета, хотя виджет Transform автоматически делает это за нас (хорошая причина выбрать его!).

Аналогично, если вы хотите добавить отступ:

```
Container(padding : EdgeInsets.all(20.0), child : Text("Child2"))
```

Разработчики Flutter часто используют Container с чем-нибудь еще, и это работает. Тем не менее я бы посоветовал использовать специализированные виджеты, а Container только в качестве запасного «универсального» варианта или, например, если у вас нет конкретных целей и нужно просто обернуть другой виджет.

ConstrainedBox, FittedBox, RotatedBox, SizedBox

Flutter предлагает несколько компонентов «box», которые во многом похожи на Row, Column и Container, но предоставляют различные возможности позиционирования, определения размера и других манипуляций для одного дочернего элемента.

ConstrainedBox используется для накладывания дополнительных ограничений на его дочерний элемент. Допустим, вы хотите, чтобы второй виджет Text в предыдущем примере занимал минимум 200 пикселей в ширину, вы можете обернуть его в ConstrainedBox и задать это ограничение:

```
ConstrainedBox(constraints : BoxConstraints(minWidth : 200.0), child :
Text("Child2"))
```

Класс BoxConstraints предлагает свойства для указания ограничений по ширине и высоте, причем minWidth (минимальная ширина), minHeight (минимальная высота), maxWidth (максимальная ширина) и maxHeight (максимальная высота) являются наиболее часто используемыми.

Далее следует FittedBox, который масштабирует и позиционирует свой дочерний элемент внутри себя в соответствии со свойством fit («поместить»). Он может быть полезен, например, в предыдущем примере с масштабированием текста, когда виджет Text не увеличивался или не размещался на экране так, как мы ожидали.

Виджет FittedBox может решить эту проблему, и он прекрасно работает в сочетании с виджетом ConstrainedBox:

```
ConstrainedBox(constraints : BoxConstraints(minWidth: 200.0), child :
FittedBox(fit: BoxFit.fill, child : Text("Child2") ) )
```

Это масштабирует виджет Text, но, в отличие от предыдущего примера, он также перемещает его, позволяя остаться выровненным по центру. Он тоже увеличивает размер текста, чтобы его минимальная ширина была 200 пикселей. Если вы сравните этот код с прошлым примером масштабирования, то увидите, что такое поведение логичнее и больше похоже на ожидаемое.

Точно так же виджет RotatedBox дает нам возможность поворачивать дочерний элемент:

```
RotatedBox(quarterTurns : 3, child : Text("Child2") )
```

Свойство quarterTurns – это число четвертей оборота по часовой стрелке, на которые поворачивается дочерний элемент (например, 3 четверти – 270 градусов). Итак, если вам нужна четверть оборота, этот виджет идеален, но если вам нужны произвольные градусы, придется иметь дело с Transform.

Наконец, виджет SizedBox задает дочернему элементу определенные ширину и высоту:

```
SizedBox(width : 200, height : 400, child : Text("Child2") )
```

Попробуйте запустить этот код, и вы заметите, что в результате виджет Text будто уплывает вверх и влево от своей обычной позиции. По умолчанию текст в виджете Text выровнен по левому верхнему углу, поэтому при присвоении ему размера происходит «уплывание» в левый верхний угол родительского виджета, который теперь занимает указанные 200×400 пикселей. Что именно виджет SizedBox сделает со своим дочерним элементом, будет зависеть от того, как этот дочерний элемент реагирует на указание ему ширины и высоты (при условии что он вообще поддерживает эти свойства).

Divider

Виджет Divider (разделитель) очень прост. Он отображает горизонтальную линию толщиной в один пиксель с небольшим отступом с обеих сторон. Просто добавьте его между элементами Text:

```
Text("Child1"),
Divider(),
Text("Child2"),
Divider(),
Text("Child3")
```

и вы ничего не увидите! Это потому, что Divider может быть только горизонтальным, а когда макет находится в Row, он не отображается. Итак, просто измените Row на Column, и вы увидите несколько красивых линий между виджетами текста!

Card

Card (карточка) – это виджет Material Design, представляющий собой прямоугольник с закругленными углами и небольшой тенью. Как правило, он используется для отображения списка информации. Использовать его очень просто. Это показано в листинге 3-2.

Листинг 3-2. *Card в действии*

```
import "packageflutter/material.dart";
```

```
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext context) {
      return MaterialApp(title : "Flutter Playground",
         home : Scaffold(
            body : Center(
               child : Card(
                  child : Column(mainAxisSize: MainAxisSize.min,
                     children : [
                        Text("Child1"),
                        Divider(),
                        Text("Child2"),
                        Divider(),
                        Text("Child3")
                     ]
                  )
               )
            )
         )
      );
   }
}
```

Вы можете заменить оператор return в предыдущем примере кода или просто посмотреть на рис. 3-2.

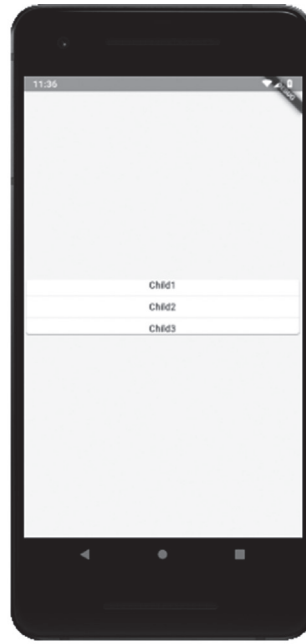


Рисунок 3-2. *Виджет Card*

Виджет `Card` не обладает большим количеством свойств, но вот самые интересные: `color` позволит вам задать цвет фона; `elevation` – установить размер тени; `shape` – изменить закругление углов.

Drawer

Виджет `Drawer` чаще всего задается в качестве значения свойства `drawer` виджета `Scaffold`, хотя это не обязательно. Он представляет собой панель Material Design, которая вызывается свайпом слева – это обычное меню. Другой виджет, `AppBar`, обычно идет вместе с `Drawer`, потому что он автоматически предоставляет соответствующий `IconButton` (виджет кнопки, на которой отображается только значок), чтобы показать и скрыть `Drawer` (что также можно сделать с помощью свайпа слева направо).

Как показано в листинге 3-3, код для `Drawer` прост, если он внутри `Scaffold`.

Листинг 3-3. *Drawer в действии*

```
import "packageflutter/material.dart";

void main() => runApp(MyApp());
class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext context) {
      return MaterialApp(title : "Flutter Playground",
```

```
        home : Scaffold(
           appBar : AppBar(
              title : Text("Flutter Playground!")
           ),
           drawer : Drawer(
              child : Column(
                 children : [
                    Text("Item 1"),
                    Divider(),
                    Text("Item 2"),
                    Divider(),
                    Text("Item 3")
                 ]
              )
           ),
           body : Center(
              child : Row(
                 children : [
                    Text("Child1"),
                    Text("Child2"),
                    Text("Child3")
                 ]
              )
           )
        )
     );
  }
}
```

Здесь вы можете увидеть `AppBar`, а также `Drawer`. При желании вы можете задать любой контент, но обычно используется `ListView`, у которого первым элементом является виджет `DrawerHeader`, отображающий информацию о пользователе. Но опять же, его использование не является обязательным. Помимо дочернего виджета, `Drawer` имеет свойство `elevation`, такое же, как у виджета `Card`. На рис. 3-3 показано, как это выглядит. До и после того, как пользователь щелкнет на значок меню (еще его называют «hamburger»).

Рисунок 3-3. *Виджет Drawer до и после раскрытия*

Это все, что нужно знать о `Drawer`!

Примечание. Виджет `CupertinoNavigationBar` – это грубый эквивалент виджета `AppBar`, который обычно используется для приложений Material (Android).

Навигация

Виджеты навигации позволяют пользователю перемещаться по вашему приложению, при необходимости ваше приложение может делать это и без его вмешательства.

Сначала поговорим о виджете `Navigator` (навигатор). Поскольку чаще всего вы запускаете свое приложение с помощью `WidgetsApp` или `MaterialApp`, вы автоматически получаете виджет `Navigator` (вы можете создать его вручную, но это не очень распространенная практика). `Navigator` – это стек, который управляет набором дочерних виджетов. Иными словами, виден только последний добавленный элемент, а остальные находятся под ним. Эти элементы являются различными экранами ваших приложений, которые называются *маршрутами* (route). Навигатор предоставляет такие методы, как `push()` и `pop()` для добавления и удаления маршрутов.

Вы уже несколько раз наблюдали использование `MaterialApp` и видели, как применяется его свойство `home`. И что? А то, что значение этого свойства – первый маршрут в вашем приложении! Вы использовали `Navigator`, даже не подозревая об этом!

Вы можете сами добавить маршруты в Navigator с помощью push(). Например:

```
Navigator.push(context, MaterialPageRoute<void>(
   builder : (BuildContext context) {
      return Scaffold(
         body : Center(child : Text("My new page"))
      );
   }
));
```

Вы всегда используете виджет MaterialPageRoute при вызове push(), и это требует применения паттерна builder. Он необходим для осуществления перехода на новую страницу, так как заданный виджет (в нашем примере Scaffold) будет создаваться и пересоздаваться много раз в разных контекстах. Поэтому прямое указание дочерних элементов может привести к ситуации, когда ваш код будет выполняться в неверном контексте. Паттерн builder позволяет избежать этой проблемы, поскольку всегда использует нужный контекст.

Когда вы вызываете push() для добавления маршрута в стек, он сразу отображается. Чтобы вернуться к предыдущему маршруту, вы должны вызвать метод pop() у Navigator, передавая текущий контекст сборки:

```
Navigator.pop(context);
```

Маршрут всегда должен начинаться со знака /, а после должно идти название страницы, которую вы хотите открыть. Например, вы добавляете маршрут в MaterialApp следующим образом:

```
routes : <String, WidgetBuilder> {
   "/announcements" : (BuildContext bc) => Page(title : "P1"),
   "/birthdays" : (BuildContext bc) => Page(title : "P2"),
   "/data" : (BuildContext bc) => Page(title : "Pe"),
}
```

Теперь вы можете вызвать маршрут по имени:

```
Navigator.pushNamed(context, "/birthdays");
```

Вы также можете вкладывать виджеты Navigator друг в друга. Иными словами, маршрут Navigator может иметь дочерний Navigator. Это позволяет осуществлять дополнительную навигацию, не меняя первичную.

BottomNavigationBar

Иногда Navigator не лучший выбор для навигации по приложению. Важным фактором является отсутствие визуального представления этого виджета. К счастью, Flutter предлагает несколько виджетов для визуальной навигации, один из них – BottomNavigationBar. Он представляет собой панель внизу экра-

на со значками и текстом, по которым пользователь может нажимать, чтобы перемещаться по вашему приложению.

Фактически этот виджет *не выполняет* никакой навигации, поэтому его название немного некорректно. В действительности вся навигация осуществляется *внутри* вашего кода. Тем не менее обычно `BottomNavigationBar` используется для навигации, и вот один из таких способов.

Листинг 3-4. *BottomNavigationBar*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
   MyApp({Key key}) : super(key : key);
   @override
   _MyApp createState() => _MyApp();
}

class _MyApp extends State {
   var _currentPage = 0;

   var _pages = [
      Text("Page 1 - Announcements"),
      Text("Page 2 - Birthdays"),
   Text("Page 3 - Data")
   ];

   @override
   Widget build(BuildContext context) {
      return MaterialApp(title : "Flutter Playground",
         home : Scaffold(
            body : Center(child : _pages.elementAt(_currentPage)),
            bottomNavigationBar : BottomNavigationBar(
               items : [
                  BottomNavigationBarItem(
                     icon : Icon(Icons.announcement),
                     title : Text("Announcements")
                  ),
                  BottomNavigationBarItem(
                     icon : Icon(Icons.cake),
                     title : Text("Birthdays")
                  ),
                  BottomNavigationBarItem(
                     icon : Icon(Icons.cloud),
                     title : Text("Data")
                  ),
               ],
```

```
                currentIndex : _currentPage,
                fixedColor : Colors.red,
                onTap : (int inIndex) {
                   setState(() { _currentPage = inIndex; });
                }
             )
          )
       );
   }
}
```

На рис. 3-4 показано, что производит этот код.

Рисунок 3-4. *Виджет BottomNavigationBar*

Итак, мы начнем с создания виджета с состоянием. Это необходимо, потому что виджет верхнего уровня создается один раз, и если он не имеет состояния, то при нажатии на вкладку ничего не изменится. Следовательно, мы должны сделать виджет с состоянием, чтобы обеспечить корректное поведение. Вы помните, что при работе с состоянием необходимо создавать два класса: первый должен наследоваться от `StatefulWidget`, а второй от `State`. Это может показаться странным (по мне, так точно!), но класс `_MyApp`, который на самом деле

определяет вкладки на панели, наследуется от State, а не от StatefulWidget. Считаете вы это странным или нет, но фишка в том, чтобы распознать паттерн. Обычно класс StatefulWidget выглядит так, как показано в примере, а класс State похож на наследников StatelessWidget, которых вы уже видели ранее.

Переменная, которая хранит текущий индекс, находится в классе _MyApp и называется _currentPage. Это значение передается методу elementAt() из списка _pages. А уже это определяет, какой именно элемент отображается внутри виджета Center (который может быть целым набором виджетов, а не одним Text). Свойство bottomNavigationBar виджета Scaffold принимает в качестве значения экземпляр BottomNavigationBar, в котором есть свойство items. Это свойство представляет собой список виджетов BottomNavigationBarItem. Каждый из них может иметь значок и заголовок. Flutter поставляет набор иконок в классе Icons, поэтому вам не нужно париться по этому поводу! Так что вам не придется запоминать или искать значки, когда они вам понадобятся! Свойство currentIndex элемента BottomNavigationBar сообщает нам, какой из элементов на панели выбран в настоящий момент, а свойство fixedColor задает цвет выбранному элементу.

Теперь, когда пользователь нажимает на один из элементов, ничего не происходит. Чтобы исправить это, существует свойство onTap. Это функция, которой передается индекс выбранного элемента. Теперь мы знаем, какой элемент из _pages нужно отобразить, но как обновится значение _currentPage? Здесь в игру вступает метод setState(), расширяющий класс State. Все, что нам нужно сделать, – это вызвать этот метод и обновить в нем переменную _currentPage. Виджет сам обновится. Поскольку _currentPage теперь другой, то пользователь перейдет на новую страницу.

TabBar (CupertinoTabBar) и TabBarView (CupertinoTabView)

Другой вездесущий элемент навигации – это TabBar и его эквивалент iOS Cupertino TabBar. Вместе с ними используются виджеты TabBarView и CupertinoTabView соответственно (обратите внимание, что здесь и далее мы будем говорить только о TabBar и TabBarView, но все сказанное относится к CupertinoTabBar и CupertinoTabView).

TabBarView представляет собой стек экранов (или views, если хотите), где виден только один, а пользователь может перемещаться между ними. Чтобы увидеть остальные экраны, нужно взаимодействовать с TabBar. Нужно тапнуть на значок одной из вкладок или свайпнуть для переключения между ними. Обычно между экранами есть анимация, например слайд.

Давайте рассмотрим пример в листинге 3-5 и на рис. 3-5.

Листинг 3-5. *Виджет TabBar*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
```

```
   @override
   Widget build(BuildContext context) {

      return MaterialApp(
         home : DefaultTabController(
            length : 3,
            child : Scaffold(
               appBar : AppBar(title : Text("Flutter Playground"),
                  bottom : TabBar(
                     tabs : [
                        Tab(icon : Icon(Icons.announcement)),
                        Tab(icon : Icon(Icons.cake)),
                        Tab(icon : Icon(Icons.cloud))
                     ]
                  )
               ),
               body : TabBarView(
                  children : [
                     Center(child : Text("Announcements")),
                     Center(child : Text("Birthdays")),
                     Center(child : Text("Data"))
                  ]
               )
            )
         )
      );
   }
}
```

Виджет `TabController` будет сам отвечать за отслеживание текущей вкладки и отображение содержимого каждой из них. Вы можете создать его вручную, но это потребует дополнительной работы, поэтому большую часть времени вы просто будете использовать виджет `DefaultTabController` в качестве значения свойства `home` виджета `MaterialApp`, который обо всем позаботится.

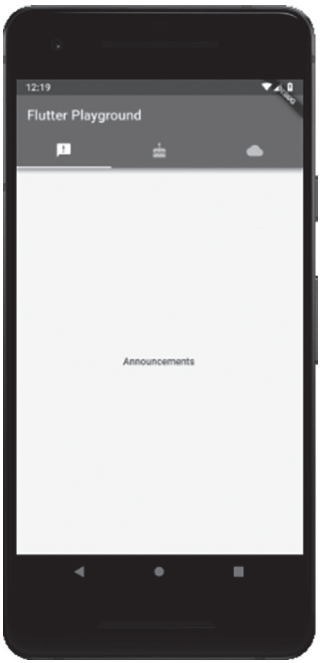


Рисунок 3-5. *Виджет TabBar*

Однако, сделав это, вы должны сообщить `TabController` через свойство `length` о том, сколько всего вкладок существует. После этого вам нужно описать каждую вкладку для `TabController`, которому вы предоставляете массив (свойство `tabs`), где каждый элемент – это виджет `Tab`. Здесь мы просто указываем значок для каждого.

Как только сами вкладки определены в `TabController`, мы должны указать контент для каждой из них, и это делается с помощью установки виджета `TabBarView` в качестве свойства `body`. Каждый дочерний элемент в списке может быть настолько сложным деревом виджетов, насколько вам нужно. В примере выше заданы же только виджеты `Center` с дочерними `Text` внутри.

Далее взаимодействие между вкладками происходит автоматически, и пользователь может свободно перемещаться между ними.

Stepper

Последний виджет навигации, который я хочу обсудить, – это виджет `Stepper`. Он необходим, чтобы провести пользователя через определенную последовательность событий. Представьте себе процесс покупки на Amazon или у другого онлайн-продавца. Сначала необходимо ввести информацию о доставке, потом нажать кнопку **Продолжить**. Затем вы вводите информацию об оплате и нажимаете кнопку **Продолжить**. Наконец, вы должны решить, нужна ли вам подарочная упаковка и другие услуги. Вы нажимаете кнопку в последний раз

и делаете заказ. Это последовательность из трех шагов, а Stepper обеспечивает ту же функциональность в приложении Flutter.

Посмотрите на пример этого кода в листинге 3-6.

Листинг 3-6. *Шаг за шагом с виджетом Stepper*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
   MyApp({Key key}) : super(key : key);
   @override
   _MyApp createState() => _MyApp();
}

class _MyApp extends State {

   var _currentStep = 0;

   @override
   Widget build(BuildContext context) {
      return MaterialApp(title : "Flutter Playground",
         home : Scaffold(
            appBar : AppBar(title : Text("Flutter Playground")),
            body : Stepper(
               type : StepperType.vertical,
                  currentStep : _currentStep,
               onStepContinue : _currentStep < 2 ?
                  () => setState(() => _currentStep += 1) : null,
               onStepCancel : _currentStep > 0 ?
                  () => setState(() => _currentStep -= 1) : null,
               steps : [
                  Step(
                     title : Text("Get Ready"), isActive : true,
                     content : Text("Let's begin...")
                  ),
                  Step(
                     title : Text("Get Set"), isActive : true,
                     content : Text("Ok, just a little more...")
                  ),
                  Step(
                     title : Text("Go!"), isActive : true,
                     content : Text("And, we're done!")
                  )
               ]
            )
         )
```

```
      );
   }
}
```

На рис. 3-6 показано, как это выглядит.

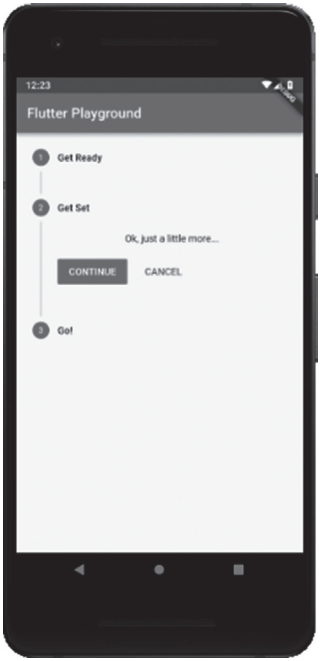


Рисунок 3-6. *Мы уже столько нашагали (и это только разминка)!*

Пока мы не доберемся до `Stepper`, вложенного в `Scaffold`, большая часть кода должна быть вам более-менее понятна. Сначала укажите, хотите ли вы, чтобы ваши шаги отображались вертикально или горизонтально через свойство `type`. С помощью переменной `_currentStep` укажите, на каком этапе находится пользователь в данный момент. Это виджет с состоянием, так как значение переменной `_currentStep` определяет то, какой шаг отображается, что точно соответствует понятию «состояние» во Flutter.

Еще нам следует предоставить код для `Stepper`, обрабатывающий нажатия пользователя на кнопки **Continue** (Продолжить) и **Cancel** (Отмена), реализованные `Stepper`. Значение `_currentStep` увеличивается при нажатии кнопки **Continue**, пока мы не окажемся на последнем шаге, и уменьшается при нажатии кнопки **Cancel**, пока мы не вернемся к первому этапу. Это позволяет пользователю произвольно перемещаться по последовательности.

Далее нам нужно определить шаги последовательности; каждый шаг представлен виджетом `Step`. Он принимает текст заголовка, чтобы отобразить его рядом с кругом. Свойство `isActive` делает серым виджет отдельного шага, если установлено значение `false` (обратите внимание, что это ничего не ме-

няет, кроме изменения цвета круга – ваш код должен сам обрабатывать необходимые вам сценарии). После этого мы определяем контент, который может быть таким сложным деревом виджетов, каким захотите.

Каждый Step может иметь подзаголовок (свойства subtitle), если необходимо, и свойство state, которые определяют стиль компонента и возможность взаимодействовать с ним. Опять же, ваш код должен обеспечивать функциональность для поддержки этого процесса. Также обратите внимание, что виджет Stepper предоставляет свойство onStepTapped, которое представляет собой функцию, вызываемую нажатием пользователя на один из шагов. Очевидно, что чаще всего вы пишете код для прямого перехода к выбранному шагу.

Ввод данных

Виджеты ввода (input) используются для получения данных, введенных пользователем (очевидно!). Flutter предоставляет широкий спектр таких виджетов, некоторые из них могут вас удивить.

Form (форма)

Во Flutter рассмотрение механизмов ввода данных начинается с виджета Form. Что не совсем верно, ведь Form не является обязательным. Но поскольку он предлагает такую опцию и часто используется для ввода данных, давайте поговорим о нем, как если бы он был для нас обязательным!

Form – это контейнер для полей формы: есть виджет FormField, который оборачивает все поля ввода и делает их дочерними. Преимущество этого виджета в том, что он предоставляет вам расширенные возможности, такие как сохранение данных, их сброс и валидацию. Без Form нам пришлось бы реализовывать это вручную, так почему бы не использовать готовое решение?

Давайте рассмотрим пример Form – это типичная форма ввода, которая также продемонстрирует и другие механизмы, связанные с вводом данных.

Листинг 3-7. *Виджет Form и его элементы*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
   MyApp({Key key}) : super(key : key);
   @override
   _MyApp createState() => _MyApp();
}

class LoginData {
   String username = "";
   String password = "";
}
```

```
class _MyApp extends State {

   LoginData _loginData = new LoginData();
   GlobalKey<FormState> _formKey = new GlobalKey<FormState>();

   @override
   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         body : Container(
            padding : EdgeInsets.all(50.0),
            child : Form(
               key : this._formKey,
               child : Column(
                  children : [
                     TextFormField(
                        keyboardType :
                           TextInputType.emailAddress,
                        validator : (String inValue) {
                           if (inValue.length == 0) {
                              return "Please enter username";
                           }
                           return null;
                        },
                        onSaved: (String inValue) {
                           this._loginData.username = inValue;
                        },
                        decoration : InputDecoration(
                           hintText : "none@none.com",
                           labelText : "Username (eMail address)"
                        )
                     ),
                     TextFormField(
                        obscureText : true,
                           validator : (String inValue) {
                              if (inValue.length < 10) {
                                return "Password must be >=10 in length";
                              }
                              return null;
                           },
                           onSaved : (String inValue) {
                              this._loginData.password = inValue;
                           },
                           decoration : InputDecoration(
                              hintText : "Password",
                              labelText : "Password"
                           )
```

```
                    ),
                    RaisedButton(
                        child : Text("Log In!"),
                        onPressed : () {
                            if (_formKey.currentState.validate()) {
                              _formKey.currentState.save();
                              print("Username: ${_loginData.username}");
                              print("Password: ${_loginData.password}");
                            }
                        }
                    )
                ]
            )
        )
    ));
  }
}
```

Посмотрите на результат выполнения этого кода на рис. 3-7. Это не суперкруто, но приятно видеть, что код делает именно то, что вы от него хотите.

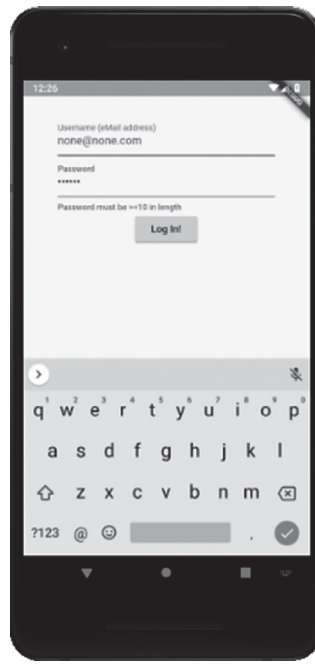


Рисунок 3-7. *Виджет Form*

Следом за стандартными функциями import и main() мы имеем дело со Stateful Widget, поэтому у нас есть обычное определение класса для него. Но, прежде чем мы дойдем до класса State, который, как вы знаете, идет вместе с ними, рассмотрим еще один небольшой класс: LoginData. Экземпляр этого класса будет хранить введенные имя пользователя и пароль. Это типичный шаблон при работе с формами Flutter. Он хорош тем, что упрощает работу, объединяя все входные данные в одном объекте.

Далее идет класс состояния (State) _MyApp. Он выглядит так же, как и любой другой класс с состоянием, который вы видели ранее, но здесь есть несколько отличий. Во-первых, у нас есть экземпляр LoginData, о котором я упоминал. Затем идет GlobalKey – уникальный ключ, который не повторяется нигде в текущем приложении. Обычно такой ключ используется в качестве поля key у виджета, что необходимо для замены одного виджета другим в дереве виджетов. Если тип (runtimeType) и ключ (key) этих виджетов равны, то новый виджет заменяет старый, обновляя родительский элемент. В противном случае старый элемент удаляется из дерева объектов, а вместо него создается новый и вставляется в дерево. Использование GlobalKey в качестве ключа (в отличие от LocalKey, который обеспечивает уникальность только в текущем родительском элементе) позволяет элементу перемещаться по дереву виджетов без потери состояния. Когда найден новый виджет (это означает, что его key и runtimeType не совпадают с предыдущим виджетом в том же месте дерева) с тем же GlobalKey в другом месте дерева, то новый виджет перемещается на его место.

Хотелось бы сказать, что свойство key чрезвычайно полезно, потому что оно дает возможность перемещать виджеты, не изменяя их состояние, хотя, если честно, этим редко пользуются. Например, добавьте новую переменную в класс _MyApp следующим образом:

```
GlobalKey _btnKey = newGlobalKey();
```

Затем в RaisedButton добавьте key, ссылающийся на него:

```
key: _btnKey,
```

Наконец, в обработчике onPressed кнопки сделайте следующее:

```
print ((
   _btnKey.currentWidget as RaisedButton).child as Text).data
);
```

Результатом будет выведенный из данного виджета текст. Чтобы все работало, мы должны привести _btnKey.currentWidget к RaisedButton, используя ключевое слово as, так как currentWidget – это Widget, который не имеет свойства Text. После приведения можно увидеть свойство data, это будет наш текст. Таким образом, если вы знаете ключ (будь то GlobalKey или LocalKey), то можете получить доступ к любому свойству любого виджета или выполнить вызов его методов. Но я думаю, что не стоит делать подобное, это чуждо реактивной

природе Flutter. Для управления такими взаимодействиями принято использовать состояние. Но это будет козырем в вашем рукаве. А еще использование key поможет вам лучше понять внутренности Flutter.

Далее идет метод build(). Он выглядит знакомо, но теперь у нас в дереве есть виджет Form. Обычно виджет, который является единственным дочерним элементом другого виджета, не нуждается в названии, поэтому вы не видели свойство key ранее, но оно является ссылкой на _formKey.

Как видите, у Form есть свойство child, поэтому если мы хотим задать в Form несколько полей, мы просто добавим контейнер, в данном случае Column.

В этом Column есть три поля ввода и кнопка **Log In**. Первые два используют виджет TextFormField, эффективно объединяющий два других: FormField, который должен оборачивать все поля в форме, и TextField – виджет для ввода текста пользователем (есть также соответствующий CupertinoTextField). Имя пользователя является TextFormField; поскольку именем пользователя часто является email (распространенная, но не особенно хорошая практика, не обеспечивающая безопасность), то мы хотим, чтобы отображаемая клавиатура была удобна для ввода электронной почты. Свойство keyboardType позволяет нам это сделать. В классе TextInputType есть несколько констант для различных типов клавиатуры, в данном случае применяется emailAddress.

Виджет TextFormField также имеет свойство validator, которое определяет функцию, выполняющую проверку поля при нажатии кнопки **Log In**. Эта функция может делать все, что вы хотите, но она должна либо вернуть строку с сообщением об ошибке (красный текст под полем ввода), либо null, если ошибок нет.

Обратите внимание, что сами данные в поле никогда нигде не сохраняются; они только временно существуют в Form. Это не очень хорошо, и чтобы решить эту проблему, нам нужно реализовать функцию для свойства onSaved. Она будет срабатывать при вызове у Form метода save(), позже вы увидите, как это происходит (на самом деле вызов произойдет не в самой Form, но это вы тоже увидите в ближайшее время). Функция обработчика onSaved просто сохранит значение inValue, переданное в поле username объекта _loginData.

Хотя это и необязательно, но можно использовать свойство InputDecoration для изменения стиля текста. Также часто используется свойство hintText для отображения подсказки (отображается в поле, пока ничего не введено) и свойство labelText, надпись, отображаемая над полем.

Поле пароля (password) аналогично полю имени пользователя (username), за исключением того, что, будучи паролем, введенные пользователем символы не должны отображаться на экране, поэтому свойство obscureText имеет значение true. Во втором случае у нас есть другая функция validator, выполняющая проверку данных, и обработчик onSaved для сохранения данных, а также экземпляр InputDecoration.

Наконец, мы переходим к кнопке **Log In**. С ней мы делаем пару интересных вещей. Во-первых, вызывается метод validate() через переменную _formKey. Она даст нам ссылку на виджет, внутри которого есть свойство currentState,

которое содержит значение, введенное в форме. Это объект (в нашем случае – строка, введенная пользователем), для которого фактически вызывается метод `validate()` виджета, и так как каждое поле ввода имеет подобную функцию, то мы можем пройтись по всем виджетам формы и провести проверку введенных данных, при необходимости отображая ошибки. Также мы вызываем `save()` у `currentState`, что приводит к тому, что все обработчики `onSaved` запускаются и данные формы сохранятся в `_loginData`. Наконец, мы печатаем эту информацию в консоль, чтобы убедиться, что все сработало должным образом.

Checkbox

Да, вы знакомы с `Checkbox`! Это маленький ящик, который нужно… подождите, подождите… отметить!

Я очень рад, что у Flutter есть такой виджет, это действительно замечательно.

Примечание. Листинг 3-8 демонстрирует `Checkbox`, а также переключатель `Switch`, `Slider` и `Radio`, а рис. 3-8 покажет, как это выглядит. В следующих разделах вы с ними познакомитесь.

Листинг 3-8. *Checkbox, Switch, Slider и Radio*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
   MyApp({Key key}) : super(key : key);
   @override
   _MyApp createState() => _MyApp();
}

class _MyApp extends State {

   GlobalKey<FormState> _formKey = new GlobalKey<FormState>();
   var _checkboxValue = false;
   var _switchValue = false;
   var _sliderValue = .3;
   var _radioValue = 1;

   @override
   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         body : Container(
            padding : EdgeInsets.all(50.0),
            child : Form(
               key : this._formKey,
               child : Column(
                  children : [
```

```
            Checkbox(
               value : _checkboxValue,
               onChanged : (bool inValue) {
                  setState(() { _checkboxValue = inValue; });
               }
            ),
            Switch(
               value : _switchValue,
               onChanged : (bool inValue) {
                  setState(() { _switchValue = inValue; });
               }
            ),
            Slider(
               min : 0, max : 20,
               value : _sliderValue,
               onChanged : (inValue) {
                  setState(() => _sliderValue = inValue);
               }
            ),
            Row(children : [
               Radio(value : 1, groupValue : _radioValue,
               onChanged : (int inValue) {
                  setState(() { _radioValue = inValue; });
               }
            ),
            Text("Option 1")
         ]),
         Row(children : [
            Radio(value : 2, groupValue : _radioValue,
               onChanged : (int inValue) {
                  setState(() { _radioValue = inValue; });
               }
            ),
            Text("Option 2")
         ]),
         Row(children : [
            Radio(value : 3, groupValue : _radioValue,
               onChanged : (int inValue) {
                  setState(() { _radioValue = inValue; });
               }
            ),
            Text("Option 3")
         ])
      ]
)
```

```
          )
      )
  ));
}
```

Рисунок 3-8. *Группа виджетов Checkbox, Switch, Slider и Radio*

Да, это все! Если `StatefulWidget` содержит переменную `checkboxValue`, значит, все идет хорошо. Кроме того, вы можете реализовать функцию `onChanged`, чтобы описать дополнительную логику при изменении состояния. И у `Checkbox` есть специальное свойство `tristate` типа `bool`, которое допускает три значения: отмеченный (`true`), неотмеченный (`false`) и `null`. Последнее состояние будет отображаться пунктиром.

Следует отметить, что виджет `Checkbox` не поддерживает отображения текстовой подсказки, хотя это часто встречается в таких компонентах. Чтобы достичь этого, вам нужно создать текст самостоятельно, поместив `Checkbox` и виджет `Text` в контейнер `Row` (при условии что вы хотите добавить метку рядом с `Checkbox`, в противном случае используйте Column или другую структуру компоновки).

Switch (CupertinoSwitch)

Виджет `Switch` и его аналог для iOS `CupertinoSwitch` похож на `Checkbox`, но с другим визуальным представлением: он выглядит как маленький переклю-

чатель. Фактически, если вы вернетесь к коду и замените Checkbox на Switch, все будет работать!

Обратите внимание, что если значение свойства onChanged не задано (установлено в null), Switch будет отключен и не будет реагировать на взаимодействие с пользователем, так же как и Checkbox.

Slider (CupertinoSlider)

Виджет Slider представляет собой ползунок, который пользователь передвигает для выбора значения из предопределенного диапазона. CupertinoSlider – это версия для iOS, и работает она аналогично. Вот вам пример:

```
Slider(
   min : 0, max : 20,
   value : _sliderValue,
   onChanged : (inValue) {
      setState(() => _sliderValue = inValue);
})
```

Свойства min и max определяют нижний и верхний пределы диапазона значений, между которыми находится текущее значение. Это главные свойства Slider. Так как Slider – это StatefulWidget, то его текущее значение должно быть переменной в текущем State. Наконец, onChanged требуется для установки значения в State при перемещении Slider.

Существуют такие свойства, как activeColor и inactiveColor для настройки цвета активной и неактивной частей Slider. Вы также можете задать количество делений в пределах диапазона (при нулевом значении Slider автоматически создает деления, представляющие собой непрерывный и дискретный набор значений в диапазоне от минимального до максимального значений). Существуют также обработчики событий, когда пользователь начинает перемещать Slider(onChangeStart) и когда он его отпускает (onChangeEnd).

Radio

О подобном размышлять немного странно, но я в таком возрасте, что могу ностальгировать по временам, когда были автомобильные радиоприемники с рядом кнопок, по одной на сохраненную радиостанцию, и когда вы нажмете одну, остальные выскочат обратно. Держу пари, что многие из вас никогда такого не видели! А вот во Flutter есть виджет под названием Radio, который работает по тому же принципу!

Виджет Radio очень похож на CheckBox или Switch, но, в отличие от них, он никогда не существует самостоятельно. В виджете Radio всегда есть один или несколько родственных виджетов Radio, и они являются взаимоисключающими: выбор одного любого Radio приводит к отмене выбора другого в его группе.

```
Column(children : [
   Row(children : [
```

```
      Radio(value : 1, groupValue : _radioValue,
         onChanged : (int inValue) {
            setState(() { _radioValue = inValue; });
         }
      ),
      Text("Option 1")
   ]),
   Row(children : [
      Radio(value : 2, groupValue : _radioValue,
         onChanged : (int inValue) {
            setState(() { _radioValue = inValue; });
         }
      ),
      Text("Option 2")
   ]),
   Row(children : [
      Radio(value : 3, groupValue : _radioValue,
         onChanged : (int inValue) {
            setState(() { _radioValue = inValue; });
         }
      ),
      Text("Option 3")
   ])
])
```

Здесь присутствуют три виджета Radio, каждый со своим текстовым обозначением. Обратите внимание, что у всех одинаковое значение свойства groupValue. Так задумано: благодаря тому что все они имеют одну и ту же ссылку _radioValue, они становятся частью одной и той же группы, что передает им взаимную исключительность (один включили, остальные выключились), о которой я упоминал. У каждого из них есть свое значение, поэтому при выборе первого Radio его значение передается в _radioValue благодаря вызову setState() в обработчике onChanged. Код, использующий виджеты Radio, может проверить это значение, чтобы определить, какое из них было выбрано.

Выбор даты и времени (CupertinoDatePicker, CupertinoTimerPicker)

Выбор даты и времени в приложении – это обычное дело, поэтому и Flutter предоставляет для этого готовые виджеты. Точнее, он предоставляет функции вызова, чтобы показать компоненты пользовательского интерфейса для этой цели, по крайней мере на Android. На этой платформе у нас есть функции showDatePicker() и showTimePicker(), как в листинге 3-9.

Листинге 3-9. *Выбор даты и времени*

```
import "package:flutter/material.dart";
```

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
      return MaterialApp(home : Scaffold(body : Home()));
   }
}

class Home extends StatelessWidget {

   Future<void> _selectDate(inContext) async {
      DateTime selectedDate = await showDatePicker(
         context : inContext,
         initialDate : DateTime.now(),
         firstDate : DateTime(2017),
         lastDate : DateTime(2021)
      );
      print(selectedDate);
   }

   Future<void> _selectTime(inContext) async {
      TimeOfDay selectedTime = await showTimePicker(
         context : inContext,
         initialTime : TimeOfDay.now(),
      );
      print(selectedTime);
   }
   @override
   Widget build(BuildContext inContext) {
      return Scaffold(
         body : Column(
            children : [
               Container(height : 50),
               RaisedButton(
                  child : Text("Test DatePicker"),
                  onPressed : () => _selectDate(inContext)
               ),
               RaisedButton(
                  child : Text("Test TimePicker"),
                  onPressed : () => _selectTime(inContext)
               )
            ]
         )
      );
   }
}
```

Обе эти функции асинхронны, поэтому мы используем методы `_selectDate()` и `_selectTime()`, которые вызываются двумя кнопками в основном макете.

Рисунок 3-9. *Выбор даты и времени*

Как видите (и в коде, и на рис. 3-9), они используют `showDatePicker()` и `showTimePicker()`. Для первого требуется контекст сборки, `initialDate`, выбранный по умолчанию, а также `firstDate` и `lastDate`, которые предоставляют выбор (в данном случае – годы). Объект `DateTime` возвращается и отображается. Для `showTimePicker()` необходимы только контекст сборки и `initialTime`.

Виджеты `CupertinoDatePicker` и `CupertinoTimerPicker` для iOS представляют собой обычные виджеты, поэтому функции для их вызова не нужны.

Обратите внимание, что есть три других способа выбрать дату, доступных на Android: `DayPicker` для выбора дня месяца, `MonthPicker` для выбора месяца и `YearPicker` для выбора года.

Dismissible

Виджет `Dismissible` – это элемент, от которого пользователь может избавиться, смахнув его в заданном направлении. У виджета есть свойство `direction`, которое определяет, в каком направлении его можно перетащить. Когда пользователь перетаскивает его, дочерний элемент выходит из поля зрения, и если необязательное свойство `resizeDirection` не равно `null`, `Dismissible` анимирует его высоту или ширину, в зависимости от того, что перпендикулярно направлению отклонения.

Вот пример:

```
Dismissible(
   key : GlobalKey(),
   onDisDisDismissed : (direction) { print("Goodbye!"); },
   child : Container(
      color : Colors.yellow, width : 100, height : 50,
      child : Text("Swipe me")
   )
)
```

Если хотите, вы также можете добавить специальный «фоновый виджет» с помощью свойства background. В этом случае указанный виджет прячется под дочерним элементом Dismissible и отображается, когда этот дочерний элемент перетаскивают.

Функция onDismissed будет вызываться при изменении размеров виджета до нуля, если задано свойство resizeDuration, или сразу после анимации перетаскивания, если не задано. Поле **Key** также должно быть определено, чтобы этот метод сработал; в показанном примере он не участвует, вместо него я использую экземпляр GlobalKey.

Диалоговые и всплывающие окна

Существуют способы взаимодействия с пользователем, чтобы показать ему контент, не являющийся частью текущей страницы. В общем, это диалоги (dialogs), в которых мы запрашиваем сведения, всплывающие окна (popups), где показываем срочную информацию, которая требует внимания, и сообщения (messages), в них вы встретите быстрые и временные фрагменты информации.

Tooltip

Виджет Tooltip (подсказка) удобен для отображения описания какого-либо другого виджета, когда вы выполняете соответствующее действие (например, длительное нажатие кнопки). Вы оборачиваете целевой виджет в Tooltip, например так:

```
Tooltip(
   message : "Tapping me will destroy the universe. Ouch!",
   child : RaisedButton(
      child : Text("Do Not Tap!"),
      onPressed : () { print("BOOM!"); }
   )
)
```

На самом деле некоторые виджеты предусматривают всплывающие подсказки, которые автоматически оборачивают виджет в Tooltip, но вы также можете сделать это вручную.

Обычно всплывающая подсказка отображается под виджетом, который она оборачивает, но вы можете установить значение `false` для ее свойства `preferBelow`, чтобы отменить ее (это произойдет автоматически, если для ее отображения недостаточно места). Вы также можете настроить свойство `verticalOffset`, чтобы определить расстояние между всплывающей подсказкой и ее целевым виджетом.

SimpleDialog (CupertinoDialog)

`SimpleDialog` – это всплывающий элемент, который предлагает пользователю выбор между несколькими вариантами. `SimpleDialog` может иметь текст заголовка, который отображается над параметрами. Обычно выбор визуализируется с помощью виджета `SimpleDialogOption`. Экземпляр `SimpleDialog` передается в функцию `showDialog()` для отображения, как показано в листинге 3-10.

Листинг 3-10. *SimpleDialog*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
      return MaterialApp(home : Scaffold(body : Home()));
   }
}

class Home extends StatelessWidget {
   @override
   Widget build(BuildContext inContext) {

      Future _showIt() async {
         switch (await showDialog(
            context : inContext,
            builder : (BuildContext inContext) {
               return SimpleDialog(
                  title : Text("What's your favorite food?"),
                  children : [
                     SimpleDialogOption(
                        onPressed : () {
                           Navigator.pop(inContext, "brocolli");
                        },
                        child : Text("Brocolli")
                     ),
                     SimpleDialogOption(
                        onPressed : () {
                           Navigator.pop(inContext, "steak");
```

```
                    },
                    child : Text("Steak")
                )
            ]
        );
      }
    )) {
      case "brocolli": print("Brocolli"); break;
      case "steak": print("Steak"); break;
    }
  }
  return Scaffold(
    body : Center(
      child : RaisedButton(
        child : Text("Show it"),
        onPressed : _showIt
      )
    )
  );
 }
}
```

Как это выглядит на экране, можно увидеть на рис. 3-10.

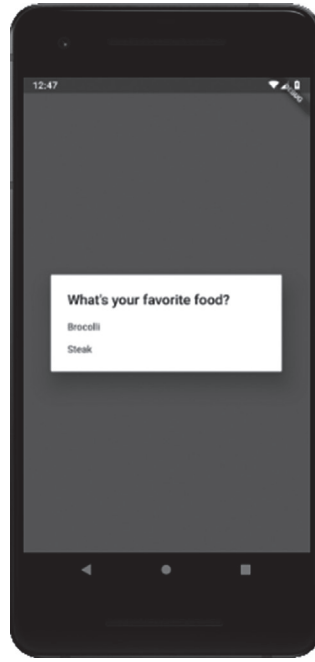


Рисунок 3-10. *Простой SimpleDialog*

Когда RaisedButton нажата, он вызывает функцию _timeForADialog(). Эта функция ожидает возвращаемого значения из showDialog() в качестве значения оператора switch. Когда пользователь тапает по одному из параметров, диалог должен быть скрыт, за что отвечает вызов Navigator.pop(). В этот момент dialog находится в верхней части стека навигации, следовательно, pop() его скрывает. Второй аргумент pop() – возвращаемое значение, которое затем обрабатывают два оператора case, чтобы вывести print() на консоль.

Существует виджет CupertinoDialog и соответствующий виджет Cupertino DialogAction для обеспечения такого же вида диалога на iOS, и работают они аналогичным образом.

Примечание. Структура здесь немного отличается от того, что вы видели раньше. Причина в том, что если вы попытаетесь вызвать showDialog() непосредственно из обработчика onPressed от RaisedButton, то обнаружите ошибку, сообщающую о необходимости MaterialLocalization. Проблема в том, что showDialog() должен вызываться в контексте сборки, где в качестве родительского элемента выступает MaterialApp, который по умолчанию включает в себя виджет MaterialLocalization, связанный с локализацией приложений. Контекст сборки внутри обработчика onPressed от RaisedButton не имеет такого элемента (даже если метод build() возвратил MaterialApp в качестве виджета верхнего уровня, который представляет другой контекст сборки, нежели переданный в сам build()). Решение заключается в создании виджета верхнего уровня MaterialApp, а затем свойства home как указателя на другой виджет, в данном случае Scaffold, который содержит виджет Home в качестве дочернего элемента (Scaffold здесь необязателен, но он необходим для других последующих примеров, которые будут на этом основаны). Таким образом, контекст компоновки для виджета верхнего уровня применяется к вызову showDialog(), у которого в качестве родительского элемента есть MaterialApp, и, таким образом, ошибки исключаются. Хотя я не делал так в большинстве примеров кода, вы видите здесь более типичную структуру. До сих пор это не имело значения, поэтому я сохраню код упрощенным (и продолжу так делать, за исключением случаев, когда это имеет значение, как здесь).

AlertDialog (CupertinoAlertDialog)

AlertDialog очень похож на SimpleDialog, за исключением того, что он предназначен для срочных ситуаций, которые требуют немедленного внимания и, как правило, не требуют какого-либо другого двоичного выбора (или вообще выбора). Основываясь на примере кода SimpleDialog, все, что нам нужно изменить, – это функция _showIt():

```
_showIt() {
   return showDialog(
      context : inContext,
      barrierDismissible : false,
      builder : (BuildContext context) {
         return AlertDialog(
            title : Text("We come in peace..."),
            content : Center(child :
```

```
            Text("...shoot to kill shoot to kill shoot to kill")
         ),
         actions : [
            FlatButton(
               child : Text(''Beam me up, Scotty!"),
               onPressed : () { Navigator.of(context).pop(); }
            )
         ]
      );
   }
);
}
```

Как и раньше, используется `showDialog()`, но на этот раз функция `builder()` возвращает `AlertDialog`. С помощью свойства `content` мы сообщаем `AlertDialog`, что отображать. Затем свойство `actions` позволяет нам предоставить массив элементов, по которым пользователь может щелкнуть, в данном случае это `FlatButton`. Как и в `SimpleDialog`, нам нужно убрать (`pop()`) диалоговое окно из стека навигатора, но на этот раз возвращать нечего, поэтому второй аргумент не требуется. Свойство `barrierDismissable`, установленное в `false`, обязует пользователя тапнуть `FlatButton`; диалоговое окно не может быть закрыто, если щелкнуть в другом месте экрана, как с `SimpleDialog`. Это подходит для информационного всплывающего окна, предназначенного для оповещения пользователя о чем-то важном.

Обратите внимание, что существует версия этого диалога для iOS, которая называется `CupertinoAlertDialog`, и вы используете ее точно так же.

SnackBar

`SnackBar` представляет собой компонент для оповещений, который показывает временное сообщение в нижней части экрана, которое, при желании, можно закрыть. Основываясь на том же примере, что и для `SimpleDialog` и `AlertDialog`, мы изменим функцию `_showIt()`, как показано здесь:

```
_showIt() {
   Scaffold.of(inContext).showSnackBar(
      SnackBar(
         backgroundColor : Colors.red,
         duration : Duration(seconds : 5),
         content : Text("I like pie!"),
         action : SnackBarAction(
            label : "Chow down",
            onPressed: () {
               print("Gettin' fat!");
            }
         )
```

```
        )
    );
}e
```

На рис. 3-11 представлен результат.

Рисунок 3-11. *Виджет SnackBar (внизу)*

Мы должны использовать вызов `Scaffold.of(inContext)`, чтобы получить ссылку на `Scaffold` – родительский виджет, вызывающий эту функцию. Этот `Scaffold` содержит метод `showSnackBar()`, который мы и вызываем. Мы можем дополнительно установить `backgroundColor`, а также продолжительность, для последнего требуется экземпляр класса `Duration` (который может принимать значения в разных форматах, таких как часы, минуты и секунды). `Content` – это текст, отображаемый на `SnackBar`. Свойство `action` необязательно, но если оно присутствует, отображается часть текста, на которую можно нажать. Обычно вы можете скрыть `SnackBar` при нажатии, но ничто не заставляет вас на него нажимать. Если вы этого не сделаете, то `SnackBar` автоматически исчезнет по истечении указанной длительности (или продолжительности по умолчанию, если она не задана).

BottomSheet (CupertinoActionSheet)

Нижние шторки, предоставляемые виджетом `BottomSheet` (дословно: нижний лист) и его аналогом для iOS `CupertinoActionSheet`, представляют собой виджеты, отображаемые в нижней части экрана, чтобы показать пользователю дополнительный контент и попросить его о выборе. Это нечто среднее меж-

ду SimpleDialog и SnackBar. Давайте продолжим изменять предыдущий пример и снова перепишем функцию _showIt(), результаты которой показаны на рис. 3-12.

```
_showIt() {
   showModalBottomSheet(context : inContext,
      builder : (BuildContext inContext) {
       return new Column(
            mainAxisSize : MainAxisSize.min,
            children : [
               Text("What's your favorite pet?"),
               FlatButton(child : Text("Dog"),
                  onPressed : () { Navigator.of(inContext).pop(); },
               ),
               FlatButton(child : Text("Cat"),
                  onPressed : () { Navigator.of(inContext).pop(); },
               ),
               FlatButton(child : Text("Ferret"),
                  onPressed : () { Navigator.of(inContext).pop(); }
               )
            ]
         );
      }
   );
}
```

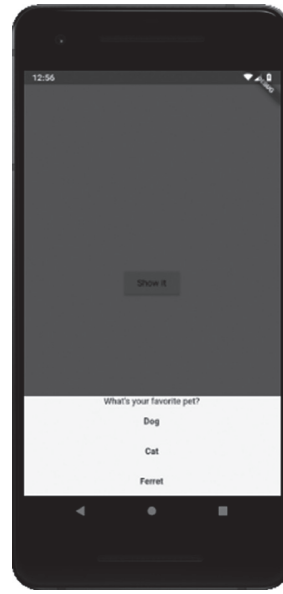


Рисунок 3-12. *BottomSheet... шторка не сверху или сбоку, а снизу!*

На самом деле есть два варианта `BottomSheet`, один из которых отображается с помощью вызова `showModalBottomSheet()`, а другой – вызова `showBottomSheet()` виджета `Scaffold`. Разница в том, что первый не позволяет пользователю взаимодействовать с другими частями приложения до тех пор, пока диалог не будет скрыт (это называется модальный, modal), тогда как другой называется «постоянный» (persistent), потому что он остается на экране до тех пор, пока не будет удален, но при этом не запрещает взаимодействие с другими элементами страницы. В любом случае `BottomSheet` создаётся одинаково. Интерактивность отображаемого контента зависит только от вас. В этом примере у меня есть текстовый заголовок с тремя виджетами `FlatButton` под ним. Нажатие на любой из них приводит к тому, что `BottomSheet` будет скрыт через вызов `Navigator.of(inContext).pop()`, который вы уже видели несколько раз.

Подведем итоги главы

Это была большая глава! Я думаю, сейчас отличный момент, чтобы немного отдохнуть.

В этой главе вы начали понимать то, какие виджеты предоставляет Flutter, но впереди еще много интересного, включая дополнительные виджеты и разные API.

Итак, перекусите, сделайте растяжку, прогуляйтесь, если нужно, и встретимся в главе 4!

ГЛАВА 4

СКАЖИ ПРИВЕТ МОЕМУ МАЛЕНЬКОМУ ДРУГУ FLUTTER. ЧАСТЬ II

В последней главе вы начали изучение виджетов, с которыми работает Flutter. А теперь мы продолжим их рассматривать и затем бегло пройдемся по некоторым API.

Виджеты стиля

Во Flutter богатая система стилизации виджетов различными способами.

Обратите внимание, что для следующих четырех разделов листинг 4-1, хотя и не напечатан здесь, будет полностью рабочим примером. На рис. 4-1 показан результат его запуска, поэтому обращайтесь к этому скриншоту при прочтении следующих четырех разделов.

Рисунок 4-1. *Демонстрация следующих четырех разделов книги (поверьте, в цвете это выглядит лучше)*

Theme и ThemeData

Виджет Theme (тема) применяет тему к дочерним виджетам. Они включают в себя цвета и настройки шрифтов.

Если посмотрите на виджет MaterialApp, то увидите, что у него есть свойство theme, которое можно использовать для объявления темы, применяемой ко всему приложению. Виджет Theme вступает в игру, когда вы хотите переопределить тему во всем приложении для необходимого подмножества виджетов. Или же вы можете просто обернуть все дерево виджетов приложения в виджет Theme и применить тему таким образом.

Есть два варианта при работе с виджетом Theme: расширение родительской темы или создание новой. Расширение родительской темы полезно, когда вы хотите только изменить подмножество элементов. Сделать это легко:

```
Theme(
   data : Theme.of(context).copyWith(accentColor : Colors.red),
   child : // Ваше дерево виджетов будет стилизовано этой темой
)
```

Метод Theme.of() – это как спросить «эй, Flutter, какая тема подойдёт к этому виджету?» Независимо от того, у какого родительского виджета есть тема, он найдет ее (и помните: даже если вы нигде не задаете тему намеренно, всегда есть тема по умолчанию). Этот метод возвращает объект ThemeData, у которого есть метод copyWith(), который возвращает новый объект ThemeData, но все переданные ему свойства переопределяют то, что было в ней ранее. Здесь мы заставляем свойство accentColor (новой ThemeData) использовать Colors.red, переопределяя все, что было раньше. Теперь все виджеты под этим Theme будут выделены красным цветом.

Создать совершенно новую Theme еще проще:

```
Theme(
   data : ThemeData( accentColor : Colors.red ),
   child : // Ваше древо виджетов будет стилизовано этой темой
);
```

Не нужно получать родительский ThemeData; вы просто создаете новый экземпляр и задаете желаемые свойства. ThemeData поддерживает много свойств, даже слишком много, чтобы перечислять их здесь, поэтому вам нужно обратиться к документации по Flutter, чтобы увидеть, что именно там есть.

Теперь, когда вы определили виджет Theme, вы все равно должны использовать его в отдельных виджетах. Но гораздо проще, когда Theme на месте:

```
Theme (
   data : ThemeData( accentColor : Colors.red ),
   child : Container(
      color : Theme.of(context).accentColor,
      child : Text(
```

```
         "Text with a background color,"
         style : Theme.of(context).textTheme.title,
      )
   )
)
```

Запомните ключевой момент: поскольку Container здесь обернут в Theme, Theme.of(context) вернет ThemeData этой темы; если Container не был обернут в Theme, то будут использоваться ThemeData уровня приложения для Theme (указанные в свойстве theme виджета MaterialApp). Если тема не была указана в MaterialApp, тогда Theme и ThemeData будут создаваться под капотом и использоваться по умолчанию.

Opacity

Виджет Opacity (полупрозрачность) очень прост: он делает дочерний элемент в нужной степени прозрачным. В качестве простой демонстрации замените второй Text в предыдущем примере:

```
Opacity(opacity: .25, child : Text("Faded") )
```

При повторном запуске вы увидите, что текст теперь полупрозрачен (или если сформулировать иначе: непрозрачен на 25 %, 100 % – полностью непрозрачен).

DecoratedBox

Виджет DecoratedBox – это именно то, что написано: коробка, которая украшена! Если подробнее, он «украшает» другой контейнерный виджет, дочерний элемент DecoratedBox. Практически всегда вместе с DecoratedBox используется виджет BoxDecoration, который определяет желаемое «украшение».

Рассмотрим пример:

```
DecoratedBox(
   decoration : BoxDecoration(
      gradient : LinearGradient(
         begin : Alignment.topCenter,
         end : Alignment.bottomCenter,
         colors : [ Color(0xFF000000), Color(0xFFFF0000) ],
         tileMode : TileMode.repeated
      )
   ),
   child : Container(width : 100, height : 100,
      child : Text("Hello",
         style : TextStyle(color : Colors.white)
      )
   )
)
```

Здесь мы оборачиваем DecoratedBox вокруг Container, который является родительским для виджета Text. Сам по себе DecoratedBox ничего не отобразит; вот где начинает работать BoxDecoration. Нам нужно дать ему эту возможность, для чего и предназначен Container. Text внутри – это просто дополнительный бонус, показывающий, что декорируется именно Container, а не Text.

Когда мы решаем, как «украсить», в игру вступает свойство decoration, значением которого является виджет BoxDecoration. Он позволяет декорировать цветом или изображениями (например, чтобы фоновое изображение было применено к Container) либо играть с границами (например, с закругленными углами) и применять тени или градиенты, последний из которых здесь показан. LinearGradient – это один из нескольких классов градиентов (в дополнение к RadialGradient и SweepGradient), которые могут вам пригодиться. У него вы можете задать место начала и окончания градиента, используемые цвета и способ повторения градиента, если необходимо.

DecoratedBox в сочетании с BoxDecoration – это удобный и гибкий способ добавления стилей любому элементу контейнера.

Transform

Виджет Transform (изменять) применяет своего рода геометрическое преобразование к своему дочернему элементу. Практически любой вид преобразования может быть закодирован с его помощью. В качестве примера:

```
Center(
   child : Container(
      color : Colors.yellow,
      child : Transform(
         alignment : Alignment.bottomLeft,
         transform : Matrix4.skewY(0.4)..rotateZ(-3 / 12.0),
         child : Container(
            padding : const EdgeInsets.all(12.0),
            color : Colors.red,
            child : Text("Eat at Joe's!")
         )
      )
   )
)
```

Показанный Transform вращает и наклоняет красное поле с желтым фоном, на котором есть текст, сохраняя при этом нижний левый угол окна, прикрепленный к его первоначальному расположению. Может, это и не особенно полезный пример, но он демонстрирует возможности, которые предоставляет этот виджет, если вы знакомы с матричными преобразованиями.

В дополнение к этому конструктору есть также Transform.rotate(), Transform.scale() и Transform.translate(), каждый из которых возвращает виджет

Transform, специально настроенный на три наиболее распространенных типа преобразований, а именно вращение, масштабирование и перемещение. Готовые классы значительно проще в использовании, поскольку не требуют знания матричных операций (они принимают простое подмножество аргументов, которое, если хотите, будет менее математическим), поэтому если вам нужен один из этих распространенных типов преобразования, я очень рекомендую использовать их вместо конструктора Transform().

Анимации и переходы

Анимации в пользовательском интерфейсе – это большой бизнес в наши дни! Пользователи ждут, что их приложения будут визуально привлекательны. Именно в этих целях Flutter предоставляет виджеты анимаций. Учитывая, что демонстрация скриншотов на эту тему не имеет особого смысла, я их не привожу. Но вы могли бы запустить Android Studio и создать базовый проект. Это будет отличным упражнением, чтобы проверить ваше понимание и показать вам, как все происходит.

AnimatedContainer

Для относительно простых анимаций идеально подходит виджет AnimatedContainer. Он постепенно меняет нужное свойство самого себя в течение определенного периода времени. Это происходит автоматически – вы просто задаете ему начальные, а затем конечные значения, и он будет анимировать нужное свойство между этими значениями.

Вот простой пример:

```
class _MyApp extends State {

   var _color = Colors.yellow;
   var _height = 200.0;
   var _width = 200.0;

   @override
   Widget build(BuildContext context) {
      return MaterialApp(home : Scaffold(
         body : Center(child : Column(
            mainAxisAlignment : MainAxisAlignment.center,
            children : [
               AnimatedContainer(
                  duration : Duration(seconds : 1),
                  color : _color, width : _width, height : _height
               ),
               RaisedButton(
                  child : Text("Animate!"),
```

```
                onPressed : () {
                    _color = Colors.red;
                    _height = 400.0;
                    _width = 400.0;
                    setState(() {});
                }
            )
        ]
    ))
));
  }
}
```

Здесь у нас есть AnimatedContainer со свойством duration, установленным на одну секунду – столько времени займет анимация. Мы устанавливаем исходные свойства color, width и height для значений переменных, определенных в State. Затем, когда пользователь нажимает RaisedButton, значения всех трех переменных изменяются и вызывается setState(), что вызывает перерисовку, но теперь Flutter будет делать это в течение одной секунды, постепенно увеличивая и окрашивая в красный цвет AnimatedContainer.

Вы также найдете DecoratedBoxTransition, который можно использовать для анимации различных свойств DecoratedBox, поэтому он концептуально очень похож на AnimatedContainer, но для определенного целевого виджета.

AnimatedCrossFade

AnimatedCrossFade – это виджет, специально разработанный для плавного перехода между двумя элементами. Плавный переход – это когда один элемент исчезает, а другой появляется в том же месте. Он прост в использовании:

```
class _MyApp extends State {

   var _showFirst = true;

   @override
   Widget build(BuildContext context) {
      return MaterialApp(home : Scaffold(
         body : Center(child : Column(
            mainAxisAlignment : MainAxisAlignment.center,
            children : [
               AnimatedCrossFade(
                  duration : Duration(seconds : 2),
                  firstChild : FlutterLogo(
                     style : FlutterLogoStyle.horizontal,
                     size : 100,0
                  ),
```

```
                secondChild : FlutterLogo(
                    style : FlutterLogoStyle.stacked,
                    size : 100,0
                ),
                crossFadeState : _showFirst ?
                    CrossFadeState.showFirst :
                    CrossFadeState.showSecond,
            ),
            RaisedButton(
                child : TextC'Cross-Fade!"),
                onPressed : () {
                    _showFirst = false;
                    setState(() {});
                }
            )
        ]
    ))
));
  }
}
```

Во-первых, это первое знакомство с виджетом FlutterLogo. Как вы могли догадаться, он отображает виджет Flutter с различными стилями. Вам не нужно добавлять его в качестве ресурса или чего-то в этом роде, он используется автоматически.

Мы встроили парочку таких в виджеты AnimatedCrossFade, установив свойства firstChild и secondChild для экземпляров FlutterLogo.

Как и у AnimatedContainer, у него есть свойство duration, которое здесь равно двум секундам.

Свойство crossFadeState самое важное: оно сообщает, какой из двух виджетов отображать. Если установлено значение CrossFadeState.showFirst, оно показывает первый. Когда значение CrossFadeState.showSecond – второй.

Это основано на значении логической переменной _showFirst, которая начинается с true, поэтому первое изображение появляется, но затем устанавливается в false при нажатии RaisedButton, и вуаля, у нас есть плавный переход!

Обратите внимание, что существует также FadeTransition, который анимирует прозрачность элемента.

Вы можете, если хотите, создать свой собственный AnimatedCrossFade с двумя виджетами FadeTransition, работающими одновременно (я не проверял, но готов поспорить, что именно так реализован AnimatedCrossFade).

AnimatedDefaultTextStyle

AnimatedDefaultTextStyle – хороший выбор для анимации текста. Он работает очень схоже с AnimatedContainer и AnimatedCrossFade:

```
class _MyApp extends State {

   var _color = Colors.red;
   var _fontSize = 20.0;

   @override
   Widget build(BuildContext context) {
      return MaterialApp(home : Scaffold(
         body : Center(child : Column(
            mainAxisAlignment : MainAxisAlignment.center,
            children : [
               AnimatedDefaultTextStyle(
                  duration : const Duration(seconds : 1),
                  style : TextStyle(
                     color : _color, fontSize : _fontSize
                  ),
                  child : Text("I am some text")
               ),
               RaisedButton(
                  child : Text("Enhance! Enhance! Enhance!"),
                  onPressed : () {
                     _color = Colors.blue;
                     _fontSize = 40.0;
                     setState(() {});
                  }
               )
            ]
         ))
      ));
   }
}
```

Здесь Text, дочерний по отношению к AnimatedDefaultTextStyle, увеличивается на 100 %, а его цвет меняется в течение одной секунды. Думаю, здесь не нужно много объяснять, учитывая, насколько похожи последние три виджета.

Несколько других: AnimatedOpacity, AnimatedPosition, PositionTransition, SlideTransition, AnimatedSize, ScaleTransition, SizeTransition и RotationTransition

Я собираюсь сэкономить немного места в книге и просто упомянуть, не показывая примеров, что помимо виджетов, которые вы видели в этом разделе, есть и другие, которые вы видите в заголовке. Они могут использоваться точно так же, как и предыдущие, для анимации прозрачности элемента, положения элемента, его размеров или поворота.

Обратите внимание, что виджет AnimatedOpacity следует использовать с умом, потому что анимация полупрозрачности является относительно дорогостоящей операцией (это также относится к виджетам AnimatedCrossFade и FadeTransition).

Также обратите внимание, что виджет AnimatedPosition работает, только если его дочерний элемент является элементом виджета Stack, который мы еще не обсуждали. Вкратце, он позволяет отображать несколько дочерних элементов, накладывающихся друг на друга (независимо от того, одинакового они размера или нет, это означает, что если больший элемент находится ниже меньшего, то более крупный элемент может «выглянуть» из-за меньшего элемента). Вы определенно встретите его снова в последующих главах, но запомните, что стек навигатора – это совсем другое понятие. Виджет Stack – это просто контейнер для других элементов, которые могут быть расположены друг над другом.

Интересное примечание о виджетах *Transition: все они поддерживают физику, что позволяет создавать не просто линейные анимации. Это верно и для виджетов Animated*, но реализация в виджетах Transition имеет немного более интересную визуализацию, а это означает, что вы можете получить более захватывающие анимации.

Drag и Drop

Хотя взаимодействие с перетаскиванием мало распространено на мобильных устройствах, оно часто встречается на стационарных компьютерах, и Flutter его поддерживает. Он делает это с помощью двух основных виджетов: Draggable и DragTarget. Они довольно просты в использовании:

```
class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext context) {
      return MaterialApp(home : Scaffold(
         body : Center(child : Column(mainAxisAlignment :
         MainAxisAlignment.center,
         children : [
            DragTarget(
               builder : (BuildContext context,
                  List<String> accepted,
                  List<dynamic> rejected) {
                     return new Container(width : 200, height : 200,
                        color : Colors.lightBlue);
                  },
                  onAccept : (data) => print(data)
               ),
               Container(height : 50),
               Draggable(
                  data : "I was dragged",
```

```
                    child : Container(width : 100, height : 100,
                       color : Colors.red),
                    feedback : Container(width : 100, height : 100,
                       color : Colors.yellow)
                )
            ]
        ))
    ));
  }
}
```

Во-первых, у нас есть DragTarget, куда можно перетаскивать объект. Нам нужно указать тип данных, который будет принимать эта цель, и в данном случае это старая добрая String. Функция builder() возвращает Container, но она может возвращать все, что мы хотим.

Затем второй Container добавляется в макет Column, чтобы дать нам немного места между DragTarget и Draggable. В первую очередь нам нужно предоставить свойство data, представляющее собой произвольные данные, которые вы хотите предоставить DragTarget; а затем свойство feedback – виджет, который пользователь будет физически (или виртуально? Физически/виртуально? Какое слово больше подходит?!) перетаскивать.

Видите ли, перетаскивание работает, потому что оригинальный виджет, который указывается с помощью child, никогда не перемещается. Вместо этого, когда пользователь начинает перемещать дочерний Container, виджет, определяемый feedback, визуализируется и начинает перетаскиваться. Когда он помещается в DragTarget, там запускается функция-обработчик onAccept, получающая значение свойств данных Draggable.

Существует множество других функций callback, которые могут запускаться в различных ситуациях на обоих виджетах, но наиболее полезный обработчик – функция onDragComplete, которая запускается при опускании Draggable на DragTarget. Обычно это место, где вы скрываете исходный дочерний виджет или делаете что-то еще.

Наконец, есть LongPressDraggable – виджет, который можно использовать вместо Draggable. Разница в том, что процесс перетаскивания запускается только после долгого нажатия. Это небольшая разница взаимодействия, которая зависит только от вас.

Просмотр данных

Использование специальных виджетов для отображения наборов данных является типичным шаблоном любого приложения. Flutter предоставляет для этого несколько специальных виджетов (вы всегда можете создать свой собственный с различными компонентами прокрутки, но обычно это не требуется).

Table

Виджет Table (таблица) – пожалуй, самый простой из виджетов «data views», используемых для отображения коллекций данных. Если вы знакомы с таблицей HTML, то у вас уже есть общее представление о виджете Table: отображение элементов в организации строк и столбцов. Посмотрите на пример кода в листинге 4-2 и результат на рис. 4-2.

Листинг 4-2. *Создание таблицы с помощью виджета Table*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         body : Column(children : [
            Container(height : 100),
            Table(
               border : TableBorder(
                  top : BorderSide(width : 2),
                  bottom : BorderSide(width : 2),
                  left : BorderSide(width : 2),
                  right : BorderSide(width : 2)
               ),
               children : [
                  TableRow(
                     children : [
                        Center(child : Padding(
                           padding : EdgeInsets.all(10),
                           child : Text("1"))
                        ),
                        Center(child : Padding(
                           padding : EdgeInsets.all(10),
                           child : Text("2"))
                        ),
                        Center(child : Padding(
                           padding : EdgeInsets.all(10),
                           child : Text("3"))
                        )
                     ]
                  )
               ]
            )
         ])
      ));
```

```
    }
}
```

Рисунок 4-2. *Базовый пример Table!*

Все просто, да? Вы можете задать границу `Table`, но по умолчанию ее не будет вообще. Затем вам нужно просто добавить несколько строк через дочерние элементы, которые могут быть любым виджетом или деревом виджетов, но в верхней части должен быть экземпляр `TableRow`, а дочерние элементы каждого – это ячейка или столбец в строке. Каждая строка в `Table` должна иметь одинаковое количество дочерних элементов. Вы можете вручную задать ширину столбцов с помощью свойства `columnWidth`, а вертикальное выравнивание содержимого в каждой ячейке можно настроить с помощью свойства `defaultVerticalAlignment`.

DataTable

Отображение данных в табличной форме очень распространено в Ul, поэтому Flutter предоставляет для этой цели виджет `DataTable`.

Листинг 4-3. *Виджет DataTable*

```
import "package:flutter/material.dart";
```

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         body : Column(children : [
            Container(height : 100),
            DataTable(sortColumnIndex : 1,
               columns : [
                  DataColumn(label : Text("First Name")),
                  DataColumn(label : Text("Last Name"))
               ],
               rows : [
                  DataRow(cells : [
                     DataCell(Text("Leia")),
                     DataCell(Text("Organa"), showEditIcon : true)
                  ]),
                  DataRow(cells : [
                     DataCell(Text("Luke")),
                     DataCell(Text("Skywalker"))
                  ]),
                  DataRow(cells : [
                     DataCell(Text("Han")),
                     DataCell(Text("Solo"))
                  ])
               ]
            )
         ])
      ));
   }
}
```

Проще говоря, `DataTable` требует, чтобы вы указали ему, что представляют собой столбцы в таблице и, конечно же, каковы строки данных для отображения. Каждый столбец определяется с помощью экземпляра `DataColumn`, а каждая строка – экземпляром `DataRow`, который содержит коллекцию ячеек, членами которых являются экземпляры `DataCell`. Хотя это и не обязательно, вы можете предоставить свойство `sortColumnIndex`, чтобы указать, по какому столбцу сортируются данные в настоящее время. Обратите внимание, что это всего лишь визуальный индикатор – ваш код отвечает за логику сортировки данных (большую часть времени вы не будете предоставлять статичные данные, как в этом примере; вместо этого у вас будет какая-то функция, которая создает список). Вы можете увидеть большую часть этого на рис. 4-3.

Рисунок 4-3. *DataTable*

`DataColumn` может отображать всплывающую подсказку при длительном нажатии на столбец. `DataCell` может включать свойство `showEditIcon`, которое, если оно `true`, показывает небольшой значок карандаша, чтобы указать, что ячейка может быть отредактирована. Однако фактическое редактирование должно быть обеспечено вашим кодом.

Обратите внимание, что `DataTable` – довольно дорогой виджет в вычислительном отношении из-за процесса компоновки, который он должен реализовать. Поэтому если у вас много данных для отображения, рекомендуется вместо него использовать виджет `PaginatedDataTable`. Он работает так же, как `DataTable`, но разбивает данные на страницы, между которыми пользователь может перемещаться. Таким образом, нужно отобразить только одну часть за раз, это дешевле.

GridView

Виджет `GridView` отображает двумерную сетку виджетов. Он может прокручиваться в любом направлении в соответствии со свойством `scrollDirection` (по умолчанию `Axis.vertical`) и предоставляет несколько макетов, наиболее распространенный генерируется конструктором `GridView.count()`, как показано в листинге 4-4.

Листинг 4-4. *GridView полон логотипов Flutter*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         body : GridView.count(
            padding : EdgeInsets.all(4.0),
            crossAxisCount : 4, childAspectRatio : 1.0,
            mainAxisSpacing : 4.0, crossAxisSpacing : 4.0,
            children: [
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo()),
               GridTile(child : new FlutterLogo())
            ]
         )
      ));
   }
}
```

Это создает макет, показанный на рис. 4-4, с фиксированным количеством элементов (называемых плитками) на поперечной оси. Другие варианты включают `GridView.extentQ`, который создает макет с плитками, имеющими максимальный размер поперечной оси. Вы также можете использовать конструктор `GridView.builderQ`, если у вас есть «бесконечное» количество плиток для отображения.

Рисунок 4-4. *GridView*

Обратите внимание, что `GridView` очень похож на `ListView`, который в некотором смысле представляет собой линейный `GridView` (мы вот-вот рассмотрим `ListView`).

ListView и ListTile

Виджет `ListView` – вероятно, наиболее важный из виджетов отображения данных. Вы будете часто использовать его со списком прокручиваемых элементов для отображения. Простейшая форма его кодирования выглядит, как показано в листинге 4-5.

Листинг 4-5. *Код простого статического ListView*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext inContext) {
     return MaterialApp(home : Scaffold(
        body : ListView(children : [
           ListTile(leading: Icon(Icons.gif), title: Text("1")),
```

```
            ListTile(leading: Icon(Icons.book), title: Text("2")),
            ListTile(leading: Icon(Icons.call), title: Text("3")),
            ListTile(leading: Icon(Icons.dns), title: Text("4")),
            ListTile(leading: Icon(Icons.cake), title: Text("5")),
            ListTile(leading: Icon(Icons.pets), title: Text("6")),
            ListTile(leading: Icon(Icons.poll), title: Text("7")),
            ListTile(leading: Icon(Icons.face), title: Text("8")),
            ListTile(leading: Icon(Icons.home), title: Text("9")),
            ListTile(leading: Icon(Icons.adb), title: Text("10")),
            ListTile(leading: Icon(Icons.dvr), title: Text("11")),
            ListTile(leading: Icon(Icons.hd), title: Text("12")),
            ListTile(leading: Icon(Icons.toc), title: Text("3")),
            ListTile(leading: Icon(Icons.tv), title: Text("14")),
            ListTile(leading: Icon(Icons.help), title: Text("15"))
          ])
      ));
  }
}
```

Дочерним объектом `ListView` может быть что угодно, но обычно вы будете использовать виджеты `ListTile` (точнее, несколько из них). `ListTile` – это виджет, представляющий собой строку с фиксированной высотой, которая содержит текст и начальный или конечный значок. `ListTile` может отображать до трех строк текста, включая подзаголовок (subtitle). В этом примере свойство `leading` («ведущий элемент») используется для отображения `Icon` (иконка) перед текстом, который вы можете видеть на рис. 4-5.

Рисунок 4-5. *Виджет ListView в сочетании с ListTile*

`ListView` может прокручиваться вертикально или горизонтально в зависимости от установки свойства `scrollDirection`. Вы даже можете настроить способ управления прокруткой в `ListView`, настроив свойство `physics`, экземпляр `Scroll Physics`.

`ListView` предоставляет несколько различных конструкторов, один из которых показан в примере. Существует также конструктор `ListView.builder()`, который использует функцию `builder` для визуализации строк. `ListView.separated()` также доступен, и это предоставит вам `ListView`, в котором между каждым элементом коллекции будет отображаться заданный вами разделитель. Конструктор `ListView.custom()` дает вам большую гибкость в настройке, чтобы `ListView` выглядел и работал так, как вы захотите.

Существует также виджет `PageView`, который поддерживает разбиение на страницы. Это хороший выбор, если у вас есть много элементов, которые вы хотите отобразить, но не просто отобразить, а разбить на группы, где каждая группа становится отдельной страницей.

Остальные виджеты

Некоторые виджеты не поддаются классификации. Но ненадолго, потому что теперь есть категория «разное» (miscellaneous) специально для них!

CircularProgressIndicator (CupertinoActivityIndicator) и LinearProgressIndicator

Что вы показываете пользователям своих приложений во время длительного ожидания? Есть много вариантов, но CircularProgressIndicator – один из лучших. Это примитивный анимированный круг, но он выполняет свою работу и очень прост в использовании:

```
CircularProgressIndicator()
```

Да, в общем-то, это все, что вам нужно! Остальное Flutter нарисует за вас. Есть несколько опций, которые могут вас заинтересовать. Во-первых, strokeWidth – позволяет задать толщину круга. Свойство backgroundColor дает возможность установить цвет фона за индикатором. Наконец, valueColor позволяет определить цвет самого круга. К сожалению, это не так просто, как установить цвет из класса Colors. Нет, вы должны предоставить экземпляр класса Animation или одного из его потомков. Почти всегда это будет класс AlwaysStoppedAnimation, у которого есть конструктор, принимающий color в качестве аргумента, так что это не намного сложнее.

Для iOS можно использовать CupertinoActivityIndicator, который выглядит и работает примерно так же. Чтобы работать с ним, вам нужно импортировать package:flutter/cupertino.dart, как и для всех виджетов Cupertino. Кроме того, CupertinoActivityIndicator не обладает той же гибкостью, что и CircularProgressIndicator: у него есть только свойство radius, чтобы определить размер круга, – никаких настроек цветов нет.

Наконец, есть LinearProgressIndicator, который показывает прогресс в виде цветной линии:

```
LinearProgressIndicator(
   value : .25,
   backgroundColor : Colors.yellow
)
```

Здесь value представляет собой число от нуля до единицы, которое определяет текущий прогресс и окрашивает эту часть цветовой шкалы. BackgroundColor – это цвет части индикатора, соответствующей оставшемуся прогрессу, в то время как значение valueColor (которое, подобно CircularProgressIndicator, принимает в качестве значения экземпляр Animation) – это завершенная часть. Таким образом, в нашем примере 75 % панели будут окрашены в желтый цвет, а 25 % – в цвет по умолчанию, поскольку значение valueColor не указано.

Icon

Виджет Icon (иконка) позволяет отображать значки стиля Material. Чтобы использовать его, достаточно вызвать конструктор:

```
Icon(Icons.radio)
```

Класс Icons содержит список доступных значков в стилистике Material, и их довольно много. Тем не менее вы также можете добавлять и свои собственные иконки, так как они реализуются с помощью шрифтов – вместо буквы на экране рисуется заданная иконка. Вы можете добавить и свои шрифты, если необходимы другие значки (например, существует популярная среди веб-разработчиков коллекция значков Font Awesome).

Для этого нам нужно перейти в файл pubspec.yaml, который упоминался в главе 1. Вкратце, этот файл содержит конфигурацию, которую Flutter использует для создания и запуска вашего приложения. В нем перечислены зависимости вашего проекта, его название, какая версия Flutter требуется и многое другое. В зависимости от ваших потребностей вам, возможно, никогда не придется прикасаться к нему после создания проекта. По умолчанию файл pubspec.yaml будет выглядеть примерно так:

```
name: flutter_playground
description: flutter playground

version: 1.0.0+1
environment:
   sdk: ">=2.0.0-dev.68.0 <3.0.0"

dependencies:
   flutter:
      sdk: flutter
      cupertino_icons: ^0.1.2

dev_dependencies:
   flutter_test:
      sdk: flutter

flutter:
   uses-material-design: true
```

Это файл pubspec.yaml, созданный для приложения Flutter Playground, сгенерированного мной при написании этой главы (который представляет собой базовый проект приложения Flutter). Обратите внимание, что я удалил поясняющие комментарии, дающие вам подсказки о других возможностях, которые вы можете реализовать, включая добавление новых шрифтов для значков! Например, они объясняют, как вы можете добавить файл True Type Font (TTF) в ваш проект. Чтобы добавить Font Awesome, вы можете сделать так:

```
flutter:
   fonts:
      - family: FontAwesome
   fonts:
      - asset: fonts/font-awesome-400.ttf
```

Как только вы это сделаете, то сможете создать экземпляр IconData, который задает код значка (его вы можете найти на веб-сайте fontawesome.com) и семейство шрифтов, к которому он принадлежит, например:

```
Icon(IconData(0xf556, fontFamily : "FontAwesome"))
```

Это несложно. Но есть еще более простой способ, по крайней мере для некоторых шрифтов, например Font Awesome, который считается самой популярной коллекцией иконок на основе шрифтов. Вы можете добавить готовый плагин, который уже будет содержать нужный шрифт и коды значков. Плагин – это то, что расширяет стандартные возможности Dart/Flutter. Это такой код Dart, который вы можете импортировать в свой проект по мере необходимости. Чтобы его использовать, вам просто нужно добавить одну строку в pubspec.yaml в разделе dependencies (зависимости):

```
dependencies:
   font_awesome_flutter: ^8.4.0
```

Это означает, что нам нужна версия плагина font_awesome_flutter 8.4.0 или более новая. Информацию об этом плагине можно найти по адресу https://pub.dartlang.org/packages, еще вы можете найти там множество других полезных плагинов, которые добавляются так же. Это будет не последний плагин, который мы увидим в данной книге.

Затем вам нужно сообщить Android Studio, что нужно скачать зависимости. Для этого прямо над файлом есть специальные подсказки. Нажмите **Packages Get**, и зависимости будут загружены. Эта зависимость включает в себя необходимый файл TTF и дополнительный код для нашего использования.

Теперь можно импортировать пакет в любой файл, где он вам понадобится:

```
import "package:font_awesome_flutter/font_awesome_flutter.dart";
```

Преимущество этих «манипуляций» заключается в том, что вместо ручного указания кода для иконки вы можете просто написать:

```
Icon(FontAwesomeIcons.angry)
```

Это делается так же легко, как и с помощью встроенных значков, но теперь у вас есть гораздо больше иконок благодаря Font Awesome!

Мы будем рассматривать pubspec.yaml по мере необходимости, но это была хорошая демонстрация его возможностей, доступных Flutter.

Image

Наряду с Icon стоит также выделить виджет Image (изображение), который, как вы можете догадаться, используется для отображения какой-либо картинки. Он предлагает несколько различных конструкторов, каждый для показа изображений из различных источников. Я же собираюсь рассказать только о двух наибо-

лее распространенных: `Image.asset()` для загрузки картинки из ресурсов самого приложения и `Image.network()` для загрузки его из сети.

Во-первых, `Image.asset()` позволяет загрузить изображение, которое включено в ресурсы приложения:

```
Image.asset("img/ron.jpg")
```

Кажется простым, да? Но одна часть отсутствует: мы должны рассказать Flutter о нашем изображении, которое называется `asset`. Для этого мы возвращаемся в `pubspec.yaml` и добавляем новый раздел под заголовком Flutter:

```
assets:
- img/ron.jpg
```

Каждый ресурс, который вы хотите включить в список «активов» (assets), должен быть объявлен в этом разделе. В противном случае Flutter SDK не будет знать о том, что его необходимо добавить в установочный пакет. Вы также можете использовать сокращенную запись `img/`, чтобы включить все файлы из каталога `img`. Но обратите внимание, что будут добавлены только файлы, находящиеся непосредственно в `img/`, – ничего из подкаталогов `img/` включено *не будет* (или вам нужно будет добавить запись для каждого подкаталога).

Обратите внимание, что ресурсами (как часть `assets`) могут быть не только изображения, но также и любые другие файлы, например текстовые в формате JSON. Загружайте их, используя объект `rootBundle`, который доступен в коде вашего приложения. Например, чтобы загрузить файл `settings.json`:

```
String settings =
   await rootBundleloadString("textAssets/settings.json");
```

Немного дополнительной информации: когда сборка завершена, Flutter SDK создает специальный архив, который поставляется вместе с вашим приложением, он называется `asset bundle` («комплект активов»). Вы можете использовать находящиеся в нем ресурсы во время выполнения, как показано в примере `settings.json` (и очевидно, что `Image.asset()` делает это под капотом).

Примечание. Во Flutter есть и дополнительные возможности для внедрения ресурсов, например изображения разных размеров для разных разрешений экрана. Однако для целей книги нам достаточно базовой информации, поэтому если вы считаете, что вам нужно больше, вам придется изучить `flutter.io` самостоятельно.

Загрузка изображения из сети еще проще, поскольку нет даже `assets`, которые нужно было бы объявлять:

```
Image.network(
   "http://zammetti.com/booksarticles/img/darkness.png"
)
```

Да, это все! Если устройство подключено к интернету, изображение будет загружено и показано точно так же, как если бы оно было размещено на смартфоне вместе с приложением (хотя и немного медленнее, учитывая внутреннюю задержку сети).

Chip

Chip – это небольшие визуальные элементы, которые обычно предназначены для отображения мелких деталей и элементов, а также для представления различных сущностей, таких как пользователи, или быстрых действий, которые пользователь может выполнить.

Типичное использование Chip – отображение информации о пользователе. Для лучшего понимания обратите внимание на листинг 4-6 и рис. 4-6.

Листинг 4-6. *Просто Chip*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         body : Center(child :
            Chip(
               avatar : CircleAvatar(
                  backgroundImage : AssetImage("img/ron.jpg")
               ),
               backgroundColor : Colors.grey.shade300,
               label : Text("Frank Zammetti")
            )
         )
      ));
   }
}
```

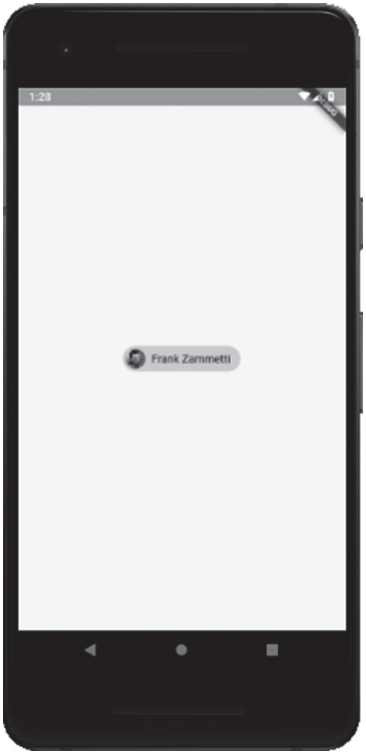


Рисунок 4-6. *Chip, просто Chip*

Свойство `avatar` не является обязательным и обычно показывает либо изображение, либо инициалы пользователя. Оно принимает значение, которое само по себе является виджетом, поэтому теоретически вы можете поместить здесь все, что захотите. Я обычно использую виджет `CircleAvatar`, который может показывать изображение или текст (обычно инициалы человека, когда `Chip` представляет человека) либо содержать дочерние виджеты. Здесь я использовал то же изображение, что и в предыдущем примере, чтобы показать маленькую картинку меня. Свойство `backgroundColor`, разумеется, отвечает за цвет `Chip`, а свойство `label` – текст, отображаемый рядом с аватаркой.

Если вы добавите свойство `onDeleted`, то `Chip` будет содержать кнопку удаления. Вам нужно будет самостоятельно реализовать процесс удаления, поскольку `Chip` предоставляет чисто визуальное оформление.

FloatingActionButton

Виджет `FloatingActionButton` очень распространен на устройствах Android, в меньшей степени на устройствах iOS. Это круглая кнопка, которая располагается поверх основного контента и предоставляет пользователю быстрый до-

ступ к основным функциям. Например, это может быть кнопка, которая вызывает экран добавления события в приложении календаря.

FloatingActionButton редко создают вручную, хотя это возможно. Также «плохим тоном» считается наличие более одной такой кнопки на экране, но опять же, технически вы можете это сделать. Чаще всего вы будете указывать подобную кнопку в качестве значения свойства floatingActionButton виджета Scaffold, как показано в листинге 4-7.

Листинг 4-7. *FloatingActionButton как часть Scaffold*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         floatingActionButton : FloatingActionButton(
            backgroundColor : Colors.red,
            foregroundColor : Colors.yellow,
            child : Icon(Icons.add),
            onPressed : () { print("Ouch! Stop it!"); }
         ),
         body : Center(child : Text("Click it!"))
      ));
   }
}
```

Как правило, дочерним элементом FloatingActionButton будет Icon, как на рис. 4-7.

Рисунок 4-7. *FloatingActionButton делает свою, эм, плавающую штуку*

Свойство `backgroundColor` окрашивает саму кнопку в любой цвет, который вам нравится, а цвет `foregroundColor` окрашивает сам значок или текст на кнопке. Свойство `onPressed` необязательно, но если оно не указано, кнопка отключается и не реагирует на прикосновения. Это не очень хорошо, так что вам нужно определить функцию для реализации любой функциональности кнопки.

Вы также можете настроить тень с помощью свойства `elevation` и даже сделать кнопку квадратной, установив в качестве значения свойства `shape` (форма) потомка класса `RoundedRectangleBorder`.

PopupMenuButton

Виджет `PopupMenuButton` реализует общую парадигму «трехточечного» меню для отображения всплывающего диалога с набором опций. Этот виджет можно разместить где угодно, и он будет отображаться в виде трех вертикальных точек. У него есть свойство `onSelected`, которое вызывает предоставляемую вами функцию. Затем вы можете реализовать любое необходимое поведение. Вот пример:

Листинг 4-8. *PopupMenuButton и его меню*

```
import "package:flutter/material.dart";

void main() => runApp(MyApp());
```

```
class MyApp extends StatelessWidget {

   @override
   Widget build(BuildContext inContext) {
      return MaterialApp(home : Scaffold(
         body : Center(child :
            PopupMenuButton(
               onSelected : (String result) { print(result); },
               itemBuilder : (BuildContext context) =>
                  <PopupMenuEntry<String>>[
                     PopupMenuItem(
                        value : "copy", child : Text("Copy")
                     ),
                     PopupMenuItem(
                        value : "cut", child : Text("Cut")
                     ),
                     PopupMenuItem(
                        value : "paste", child : Text("Paste")
                     )
                  ]
            )
         )
      ));
   }
}
```

Вы наверняка знаете, как это выглядит, и без скриншота! Виджет PopupMenu Button использует ранее описанный шаблон builder для создания списка виджетов PopupMenuItem. Эти виджеты могут иметь любой дочерний элемент, который вы считаете подходящим, но чаще всего это просто Text.

Для начала вы устанавливаете value у каждого элемента, а затем ваша функция onSelected передает это значение и реализует необходимое поведение (в нашем случае будет происходить вывод в консоль).

Другие поддерживаемые свойства включают возможность выбора элемента по умолчанию (свойство initialValue), возможность реагировать на отмену без выбора элемента (предоставляет реализацию onCanceled), а также вы можете настраивать свойства elevation (тень) и padding (отступ).

Базовые библиотеки

В дополнение к большому разнообразию виджетов Flutter также предлагает доступ к набору вспомогательных API, упакованных в библиотеки. Условно их можно разделить на три категории: базовые библиотеки Flutter, библиотеки Dart и другие вспомогательные библиотеки. Мы рассмотрим каждую группу, но последние две объединим.

Обратите внимание, что, как и в случае с виджетами, это будет краткий, но очень полезный обзор. Существует больше API-интерфейсов, чем показано в текущем разделе. Также для многих из рассмотренных API не будет примеров кода или подробных сведений, просто базовое «вот что-то, что может вас заинтересовать», как и раньше, я постараюсь указать на детали, которые *будут* интересны большинству разработчиков. Я думаю, вы должны потратить свое время, чтобы увидеть все, что вам доступно на сайте `flutter.io`. Часто вам нужно будет обращаться именно к онлайн-документации, чтобы получить исчерпывающую информацию по конкретным возможностям фреймворка. А еще там есть примеры кода, много примеров. После же прочтения текущего раздела вы узнаете, что вам будет доступно и как потом найти подробную информацию. Это и есть первоначальная цель раздела!

Основные библиотеки фреймворка Flutter

Базовые библиотеки Flutter предоставляют основную функциональность фреймворка. Многие из них используются внутри виджетов, так что вы можете не обращаться к библиотекам напрямую.

Обратите внимание, что для использования библиотек вы сначала должны их импортировать, синтаксис импорта: `package:flutter/<library-name>.dart`.

animation

Библиотека `animation` предоставляет множество функций для реализации различных анимаций в приложениях Flutter. Вот несколько интересных примеров:

- `Animation` – этот класс содержит основную информацию об анимациях;
- `AnimationController` – этот класс позволяет управлять анимациями – их запуском, остановкой, сбросом и повтором;
- `Curve` – этот класс определяет, с помощью какой временной функции будет происходить анимация. Например, замедляться в начале и ускоряться в конце. Существует множество подклассов `Curve`, включая `Cubic`, `ElasticIn` `OutCurve`, `Interval` и `Sawtooth`, которые определяют ускорение и замедление анимаций;
- `Tween` – как и `Curve`, этот класс содержит данные, определяющие «двойные» (`tween`) анимации – переход из одного заданного состояния в новое. Как и `Curve`, `Tween` включает много подклассов для типовых анимаций, таких как `ColorTween` (анимация изменения цвета между парой значений), `TextStyleTween` (анимация между двумя стилями текста, например переход от обычного текста к полужирному) и `RectTween` (анимация изменения размера прямоугольного виджета).

foundation

Эта библиотека содержит базовые классы, функции и многое другое, являющиеся своего рода фундаментом (`foundation`) для других компонентов. Все остальные слои Flutter будут использовать эту библиотеку. Примеры классов, входящих в `foundation`:

- `Key` – мы встречали его раньше в подклассах `GlobalKey` и `LocalKey`;
- `kReleaseMode` – константа, которая, если равна true, компилирует приложение в режиме release (выпуск);
- `required` – константа, используемая для пометки параметра (в методе или функции) как обязательного. Да, вы можете пользоваться этим в ваших приложениях!
- `debugPrintStack()` – функция, которая выводит текущий стек вызовов в консоль;
- `debugWordWrap()` – функция, которая позволяет переносить длинные тексты с учетом заданных ограничений;
- `TargetPlatform` – перечисление (enumeration), которое содержит значения, соответствующие различным поддерживаемым платформам (на момент написания книги это были android, fuscia и iOS).

gestures

Библиотека `gestures` (жесты) содержит код для распознавания различных пользовательских жестов, распространенных на смартфонах и планшетах, таких как двойное касание, свайпы и операции перетаскивания. Здесь вы найдете такие вещи, как:

- `DoubleTapGestureRecognizer` – класс, который обрабатывает двойное касание;
- `PanGestureRecognizer` – класс, который распознает движения перетаскивания в горизонтальном и вертикальном направлениях;
- `ScaleGestureRecognizer` – класс для распознавания жестов, обычно используемых для увеличения и уменьшения масштаба.

painting

Библиотека `painting` (рисование) включает в себя множество классов, которые реализуют механизмы отрисовки, являющиеся частью движка Flutter. Эта библиотека предоставляет такие классы и методы для рисования, как, например, изменение масштаба изображений, добавление границы или теней вокруг контейнеров. С некоторыми механизмами вы уже знакомы из предыдущих разделов и примеров:

- `Alignment` – класс, определяющий точку внутри прямоугольника;
- `AssetImage` – класс, который извлекает изображение из `AssetBundle`;
- `Border` – класс, определяющий границу контейнера;
- `Gradient` – класс для отображения градиента цвета в 2D;
- `TextDecoration` – класс, используемый для оформления текста;
- `debugDisableShadows` – свойство, которое можно установить в значение `true`, чтобы превратить все тени в сплошные цветные блоки – это очень удобно для отладки;
- `BorderStyle` – перечисление со значениями, определяющими стиль рисования линий для границ (`none`, `solid` или список значений);
- `TextAlign` – перечисление со значениями, определяющими выравнивание текста (`center`, `end`, `justify`, `left` или `start`).

services

Эта библиотека содержит сервисные (`service`) классы и методы для работы с возможностями операционной системы. Вот некоторые из них:

- `AssetBundle` – класс, который состоит из набора ресурсов, используемых приложением (мы кратко говорили об этом ранее). Такие файлы, как изображения или тексты, могут быть помещены в `AssetBundle`;
- `ByteData` – класс для работы с массивом фиксированной длины, содержащим информацию в виде набора байтов. Он также обеспечивает произвольный доступ к числам (целые, с плавающей запятой), представленным этими байтами;
- `Clipboard` – класс, содержащий методы для работы с системным буфером обмена (методы `getData()` и `setData()`). Они используют класс `ClipboardData` для хранения данных, помещаемых или извлекаемых из буфера обмена;
- `HapticFeedback` – класс, который обеспечивает доступ к движку тактильной обратной связи (например, вибрация при нажатии экранной кнопки). Здесь можно найти такие методы, как `heavyImpact()`, `mediumImpact()` и `lightImpact()` для сильной, средней и легкой вибрации соответственно;
- `SystemSound` – класс, который предоставляет метод `play()` для воспроизведения одного из коротких звуков системной библиотеки, указанных в `SystemSoundType`;
- `DeviceOrientation` – перечисление со значениями `landscapeLeft` и `portrait Down`, которые можно использовать для определения и изменения ориентации устройства.

widgets

Существует также отдельная библиотека виджетов (`widgets`), и если вы думаете, что она содержит все виджеты Flutter, то вы в значительной степени правы! Нет особого смысла сейчас останавливаться на этой библиотеке, так как вы уже видели большую часть ее содержимого и продолжите с ним знакомиться при написании кода в последующих главах. Но если вас когда-нибудь заинтересует, где живут виджеты, то ответ находится в этой библиотеке, и вы можете перейти к описанию виджетов через документацию данной библиотеки (хотя, поскольку есть отдельная документация для виджетов в открытом виде, нет особого смысла следовать по этому пути – но вы можете).

Библиотеки Dart

Библиотеки `Dart` предоставляются самим Dart. Чтобы импортировать их, используйте форму импорта `dart:<library-name>.dart`.

core

Технически есть библиотека под названием `core` (ядро), которая содержит встроенные типы, коллекции и другие базовые механизмы, которые нужны каждой программе Dart. Таким образом, в отличие от других библиотек Dart, вам не нужно специально импортировать ее. Это происходит автоматически при написании программы Dart! Так что я собираюсь кое-что пропустить, потому что многое вы либо уже видели, либо еще увидите, так что двигаемся дальше.

ui

Учитывая, что Google работает и с Flutter, и с Dart, иногда можно встретить, так сказать, перекрестное опыление, и библиотека `ui` (user interface, пользовательский интерфейс) – один из таких примеров. Она содержит встроенные типы и основные примитивы для приложений Flutter. Однако библиотека `ui` предоставляет сервисы более низкого уровня, используемые Flutter для начальной загрузки приложений: классы для управления подсистемами ввода, графики, текста, компоновки и отображения на экране. Вы вряд ли будете использовать её очень часто, прямо в коде приложения. Я думаю, что лучше увидеть ее возможности в правильном контексте, поэтому не буду вдаваться в подробности сейчас.

async

Эта библиотека обеспечивает поддержку асинхронного (`asynchronous` или `async`) программирования. Хотя классов в ней немного больше, я думаю, будет справедливо сказать, что эти два – настоящие звезды:

- `Future` – класс для представления результатов асинхронных операций, которые будут завершены позже. Вы обнаружите, что многие методы во

Flutter и Dart возвращают Future. У Future есть метод then(), это функция, которая будет выполняться, когда Future завершится и вернет свое значение. В будущем вы увидите много упоминаний этого класса;

- Stream – класс, обеспечивающий асинхронный доступ к потоку данных. Он содержит метод listen(), который будет выполняться каждый раз, когда в Stream будет происходить обновление данных.

Будет справедливо сказать, что информации об этой библиотеке достаточно, поскольку, за редким исключением, это почти все, что вам когда-либо понадобится!

collection

Библиотека core уже содержит механизмы, связанные с коллекциями (collection) данных, но отдельная библиотека collection дополняет их такими вещами, как:

- DoubleLinkedQueue – класс Queue (еще один класс в этой библиотеке!), основанный на реализации двойного связанного списка;
- HashSet – неупорядоченный класс реализации Set на основе хеш-таблицы;
- SplayTreeMap – класс Map, в котором хранятся объекты, которые можно упорядочивать относительно друг друга;
- UnmodifiableListView – нужен для отображения неизменяемого списка (list) данных.

convert

В этой библиотеке вы найдете утилиты для преобразования (convert) данных между различными представлениями, включая распространенные форматы JSON и UTF-8. Вот примеры классов, которые вы будете чаще всего использовать:

- JsonCodec – класс, который кодирует и декодирует строки и объекты JSON. Методы json.encode() и json.decode() – ваши основные точки входа (обратите внимание, что json – это экземпляр JsonCodec, который автоматически доступен после импорта библиотеки convert);
- Utf8Codec – класс, в котором вы найдете автоматически созданный экземпляр с именем uft8. Он также содержит методы encode() и decode(), которые можно использовать для преобразования между строками Unicode и их соответствующими байтовыми значениями;
- AsciiCodec – класс, который с помощью своего автоматического экземпляра ascii позволяет кодировать строки в виде байтов ASCII с помощью метода encode() и декодировать байты ASCII в строки с помощью decode();

- `Base64Codec` – класс, используемый для кодирования и декодирования объектов в base64, опять же с помощью методов `encode()` и `decode()`, и доступного экземпляра `base64` (вы уже видите закономерность?!).

Обратите внимание, что в дополнение к экземплярам `json` и `base64`, поскольку кодирование/декодирование JSON и base64 очень распространено, вы также найдете функции верхнего уровня `base64Encode()`, `base64Decode()`, `jsonEncode()` и `jsonDecode()`.

io

Библиотека `io` (input-output, ввод-вывод) предоставляет различные механизмы для работы с файлами, сокетами, сетью и другими функциями ввода/вывода. Вероятно, наиболее важные компоненты – это:

- `File` – класс, представляющий файл в файловой системе. Среди множества доступных операций вы можете вызвать `copy()`, `create()`, `openRead()`, `openWrite()`, `rename()` и `length()`;
- `Directory` – класс, представляющий каталог в файловой системе. Среди множества доступных операций стоит выделить `create()`, `list()`, `rename()` и `delete()`;
- `HttpClient` – класс, который можно использовать для получения контента с удаленного сервера по протоколу HTTP. Вместе с ним существует класс Cookie для работы с HTTP-файлами `cookie`, `HttpClientBasicCredentials` для поддержки BASIC Auth, HttpHeaders для работы с HTTP-заголовками и даже HttpServer, если ваше приложение работает в качестве HTTP-сервера!
- `Socket` – класс для выполнения низкоуровневой связи через сокет TCP;
- `exit()` – функция верхнего уровня для выхода из процесса Dart VM с заданным кодом ошибки. Вряд ли вы захотите использовать это в мобильном приложении, но если вы пишете стандартную программу Dart, то вам может понадобиться данный метод.

В библиотеке io определенно гораздо больше классов, связанных с HTTP-коммуникациями, но выше описано то, что вы будете использовать чаще всего.

math

Во всех языках программирования есть математические функции (я уверен, что вы можете найти хотя бы один язык, в котором их нет, но это просто странно!), и Dart не исключение благодаря библиотеке `math`. Здесь вы найдете математические константы и функции, включая генерацию случайных чисел.

- `Random` – класс для генерации случайных чисел, включая криптографически безопасные случайные числа с помощью метода `secure()`.

- `pi` – почтенная константа, которую вы знаете и любите. Я люблю!
- `cos()` – функция для получения косинуса значения, сначала требуя преобразования радиан в `double`. Большинство других тригонометрических функций, которые вы знаете и любите (или ненавидите, в зависимости от того, как вы учились!), также присутствуют здесь: `acos()`, `asin()`, `atan()`, `sin()` и т. д.
- `max()` – возвращает большее из двух чисел.
- `min()` – возвращает меньшее из двух чисел.
- `sqrt()` – возвращает квадратный корень числа.

Вспомогательные библиотеки

Наконец, у нас есть еще несколько вспомогательных библиотек. Конечно, их немного больше, чем описано ниже, но для наших примеров будет достаточно и трех.

crypto

Если вам нужна криптография, то библиотека `crypto` – это именно то, что вам нужно! Например, если вам нужно получить хеш (hash), то вы можете сделать это различными способами:

- `MD5` – класс для генерации хешей MD5. Вам даже не нужно создавать его экземпляр, потому что библиотека сама предоставляет вам объект `md5`. Думаю, что пока вы не должны использовать MD5, за исключением случаев обратной совместимости.
- `Sha1` – лучший класс для хеширования, по сравнению с MD5, в комплекте с собственным экземпляром `sha1`.
- `Sha256` – `Sha1` недостаточно хорош для вас? Хорошо, хорошо, вы можете использовать Sha256 вместо него! Экземпляр `sha256` готов и ждет вас.

collection

Подождите-ка, мы уже видели библиотеку `collection`, не так ли? Действительно! Я думаю, Google решил, что вы еще не получили свою коллекцию, так что есть еще одна! В ней вы найдете еще больше коллекций, таких как:

- `CanonocalizedMap` – класс для работы с отображением (`map`) «ключ–значение», ключи которого преобразуются в канонические значения указанного типа. Это может пригодиться, когда вы хотите, чтобы в коллекции были ключи без учета регистра или `null`;

- `DelegatingSet` – класс, который делегирует все операции над заданным множеством (set) значений. Удобно, когда вы хотите скрыть методы изменения объекта `Set`;
- `UnionSet` – класс, который позволяет объединять несколько множеств (set) в одно;
- `binarySearch()` – функция двоичного поиска по списку;
- `compareNatural()` – функция для сравнения двух строк в соответствии с естественным порядком сортировки;
- `mergeMaps()` – функция, которая объединяет два экземпляра `Map` и возвращает новый объединенный `Map`;
- `shuffle()` – функция, которая переставляет элементы списка случайным образом.

convert

И, как и в случае с `collection`, если вы думали, что у вас достаточно способов конвертировать одни данные в другие, то Google явно не согласен, потому что есть еще одна библиотека `convert`! В ней есть несколько интересных возможностей:

- `HexCodec` – класс для всех ваших массивов шестнадцатеричных строк! Эта библиотека предоставляет вам экземпляр для использования и содержит типичные методы `encode()` и `decode()`;
- `PercentCodec` – это немного странно названный класс, потому что под percent (процент) он подразумевает преобразование строки в формат URL-адреса, например замену пробелов на %20. Как и в случае с `HexCodec`, вы обнаружите, что экземпляр `percent` уже готов к использованию.

Итого

В этой главе, как и в предыдущей, мы прокатились на самолете на высоте нескольких километров и полюбовались пейзажем Flutter! В процессе работы вы получили представление о многих (о большинстве!) виджетах, с которыми поставляется Flutter. Вы также познакомились с новыми API, которые, как и предыдущие две главы, создают необходимую основу для написания приложений Flutter!

И в следующей главе мы займемся именно этим! Первое созданное приложение будет не слишком сложным с технической точки зрения, но оно послужит отличным первым шагом в мир Flutter.

Давайте уже перейдем к кодированию, хорошо?

ГЛАВА 5

FLUTTERBOOK. ЧАСТЬ I

Итак, друг мой, пришло время немного повеселиться! Мы пробежались по Flutter, и у вас хорошая база знаний, так что пришло время использовать их и начать создавать приложения! В следующих пяти главах мы создадим три приложения, начиная с FlutterBook.

В процессе вы получите реальный опыт работы с Flutter, именно то, что вам нужно для достижения следующего уровня освоения Flutter.

Итак, давайте вернемся к приложению, начав с разговора о том, что же мы собираемся сделать!

Что мы делаем?

Термин PIM стал популярным во времена устройств Palm Pilot, хотя существовал и до этого. PIM расшифровывается как Personal Information Manager (персональный информационный менеджер) и, по сути, представляет собой причудливое название приложения (или устройства, в случае с Palm Pilot), которое хранит необходимую пользователю информацию в структурированном виде. До эпохи электроники у вас мог быть небольшой блокнот с разделами для различной информации, PIM – это примерно то же самое. Для большинства людей существует четыре основных вида данных PIM: встречи, контакты, заметки и задачи. Могут быть и другие, и даже между этими четырьмя могут быть «накладки», но именно четыре раздела, как правило, считаются основными, и именно их будет содержать наш FlutterBook.

В этом приложении будут представлены четыре «сущности», которые я буду использовать для общего обозначения встреч, контактов, заметок и задач. Приложение позволит пользователю создавать, сохранять, просматривать, ре-

дактировать и удалять элементы каждого типа. Когда мы создадим приложение, мы разобьем его на модули, чтобы позже вы могли добавить другие модули для работы с иными типами данных. Например, закладки или, может быть, рецепты, если вы повар. Вы сможете добавить их без особых затруднений, потому что мы разработаем код, который будет модульным и легко расширяемым.

Хорошо, конечно, говорить об этом, но, может, стоит увидеть? Именно поэтому я покажу вам рис. 5-1.

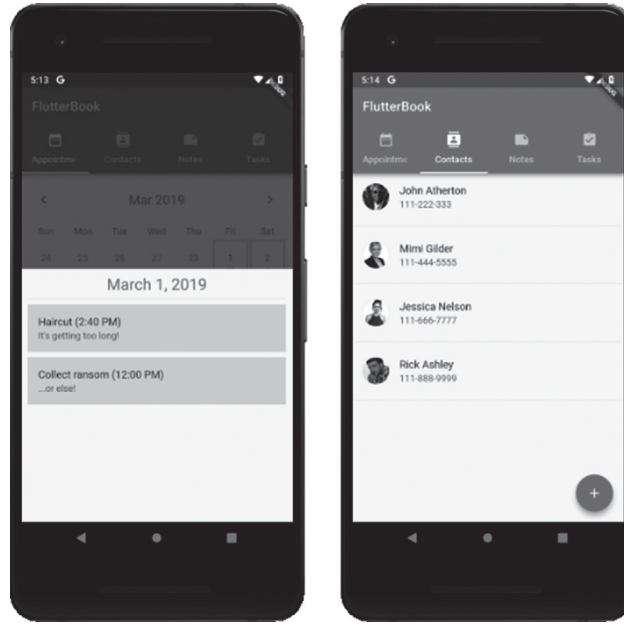


Рисунок 5-1. *FlutterBook, экран списка контактов и встреч*

Как видите, в верхней части находятся вкладки, по которым пользователь может щелкнуть, чтобы перемещаться между четырьмя типами данных (между ними можно также перемещаться с помощью свайпов, правда, пролистывание немного проблематично из-за функциональности, представленной на экранах, но об этом мы поговорим позже).

У каждой вкладки будет два экрана для работы: экран списка и экран ввода. Здесь вы можете видеть экраны со списками, хотя для встреч слева термин «список» немного ошибочен, потому что на самом деле вы видите гигантский календарь, с которым пользователь может взаимодействовать. Однако для контактов это действительно список.

Для заметок и задач используется аналогичный шаблон, как на рис. 5-2.

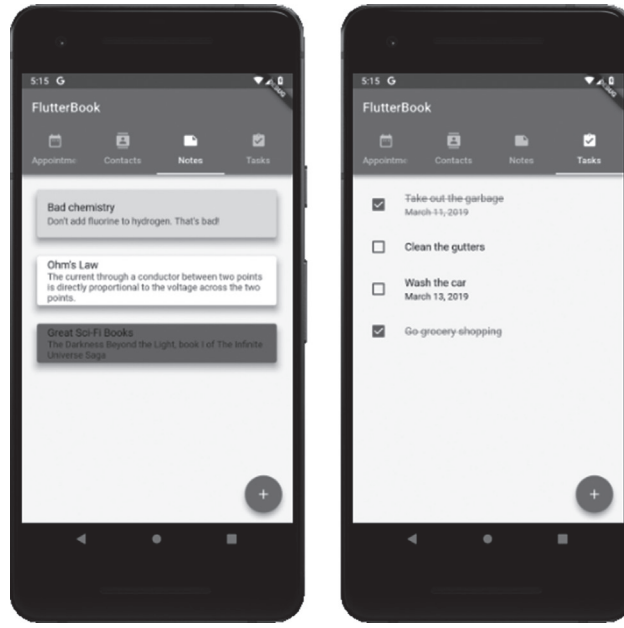


Рисунок 5-2. *FlutterBook, экраны вкладок Notes и Tasks*

Каждый экран со списком немного отличается от других, так как отличаются типы объектов: встречи (`appointments`) должны быть в календаре, в то время как контакты (`contacts`) должны показывать аватарки, заметки (`notes`) выглядят (примерно) как карточки (`Cards`), а список задач (`tasks`) позволяет пользователю отмечать выполненные пункты. Все это разнообразие возможностей предоставляет Flutter!

Экраны ввода, которые рассмотрим для каждого типа вкладок, дадут вам представление о том, как всё это выглядит.

Примечание. В этой главе и фактически во всех остальных главах данной книги я удалил комментарии, операторы `print()` и некоторые пробелы, так что код в репозитории с примерами будет выглядеть немного иначе. Но не бойтесь, фактически исполняемый код идентичен примерам в книге.

Старт проекта

Для создания FlutterBook я просто использовал мастера новых проектов, предоставленного Android Studio, так и начались все примеры в этой книге. Это дает нам необходимую структуру и полностью работающее приложение. Далее мы начинаем добавлять и редактировать элементы по мере необходимости, начиная с настройки проекта.

Конфигурации и библиотеки

Файл pubspec.yaml, показанный в листинге 5-1, содержит большую часть необходимых настроек, но поскольку этот проект потребует от нас использования плагинов (plugins, сторонние загружаемые библиотеки), нам нужно их добавить. Вы можете увидеть это в разделе dependencies (зависимости):

Листинг 5-1. *Файл pubspec.yaml*

```
name: flutter_book
description: flutter_book
version: 1.0.0+1
environment:
   sdk: ">=2.1.0 <3.0.0"
dependencies:
   flutter:
      sdk: flutter
   scoped_model:1.0.1
   sqflite: 1.1.2
   path_provider: 0.5.0+1
   flutter_slidable: 0.4.9
   intl: 0.15.7
   image_picker: 0.4.12+1
   flutter_calendar_carousel:1.3.15+3
   cupertino_icons: ^0.1.2

dev_dependencies:
   flutter_test:
      sdk: flutter

flutter:
   uses-material-design: true
```

Внимание! Помните, что файлы YAML чувствительны к отступам! Например, если одна из этих зависимостей имеет неправильный отступ (здесь «правильно» – два пробела от его родительского элемента), вы столкнетесь с проблемами. Обратите внимание, что потомком flutter является sdk, но scoped_model – потомок зависимостей, а не flutter, поэтому scoped_model должен быть в двух пробелах справа от dependencies, а не в двух пробелах справа от flutter. Это типичная ошибка (просто спросите моего замечательного технического рецензента!), особенно если вы новичок в структуре YAML.

Здесь довольно много плагинов, и вы, конечно, познакомитесь не со всеми, но я дам вам общий обзор:

- scoped_model – предоставляет очень хороший способ управления состоянием во всем приложении;
- sqflite – поскольку хранение данных – это требование приложения, мы должны выбрать, как это сделать, и я решил использовать популярную базу

данных SQLite, к которой этот плагин предоставляет нам доступ (и нет, в названии нет опечатки!);

- `path_provider` – для контактов нам нужно будет сохранить изображение аватара, если оно есть, и SQLite не лучшее место для этого. Вместо него мы будем использовать файловую систему. Каждое приложение получает свой собственный каталог документов, в котором мы можем хранить произвольные файлы, и плагин `path_provider` помогает нам в этом;
- `flutter_slidable` – нужен при удалении контактов, заметок и задач, теперь пользователь может свайпнуть эти виджеты на экране списка;
- `intl` – нам понадобятся функции форматирования даты и времени, поскольку некоторые из наших объектов работают с датами и временем;
- `image_picker` – этот плагин обеспечивает инфраструктуру, которая понадобится приложению, чтобы позволить пользователю добавлять аватарки для контактов из галереи или с камеры устройства;
- `flutter_calendar_carousel` – этот виджет предоставляет календарь для экрана списка встреч.

Все остальное в файле `pubspec.yaml` к настоящему моменту должно показаться вам знакомым, и, помимо перечисленных зависимостей, здесь есть те строки, которые мастер проектов создал для нас.

Структура UI

Базовая структура UI (пользовательского интерфейса) приложения показана на рис. 5-3.

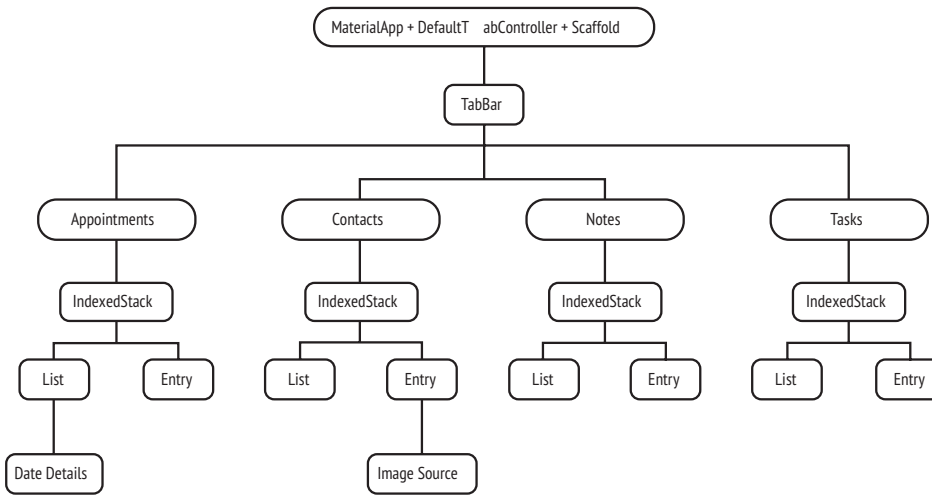


Рисунок 5-3. *Базовая структура UI*

Хотя это не показывает все до мельчайших деталей, схема дает нам картину объектов верхнего уровня.

Сверху – основной виджет MaterialApp, под которым находится DefaultTab Controller, а под ним – Scaffold. Под ними – TabBar. Под TabBar расположено четыре основных экрана, по одному для каждой из четырех вкладок. У каждого из экрана по два «подэкрана», экраны списка и ввода – это дочерние элементы виджета IndexedStack, который позволяет отображать любой из двух экранов, просто изменяя поле index (порядок экрана внутри IndexedStack). Под экраном списка встреч мы видим BottomSheet, который показывает детали для выбранной даты, а под экраном ввода контактов отображается диалоговое окно, в котором пользователь может выбрать источник изображения (камеру или галерею).

Элементы списка и экраны ввода для каждого типа, конечно, сложнее, но мы рассмотрим их позже. Давайте сначала поговорим об основной структуре приложения с точки зрения кода.

Структура кода приложения

Что касается структуры каталогов приложения, она на 100 % стандартна, здесь нет ничего нового. Весь код приложения находится в каталоге lib, как всегда, хотя на этот раз, учитывая количество файлов и желание сделать его хотя бы несколько модульным, каждый тип сущности получает свой собственный каталог. Таким образом, каталог lib/contacts содержит файлы, связанные с контактами, lib/notes-файлы, связанные с заметками, и т. д.

В каждом из них вы найдете один и тот же базовый набор файлов, и во всех последующих случаях xxx – это имя объекта Appointments, Contacts, Notes или Tasks:

- xxx.dart – основная точка входа на каждый из этих экранов;
- xxxList.dart – экран списка;
- xxxEntry.dart – экран ввода;
- xxxModel.dart – эти файлы содержат классы для представления каждого типа данных, а также объект model, как того требует scoped_model (его мы обсудим позже);
- xxxDBWorker.dart – эти файлы содержат код, который работает с SQLite. Он обеспечивает уровень абстракции над базой данных, так что вы можете изменить механизм хранения данных, не изменяя код приложения, а просто изменив эти файлы.

Отправная точка

Пришло время начать смотреть на код! Как обычно, все начинается в файле main.dart в корне проекта:

```
import "dart:io";
import "package:flutter/material.dart";
import "package:path_provider/path_provider.dart";
import "appointments/Appointments.dart";
import "contacts/Contacts.dart";
import "notes/Notes.dart";
import "tasks/Tasks.dart";
import "utils.dart" as utils;

void main() {
   startMeUp() async {
      Directory docsDir =
         await getApplicationDocumentsDirectory();
      utils.docsDir = docsDir;
      runApp(FlutterBook());
   }
   startMeUp();
}
```

Давайте на этом остановимся и обсудим представленный фрагмент (я обычно разбиваю листинг, чтобы представить его в более удобоваримой форме и дать вам понять, что происходит).

Во-первых, у нас есть `import`. Вы уже знаете, что `material.dart` – это код классов `Material` Flutter. Нам нужна библиотека `io` и плагин `path_provider` для получения каталога документов приложения (мы скоро к этому вернемся). Остальное – классы приложения. Четыре файла экранов импортируются, а затем добавляется файл `utils.dart`. Мы рассмотрим его в следующем разделе, но вкратце: он содержит функции и переменные, которые имеют глобальное значение для всего кода и поэтому находятся в этом файле.

Далее следует обычная функция `main()`, с которой начинается запуск приложения. Тут есть маленький нюанс: нам нужно получить каталог документов приложения. Импорт `path_provider.dart` предоставляет нам функцию `getApplicationDocumentsDirectory()`, которая возвращает объект `Directory`, предоставляемый импортированной библиотекой `Dart:io`. В дополнение к этой функции данный плагин также предоставляет метод `getExternalStorageDirectory()`, который доступен только на Android (и только на некоторых устройствах), поэтому обычно перед его использованием следует проверить тип ОС. Этот метод возвращает путь к корневой папке хранилища на внешнем устройстве (обычно на SD-карте), где приложение может считывать и записывать данные. Наконец, есть функция `getTemporaryDirectory()`. Она возвращает путь к временному каталогу приложения (где вы обычно храните временные данные, в отличие от `getApplicationDocuments Directory()`, который обеспечивает долговременное хранение.

Однако есть проблема: мы должны убедиться, что никакой другой код не выполняется до тех пор, пока наш асинхронный код не завершится, потому что в противном случае мы будем получать исключения (`exceptions`) из-за от-

сутствия доступа к файлу базы данных. Как вы увидите позже, каждая из четырех баз данных, по одной для каждой вкладки, представляет собой отдельный файл SQLite, хранящийся в каталоге документов приложения. И если новый код вызывается до определения docDir, то во время загрузки экранов и создания основного виджета у нас будут проблемы. Итак, для этого я создаю функцию внутри main() (да, вы можете сделать это в Dart!) и гарантирую, что данная функция асинхронна, потому что мы будем ждать вызова getApplication DocumentsDirectory(). Как только она возвращает Directory, этот объект сохраняется в utils.docsDir (так что нам нужно получить ссылку на этот каталог только один раз), а затем вызывает обычный runApp(), передавая ему новый экземпляр класса FlutterBook.

Примечание. Это не обязательно лучший способ, так как пользовательский интерфейс не будет создан до тех пор, пока getApplicationDocumentsDirectory() не завершится. Обычно это не очень хорошо с точки зрения пользовательского интерфейса, но, учитывая, что это не займет слишком много времени, я думаю, это самый простой способ получить нужный результат.

Затем создается основной виджет, в который входит класс FlutterBook, как вы можете видеть ниже:

```
class FlutterBook extends StatelessWidget {

  Widget build(BuildContext inContext) {
    return MaterialApp(
      home : DefaultTabController(
        length : 4,
        child : Scaffold(
          appBar : AppBar(
            title : Text("FlutterBook"),
            bottom : TabBar(
              tabs : [
                Tab(icon : Icon(Icons.date_range),
                  text : "Appointments"),
                Tab(icon : Icon(Icons.contacts),
                  text : "Contacts"),
                Tab(icon : Icon(Icons.note),
                  text : "Notes"),
                Tab(icon : Icon(Icons.assignment_turned_in),
                  text : "Tasks")
              ]
            )
          ),
          body : TabBarView(
            children : [
              Appointments(), Contacts(), Notes(), Tasks()
```

```
                    ]
                )
            )
        )
    );
  }
}
```

Во-первых, у нас есть MaterialApp, который я упоминал ранее, с DefauItTab Controller в качестве главного экрана. DefaultTabController – это тип Tab Controller, отвечающий за координацию выбора вкладок в TabBar, который, как вы можете видеть, является нижним дочерним элементом AppBar под Scaffold. Controller заботится о переключении между дочерними элементами, которые определяются свойством tabs у TabBar. Каждая запись во вкладках – это объект Tab, который может иметь значок и надпись, поэтому я решил показать оба случая. С такой конфигурацией вам не нужно ничего делать, чтобы переключаться между экранами; эти виджеты позаботятся обо всем за вас.

Наконец, тело Scaffold должно быть TabBarView, чтобы его можно было правильно отображать с помощью TabBar и правильно управлять им с помощью DefaultTabController. Дочерние элементы нашего TabBarView – четыре экрана, по одному для каждой вкладки (и, очевидно, именно в этом и заключается основная часть приложения; мы же вскоре приступим к реализации, но у нас есть несколько других тем, на которых нужно остановиться подробнее, начнем с utils.dart).

Глобальные утилиты

Файл utils.dart содержит глобальные данные, о которых я говорил ранее, поэтому давайте посмотрим на них:

```
import "dart:io";
import "package:flutter/material.dart";
import "package:path_provider/path_provider.dart";
import "package:intl/intl.dart";
import "BaseModel.dart";

Directory docsDir;
```

Как вы видели ранее, docDir – это каталог документов приложения, который был использован в main().

Далее мы находим в этом файле функцию selectDate():

```
Future selectDate(
   BuildContext inContext, BaseModel inModel,
   String inDateString
) async {
```

```
    DateTime initialDate = DateTime.now();
    if (inDateString != null) {
      List dateParts = inDateString.split(",");
      initialDate = DateTime(
        int.parse(dateParts[0]),
        int.parse(dateParts[1]),
        int.parse(dateParts[2])
      );
    }

    DateTime picked = await showDatePicker(
      context : inContext, initialDate : initialDate,
      firstDate : DateTime(1900), lastDate : DateTime(2100)
    );

    if (picked != null) {
      inModel.setChosenDate(
        DateFormat.yMMMMd("en_US").format(picked.toLocal())
      );
      return "${picked.year},${picked.month},${picked.day}";
    }
}
```

Сейчас эту функцию будет немного сложно объяснить полностью, потому что она зависит от вещей, с которыми вы еще незнакомы, но вы можете вернуться к ней позже, когда у вас будет нужная информация.

Во-первых, метод `selectDate()` используется для выбора даты на экранах добавления встреч, контактов и задач (дата встречи, день рождения контакта или срок выполнения задачи). Он должен быть универсальным и работать со всеми тремя типами вкладок (и, возможно, с другими позже). Итак, то, что передается в метод `selectDate()`, – это контекст `inContext` экрана ввода, с которого он вызывается, наряду с объектом `BaseModel` и датой в строковой форме. `BaseModel` – объект, для которого в конечном итоге будет выбрана нужная дата. Переданная дата, которая является необязательной, будет иметь формат `yyyy`, `mm`, `dd`, и это общий формат во всем коде. Причина в том, что при сохранении даты в SQLite тип данных даты недоступен, поэтому имеет смысл сохранить ее в виде строки. Я выбрал этот формат, потому что он облегчает создание объекта `DateTime`, поскольку его конструктор принимает фрагменты информации именно в таком порядке. Если дата передается, функция `split()` используется для ее разделения на части, из которых создается `DateTime`, сначала год, затем месяц, потом день, результат отображается в виде строки.

`initialDate` – это день, который будет заранее выбран при отображении всплывающего календаря, это применимо только при редактировании объекта (при создании не будет указано `initialDate`, чтобы календарь выбрал текущий день).

Затем вызывается метод `showDatePicker()`, который вы видели в предыдущих двух главах. Он отображает всплывающий календарь и возвращает

экземпляр DateTime. Обратите внимание, что диапазон выбираемых дат начинается с 1900-го года, а заканчивается 2100-м. Логичнее было бы ограничить его в зависимости от типа объекта (нет смысла создавать встречу для даты в прошлом), и именно здесь вступают в игру firstDate и lastDate. Но, просто чтобы уменьшить объем кода, вместо этого я выбрал диапазон, который, по крайней мере, номинально работал бы для всех типов вкладок.

Когда showDatePicker() завершается, то мы узнаем, выбрал пользователь дату или нет. Возвращенное значение будет null, если пользователь нажмет кнопку **Cancel**. В противном случае для выбранной даты у нас будет объект DateTime. Теперь, как я упоминал ранее, мы должны сохранить дату в экземпляре BaseModel с помощью вызова функции setChosenDate(). Переданное значение должно быть в понятном формате, а так как у класса DateTime нет метода toString(), то мы используем возможности intl.dart, предварительно его импортировав. В частности, функция DateFormat.yMMMMd.format() предоставляет строку в форме «MONTH dd, yyyy», где MONTH – полное имя месяца (январь, февраль, март и т. д.). Этот плагин содержит множество вариантов форматирования даты и времени, а также другие механизмы интернационализации и локализации приложений. Для получения дополнительной информации смотрите документацию по intl.dart: https://pub.dartlang.org/packages/intl (я буду редко описывать эти модули целиком, слишком много подробностей, поэтому мы обсудим только код path_provider, который предлагает необходимые возможности!).

Тем не менее мы еще не закончили! Код, вызвавший эту функцию, нуждается в получении даты. Формат, в котором дата была возвращена, остается прежним: yyyy,mm,dd.

Как я уже сказал, функция setChosenDate() будет иметь больше смысла, когда вы узнаете о моделях, а позже увидите ее использование – в следующей главе, поэтому давайте не будем на ней сейчас останавливаться, а лучше рассмотрим применение моделей для управления состояниями виджетов.

Управление состоянием

Концепция State и управления состояниями определяют то, как ваши виджеты создают и используют данные и как ваш код взаимодействует с ними – это тема, которую чаще всего оставляют на усмотрение самих разработчиков. Flutter на момент написания данной книги не говорил ничего определенного по тому, как правильно работать с состояниями (ходят слухи, что вскоре у Flutter может появиться канонически «правильный» подход к управлению состояниями, но пока это только слухи).

Конечно, есть потомки виджета StatefulWidget, который умеет управлять своим состоянием и который мы изучили в предыдущих главах (вы также увидите его снова до конца книги). Это действительно управление состояниями. Но на самом деле это только один *тип* состояний: локальные. Другими слова-

ми, такие состояния локальны для отдельного виджета, и для управления ими обычно достаточно потомков State и StatefulWidget.

Но есть другой тип состояния – состояние, которое вы можете считать «глобальным». Другими словами, это состояние может быть необходимо вне виджета и, во многих случаях, вне времени его жизни. Например, дочерним виджетам может быть необходимо состояние родительских виджетов. Или наоборот, родительский виджет (а может, и не *прямой* родительский класс) нуждается в доступе к состоянию его дочернего элемента. Если все, с чем вам нужно работать, – это виджеты с отслеживанием состояния, а парадигма setState() используется по умолчанию, то с подобными задачами может быть непросто справиться.

Как я уже сказал, Flutter не дает однозначного ответа. Существует множество решений для управления состояниями, доступных во Flutter, помимо setState(), и это лишь некоторые из них: BLoC, Redux и scoped_model. Их, вероятно, еще дюжина, каждый со своими плюсами и минусами. Таким образом, используемый вами подход к управлению состоянием будет зависеть от многих факторов, включая цели вашего проекта, конкретные взаимодействия между состояниями, которые вам нужны, и, в конце концов, ваши личные предпочтения относительно структурирования своего кода.

В проекте FlutterBook и фактически в оставшейся части этой книги я сосредоточусь на одном конкретном подходе из этого списка – scoped_model. Подход scoped_model – возможно, самый простой вариант, потому что он делает код проще, и мне это нравится! Чем проще, тем лучше! Вы, конечно, обязательно должны изучить другие варианты и посмотреть, что подходит под вашу ментальность и формат мышления. Если этим вариантом окажется scoped_model, то отлично! Если нет, то нет проблем, мы все еще можем быть друзьями, и, по крайней мере, прочитав эту книгу, вы будете хорошо понимать scoped_model, чтобы проводить осмысленное сравнение с остальными.

Итак, что такое scoped_model? Ну, это всего лишь три простых класса, которые предоставляют (в сочетании с тремя простыми шагами с вашей стороны) специальное хранилище данных для дерева виджетов.

Первый класс Model из библиотеки scoped_model требует, чтобы вы создали от него потомка и именно здесь разместили свою логику обработки данных и ваши переменные. Обратите внимание, что вам может не потребоваться какая-либо реальная логика, и это совершенно нормально (хотя и немного нетипично). Основная цель помещения кода в потомка Model заключается в том, чтобы вы могли вызвать метод notifyListeners() базового класса (который можно вызывать только из подкласса Model). Это секретный ингредиент! Вызов этого метода информирует любой виджет, который был «подключен» к классу модели, о том, что модель изменилась, и они должны, при необходимости, перестроить себя.

Второй шаг – подключение scoped_model к дереву виджетов. Эта часть очень проста: оберните виджет во второй класс, о котором вы должны знать, ScopedModel. Например, если ваш самый верхний виджет – это Column, то вы можете сделать следующее:

```
return ScopedModel<your-model-class-here>(
   child : Column(...)
)
```

На самом деле вам даже не нужно помещать самый верхний виджет в общее дерево, хотя это и наиболее популярный вариант, чтобы любой виджет в дереве мог иметь доступ к вашей модели. Но если доступ к модели требуется только подмножеству виджетов, вы можете вместо этого выбрать родительский виджет, даже если он не самый верхний, и обернуть его в ScopedModel. В любом случае вы должны сообщить ScopedModel тип через шаблонную нотацию <your-model-class-here> (это потомок базового класса Model).

И наконец, для всех дочерних виджетов, на которые мы хотим повлиять и чей родитель обернут в ScopedModel, нужно использовать класс ScopedModel Descendent. Как и в ScopedModel, вам не нужно оборачивать каждый виджет отдельно; оборачивание одного покроет и все его дочерние элементы. Любые виджеты, обернутые этим классом, будут перестраиваться при изменении модели (конечно, при условии что алгоритм сравнения Flutter определит, что это необходимо). Синтаксис ScopedModelDescendent немного отличается от ScopedModel, потому что требуется паттерн builder:

```
return ScopedModel<your-model-class-here>(
   child :
      ScopedModelDescendent<your-model-class-here>(
         builder : (BuildContext inContext, Widget inChild,
            <your-model-class-here> inModel) {
               return Column(...);
      )
   );
)
```

Теперь если у вас есть виджет Text внутри Column, с помощью которого вы хотите отобразить значение переменной из inModel, то вы можете сделать

```
Text(inModel.myVariable)
```

И вуаля, у вас есть готовое к использованию хранилище данных, которое является состоянием вашего приложения. Оно обновит ваш пользовательский интерфейс при изменении данных, без использования класса State (верно, вы можете делать все это с виджетами StatelessWidget!) и более глобальным способом.

Последний фрагмент головоломки – это изменение состояния. Чтобы с этим разобраться, давайте посмотрим на реальный класс модели, файл BaseModel. dart из FlutterBook. Прежде чем мы начнем это исследование, позвольте мне сказать, что каждый тип вкладки, с которой работает FlutterBook, имеет свой собственный класс модели. Вам *не нужно* делать это таким образом – у вас ведь может быть один класс модели, который сразу содержит данные для всех четырех типов вкладок. Но я считаю, что логичнее их разделять. Дело в том, что

наши модели данных имеют несколько общих характеристик, поэтому, вместо того чтобы дублировать код, я создал класс BaseModel, который наследуется от Model. Классы моделей для отдельных типов вкладок наследуются от BaseModel, а это означает, что они также происходят от класса scoped_model, как нам и нужно.

```
import "package:scoped_model/scoped_model.dart";
```

Очевидно, что не получится использовать библиотеку scoped_model, если мы ее не импортируем, поэтому начинаем именно с этого. Затем идет класс Base Model:

```
class BaseModel extends Model {
```

Видите, он действительно наследуется от класса Model из библиотеки scoped_ model!

```
int stackIndex = 0;
List entityList = [ ];
var entityBeingEdited;
String chosenDate;
```

В этой модели мы постарались выделить все общие части наших вкладок. Помните, что каждый из четырех экранов во вкладках на самом деле содержит два экрана (список объектов и редактирование), дочерних элемента IndexedStack? Так вот, то, что именно сейчас отображается на экране, будет зависеть от значения переменной stackIndex. Кроме того, поскольку все четыре типа вкладок имеют списки с данными, здесь определено поле enityList. А свойство entityBeingEdited будет ссылкой на объект, который пользователь выберет, если захочет изменить существующий элемент списка. Таким способом данные для вкладки передаются с экрана списка на экран ввода. Наконец, переменная chosenDate будет хранить дату, выбранную пользователем при редактировании записи. Вы увидите, зачем это необходимо, уже в ближайшее время, но пока давайте продолжим с классом BaseModel.

```
void setChosenDate(String inDate) {
   chosenDate = inDate;
   notifyListeners();
}
```

Когда пользователь выбирает дату, он делает это через всплывающее окно, но затем выбранная дата должна вернуться в модель. Вызов метода setChosen Date() реализует данную логику. Как видите, последнее, что он делает, – это вызывает notifyListeners(). Именно данный вызов обновляет экран, чтобы показать выбранную дату. Без этого данные будут сохранены в модели, но пользователь не узнает об этом, посмотрев на экран, потому что виджеты, обернутые в ScopedModel (и ScopedModelDescendent), не будут перерисованы.

```
void loadData(String inEntityType, dynamic inDatabase) async {
   entityList = await inDatabase.getAll();
   notifyListeners();
}
```

Метод `loadData()` будет вызываться всякий раз, когда объект добавляется или удаляется из `entityList` (код, который вы увидите в ближайшее время). Это использует класс `xxxDBWorker`, который знает, как контактировать с SQLite. Еще раз, мы перейдем к этому в ближайшее время, но на данный момент просто обратите внимание, что результат вызова метода `getAll()` заменяет `entityList`, а затем `notifyListeners()` снова вызывается, чтобы список на вкладке перерисовывал себя.

Наконец, у нас есть метод `setStackIndex()`:

```
void setStackIndex(int inStackIndex) {
   stackIndex = inStackIndex;
   notifyListeners();
}
```

Он будет вызываться всякий раз, когда мы захотим перемещать пользователя между экранами списка и ввода.

Я понимаю, что у вас еще нет полного контекста использования кода `Base Model`, но скоро будет! На данный момент основные понятия `scoped_model` – вот что важно, и надеюсь, они начинают обретать какой-то смысл. Кстати, сейчас мы увидим, как все это работает, на примере вкладки **Notes**.

Начнем с простого: заметки

Я думаю, что из четырех типов вкладок код для заметок наиболее простой. Поэтому начнем с него.

Точка отсчета: Notes.dart

Как вы помните, каждая из четырех вкладок имеет главный экран, который является основным ее содержимым. Файл `Notes.dart` реализует код для этого экрана и начинается с импорта:

```
import "package:flutter/material.dart";
import "package:scoped_model/scoped_model.dart";
import "NotesDBWorker.dart";
import "NotesList.dart";
import "NotesEntry.dart";
import "NotesModel.dart" show NotesModel, notesModel;
```

Помимо обычных компонентов, таких как `material.dart`, у нас есть `scoped_model.dart`. Вы увидите, что все дерево виджетов для этого экрана будет иметь

доступ к модели заметок. Нам также нужно добавить файл NotesDBWorker.dart, чтобы мы могли загрузить заметки. Затем нам нужен исходный код для двух подэкранов: NotesList.dart и NotesEntry.dart. Наконец, нам нужна модель для заметок в NotesModel.dart. Продолжим рассматривать файл, описывающий данный экран:

```
class Notes extends StatelessWidget {
```

Обратите внимание, что это виджет без сохранения состояния. Помните: использование scoped_model означает, что вы имеете дело с состоянием, но это не значит, что вы должны использовать только потомков StatefullWidget.

После идет конструктор:

```
Notes() {
   notesModel.loadData("notes", NotesDBWorker.db);
}
```

Напомним, что у BaseModel есть метод loadData(), и он написан обобщенно, поэтому будет работать с любым типом вкладок. Тем не менее единственная причина, по которой метод может быть записан обобщенно, в том, что здесь конструктор его вызывает и предоставляет информацию об объекте – а именно тип объекта и ссылку на базу данных для этого типа объекта. После вызова конструктора Notes() свойство entityList будет содержать список заметок, загруженных из базы данных SQLite, и поэтому позже эти данные мы сможем отобразить на экране. Технически, поскольку эта загрузка данных происходит асинхронно, экран списка отобразится до того, как данные будут загружены, однако благодаря scoped_model и методу loadData(), вызывающему notifyListeners(), экран получает уведомление и перерисует себя уже с данными.

```
Widget build(BuildContext inContext) {

   return ScopedModel<NotesModel>(
      model : notesModel,
      child : ScopedModelDescendant<NotesModel>(
         builder : (BuildContext inContext, Widget inChild,
            NotesModel inModel
         ) {
            return IndexedStack(
               index : inModel.stackIndex,
               children : [ NotesList(), NotesEntry() ]
            );
         }
      )
   );
}
```

Наконец, рассмотрим виджет, возвращаемый из метода build(). Он начинается со ScopedModel и содержит ScopedModelDescendent. IndexedStack ис-

пользуется для хранения двух экранов. Примечательно, что значение индекса IndexedStack – это ссылка на поле stackIndex в экземпляре NotesModel.

Управление экранами в IndexedStack реализовано очень просто: установите значение stackIndex равным 0, и будет показан NotesList, а если задать 1 – отобразится NotesEntry.

Модель: NotesModel.dart

Класс модели для вкладки с заметками находится в NotesModel.dart. NotesModel – это не просто класс модели, но также класс, описывающий заметку.

Прежде всего мы начнем с импорта:

```
import "../BaseModel.dart";
```

Наш класс будет наследоваться от BaseModel, который, в свою очередь, наследуется от Model из scoped_model, поэтому нам нужен этот импорт.

Далее у нас идет определение класса с данными:

```
class Note {

   int id;
   String title;
   String content;
   String color;

   String toString() {
      return "{ id=$id, title=$title, "
      "content=$content, color=$color }";
   }
}
```

Экземпляры этого класса – заметки (note). Каждая заметка содержит уникальный идентификатор, заголовок, описание и цвет, используемый для задания фона на экране списка. Хотя это и не требуется, я также добавил метод toString(), который переопределяет реализацию по умолчанию, предоставляемую классом Object. Реализация по умолчанию не совсем удобна, поскольку она просто говорит, к какому типу относится объект. Переопределенная же версия показывает информацию о заметке, что очень удобно при отладке.

Далее идет сам класс модели:

```
class NotesModel extends BaseModel {

   String color;

   void setColor(String inColor) {
      color = inColor;
      notifyListeners();
   }
}
```

Большая часть того, что нужно этому классу, обеспечивается BaseModel, поэтому нам необходимо реализовать только задание цвета. Немного забежим вперед: это необходимо потому, что, когда пользователь выбирает цвет, изменение значений в экземпляре Note не отразится на модели, и экран не узнает об изменении. Поэтому нам нужно где-то это описать.

Есть еще одна важная строка в этом файле:

```
NotesModel notesModel = NotesModel();
```

У нас есть определение класса, но пока нет ни одного его экземпляра. Его мы и создадим новой строкой. Этот код выполнится только один раз, независимо от того, сколько раз или откуда он импортирован, а это гарантирует, что у нас всегда будет только один экземпляр NotesModel, и это именно то, что нам нужно!

Слой базы данных: NotesDBWorker.dart

Следующий файл для просмотра – NotesDBWorker.dart, который содержит весь код для работы с SQLite. Разберем импорты:

```
import "package:path/path.dart";
import "package:sqflite/sqflite.dart";
import "../utils.dart" as utils;
import "NotesModel.dart";
```

Пожалуй, тут не много удивительного. Модуль path.dart содержит функции для работы с путями в файловой системе кросс-платформенным способом. Большинство этих операций типичны, но мы чуть подробнее поговорим об одной из них, которая появится позднее.

Далее идет сам класс NotesDBWorker:

```
class NotesDBWorker {

   NotesDBWorker._();
   static final NotesDBWorker db = NotesDBWorker._();
```

Первый шаг – убедиться, что существует только один экземпляр этого класса, поэтому мы собираемся реализовать паттерн singleton. Начинаем с создания закрытого конструктора, как видно в первой строке. Во второй строке вызывается конструктор, а экземпляр класса БД хранится статически и не может быть изменен.

Далее нам нужен экземпляр класса Database – ключевого класса при работе с SQLite через плагин sqflite:

```
Database _db;

Future get database async {
   if (_db == null) {
      _db = await init();
```

```
    }
    return _db;
}
```

Когда вызывается метод get для получения базы данных, мы проверяем, есть ли что-то в _db. Если есть, то вернем текущее значение, если нет, то вызовем метод init(). Это гарантирует, что в одном экземпляре NotesDBWorker будет только один объект Database, именно этого мы и хотели для сохранения целостности данных.

К слову, о методе init():

```
Future<Database> init() async {
   String path = join(utils.docsDir.path, "notes.db");
   Database db = await openDatabase(
      path, version : 1, onOpen : (db) { },
      onCreate : (Database inDB, int inVersion) async {
         await inDB.execute(
            "CREATE TABLE IF NOT EXISTS notes ("
               "id INTEGER PRIMARY KEY,"
               "title TEXT,"
               "content TEXT,"
               "color TEXT"
            ")"
         );
      }
   );
   return db;
}
```

Ключевая задача – убедиться, что база данных существует. Она будет храниться в виде файла в каталоге документов приложения, поэтому нам нужен путь к нему. Здесь используется одна функция из библиотеки path – функция join(), которая объединяет путь к каталогу документов с именем файла notes.db. Как только мы это сделаем, нужно создать объект Database из сформированного пути при помощи метода openDatabase(). Мы передаем этому методу путь, версию базы данных, а также callback-функцию, вызываемую при открытии базы данных. Далее с помощью метода execute() мы выполняем SQL-запрос для создания таблицы notes, если такая еще создана. После этого возвращается экземпляр базы данных, который сохраним в _db. Теперь мы готовы выполнять операции с базой данных!

Но прежде чем мы перейдем к этим операциям, нам нужно создать две вспомогательные функции. Проблема в том, что SQLite и sqflite ничего не знают о нашем классе Note, так как все, с чем они могут работать, – это отображения данных, реализуемые классом Map. Итак, нам нужно предоставить некоторые функции, которые могут преобразовать объект из Map в Note и наоборот. Здесь нет ничего сложного:

```
Note noteFromMap(Map inMap) {
   Note note = Note();
   note.id = inMap["id"];
   note.title = inMap["title"];
   note.content = inMap["content"];
   note.color = inMap["color"];
   return note;
}
Map<String, dynamic> noteToMap(Note inNote) {
   Map<String, dynamic> map = Map<String, dynamic>();
   map["id"] = inNote.id;
   map["title"] = inNote.title;
   map ["content"] = inNote.content;
   map["content"] = inNote.content;
   return map;
}
```

Я даже могу поспорить, для вас все совершенно очевидно, так что давайте перейдем к более захватывающим вещам: создание заметки в базе данных!

Примечание. Именно поэтому я и хотел, чтобы у меня не было одного `DBWorker` для всех вкладок. Помимо фактического отличия операторов SQL, с которыми я мог бы справиться с помощью оператора `switch`, в настоящее время у Dart нет ничего похожего на рефлексию в языке Java. Скорее всего, это появится в будущем, но на момент написания книги не было возможностей использовать динамическое преобразование данных, поэтому пришлось использовать ручное присваивание. Мне нравится Dart, но иногда я скучаю по JavaScript! [Уже есть такая библиотека `https://pub.dev/packages/reflectable`. – *Прим. перев.*]

```
Future create(Note inNote) async {
   Database db = await database;
   var val = await db.rawQuery(
      "SELECT MAX(id) + 1 AS id FROM notes"
   );
   int id = val.first["id"];
   if (id == null) { id = 1; }
   return await db.rawInsert(
      "INSERT INTO notes (id, title, content, color) "
      "VALUES (?, ?, ?, ?)",
      [ id, inNote.title, inNote.content, inNote.color ]
   );
}
```

Создание заметки состоит из трех этапов. Во-первых, нам нужно получить ссылку на объект базы данных. Во-вторых, нам нужно придумать уникальный

идентификатор для заметки. Чтобы сделать это, мы запрашиваем существующие заметки и просто увеличиваем наибольший найденный идентификатор. Однако если это первая заметка, то мы получим null и явно столкнемся с подобной ситуацией (на практике null для идентификатора работает, но мне кажется, что это «грязно», поэтому дополнительная проверка гарантирует, что мы всегда имеем дело с действительным числовым идентификатором).

Третий шаг – вызов метода rawInsert() с простым SQL-запросом, который выполняет вставку значений. Эти значения взяты из объекта Note, переданного как inNote. Как вы можете видеть, мы возвращаем объект Future, который, в свою очередь, возвращается методом rawInsert(), поэтому функция create() является асинхронной. На этом все готово!

Примечание. Если вы посмотрите API для объекта Database, то увидите, что, помимо метода rawInsert(), есть также метод insert() и аналогичное разделение для других операций. Зачем использовать одно вместо другого? По правде говоря, у меня нет веских аргументов в пользу одного из подходов! Метод insert() является более абстрактным и избавляет вас от необходимости писать SQL-запросы самостоятельно, в отличие от rawInsert(). Лично я предпочитаю работать с SQL, но если вы предпочитаете решения более высокого уровня, то можете использовать insert() вместо rawInsert(), и нет никаких веских причин для того, чтобы предпочесть последний, помимо желания писать SQL.

Далее нам нужна возможность получить указанную заметку. Хоть сейчас это и неочевидно, мы просто реализуем операции CRUD, то есть **Create** (создать), **Read** (прочитать), **Update** (обновить) и **Delete** (удалить).

```
Future<Note> get(int inID) async {
   Database db = await database;
   var rec = await db.query(
      "notes", where : "id = ?", whereArgs : [ inID ]
   );
   return noteFromMap(rec.first);
}
```

Вызывающая сторона передает идентификатор заметки, которую хочет получить, а сам запрос происходит с помощью метода query(). Этот метод принимает имя таблицы для запроса и условие where (есть несколько форм, которые данный метод может принять, это только одна из них) плюс значения для него. Здесь нам просто нужно запросить поле id. Результатом этого вызова будет объект Map, поэтому нам нужна функция noteFromMap() для возврата объекта Note.

Ниже представлен метод getAll(), который позволяет извлечь все заметки за один вызов:

```
Future<List> getAll() async {
   Database db = await database;
   var recs = await db.query("notes");
   var list = recs.isNotEmpty ?
```

```
    recs.map((m) => noteFromMap(m)).toList() : [ ];
    return list;
}
```

Здесь методу `query()` просто нужно имя таблицы, и он извлечет из нее все записи. Если у нас нет записей, то возвращается пустой список, но если мы их получили, то вызовется функция `map()` с методом `noteFromMap()` внутри. Это преобразует каждый элемент списка из `Map` в `Note`. Наконец, преобразуем результат в новый список методом `toList()`.

Обновление заметки происходит следующим образом:

```
Future update(Note inNote) async {

    Database db = await database;
    return await db.update("notes", noteToMap(inNote),
       where : "id = ?", whereArgs : [ inNote.id ]
    );
}
```

Не слишком сложно, не так ли? Метод `update()` принимает имя таблицы, объект `Map` с новыми значениями (которые мы получаем, вызывая `noteToMap()`), и условие `where` для идентификации записи по `id`. Этот метод знает, как взять `Map` и преобразовать его содержимое к форме, которая подходит для записи данных в базу.

Последний метод, который нужно рассмотреть, – это `delete()`:

```
Future delete(int inID) async {

    Database db = await database;
    return await db.delete(
       "notes", where : "id = ?", whereArgs : [ inID ]
    );
}
```

Это все, что нужно! Надеюсь, что объяснения не нужны. Итак, давайте перейдем к коду экранов, начиная со списка.

Экран списка: NotesList.dart

Экран списка заметок начинается с импорта:

```
import "package:flutter/material.dart";
import "package:scoped_model/scoped_model.dart";
import "package:flutter_slidable/flutter_slidable.dart";
import "NotesDBWorker.dart";
import "NotesModel.dart" show Note, NotesModel, notesModel;

class NotesList extends StatelessWidget {
```

Из новенького здесь импорт flutter_slidable.dart; в остальном это обычные импорты и совершенно обычное начало класса. Давайте пропустим описание библиотек, пока не встретим их, а вместо этого посмотрим на метод build():

```
Widget build(BuildContext inContext) {

   return ScopedModel<NotesModel>(
      model : notesModel,
      child : ScopedModelDescendant<NotesModel>(
         builder : (BuildContext inContext, Widget inChild,
            NotesModel inModel
         ) {
            return Scaffold(
```

У нас есть ScopedModel, который ссылается на экземпляр notesModel. Он содержит ScopedModelDescendent в качестве child, так что все дочерние элементы в этом классе могут получить доступ к модели. Далее идет функция builder(), и мы начинаем создавать виджет на базе Scaffold, как это часто бывает с экраном приложений Flutter.

```
floatingActionButton : FloatingActionButton(

   child : Icon(Icons.add, color : Colors.white),
   onPressed : () {
      notesModel.entityBeingEdited = Note();
      notesModel.setColor(null);
      notesModel.setStackIndex(1);
   }
)
```

Scaffold имеет кнопку floatingActionButton, которая позволяет пользователю создавать новые заметки. Она находится в правом нижнем углу, поверх содержимого экрана. При нажатии на нее выполняется функция onPressed, этим мы инициализируем процедуру ввода, которую начнем с создания нового экземпляра Note и сохранения его в модели как entityBeingEdited. Этот объект мы сохраним в базе данных, как только это потребуется.

Пользователь может выбрать на экране ввода цвет заметки. Вспомните, как мы говорили о том, что экран будет перерисовывать себя при изменении модели. И это будет происходить вне зависимости от того, изменял пользователь цвет или нет. Однако передать цвет в новом объекте Note недостаточно, поскольку scoped_model не будет видеть изменений в этом объекте (scoped_model не может наблюдать за свойствами вложенных объектов), поэтому, как вы видели ранее, NoteModel имеет свое свойство color. Изначально мы хотим, чтобы цвет не был выбран, поэтому вызываем метод setColor() с аргументом null, который устанавливает для модели свойство color и вызывает notifyListeners(), так что экран обновляется (на самом деле это не имеет значения, поскольку на этом этапе экран ввода не отображается).

Наконец, мы перемещаем пользователя на экран ввода, вызывая setStack Index( ) и передавая ему значение, равное единице. После этого определяется свойство body виджета Scaffold, и именно здесь мы начинаем рисовать список заметок:

```
body : ListView.builder(
   itemCount : notesModel.entityList.length,
   itemBuilder : (BuildContext inBuildContext, int inIndex) {
      Note note = notesModel.entityList[inIndex];
      Color color = Colors.white;
      switch (note.color) {
         case "red" : color = Colors.red; break;
         case "green" : color = Colors.green; break;
         case "blue" : color = Colors.blue; break;
         case "yellow" : color = Colors.yellow; ;
         case "grey" : color = Colors.grey; break;
         case "purple" : color = Colors.purple; break;
      }
```

Мы используем виджет ListView, потому что нам нужен прокручиваемый список элементов. Это требует использования конструктора builder(), который принимает число элементов в списке через itemCount (а это свойство length у entityList), а затем функцию itemBuilder, которая создает виджеты для всех элементов в списке. Для каждого из них мы получаем объект Note по его индексу, и первое, что нам нужно сделать, – это разобраться с цветом. По умолчанию мы предполагаем, что цвет не был указан и заметка будет белой. Для всех остальных вариантов устанавливаем правильный цвет из коллекции Colors (обратите внимание, что значения этих констант являются объектами, а не простыми строками или числами, поэтому я не сохраняю эти значения в базе напрямую).

После определения цвета может быть возвращен виджет Container:

```
return Container(
   padding : EdgeInsets.fromLTRB(20, 20, 20, 0),
   child : Slidable(
      delegate : SlidableDrawerDelegate(),
      actionExtentRatio : .25,
      secondaryActions : [
         IconSlideAction(
            caption : "Delete",
            color : Colors.red,
            icon : Icons.delete,
            onTap : () => _deleteNote(inContext, note)
         )
      ]
```

Мы делаем небольшие отступы слева, сверху и справа. Это удерживает заметки вдали от краев экрана, что эстетически более приятно и обеспечивает некоторое пространство между элементами списка.

Далее мы переходим к Slidable, который видели в разделе import. Этот виджет – просто тип контейнера, который поддерживает смещение. Во многих мобильных приложениях, когда есть список элементов, вы можете перемещать («свайпать») их влево/вправо, чтобы открыть кнопки для дополнительных функций. Именно такую задачу и решает виджет Slidable. Вам достаточно предоставить ему объект delegate, который управляет анимацией слайда (в нашем примере это просто экземпляр класса SlidableDrawerDelegate()). Вы также должны указать, как далеко виджет может быть перемещен, и здесь .25 означает 25 % ширины экрана. Затем вы должны указать свойства actions и/или secondaryActions. Свойство actions указывает, какие функции будут отображаться после перемещения элемента вправо, тогда как secondaryActions – это то, какие функции будут отображаться, когда элемент перемещается влево. В примере есть только одно действие для удаления, и чаще всего вы видите кнопки для удаления справа (хотя нет правила, согласно которому это должно быть именно так), так что secondaryActions – это все, что я использовал.

Каждый из объектов в списке secondaryActions, которых у вас может быть сколько угодно, – это объекты класса IconSlideAction, также предоставляемого плагином Slidable. Эти объекты позволяют вам определить, какой заголовок, значок и цвет будут у action, а также что делать с элементами. Скоро мы рассмотрим метод _deleteNote(), но сначала нужно еще немного настроить конфигурацию виджета:

```
child : Card(
   elevation : 8, color : color,
   child : ListTile(
      title : Text("${note.title}"),
      subtitle : Text("${note.content}"),
      onTap : () async {
         notesModel.entityBeingEdited =
            await NotesDBWorker.db.get(note.id);
         notesModel.setColor(notesModel.entityBeingEdited.color);
         notesModel.setStackIndex(1);
      }
   )
)
```

Внутри Container и Slidable каждая заметка представлена Card, которая, как вы помните, отображает рамку с тенью (в соответствии с рекомендациями Google Material Design). Такие виджеты выглядят как стикеры, так что я подумал, что это хороший выбор. Я немного увеличу значение elevation, чтобы сделать тени у ячеек более выраженными, а также задам выбранный ранее цвет с помощью color. Дочерний элемент Card – это виджет ListTile, который использует общий способ компоновки с полями title и subtitle, необходи-

мыми для отображения информации об отдельном элементе списка. Заметка будет вертикально развернута так, чтобы показать весь контент. `ListTile` – это очень распространенный виджет, который обычно используется в качестве дочернего элемента `ListView`, но он не обязательно должен быть его прямым потомком (он даже не должен быть его *косвенным* потомком). В следующей главе вы будете чаще встречать этот виджет и увидите другие его возможности.

Теперь, когда заметка выбрана, мы хотим, чтобы пользователь мог ее отредактировать. Это выглядит почти так же, как создание новой заметки с одним важным исключением: заметка извлекается из базы данных. На самом деле в этом нет необходимости, так как у нас уже есть эти данные в свойстве `entityList` нашей модели. Тем не менее я подумал, что в демонстрационных целях лучше сделать именно так.

Наконец, у нас есть метод _deleteNote(), который мы пропустили ранее:

```
Future _deleteNote(BuildContext inContext, Note inNote) {

   return showDialog(
      context : inContext,
      barrierDismissible : false,
      builder : (BuildContext inAlertContext) {
         return AlertDialog(
            title : Text("Delete Note"),
            content : Text(
               "Are you sure you want to delete ${inNote.title}?"
            ),
            actions : [
               FlatButton(child : Text("Cancel"),
                  onPressed: () {
                     Navigator.of(inAlertContext).pop();
                  }
               ),
               FlatButton(child : Text("Delete"),
                  onPressed : () async {
                     await NotesDBWorker.db.delete(inNote.id);
                     Navigator.of(inAlertContext).pop();
                     Scaffold.of(inContext).showSnackBar(
                        SnackBar(
                           backgroundColor : Colors.red,
                           duration : Duration(seconds : 2),
                           content : Text("Note deleted")
                        )
                     );
                     notesModel.loadData("notes", NotesDBWorker.db);
                  }
               )
```

```
            ]
          );
        }
    );
}
```

При удалении каких-либо данных хорошей практикой является уточнение намерений пользователя, и обычно это делается с помощью отображения диалога с вопросом. В нашем случае необходимо вызвать `showDialog()` и передать в него соответствующий `BuildContext`.

Затем внутри функции builder(), которая требуется для `showDialog()`, мы создаем `AlertDialog`, содержимое которого запрашивает подтверждение и показывает заголовок заметки. Потом для `actions` мы реализуем кнопки (`FlatButton`) отмены (**Cancel**), которая просто вызывает `pop()` для закрытия диалога, и удаления (**Delete**). При тапе на последнюю мы вызываем метод `delete()` класса `NotesDBWorker`, передавая ему идентификатор заметки. Затем мы закрываем диалоговое окно и используем метод Scaffold `showSnackBar()`, чтобы показать сообщение о том, что заметка была удалена. Оно будет отображаться в течение двух секунд в соответствии со значением `duration`. Наконец, необходимо вызвать метод `loadData()` в `notesModel`, чтобы список был обновлен. Напомню, что `loadData()` перезагрузит все заметки из базы данных, а затем вызовет `notifyListeners()`, что запустит перерисовку экрана.

Экран ввода: NotesEntry.dart

Теперь мы подошли к заключительной части реализации заметок, экрану ввода. Как вы видите на рис. 5-4, это довольно простой экран.

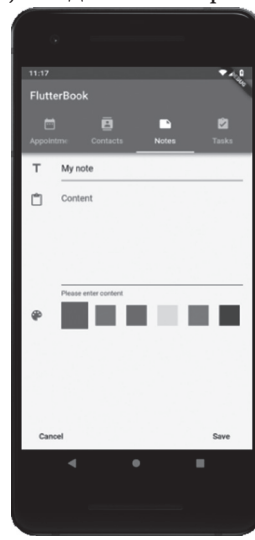


Рисунок 5-4. *Экран редактирования заметки*

На экране вы видите поле **Title** (заголовок) и поле **Content** (содержимое), которое я еще не ввел (на скриншоте вы видите сообщение об ошибке для Content, ведь я пытался сохранить заметку, не вводя ничего). Еще здесь есть кнопки отмены (**Cancel**) и сохранения (**Save**), первая возвращает пользователя на экран списка, а вторая сохраняет новую заметку (и, как я надеюсь, вы уже поняли, запускает перерисовку экрана списка, чтобы показать новую заметку).

Как всегда, начинаем с импорта:

```
import "package:flutter/material.dart";
import "package:scoped_model/scoped_model.dart";
import "NotesDBWorker.dart";
import "NotesModel.dart" show NotesModel, notesModel;

class NotesEntry extends StatelessWidget {
```

Здесь нет ничего нового, подобное вы уже видели раньше. Имейте в виду, что это все еще виджет без состояния, несмотря на то что приходится иметь дело с каким-то состоянием.

Видим что-то новенькое:

```
final TextEditingController _titleEditingController =
   TextEditingController();
final TextEditingController _contentEditingController =
   TextEditingController();
```

Виджет TextFormField – то, с помощью чего будет вводиться заголовок и содержимое. Он должен иметь связанный с ним TextEditingController, чтобы были доступны такие возможности, как значение по умолчанию и различные события, которые могут происходить, когда пользователь печатает. Так как нам понадобится доступ к этим возможностям, мы создадим TextEditingController и подключим его к полям TextFormField (если использовать просто TextFormFields без контроллеров, то мы не сможем сделать ничего подобного, без использования грязных трюков!).

Поскольку у нас есть понятие обязательных для заполнения полей, у нас будет form, а form требует key:

```
final GlobalKey<FormState> _formKey = GlobalKey<FormState>();
```

Нас не очень волнует, что это за ключ, только то, что он у нас есть, поэтому создается простой GlobalKey.

Далее у нас есть работа, которую нужно выполнить при создании класса, с этим нам поможет constructor:

```
NotesEntry() {
   _titleEditingController.addListener((){
      notesModel.entityBeingEdited.title =
         _titleEditingController.text;
   });
```

```
   _contentEditingController.addListener((){
      notesModel.entityBeingEdited.content =
         _contentEditingController.text;
   });
}
```

Видите! Нам действительно нужен был доступ к этим двум контроллерам! Хитрость в том, что каждый раз, когда изменяется значение TextFormField, к которому подключен контроллер, соответствующее значение в entityBeingEdited необходимо обновить. Для этого вызываем addListener() с функцией, которая выполнит эту задачу. Без этого все, что пользователь вводит на экран, не отразилось бы на модели.

Теперь снова просыпается метод build():

```
Widget build(BuildContext inContext) {
   _titleEditingController.text =
      notesModel.entityBeingEdited.title;
   _contentEditingController.text =
      notesModel.entityBeingEdited.content;
```

Поскольку этот экран можно эффективно использовать в двух режимах, добавляя и редактируя заметку, нам нужно убедиться, что предыдущие значения заголовка и содержимого отображаются на экране при редактировании. Когда экран находится в режиме добавления, он будет просто устанавливать значения null, поскольку это значение по умолчанию для String, которое используется для полей title и content класса Note. TextFormField все корректно обработает, делая поля пустыми; в противном случае будет показано текущее значение при редактировании заметки.

Теперь мы начинаем создавать виджет верхнего уровня, который возвращает build():

```
return ScopedModel(
   model : notesModel,
   child : ScopedModelDescendant<NotesModel>(
      builder : (BuildContext inContext, Widget inChild,
         NotesModel inModel
   ) {
   return Scaffold(
```

Пока что ничего нового: все выглядит, как и создание виджета на экране списка. Но после этого у нас идет что-то необычное:

```
bottomNavigationBar : Padding(
   padding :
      EdgeInsets.symmetric(vertical : 0, horizontal : 10),
   child : Row(
      children : [
```

```
            FlatButton(
               child : Text("Cancel"),
               onPressed : () {
                  FocusScope.of(inContext).requestFocus(FocusNode());
                  inModel.setStackIndex(0);
               }
            ),
            Spacer(),
            FlatButton(
               child : Text("Save"),
               onPressed : () { _save(inContext, notesModel); }
            )
         ]
      )
)
```

BottomNavigationBar виджета Scaffold, который позволяет размещать некоторое статическое содержимое внизу экрана. Оно не будет прокручиваться, даже если то, что находится над ним, требует прокрутки. Cancel перемещает пользователя назад к экрану списка с помощью вызова setStackIndex(). Но сначала нам нужно скрыть программную клавиатуру, если она открыта. В противном случае она будет скрывать ListView на экране списка заметок. Класс FocusScope устанавливает область видимости, в которой виджеты могут получать фокус. Flutter отслеживает через дерево виджетов элемент, который сейчас действительно находится в фокусе пользователя. Когда вы получаете FocusScope данного контекста через static-метод of(), вы можете вызвать метод requestFocus() для передачи фокуса в определенное место.

Вторая кнопка – **Save**, и это просто вызов метода _save(), к которому мы вернемся после просмотра кода виджета. К слову, о нем:

```
body : Form(
   key : _formKey,
   child : ListView(
      children : [
         ListTile(
            leading : Icon(Icons.title),
            title : TextFormField(
               decoration : InputDecoration(hintText : "Title"),
               controller : _titleEditingController,
               validator : (String inValue) {
                  if (inValue.length == 0) {
                     return "Please enter a title";
                  }
                  return null;
               }
```

```
            )
        )
```

В предыдущих двух главах вы видели, что можно иметь виджет Form и события проверки для полей ввода. Именно это нам здесь и нужно, и тот _formKey, который был создан ранее, используется и здесь. Дочерние элементы – это виджеты ListTile, и здесь вы можете увидеть еще одну вещь, которую предоставляет виджет: поле leading. Им может быть некоторый контент в левой части ListTile, например значок, показанный здесь. Виджет ListTile также поддерживает свойство trailing, чтобы делать то же самое с правой стороны, но нам это не нужно.

Поле title у ListTile – это первый TextFormField. Может показаться странным, что свойство с именем title – это не просто текстовая строка, но в этом и прелесть Flutter: это (обычно) не имеет значения! Вы можете поместить туда все, что захотите, при условии что это виджет (будет ли он хорошо выглядеть или работать так, как вы ожидаете, – это другой вопрос). У TextFormField есть свойство decoration, значением которого является объект InputDecoration. У данного объекта много свойств, включая labelText (текст, описывающий поле), enabled (для визуального включения или отключения поля), suffixIcon (значок, который появляется после редактируемой части текстового поля или после suffix или suffixText в контейнере для decoration). Еще у него есть свойство hintText. Установка этого значения приводит к отображению слова «Title» в виде слегка затемненного текста всякий раз, когда поле для ввода оказывается пустым. Другими словами, hintText выполняет ту же функцию, что и текстовая метка. Как видите, свойство контроллера ссылается на TextEditingController, созданный ранее для этого поля. Также здесь задан validator, который проверяет текущее значение, чтобы убедиться, что оно было введено. К тому же валидатор возвращает строку ошибки, и если такая ошибка есть, то отобразится подсказка красного цвета под проверяемым полем. Обратите внимание, что подсказка отобразится после того, как будет вызван соответствующий метод валидации. Наша проверка отобразится после вызова метода _save(), к которому мы скоро перейдем.

Однако до этого у нас есть еще один TextFormField для поля content:

```
ListTile(
   leading : Icon(Icons.content_paste),
   title : TextFormField(
      keyboardType : TextInputType.multiline,
      maxLines : 8,
      decoration : InputDecoration(hintText : "Content"),
      controller : _contentEditingController,
      validator : (String inValue) {
         if (inValue.length == 0) {
            return "Please enter content";
         }
```

```
            return null;
        }
    )
)
```

Это почти то же самое, что и в поле заголовка, за исключением одного: *maxLines*. Он определяет то, сколько строк текста можно ввести в поле, и меняет высоту виджета соответственно. Здесь будет достаточно места для восьми строк текста. Если вы знаете HTML, это фактически заставляет `TextFormField` работать как `<textarea>`.

Теперь перейдем к части, отвечающей за те цветовые блоки, которые пользователь может использовать для выбора цвета заметки:

```
ListTile(
   leading : Icon(Icons.color_lens),
   title : Row(
      children : [
```

Мы начнем с другого свойства `ListTile`, с `leading`, показывающего значок цветовой палитры. На этот раз `title` представляет собой `Row`, так что все блоки могут быть расположены рядом друг с другом.

Из-за повторяющегося кода я собираюсь показать только один блок. Другие блоки идентичны этому коду, за исключением ссылок на цвета.

```
GestureDetector(
   child : Container(
      decoration : ShapeDecoration(
         shape : Border.all(width : 18, color : Colors.red) +
         Border.all(width : 6,
            color : notesModel.color == "red" ?
            Colors.red : Theme.of(inContext).canvasColor
         )
      )
   ),
   onTap : () {
      notesModel.entityBeingEdited. color = "red";
      notesModel.setColor("red");
   }
),
Spacer(),
```

и так для каждого цвета...

Каждый блок содержит виджет `GestureDetector`, который дает нам элемент, реагирующий на различные сенсорные события. Здесь мы заботимся только о событиях тапа, поэтому предусмотрена функция `onTap()`. Внутри `GestureDetector` находится `Container`, а у него есть поле `decoration`, которое определяет прямо-

угольник с рамкой (Border) со всех сторон. Рамке задается граница шириной 18 пикселей, поскольку контента нет, это приводит к заполнению рамки, поэтому границы в некотором смысле «схлопываются» в сплошную рамку. Затем добавляется еще одна граница, снова используя конструктор all(), чтобы поместить рамку шириной в 6 пикселей вокруг этого поля. Если значение свойства color в модели красного цвета, то цвет границы становится красным. В противном случае она имеет тот же цвет, что и фон, который мы можем получить, запросив тему, связанную с BuildContext. CanvasColor – это фон, на котором все нарисовано, поэтому нам нужен именно этот элемент Theme. Смысл в том, чтобы контейнер становился толще благодаря внешней границе только тогда, когда она выбрана.

При нажатии на прямоугольник с цветом происходит установка color в объекте entityBeingEdited, а еще он устанавливается как атрибут color модели с помощью вызова setColor(). Этот вызов также приводит к вызову notifyListeners(), что в итоге приводит к тому, что граница теперь отображается в цвете блока – так достигается эффект большего размера.

Последний фрагмент кода, который мы рассмотрим в этой главе, – это метод _save():

```
void _save(BuildContext inContext, NotesModel inModel) async {
   if (!_formKey.currentState.validate()) { return; }
   if (inModel.entityBeingEdited.id == null) {
      await NotesDBWorker.db.create(
         notesModel.entityBeingEdited
      );
   } else {
      await NotesDBWorker.db.update(
         notesModel.entityBeingEdited
      );
   }

notesModel.loadData("notes", NotesDBWorker.db);

inModel.setStackIndex(0);
   Scaffold.of(inContext).showSnackBar(
      SnackBar(
         backgroundColor : Colors.green,
            duration : Duration(seconds : 2),
         content : Text("Note saved")
      )
   );
}
```

Очевидно, что этот метод сохраняет заметки в базе данных. Во-первых, форма проверяется, и если она не прошла валидацию, метод завершается досрочно.

Поскольку у нас нет флага для определения того, создаем мы новую заметку или изменяем существующую, то мы можем проверить ее на наличие идентификатора (`id`). Если у заметки не обнаружен идентификатор (`id == null`), значит, мы ее создаем, иначе редактируем. Итак, если мы редактируем нашу заметку, то необходимо вызвать `update()`, а если создаем, то `create()`. В любом случае, в `NotesDBWorker` передаем `entityBeingEdited`.

Далее у нас есть несколько заключительных задач. Во-первых, необходимо вызвать `loadData()`, чтобы экран списка был обновлен, а затем вернуть пользователя обратно на экран списка с помощью вызова `setStackIndex()`. Наконец, в течение двух секунд мы отображаем `SnackBar`, чтобы указать, что заметка была сохранена.

С заметками покончено!

Внимание. При работе с Flutter меня часто напрягала работа (или отсутствие) механизма hot reload (перерисовка приложения при внесении изменений). Хотя горячая перезагрузка, несомненно, – это огромное преимущество для повышения производительности разработчика, она может вызывать проблемы, если вы забыли, что при горячей перезагрузке изменения *не сохраняются* в вашем приложении. Это означает, что если ваше приложение работает в эмуляторе, и вы вносите изменение, а затем оно само перезагружается с помощью `hot reload`, то увидите это изменение в эмуляторе, как и ожидалось, но если затем закрыть приложение и перезапустить его, изменений *не будет*. Изменение будет присутствовать только до перезапуска приложения и в действительности не применяется, пока вы не выполните полную сборку (`build`) и развертывание приложения (`deploy`). Бывало, что я забывал об этом и бился головой о стол, когда что-то начинало внезапно работать, по какой-то непонятной мне причине, а потом переставало. Я призываю вас запомнить это.

Что в итоге

Ура, мы сделали это! Но FlutterBook еще не закончен! В вашем первом опыте создания настоящего приложения Flutter вы познакомились с общей архитектурой приложения, конфигурацией проекта, включая добавление плагинов, переход между частями приложения, управление состоянием, хранение данных с помощью SQLite, и множеством виджетов! Конечно, это еще не законченное приложение, но начало положено.

В следующей главе мы завершим FlutterBook, добавив код для трех других объектов: встреч, контактов и задач. В конце концов, у вас будет полноценное, пригодное для использования приложение!

ГЛАВА 6

FLUTTERBOOK. ЧАСТЬ II

В предыдущей главе мы начали писать код FlutterBook, в частности заметки. В этой главе мы закончим с этим приложением, рассмотрев задачи, встречи и контакты.

Это может показаться объемным, но это не так: если вы сравните код для четырех вкладок, то увидите, что он на 90 % идентичен. Для каждой работает одна и та же структура: файл основного кода (например, `Notes.dart`), а затем экран списка и экран ввода, каждый из которых имеет свои собственные исходные файлы. Код в каждом из них будет в основном таким же (или чрезвычайно похожим), что и у вкладки `Notes`. Все четыре экрана списка несколько отличаются, так что мы рассмотрим их более подробно, а вот экраны ввода очень похожи, за исключением нескольких моментов.

Итак, я собираюсь показать вам отличия этого кода от кода в предыдущей главе. Таким образом, мы рассмотрим только фрагменты исходных кодов. Суть в том, что если я об этом ничего не скажу, вы подумаете, что нет никаких отличий от кода заметок из предыдущей главы (кроме таких мелочей, как имена переменных, полей и т. п.).

Сделаем это: задачи

Первая вкладка, которую мы рассмотрим в этой главе, – это `Tasks` (задачи). Задачи – это просто: для их описания требуется только строка текста и, возможно, срок выполнения. Как вы видели на скриншоте предыдущей главы, список позволяет пользователю отмечать выполненные задачи. Поэтому код довольно простой, возможно, даже проще, чем код заметок.

TasksModel.dart

Во-первых, каждая вкладка имеет модель и экран. У задач есть класс `Task`, и единственным отличием между классом `Note` и `Task` являются поля в классе:

```
int id;
String description;
String dueDate;
String completed = "false";
```

Экземпляр этого класса будет храниться в базе данных, поэтому нам необходимо уникальное поле `id`. Кроме того, у нас есть описание задачи, дата (которая будет необязательной) и флаг, чтобы указать, выполнена задача или нет. Вы можете подумать, что тип поля `completed` должен быть `bool`, и в целом

я с вами согласен! Но поскольку данная сущность хранится в таблице SQLite, а SQLite не предлагает нам поддержку типа bool, то completed должен храниться в виде строки. Конечно, можно выполнять преобразования из логического типа в строку у нас в коде, но нам не нужны такие трудности, к тому же вы сами сможете сделать это, если захотите, опираясь на предыдущий опыт.

После этого идет описание модели:

```
class TasksModel extends BaseModel { }
```

Думаете, я ошибся и что-то забыл? Нет! TasksModel действительно пуст! Видите ли, на экране ввода нет полей, которые были бы связаны только с задачами. Поэтому в модели не должно быть ничего. Напомню, что BaseModel предоставляет общий код, который должен быть у всех четырех моделей, но задачам не нужно ничего, кроме этого, следовательно, это просто пустой класс (за исключением того, что уже есть в BaseModel!).

TasksDBWorker.dart

Как и заметки, вкладка **Tasks** должна иметь свой класс для работы с базой данных (DBWorker), но есть только одно существенное отличие от класса Notes (опять же, помимо базовых вещей, таких как имена переменных и методов и т. д.), и это SQL:

```
CREATE TABLE IF NOT EXISTS tasks (
   id INTEGER PRIMARY KEY, description TEXT,
   dueDate TEXT, completed TEXT
)
```

Tasks.dart

Отправная точка для экрана задачи, подобно TasksDBWorker.dart, почти идентична Notes.dart, который вы видели в предыдущей главе, структура экрана с IndexedStack и все остальное повторяется, поэтому давайте перейдем к коду, в котором есть реальные различия.

TasksList.dart

Как упоминалось ранее, каждый из четырех списков немного отличается друг от друга, хотя вы обнаружите, что большой процент кода идентичен. Основное отличие задач в том, что они могут быть завершены. Давайте рассмотрим виджет, возвращаемый функцией build(), который снова является ScopedModel:

```
body : ListView.builder(
   padding : EdgeInsets.fromLTRB(0, 10, 0, 0),
   itemCount : tasksModel.entityList.length,
   itemBuilder : (BuildContext inBuildContext, int inIndex) {
```

```
      Task task = tasksModel.entityList[inIndex];
      String sDueDate;
      if (task.dueDate != null) {
         List dateParts = task.dueDate.split(",");
         DateTime dueDate = DateTime(int.parse(dateParts[0]),
            int.parse(dateParts[1]), int.parse(dateParts[2]));
         sDueDate = DateFormat.yMMMMd(
            "en_US"
         ).format(dueDate.toLocal());
      }
```

Дата выполнения, если она есть, использует метод `split()` для разделения на три отдельные части (помните из предыдущей главы, что она хранится как «год,месяц,день»), и эти части передаются в конструктор `DateTime`, чтобы получить объект `DateTime` с указанным сроком выполнения. Затем мы используем одну из функций форматирования, которые предлагает класс `DateFormat`. Этот служебный класс из пакета `intl` предоставляет множество функций для форматирования даты и времени, а также решения других проблем интернационализации. Сложности работы с этими функциями немного выходят за рамки нашей книги. Но суть в том, что вызов функции `yMMMMD()`, а затем передача возвращаемого значения в функцию `format()` и вызов `toLocal()` у `dueDate` возвращают нам дату в удобном для отображения формате.

Затем мы можем начать создавать пользовательский интерфейс, который, как и в заметках, использует `Slidable` в качестве основы:

```
return Slidable(delegate : SlidableDrawerDelegate(),
   actionExtentRatio : .25, child : ListTile(
      leading : Checkbox(
         value : task.completed == "true" ? true : false,
      onChanged : (inValue) async {
         task.completed = inValue.toString();
         await TasksDBWorker.db.update(task);
         tasksModel.loadData("tasks", TasksDBWorker.db);
      }
   ),
```

На этот раз свойству `leading` передадим `Checkbox`, который пользователь может изменить по завершении задачи. Значение берется из ссылки на задачу. Поскольку `completed` – это строка, а не логическое значение, оно не может быть значением свойства `value` напрямую, поэтому получим логический тип, используя тернарную запись. После мы должны прикрепить обработчик события `onChanged`, когда `Checkbox` станет отмечен (или не отмечен). Это легко: возьмите логическое значение, переданное в функцию `onChanged`, и установите его в качестве значения `task.completed`, вызвав для него метод `toString()`. Затем попросите `TasksDBWorker` обновить задачу и, наконец, попросите `TasksModel` обновить список с помощью вызова метода `loadData()`, который предоставляет `BaseModel`.

У нас есть еще одна часть для Slidable:

```
title : Text(
   "${task.description}",
   style : task.completed == "true" ?
      TextStyle(color :
         Theme.of(inContext).disabledColor,
         decoration : TextDecoration.lineThrough
      ) :
      TextStyle(color :
         Theme.of(inContext).textTheme.title.color
      )
   ),
   subtitle : task.dueDate == null ? null :
      Text(sDueDate,
         style : task.completed == "true" ?
            TextStyle(color :
               Theme.of(inContext).disabledColor,
               decoration : TextDecoration.lineThrough) :
               TextStyle(color :
                  Theme.of(inContext).textTheme.title.color)
               ),
               onTap : () async {
                  if (task.completed == "true") { return; }
                     tasksModel.entityBeingEdited =
                        await TasksDBWorker.db.get(task.id);
                  if (tasksModel.entityBeingEdited.dueDate == null) {
                     tasksModel.setChosenDate(null);
                  } else {
                     tasksModel.setChosenDate(sDueDate);
                  }
                  tasksModel.setStackIndex(1);
               }
            ),
            secondaryActions : [
               IconSlideAction(
                  caption : "Delete",
                  color : Colors.red,
                  icon : Icons.delete,
                  onTap : () => _deleteTask(inContext, task)
               )
            ]
         );
      }
)
```

Все это должно показаться вам довольно знакомым, но вы должны понять, что мы не можем редактировать выполненную задачу. Чтобы добавить такое ограничение, необходима проверка. Поэтому добавим `task.completed` в обработчик события `onTap()`.

Как и в случае с заметками, здесь есть метод `deleteTask()`, который вызван через `secondActions` объекта `Slidable`.

TasksEntry.dart

Экран ввода (рис. 6.1) содержит только два поля, одно из которых необходимо обязательно заполнить:

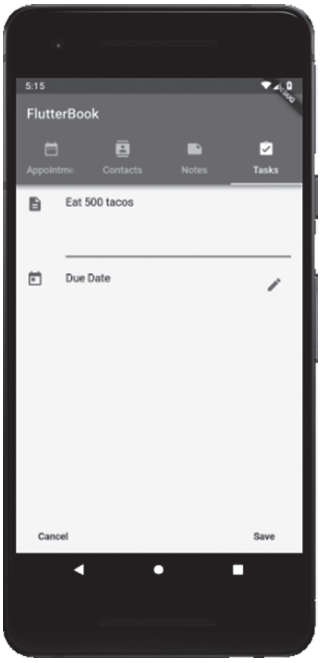


Рисунок 6-1. *Экран ввода задач*

Единственный код, на котором нам нужно сосредоточиться, – это поле даты выполнения:

```
ListTile(leading : Icon(Icons.today),
   title : Text("Due Date"), subtitle : Text(
      tasksModel.chosenDate == null ? "" : tasksModel.chosenDate
   ),
   trailing : IconButton(
      icon : Icon(Icons.edit), color : Colors.blue,
      onPressed : () async {
         String chosenDate = await utils.selectDate(
```

```
            inContext, tasksModel,
            tasksModel.entityBeingEdited.dueDate);
         if (chosenDate != null) {
            tasksModel.entityBeingEdited.dueDate = chosenDate;
         }
      }
   )
)
```

Здесь мы видим использование функции `utils.selectDate()`, которую мы кратко рассмотрели в предыдущей главе. Она возвращает строку в формате «год, месяц, день», в котором она сохраняется в базе данных. Конечно, если возвращается `null`, тогда дата не была выбрана, а значит, нам не нужно записывать значение в поле `dueDate`.

Назначим свидание: Appointments (встречи)

Далее следует вкладка **Appointments** (встречи), с ее помощью мы рассмотрим некоторые, отличные от изученных ранее возможности, включая удобный плагин для экрана со списком. Но сначала рассмотрим модель.

AppointmentsModel.dart

У нас есть класс для описания встречи, который называется `Appointment`. Он содержит следующие поля:

```
int id;
String title;
String description;
String apptDate;
String apptTime;
```

С полями идентификатора (`id`), а также заголовка (`title`) и описания (`description`) все понятно. Как и у задачи, у встречи есть дата, которая называется `apptDate`, а также здесь есть время – `apptTime`. Оба этих поля являются строками для удобства хранения в базе данных. Для преобразования типов нам необходим код, который мы скоро разберем.

После определения класса нам нужно добавить метод, который нужен для работы со временем на экране:

```
class AppointmentsModel extends BaseModel {
   String apptTime;
   void setApptTime(String inApptTime) {
      apptTime = inApptTime;
      notifyListeners();
   }
}
```

Как и с задачами, поле chosenDate и метод setChosenDate() в BaseModel будут использоваться для задания даты. Поскольку время есть только у встреч, нам нужны apptTime и setApptTime() в AppointmentsModel, код метода setApptTime() аналогичен коду setChosenDate().

AppointmentsDBWorker.dart

У встреч тоже есть свой класс для работы с базой данных (DBWorker). Он почти идентичен двум предыдущим, за исключением определения таблицы, которое выглядит следующим образом:

```
CREATE TABLE IF NOT EXISTS appointments (
   id INTEGER PRIMARY KEY, title TEXT,
   description TEXT, apptDate TEXT, apptTime TEXT
)
```

Appointments.dart

Как и в случае с задачами, определение главного экрана для встреч такое же, как и для заметок, поэтому сразу перейдем к экрану списка, в котором есть на что посмотреть.

AppointementsList.dart

Во-первых, здесь появляются новые import:

```
import
   "package:flutter_calendar_carousel/"
   "flutter_calendar_carousel.dart";
import "package:flutter_calendar_carousel/classes/event.dart";
import "package:flutter_calendar_carousel/classes/event_list.dart";
```

Calendar Carousel – это плагин, предоставляющий приложению виджет просмотра календаря (calendar). У него есть много опций, таких как различные режимы для отображения дат, обработчики касаний при нажатии на определенную дату и т. д.

Flutter не предлагает ничего подобного из коробки, отсюда и необходимость в плагине.

Итак, давайте создадим виджет и посмотрим, как он используется:

```
class AppointmentsList extends StatelessWidget {

Widget build(BuildContext inContext) {
   EventList<Event> _markedDateMap = EventList();
      for (
         int i = 0; i < appointmentsModel.entityList.length; i++
      ) {
```

```
Appointment appointment =
   appointmentsModel.entityList[i];
List dateParts = appointment.apptDate.split(",");
DateTime apptDate = DateTime(
   int.parse(dateParts[0]), int.parse(dateParts[1]),
   int.parse(dateParts[2]));
   _markedDateMap.add(apptDate, Event(date : apptDate,
      icon : Container(decoration : BoxDecoration(
         color : Colors.blue))
   ));
}
```

`Calendar Carousel` предоставляет способ отображения дат, поддерживающих назначенные на них события. Для этого у него есть свойство `markedDatesMap`, которое принимает `Map`, содержащий в качестве ключей объекты `DateTime`, а в качестве значения объекты `Event`, которые описывают событие. Создадим отображение (`map`), перебирая `appointmentsModel.entityList`, который представляет собой массив встреч, полученных из базы данных. Для каждой встречи мы приведем свойство `apptDate` в такой вид, который можно передать конструктору `DateTime` и получить его экземпляр. Потом мы создаем `Event` и добавляем его в `_markedDateMap`. Объекту `Event` можно задать дату и свойство `icon`. Для отображения даты на экране я использую виджет `Container` вместе с `BoxDecoration`, без дополнительных свойств. В результате поле применяет минимум возможного пространства и выглядит как квадрат размером всего несколько пикселей, рис. 6-2.

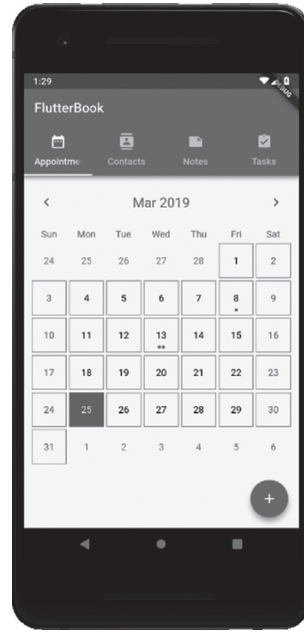


Рисунок 6-2. *Экран списка встреч с индикатором даты*

На 8-е и 13-е назначены встречи, поэтому мы видим точки. Обратите внимание, что если 13-го числа будет несколько событий, точек будет столько же. Идеально!

Теперь мы можем перейти к виджету, описанному методом `build()`:

```
return ScopedModel<AppointmentsModel>(
   model : appointmentsModel,
   child : ScopedModelDescendant<AppointmentsModel>(
      builder : (inContext, inChild, inModel) {
         return Scaffold(
            floatingActionButton : FloatingActionButton(
               child : Icon(Icons.add, color : Colors.white),
               onPressed : () async {
                  appointmentsModel.entityBeingEdited =
                     Appointment();
                  DateTime now = DateTime.now();
                  appointmentsModel.entityBeingEdited.apptDate =
                     "${now.year},${now.month},${now.day}";
                  appointmentsModel.setChosenDate(
                     DateFormat.yMMMMd("en_US").format(
                        now.toLocal()));
                  appointmentsModel.setApptTime(null);
                  appointmentsModel.setStackIndex(1);
               }
            ),
```

Подобный код должен быть вам знаком, так что не будем на нем останавливаться. Вместо этого давайте посмотрим, что будет дальше:

```
body : Column(
   children : [
      Expanded(
         child : Container(
            margin : EdgeInsets.symmetric(horizontal : 10),
            child : CalendarCarousel<Event>(
               thisMonthDayBorderColor : Colors.grey,
               daysHaveCircularBorder : false,
               markedDatesMap : _markedDateMap,
               onDayPressed : (DateTime inDate, List<Event> inEvents) {
                     _showAppointments(inDate, inContext);
               }
            )
         )
      )
   ]
)
```

Наша цель – заставить Calendar Carousel растягиваться, чтобы заполнить экран. В этом поможет виджет Expanded: он растягивает свой дочерний элемент, чтобы заполнить все доступное пространство внутри Row, Column или Flex. Обратите внимание, что родительский виджет должен иметь такую возможность. На данный момент подобная возможность присутствует только у трех представленных выше виджетов. Я выбрал Column, так как в данном случае различий нет. Вместо того чтобы использовать виджет CalendarCarousel в качестве дочернего элемента Expanded, я использую Container, чтобы можно было установить вокруг него отступ. Я подумал, что лучше не растягивать его до самого края экрана и избежать столкновения с TabBar или FAB.

Определение CalendarCarousel в нашем случае – это достаточно простая задача. Я задаю каждой дате серую рамку, а также гарантирую, что они квадратные, установив daysHaveCircularBorder в false.

Затем идет markedDatesMap, о котором мы говорили ранее, он указывает на заполненный ранее _markedDateMap. А далее мы описываем обработчик события нажатия на какую-либо дату. При нажатии я хочу показать имеющиеся события для выбранной даты в BottomSheet. Для этого используется метод _showAppointments():

```
void _showAppointments(
   DateTime inDate, BuildContext inContext) async {

   showModalBottomSheet(context : inContext,
      builder : (BuildContext inContext) {
         return ScopedModel<AppointmentsModel>(
            model : appointmentsModel,
            child : ScopedModelDescendant<AppointmentsModel>(
```

Этот код принимает дату, на которую кликает пользователь, и BuildContext вызвавшего виджета. Обратите внимание на то, что хотя функция onDayPressed принимает список событий, нам необходимо использовать данные из entityList в model, так как Event не содержит нужных нам данных. Вот почему виджет, возвращаемый функцией builder для showModalBottomSheet(), начинается со ScopedModel и ссылается на нашу AppointmentsModel.

Функция builder будет следующей:

```
builder : (BuildContext inContext, Widget inChild,
   AppointmentsModel inModel) {
      return Scaffold(
         body : Container(child : Padding(
         padding : EdgeInsets.all(10), child : GestureDetector(
```

Пока ничего нового, верно? Я снова почувствовал, что здесь просто необходим отступ (padding), поэтому body начинается с контейнера с таким отступом.

Поскольку то, что показано в BottomSheet, со скриншота из предыдущей главы, представляет собой список встреч с вертикальной прокруткой, я использую Column:

```
child : Column(
   children : [
      Text(DateFormat.yMMMMd("en_US").format(inDate.toLocal()),
         textAlign : TextAlign.center,
         style : TextStyle(color :
            Theme.of(inContext).accentColor, fontSize : 24)
         ),
         Divider(),
```

Первый дочерний элемент – это просто красиво отформатированный `Text`, центрированный на `BottomSheet` с помощью свойства `textAlign`, а его значение – выбранная дата. Обратите внимание на способ получения цвета текста: функция `Theme.of()` всегда доступна и дает вам ссылку на тему, активную в данный момент для приложения. Получив эту ссылку, у вас будет доступ к ее содержимому, например значению `accentColor`, в нашем случае он синий. Также здесь указан `fontSize`. Ниже я добавил виджет `Divider`, чтобы отделить дату от списка встреч.

Затем следует еще один `Expanded`, так что список встреч заполнит оставшееся пространство внутри макета `Column`. Затем мы создаем контейнер для каждого элемента из нашего списка. Для правильного отображения встреч нам нужно отфильтровать весь список по требуемой нам дате:

```
Expanded(

   child : ListView.builder(
      itemCount : appointmentsModel.entityList.length,
      itemBuilder : (BuildContext inBuildContext, int inIndex) {
         Appointment appointment =
            appointmentsModel.entityList[inIndex];
         if (appointment.apptDate !=
            "${inDate.year},${inDate.month},${inDate.day}") {
            return Container(height : 0);
         }
         String apptTime = "";
         if (appointment.apptTime != null) {
            List timeParts = appointment.apptTime.split(",");
            TimeOfDay at = TimeOfDay(
               hour : int.parse(timeParts[0]),
               minute : int.parse(timeParts[1]));
            apptTime = " (${at.format(inContext)})";
         }
```

Для любой встречи, которая не относится к выбранной дате, мы возвращаем `Container` с нулевой высотой. Это необходимо, потому что возврат `null` из функции `itemBuilder` приведет к исключению (`exception`).

Время приводится к такому формату, который мы можем передать свойствам из `TimeOfDay`. Метод `format()` возвращает нам отформатированное время с учетом текущего региона.

Каждая встреча должна поддерживать редактирование и удаление, для этого используем виджет Slidable:

```
return Slidable(delegate : SlidableDrawerDelegate(),
   actionExtentRatio : .25, child : Container(
      margin : EdgeInsets.only(bottom : 8),
      color : Colors.grey.shade300,
      child : ListTile(
         title : Text("${appointment.title}$apptTime"),
         subtitle : appointment.description == null ?
            null : Text("${appointment.description}"),
         onTap : () async {
            _editAppointment(inContext, appointment);
         }
      )
   ),
```

Если у встречи есть описание, оно отображается как подзаголовок. В противном случае значение равно null, и ничего не отобразится. Метод _editAppointment мы скоро рассмотрим, но сначала завершим определение виджета с помощью свойства secondActions объекта Slidable:

```
secondaryActions : [
   IconSlideAction(caption : "Delete", color : Colors.red,
      icon : Icons.delete,
      onTap : () => _deleteAppointment(inBuildContext, appointment)
   )
]
```

Этот код не отличается от кода для заметок или задач, так же как и код метода _deleteAppointment(), поэтому мы пропустим его. Однако у нас все еще есть метод _editAppointment():

```
void _editAppointment(BuildContext inContext, Appointment
   inAppointment) async {

   appointmentsModel.entityBeingEdited =
      await AppointmentsDBWorker.db.get(inAppointment.id);
   if (appointmentsModel.entityBeingEdited.apptDate == null) {
      appointmentsModel.setChosenDate(null);
   } else {
      List dateParts =
         appointmentsModel.entityBeingEdited.apptDate.split(",");
      DateTime apptDate = DateTime(
         int.parse(dateParts[0]), int.parse(dateParts[1]),
         int.parse(dateParts[2]));
         appointmentsModel.setChosenDate(
```

```
            DateFormat.yMMMMd("en_US").format(apptDate.toLocal()));
    }
    if (appointmentsModel.entityBeingEdited.apptTime == null) {
       appointmentsModel.setApptTime(null);
    } else {
       List timeParts =
          appointmentsModel.entityBeingEdited.apptTime.split(",");
       TimeOfDay apptTime = TimeOfDay(
          hour : int.parse(timeParts[0]),
          minute : int.parse(timeParts[1]));
          appointmentsModel.setApptTime(apptTime.format(inContext));
    }
    appointmentsModel.setStackIndex(1);
    Navigator.pop(inContext);
}
```

Он почти идентичен методу редактирования заметок, но здесь мы должны добавить обработку времени. Поскольку дата и время необязательны, мы должны обрабатывать их только в том случае, если они не пустые. И если они не пустые, мы должны привести их к формату, поддерживаемому `setChosenDate()` и `setApptTime()`, и передать их в соответствующие методы.

AppointmentsEntry.dart

Последняя часть раздела Appointments – это экран ввода, который показан на рис. 6-3.

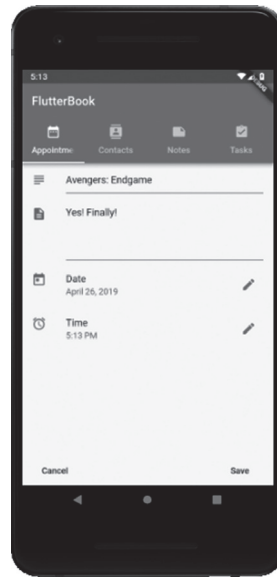


Рисунок 6-3. *Экран редактирования встречи*

Это достаточно простой экран. Он имеет только одно обязательное поле title, многострочное описание (description) и два поля для выбора даты и времени. Технически поле Date тоже обязательное, потому что встреча без даты не имеет смысла. Поэтому по умолчанию устанавливается текущая дата. Следовательно, у встречи всегда будет дата, и пользователь может изменить ее, если текущая не подходит.

Что касается кода, то здесь нет ничего нового, кроме части для получения времени встречи, которая очень похожа на часть с датой, но мы все равно посмотрим на код:

```
        ListTile(leading : Icon(Icons.alarm),
           title : Text("Time"),
           subtitle : Text(appointmentsModel.apptTime == null ?
              "" : appointmentsModel.apptTime),
           trailing : IconButton(
              icon : Icon(Icons.edit), color : Colors.blue,
              onPressed : () => _selectTime(inContext)
           )
        )
     ]
  )
)
```

Это совершенно обычное определение поля, за исключением вызова _select Time в обработчике onPressed, который выглядит следующим образом:

```
Future _selectTime(BuildContext inContext) async {

   TimeOfDay initialTime = TimeOfDay.now();
   if (appointmentsModel.entityBeingEdited.apptTime != null) {
      List timeParts =
         appointmentsModel.entityBeingEdited.apptTime.split(",");
      initialTime = TimeOfDay(hour : int.parse(timeParts[0]),
         minute : int.parse(timeParts[1])
      );
   }
   TimeOfDay picked = await showTimePicker(
      context : inContext, initialTime : initialTime);
      if (picked != null) {
         appointmentsModel.entityBeingEdited.apptTime =
            "${picked.hour},${picked.minute}";
         appointmentsModel.setApptTime(picked.format(inContext));
      }
}
```

Итак, если пользователь редактирует существующую встречу, то мы должны установить initialTime в TimePicker, который мы вызываем с помощью show

TimePicker(). Следовательно, мы должны взять его в entityBeingEdited. Затем, когда пользователь выберет время, свойство apptTime обновится в entityBeingEdited, и, наконец, следует вызов setApptTime(), необходимый для перерисовки экрана.

Как с вами связаться: контакты

Последняя вкладка, которую мы рассмотрим, – это Contacts (контакты), я оставил ее напоследок, по моему мнению, она сложнее всех.

ContactsModel.dart

Как и в случае с тремя другими вкладками, мы начнем с модели и класса Contact. Я просто покажу вам поля в классе, поскольку в общем-то он похож на предыдущие:

```
int id;
String name;
String phone;
String email;
String birthday;
```

Вообще, у контактов может быть много информации, но мы остановимся на ключевых. Это поля имени (name), телефона (phone) и адреса электронной почты (email). В этот список я добавил день рождения (birthday), чтобы иметь еще один пример работы с DatePicker.

Совет: я рекомендую доработать все приложения в этой книге в качестве учебных упражнений, и добавление контактам дополнительной информации идеально для этого подойдет.

Что касается модели:

```
class ContactsModel extends BaseModel {
   void triggerRebuild() {
      notifyListeners();
   }
}
```

Метод triggerRebuild() нам действительно необходим, поскольку notifyListeners() должен вызываться из модели. Напомню, что метод notifyListeners() вызывает перерисовку экрана. Мы будем его использовать при редактировании контакта. Например, при задании или изменении аватара.

ContactsDBWorker.dart

Код базы данных для контактов снова идентичен предыдущим, за исключением создания таблицы:

```
CREATE TABLE IF NOT EXISTS contacts (
   id INTEGER PRIMARY KEY,
   name TEXT, email TEXT, phone TEXT, birthday TEXT
)
```

Contacts.dart

Базовый макет экрана контактов не имеет ничего нового, поэтому идем дальше.

ContactsList.dart

Экран списка контактов – это просто ListView. Единственное отличие от уже готовых экранов в том, что здесь есть изображение аватара для каждого контакта.

Итак, начинаем:

```
return ScopedModel<ContactsModel>(
   model : contactsModel,
   child : ScopedModelDescendant<ContactsModel>(
      builder : (BuildContext inContext, Widget inChild,
         ContactsModel inModel) {
         return Scaffold(
            floatingActionButton : FloatingActionButton(
               child : Icon(Icons.add, color : Colors.white),
               onPressed : () async {
                  File avatarFile =
                     File(join(utils.docsDir.path, "avatar"));
                  if (avatarFile.existsSync()) {
                     avatarFile.deleteSync();
                  }
                  contactsModel.entityBeingEdited = Contact();
                  contactsModel.setChosenDate(null);
                  contactsModel.setStackIndex(1);
               }
            )
```

Все начинается со ScopedModel. Свойство model ссылается на contactsModel, а затем идет объявление ScopedModelDescendant. Далее идет функция builder, возвращающая Scaffold, который нам нужен, чтобы мы могли использовать FAB для создания нового контакта.

Теперь обработчик событий onPressed от FAB – это место, где мы начинаем видеть кое-что новое и захватывающее. Вы увидите, что при создании контакта можно добавить к нему изображение аватара. Оно будет храниться в каталоге документов приложения, а не в базе данных из-за того, что я хочу вам показать, как работать с файлами. Но при редактировании контакта, нового или существующего, может присутствовать временный файл изображения, если пользователь ранее редактировал контакт. Итак, чтобы начать с создания нового контакта, мы должны убедиться, что временного файла нет. Класс File представляет собой класс Dart из пакета io, и его конструктор принимает в качестве аргумента путь к файлу. Вы видели utils.docsDir, полученный в предыдущей главе, и его свойство path – путь к каталогу документов. Таким образом, передавая это методу join() (функция из библиотеки Path, которая знает, как объединить части в полный путь до файла с учетом конкретной платформы), мы получаем правильный путь вместе с именем аватара. Класс File предоставляет несколько методов, один из которых – existSync(). Этот метод возвращает true, если файл существует, false, если нет, и метод выполняется синхронно, что нам и нужно. В противном случае нам придется ждать его (или же ждать, пока Future не будет решено). Существует также асинхронная версия exists(). Если файл уже существует, то вызывается метод deleteSync(), чтобы избавиться от него (также доступен асинхронный метод delete()). После этого создается новый контакт, и пользователь, как обычно, переходит на экран ввода.

Далее у нас есть ListView, который содержит контакты:

```
body : ListView.builder(

   itemCount : contactsModel.entityList.length,
   itemBuilder : (BuildContext inBuildContext, int inIndex) {
      Contact contact = contactsModel.entityList[inIndex];
      File avatarFile =
         File(join(utils.docsDir.path, contact.id.toString()));
      bool avatarFileExists = avatarFile.existsSync();
```

Каждый контакт по очереди извлекается из модели, и создается ссылка на файл аватара, если он задан. Файл использует идентификатор контакта в качестве имени файла, поэтому их легко связать. На этот раз результат вызова existSync() сохраняется в avatarFileExists по причине, которую вы можете увидеть в следующем фрагменте кода:

```
return Column(children : [
   Slidable(

      delegate : SlidableDrawerDelegate(),
      actionExtentRatio : .25, child : ListTile(
         leading : CircleAvatar(
            backgroundColor : Colors.indigoAccent,
            foregroundColor : Colors.white,
```

```
            backgroundImage : avatarFileExists ?
               FileImage(avatarFile) : null,
            child : avatarFileExists ? null :
               Text(contact.name.substring(0, 1).toUpperCase())
      ),
      title : Text("${contact.name}"),
      subtitle : contact.phone == null ?
         null : Text("${contact.phone}"),
```

Каждый дочерний элемент `ListView` представляет собой макет `Column` и будет содержать два элемента: `Slidable`, который содержит сам контакт, и `Divider`, поэтому `Column` необходим. `Slidable` похож на все остальные, которые вы видели, за исключением `leading`. Здесь это `CircleAvatar` – виджет, который показывает изображение и обрезает его в круглую форму. Обычно он используется для отображения аватаров людей в списке, поэтому очень удобно его использовать. Единственная хитрость в том, что `backgroundImage` должен быть либо действительной ссылкой `FileImage`, либо `null`. Вот тут и появляется флаг `avatarFileExists`. Когда он `true`, `avatarFile`, который запоминает экземпляр `File`, оборачивается в виджет `FileImage` – виджет для отображения изображения на основе ссылки на файл в файловой системе. Если он `false`, `backgroundImage` будет `null`.

Нам также нужен этот флаг, ведь если у контакта нет изображения аватара, мы хотим отображать первую букву его имени – это типичный шаблон в контактах. Таким образом, дочерний элемент `CircleAvatar` будет `null`, если изображение есть, или текстовым виджетом, если его нет. В последнем случае метод `substring()` класса `String`, поле `contact.name`, используется для получения этой первой буквы, а метод `toUpperCase()` применяется для преобразования его в верхний регистр.

Остальную конфигурацию для `Slidable` вы уже знаете, поэтому давайте посмотрим на обработчик `onTap` и запуск редактирования контакта:

```
onTap : () async {

   File avatarFile =
      File(join(utils.docsDir.path, "avatar"));
   if (avatarFile.existsSync()) {avatarFile.deleteSync(); }
   contactsModel.entityBeingEdited =
      await ContactsDBWorker.db.get(contact.id);
   if (contactsModel.entityBeingEdited.birthday == null) {
      contactsModel.setChosenDate(null);
   } else {
      List dateParts =
         contactsModel.entityBeingEdited.birthday.split(",");
      DateTime birthday = DateTime(
         int.parse(dateParts[0]), int.parse(dateParts[1]),
         int.parse(dateParts[2]));
```

```
      contactsModel.setChosenDate(
         DateFormat.yMMMMd("en_US").format(birthday.toLocal())
      );
   }
   contactsModel.setStackIndex(1);
}
```

Этот обработчик также не слишком отличается от предыдущих, но здесь мы снова имеем дело с временным изображением аватара, поэтому оно удаляется, если уже существует. Дата `birthday` также должна быть проверена и установлена в модели для отображения на экране редактирования, затем осуществляется обычная навигация по экрану с помощью вызова `setStackIndex()`.

Для завершения настройки `Slidable` и `ListView` необходимо рассмотреть `secondaryActions`:

```
   secondaryActions : [
      IconSlideAction(caption : "Delete", color : Colors.red,
         icon : Icons.delete,
         onTap : () => _deleteContact(inContext, contact))
   ]
),
Divider()
```

Здесь вы также можете увидеть `Divider`, который завершит функцию `itemBuilder()`.

Теперь посмотрим, как удалить контакт:

```
Future _deleteContact(BuildContext inContext,
   Contact inContact) async {

   return showDialog(context : inContext,
      barrierDismissible : false,
      builder : (BuildContext inAlertContext) {
         return AlertDialog(title : Text("Delete Contact"),
            content : Text(
               "Are you sure you want to delete ${inContact.name}?"
            ),
            actions : [
               FlatButton(child : Text("Cancel"),
                  onPressed: () {
                     Navigator.of(inAlertContext).pop();
                  }
               ),
   FlatButton(child : Text("Delete"),
      onPressed : () async {
         File avatarFile = File(
            join(utils.docsDir.path, inContact.id.toString()));
```

```
            if (avatarFile.existsSync()) {
               avatarFile.deleteSync();
            }
            await ContactsDBWorker.db.delete(inContact.id);
            Navigator.of(inAlertContext).pop();
            Scaffold.of(inContext).showSnackBar(
               SnackBar(backgroundColor : Colors.red,
                  duration : Duration(seconds : 2),
                  content : Text("Contact deleted")));
            contactsModel.loadData("contacts", ContactsDBWorker.db);
      }
)
```

Мы видим типичный код функции удаления из других вкладок, но у нас остались файлы аватарок. Удаление контакта из базы данных недостаточно; мы должны также удалить его файл аватара, если он есть, поэтому мы снова получаем на него ссылку и, если он существует, вызываем `deleteSync()`, чтобы избавиться от него. После этого мы просто удаляем контакт из базы данных, показываем `SnackBar`, и все готово!

ContactsEntry.dart

У нас есть еще один фрагмент FlutterBook, и это экран редактирования контактов, который вы можете видеть на рис. 6-4.

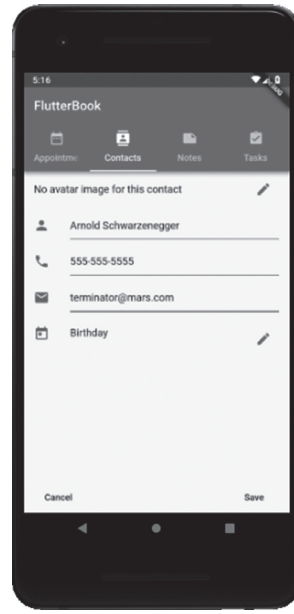


Рисунок 6-4. *Экран ввода для контактов*

Это достаточно простой экран: три виджета TextFormField, только один из которых (name) необходим, а затем поле birthday со значком редактирования для отображения DatePicker. Это достаточно просто, но я все равно хочу показать код, потому что функциональность работы с изображениями аватара реализуется в нескольких местах, и именно здесь код существенно отличается от кода экранов ввода/редактирования для остальных трех вкладок.

```
return ScopedModel(model : contactsModel,
   child : ScopedModelDescendant<ContactsModel>(
      builder : (BuildContext inContext, Widget inChild,
         ContactsModel inModel) {
         File avatarFile =
            File(join(utils.docsDir.path, "avatar"));
         if (avatarFile.existsSync() == false) {
            if (inModel.entityBeingEdited != null &&
               inModel.entityBeingEdited.id != null
            ) {
               avatarFile = File(join(utils.docsDir.path,
                  inModel.entityBeingEdited.id.toString()
               ));
            }
         }
```

Первое, с чем мы имеем дело, – это то, что экран может отображаться при создании нового контакта или при редактировании существующего. В случае создания не будет изображения аватара, но при редактировании может быть: помните, что build() будет вызываться при изменении модели, что и происходит, когда пользователь выбирает аватар. Итак, поскольку мы находимся внутри метода build(), мы должны увидеть, существует ли временное изображение аватара. Если его нет, мы проверяем entityBeingEdited. Если у него задан id, что говорит о редактировании существующего контакта, то мы пытаемся получить ссылку на его фактический файл аватара (в отличие от временного файла с именем avatar).

Позже мы сохраняем ссылку на файл. Это понадобится, когда мы начнем рендеринг полей, но сначала нам нужно решить несколько «предварительных» вопросов:

```
return Scaffold(bottomNavigationBar : Padding(
   padding :
      EdgeInsets.symmetric(vertical : 0, horizontal : 10),
   child : Row(
      children : [
         FlatButton(child : Text("Cancel"),
            onPressed : () {
               File avatarFile =
                  File(join(utils.docsDir.path, "avatar"));
```

```
                if (avatarFile.existsSync()) {
                   avatarFile.deleteSync();
                }
                FocusScope.of(inContext).requestFocus(FocusNode());
                inModel.setStackIndex(0);
             }
          ),
          Spacer(),
          FlatButton(child : Text("Save"),
             onPressed : () { _save(inContext, inModel); })
   ]
)),
```

Это типичный запуск формы ввода, но в обработчике onPressed кнопки **Cancel** мы работаем с возможным временным файлом аватара. Несмотря на то что он был удален до отображения этого экрана, нам все равно лучше удалить его сейчас, если он существует (если пользователь выбрал аватар, но затем отменил его). Как только это будет сделано, программная клавиатура будет скрыта, как обсуждалось ранее, и пользователь вернется к экрану списка. Кнопка **Save** просто вызывает _save(), как всегда, мы увидим это позже.

Но сначала давайте определим фактическую форму:

```
body : Form(key : _formKey, child : ListView(
   children : [
      ListTile(title : avatarFile.existsSync() ?
         Image.file(avatarFile) :
         Text("No avatar image for this contact"),
         trailing : IconButton(icon : Icon(Icons.edit),
            color : Colors.blue,
            onPressed : () => _selectAvatar(inContext)
         )
      )
```

Теперь вы можете видеть, где эта ссылка на avatarFile вступает в игру: заголовок ListTile будет либо изображением, либо текстовым виджетом, сообщающим, что изображение аватара не было выбрано. Когда это изображение, avatarFile передается в конструктор Image.file(), и аватар отображается. Обратите внимание, что здесь я ничего не делал с масштабированием или обрезанием. Он просто отобразит изображение любого размера (вы можете изменить это в качестве упражнения!). Конечное свойство ListTile предоставляет IconButton для пользователя, чтобы выбрать изображение аватара, и код для этого мы рассмотрим в ближайшее время, потому что в нем есть кое-что новое и интересное! Во-первых, давайте продолжим описывать форму:

```
ListTile(leading : Icon(Icons.person),
```

```
   title : TextFormField(
      decoration : InputDecoration(hintText : "Name"),
      controller : _nameEditingController,
      validator : (String inValue) {
         if (inValue.length == 0) {
            return "Please enter a name";
         }
         return null;
      }
   )
),
ListTile(leading : Icon(Icons.phone),
   title : TextFormField(
      keyboardType : TextInputType.phone,
      decoration : InputDecoration(hintText : "Phone"),
      controller : _phoneEditingController)
),
ListTile(leading : Icon(Icons.email),
   title : TextFormField(
      keyboardType : TextInputType.emailAddress,
      decoration : InputDecoration(hintText : "Email"),
      controller : _emailEditingController)
),
ListTile(leading : Icon(Icons.today),
   title : Text("Birthday"),
   subtitle : Text(contactsModel.chosenDate == null ?
      "" : contactsModel.chosenDate),
   trailing : IconButton(icon : Icon(Icons.edit),
      color : Colors.blue,
      onPressed : () async {
         String chosenDate = await utils.selectDate(
            inContext, contactsModel,
            contactsModel.entityBeingEdited.birthday
         );
         if (chosenDate != null) {
            contactsModel.entityBeingEdited.birthday = chosenDate;
         }
      }
   )
)
```

Все это вы видели раньше, за исключением свойства keyboardType, которое позволяет нам указывать клавиатуру с учетом типа вводимых данных. Как видите, ему доступны такие свойства, как phone, emailAddress, и их значения, которые, я думаю, говорят сами за себя!

Теперь мы подошли к методу _selectAvatar(), который вызывается, когда пользователь щелкает иконку IconButton рядом с виджетом изображения аватара:

```
Future _selectAvatar(BuildContext inContext) {
   return showDialog(context : inContext,
      builder : (BuildContext inDialogContext) {
         return AlertDialog(content : SingleChildScrollView(
            child : ListBody(children : [
               GestureDetector(child : Text("Take a picture"),
               onTap : () async {
                  var cameraImage = await ImagePicker.pickImage(
                     source : ImageSource.camera
                  );
                  if (cameraImage != null) {
                     cameraImage.copySync(
                        join(utils.docsDir.path, "avatar")
                     );
                     contactsModel.triggerRebuild();
                  }
                  Navigator.of(inDialogContext).pop();
               }
            )
```

Задача состоит в том, чтобы показать диалоговое окно, в котором пользователь выбирает источник изображения аватара: галерею или камеру. Итак, мы вызываем showDialog(), а затем возвращаем AlertDialog из его функции builder. Внутри AlertDialog мы начинаем с виджета SingleChildScrollView, который содержит один виджет с возможностью прокрутки. Зачем использовать это здесь? Честно говоря, нет конкретной причины, кроме как показать вам дополнительный способ реализации. В этом случае прокрутка (scroll) не используется, но что, если у вас есть еще несколько источников изображений, которые вы хотите показать пользователю? Вместо того чтобы делать диалог огромного размера, способного вместить все его содержимое, вы можете просто позволить ему прокручиваться.

Так или иначе, внутри SingleChildScrollView хранится виджет ListBody, который последовательно размещает свои дочерние элементы вдоль заданной оси и приводит их к размерам родительского элемента на другой оси. Поскольку нам нужны элементы, по которым можно щелкнуть, я решил использовать здесь виджеты GestureDetector, а не кнопки или что-то еще. Таким образом, теперь у нас есть событие onTap, применимое к элементу, который является текстовым виджетом и при нажатии запускает камеру. Класс ImagePicker предоставляется плагином image_picker, который предлагает функции для доступа к источникам изображений; местоположение, из которого вы хотите получить изображение, указывается в свойстве source, передаваемом функцией ImagePicker.

pickImage(). Далее, если значение cameraImage не равно null (оно будет равно null, если фото не было сделано), то мы используем метод copySync(), который нам доступен, так как нам возвращается экземпляр класса File, чтобы скопировать изображение и установить его на аватар.

Затем мы должны сказать модели, что она изменилась, хотя на самом деле это не так! Мы должны сделать это, потому что нам нужен вызов метода build(), чтобы изображение было показано (помните этот код?). Итак, вызывается метод contactsModel.triggerRebuild(), который просто вызывает notifyListeners(), и это заставляет изображение отображаться после перерисовки экрана. Затем мы просто вызываем метод pop() для скрытия диалогового окна, получая ссылку на BuildContext, и двигаемся дальше.

Другой элемент в диалоге предназначен для выбора изображения из галереи, и это тот же код, только с другим источником, указанным в вызове pick Image():

```
GestureDetector(child : Text("Select From Gallery"),
   onTap : () async {
      var galleryImage = await ImagePicker.pickImage(
         source : ImageSource.gallery
      );
      if (galleryImage != null) {
         galleryImage.copySync(
            join(utils.docsDir.path, "avatar")
         );
         contactsModel.triggerRebuild();
      }
      Navigator.of(inDialogContext).pop();
   }
)
```

Наконец, есть метод _save(), но для краткости я просто покажу вам парочку строк, которые отличаются от иных методов _save():

```
id = await ContactsDBWorker.db.create(
   contactsModel.entityBeingEdited
);
```

Другой код, с которым вы уже знакомы:

```
File avatarFile = File(join(utils.docsDir.path, "avatar"));
if (avatarFile.existsSync()) {
   avatarFile.renameSync(
      join(utils.docsDir.path, id.toString())
   );
}
```

Единственное, что уникально для контактов, – это, конечно, изображение аватара, и мы должны это учитывать. Если есть временный файл `avatar`, то мы используем функцию `renameSync()`, чтобы присвоить ему имя, соответствующее идентификатору контакта. Идентификатор возьмем из метода `ContactsDBWorker.db.create()`. Конечно, при обновлении существующего контакта мы уже знаем этот идентификатор, так что можем двигаться дальше.

Подведем итоги

В этой главе мы завершили наш обзор приложения FlutterBook. Вы видели, как кодировались вкладки встреч, контактов и задач, включая такие моменты, как получение изображений из галереи или камеры и выбор времени и даты. При этом у нас есть полное приложение PIM, которое вы можете использовать в соответствии с «практическим» названием этой книги!

В следующей главе мы начнем создавать второе из трех наших приложений, и в процессе вы увидите некоторые новые возможности Flutter и даже вкусите программирование с точки зрения сервера и взаимодействия с ним приложения Flutter.

Звучит как отличное обучающее развлечение, не так ли? Этого я и добиваюсь!

ГЛАВА 7

FLUTTERCHAT. ЧАСТЬ I: СЕРВЕР

В последних двух главах мы создавали приложение, похожее на остров: все его данные хранятся на устройстве, на котором они запущены. Это полезно для многих видов приложений, но для остальных вам понадобится сервер для обмена какими-либо данными (или просто для того, чтобы сделать их доступными из других мест, кроме устройства, на котором работает приложение). На самом деле это важная часть современной разработки приложений.

В этой и следующей главах мы создадим приложение, которое использует сервер. Хотя данная книга явно не о создании серверов, но в этой главе мы будем заниматься именно этим. Можете расценивать это как бонус!

Во-первых, мы поговорим о проекте, который создаем, а затем о двух технологиях, о которых вы, возможно, слышали: Node и WebSockets. Если вы с ними уже знакомы, то можете пропустить эти два раздела и перейти прямо к разделу создания приложений, но если нет, то продолжайте читать, чтобы получить краткое представление о разработке серверных приложений, – но сначала давайте поговорим о том, что мы будем строить!

Можем ли мы это построить? Да, мы можем! Но... что «это»?!

Приложение, которое мы создадим, будет названо FlutterChat, и на тот случай, если имя его не выдает, это будет чат! С FlutterChat вы сможете общаться с другими пользователями в режиме реального времени, используя сервер.

Приложение предоставит пользователям возможность создавать комнаты, где они смогут собираться и общаться друг с другом. Будет основной экран со списком всех комнат, которые знает сервер, и мы также предоставим возможность перечислить всех пользователей, которых он знает.

Пользователям необходимо зарегистрироваться на сервере, указав имя пользователя и пароль, и они смогут в любое время присоединиться к серверу с их помощью.

Кроме того, мы предоставим возможность делать комнаты приватными. К ним смогут присоединиться только приглашенные пользователи, поэтому, конечно, мы предоставим механизм для их приглашения.

Наконец, пользователь, создающий комнату, будет иметь несколько избранных «административных» привилегий: он будет единственным, кто сможет закрыть комнату, а также сможет исключать недисциплинированных пользователей из нее.

Для этого приложения мы будем использовать встроенные функции навигации Flutter, которые вы не видели во FlutterBook (помните, что в нем использовался собственный механизм навигации). Что касается интерфейса, мы будем использовать виджет Drawer, который будет позволять управлять навигацией (предоставление пользователю возможности перемещаться между основными экранами независимо от того, в какой комнате он находится), списком пользователей, а также просматривать экран About, который мы создадим просто так!

Это несложное приложение, и если вы когда-либо использовали приложение чата, то вы уже знакомы с большинством основных концепций. Но это будет отличной демонстрацией возможностей, которые вы еще не использовали в прошлом приложении, а также приличным маленьким приложением, которое вы могли бы, при желании, использовать.

И прежде чем мы перейдем к коду Flutter, давайте поговорим о некоторых моментах разработки серверных приложений, начиная с Node.

Node

Райан Даль. У этого кота есть талант, говорю тебе!

Райан – создатель фантастического программного обеспечения под названием Node (или Node.js). Райан впервые представил его на европейском JSConf в 2009 году, и это было переломным моментом, о чем свидетельствуют нескончаемые аплодисменты на его презентации.

Node – это платформа для запуска в первую очередь (хотя и не только!) кода на стороне сервера, обладающая высокой производительностью и способная с легкостью обрабатывать множество запросов. Она основана на наиболее популярном языке на планете – JavaScript. Он довольно легкий и понятный, и это предоставляет огромные возможности разработчикам, во многом благодаря своей асинхронной и управляемой событиями модели программирования. В Node почти все, что вы делаете, не блокируется, то есть код не будет задерживать обработку других потоков запросов. Плюс Node использует популярный и хорошо настроенный движок JavaScript V8 от Google для выполнения кода, который используется в браузере Chrome и обеспечивает очень высокую производительность.

Неудивительно, что так много крупных игроков и сайтов используют Node в той или иной степени. Более того, это не мелкие компании. Мы говорим о названиях, которые вы, несомненно, знаете, включая DuckDuckGo, eBay, LinkedIn, Microsoft, Walmart и Yahoo.

Node – это первоклассная среда выполнения, в которой вы можете делать такие вещи, как взаимодействие с локальной файловой системой, доступ к реляционным базам данных, вызов удаленных систем и многое другое. Раньше вам приходилось использовать «правильную» среду выполнения, такую как Java или .Net, чтобы сделать все это, поскольку JavaScript не работал на серверах. С Node

это больше не так. Для ясности, Node сам по себе не является сервером, хотя он чаще всего используется для создания серверных приложений. Это среда выполнения JavaScript, в которой работают также многие несерверные приложения, включая инструменты разработчика, с которыми вы когда-либо сталкивались, даже если вы не знали об участии в этом Node!

Скачивание, установка и запуск Node – тривиальные задачи, независимо от выбора операционной системы. Не существует сложных установок со множеством зависимостей, а также нет большого набора конфигурационных файлов, с которыми нужно разбираться перед запуском приложения Node. Это 5-минутная задача, в зависимости от скорости вашего интернет-соединения и скорости печати. Запомните только один адрес: `http://nodejs.org`. Это ваш универсальный источник для всего, связанного с Node, начиная с первой страницы загрузки, как вы можете видеть на рис. 7-1.

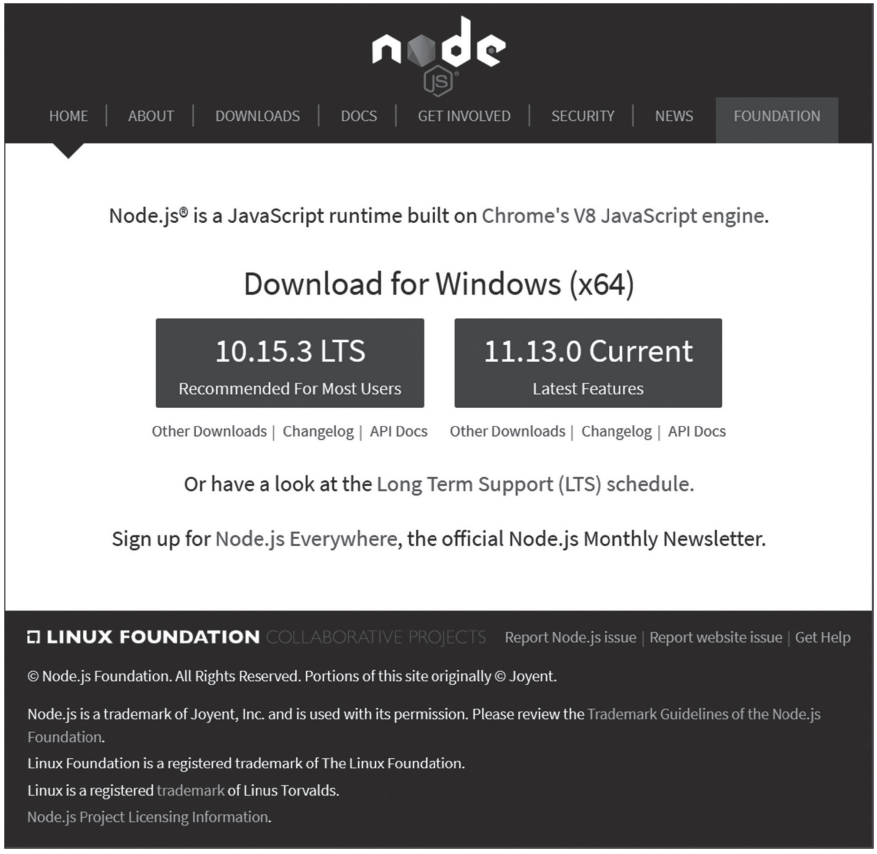


Рисунок 7-1. *У Node простой веб-сайт, который выполняет свою работу*

Я бы предложил вам установить последнюю доступную версию, но в нашем случае было бы лучше выбрать версию с долгосрочной поддержкой (LTS), по-

тому что она, как правило, более стабильна. Однако это не должно (он скрестил пальцы) иметь значение для целей нашей книги. Однако, для справки, я разработал весь код с использованием версии 10.15.3, поэтому если у вас возникнут какие-либо проблемы, я бы предложил выбрать эту версию, вы можете загрузить ее по ссылке Other Downloads, а затем по ссылке Previous Releases (оттуда вы сможете скачать любую понравившуюся вам предыдущую версию).

Например, Node для Windows предоставляет совершенно обычный и простой установщик, который проведет вас по необходимым (и чрезвычайно простым) этапам. В MacOS X типичный мастер установки сделает то же самое.

После завершения установки вы будете готовы использовать Node. Установщик должен был добавить каталог Node в вашу системную переменную path. Итак, в качестве первого простого теста перейдите в командную строку, введите `node` и нажмите **Enter**. Вас приветствует приглашение >. Теперь Node принимает ваши команды в интерактивном режиме. Для подтверждения введите следующее:

```
console.log("test");
```

Нажмите **Enter**, и вы увидите нечто похожее на рис. 7-2 (исключая различия между платформами).

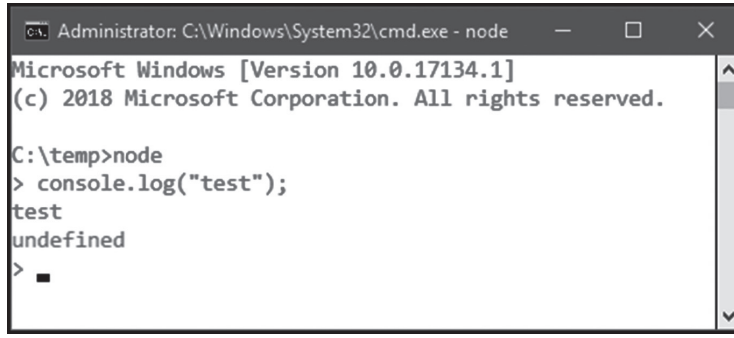


Рисунок 7-2. *Скажи привет моему маленькому другу Node*

Взаимодействие с Node в режиме командной строки хорошее, но ограниченное. Вам следует запустить сохраненный файл JavaScript с помощью Node. Как и всегда, это довольно легко. Просто создайте текстовый файл с именем `test.js` (это может быть что угодно) и введите в него приведенный ниже код (и сохраните его):

```
var a = 5;
var b = 3;
var c = a   * b;
console.log(a + " * " + b + " = " + c);
```

Чтобы запустить этот файл, вам нужно находиться в том же каталоге и ввести `node test.js`. Затем нажмите **Enter**, и вы увидите то же самое, что и на рис. 7-3.

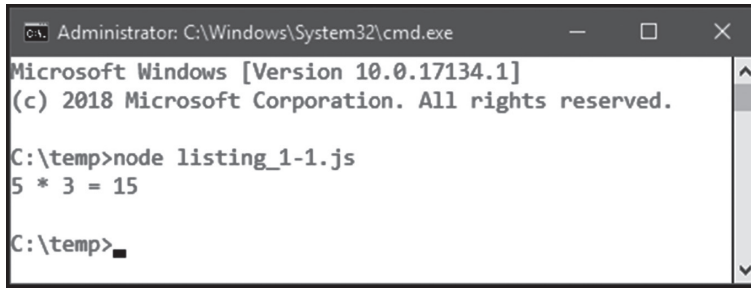


Рисунок 7-3. *Простой пример Node*

Очевидно, что этот фрагмент очень прост, зато он демонстрирует, что Node может прекрасно выполнять старый добрый JavaScript. Вы можете немного поэкспериментировать и увидите, что Node запустит любой базовый JavaScript, который вы захотите. Эта возможность, наряду с первоклассной средой выполнения и доступом к средствам операционной системы, позволяет создавать сложные инструменты, такие как React Native (точнее, его инструменты командной строки).

В этом разделе нет исчерпывающей информации о Node. В Node намного больше интересного, и если вы в этом новичок, я рекомендую вам ознакомиться с сайтом `nodejs.org`. Однако для целей книги базовых знаний будет достаточно.

Примечание. Когда я начал писать эту главу, я рассмотрел другие варианты реализации серверных приложений. Я думал о создании сервера RESTful с использованием Express поверх Node, где Express – это библиотека, которую вы можете добавить, что делает создание серверов RESTful очень простым. Но для современных требований чата это не подойдет. Затем я подумал об использовании Firebase – системы баз данных Google реального времени. Дело в том, что в интернете есть множество учебных пособий по написанию приложений Flutter, которые подключаются к Firebase, и есть даже одно или два, которые создают приложение для чата. Итак, я решил пойти другим путем и добавить такой контент, который, возможно, отсутствовал в мире обучения Flutter-разработчиков.

Я думаю, что мой подход делает вещи проще и, безусловно, более самодостаточными. Я просто хотел как-то обосновать свой выбор, который, надеюсь, придется вам по душе.

Сохранение линий связи открытыми: socket.io

Теперь, когда вы знаете что-то о Node, давайте поговорим о следующем компоненте, который нам понадобится: WebSocket и socket.io. Но сначала немного истории!

Примечание. Обзор будет в основном ориентирован на веб-технологии, но будьте уверены, что все, о чем я говорю, применимо к разработке мобильных приложений (не важно, на базе Flutter или нет). Не

беспокойтесь, если вы не знаете JavaScript, – у вас не будет никаких проблем с пониманием кода, потому что я сделал его очень простым, и, честно говоря, он все равно немного похож на Dart, с несколькими синтаксическими различиями. Конечно, после этого вы не станете опытным разработчиком JavaScript, но даже если вы с ним незнакомы, у вас не возникнет проблем.

Всемирная паутина, она же веб, изначально была задумана как место, где клиент (и, соответственно, мобильные приложения, использующие веб для связи с другими устройствами) должен был запрашивать информацию с сервера, но это исключает множество интересных возможностей или, по крайней мере, усложняет их.

Например, если у вас есть сервер, который предоставляет цены на акции для отображения на информационной панели, клиент должен постоянно запрашивать обновление данных. Это типичный подход к клиент-серверному взаимодействию. Недостаток в первую очередь заключается в том, что он требует постоянно совершать новые запросы от клиента к серверу, а также приведет к тому, что актуальность цены будет зависеть от интервала между запросами, который вы обычно не хотите делать слишком маленьким из-за боязни перегрузки сервера. Цены неактуальны, а это очень неудобно, если вы инвестор.

Через некоторое время появился AJAX. AJAX расшифровывается как Asynchronous Javascript And XML (асинхронный JavaScript и XML). Этот метод позволяет веб-страницам отправлять запросы на сервер, не обновляя всю страницу целиком, как изначально работали веб-сайты. Это был переломный момент! Теперь страница может запрашивать данные, например цены на акции, и обновлять только часть страницы, а не всю. Поверьте мне, я работал до и после появления AJAX, это было колоссально!

Наибольшее значение в концепции AJAX играет именно *асинхронная* часть аббревиатуры, а не JavaScript/XML.

С появлением методов AJAX следующим эволюционным шагом было исследование способов *двунаправленной* связи, чтобы сервер мог предоставлять клиенту новые данные без специального запроса со стороны последнего. Для этого были разработаны некоторые хитрые приемы, один из них – длительный запрос (long-polling), иногда называемый Comet. Длительный запрос – это такой способ общения между клиентом и сервером, при котором команда о завершении текущего запроса не является обязательной. Таким образом данное соединение будет оставаться открытым, и сервер сможет беспрепятственно передавать данные клиенту в любое время. Это называется «hanging-GET» («подвисший GET») или «pending-POST» («ожидающий POST») в зависимости от метода HTTP, используемого для создания соединения.

По многим причинам это может быть сложно реализовать, но главная сложность заключается в том, что для поддержания соединения на сервере запускается отдельный поток (thread). Учитывая, что это HTTP-соединение, накладные расходы несущественны, однако вскоре ваш сервер может «упасть на колени», не справившись с количеством одновременно обслуживаемых соединений.

В последние годы был создан протокол WebSocket для обеспечения такого рода постоянных соединений без проблем с длительными запросами или другими подходами, именно это нам и нужно для чата!

WebSocket – это стандарт Internet Engineering Task Force (IETF), который обеспечивает двунаправленную связь между клиентом и сервером. В этом помогает специальный запрос-рукопожатие (handshake), когда устанавливается обычное соединение HTTP. Для этого клиент отправляет запрос, который выглядит примерно так:

```
GET ws://websocket.apress.com/ HTTP/1.1
Origin: http://apress.com
Connection: Upgrade
Host: websocket.apress.com
Upgrade: websocket
```

Вы заметили значение заголовка `Upgrade` (обновление)? Это волшебный кусочек. Если сервер, поддерживающий WebSocket, увидит это, то он предоставит следующий ответ:

```
HTTP/1.1 101 WebSocket
Date: Mon, 21 Dec 2017
Connection: Upgrade
Upgrade: WebSocket
```

Сервер «соглашается на upgrade» на языке WebSocket. Как только рукопожатие завершается, HTTP-запрос прерывается, но основное TCP/IP-соединение, на котором он находился, остается. Это постоянное соединение, с которым клиент и сервер могут общаться в режиме реального времени, без необходимости каждый раз его восстанавливать.

WebSocket также поддерживается в JavaScript API, который можно использовать для установки соединений, а также отправки и получения сообщений (сообщения – это то, что мы называем данными, которые передаются через соединение WebSocket в любом направлении). Это полезно знать для реализации сервера на Node, но недостаточно для использования WebSockets во FlutterChat, которое реализуется на Dart.

К счастью, существует как готовая библиотека для Node, так и ее реализация на Dart, которая абстрагирует работу с WebSockets, предоставляя готовые механизмы. Эта библиотека называется socket.io, и именно ее мы и будем использовать в нашем проекте. Проще говоря, использование socket.io, помимо импорта библиотеки, требует вызовов чуть большего количества функций: одной для соединения двух устройств (зачастую клиента и сервера, но ни одно из них на самом деле не обязано быть сервером), второй для отправки сообщения с одного устройства на другое (включая отправку сообщения на все подключенные устройства) и третьей для приема сообщений от других устройств.

Предположим, что клиентское приложение (которое считается веб-приложением на основе JavaScript, использующим библиотеку socket.io) хранит

свои настройки на сервере. Затем, если пользователь хочет очистить их, он может отправить (emit) сообщение clearPreferences на сервер вместе с объектом, который содержит идентификатор пользователя. Для этого ему понадобится экземпляр socket.io, который уже предположительно создан и на который ссылается переменная io. Клиент будет использовать метод emit() для отправки сообщения, например так:

```
io.emit("clearPreferences", { "userlD" : "user123" });
```

Чтобы это сделать, сервер должен ожидать нужное сообщение от клиента. Вы должны зарегистрировать функцию callback с экземпляром socket.io для каждого прослушиваемого сообщения, и тут в игру вступает метод on():

```
io.on("clearPreferences", function(inData) {
   database.execute(
      'delete from user_preferences where userID=${inData.userID}'
   );
});
```

После этого при каждом получении сообщения clearPreferences выполняется функция callback и запрос к базе данных, чтобы удалить все, что хочет пользователь (не зацикливайтесь на вопросах использования базы данных, это просто пример).

Теперь предположим, что вы переносите это веб-приложение на Flutter. Со стороны Dart понятия те же, но синтаксис немного отличается. Он вместо emit() для отправки сообщения использует метод с точным названием sendMessage():

```
io.sendMessage("clearPreferences", { "userlD" : "user123" });
```

Как видите, помимо другого названия метода, все то же самое. Регистрация функции callback для сообщения почти такая же, но Dart использует метод subscribe() вместо on():

```
io.subscribe("preferencesCleared", () {
   // Сделай что-нибудь... или нет - выбор за тобой!
});
```

Если конкретнее, вы можете подписаться на сообщения как с клиента, так и на сервере, ведь сервер тоже может отправлять сообщения клиенту. Я упоминаю это потому, что мы не будем писать сервер на Dart и Flutter, но концепция применима независимо от того, что грань между клиентом и сервером при работе с WebSockets и socket.io туманна. Ничего, кроме логики, не делает одно устройство клиентом, а другое – сервером. В этом сила данного механизма!

Как видите, будь то версия JavaScript или Dart, клиентская или серверная части уравнения (socket.io API) невероятно проста и чрезвычайно эффективна. Данная библиотека предлагает более продвинутые возможности, такие как пространства имен и комнаты, которые позволяют вам разделять сообщения на логические группы, и еще многое другое. Однако это все, что вам нужно

знать для FlutterChat. Есть один маленький нюанс, связанный с установлением соединения, но я объясню это в контексте серверного кода FlutterChat, который мы рассмотрим прямо сейчас!

Код сервера FlutterChat

Чтобы начать работать с кодом сервера, мы должны создать приложение Node. Это очень просто: создайте пустой каталог, а затем выполните в нем команду (через консоль или командную строку):

```
npm init
```

«Что, черт возьми, такое NPM?! Ты не рассказывал об этом!» Я чувствую ваше недовольство. Расслабьтесь, это легко объяснить!

Node Package Manager (NPM) – это менеджер пакетов Node и инструмент, который поставляется с Node, и он... подождите, подождите... *управляет пакетами*! Пакеты – это просто дополнительные библиотеки и модули, загруженные из центрального репозитория, которые можно добавить в приложение на Node, о котором знает NPM.

Тем не менее NPM также выполняет другие функции, одна из которых – инициализация проекта.

Результатом выполнения предыдущей команды станет интерактивный процесс, который задаст вам несколько простых вопросов о вашем приложении, большинство из которых, откровенно говоря, не имеют для нас значения, поэтому вы можете либо принять значения по умолчанию, либо ввести почти все, что угодно. Конечный результат – вот что имеет значение, им будут несколько файлов, созданных в каталоге, самый важный из них – `package.json`. Он описывает ваше приложение для NPM и в конечном счете для Node. Еще он выполняет функцию, аналогичную функции файла `pubspec.yaml` в приложении Flutter, позволяя указывать такие вещи, как зависимости. И именно это нам и нужно сделать!

Здесь у нас есть выбор: мы можем отредактировать файл `package.json` или добавить его. Затем вам нужно добавить зависимость socket.io следующим образом:

```
"dependencies": {
   "socket.io": "2.2.0"
}
```

После этого мы можем выполнить еще одну команду:

```
npm install
```

Это заставит NPM прочитать файл `package.json`, просмотреть необходимые зависимости и установить их для нас из центрального репозитория. Кроме того, мы можем пропустить редактирование файла вручную, просто выполнив эту команду:

```
npm install socket.io -save
```

Это заставит NPM загрузить socket.io, «установить» его в наш проект (создать каталог `node_modules` и поместить туда код socket.io) и автоматически добавить зависимость в `package.json`.

Любой подход приведет к одному и тому же результату, но все зависит от способа, который вы предпочитаете. Однако следует помнить об одном различии: второй подход приведет к тому, что ваш проект получит последнюю версию запрошенного модуля.

Если вы захотите указать версию, то начните с редактирования `package.json` (есть способы указать версию из командной строки, но это немного сложнее).

Примечание. Если вы загрузили исходный код книги, что вам следовало бы сделать, перейдите в командную строку каталога `flutter_chat_server` и выполните `npm install`, прежде чем сделать что-либо еще. Затем запустите сервер, выполнив команду `npm start`. Из-за свойства `main` в `package.json` npm узнает, что `server.js` является главной точкой входа в приложение, и запустит Node, передав этот файл в качестве аргумента. Кроме того, вы можете сделать это вручную, выполнив `node server.js`.

Два Bits of State и Object заходят в Bar...

Если мы максимально упростим серверный код, то данные не будут сохраняться. Любые данные или состояния будут существовать в памяти только во время работы сервера. Так что если сервер перезапустится, то все будет потеряно. Но мы можем рассматривать это как фичу, а не как багу – это означает, что сервер, возможно, более безопасен (поймите меня правильно, это приложение никоим образом не обеспечит такую же безопасность, как системы ФБР/ЦРУ/АНБ, – это далеко не так).

С учетом вышесказанного мы начнем с создания файла `server.js`, в котором будет размещен весь код нашего сервера.

Вот первый фрагмент его кода:

```
const users = { };
```

Это отображение (map) пользователей. Оно будет основано на имени пользователя и значении объекта, называемого *объектом дескриптора пользователя* (user descriptor object) и имеющего следующий вид:

```
{userName : "", password : ""}
```

Довольно просто, да?

У нас есть еще одно отображение для комнат:

```
const rooms = { };
```

Структура *объектов дескриптора комнаты* имеет название комнаты и следующую форму:

```
{ roomName : "", description : "", maxPeople : 99,
   private : true|false, creator : "",
   users : [
      <username> : { userName : "" }, ...
   ]
}
```

Каждая комната может иметь описание (description) темы разговора, еще мы сможем задать максимальное количество пользователей, разрешенное в комнате, с помощью свойства `maxPeople`. Свойство `private` делает комнату закрытой (`true`) или нет (`false`), а `creator` – это имя пользователя, который ее создал. Отображение users основано на имени пользователя и представляет собой список пользователей, находящихся в данный момент в комнате.

Еще нам необходимо создать объект socket.io. Это делается одной строкой:

```
const io = require("socket.io")(
   require("http").createServer(
      function() {}
   ).listen(80)
);
```

Здесь я сделал код более понятным, по сравнению с длинной строкой из комплекта загрузки.

Суть в том, что мы создаем HTTP-сервер для Node, импортировав модуль `http`, что и делает код выше. Вместо того чтобы сохранять ссылку на объект, который мне больше не нужен, я вызываю для него метод `createServer()`, передавая в него пустую функцию. Как правило, без socket.io было бы гораздо сложнее провернуть такое, поэтому я рад, что есть такая возможность! Поскольку socket.io возьмет на себя всю ответственность, то для выполнения контракта вызова `createServer()` достаточно пустой функции. Значение, возвращаемое вызовом `createServer()`, запускается с помощью `listen()` на 80-м порту для входящих запросов, как это обычно делает маленький хороший HTTP-сервер!

Однако, поскольку мы используем для проекта библиотеку socket.io, нам предстоит еще один шаг: взять возвращаемое значение из вызова `listen()`, которое является полностью активным HTTP-сервером, и передать его в конструктор socket.io. Это позволяет socket.io взять на себя управление сервером и реализовать на нем самую древнюю и темную черную магию, чтобы сделать его подходящим сервером WebSocket.

Конечно, на текущем этапе сервер не будет ничего отвечать на запросы и подключения, но все еще впереди!

Поймай меня, если сможешь: сообщения

Все начинается с сообщения серверу socket.io. Логика при перехвате сообщения – это то, что может иметь следующую реализацию:

```
io.on("connection", io => {
   console.log("Connection established with a client");
   // Больше сведений (продолжение в следующей главе!)
});
```

Внутри функции, передаваемой в качестве второго аргумента в метод `io.on()`, у нас есть вызов `console.log()`, так что мы увидим сообщение в консоли при подключении клиента.

Нам предстоит написать еще много кода, обрабатывающего все наши сообщения из мобильного приложения, этим мы и займемся далее. Теперь давайте рассмотрим первый фрагмент этих «сведений»: проверку пользователей.

Заходим в парадную дверь: проверка пользователей

Для всех обработчиков (handlers) сообщений, обсуждаемых здесь и далее, я покажу вам диаграмму, которая детализирует данные, поступающие в обработчик (inData), а также данные, выходящие из него через callback или широковещательное сообщение (или оба). У некоторых обработчиков, в зависимости от происходящего в них, есть несколько вариантов выходных сообщений, которые будут рассмотрены в качестве альтернативного пути. Ссылайтесь на эти диаграммы при просмотре кода, чтобы получить целостное представление о потоках данных внутрь и наружу. Начнем с того, что на рис. 7-4 показана схема для первого обсуждаемого обработчика.

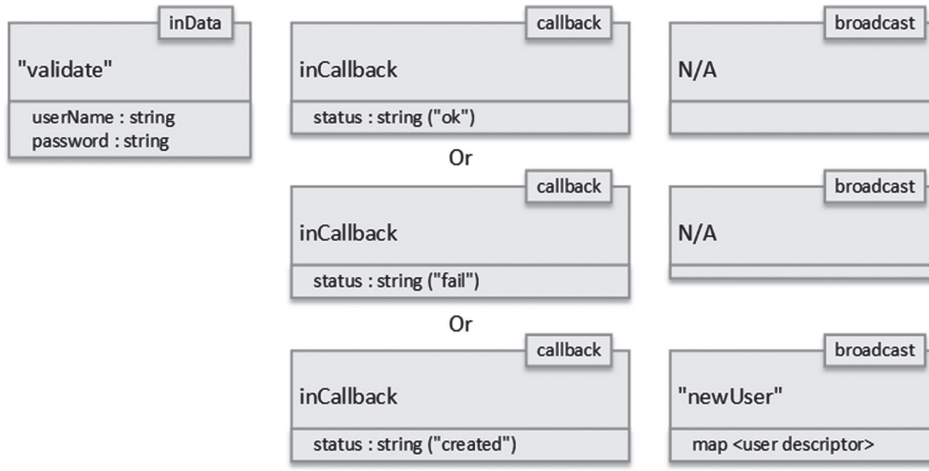


Рисунок 7-4. *Детали сообщения validate для проверки пользователя*

Первое, что происходит, когда пользователь запускает приложение Flutter Chat на своем мобильном устройстве, – это то, что ему предлагается ввести имя пользователя и пароль (если это в первый раз, в дальнейшем же нужные данные будут подставляться автоматически). Пользователи вводят свои учетные данные, и затем приложение отправляет сообщение «`validate`» (подтвердить).

Сервер должен ответить на него, чтобы подтвердить, действителен ли пользователь:

```
io.on("validate", (inData, inCallback) => {

   const user = users[inData.userName];
   if (user) {
      if (user.password === inData.password) {
         inCallback({ status : "ok" });
      } else {
         inCallback({ status : "fail" });
      }
   } else {
      users[inData.userName] = inData;
      io.broadcast.emit("newUser", users);
      inCallback({ status : "created" });
   }
});
```

Как и в случае сообщения о соединении, мы вызываем `io.on()`, чтобы зарегистрировать обработчик для этого сообщения, в котором сначала пытаемся найти пользователя в нашем списке. Данные, передаваемые в обработчик, поступают в переменную `inData`, и конкретно для этого сообщения мы ожидаем, что `client` отправит какой-либо объект в формате { `userName: «», password : «»` }. Если пользователь найден, то нам просто нужно подтвердить, что пароль совпадает.

Если это так, вызывается callback, который был передан через `inCallback`. Может показаться странным, что сервер вызывает функцию, существующую на *клиенте*, но в этом и заключается прелесть абстракции, которую нам предоставляет библиотека socket.io! Так как это сообщение специфично для конкретного пользователя, то нет необходимости отправлять обратное сообщение с помощью метода emit. Именно поэтому подход с callback-функциями идеален. Вместо него мы могли бы отправить клиенту другое сообщение, на которое клиент ответил бы новым запросом, что имитировало бы механизм обратного вызова (callback), но потребовало больше кода и действий. Когда же требуется реализовать подход типа «запрос–ответ», то использование клиентских callback-функций гораздо более удобно.

Если пароль совпадает, мы отправляем обратно объект { `status : «ok»` }. В противном случае мы отправляем обратно { `status : «fail»` }. Этот объект со свойством status (статус), который вы видите, является общим для всех обработчиков сообщений, хоть он и полностью определяется самим приложением. Я мог бы здесь возвращать простые строки, но мне нравится, что у всех моих вызовов одинаковая базовая структура как на входе, так и на выходе, поэтому я остановился на этой парадигме. Но помните socket.io все равно – вы можете отправлять и получать все, что угодно (при условии что это упорядоченная (marshaled) информация).

Как вы увидите в следующей главе, эти объекты могут преобразовываться в отображения (класс Map) языка Dart, именно это нам и нужно, так как позволяет легко передавать произвольные данные туда и обратно.

Итак, если пользователь не найден, значит, он новый (или что сервер перезапустился, или что пользователь очистил данные для приложения на своем устройстве). Во всех случаях мы добавляем пользователя в список подключенных. Затем мы делаем две вещи: вызываем `io.broadcast.emit()` и callback-функцию. С вызовом callback вы уже знакомы, но что происходит с `io.broadcast.emit()`?

Смысл в том, чтобы все подключенные клиенты знали, что на сервере появился новый пользователь. Помните, что приложение должно уметь показывать список пользователей на сервере. Данное широковещательное сообщение содержит обновленный список пользователей (как видите, это второй аргумент `io.broadcast.emit()`), который получит наше приложение. Как вы увидите в следующей главе, это приведет к обновлению списка пользователей в ScopedModel и, как следствие, перерисовке экрана с пользователями внутри мобильного приложения.

Так что да, вы можете передавать сообщения и вызывать callback-функцию из одного и того же обработчика (также вы можете отправлять столько сообщений, сколько хотите, и технически можете вызывать callback-функцию несколько раз, но зачем?).

Благодаря socket.io у нас получилось все достаточно просто. На самом деле я думаю, что данный обработчик был сложнее остальных! Давайте посмотрим на следующий!

Создание комнаты

Первая функция, которую сервер должен поддерживать после проверки пользователей, – это создание комнат (см. рис. 7-5).

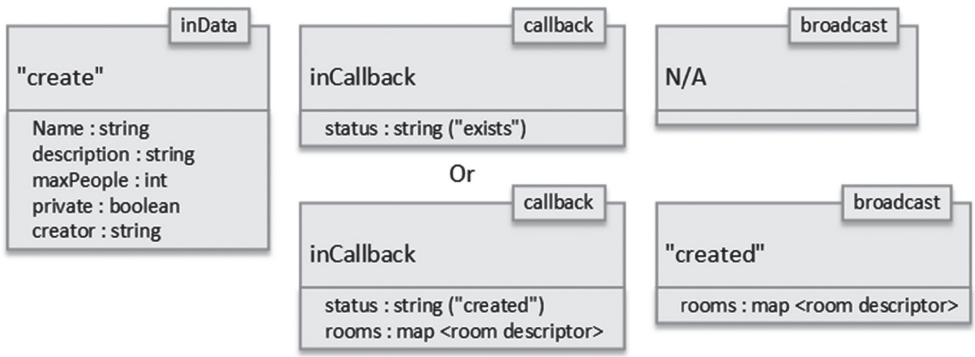


Рисунок 7-5. *Структуры данных и шаги обработчика сообщения create*

Код выглядит следующим образом:

```
io.on("create", (inData, inCallback) => {
   if (rooms[inData.roomName]) {
```

```
      inCallback({ status : "exists" });
   } else {
      inData.users = { };
      rooms[inData.roomName] = inData;
      io.broadcast.emit("created", rooms);
      inCallback({ status : "created", rooms : rooms });
   }
});
```

Быстрая проверка в коллекции `rooms` сообщает нам о существовании комнаты с указанным названием, и в случае успеха объект со статусом «`exists`» (существует) отправляется обратно, чтобы приложение могло уведомить об этом пользователя. В противном случае пустой список пользователей добавляется ко входящему объекту `inData`, который затем прикрепляется к коллекции комнат с указанием `roomName` в качестве ключа.

Затем, чтобы предупредить всех пользователей о существовании новой комнаты, отправляется сообщение «`created`» (создана). Полный список комнат рассылается всем пользователям (я признаю, это не самый эффективный механизм, но если у вас небольшое количество комнат, то это довольно простая реализация – опять же, я не утверждаю, что это готовый к работе код, который можно использовать для поддержки тысяч пользователей!).

Наконец, вызывается callback-функция, чтобы сообщить пользователю, создающему комнату, о том, что работа выполнена. Она также обновит список комнат. Это важно, потому что при широковещательной отправке сообщения оно не будет отправляться на тот сокет, который вызвал эту отправку. Другими словами, пользователь, отправивший сообщение о создании комнаты, сам не получит это уведомление. Поэтому вызов callback-функции клиента здесь необходим, чтобы обойти данное ограничение.

Покажите мне все комнаты: просмотр списка комнат

Теперь, когда у нас есть способ создавать комнаты, было бы неплохо иметь возможность отображать их список, не так ли? Я думаю, вы согласитесь! Для этого у нас будет сообщение «`listRooms`» (см. рис. 7-6).

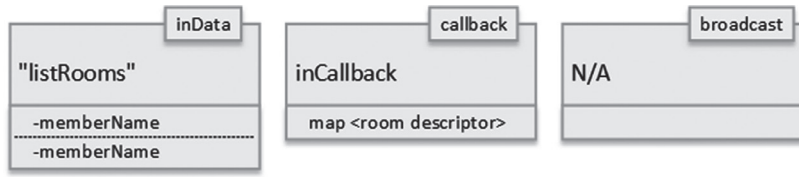


Рисунок 7-6. *Структуры данных и шаги обработчика сообщения listRooms*

Я надеюсь, что вы готовы к печати большого куска кода, потому что этот пример просто огромен. Готовы? Вы действительно готовы к тому, насколько огромным он будет? Хорошо, вот он:

```
io.on("listRooms", (inData, inCallback) => {
   inCallback(rooms);
});
```

Да, это все! Все, что нам нужно сделать, – это вернуть коллекцию комнат в callback-функцию клиента. Это сообщение «`listRooms`» необходимо только в одном случае: когда пользователь впервые заходит на основной экран. Помните, что обработчик сообщений «`create`» транслирует полный список комнат всем клиентам при каждом создании комнаты (и, как вы увидите позже, при каждом закрытии комнаты). Таким образом у клиентов будет обновленный список комнат каждый раз после подключения, но его не будет сразу после входа в систему. В этом случае отправляется «`listRooms`», однако, как вы увидите в следующей главе, он также отправляется каждый раз, когда пользователь заходит на основной экран, что немного излишне, но упрощает код.

Это все, что должен сделать обработчик сообщения «`listRooms`». Ах да, и последнее замечание: `inData` здесь не нужен, но функция-обработчик всегда будет передавать данные, нужны вам они или нет. Так что `inData` есть в списке аргументов анонимной функции только для удовлетворения требований API.

Не забывайте о людях: список пользователей

По аналогии с получением списка комнат мы должны иметь возможность получить и список пользователей.

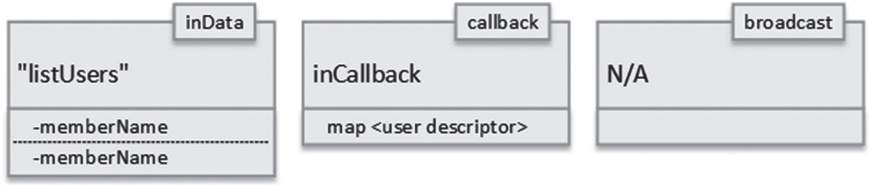


Рисунок 7-7. *Подробности обмена сообщениями для обработчика listUsers*

Модель ввода/вывода такая же, как в списке комнат:

```
io.on("listUsers", (inData, inCallback) => {
   inCallback(users);
});
```

И так же, как со списком комнат, у клиентов будет список пользователей на сервере, а получать они его будут каждый раз, когда регистрируется новый пользователь. Логика у списков пользователей и комнат будет различна. И это различие заключается в том, что для комнат есть сценарии, при которых список необходимо очищать, а для пользователей нет такого сценария. Это происходит из-за того, что список пользователей загружается каждый раз при переходе на соответствующий экран.

Стук в дверь: вход в комнату

Теперь, когда мы можем создавать комнаты и отображать их список, нам необходима возможность войти в заданную комнату, вот тут-то и вступает в действие обработчик сообщения «join» (см. рис. 7-8).

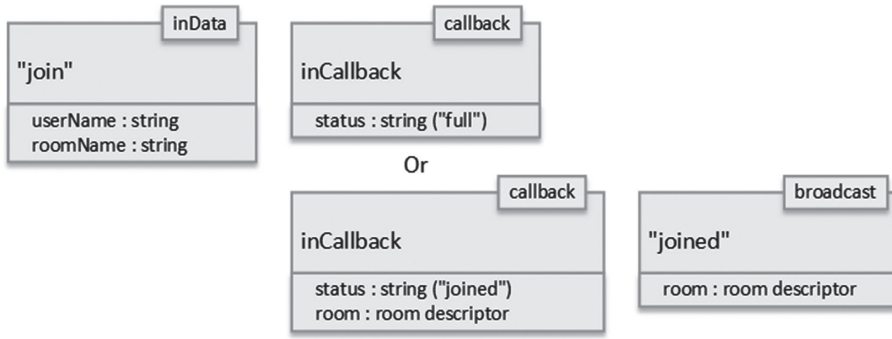


Рисунок 7-8. *Подробности обмена сообщениями для обработчика сообщений «join»*

Этот требует немного логики, как вы можете видеть:

```
io.on("join", (inData, inCallback) => {
   const room = rooms[inData.roomName];
   if (Object.keys(room.users).length >= rooms.maxPeople) {
      inCallback({ status : "full" });
   } else {
      room.users[inData.userName] = users[inData.userName];
      io.broadcast.emit("joined", room);
      inCallback({status : "joined", room : room });
   }
});
```

Сначала мы получаем ссылку на объект дескриптора комнаты на основе запрошенного имени `roomName`. Затем выполняется проверка, чтобы убедиться, что в комнате есть место (помните переменную `maxPeople`?), если места нет, то возвращается объект со статусом «`full`». В этом случае приложение сообщит пользователю о том, что он не может войти.

Если комната не заполнена, то пользователь добавится в список подключенных к этой комнате. Таким образом комната знает, кто в ней находится.

Наконец, сообщение «`joined`» (присоединен) передается всем клиентам, и, как и при создании комнаты, вызывается callback-функция, чтобы предоставить отправителю ту же информацию, которая является описанием комнаты. Затем клиентское приложение переместит пользователя на экран комнаты и заполнит список пользователей в ней, подробности вы увидите в следующей главе. Для пользователей, которых нет в комнате, это сообщение будет игнорироваться, поскольку оно к ним не относится.

Не надо кричать: отправка сообщения в комнату

Возможность создавать, просматривать список и присоединяться к комнате была бы бесполезной, если бы мы не могли публиковать пользовательские сообщения, поэтому давайте позаботимся об этом позже через скучно названное сообщение «post», как на рис. 7-9.

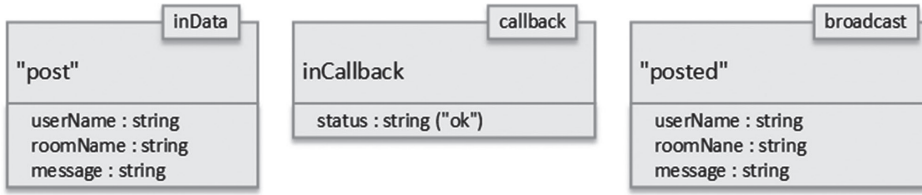


Рисунок 7-9. *Подробности обмена данными для обработчика сообщения post*

Обработчик этого события неожиданно прост:

```
io.on("post", (inData, inCallback) => {
   io.broadcast.emit("posted", inData);
   inCallback({ status : "ok" });
});
```

Все просто: нет необходимости сохранять сообщения на сервере, ведь достаточно его просто передать другим клиентам вместе с требуемыми данными. Как и в случае с сообщением «join», все пользователи, которых нет в комнате, будут игнорировать это сообщение, потому что оно не имеет к ним отношения. Наконец, хотя в этом и нет необходимости, callback-функция вызывается с простым статусом «ok», просто для обеспечения согласованности всех наших обработчиков.

Псс! Эй! Эй, ты! Иди сюда: приглашение пользователя в комнату

Когда вы находитесь в комнате, то можете приглашать других пользователей присоединиться к вам. Для этого пользователь отправляет сообщение «invite».

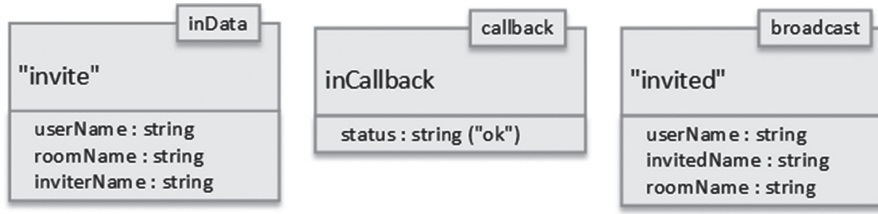


Рисунок 7-10. *Подробности обмена сообщениями для обработчика invite*

Код очень похож на код обработчика для отправки сообщения:

```
io.on("invite", (inData, inCallback) => {
   io.broadcast.emit("invited", inData);
```

```
   inCallback({ status : "ok" });
});
```

К сожалению, возможность передать сообщение конкретному пользователю отсутствует в нашей реализации, поэтому оно передается всем пользователям, а реагирует на него только тот, чье имя указано в `inData` (вместе с комнатой, в которую они приглашены, и тем, кто пригласил). Как и в случае с обработчиком «`post`», callback-функция для «`invite`» вызывается просто для согласованности.

С меня хватит: выход из комнаты

Пользователь может покинуть комнату в любое время – это ведь не тюрьма! Рисунок 7-11 доказывает, что это так.

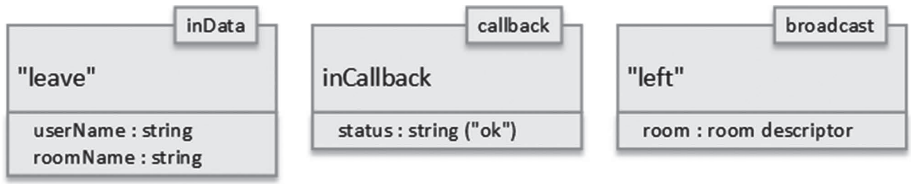


Рисунок 7-11. *Детали сообщения для обработчика leave*

Сообщение о выходе реализовано следующим образом:

```
io.on("leave", (inData, inCallback) => {
   const room = rooms[inData.roomName];
   delete room.users[inData.userName];
   io.broadcast.emit("left", room);
   inCallback({status : "ok" });
});
```

Выход из комнаты означает, что пользователь должен быть удален из коллекции пользователей в конкретной комнате, поэтому сначала мы получаем ссылку на объект `room`, а потом пользователь удаляется из коллекции `users`. После этого код должен отправить сообщение «`left`», чтобы предоставить всем клиентам обновленный список пользователей в комнате, а затем вызвать callback-функцию. Так делают, чтобы отправивший событие клиент мог завершить свой обработчик ухода из комнаты.

Вы не должны идти домой, но и здесь вы остаться не можете: закрытие комнаты

Наконец, мы подошли к первой из двух «административных» функций, которые могут применяться только человеком, создавшим комнату. Первая – закрытие комнаты, рис. 7-12.

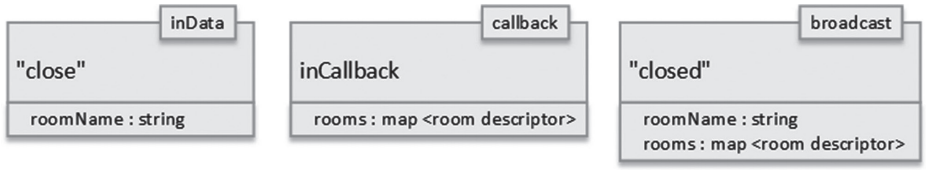


Рисунок 7-12. *Детали сообщения для обработчика close*

Код, участвующий в этом процессе, короткий и приятный:

```
io.on("close", (inData, inCallback) => {
   delete rooms[inData.roomName];
   io.broadcast.emit("closed",
      { roomName : inData.roomName, rooms : rooms }
   );
   inCallback(rooms);
});
```

Логика закрытия комнаты сложнее, чем просто ее удаление. Мы должны удалить комнату из списка и оповестить пользователей о том, что она закрыта. Это уведомление, в свою очередь, вызовет callback-функцию, которая выполнится на клиенте. Пользователи, оказавшиеся в закрытой комнате, будут оттуда выгружены и получат сообщение о закрытии комнаты.

Кое-кто ведет себя глупо: исключение пользователя из комнаты

Наконец, у нас есть еще одно «административное» сообщение – на этот раз для удаления пользователей из комнаты, как показано на рис. 7-13.

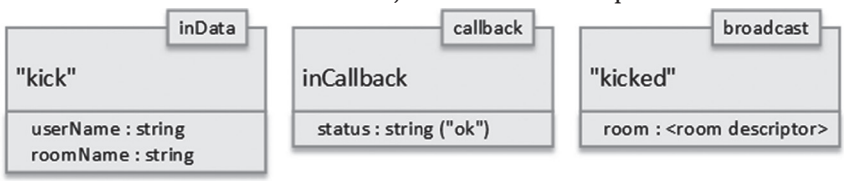


Рисунок 7-13. *Детали обработчика сообщения kick*

Как видите, в этой функции задействовано немного больше кода:

```
io.on("kick", (inData, inCallback) => {
   const room = rooms[inData.roomName];
   const users = room.users;
   delete users[inData.userName];
   io.broadcast.emit("kicked", room);
   inCallback({ status : "ok" });
});
```

Это требует извлечения объекта дескриптора комнаты из коллекции rooms, затем коллекции users внутри нее, а потом удаления из нее пользователя.

После этого сообщение «kicked» передается всем пользователям вместе с обновленной информацией о комнате. Callback-функция вызывается, хотя этого и не требуется.

И с этим окончательным обработчиком сообщений у нас есть полный сервер, который реализует всю функциональность, необходимую для работы Flutter Chat!

Итого

В этой главе мы построили серверную часть приложения FlutterChat. Здесь вы познакомились с Node.js и socket.io, а также увидели сообщения, необходимые для работы приложения. Теперь у нас есть сервер, готовый общаться с клиентами.

В следующей главе мы рассмотрим клиентскую часть и само приложение на основе Flutter. Вы увидите, как оно подключается к только что созданному серверу, и сделаете FlutterChat полноценным и функционирующим.

ГЛАВА 8

FLUTTERCHAT. ЧАСТЬ II: КЛИЕНТ

В предыдущей главе мы создали серверную часть FlutterChat, предоставляющую API на основе WebSocket и socket.io для клиентского приложения.

Наконец-то пришло время сделать клиентскую часть. Теперь вспомните все, чему уже научились, мы начинаем делать FlutterChat для смартфонов!

Model.dart

Хотя это может показаться странным, вместо того чтобы начинать с обычного файла main.dart, мы начнем с `Model.dart`, который содержит код единственной scoped-модели, используемой в приложении (как вы уже видели при создании FlutterBook, библиотека `scoped_model` указана в зависимостях проекта через файл `pubspec.yaml`). Файл `Model.dart` содержит класс `FlutterChatModel`, который наследуется от `Model` и включает следующие свойства:

- `BuildContext rootBuildContext` – это ссылка на `BuildContext` корневого (root) виджета приложения. Вскоре вы поймете, зачем это свойство может вам понадобиться. Также обратите внимание, что оно не является состоянием и может использоваться в нескольких местах. Для данного свойства нет необходимости реализовывать свой setter-метод, так как нет необходимости вызывать `notifyListeners()` при установке нового значения;
- `Directory docsDir` – каталог документов приложения. Смотрите комментарий о `rootBuildContext` и причинах его появления в модели, поскольку они также относятся и к этому свойству – без специального setter-метода;
- `String greeting = «»` – это текст приветствия, который будет отображаться на главном экране;
- `String userName = «»` – это, очевидно, имя пользователя!
- `static final String DEFAULT_ROOM_NAME = «Not currently in a room»` – это текст, который будет отображаться в `AppDrawer`, когда пользователя нет в комнате;
- `String currentRoomName = DEFAULT_ROOM_NAME` – это название комнаты, в которой находится пользователь, или же строка, которая указывает, что он еще туда не вошел;
- `List currentRoomUserList = [ ]` – список пользователей текущей комнаты;

- `bool currentRoomEnabled = false` – флаг, который отображает, находится ли пользователь в данной комнате. И если он равен `true`, то мы отображаем текущую комнату в списке комнат;
- `List currentRoomMessages = [ ]` – список сообщений в текущей комнате;
- `List roomList = [ ]` – список комнат на сервере;
- `List userList = [ ]` – список пользователей на сервере;
- `bool creatorFunctionsEnabled = false` – это флаг, который включает функции администратора чата;
- `Map roomInvites = { }` – список приглашений, полученных пользователем.

Примечание. Дабы сократить текст, я не стал печатать импорты и исходные коды. Если здесь появятся какие-либо новые и интересные импорты, то я упомяну их, но в остальном это только те модули, с которыми вы уже знакомы.

Это типичный класс модели, мы сталкивались с подобными во FlutterBook, но здесь есть различные методы и свойства, например метод для приветствия:

```
void setGreeting(final String inGreeting) {
   greeting = inGreeting;
   notifyListeners();
}
```

В конце вызывается `notifyListeners()`, чтобы любой код, подписанный на изменение данного поля, был оповещен.

Чтобы сэкономить немного места, мы пропустим рассмотрение `setUserName()`, `setCurrentRoom()`, `setCreatorFunctionsEnabled()` и `setCurrentRoomEnabled()`, поскольку они очень похожи на `setGreeting()`, различие только в том, что они ссылаются на разные свойства.

Вместо этого давайте перейдем к `addMessage()`, который немного отличается и будет вызываться, когда сервер сообщает клиенту о новом сообщении, опубликованном в комнате:

```
void addMessage(final String inUserName,
   final String inMessage) {
   currentRoomMessages.add({ "userName" : inUserName,
      "message" : inMessage });
   notifyListeners();
}
```

Здесь, вместо того чтобы просто присвоить значение, нам необходимо вызвать метод `currentRoomMessages.add()`. Метод `setRoomList()` работает немного по-другому:

```
void setRoomList(final Map inRoomList) {
```

```
    List rooms = [ ];
    for (String roomName in inRoomList.keys) {
        Map room = inRoomList[roomName];
        rooms.add(room);
    }
    roomList = rooms;
    notifyListeners();
}
```

Мы снова обновляем список, поэтому используем метод `add()`, но на этот раз передаваемый параметр имеет тип `Map`. Далее нам нужно перебрать все ключи в `Map`, а затем для каждого вытащить значение и добавить его в список наших комнат.

После этого идут методы `setUserList()` и `setCurrentRoomUserList()`, они такие же, как `setRoomList()`, так что мы их пропустим. Далее – метод `addRoomInvite()`:

```
void addRoomInvite(final String inRoomName) {
    roomInvites[inRoomName] = true;
}
```

Приглашение в комнату приводит к тому, что на экране некоторое время отображается `SnackBar`. После того как он исчезнет, мы узнаем, может ли пользователь войти в данную комнату, поэтому в коллекции `roomInvites` указывается имя комнаты и значение `bool`. Если это значение `true`, то у пользователя есть приглашение в комнату, и он может войти. Затем нам понадобится возможность отозвать приглашение при закрытии комнаты; в противном случае, если кто-то создаст комнату с таким же именем, у пользователя появится некорректная возможность зайти в нее, поэтому сделаем метод `removeRoomInvite()`:

```
void removeRoomInvite(final String inRoomName) {
    roomInvites.remove(inRoomName);
}
```

Когда пользователь выходит из комнаты, появляется возможность очистки, например очистка списка сообщений для комнаты, для этого у нас есть метод `clearCurrentRoomMessages()`:

```
void clearCurrentRoomMessages() {
    currentRoomMessages = [ ];
}
```

Наконец, создается экземпляр модели `FlutterChatModel`:

```
FlutterChatModel model = FlutterChatModel();
```

Это будет единственный экземпляр данного класса в приложении, поэтому считаем, что с этой моделью мы разобрались!

Connector.dart

Далее мы рассмотрим файл Connector.dart. Его цель – создать единый модуль, взаимодействующий с сервером и используемый остальными частями приложения. Он избавит нас от дублирования кода и необходимости импортировать зависимости в несколько мест (например, socket.io). Для данного файла нам понадобятся два новых импорта:

```
import "package:flutter_socket_io/flutter_socket_io.dart";
import "package:flutter_socket_io/socket_io_manager.dart";
```

Они необходимы для использования socket.io. Здесь нас интересуют только два класса: SocketIO из библиотеки flutter_socket_io.dart и SocketIOManager из библиотеки socket_io_manager.dart. Но я немного забегаю вперед!

Этот код начинается достаточно просто:

```
String serverURL = "http://192.168.9.42";
```

Когда вы запустите приложение, вам нужно будет задать IP-адрес, на котором работает ваш сервер. Чтобы проверить ваши навыки, у меня есть предложение: попробуйте добавить поле для IP-адреса в диалог входа в систему, который мы скоро рассмотрим. Таким образом IP-адрес, который мы захардкодили, можно будет изменить, что повысит удобство использования приложения. Затем мы находим экземпляр класса SocketIO:

```
SocketIO _io;
```

Ну, технически это объявление, а не экземпляр! Скоро мы его построим. Но сначала поговорим о двух служебных функциях. Каждый раз, когда мы делаем запрос к серверу, приложение будет отображать на экране надпись «Please wait...» («пожалуйста, подождите...»). Она заблокирует любые пользовательские действия. Чаще всего операция будет настолько быстрой, что пользователь увидит на экране только вспышку, и это нормально. Если операция занимает больше времени, то появляется наша надпись. Для этого мы просто используем dialog:

```
void showPleaseWait() {
   showDialog(context : model.rootBuildContext,
      barrierDismissible : false,
      builder : (BuildContext inDialogContext) {
         return Dialog(
            child : Container(width : 150, height : 150,
            alignment : AlignmentDirectional.center,
            decoration :
               BoxDecoration(color : Colors.blue[200])
```

Вызывается уже знакомая функция showDialog(), и мы видим, где начинает работать свойство модели rootBuildContext. Проблема в том, что этот индика-

тор должен перекрывать *весь* экран, *все* дерево виджетов, а не только его подмножество. Так что стоит всегда задавать контекст для корневого виджета. Но, как правило, в коде это доступно не отовсюду. Когда мы доберемся до файла `main.dart`, то вы увидите, что мы получаем ссылку на `buildContext` корневого виджета и сохраняем ее в модели, чтобы она была доступна везде, где может понадобиться.

Очень важно установить свойство `barrierDismissable` в значении `false`, иначе пользователь сможет закрыть наш диалог «Please wait...». Затем мы просто отображаем обычный диалог. Его содержание сводится к некоторому тексту с объяснениями того, что происходит, и вращающемуся `CircularProgressIndicator`:

```
child : Column(
   crossAxisAlignment : CrossAxisAlignment.center,
   mainAxisAlignment : MainAxisAlignment.center,
   children : [
      Center(child : SizedBox(height : 50, width : 50,
         child : CircularProgressIndicator(
            value : null, strokeWidth : 10)
      )),
      Container(margin : EdgeInsets.only(top : 20),
         child : Center(child :
            Text("Please wait, contacting server...",
            style : new TextStyle(color : Colors.white)
         ))
      )
   ]
)
```

Размещение `CircularProgressIndicator` внутри `SizedBox` с определенной шириной и высотой позволяет нам контролировать размер индикатора. Может показаться странным установка свойства `value` в значение `null` и то, что мы его никогда не обновляем, но при этом индикатор отображает анимацию для текущей операции. Проще говоря: он всегда показывает вращающуюся анимацию! Если бы операция была конечной, то вы могли бы постепенно обновлять свойство `value`, чтобы дать реальное представление пользователю об общем прогрессе, но здесь другая ситуация. Обратите внимание, я установил значение `strokeWidth`, чтобы сделать индикатор толще обычного (на мой взгляд, так выглядит лучше).

Еще нам нужен способ скрыть это диалоговое окно, когда мы дождемся ответа от сервера, для этого служит функция `hidePleaseWait()`:

```
void hidePleaseWait() {
   Navigator.of(model.rootBuildContext).pop();
}
```

Это стандартный способ скрыть диалог, отличие в том, что он снова должен использовать rootBuildContext.

Далее следует функция connectToServer(), которая вызывается из диалогового окна входа в систему, как только пользователь вводит свои учетные данные:

```
void connectToServer(final BuildContext inMainBuildContext,
   final Function inCallback) {
      _io = SocketIOManager().createSocketIO(
         serverURL, "/", query : "",
         socketStatusCallback : (inData) {
            if (inData == "connect") {
               _io.subscribe("newUser", newUser);
               _io.subscribe("created", created);
               _io.subscribe("closed", closed);
               _io.subscribe("joined", joined);
               _io.subscribe("left", left);
               _io.subscribe("kicked", kicked);
               _io.subscribe("invited", invited);
               _io.subscribe("posted", posted);
               inCallback();
            }
         }
      );
      _io.init();
      _io.connect();
   }
```

Вот где используется объект SocketIO с помощью вызова SocketIOManager.createSocketIO() и передачи ему serverURL. Этот метод также имеет два дополнительных параметра. Первый параметр – это path, необходимый для уточнения пути (например, serverURL/chat). По умолчанию туда передается пустая строка или «/»; второй параметр, query, может использоваться для добавления дополнительных параметров к запросу, частый пример – это token для авторизации.

Свойство socketStatusCallback принимает функцию, запускаемую при изменении статуса WebSocket. Может приходить несколько статусов, но нам важен только один – connect. Он означает, что соединение с сервером установлено. Только теперь мы можем определить обработчики для различных сообщений, которые сервер отправляет клиентам. Они называются subscriptions (подписки), поэтому вызывается метод subscribe(), в который передаются сообщение и обработчик, вызываемый при получении данного сообщения.

Наконец, методы init() и connect() должны быть вызваны для инициализации и подключения к серверу, и, если не возникло ошибок, мы сможем использовать callback, определенный ранее. После этого наш клиент может от-

правлять сообщения на сервер и обрабатывать сообщения, принимаемые от него.

Связанные с сервером функции сообщений

Сначала мы рассмотрим функции, которые отправляют сообщения на сервер, и первая такая функция validate() вызывается из диалога входа в систему и предназначена для проверки того, что ввел пользователь:

```
void validate(final String inUserName, final String
   inPassword, final Function inCallback) {
      showPleaseWait();
      _io.sendMessage("validate",
         "{ \"userName\" : \"$inUserName\", "
         "\"password\" : \"$inPassword\" }",
         (inData) {
            Map<String, dynamic> response = jsonDecode(inData);
            hidePleaseWait();
            inCallback(response["status"]);
         }
      );
   }
```

validate() принимает имя пользователя, его пароль и ссылку на callback-функцию. Сначала вызывается функция showPleaseWait(), которую мы рассматривали ранее, чтобы заблокировать экран. Затем метод sendMessage() объекта _io, который отправляет серверу сообщение «validate» и строку JSON, содержащую имя пользователя и пароль. Функция callback использует jsonDecode(), чтобы получить объект Map, в котором будут содержаться данные от сервера. Затем вызывается hidePleaseWait() для снятия блокировки экрана, а следом – inCallback, передавая ему свойство status из отображения.

В некоторых местах все отображение (map) будет передано в callback, как, например, в случае с функцией listRooms():

```
void listRooms(final Function inCallback) {
   showPleaseWait();
   _io.sendMessage("listRooms", "{}", (inData) {
      Map<String, dynamic> response = jsonDecode(inData);
      hidePleaseWait();
      inCallback(response);
   });
}
```

Обе эти функции и их базовая структура повторяются несколько раз в других местах. Основная идея отображения диалога «Please wait..» – это ожидание отправки нашего сообщения и ответа от сервера. Затем, когда мы его дожда-

лись, у нас выполнится callback-функция, которая скроет диалог и выполнит действие в зависимости от ответа. Разница лишь в том, какое сообщение отправлено, какие аргументы оно принимает и что сервер отправляет обратно. Таким образом, я собираюсь описать все эти функции здесь:

- `create()` – вызывается на главном экране для создания комнаты. Она передает имя комнаты, ее дескриптор, максимальное количество людей независимо от того, приватная комната или нет, имя администратора и callback, в который передаются `status` и свойство `rooms` – полный и обновленный список комнат на сервере;
- `join()` – вызывается, когда пользователь кликает на нужную комнату, чтобы присоединиться (или войти) в нее. Эта функция передает имя пользователя, имя комнаты и callback, которому передается свойство `status` из ответа и дескриптора комнаты;
- `left()` – вызывается, когда пользователь покидает комнату, в которой он находится. Передает имя пользователя, имя комнаты и callback (которому ничего не передается);
- `listUsers()` – вызывается из `AppDrawer` для получения обновленного списка пользователей на сервере. Эта функция принимает только ссылку на callback, который получает список пользователей;
- `invite()` – вызывается, когда пользователь приглашает кого-то в комнату. Передается имя приглашаемого пользователя, имя комнаты, в которую он приглашен, имя приглашающего пользователя и callback (которому ничего не передается);
- `post()` – вызывается для отправки сообщения в текущую комнату. Передает имя пользователя, имя комнаты, публикуемое сообщение и callback, который получает свойство status;
- `close()` – вызывается администратором, чтобы закрыть комнату. Передает имя комнаты и callback, которому ничего не передается;
- `kick()` – вызывается администратором для удаления пользователя из комнаты. Передает имя пользователя, имя комнаты и callback, которому ничего не передается.

Связанные с клиентом обработчики сообщений

Следующая группа функций, которые мы рассмотрим, касается сообщений, поступающих с сервера.

Названия этих функций имитируют имя сообщения, отправляемого сервером. Первая из них – `newUser()`:

```
void newUser(inData) {
   Map<String, dynamic> payload = jsonDecode(inData);
```

```
   model.setUserList(payload);
}
```

Она вызывается при создании нового пользователя. Сервер отправляет полный список пользователей, и эта функция просто задает его в модели.

Функция `create()` используется при создании новой комнаты. Она выглядит так же, как `newUser()`, за исключением вызова `model.setRoomList()`, поэтому давайте пропустим ее и перейдем к `closed()`:

```
void closed(inData) {
   Map<String, dynamic> payload = jsonDecode(inData);
   model.setRoomList(payload);
   if (payload["roomName"] == model.currentRoomName) {
      model.removeRoomInvite(payload["roomName"]);
      model.setCurrentRoomUserList({});
      model.setCurrentRoomName(
         FlutterChatModel.DEFAULT_ROOM_NAME);
      model.setCurrentRoomEnabled(false);
      model.setGreeting(
         "The room you were in was closed by its creator.");
      Navigator.of(model.rootBuildContext
      ).pushNamedAndRemoveUntil("/", ModalRoute.withName("/"));
   }
}
```

Здесь у нас чуть больше работы! Во-первых, в модели задается обновленный список комнат. Далее, если в закрытой комнате в данный момент находится пользователь, то нам нужно выполнить очистку. Если в неё есть приглашение, оно должно быть удалено, чтобы избежать приглашения пользователя в комнату с тем же именем, но созданную позже. Потом очищается список пользователей в текущей комнате, а ее название заменяется на имя по умолчанию, которое также отображается в заголовке `AppDrawer`. Еще мы отключаем ссылку `Current Room` внутри `AppDrawer`, а приветствие на домашнем экране сообщает о том, что комната была закрыта. Так пользователь узнает, что произошло. Наконец, нам нужно перейти к домашнему экрану с помощью метода `pushNamedAndRemoveUntil()`. Эта функция (одна из нескольких, которые можно использовать для навигации) гарантирует, что мы всегда возвращаемся к домашнему экрану. Таким образом, наш `Navigator` всегда находится в известном состоянии после этого перемещения.

Когда пользователь присоединяется к комнате, то сервер отправляет сообщение «joined», поэтому у нас есть соответствующая функция-обработчик `joined()`:

```
void joined(inData) {
   Map<String, dynamic> payload = jsonDecode(inData);
   if (model.currentRoomName == payload["roomName"]) {
      model.setCurrentRoomUserList(payload["users"]);
```

```
    }
}
```

Мы заботимся об этом сообщении только тогда, когда пользователь находится в комнате, и если это так, то обновленный список пользователей сохраняется в модели. Похожим образом работает функция `left()`, которая выполняется, когда пользователь покидает комнату.

Если администратор *исключает* пользователя из комнаты, то включается обработчик сообщения `kicked`! Эта функция аналогична `closed()`, потому что с точки зрения пользователя комната в некотором смысле закрылась – по крайней мере, для него! Единственным отличием является текст на главном экране, который говорит, что его исключили. Давайте сэкономим время и не будем это рассматривать. Лучше взгляните, что происходит, когда пользователя приглашают в комнату:

```
void invited(inData) async {
   Map<String, dynamic> payload = jsonDecode(inData);
   String roomName = payload["roomName"];
   String inviterName = payload["inviterName"];
   model.addRoomInvite(roomName);
   Scaffold.of(model.rootBuildContext).showSnackBar(
      SnackBar(backgroundColor : Colors.amber,
         duration : Duration(seconds : 60),
         content : Text("You've been invited to the room "
            "'$roomName' by user '$inviterName'.\n\n"
            "You can enter the room from the lobby."
         ),
         action : SnackBarAction(label : "Ok", onPressed: () {})
      )
   );
}
```

Здесь мы должны извлечь некоторую информацию из ответа, а именно название комнаты и имя пользователя, который их пригласил. Затем добавляется приглашение в эту комнату, так что, когда они нажмут на нее на основном экране, мы узнаем, как их впустить. Затем мы должны показать им `SnackBar`, чтобы сообщить о приглашении. Мы оставим его на целую минуту, чтобы они его не пропустили, иначе нет никаких признаков того, что приглашение пришло (эй, вот вам еще одно упражнение: добавьте какой-нибудь индикатор в список комнат на главном экране!). Мы также покажем им кнопку **Ok**, чтобы закрыть `SnackBar`, если они захотят.

Наконец, мы подошли к последней функции, которая обрабатывает сообщения, отправленные в комнату:

```
void posted(inData) {
   Map<String, dynamic> payload = jsonDecode(inData);
```

```
    if (model.currentRoomName == payload["roomName"]) {
      model.addMessage(payload["userName"], payload["message"]);
    }
}
```

Еще раз, новые сообщения отправляются всем пользователям, поэтому мы должны игнорировать все сообщения, не относящиеся к текущей комнате. Однако для пользователей, которые находятся внутри комнаты, вызов `model.addMessage()` добавит сообщение и уведомит их.

И теперь у нас есть полный API для связи с сервером, на базе которого мы можем написать код нашего клиентского приложения. Первая часть этой головоломки находится в файле main.dart.

main.dart

Есть несколько задач, которые нужно выполнить в `main()` перед созданием пользовательского интерфейса. Это может занять некоторое время, так что мы сперва выполняем их:

```
void main() {

  startMeUp() async {

    Directory docsDir =
      await getApplicationDocumentsDirectory();
    model.docsDir = docsDir;

    var credentialsFile =
      File(join(model.docsDir.path, "credentials"));
    var exists = await credentialsFile.exists();

    var credentials;
    if (exists) {
      credentials = await credentialsFile.readAsString();
    }
```

Опять же, есть функция `startMeUp()`, которая вызывается в самом конце `main()`, поэтому мы можем выполнить внутри нее некоторую асинхронную задачу. Первая такая задача – получить каталог документов приложения, как в предыдущем проекте. В нашем случае это будет файл для хранения имени и пароля пользователя (*учетные данные*). Следующий шаг – попытаться прочитать этот файл. Если он существует, то мы читаем его содержимое как строку. Мы разберемся с этим через минуту, но сначала создадим пользовательский интерфейс:

```
runApp(FlutterChat());
```

Прежде чем мы доберемся до класса `FlutterChat`, нам нужно разобраться с учетными данными. Цель в том, чтобы при наличии файла с логином и паро-

лем мы могли сразу же проверить пользователя на сервере. Если такого файла нет, то мы должны показать пользователю диалог входа в систему. Итак:

```
if (exists) {
   List credParts = credentials.split("============");
   LoginDialog().validateWithStoredCredentials(credParts[0],
      credParts[1]);
} else {
   await showDialog(context : model.rootBuildContext,
      barrierDismissible : false,
      builder : (BuildContext inDialogContext) {
         return LoginDialog();
      }
   );
}
```

Содержимое файла представляет собой простую строку в виде `xxx==========yyy`, где `xxx` – это имя пользователя, а `yyy` – пароль. Вы спросите, почему их разделяют 12 знаков равенства? Все просто: имя пользователя и пароль ограничены десятью символами, мы увеличиваем их количество на 2, чтобы, если пользователь вводит десять знаков равенства в имени пользователя, мы все равно могли разделить нашу строку на части с помощью данной 12-символьной подстроки. Да, я мог бы использовать один символ, возможно запятую, и просто запретить этот символ в имени пользователя, но я хотел дать пользователям полную свободу!

Как видите, если файла учетных данных не существует, то открывается диалоговое окно входа в систему. Мы рассмотрим это в следующем разделе. Как упоминалось ранее, после этого вызывается `startMeUp()`, и именно здесь все начинается.

Примечание. Существует пограничный сценарий, когда при регистрации пользователя перезапускается сервер, а *другой* пользователь регистрируется с таким же `userName`, то при попытке первого пользователя выполнить повторную проверку произойдет сбой, поскольку пароли не совпадут. В этом случае код в `validateWithStoredCredentials()` удалит файл учетных данных и предупредит пользователя. При перезапуске приложения им будет предложено ввести новые учетные данные.

Теперь вернемся к классу `FlutterChat`:

```
class FlutterChat extends StatelessWidget {
   @override
   Widget build(final BuildContext context) {
      return MaterialApp(
         home : Scaffold(body : FlutterChatMain())
      );
   }
}
```

Он начинается с паттерна, с которым вы уже должны быть хорошо знакомы: MaterialApp со Scaffold внутри. Body указывает на класс FlutterChatMain, с которого начинается наш пользовательский интерфейс:

```
class FlutterChatMain extends StatelessWidget {

   @override
   Widget build(final BuildContext inContext) {
      model.rootBuildContext = inContext;
```

Как вы видели в файле Model.dart, rootBuildContext кешируется для использования другим кодом, и поскольку он представлен только в методе build(), нам необходимо сделать это внутри данного метода. Далее создается виджет:

```
return ScopedModel<FlutterChatModel>(model : model,
   child : ScopedModelDescendant<FlutterChatModel>(
   builder : (BuildContext inContext, Widget inChild,
      FlutterChatModel inModel) {
         return MaterialApp(initialRoute : "/",
            routes : {
               "/Lobby" : (screenContext) => Lobby(),
               "/Room" : (screenContext) => Room(),
               "/UserList" : (screenContext) => UserList(),
               "/CreateRoom" : (screenContext) => CreateRoom()
            },
            home : Home()
```

Поскольку в этом приложении мы собираемся использовать встроенный во Flutter модуль Navigator, а не подход «сделай сам», принятый во FlutterBook, первая задача – определить маршруты (другими словами, экраны) приложения. Их четыре: /Lobby (список комнат), /Room (внутри комнаты), /UserList (список пользователей на сервере) и /CreateRoom (создание комнаты). Они называются *именованными маршрутами* (named routes), потому что у них есть имена! Без них вы все еще можете перемещаться между экранами, но вам придется вручную манипулировать стеком навигации, что приведет к большому количеству дублирующегося кода. С использованием именованных маршрутов код становится намного чище, и вы уже видели это в Connector.dart.

Как вы можете догадаться, названия маршрутов могут представлять иерархию и быть настолько сложными, насколько вам нравится. Итак, если у вас есть страница pageA, которая содержит две «дочерние» страницы 1a и 2a, то вы можете назвать их /pageA, /pageA/1a и /pageA/2a. Здесь они находятся на одном уровне, поэтому я постарался реализовать все максимально просто (можно утверждать, что поскольку эти экраны запускаются с основного экрана, то они должны называться /Lobby/Room и /Lobby/CreateRoom, справедливо, но в нашем случае это ничего не упростит).

Значение initialRoute сообщает навигатору, какой экран отображать при старте, и это соответствует свойству home. Обратите внимание, что использова-

ние свойства `home` и одновременное указание корневого (root) маршрута с именем «/» – это ошибка. Но если вы удалите свойство `home`, то у вас останется только «/». И тогда вам понадобится дополнительный код, чтобы вернуться к главному экрану, поэтому позвольте Flutter и Navigator сделать это за вас.

LoginDialog.dart

Если мы не сохраним данные пользователя, то отобразится диалоговое окно входа в систему, в котором пользователь может авторизоваться либо зарегистрироваться. Это стандартный диалог входа, как показано на рис. 8-1.

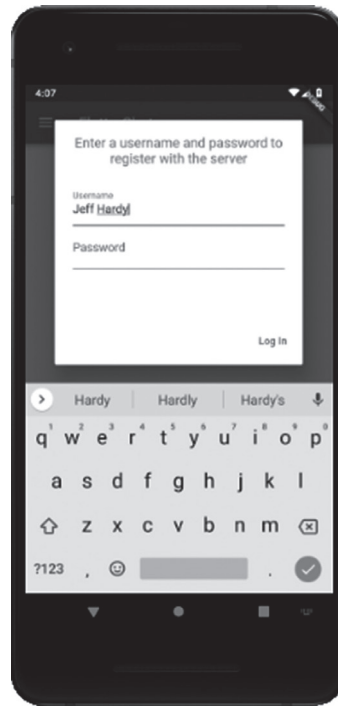


Рисунок 8-1. *Диалог входа (авторизации)*

Просто введите имя пользователя, пароль и нажмите кнопку **Log In**. Этот код начинается довольно обыденно:

```
class LoginDialog extends StatelessWidget {

   static final GlobalKey<FormState> _loginFormKey =
      new GlobalKey<FormState>();
```

Мы будем иметь дело с формой, поэтому нам потребуется `GlobalKey` для некоторой проверки. В конечном итоге мы будем заполнять две переменные:

```
String _userName;
String _password;
Widget build(final BuildContext inContext) {
   return ScopedModel<FlutterChatModel>(model : model,
      child : ScopedModelDescendant<FlutterChatModel>(
      builder : (BuildContext inContext, Widget inChild,
         FlutterChatModel inModel) {
         return AlertDialog(content : Container(height : 220,
            child : Form(key : _loginFormKey,
               child : Column(children : [
                  Text("Enter a username and password to "
                     "register with the server",
                     textAlign : TextAlign.center, fontSize : 18
                     style : TextStyle(color :
                        Theme.of(model.rootBuildContext).accentColor)
                  ),
                  SizedBox(height : 20)
```

Поскольку здесь задействовано состояние, мы оборачиваем все в Scoped Model. А затем внутри него используем ScopedModelDescendant. С таким подходом мы познакомились ранее. Затем идет функция builder(), которая возвращает AlertDialog. Содержимое этого диалога представляет собой Form, которая ссылается на _loginFormKey. Потом идет Column с элементами, первый из которых – это заголовок. Он получает свой цвет из активной Theme вашего MaterialApp. Обратите внимание на использование rootBuildContext, из которого мы и берем активную Theme. Чтобы добавить немного пространства между текстом заголовка и полями формы, мы используем SizedBox:

```
TextFormField(
   validator : (String inValue) {
      if (inValue.length == 0 || inValue.length > 10) {
         return "Please enter a username no "
            "more than 10 characters long";
         }
         return null;
   },
   onSaved : (String inValue) { _userName = inValue; },
   decoration : InputDecoration(
      hintText : "Username", labelText : "Username")
),
TextFormField(obscureText : true,
   validator : (String inValue) {
      if (inValue.length == 0) {
         return "Please enter a password";
      }
```

```
      return null;
   },
   onSaved : (String inValue) { _password = inValue; },
   decoration : InputDecoration(
      hintText : "Password", labelText : "Password")
)
```

Здесь не должно быть сюрпризов. Существует ограничение на количество символов для имени пользователя (это важно, потому что мы используем токены в нашем `main.dart`), а также необходимо убедиться, что пользователь ввел пароль (а не только имя пользователя).

Далее следует кнопка **Log In**, которая содержится в коллекции действий для диалога:

```
actions : [
   FlatButton(child : Text("Log In"),
      onPressed : () {
         if (_loginFormKey.currentState.validate()) {
            _loginFormKey.currentState.save();
            connector.connectToServer(() {
               connector.validate(_userName, _password,
                  (inStatus) async {
                  if (inStatus == "ok") {
                     model.setUserName(_userName);
                     Navigator.of(model.rootBuildContext).pop();
                     model.setGreeting("Welcome back, $_userName!");
```

После нажатия кнопки и при условии, что данные формы пройдут проверку, состояние формы сохранится. Это запустит выполнение обработчиков `onSaved` для отдельных полей. В этих обработчиках мы заполним наши переменные `_userName` и `_password`. Затем идет вызов `Connector.connectToServer()`. Как вы помните, он устанавливает соединение с сервером и настраивает все обработчики сообщений. Этот метод передает функцию callback, вызываемую после установки соединения. Этот callback, в свою очередь, вызовет функцию `connector.validate()`. Она передает `_userName` и `_password` на сервер для проверки. Если возвращается статус «ok», то пользователь уже известен серверу, и пароль был верным, поэтому мы можем продолжить, что означает сохранение имени пользователя в модели, затем идет `pop()`, удаление диалога и настройка приветствия на главном экране (которое мы рассмотрим далее). Если статус не возвращается, то отображается `SnackBar`, указывающий, что имя пользователя уже занято:

```
} else if (inStatus == "fail") {
   Scaffold.of(model.rootBuildContext
   ).showSnackBar(SnackBar(backgroundColor : Colors.red,
      duration : Duration(seconds : 2),
```

```
      content : Text("Sorry, that username is already taken")
   ));
```

Если имя пользователя новое для сервера, то приходит сообщение «created»:

```
} else if (inStatus == "created") {
   var credentialsFile = File(join(
      model.docsDir.path, "credentials"));
   await credentialsFile.writeAsString(
      "$_userName============$_password");
   model.setUserName(_userName);
   Navigator.of(model.rootBuildContext).pop();
   model.setGreeting("Welcome to the server, $_userName!");
}
```

Здесь нам нужно куда-нибудь сохранить учетные данные, поэтому мы создаем экземпляр класса `File`. Затем используем функцию `join()` для создания пути к каталогу документов приложения. Путь к этому каталогу будет получен при запуске приложения. Далее применяем метод `writeAsString()`, который запишет логин и пароль в файл учетных данных, используя странный разделитель из знаков равенства! После этого мы выполняем ту же настройку, что и в случае «ok», но с другим приветствием.

Вход для существующих пользователей

Теперь у нас есть диалоговое окно для входа в систему. Оно также содержит код, который работает, когда приложение запускается и находит существующий файл учетных данных. В этом случае к серверу все еще нужно обращаться, но пользовательского интерфейса нет; это происходит автоматически, когда в игру вступает функция `validateWithStoredCredentials()`:

```
void validateWithStoredCredentials(final String inUserName,
   final String inPassword) {

      connector.connectToServer(model.rootBuildContext, () {
      connector.validate(inUserName, inPassword, (inStatus) {
         if (inStatus == "ok" || inStatus == "created") {
            model.setUserName(inUserName);
            model.setGreeting("Welcome back, $inUserName!");
```

Как и прежде, сначала вызывается `connector.connectToServer()`, а затем `connector.validate()`, передавая ему имя пользователя и пароль, которые будут получены из файла учетных данных. В этом случае логика немного проще, потому что с точки зрения пользователя он уже регистрировался ранее, однако для сервера это новый пользователь, за исключением ситуации, когда кто-то уже успел зарегистрироваться с таким же именем. Это немного увеличит нагрузку на сервер, но он всего лишь машина, и кого волнуют его чувства?! (конечно, если

мы не в эпизоде сериала *«Звездный путь: следующее поколение»*, где обсуждается личность данных!). Однако мы заботимся о чувствах пользователей! Таким образом, независимо от того, вернем ли мы сообщение «ok» или «created», мы оповестим пользователя, что сервер его не забыл, даже если это не так!

А еще мы можем вернуть ошибку. Например, если логин был занят другим человеком, то пароль наверняка не подойдет, и пользователь не сможет авторизоваться. Но здесь мы знаем причину неправильного пароля: новый пользователь взял тот же логин после перезапуска и успел авторизоваться первым. Давайте обработаем эту ошибку:

```
} else if (inStatus == "fail") {
   showDialog(context : model.rootBuildContext,
      barrierDismissible : false,
      builder : (final BuildContext inDialogContext) =>
      AlertDialog(title : Text("Validation failed"),
         content : Text("It appears that the server has "
            "restarted and the username you last used "
            "was subsequently taken by someone else. "
            "\n\nPlease re-start FlutterChat and choose "
            "a different username."
         )
```

Поскольку это вообще «game over», то мы покажем `AlertDialog` и сделаем так, что его нельзя будет отклонить без заданных нами действий. Для этого нам нужно установить значение для `barrierDismissable`, равное `false`. Это гарантирует, что при нажатии за пределы диалогового окна оно не скроется, как это происходит по умолчанию. Это сообщение объяснит ситуацию. Затем мы напишем обработчик для кнопки **Ok**:

```
actions : [
   FlatButton(child : Text("Ok"),
   onPressed : () {
      var credentialsFile = File(join(
         model.docsDir.path, "credentials"));
      credentialsFile.deleteSync();
      exit(0);
   })
]
```

Теперь мы знаем, что это имя пользователя нельзя использовать, поэтому нам нужно удалить файл учетных данных, чтобы избежать повторения при следующем запуске. Наконец, вызывается функция `exit()`, то есть функция, которую Flutter предоставляет для завершения приложения. При следующем запуске пользователю будет предложено ввести имя пользователя и пароль.

Теперь давайте посмотрим, где же используются приветственные сообщения – домашний экран.

Home.dart

Домашний экран в файле `Home.dart` – это первый экран, который видит пользователь (а также то, к чему он возвращается при возникновении различных событий, включая закрытие комнаты или удаление из нее). Как вы видите на рис. 8-2, этот экран довольно прост.

Рисунок 8-2. *Домашний экран*

Его код тоже вполне понятен:

```
class Home extends StatelessWidget {
   Widget build(final BuildContext inContext) {
      return ScopedModel<FlutterChatModel>(model : model,
         child : ScopedModelDescendant<FlutterChatModel>(
            builder : (BuildContext inContext, Widget inChild,
               FlutterChatModel inModel) {
               return Scaffold(drawer : AppDrawer(),
                  appBar : AppBar(title : Text("FlutterChat")),
                  body : Center(child : Text(model.greeting))
               );
            }
         )
      );
```

```
    }
}
```

Да, вот так просто! В конечном итоге это всего лишь виджет Text внутри виджета `Center`. Виджет `Text` получит значение из свойства `model.greeting`, чтобы отобразить его содержимое пользователю. Все остальное ничем не примечательно!

AppDrawer.dart

Боковое меню (`AppDrawer`) реализуется в файле `AppDrawer.dart` – это один из способов навигации пользователя по приложению. Его можно увидеть на рис. 8-3.

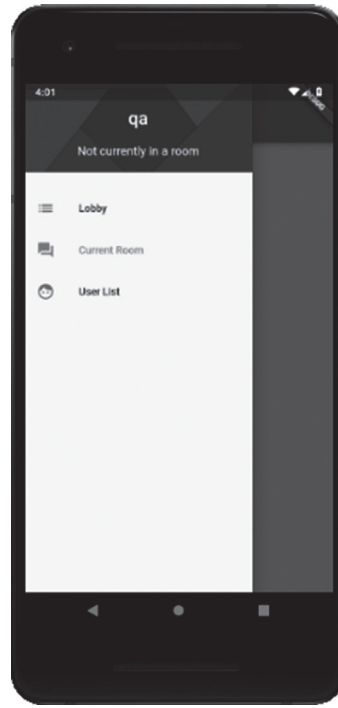


Рисунок 8-3. *AppDrawer*

Сверху мы видим заголовок с красивым фоном, над которым стоит имя пользователя и название комнаты, в этом случае название является заглушкой, которую вы видели в коде `Model.dart`.

Класс `AppDrawer` начинается знакомо:

```
class AppDrawer extends StatelessWidget {
   Widget build(final BuildContext inContext) {
      return ScopedModel<FlutterChatModel>(model : model,
```

```
child : ScopedModelDescendant<FlutterChatModel>(
builder : (BuildContext inContext, Widget inChild,
   FlutterChatModel inModel) {
      return Drawer(child : Column(children : [
         Container(decoration : BoxDecoration(image :
            DecorationImage(fit : BoxFit.cover,
               image : AssetImage("assets/drawback01.jpg")
         ))
```

В конечном итоге создается виджет Drawer, а внутри него макет Column. Первый элемент в этом макете – Container, который украшен DecorationImage. Как следует из названия, это виджет для украшения изображения. AssetImage – это изображение, созданное из файла drawback01.jpg в каталоге ресурсов. Использование значения BoxFit.cover для свойства fit говорит Flutter о том, что размер изображения должен быть как можно меньше, но при этом заполнить прямоугольник с рамкой.

После этого идет дочерний элемент Container, где отображаются имя пользователя и текущая комната:

```
child : Padding(
   padding : EdgeInsets.fromLTRB(0, 30, 0, 15),
   child : ListTile(
      title : Padding(padding : EdgeInsets.fromLTRB(0,0,0,20),
         child : Center(child : Text(model.userName,
            style : TextStyle(color : Colors.white, fontSize : 24)
         ))
      ),
      subtitle : Center(child : Text(model.currentRoomName,
         style : TextStyle(color : Colors.white, fontSize : 16)
      ))
```

Во-первых, для обеспечения подходящего расстояния между этими значениями используется отступ (padding). Затем я использую ListTile, чтобы текст с именем пользователя был крупнее названия комнаты, для чего применяются поля title и subtitle соответственно. Я также добавляю несколько отступов в названии, чтобы контролировать интервал между этими двумя текстами и избежать их слишком большого скопления. Конечно, цвет должен отличаться от черного, используемого по умолчанию, в противном случае текст будет плохо виден на заднем фоне. Я также настрою fontSize, чтобы шрифт выглядел так, как я хочу. Отображаемый текст поступает из соответствующих полей модели, чтобы они автоматически обновлялись.

Затем у пользователя появляется возможность выбора из трех элементов на основном экране:

```
Padding(padding : EdgeInsets.fromLTRB(0, 20, 0, 0),
   child : ListTile(leading : Icon(Icons.list),
```

```
      title : Text("Lobby"),
      onTap: () {
         Navigator.of(inContext).pushNamedAndRemoveUntil(
            "/Lobby", ModalRoute.withName("/"));
         connector.listRooms((inRoomList) { model.setRoomList(inRoomList);
         });
      }
   )
)
```

Это все еще `ListTile` с добавлением отступа, чтобы я мог красиво распределить элементы. Каждый элемент получит значок, отражающий его функциональность. Когда срабатывает обработчик `onTap`, появляется необходимость выполнения двух задач. Во-первых, мы получаем ссылку на `Navigator` для `inContext` и вызываем метод `pushNamedAndRemoveUntil()`, указывая имя маршрута для навигации. Затем вызывается метод для получения обновленного списка комнат. Теоретически в этом нет необходимости, поскольку сервер будет сообщать о добавлении или закрытии комнаты, и список будет обновляться автоматически. Но повторная загрузка не будет лишней, если вы хотите удостовериться, что список действительно обновился. Наконец, обновленный список комнат записывается в модели и отображается пользователю.

Следующие два элемента, `Current Room` и `User List`, реализуются по аналогии с только что просмотренным кодом, за исключением одного: при переходе в текущую комнату не требуется никаких обращений к серверу или обновления модели, так что код просто осуществляет навигацию, и все. Ну и конечно же, элемент `User List` вызывает методы `connector.listUsers()` и `model.setUserList()` вместо методов комнаты, о чем вы, наверное, уже догадались! Итак, мы не будем рассматривать этот код, а просто перейдем к экрану Lobby.

Lobby.dart

Основной экран показан на рис. 8-4. Он содержится в файле `Lobby.dart` и представляет собой простой `ListView`, который показывает комнаты на сервере. Мы используем иконку замка, чтобы указать на закрытость/открытость комнаты. Также отображается название комнаты и ее описание.

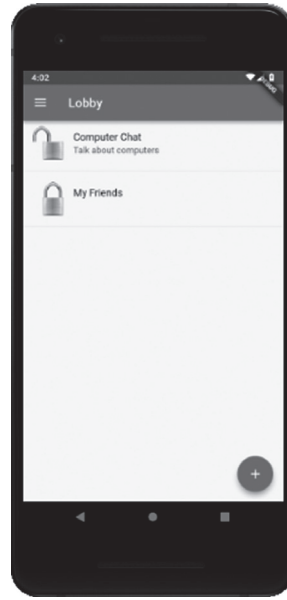


Рисунок 8-4. *Экран Lobby (список комнат)*

Щелчок по одному из элементов списка либо открывает комнату, либо указывает пользователю, что комната закрыта и он не может в нее войти (при условии что у него нет приглашения). Для создания новой комнаты также используют FAB. Это может сделать любой пользователь.

```
class Lobby extends StatelessWidget {
   Widget build(final BuildContext inContext) {
      return ScopedModel<FlutterChatModel>(model : model,
      child : ScopedModelDescendant<FlutterChatModel>(
         builder : (BuildContext inContext, Widget inChild,
         FlutterChatModel inModel) {
            return Scaffold(drawer : AppDrawer(),
               appBar : AppBar(title : Text("Lobby")),
               floatingActionButton : FloatingActionButton(
                  child : Icon(Icons.add, color : Colors.white),
                  onPressed : () {
                     Navigator.pushNamed(inContext, "/CreateRoom");
                  }
               )
```

Код начинается с привычного паттерна, как и везде, где используется `scoped_model`, так как без данных он не будет работать! То, что нас интересует, находится в обработчике события `onPressed` для FAB. Здесь мы выбираем маршрут `/CreateRoom`, который покажет пользователю экран для создания комнаты. Это описано в следующем разделе, поэтому пока продолжим:

```
body : model.roomList.length == 0 ?
   Center(child :
      Text("There are no rooms yet. Why not add one?")) :
      ListView.builder(itemCount : model.roomList.length,
      itemBuilder : (BuildContext inBuildContext, int inIndex) {
         Map room = model.roomList[inIndex];
         String roomName = room["roomName"];
         return Column(children : [
```

Существует вероятность того, что на сервере нет комнат, поэтому вместо пустого экрана я решил вставить сообщение об этом в центре экрана. Если комнаты есть, то мы добавим ListView. Описания для каждой комнаты мы возьмем из model.roomList, а затем добавим нужные виджеты в Column. Таким способом я хочу показать комнату в ListTile, а затем добавить Divider, для этого я использую виджет со свойством children.

ListTile для комнаты выглядит следующим образом:

```
ListTile(leading : room["private"] ?
   Image.asset("assets/private.png") :
   Image.asset("assets/public.png"),
   title : Text(roomName), subtitle : Text(room["description"])
```

Во-первых, это значок замка, который находится в leading. Значение private в отображении (map) комнаты говорит нам, является ли она закрытой. Для определения того, какое изображение должно отображаться, мы используем тернарный оператор. Затем отображаются title и subtitle, что уже привычно для ListTile.

Благодаря обработчику onTap можно нажать на отдельную комнату.

```
onTap : () {
   if (room["private"] &&
      !model.roomInvites.containsKey(roomName) &&
      room["creator"] != model.userName) {
         Scaffold.of(
            inBuildContext).showSnackBar(SnackBar(
               backgroundColor : Colors.red,
               duration : Duration(seconds : 2),
               content : Text("Sorry, you can't "
                  "enter a private room without an invite")
            ));
```

Сначала мы определяем, закрыта ли комната. Если это так, то мы проверяем наличие приглашения у пользователя. Еще мы проверяем, является ли он администратором комнаты. Если комната закрыта, а у пользователя нет приглашения, и он ее не создавал, то показываем ему SnackBar с сообщением о том, что он не может войти в комнату без приглашения.

Теперь смотрим, что дальше – или комната открыта, или у пользователя есть приглашение, или он создатель:

```
} else {
   connector.join(model.userName, roomName,
      (inStatus, inRoomDescriptor) {
      if (inStatus == "joined") {
         model.setCurrentRoomName(inRoomDescriptor["roomName"]);
         model.setCurrentRoomUserList(inRoomDescriptor["users"]);
         model.setCurrentRoomEnabled(true);
         model.clearCurrentRoomMessages();
         if (inRoomDescriptor["creator"] == model.userName) {
            model.setCreatorFunctionsEnabled(true);
         } else {
            model.setCreatorFunctionsEnabled(false);
         }
         Navigator.pushNamed(inContext, "/Room");
```

Вход в комнату влечет за собой немного работы по настройке. Во-первых, сервер получает уведомление о входе пользователя в комнату благодаря методу `connector.join()`, отправляющему сообщение «`join`». Если возвращается «`joined`», то пользователь входит в комнату. В этом случае записывается текущее имя комнаты, а также список пользователей в комнате, которые вернул сервер. Для начала мы должны установить новое название комнаты в боковом меню (`AppDrawer`) и убедиться, что нет списка старых сообщений. Если наш пользователь является администратором, то мы включаем соответствующие функции. Наконец, переходим по маршруту `/Room` для отображения экрана комнаты, который будет рассмотрен в заключительном разделе этой главы.

Последний сценарий, с которым мы должны разобраться, – это когда сервер отвечает, что комната заполнена, потому что в каждой комнате есть ограничение по количеству людей. Итак, мы находим другую логическую ветку:

```
} else if (inStatus == "full") {
   Scaffold.of(inBuildContext).showSnackBar(SnackBar(
      backgroundColor : Colors.red,
      duration : Duration(seconds : 2),
      content : Text("Sorry, that room is full")
   ));
}
```

Как и в случае отсутствия приглашения, `SnackBar` оповещает пользователя о том, что комната заполнена. На этом мы заканчиваем с экраном Lobby! Теперь давайте разберемся с тем, что происходит при нажатии кнопки **FAB**, которая ведет нас в `CreateRoom.dart`.

CreateRoom.dart

Пришло время создать несколько комнат! Ох уж эта сила богов, сила творения, заключенная во Flutter! Рисунок 8-5 демонстрирует эту магическую мощь.

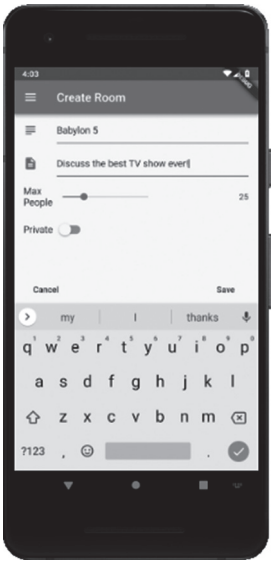


Рисунок 8-5. *Экран создания комнаты*

Это достаточно простой экран, потому что создать комнату несложно. Требуется только название комнаты. В описании нет необходимости, а максимальное количество людей в комнате задано по умолчанию (хотя его можно настроить с помощью `Slider`). Комнату также можно сделать закрытой, активировав для этого виджет `Switch`. Затем нажмите **Save**, и комната создана!

Так как в этот раз мы будем создавать виджет с состоянием, у нас будет два класса: фактический класс виджета и соответствующий ему объект состояния. Мы начнем с класса виджета:

```
class CreateRoom extends StatefulWidget {
   CreateRoom({Key key}) : super(key : key);
   @override
   _CreateRoom createState() => _CreateRoom();
}
```

Просто шаблонный код, ничего нового. Поэтому мы перейдем к объекту `_CreateRoom`, который наследуется от `State`:

```
class _CreateRoom extends State {

   String _title;
   String _description;
   bool _private = false;
```

```
   double _maxPeople = 25;
   final GlobalKey<FormState> _formKey= GlobalKey<FormState>();
```

Нам понадобится несколько переменных для полей формы и GlobalKey для самой Form. Метод build() может начинаться так:

```
Widget build(final BuildContext inContext) {
   return ScopedModel<FlutterChatModel>(model : model, child :
      ScopedModelDescendant<FlutterChatModel>(
      builder : (BuildContext inContext, Widget inChild,
         FlutterChatModel inModel) {
            return Scaffold(resizeToAvoidBottomPadding : false,
               appBar : AppBar(title : Text("Create Room")),
               drawer : AppDrawer(), bottomNavigationBar :
                  Padding(padding : EdgeInsets.symmetric(
                     vertical : 0, horizontal : 10
                  ),
                  child :
                       SingleChildScrollView(child : Row(children : [
```

Как обычно, при работе с моделью мы используем ScopedModel со ScopedModel Descendant внутри, а также функцию builder() для получения виджета, которому нужен доступ к модели. Как и на экранах Home и Lobby, создается Scaffold, и на этот раз мы передаем свойство resizeToAvoidBottomPadding и устанавливаем его в false. Оно управляет изменением размера виджетов внутри Scaffold при отображении экранной клавиатуры. Обычно вы хотите, чтобы оно было по умолчанию установлено в true, дабы избежать исчезновения виджетов под клавиатурой. Однако в некоторых случаях виджеты сами скрываются при появлении клавиатуры, что может быть нежелательным. В таком случае установка свойства resizeToAvoidBottomPadding в false решит проблему. Но если виджеты находятся в контейнере с прокруткой (scroll), то пользователь может их пролистывать. Помимо этого, мы задаем title для appBar и устанавливаем AppDrawer. Еще у нас есть bottomNavigationBar с небольшим отступом вокруг, чтобы между кнопками и краями экрана было пространство в несколько пикселей (только для удобства). Затем определяются сами кнопки:

```
FlatButton(child : Text("Cancel"),
   onPressed : () {
      FocusScope.of(inContext).requestFocus(FocusNode());
      Navigator.of(inContext).pop();
   }
),
Spacer()
```

Сначала идет кнопка (**Cancel**), и все, что нужно сделать при нажатии, – это скрыть клавиатуру, а затем убрать экран (помните, что это маршрут, то есть от-

дельный экран, а не диалог). Затем появляется Spacer, который сдвигает вправо вторую кнопку **Save**, которая выглядит следующим образом:

```
FlatButton(child : Text("Save"),

   onPressed : () {
      if (!_formKey.currentState.validate()) { return; }
      _formKey.currentState.save();
      int maxPeople = _maxPeople.truncate();
      connector.create(_title, _description,
         maxPeople, _private,
         model.userName, (inStatus, inRoomList) {
         if (inStatus == "created") {
            model.setRoomList(inRoomList);
            FocusScope.of(inContext).requestFocus(FocusNode());
            Navigator.of(inContext).pop();
         } else {
            Scaffold.of(inContext).showSnackBar(SnackBar(
               backgroundColor : Colors.red,
               duration : Duration(seconds : 2),
               content : Text("Sorry, that room already exists")
            ));
         }
      });
```

Сначала проверяются поля формы с помощью `validate()`, а затем сохраняется ее состояние. Далее должно быть обрезано (truncate) значение `_maxPeople`. Мы же хотим получить целочисленное значение типа `int`, а `Slider` возвращает нам число с плавающей запятой и дробной частью. Когда все это произойдет, мы сможем вызвать метод `connector.create()`, который сообщит серверу о создании комнаты. Нужно обработать два возможных результата: либо комната была создана, либо нет. Последнее происходит при условии, что такое имя уже используется. Для этого мы проверяем аргумент `inStatus`, переданный в callback-функцию. Если он равен «`created`», значит, сервер отправил нам обновленный список комнат, и мы сохраняем его в модели. Затем клавиатура скрывается, и экран убирается из стека `Navigator`, возвращая пользователя к экрану Lobby. Однако если комната не была создана, то отображается SnackBar, который сообщает пользователю о том, что имя уже занято, и предлагает выбрать новое.

Строим форму

Теперь нам просто нужно построить саму форму. Код тот же, что и в предыдущих примерах форм.

```
body : Form(key : _formKey, child : ListView(
   children : [
```

```
      ListTile(leading : Icon(Icons.subject),
         title : TextFormField(decoration :
            InputDecoration(hintText : "Name"),
            validator : (String inValue) {
               if (inValue.length == 0 || inValue.length > 14) {
                  return "Please enter a name no more "
                     "than 14 characters long";
               }
               return null;
            },
            onSaved : (String inValue) {setState(() { _title = inValue; });
            }
         )
      )
```

Каждое поле в форме содержится в `ListTile`, начиная с поля **Name**. `validator` гарантирует, что длина ограничивается 14 символами (я выбрал такую длину, чтобы не было переносов или обрезаний текста, как бывает с более длинным именем).

Поле с описанием определено аналогично, за исключением того, что я не применял к нему никаких ограничений:

```
ListTile(leading : Icon(Icons.description),
   title : TextFormField(decoration :
      InputDecoration(hintText : "Description"),
      onSaved : (String inValue) {
         setState(() { _description = inValue; });
      }
   )
)
```

Затем идет поле `Max People`, и здесь мы сталкиваемся с чем-то новым, виджетом `Slider`:

```
ListTile(title : Row(children : [ Text("Max\nPeople"),
   Slider(min : 0, max : 99, value : _maxPeople,
      onChanged : (double inValue) {
         setState(() { _maxPeople = inValue; });
      }
   )
]),
trailing : Text(_maxPeople.toStringAsFixed(0))
)
```

Это достаточно простой виджет, требующий минимального (min) и максимального (max) значений для определения его конечных точек, и в этом случае свойство `value` связывается со свойством `_maxPeople`. Для этого поля нет провер-

ки, но всякий раз, когда значение изменяется, нам нужно задавать его в State. Наконец, возникает проблема: когда пользователь перемещает Slider, значение, которое он выбирает, не отображается, и на нем нет никаких вспомогательных отметок. Чтобы исправить это, я поставил виджет Text в поле trailinge и задал для отображения значение _maxPeople. Конечно, для отображения Text требуется текст, а _maxPeople – это число. К счастью, double использует несколько методов преобразования его в строку, один из них – toStringAsFixed() (есть также toStringAsExponential() и toStringAsPrecision()). Он делает именно то, что нам нужно: преобразует double в строку и позволяет установить значение с точностью до десятых. Конечно, мне *не нужны* числа после запятой, поэтому я передаю этому методу значение 0.

Остается только одно поле, которое делает комнату приватной:

```
ListTile(title : Row(children : [ Text("Private"),
   Switch(value : _private,
      onChanged : (inValue) {
         setState(() { _private = inValue; });
      }
   )
]))
```

Впервые вы видите использование виджета Switch (переключатель). Это удобно, поскольку это бинарный выбор: комната либо публичная, либо приватная. Checkbox бы тоже сработал, но я решил познакомить вас с работой Switch!

UserList.dart

Экран списка пользователей, как показано на рис. 8-6, является следующим интересующим нас фрагментом кода и содержится в файле UserList.dart.

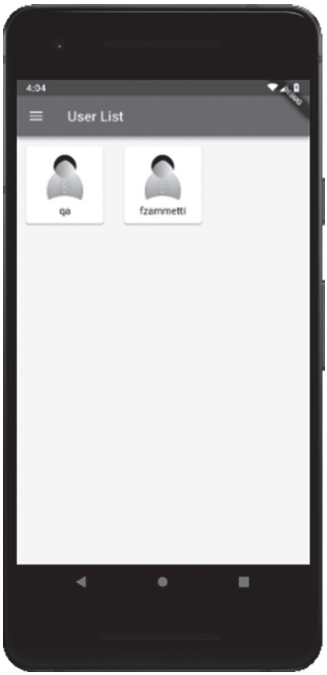


Рисунок 8-6. *Экран списка пользователей*

Сам экран очень прост: это всего лишь GridView с элементом для каждого зарегистрированного пользователя. Каждый элемент пользовательской сетки размещается в виджете Card и использует базовый значок просто для удобства (можно позволить пользователям выбирать значок аватара, как для контактов во FlutterBook, еще одно прекрасное самостоятельное упражнение, не так ли?!).

Код начинается следующим образом:

```
class UserList extends StatelessWidget {
   Widget build(final BuildContext inContext) {
      return ScopedModel<FlutterChatModel>(model : model,
         child : ScopedModelDescendant<FlutterChatModel>(
         builder : (BuildContext inContext, Widget inChild,
            FlutterChatModel inModel) {
            return Scaffold(drawer : AppDrawer(),
               appBar : AppBar(title : Text("User List")),
               body : GridView.builder(
                  itemCount : model.userList.length,
                  gridDelegate :
                     SliverGridDelegateWithFixedCrossAxisCount(
                        crossAxisCount : 3
                     )
```

Он начинается так же, как и любой другой класс, который вы видели. Нам нужны данные из модели, так что все, как обычно, соответствует иерархии ScopedModel/ScopedModelDescendant/builder( ). Мы строим экран, поэтому возвращаемым корневым виджетом является Scaffold, на который ссылается AppDrawer, чтобы мы не потеряли его на этом экране. Затем начинается body. Как я уже сказал, это GridView, поэтому мы используем конструктор этого класса builder( ) и передаем ему length коллекции model.userList в качестве значения свойства itemCount. Далее предоставляется gridDelegate типа SliverGridDelegate WithFixedCrossAxisCount (Flutter не знает коротких имен классов!). Здесь мы с помощью свойства crossAxisCount указываем, что хотим отобразить три элемента в строке.

Затем начинаем строить наши элементы с помощью функции itemBuilder:

```
itemBuilder : (BuildContext inContext, int inIndex) {
   Map user = model.userList[inIndex];
   return Padding(padding : EdgeInsets.fromLTRB(10,10,10,10),
      child : Card(child : Padding(padding :
         EdgeInsets.fromLTRB(10, 10, 10, 10),
         child : GridTile(
            child : Center(child : Padding(
               padding : EdgeInsets.fromLTRB(0, 0, 0, 20),
               child : Image.asset("assets/user.png")
            )),
            footer : Text(user["userName"],
            textAlign : TextAlign.center)
         )
      )
   ));
}
```

Для каждого элемента мы получаем дескриптор пользователя из отображения userList в нашей модели.

Затем строится виджет Card, обернутый в Padding, с пространством вокруг него (чтобы элементы в GridView не наезжали друг на друга). Дочерний элемент Card – это наш добрый сосед GridTile, типичный дочерний элемент GridView. Дочерним элементом GridTile является виджет Image (отображает изображение «user.png» из ресурсов приложения), обернутый в Padding, дабы контролировать пространство вокруг него (в данном случае только ради отступа снизу, чтобы отделить его от имени пользователя), и это все обернуто в Center, чтобы центрировать его на Card. Наконец, нижний элемент Card – это виджет, используемый для отображения имени пользователя.

Вот так просто можно сделать список пользователей! Это легко, если не добавлять ничего другого, но именно это мы сделаем на следующем экране.

Room.dart

Наконец, мы подошли к коду экрана Room, наиболее существенному фрагменту, на который стоит обратить внимание. Здесь вы также познакомитесь с парочкой новых концепций Flutter! Сначала взгляните на рис. 8-7, чтобы знать, как он выглядит.

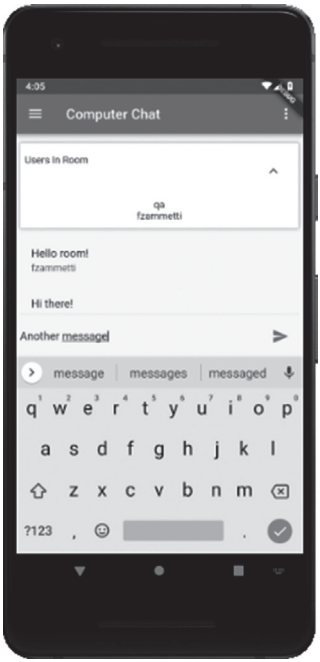


Рисунок 8-7. *Экран комнаты*

В верхней части вы видите виджет `ExpansionPanelList`. Он предоставляет список дочерних элементов, которые могут быть развернуты и свернуты по желанию пользователя.

Мы будем использовать его для отображения списка пользователей в комнате. Он должен быть расширяющимся и сворачивающимся, потому что под ним находится основная цель экрана Room – список сообщений. В самом низу находится область, в которую пользователь может ввести сообщение и отправить его в чат. Для этого используется кнопка `IconButton`, которая представляет собой просто значок. В правом верхнем углу находится трехточечное меню, или меню переполнения (overflow), как его иногда называют. Там вы найдете следующие функции: выход из комнаты, приглашение пользователя в комнату, удаление пользователя и закрытие комнаты (последние две предназначены только для администратора). Функция приглашения приведет к диалогу выбора пользователя, но мы дойдем до всего этого чуть позже.

А сначала давайте посмотрим, как все это начинается (кроме блока `imports`, конечно):

```
class Room extends StatefulWidget {
   Room({Key key}) : super(key : key);
   @override
   _Room createState() => _Room();
}

class _Room extends State {
   bool _expanded = false;
   String _postMessage;
   final ScrollController _controller = ScrollController();
   final TextEditingController _postEditingController =
      TextEditingController();
```

Это виджет с состоянием, и нам понадобится некоторое локальное состояние для сворачивания и разворачивания `ExpansionPanelList`. Конечно, я мог бы поместить его в `scoped_model`, но, как правило, если изменения касаются одного виджета, то имеет смысл сделать его виджетом с состоянием. Однако, как я уже говорил, Flutter не накладывает никаких ограничений.

Существует пара переменных уровня класса, которые определяют, будет ли список пользователей расширен (когда поле `_expanded` имеет значение `true`) или свернут (когда оно равно `false`). У нас также есть переменная `_postMessage`, которая будет содержать сообщение, отправляемое пользователем. Кроме того, у нас есть `ScrollController`, на который ссылается переменная `_controller`. Вы часто будете иметь дело с этим объектом, поскольку большинство компонентов прокрутки (scroll) содержат его по умолчанию. Тем не менее в этом приложении в нем есть конкретная потребность, о которой я расскажу позже, когда мы рассмотрим код списка сообщений. После этого, наконец, идет `TextEditingController`, который, как вы знаете, используется при работе с виджетами `TextField`, именно его мы используем для ввода сообщений пользователем.

Меню

Далее идет метод `build()`:

```
Widget build(final BuildContext inContext) {
   return ScopedModel<FlutterChatModel>(model : model,
      child : ScopedModelDescendant<FlutterChatModel>(
      builder : (BuildContext inContext, Widget inChild,
         FlutterChatModel inModel) {
         return Scaffold(resizeToAvoidBottomPadding : false,
            appBar : AppBar(title : Text(model.currentRoomName),
               actions : [
                  PopupMenuButton(
                     onSelected : (inValue) {
                        if (inValue == "invite") {
                           _inviteOrKick(inContext, "invite");
```

К этому моменту первые несколько строк должны вам уже надоесть, пока вы не увидите новую строку PopupMenuButton. PopupMenuButton – это виджет, который предоставляет *всплывающее* меню, посмотрите на рис. 8-8.

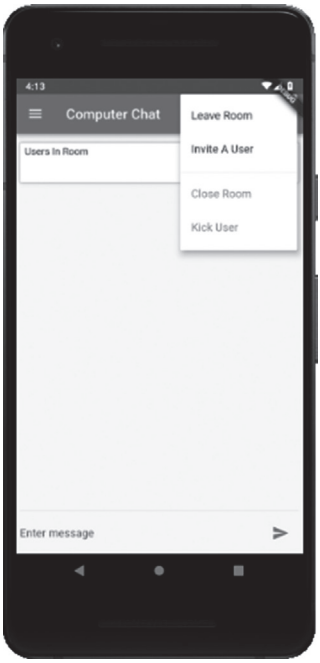


Рисунок 8-8. *Меню функций комнаты*

Хотя мы еще не добавили пункты меню – этот код скоро появится, – мы уже начали с кода, который будет выполняться при выборе в меню элемента, функции обработчика onSelected. Эта функция получает строковое значение, связанное с элементом меню, который был нажат, поэтому мы запускаем оператор if для верного исполнения действий. В случае с пунком «invite» мы вызываем метод _inviteOrKick(), который рассмотрим позже (эта функция обрабатывает как приглашение пользователя, так и его удаление из комнаты).

Затем следует ветка для пункта «leave»:

```
} else if (inValue == "leave") {
   connector.leave(model.userName, model.currentRoomName, () {
      model.removeRoomInvite(model.currentRoomName);
      model.setCurrentRoomUserList({});
      model.setCurrentRoomName(
         FlutterChatModel.DEFAULT_ROOM_NAME
      );
      model.setCurrentRoomEnabled(false);
      Navigator.of(inContext).pushNamedAndRemoveUntil("/",
         ModalRoute.withName("/")
```

```
      );
});
```

Выход из комнаты требует от нас выполнения некоторых типовых задач по очистке, начиная с удаления любых существующих приглашений в комнату. Кто-то может возразить, что если вы уйдете, то все равно сможете войти в комнату, в которую вас пригласили, но я считаю, что «эй, вы ушли – скатертью дорожка!». Список пользователей в комнате также должен быть очищен, а строка с названием комнаты должна отобразить текст по умолчанию. Связанный с AppDrawer элемент Current Room тоже должен быть недоступен, и наконец-таки мы возвращаем пользователя обратно на домашний экран.

Администратор может вообще закрыть комнату:

```
} else if (inValue == "close") {
   connector.close(model.currentRoomName, () {
      Navigator.of(inContext).pushNamedAndRemoveUntil("/",
      ModalRoute.withName("/") );
});
```

Здесь нет никакой работы, кроме как сообщить серверу, что комната закрыта, а затем перейти на домашний экран.

Также добавим ветку для удаления пользователя из комнаты:

```
} else if (inValue == "kick") {
   _inviteOrKick(inContext, "kick");
}
```

Это то же самое, что и код приглашения, поэтому давайте чуть позже посмотрим, что стоит за этой функцией. А сначала вернемся и создадим пункты меню с помощью функции itemBuilder:

```
itemBuilder : (BuildContext inPMBContext) {
   return <PopupMenuEntry<String>>[
      PopupMenuItem(value:"leave",child:Text("Leave Room")),
      PopupMenuItem(value:"invite",child:Text("Invite A User")),
      PopupMenuDivider(),
      PopupMenuItem(value : "close", child : Text("Close Room"),
         enabled : model.creatorFunctionsEnabled),
      PopupMenuItem(value : "kick", child : Text("Kick User"),
         enabled : model.creatorFunctionsEnabled)
   ];
}
```

Мы должны вернуть массив виджетов PopupMenuEntry, и каждый PopupMenu Entry в этом массиве имеет свойство value (значения, которые вы должны распознать!) и дочерний (child) виджет Text для фактического текста, который будет отображаться. Для пунктов **Close Room** (Закрыть комнату) и **Kick User** (Исключить пользователя) свойства enabled ссылаются на свойство creatorFunctionsEnabled

нашей модели (просто чтобы показать, что вы действительно можете без проблем использовать вместе локальное и глобальное состояния), чтобы определить, включены эти элементы или нет.

Содержимое главного экрана

После того как меню построено, мы продолжаем:

```
drawer : AppDrawer(),

body : Padding(padding : EdgeInsets.fromLTRB(6, 14, 6, 6),
   child : Column(
      children : [
         ExpansionPanelList(
            expansionCallback : (inIndex, inExpanded) =>
               setState(() { _expanded = !_expanded; }),
            children : [
               ExpansionPanel(isExpanded : _expanded,
                  headerBuilder : (BuildContext context,
                     bool isExpanded) => Text(" Users In Room"),
                     body : Padding(padding:EdgeInsets.fromLTRB(0,0,0,10),
                     child : Builder(builder : (inBuilderContext) {
                        List<Widget> userList = [ ];
                        for (var user in model.currentRoomUserList) {
                           userList.add(Text(user["userName"]));
                        }
                        return Column(children : userList);
                     })
                  )
               )
            ]
         )
```

Итак, после drawer появляется кое-что новенькое. Во-первых, Padding находится сверху, так что я могу контролировать расстояние между всеми элементами на экране. Я сдвигаю все на 14 пикселей ниже, чтобы убрать тень под строкой состояния (status bar), и на несколько пикселей влево, вправо и вниз только потому, что так на мой взгляд виджет body выглядит лучше, так как не примыкает к краям экрана.

Затем идет макет Column и его первый дочерний элемент ExpansionPanelList, в котором отображается список пользователей. Первое, что нам нужно сделать, – это подключить обработчик expensionCallback, который запускался бы каждый раз, когда пользователь разворачивает или сворачивает панель. Интересно, что по умолчанию ничего не произойдет, кроме отображения маленькой стрелочки, если вы сами не напишите нужный код. Как только вы это сделаете, то заметите флаг _expended, который нам нужен в первом дочернем элементе

ExpansionPanelList – ExpansionPanel со списком пользователей. Флаг _expanded становится значением свойства isExpanded. В функции headerBuilder мы создаем заголовок (header) у ExpansionPanel. Это простой виджет Text с двумя пробелами в начале, так как по умолчанию данный виджет будет автоматически прижиматься к левому краю панели, но, как вы догадываетесь, мне это не нравится! Вместо того чтобы обернуть все в Padding, который наверняка сработал бы, я просто добавляю два пробела к строке, и все будет в порядке.

В дополнение к заголовку у ExpansionPanel обычно всегда есть body, и оно также требует выравнивания. Для этого я использую Padding, чтобы обеспечить некоторое пространство под списком пользователей, по тем же причинам, что и в заголовке: без него последний пользователь в списке будет отображаться некорректно.

Теперь child-элемент Padding становится интереснее. Вы видели уже много различных функций builder, но я никогда не говорил о том, что вы можете почти всегда использовать универсальную функцию Builder(). Иногда она необходима, чтобы использовать данные, к которым нет прямого доступа (конечно, можно использовать общую модель, но можно и без нее). В этом примере на самом деле не было необходимости в Builder(), но я решил, что это будет отличное место для демонстрации. Опять же, Flutter дает вам множество вариантов решения задач.

Внутри функции Builder() находится простой цикл для прохода по списку пользователей в комнате, где каждый пользователь отображается как виджет с текстом, а сам список отображается в Column.

Далее следует список сообщений, которому кое-что предшествует:

```
Container(height : 10),
   Expanded(child : ListView.builder(controller : _controller,
      itemCount : model.currentRoomMessages.length,
      itemBuilder : (inContext, inIndex) {
         Map message = model.currentRoomMessages[inIndex];
         return ListTile(subtitle : Text(message["userName"]),
            title : Text(message["message"])
         );
      }
))
```

Container – это еще один способ добавить пространства в макет. Я задал его без содержимого, но с определенной высотой, и добавил некоторое расстояние между списком пользователей и списком сообщений, все это без специального виджета Padding. Далее идет ListView для сообщений, надеюсь, что вам уже стало ясно, как он работает, и вы представляете себе его реализацию, так как видели виджеты ListView уже несколько раз. Для каждого элемента в списке создается ListTile с title, содержащим текст сообщения, и subtitle, именем пользователя, опубликовавшего это сообщение.

Затем появляется виджет Divider и область ввода сообщения для пользователя:

```
Divider(),
Row(children : [
   Flexible(
      child : TextField(controller : _postEditingController,
      onChanged : (String inText) =>
         setState(() { _postMessage = inText; }),
      decoration : new InputDecoration.collapsed(
         hintText : "Enter message"),
   )),
   Container(margin : new EdgeInsets.fromLTRB(2, 0, 2, 0),
      child : IconButton(icon : Icon(Icons.send),
         color : Colors.blue,
         onPressed : () {
            connector.post(model.userName,
               model.currentRoomName, _postMessage, (inStatus) {
               if (inStatus == "ok") {
                  model.addMessage(model.userName, _postMessage);
                  _controller.jumpTo(
                     _controller.position.maxScrollExtent);
               }
         });
      }
   )
)
```

Во-первых, область ввода – это TextField и IconButton, стоящие рядом друг с другом, поэтому использование Row имеет смысл. Но я не хочу устанавливать точную ширину для каждого из виджетов, так как я не знаю размеров экрана. И, как всегда, во Flutter есть подходящий виджет: Flexible. Он позволяет вам контролировать то, как компоненты внутри виджета Flex, Row или Column растягиваются и заполняют доступное пространство. Здесь цель проста: позволить TextField заполнить столько места, сколько доступно, учитывая размеры соседней кнопки (IconButton). Поэтому я помещаю TextField во Flexible, а затем Flexible в Row в качестве первого дочернего элемента. Второй дочерний элемент Row – это Container, содержащий IconButton. IconButton находится внутри Padding, так что у меня будет отступ в несколько пикселей слева и справа от IconButton. Затем создается IconButton. Flutter предлагает хорошую иконку для отправки сообщения, которая вполне нам подойдет.

Когда кнопка нажата, вызывается метод connector.post() с передачей имени пользователя, имени комнаты и сообщением, которое они ввели. Предполагая, что мы получаем ответ **ok**, сообщение добавляется в список сообщений комнаты, и, наконец, используется ScrollController, который я упомянул ра-

нее. Смысл в том, что поскольку сообщение появится в нижней части `ListView`, его может быть не видно, если общее количество сообщений выходит за границы экрана. Таким образом, `_controller` предоставляет метод `jumpTo()`, которому мы передаем `_controller.position.maxScrollExtent`, для того чтобы реализовать прокрутку списка к новому сообщению.

Приглашение или исключение пользователей

Последнее, на что нужно обратить внимание, – это на приглашение пользователя в комнату или его удаление. При нажатии любого из этих пунктов меню появляется диалоговое окно, подобное показанному на рис. 8-9, с надписью «`kicked`» (выгнать) или «`invite`» (пригласить), в зависимости от ситуации.

Рисунок 8-9. *Диалог приглашения пользователя*

Итак, код начинается так:

```
_inviteOrKick(final BuildContext inContext,
   final String inInviteOrKick) {
      connector.listUsers((inUserList) {
         model.setUserList(inUserList);
```

Первое, что мы хотим сделать, – это получить от сервера обновленный список пользователей. Как и в некоторых других ситуациях, это *может* быть излишним, но лучше это сделать. Обратите внимание, что при удалении это

и правда лишнее, так как в коде уже есть обновление списка пользователей в комнате. Пока что наш код не разветвляется для обработки разных значений аргумента inInviteOrKick, поэтому обращение к серверу происходит в любом случае. Это немного неэффективно, но при условии, что наш сервер работает нормально, не придавайте этому особого значения.

Как только приходит ответ, мы можем показать диалог:

```
showDialog(context : inContext,
   builder : (BuildContext inDialogContext) {
      return ScopedModel<FlutterChatModel>(model : model,
         child : ScopedModelDescendant<FlutterChatModel>(
            builder : (BuildContext inContext, Widget inChild,
               FlutterChatModel inModel) {
               return AlertDialog(
                  title : Text("Select user to $inInviteOrKick"
               )
```

Все начинается достаточно просто, и с помощью такого кода вы можете легко задавать название диалогового окна.

Далее мы начинаем конструировать содержимое диалога:

```
content : Container(width : double.maxFinite,
   height : double.maxFinite / 2,
   child : ListView.builder(
      itemCount : inInviteOrKick == "invite" ?
         model.userList.length : model.currentRoomUserList,
         itemBuilder:(BuildContext inBuildContext, int inIndex) {
         Map user;
         if (inInviteOrKick == "invite") {
            user = model.userList[inIndex];
         } else {
            user = model.currentRoomUserList[inIndex];
         }
         if (user["userName"] == model.userName) { return Container(); }
```

Я хочу, чтобы диалоговое окно заполняло экран, поэтому использую небольшую хитрость, задав ширине (width) значение, равное double.maxFinite, а высоте (height) – double.maxFinite/2. Это фактически заставляет Flutter изменять размер окна до максимального размера, который будет занимать большую часть экрана.

Далее создается виджет ListView, содержащий список пользователей. Из какого списка мы бы ни получали данные, будь то model.userList для приглашения пользователя или model.currentRoomUserList для его исключения, нам необходимо задать его длину (length). Далее, когда мы нажимаем на текущего пользователя, его нужно удалить, но мы не можем вернуть null, так как получим exception. Поэтому вернем пустой Container.

Если это не текущий пользователь, то Container возвращается с актуальным содержимым:

```
return Container(decoration : BoxDecoration(
   borderRadius : BorderRadius.all(Radius.circular(15.0)),
   border : Border(
      bottom : BorderSide(), top : BorderSide(),
      left : BorderSide(), right : BorderSide()
   )
```

Сначала я применяю BoxDecoration, чтобы закруглить углы свойством border Radius. Так что вы можете закруглить любой или вообще все углы. Конечно, без углов это выглядит немного странно, поэтому я задаю границу с помощью свойства Border. Значения по умолчанию выглядят достаточно хорошо, поэтому используем их.

В момент написания книги я чувствовал себя немного психоделично, поэтому хотел красивых цветов! К счастью, свойство gradient класса BoxDecoration позволяет мне это сделать:

```
gradient : LinearGradient(
   begin : Alignment.topLeft, end : Alignment.bottomRight,
   stops : [ .1, .2, .3, .4, .5, .6, .7, .8, .9],
   colors : [
      Color.fromRGBO(250, 250, 0, .75),
      Color.fromRGBO(250, 220, 0, .75),
      Color.fromRGBO(250, 190, 0, .75),
      Color.fromRGBO(250, 160, 0, .75),
      Color.fromRGBO(250, 130, 0, .75),
      Color.fromRGBO(250, 110, 0, .75),
      Color.fromRGBO(250, 80, 0, .75),
      Color.fromRGBO(250, 50, 0, .75),
      Color.fromRGBO(250, 0, 0, .75)
   ]
)),
margin : EdgeInsets.only(top : 10.0),
child : ListTile(title : Text(user["userName"])
```

Есть несколько вариантов Gradient, например LinearGradient – это класс, который создает горизонтальный или вертикальный линейный градиент, а также есть RadialGradient и SweepGradient (которые вы можете изучить самостоятельно). Для этого вам нужно задать начало и конец градиента. Затем необходимо определить промежуточные точки «остановки» (stops) вдоль градиента, которые задают распределение цветов. Значения stops переходят от нуля к единице, и вы можете разделить их по своему усмотрению. Здесь я хочу, чтобы каждый цвет занимал равное пространство, поэтому делаю остановки каждую десятую часть. Далее определяются сами цвета (colors). Есть несколько

способов определить их с помощью Flutter, и вы уже видели коллекцию `Colors`, но здесь я хотел быть более точным, поэтому использовал значения RGB (Red-Green-Blue, красный – зеленый – синий). Технически это RGBO, где O – непрозрачность (opacity), которая установлена на 0.75, что делает все цвета непрозрачными на 75 % (или прозрачными на 25 %, если так проще). Таким образом, полупрозрачный цвет немного сливается с фоном. Это чуть притупляет цвета, так как задний фон является белым. Дальше я добавляю отступ, чтобы между первым указанным пользователем и текстом заголовка было пространство, а затем для каждого пользователя создается `ListTile`.

Наконец, нам нужно реализовать то, что происходит, когда пользователь нажимает на элемент списка:

```
onTap : () {
   if (inInviteOrKick == "invite") {
      connector.invite(user["userName"],
         model.currentRoomName, model.userName, () {
            Navigator.of(inContext).pop();
      });
   } else {
      connector.kick(user["userName"],model.currentRoomName,() {
         Navigator.of(inContext).pop();
      });
   }
}
```

Это легко: если мы приглашаем пользователя, то вызываем `connect.invite()` и передаем ему имя выбранного пользователя, имя текущей комнаты и имя приглашающего пользователя, чтобы его можно было отобразить приглашенному человеку. Или если мы выгоняем его, то вызывается метод `connector.kick()`, в который передается `userName` выбранного пользователя и название комнаты, из которой его выгнали. И в обоих случаях диалог закрывается.

Итого

В этой главе мы рассмотрели `FlutterChat`, создав клиентское приложение на основе Flutter. В нем вы увидели новые возможности фреймворка (или то, что мы раньше не использовали в реальном приложении), такие как виджеты `StatefulWidget`, `PopupMenuButton`, `ExpansionPanel`, реальное использование `GridView`, компоненты `Slider` и `Switch` и, конечно же, создание соединений с помощью socket.io и WebSocket. В качестве бонуса вы немного поиграли с Node и написали сервер!

В следующей главе мы поработаем над последним из трех наших приложений, оно откроет вам совершенно новое направление и даст несколько иной взгляд на Flutter – мы создадим игру!

ГЛАВА 9

FLUTTERHERO: ИГРА FLUTTER

«Одна работа, никакого безделья, бедняга Джек не знает веселья» (англ. «All work and no play makes Jack a dull boy», цитата из фильма «*Сияние*» 1980 года). К счастью, вам удастся избежать судьбы Джека.

В этой книге вы смотрели на Flutter сквозь призму написания полезного кода и приложений. Но это не все, что вы можете с ним сделать. Вы можете придумать что-нибудь более легкомысленное, веселое, скажем, написать игру!

Игры считаются отличными проектами для любого разработчика, потому что они затрагивают очень много различных дисциплин в программировании, от графики и звука до искусственного интеллекта, структур данных, алгоритмической эффективности и т. д. Как ведущего архитектора разработчики иногда спрашивают меня, как бы им отточить свои навыки. У меня всегда один ответ: написать игру! Я не верю, что какой-либо другой проект предоставляет такой же уровень неординарности, креативности и возможности обучения.

А самое главное – игры весело писать!

В этой главе вы будете использовать Flutter, чтобы написать игру. Преимущество данной книги состоит в том, что вы встретите несколько новых концепций Flutter. В конце концов, вы будете учиться и, надеюсь, веселиться!

Итак, давайте выясним, какую игру мы придумаем и создадим. Что нужно каждой великой игре? История!

История такова

Жители Горгоны 6 представляют собой космическое противоречие: технологически развитая цивилизация, но отстающая интеллектуально!

Они могут строить быстрые, гладкие, но очень хрупкие космические корабли! Достаточно просто столкнуться с рыбой-космонавтом (и, будучи мирной расой, горгонианцы не разрабатывают никакого оружия).

Да, я сказал рыба-космонавт! Но я отвлекся.

Для горгонианцев это настоящая проблема, потому что их звездная система поражена паразитами: она кишит космическими зубастиками и разного рода опасностями! У них есть гигантская космическая рыба с третьей луны Вальтракса, разумные существа-машины с протопланеты 10101110, космические пришельцы (но у кого в Солнечной системе их нет?) и астероиды, летающие вокруг. Всё это блокирует работу космических путей сообщения и прогулочных круизов (при всей хрупкости их кораблей, круизы можно считать еще одним противоречием горгонианцев, учитывая множество разрушенных кусочков кораблей вокруг).

К счастью, есть решение этих проблем: на окраине Солнечной системы находится массивный кристалл неизвестного происхождения, который излучает особый тип энергии, уничтожающий космических вредителей, по крайней мере на некоторое время. Горгонианцы придумали, как собирать эту энергию. Таким образом, они посылают космические грузовые корабли (и, будучи горгонианскими кораблями, они выглядят круто!), чтобы собирать энергию и возвращать ее на родину.

Ваша работа как одного из смельчаков – может быть, даже *героев*-пилотов «флота хрустальных кораблей» – пробраться сквозь космических паразитов, чтобы извлечь энергию из кристалла и затем вернуть ее домой. Когда вы соберете достаточно энергии, паразиты будут уничтожены, и вы станете героем Горгоны!

По крайней мере, на некоторое время.

Конечно, вы получите за это несколько очков – давайте назовем их космическими кредитами, – с помощью которых вы сможете сохранить свою привычку лизать галлюциногенных ящериц-горгонианцев.

Именно так, друзья, вы и продумываете простую игру, которую можно кодировать с помощью Flutter. Я сразу скажу, что если вы ожидаете геймплея уровня Apex Legends, Halo или Red Dead Redemption, вы будете сильно разочарованы. Не ждите от игры класса AAA. Вряд ли вы захотите играть в нее постоянно, но, я надеюсь, вы ее полюбите! И она станет хорошим опытом, к которому я вас веду.

Итак, с историей мы разобрались, давайте приступим к работе, потому что паразиты сами себя не уничтожат (однако они могут быть такими же глупыми, как горгонианцы).

Базовая компоновка

Итак, как выглядит эта игра? Что ж, если вы когда-нибудь играли в старую 8-битную игру, скажем, с лягушкой, которая прыгает по дощечкам различного типа, чтобы добраться до цели на другой стороне, то наша игра будет вам понятна. Рисунок 9-1 показывает, как она выглядит.

Рисунок 9-1. *Может, мне стоило назвать игру Space Frogger?*

Ваш корабль располагается в нижней части экрана, недалеко от вашего дома. Чтобы управлять им, поместите палец в любом месте экрана, оно станет точкой привязки, или нулевой позицией виртуального джойстика. Теперь просто двигайте пальцем в любом из восьми направлений компаса, и ваш корабль будет двигаться за вами. Ваша цель – пройти через полосы паразитов (астероиды, инопланетяне, разумные машины и космические рыбы). Когда вы достигнете вершины, то дотронетесь до кристалла, и полоса энергии сверху заполнится. Затем вы возвращаетесь через паразитов, касаетесь вашего дома, энергия передается, и в этот момент все паразиты взрываются, а вы получаете несколько очков. Затем паразиты возвращаются, и все повторяется снова. Конечно, вы взорветесь, если коснетесь чего-либо, кроме кристалла или вашего дома.

Как я уже сказал, это несложная игра, и она сделана с помощью Flutter, так что миссия выполнима.

Структура каталога и исходные файлы компонентов

Начнем с обсуждения структуры каталогов и, что более важно, некоторых файлов, которые в нем содержатся. Все это показано на рис. 9-2. Это совершенно стандартная структура приложения Flutter, которую вы узнали и, я надеюсь, полюбили. В каталоге ресурсов вы найдете кучу изображений и несколько аудиофайлов. Названия файлов должны выражать их суть, но цифры требуют некоторого объяснения.

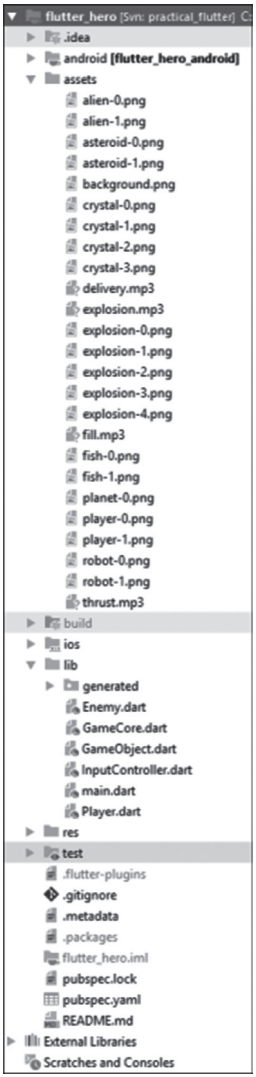


Рисунок 9-2. *Структура каталога приложения и составляющие файлы источника/ресурса*

Как вы скоро увидите, каждое из изображений представляет собой часть объекта в игре, который будет использоваться общим классом `GameObject`. Например, есть `GameObject` для корабля игрока, который использует изображения `player-0.png` и `player-l.png`, и `GameObject` для кристалла. Этот общий класс включает логику для анимации объектов. Анимация в этом контексте похожа на старый трюк, как в детстве, когда вы берете блокнот, рисуете серию «кадров», возможно, фигуру прыгающей палки; а затем быстро переворачиваете страницы, чтобы анимировать рисунок. В нашем примере каждый кадр ани-

мации обозначается номером в имени файла. Итак, для кристалла есть четыре кадра анимации (так сказать, в вашем блокноте четыре страницы): `crystal-0.png`, `crystal-l.png`, `crystal-2.png` и `crystal-3.png`, а класс `GameObject` знает, как «переворачивать страницы».

Примечание. Планета не анимирована, поэтому для нее есть только один кадр, однако этот объект также обернут в `GameObject`. Для этого, как вы увидите позже, необходимо использовать ту же схему именования и назвать изображение `planet-0.png`, чтобы `GameObject` мог корректно работать с ним.

MP3-файлы озвучивают различные события, и, надеюсь, их названия объяснят вам их предназначение, но если нет, то дальнейшее использование поможет вам разобраться.

Кроме того, у вас есть небольшие полезные файлы исходных кодов в папке `lib`, дополняющие `main.dart`, и мы дойдем до каждого из них позже, хотя, как и в случае с ресурсами, их названия подскажут вам их предназначение.

Но прежде всего давайте немного поговорим о `pubspec.yaml`.

Конфигурация: pubspec.yaml

Файл `pubspec.yaml` примерно на 99% такой же, как и все остальные, которые вы видели ранее, за исключением одного нового элемента:

```
name: flutter_hero
description: FlutterHero
version: 1.0.0+1
environment:
   sdk: ">=2.1.0 <3.0.0"
dependencies:
   flutter:
      sdk: flutter
   cupertino_icons: ^0.1.2
   audioplayers: 0.11.0
dev_dependencies:
   flutter_test:
      sdk: flutter
flutter:
   uses-material-design: true
   assets:
      - assets/
```

В этой игре, как и в большинстве других, есть звуковое сопровождение! На момент написания этой книги Flutter не имел подходящего для нашей игры API работы с аудио, который бы позволял воспроизводить произвольные звуковые файлы по мере необходимости, а иногда и одновременно. Поэтому мы используем плагин. К счастью, есть несколько вариантов, но самый популярный –

это плагин audioplayers (https://pub.dartlang.org/packages/audioplayers). Audioplayers – версия более раннего плагина с именем audioplayer, которая расширяет функциональность старого. Этот плагин позволяет нам воспроизводить аудиофайлы, хранящиеся удаленно в интернете, локально на устройстве пользователя, или, что очень важно для нас, в качестве ресурсов нашего проекта. С помощью этого плагина вы можете воспроизводить файлы, управлять их воспроизведением (пауза, остановка, выбор определенного момента аудио), зацикливать звук и отслеживать события во время воспроизведения, чтобы вы могли, например, показать индикатор прогресса.

В приложении FlutterHero мы не собираемся работать со всем набором этих функций! Нам просто нужно уметь воспроизводить звуки во время определенных событий. Мы рассмотрим API для этого плагина, когда увидим первое использование звука – он довольно элементарен.

Помимо плагина audioplayers, в конфигурации также указан каталог ресурсов (assets), который содержит все те файлы, которые вы видели ранее, как изображения, так и аудио. Я хотел разделить их на assets/images и assets/audio, но, чтобы audioplayers мог их найти, потребовалось бы больше работы. Поэтому, учитывая небольшое количество необходимых ресурсов, я просто скинул все в единый каталог.

Класс GameObject

Теперь переходим к коду! Обычно я начинаю с main.dart, но здесь я расскажу вам о классе GameObject. Возможно, вы не сразу все поймете, но это изменится, когда вы увидите, как используется этот класс и его подклассы.

GameObject включает два дочерних подкласса, как показано на рис. 9-3.

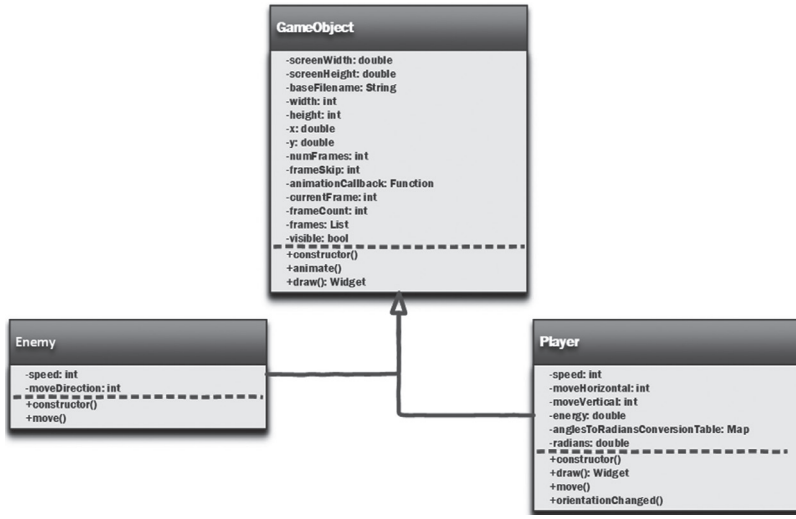


Рисунок 9-3. *Иерархия классов: GameObject и его дочерние элементы*

Основная идея – это простое объектно-ориентированное программирование: класс GameObject содержит данные и функциональность, общие для всех объектов в игре, а затем подклассы расширяют его при необходимости. Например, каждому игровому объекту (игрок, кристалл, паразит и планета) нужны такие данные, как:

- ширина и высота экрана;
- базовое имя файла для их изображений (например, «planet-*.png» или «crystal- *.png», где * будут номерами кадров);
- ширина и высота объекта;
- X и Y объекта на экране;
- общее количество кадров в цикле анимации; сколько игровых фреймов пропустить при анимации (я знаю, это немного сбивает с толку, но не волнуйтесь, это ненадолго!); текущий кадр анимации; счетчик для определения того, когда пора переходить к следующему кадру; функция, которая будет вызываться после завершения цикла анимации; и, конечно же, все кадры;
- необходимость показать или скрыть объект.

Каждый игровой объект включает следующие общие функции:

- конструктор для его настройки;
- метод для его анимации;
- способ нарисовать его на экране.

Но у подкласса Enemy, который будет представлять рыбу, роботов, инопланетян и астероиды, есть дополнительные поля:

- скорость перемещения по экрану;
- направление движения (влево или вправо).

Класс Player, очевидно, представляет корабль игрока, и у него тоже есть специфические поля, помимо поставляемых GameObject:

- скорость движения;
- направление: влево или вправо (горизонтальное движение), вверх или вниз (вертикальное движение);
- количество энергии кристалла на борту;
- на сколько градусов (в радианах) он повернут (позволяет нам использовать только одно изображение и менять его ориентацию и направление движения), а также некоторые таблицы данных, которые избавляют нас от необходимости повторять математические операции, что, в свою оче-

редь, повышает производительность (никогда не забывайте о производительности в играх!);

- метод, используемый при изменении ориентации судна (в зависимости от направления движения), чтобы его можно было корректно вращать.

Итак, теперь, когда у вас есть общее представление об этих классах, давайте перейдем к коду `GameObject`:

```
class GameObject {
   double screenWidth = 0.0;
   double screenHeight =0.0;
```

Он начинается как обычный класс, не имеющий предка. Наши первые два свойства данных: ширина (width) и высота (height) экрана. Как вы увидите позже, Flutter предоставляет API для получения этой информации, которая извлекается во время запуска приложения и передается любому экземпляру `GameObject` при создании. Это позволяет избежать необходимости вызывать API снова и снова, а так как эта информация необходима различным игровым объектам, то мы сохраним ее.

```
String baseFilename = "";
```

`baseFilename` – это часть имени графического файла, которая не изменяется. Проще говоря, это название объекта, будь то рыба (fish), игрок (player), планета (planet) или что-то еще.

```
int width = 0;
int height = 0;
```

Ширина и высота объекта тоже необходимы. Мы могли бы найти эту информацию с помощью механизмов Flutter для работы с картинками, но проще предоставить размеры в коде, тогда она никогда не изменится.

```
double x = 0.0;
double y = 0.0;
```

Горизонтальное (x) и вертикальное (y) расположения тоже понадобятся каждому игровому объекту на экране.

```
int numFrames = 0;
int frameSkip = 0;
int currentFrame = 0;
int frameCount = 0;
List frames = [ ];
Function animationCallback;
```

Все эти шесть свойств связаны с анимацией. Свойство `numFrames` – это общее количество кадров. Свойство `frameSkip` определяет, сколько итераций (или тиков, `ticks`) основного игрового цикла должно пройти до показа следующего

кадра анимации. Свойство currentFrame указывает, какой кадр анимации отображается в данный момент. Свойство frameCount увеличивается с каждым тиком основного игрового цикла, а когда оно достигает значения frameSkip, значение currentFrame увеличивается. Свойство frames (кадры) представляет собой список изображений для игрового объекта, по одному на кадр анимации. Наконец, свойство animationCallback – это ссылка на функцию, которая будет вызываться каждый раз, когда заканчивается цикл анимации. Скоро вы поймете, почему это необходимо, а пока давайте продолжим.

```
bool visible = true;
```

Паразит и игрок должны быть скрыты в определенное время, а свойство visible (видимый) определяет, когда игровой объект виден или нет.

Переходим к конструктору:

```
GameObject(double inScreenWidth, double inScreenHeight,
   String inBaseFilename, int inWidth, int inHeight,
   int inNumFrames, int inFrameSkip,
   Function inAnimationCallback) {
   screenWidth = inScreenWidth;
   screenHeight = inScreenHeight;
   baseFilename = inBaseFilename;
   width = inWidth;
   height = inHeight;
   numFrames = inNumFrames;
   frameSkip = inFrameSkip;
   animationCallback = inAnimationCallback;
   for (int i = 0; i < inNumFrames; i++) {
      frames.add(Image.asset("assets/$baseFilename-$i.png"));
   }
}
```

Довольно просто, да? Все входящие аргументы сохраняются в соответствующих свойствах, а затем нам нужно загрузить кадры анимации. Здесь вы можете увидеть, как каждый виджет Image, созданный с помощью конструктора asset(), использует baseFilename для генерации имени. Мы делаем так ради производительности: однократная загрузка изображений – хорошая идея. Flutter достаточно умен, чтобы кешировать картинки, если мы загружаем их дважды, но, думаю, лучше спроектировать наше приложение так, чтобы они загружались при создании GameObject, а не во время анимации.

К слову о коде анимации:

```
void animate() {
   frameCount = frameCount + 1;
   if (frameCount > frameSkip) {
      frameCount = 0;
      currentFrame = currentFrame + 1;
```

```
      if (currentFrame == numFrames) {
         currentFrame = 0;
         if (animationCallback != null) { animationCallback(); }
      }
   }
}
```

Все, что здесь делается, – это вызов анимации на каждый тик, в нашем случае таких тиков 60. Когда значение текущего кадра достигает значения `frameSkip`, мы увеличиваем его до следующего кадра. Когда кадры кончаются, мы переходим обратно к первому кадру и вызываем `animationCallback`, если он указан.

Помимо анимирования, `GameObject` также должен знать, как отрисовать себя на экране. Поскольку все во Flutter является виджетом, как вы уже хорошо знаете, смысл в том, чтобы получить с помощью метода `build()` правильный виджет для включения его в общее дерево. В этом нам поможет метод `draw()`:

```
Widget draw() {
   return visible ?
      Positioned(left : x, top : y, child : frames[currentFrame])
         : Positioned(child : Container());
}
```

Не хочу забегать вперед, но мы будем использовать виджет `Stack` в качестве родительского для всех наших игровых объектов. Это связано с тем, что в `Stack` вы можете использовать виджеты `Positioned`, которые можно полностью разместить внутри `Stack`. Если `Stack` покрывает весь экран, то фактически мы получим канву (canvas, холст для рисования), идеально подходящую для разработки игр, поскольку сможем контролировать точное расположение объектов на экране вплоть до уровня пикселей. Именно это мы и собираемся сделать, поэтому метод `draw()` должен вернуть виджет `Positioned`, который содержит виджет `Image`, связанный с текущим кадром анимации объекта. Кроме того, объект может быть скрыт. Ваш способ сделать это может показаться Flutter немного странным. Это не `hidden = true` или что-то в этом роде, как во многих других платформах, здесь нет возможности спрятать элемент, не удаляя его из дерева виджетов. Поэтому мы и удалим его! Однако, как вы скоро увидите, возврат `null` не сработает, потому что это сломает дерево виджетов. Так что вместо `null` мы вернем пустой `Container`. Это сработает, хоть это и немного странно.

Расширение GameObject: класс Enemy

С базовыми знаниями `GameObject` мы можем рассмотреть подклассы, начнем с `Enemy`. Главное, что отличает `Enemy` от обычного `GameObject`, – это то, что `Enemy` может двигаться.

```
class Enemy extends GameObject {
```

```
int speed = 0;
int moveDirection = 0;
```

Перемещения противника (enemy) очень просты: он движется либо влево, либо вправо с заданной скоростью (speed), где скорость означает, на сколько пикселей он перемещается за тик основного игрового цикла. Значение moveDirection будет 0 для левого или 1 для правого.

Итак, у нас есть конструктор:

```
Enemy(double inScreenWidth, double inScreenHeight,
   String inBaseFilename, int inWidth, int inHeight,
   int inNumFrames, int inFrameSkip, int inMoveDirection,
   int inSpeed) :
      super(inScreenWidth, inScreenHeight, inBaseFilename,
      inWidth, inHeight, inNumFrames, inFrameSkip, null) {
   speed = inSpeed;
   moveDirection = inMoveDirection;
}
```

Поскольку этот класс расширяет GameObject, то он поддерживает все те же свойства, поэтому их необходимо задать. Здесь в игру вступает вызов super(). Как видите, сигнатура конструктора Enemy включает в себя все, что делает конструктор GameObject, плюс элементы, специфичные для Enemy. Сначала вызов super() задает свойства, общие для GameObject, затем код внутри конструктора Enemy устанавливает дополнительные свойства, специфичные для Enemy.

Со всеми этими данными мы можем реализовать метод move():

```
void move() {
   if (moveDirection == 1) {
      x = x + speed;
      if (x > screenWidth + width) { x = -width.toDouble(); }
   } else {
      x = x - speed;
      if (x < -width) { x = screenWidth + width.toDouble(); }
   }
}
```

Теперь вы понимаете, почему нужны ширина и высота экрана. Именно так мы узнаем обо всех перемещениях врага. Еще они помогают нам задать его новое местоположение. Итак, в нашей концепции вражеская рыба движется прямо по экрану. Когда она находится за правым краем экрана, значение «x» устанавливается в отрицательное значение, что помещает его в левую часть экрана. Затем она продолжает двигаться как прежде. При перемещении влево происходит то же самое, но в обратном направлении. Это все, что нужно для перемещения врагов.

Расширение GameObject: класс Player

Другой класс, который происходит из GameObject, предназначен для игрока:

```
class Player extends GameObject {
   int speed = 0;
   int moveHorizontal = 0;
   int moveVertical = 0;
```

Как и враг, игрок может двигаться, поэтому нам нужно знать, как быстро он двигается (speed) и в каком направлении (moveHorizontal и moveVertical). В отличие от вражеских паразитов, игрок может двигаться вверх, вниз, влево и вправо, а также влево вверх, влево вниз, вправо вверх и вправо вниз. Следовательно, нам нужно использовать две переменные для отслеживания направления движения. Но игрок также может не двигаться, поэтому у каждого направления есть три возможных значения вместо двух, как для врагов: 0 для обоих означает при отсутствии движения, в то время как для moveHorizontal «-1» означает влево, а «1» – вправо; moveVertical «-1» означает вверх, а «1» – вниз.

```
double energy = 0.0;
```

Игрок также может в любой момент времени получить часть энергии кристалла с борта. Поэтому нам нужна переменная, чтобы отслеживать и это.

```
Map anglesToRadiansConversionTable = {
   "angle45" : 0.7853981633974483,
   "angle90" : 1.5707963267948966,
   "angle135" : 2.3387411976724017,
   "angle180" : 3.141592653589793,
   "angle225" : 3.9269908169872414,
   "angle270" : 4.71238898038469,
   "angle315" : 5.497787143782138
};
double radians = 0.0;
```

В разработке игр вы почти всегда ищете маленькие хитрости для оптимизации, экономии памяти или циклов. В этом случае есть два трюка с кораблем игрока. Во-первых, корабль всегда должен быть направлен в сторону движения. Таким образом, нам понадобилось бы восемь разных изображений: по одному для каждого при движении вверх, вниз, влево, вправо, вверх влево, вверх вправо, вниз влево и вниз вправо. Но поскольку корабль анимирован и при условии что каждое направление использует по два кадра, мы бы использовали 16 разных изображений! Выглядит неэффективно. Поэтому будет только два изображения, по одному на каждый кадр. Чтобы обеспечить восемь различных направлений, они будут вращаться в реальном времени до нужной ориентации. Flutter предоставляет несколько способов поворота изображения, и вскоре мы узнаем, как на самом деле лучше это сделать. Но прежде обсудим второй трюк,

который поможет нам оптимизировать приложение. Чтобы повернуть корабль, нам нужно сообщить Flutter, на сколько его повернуть. Это значение указывается в радианах. Но мы-то хотим повернуть его на несколько градусов! Мы могли бы каждый раз переводить градусы в радианы, но постараемся этого избежать. Самый простой способ заключается в предварительном расчете радиана для каждого угла в градусах, на который мы хотим повернуть, и сохранении этих значений на карте для удобства поиска. Для этого мы используем свойство `anglesToRadiansConversionTable`. Поскольку нам нужно отследить фактическое количество радиан, на которые объект поворачивается, мы используем свойство `radians`. Вы скоро увидите в использовании каждое из них.

Структура будет следующей:

```
Player(double inScreenWidth, double nScreenHeight,
   String inBaseFilename, int inWidth, int inHeight,
   int inNumFrames, int inFrameSkip, int inSpeed,
) : super(inScreenWidth, inScreenHeight, inBaseFilename,
   inWidth, inHeight, inNumFrames, inFrameSkip, null) {
   speed = inSpeed;
}
```

Так как `Player` наследуется от `GameObject`, сначала мы вызываем конструктор `GameObject`, а затем устанавливаем скорость, единственное отличительное значение `Player`, которое необходимо задать во время создания. Обратите внимание на `null` в качестве последнего аргумента конструктора `GameObject` – это callback-функция анимации, которая не нужна игроку, поэтому передается `null`.

Теперь `GameObject` предоставляет метод `draw()`, но для игрока сам процесс рисования немного отличается, поэтому нам нужно переопределить этот метод:

```
@override
Widget draw() {
   return visible ?
   Positioned(left : x, top : y, child : Transform.rotate(
      angle : radians, child : frames[currentFrame]))
   : Positioned(child : Container());
}
```

Разница здесь только во вращении. Для этого мы оборачиваем виджет `Image` в виджет `Transform`, который Flutter предоставляет для применения преобразования к дочернему элементу. Хотя использование самого `Transform` требует от вас предоставления матрицы преобразования, которая может быть математически сложной в зависимости от того, чего вы пытаетесь достичь, этот класс предоставляет несколько конструкторов для наиболее распространенных преобразований. К ним относятся `Transform.scale()` для масштабирования дочернего элемента (увеличения или уменьшения его размера), `Transform`.

translate() для его смещения и Transform.rotate() для поворота дочернего элемента вокруг своей оси. Как видите, этот конструктор требует угла поворота в радианах, поэтому здесь вы можете увидеть, как используется свойство radians. Установка значения происходит в методе orientationChanged(), с которым вы познакомитесь позже, и вызывается из кода, который обрабатывает пользовательский ввод:

```
void orientationChanged() {
   radians = 0.0;
   if (moveHorizontal == 1 && moveVertical == -1) {
      radians = anglesToRadiansConversionTable["angle45"];
   } else if (moveHorizontal == 1 && moveVertical == 0) {
      radians = anglesToRadiansConversionTable["angle90"];
   } else if (moveHorizontal == 1 && moveVertical == 1) {
      radians = anglesToRadiansConversionTable["angle135"];
   } else if (moveHorizontal == 0 && moveVertical == 1) {
      radians = anglesToRadiansConversionTable["angle180"];
   } else if (moveHorizontal == -1 && moveVertical == 1) {
      radians = anglesToRadiansConversionTable["angle225"];
   } else if (moveHorizontal == -1 && moveVertical == 0) {
      radians = anglesToRadiansConversionTable["angle270"];
   } else if (moveHorizontal == -1 && moveVertical == -1) {
   radians = anglesToRadiansConversionTable["angle315"];
   }
}
```

Проверка выполняется для каждого из четырех основных направлений, плюс четыре комбинации, чтобы определить, в каком направлении движется игрок. Затем выполняется поиск в anglesToRadiansConversionTable, и получившиеся радианы сохраняются в свойстве radians. Хоть это и не хитроумный код, но он отлично справляется со своей задачей и избегает дорогостоящей математической операции.

Совет. На практике это не будет таким уж дорогостоящим, но в играх всегда лучше думать об оптимизации во время написания кода. Это справедливо для всех видов программирования, но в особенности для игр, где каждая итерация выполняется в основном игровом цикле, о чем мы вскоре поговорим. Конечно, вы должны избегать слишком дорогостоящих операций и думать об оптимизации заблаговременно. Но на практике подобная таблица довольно распространена.

Последний метод для перемещения игрока:

```
void move() {
   if (x > 0 && moveHorizontal == -1) { x = x - speed; }
   if (x < (screenWidth - width) && moveHorizontal == 1) {
      x = x + speed;
   }
```

```
   if (y > 40 && moveVertical == -1) { y = y - speed; }
   if (y < (screenHeight - height - 10) && moveVertical == 1) {
      y = y + speed;
   }
}
```

Это код будет вызываться один раз за тик основного игрового цикла. Мы задаем горизонтальное направление движения, а затем вертикальное. Напомню, что существует восемь возможных направлений движения игрока. Помимо четырех очевидных, существует еще столько же промежуточных, которые нам также необходимо обработать. Когда сработает один из операторов `if`, имеющих дело с «x», и один из операторов, имеющих дело с «y», это даст комбинацию вертикального и горизонтального движений. Конечно, мы должны убедиться, что игрок не может попасть за пределы экрана, этим занимается проверка границ в каждом из операторов `if`. Она учитывает как сторону игрока, так и пространство для счета и индикатор энергии.

Место, где все начинается: main.dart

Как всегда, наше приложение запускается в исходном файле `main.dart`:

```
import "package:flutter/material.dart";
import "package:flutter/services.dart";
import "InputController.dart" as InputController;
import "GameCore.dart";

void main() => runApp(FlutterHero());

class FlutterHero extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
      SystemChrome.setEnabledSystemUIOverlays(
         [SystemUiOverlay.bottom]
      );
      return MaterialApp(
         title : "FlutterHero", home : GameScreen()
      );
   }
}

class GameScreen extends StatefulWidget {
   @override
   GameScreenState createState() => new GameScreenState();
}
```

Модуль `services.dart` – это что-то новенькое. Он предоставляет нам доступ к возможностям смартфона для таких задач, как взаимодействие с буфером об-

мена, тактильной обратной связи (вибрация устройства), воспроизведение системных звуков, выделение текста, и это лишь некоторые из них. Также он позволяет нам контролировать «хромированную рамку» (chrome) нашего приложения, а именно строку состояния системы сверху (status bar) и кнопки навигации (navigation bar) на Android. Если вы перейдете к классу верхнего уровня `FlutterHero`, в методе `build()` вы увидите вызов `SystemChrome.setEnabledSystemUIOverlays()`. Это предоставляется сервисным модулем, и этот метод позволяет нам передать ему массив хромированных идентификаторов для включения. Здесь я специально включаю элемент `SystemUiOverlay.bottom`, который представляет собой программные кнопки навигации в Android. Так как это все, что находится в массиве, строка состояния в верхней части будет скрыта, предоставляя нашей игре (почти) полноэкранный режим.

Конечно, перед этим вы могли заметить, что мы создаем в `GameScreen` виджет с состоянием (stateful), который является домашним экраном, определенным в `MaterialApp` класса `FlutterHero`.

Как и следовало ожидать (учитывая, что `GameScreen` – это виджет с состоянием), он будет связан с классом состояния `GameScreenState`:

```
class GameScreenState extends State with
   TickerProviderStateMixin {
   @override
   Widget build(BuildContext inContext) {
      if (gameLoopController == null) {
         firstTimeInitialization(inContext, this);
      }
```

Мы не собираемся использовать `scoped_model` в этом приложении, а просто возьмем базовые возможности `State`, предоставленные Flutter. Но, помимо `State`, в этом классе есть еще кое-что новое: `TickerProviderStateMixin`. Мы рассмотрим все это в следующем разделе, но я заранее скажу, что оно связано с основным игровым циклом, который будет выполняться по 60 раз в секунду на протяжении всей игры.

Метод `build()` начинается с проверки `gameLoopController`, который является частью файла `GameCore.dart`. Я пока пропущу его описание, но если `gameLoopController` равен `null`, то производится вызов `firstTimeInitialization()`, с передачей ему в качестве параметров `BuildContext`, а также ссылки на сам класс `GameScreenState`. Эту функцию мы тоже рассмотрим позже, но по названию можно догадаться, что она выполняет задачи, которые происходят при первом запуске метода `build()` (мы встретим `build()` в коде игры еще не раз). Проблема в том, что есть задачи, которые необходимо выполнить для настройки игры. Они возникают только при наличии `BuildContext`. Так что эти задачи должны выполняться в методе `build()`. Однако они должны произойти только один раз, поэтому и выполняется проверка `gameLoopController`.

Но, как я уже сказал, мы рассмотрим все это в следующем разделе!

Продолжая изучать метод `build()`, давайте построим наше дерево виджетов:

```
List<Widget> stackChildren = [
   Positioned(left : 0, top : 0,
      child : Container(width : screenWidth, height : screenHeight,
      decoration : BoxDecoration(image : DecorationImage(
         image : AssetImage("assets/background.png"),
         fit : BoxFit.cover
      ))
   )
),
```

Интересно, что во всех предыдущих методах `build()`, которые вы видели, вы почти сразу могли заметить оператор return для возврата виджета, и все дочерние виджеты были определены как «встроенные» в него. А здесь мы сначала создаем список (list). Ситуация такова, что существует дополнительная логика, которая должна выполняться при создании дерева виджетов (например, применение циклов и прочее), которая не может быть выполнена при создании одного монолитного дерева виджетов, как мы обычно это делаем.

Поскольку виджетом, который мы хотим получить, является `Stack`, и он принимает `List` в качестве значения `children`, то мы можем реализовать всю логику генерации за пределами конструктора этого виджета – сначала мы создаем список виджетов, а затем просто передаем ссылку на него в конструктор `Stack`. Именно это здесь и происходит.

Первый элемент в списке – это `Positioned` внутри `Container`, который использует `BoxDecoration` для фонового изображения. Значение `fit`, равное `BoxFit.cover`, гарантирует заполнение (cover) экрана независимо от его фактических размеров. Ширина и высота `Container` – это значения переменных `screenWidth` и `screenHeight` соответственно. Как вы скоро узнаете, эти значения запрашиваются у операционной системы во время первичной инициализации, так что где бы эта информация ни потребовалась, она будет доступна без повторных запросов.

```
Positioned(left : 4, top : 2,
   child : Text(‘Score: ${score.toString().padLeft(4, "0")}’,
      style : TextStyle(color : Colors.white,
         fontSize : 18, fontWeight : FontWeight.w900)
   )
),
```

Следующий `Positioned` содержит `Text`, это наш игровой счет (score). Как видите, этот виджет расположен в точке с координатами `left` (соответствует `x`) и `top` (соответствует `y`), равными 4 и 2. В этом весь смысл использования `Stack`: мы можем расположить наши элементы как захотим. `Stack` автоматически будет заполнять свой родительский элемент, в данном случае экран, поэтому мы можем отобразить его во весь экран. Удобно для игры, как думаете?!

Следом идет еще один `Positioned`, на этот раз с `LinearProgressIndicator` для индикатора энергии:

```
Positioned(left : 120, top : 2, width : screenWidth - 124, height : 22,
   child : LinearProgressIndicator(
      value : player.energy,
      backgroundColor : Colors.white,
      valueColor : AlwaysStoppedAnimation(Colors.red)
   )
),
crystal.draw()
];
```

Настройка свойства `valueColor` здесь важна, потому что по умолчанию индикатор прогресса в Flutter отображается с анимацией выполнения. Он будет вращаться, если он циклический, или заполняться, если линейный. Но нам это не нужно. Нам нужен индикатор, который постепенно заполняется при контакте корабля с кристаллом и постепенно опустошается при контакте с планетой, чтобы указывать на поступление или отток энергии на корабль и обратно. Все под контролем нашего кода, а не Flutter (эй, Flutter, это моя игра, делаю, что хочу!). Таким образом, чтобы указать на то, какой цвет должен заполнять индикатор, достаточно задать только один цвет, для этого мы используем экземпляр виджета `AlwaysStoppedAnimation`. Это специальный виджет, с которым работают классы индикаторов прогресса и который обеспечивает необходимое нам поведение! Конечно, цвет заполненной части – это важная информация, поэтому она передается в конструктор `AlwaysStoppedAnimation`. Обратите внимание на то, что ширина `Positioned`, в котором находится `LinearProgress Indicator`, устанавливается динамически, используя ширину экрана минус пространство, занимаемое полем `score` (текущий счет).

Кристалл также добавляется здесь. Он является последним элементом в списке.

Затем мы добавляем наших вражеских паразитов:

```
for (int i = 0; i < 3; i++) {
   stackChildren.add(fish[i].draw());
   stackChildren.add(robots[i].draw());
   stackChildren.add(aliens[i].draw());
   stackChildren.add(asteroids[i].draw());
}
```

Далее добавляются планета и игрок:

```
stackChildren.add(planet.draw());
stackChildren.add(player.draw());
```

Благодаря `Stack` в игре присутствует ось `z`. Это означает, что элементы, добавленные в список позже, появятся поверх добавленных ранее. Поэтому мы должны убедиться, что добавляем их в правильном порядке. Так что игрок должен быть добавлен после планеты, а планета не должна перекрывать корабль, когда игрок летит рядом с ней. Корабль должен отображаться поверх планеты.

Следовательно, он должен быть выше по шкале z и поэтому должен быть добавлен после планеты.

Теперь, хотя они будут отображаться только в определенное время, мы добавляем взрывы:

```
for (int i = 0; i < explosions.length; i++) {
   stackChildren.add(explosions[i].draw());
}
```

Напомню, что если какой-либо игровой объект не виден в данный момент, метод draw() вернет пустой Container. Именно так обстоит дело со взрывами. Таким образом, большую часть времени этот цикл может рисовать кучу пустых виджетов Container, но это нормально, это просто способ управления видимостью во Flutter.

И вот теперь, наконец, создается виджет Scaffold в операторе return:

```
return Scaffold(body : GestureDetector(
   onPanStart : InputController.onPanStart,
   onPanUpdate : InputController.onPanUpdate,
   onPanEnd : InputController.onPanEnd,
   child : Stack(children : stackChildren)
));
```

Тело Scaffold не возвращает Stack напрямую, как вы могли ожидать. Нам нужно будет разработать для игрока метод управления, чтобы контролировать свой корабль, и мы это сделаем с помощью механизма управления жестами. Итак, нам нужен компонент для распознавания событий от сенсорного экрана, и виджет GestureDetector предназначен именно для этого.

Он распознает всевозможные жесты, нажатия и тому подобное, и одним из таких жестов является Pan. Это когда пользователь нажимает на экран и затем перемещает палец в разные стороны, не отрывая его. Если вы разрабатывали веб-сайт, на котором пользователи большую часть времени используют мышь, вам знакомы такие события, как mouseDown, mouseMove и MouseUp. В мобильных приложениях их нет, хотя фактически именно это нам и нужно. Три события Pan концептуально имитируют их (допустим, что палец игрока – это указатель мыши): onPanStart для mouseDown, onPanUpdate для mouseMove и onPanEnd для mouseUp. Код, который обрабатывает эти события, реализуется с помощью класса InputController, но мы вернемся к нему позже. Так как дочерним элементом GestureDetector является Stack, это означает, что жесты будут обрабатываться в любом месте экрана, потому что Stack занимает весь экран (он автоматически заполняет свой родительский элемент, как и GestureDetector). Наконец, как уже упоминалось ранее, поле children у Stack ссылается на созданный ранее список.

Помните, что все, о чем мы только что говорили, находится внутри метода build() виджета верхнего уровня. Это значит, что каждый раз, когда изменяется состояние, build() будет вызываться снова, а экран – перерисовываться.

Это ключ к тому, чтобы все это работало как игра. Далее мы обсудим основной игровой цикл, который я упоминал несколько раз, а также основную логику, из которой состоит игра.

Основной игровой цикл и основная игровая логика

Основная логика игры содержится в файле `GameCore.dart` и начинается, как всегда, с импорта:

Начнем

```
import "dart:math";
import "package:flutter/material.dart";
import "package:audioplayers/audio_cache.dart";
import "InputController.dart" as InputController;
import "GameObject.dart";
import "Enemy.dart";
import "Player.dart";
```

Пакет `math` необходим, потому что нам нужно сгенерировать несколько случайных чисел, а он включает в себя эту функциональность. Модуль `audio_cache.dart` – это часть плагина аудиоплееров, а также интерфейс, который мы будем использовать для загрузки и воспроизведения звуковых файлов. Остальное – различные модули самого `FlutterHero`.

Еще у нас есть целая куча переменных:

- `State state` – это ссылка на класс `State`;
- `Random random = new Random()` – класс `Random`, который позволяет нам… вы догадались… генерировать случайные числа! Я создаю его один раз, потому что, хотя он и понадобится нам несколько раз, нет смысла иметь более одного экземпляра;
- `int score = 0` – текущий счет игры;
- `double screenWidth` – ширина экрана;
- `double screenHeight` – высота экрана;
- `AnimationController gameLoopController` – мы поговорим об этом через мгновение!
- `Animation gameLoopAnimation` – идет вместе с `gameLoopController`;
- `GameObject crystal` – единственный игровой объект кристалла;
- `List fish` – список рыб, вражеских объектов-вредителей;
- `List robots` – список роботов, вражеских объектов-вредителей;

- `List aliens` – список инопланетян, вражеских объектов-вредителей;
- `List asteroids` – список астероидов, вражеских объектов-вредителей;
- `Player player` – единственный объект игрока;
- `GameObject planet` – единственный игровой объект планеты;
- `List explosions = [ ]` – список взрывов, которые также являются `GameObject` (если на экране нет взрывов, список будет пуст);
- `AudioCache audioCache` – кеш звуков, которые можно воспроизвести (позже расскажу подробнее).

С переменными мы разобрались, теперь перейдем к коду.

Первичная инициализация

Первый фрагмент кода – это функция `firstTimeInitialization()`, которую вы видели в методе `build()` основного виджета, помните? Этот вызов будет сделан тогда и только тогда, когда значение переменной `gameLoopController` было `null`. Ну, и наконец:

```
void firstTimeInitialization(BuildContext inContext, dynamic inState) {
   state = inState;
```

Код в этом модуле будет нуждаться в доступе к объекту `GameScreenState`, так как он содержит состояние для основного виджета. Так что ссылка на объект передается, а затем сохраняется в переменной `state`.

Теперь разберемся со звуком:

```
audioCache = new AudioCache();
audioCache.loadAll([ "delivery.mp3", "explosion.mp3",
   "fill.mp3", "thrust.mp3" ]);
```

Плагин `audioplayers` использует различные способы работы со звуком, и класс `AudioCache` – один из них. Его используют для предварительной загрузки звуков и, что очень важно для игр, своевременного воспроизведения. Это необходимо, чтобы иметь возможность воспроизводить звуки, которые добавлены в наше приложение. Итак, странно это или нет, мы создаем экземпляр класса и затем вызываем метод `loadAll()`, передавая ему список имен звуковых файлов для загрузки. Теперь мы готовы воспроизводить эти звуки в любое время.

Затем нам нужно получить размеры экрана:

```
screenWidth = MediaQuery.of(inContext).size.width;
screenHeight = MediaQuery.of(inContext).size.height;
```

Класс `MediaQuery` предоставляется библиотекой `material.dart` и позволяет нам получать информацию о нужном мультимедиаобъекте, например экране. Вызов метода `of()` для входящего `BuildContext` возвращает нам объект

MediaQueryData, который мы можем разобрать, чтобы получить ширину и высоту экрана.

Пришло время создать игровой объект!

```
crystal = GameObject(screenWidth, screenHeight, "crystal", 32, 30, 4, 6, null);
planet = GameObject(screenWidth, screenHeight, "planet",
   64, 64, 1, 0, null);
player = Player(screenWidth, screenHeight, "player", 40, 34, 2, 6, 2);
fish = [
   Enemy(screenWidth,screenHeight, "fish", 48, 48, 2, 6, 1, 4),
   Enemy(screenWidth,screenHeight, "fish", 48, 48, 2, 6, 1, 4),
   Enemy(screenWidth,screenHeight, "fish", 48, 48, 2, 6, 1, 4)
];
```

Кристалл и планета – это старые добрые экземпляры GameObject, в то время как игрок и вражеские паразиты – экземпляры Player и Enemy соответственно. Роботы, инопланетяне и астероиды созданы так же, как рыбы, поэтому нет смысла это объяснять. Обратите внимание, что они должны быть созданы именно здесь, потому что нам нужно, чтобы screenWidth и screenHeight уже были определены.

Коротко об анимациях во Flutter

Flutter предоставляет богатую поддержку анимаций, но в конечном итоге всё сводится к нескольким ключевым классам, даже если вы не используете их напрямую (например, виджеты, которые создают свою собственную анимацию, скрыто используют эти классы). Сначала вам нужен объект Ticker, затем объект Animation и, наконец, AnimationController.

Ticker – это объект, который посылает сигнал с регулярным интервалом, обычно 60 раз в секунду. Каждый раз, когда этот объект «тикает», выполняются заданные callback-функции для реализации связанных с анимацией процессов.

Объект Animation связан с генерацией числа на каждом тике. Это число – часть последовательности между двумя определенными значениями в заданный промежуток времени. Оно может изменяться линейно или с помощью сложных кривых.

AnimationController – это объект, который управляет анимацией. Он может запускать, останавливать и ставить на паузу анимацию. Он также может отменять анимацию (и помните, что «анимация» не означает ничего, кроме генерации следующего значения в последовательности – ни одно из них на самом деле не знает о том, что находится на экране).

AnimationController связывается с Ticker, который чаще всего связан с объектом State. Таким образом, после каждого «тика» Ticker отправляется сигнал в AnimationController. Потом он отправляет сигнал объекту Animation, который генерирует новое значение. Затем ваш код подключается к событиям

жизненного цикла Animation и делает все необходимое для отрисовки анимированных элементов на экране. В конечном счете ваш код (или код виджета Flutter, который вы используете) отвечает за фактическое размещение объекта на экране и его перемещение (или другие изменения, поэтому помните, что анимация является общей концепцией и применима не только к перемещению элемента, мы также можем анимировать изменение размера или другие свойства, имеющие числовое значение).

Итак, представьте, что у вас есть Ticker, срабатывающий 60 раз в секунду. А еще представьте, что Animation выплевывает линейный набор чисел от 0 до 500 под управлением AnimationController. Наконец, представьте, что вы подключаетесь к жизненному циклу Animation, так что каждый раз, когда генерируется число, вы обновляете местоположение X одного из наших врагов на экране. Это приведет к повторному запуску метода build(), и вы увидите, что объект начал перемещаться. Другими словами, у вас есть анимация!

Это основная концепция, поэтому теперь давайте посмотрим на реальный код, который применяет эту теорию на практике:

```
gameLoopController = AnimationController(vsync : inState,
   duration : Duration(milliseconds : 1000));
gameLoopAnimation = Tween(begin : 0, end : 17).animate(
   CurvedAnimation(parent : gameLoopController, curve : Curves.linear)
);
gameLoopAnimation.addStatusListener((inStatus) {
   if (inStatus == AnimationStatus.completed) {
      gameLoopController.reset();
      gameLoopController.forward();
   }
});gameLoopAnimation.addListener(gameLoop);
```

Во-первых, создается экземпляр AnimationController, который с помощью свойства vsync связывается с Ticker, в нашем случае это объект inState класса GameScreenState. Если вы посмотрите на код из предыдущих разделов, то увидите, что GameScreenState также наследуется от TickerProviderStateMixin. Это превращает inState в Ticker! Мы также сообщаем AnimationController о том, как долго хотим анимировать значения с помощью свойства duration, и в нашем случае это одна секунда (1000 миллисекунд).

Далее мы должны создать объект Animation и связать его с AnimationController. Есть несколько подклассов, которые мы могли бы использовать. Я выбрал, пожалуй, самый простой: Tween. Он позволит нам задать последовательность от начального до конечного значения, которые здесь составляют от 0 до 17.

Почему именно эти значения? Смысл в том, чтобы создать так называемый «основной игровой цикл» (main game loop). Это хитрый способ сообщить о том, что мы хотим, чтобы какая-либо функция выполнялась один раз при смене кадра. Но сколько времени занимает такая итерация? Что ж, здесь мы делим

общее время на количество тиков. Это означает, что 1000 делится на 60. Получается 16.666667. Округлите до 17 – вот он, наш диапазон. Короче говоря, мы хотим, чтобы функция основного игрового цикла выполнялась один раз в 17 миллисекунд, то есть она будет выполняться 60 раз в секунду (примерно). Вот что делает анимация gameLoopAnimation: она выдает числа от 0 до 17 в течение 1 секунды, один раз в 17 миллисекунд.

Теперь, когда все настроено, как надо, мы должны подключиться к gameLoop Animation, чтобы выполнять свою работу. Происходит это в двух местах. Во-первых, вы должны заметить, что через одну секунду эта анимация будет завершена. Последовательность значений будет исчерпана. Нам, очевидно, нужно, чтобы анимация повторялась снова и снова. Поэтому мы задаем функцию-слушатель (listener), которая выполняется каждый раз, когда меняется статус нашего объекта Animation. Эта функция будет вызываться два раза – когда анимация начинается и заканчивается. Нас интересует только окончание анимации (когда inStatus имеет значение completed) – в этом случае мы вызываем методы reset() и forward() нашего AnimationController. Первый метод сбрасывает значение в начальное, а второй запускает анимацию снова.

А еще мы должны знать о том, когда генерируется новое число в последовательности, чтобы мы могли вызвать функцию gameLoop, которая реализует основной игровой цикл. Метод addListener() в экземпляре Animation делает именно это.

Когда основной игровой цикл подключен и готов к работе, нам просто нужно сбросить все переменные состояния игры:

```
resetGame(true);
```

Мы собираемся разобрать его позже, так что давайте пока это пропустим. Дальше идет:

```
InputController.init(player);
```

Объект InputController отвечает за обработку пользовательского ввода, но и о нем я расскажу немного позже. Но взгляните, в этой функции есть еще одна строка:

```
gameLoopController.forward();
```

Здесь запускается игровой цикл, а значит, наша игра запущена. Ура! Наши объекты научились двигаться по экрану!

Сброс состояния игры

В начале игры, во время взрыва игрока или же когда энергия доставляется на планету, игру необходимо перезапустить. С этим нам поможет функция resetGame():

```
void resetGame(bool inResetEnemies) {
   player.energy = 0.0;
   player.x = (screenWidth / 2) - (player.width / 2);
```

```
   player.y = screenHeight - player.height - 24;
   player.moveHorizontal = 0;
   player.moveVertical = 0;
   player.orientationChanged();
```

Сначала мы избавляем корабль от энергии, центрируем его по горизонтали и смещаем к нижней границе экрана. Затем мы должны сбросить значения `moveHorizontal` и `moveVertical`, которые определяют направление движения, и сообщить об этом остальному коду с помощью вызова `orientationChanged()`.

```
crystal.y = 34.0;
randomlyPositionObject(crystal);
```

Следом за игроком и кристалл возвращается в стартовую позицию. Обратите внимание, что после первого вызова функции `resetGame()` нет смысла устанавливать свойство «y» нашего кристала, поскольку оно не меняется, но повторная установка не принесет вреда. Свойство «x» устанавливается функцией `randomlyPositionObject()`, которую мы рассмотрим позже, но по названию понятно, что она делает!

Планета задается практически так же:

```
planet.y = screenHeight - planet.height - 10;
randomlyPositionObject(planet);
```

Свойство «y» должно учитывать высоту планеты, чтобы она не свисала с нижней части экрана (10 пикселей – это произвольное значение, но я выбрал его так, чтобы начальная позиция корабля была примерно по центру вертикальной оси планеты).

Следом идут противники (возможно):

```
if (inResetEnemies) {
   List xValsFish = [ 70.0, 192.0, 312.0 ];
   List xValsRobots = [ 64.0, 192.0, 320.0 ];
   List xValsAliens = [ 44.0, 192.0, 340.0 ];
   List xValsAsteroids = [ 24.0, 192.0, 360.0 ];
   for (int i = 0; i < 3; i++ ) {
      fish[i].x = xValsFish[i];
      robots[i].x = xValsRobots[i];
      aliens[i].x = xValsAliens[i];
      asteroids[i].x = xValsAsteroids[i];
      fish[i].y = 110.0;
      robots[i].y = fish[i].y + 120;
      aliens[i].y = robots[i].y + 130;
      asteroids[i].y = aliens[i].y + 140;
      fish[i].visible = true;
      robots[i].visible = true;
      aliens[i].visible = true;
```

```
      asteroids[i].visible = true;
   }
}
```

Когда метод `resetGame()` впервые вызывается из `firstTimeInitialization()`, ему передается значение `true` в качестве значения аргумента `inResetEnemies`. Это приводит к выполнению показанного блока кода. Когда игрок взрывается, передается `false`, чтобы пропустить эту настройку, так как нет смысла сбрасывать позиции вражеских объектов, а когда энергия доставляется на планету – снова передается `true`.

Логика сброса здесь проста: у нас есть четыре списка (по одному для каждого типа врагов), которые содержат значения «x» для каждого врага. Вместо того чтобы вычислять их динамически, я решил, что будет проще использовать «магические числа». Важно отметить, что это позволило легко ввести нужный разброс значений без большого количества кода: расстояние между врагами изменяется, чтобы избежать «пустых коридоров», которые игрок смог бы слишком легко пройти. Также я меняю значения «y» так, чтобы ряды врагов становились плотнее при приближении к кристаллу. Мы еще должны убедиться, что все враги отображаются на экране (значение их свойства `visible` равно `true`), потому что после передачи энергии на планету они взрываются и исчезают. Поэтому их нужно будет показать заново.

Осталось только две небольшие задачи:

```
explosions = [ ]; player.visible = true;
```

Позже вы увидите, как обрабатываются взрывы, а пока достаточно знать, что список `explosions` необходимо очистить. Также нам необходимо сделать игрока видимым, на случай если он успел взорваться. Это воскресит его и позволит повторить попытку.

Основной игровой цикл

Наконец, мы подошли к основной функции игрового цикла, которая вызывается 60 раз в секунду, каждые 17 миллисекунд, как это указано при настройке анимации:

```
void gameLoop() {
crystal.animate();
```

Первое, что нужно сделать, – это запросить анимацию кристалла. Как вы видели в коде `GameObject`, это запускает циклическую смену изображений (кадров), что добавляет нашему кристаллу немного цветных переливов.

Далее мы должны оживить и переместить врагов-паразитов:

```
for (int i = 0; i < 3; i++) {
   fish[i].move();
   fish[i].animate();
```

```
    robots[i].move();
    robots[i].animate();
    aliens[i].move();
    aliens[i].animate();
    asteroids[i].move();
    asteroids[i].animate();
}
```

Паразиты, по три каждого типа, оживают и начинают двигаться. Сохранение логики этих функций в классах `GameObject` и `Enemy` теперь должно иметь для вас смысл: это избавляет нас от необходимости писать лишний код. Затем пора двигаться и оживать игроку:

```
player.move();
player.animate();
```

Обратите внимание, что точный порядок всех этих вызовов не имеет особого значения. Вызов `player.animate()` перед `player.move()`, или анимирование игрока раньше паразита, – это не имеет принципиальной разницы.

Теперь мы перейдем к нашим взрывам:

```
for (int i = 0; i < explosions.length; i++) {
    explosions[i].animate();
}
```

В это время на экране может не быть взрывов, или он может быть один (если игрок ударил врага), или их может быть двенадцать (если энергию только что доставили на планету, после чего все паразиты взрываются). Итак, это простой цикл, где каждая итерация дает возможность анимировать отдельный взрыв в списке.

Пока что все было очень просто, мы обновляли местоположение и внешний вид наших игровых объектов. Но, конечно, это еще не все: у нас также должна быть логика, чтобы сделать игру настоящей. Наша логика будет следующей:

```
if (collision(crystal)) {
    transferEnergy(true);
} else if (collision(planet)) {
    transferEnergy(false);
} else {
    if (player.energy > 0 && player.energy < 1) {
        player.energy = 0;
    }
}
```

Первая часть этого кода проверяет, столкнулся ли игрок с кристаллом или планетой. Функция `collision()` реализует эту проверку, но мы рассмотрим ее позже. А пока запомните, что она просто возвращает `true`, если игрок и указанный объект столкнулись, иначе – `false`.

Если мы коснулись кристалла, то нам нужно передать энергию на корабль (или на планету с корабля), для этого есть функция `transferEnergy()`, которую мы скоро рассмотрим. Передача `true` говорит о том, что столкновение произошло с кристаллом, а `false` означает планету, вы можете увидеть это на ветке `else if`.

Следующая ветвь `else` позволяет избежать «читерства»: если у игрока есть энергия, но корабль не успел заполниться ею на 100 %, то вся энергия теряется. Так игрок не сможет собрать лишь часть энергии и при этом получить за нее полный кредит после доставки на планету. Это было бы ужасно для экономики солнечной системы Горгоны (а также, скорее всего, привело бы к войнам между капитанами, правда, из-за хрупкости кораблей эти войны были очень короткими, но что-то я отвлекся), поэтому мы должны это предотвратить прямо здесь и сейчас! Поскольку подобная ситуация может возникнуть *только* в том случае, если игрок не контактирует ни с кристаллом, ни с планетой, ветвь `else` для этой логики отлично подходит.

Далее нам нужно проверить наличие столкновений с паразитами:

```
for (int i = 0; i < 3; i++) {
   if (collision(fish[i]) || collision(robots[i]) ||
      collision(aliens[i]) || collision(asteroids[i])) {
         audioCache.play("explosion.mp3");
         player.visible = false;
         GameObject explosion = GameObject(screenWidth,
            screenHeight, "explosion", 50, 50, 5, 4,
            () { resetGame(false);}
         );
         explosion.x = player.x;
         explosion.y = player.y;
         explosions.add(explosion);
         score = score - 50;
         if (score < 0) { score = 0; }
      }
   }
```

Очевидно, что нам требуется проверять каждого врага в отдельности, отсюда и потребность в циклах. Чтобы избежать вложенных циклов, я проверяю сразу все виды врагов на каждой итерации. Если происходит какое-либо столкновение, мы сначала воспроизводим звук взрыва. `audioCache`, который мы создали ранее, обеспечивает корректное воспроизведение аудио с помощью метода `play()`. Все, что вам нужно сделать, – это передать ему имя файла для проигрывания (без префикса «`assets/`», поскольку `audioplayers` предполагают, что именно там находятся файлы). Легкотня! Далее игрок должен быть скрыт, потому что на его месте появится взрыв. Так что следующий наш шаг – создание экземпляра `GameObject` для взрыва. Он будет там, где находится игрок (ошибочка, находился!), и тогда `GameObject` добавляется в список взрывов (который

означает, что он будет анимирован, начиная со следующего кадра). Результат – взрывающийся корабль! Несколько очков вычитаются из баланса игрока за проигрыш (счет не должен быть отрицательным), и мы закончили.

Остается только одна задача, одна строка кода, но очень важная:

```
state.setState((){});
```

Без этого ничего не происходит! Без обновления состояния (переменная `state`) игрового поля Flutter не узнает, что что-то изменилось, так что `build()` не запустится снова и экран не обновится. Как думаете, это важный шаг?!

Теперь давайте рассмотрим функцию `collision()` и разберемся, что же она собой представляет.

Проверка на наличие столкновений

Большинство видеоигр должны уметь обнаруживать столкновение двух объектов. Так что и нам нужно знать о моменте, когда корабль ударит по любому из паразитов. Существует несколько способов сделать это, каждый со своими плюсами и минусами. Среди них есть один простой способ под названием «ограничительные рамки» (bounding boxes). Этот подход проверяет границы двух прямоугольников, и если они пересекаются, то происходит столкновение.

Как показано в примере на рис. 9-4, каждый игровой объект имеет квадратную/прямоугольную область вокруг него, называемую ограничительной рамкой. Эта рамка определяет границы области, занимаемой объектом. Обратите внимание, что на рисунке верхний левый угол ограничительной рамки объекта 2 находится внутри ограничительной рамки объекта 1. Это и есть область столкновения (collision area). Вы можете обнаружить такую область, выполнив ряд простых проверок, сравнивающих границы каждого объекта. Если какое-либо из условий неверно, то столкновения не произошло. Например, если низ объекта 1 находится выше верха объекта 2, то столкновения не было. Фактически, поскольку вы имеете дело с квадратным или прямоугольным объектом, у вас есть только четыре условия для проверки, любое из которых, будучи ложным, исключает возможность столкновения.

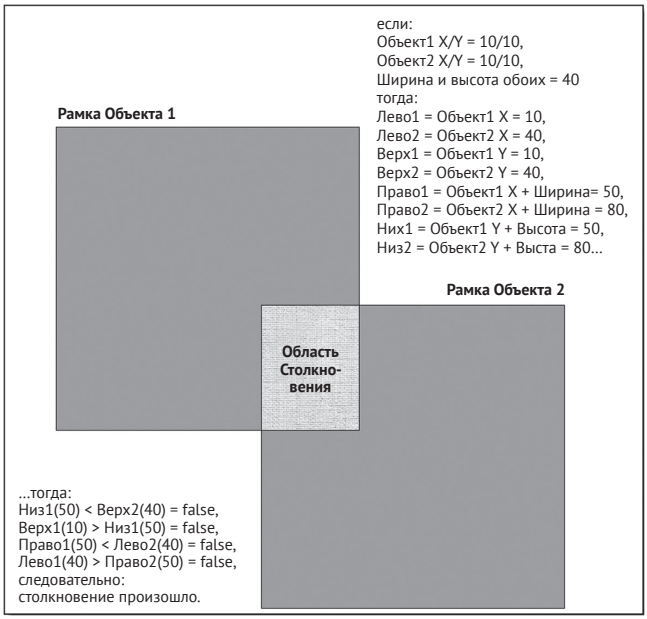


Рисунок 9-4. *Основная концепция работы ограничительных рамок*

Однако этот алгоритм не дает идеальных результатов. Например, вы иногда будете видеть, как корабль «ударяет» объект, в то время как он внешне его даже не касается. В других случаях объекты могут немного соприкоснуться, но это не будет зарегистрировано как столкновение. Вращение корабля также имеет значение, поскольку простой алгоритм не сможет обработать измененную геометрию объекта, так как он не будет квадратным (и идеально выровненным по вертикали или горизонтали). Это можно исправить более сложной версией алгоритма или с помощью проверок на уровне отдельных пикселей. Тем не менее подход ограничительных рамок, показанный здесь, дает «достаточно хорошие» результаты – в игру можно играть даже с такой погрешностью.

Теперь мы можем рассмотреть функцию `collision()`, на которую ссылались в предыдущем разделе:

```
bool collision(GameObject inObject) {
   if (!player.visible || !inObject.visible) { return false; }
   num left1 = player.x;
   num right1 = left1 + player.width;
   num top1 = player.y;
   num bottom1 = top1 + player.height;
   num left2 = inObject.x;
   num right2 = left2 + inObject.width;
   num top2 = inObject.y;
   num bottom2 = top2 + inObject.height;
   if (bottom1 < top2) { return false; }
```

```
   if (top1 > bottom2) { return false; }
   if (right1 < left2) { return false; }
   return left1 <= right2;
}
```

Если игрок или объект, который мы проверяем на столкновение, не отображается, то нет необходимости проверять дальше, ведь игровые объекты скрываются только при взрыве. После этого мы рассчитываем сравниваемые значения, то есть координаты верхней (top), нижней (bottom), левой (left) и правой (right) границ игрока и заданного объекта. Это всего лишь четыре простые проверки, которые сообщат вам, произошло ли столкновение.

Размещение объекта в случайной точке

После того как игрок заберет всю энергию из кристалла, или перенесет ее на планету, или игра перезагрузится, кристалл и планета располагаются случайным образом с помощью вызова метода `randomlyPositionObject()`:

```
void randomlyPositionObject(GameObject inObject) {
   inObject.x = (random.nextInt(
      screenWidth.toInt() - inObject.width)).toDouble();
   if (collision(inObject)) {
      randomlyPositionObject(inObject);
   }
}
```

Как видим, мы используем метод `nextInt()` объекта `random`, созданного при запуске приложения. Требуемое значение должно быть в диапазоне от нуля до ширины экрана минус ширина объекта, чтобы объект всегда был на экране и не задевал его края. Только горизонтальное положение объекта является случайным, поэтому результирующее случайное значение устанавливается в его свойство «x». Другой объект не может находиться в том же месте, что и игрок, поэтому мы вызываем `collision()` для проверки этого условия, и если происходит столкновение, то `randomlyPositionObject()` вызывается рекурсивно, пока не будет выбрана позиция без столкновения.

Передача энергии

Когда корабль «сталкивается» с кристаллом или планетой, энергия должна передаваться. Для этого вызывается функция `transferEnergy()`:

```
void transferEnergy(bool inTouchingCrystal) {
   if (inTouchingCrystal && player.energy < 1) {
```

Если игрок прикасается к кристаллу, то мы должны убедиться, что корабль еще не заполнен. Значение `energy` варьируется от 0 до 1, поскольку это соответствует диапазону значений виджета `LinearProgressIndicator`.

Когда происходит первый контакт, нам нужно воспроизвести соответствующий звук:

```
if (player.energy == 0) { audioCache.play("fill.mp3"); }
```

При первом касании кристалла энергия, конечно, будет равна нулю, поэтому это условие выполняется.

В дальнейшем энергия увеличивается на одну сотую каждую итерацию игрового цикла, пока игрок получает энергию. Также значение energy ограничивается сверху единицей:

```
player.energy = player.energy + .01;
if (player.energy >= 1) {
   player.energy = 1;
   randomlyPositionObject(crystal);
}
```

Кроме того, когда корабль заполнен, кристалл перемещается в случайную позицию и перестает собирать энергию.

Ветка `else if`, для контакта с планетой, будет следующей:

```
} else if (player.energy > 0) {
```

Конечно, это работает только при наличии энергии на корабле, поэтому мы проверяем ее.

Затем, как и при контакте с кристаллом, мы хотим воспроизвести другой звук при контакте с планетой, поэтому:

```
if (player.energy >= 1) {
   audioCache.play("delivery.mp3");
}
```

И, как и в случае с кристаллом, энергия на корабле регулируется:

```
player.energy = player.energy - .01;
```

Конечно, при изменении значения energy специальный индикатор должен отобразить эти изменения.

Еще есть немного логики, которую мы должны реализовать при передаче энергии на планету:

```
if (player.energy <= 0) {
   player.energy = 0;
   audioCache.play("explosion.mp3");
   score = score + 100;
   for (int i = 0; i < 3; i++) {
      Function callback;
      if (i == 0) {
         callback = () { resetGame(true); };
      }
```

```
      fish[i].visible = false;
      GameObject explosion = GameObject(screenWidth,
         screenHeight, "explosion", 50, 50, 5, 4, callback);
      explosion.x = fish[i].x;
      explosion.y = fish[i].y;
      explosions.add(explosion);
      robots[i].visible = false;
   }
}
```

Здесь мы гарантируем, что энергия не может быть меньше нуля, воспроизводим звук взрыва и увеличиваем счет игрока. Это все потому, что пришло время взрывать паразитов! Цикл выполняется, паразиты скрываются, и на их месте появляется взрыв. Обратите внимание, что callback-функция, которую вы видели ранее при просмотре класса `GameObject`, подключена только к первому объекту взрыва. Это позволяет перезапустить игру после завершения анимации.

Примечание. Код, который вы видите здесь для рыб, повторяется для роботов, инопланетян и астероидов, поэтому я сэкономил немного места, не показывая их здесь.

Результат выполнения этого кода показан на рис. 9-5 – красивое массовое убийство паразитов!

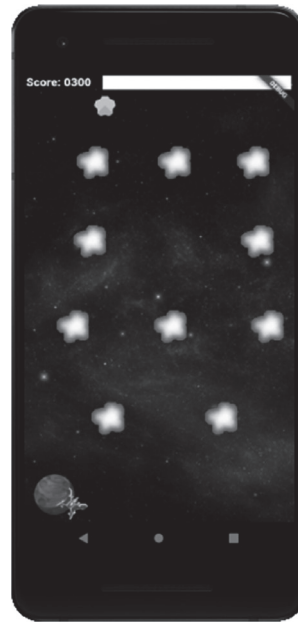


Рисунок 9-5. *Бум! Вы все проиграли!*

Но они возвращаются, так что победа нашего героя была недолгой. :(

Все под контролем: InputController.dart

Последний фрагмент кода, который мы рассмотрим, – это класс InputController, и вы уже видели подключение его к событиям GestureDetector. Данный контроллер реализует все элементы управления движением игрока и начинается следующим образом:

```
import "packageflutter/material.dart";
import "Player.dart";

double touchAnchorX;
double touchAnchorY;
int moveSensitivity = 20;
```

После очевидного импорта у нас есть три переменные. Способ работы схемы управления заключается в том, что игрок помещает палец в любом месте на экране, и оно становится «якорной точкой» (anchor point). Представьте себе игровой джойстик. Центральное положение и есть якорная точка. Теперь, когда игрок перемещает палец, новая позиция относительно этой якорной точки представляет движение в определенном направлении. Если его палец, скажем, на 20 пикселей выше якорной точки, то он хочет переместить корабль вверх. Если пользователь убирает палец и затем нажимает в другое место, у нас появляется новая якорная точка. В каком-то смысле это «виртуальный джойстик» в любом удобном месте экрана. Итак, нам нужны две переменные, чтобы записать координаты X и Y якорной точки. Нам также нужно знать, на сколько пикселей требуется сместить палец, чтобы началось перемещение корабля – это так называемая «чувствительность» джойстика. После некоторых экспериментов я остановился на 20.

Нам, очевидно, также нужна ссылка на игрока:

```
Player player;
```

И эта ссылка сохраняется при вызове метода init() из метода firstTime Initialization():

```
void init(Player inPlayer) { player = inPlayer; }
```

Надеюсь, вы помните, что сначала нам нужно обработать три события Gesture Detector: onPanStart (когда игрок опускает палец вниз), onPanUpdate (когда он двигает пальцем) и onPanEnd (когда он поднимает палец).

Сначала запускается событие onPanStart:

```
void onPanStart(DragStartDetails inDetails) {
   touchAnchorX = inDetails.globalPosition.dx;
   touchAnchorY = inDetails.globalPosition.dy;
   player.moveHorizontal = 0;
   player.moveVertical = 0;
}
```

Задача проста: сохранить новую якорную точку и убедиться, что игрок не двигается. Объект DragStartDetails, переданный в этот метод, содержит несколько фрагментов информации. Наиболее важными для нас являются globalPosition.dx и globalPosition.dy для горизонтальной (x) и вертикальной (y) позиций перетаскивания.

Далее идет функция onPanUpdate(), где выполняется большая часть работы этого класса:

```
void onPanUpdate(DragUpdateDetails inDetails) {
   if (inDetails.globalPosition.dx < touchAnchorX - moveSensitivity) {
      player.moveHorizontal = -1;
      player.orientationChanged();
   } else if (inDetails.globalPosition.dx >
      touchAnchorX + moveSensitivity) {
      player.moveHorizontal = 1;
      player.orientationChanged();
   } else {
      player.moveHorizontal = 0;
      player.orientationChanged();
   }
   if (inDetails.globalPosition.dy < touchAnchorY - moveSensitivity) {
      player.moveVertical = -1;
      player.orientationChanged();
   } else if (inDetails.globalPosition.dy >
      touchAnchorY + moveSensitivity) {
      player.moveVertical = 1;
      player.orientationChanged();
   } else {
      player.moveVertical = 0;
      player.orientationChanged();
   }
}
```

Это может выглядеть сложно, но это не так: все, что здесь делается, – это простое сравнение координат для определения того, в какую сторону игрок направляет свой корабль. Затем вызывается метод, который изменяет ориентацию (направление движения) корабля. Для начала мы проверяем изменение координат по оси «x», а затем повторяем эту же логику для оси «y». В результате этих манипуляций мы получаем полноценный виртуальный джойстик, который управляет направлением движения корабля.

Наконец, нам просто нужно обработать событие onPanEnd:

```
void onPanEnd(dynamic inDetails) {
   player.moveHorizontal = 0;
   player.moveVertical = 0;
}
```

Все, что нам нужно здесь сделать, – это остановить любое активное движение.

Поздравляю! У нас есть полностью готовая игра, написанная на Flutter!

Что дальше?

Вот и все! Вы сделали это! Три готовых Flutter-приложения, а последнее – игра! В этой главе вы узнали о таких вещах, как виджет `Positioned`, генерация случайных чисел, обработка жестов, `AnimationController`, `Tween`, анимации и проигрывание звуков. Вы также узнали (если, конечно, не встречались с этим раньше), как проектировать игры.

Я искренне надеюсь, что вам понравилась книга «Flutter на практике» и вы многому научились. У меня никогда не было цели сделать из вас эксперта по Flutter – это слишком обширная тема для одной книги! Но если я выполнил свою задачу хотя бы частично, то у вас теперь есть прочная основа, на которой можно дальше развивать свои знания. Я также предоставил вам необходимые строительные блоки для создания собственных приложений на основе Flutter.

Итак, погрузитесь в дальнейшее изучение, не сидите сложа руки, идите и создавайте великие мобильные приложения с помощью Flutter и, в некоторой степени, я надеюсь, с помощью данной книги!

УКАЗАТЕЛЬ

G

H

I

O

P

R

S

T

Ф

Ц

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.
При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. +7 (499) 782-38-89
Электронный адрес: **books@alians-kniga.ru**.

Фрэнк Заметти

Flutter на практике

Прокачиваем навыки мобильной разработки с помощью открытого фреймворка от Google

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Тищенко А.С.*
Науч. редактор *Черников В. Н.*
Корректоры *Синяева Г. И.*
Верстка *Орлов И. Ю.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×90 1/16. Гарнитура «PT Serif».
Печать цифровая. Усл. печ. л. 23,99.
Тираж 200 экз.
Веб-сайт издательства: **www.dmkpress.com**