Данная книга в сжатой форме описывает целостный процесс создания приложений для смартфонов и планшетов:

- особенности мобильных операционных систем и устройств;
- выбор инструментов для разработки;
- подготовка рабочей документации в духе Agile;
- проектирование структуры и архитектуры решения;
- создание автоматизированного конвейера Continues Integration/ Continues Delivery;
- мониторинг работоспособности конечного продукта на устройствах реальных пользователей.

Все примеры приведены на языке C#.

Издание рекомендовано как опытным, так и начинающим программистам, интересующимся разработкой мобильных приложений. Также оно может быть использовано в качестве пособия в вузах, центрах дополнительного образования и др.

Желаем приятного чтения!

За плечами **Вячеслава Черникова** более 15 лет работы в области разработки и эксплуатации программного обеспечения. Он является экспертом в области облачных, мобильных и инновационных технологий Microsoft; в прошлом Microsoft MVP, Nokia Champion, Qt Ambassador, сертифицированный разработчик на Qt и Xamarin, победитель российских и международных конкурсов, хакатонов. Стаж преподавания студентам и школьникам более 7 лет. Вячеслав Черников активно выступает на конференциях, пишет руководства и статьи, проводит вебинары и учебные курсы по разработке мобильных и облачных бизнес-приложений.

Интернет-магазин:
www.dmkpress.com

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

www.дмк.рф

ISBN 978-5-97060-805-0

Разработка **мобильных приложений** на C# для iOS и Android

Вячеслав Черников

Разработка **мобильных приложений** на C# для iOS и Android

дмк издательство

Binwell University

Вячеслав Черников

Разработка мобильных приложений на C# для iOS и Android

Сравнение нативных и кроссплатформенных
инструментов на примере Xamarin, ReactNative,
PhoneGap, Qt и Flutter.
Проектирование и техническая документация для кода.
Архитектура и структура проекта,
раскладываем все по местам.
Mobile DevOps и автоматизация сборки,
тестирования, поставки и мониторинга.
Практические советы на каждый день

ДМК ПРЕСС
издательство
Москва, 2020

УДК 004.4
ББК 32.973.202
Ч49

Черников В. Н.
Ч49 Разработка мобильных приложений на C# для iOS и Android. – М.: ДМК Пресс, 2020. – 188 с.: ил.

ISBN 978-5-97060-805-0

Данная книга в сжатой форме описывает целостный процесс создания приложений для смартфонов и планшетов. Рассматриваются особенности мобильных операционных систем и устройств, выбор инструментов для разработки, подготовка рабочей документации в духе Agile, проектирование структуры и архитектуры решения, создание автоматизированного конвейера Continues Integration/Continues Delivery, а также мониторинг работоспособности конечного продукта на устройствах реальных пользователей. Все примеры приведены на языке C#.

Издание может быть рекомендована как опытным, так и начинающим программистам, интересующимся разработкой мобильных приложений. Также оно может быть использовано в качестве пособия в вузах, центрах дополнительного образования и др.

УДК 004.4
ББК 32.973.202

ISBN 978-5-97060-805-0

Содержание

Вступительное слово от издательства

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** или **www.дмк.рф** 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. Издательство «ДМК Пресс» очень серьезно относится 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.

Введение

Данная книга представляет собой практическое руководство для инженеров, уже овладевших основами разработки, а также руководителей или fullstack-разработчиков, которым в том числе необходимо создавать и поддерживать мобильные приложения. Все примеры даны на языке C# (фреймворк Xamarin.Forms).

Книга разделена на две части: в первой рассмотрен процесс выбора инструментов, проектирования и создания «скелета» (базовой структуры) проекта, а во второй – представлены практические решения для самых частых задач, с которыми сталкиваются разработчики. Ниже приведено краткое содержание каждой части и главы.

Часть I. Закладываем правильный фундамент

Как не построить большой дом без фундамента и грамотного проекта, так и реальное программное обеспечение без правильной архитектуры превращается в «миску спагетти» через 1–2 года развития.

В **главе 1** мы начнем с общего знакомства с инструментами кроссплатформенной разработки, включая PhoneGap, ReactNative, Flutter, Xamarin и Qt. Сравним заложенные в эти фреймворки архитектуры, что позволит нам лучше понять их работу в сравнении с нативными.

В **главе 2** мы опишем алгоритм создания «скелета» приложения (архитектура и структура кода) на основе легковесной онлайн-документации.

С **главы 3** начнется погружение в архитектуру, и далее в **главе 4** мы рассмотрим, какие есть особенности реализации различных компонентов приложения на базе MVVM и многослойной архитектуры, а также то, каким образом эти модули лучше связывать между собой.

И завершим мы первую часть книги описанием Mobile DevOps для выстраивания коммуникации в команде на базе технической документации, а также использования облачных инструментов для автоматической сборки и тестирования приложений (**глава 5**).

Часть II. Практические советы на каждый день

В этой части будут представлены практические советы по следующим темам:

- **глава 6** «Иконочные шрифты вместо растровых картинок»;
- **глава 7** «Работаем с состояниями экранов»;
- **глава 8** «Дополнительные анимации при переходе экрана из одного состояния в другое»;
- **глава 9** «Использование FastGrid для создания сложного интерфейса»;
- **глава 10** «Работа с сетевыми сервисами Json/REST»;
- **глава 11** «Авторизация с помощью нативных библиотек Facebook, ВКонтакте, а также с помощью OAuth».

Часть I

ЗАКЛАДЫВАЕМ ПРАВИЛЬНЫЙ ФУНДАМЕНТ

Глава 1

Особенности разработки мобильных приложений

Архитектуру программных продуктов можно сравнить со скелетом, расположением и связями внутренних органов человека. Именно поэтому в реальных проектах архитектуре следует уделять особое внимание. Чтобы лучше понимать особенности разработки мобильных приложений (кроссплатформенных и нативных), мы рассмотрим архитектуры популярных кроссплатформенных фреймворков.

Самих фреймворков сейчас существует очень много, но с архитектурной точки зрения они в основном аналогичны PhoneGap, ReactNative, Flutter, Xamarin и Qt. В качестве целевых платформ мы остановимся на iOS, Android и Windows UWP.

1.1. Нативные и кроссплатформенные инструменты разработки

Исторически на рынке компьютеров всегда была конкуренция и каждый производитель предоставлял оптимальный набор так называемых нативных (родных) инструментов для разработки приложений под свои операционные системы и устройства. Нативные средства разработки обеспечивают максимальную производительность и доступ к возможностям операционной системы.

Однако часто оказывалось, что эти инструменты были несовместимы друг с другом не только на уровне языка разработки, принятых соглашений и архитектур, но и на уровне механизмов работы с операционной системой и библиотеками. В результате для реа-

лизации одних и тех же алгоритмов, пользовательских или бизнес-сценариев требовалось написать приложение для нескольких сред на разных языках программирования. Например, если надо поддерживать две платформы, то требуется увеличение трудозатрат и команды в два раза. Плюс в два раза больше бюджетов на поддержку и развитие. Можно добавить, что во многих компаниях уже скопилась большая база кода, который также хотелось бы унаследовать в новых решениях.

Вторым важным моментом является наличие необходимых компетенций (знаний и опыта) внутри команды – если их нет, то потребуется время на обучение.

Для решения подобных проблем на рынке уже давно существуют инструменты кроссплатформенной разработки, предлагающие:

- максимизировать общую базу кода на едином языке программирования, чтобы продукт было проще разрабатывать и поддерживать;
- использовать существующие компетенции и специалистов для реализации приложений на новых платформах.

Так как языков программирования (и сред) сейчас наплодилось очень много (и специалистов, владеющих этими языками), то и инструментов для кроссплатформенной разработки существует изрядное количество. В данной книге нас интересуют только инструменты для создания мобильных бизнес-приложений, поэтому в следующих главах мы подробнее разберем, как они работают. А пока чуть детальнее каждый из плюсов кроссплатформенной разработки.

Общая база кода. В зависимости от выбранного инструмента разработчик может разделять между платформами ресурсы приложения (картинки, шрифты и прочие файлы), логику работы с данными, бизнес-логику и описание интерфейса. И если с ресурсами и бизнес-логикой все просто, то вот с интерфейсом следует быть осторожнее, так как для каждой из платформ есть свои рекомендации и требования.

Использование существующих компетенций и команды. Здесь стоит учитывать не только язык программирования, но и понимание механизмов работы операционных систем iOS/Android/Windows, а также набор дополнительных библиотек и инструментов разработки.

Рис. 1.1 ❖ Отличие нативной и кроссплатформенной мобильной разработки

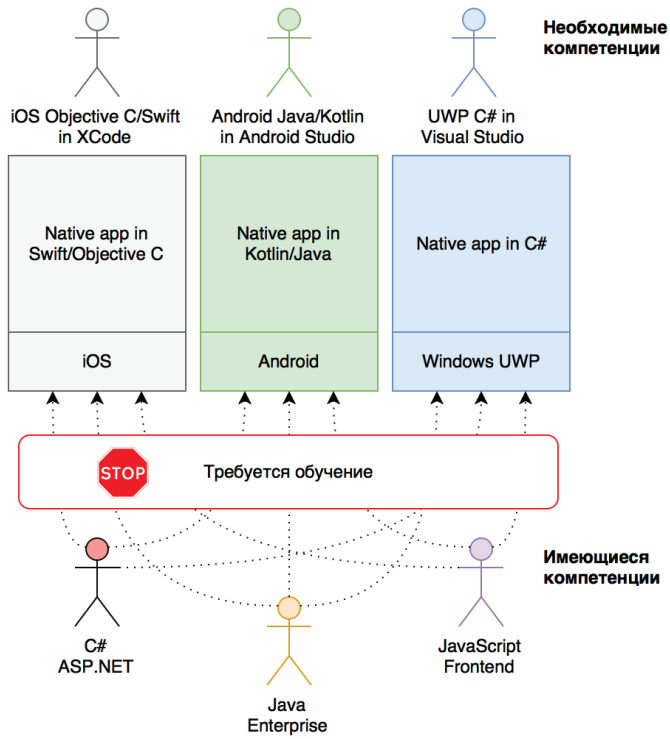


Рис. 1.2 ❖ Компетенции мобильной разработки

Итак, нативные инструменты предоставляются самими владельцами экосистем и позволяют получить максимум из возможностей целевой операционной системы, имеют полный доступ к родным API,

оптимальную производительность и требуют отдельной команды разработки под каждую платформу.

Кроссплатформенные фреймворки позволяют сократить трудозатраты и ускорить выпуск приложений в том случае, если требуется поддержка нескольких платформ одновременно и имеются (или развиваются) необходимые компетенции. В долгосрочной перспективе кроссплатформенные решения помогут сэкономить приличное количество человеко-часов, но для этого стоит учитывать особенности выбранного инструмента. Также одной кроссплатформенной командой разработки проще управлять, чем несколькими нативными.

1.2. Архитектура iOS/Android и нативные API

Главный принцип, лежащий в основе кроссплатформенных решений, – разделение кода на две части:

- **кроссплатформенную**, живущую в виртуальном окружении и имеющую ограниченный доступ к возможностям целевой платформы через специальный мост;
- **нативную**, которая обеспечивает инициализацию приложения, управление жизненным циклом ключевых объектов и имеет полный доступ к системным API.

Рис. 1.3 ❖ Общая архитектура кроссплатформенных фреймворков

Для того чтобы связывать между собой мир нативный и мир кроссплатформенный, необходимо использовать специальный **мост** (bridge), который и определяет возможности и ограничения кроссплатформенных фреймворков.

Использование bridge всегда негативно сказывается на производительности за счет преобразования данных между «мирами», а также конвертации вызовов API и библиотек. Сам по себе кроссплатформенный мир имеет сопоставимую с нативным производительность.

Итак, все кроссплатформенные приложения обязаны иметь нативную часть, иначе операционная система просто не сможет их запустить. Поэтому давайте рассмотрим подробнее, какие системные API и механизмы предоставляются самими iOS, Android и Windows.

1.2.1. Нативный iOS

Начнем мы наш обзор операционных систем с iOS, которая, в свою очередь, основана на Mac OS X, созданной из NeXTSTEP OS, являющейся полноценной Unix-системой. Поэтому iOS стоит воспринимать как полноценную Unix-систему без командной строки.

Нативные интерфейсы низкого уровня в iOS реализованы по аналогии с Unix (для C). Для iOS-разработчика выбор языков ограничивается Objective C и Swift, ведь именно для них реализованы нативные инструменты и API. Также можно использовать C/C++, но это будет либо от острой необходимости (есть существующие наработки), либо из сильного любопытства, так как потребуется высокая квалификация и написание приличной базы вспомогательного кода. Общая архитектура iOS представлена ниже.

Дополнительно на схеме мы отметили подсистемы, которые имеют значение для кроссплатформенных фреймворков:

- **WebKit** используется в гибридных приложениях на базе PhoneGap или аналогов для запуска приложений и фактически выступает средой выполнения веб-приложений;
- **JavaScript Core** используется в ReactNative и аналогах для быстрого выполнения JS-кода и обмена данными между Native и JS;
- **OpenGL ES** используется в играх и приложениях на Qt/QML, Flutter или аналогах для отрисовки интерфейса;

- **UIKit** отвечает за нативный пользовательский интерфейс приложения, что актуально для ReactNative и Xamarin.

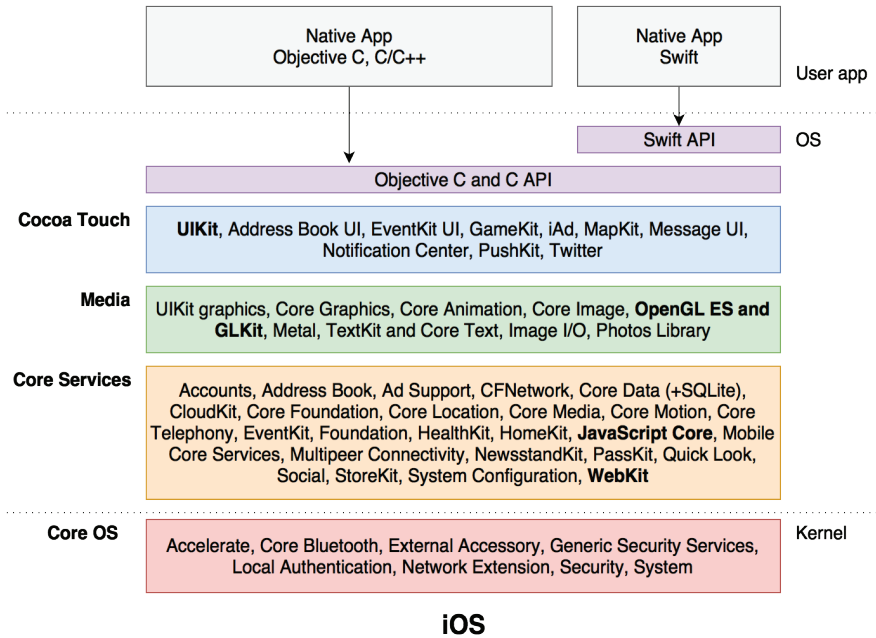


Рис. 1.4 ❖ Архитектура iOS

Как видим, из коробки iOS предоставляет готовые интерфейсы для Objective C (Swift работает в качестве надстройки), плюс имеются механизмы для кроссплатформенных HTML/JS-приложений (WebKit, JavaScriptCore). С iOS API на уровне системных вызовов могут работать любые фреймворки, поддерживающие Unix-вызовы, но для полноценного взаимодействия с Objective C API из других языков будет необходимо написать специальные обертки.

В iOS недоступна компиляция Just In Time, кроме компиляции JavaScript с помощью WebKit. Это связано с тем, что в iOS закрыт доступ к записываемой исполняемой памяти (writable executable memory), что не позволяет генерировать исполняемый код динамически.

Ввиду ограничений iOS все приложения, требующие JIT (кроме JavaScript) должны быть скомпилированы в машинный код (Ahead Of Time compilation, AOT), что может стать неожиданностью для разработчиков Java и .NET. Ограничение это продиктовано повышенными требованиями к безопасности и производительности.

1.2.2. Нативный Android

Android также является Unix-системой и большей частью основан на Linux со всеми вытекающими плюсами и минусами. Однако уши Linux не сильно торчат у Android, так как поверх ядра ОС создана своя инфраструктура, включающая виртуальную машину Java (Java Virtual Machine, JVM) для запуска приложений. JVM выступает посредником между пользовательским кодом и набором системных API, доступных для Java-приложений. Поддержка языка Kotlin является надстройкой над той инфраструктурой, которая доступна Java.

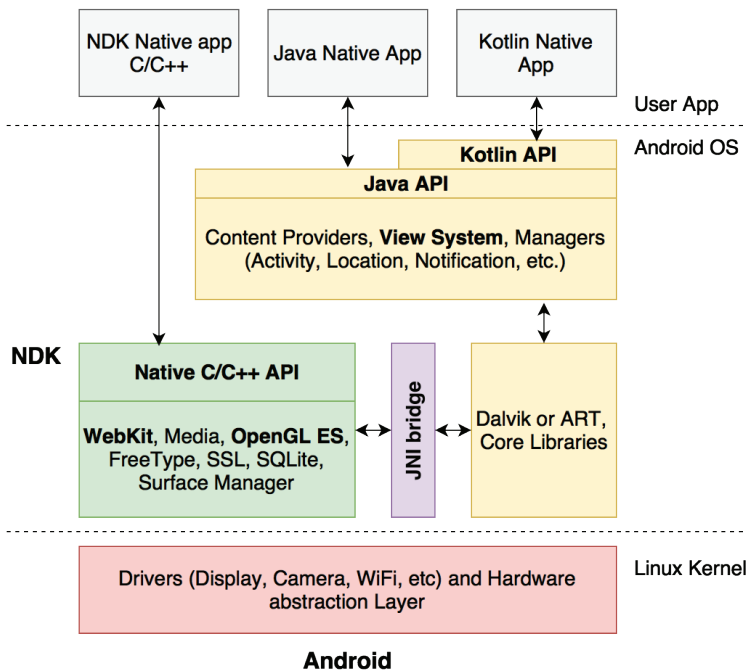


Рис. 1.5 ❖ Архитектура Android

Как видим, в Android разработчику доступно сразу целых две подсистемы: Native Development Kit (Android NDK) и Android SDK. С помощью NDK можно получить доступ к низкоуровневым механизмам Android. Разработка ведется на C/C++. При использовании Android SDK разработчик оказывается внутри Java-машины Dalvik (или Android Runtime, ART) и имеет только те возможности, которые предоставляет Java API.

Связующим звеном между библиотеками низкого уровня (на C/C++) и инфраструктурой Java выступает специальный JNI bridge (Java Native Interface), который и позволяет двум мирам взаимодействовать друг с другом. JNI выступает единым и универсальным связующим звеном, однако, как и любой мост, ведет к падению производительности, если начинает использоваться неэффективно.

JNI снижает производительность приложений, когда большой поток команд и данных передается через мост.

Помимо JNI bridge, в архитектуре Android также стоит обратить внимание на наличие подсистем **WebKit** (для PhoneGap), **OpenGL ES** (для Qt, Flutter и игр) и **View System** (грубо говоря, **iOS UIKit**; для ReactNative и Xamarin), аналогичные модулям в iOS. Однако в сравнении с iOS ограничений меньше – можно использовать JIT не только для JavaScript, но и других языков, плюс нет жесткой привязки к JS-движку.

Сам по себе Android до недавнего времени использовал JIT для Java-приложений, что не самым лучшим образом сказывалось на производительности. Начиная с версии 5.0, в Android добавили механизм AOT-компиляции байт-кода (как часть ART), что улучшило поведение программ, однако не сняло ограничения JNI bridge. Забегая вперед, отметим, что JNI будет использоваться в приложениях на Xamarin и Qt.

1.2.3. Нативный Windows UWP

Напоследок давайте рассмотрим архитектуру Windows UWP, которая является самой всеядной и предоставляет большое количество различных интерфейсов и механизмов взаимодействия, включая Win-

dows Bridges (**https://developer.microsoft.com/en-us/windows/bridges**).

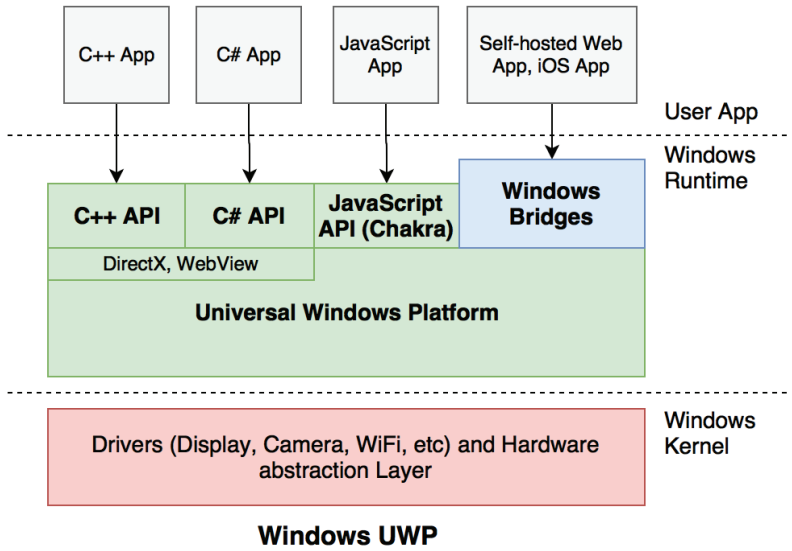


Рис. 1.6 ❖ Архитектура Windows UWP

Помимо традиционных API для C++/C#, Windows UWP также предоставляет механизмы работы с JavaScript на базе движка Chakra, который используется в Edge. Microsoft поддерживает open source версию ReactNative на Windows UWP: **https://github.com/Microsoft/react-native-windows**.

Также система имеет **WebView** и подходит для приложений в духе PhoneGap. Реализации OpenGL ES нет, вместо нее доступен только DirectX. Qt работает, но с большими ограничениями. Поддержка Flutter имеется, но еще далека от стабильности.

В качестве «диковинных» решений в Windows также доступны различные технологии бриджинга, например для запуска self-hosted сайтов (**https://developer.microsoft.com/en-us/windows/bridges/hosted-web-apps**) в качестве локальных приложений, а также классических десктопных Win32-программ или даже iOS-приложений (**https://developer.microsoft.com/en-us/windows/bridges/ios**).

Для нас же важно, что Windows UWP обеспечивает все необходимые механизмы для работы PhoneGap, Flutter, ReactNative и Qt. Если рассматривать «Классический Xamarin», то он работает только в iOS/

Android (в Windows C#/.NET и так являются родными), однако библиотека Xamarin.Forms отлично функционирует поверх родных Windows UWP API.

1.3. Архитектуры кроссплатформенных фреймворков

Итак, мы рассмотрели архитектуры iOS, Android и Windows UWP. Как вы могли заметить, все операционные системы имеют те или иные технические возможности по запуску кроссплатформенных приложений. Самое простое с технической точки зрения – использование WebView, которое есть у всех ОС (актуально для PhoneGap). Вторым вариантом является использование механизмов низкого уровня вроде OpenGL/DirectX и языка C/C++ (Qt) или скомпилированного Dart (Flutter) – это позволит получить высокую производительность, но не всегда нативный Look'n'Feel. Если же вам будет нужен полностью нативный пользовательский интерфейс и нативная производительность с минимальными накладными расходами, то здесь начинают задействоваться системные API верхнего уровня – такой подход реализуется только в Xamarin и ReactNative.

Чтобы лучше понять возможности и ограничения каждого из фреймворков, давайте рассмотрим, как архитектурно они устроены и какие из этого следуют возможности и ограничения.

1.3.1. PhoneGap

Решения на базе PhoneGap используют WebView и являются достаточно простыми с точки зрения реализации – создается небольшое нативное приложение, которое фактически просто отображает встроенный веб-браузер и single-page HTML. Нет никаких нативных контролов и прямого доступа к API – все интерфейсные элементы внутри веб-страницы просто стилизуются под родные. Для доступа к системной функциональности подключаются специальные плагины, которые добавляют JS-методы внутрь веб-браузера и связывают их с нативной реализацией на каждой платформе.

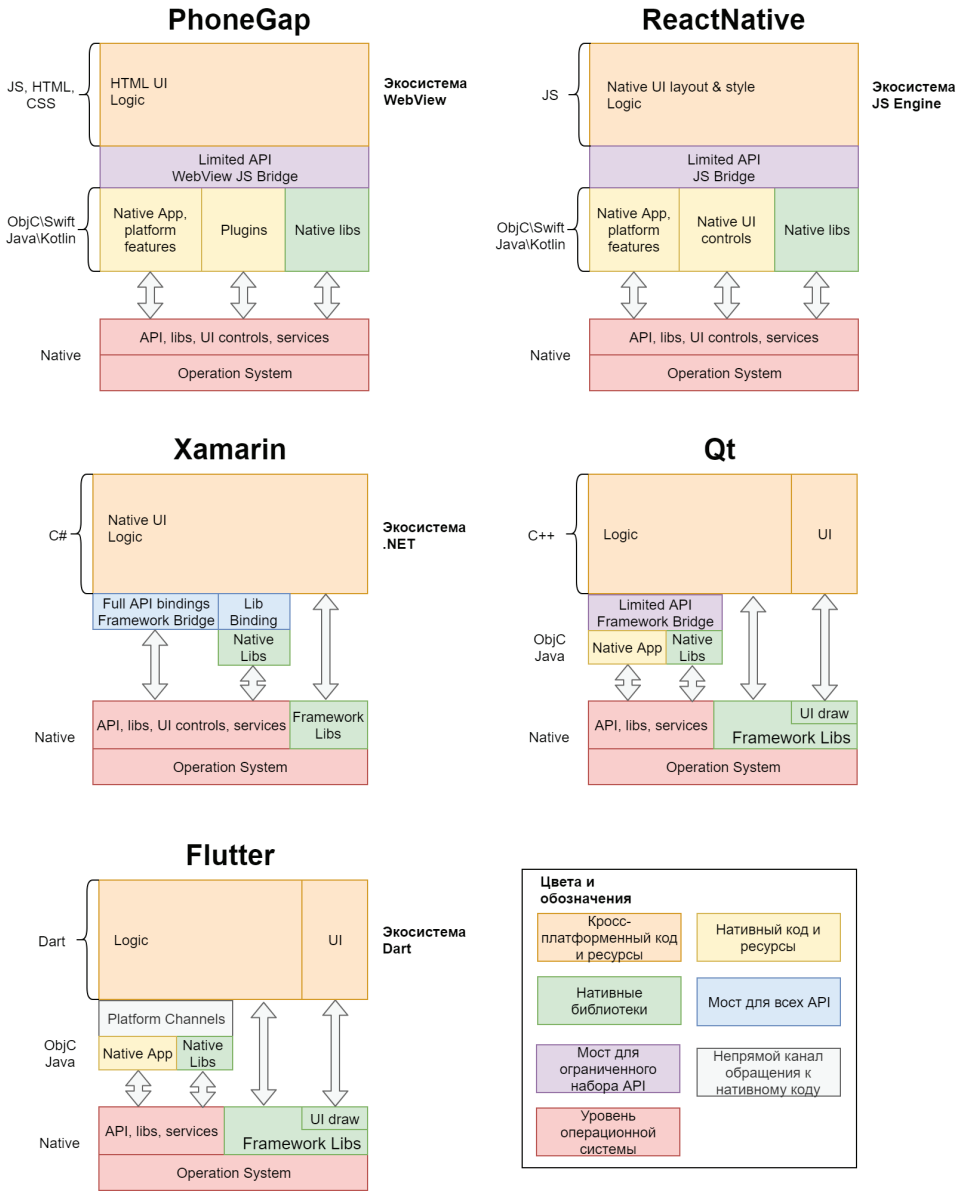


Рис. 1.7 ❖ Архитектуры кроссплатформенных фреймворков

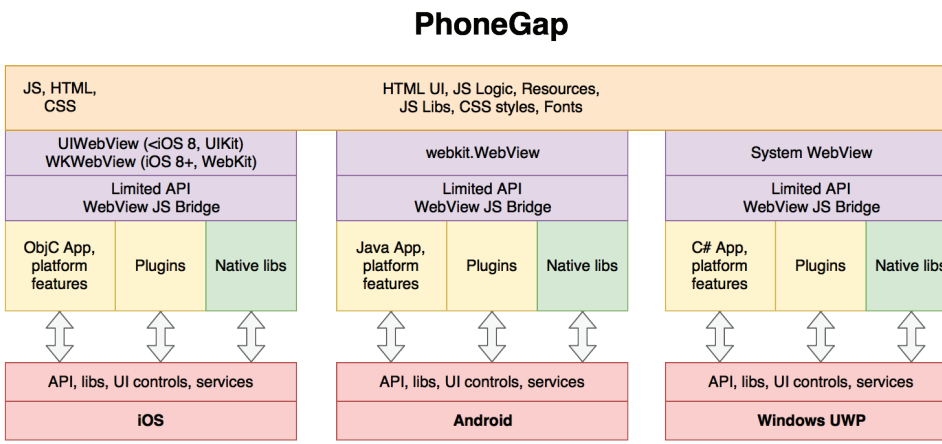


Рис. 1.8 ❖ Архитектура PhoneGap

Как видим, PhoneGap позволяет разделять практически весь код между платформами, однако все еще требуется реализация нативной части на Objective C и Java (и C# для Windows). Вся жизнь приложения проходит внутри WebView, поэтому веб-разработчики почувствуют себя как рыба в воде. До тех пор пока не возникнет потребность в платформенной функциональности – здесь уже будет необходимо хорошее понимание iOS и Android.

Также PhoneGap (он же Apache Cordova) используется в популярном фреймворке Ionic, который предоставляет большое количество готовых плагинов для системной функциональности.

Интерфейс приложений на основе WebView не является нативным, а только делается похожим на него с помощью HTML/CSS-стилей.

При разработке приложений на PhoneGap требуется опыт HTML, JavaScript, CSS, а также Objective C, Java и хорошие инженерные знания для интеграции нативной и кроссплатформенной частей. Пользовательский интерфейс организован по принципу одностраничного HTML – в реальных приложениях со сложным интерфейсом будут подергивания и подтормаживания (особенности мобильных WebView, которые еще и могут отличаться у разных производителей). Для передачи данных через мост их необходимо сериализовать/десериализовать в Json. В целом мост используется редко, так как вся жизнь приложения проходит внутри WebView.

Для передачи сложных структур данных и классов между нативной частью и WebView их необходимо сериализовать/десериализовать в формате JSON.

Напоследок отметим, что PhoneGap уже достаточно зрелое решение с большим количеством готовых плагинов.

1.3.2. ReactNative

Одним из интересных решений в области кроссплатформенной разработки мобильных приложений является ReactNative, созданный в Facebook. Этот фреймворк дает возможность использовать JavaScript для описания нативного интерфейса и логики работы приложений. Сам по себе JS-движок обеспечивает производительность, сопоставимую с нативной. Однако не стоит забывать, что и в архитектуре ReactNative присутствует мост, снижающий скорость работы с платформенной функциональностью и UI.

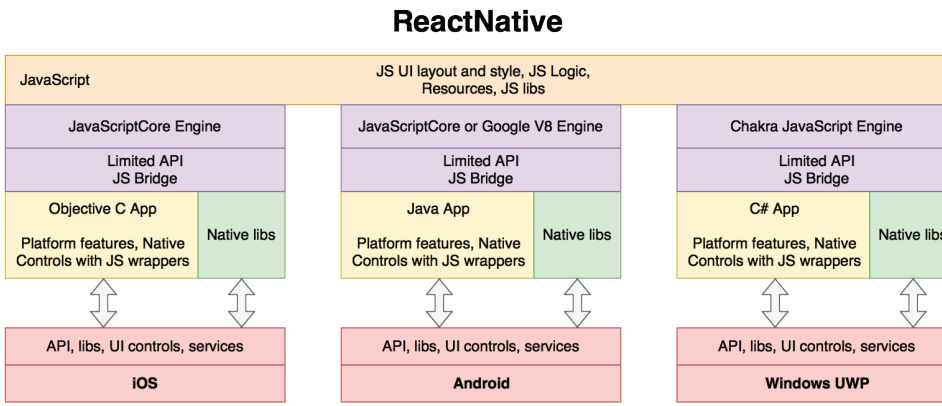


Рис. 1.9 ❖ Архитектура ReactNative

При создании приложений на ReactNative разработчику будет необходимо также реализовывать нативную часть на Objective C, Java или C#, которая инициализирует JS-движок и свой JS-код. Далее JS-приложение берет управление в свои руки и при помощи ReactNative начинает создавать нативные объекты и управлять ими из JavaScript. Стоит добавить, что архитектура ReactNative позволяет осу-

ществлять обновление JS-кода без перезапуска приложения (hot reloading). Это допускает обновление кроссплатформенной части без необходимости перепубликации приложений в AppStore и Google Play. Также можно использовать библиотеки из Npm и большое количество сторонних плагинов.

Необходимо учитывать, что из-за ограничений iOS (нет возможности реализовать JIT) код JavaScript на лету интерпретируется, а не компилируется. В целом это не сильно сказывается на производительности в реальных приложениях, но помнить об этом стоит.

Для передачи сложных структур данных и классов между нативной частью и JS-движком их необходимо сериализовать/десериализовать в формате JSON.

При создании приложений на ReactNative требуется опыт JavaScript, а также хорошие знания iOS и Android. Интеграцию нативной и кроссплатформенной частей легко сделать по официальной документации. Пользовательский интерфейс является полностью нативным, но имеет ограничения и особенности при стилизации из JS-кода, к которым придется привыкнуть. Для передачи данных через мост их необходимо сериализовать/десериализовать в Json. Плюс мост используется для управления нативными объектами, что также может вести к падению производительности при неэффективном использовании (например, часто менять свойства нативных UI-объектов из JS-кода при анимациях в ручном режиме).

Также следует учитывать юность фреймворка – имеются узкие места или ошибки, о которых узнаешь только во время разработки. И практически всегда требуется реализация нативной части на Objective C и Java.

1.3.3. Qt

Qt является одним из старейших кроссплатформенных фреймворков и используется очень широко для разработки embedded и десктопных приложений. Архитектура Qt позволяет портировать его в те операционные системы, которые имеют API для C++. И iOS, и Android (NDK), и Windows такой возможностью обладают, хотя и все со своими особенностями.

Один из главных плюсов Qt – собственная эффективная система отрисовки пользовательского интерфейса либо на базе растрового движка (например, CoreGraphics в iOS), либо на базе Open GL (ES). Именно это и делает фреймворк портируемым. То есть в Qt используются свои механизмы отрисовки UI – приложение будет выглядеть нативным настолько, насколько вы его сами стилизуете.

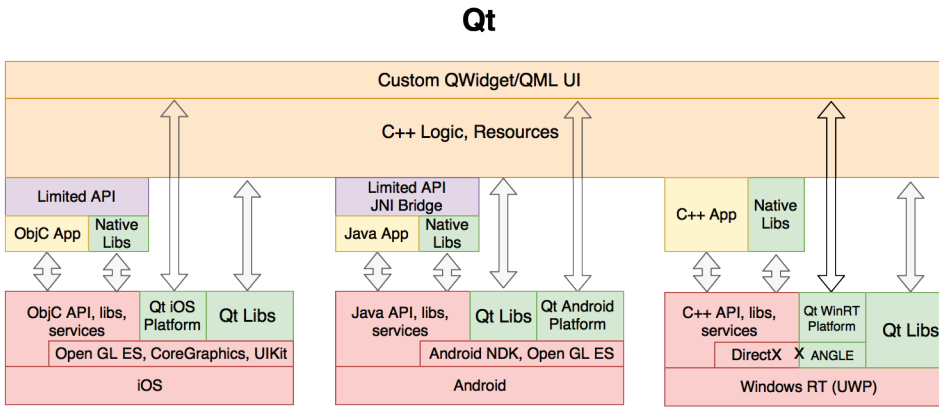


Рис. 1.10 ❖ Архитектура Qt

Как видим, на iOS используются стандартные модули CoreGraphics и UIKit для отрисовки пользовательского интерфейса. В Android ситуация чуть посложнее, так как Qt использует механизмы NDK для отрисовки UI, а для доступа к Java API и управления приложением используется уже знакомый нам мост JNI. Также в iOS и Android может использоваться Open GL ES для отрисовки QML или работы с 3D.

В Windows имеется прямой доступ к C++ API, и все работало бы отлично, если бы не необходимость использовать конвертацию вызовов Open GL ES в вызовы DirectX (растровая отрисовка не удовлетворяет по производительности, а Open GL ES нет в Windows UWP). В этом помогает библиотека ANGLE.

Интерфейс приложений на основе Qt не является нативным, а только делается похожим на него с помощью стилизации.

В целом Qt можно было бы рекомендовать как вещь в себе – только готовые модули самого фреймворка плюс платформонезависимые

библиотеки на C++. Но в реальных проектах его использовать будет очень непросто – неродной UI, отсутствуют сторонние компоненты (только библиотеки «из коробки»), возникают сложности при сборке и отладке приложения, а также при доступе к нативной функциональности. Из плюсов – высокая производительность кода на C++.

1.3.4. Flutter

Данный фреймворк был впервые представлен корпорацией Google только в 2015 году, однако быстро получил популярность со стороны разработчиков за свою простоту и высокую производительнсть. С точки зрения архитектуры Flutter похож на Qt – ядро этого фреймворка реализовано на C++, и пользовательский интерфейс создается с помощью собственного движка, не являясь нативным. Однако (в отличие от Qt) во Flutter реализованы простые механизмы интеграции с функциональностью операционной системы, не требуя большого количества оберток.

Так как нужные элементы пользовательского интерфейса рисуются на экране самим Flutter, то на нативном уровне приложение состоит из одного экрана, показывающего отрисованную движком Flutter картинку, и, как результат, имеет очень высокую производительность (до 120 кадров в секунду). Также сам Flutter берет на себя взаимодействие с пользователем и отлавливает жесты, касания и другие события.

Важно отметить, что приложение для Flutter необходимо разрабатывать на языке Dart, который очень похож на другие современные C-подобные языки, однако получил популярность только среди Flutter-разработчиков.

Для отрисовки пользовательского интерфейса в iOS/Android Flutter использует высокропроизводительный графический движок Skia, работающий поверх OpenGL или других низкоуровневых механизмов, задействующих возможности графических процессоров. С одной стороны, это позволяет получить очень отзывчивый пользовательский интерфейс, с другой – интерфейс может казаться ненативным и потребовать заметной доработки для реальных приложений.

Flutter в качестве моста для интеграции с операционной системой использует так называемые каналы платформы (Platform Channels),

которые позволяют передавать и обрабывать сообщения между Dart-кодом и нативной частью. Фактически это общая шина данных, через которую отправляются сообщения. При этом стоит помнить, что, как и любой мост, каналы платформы также преобразуют данные, что может привести к потерям производительности при неумелом использовании.

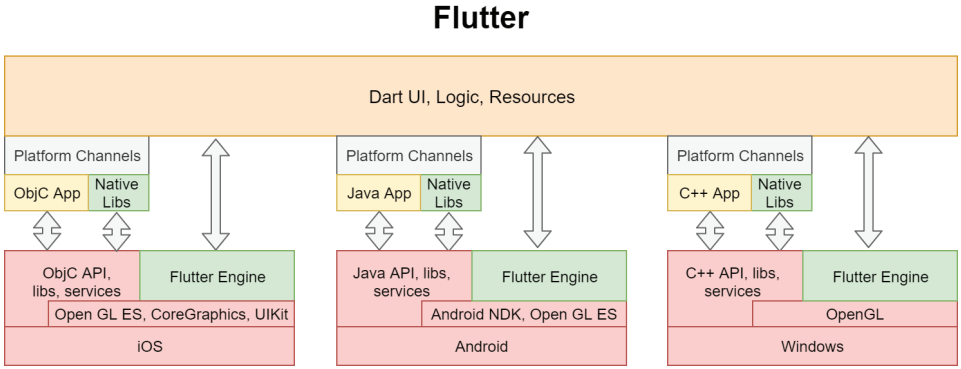


Рис. 1.11 ❖ Архитектура Flutter

Одним из плюсов Flutter является его легкая портируемость на новые платформы – уже сейчас заявлена поддержка iOS, Android, Windows, macOS, Linux и Web-приложений. По аналогии с ReactNative во Flutter реализованы механизмы автоматического обновления (hot reload) пользовательского интерфейса на этапе разработки – вы вносите изменения в код и сразу видите конечных результат без необходимости запуска приложения на эмуляторе/смартфоне. Также приложения на Flutter компилируются в машинный код (AOT-компиляция), что также положительно сказывается на производительности.

В целом Flutter является простым и удобным инструментом для создания мобильных приложений. Одновременно минусом и плюсом можно назвать использование языка Dart: современен, удобен и прост, но требует обучения и популярен только для Flutter. Ну и плюс юность фреймворка – простые приложения делаются быстро, но шаг влево-вправо – и можно нактнуться на баг или отстуствие готового компонента.

1.3.5. Xamarin

Xamarin сейчас доступен в open source и появился в качестве развития проекта Mono (**http://www.mono-project.com**), открытой реализации инфраструктуры .NET для Unix-систем. Изначально Mono поддерживался компанией Novell и позволял запускать .NET-приложения в Linux и других открытых ОС.

Для взаимодействия с родными (для C) интерфейсами операционных систем в Mono используется механизм P/Invoke (**http://www.mono-project.com/docs/advanced/pinvoke/**). На основе Mono были созданы фреймворки MonoTouch и MonoDroid, которые затем переименовали в Xamarin.iOS и Xamarin.Android и теперь вместе называют «классическим Xamarin» (Xamarin Classic).

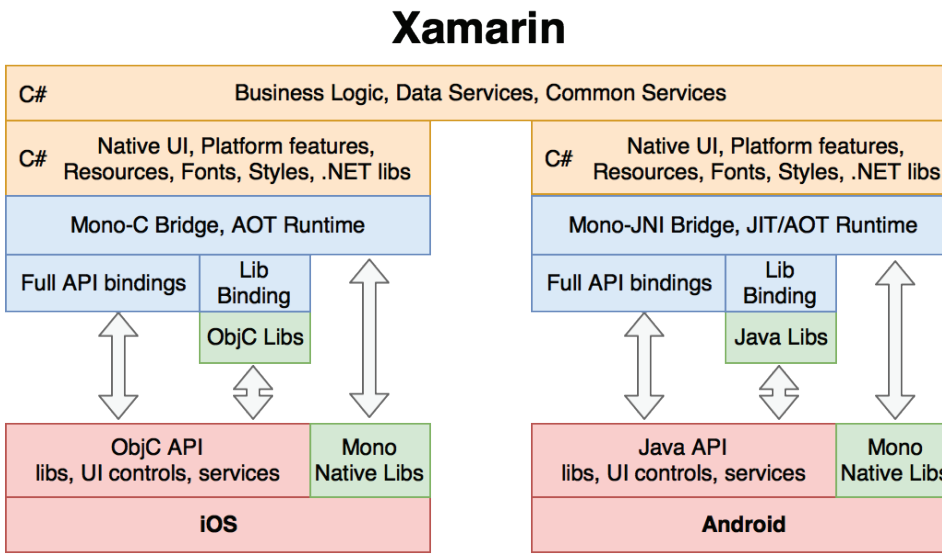


Рис. 1.12 ❖ Архитектура Xamarin

Классический Xamarin предоставляет полный доступ к нативным API, то есть можно создавать нативные приложения iOS/Android с помощью C# без единой строчки на Objective C и Java. Нативные библиотеки подключаются через механизм байндинга (Native Library Binding). Взаимодействие с ОС происходит через мост и механизм оберток (wrappers), однако нет необходимости сериализовать данные, так как

осуществляется автоматический маршалинг и есть возможность прямой передачи ссылок между средами Mono Managed и Native. Можно также использовать большое количество .NET-библиотек из NuGet.

Инфраструктура .NET/Mono предполагает использование JIT по аналогии с Java, когда приложение компилируется в промежуточный байт-код и уже потом интерпретируется во время исполнения. Но из-за ограничений iOS нет возможности использовать JIT, и поэтому байт-код Xamarin.iOS-приложений компилируется в нативный бинарный и статически линкуется вместе с библиотеками. Такая компиляция называется AOT (Ahead Of Time) и является обязательной в Xamarin.iOS. В Xamarin.Android же помимо AOT доступен и режим JIT, когда виртуальное окружение Mono работает параллельно с Dalvik/ART и компилирует код во время исполнения.

Как видим, общая база кода между платформами ограничивается бизнес-логикой и механизмами работы с данными. К сожалению, UI и платформенную функциональность приходится реализовывать отдельно для каждой платформы. В результате шарить можно не более 30–40 % от общей базы кода мобильных приложений. Для достижения большего результата необходимо использовать Xamarin.Forms.

Ключевым преимуществом классического Xamarin является использование языка C# для всей базы кода, включая UI-тесты, и, как следствие, разработчиков, которые уже хорошо знакомы с .NET. Также обязательным является хорошее знание и понимание механизмов iOS/Android, их классовых моделей, архитектур, жизненных циклов объектов и умение читать примеры на Objective C и Java.

Производительность C#-кода сопоставима с производительностью нативного кода в iOS/Android, но при взаимодействии с ОС используется мост, который может замедлять приложение при неэффективном использовании.

Приложение на Xamarin.iOS/Xamarin.Android обычно состоит из shared (общей) части, которая упаковывается в .NET-библиотеку, и платформенной части, которая имеет полный доступ к API, включая нативный пользовательский интерфейс. В платформенной части содержится описание экранов, ресурсы, стили, шрифты – практически 100%-ная структура нативного проекта на Objective C или Java, только на C#.

Классический Xamarin является достаточно зрелым решением и обеспечивает максимально близкий к нативному опыт разработки для C#-программистов и использованием привычных инструментов вроде Visual Studio.

1.3.6. Xamarin.Forms

Если у вас стоит цель максимизировать общую базу кода, то классический Xamarin здесь явно проигрывает всем остальным фреймворкам (PhoneGap, ReactNative, Flutter, Qt и их аналогам). Это понимали и в самом Xamarin, поэтому выпустили решение, позволяющее использовать единое описание UI и простые механизмы доступа к платформенным фичам, – Xamarin.Forms.

Библиотека Xamarin.Forms работает поверх описанного ранее классического Xamarin и фактически предоставляет механизмы виртуализации пользовательского интерфейса и дополнительную инфраструктуру.

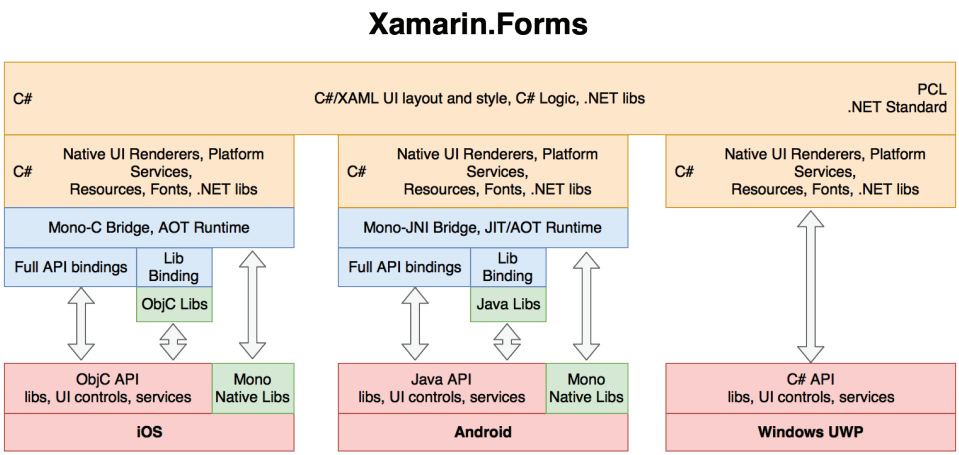


Рис. 1.13 ❖ Архитектура Xamarin.Forms

Xamarin.Forms (XF) решает своего рода задачу «последней мили», предоставляя единый API для работы с пользовательским интерфейсом в разных операционных системах (iOS, Android, Windows UWP/WPF, Linux Gtk#, Mac OS X, Tizen). При этом сам интерфейс остается полностью родным.

Для того чтобы лучше понять, как работает XF, давайте рассмотрим простую кнопку. Одним из базовых механизмов являются рендереры (renderers), благодаря которым при отображении кнопки Xamarin.Forms фактически на экран добавляется нативный контрол, а свойства XF-кнопки динамически пробрасываются в свойства нативной кнопки на каждой платформе. В ReactNative используются похожие механизмы.

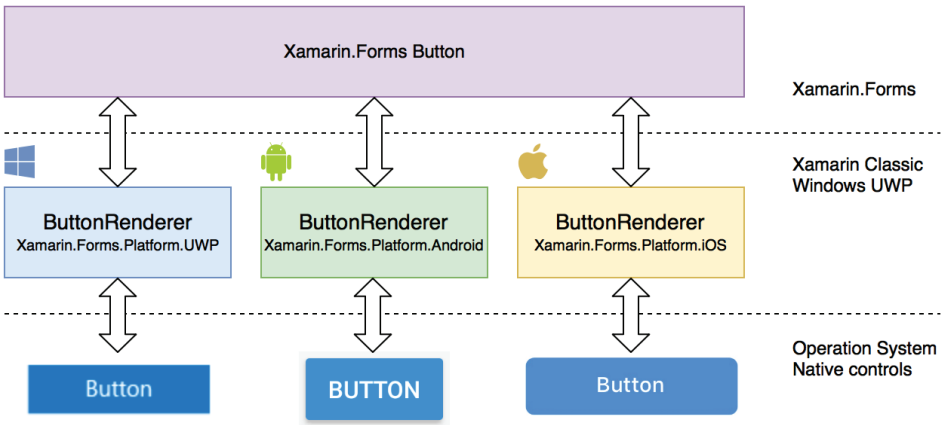


Рис. 1.14 ❖ Нативные контролы в Xamarin.Forms

Общая (shared) часть на Xamarin.Forms обычно реализуется в виде библиотеки (Portable/PCL или .NET Standard) и имеет доступ к базе компонентов в NuGet. Платформенная часть реализуется на базе Xamarin Classic и имеет полный доступ к API, а также возможность подключения сторонних библиотек. При этом общий процент кода между платформами обычно доходит до 85. Также Xamarin.Forms можно использовать в режиме Embedded для создания отдельных экранов и View внутри приложений на классическом Xamarin.iOS и Xamarin.Android.

Если вам будем достаточно уже доступных в Xamarin.Forms компонентов и плагинов, то не потребуется глубоких знаний в iOS/Android/Windows. В сообществе и NuGet также доступно большое количество готовых плагинов и примеров.

Несмотря на то что классический Xamarin является зрелым и стабильным решением, Xamarin.Forms еще достаточно молодая и активно развивающаяся над ним надстройка, поэтому могут проявляться проблемы и узкие места, с которыми стоит быть внимательным.

Глава 2

Процесс разработки и документация

Во время разработки программного обеспечения необходимо учитывать интересы сразу нескольких групп участников: бизнес-заказчики, проектировщики, тестировщики, разработчики и дизайнеры. Спецификой мобильных бизнес-приложений является их невысокая сложность в сравнении с корпоративным backend-сервисами – минимум бизнес-логики (она вся на сервере), почти всегда статический и относительно простой интерфейс.

Во время развития проекта команды могут столкнуться со следующими проблемами:

1) отсутствие или несоблюдение архитектурных паттернов, которое ведет к хаотичному расположению файлов в структуре решения. Также создаются излишние связи между классами и подсистемами. Все это усложняет и замедляет развитие продукта, так как требуется много времени на распутывание «лапши»;
2) сложность работы с проектной документацией – требуется долгое чтение для составления полной картины, детали все равно выпадают из головы;
3) отсутствие единой документации (кроме ТЗ) для всей команды, которая бы позволила проще находить общий язык и при этом сама была достаточно компактной и простой для восприятия. «Документация отдельно, код отдельно» – редко обозначения и названия из документации используются в коде, что усложняет его разработку и развитие.

В данной главе мы расскажем о пошаговом процессе подготовки технической документации, которая позволит *создать «скелет» проекта на основе пользовательского интерфейса*. Также мы остановимся на необходимом минимуме электронных документов, на которые могут в дальнейшем опираться участники команды.

2.1. Первичная документация

Ключевую идею, с которой мы начнем, можно коротко сформулировать следующим образом:

> **Мобильное бизнес-приложение** – это в первую очередь **пользовательский интерфейс** для взаимодействия с внешним сервисом.

При разработке технической документации на проект это стоит обязательно учитывать, так как интерфейс нагляден и на его основе проще проводить разделение проекта на разделы. Да и сама модель предметной области очень хорошо описывается интерфейсом – в ней необходимо учитывать в основном те данные (и их производные), которые вводятся пользователем, отображаются на экране и управляют его поведением. Бизнес-сценарии также напрямую завязаны на поведение пользовательского интерфейса.

В то же самое время большинство ТЗ готовится для бизнес-заказчиков и описывает не конкретные экраны или сервисы, а целые бизнес-сценарии и блоки функциональности. В дальнейшем эта документация и спецификации дизайна используются командой разработки. Для кодирования и последующей реализации используются многократные перечитки и пересказы ТЗ.

В следующих разделах мы рассмотрим минимально необходимый набор документов, которые позволят команде использовать простые чек-листы для контроля реализации.

Прежде чем мы перейдем к разбору артефактов и извлечению из них полезных данных, давайте рассмотрим весь процесс разработки целиком. Для простоты мы выберем итерационный процесс разработки, так как при использовании других методологий возникают те же самые классы задач, только последовательность их выполнения может отличаться.

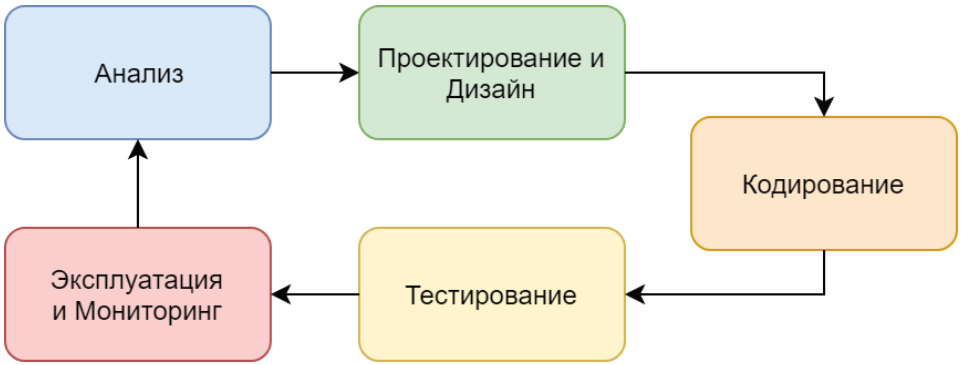


Рис. 2.1 ❖ Итерационный процесс разработки

Итак, у проекта обычно выделяют следующие производственные классы задач:

- анализ;
- проектирование и дизайн;
- кодирование;
- тестирование;
- эксплуатация.

Можно выделить и больше, но они по факту будут являться производными от обозначенных.

На этапе анализа производится поиск решения, описание общих требований к приложению. На выходе с этапа аналитики появляются спецификации, которые являются вводными для этапа проектирования.

Так как наше руководство предназначено в первую очередь для инженеров, то считаем, что бриф или базовое ТЗ у вас есть.

Дальше начинается самое интересное – проектирование пользовательского интерфейса. Этот этап является ключевым и при правильном подходе очень сильно облегчает и упрощает процесс разработки. Если же данный этап пропущен, то дальше успех проекта будет зависеть только от опыта команды.

> **На этапе проектирования самым важным является продумывание пользовательского интерфейса и создание схем экранов.**

Если начинать сразу с дизайна (вместо схем экранов), то придется постоянно переделывать его (дизайн) для согласования с заказчиком.

На этапе проектирования это сильно замедлит процесс. Дизайн фактически является производным от UX и наполняет изначальные схемы эмоциями, выправляет композицию, добавляет анимации и другие аспекты внешнего вида и визуального поведения. Схемы экранов в свою очередь создают структуру приложения и моделей данных – какие данные отображать, как они будут сгруппированы, как будут влиять на поведение интерфейса.

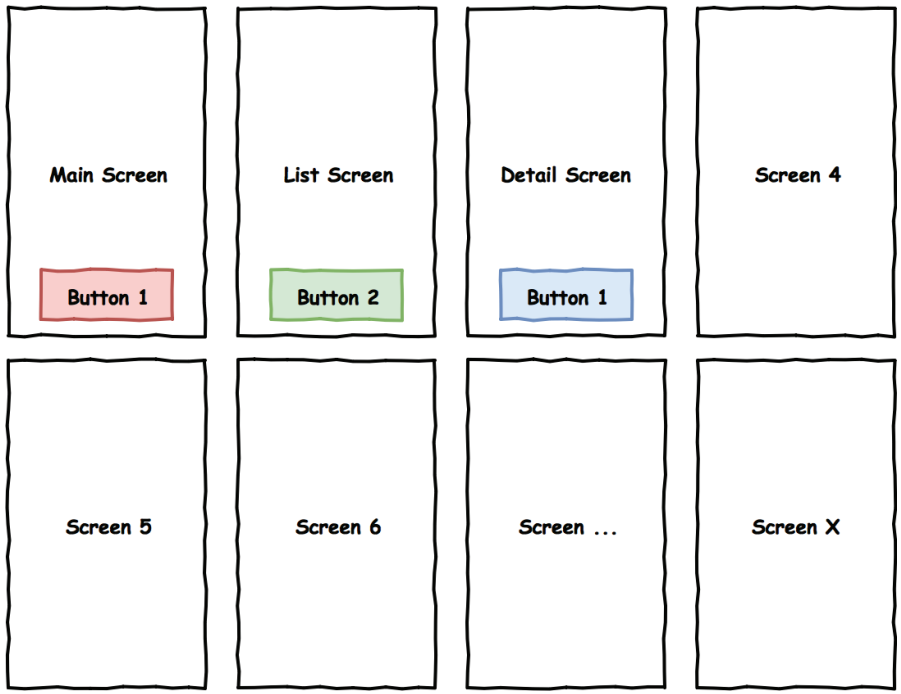


Рис. 2.2 ❖ Схемы экранов

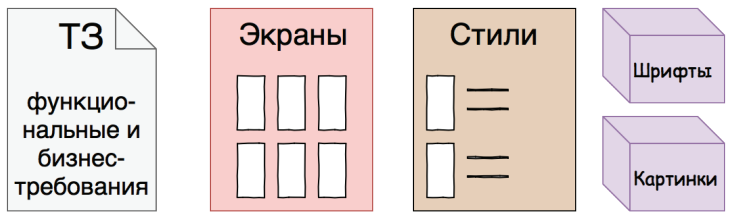


Рис. 2.3 ❖ Типовые документы для этапа разработки

На выходе с этапа проектирования будет получен комплект необходимых спецификаций и ресурсов, которые вместе с ТЗ уйдут разработчику. Минусом данного подхода является то, что разработчики должны самостоятельно создавать фундамент проекта, не имея четкой структуры и работая каждый со своим модулем. Все это приводит к мнимому усложнению кода, что ведет к быстрому росту технического долга.

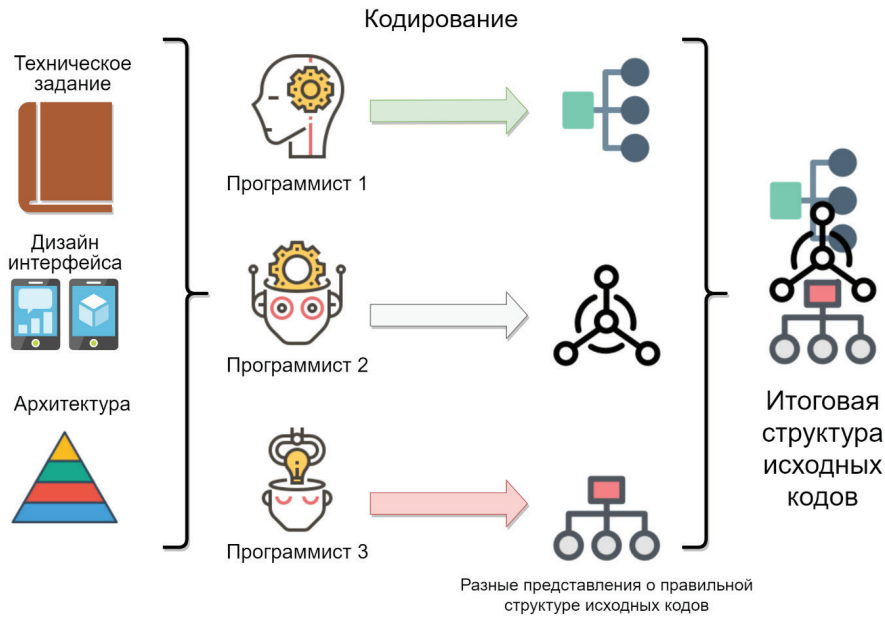


Рис. 2.4 ❖ Результат отсутствия технической документации при старте разработки

Далее будет описан подход, который позволит подготовить дополнительную техническую документацию и создать понятную структуру проекта, а также придерживаться единых терминов и обозначений при разработке.

На подготовку технической документации вам потребуется от 2 до 16 часов в зависимости от размеров проекта и квалификации сотрудника (первый раз дольше, потом быстрее). На создание «скелета» проекта с помощью описанных далее алгоритмов – от 6 до 24 часов.

Итого при старте проекта можно выделить 1–5 дней на проработку нужной документации и закладку правильного фундамента. Это заметно снизит сложность проекта и позволит всем членам команды говорить на одном языке. Важно помнить, что документацию и структуру проекта требуется поддерживать в актуальном состоянии, благо на это необходимо всего несколько часов в месяц.

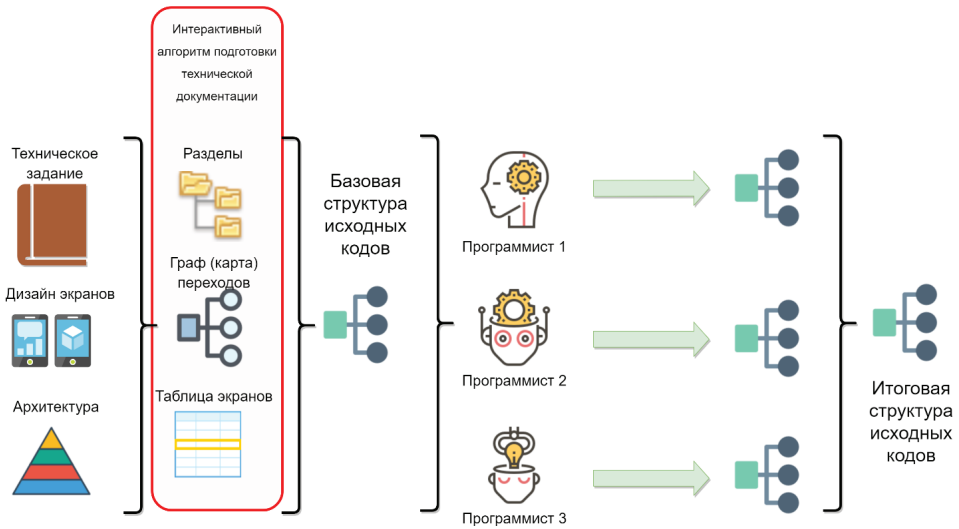


Рис. 2.5 ❖ Результат добавления алгоритма (процесса) для подготовки технической документации и создания «скелета» проекта на ее основе

2.2. Экраны, данные и логика

Напомним еще раз, что мобильные приложения – это в первую очередь пользовательский интерфейс, поэтому и проектирование лучше начать со схем экранов и последовательности переходов между ними. Это необходимо, для того чтобы определить набор шагов, которые предстоит пройти пользователю для получения желаемого результата. Ведь бизнес-приложение создается для определенного набора ключевых сценариев (последовательности действий пользователя), с помощью которых человек может решить свои задачи.

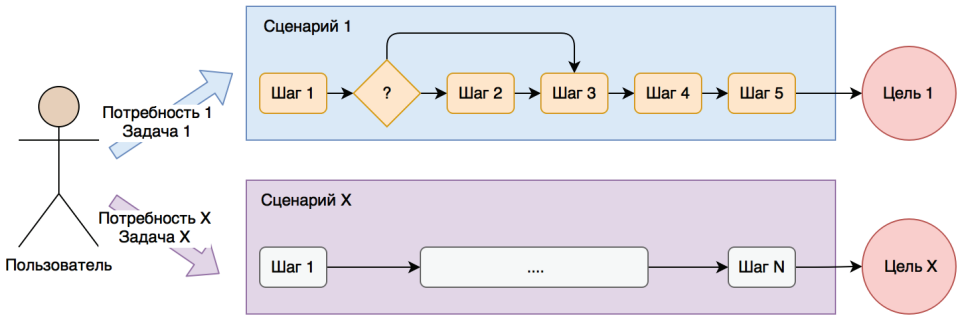


Рис. 2.6 ❖ Пользовательские бизнес-сценарии

Так как среднее время контакта человека со смартфоном составляет всего несколько минут, то количество шагов в бизнес-приложениях не должно быть большим – для пользователя в первую очередь важно получить результат (выполнить задачу или удовлетворить потребность) за время контакта с устройством. Для сложных приложений с большим количеством функциональных возможностей следует учитывать этот фактор. Хорошим выбором станет разделение приложения на относительно короткие сценарии не более 10 шагов каждый.

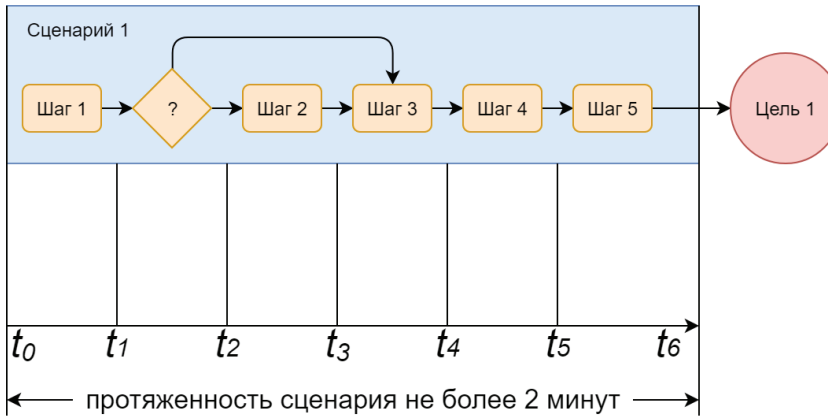


Рис. 2.7 ❖ Длительность бизнес-сценариев в мобильных приложениях

Для того чтобы определить глубину ключевых сценариев, можно использовать карту переходов и состояний, подробнее о которой будет рассказано немного позже. Но для начала требуется привести в порядок структуру интерфейса.

2.2.1. Группировка экранов и сквозное именование

Итак, у нас на руках есть схемы экранов от проектировщиков/дизайнеров и последовательность переходов между ними.

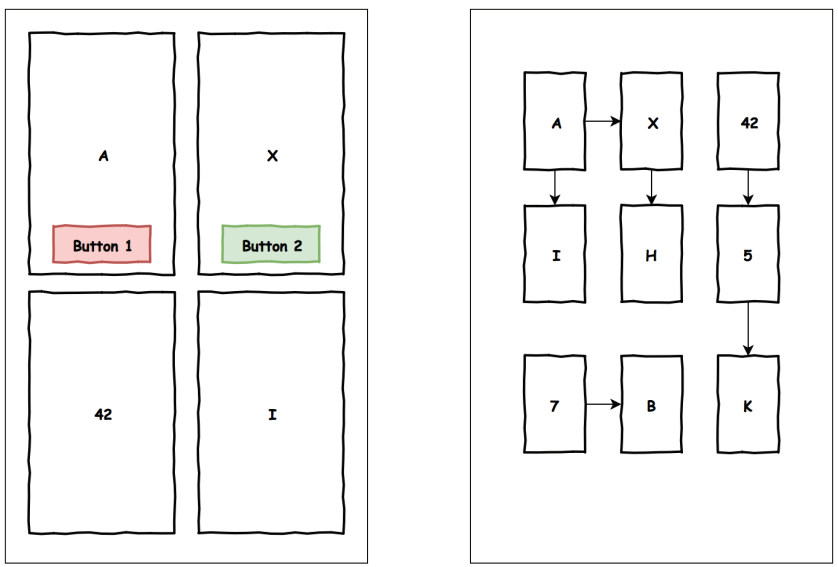
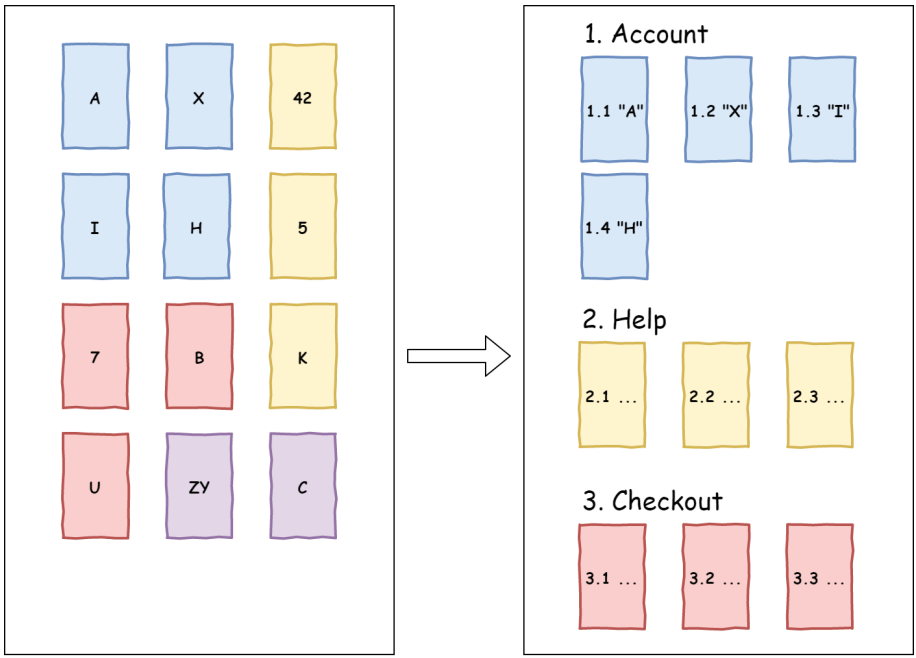


Рис. 2.8 ❖ Список экранов и последовательность переходов

Для того чтобы разбить приложение на части (разделы предметной области), мы пойдем от экранов. Еще раз напомним, что мобильное приложение – это в первую очередь интерфейс взаимодействия с пользователем, поэтому наши экраны и являются прямым отражением доступной ему модели предметной области.



Названия файлов и структура **"от дизайнера"**

Названия файлов и структура **"для программиста"**

Рис. 2.9 ❖ Структурирование дизайна экранов

Первым делом необходимо выделить экраны, которые связаны между собой, – обычно они должны идти друг за другом в пользовательских сценариях. Например, часто в приложениях можно выделить раздел Account с просмотром и редактированием всей информации, связанной с профилем пользователя.

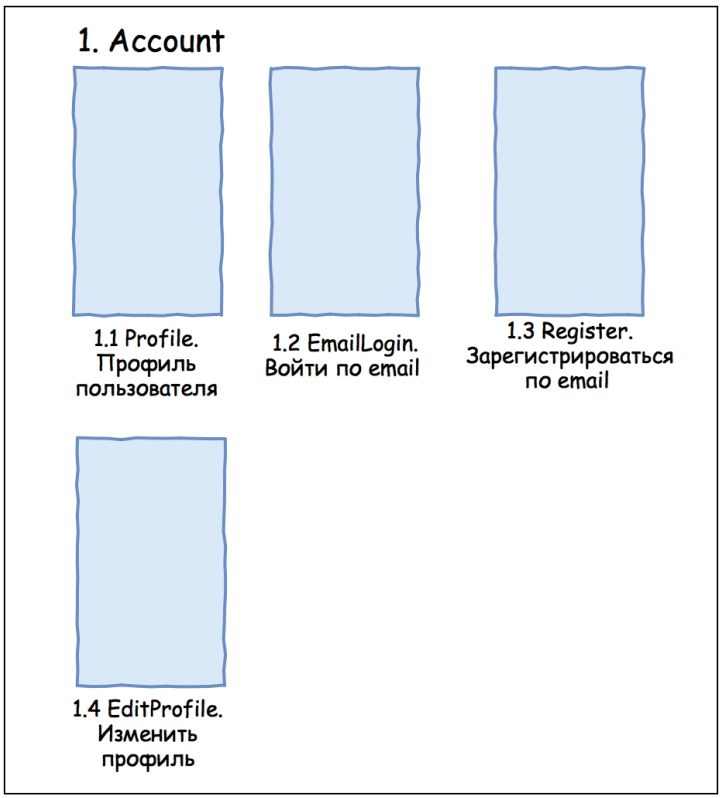


Рис. 2.10 ❖ Именование и нумерация экранов

Если вы опытный программист, то легко справитесь с разделением списка экранов на связанные разделы. В любом случае потребуется немного практики.

Итак, у нас могут получиться следующие разделы:

1) Account;
2) Help;
3) Checkout;
4) Catalog.

Каждый раздел должен иметь название и номер. Названия разделов следует использовать для горизонтального разделения слоя работы с данными, бизнес-логики и пользовательского интерфейса. Это позволит в дальнейшем проще развивать проект.

Слой работы с данными (группы методов для различных серверных API) в этом случае разделится на разделы (репозитории, если вам так привычнее), каждый из которых будет обслуживать свой набор экранов:

```
DAL\DataServices (Repositories)
AccountDataService.cs (или AccountRepository.cs)
HelpDataService.cs
CheckoutDataService.cs
CatalogDataService.cs
```

В дальнейшем каждый из репозиториев может полностью скрывать всю работу с сервером, дисковым кешем и локальной СУБД. Это позволит на уровне бизнес-логики работать с репозиториями как с черными ящиками.

Дальше предстоит пронумеровать и назвать экраны (страницы, окна). На выходе у нас получится древовидная (хотя и плоская) структура интерфейса без учета последовательности переходов между экранами и их вложенности.

Имена экранов будут использоваться у нас в названиях классов для соответствующих страниц (Page) и ViewModel (или Controller для MVC):

```
1.1 Profile
ProfilePage
ProfileViewModel

1.2 EmailLogin
EmailLoginPage
EmailLoginViewModel
```

В первую очередь это важно для разработчиков, которые фактически получают готовую структуру проекта:

- слой доступа к данным разбивается на разделы приложения – создаем структуру Data Access Layer (DAL);

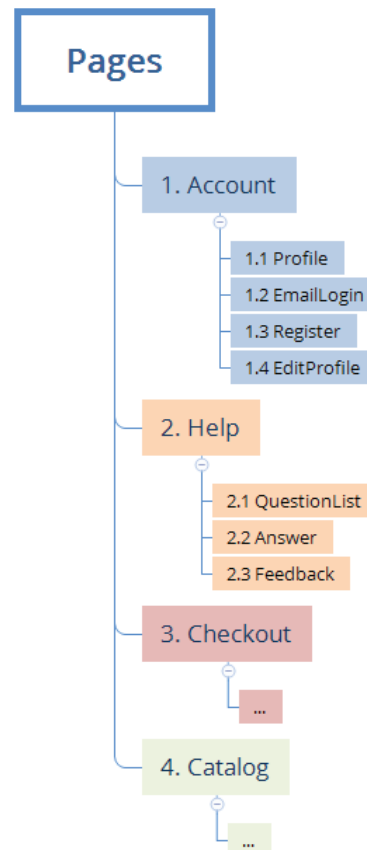


Рис. 2.11 ❖ Структура экранов по разделам

- добавляем нужные Pages и ViewModels – это создает структуру слоев работы с пользовательским интерфейсом (UI) и бизнес-логикой (BL).

Как видим, уже вырисовывается «скелет» проекта. Слой DAL можно легко выделить в отдельную библиотеку. Если же у вас используется типовая архитектура или шаблон проекта (Base-классы, NavigationService и т. д.), то считайте, что костяк приложения у вас уже имеется.

Пример структуры проекта представлен ниже.

```
UI (User Interface, пользовательский интерфейс)
    |Pages
        |Account
            ProfilePage.xaml
            ...

BL (Business Logic, бизнес-логика)
    |ViewModels
        |Account
            ProfileViewModel.cs
            ...

DAL (Data Access Layer, доступ к данным)
    |DataObjects (Models)
        ProfileObject.cs (ProfileModel.cs)
        ProductObject.cs

        ...
    |DataServices (Repositories)
        AccountDataService.cs
        ...
```

Для того чтобы дальше реализовывать поведение экранов, нам потребуется дополнительная информация, поэтому продолжим знакомство с необходимыми артефактами.

2.2.2. Таблица экранов

После того как у нас есть схемы экранов, мы также можем провести их анализ до фактического старта разработки. Следующим полезным артефактом для нас станет таблица экранов, оперировать которой будут не только разработчики, но и тестировщики. В сводной таблице легко собрать воедино всю текстовую информацию. Ключевыми столбцами таблицы для нас станут:

1) номер экрана;
2) краткое название (Name);
3) список возможных состояний (States);
4) поля ввода для валидации (Validation);
5) описание экрана и его поведения (Behavior).

Как видим, в представленном наборе полей собрана та информация, которая позволит корректно проверить работу каждого экрана в отдельности. Можно также каждому разделу присвоить свой цвет – это упростит работу с картой переходов и состояний.

	A	B	C	D	E	F
1		**Name**	**States**	**Validation**	**Behavior**	**AutomationId**
2	**1**	**Account**				
3	1.1	Profile	Loading Normal NoInternet NoData		Необходимо отобразить краткую информацию о профиле, список последних уведомлений и активностей, а также текущий рейтинг.	EditButton
4	1.2	EmailLogin		Email (>5 and <160, default email mask) Password (>2 and <24, any chars)	Если пользователь ввел какое-либо неверное значение в поля "Ваш Email" и "Пароль" и нажал "Войти", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Войти", то отображается всплывающий индикатор загрузки и в случае успешной авторизации осуществляется переход на экран 1.1 Profile. Если авторизоваться не удалось, то должен отобразиться соответствующий всплывающий диалог.	EmailField PasswordField LoginButton RegisterButton
5	1.3	Register		Email (>5 and <160, default email mask) Password (>2 and <24, any chars) Phone (12 digits, +X (XXX) XXX-XX-XX) FirstName (>2 any chars) LastName (>2 any chars)	Если пользователь ввел какое-либо неверное значение в поля "Ваш Email", "Пароль", "Телефон", "Имя", "Фамилия" и нажал "Зарегистрироваться", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Зарегистрироваться", то отображается всплывающий индикатор загрузки и в случае успешной регистрации осуществляется переход на экран 1.1 Profile. Если зарегистрироваться не удалось, то должен отобразиться соответствующий всплывающий диалог.	EmailField PasswordField PhoneField FirstNameField LastNameField RegisterButton
6	1.3	EditProfile		Phone (12 digits, +X (XXX) XXX-XX-XX) FirstName (>2 any chars) LastName (>2 any chars)	Если пользователь ввел какое-либо неверное значение в поля "Телефон", "Имя", "Фамилия" и нажал "Сохранить", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Сохранить", то отображается всплывающий индикатор загрузки и в случае успешного сохранения осуществляется переход на экран 1.1 Profile. Если обновить данные профиля не удалось, то должен отобразиться соответствующий всплывающий диалог.	PhoneField FirstNameField LastNameField SaveButton
7						
8	**2**	**Help**				

Рис. 2.12 ❖ Таблица экранов

Дополнительно в эту таблицу могут быть добавлены следующие столбцы:

6) список всплывающих уведомлений (alerts, sheets, dialogs);
7) идентификаторы UI-контролов (например, LoginButton) для написания автоматизированных UI-тестов;
8) используемые модели (Models/Data Objects) данных;
9) используемые на каждом экране методы DAL;
10) используемые стили (Styles).

О каждом экране по столбцам достаточно просто вписывать короткие обозначения, которые в дальнейшем будут использоваться в программном коде и понятны в первую очередь разработчикам. Кроме столбца **Behaviour** (Описание экрана и его поведение), здесь ограничений лучше не делать.

Отдельно остановимся на состояниях экранов. Большинство современных приложений работает через интернет, поэтому стоит корректно отображать пользователю информацию о состоянии загрузки, а именно:

- индикатор прогресса загрузки;
- загруженные данные;
- сообщение об отсутствии интернет-соединения;
- сообщение об ошибке (недоступен сервер, ошибки сервера);
- заглушку, если сервер вернул пустые данные (например, пустой список).

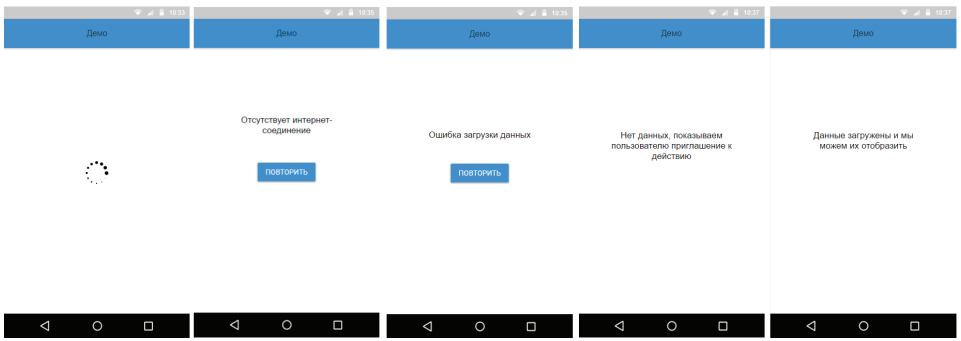


Рис. 2.13 ❖ Состояния экранов

Хорошей практикой считается, если каждый экран, загружающий данные из сети (или из локальной СУБД), будет корректно отображать пользователю каждое из описанных состояний. Фактически отдельное состояние описывается своим набором визуальных элементов (тексты, картинки, кнопки, другие элементы), и на уровне программного кода легко управлять переключением из одного состояния в другое. Также можно фиксировать негативные сценарии (ошибка загрузки, пустые данные) для дальнейшего анализа и устранения на стороне сервера или приложения. Реализация компонента

StateContainer, который позволяет управлять состояниями экрана, описана в главе 8.

Можно взять за правило и на всех экранах, загружающих данные, добавлять переключение состояний. Это упростит взаимодействие пользователя с приложением. Можно также использовать различные анимации или графику в негативных состояниях (ошибки, пустые данные), чтобы сгладить эффект.

Итак, у нас уже есть схемы экранов, список Page и ViewModel, а также детальная информация по каждому экрану. Каркас приложения можно построить, однако сейчас у нас экраны описаны независимо друг от друга, и нет четкой и понятной последовательности переходов. Поэтому следующим полезным артефактом для нас станет карта переходов и состояний.

2.2.3. Карта переходов и состояний

Для того чтобы лучше понять основные пользовательские сценарии, а также обозначить связи между экранами, можно использовать карту переходов и состояний. Плюсами карты являются ее компактность и наглядность. Даже для больших проектов карту переходов можно распечатать на принтере A4 и повесить над рабочим столом.

Итак, карта переходов начинается с точки старта – момента запуска приложения пользователем. Точек старта может быть несколько, например один вариант запуска для авторизованного пользователя, второй – для неавторизованного, а третий – из Push-уведомления.

Дальше добавляются прямоугольники для каждого экрана и обозначаются стрелками последовательности переходов. Можно добавить идентификаторы (AutomationId) кнопок или событий, из-за которых произошел переход, и для наглядности еще указать данные, которые будут передаваться на новый экран.

Также у нас уже есть таблица экранов (предыдущая глава), где обозначены возможные состояния каждого из них, которые необходимо отобразить на карте переходов. Это позволит лучше понять возможные прерывания пользовательских сценариев, например в случае ошибок или пустых данных. Если состояние прерывает (человек не может идти дальше) пользовательский сценарий, то обозначаем его минусом «-», если не прерывает – плюсом «+». Стрелочки «назад» можно не добавлять.

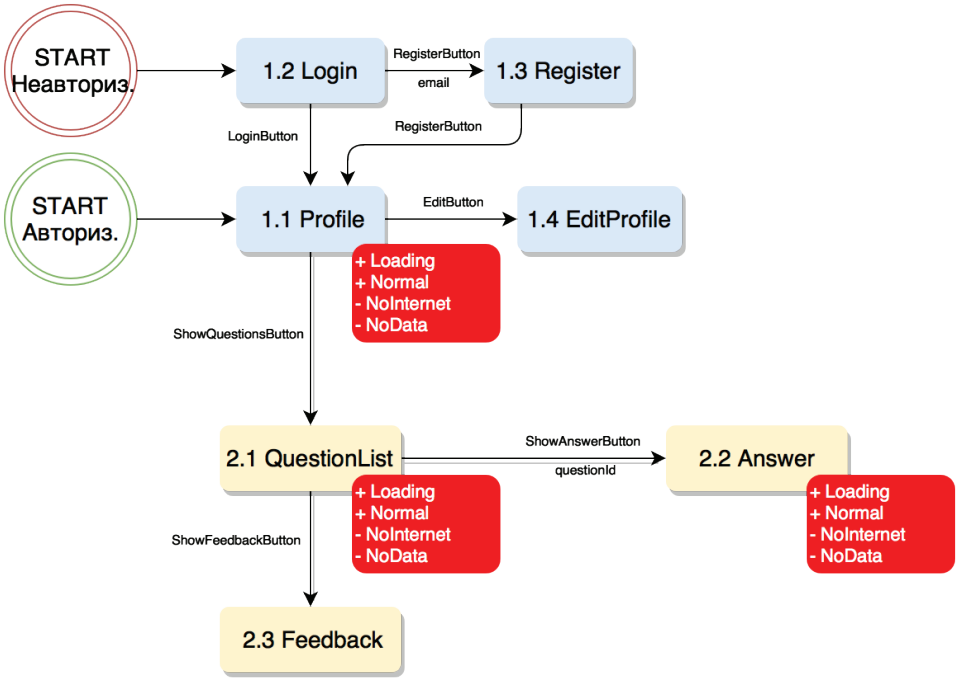


Рис. 2.14 ❖ Карта переходов

Как видим, теперь у нас имеется практически вся необходимая для разработки информация в компактном виде. Эти три онлайн-документа (список экранов, таблица экранов, карта переходов) могут обновляться по мере развития проекта.

Для создания описанных выше артефактов нам будет достаточно три онлайн-инструмента:

- текстовый редактор (Microsoft Word, Google Docs);
- табличный редактор (Microsoft Excel, Google Sheets);
- графический редактор (Draw.io, Microsoft Visio, Google Draw).

На подготовку каждого из артефактов уходит не больше одного дня, зато в дальнейшем это очень сильно упрощает процесс разработки, тестирования и развития продукта. За время медитативной подготовки документов и схем команда глубже понимает проект целиком и может уже финально оценить сложность и длительность его разработки (цифры для внутреннего использования).

2.3. Стили и ресурсы

В современных приложениях с пользовательским интерфейсом широко применяются таблицы стилей. В инструментах разработки (и в нативных, и в кроссплатформенных) этот механизм также реализован, поэтому до старта кодирования лучше сделать единую таблицу стилей, так как в противном случае есть высокий риск взрывного увеличения их количества, когда каждый разработчик придумывает свои имена для внешне одинаковых элементов пользовательского интерфейса. Чтобы этого избежать, необходимо заранее подготовить если и не описания, то хотя бы названия стилей с привязкой к дизайну.

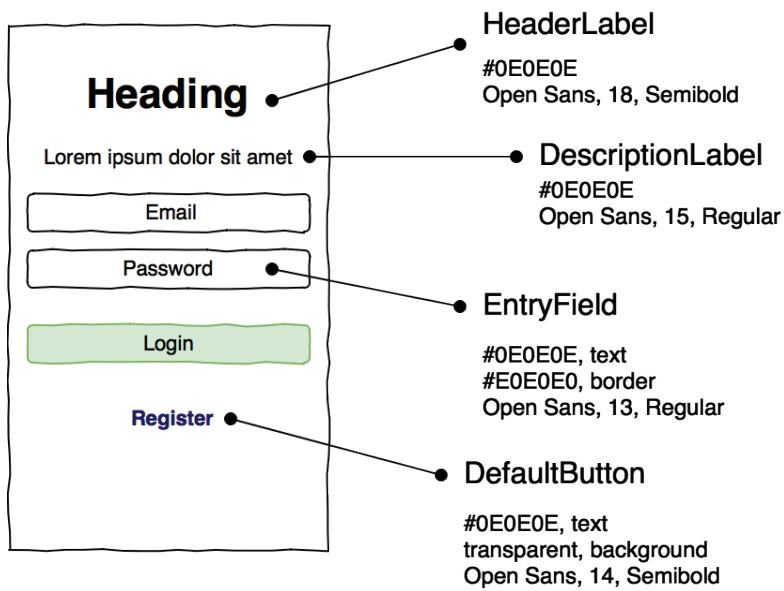


Рис. 2.15 ❖ Стили в приложении

Сама по себе информация по стилю должна приходить от дизайнера (например, через сервис Zeplin.io или Sketch). Для нас же важен на текущем этапе не просто набор свойств для каждого типа объекта (цвет шрифта у текста, задний фон у страницы и т. д.), а их связь со страницами. Необходимо для всех объектов на экране указать названия связанных стилей.

	A	B	E	F	G
1		**Name**	**Behavior**	**AutomationId**	**Styles**
2	**1**	**Account**			
3	1.1	Profile	Необходимо отобразить краткую информацию о профиле, список последних уведомлений и активностей, а также текущий рейтинг.	EditButton	HeaderLabel DescriptionLabel EntryField DefaultButton
4	1.2	EmailLogin	Если пользователь ввел какое-либо неверное значение в поля "Ваш Email" и "Пароль" и нажал "Войти", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Войти", то отображается всплывающий индикатор загрузки и в случае успешной авторизации осуществляется переход на экран 1.1 Profile. Если авторизоваться не удалось, то должен отобразиться соответствующий всплывающий диалог.	EmailField PasswordField LoginButton RegisterButton	HeaderLabel DescriptionLabel EntryField DefaultButton
5	1.3	Register	Если пользователь ввел какое-либо неверное значение в поля "Ваш Email", "Пароль", "Телефон", "Имя", "Фамилия" и нажал "Зарегистрироваться", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Зарегистрироваться", то отображается всплывающий индикатор загрузки и в случае успешной регистрации осуществляется переход на экран 1.1 Profile. Если зарегистрироваться не удалось, то должен отобразиться соответствующий всплывающий диалог.	EmailField PasswordField PhoneField FirstNameField LastNameField RegisterButton	HeaderLabel DescriptionLabel EntryField DefaultButton
6	1.3	EditProfile	Если пользователь ввел какое-либо неверное значение в поля "Телефон", "Имя", "Фамилия" и нажал "Сохранить", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Сохранить", то отображается всплывающий индикатор загрузки и в случае успешного сохранения осуществляется переход на экран 1.1 Profile. Если обновить данные профиля не удалось, то должен отобразиться соответствующий всплывающий диалог.	PhoneField FirstNameField LastNameField SaveButton	HeaderLabel DescriptionLabel EntryField DefaultButton
7					
8	**2**	**Help**			

Рис. 2.16 ❖ Таблица экранов со списком стилей

На основе этой таблицы можно сразу описывать необходимые стили еще на ранних этапах проекта, что позволит на этапе разработки сфокусироваться на компоновке, а не стилизации элементов интерфейса.

Следующим шагом мы рассмотрим ту функциональность, которая может быть скрыта от пользователя.

2.4. Скрытая функциональность

Отдельно на этапе технического проектирования необходимо выделить фоновую (скрытую) функциональность вроде механизмов

работы с СУБД, кешера, обработчика Push-уведомлений, фоновой синхронизации или сервиса управления корзиной. Данная функциональность просто не должна ускользнуть на этапе старта работ, поэтому ее лучше описать отдельно.

В каком виде все это описывать – решать вам, но это не мешает нам указывать названия фоновых сервисов (реализуют нужную функциональность), которые мы в дальнейшем будем использовать в коде. Например:

`BL\Services\CartService`	Фоновый Singleton-сервис синхронизации корзины. Запускается при старте приложения. Должен поддерживать работу в многопоточном окружении с очередью сообщений об изменении корзины
`BL\Services\CatalogCacheService`	Реализует табличное хранилище в виде Singleton-сервиса для временного кеширования списка товаров и информации о продавце между экранами
`DAL\DataServices\SyncDataService`	Реализует механизмы поэтапной синхронизации данных. Должен поддерживать работу в ограниченном окружении нативного фонового запуска

2.5. Пользовательские сценарии

Как мы уже отмечали в главе 1, все приложения создаются для решения определенных пользовательских задач. То есть фактически это набор переходов между экранами и действий пользователя (нажатие на кнопку, выбор элемента и т. д.) на них. Поэтому и наши сценарии мы также привяжем к экранам и элементам пользовательского интерфейса.

1. Account

Case 1.1 Авторизация по email/паролю

Начальные условия		Устройство имеет стабильное и высокоскоростное интернет-соединение, приложение запущено в первый раз или пользователь вышел из своего профиля. Отображается экран 1.2 EmailLogin
Ожидаемый результат		Пользователю отображается экран 1.1 Profile в состоянии Normal
UI	**Действия пользователя**	**Реакция системы**
1.2	Ввел Email, Пароль и нажал кнопку **Войти**	Если введены корректные email и пароль, то отобразится индикатор загрузки и осуществится автоматический переход на экран 1.1 Profile. В случае ввода неверных данных должны отображаться сообщения об ошибках
1.1	Ожидание	На экране должны отобразиться загруженные данные из профиля пользователя (состояние экрана Normal)

Рис. 2.17 ❖ Пример описание пользовательского сценария

Как видим, в сценарии указаны элементы пользовательского интерфейса, что позволит в будущем проще переложить их на автоматизированное UI-тестирование.

Теперь у нас есть и вся необходимая информация как для разработки, так и тестирования (ручного и автоматизированного).

2.6. Финальный набор артефактов и их обновление

Итак, помимо первоначального ТЗ и дизайна мы также получили набор дополнительных артефактов.

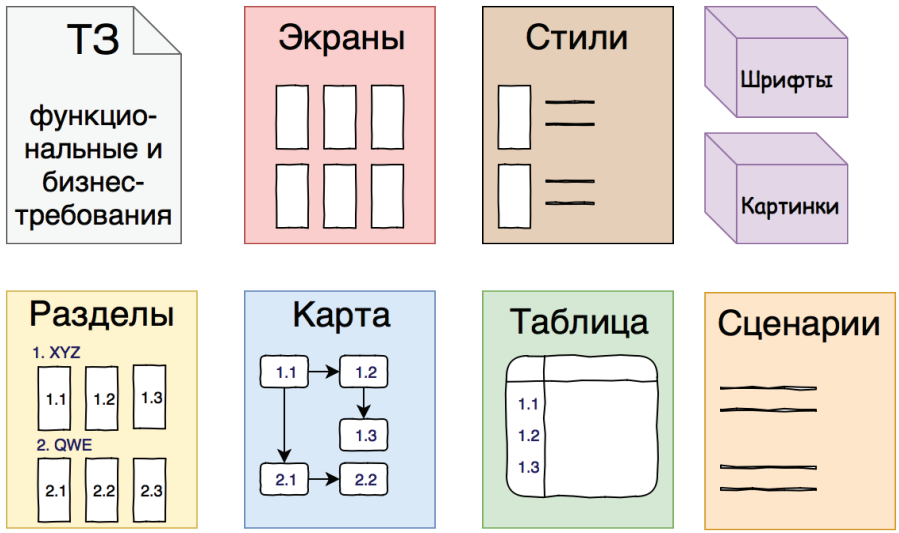
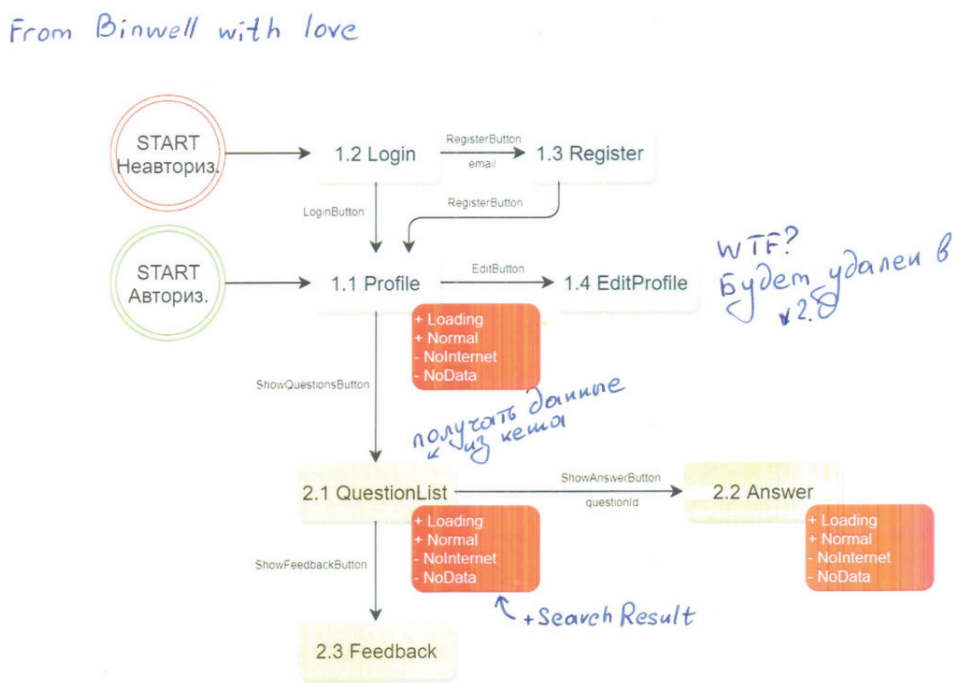


Рис. 2.18 ❖ Финальный список артефактов после технического проектирования

Каждый из артефактов требует относительно немного времени на создание и не требует сложных знаний и опыта. Вместе они полностью описывают интерфейс приложения и весь пользовательский опыт. Данные документы могут быть созданы даже специалистами средней квалификации, а их использование очень сильно упрощает и структурирует дальнейшую разработку.

В независимости от того, будете ли вы использовать простые онлайн-документы или специальную систему управления требованиями, следует обновлять документацию при каждом крупном релизе. Стоит контролировать, чтобы указанные в документе названия использовались всей командой при написании кода, тестовых скриптов и планировании работ (например, дробить задачи по разделам).

Небольшой объем (мало букв) документации позволяет использовать ее в качестве чек-листа во время разработки. Очень полезным для понимания проекта оказывается карта переходов и состояний, поэтому ее можно распечатать в бумажном виде и делать пометки прямо на листе.



Name Account	States	Validation	Behavior	AutomationId	Styles
.1 Profile	Loading Normal NoInternet NoData		Необходимо отобразить краткую информацию о профиле, список последних уведомлений и активностей, а также текущий рейтинг.	EditButton	HeaderLabel DescriptionLabel EntryField DefaultButton
.2 EmailLogin		Email (>5 and <160, default email mask) Password (>2 and <24, any chars)	Если пользователь ввел какое-либо неверное значение в поля "Ваш Email" и "Пароль" и нажал "Войти", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Войти", то отображается всплывающий индикатор загрузки и в случае успешной авторизации осуществляется переход на экран 1.1 Profile. Если авторизоваться не удалось, то должен отобразиться соответствующий всплывающий диалог.	EmailField PasswordField LoginButton RegisterButton	HeaderLabel DescriptionLabel EntryField DefaultButton
.3 Register		Email (>5 and <160, default email mask) Password (>2 and <24, any chars) Phone (12 digits, +X (XXX) XXX-XX-XX) FirstName (>2 any chars) LastName (>2 any chars)	Если пользователь ввел какое-либо неверное значение в поля "Ваш Email", "Пароль", "Телефон", "Имя", "Фамилия" и нажал "Зарегистрироваться", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Зарегистрироваться", то отображается всплывающий индикатор загрузки и в случае успешной регистрации осуществляется переход на экран (1.1 Profile). Если зарегистрироваться не удалось, то должен отобразиться соответствующий всплывающий диалог.	EmailField PasswordField PhoneField FirstNameField LastNameField RegisterButton	HeaderLabel DescriptionLabel EntryField DefaultButton
.3 EditProfile Обновл. ver1.5		Phone (12 digits, +X (XXX) XXX-XX-XX) FirstName (>2 any chars) LastName (>2 any chars)	Если пользователь ввел какое-либо неверное значение в поля "Телефон", "Имя", "Фамилия" и нажал "Сохранить", то рядом с соответствующим полем должна отбразиться ошибка валидации. Если ошибок нет и пользователь нажал "Сохранить", то отображается всплывающий индикатор загрузки и в случае успешного сохранения осуществляется переход на экран 1.1 Profile. Если обновить данные профиля не удалось, то должен отобразиться соответствующий всплывающий диалог.	PhoneField FirstNameField LastNameField SaveButton	HeaderLabel DescriptionLabel EntryField DefaultButton

Рис. 2.19 ❖ Примеры использования технической документации в бумажном виде

Глава 3

Архитектура приложения

Итак, мы уже познакомились с тем, как работает Xamarin.Forms и как провести техническое проектирование своими силами. Теперь у нас есть понимание о модели предметной области, и пришло время переходить к архитектуре и структуре решения – как мы будем распределять наши классы по папкам, чтобы потом было легко находить нужный код.

3.1. Многослойный MVVM

В мобильных приложениях традиционно применяется многослойная архитектура с отделением слоев доступа к данным, слоя бизнес-логики и слоя отображения пользовательского интерфейса.

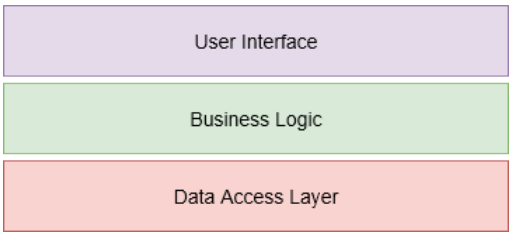


Рис. 3.1 ❖ Классическая трехуровневая архитектура

Так как родным для Xamarin.Forms является архитектурный паттерн MVVM, то именно его рекомендуется использовать в мобильных приложениях. MVVM описывает связь View (обычно это экраны приложения – Page), ViewModel и Model.

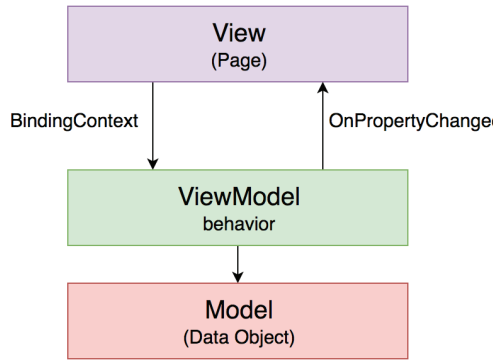


Рис. 3.2 ❖ Паттерн MVVM

Таким образом, типовая архитектура приложения на базе Xamarin.Forms будет следующей:

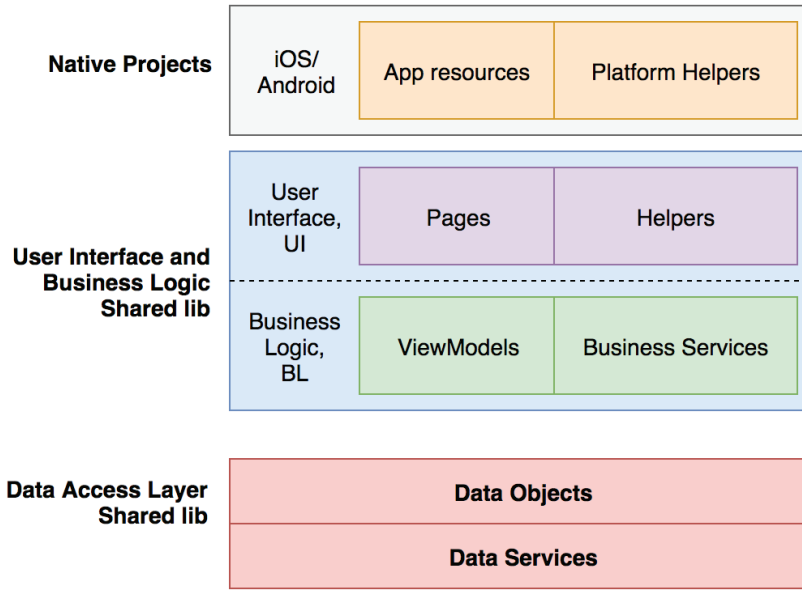


Рис. 3.3 ❖ Базовая архитектура приложения на Xamarin.Forms

В рамках данной книги мы остановимся на представленной архитектуре, так как она является классической для Xamarin.Forms. Более подробно каждый из модулей будет описан в следующих разделах.

3.2. Декомпозиция по слоям

Если вспомнить основы, то программа – это набор алгоритмов и данных. Мобильные приложения не стали исключением. Архитектура позволяет отделить алгоритмы и данные различного предназначения друг от друга.

В мобильных приложениях условно можно выделить следующие виды алгоритмов:

1) управление поведением и внешним видом компонентов пользовательского интерфейса (user interface, UI);
2) логика взаимодействия с пользователем и бизнес-сценарии (business logic, BL);
3) логика получения, хранения и преобразования данных (data access layer, DAL);
4) платформенная функциональность, не связанная с пользовательским интерфейсом (platform).

Есть также множество дополнительных алгоритмов вроде инициализации приложения или дополнительных вспомогательных классов и расширений (Extensions), но их не так просто классифицировать, так как они специфичны для проектов, команд и выбранных библиотек.

Справа показана структура пустого проекта на Xamarin.Forms. Дальше важно понимать, в какие папки складывать файлы, чтобы код сохранял простоту.

Если же переходить к тому, как поддерживать код «в тонусе» (минимальный технический долг), то важным для команды является следование единым соглашениям. Ниже мы рассмотрим пример разделения классов по папкам, который будет соответствовать описанной архитектуре.

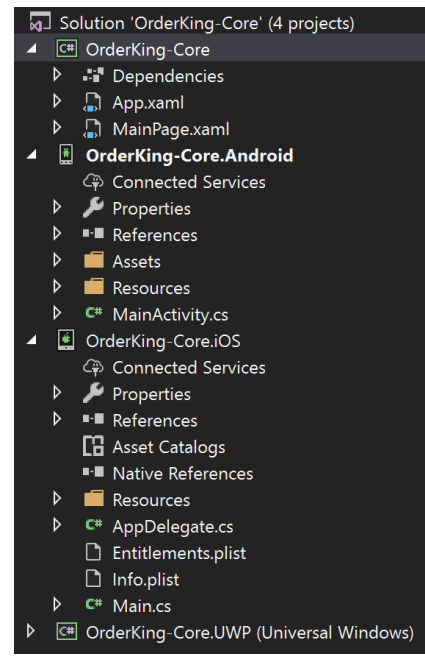


Рис. 3.4 ❖ Структура пустого проекта на Xamarin.Forms

Но для начала вспомним о данных. Здесь важно понимать, о каких из них будет идти речь. Есть данные, которые приходят с сервера (data transfer object, dto), а есть те, что обрабатываются в приложении (models, entities, data objects). Отметим, что удобнее сразу получать готовые данные со слоя DAL, чтобы дальше с ними было проще работать. Подробнее об этом мы поговорим в разделе 3.5.

Также в мобильных приложениях нет такого количества данных, чтобы требовалось делать толстые модели и «размазывать» бизнес-логику по ним (подход из больших корпоративных систем). Достаточно обычных POCO (Plain Old CLR Object) без какой-либо логики. Итак, все готовые данные приходят со слоя DAL, там же внутри спрятаны классы DTO, о которых не знают другие слои. Ниже показано различие между «толстыми» моделями и POCO-объектами.

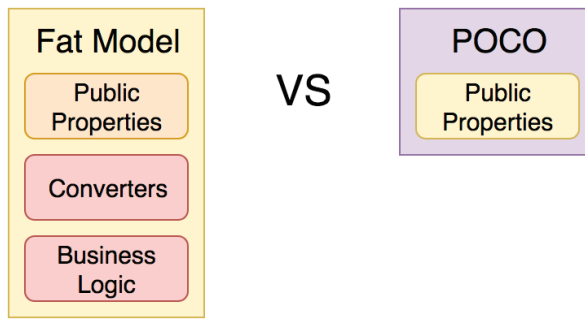


Рис. 3.5 ❖ Отличие «толстой» Model от POCO-объекта

Далее мы будем придерживаться следующих обозначений:

1) Data Objects – плоские (POCO) модели данных, с которыми будет дальше работать бизнес-логика;
2) Data Services – сервисы получения, преобразования и хранения данных;
3) Business Services – сервисы обработки данных и бизнес-сценарии;
4) Platform Services – сервисы прямого доступа к платформенной функциональности.

Финальная архитектура показана ниже, но подробнее она будет рассмотрена в разделе 3.6.

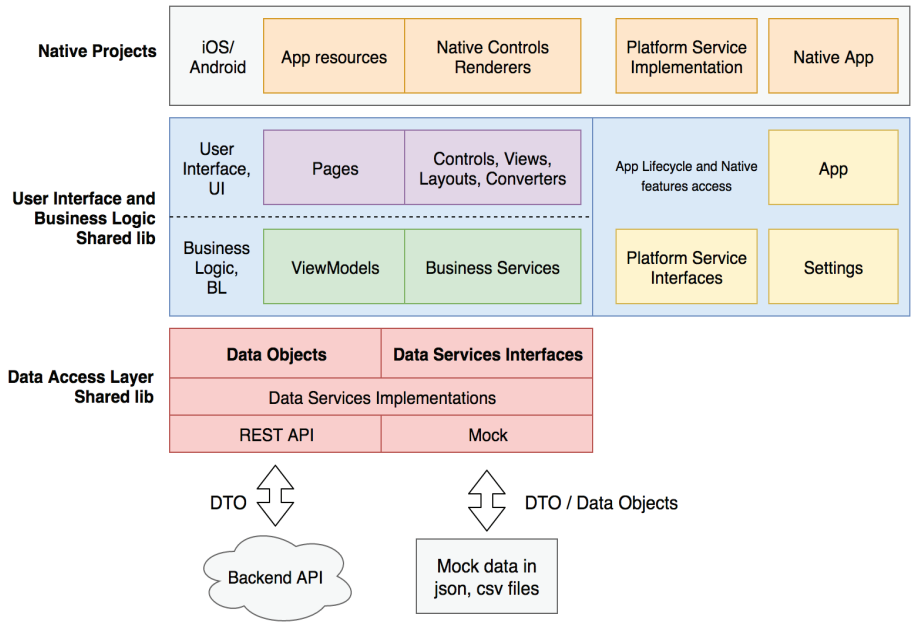


Рис. 3.6 ❖ Полная архитектура приложения на Xamarin.Forms

В заключение давайте отметим самые частые ошибки, которые допускаются при распределении алгоритмов по слоям:

1) преобразование и склейка моделей данных осуществляется внутри ViewModels – это ведет к дублированию кода и частым ошибкам преобразования, плюс засоряет код самой ViewModel;
2) управление пользовательским интерфейсом (цвет, размеры контролы) из ViewModels – это усложняет дальнейшую поддержку и развитие, так как требует держать в голове больше классов для деликатной доработки и багфикса;
3) использование IOC в DAL для доступа к внешним данным и сервисам – это ведет к тому, что самый нижний слой начинает зависеть от классов более высокого, что является грубым нарушением архитектуры. И самое главное – это усложняет и запутывает развитие системы.

3.3. Связи внутри слоев

Связи классов вообще являются темой очень непростой. Главный принцип гласит: чем меньше у класса зависимостей от других классов, тем лучше. Лишние связи усложняют код и делают его развитие очень тяжелым. Важно понимать, какие связи ведут к образованию «лапшекода», а какие – нет.

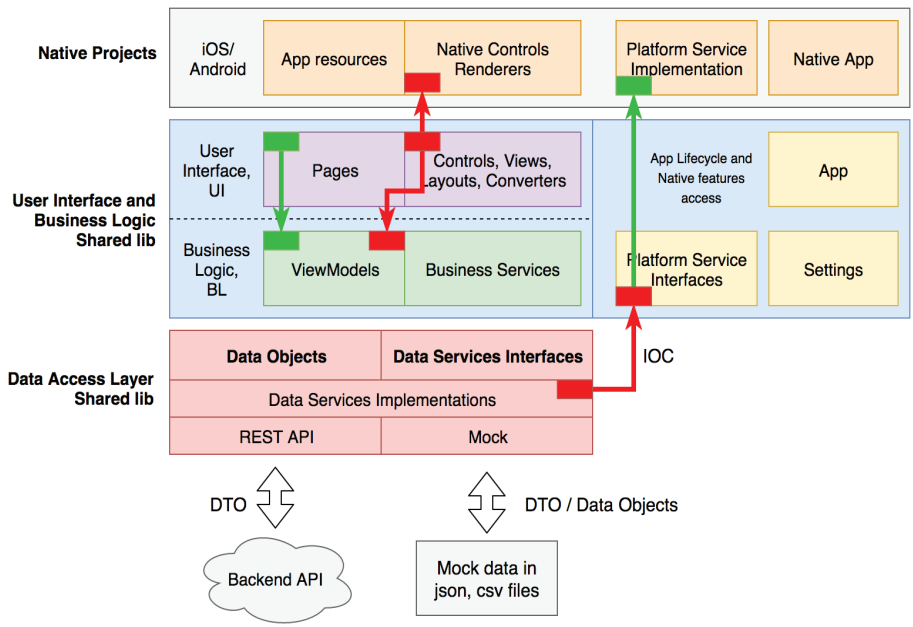


Рис. 3.7 ❖ Правильные и неправильные зависимости

Чем длиннее отдельная «лапша», тем сложнее потом поддерживать продукт. Также «лапшекод» часто выступает источником утечек памяти, когда объекты не могут быть удалены сборщиком мусора из-за взаимных ссылок.

Выделяют следующие связи между классами и их экземплярами:

1) обобщение/наследование (класс X наследуется от класса Y);
2) ассоциации (экземпляр класса X включает экземпляр(ы) класса Y в качестве одного из своих полей, например `Store.Products`, где `Products` – это `List<ProductObject>`);

3) реализация (класс X реализует интерфейс Y);
4) зависимость (экземпляр класса X обращается к полю или методу у экземпляра класса Y).

С точки зрения кода чаще всего ошибки допускаются в работе с зависимостями – именно они являются основным источником «лапшекода». Связи видов 1, 2, 3 мы оставим за пределами данной книги – так как они обычно не ведут к большому количеству проблем и уже самим фактом своего существования структурируют классы в единую иерархию.

Итак, «зависимости» (4-й вид связей) могут быть представлены в следующем виде:

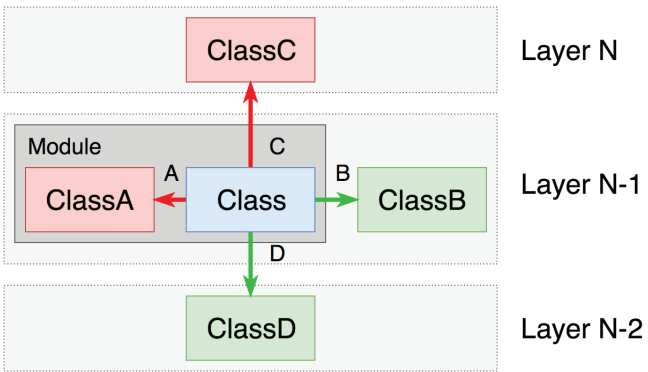


Рис. 3.8 ❖ Зависимости между классами с учетом архитектуры

О вертикальных связях C и D мы поговорим в разделе 3.5, а сейчас сфокусируемся на A и B.

Связь вида B (с внешним модулем) является нормальной и обычно не ведет к патологиям. Объект класса из одного модуля просто обращается к методам или свойствам другого объекта. Единственное, здесь нужно быть внимательным, чтобы не было циклических или двунаправленных связей, – это ведет к образованию «лапши».

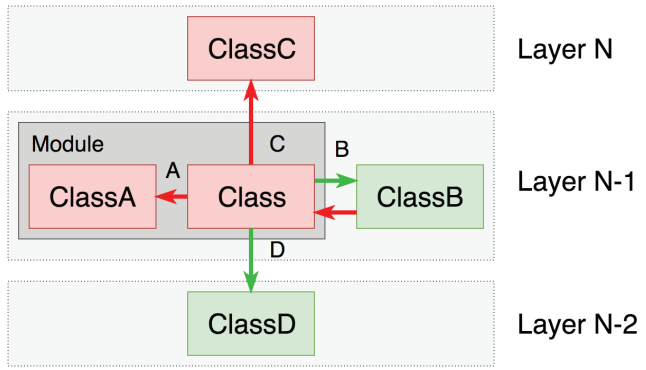


Рис. 3.9 ❖ Пример циклических зависимостей между Class и ClassB

А вот связи вида A являются проблемными, и их необходимо избегать. Также на подобные связи необходимо обращать внимание во время code review. Давайте посмотрим, как лучше распутать связи вида A в различных модулях (рис. 3.10).

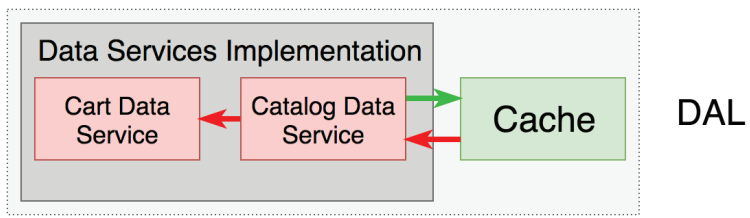


Рис. 3.10 ❖ Неправильные зависимости на уровне DAL

Рекомендации по DAL: избегать зависимостей между сервисами (репозиториями) и «размазываний» логики по ним. Каждый Data Service должен быть изолирован от соседей.

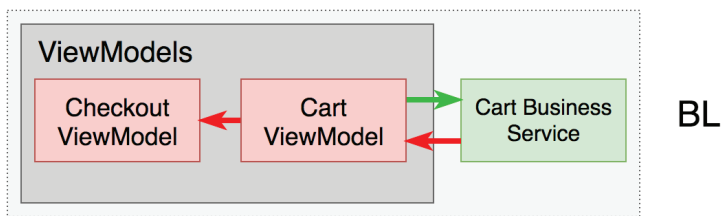


Рис. 3.11 ❖ Неправильные зависимости на уровне BL

Рекомендации по BL: избегать зависимостей Business Services -> ViewModel; использовать общую шину MessageBus (см. раздел 4.2) для передачи данных между ViewModels.

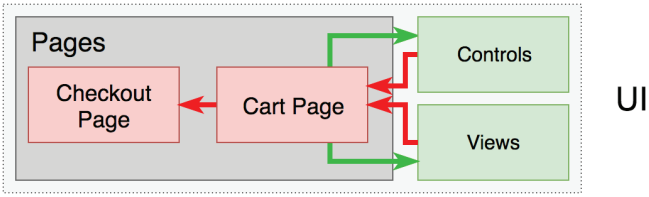


Рис. 3.12 ❖ Неправильные зависимости на уровне UI

Рекомендации по UI: избегать зависимостей между Pages, Controls и Views. Допускаются только ассоциации, когда, например, одни элементы пользовательского интерфейса (Views) являются частью более сложного View.

Придерживаясь описанных рекомендаций и уделяя особое внимание связям вида A, вы сможете оперативно выявлять «лапшекод», предотвращая его развитие в проекте. Но всегда стоит учитывать, что в реальных проектах иногда возникают ситуации, когда приходится нарушать принятые паттерны, чтобы «срезать путь». Но это должно быть редким и локализованным исключением.

3.4. Связи между слоями

Теперь мы можем подробнее рассмотреть вертикальные связи C и D.

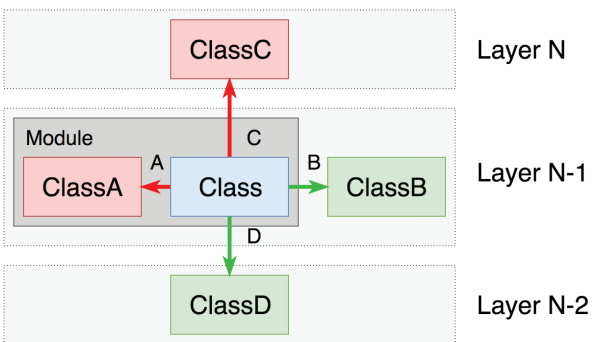


Рис. 3.13 ❖ Зависимости вида C и D

Многослойная архитектура предполагает, что классы слоя N будут иметь доступ только к слою N-1 и в редких случаях N-2 и ниже. Если следовать этой логике, связей вида C быть не должно ни при каких обстоятельствах.

По правилам архитектуры не должно быть зависимостей класса из слоя N-1 от класса из слоя N. Это означает, что ViewModel не меняет напрямую свойства у интерфейсных элементов (например, цвет кнопки) и Repository не должен обращаться напрямую к настройкам приложения или вызывать PlatformService для получения координат или информации об устройстве.

В тех случаях, когда связь вида C все же необходима (доступ к платформенной функциональности из ViewModel), следует использовать Inverse Of Control (IOC), когда на слое N-1 создается интерфейс (Interface), который реализован на слое N и доступен через IOC. В большинстве проектов нет необходимости использовать IOC для связей между слоями и тем более классами, однако исключение составляет платформенный функционал. Для доступа к нему подойдет DependencyService, являющийся частью Xamarin.Forms. IOC, с одной стороны, упрощает код, но с другой – снижает его читаемость и наглядность, поэтому его стоит использовать «штучно», когда без него просто не обойтись.

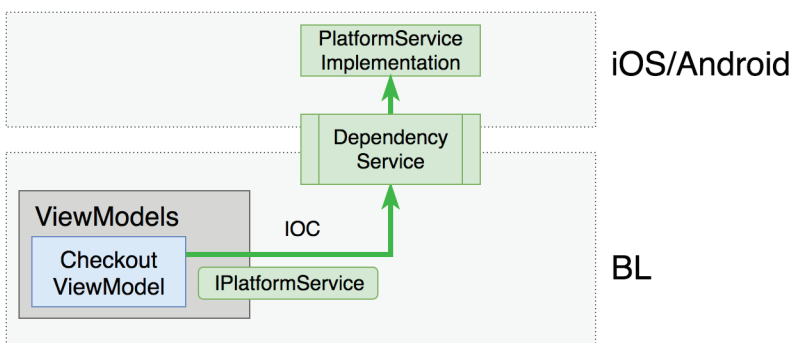


Рис. 3.14 ❖ Использование IOC для доступа к верхнему слою

Итак, связей вида C быть не должно, это легко выявить во время Code Review.

Остаются связи вида D. Это нормальные здоровые зависимости. Давайте подробнее их рассмотрим.

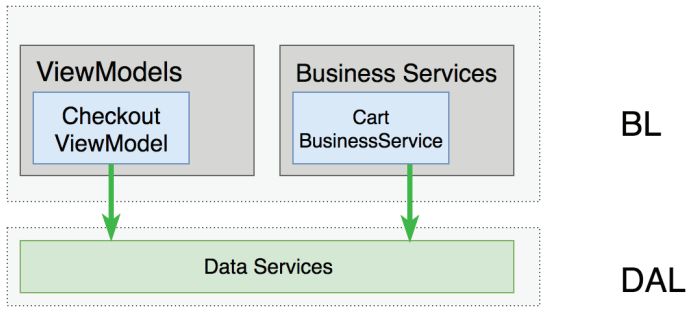


Рис. 3.15 ❖ Здоровые зависимости вида D (BL -> DAL)

Слой DAL предоставляет наружу интерфейсы для доступа к репозиториям и структуры (модели) данных. Поэтому BL (ViewModels и Business Services) могут только вызывать нужные методы и получать нужные данные. Это предотвращает появление лишних зависимостей. DAL должен быть вещью в себе и ничего не знать о вышележащих слоях. Крайне не рекомендуется использовать IOC или другие механизмы на слое DAL. Все необходимые данные должны передаваться в DAL напрямую через вызов соответствующих методов на вышележащих слоях.

Следующая на очереди у нас будет связка UI-BL.

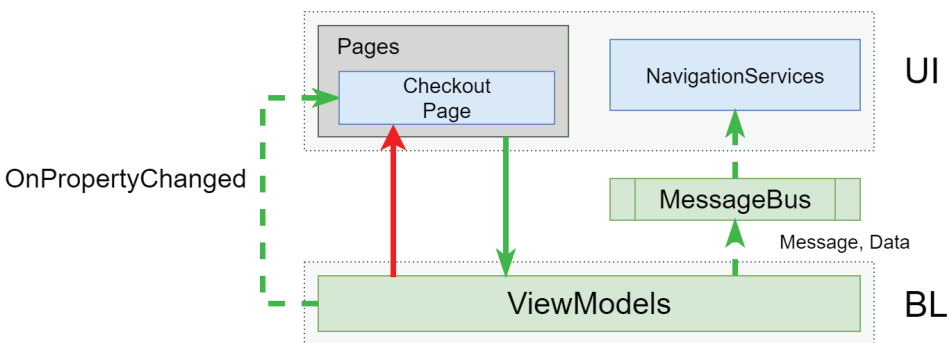


Рис. 3.16 ❖ Зависимости видов C и D (BL -> UI)

Как уже отмечалось ранее, BL не может обращаться напрямую к UI. Для связи UI и BL необходимо использовать механизмы Binding, когда

данные передаются не напрямую, а через события: у ViewModel изменилось свойство, это сгенерировало событие OnPropertyChanged, после которого нужные данные присваиваются UI-элементу автоматически (через механизмы BindableObject, являющегося общим предком всех View в Xamarin.Forms). От UI во ViewModel данные также попадают через Binding. В редких случаях допустимо прямое обращение к ViewModel или Business Service из кода UI. Но это должно быть только от острой необходимости.

Если ViewModel требуется совершить прямое управление UI (например, открыть новую страницу приложения), то следует использовать общую шину (MessageBus), в которую ViewModel будет посылать сообщение, считываемое внешним UI-сервисом (NavigationService, DialogService).

Доступ к нативной функциональности возможен и допустим как со слоя UI, так и со слоя BL. А на DAL зависимостей от «нативки» быть не должно – это усложняет развитие продукта и поиск проблем.

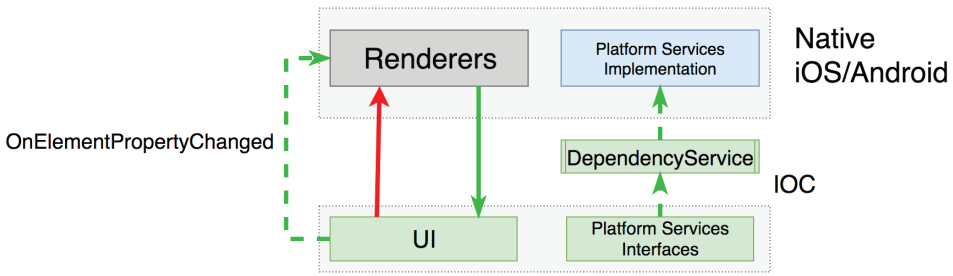


Рис. 3.17 ❖ Зависимости между UI и платформенной частью

Для того чтобы ViewModels и Business Services имели доступ к платформенной функциональности, следует использовать IOC, как отмечалось ранее.

Слой UI прямых зависимостей от платформы не имеет, а только предоставляет необходимые свойства, события и методы, которые будут отслеживаться, изменяться или вызываться из Native-части через Renderers. Таким образом, UI у нас не должен иметь прямых зависимостей от BL (кроме редких исключений) или Native. Если такие зависимости есть – это признаки проблемной реализации.

3.5. Структуры данных на основе UI

О данных есть много разных мнений, и все будет зависеть от того, как хорошо вы освоили те или иные подходы проектирования. Когда речь заходит о данных, то бизнес-аналитики и проектировщики (формализуют модель предметной области) обычно работают с некой умозрительной, пусть и достоверной бизнес-моделью. Это позволяет грамотно формализовать бизнес-требования, но оставляет много вопросов у разработчиков, до которых очередь доходит уже после анализа и проектирования. В результате получается ситуация «модели данных отдельно, дизайн отдельно, код отдельно».

Можно выделить четыре вида данных, которые используются в мобильных приложениях:

- Models – модели данных, привязанные к пользовательскому интерфейсу, содержат бизнес-логику;
- Data Objects – модели данных, привязанные к пользовательскому интерфейсу, но без бизнес-логики;
- Entities – объекты «предметной области», могут содержать бизнес-логику и поля модели предметной области;
- Data Transfer Object (DTOs) – структуры, используемые для обмена с внешними веб-сервисами (REST/SOAP API).

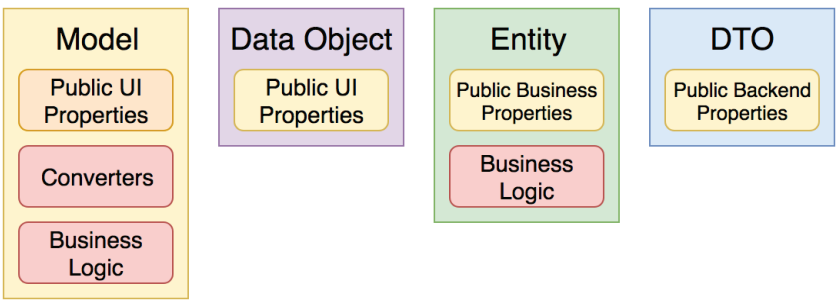


Рис. 3.18 ❖ Различные виды данных в приложениях

В некоторых проектах может быть еще больше видов данных, но и три уже становится сложно поддерживать по мере развития продукта. Достаточно двух, надо только понять, каких.

Если рассматривать не академический, а практический подход, то реальным проектированием модели предметной области занимаются проектировщики пользовательского интерфейса. Именно они разбивают приложение на экраны и продумывают, какие данные отображать пользователю, как их группировать, как будет меняться поведение и внешний вид экрана в зависимости от данных. А ведь какой бы ни была система, ей будет необходим интерфейс взаимодействия с человеком, – всегда лучше начинать с него.

Напомним, что мобильные приложения являются пользовательским интерфейсом для внешнего бизнес-процесса, а следовательно, модель предметной области, доступная пользователю, также описывается интерфейсом.

Нет необходимости придумывать какие-то умозрительные модели предметной области, если есть пользовательский интерфейс. Именно его предстоит создать разработчику. В паттерне MVVM за данные отвечают Models, однако термин «model» ассоциируется со сложной структурой данных с бизнес-логикой в придачу. Чтобы избежать подобных ассоциаций, мы будем придерживаться термина Data Object, который будет является POCO-объектом, как отмечалось в разделе 3.2.

Готовые Data Objects должны сразу приходить со слоя DAL, чтобы потом не требовалось возиться с их преобразованием и форматированием. Data Objects должны содержать только те данные, которые необходимы пользовательскому интерфейсу (плюс служебные поля вроде id объекта в базе данных), избегая мусора и неиспользуемых полей.

Таким образом, у нас в проекте остаются только DTO и Data Objects. Структура DTO определяется серверным разработчиком, и в главе 5 мы рассмотрим то, как с ними удобнее работать. А сейчас давайте сфокусируемся на Data Objects.

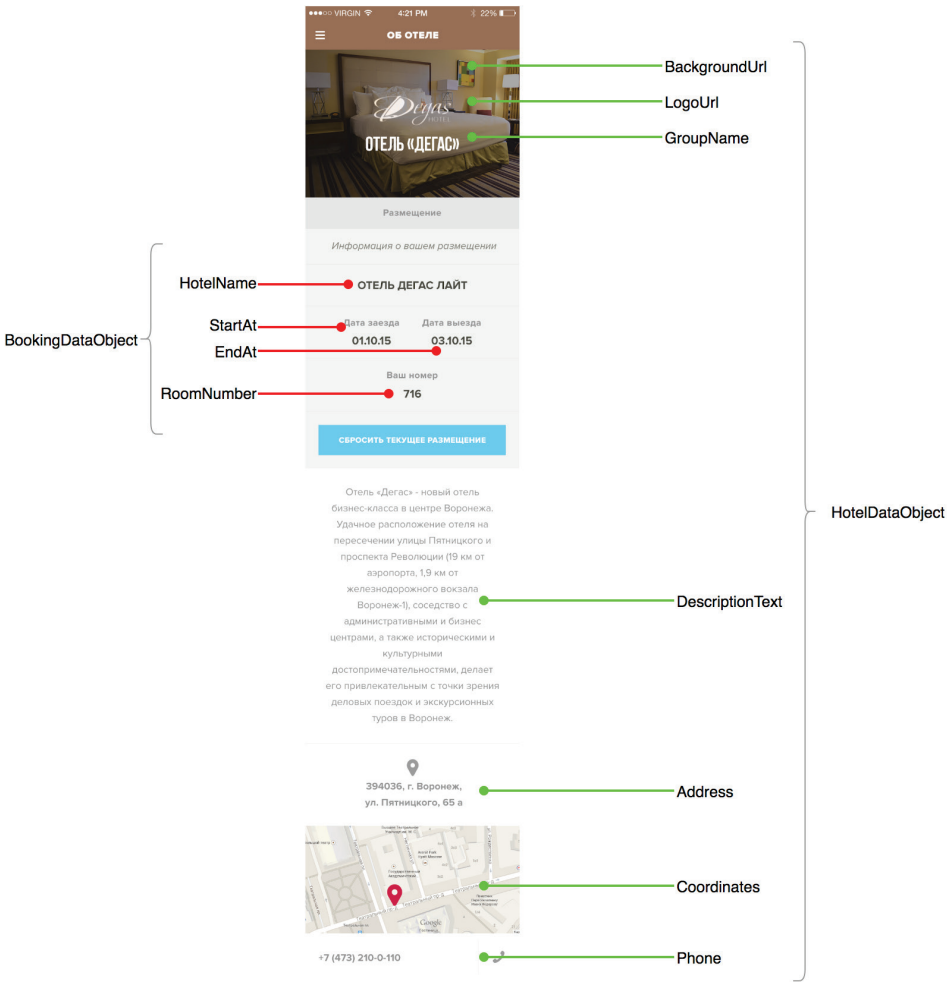


Рис. 3.19 ❖ Выделение Data Objects на основе интерфейса

Обычно при разговоре про MVVM у программиста в голове возникает образ «одна Model-одна View-одна ViewModel». Это часто является стопором, так как возникает соблазн для каждого экрана сделать свою единую Model. Но зачем, если мы можем использовать несколько разных структур данных на одном экране? В этом нам помогут Data Objects.

На схемах выше показано, как создавать Models на базе экранов. Первым шагом необходимо выделить на экране те данные, которые

будут изменяться. Затем следует сгруппировать эти данные в логические структуры и придумать названия для каждой, выделяя массивы и списки (например, `ProductObject`, `List<ReviewObject>`, `ProductDescriptionObject`). После этого следует просмотреть остальные экраны приложения и проверить, нет ли на них тех же самых данных. Если есть – проверить полноту наших структур.

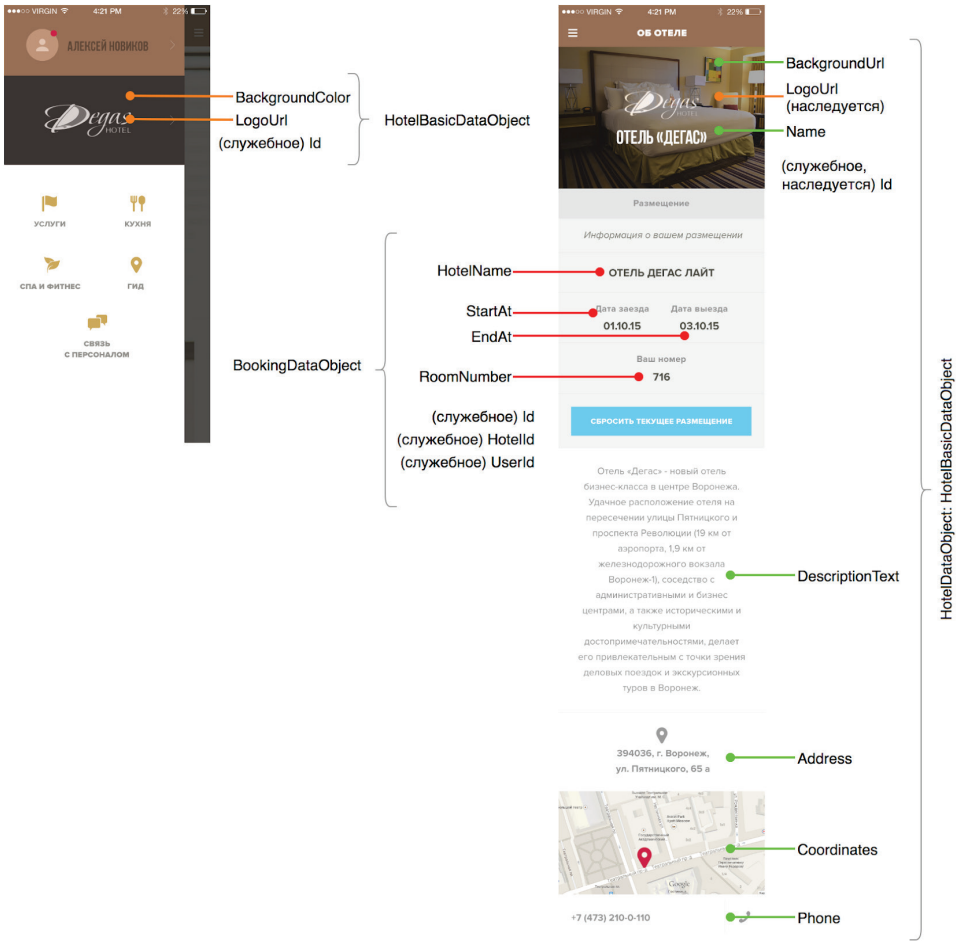


Рис. 3.20 ❖ Доработка Data Objects на основе интерфейса

Какой-либо единый и простой алгоритм для этапа проектирования данных выделить сложно – большинство программистов уже интуитивно использует тот или иной подход к выделению ключевых

структур данных. Здесь важно понимать, что одни и те же структуры должны и будут использоваться на разных экранах, поэтому должны содержать все необходимые поля. При необходимости можно выделять полную и сокращенную версии одних и тех же «сущностей» – например, полное описание о продукте и его сокращенная версия в списке товаров.

Описанный выше подход позволяет выделить те данные, которые только отображаются пользователю. Если же требуется что-либо ввести (например, email и пароль для авторизации), то такие поля лучше делать в виде отдельных Properties у ViewModel, чтобы было проще их потом проверять, а в финальный Data Object уже записывать корректные значения.

3.6. Типовая архитектура приложения на Xamarin.Forms

В данном разделе мы соберем в одном месте описание типовой архитектуры бизнес-приложений на Xamarin.Forms.

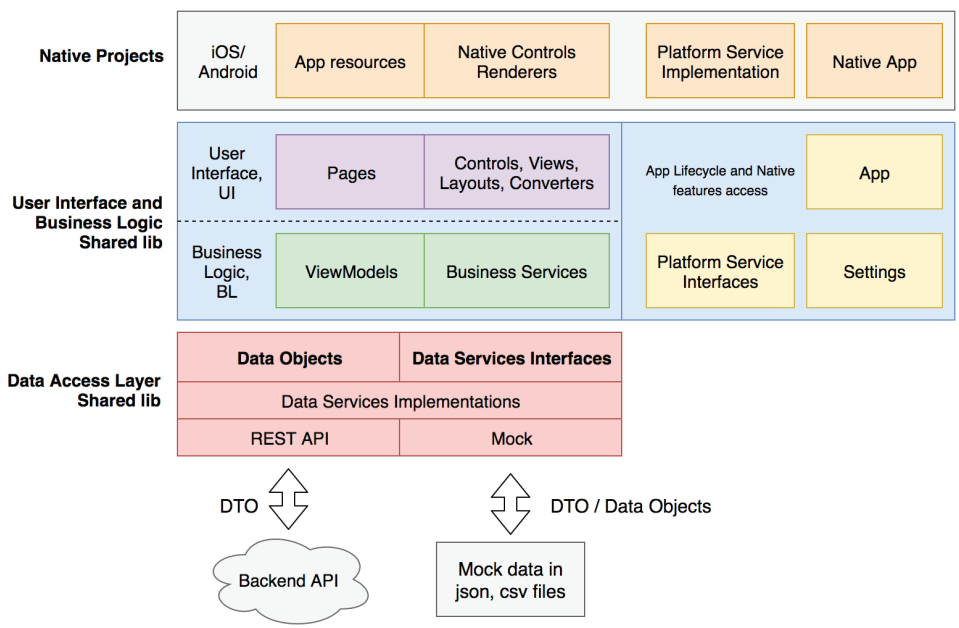


Рис. 3.21 ❖ Типовая архитектура приложения на Xamarin.Forms

3.6.1. Слой работы с данными (Data Access Layer, DAL)

Данный слой реализует полностью всю логику получения и хранения данных, включая механизмы синхронизации и работы с локальной СУБД (SQLite и пр.). DAL выделен в отдельную библиотеку, и модули более высокого уровня должны получать из него готовые модели данных (Data Objects), содержащие только те поля, которые необходимы для реализации бизнес-логики и пользовательского интерфейса. В описанной представленной структуре DAL можно выделить следующие компоненты:

- DTO (Data Transfer Objects) – структуры данных, возвращаемые и принимаемые серверными API;
- Refit REST API interface – спецификации серверного API для библиотеки Refit;
- Data Service Implementation – реализация сервисов получения, синхронизации и хранения данных. Для каждого раздела приложения создается свой репозиторий;
- Data Objects – модели предметной области, адаптированные для пользовательского интерфейса;
- Data Service Interfaces – программные интерфейсы, описывающие отдельные сервисы доступа к данным.

Обязательно покрытие Unit-тестами для REST API, чтобы своевременно выявлять ошибки в работе серверных методов. Также рекомендуется покрытие реализации Data Services с помощью Unit-тестов.

3.6.2. Слой бизнес-логики

Данный слой содержит реализацию сценариев взаимодействия с пользователем и содержит в себе два вида классов:

- ViewModels – реализация поведения отдельных экранов по принципу «один page – одна viewmodel»;
- Business Services – реализация сервисов, обслуживающих бизнес-логику.

Слой Business Logic (BL) тестируется вместе с пользовательским интерфейсом с помощью UI-тестов, имитирующих работу человека на реальных устройствах. Это является дополнением к ручному тестированию.

3.6.3. Слой пользовательского интерфейса

Данный слой полностью содержит описание пользовательского интерфейса, его поведения и вспомогательные классы:

- Pages – описание отдельных страниц (экранов) приложения;
- Controls – различные компоненты пользовательского интерфейса, требующие нативной реализации через рендереры;
- Views – комплексные компоненты пользовательского интерфейса, состоящие из других UI-элементов;
- Converters – различные конвертеры данных внутри механизма Binding в формат, необходимый для отображения пользователю;
- Behaviors – описывает сценарии поведения элементов пользовательского интерфейса;
- DataTemplateSelectors – поставщики ячеек для списков. В зависимости от типа данных возвращают соответствующую ячейку;
- Effects – визуальные эффекты, применяемые к различным элементам пользовательского интерфейса;
- Extensions – наборы расширенных методов и свойств для классов с элементами пользовательского интерфейса;
- Layouts – специализированные компоновщики пользовательского интерфейса для тех ситуаций, когда использование стандартных компоновщиков не оправдано с точки зрения производительности или сложности;
- ViewCell – ячейки для списков данных.

Для тестирования пользовательского интерфейса вместе с бизнес-логикой необходимо использовать автоматизированные UI-тесты, имитирующие поведение реальных пользователей на реальных устройствах. Это является дополнением к ручному тестированию.

3.6.4. Дополнительные классы

Также частью библиотеки с общей частью приложения являются следующие элементы:

- App.cs – управление жизненным циклом приложения;
- Settings.cs – единый реестр настроек приложения, реализованный поверх механизмов iOS/Android;

- Platform Service Interfaces – программные интерфейсы для доступа к нативной функциональности, реализуемой с помощью DependencyService.

Для тестирования данных механизмов достаточно использовать автоматизированные UI-тесты и ручное тестирование.

3.6.5. Нативная часть

Содержит реализацию платформенной функциональности и контролов, а также инициализирует и запускает приложение:

- Native App (по умолчанию AppDelegate.cs для iOS и MainActivity.cs для Android) – инициализация и настройка приложения, специфичное для каждой платформы;
- Platform Services (Implementation) – реализация платформенных сервисов, доступных через DependencyService;
- Renderers – платформенная реализация элементов пользовательского интерфейса;
- Resources – ресурсы приложения, картинки, шрифты.

Для тестирования данных механизмов достаточно использовать автоматизированные UI-тесты и ручное тестирование.

Глава 4

Базовая инфраструктура и ее применение

Каждое приложение начинается с создания пустого проекта и его дальнейшего расширения. Применение подхода, описанного в главе 2, и архитектуры из главы 3 позволит создать минимально необходимый набор классов и папок в вашем проекте. Однако для полноценной реализации «скелета» проекта, необходимо использовать базовые классы (Base*) для всех ключевых компонентов и минимальный набор инфраструктурных классов. В данной главе мы опишем общие рекомендации для создания подобного фундамента. Вы можете использовать описанные ниже классы или реализации, доступные в виде Nuget-библиотек (например, на базе MVVM Light или аналогичных библиотек).

4.1. Фундамент Data Access Layer (DAL)

Как уже многократно отмечалось ранее, слой DAL должен возвращать готовые данные для ViewModel и выступать в роли черного ящика. Это позволит на уровне BL ничего не знать о структуре и внутреннем устройстве DAL, всегда получая только нужные данные. Реализацию DAL лучше упаковать в отдельную библиотеку и покрыть unit-тестами.

4.1.1. Класс `DataServices` как единая точка входа в слой DAL

Для того чтобы удобнее взаимодействовать с DAL, можно использовать единую точку входа – статический класс `DataServices`, который будет предоставлять необходимые методы и свойства. На всякий слу-

чай напомним, что каждый раздел (набор экранов), описанный в документации, должен иметь свой сервис данных (см. главу 2).

Листинг 4.1 ❖ Пример класса `DataServices`

```
public static class DataService {
    static bool _isInitialized;

    public static IAuthorizationDataService Authorization { get; private set; }

    public static void Init(
        string dbFilePath,
        string baseUrl,
        bool isMock = false) {

        if (_isInitialized)
            return;

        _isInitialized = true;

        if (isMock) {
            Authorization = new AuthorizationMockDataService();
        }
        else {
            // см. Главу 12
            // Здесь добавить реализацию для онлайн-сервиса,
            // обращающегося к внешнему API с baseUrl
            // и сохраняющего данные в локальную базу данных
            // в файле dbFilePath
        }
    }
}
```

Как видим, `DataServices` предоставляет механизмы инициализации сервисов, управления авторизацией на уровне HTTP (Token, Cookies и прочее) и свойства для доступа к отдельным сервисам. Для обращения к нужному методу в DAL будет достаточно, например, `DataServices.Authorization.Login();`.

Наборы методов мы также можем получить из таблицы экранов или определить на этапе реализации. Имена методов лучше делать понятными, а каждый метод должен будет возвращать данные (Models, Entities, Data Objects), которые сразу будут адаптированы для пользовательского интерфейса.

4.1.2. Data Objects и Data Services

Если рассматривать структуры данных, то обычно мобильные разработчики работают с уже понятными серверными API и описанными форматами данных. Данные, которые приходят от сервера или передаются на него, обычно называют Data Transfer Object (DTO).

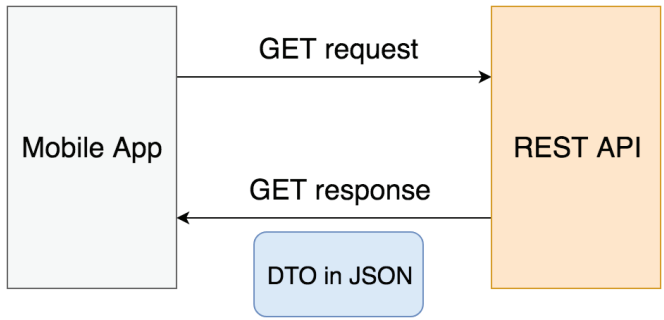


Рис. 4.1 ❖ Взаимодействие мобильного приложения с REST API

В идеальном мире DTO приходят в таком формате, как необходимо пользовательскому интерфейсу, чтобы не приходилось возиться с преобразованием данных. Однако в реальности такое бывает крайне редко, да и интерфейс со временем меняется. Поэтому в проектах часто используются дополнительные структуры данных, адаптированные для UI.

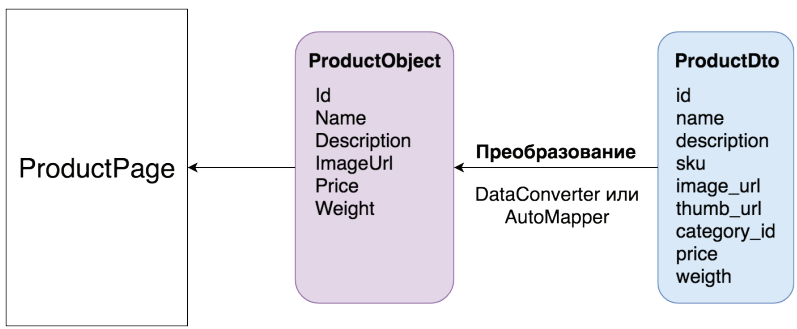


Рис. 4.2 ❖ Получение данных для Page со стороны серверного API

Как уже отмечалось ранее, в мобильных бизнес-приложениях относительно невысокая (а часто и очень низкая) сложность бизнес-логики, поэтому объекты с данными для UI лучше реализовывать в виде POCO (см. раздел 3.5) – без какой-либо логики. Это позволяет сфокусировать бизнес-логику только на уровне ViewModel.

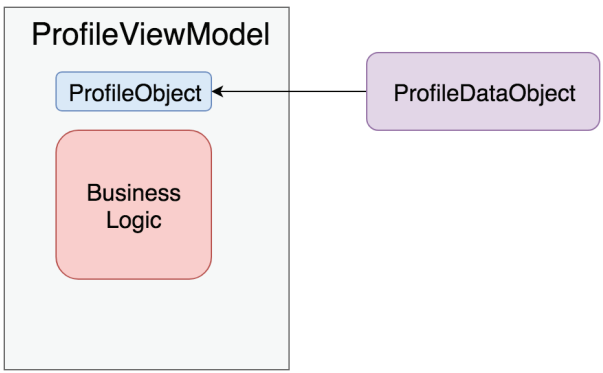


Рис. 4.3 ❖ Объект класса ProfileDataObject внутри ViewModel

В данной книге мы будем придерживаться названия Data Objects для объектов, приходящих из DAL. Процесс их получения будет описан далее.

Одним из элементов наших Data Services станет механизм получения Data Object из DTO.

В зависимости от количества ваших Data Object можно использовать три подхода для преобразования данных (чтобы в дальнейшем их было проще поддерживать и развивать):

- единый статический класс DataConverter – до 30 сущностей во всем приложении;
- свой статический класс DataConverter для каждого раздела – до 30 сущностей в каждом разделе (или более 100 сущностей в проекте);
- спрятать логику преобразования в сам Data Object (например, реализовав специальный конструктор, который на вход будет принимать объект DTO, или переопределив оператор присваивания) – если более 30 сущностей.

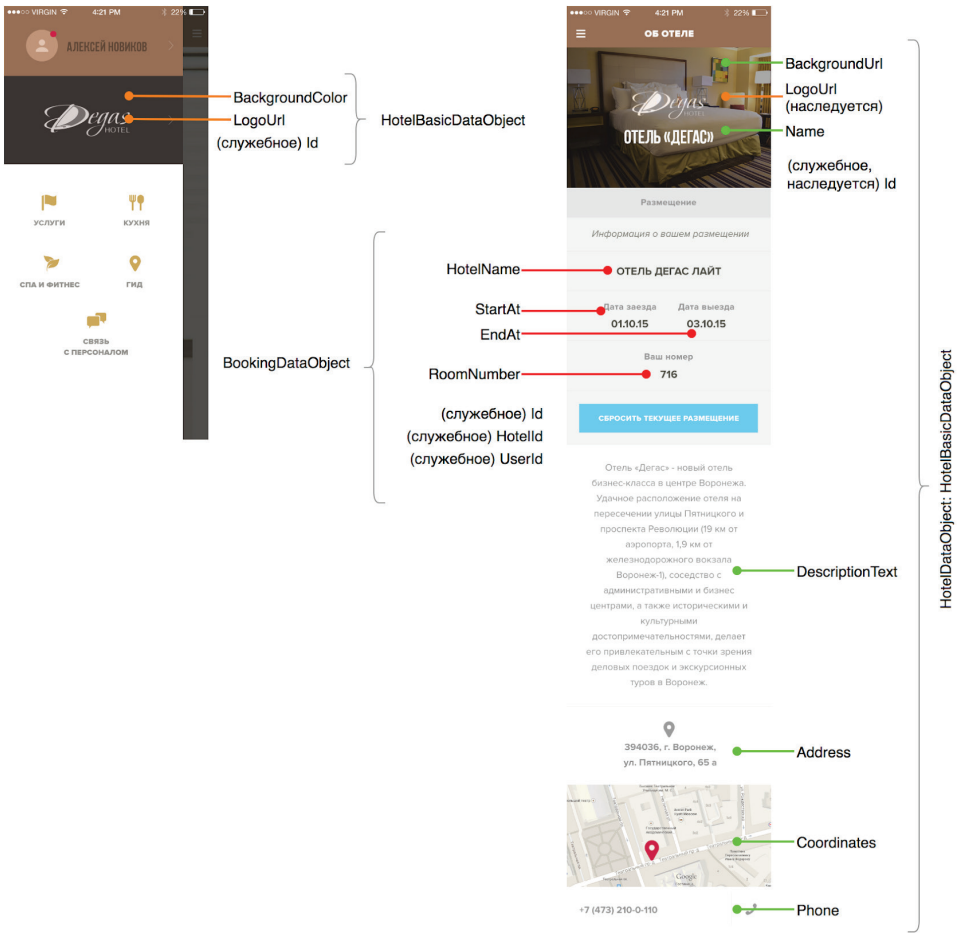


Рис. 4.4 ❖ Получение структур Data Object на основе пользовательского интерфейса

Важно заметить, что бизнес-логика должна не просто получать готовые данные из DAL, но и узнавать о том, как прошел процесс получения этих данных и были ли ошибки. Для этого необходимо (никаких исключений!) возвращать из DAL структуры `RequestResult`, которые помимо самих данных будут содержать статус ошибки и сообщение от сервера (если была ошибка) или базы данных. Использование `struct` вместо `class` позволяет убрать дополнительную проверку на `null` (структуры передаются по значению, а не по ссылкам, как

объекты классов), так как именно такие ошибки являются основной проблемой DAL.

Самой частой причиной крешей в приложениях является остутствие проверок на `null`. Обязательно делайте проверки на `null` в ваших методах DAL. Это сэкономит нервы вам и вашим пользователям.

Листинг 4.2 ❖ Реализация структуры `RequestResult`

```
public struct RequestResult<T> {
    public readonly string Message;
    public readonly RequestStatus Status;
    public readonly T Data;
    public bool IsValid => Status == RequestStatus.Ok && Data != null;

    public RequestResult(T data, RequestStatus status, string message = null) {
        Data = data;
        Status = status;
        Message = message;
    }

    public override string ToString() {
        return $@"Result: {Status}, Data: {Data}, Message: {Message}";
    }
}

public struct RequestResult {
    public readonly string Message;
    public readonly RequestStatus Status;
    public bool IsValid => Status == RequestStatus.Ok;

    public RequestResult(RequestStatus status, string message = null) {
        Status = status;
        Message = message;
    }

    public override string ToString() {
        return $@"Result: {Status},  Message: {Message}";
    }
}

public enum RequestStatus {
    Unknown = 0,
```

```
    Ok = 200,
    NotModified = 304,
    BadRequest = 400,
    Unauthorized = 401,
    Forbidden = 403,
    NotFound = 404,
    InternalServerError = 500,
    ServiceUnavailable = 503,
    Canceled = 1001,
    InvalidRequest = 1002,
    SerializationError = 1003,
    DataBaseError = 505
}
```

Итак, каждый (или почти каждый) метод DAL получает от сервера DTO-объекты и преобразует их в нужные Data Object, используя единые механизмы и возвращая `RequestResult`, содержащий готовый объект в качестве поля Data. Преобразование мы разобрали, можем перейти к получению. Для этого необходимо реализовать базовые классы, которые и будут получать/отправлять данные на сервер и работать с базой данных.

В нашем примере мы остановимся на чтении тестовых данных (Mock), которые будут добавлены в проект DAL в виде JSON-файлов в папку Resources. Mock-данные позволят вести разработку без сервера, подставляя любые данные и статусы в `RequestResult` для разработки и тестирования.

Листинг 4.3 ❖ Реализация класса `BaseMockDataService`

```
public class BaseMockDataService {
    protected async Task<RequestResult<T>> GetMockData<T>(string fileId) where
T : class {

        try {
            var data = JsonConvert.DeserializeObject<T>(GetFileContent(fileId));
            return new RequestResult<T>(data, RequestStatus.Ok);
        }
        catch (Exception e) {
            return new RequestResult<T>(default(T), RequestStatus.
InternalServerError, e.Message);
        }
    }
```

```
    static string GetFileContent(string fileId) {
        var assembly = typeof(BaseMockDataService).GetTypeInfo().Assembly;
        var stream = assembly.GetManifestResourceStream(fileId);

        using (var reader = new StreamReader(stream))
            return reader.ReadToEnd();
    }
}
```

DAL должен восприниматься только как черный ящик для записи и получения готовых данных, поэтому какую-либо бизнес-логику необходимо реализовать на уровне Business Services.

4.2. Фундамент Business Layer (BL)

Одно из самых больших заблуждений начинающих мобильных разработчиков заключается в том, что в приложениях есть «сложная бизнес-логика». Что это такое, мало кто может ответить, и все заканчивается одним-двумя примерами сервисов бизнес-логики вроде синхронизации данных. На самом деле начинающие разработчики часто понимают под бизнес-логикой симбиоз из алгоритмов преобразования данных (из DTO в Fat Model или еще какой-нибудь формат), алгоритмов получения данных из DAL (вместо DAL реализуя обращение к серверу напрямую внутри ViewModel) и алгоритмов управления UI (которые надо оставлять внутри классов UI). Как мы уже отмечали ранее, алгоритмы преобразования данных лучше оставить в слое DAL. Поэтому наша бизнес-логика будет получать данные в готовом виде.

4.2.1. Реализация фоновых задач и сервисов бизнес-логики

Сервисы бизнес-логики (Business Services) необходимо использовать в тех случаях, когда требуется вынести те или иные алгоритмы (например, обработку корзины или временное хранение данных) из ViewModel. Также фоновые сервисы бизнес-логики могут выполнять работу по таймеру, например проверять обновление данных на сервере.

Для реализации Business Service можно использовать один из следующих паттернов:

- **Singleton** – создается единственный экземпляр класса (например, `CartService.Instance`), и все обращения из внешнего мира идут к этому объекту. Удобно использовать для фоновой обработки очереди событий или синхронизации данных. Простой и безопасный способ нереализации синглетонов – использование класса `Lazy`.

Листинг 4.4 ❖ Пример создания синглетона

```
public sealed class CartService
{
    static readonly Lazy<CartService> LazyInstance =
           new Lazy<CartService>(() => new CartService(), true);
    public static CartService Instance => LazyInstance.Value;

    // Реализация методов и свойств класса CartService
}
```

- **Статический класс (static class)** – набор статических методов и свойств. Например, для реализации кеша в ОЗУ приложение один раз получает данные и хранит их в памяти до закрытия приложения.

При реализации классов бизнес-логики стоит учитывать, что методы будут вызывать в многопоточном окружении, поэтому следует использовать структуры данных из набора Concurrent (`ConcurrentDictionary`, `ConcurrentList`, `ConcurrentQueue`, `ConcurrentStack`), упаковывать методы в Task и использовать механизмы синхронизации потоков (например, lock).

На этапе создания «скелета» проекта будет достаточно описать пустые классы бизнес-сервисов, реализацию можно будет добавить позже.

4.2.2. Фундамент для ViewModels

Главная задача ViewModel – реализация логики взаимодействия с пользователем. Для этого изменяемые поля данных и команды из ViewModel подключаются к Page через механизм Binding. View автоматически подписывается на события `OnPropertyChanged`, исходящие от ViewModel, и обновляет компоненты пользовательского интерфейса

новыми данными в ответ на это событие. Из коробки Xamarin.Forms не содержит базовых классов, реализующих INotifyPropertyChange, поэтому для реализации ViewModel нам дополнительно потребуется своя реализация этого интерфейса.

Листинг 4.5 ❖ Реализация Bindable

```
public class Bindable: INotifyPropertyChanged {
    readonly ConcurrentDictionary<string, object> _properties = new
ConcurrentDictionary<string, object>();

    public event PropertyChangedEventHandler PropertyChanged;

    [NotifyPropertyChangedInvocator]
    protected void OnPropertyChanged([CallerMemberName] string propertyName =
null) {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(
propertyName));
    }

    protected T Get<T>(T defValue = default(T), [CallerMemberName] string name
= null) {
        return !string.IsNullOrEmpty(name) && _properties.TryGetValue(name, out
var value)
                   ? (T)value
                   : defValue;
    }

    protected bool Set(object value, [CallerMemberName] string name = null) {
        if (string.IsNullOrEmpty(name))
            return false;

        var isExists = _properties.TryGetValue(name, out var getValue);
        if (isExists && Equals(value, getValue))
            return false;

        _properties.AddOrUpdate(name, value, (s, o) => value);

        OnPropertyChanged(name);

        return true;
    }
}
```

Как видим, методы Get() и Set() позволяют сохранять в локальный кеш необходимые свойства. Метод Set() также самостоятельно генерирует события OnPropertyChanged.

Именно от Bindable и будет наследоваться наша BaseViewModel. Это позволит удобно реализовывать генерацию OnPropertyChanged.

Также для сокращения кода в будущих потомках ViewModel будет удобно реализовать дополнительные методы создания объектов интерфейса ICommand.

Листинг 4.6 ❖ Реализация методов создания ICommand в BaseViewModel

```
readonly ConcurrentDictionary<string, ICommand> _cachedCommands = new
ConcurrentDictionary<string, ICommand>();

protected ICommand MakeCommand(Action commandAction, [CallerMemberName] string
propertyName = null) {
    return GetCommand(propertyName) ?? SaveCommand(new Command(commandAction),
propertyName);
}

protected ICommand MakeCommand(Action<object> commandAction, [CallerMemberName]
string propertyName = null) {
    return GetCommand(propertyName) ?? SaveCommand(new Command(commandAction),
propertyName);
}

ICommand SaveCommand(ICommand command, string propertyName) {
    if (string.IsNullOrEmpty(propertyName))
        throw new ArgumentNullException(nameof(propertyName));

    if (!_cachedCommands.ContainsKey(propertyName))
        _cachedCommands.TryAdd(propertyName, command);

    return command;
}

ICommand GetCommand(string propertyName) {
    if (string.IsNullOrEmpty(propertyName))
        throw new ArgumentNullException(nameof(propertyName));

    return _cachedCommands.TryGetValue(propertyName, out var cachedCommand)
        ? cachedCommand
        : null;
}
```

Теперь у нас есть все инфраструктурные методы для последующей экономии кода. А мы переходим к описанию тех методов, которые бу-

дут вызываться в неизменном виде. Речь про методы, реализующие отправку событий для NavigationService и DialogService.

Листинг 4.7 ❖ Пример отправки события перехода на новый экран и показа всплывающего уведомления

```
protected void NavigateTo(object toName,
            NavigationMode mode = NavigationMode.Normal,
            string toTitle = null,
            Dictionary<string, object> navParams = null,
            bool newNavigationStack = false,
            bool withAnimation = true,
            bool withBackButton = false,
            object fromName = null) {
    NavigateTo(toName, fromName ?? _classShortName, mode, toTitle, navParams,
newNavigationStack, withAnimation, withBackButton);
}

protected static void NavigateTo(object toName,
    object fromName,
    NavigationMode mode = NavigationMode.Normal,
    string toTitle = null,
    Dictionary<string, object> dataToLoad = null,
    bool newNavigationStack = false,
    bool withAnimation = true,
    bool withBackButton = false) {

    if (toName == null) return;

    MessageBus.SendMessage(Consts.DialogHideLoadingMessage);

    MessageBus.SendMessage(Consts.NavigationPushMessage,
        new NavigationPushInfo {
            To = toName.ToString(),
            From = fromName?.ToString(),
            Mode = mode,
            NavigationParams = dataToLoad,
            ToTitle = toTitle,
            NewNavigationStack = newNavigationStack,
            WithAnimation = withAnimation,
            WithBackButton = withBackButton
        });
}
```

Внутри ViewModel часто бывает удобно проверять наличие интернет-соединения (если его нет – переход в режим офлайн или отображение подсказки пользователю). В этом нам поможет библиотека Plugin.Connectivity.

Листинг 4.8 ❖ Реализация свойства IsConnected

```
public bool IsConnected => !CrossConnectivity.IsSupported || CrossConnectivity.
IsSupported && CrossConnectivity.Current.IsConnected;
```

При работе с сетью также может быть полезно передавать CancellationToken во все запросы, чтобы при закрытии окна происходила отмена всех запущенных во ViewModel сетевых операций.

Листинг 4.9 ❖ Создание CancellationToken и метод CancelNetworkRequests

```
readonly CancellationTokenSource _networkTokenSource = new
CancellationTokenSource();

public CancellationToken CancellationToken => _networkTokenSource?.Token ??
CancellationToken.None;

public void CancelNetworkRequests() {
    _networkTokenSource.Cancel();
}
```

Для того чтобы ViewModel могла реагировать на жизненный цикл Page, она должна корректно реализовать виртуальные методы OnPageAppearing(), OnPageDisappearing() плюс принимать параметры навигации. Метод SetNavigationParams() позволяет передать данные во ViewModel при переходе с одного окна на другое.

Листинг 4.10 ❖ Методы SetNavigationParams, OnPageAppearing и OnPageDisappearing

```
public virtual void SetNavigationParams(Dictionary<string, object> navParams) {
    NavigationParams = navParams;
}

public virtual Task OnPageAppearing() {
    return Task.FromResult(0);
}

public virtual Task OnPageDisappearing() {
    return Task.FromResult(0);
}
```

На этом мы завершили рассмотрение базовых классов, необходимых для реализации MVVM без нарушения паттерна, а также излишнего запутывания и усложнения кода.

4.3. Фундамент User Interface Layer (UI)

В предыдущих главах мы уже отмечали, что для грамотной декомпозиции приложения по слоям необходимо использовать следующие дополнительные классы, отделяющие UI от BL:

- **NavigationService** – сервис, который будет управлять открытием и закрытием страниц (Page) приложения;
- **DialogService** – сервис для отображения пользователю различных диалогов: индикатор загрузки, всплывающие уведомления, запрос на ввод данных;
- общая шина **MessageBus** для обмена данными между ViewModels, а также ViewModels и NavigaionService/DialogService.

Это минимально необходимый набор классов для грамотной реализации MVVM. Вы также можете использовать готовые классы из набора библиотек MvvmCross, MvvmLight, Prism и других MVVM-фреймворков.

4.3.1. Реализация MessageBus

Для того чтобы реализовать общую шину, мы реализуем свой singleton-класс на базе `MessaginCenter`.

Листинг 4.11 ❖ Реализация `MessageBus`

```
public class MessageBus {
    static readonly Lazy<MessageBus> LazyInstance = new Lazy<MessageBus>(() =>
new MessageBus(), true);
    static MessageBus Instance => LazyInstance.Value;

    MessageBus() {}

    public static void SendMessage(string message) {
        MessagingCenter.Send(Instance, message);
    }
```

```
    public static void SendMessage<TArgs>(string message, TArgs args) {
        MessagingCenter.Send(Instance, message, args);
    }
}
```

Как видим, единственный экземпляр MessageBus выступает в качестве отправителя сообщений в MessagingCenter. В дальнейшем остальные компоненты могут просто подписаться в MessagingCenter на получение событий от MessageBus.

Листинг 4.12 ❖ Использование MessageBus

```
// Пример: подписка на события из MessageBus

MessagingCenter.Subscribe<MessageBus, DialogToastInfo>(this,
"DialogToastMessage", DialogToastCallbackMethod);

// Пример: отправка события с помощью MessageBus

MessageBus.SendMessage("DialogToastMessage",
                new DialogToastInfo {
                    Text = text,
                    IsCenter = isCenter,
                    IsLongTime = isLongTime
                });
```

Этот MessageBus будет использоваться в NavigationService и DialogService.

4.3.2. Реализация NavigationService

Для того чтобы осуществлять переходы между окнами, в Xamarin.Forms реализованы специальные классы Navigation на уровне UI. К сожалению, использование данных механизмов на уровне ViewModels ведет к появлению зависимостей от UI, что является грубым нарушением архитектуры. Для решения этой проблемы часто предлагается воспользоваться готовым MVVM-фреймворком, но они перегружены лишней функциональностью и сильно дублируют реализованные в Xamarin.Forms модули. Поэтому мы рекомендуем использовать свой NavigationService в виде белого ящика, что позволит филигранно доработать его под нужды вашего проекта.

Основные задачи, которые предстоит решать NavigationService:

- получение событий на открытие/закрытие окон;
- манипуляции стеком навигации в ответ на полученное событие.

Существуют различные возможности связывания ViewModel и Page, мы же остановимся на привычном для Asp.Net подходе **Convention over configuration** (Соглашение вместо конфигурирования). В этом подходе для связывания View и Controller (речь про Asp.net) их просто необходимо положить в правильную папку и назвать правильным образом (например, MainView и MainViewController). Наш пример включает использование рефлексии (reflection) для поиска нужных классов. Это не самый лучший подход с точки зрения производительности, но он нагляден и выполняется всего один раз при старте приложения. Вы можете реализовать свои механизмы.

Как уже отмечалось в главе 2, мы будем называть наши ViewModel и Page согласно документации. На уровне NavigationService будет достаточно пройти по именам всех классов в библиотеке .NET Standard, содержащей UI и BL, и сохранить у себя в кеше информацию об этих классах.

Листинг 4.13 ❖ Поиск классов Pages и ViewModels

```
static string GetTypeBaseName(MemberInfo info) {
    if (info == null) throw new ArgumentNullException(nameof(info));
    return info.Name.Replace(@"Page", "").Replace(@"ViewModel", "");
}

static Dictionary<string, Type> GetAssemblyPageTypes() {
    return typeof(BasePage).GetTypeInfo().Assembly.DefinedTypes
        .Where(ti => ti.IsClass && !ti.IsAbstract && ti.Name.Contains(@"Page")
&& ti.BaseType.Name.Contains(nameof(BasePage)))
        .ToDictionary(GetTypeBaseName, ti => ti.AsType());
}

static Dictionary<string, Type> GetAssemblyViewModelTypes() {
    return typeof(BaseViewModel).GetTypeInfo().Assembly.DefinedTypes
        .Where(ti => ti.IsClass && !ti.IsAbstract && ti.Name.
Contains(@"ViewModel") &&
                    ti.BaseType.Name.Contains(@"ViewModel"))
        .ToDictionary(GetTypeBaseName, ti => ti.AsType());
}
```

Инициализировать список ViewModel и Page будет необходимо в конструкторе, где также уместно подписаться на события навигации:

Листинг 4.14 ❖ Конструктор NavigationService

```
static readonly Lazy<NavigationService> LazyInstance = new
Lazy<NavigationService>(() => new NavigationService(), true);
readonly Dictionary<string, Type> _pageTypes;
readonly Dictionary<string, Type> _viewModelTypes;
Page _rootPage;

public static NavigationService Instance => LazyInstance.Value;

NavigationService() {
    _pageTypes = GetAssemblyPageTypes();
    _viewModelTypes = GetAssemblyViewModelTypes();
    MessagingCenter.Subscribe<MessageBus, NavigationPushInfo>(this, Consts.
NavigationPushMessage, NavigationPushCallback);
    MessagingCenter.Subscribe<MessageBus, NavigationPopInfo>(this, Consts.
NavigationPopMessage, NavigationPopCallback);
}
```

Как видим, ничего сложного. Сам NavigationService будет реализован в виде синглетона. А вот код обработчиков событий навигации:

Листинг 4.15 ❖ Обработчики событий навигации

```
async void NavigationPushCallback(MessageBus bus, NavigationPushInfo
navigationPushInfo) {
    try {
        if (navigationPushInfo == null) throw new ArgumentNullException(nameof(
navigationPushInfo));
        if (string.IsNullOrEmpty(navigationPushInfo.To?.ToString())) throw new
FieldAccessException(@"'To' page value should be set into NavigationPushInfo");

        await Push(navigationPushInfo);
    }
    catch (Exception e) {
        throw;
    }
}

async void NavigationPopCallback(MessageBus bus, NavigationPopInfo
```

```
navigationPopInfo) {
    if (navigationPopInfo == null) throw new ArgumentNullException(nameof(navig
ationPopInfo));
    await Pop(navigationPopInfo);
}

Task Push(NavigationPushInfo pushInfo) {
    switch (pushInfo.Mode) {
        case NavigationMode.Normal:
            return PushNormal(pushInfo);
        case NavigationMode.Modal:
            return PushModal(pushInfo);
        case NavigationMode.Custom:
            return PushCustom(pushInfo);
        case NavigationMode.Root:
            return PushRoot(pushInfo);
        case NavigationMode.PopUp:
            return PushPopUp(pushInfo);
        default:
            throw new NotImplementedException();
    }
}

Task Pop(NavigationPopInfo popInfo) {
    switch (popInfo.Mode) {
        case NavigationMode.Normal:
            return PopNormal(popInfo);
        case NavigationMode.Modal:
            return PopModal(popInfo);
        case NavigationMode.Custom:
            return PopCustom(popInfo);
        case NavigationMode.PopUp:
            return PopPopUp(popInfo);
        case NavigationMode.Root:
            return PopRoot(popInfo);
        default:
            throw new NotImplementedException();
    }
}
```

Полные исходные коды методов `Push*` и `Pop*` смотрите в репозитории Order King.

Для того чтобы иметь возможность расширенного управления процессом навигации, необходимо передавать корректные значения в параметрах навигации.

Листинг 4.16 ❖ Структуры `NavigationPushInfo` и `NavigationPopInfo`

```
public enum NavigationMode {
    Normal,
    Modal,
    Custom,
    Root,
    PopUp
}

public class NavigationPushInfo {
    public string From { get; set; }
    public string To { get; set; }
    public string ToTitle { get; set; }
    public Dictionary<string, object> NavigationParams { get; set; }
    public NavigationMode Mode { get; set; } = NavigationMode.Normal;
    public bool WithAnimation { get; set; } = true;
    public bool WithBackButton { get; set; } = true;
    public bool NewNavigationStack { get; set; }
}

public class NavigationPopInfo {
    public NavigationMode Mode { get; set; } = NavigationMode.Normal;
    public bool WithAnimation { get; set; } = true;

}
```

В большинстве приложений возникает необходимость расширенного управления навигаций – например, переход на главный экран после успешной авторизации, состоящей из нескольких шагов (и Page). Для этого можно реализовать обработчик для режима Custom внутри методов `PushCustom` и `PopCustom` – здесь вы можете самостоятельно манипулировать стеком навигации, но следует быть внимательным и осторожным.

Чтобы при навигации происходило связывание объектов Page и ViewModel, необходимо использовать методы BasePage, о которых мы поговорим в разделе 4.4.

Листинг 4.17 ❖ Создание страниц и связывание Page-ViewModel

```
Page GetInitializedPage(string toName,
    NavigationMode mode = NavigationMode.Normal,
    Dictionary<string, object> navParams = null,
    bool newNavigationStack = false,
    bool withAnimation = true,
    bool withBackButton = true,
    string toTitle = null) {

    var page = GetPage(toName);
    var viewModel = GetViewModel(toName);
    viewModel.SetNavigationParams(navParams);
    page.SetViewModel(viewModel);

    if (!string.IsNullOrEmpty(toTitle)) page.Title = toTitle;

    return newNavigationStack
        ? new NavigationPage(page)
        : (Page) page;
}

Page GetInitializedPage(NavigationPushInfo navigationPushInfo) {
    return GetInitializedPage(navigationPushInfo.To, navigationPushInfo.Mode,
navigationPushInfo.NavigationParams,
        navigationPushInfo.NewNavigationStack, navigationPushInfo.
WithAnimation,
        navigationPushInfo.WithBackButton, navigationPushInfo.ToTitle);
}

BasePage GetPage(string pageName) {
    if (!_pageTypes.ContainsKey(pageName)) throw new
KeyNotFoundException($@"Page for {pageName} not found");

    BasePage page;
    try {
        var pageType = _pageTypes[pageName];
        var pageObject = Activator.CreateInstance(pageType);
        page = pageObject as BasePage;
    }
    catch (Exception e) {
        throw new TypeLoadException($@"Unable create instance for {pageName}
Page", e);
    }
```

```
    return page;
}

BaseViewModel GetViewModel(string pageName) {
    if (!_viewModelTypes.ContainsKey(pageName)) throw new KeyNotFoundException(
$@"ViewModel for {pageName} not found");

    BaseViewModel viewModel;
    try {
        viewModel = Activator.CreateInstance(_viewModelTypes[pageName]) as
BaseViewModel;
    }
    catch (Exception e) {
        throw new TypeLoadException($@"Unable create instance for {pageName}
ViewModel", e);
    }

    return viewModel;
}
```

На этом мы описали все ключевые механизмы работы NavigationService.

4.3.3. Реализация DialogService

Для того чтобы сделать приложение более интерактивным, далеко не всегда необходимо создавать целые экраны. Иногда достаточно просто показать диалоговое сообщение.

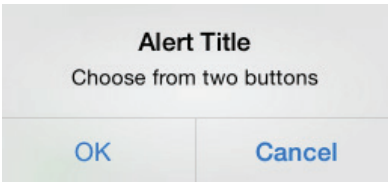


Рис. 4.5 ❖ Пример всплывающего диалога

При разработке приложения с паттерном MVVM логика взаимодействия с пользователем реализуется во ViewModel, однако диалоговые сообщения являются частью UI. И если ViewModel начинает напрямую обращаться к UI, то это ведет к нарушению архитектуры и запутыванию кода (см. раздел 3.4).

Чтобы разорвать обратную связь ViewModel-UI, нам необходимо реализовать свой DialogService. По традиционной схеме он будет получать события от MessageBus и отображать необходимые диалоги и уведомления (индикатор загрузки, Toast-уведомление и прочее).

Листинг 4.18 ❖ Конструктор DialogService и подписка на события

```
DialogService() {
    MessagingCenter.Subscribe<MessageBus, DialogAlertInfo>(this, Consts.
DialogAlertMessage, DialogAlertCallback);
    MessagingCenter.Subscribe<MessageBus, DialogSheetInfo>(this, Consts.
DialogSheetMessage, DialogSheetCallback);
    MessagingCenter.Subscribe<MessageBus, DialogQuestionInfo>(this, Consts.
DialogQuestionMessage, DialogQuestionCallback);
    MessagingCenter.Subscribe<MessageBus, DialogEntryInfo>(this, Consts.
DialogEntryMessage, DialogEntryCallback);

    MessagingCenter.Subscribe<MessageBus, string>(this, Consts.
DialogShowLoadingMessage, DialogShowLoadingCallback);

    MessagingCenter.Subscribe<MessageBus>(this, Consts.
DialogHideLoadingMessage, DialogHideLoadingCallback);

    MessagingCenter.Subscribe<MessageBus, DialogToastInfo>(this, Consts.
DialogToastMessage, DialogToastCallback);
}
```

Листинг 4.19 ❖ Пример реализации метода DialogService

```
void DialogToastCallback(MessageBus bus, DialogToastInfo toastInfo) {
    if (toastInfo == null) throw new ArgumentNullException(nameof(toastInfo));
    Device.BeginInvokeOnMainThread(() =>
        UserDialogs.Instance.Toast(new ToastConfig(toastInfo.Text) {
            Duration = TimeSpan.FromSeconds(toastInfo.IsLongTime ? 2 : 1)})
    );
}
```

Для реализации выше используется библиотека Acr.UserDialogs, но вы также можете легко заменить и доработать предложенные механизмы под нужды вашего проекта.

4.3.4. Реализация BasePage

Для реализации MVVM нам потребуется создать свой базовый класс для страниц приложения. BasePage будет наследоваться от Content-

Page, поэтому для других типов страниц (например, TabbedPage) вам потребуется создать свои базовые классы (например, BaseTabbedPage). К сожалению, это приведет к небольшому дублированию кода, но зато заметно упростит дальнейшую разработку.

Листинг 4.20 ❖ Интерфейс IBasePage

```
public interface IBasePage : IDisposable {
    void SetViewModel(BaseViewModel viewModel);
}
```

Итак, в обычной Page уже есть поле `BindingContext`, и наша задача реализовать в BasePage следующие дополнительные поля и методы:

- свойство ViewModel – ссылка на ViewModel, привязанную к Page;
- переопределить метод `OnPageAppearing`, чтобы в нем вызывать аналогичный метод у привязанной ViewModel;
- переопределить метод `OnPageDisappearing`, чтобы в нем вызывать аналогичный метод у привязанной ViewModel.

Также будет полезно переопределить метод `Dispose`, чтобы вызывать соответствующий метод у ViewModel.

Листинг 4.21 ❖ Реализация BasePage

```
public class BasePage: ContentPage, IBasePage {
    public BaseViewModel ViewModel => BindingContext as BaseViewModel;

    ~BasePage() {
        Dispose();
    }

    public void Dispose() {
        BindingContext = null;
    }

    public void SetViewModel(BaseViewModel viewModel) {
        BindingContext = ViewModel = viewModel;
    }

    protected override void OnAppearing() {
        base.OnAppearing();
        Task.Run(() => ViewModel?.OnPageAppearing());
    }
```

```
    protected override void OnDisappearing() {
        base.OnAppearing();
        Task.Run(() => ViewModel?.OnPageDissapearing());
    }

    protected override void OnParentSet() {
        base.OnParentSet();
        if (Parent != null) return;
        Dispose();
    }

    public virtual void Dispose() {
        ViewModel?.Dispose();
        BindingContext = null;
    }
}
```

Как мы уже видели в коде NavigationService, связывание Page и ViewModel происходит на этапе создания экземпляров страниц, в этом помогает метод `BasePage.SetViewModel`.

Если вам необходимо использовать IBasePage для создания, например, `BaseContentPage`, `BaseTabbedPage`, `BaseCarouselPage` и др., то каждый Base-класс должен включать в себя одну и ту же реализацию методов `IBasePage`, как было показано в листинге 4.11.

Итак, мы рассмотрели основные вопросы закладывания фундамента для вашего проекта в виде рабочей технической документации и «скелета» проекта. Теперь можно перейти к автоматизации самого процесса разработки мобильных приложений.

Глава 5

Mobile DevOps

По мере того как DevOps набирает популярность в мире больших проектов и команд, его пытаются также адаптировать и к относительно небольшим мобильным приложениям. Так появился немного маркетинговый термин Mobile DevOps. Однако сам по себе DevOps – это не только культура, но и набор практических методик, используемых в команде, работающей над созданием и развитием продукта.

В нашей заключительной главе мы рассмотрим, что такое Mobile DevOps и автоматизация, что тестировать в мобильных приложениях и как следить за здоровьем продукта, ушедшего в реальную эксплуатацию. В качестве единого инструмента Mobile DevOps мы будем использовать Visual Studio App Center.

5.1. Про DevOps

Сам по себе DevOps медленно шагает по планете в обнимку с гибкими методиками управления. Однако если присмотреться поглубже, то DevOps – это в первую очередь культура, в которой вся команда, развивающая и обслуживающая ИТ-систему, работает как одно целое. Именно общение внутри команды является обязательным для выстраивания здорового взаимодействия между бизнесом, программистами, системными администраторами и тестировщиками.

Вторым важным элементов любого DevOps-процесса является обучение команды. В идеальном мире команда (начиная с бизнеса) должна учиться все лучше удовлетворять нужды конечного пользователя, создавая ценность для потребителя. Но на практике еще и ценность надо научиться считать и делать из этого грамотные выводы. Для обучения в любом случае необходима обратная связь от реаль-

ных пользователей. Такой связью становятся системы непрерывного мониторинга жизнеспособности ИТ-систем и разнообразные системы логирования.

И чтобы этот DevOps существовал не только в головах у разработчиков, его необходимо реализовать на практике с помощью инструментов, которые подойдут под нужды задачи. Частью DevOps-культуры также является стремление к автоматизации рутинных действий. Это делает процесс разработки и развития ИТ-систем менее болезненным и более эффективным.

Сам по себе DevOps появился как подход для больших команд и компаний, работающих над крупными ИТ-системами, состоящими из большого числа модулей, часто написанных на разных языках программирования и предназначенных для разных сред выполнения. Однако по мере популяризации термина и подхода DevOps стал адаптироваться различными командами, решающими широкий спектр задач.

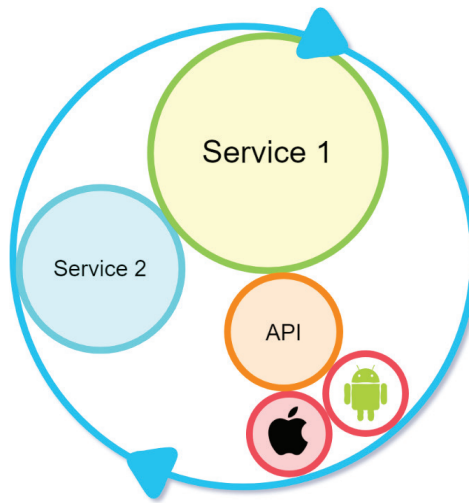


Рис. 5.1 ❖ Единый цикл развития ИТ-комплекса

Чаще всего DevOps ассоциируется с крупными цифровыми продуктами, разработкой и обслуживанием которых занимаются десятки, сотни, а иногда и тысячи специалистов. У многих команд уже выработались свои практики и свои инструменты для поддержания процес-

са DevOps – системы автоматической сборки, тестирования, развертывания и мониторинга. Обычно это все создается на собственной инфраструктуре, но активно развиваются и облачные инструменты CI/CD.

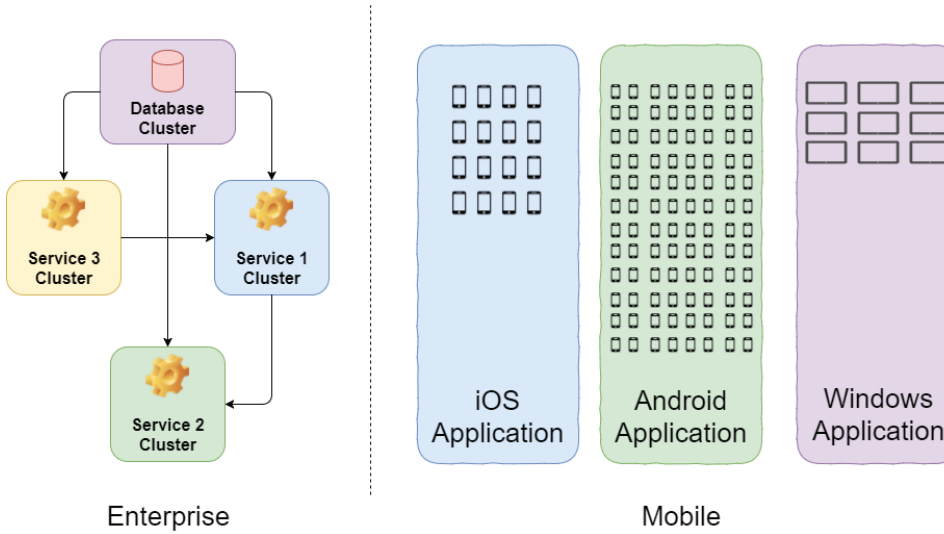


Рис. 5.2 ❖ Отличие задач корпоративной и мобильной разработки

Все это мало ассоциируется с разработкой мобильных приложений. Mobile DevOps является уменьшенной версией обычного DevOps, так как мобильные приложения – это в первую очередь удобный интерфейс для взаимодействия с внешними ИТ-системами. Меньший масштаб команды и свои специфические проблемы, характерные именно для мобильных приложений:

- различные платформы;
- различные устройства.

5.2. Особенности Mobile CI/CD

Итак, с культурой взаимодействия и рабочей документацией мы определились, и теперь можно переходить к инструментальной поддержке в виде CI/CD-конвейера. Чем больше операций получится автоматизировать, тем лучше.

У мобильной разработки есть три отличительных проблемных места:

- различные операционные системы и их версии. Приложение должно корректно работать на широком спектре самых различных ОС, каждая из которых имеет свои особенности и ограничения;
- различные варианты архитектуры центрального процессора. Железо смартфонов и планшетов постоянно улучшается, однако не надо забывать и о «старичках» пятилетней давности, которые могут быть на руках у ваших реальных пользователей;
- различные разрешения экранов устройств. Независимо от количества пикселей или соотношения сторон интерфейс мобильного приложения должен корректно отображаться на всех устройствах.

В реальной практике невозможно обойтись без ручного тестирования на смартфонах или планшетах. Однако часть кода легко покрывается автоматическими тестами на базе Unit Testing (далее просто unit-тесты).

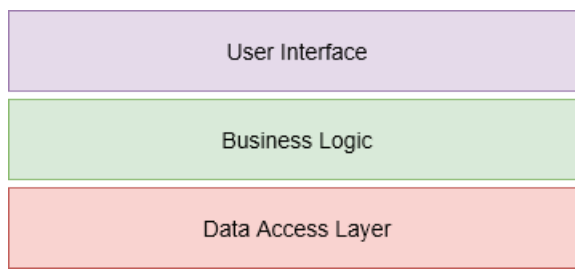


Рис. 5.3 ❖ Многослойная архитектура мобильного приложения

Полное покрытие тестами разумнее осуществлять по двум направлениям:

- unit-тесты (функциональные, интеграционные) для слоя доступа к данным (Data Access Layer) или Data Services;
- UI-тесты (функциональные, регрессионные) для слоев Business Logic и User Interface.

Покрывать все unit-тестами в мобильных приложениях не представляется возможным, плюс это снижает скорость разработки и обновления системы. Для этапа автоматического unit-тестирования в рамках CI/CD-конвейера будет полезным покрыть следующие механизмы слоя DAL:

- методы доступа к Backend API;
- методы доступа к данным (Data Services).

Подробнее о unit-тестировании мы расскажем в разделе 9.4.

Если говорить об автоматических UI-тестах, то можно протестировать следующее:

- работоспособность приложения на реальных устройствах с нужными характеристиками;
- возможность выполнить ключевые бизнес-сценарии.

По результатам UI-тестов собираются скриншоты с каждым шагом на каждом устройстве, на их основе QA-инженер в ручном режиме может просмотреть правильность верстки приложения на разных разрешениях и размерах экранов.

Удобство, плавность анимаций и другие пользовательские характеристики сложно отдать на откуп автотестам, их лучше оставить живым тестировщикам и бета-пользователям.

5.3. Конвейер CI/CD

Инструментарий для сборки и публикаци установочных пакетов существует уже очень давно. Обычно это набор скриптов, который использовался на build-машине в команде мобильной разработки. В последнее время, однако, стали набирать популярность облачные сервисы для реализации CI/CD. В нашем руководстве мы остановимся на App Center от Microsoft, который является универсальным инструментом в режиме «единого окна» и включает в себя полный набор необходимых сервисов: сборка, тестирование, дистрибуция и мониторинг.

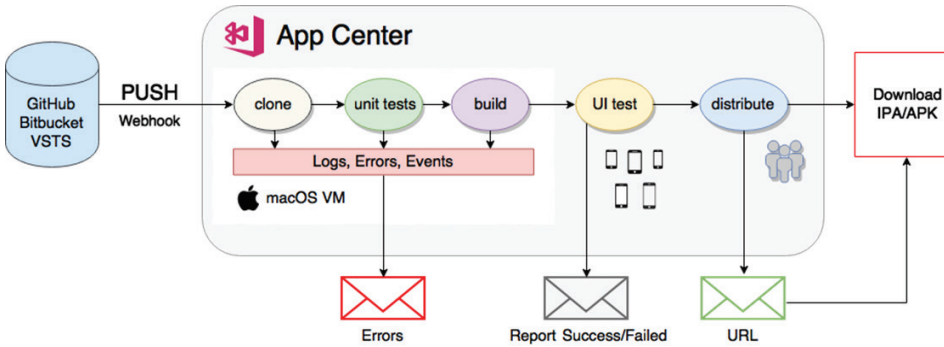


Рис. 5.4 ❖ Работа конвейера CI/CD в облачном сервисе App Center

Вы можете запускать CI/CD-конвейер как самостоятельно командой Slack, так и автоматически по каждому новому Push в репозиторий. С использованием облачного CI/CD лучше настроить автоматический вариант – это сократит путь получения информации об ошибках. Ниже показаны необходимые настройки конвейера сборки в App Center (добавить проект в App Center, подключить репозиторий кода и настроить Build для разных веток).

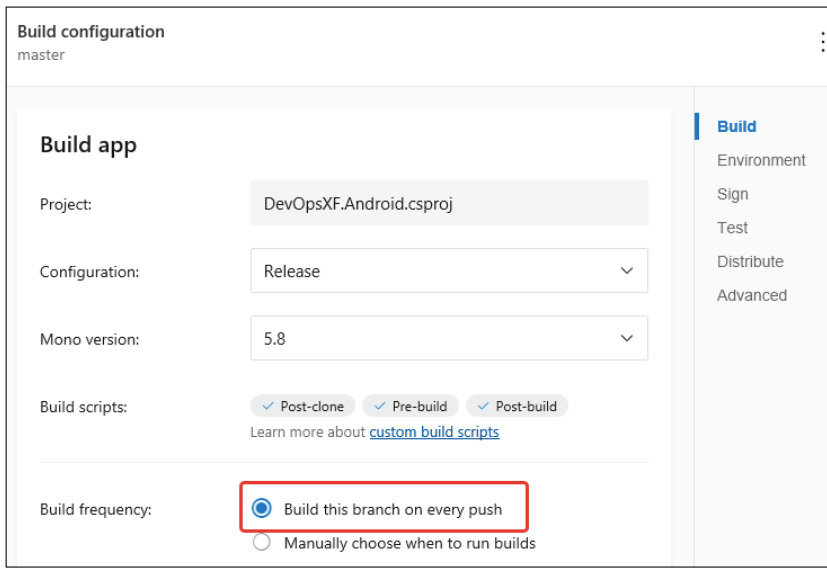


Рис. 5.5 ❖ Запуск конвейера и сборка для каждого commit в репозиторий

Ручной режим лучше оставить для более сложных сценариев – например, подготовка отдельной сборки для A/B-тестирования. Или запуск более широкого набора функциональных тестов перед поставкой бета-пользователям.

Для того чтобы реализовать возможность запускать свои автотесты в App Center, вам потребуется написать bash-скрипты для iOS/Android и положить их в папку с проектом:

- **appcenter-post-clone.sh** – запускается сразу после того, как удаленный репозиторий склонирован на build-машину (mac для iOS/Android). Здесь можно запускать unit-тесты;
- **appcenter-pre-build.sh** – выполняется перед сборкой приложения, здесь можно, например, прописывать BUILD_ID в версии приложения (например, x.x.BUILD_ID);
- **appcenter-post-build.sh** – запускается сразу после успешной сборки приложения. На этом шаге можно запускать Smoke-тесты на реальных смартфонах/планшетах.

Так как сборка (включая упаковку и подпись сертификатами) реальных мобильных приложений занимает достаточно длительное время (более 5–10 мин), то можно запускать unit-тесты на шагах post-clone или pre-build; это позволит провести быструю диагностику ключевых механизмов. А вот smoke-тестирование, крайне желательное в мобильной разработке, необходимо делать уже после сборки. Это позволит проверить, что приложение как минимум будет запускаться на реальных смартфонах с нужными версиями ОС.

Build-машина

Для того чтобы собирать приложения, в App Center используются виртуальные маки, работающие на железе Apple. Машины имеют богатый набор возможностей, которые вы можете также использовать в своих bash-скриптах.

Если вы еще не писали shell-скрипты для bash, то нужно будет немного попрактиковаться и почитать документацию: **bing.com/search?q=bash+для+начинающих**.

Языки и среды
Java 1.7.0_80
Java 1.8.0_162
Java 9.0.4
Node.js 6.12.3
PowerShell 6.0.1
Python 2.7.10
Python 3.6.4
Ruby 2.5.0p0
.NET Core SDK 2.0.3
Go 1.9.3

Управление пакетами
Bundler 1.16.0
Carthage 0.27.0
CocoaPods 1.4.0
Homebrew 1.5.2
NPM 3.10.10
Yarn 1.3.2
NuGet 4.3.0
pip 9.0.1

Управление проектами
Apache Maven 3.5.2
Gradle 4.5

Утилиты
curl 7.54.0
Git 2.16.1
Git LFS 2.3.4
GNU Wget 1.19.4
Subversion (SVN) 1.9.7
fastlane 2.78.0

Xcode
Xcode 8.0-9.2
Nomad CLI 2.7.4
Nomad CLI IPA 0.14.3
xcpretty 0.2.8
xctool 0.3.4

Рис. 5.6 ❖ Предустановленный набор утилит на виртуальный macOS

В нашем примере мы создали автоматический CI/CD-конвейер для ветки master в репозитории на GitHub.

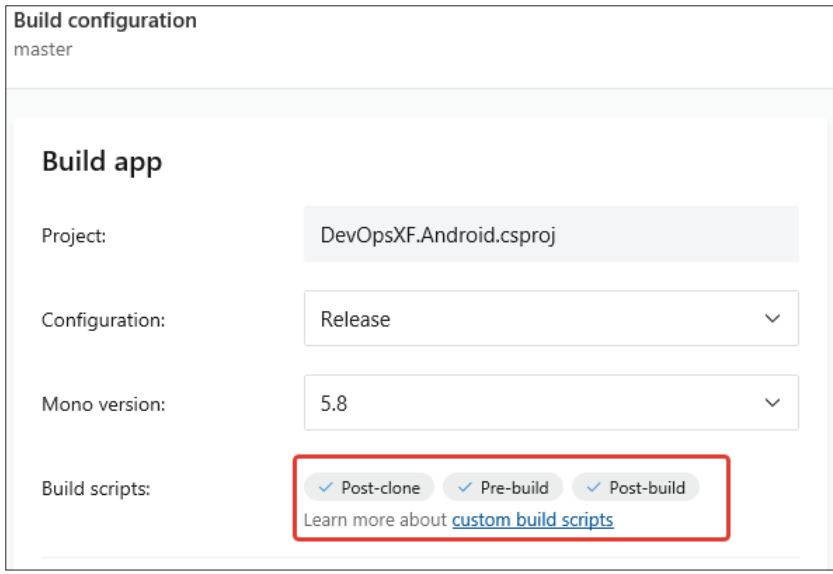


Рис. 5.7 ❖ Автоматическое определение нужных скриптов в исходных кодах

Как видим, App Center автоматически нашел наши скрипты, которые были добавлены в папку с проектом (где лежат файлы .xcodeproj, build.gradle, .csproj, .sln или package.json).

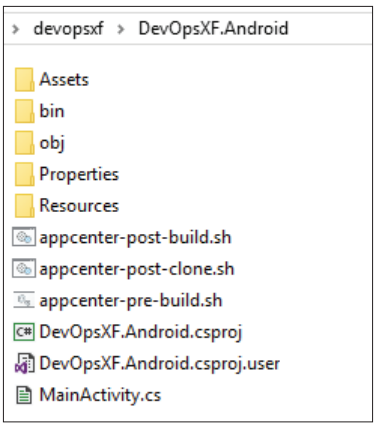


Рис. 5.8 ❖ Файлы со скриптами в папке проекта

При написании скриптов может быть необходимо использовать переменные окружения bash – внешний скрипт или программа записывает в сессию bash свою переменную, например APPCENTER_SOURCE_DIRECTORY, и это позволяет использовать значение этой переменной в своих скриптах. Ключевые предустановленные переменные окружения в App Center:

APPCENTER_BUILD_ID	Номер сборки увеличивается на 1 при каждом билде
APPCENTER_BRANCH	Ветка репозитория, из которой запускается сборка
APPCENTER_OUTPUT_DIRECTORY	Папка, в которой будут сохраняться результаты сборки – пакеты IPA/APK, другие бинарные файлы
APPCENTER_TRIGGER	Как был запущен сценарий сборки – вручную на сайте (manual) или автоматически при push в репозиторий (continuous)

Подробнее: **docs.microsoft.com/en-us/appcenter/build/custom/variables**.

Вы также можете настроить свои переменные окружения в параметрах сборки.

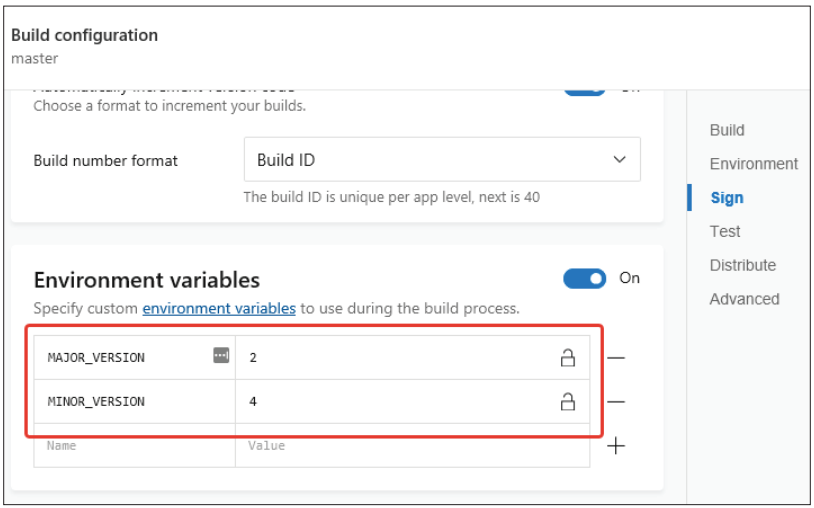


Рис. 5.9 ❖ Свои переменные окружения в настройках конвейера

В ваших скриптах можете использовать переменные **$MAJOR_VERSION** и **$MINOR_VERSION**, как будет показано в примере для **appcenter-pre-build.sh**.

Теперь рассмотрим каждый скрипт в отдельности. Начнем мы с шага post-clone, на котором будут запускаться Unit-тесты.

Листинг 5.1 ❖ Исходные коды **appcenter-post-clone.sh**

```
#!/usr/bin/env bash -e

echo "Found Unit test projects:"
find $APPCENTER_SOURCE_DIRECTORY -regex '.*UnitTests.*\.csproj' -exec echo {} \;
echo
echo "Run Unit test projects:"
find $APPCENTER_SOURCE_DIRECTORY -regex '.*UnitTests.*\.csproj' | xargs dotnet
test;
```

Как видим, скрипт ищет папки с .csproj-файлами (проекты Visual Studio), название которых содержит UnitTests и запускает в них unit-тесты на базе NUnit. В нашем примере unit-тесты реализованы на базе .Net Core. Вы можете использовать любые привычные инструменты для unit-тестирования в зависимости от стека, на котором будете разрабатывать приложения.

Листинг 5.2 ❖ Исходные коды **appcenter-pre-build.sh**

```
#!/usr/bin/env bash

MANIFEST="$APPCENTER_SOURCE_DIRECTORY/DevOpsXF.Android/Properties/
AndroidManifest.xml"
NEW_VERSION=${MAJOR_VERSION}.${MINOR_VERSION}.${APPCENTER_BUILD_ID}

sed -i -e "s/versionName=\".*\"/versionName=\"${NEW_VERSION}\"/g" $MANIFEST
```

На этом шаге вы можете добавить `BUILD_ID` к версии приложения в формате x.x.BUILD_ID. Также здесь мы можем выполнить какие-либо дополнительные действия перед сборкой.

Листинг 5.3 ❖ Исходные коды **appcenter-post-build.sh**

```
#!/usr/bin/env bash -e

appName="YOUR_APP_NAME"
appCenterLoginApiToken="YOUR_API_TOKEN"

uiTestProjectName="DevOpsXF.UITests"
appFileName="com.binwell.DevOpsXF.apk"
locale="ru_RU"

if [ "$APPCENTER_BRANCH" == "master" ];
then
    msbuild $APPCENTER_SOURCE_DIRECTORY/$uiTestProjectName/
/p:Configuration=Release
    appcenter test run uitest --app $appName --devices "slavachernikoff/
Smoke" --test-series "Smoke" --include-category "Smoke" \
    --app-path $APPCENTER_OUTPUT_DIRECTORY/$appFileName --locale $locale
--build-dir $APPCENTER_SOURCE_DIRECTORY/$uiTestProjectName/bin/Release \
    --token $appCenterLoginApiToken
fi
```

Если сборка дошла до этого шага, значит, у вас на руках есть установочные пакеты APK/IPA. У многих команд CI/CD-конвейер на этом шаге обрывался, так как требовалась проверка на реальных устройствах, а свои фермы для автотестов были дорогим удовольствием. Мы будем использовать App Center Test Cloud для автоматических Smoke-

тестов на запуск приложения. Подробнее о Smoke-тестировании будет рассказано далее.

5.4. Тестирование

С автоматическим тестированием многие разработчики уже знакомы, поэтому мы перейдем сразу к практике и рассмотрим, как это использовать в рамках единого CI/CD-конвейера.

В мобильных приложениях (как и UI-приложениях вообще) нет возможности покрыть весь код автотестами на базе консольных Unit Tests. Многое требует взаимодействия с пользовательским интерфейсом или платформенной функциональностью. Если рассмотреть многослойную архитектуру, которой обычно придерживаются в мобильных проектах, то можно выделить DAL, на 100 % поддающийся автотестам.

Рис. 5.10 ❖ Типовая архитектура приложения на базе Xamarin.Forms

Отдельно можно покрыть тестами механизмы интеграции с внешними REST-сервисами, это позволит оперативно узнавать об изменениях и проблемах внешних API (используемых в приложении), если станет частью автоматического конвейера.

Для того чтобы понять, как можно включить автоматическое UI-тестирование в ваш процесс разработки, давайте определимся с классами проблем, которые можно выявить с их помощью:

- падения (crashes) приложения на всех или некоторых моделях устройств, логи и crash-репорты помогают быстрее найти проблему;
- несоответствие поведения интерфейса нужным сценариям – неверно осуществляется переход между экранами или не работают элементы пользовательского интерфейса;
- проблемы с верстской на всех или некоторых моделях устройств – анализ скриншотов с разных устройств позволяет найти ошибки в интерфейсе, когда что-нибудь обрезается, не умещается или не отображается.

Плюс UI-тестирование позволяет снять «зуд» ручного тестирования, когда одни и те же сценарии тестировщик вручную проверяет на большом парке устройств день за днем. В зависимости от того, для какой платформы или фреймворка вы ведете разработку (iOS, Android, Xamarin, ReactNative), вам доступны следующие фреймворки для тестирования:

- Appium (JUnit/Java) для Android и iOS;
- Espresso (Java) для Android;
- Calabash для Android и iOS;
- Xamarin.UITest для Android и iOS;
- XCUITest для iOS.

В нашем примере мы остановимся на Xamarin.UITest для Android, который позволяет писать тестовые скрипты на C#.

Главная задача, которую решают UI-тесты, – имитация действий реального пользователя на реальном устройстве: нажатия на экран, ввод данных, жесты. Тестировщик или разработчик может сам написать тест на привычном ему языке программирования или фреймворке. При этом следует учитывать, что тесты могут не просто взаимодействовать с приложением как с черным ящиком (выбирать элементы по содержащимся в них текстам или координатам экрана),

но и нажимать, например, на кнопку с внутренним идентификатором `CalculateButton`, указанным в коде программы. Для этого нужно установить значение в специальное поле (см. документацию по тестовым фреймворкам) у UI-компонента. В случае с Xamarin.Forms необходимо установить свойство `AutomationId`, которое будет автоматически присвоено нативным полям `AccessibilityIdentifier` в iOS и `ContentDescription` в Android.

```
<StackLayout
    VerticalOptions="Center"
    HorizontalOptions="Center">
    <Label Text="Enter nummber of Euros (EUR)"  />
    <StackLayout Orientation="Horizontal">
        <Entry Placeholder="0" Keyboard="Numeric" AutomationId="ValueEntry" x:Name="ValueEntry" WidthRequest="60"/>
        <Label Text="EUR" VerticalOptions="Center"/>
    </StackLayout>
    <Button Text="Calculate" Clicked="CalculateOnClicked" AutomationId="CalculateButton"/>
</StackLayout>
```

Рис. 5.11 ❖ Указание свойства AutomationId для автоматических тестов

Подробнее о том, как писать UI-тесты, вы можете узнать из официальной документации: **https://docs.microsoft.com/en-us/appcenter/test-cloud/uitest/**.

Если вы используете другие фреймворки, то найти правильную документацию можно здесь: **https://docs.microsoft.com/en-us/appcenter/test-cloud/supported-frameworks**.

Тестирование должно носить системный характер, поэтому нет необходимости «всегда тестировать все на всех устройствах». Это долго и очень дорого. Если рассматривать процесс разработки мобильных приложений, то можно выделить следующие наборы автоматических функциональных UI-тестов:

- **Smoke-тесты**, их лучше добавить как элемент автоматического CI/CD-конвейера. Проверяют, что приложение просто запускается на реальных устройствах. Находят проблемы после обновления зависимых библиотек (не поддерживаются на самых старых или самых новых версиях ОС или на определенной архитектуре процессора) и позволяют быть уверенными, что сборка будет запускаться;
- **Acceptance (Beta, Nightly)** – набор ключевых пользовательских сценариев, для реализации которых приложение создается. Позволяют проверить, что как минимум все ключевые элементы

пользовательского интерфейса и бизнес-сценарии выполняются корректно;

- **GUI** – расширенный набор коротких тестов для проверки всех экранов (верстка, работоспособность ключевых контролов) на максимально широком парке устройств.

Предложенные наборы тестов позволят вам провести комплексное тестирование приложения в автоматическом режиме, разделив его на этапы (в примере показана разбивка на двухнедельные спринты).

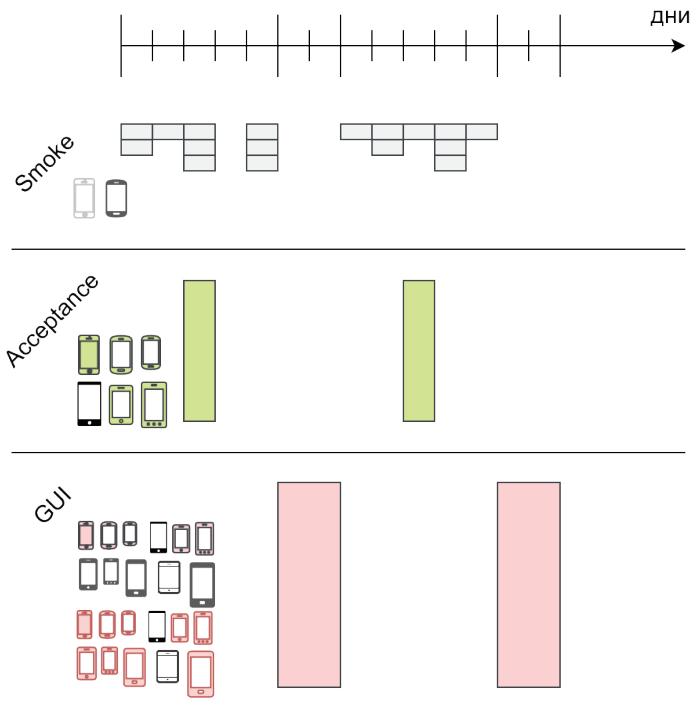


Рис. 5.12 ❖ Запуск различных тестов

И если Smoke-тесты лучше сделать элементом автоматического конвейера, то вот тесты Acceptance и GUI будут запускаться тестировщиком вручную. Перед этим их лучше отладить на локальной машине.

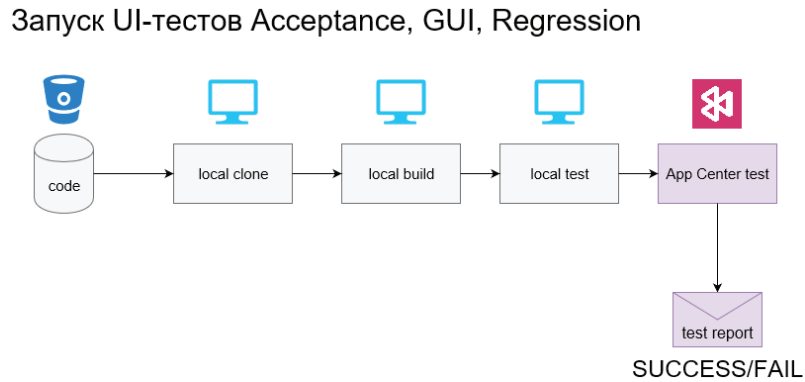


Рис. 5.13 ❖ Процесс отправки UI-тестов в App Center с предварительным тестированием

Для того чтобы как-то разделять между собой разные виды тестов (по умолчанию всегда запускаются все тесты в проекте), тестировщик может использовать механизм категорий, доступных в популярных фреймворках. Для Xamarin.UITest необходимо указать атрибут `Category` у каждого теста или отдельного метода.

```
[TestFixture(Platform.Android)]
[TestFixture(Platform.iOS)]
[Category("Smoke")]
1 reference | Slava Chernikoff, 1 day ago | 1 author, 2 changes
public class SmokeTests : BaseTests {
    0 references | Slava Chernikoff, 1 day ago | 1 author, 1 change
    public SmokeTests(Platform platform) : base(platform) {
    }

    [Test]
    0 references | Slava Chernikoff, 1 day ago | 1 author, 2 changes
    public void AppLaunches() {
        App.Screenshot("SMOKE testing screenshot");
    }
}
```

Рис. 5.14 ❖ Указание Category для разделения тестов по видам и наборам

Далее необходимо учитывать эти категории при отправке тестов в App Center:

```
appcenter test run uitest --app "binwell/DevOpsXF-Android" --devices "binwell/smoke" --test-series "smoke"  --include-category "Smoke" \
--app-path $APPCENTER_OUTPUT_DIRECTORY/com.binwell.DevOpsXF.apk --locale "ru_RU" \
--build-dir $APPCENTER_SOURCE_DIRECTORY/"DevOpsXF.UITests"/bin/Release
```

Рис. 5.15 ❖ Запуск теста в облаке с помощью командной строки

После завершения каждого теста на почту приходит уведомление с результатами тестирования, которые также доступны на сайте.

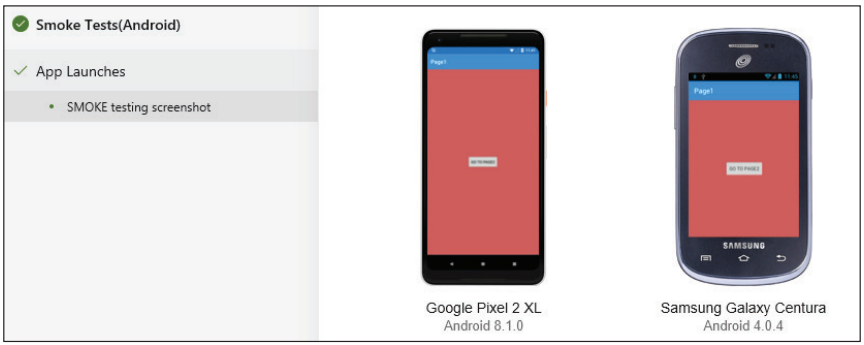


Рис. 5.16 ❖ Просмотр результатов прогона автотеста в облаке

Можно посмотреть более детальную информацию по каждому шагу на каждом тестовом устройстве, включая скриншот и техническую информацию.

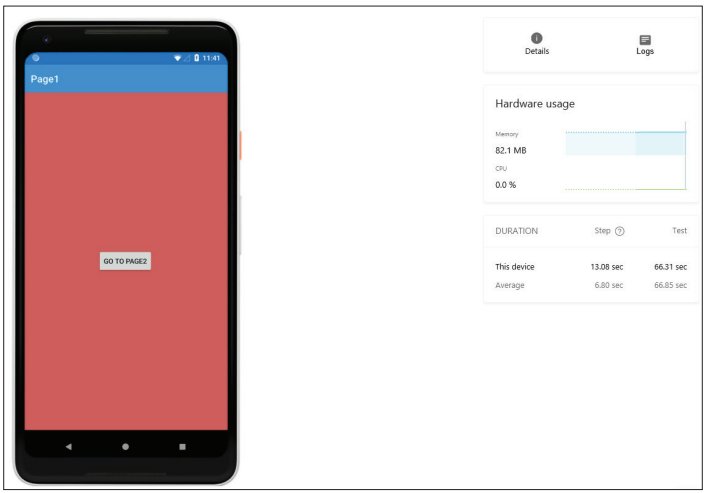


Рис. 5.17 ❖ Просмотр информации об отдельном шаге автотеста

5.5. Дистрибуция

Перед тем как приложение попадет в AppStore, Google Play или Microsoft Store, его необходимо протестировать реальным пользователям. Для того чтобы распространять приложение, необходимо использовать группы дистрибуции App Center. В каждую группу по email добавляются пользователи и в каждую группу можно добавлять релизы, о которых оповестят почтовые уведомления.

Рис. 5.18 ❖ Email с информацией о новой сборке

Если вы используете App Center SDK Distribute, то возможно также отображение алерта внутри самого приложения, установленного у бета-пользователей.

Подробнее о механизмах In-App Update вы можете узнать в документации: **https://docs.microsoft.com/en-us/appcenter/sdk/distribute/android**.

Итак, для реализации сценария с автоматической сборкой нам будет достаточно создать группу дистрибуции Dev, куда будут автоматически публиковаться все успешные билды, созданные и протестированные автоматическим конвейером. Все участники команды будут получать уведомление о наличии новой версии.

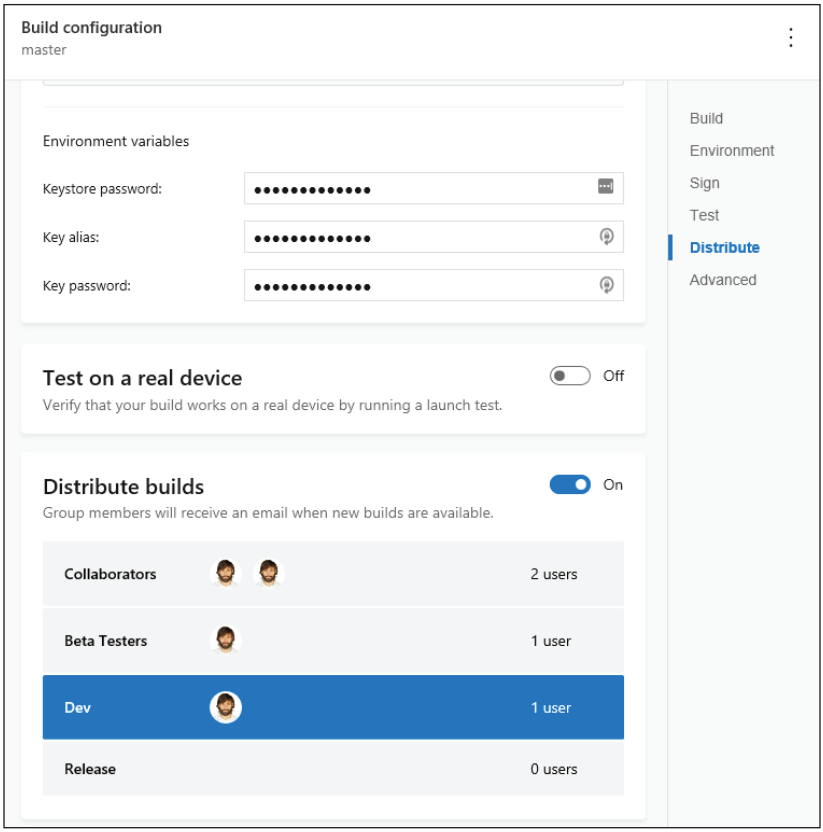


Рис. 5.19 ❖ Выбор группы дистрибуции

Если планируется раздавать публичные URL для загрузки приложения, например, полевыми тестировщиками, то вам будет необходимо также сделать открытую группу дистрибуции.

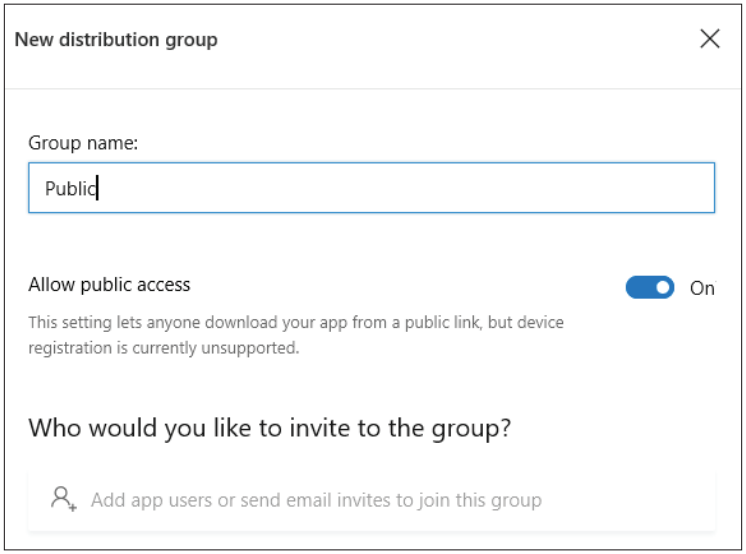


Рис. 5.20 ❖ Создание публичной группы

Также с помощью App Center вы можете упростить процесс публикации приложения в AppStore и Google Play. Прямая публикация в Microsoft Store для Windows UWP пока не поддерживается.

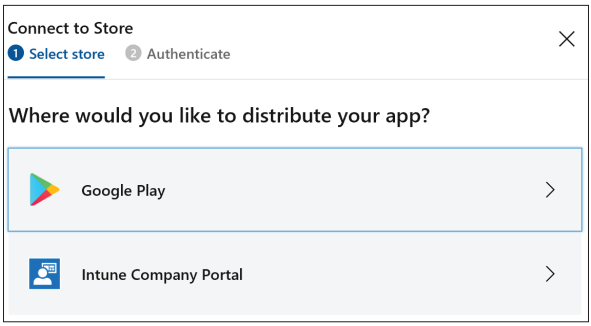


Рис. 5.21 ❖ Отправка финальной сборки в магазин приложений

Подробнее о данной функциональности вы можете почитать в официальной документации: **https://docs.microsoft.com/en-us/appcenter/distribution/stores/**.

5.6. Мониторинг

Так как приложение будет работать у большого количества пользователей на большом количестве реальных и разных устройств, то оперативный сбор обратной связи о проблемах является обязательным. Для этого в ваше приложение необходимо интегрировать App Center SDK: **docs.microsoft.com/en-us/appcenter/sdk**.

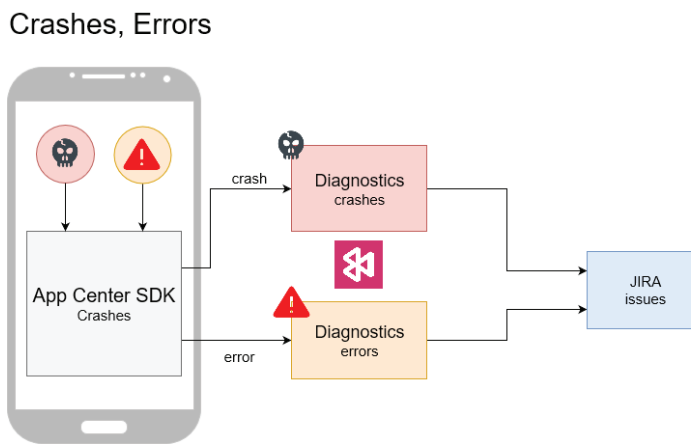


Рис. 5.22 ❖ Сбор ошибок с помощью App Center SDK

Сбор падений (crashes) уже давно является обязательной опцией для всех мобильных приложений. Логи с устройств и детальные отчеты об ошибках позволят лучше понять и устранить причины падения.

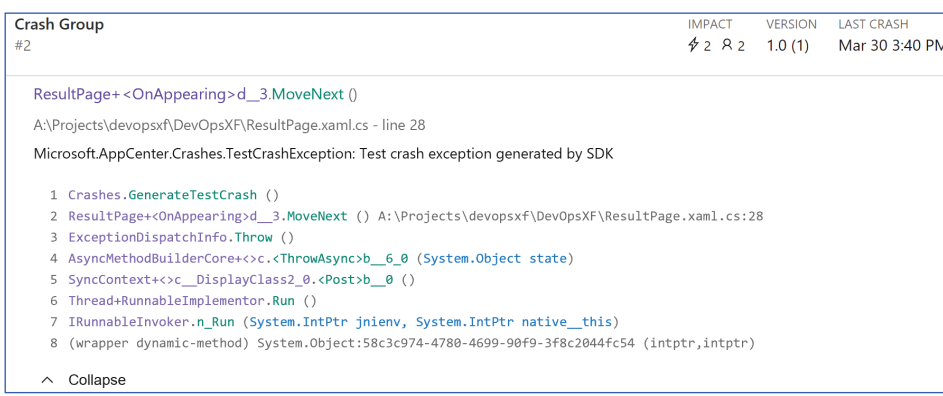


Рис. 5.23 ❖ Информация о креше с указанием вероятного места падения

Для того чтобы информация о крешах как можно быстрее попадала к команде, можно использовать интеграцию с VSTS, Jira или GitHub.

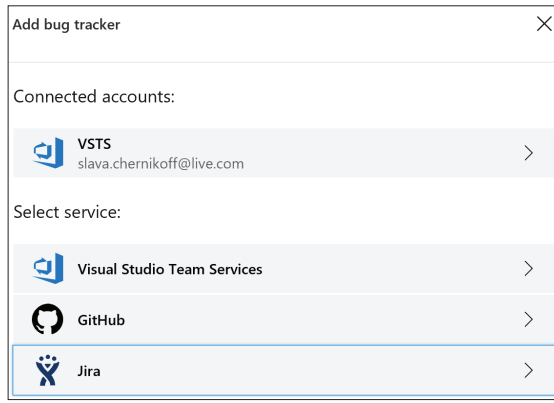


Рис. 5.24 ❖ Интеграция с внешними сервисами

Подробнее: **docs.microsoft.com/en-us/appcenter/crashes**.

Помимо крешей, также критически важным является логирование ошибок в узловых модулях приложения – например, ошибок от сервера.

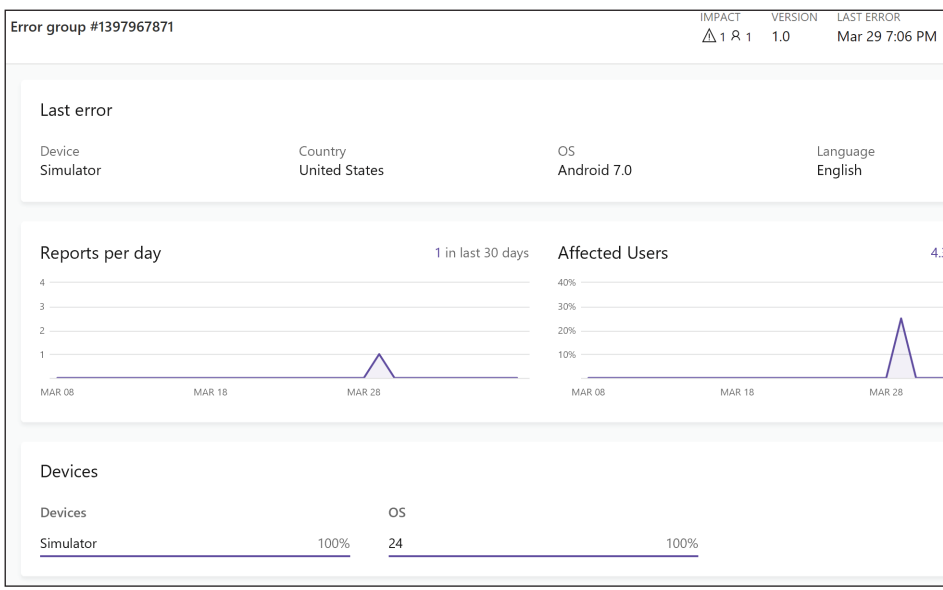


Рис. 5.25 ❖ Просмотр статистики по ошибке

Для этого необходимо внутри ключевых блоков try-catch добавить фиксацию ошибок. Это помогает выявлять проблемы в работе сервера, если записывать в App Center информацию о возвращаемых сервером ошибках. Эти отчеты будут полезны команде Backend-разработки. Подробнее: **docs.microsoft.com/en-us/appcenter/errors**.

Чтобы лучше понимать поведение пользователя (и рассчитать различные показатели), маркетологи часто используют механизм событий. Например, нажал пользователь на кнопку (событие 1) на определенном экране или вернулся обратно (событие 2)?

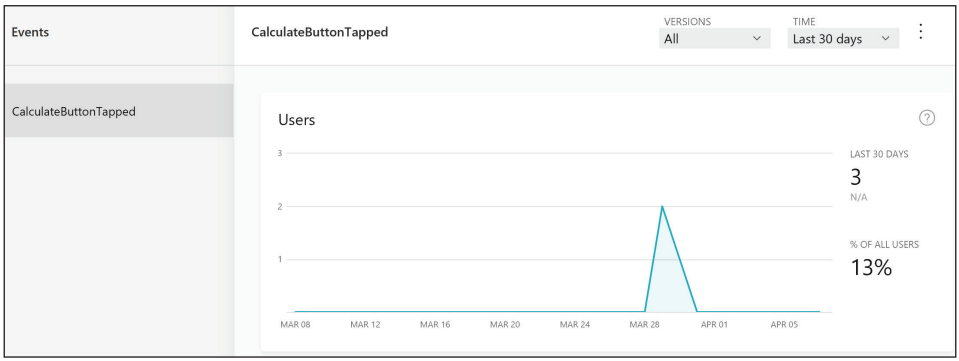


Рис. 5.26 ❖ Сбор событий о действиях пользователей

Также можно использовать events для сбора других событий внутри приложения, например успешная авторизация пользователя или открытие карточки товара с фиксацией productId. События сохраняются в App Center и доступны для выгрузки в Azure BLOB Storage или Azure Application Insights.

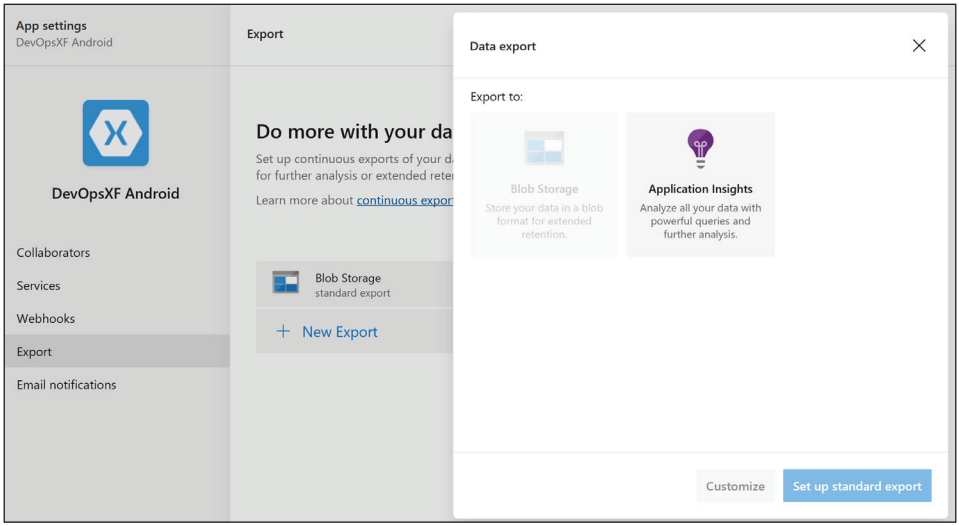


Рис. 5.27 ❖ Экспорт пользовательских событий в Azure Application Insights для анализа данных

Подробнее про аналитику: **docs.microsoft.com/en-us/appcenter/analytics**.

Конец первой части.

Часть II

ПРАКТИЧЕСКИЕ СОВЕТЫ НА КАЖДЫЙ ДЕНЬ

Глава 6

Иконочные шрифты вместо растровых картинок

В современной практике мобильной разработки приходится работать с большим количеством разрешений экранов, поэтому в приложении нужно иметь множество изображений разного размера, чтобы иконки отображались корректно, без артефактов и лишнего использования памяти. Несколько лет назад в мире веб-разработки стали активно использоваться иконочные шрифты Font Awesome, которые позволяли уйти от большого количества изображений и гибко адаптировать верстку под различные разрешения и плотность пикселей. Со временем данный подход перекочевал и в сферу мобильной разработки.

Работает этот механизм так же, как и отображение обычных шрифтов, – операционная система использует файлы шрифтов, содержащие нужные векторные изображения, и сама масштабирует и отрисовывает их на экране. Важно отметить, что в самом шрифтовом файле каждая иконка ассоциируется с той или иной буквой. То есть ваше приложение говорит операционной системе: нарисуй букву «а», используя шрифт X. При этом в самом шрифте X для буквы «а» привязана векторная иконка с нужной графикой.

Сейчас существует большое количество бесплатных и открытых иконочных шрифтов, однако наиболее целесообразным будет создание своих иконочных шрифтов, содержащих нужные наборы изображений для каждой платформы.

Начнем мы с создания иконочного шрифта. Для этого нам понадобятся изображения в формате SVG и бесплатный сервис Glyphter (**https://glyphter.com**), который из коробки позволяет использовать большое количество иконок.

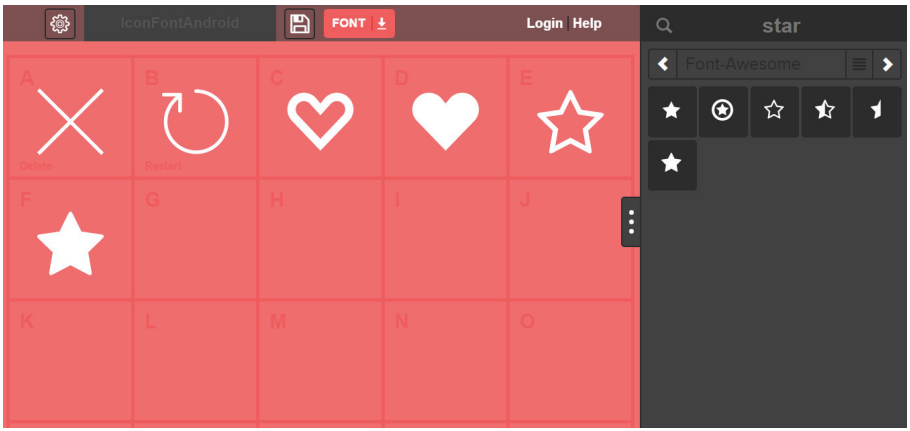


Рис. 6.1 ❖ Пример создания иконочного шрифта с помощью приложения Glyphter

Сами иконки рекомендуется создавать в стилистике целевой платформы или использовать готовые наборы. Далее нам необходимо эти шрифты скачать в формате TTF и поместить в корректные папки:

- Assets\Fonts для Android;
- Resources\Fonts для iOS.

Для iOS также необходимо прописать новые шрифты в файле Info.plist, добавив раздел `UIAppFonts`, и явно указать ссылку на ttf-файл с иконками.

```
<key>UIAppFonts</key>
<array>
    <string>Fonts/icons-ios.ttf</string>
</array>
```

Рис. 6.2 ❖ Добавление иконочного шрифта в Info.plist на iOS

Для того чтобы использовать иконочный шрифт, потребуется создать свой простой наследник `Label`, который мы назовем `IconLabel`.

Листинг 6.1 ❖ Реализация `IconLabel` в кроссплатформенной части проекта

```
public class IconLabel : Label {
    public enum Icons {
        Empty = char.MinValue,
```

```
        Close = 'A',
        Refresh = 'B',
        Hearth = 'C'
    }

    public static readonly BindableProperty IconProperty = BindableProperty.
Create(nameof(Icon), typeof(Icons), typeof(IconLabel), Icons.Empty,
BindingMode.Default, propertyChanged: IconPropertyChanged);

    static void IconPropertyChanged(BindableObject bindable, object oldValue,
object newValue) {
        if (!(bindable is IconLabel iconLabel) || !(newValue is Icons))
            return;
        iconLabel.Text = Convert.ToString((char)(IconLabel.Icons)newValue);
    }

    public Icons Icon {
        get => (Icons)GetValue(IconProperty);
        set => SetValue(IconProperty, value);
    }

    public IconLabel() {
        Icon = Icons.Empty;
        if (Device.RuntimePlatform == Device.iOS) FontFamily = "icons-ios";
        LineBreakMode = LineBreakMode.NoWrap;
    }
}
```

Так как задача `Label` (и нашего потомка `IconLabel`) состоит в том, чтобы показывать текст, то по умолчанию нам бы пришлось каждый раз устанавливать строку «A» в качестве значения поля `IconLabel.Text`, когда необходимо показать иконку крестика. Чтобы упростить работу с иконками, мы будем использовать специальное множество `Icons`, где каждому элементу множества будет соответствовать символ (char), прописанный в иконочном шрифте. Это позволит нам задавать свойство `IconLabel.Icon` с помощью `Icons.Close`.

Для iOS все необходимые механизмы работают из коробки, достаточно просто указать правильный `FontFamily`, ссылающийся на иконочный шрифт. А вот для Android будет необходимо реализовать свой платформенный рендерер:

Листинг 6.2 ❖ Реализация IconLabelRenderer для Android

```
public class IconLabelRenderer: LabelRenderer {
    readonly Typeface _typeface;

    public IconLabelRenderer(Context context) : base(context) {
        _typeface = FontCacher.Instance.GetFont(Context, "icons-android.ttf");
    }

    protected override void OnElementChanged(ElementChangedEventArgs<Label> e)
{
        base.OnElementChanged(e);

        try {
            Control.SetTypeface(_typeface, TypefaceStyle.Normal);
            Control.Gravity = GravityFlags.Center;
            Control.SetTextSize(ComplexUnitType.Dip, (float)Element.FontSize);
        }
        catch (Exception ee) {
            // skip
        }

    }

    protected override void OnElementPropertyChanged(object sender,
PropertyChangedEventArgs e) {
        base.OnElementPropertyChanged(sender, e);
        if (e.PropertyName == Label.FontSizeProperty.PropertyName ||
            e.PropertyName == Label.TextProperty.PropertyName ||
            e.PropertyName == Label.FormattedTextProperty.PropertyName ||
            e.PropertyName == Label.FontAttributesProperty.PropertyName ||
            e.PropertyName == Label.FontFamilyProperty.PropertyName ||
            e.PropertyName == Label.FontProperty.PropertyName ||
            e.PropertyName == Label.FormattedTextProperty.PropertyName ||
            e.PropertyName == Label.HorizontalTextAlignmentProperty.PropertyName
||
            e.PropertyName == Label.VerticalTextAlignmentProperty.PropertyName
||
            e.PropertyName == IconLabel.IconProperty.PropertyName)
            try {
                Control.SetTypeface(_typeface, TypefaceStyle.Normal);
                Control.Gravity = GravityFlags.Center;
                Control.SetTextSize(ComplexUnitType.Dip, (float)Element.
```

```
FontSize);
            }
            catch (Exception ee) {
                // skip
            }
    }
}

public class FontCacher {
    static FontCacher _instance;
    public static FontCacher Instance => _instance ?? (_instance = new
FontCacher());
    FontCacher() { }
    readonly Dictionary<string, Typeface> _cache = new Dictionary<string,
Typeface>();

    public Typeface GetFont(Context context, string name) {
        if (string.IsNullOrEmpty(name))
            return Typeface.Default;
        if (_cache.ContainsKey(name))
            return _cache[name];

        var filename = name.Replace(" ", "");
        var filenameLower = filename.ToLowerInvariant();
        // if no extension given then assume and add .ttf

        if (filename.LastIndexOf(".", StringComparison.Ordinal) != filename.
Length - 4)
            filename = $"{filename}.ttf";

        if (filenameLower.LastIndexOf(".", StringComparison.Ordinal) !=
filenameLower.Length - 4)
            filenameLower = $"{filenameLower}.ttf";

        try {
            var typeface = Typeface.CreateFromAsset(context.Assets, $"Fonts/
{filename}");
            _cache.Add(name, typeface);
            return typeface;
        }
        catch {
            try {
```

```
                var typeface = Typeface.CreateFromAsset(context.Assets, $"Fonts/
{filenameLower}");
                _cache.Add(name, typeface);
                return typeface;
            }
            catch (Exception ee) {
                // skip
            }
        }

        return Typeface.Default;
    }
}
```

Помимо нативного рендерера мы также добавили специальный вспомогательный класс `FontCacher`, который позволит кешировать загруженные шрифты в оперативной памяти.

В конце важно добавить, что шрифты (в том числе и иконочные) могут содержать изображение только одного цвета, поэтому для случаев, когда требуется использовать разноцветные иконки (сразу несколько цветов), описанные механизмы не подойдут.

Глава 7

Работаем с состояниями экранов

В главе 2 мы уже упоминали о необходимости управлять различными состояниями экранов, поэтому ниже рассмотрим вариант реализации данного подхода на практике.

Мобильные приложения, в отличие от веб-сайтов, должны гораздо быстрее взаимодействовать с пользователем, поэтому показывать длительное время пустой экран во время загрузки данных считается не очень правильным. Дополнительно приложение должно уведомлять пользователя об ошибках загрузки данных или отсутствии интернет-соединения. Ленивые разработчики могут обойтись отображением всплывающих уведомлений в духе «Ошибка загрузки данных», но мы пойдем другим путем.

Напомним еще раз основные состояния одного (!) экрана, который загружает данные из интернета:

- загрузка данных (индикатор загрузки по центру экрана);
- отсутствует интернет-соединение (сопроводительный текст, возможно, красивая картинка и кнопка **Повторить**);

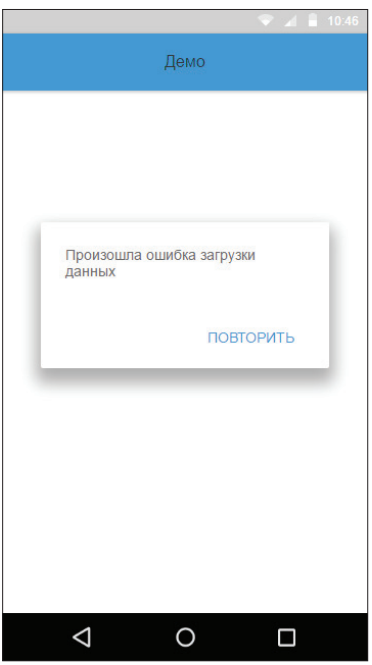


Рис. 7.1 ❖ Как делать не надо – показывать всплывающее уведомление с сообщением об ошибке

- ошибка загрузки данных (сопроводительный текст, возможно, красивая картинка и кнопка **Повторить**);
- нет данных (например, пустая корзина покупок);
- отображение данных (например, загруженный список товаров).

Рис. 7.2 ❖ Пример различных состояний одного экрана

У программиста могут начать шевелиться волосы при мыслях о том, сколько кода надо будет написать, чтобы заменять содержимое одного экрана, при расчете, что таких экранов могут быть десятки, а каждое из состояний может быть достаточно сложным. Рано паниковать – простое и элегантное решение предложил Патрик МакКерли (Patrick McCurley) (**https://github.com/xDelivered-Patrick/Xamarin.Forms.Essentials**). Мы возьмем это решение за основу и немного доработаем.

В основе данного подхода лежит идея описывать все состояния экрана при создании страницы и управлять их сменой с помощью ViewModel. Забегая вперед, отметим, что решение достаточно простое и может быть использовано для управления не только состояниями всего окна, но и отдельных его частей.

В листинге 7.1 показано XAML-описание одной страницы с поддержкой смены состояний.

Листинг 7.1 ❖ Описание различных состояний одного экрана с помощью XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage x:Class="ApiDemo.DemoPage"
             xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:stateContainerDemo="clr-namespace:StateContainerDemo;assembly
=ApiDemo">
    <stateContainerDemo:StateContainer State="{Binding State}">

        <stateContainerDemo:StateCondition State="Loading">
            <ActivityIndicator IsRunning="True" />
        </stateContainerDemo:StateCondition>

        <stateContainerDemo:StateCondition State="Normal">
            <Label Text="Данные загружены и можем их отобразить"/>
        </stateContainerDemo:StateCondition>

        <stateContainerDemo:StateCondition State="Error">
            <StackLayout>
                <Label Text="Ошибка загрузки данных" />
                <Button Command="{Binding LoadDataCommand}" Text="ПОВТОРИТЬ" />
            </StackLayout>
        </stateContainerDemo:StateCondition>

        <stateContainerDemo:StateCondition State="NoInternet">
```

```
            <StackLayout>
                <Label Text="Отсутствует интернет-соединение" />
                <Button Command="{Binding LoadDataCommand}" Text="ПОВТОРИТЬ" />
            </StackLayout>
        </stateContainerDemo:StateCondition>

        <stateContainerDemo:StateCondition State="NoData">
            <Label Text="Нет данных, показываем пользователю приглашение
к действию" />
        </stateContainerDemo:StateCondition>
    </stateContainerDemo:StateContainer>
</ContentPage>
```

Просто и понятно. При этом крупные блоки для состояний можно вынести в виде отдельных View для повторного использования.

В листинге 7.2 показан пример класса-обертки, содержащего описание отдельного состояния экрана.

Листинг 7.2 ❖ Реализация класса StateCondition

```
[ContentProperty("Content")]
public class StateCondition: View
{
    public object State { get; set; }
    public View Content { get; set; }
}
```

Для того чтобы удобно управлять переключением между состояниями, мы будем использовать enum, содержащий все возможные состояния, как это показано в листинге 7.3.

Листинг 7.3 ❖ Перечисление States, содержащее все возможные состояния для экранов в приложении

```
public enum States {
    Loading,
    Normal,
    Error,
    NoInternet,
    NoData
}
```

Мы немного доработали State Container от Патрика МакКерли, добавив простые анимации смены состояния, чтобы все работало плавно. Полный код компонента показан в листинге 7.4.

Листинг 7.4 ❖ Реализация класса StateContainer

```
[ContentProperty("Conditions")]
public class StateContainer : ContentView {
    public List<StateCondition> Conditions { get; set; } = new
List<StateCondition>();

    public static readonly BindableProperty StateProperty = BindableProperty.
Create(nameof(State), typeof(object), typeof(StateContainer), null,
BindingMode.Default, null, StateChanged);

    public static void Init()
    {
        //for linker
    }

    private static async void StateChanged(BindableObject bindable, object
oldValue, object newValue)
    {
        var parent = bindable as StateContainer;
        if (parent != null)
            await parent.ChooseStateProperty(newValue);
    }

    public object State
    {
        get { return GetValue(StateProperty); }
        set { SetValue(StateProperty, value); }
    }

    private async Task ChooseStateProperty(object newValue)
    {
        if (Conditions == null && Conditions?.Count == 0) return;

        try
        {
            foreach (var stateCondition in Conditions.Where(stateCondition
=> stateCondition.State != null && stateCondition.State.ToString().
Equals(newValue.ToString()))) {
                if (Content != null)
                {
                    await Content.FadeTo(0, 100U); // Быстрая анимация скрытия
                    Content.IsVisible = false;     // Полностью скрываем
                                                      с экрана старое состояние
```

```
                    await Task.Delay(30); // Позволяем UI-потоку отработать свою
                                          очередь сообщений и гарантировано
                                          скрыть предыдущее состояние
                }

                // Плавно показываем новое состояние
                stateCondition.Content.Opacity = 0;
                Content = stateCondition.Content;
                Content.IsVisible = true;
                await Content.FadeTo(1);

                break;
            }
        } catch (Exception e)
        {
            Debug.WriteLine($"StateContainer ChooseStateProperty {newValue}
error: {e}");
        }
    }
}
```

Для получения статуса интернет-соединения мы будем использовать ранее описанное свойство `IsConnected` из BaseViewModel.

Листинг 7.5 ❖ Пример проверки наличия интернет-соединения с помощью плагина ConnectivityPlugin внутри ViewModel

```
if (!IsConnected){
    State = States.NoInternet; // Меняем свойство у ViewModel
    return;
}
```

Как видим, `StateContainer` – это обычный `ContentView`, который может вполне спокойно размещаться на экране со статическим или уже загруженным контентом. Это позволит реализовать механизмы частичной дозагрузки данных, например когда у вас уже есть название и ссылка на фотографию, которые можно отображать пользователю без необходимости ожидания, как это показано на рис. 7.3.

Рис. 7.3 ❖ Пример использования `StateContainer` для сценария частичной дозагрузки данных

В оригинальную реализацию от Patrick мы добавили простые анимации смены состояний, что добавляет плавности и придает приложению законченный вид. Как уже отмечалось в главе 2, рекомендуется использовать StateContainer для всех экранов, загружающих данные из интернета или локальной базы данных, что сделает поведение приложения более понятным для конечных пользователей.

Глава 8

Дополнительные анимации при переходе экрана из одного состояния в другое

Описанный ранее `StateContainer` хорошо работает, когда у нас все состояния существуют независимо друг от друга и между ними достаточно простого перехода «один исчез – второй появился».

Но что делать, если необходимо реализовать комплексный и анимированный переход из одного состояния в другое? Чтобы выезжало, вращалось и прыгало.

В качестве примера давайте рассмотрим экран ввода адреса и работы с картой, как это реализуется в большинстве навигаторов.

Представим, что у нас анимированные переходы между следующими состояниями ОДНОГО экрана.

Если вы были студентом технической специальности, то наверняка помните курс, посвященный конечным автоматам. Эта простая, но очень емкая модель (конечный автомат, он же finite state machine, он же FSM) поможет нам в решении поставленной задачи.

Чаще всего требуется анимировать следующие свойства у элементов пользовательского интерфейса:

- **Scale** – масштаб элемента;
- **Opacity** – прозрачность;
- **Translation** – дополнительное смещение по x, y относительно полученного при компоновке положения;
- **Rotation** – вращение вокруг осей x, y, z.

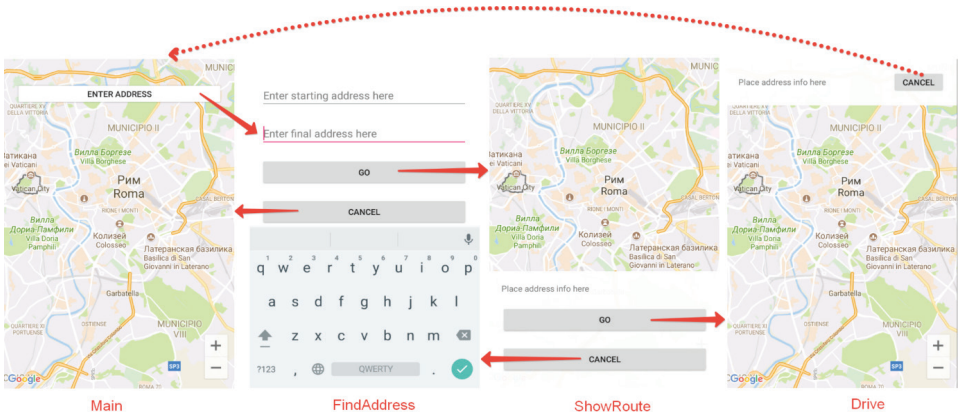


Рис. 8.1 ❖ Пример комплексных анимаций внутри одного окна

В Xamarin.Forms для задания обозначенных свойств используются механизмы ОС низкого уровня, что отлично сказывается на производительности – нет проблем анимировать сразу целую кучу объектов. В нашем примере мы остановимся именно на этих свойствах, но при желании вы сможете самостоятельно расширить описанные ниже механизмы.

Если описать конечный автомат человеческим языком, то это некий объект, который может находиться в различных устойчивых состояниях (например, **Загрузка** или **Ошибка**). Свои состояния автомат меняет под воздействием внешних событий. Количество состояний конечно. Для сценария, описанного выше, у нас получается такой конечный автомат:

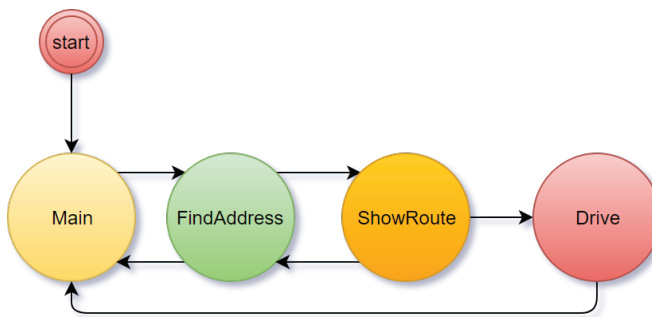


Рис. 8.2 ❖ Конечный автомат,
соответствующий описанному выше набору анимаций

Предположим, что необходимо реализовать следующие анимации при переходе из состояния в состояние:

- при входе в FindAddress нужно скрыть с анимацией старый контент и плавно показать новый. Плюс для пикантности будем анимировать кнопки во время появления;
- при переходе в ShowRoute необходимо скрыть старое состояние, а снизу экрана должна выехать табличка с информацией о маршруте;
- при переходе в Drive необходимо скрыть старое состояние, и сверху должна выехать табличка с информацией о маршруте;
- при переходе в Main (кроме первого запуска) необходимо скрыть текущее состояние и плавно отобразить кнопку, добавим к ней также небольшую анимацию изменения масштаба.

Для реализации конечного автомата мы возьмем самую простую реализацию:

- у автомата есть фиксированный набор состояний, которые задаются при инициализации;
- каждое из состояний описывается набором необходимых анимаций (конечные значения properties) для элементов UI;
- при входе в новое состояние параллельно запускаются все анимации из массива, добавленного при инициализации автомата.

Никакую историю переходов хранить не будем, также неважно, по какому пользовательскому событию автомат перешел из одного состояния в другое. Есть только переход в новое состояние, который сопровождается анимациями.

Итак, простейший автомат, который мы назовем Storyboard, показан в листинге 8.1. В коде ниже опущены проверки входных данных на null, что в реальных проектах может быть обязательным.

Листинг 8.1 ❖ Реализация класса Storyboard

```
public enum AnimationType {
    Scale,
    Opacity,
    TranslationX,
    TranslationY,
    Rotation
}

public class Storyboard {
```

```
    readonly Dictionary<string, ViewTransition[]> _stateTransitions = new
Dictionary<string, ViewTransition[]>();

    public void Add(object state, ViewTransition[] viewTransitions) {
        var stateStr = state?.ToString().ToUpperInvariant();
        _stateTransitions.Add(stateStr, viewTransitions);
    }

    public void Go(object newState, bool withAnimation = true) {
        var newStateStr = newState?.ToString().ToUpperInvariant();

        // Get all ViewTransitions
        var viewTransitions = _stateTransitions[newStateStr];

        // Get transition tasks
        var tasks = viewTransitions.Select(viewTransition => viewTransition.
GetTransition(withAnimation));

        // Run all transition tasks
        Task.WhenAll(tasks);
    }
}

public class ViewTransition {
        // Skipped. Смотри полный пример в репозитории

    public async Task GetTransition(bool withAnimation) {
        VisualElement targetElement;
        if( !_targetElementReference.TryGetTarget(out targetElement) )
           throw new ObjectDisposedException("Target VisualElement was
disposed");

        if( _delay > 0 ) await Task.Delay(_delay);

        withAnimation &= _length > 0;

        switch ( _animationType ) {
            case AnimationType.Scale:
                if( withAnimation )
                   await targetElement.ScaleTo(_endValue, _length, _easing);
                else
                   targetElement.Scale = _endValue;
```

```
                break;

                // See complete sample in repository below
            default:
                throw new ArgumentOutOfRangeException();
        }
    }
}
```

Как видим, при переходе в новое состояние параллельно происходят плавные изменения необходимых свойств. Есть также возможность перейти в новое состояние без анимации.

Итак, автомат у нас есть, и мы можем подключить его для задания необходимых состояний элементов, как это показано в листинге 8.2.

Листинг 8.2 ❖ Пример добавления одного состояния с указанием необходимых анимаций

```
_storyboard.Add(States.Drive, new[] {
   new ViewTransition(ShowRouteView, AnimationType.TranslationY, 200),
   new ViewTransition(ShowRouteView, AnimationType.Opacity, 0, 0, delay: 250),
   new ViewTransition(DriveView, AnimationType.TranslationY, 0, 300, delay: 250),
                                                        // Active and visible
   new ViewTransition(DriveView, AnimationType.Opacity, 1, 0)
                                                        // Active and visible
});
```

Как видим, для состояния `Drive` мы задали массив индивидуальных анимаций. `ShowRouteView` и `DriveView` – обычные `View`, заданные в XAML, пример ниже.

А вот для перехода в новое состояние достаточно просто вызвать метод `Go()`:

```
_storyboard.Go(States.ShowRoute);
```

Кода получается относительно немного, и групповые анимации создаются по факту просто набором чисел. Работать наш конечный автомат может не только со страницами, но и с отдельными `View`, что расширяет варианты его применения.

На рис. 8.3 показан пример XAML, в котором описаны все элементы пользовательского интерфейса.

```
<ContentView AbsoluteLayout.LayoutBounds="0, 20, 1, 80" AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" Padding="20" x:Name="MainView"
    <Button Text="Enter address" BackgroundColor="White" BorderColor="Black" BorderRadius="6" BorderWidth="2" Clicked="GotoFindAddressClicked"
            HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" x:Name="EnterAddressButton" Scale="0.9" />
</ContentView>
<ScrollView  AbsoluteLayout.LayoutBounds="0,0,1,1" AbsoluteLayout.LayoutFlags="All" BackgroundColor="#FFFFFF" x:Name="FindAddressView" >
    <StackLayout HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" Spacing="20" Padding="20">
        <Entry Placeholder="Enter starting address here" Margin="0,20,0,0"/>
        <Entry Placeholder="Enter final address here" x:Name="FinalAddressEntry"/>
        <Button Text="Go" Clicked="GotoShowRouteClicked" x:Name="ShowRouteGoButton" BorderWidth="1"/>
        <Button Text="Cancel" Clicked="GotoMainClicked" x:Name="ShowRouteCancelButton" BorderWidth="1"/>
    </StackLayout>
</ScrollView>
<StackLayout BackgroundColor="#E0EEEEEE" AbsoluteLayout.LayoutBounds="0,1,1,200" AbsoluteLayout.LayoutFlags="YProportional, WidthProportional"
             x:Name="ShowRouteView" Padding="20" Spacing="20" IsClippedToBounds="False">
    <Label Text="Place address info here" HorizontalOptions="FillAndExpand" VerticalOptions="Center" VerticalTextAlignment="Center"/>
    <Button Text="Go" Clicked="GotoDriveClicked" BorderWidth="1"/>
    <Button Text="Cancel" Clicked="GotoFindAddressClicked" BorderWidth="1"/>
</StackLayout>
<StackLayout BackgroundColor="#E0FFFFFF" AbsoluteLayout.LayoutBounds="0,0,1,80" AbsoluteLayout.LayoutFlags="XProportional,WidthProportional"
             x:Name="DriveView" Orientation="Horizontal" Padding="20">
    <Label Text="Place address info here" HorizontalOptions="FillAndExpand" VerticalOptions="Center" VerticalTextAlignment="Center"/>
    <Button Text="Cancel" Clicked="GotoMainClicked" HorizontalOptions="End" BorderWidth="1"/>
</StackLayout>
```

Рис. 8.3 ❖ Указание имен переменных для объектов, чьи свойства будут изменяться с помощью Storyboard

Полный код проекта из статьи вы можете найти в нашем репозитории: **https://github.com/binwell-university/XamarinBookSamples**.

Глава 9

Использование FastGrid для создания сложного интерфейса

При разработке мобильных приложений для бизнеса часто возникают задачи по созданию экранов с большим количеством разнородных данных. Хорошим примером являются онлайн-магазины (OZON, Aliexpress, Инстамарт) – на главной странице приложения необходимо показывать не только большое число карточек с товарами, но и дополнительные блоки – баннеры, избранные товары, категории товаров, список брендов и пр. С точки зрения реализации задача это довольно нетривиальная, так как некоторые блоки могут отображаться или скрываться для различных пользователей, а самих товаров на странице может быть несколько сотен, а то и больше. На рис. 9.1 показаны скриншоты приложения Инстамарт, имеющего сложную компоновку интерфейса главной страницы.

Если попробовать решить задачу на базе Xamarin.Forms «в лоб», т. е. с использованием стандартных компонентов Grid и StackLayout, то разработчики столкнутся со следующими проблемами:

- долгое время создания экрана со сложной компоновкой – все возможные элементы сразу добавляются на экран, а их там сотни;
- высокое потребление памяти, так как все элементы создаются при открытии экрана.

При использовании стандартного компонента ListView возможно временно победить обозначенные проблемы, однако во весь рост

встанет вопрос с производительностью при скролле – сложные ячейки в ListView долго создаются и обновляются.

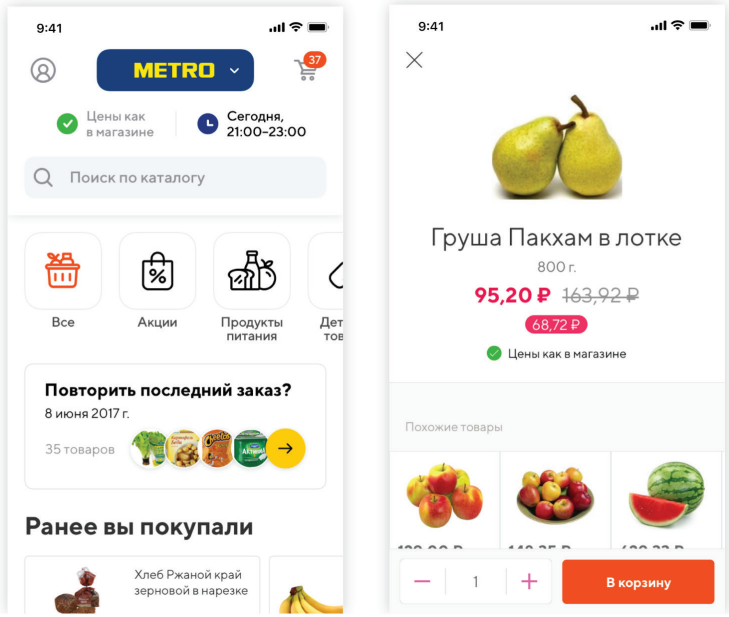


Рис. 9.1 ❖ Скриншоты приложения Инстамарт

Операционные системы iOS и Android предоставляют компоненты `UICollectionView` и `RecyclerView`, которые позволяют реализовывать сложную компоновку и повторно использовать ячейки. Это своего рода аналоги ListView, только с возможностью управления процессом компоновки ячеек на экране.

Для Xamarin.Forms существует несколько готовых компонентов, основанных на `UICollectionView`/`RecyclerView`, однако это либо сырой open source, либо закрытые и ограченные commercial-контролы, которые разработчики продают за деньги. В нашей книге мы остановимся на подходе, предложенном компанией Twin Technologies (**https://twin-techs.com**) и доработанном специалистами компании Binwell. Созданный компонент получил название `FastGrid` и представляет собой готовый контрол для Xamarin.Forms, который можно добавить в проект как в виде отдельной библиотеки, так и в виде исходных кодов, что позволит дорабатывать его под нужды вашего проекта.

Итак, давайте еще раз напомним задачу – создать экран с большим количеством разнородных элементов (и отображаемых данных), каждый блок может иметь свои размеры и поддерживать не только вертикальную, но и горизонтальную прокрутку. Как дополнительная опция – элементы должны отображаться пользователю только в случае их получения от сервера.

Для достижения максимально возможной производительности при прокрутке в компоненте `FastGrid` используется прямое указание размеров каждой ячейки – это убирает необходимость в динамическом вычислении ширины и высоты, что положительно сказывается на производительности. Для связки «ячейка–тип данных» реализован механизм на базе `DataTemplateSelector`, который возвращает нужную ячейку для указанного типа данных. Схема работы `FastGrid` в связке с `DataTemplateSelector` показана на рис. 9.2.

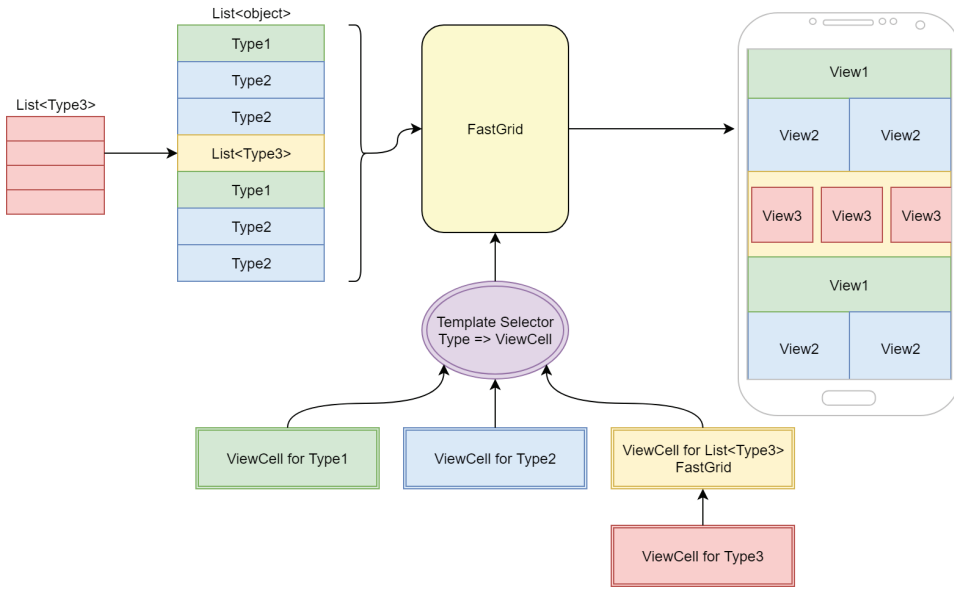


Рис. 9.2 ❖ Механизм работы FastGrid

Краткий список возможностей `FastGrid`:

- использование неограниченного количества ячеек с возможностью динамической дозагрузки данных;

- каждый вид ячеек внутри `FastGrid` сопоставляется с определенным типом данных, приходящих от серверного API;
- использование Flow Layout, который отвечает за размещение и выравнивание ячеек внутри `FastGrid`;
- возможность использования `FastGrid` внутри ячеек для реализации блоков с горизонтальной прокруткой;
- поддержка динамического добавления/удаления и обновления данных;
- функциональность Pull-to-refresh для обновления содержимого `FastGrid` с помощью привычного для пользователей механизма.

В качестве примера работы `FastGrid` мы предлагаем рассмотреть реализацию списка товаров, сгруппированного по категориям и содержащего по две ячейки в каждой строке.

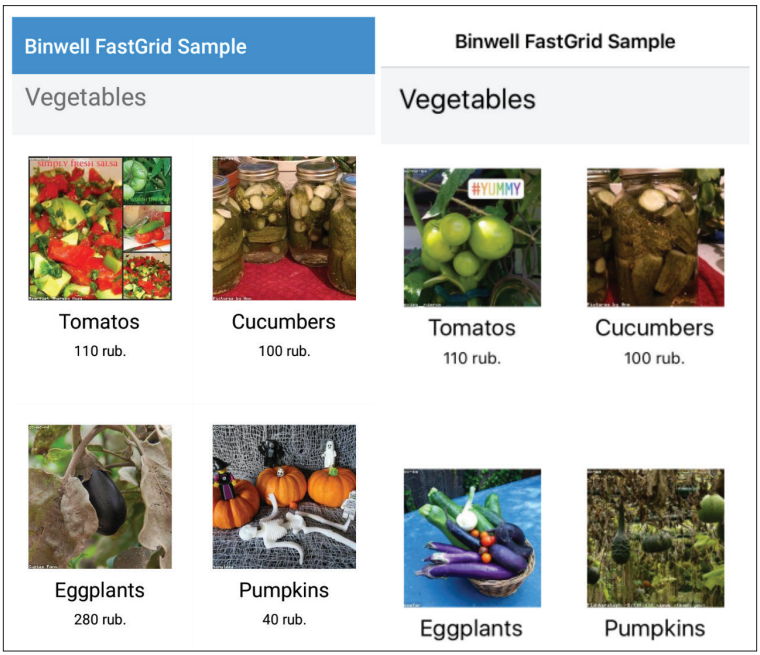


Рис. 9.3 ❖ Пример использования `FastGrid`

Для начала нам потребуется создать свою ячейку на базе `FastGridCell` с помощью C#, однако допустимо использование XAML для описания ячеек.

Листинг 9.1 ❖ Пример создания ячейки для FastGrid

```
public class ProductCell : FastGridCell {
    CachedImage _image;
    Label _name;
    Label _price;

    protected override void InitializeCell()
    {
        var screenWidth = Device.Info.ScaledScreenSize.Width;

        _image = new CachedImage
        {
            HorizontalOptions = LayoutOptions.Center,
            Aspect = Aspect.AspectFill,
            WidthRequest = screenWidth / 2 - 40,
            HeightRequest = screenWidth / 2 - 40
        };

        _name = new Label
        {
            HorizontalOptions = LayoutOptions.Center,
            FontSize = 20,
            TextColor = Color.Black
        };

        _price = new Label { HorizontalOptions = LayoutOptions.Center,
            FontSize = 14,
            TextColor = Color.Black
        };

        View = new StackLayout
        {
            BackgroundColor = Color.White,
            Padding = 20,
            VerticalOptions = LayoutOptions.FillAndExpand,
            HorizontalOptions = LayoutOptions.FillAndExpand,
            Children =
            {
                _image,
                _name,
                _price
```

```
            }
        };
    }

    protected override void SetupCell(bool isRecycled)
    {
        if (!(BindingContext is ProductObject bindingContext)) return;

        _image.Source = null;
        _image.Source = bindingContext.ImageUrl;
        _name.Text = bindingContext.Name;
        _price.Text = bindingContext.Price;
    }
}
```

Для реализации нового функционала необходимо переопределить два виртуальных метода `FastGridCell`:

- `InitializeCell()` – вызывается при первом создании ячейки (создается один раз, а дальше используется повторно). Если вы описываете ячейки в XAML, то в данном методе необходимо вызвать `InitializeComponent()`, который отвечает за создание компонента на базе XAML;
- `SetupCell()` – вызывается при повторном использовании ранее созданной ячейки. По факту при повторном использовании в качестве BindingContext указываются новые данные, и эти данные можно вручную присвоить компонентам пользовательского интерфейса.

Далее создается `FastGridTemplateSelector`, где задаются связки «тип ячейки–тип данных» и указываются необходимые размеры ячеек. Решение немного топорное, но другие реализации требуют пересчета размеров ячеек, что добавляет мороки на нативом уровне.

Листинг 9.2 ❖ Создание `FastGridTemplateSelector`

```
var size = Device.Info.ScaledScreenSize;
fastGridView.ItemTemplateSelector = new FastGridTemplateSelector(
   new FastGridDataTemplate(typeof(CategoryObject).Name,
                typeof(CategoryCell),new Size(size.Width, 70)),
   new FastGridDataTemplate(typeof(ProductObject).Name,
                typeof(ProductCell),new Size(size.Width / 2, 260))
);
```

Важно добавить, что компонента `RecyclerView` в Android использует целые числа (integer) в качестве идентификаторов типов ячеек, поэтому необходимо дополнительно вызывать метод `FastGridTemplateSelector.Prepare()`, который сопоставит ваши типы данных со значениями `integer`.

Исходные коды компонента Binwell `FastGrid` и предложенного примера вы можете изучить в репозитории **https://github.com/binwell-university/XamarinBookSamples**.

Глава 10

Работа с сетевыми сервисами Json/REST

Самым популярным в настоящее время протоколом для общения мобильных приложений с сервером является REST в связке с Json. Поэтому далее мы познакомимся с библиотекой Refit, которая заметно упрощает подключение внешних API.

Refit позволяет описать спецификации для работы с REST-сервисом в виде простого Interface с понятным набором входных и выходных параметров, включая возможность манипулировать HTTP-заголовками для отдельных запросов. Для примера возьмем демо-API сервиса httpbin.org, описанного с помощью расширения Refit и показанного в листинге 10.1.

Листинг 10.1 ❖ Описание внешнего REST API с помощью Refit

```
[Headers("Accept: application/json")]
public interface IHttpbinApi
{
    [Get("/basic-auth/{username}/{password}")]
    Task<AuthResult> BasicAuth(string username, string password,
[Header("Authorization")] string authToken, CancellationToken ctx);

    [Get("/cache")]
    Task<HttpResponseMessage> CheckIfModified([Header("If-Modified-Since")]
string lastUpdateAtString, CancellationToken ctx);

    [Post("/post")]
    Task<HttpResponseMessage> FormPost([Body(BodySerializationMethod.
UrlEncoded)] FormData data, CancellationToken ctx);
}
```

После описания данного интерфейса он подается на вход для Refit (листинг 10.2).

Листинг 10.2 ❖ Использование описания API при создании http-клиента

```
var client = new HttpClient(new HttpClientHandler()) {
                BaseAddress = new Uri("http://httpbin.org")
            };
_httpbinApiService = RestService.For<IHttpbinApi>(client);
```

Сами данные при необходимости (конвертации `camelCase` или `snake_eyes`, преобразование из enum в строковые значения) можно расширить атрибутами из библиотеки Json.net, так как именно она используется в Refit (листинг 10.3).

Листинг 10.3 ❖ Дополнительное описание свойств для модели данных, передающейся по сети

```
public class AuthResult
{
    [JsonProperty("authenticated")]
    public bool IsAuthenticated { get; set; }

    [JsonProperty("user")]
    public string Login { get; set; }
}
```

В Refit в качестве выходного значения можно получить уже преобразованные объекты DTO (Data Transfer Object) или `HttpResponseMessage`. Последний позволяет получить информацию о запросе и ответе, что может быть полезно при отладке.

При разработке мобильных приложений часто приходится учитывать фактор нестабильности сигнала в сотовых сетях, поэтому при выполнении сетевых запросов часто возникает необходимость делать повторные попытки. Это позволяет не отвлекать лишний раз пользователя просьбой повторить запрос. В листинге 10.4 показана реализация класса доступа к внешнему REST API с учетом нестабильности интернет-соединения. Напомним, что `RequestResult` описывался в разделе 4.1.

Листинг 10.4 ❖ Реализация базового класса для сервисов доступа к внешним REST API

```
public class BaseOnlineDataService  {
    const int RetryCount = 3;
```

```
    const int RetryDelayMilliseconds = 300;

    protected BaseOnlineDataService() {
    }

    protected async Task<RequestResult> MakeRequest(Task loadingTask,
CancellationToken cancellationToken) {
        try {
            await InvokeWithRetry(loadingTask, cancellationToken).
ConfigureAwait(false);
        }
        catch (Exception e) {
            // Log exception
            return new RequestResult(StatusFromException(e), e.Message);
        }

        return new RequestResult(RequestStatus.Ok);
    }

    protected async Task<RequestResult<T>> MakeRequest<T>(Task<T> loadingTask,
CancellationToken cancellationToken) where T : class {
        T result;
        try {
            result = await InvokeWithRetry(loadingTask, cancellationToken).
ConfigureAwait(false);
        }
        catch (Exception e) {
            // Log exception
            return new RequestResult<T>(default(T), StatusFromException(e),
e.Message);
        }
        return new RequestResult<T>(result, RequestStatus.Ok);
    }

    async Task InvokeWithRetry(Task loadingTask, CancellationToken
cancellationToken) {
        Exception exception;
        var retryRemained = RetryCount;
        do {
            try {
                await loadingTask.ConfigureAwait(false);
                exception = null;
            }
```

```
            catch (TaskCanceledException ce) {
                exception = ce;
                break;
            }
            catch (Exception e) {
                exception = e;
                await Task.Delay(RetryDelayMilliseconds, cancellationToken).
ConfigureAwait(false);
            }
            retryRemained--;
        } while (exception != null && retryRemained > 0);

        if (exception != null)
            throw exception;
    }

    async Task<T> InvokeWithRetry<T>(Task<T> loadingTask, CancellationToken
cancellationToken) {
        Exception exception;
        var result = default(T);
        var retryRemained = RetryCount;
        do {
            try {
                result = await loadingTask.ConfigureAwait(false);
                exception = null;
            }
            catch (TaskCanceledException ce) {
                exception = ce;
                break;
            }
            catch (Exception e) {
                exception = e;
                await Task.Delay(RetryDelayMilliseconds, cancellationToken).
ConfigureAwait(false);
            }
            retryRemained--;
        } while (exception != null && retryRemained > 0);

        if (exception != null)
            throw exception;

        return result;
    }

    static RequestStatus StatusFromCode(int code) {
```

```
        if (!Enum.TryParse(code.ToString(), out RequestStatus status))
            status = RequestStatus.Unknown;
        return status;
    }

    static RequestStatus StatusFromException(Exception exception) {
        if (exception == null)
            return RequestStatus.Ok;

        if (exception is TaskCanceledException || exception is
OperationCanceledException)
            return RequestStatus.Canceled;

        if (exception is JsonException)
            return RequestStatus.SerializationError;

        return exception is ApiException apiException
            ? StatusFromCode((int)apiException.StatusCode)
            : GetWebExceptionStatus(exception);
    }

    static RequestStatus GetWebExceptionStatus(Exception exception) {
        if (!(exception is WebException) && !(exception is
HttpRequestException))
            return RequestStatus.Unknown;

        var webException = exception as WebException ?? ((HttpRequestException)
exception).InnerException as WebException;

        return webException?.Response is HttpWebResponse response
            ? StatusFromCode((int)response.StatusCode)
            : RequestStatus.Unknown;
    }
}
```

Вместо `loadingFunction` необходимо передать ваш код обращения к Refit (листинг 10.5).

Листинг 10.5 ❖ Обращение к серверу с помощью метода-обертки MakeRequest и HTTP Basic Authentication

```
var authToken = "Basic " + Convert.ToBase64String(Encoding.UTF8.GetBytes(
$"{username}:{password}"));
return await MakeRequest(ct => _httpbinApiService.BasicAuth(username, password,
authToken, ct), cancellationToken);
```

И в завершении главы рассмотрим использование кеша при работе с сетью. Xamarin-разработчику доступны все возможности целевых платформ, поэтому для реализации кеша применяют различные СУБД. Для этих целей можно использовать мобильную СУБД Realm. Ниже представлен пример кеширования на базе Realm. В качестве Value можно хранить данные для сериализации (например, полученные данные из ответа сервера на REST-запрос вашего приложения), в качестве Key – либо URL, либо его хеш. Поле `UpdatedAt` позволяет проверить, не устарели ли данные, – если нет, то их можно использовать вместо реального обращения к серверу.

Листинг 10.6 ❖ Реализация локального кешера на базе Realm Xamarin

```
public static class LocalCache {
    private class CachedObject : RealmObject
    {
        [PrimaryKey]
        public string Key { get; set; }
        public string Value { get; set; }
        public DateTimeOffset UpdatedAt { get; set; }
    }

    private static readonly RealmConfiguration Configuration = new
RealmConfiguration("cache.realm", true);
    private static Realm Db => Realm.GetInstance(Configuration);

    public static async Task WriteToCache<T>(string key, T data, DateTimeOffset
timeStamp)
    {
        if (String.IsNullOrEmpty(key) || data == null || timeStamp ==
DateTimeOffset.MinValue) return;

        var currentValue = Db.All<CachedObject>().Where(o => o.Key == key).
ToList().FirstOrDefault();
        if (currentValue == null)
            await Db.WriteAsync(db =>
            {
                var newValue = db.CreateObject<CachedObject>();
                newValue.Key = key;
                newValue.UpdatedAt = timeStamp;
                newValue.Value = JsonConvert.SerializeObject(data);
            });
```

```
        else
            using (var transaction = Db.BeginWrite())
            {
                currentValue.Value = JsonConvert.SerializeObject(data);
                currentValue.UpdatedAt = timeStamp;
                transaction.Commit();
            }
    }
    public static DateTimeOffset CacheLastUpdated(string key)
    {
        if (String.IsNullOrEmpty(key)) return DateTimeOffset.MinValue;

        var currentValue = Db.All<CachedObject>().Where(o => o.Key == key).
ToList().FirstOrDefault();
        return currentValue?.UpdatedAt ?? DateTimeOffset.MinValue;
    }
    public static void RemoveCache(string key)
    {
        if (String.IsNullOrEmpty(key)) return;

        var currentValue = Db.All<CachedObject>().Where(o => o.Key == key).
ToList().FirstOrDefault();
        if (currentValue == null) return;

        using (var transaction = Db.BeginWrite())
        {
            Db.Remove(currentValue);
            transaction.Commit();
        }
    }
    public static T GetFromCache<T>(string key)
    {
        if (String.IsNullOrEmpty(key)) return default(T);

        var currentValue = Db.All<CachedObject>().Where(o => o.Key == key).
ToList().FirstOrDefault();
        return currentValue?.Value == null ? default(T) : JsonConvert.Deseriali
zeObject<T>(currentValue.Value);
    }

    public static void ClearCache()
    {
```

```
        Realm.DeleteRealm(Configuration);
    }
}
```

В данной главе мы рассмотрели удобные механизмы работы с сетью в ваших мобильных приложениях. В следующей главе речь пойдет о механизмах авторизации пользователей с помощью внешних популярных сервисов.

Глава 11

Авторизация с помощью Facebook, ВКонтакте и OAuth

11.1. Facebook

Социальные сети, и особенно Facebook, уже давно используются в мобильных приложениях. Сегодня мы рассмотрим, как подключить нативные Facebook SDK к проекту на базе Xamarin.Forms (iOS и Android) для удобной авторизации пользователей и получения о них базовой информации. Вы также легко сможете расширить описанные в статье методы, для того чтобы реализовать полноценное взаимодействие с этим замечательным сервисом. Тема простая и понятная, поэтому без теорий и прелюдий перейдем сразу к практике.

Для тех, кто впервые создает свое приложение в Facebook, мы кратко расскажем о том, как это делается.

Сам по себе процесс это довольно простой и потребует от вас следующих данных:

- **Package Name** для Android-проекта (например, `com.binwell.login`);
- **Bundle Identifier** для iOS-проекта (например, `com.binwell.login`).

Для Android еще потребуются Key Hashes, которые можно получить командой:

Windows:

```
keytool -exportcert -alias androiddebugkey -storepass android -keystore C:\
Users\[USERNAME]\AppData\Local\Xamarin\Mono for Android\debug.keystore |
openssl sha1 -binary | openssl base64
```

macOS:

```
keytool -exportcert -alias androiddebugkey -storepass android -keystore
/Users/[USERNAME]/.local/share/Xamarin/Mono for Android/debug.keystore |
openssl sha1 -binary | openssl base64
```

Вместо `[USERNAME]` необходимо подставить ваше имя пользователя в системе. Плюс можно прописать путь до `openssl`, если путь до него не указан в `PATH`. Скачать `openssl` для Windows можно здесь: **http://gnuwin32.sourceforge.net/packages/openssl.htm**.

На выходе мы и получим нужные `Key Hashes` следующего вида: `kGP2WMxohvxm/NiwR7H+Eb3/8qw=`.

Теперь заходим на **developers.facebook.com** и создаем новое приложение – отдельно для iOS и Android. При создании приложения мы можем использовать режим с подсказками (Quick Start), где дополнительно описано, как настроить проект. Из этого руководства нам и потребуются примеры кода.

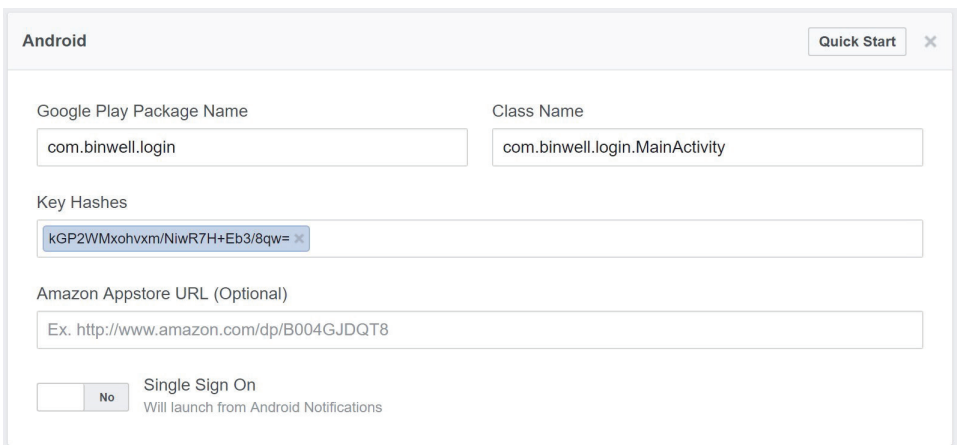


Рис. 11.1 ❖ Регистрация приложения Android

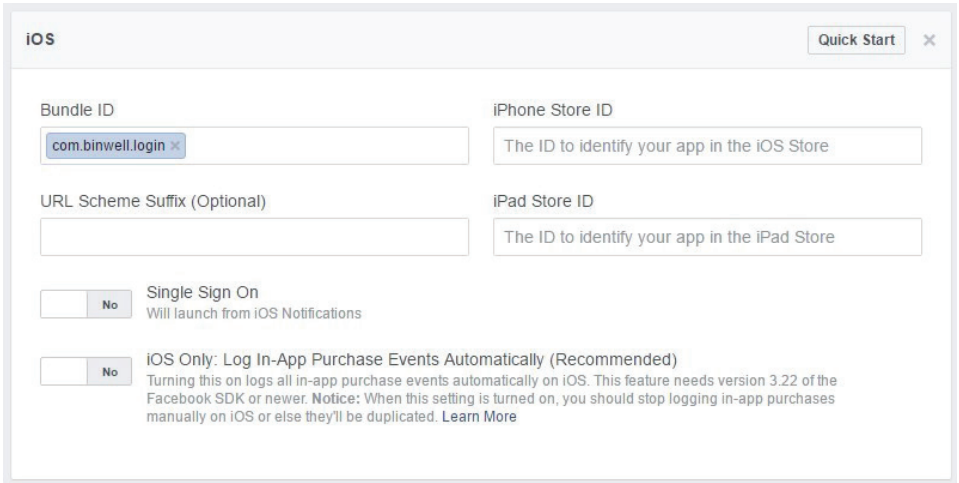


Рис. 11.2 ❖ Регистрация приложения iOS

Подключаем Facebook SDK к проектам iOS и Android

Для начала необходимо установить пакеты Facebook SDK от Xamarin для iOS и Android из Nuget (рис. 11.3).

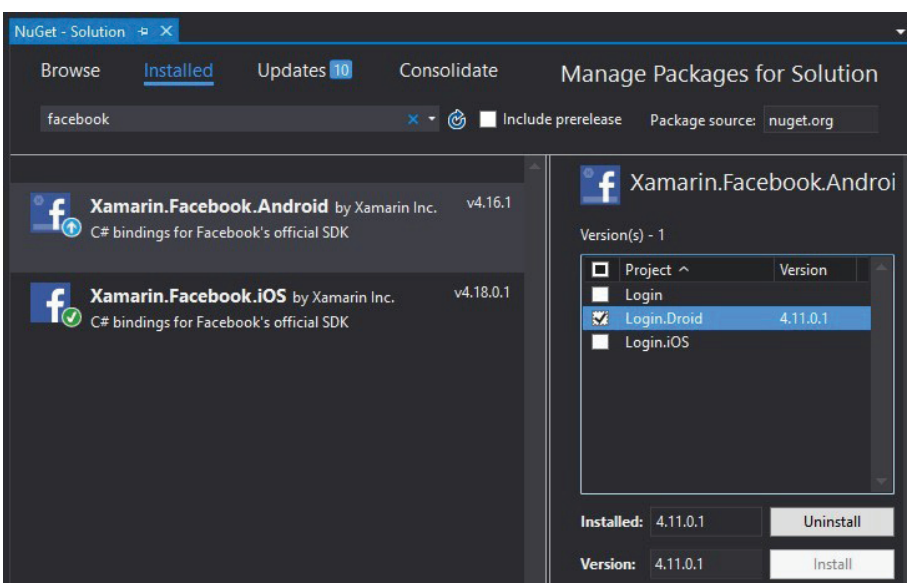


Рис. 11.3 ❖ Добавление Facebook SDK в платформенные проекты

Обратите внимание, что с Xamarin.Forms 2.3 на текущий момент совместима только версия Xamarin.Facebook.Android 4.11.0.1. Версия Xamarin.Facebook.iOS ограничений по совместимости не имеет.

Подключаем в Android

Для начала нам необходимо прописать специальные значения в файле `Resources/values/strings.xml`:

Листинг 11.1 ❖ Добавление параметров приложения Facebook

```
<string name="facebook_app_id">1102463466549096</string>
<string name="fb_login_protocol_scheme">fb1102463466549096</string>
```

где `1102463466549096` – это ваш App ID из настроек приложения Facebook. Дополнительно нам потребуется внести следующие изменения в `AndroidManifest.xml` (листинг 11.2).

Листинг 11.2 ❖ Настройки `AndroidManifest.xml`

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<application android:label="@string/app_name">
    <meta-data android:name="com.facebook.sdk.ApplicationId" android:value="@
string/facebook_app_id"/>
    <activity android:name="com.facebook.FacebookActivity" android:conf
igChanges="keyboard|keyboardHidden|screenLayout|screenSize|orientation"
android:theme="@android:style/Theme.Translucent.NoTitleBar" android:label="@
string/app_name" />
    <activity android:name="com.facebook.CustomTabActivity"
android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.VIEW" />
            <category android:name="android.intent.category.DEFAULT" />
            <category android:name="android.intent.category.BROWSABLE" />
            <data android:scheme="@string/fb_login_protocol_scheme" />
        </intent-filter>
        <provider android:authorities="com.facebook.app.
FacebookContentProvider1102463466549096" android:name="com.facebook.
FacebookContentProvider" android:exported="true" />
    </activity>
</application>
```

Далее вносим небольшие доработки в MainActivity.cs, показанные в листинге 11.3.

Листинг 11.3 ❖ Добавление кода, необходимого Facebook SDK, в код MainActivity

```
protected override void OnCreate(Bundle bundle)
{
    TabLayoutResource = Resource.Layout.Tabbar;
    ToolbarResource = Resource.Layout.Toolbar;

    base.OnCreate(bundle);
    FacebookSdk.SdkInitialize(ApplicationContext);
    Forms.Init(this, bundle);
    LoadApplication(new App());
}

protected override void OnResume()
{
    base.OnResume();
    AppEventsLogger.ActivateApp(Application);
}
```

На этом первичная инициализация Facebook SDK завершена.

Подключаем в iOS

По аналогии с Android нам будет необходимо внести правки в файл Info.plist, вставить следующие строки между <dict>...</dict> (листинг 11.4).

Листинг 11.4 ❖ Настройки Info.plist для требований Facebook SDK

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
  <key>CFBundleURLSchemes</key>
  <array>
    <string>fb1102463466549096</string>
  </array>
  </dict>
</array>
<key>FacebookAppID</key>
<string>1102463466549096</string>
```

```
<key>FacebookDisplayName</key>
<string>Binwell Social Demo</string>
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>fbapi</string>
  <string>fb-messenger-api</string>
  <string>fbauth2</string>
  <string>fbshareextension</string>
</array>
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>facebook.com</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSThirdPartyExceptionRequiresForwardSecrecy</key>
            <false/>
        </dict>
        <key>fbcdn.net</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSThirdPartyExceptionRequiresForwardSecrecy</key>
            <false/>
        </dict>
        <key>akamaihd.net</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSThirdPartyExceptionRequiresForwardSecrecy</key>
            <false/>
        </dict>
    </dict>
</dict>
```

И немного кода в `AppDelegate.cs` (листинг 11.5).

Листинг 11.5 ❖ Добавление кода в `AppDelegate` для работы Facebook SDK

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    Xamarin.Forms.Forms.Init();
    LoadApplication(new App());
```

```
    Facebook.CoreKit.Profile.EnableUpdatesOnAccessTokenChange(true);
Facebook.CoreKit.ApplicationDelegate.SharedInstance.FinishedLaunching(app,
options);

    return base.FinishedLaunching(app, options);
}

public override bool OpenUrl(UIApplication application, NSUrl url, string
sourceApplication, NSObject annotation)
{
    return Facebook.CoreKit.ApplicationDelegate.SharedInstance.
OpenUrl(application, url, sourceApplication, annotation);
}

public override void OnActivated(UIApplication application)
{
    Facebook.CoreKit.AppEvents.ActivateApp();
}
```

На этом предварительная подготовка завершена, и мы можем переходить к использованию Facebook SDK в приложении.

Интегрируем с Xamarin.Forms

Использовать Facebook SDK мы будем через механизм `DependencyService`. Для этого в первую очередь опишем нужные данные и интерфейс сервиса, как это показано в листинге 11.6.

Листинг 11.6 ❖ Кроссплатформенный интерфейс доступа к Facebook SDK на разных платформах

```
public interface IFacebookService
{
    Task<LoginResult> Login();
    void Logout();
}

public enum LoginState
{
    Failed,
    Canceled,
    Success
}
```

```
public class LoginResult
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string ImageUrl { get; set; }
    public string UserId { get; set; }
    public string Token { get; set; }
    public DateTimeOffset ExpireAt { get; set; }
    public LoginState LoginState { get; set; }
    public string ErrorString { get; set; }
}
```

Одной из целей подключения социальных сетей является возможность простого и удобного получения пользовательских данных, поэтому нам потребуется дополнительный запрос для получения email, который Facebook не отдает по умолчанию. Данные запросы будет необходимо реализовать отдельно для каждой платформы.

Реализация для Android

Для Android реализация интерфейса IFacebookService показана в листинге 11.7.

Листинг 11.7 ❖ Реализация IFacebookService для Android

```
[assembly: Dependency(typeof(AndroidFacebookService))]
namespace Login.Droid
{
    public class AndroidFacebookService: Java.Lang.Object, IFacebookService,
GraphRequest.IGraphJSONObjectCallback, GraphRequest.ICallback,
IFacebookCallback
    {
        public static AndroidFacebookService Instance => DependencyService.
Get<IFacebookService>() as AndroidFacebookService;

        readonly ICallbackManager _callbackManager = CallbackManagerFactory.
Create();
        readonly string[] _permissions = { @"public_profile", @"email",
@"user_about_me" };

        LoginResult _loginResult;
        TaskCompletionSource<LoginResult> _completionSource;
```

```
        public AndroidFacebookService()
        {
            LoginManager.Instance.RegisterCallback(_callbackManager, this);
        }

        public Task<LoginResult> Login()
        {
            _completionSource = new TaskCompletionSource<LoginResult>();
            LoginManager.Instance.LogInWithReadPermissions(Forms.Context as
Activity, _permissions);
            return _completionSource.Task;
        }

        public void Logout()
        {
            LoginManager.Instance.LogOut();
        }

        public void OnActivityResult(int requestCode, int resultCode, Intent
data)
        {
            _callbackManager?.OnActivityResult(requestCode, resultCode, data);
        }

        public void OnCompleted(JSONObject data, GraphResponse response)
        {
            OnCompleted(response);
        }

        public void OnCompleted(GraphResponse response)
        {
            if (response?.JSONObject == null)
                _completionSource?.TrySetResult(new LoginResult {LoginState =
LoginState.Canceled});
            else
            {
                _loginResult = new LoginResult
                {
                    FirstName = Profile.CurrentProfile.FirstName,
                    LastName = Profile.CurrentProfile.LastName,
                    Email = response.JSONObject.Has("email") ? response.
JSONObject.GetString("email") : string.Empty,
```

```
                    ImageUrl = response.JSONObject.GetJSONObject("picture")?.
GetJSONObject("data")?.GetString("url"),
                    Token = AccessToken.CurrentAccessToken.Token,
                    UserId = AccessToken.CurrentAccessToken.UserId,
                    ExpireAt = FromJavaDateTime(AccessToken.
CurrentAccessToken?.Expires?.Time),
                    LoginState = LoginState.Success
                };

                _completionSource?.TrySetResult(_loginResult);
            }
        }

        public void OnCancel()
        {
            _completionSource?.TrySetResult(new LoginResult { LoginState =
LoginState.Canceled });
        }

        public void OnError(FacebookException exception)
        {
            _completionSource?.TrySetResult(new LoginResult
            {
                LoginState = LoginState.Failed,
                ErrorString = exception?.Message
            });
        }

        public void OnSuccess(Java.Lang.Object result)
        {
            var facebookLoginResult = result.JavaCast<Xamarin.Facebook.Login.
LoginResult>();
            if (facebookLoginResult == null) return;

            var parameters = new Bundle();
            parameters.PutString("fields", "id,email,picture.type(large)");
            var request = GraphRequest.NewMeRequest(facebookLoginResult.
AccessToken, this);
            request.Parameters = parameters;
            request.ExecuteAsync();
        }

        static DateTimeOffset FromJavaDateTime(long? longTimeMillis)
```

```
        {
            var epoch = new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc);
            return longTimeMillis != null ? epoch.
AddMilliseconds(longTimeMillis.Value) : DateTimeOffset.MinValue;
        }
    }
}
```

Дополнительно потребуется добавить обработчик в `MainActivity.cs`, как это показано в листинге 11.8.

Листинг 11.8 ❖ Добавление метода `OnActivityResult` в MainActivity

```
protected override void OnActivityResult(int requestCode, Result resultCode,
Intent data){
    base.OnActivityResult(requestCode, resultCode, data);
    AndroidFacebookService.Instance.OnActivityResult(requestCode, (int)
resultCode, data);
}
```

Реализация для iOS

В листинге 11.9 показана реализация интерфейса IFacebookService в iOS.

Листинг 11.9 ❖ Реализиция доступа к Facebook SDK в iOS

```
[assembly: Dependency(typeof(AppleFacebookService))]
namespace Login.iOS
{
    public class AppleFacebookService: IFacebookService
    {
        readonly LoginManager _loginManager = new LoginManager();
        readonly string[] _permissions = { @"public_profile", @"email",
@"user_about_me" };

        LoginResult _loginResult;
        TaskCompletionSource<LoginResult> _completionSource;

        public Task<LoginResult> Login()
        {
            _completionSource = new TaskCompletionSource<LoginResult>();
            _loginManager.LogInWithReadPermissions(_permissions,
GetCurrentViewController(), LoginManagerLoginHandler);
```

```
            return _completionSource.Task;
        }

        public void Logout()
        {
            _loginManager.LogOut();
        }

        void LoginManagerLoginHandler(LoginManagerLoginResult result, NSError
error)
        {
            if (result.IsCancelled)
                _completionSource.TrySetResult(new LoginResult {LoginState =
LoginState.Canceled});
            else if (error != null)
                _completionSource.TrySetResult(new LoginResult { LoginState =
LoginState.Failed, ErrorString = error.LocalizedDescription });
            else
            {
                _loginResult = new LoginResult
                {
                    Token = result.Token.TokenString,
                    UserId = result.Token.UserID,
                    ExpireAt = result.Token.ExpirationDate.ToDateTime()
                };

                var request = new GraphRequest(@"me", new
NSDictionary(@"fields", @"email"));
                request.Start(GetEmailRequestHandler);
            }
        }

        void GetEmailRequestHandler(GraphRequestConnection connection, NSObject
result, NSError error)
        {
            if (error != null)
                _completionSource.TrySetResult(new LoginResult { LoginState =
LoginState.Failed, ErrorString = error.LocalizedDescription });
            else
            {
                _loginResult.FirstName = Profile.CurrentProfile.FirstName;
                _loginResult.LastName = Profile.CurrentProfile.LastName;
```

```
                _loginResult.ImageUrl = Profile.CurrentProfile.
ImageUrl(ProfilePictureMode.Square, new CGSize()).ToString();

                var dict = result as NSDictionary;
                var emailKey = new NSString(@"email");
                if (dict != null && dict.ContainsKey(emailKey))
                    _loginResult.Email = dict[emailKey]?.ToString();

                _loginResult.LoginState = LoginState.Success;
                _completionSource.TrySetResult(_loginResult);
            }
        }

        static UIViewController GetCurrentViewController()
        {
            var viewController = UIApplication.SharedApplication.KeyWindow.
RootViewController;
            while (viewController.PresentedViewController != null)
                viewController = viewController.PresentedViewController;
            return viewController;
        }
    }
}
```

Подключаем в Xamarin.Forms

Для доступа к созданным реализациям достаточно вставить следующий обработчик события `Clicked` для кнопки **Facebook Login** в вашей кроссплатформенной части, как это показано в листинге 11.10.

Листинг 11.10 ❖ Использование Facebook SDK в приложении

```
var loginResult = await DependencyService.Get<IFacebookService>().Login();

switch (loginResult.LoginState)
{
    case LoginState.Canceled:
        // Обработать
        break;
    case LoginState.Success:
        var str = $"Hi {loginResult.FirstName}! Your email is {loginResult.
Email}";
        break;
```

```
    default:
        // Обработать ошибки
        break;
}
```

На этом кодирование завершено! Делаем сборку, запускаем и... легко авторизуемся с помощью нативных SDK.

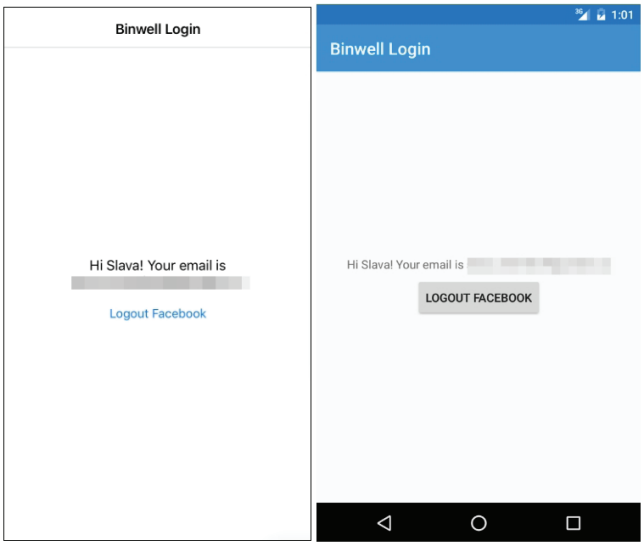


Рис. 11.4 ❖ Авторизация с помощью Facebook SDK

Полный код проекта с пошаговыми изменениями расположен в репозитории **https://github.com/binwell-university/XamarinBookSamples**.

11.2. ВКонтакте

В целом процесс интеграции ВКонтакте будет сильно напоминать работу с Facebook, так что смело заходите на страницу управления приложениями: **https://vk.com/apps?act=manage**.

Нажимаем **Создать приложение** и выбираем **Standalone-приложение**.

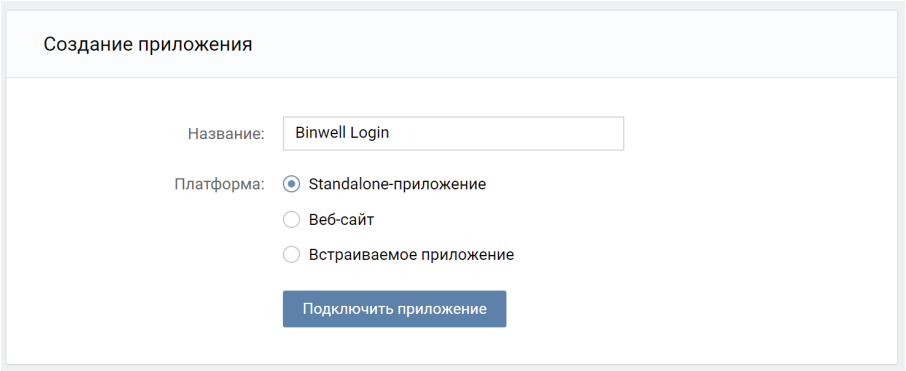


Рис. 11.5 ❖ Создание приложения ВКонтакте

Далее идем в **Настройки** и вводим данные о приложении. **Отпечаток сертификата** – это Key Hashes, полученные ранее для Facebook.

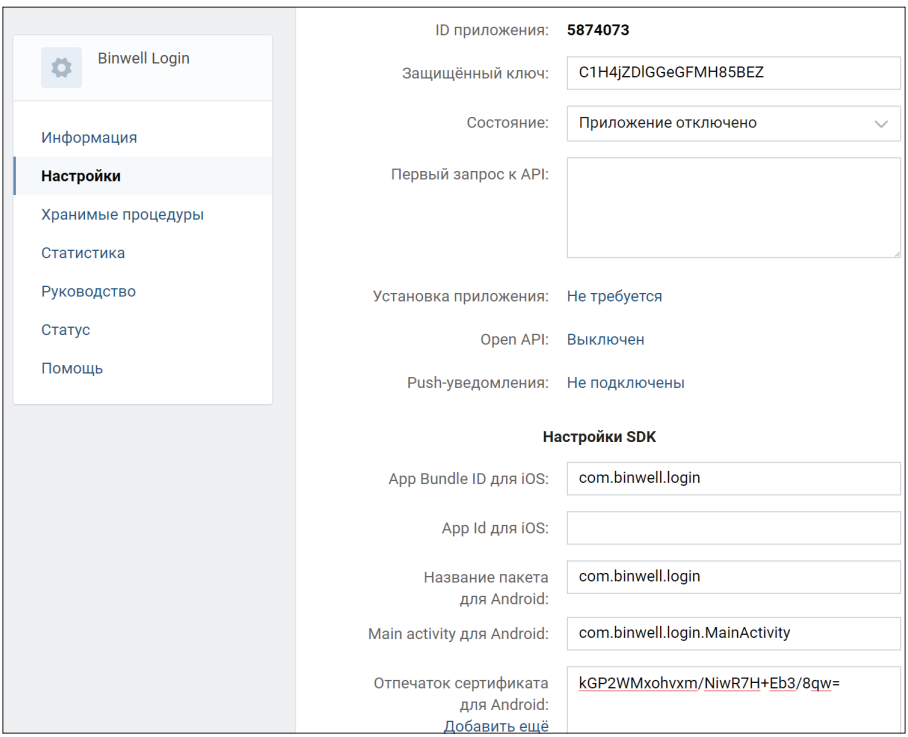


Рис. 11.6 ❖ Настройки мобильного приложения для доступа к API ВКонтакте

На этом подготовительная часть завершена.

Подключаем ВКонтакте SDK к проектам iOS и Android

Для Xamarin доступно достаточно много готовых bindings, однако полноценная библиотека для доступа к нативным возможностям ВКонтакте SDK есть всего одна: **https://www.nuget.org/packages/Xamarin.VKontakte**. Библиотека какое-то время пребывала в стадии beta и сейчас готова к использованию.

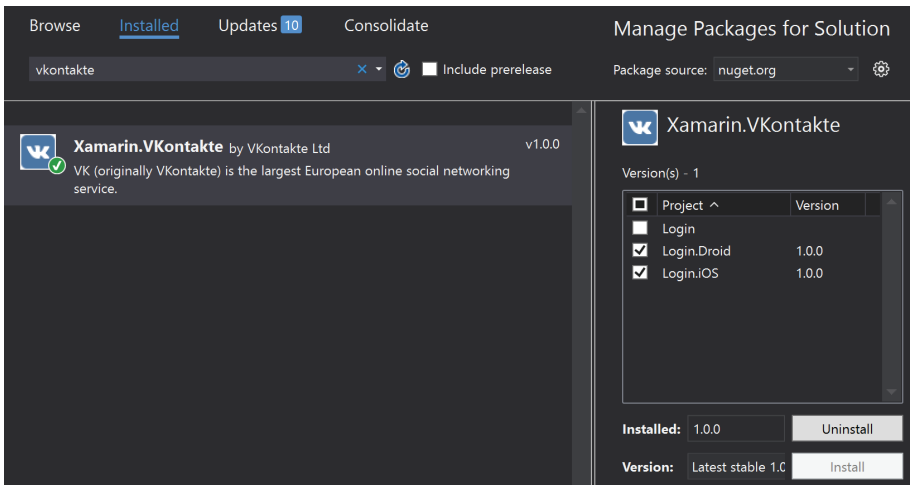


Рис. 11.7 ❖ Добавляем библиотеки Xamarin.VKontakte к вашему проекту

Подключаем в iOS

Вносим правки в `Info.plist`. Расширяем `CFBundleURLTypes` значениями для ВКонтакте, как это показано в листинге 11.11.

Листинг 11.11 ❖ Добавление нужных значений в `Info.plist` для работы ВКонтакте SDK

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
      <key>CFBundleURLName</key>
      <string>vk5874073</string>
    <key>CFBundleURLSchemes</key>
```

```
    <array>
      <string>fb1102463466549096</string>
      <string>vk5874073</string>
    </array>
  </dict>
</array>
```

Листинг 11.12 ❖ Добавляем новые LSApplicationQueriesSchemes

```
<string>vk</string>
<string>vk-share</string>
<string>vkauthorize</string>
```

Листинг 11.13 ❖ И также новый домен в NSAppTransportSecurity

```
<key>vk.com</key>
<dict>
  <key>NSExceptionRequiresForwardSecrecy</key>
  <false/>
  <key>NSIncludesSubdomains</key>
  <true/>
  <key>NSExceptionAllowsInsecureHTTPLoads</key>
  <true/>
</dict>
```

Листинг 11.14 ❖ После этого вносим правки в AppDelegate.cs

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    Xamarin.Forms.Forms.Init();

    LoadApplication(new App());

    Facebook.CoreKit.Profile.EnableUpdatesOnAccessTokenChange(true);
    Facebook.CoreKit.ApplicationDelegate.SharedInstance.FinishedLaunching(app,
options);

    VKSdk.Initialize("5874073");

    return base.FinishedLaunching(app, options);
}

public override bool OpenUrl(UIApplication application, NSUrl url, string
sourceApplication, NSObject annotation)
{
```

```
    return VKSdk.ProcessOpenUrl(url, sourceApplication)
        || Facebook.CoreKit.ApplicationDelegate.SharedInstance.
OpenUrl(application, url, sourceApplication, annotation)
        || base.OpenUrl(application, url, sourceApplication, annotation);
}
```

На этом первичная инициализация iOS завершена.

Подключаем в Android

А вот для Android придется дополнительно переопределить свой класс `Application` для корректной инициализации SDK (листинг 11.15).

Листинг 11.15 ❖ Создание класса `MainApplication` для работы Facebook SDK

```
[Application]
public class MainApplication : Application
{
    public MainApplication(IntPtr handle, JniHandleOwnership transer)
      :base(handle, transer)
    {
    }

    public override void OnCreate()
    {
        base.OnCreate();
        VKSdk.Initialize(this).WithPayments();
    }
}
```

Листинг 11.16 ❖ Теперь добавим ID приложения в `strings.xml`

```
<integer name="com_vk_sdk_AppId">5874073</integer>
<string name="vk_data_theme">vk5874073</string>
```

Листинг 11.17 ❖ И еще немного кода в `AndroidManifest.xml` между `<application …>`

```
<activity android:name="com.binwell.login.MainActivity"
android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
```

```
    <data android:scheme="@string/vk_data_theme" />
  </intent-filter>
</activity>
```

Листинг 11.18 ❖ И завершим расширением MainActivity

```
protected override async void OnActivityResult(int requestCode, Result
resultCode, Intent data)
{
    bool vkResult;
    var task = VKSdk.OnActivityResultAsync(requestCode, resultCode, data, out
vkResult);

    if (!vkResult)
    {
        base.OnActivityResult(requestCode, resultCode, data);
        AndroidFacebookService.Instance.OnActivityResult(requestCode, (int)
resultCode, data);
        return;
    }

    try
    {
        var token = await task;
        // Get token
    }
    catch (Exception e)
    {
        // Handle exception
    }
}
```

Интегрируем с Xamarin.Forms

По аналогии с Facebook мы создадим свой интерфейс в PCL-проекте для работы с новым SDK, как это показано в листинге 11.19.

Листинг 11.19 ❖ Кроссплатформенный интерфейс для доступа к ВКонтакте SDK

```
public interface IVkService {
    Task<LoginResult> Login();
    void Logout();
}
```

Реализация для iOS

В листинге 11.20 реализация для iOS будет выглядеть следующим образом:

Листинг 11.20 ❖ Реализация для iOS

```
[assembly: Dependency(typeof(AppleVkService))]
namespace Login.iOS
{
    public class AppleVkService : NSObject, IVkService, IVKSdkDelegate,
IVKSdkUIDelegate
    {
        readonly string[] _permissions = {
            VKPermissions.Email,
            VKPermissions.Offline
        };

        LoginResult _loginResult;
        TaskCompletionSource<LoginResult> _completionSource;

        public AppleVkService()
        {
            VKSdk.Instance.RegisterDelegate(this);
            VKSdk.Instance.UiDelegate = this;
        }

        public Task<LoginResult> Login()
        {
            _completionSource = new TaskCompletionSource<LoginResult>();
            VKSdk.Authorize(_permissions);
            return _completionSource.Task;
        }

        public void Logout()
        {
            _loginResult = null;
            _completionSource = null;
        }

        [Export("vkSdkTokenHasExpired:")]
        public void TokenHasExpired(VKAccessToken expiredToken)
        {
```

```
            VKSdk.Authorize(_permissions);
        }

        public new void Dispose()
        {
            VKSdk.Instance.UnregisterDelegate(this);
            VKSdk.Instance.UiDelegate = null;
            SetCancelledResult();
        }

        public void AccessAuthorizationFinished(VKAuthorizationResult result)
        {
            if (result?.Token == null)
                SetErrorResult(result?.Error?.LocalizedDescription ?? @"VK
authorization unknown error");
            else
            {
                _loginResult = new LoginResult
                {
                    Token = result.Token.AccessToken,
                    UserId = result.Token.UserId,
                    Email = result.Token.Email,
                    ExpireAt = Utils.FromMsDateTime(result.Token.ExpiresIn),
                };
                Task.Run(GetUserInfo);
            }
        }

        async Task GetUserInfo()
        {
            var request = VKApi.Users.Get(NSDictionary.FromObjectAndKey((NSStri
ng)@"photo_400_orig", VKApiConst.Fields));
            var response = await request.ExecuteAsync();
            var users = response.ParsedModel as VKUsersArray;
            var account = users?.FirstObject as VKUser;
            if (account != null && _loginResult != null)
            {
                _loginResult.FirstName = account.first_name;
                _loginResult.LastName = account.last_name;
                _loginResult.ImageUrl = account.photo_400_orig;
                _loginResult.LoginState = LoginState.Success;
                SetResult(_loginResult);
```

```
            }
            else
                SetErrorResult(@"Unable to complete the request of user info");
        }

        public void UserAuthorizationFailed()
        {
            SetErrorResult(@"VK authorization unknown error");
        }

        public void ShouldPresentViewController(UIViewController controller)
        {
            Device.BeginInvokeOnMainThread(() => Utils.
GetCurrentViewController().PresentViewController(controller, true, null));
        }

        public void NeedCaptchaEnter(VKError captchaError)
        {
            Device.BeginInvokeOnMainThread(() => VKCaptchaViewController.
Create(csaptchaError).PresentIn(Utils.GetCurrentViewController()));
        }

        void SetCancelledResult()
        {
            SetResult(new LoginResult { LoginState = LoginState.Canceled });
        }

        void SetErrorResult(string errorString)
        {
            SetResult(new LoginResult { LoginState = LoginState.Failed,
ErrorString = errorString });
        }

        void SetResult(LoginResult result)
        {
            _completionSource?.TrySetResult(result);
            _loginResult = null;
            _completionSource = null;
        }
    }
}
```

Реализация для Android

Листинг 11.21. Для Android тоже ничего необычного

```
[assembly: Dependency(typeof(AndroidVkService))]
namespace Login.Droid
{
    public class AndroidVkService : Java.Lang.Object, IVkService
    {
        public static AndroidVkService Instance => DependencyService.
Get<IVkService>() as AndroidVkService;

        readonly string[] _permissions = {
            VKScope.Email,
            VKScope.Offline
        };

        TaskCompletionSource<LoginResult> _completionSource;
        LoginResult _loginResult;

        public Task<LoginResult> Login()
        {
            _completionSource = new TaskCompletionSource<LoginResult>();
            VKSdk.Login(Forms.Context as Activity, _permissions);
            return _completionSource.Task;
        }

        public void Logout()
        {
            _loginResult = null;
            _completionSource = null;
            VKSdk.Logout();
        }

        public void SetUserToken(VKAccessToken token)
        {
            _loginResult = new LoginResult
            {
                Email = token.Email,
                Token = token.AccessToken,
                UserId = token.UserId,
                ExpireAt = Utils.FromMsDateTime(token.ExpiresIn)
            };
```

```
            Task.Run(GetUserInfo);
        }

        async Task GetUserInfo()
        {
            var request = VKApi.Users.Get(VKParameters.From(VKApiConst.Fields,
@"photo_400_orig,"));
            var response = await request.ExecuteAsync();
            var jsonArray = response.Json.OptJSONArray(@"response");
            var account = jsonArray?.GetJSONObject(0);
            if (account != null && _loginResult != null)
            {
                _loginResult.FirstName = account.OptString(@"first_name");
                _loginResult.LastName = account.OptString(@"last_name");
                _loginResult.ImageUrl = account.OptString(@"photo_400_orig");
                _loginResult.LoginState = LoginState.Success;
                SetResult(_loginResult);
            }
            else
                SetErrorResult(@"Unable to complete the request of user info");
        }

        public void SetErrorResult(string errorMessage)
        {
            SetResult(new LoginResult { LoginState = LoginState.Failed,
ErrorString = errorMessage });
        }

        public void SetCanceledResult()
        {
            SetResult(new LoginResult { LoginState = LoginState.Canceled });
        }

        void SetResult(LoginResult result)
        {
            _completionSource?.TrySetResult(result);
            _loginResult = null;
            _completionSource = null;
        }
    }
}
```

Подключаем в Xanarin.Forms

Теперь можно использовать установленное на телефон приложение ВКонтакте для авторизации пользователей в вашем приложении.

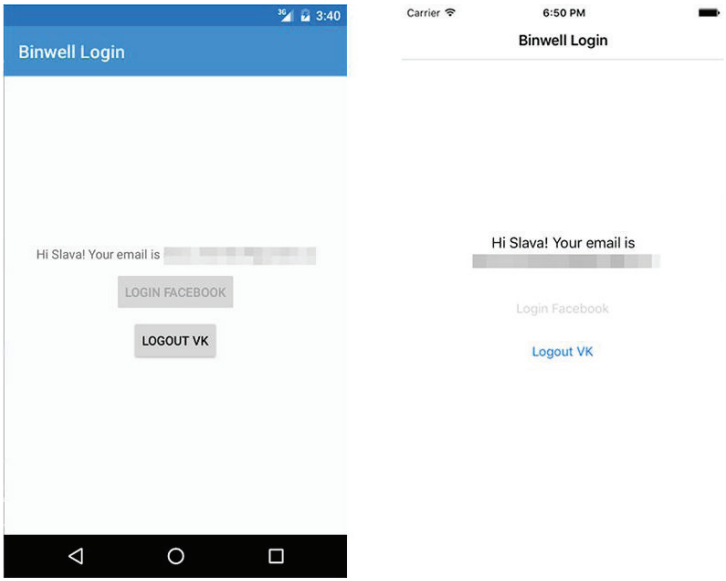


Рис. 11.8 ❖ Использование ВКонтакте SDK для авторизации

Напоминаем, что для публикации приложений (чтобы кто-нибудь кроме вас мог авторизоваться) необходимо выполнить дополнительные действия для каждой социальной сети.

11.3. OAuth

После знакомства с SDK от Facebook и ВКонтакте (**https://habrahabr.ru/company/microsoft/blog/323296/** и **https://habrahabr.ru/company/microsoft/blog/321454/**) можем перейти к одному из самых популярных (на текущий момент) механизмов внешней авторизации пользователей – OAuth (**https://oauth.net/**). Большинство популярных сервисов (вроде Twitter, Microsoft Live, Github и т. д.) предоставляют своим пользователям возможность входа в сторонние приложения с помощью одного привычного аккаунта. Научившись работать с OAuth,

вы легко сможете подключать все эти сервисы и забирать из них информацию о пользователе.

Предполагается, что вы уже знакомы с тем, как работает OAuth, а если нет – рекомендуем хорошую статью на Хабре по ссылке **https://habrahabr.ru/company/mailru/blog/115163/**. Если коротко, то при авторизации OAuth пользователь перенаправляется с одной веб-страницы на другую (обычно 2–3 шага), до тех пор пока не перейдет на конечный URL. Этот финальный переход и будет отловлен в приложении (если писать логику самому) на уровне WebView, а нужные данные (token и срок его валидности) будут указаны прямо в URL.

Небольшой список популярных сервисов, которые предоставляют возможность авторизации пользователей по OAuth: Одноклассники, Mail.ru, Dropbox, Foursquare, GitHub, Instagram, LinkedIn, Microsoft, Slack, SoundCloud, Visual Studio Online, Trello.

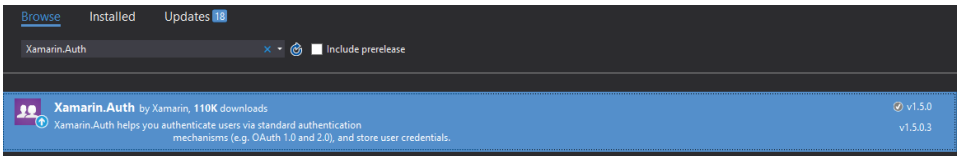
Xamarin.Auth

Для того чтобы работать с OAuth в Xamarin, мы остановимся на простой и удобной библиотеке Xamarin.Auth (**https://github.com/xamarin/Xamarin.Auth**), которая развивается уже не первый год и имеет все необходимые для нас механизмы:

1. Отображение браузера со страницами авторизации.
2. Управление потоком редиректов и процессом авторизации.
3. Получение нужных данных.
4. Предоставление механизмов для дополнительных запросов к сервису, например для получения информации о пользователе.

Также Xamarin.Auth поддерживает возможность хранения учетных данных пользователя в защищенном хранилище. В общем, зрелый и качественный компонент с необходимой функциональностью.

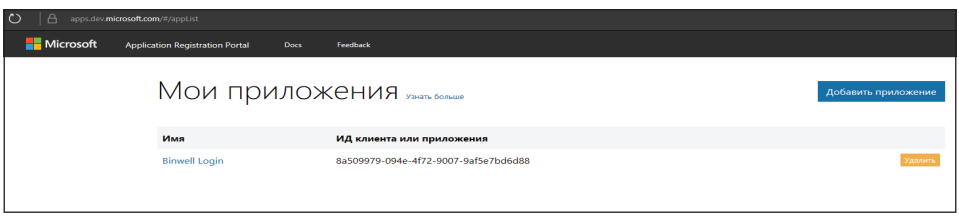
Рекомендуем устанавливать Xamarin.Auth из Nuget, так как версия в Xamarin Components уже давно устарела и не обновляется.



Напомню, что мы уже ранее рассказывали про авторизацию с помощью SDK от Facebook и ВКонтакте. В нашем примере мы вынесли всю логику авторизации в платформенные проекты, оставив в PCL только интерфейсы. Для OAuth мы пойдем тем же путем, несмотря на поддержку PCL в самом Xamarin.Auth.

Помимо Xamarin.Auth, можем также порекомендовать библиотеку Xamarin.Forms.Oauth (**https://github.com/Bigsby/Xamarin.Forms.OAuth**). Даже если вы используете классический Xamarin, в исходных кодах этого проекта можно найти множество готовых конфигураций для различных сервисов.

Мы же в качестве примера работы Oauth2 подключим авторизацию с помощью Microsoft. Первым делом создадим приложение на сайте **https://apps.dev.microsoft.com** и получим там Client ID (ИД клиента или приложения).



Подключаем авторизацию в кроссплатформенной части

На уровне кроссплатформы все как обычно – делаем простой интерфейс IOAuthService для платформенного сервиса, никаких новых зависимостей в проект не добавляем.

Листинг 11.22 ❖ Добавление кроссплатформенного интерфейса для авторизации по OAuth

```
public interface IOAuthService
{
   Task<LoginResult> Login();
   void Logout();
}
```

Ну и, конечно же, будет необходимо добавить обращение к методам `DependencyService.Get<IOAuthService>().Login()` и `DependencyService.Get<IOAuthService>().Logout()` внутри нашей страницы авторизации.

Также не составит проблем добавить поддержку нескольких OAuth-сервисов. Для этого можно добавить в методы `Login()` и `Logout()` аргумент `providerName` (тип `string`, `int` или `enum`) и в зависимости от его значения выбирать поставщика услуг.

Реализация платформенной части

Как уже отмечалось ранее, необходимо добавить библиотеки Xamarin.Auth из Nuget в каждый платформенный проект, в нашем случае – iOS и Android. Дальше пишем нашу реализацию `IOAuthService` для каждой платформы и регистрируем ее в качестве `Dependency`.

Листинг 11.23 ❖ Теперь нам достаточно создать экземпляр класса `OAuth2Authenticator` с нужными параметрами

```
var auth = new OAuth2Authenticator
       (
           clientId: "ВАШ_CLIENT_ID",
           scope: "wl.basic, wl.emails, wl.photos",
           authorizeUrl: new Uri("https://login.live.com/oauth20_authorize.srf"),
           redirectUrl: new Uri("https://login.live.com/oauth20_desktop.srf"),
           clientSecret: null,
           accessTokenUrl: new Uri("https://login.live.com/oauth20_token.srf")
       )
       {
           AllowCancel = true
       };
```

Листинг 11.24 ❖ Теперь повесим обработчик завершения авторизации

```
auth.Completed += AuthOnCompleted;
```

Все – можно показать модальное окно со встроенным веб-браузером для авторизации, получаемое через метод `auth.GetUI()`.

Листинг 11.25 ❖ Примерно так это можно сделать на iOS

```
UIApplication.SharedApplication.KeyWindow.RootViewController.
PresentViewController(auth.GetUI(), true, null);
```

Листинг 11.26 ❖ Каким может получиться код на Android при использовании Xamarin.Forms

```
Forms.Context.StartActivity(auth.GetUI(Forms.Context));
```

Листинг 11.27 ❖ После успешной авторизации вызовется наш метод `AuthOnCompleted()`, и для iOS будет необходимо скрыть модальное окно с браузером (на Android само скроется)

```
UIApplication.SharedApplication.KeyWindow.RootViewController.
DismissViewController(true, null);
```

Листинг 11.28 ❖ Теперь можно получать нужные данные (`access_token` и время его жизни в секундах – `expires_in`)

```
var token = authCompletedArgs.Account.Properties["access_token"];
var expireIn = Convert.ToInt32(authCompletedArgs.Account.Properties["expires_
in"]);
var expireAt = DateTimeOffset.Now.AddSeconds(expireIn);
```

И нам остался последний шаг – получить расширенную информацию из профиля пользователя, включая email и ссылку на аватарку. Для этого в Xamarin.Auth есть специальный класс `OAuth2Request`, с помощью которого удобно делать подобные запросы.

Листинг 11.29 ❖ Отправка `OAuth2Request`

```
var request = new OAuth2Request("GET", new Uri("https://apis.live.net/v5.0/me"),
null, account);
var response = await request.GetResponseAsync();
```

Теперь нам приходит JSON с данными пользователя, и мы можем их сохранить и отобразить в приложении.

Листинг 11.30 ❖ Использование полученных данных из Live API

```
if (response.StatusCode == HttpStatusCode.OK)
{
    var userJson = response.GetResponseText();
```

```
    var jobject = JObject.Parse(userJson);
    result.LoginState = LoginState.Success;
    result.Email = jobject["emails"]?["preferred"].ToString();
    result.FirstName = jobject["first_name"]?.ToString();
    result.LastName = jobject["last_name"]?.ToString();
    result.ImageUrl = jobject["picture"]?["data"]?["url"]?.ToString();
    var userId = jobject["id"]?.ToString();
    result.UserId = userId;
    result.ImageUrl = $"https://apis.live.net/v5.0/{userId}/picture";
}
```

Как видим, ничего сложного нет. Вопрос в том, чтобы правильно прописать URL для процесса авторизации. Ну и помнить, что поле `expires_in` содержит время в секундах (это вызывает частые вопросы).

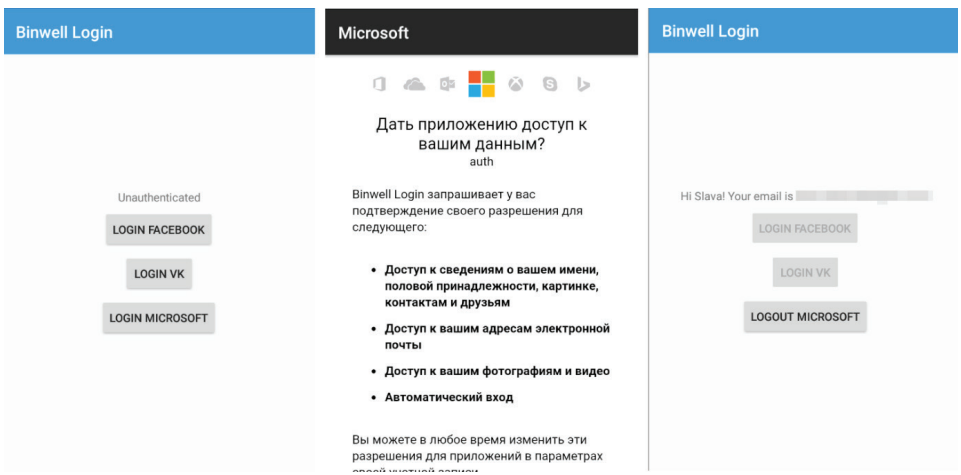


Рис. 11.9 ❖ Использование авторизации Microsoft в приложении

В реальных проектах также рекомендуем назначить обработчик ошибок на событие `auth.Error`, чтобы ни одна проблема не осталась без решения.

Сегодня мы завершили рассмотрение всех популярных способов авторизации пользователей и получения базовой информации о них через внешние сервисы. Описанные механизмы подходят как для Xamarin.Forms, так и для классического Xamarin iOS/Android. Полные исходные коды проекта со всеми примерами можно найти в репозитории **https://github.com/binwell-university/XamarinBookSamples**.

Заключение

Поздравляю! Вы добрались до конца этой непростой книги, в которой я постарался описать весь процесс разработки мобильных приложений, включая выбор инструментов, проектирование, создание «скелета» проекта, автоматизацию и решение ряда повседневных задач программиста. Объединенные в одно целое, данные улучшения делают процесс разработки более простым и понятным.

Описанные в книге подходы совместимы с базовыми принципами Agile/DevOps и могут быть легко адаптированы под различные инструменты разработки приложений с пользовательским интерфейсом.

Выражаю благодарность моим коллегам по компании Binwell, отдельно Кириллу Ашихмину и Артему Тищенко за помощь с примерами.

Буду рад получить ваши отзывы и комментарии по улучшению книги на **editor@binwell.com**!

Ваш
Черников Вячеслав

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. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Вячеслав Черников

**Разработка мобильных приложений на C#
для iOS и Android**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Чистякова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 15,28. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**