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 2 раза. Плюс в 2 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.

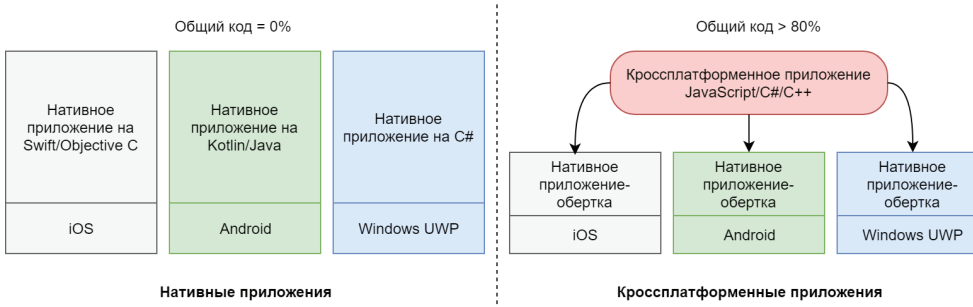

Рис. 1.1 ❖ Отличие нативной и кроссплатформенной мобильной разработки

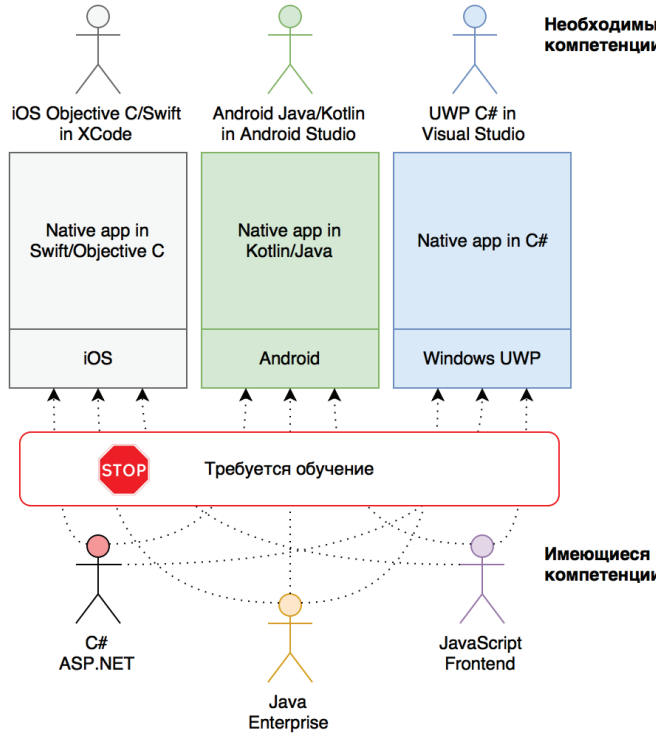

Рис. 1.2 ❖ Компетенции мобильной разработки

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.

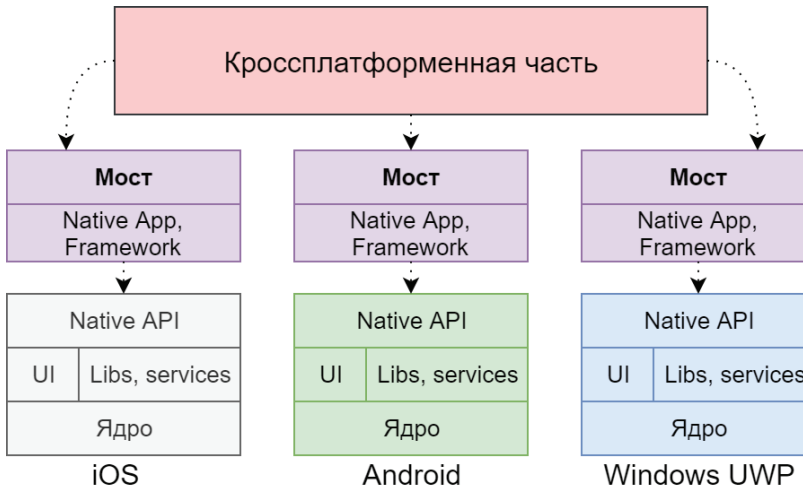

Рис. 1.3 ❖ Общая архитектура кроссплатформенных фреймворков

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.

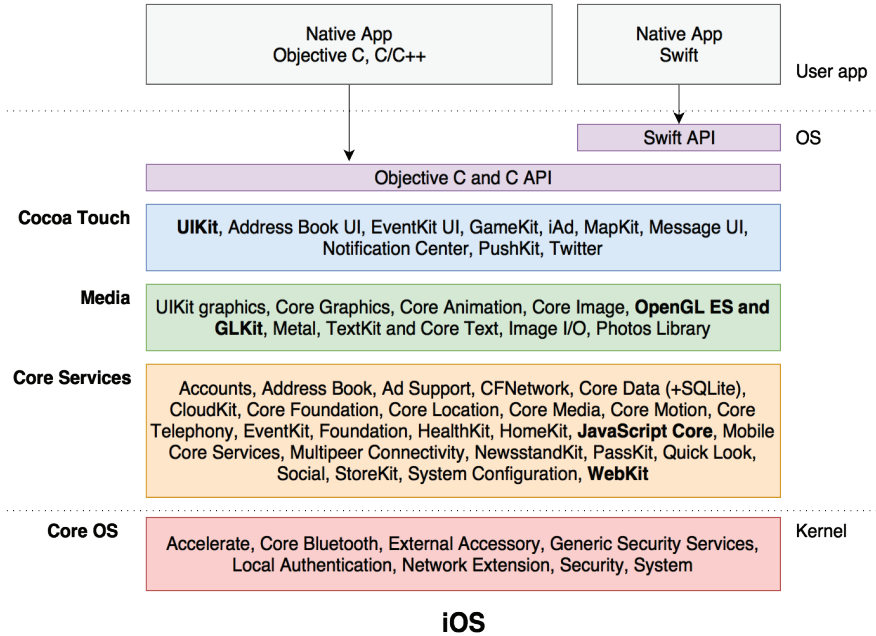

Рис. 1.4 ❖ Архитектура iOS

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.

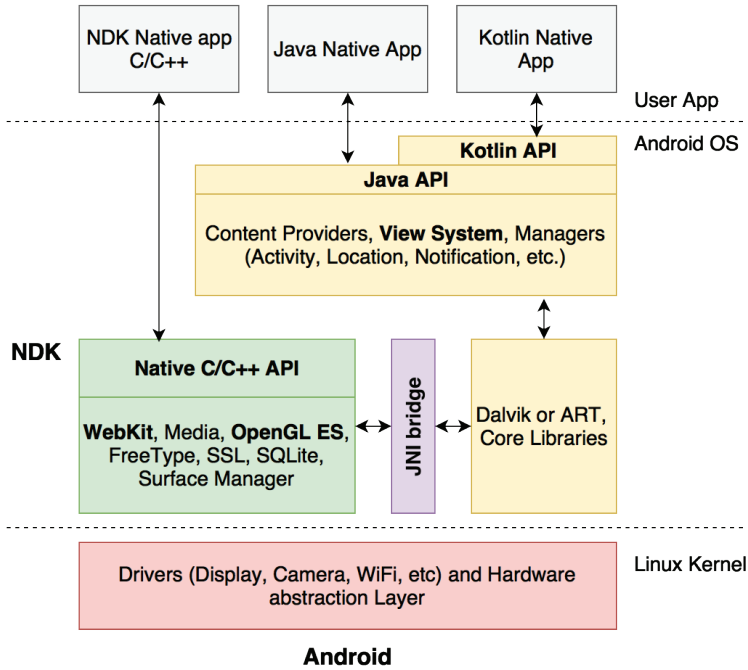

Рис. 1.5 ❖ Архитектура Android

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**).

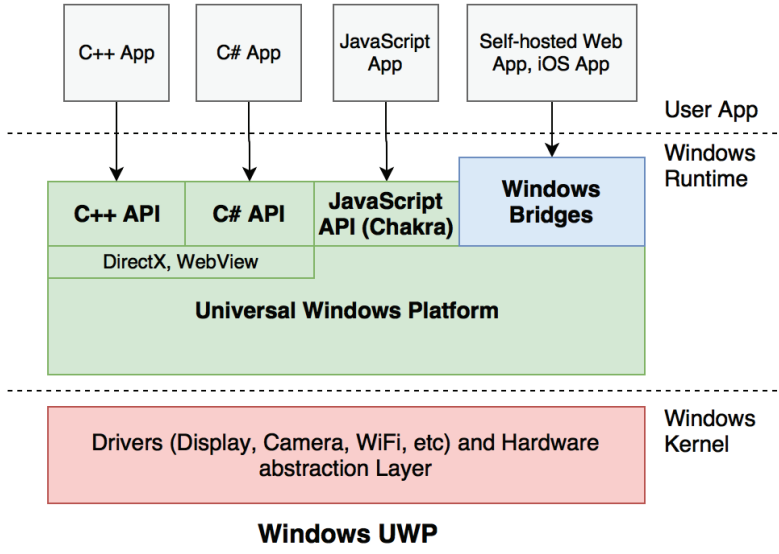

Рис. 1.6 ❖ Архитектура Windows UWP

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.

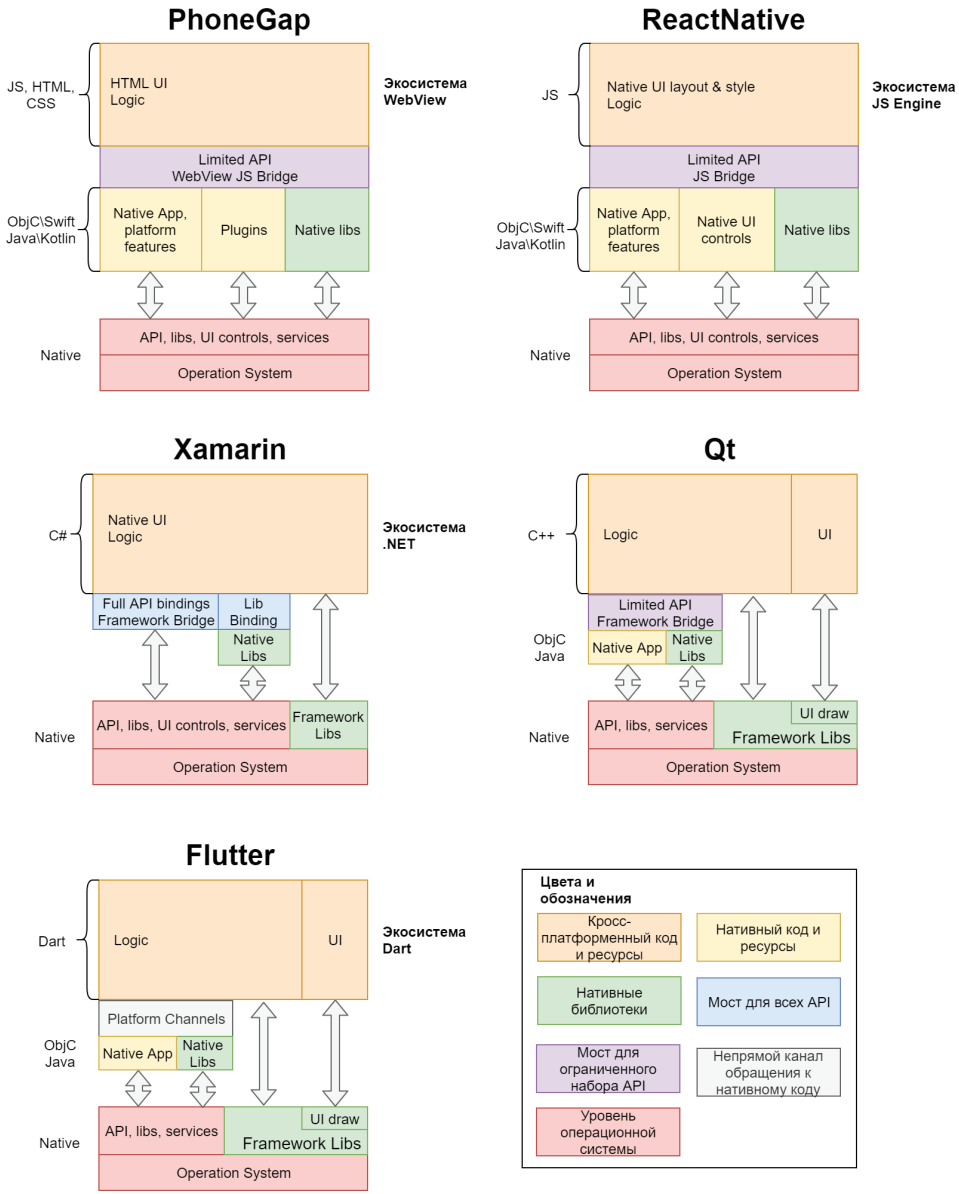

Рис. 1.7 ❖ Архитектуры кроссплатформенных фреймворков

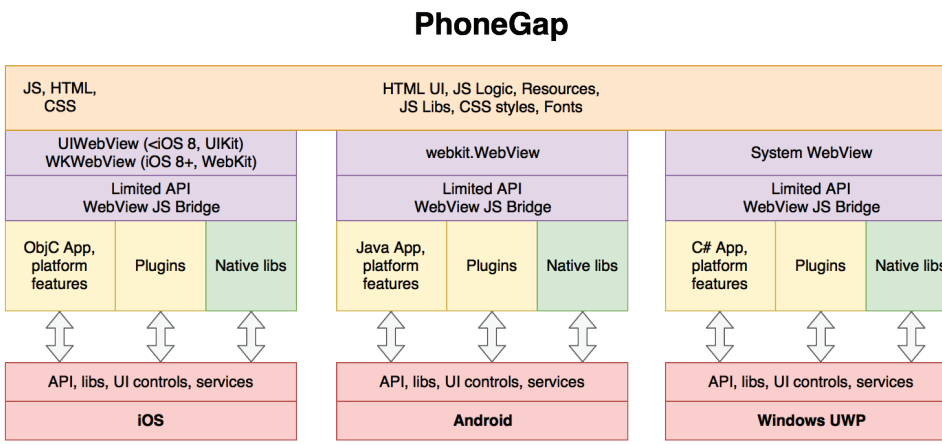

Рис. 1.8 ❖ Архитектура PhoneGap

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.

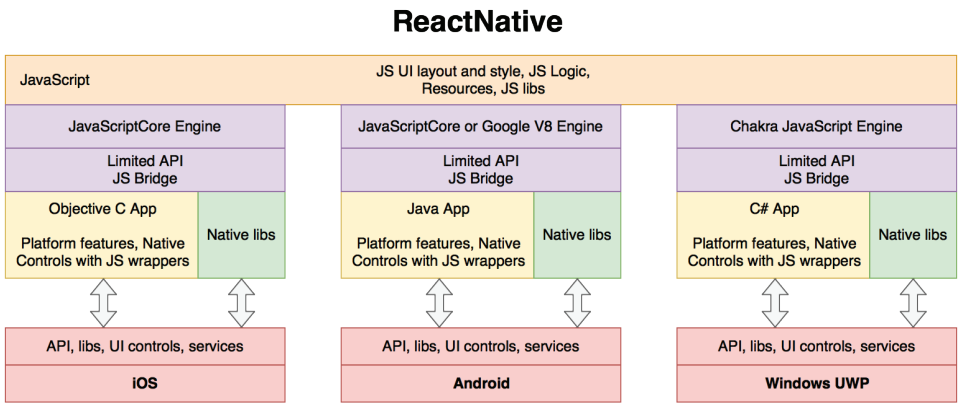

Рис. 1.9 ❖ Архитектура ReactNative

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.

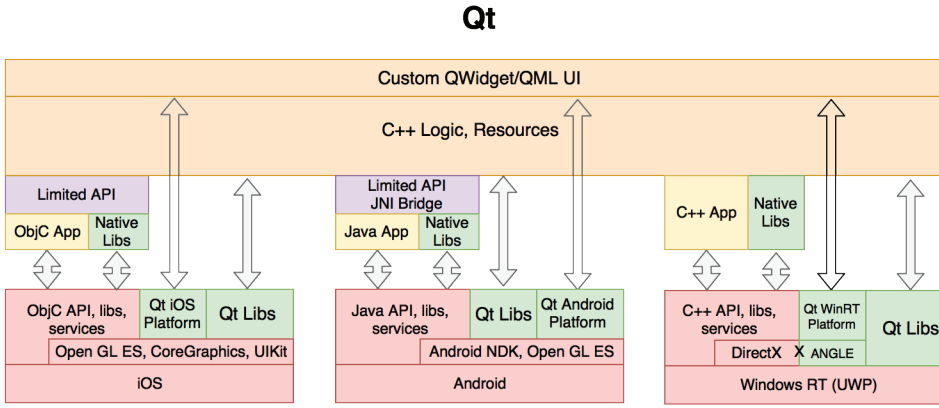

Рис. 1.10 ❖ Архитектура Qt

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.

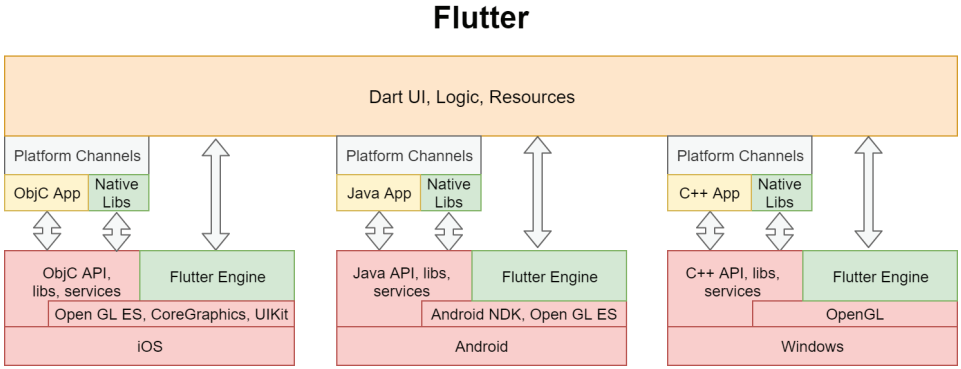

Рис. 1.11 ❖ Архитектура Flutter

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).

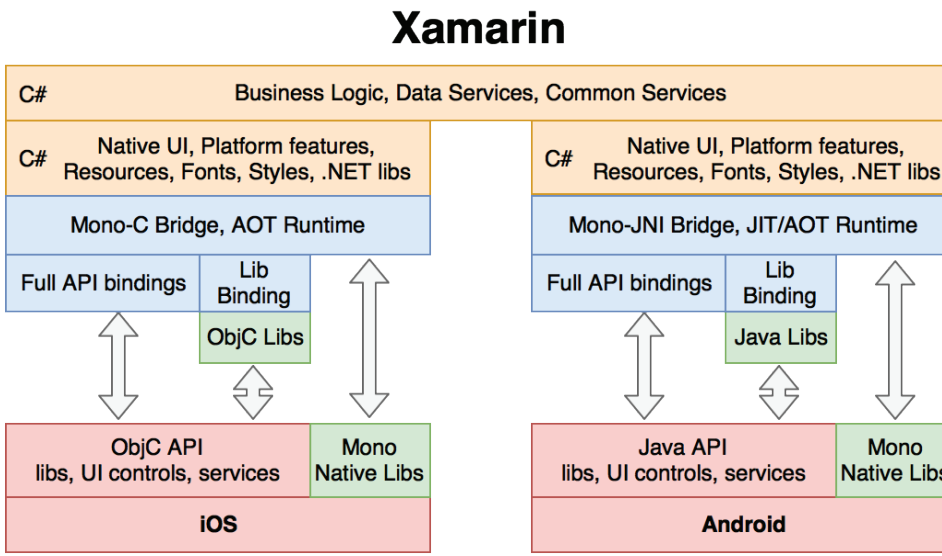

Рис. 1.12 ❖ Архитектура Xamarin

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.

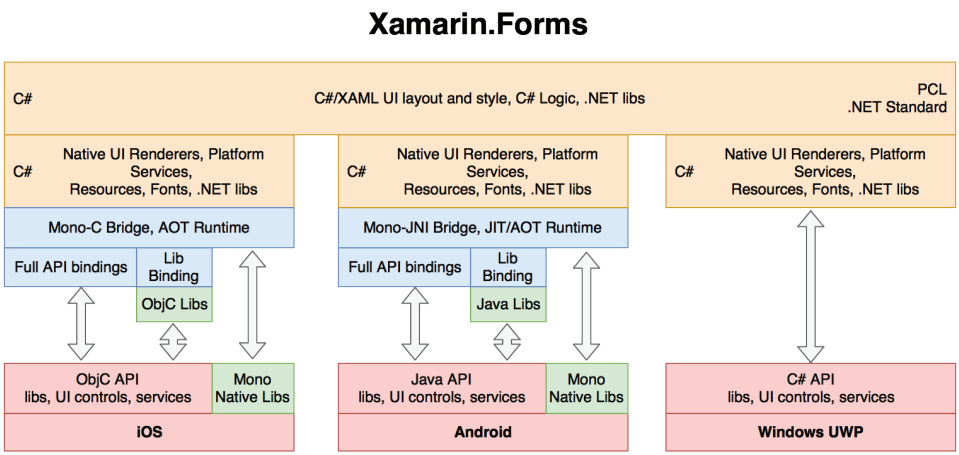

Рис. 1.13 ❖ Архитектура Xamarin.Forms

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.

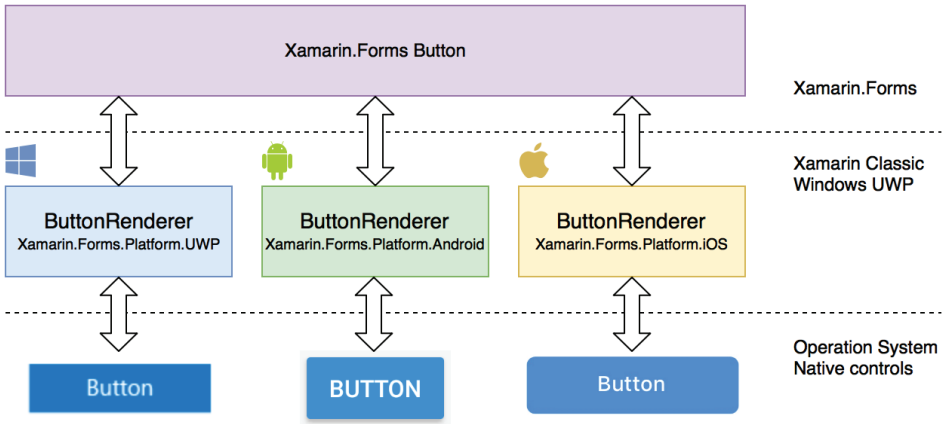

Рис. 1.14 ❖ Нативные контролы в Xamarin.Forms

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.