Магомедов О. Р., Бодренков Г. А., Чернышёв С. А. Архитектура кроссплатформенных фреймворков // Международный журнал информационных технологий и энергоэффективности. – 2022. – Т. 7 № 1(23) с. 31–43

Международный журнал информационных технологий и энергоэффективности

Сайт журнала:
http://www.openaccessscience.ru/index.php/ijcse/

УДК 004.4'2

АРХИТЕКТУРА КРОССПЛАТФОРМЕННЫХ ФРЕЙМВОРКОВ

[1]Магомедов О. Р., [2] Бодренков Г. А., [3,4] Чернышёв С. А.
[1,2,3] Санкт-Петербургский государственный экономический университет, Россия, (191023, г. Санкт-Петербург, ул. Садовая 21), e-mail: [1],smart7even@yandex.ru, [2]glebbodrenkov@gmail.com, [3]chernyshev.s.a@bk.ru.
[4]Санкт-Петербургский государственный университет промышленных технологий и дизайна, Россия (191186, г. Санкт-Петербург, ул. Большая Морская, 18), e-mail: chernyshev.s.a@bk.ru

В статье исследована архитектура кроссплатформенных фреймворков. Рассмотрены концепции, возможности и ограничения архитектур наиболее популярных кроссплатформенных технологий, проведено сравнение архитектурных подходов рассмотренных решений, проанализированы текущие планы по развитию кроссплатформенных технологий.

Ключевые слова: кроссплатформенные технологии, кроссплатформенные фреймворки, архитектура.

ARCHITECTURE OF CROSS-PLATFORM FRAMEWORKS

[1]Magomedov O. R., [2] Bodrenkov G. A., [3,4] Chernyshev S. A.
[1,2,3] *Saint Petersburg state university of economics, Russian Federation, (191023, Saint-Petersburg, Sadovaya str. 21), e-mail: [1] smart7even@yandex.ru, [2] glebbodrenkov@gmail.com,[3] chernyshev.s.a@bk.ru.*
[4] *Saint Petersburg State University of Industrial Technologies and Design, Russian Federation (191186, Saint-Petersburg, Bolshaya Morskaya str. 18), e-mail: chernyshev.s.a@bk.ru.*

The article explores the architecture of cross-platform frameworks. The concepts, capabilities and limitations of the architectures of the most popular cross-platform technologies are considered, the architectural approaches of the considered solutions are compared, current plans for the development of cross-platform technologies are analyzed.

Keywords: cross-platform technologies, cross-platform frameworks, architecture.

Введение

Кроссплатформенные решения становятся все более популярными, так как позволяют разработать приложение, которое будет работать сразу на нескольких платформах. Это особенно важно для электронной коммерции, так как позволяет сократить сроки разработки и численность команды программистов и при этом сохранить поддержку всех широко используемых платформ. Кроссплатформенные фреймворки реализуют сложные архитектуры, и в них необходимо разбираться для того, чтобы понимать возможности и ограничения каждого решения. Многие концепции, используемые при разработке таких фреймворков, могут быть использованы и при разработке архитектур других программных продуктов. К самым популярных кроссплатформенным решениям относятся: Flutter, Xamarin, React Native и Kotlin Multiplatform.

Flutter

Первая версия Flutter SDK увидела свет в конце 2018 года, где Dart занял место основного языка программирования. Он был представлен как кроссплатформенный набор инструментов для разработки приложений с графическим пользовательским интерфейсом различной сложности под iOS и Android, позволяя напрямую взаимодействовать с базовыми сервисами целевых платформ. Постепенно список целевых платформ увеличивался и на данный момент времени разработка может вестись в рамках одной кодовой базы для: iOS, Android, macOS, Windows, Linux и Web. Конечно, это не значит, что код написанный под Android сразу будет работать в Web без доработок, но сама концепция и приложенные к её реализации усилия Google способствуют тому, чтобы пристально присмотреться к этому инструменту.

Архитектурные слои

Архитектура Flutter для всех платформ, за исключением Web, состоит из трех слоев (рисунок 1 [1]):

1. Flutter framework;
2. Flutter engine;
3. Embedder.

Каждый из представленных слоев имеет некоторое количество независимых библиотек, которые расположены на различных уровнях и спроектированы таким образом, что библиотека более верхнего уровня имеет зависимость только от библиотек или библиотеки следующего за ней нижнего уровня. Такой подход делает любую из библиотек заменяемой.

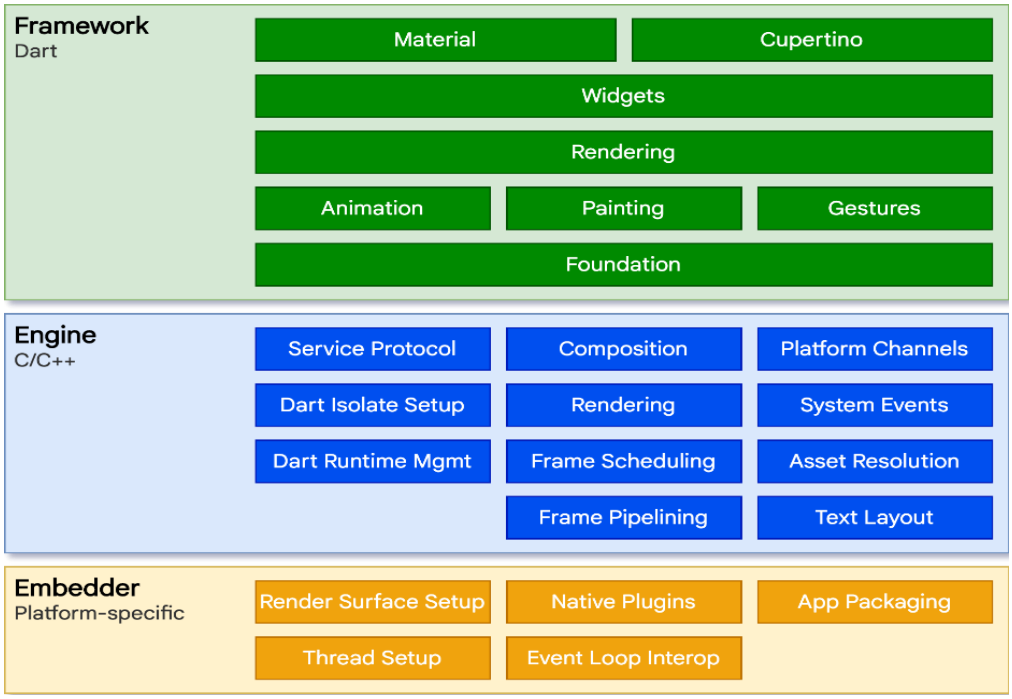

Рисунок 1 – Архитектура Flutter SDK

Слой *Embedder* позволяет упаковывать разрабатываемые приложения на Flutter под различные целевые платформы таким образом, как будто они были написаны на поддерживаемом ими языке программирования. Таким образом, слой *Embedder* обеспечивает для приложения «точку входа» и отвечает за взаимодействие с целевой операционной системой для доступа к её службам (визуализация, ввод/вывод и т.д.) и управляет циклом обработки сообщений. Для каждой из поддерживаемых платформ этот слой написан на подходящем к ней языке программирования:

- Java/Kotlin и C++ для Android;
- Objective-C и Swift для iOS и macOS;
- C++ для Windows и Linux.

Именно благодаря слою *Embedder* разрабатываемое на Flutter приложение может быть как самостоятельным, так и интегрироваться в качестве модуля в нативные приложения.

Слой *Flutter engine* в основном написан на C++ и обеспечивает низкоуровневую реализацию API Flutter: работа с графикой (графический движок Skia), операции ввода-вывода, межсетевое взаимодействие и т.д., а также отвечает за поддержку ряда специальных возможностей, среду выполнения и компиляции Dart. Именно благодаря собственному графическому движку Skia, который отвечает за отрисовку пользовательского интерфейса на любой из платформ, Flutter SDK не пошел по стопам React Native. То есть все виджеты отрисовываются средствами Flutter и нет никаких дополнительных прослоек, что сказывается на плавности их отрисовки, которое не уступает нативному.

Чаще всего разработчики и не подозревают о наличии рассмотренных ранее слоев и взаимодействуют только со слоем *Flutter framework*, написанным на Dart. Он включает в себя довольно богатый набор базовых библиотек, расположенных на различных уровнях текущего слоя. Давайте перечислим существующие уровни слоя *Flutter framework*, начиная с самого нижнего:

- Уровень *Foundational* предоставляет базовые классы и функции, которые используются для создания приложения, а также содержит API-интерфейсы для связи со слоем *Flutter engine*.
- Уровень *Rendering* обеспечивает абстракцию для работы с компоновкой виджетов, позволяет построить дерево визуализируемых объектов и динамически управлять ими, что автоматически отразится на структуре дерева.
- Уровень *Widgets* представляет собой композицию абстракций и модель реактивного программирования. Так, например, у каждого объекта на уровне рендеринга имеется соответствующий класс на уровне *Widgets*, что позволяет определять повторно используемые комбинации классов.
- Библиотеки *Material* и *Cupertino* предоставляют разработчику наборы элементов управления, которые используют примитивы композиции уровня *Widgets* для реализации виджетов в стиле Material или iOS.

Как в Dart существует концепция, что все является объектом, так и во Flutter имеется своя концепция: все является виджетом. То есть графический пользовательский интерфейс разрабатываемых приложений полностью состоит из виджетов и их различной компоновки. Каждому виджету соответствует свой элемент на уровне *Rendering*, в соответствии с чем на этапе сборки Flutter переводит виджеты, которые используются в коде, в соответствующее дерево элементов (рисунок 2 [1]):

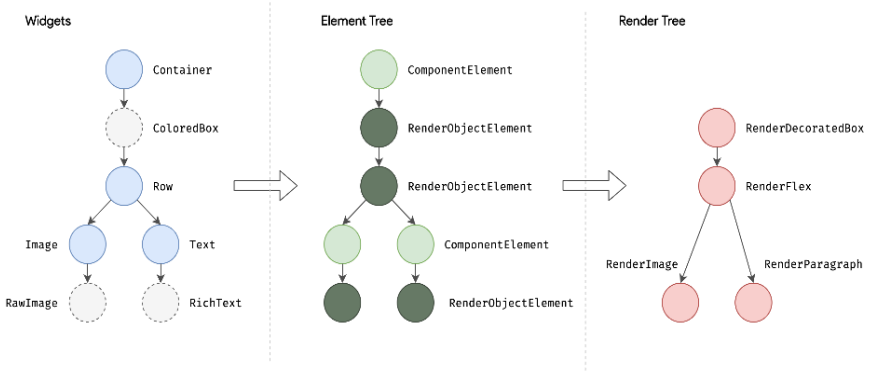

Рисунок 2 – Приведение дерева виджетов к дереву рендеринга

Дерево виджетов отвечает за конфигурирование, а именно: декларативное описание пользовательского интерфейса и хранение свойств виджетов. Дерево элементов отвечает за управление, то есть элементы управляют жизненным циклом виджетов, а также осуществляют их связывание в древовидную иерархию и с объектами рендеринга. Дерево рендеринга отвечает за отрисовку виджетов с учетом их положения и ограничений.

Архитектура Web

Еще с момента своего появления Dart компилировался в JavaScript из-за чего портирование Flutter для Web пошло по другому пути. Это связано с тем, что слой *Flutter engine* написан на C++ и ориентирован на взаимодействие с операционной системой, а не браузером (рисунок 3 [1]).

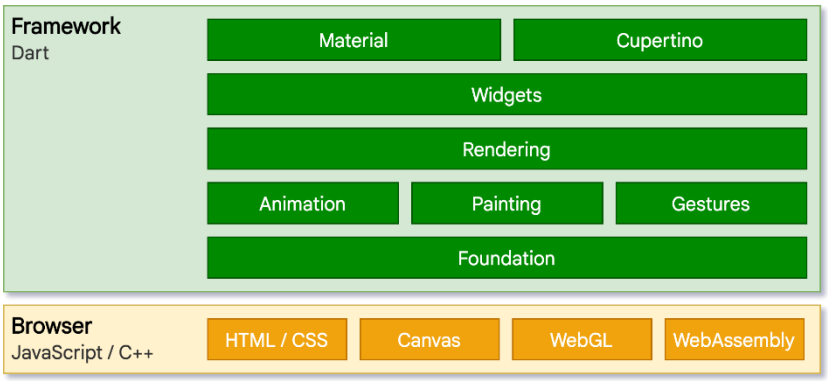

Рисунок 3 – Архитектура Flutter Web

Flutter Web обеспечивает повторную реализацию слоя *Flutter engine* поверх стандартных API-интерфейсов браузера. В связи с этим существует два способа рендеринга содержимого Flutter для web: HTML и WebGL. В первом случае Flutter использует HTML, CSS, Canvas и SVG. А для рендеринга в WebGL Flutter использует версию Skia (CanvasKit), скомпилированную в WebAssembly.

Flutter: Итог

Таким образом, можно утверждать, что архитектура Flutter хорошо спроектирована. Она позволяет улучшать фреймворк и добавлять поддержку любых платформ. Архитектура разделена на слои, что позволяет однозначно установить иерархию зависимостей и сделать компоненты фреймворка заменяемыми. К минусу можно отнести использование Skia, потому что пользовательский интерфейс, хоть и мимикрирует под интерфейс нативных платформ, но в некоторых деталях заметно отличается от него. Фреймворки, рассмотренные далее используют нативные элементы пользовательского интерфейса, но в связи с этим вынуждены реализовывать более сложную архитектуру для установления двусторонней связи между фреймворком и нативной платформой.

Xamarin

Xamarin — это платформа с открытым исходным кодом, предназначенная для построения современных производительных приложений для iOS, Android и Windows с .NET. [2] Платформа Xamarin представляет собой уровень абстракции, который обеспечивает взаимодействие между общим кодом и кодом нативной платформы.

Благодаря Xamarin в среднем 90 % кода приложения может использоваться без изменений на разных платформах. Разработчик может написать всю бизнес-логику на одном языке, а также получить близкую к нативной производительность приложения и графический пользовательский интерфейс, которые характерны для целевой платформы.

Платформа Xamarin ориентирована на разработчиков, перед которыми стоят следующие задачи:

- Совместное использование кода, тестов и бизнес-логики на различных платформах;
- Написание кроссплатформенных приложений на языке C#.

Архитектура

Структуру платформы Xamarin можно представить следующим образом (рисунок 4 [2])

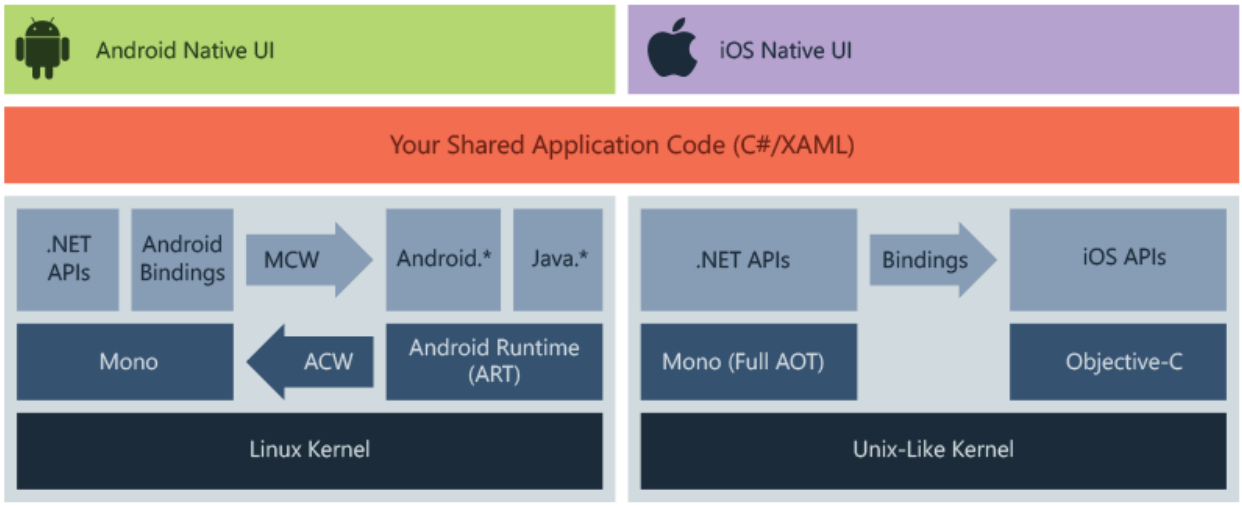

Рисунок 4 – Архитектура Xamarin

Xamarin работает поверх фреймворка Mono, который предоставляет opensource-реализацию .NET Framework [3] и может работать с разными платформами, такими как Linux, MacOS и т.д.

На уровне каждой отдельной платформы Xamarin полагается на ряд субплатформ. В частности:

- Xamarin.Android - библиотеки для создания приложений на ОС Android;
- Xamarin.iOS - библиотеки для создания приложений для iOS.

Через них приложения могут направлять запросы к прикладным интерфейсам на устройствах под управлением ОС Android или iOS.

Xamarin.Android

Приложения Xamarin.Android компилируются из языка C# в промежуточный язык, который при запуске приложения проходит Just-in-Time-компиляцию (рисунок 5 [2]). Приложения Xamarin.Android работают в среде выполнения Mono параллельно с виртуальной машиной среды выполнения Android (ART). Xamarin предоставляет привязки .NET к пространствам Android и Java, а также обращается к этим пространствам с использованием управляемых оболочек (ACW) и предоставляет ART их Android (ACW), благодаря чему обе среды могут вызывать код друг друга.

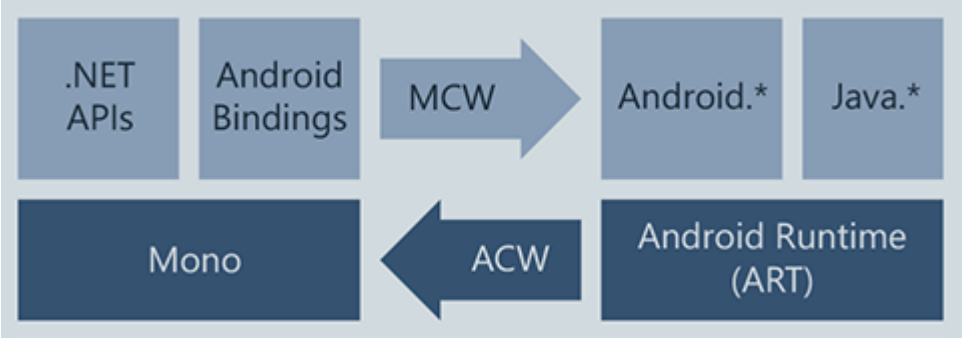

Рисунок 5 – Архитектура Xamarin.Android

Xamarin.iOS

Код приложений Xamarin.iOS собирается под целевую платформу посредством Ahead-of-Time-компиляции из языка C#. Xamarin использует селекторы, чтобы предоставить Objective-C управляемому коду C# и управляемый код C# для Objective-C. Селекторы и регисторы в совокупности называются "привязками" и обеспечивают взаимодействие между Objective-C и C# (рисунок 6 [2]).

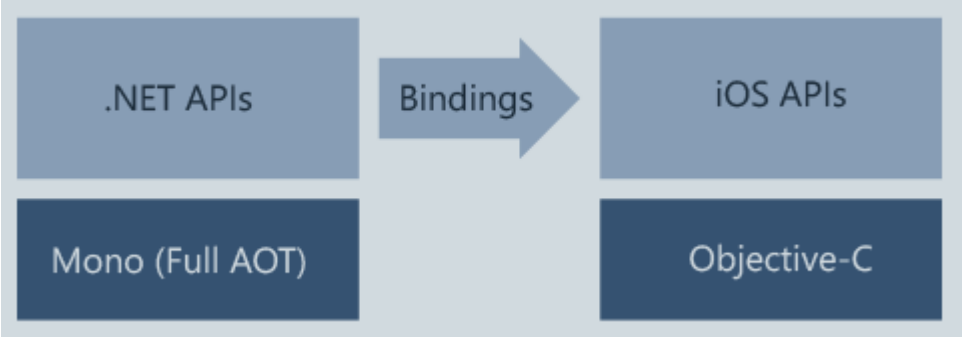

Рисунок 6 – Архитектура Xamarin.iOS

В итоге благодаря этим платформам можно отдельно создавать приложения для Android или iOS. Наиболее важной особенностью Xamarin является возможность создавать кроссплатформенные приложения - одна логика на все платформы. Для этого используется технология Xamarin.Forms, которая является слоем абстракцией над Xamarin.Android и Xamarin.iOS. С помощью Xamarin.Forms определяется визуальный интерфейс и выполняется привязка к нему бизнес-логики на C#, которая одинаково будет работать на Android, iOS и Windows.

.NET MAUI

.NET Multi-platform App UI (.NET Кроссплатформенное приложение пользовательского интерфейса) — это кроссплатформенный фреймворк, позволяющий создавать мобильные и компьютерные приложения при помощи C# и XAML [3]. В данный момент (февраль 2022 года) он находится в активной разработке и доступна версия Preview 12. Позиционируется как “эволюция Xamarin.Forms” и поддерживает миграцию приложения из этого фреймворка с небольшими изменениями в коде. Оба фреймворка похожи друг на друга, но есть одно главное отличие - в одном проекте .NET MAUI можно будет создавать кроссплатформенные приложения и добавлять специфический для платформы код непосредственно там, где это необходимо. Одна из основных целей этого фреймворка - реализовать как можно больше исходной логики приложения и макета пользовательского интерфейса в основном коде, еще больше, чем позволяет Xamarin.Forms.

Xamarin: Итог

Xamarin позволяет довольно просто создавать, поддерживать и обновлять кроссплатформенное программное обеспечение, при этом оставляя сравнимую с нативной производительность. Главными минусами Xamarin на данный момент являются:

- желательное наличие начальных знаний в языках нативных платформ для успешного использования возможностей каждой из них;
- чаще всего, приложения на Xamarin весят больше, чем приложения на нативных платформах. На рисунке 7 показано, сколько объема памяти занимает приложение, выводящее «Hello World»;
- Xamarin.Forms в ближайшее время будет заменено .NET MAUI.

Рисунок 7 – Распределение объема памяти приложения «Hello World»

React Native

React Native – кроссплатформенный фреймворк с открытым исходным кодом, разработанный Facebook. Фреймворк поддерживает все популярные платформы: Android, Android TV, iOS, macOS, tvOS, Web, Windows. Изучение фреймворка React Native будет несложным для фронтенд разработчиков, в особенности для тех, кто использовал в разработке web-фреймворк React.

Архитектура

Фундаментальным понятием в архитектуре React является Virtual DOM (виртуальное дерево элементов пользовательского интерфейса) - это абстракция, которая держит представление UI в памяти в виде дерева компонентов и синхронизирует его с пользовательским интерфейсом.

Каждое React Native приложение работает в трёх потоках [4]. Поток пользовательского интерфейса обрабатывает отображение элементов пользовательского интерфейса и жесты пользователя. JavaScript поток отвечает за управление состоянием приложения: получение, передачу и обработку данных - и построение Virtual DOM при изменении состояния приложения. Далее JavaScript поток передает его Shadow потоку, который работает в фоновом режиме и отвечает за вычисление изменений пользовательского интерфейса и уведомление потока пользовательского интерфейса о том, когда и где нужна перерисовка пользовательского интерфейса. Именно введение такой абстракции как Virtual DOM позволяет отслеживать те части пользовательского интерфейса, в которых нужны изменения, и производить только их перерисовку, не перерисовывая приложение целиком. React Native предоставляет набор абстрактных компонентов пользовательского интерфейса (View), которые преобразуются в нативные элементы каждой платформы при компиляции приложения. А также предоставляет универсальный API взаимодействия с такими внешними модулями как: сеть, файловая система и т.д.

Архитектура, представленная в React Native, имеет ряд преимуществ:

1. Она позволяет осуществить разделение процесса обновления пользовательского интерфейса на два этапа: reconciliation (согласование) и rendering (отрисовка). Такое разделение дает возможность хранить представление пользовательского интерфейса в памяти и при изменении состояния приложения на этапе reconciliation сравнивать предыдущее и текущее состояния и посылать модулю отрисовки команды на обновление только тех частей пользовательского интерфейса, для которых изменилось состояние. Такой механизм уже применялся в веб фреймворке React и был перенесен в React Native.
2. Она предусматривает использование нативных для платформ элементов графического интерфейса, но отделение этапов rendering и reconciliation позволяет также создать свой движок для управления отрисовкой пользовательского интерфейса, что

продемонстрировано в реализации библиотеки react-native-skia [5], которая позволяет использовать графический движок skia. Библиотека находится на стадии Proof of Concept (Доказательства концепции), но сам тот факт, что архитектура React Native настолько гибка, что позволяет подменять реализации модулей для отрисовки пользовательского интерфейса, неоспорим.

3. Использование JavaScript как языка программирования позволяет использовать API веб фреймворка React и JavaScript библиотек (менеджеров состояний, таких как Redux и Mobx, и других вспомогательных библиотек).

React Native: Итог

React Native стремительно развивается и постепенно решает архитектурные проблемы. Еще недавно к минусам архитектуры можно было отнести использование модуля bridge для взаимодействия потоков, что ограничивало быстродействие приложений, написанных на React Native. Сообщения для обмена информацией между потоками передавались через bridge в формате JSON. Это добавляло необходимость сериализовать и десериализовать данные, что замедляло приложение. Новый отрисовщик интерфейса улучшает обмен данными с помощью доступа к состоянию JavaScript потока напрямую используя JavaScript интерфейсы (JSI).

Тем не менее «узким местом» React Native до сих пор остается однопоточность JavaScript, что затрудняет реализацию сложных вычислений на устройстве клиента. Запуск сложного вычисления приведет к зависанию пользовательского интерфейса. Обход данного ограничения состоит в использовании процессов, но такое решение требует реализовывать межпроцессное взаимодействие, так как память процессов по умолчанию изолирована. Положительным моментом является то, что сообщество разрабатывает библиотеку для межпроцессного взаимодействия react-native-multithreading [6].

Kotlin Multiplatform

Kotlin Multiplatform (KM) [7] – это технология, позволяющая писать приложения под разные платформы с использованием Kotlin, разработанная компанией JetBrains. Kotlin Multiplatform включает в себя два фреймворка: Kotlin Multiplatform Mobile (KMM) [8] и Compose Multiplatform [9]. KMM создан для мобильной разработки под iOS и Android, а Compose Multiplatform включает поддержку Desktop платформ (Windows, macOS, Linux) и Web. Кроссплатформенная разработка с использованием Kotlin Multiplatform устроена таким образом, что часть приложения, которая не зависит от платформы, пишется на Kotlin, а пользовательский интерфейс пишется на нативном языке платформы с использованием фреймворков Compose Multiplatform и Kotlin Multiplatform Mobile (рисунок 8 [7]). К общему коду относится взаимодействие с сетью, сериализация и десериализация данных, бизнес-логика, модели данных, хранение данных, состояние приложения и т. д.

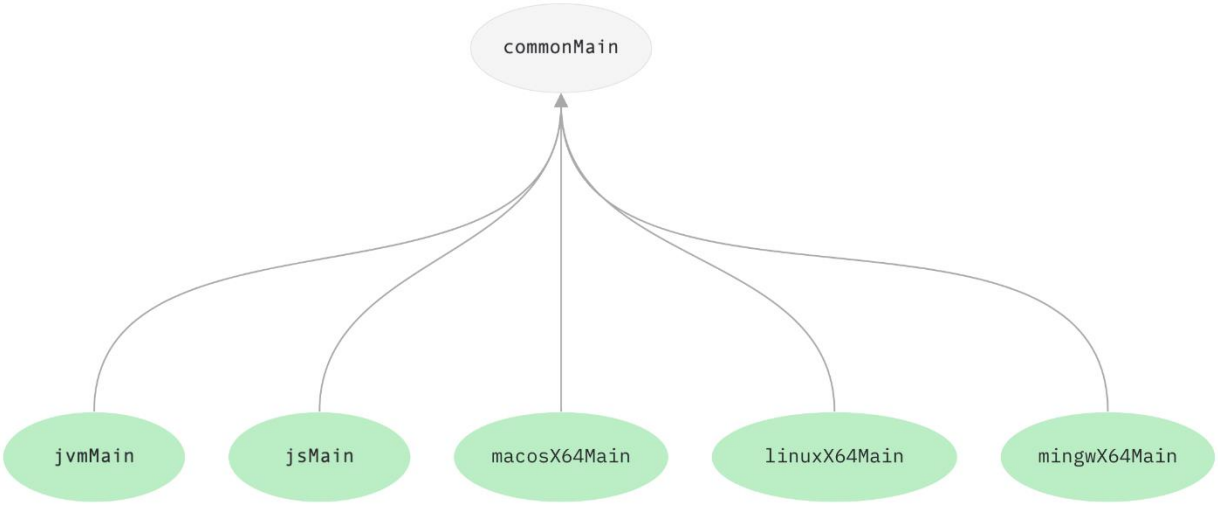

Рисунок 8 – Разделение кода приложения на общую и платформозависимую логику

За счет использования нативных элементов пользовательского интерфейса скорость работы приложений, написанных с использованием фреймворков, близка к скорости полностью нативных приложений. Механизм реализации платформозависимой логики устроен через интерфейсы. Объявляются функции, классы, интерфейсы, перечисления и свойства, которые должны быть реализованы для каждой платформы отдельно. Затем они реализуются для каждой платформы с использованием платформенных версий Kotlin, которые имеют доступ к нативному коду платформы, что позволяет использовать все нативные возможности (рисунок 9 [7]).

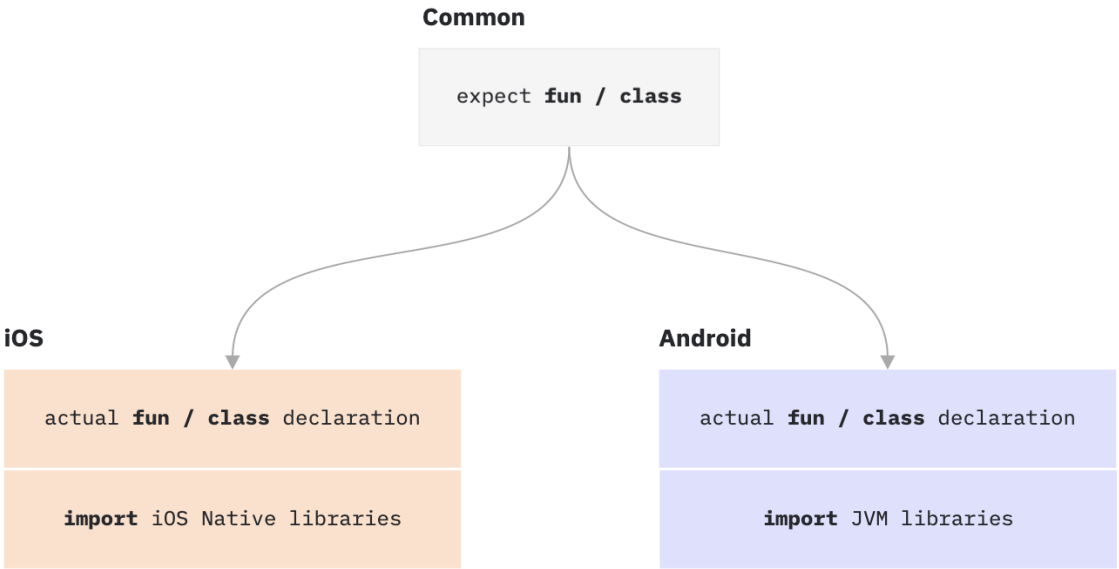

Рисунок 9 – Объявление и реализация платформозависимого интерфейса приложения

При запуске и сборке приложения с использованием Kotlin для Web, код на Kotlin транспилируется в JavaScript транспилятором Kotlin/JS. При использовании Kotlin для Android код компилируется в байт-код, исполняемый далее на Java Virtual Machine. При использовании Kotlin для других платформ (macOS, iOS, tvOS, watchOS, Linux, Windows) код приложений компилируется в нативные бинарные файлы, которые запускаются без виртуальной машины (рисунок 10 [7]).

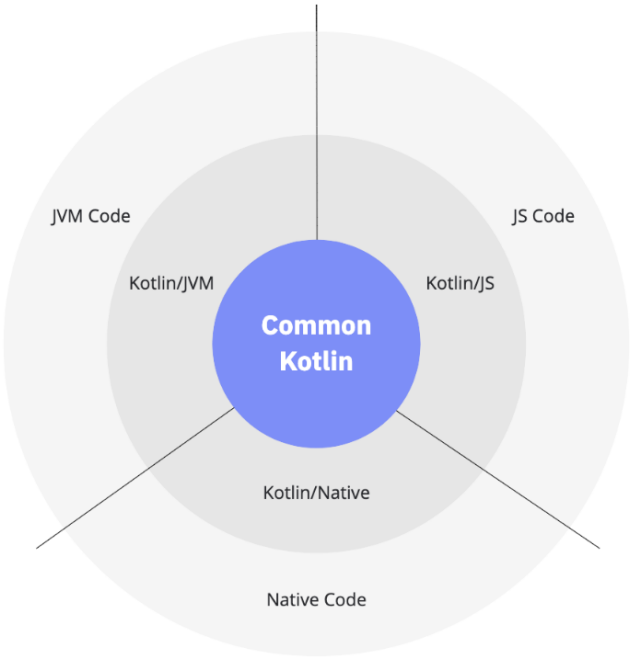

Рисунок 10 – Транспиляторы и компиляторы Kotlin

Kotlin Multiplatform: Итог

Являясь достаточно молодой технологией, Kotlin Multiplatform представил свое видение кроссплатформенности, проводя границу между пользовательским интерфейсом и бизнес-логикой приложения. Логика приложения выступает ядром, вокруг которого добавляется поддержка платформ с помощью реализации пользовательского интерфейса посредством двух технологий - Kotlin Multiplatform Mobile и Compose Multiplatform. Большим плюсом KM является его компилируемость в нативный код платформ, что дает возможность не включать виртуальную машину исполнения Kotlin в релизную сборку приложения.

Сравнение исследуемых фреймворков

У Представленных выше фреймворки есть как общие концепции, так и различия.

К общему подходу, использующемуся во всех представленных выше фреймворках, относится использование деревьев для построения пользовательского интерфейса. Фреймворки используют их для оптимизации рендеринга, что позволяет обновлять пользовательский интерфейс только тех компонентов, для которых изменилось состояние.

Необходимо выделить, что у фреймворков разный подход к пользовательскому интерфейсу. React Native, Xamarin и Kotlin Multiplatform используют нативные компоненты

для отрисовки пользовательского интерфейса, в то время как Flutter использует свой 2D движок Skia.

Также различны и способы взаимодействия с нативной платформой. Kotlin Multiplatform компилируется в нативный код, а Flutter, React Native и Xamarin запускаются в отдельном ядре и коммуницируют с ядром пользовательского интерфейса для предоставления ему состояния приложения и обработки событий пользовательского интерфейса. В связи с этим в сборку приложений данных фреймворков входят среды исполнения языков программирования, для Flutter это среда исполнения Dart, React Native - JavaScript, Xamarin - C#. Как правило среды исполнения занимают много места в памяти, поэтому сборка таких приложений становится больше.

Следует отметить, что первая стабильная версия Compose Multiplatform вышла только 12 декабря 2021 года, поэтому принимать решение об использовании данного фреймворка для больших проектов нужно с осторожностью, в то время как Xamarin, Flutter и React Native уже «прижившиеся» технологии, которые используются в продуктах ряда больших компаний.

Отметим развитие фреймворков. Flutter и React постепенно развиваются, выпуская новые версии, которые улучшают производительность, удобство разработки, устраняют ошибки. А Xamarin помимо этого эволюционирует в MAUI, что позволяет добавить поддержку десктопных операционных систем. Эта эволюция - хороший шаг в развитии фреймворка, но в то же время рассматривать Xamarin как фреймворк для написания приложения с нуля можно, если вы уверены, что сможете без особых проблем мигрировать приложение с Xamarin в MAUI, когда выйдет стабильная версия MAUI.

Вывод

Таким образом, все кроссплатформенные фреймворки реализуют сложные архитектуры для того, чтобы использовать общую кодовую базу для нескольких платформ, но при этом предоставлять доступ к нативным библиотекам и интерфейсам, таким как: взаимодействие с сетью, контакты, файловая система, уведомления и так далее. Из проведенного исследования можно заключить, что кроссплатформенные фреймворки реализуют гибкую архитектуру для того, чтобы поддерживать различные платформы и добавлять поддержку новых при необходимости. В то же время архитектура ориентирована на достижение лучшей производительности решения и удобства разработки.

Список литературы

1. Flutter architectural overview [Электронный ресурс]. – Режим доступа: https://docs.flutter.dev/resources/architectural-overview
2. Что такое Xamarin [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/xamarin/get-started/what-is-xamarin
3. .NET Multi-platform App UI documentation [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/maui/
4. Обзор архитектуры React Native [Электронный ресурс]. – Режим доступа: https://reactnative.dev/docs/architecture-overview
5. Библиотека React Native Skia [Электронный ресурс]. – Режим доступа: https://github.com/react-native-skia/react-native-skia

6. Библиотека React Native Multithreading [Электронный ресурс]. – Режим доступа: https://github.com/mrousavy/react-native-multithreading
7. Kotlin Multiplatform [Электронный ресурс]. – Режим доступа: https://kotlinlang.org/docs/multiplatform.html
8. Kotlin Multiplatform Mobile [Электронный ресурс]. – Режим доступа: https://kotlinlang.org/lp/mobile/
9. Compose Multiplatform [Электронный ресурс]. – Режим доступа: https://www.jetbrains.com/lp/compose-mpp/

References

1. Flutter architectural overview [Electronic resource]. – Access mode: https://docs.flutter.dev/resources/architectural-overview
2. What is Xamarin [Electronic resource]. – Access Mode: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin
3. .NET Multi-platform App UI documentation [Electronic resource]. – Access Mode: https://docs.microsoft.com/en-us/dotnet/maui/
4. Overview of React Native architecture [Electronic resource]. - Access mode: https://reactnative.dev/docs/architecture-overview
5. Library React Native Skia [Electronic resource]. – Access mode: https://github.com/react-native-skia/react-native-skia
6. Library React Native Multithreading [Electronic resource]. - Access mode: https://github.com/mrousavy/react-native-multithreading
7. Kotlin Multiplatform [Electronic resource]. – Access Mode: https://kotlinlang.org/docs/multiplatform.html
8. Kotlin Multiplatform Mobile [Electronic resource]. – Access Mode: https://kotlinlang.org/lp/mobile/
9. Compose Multiplatform [Electronic resource]. – Access Mode: https://www.jetbrains.com/lp/compose-mpp/