Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
НА ТЕМУ:
«Разработка мультиплатформенного клиентского приложения»

Студент ИУ9-82Б (Группа)	(Подпись, дата)	А.Г. Владиславов (И.О. Фамилия)
Руководитель ВКР	(Подпись, дата)	Д.П. Посевин (И.О. Фамилия)
Консультант	(Подпись, дата)	(И.О. Фамилия)
Консультант	(Подпись, дата)	(И.О. Фамилия)
Нормоконтролер	(Подпись, дата)	(И.О. Фамилия)

2023 г.

ЗАДАНИЕ стр. 1

Печатается отдельно, номер не проставляется

ЗАДАНИЕ стр. 2

Печатается отдельно, номер не проставляется

КАЛЕНДАРНЫЙ ПЛАН

Печатается отдельно, номер не проставляется

АННОТАЦИЯ

Темой данной работы является «Рарзаботка мультиплатформенного клиентского приложения». Объем данной работы составляет 70 страниц.

Основной объект исследования — разбор существующих технологий для разработки приложений под различные платформы. В практической части работы рассматривается реализация соответствующего приложения на основе одной из рассмотренных технологий.

Данная работа состоит из 4 глав. Первая глава посвящена разбору наиболее популярных на данный момент техлонолий. Во второй главе более подробно описана выбранная для написания кода технология. В третьей главе описан процесс разработки приложения с использованием выбранной технологии. В четвертой главе описан процесс тестирования и достигнутые результаты.

Работа содержит 36 листингов и 5 рисунков.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современных задачах создания мобильных приложений становится все более актуальным использование фреймворков для мультиплатформенной разработки. Многие компании сталкиваются с задачей выбора подходящей технологии для создания мультиплатформенных продуктов. В данной работе представлен обзор существующих технологий для кросс-платформенной разработки и анализ их основных преимуществ и недостатков. Особое внимание уделяется технологии Kotlin Multiplatform, выбранной в качестве основы для разработки мобильного приложения.

В первой части работы представлен краткий обзор предметной области и существующих технологий, таких как нативная разработка, Xamarin, React Native, Flutter и Kotlin Multiplatform. Описаны причины выбора Kotlin Multiplatform для разработки приложения.

Во второй части исследуется технология Kotlin Multiplatform. Описываются основные характеристики языка Kotlin, его возможности в контексте разработки для различных платформ, а также процесс компиляции под разные целевые платформы.

Третья часть работы посвящена разработке мультиплатформенного приложения с использованием выбранной технологии. Рассматриваются настройка проекта, подключение библиотек, а также написание "общего" и платформенного кода. Описывается процесс интеграции кода на Kotlin Multiplatform в проекты для Android и iOS, подключение сетевых запросов и баз данных, а также выбор подходящего фреймворка для создания пользовательского интерфейса.

Таким образом, данная работа представляет собой комплексное исследование технологий для разработки мультиплатформенных приложений и практическую реализацию одного из таких приложений с использованием технологии Kotlin Multiplatform. Результаты работы могут быть полезными для специалистов в области разработки мобильных приложений, а также компаний, стоящих перед выбором технологии для своих проектов.

1 Обзор предметной области

В данной главе будет проведён обзор предметной области мультиплатформенной разработки, а именно разработки приложений, которые могут работать на разных платформах без значительных изменений в исходном коде. Сначала будет произведено ознакомление с основами мультиплатформенной разработки и рассмотрим ключевые преимущества и недостатки такого подхода (раздел 1.1).

Далее будет проведён анализ популярных технологий, используемых в мультиплатформенной разработке (раздел 1.2).

В заключительной части главы (раздел 1.3) будут подведены итоги, и будет выберана технология для дальнейшего разбора в рамках данной работы.

1.1 Введение в мультиплатформенную разработку

В современном мире разработки мобильных приложений одной из ключевых задач является создание программного продукта, который будет работать на различных платформах и устройствах. Это стало возможным благодаря появлению мультиплатформенных технологий и инструментов, которые позволяют разрабатывать приложения с использованием единого кода для нескольких операционных систем, таких как iOS и Android.

Разработка под множество платформ предоставляет целый ряд преимуществ. Во-первых, она позволяет сократить время разработки, так как разработчикам не нужно создавать отдельные версии приложения для каждой платформы. Во-вторых, такой подход снижает затраты на разработку и поддержку, так как требуется меньше ресурсов для создания и обновления приложений. В-третьих, мультиплатформенные приложения позволяют охватить более широкую аудиторию, так как они могут работать на разных устройствах и операционных системах.

Однако стоит учесть, что разработка под множество платформ может иметь свои недостатки, такие как возможные компромиссы в производительности и доступности некоторых платформенных возможностей. Поэтому выбор подходящей технологии является критически важным этапом в процессе разработки мультиплатформенного приложения.

В последние годы появилось множество технологий, предназначенных для кросс-платформенной разработки, каждая из которых имеет свои особенности и

преимущества. В данной работе будет проведен обзор пяти основных технологий, которые сегодня активно используются разработчиками:

1. Нативная разработка — это создание приложений с использованием языков и инструментов, специфических для каждой платформы. Этот подход обеспечивает наилучшую производительность и доступ ко всем платформенным возможностям, однако требует значительных ресурсов и времени на разработку отдельных приложений для каждой платформы.
2. Xamarin — это инструмент для кросс-платформенной разработки на языке C#, который позволяет создавать приложения для iOS, Android и Windows с использованием единого кода. Xamarin обеспечивает хорошую производительность и доступ к платформенным возможностям, но может потребовать дополнительных усилий для адаптации пользовательского интерфейса под каждую платформу.
3. React Native — это фреймворк для разработки мобильных приложений на основе JavaScript и React, который позволяет создавать нативные приложения с использованием общего кода для iOS и Android. React Native предлагает быструю разработку и упрощенную интеграцию с веб-технологиями, однако может иметь некоторые ограничения в производительности и доступности платформенных возможностей.
4. Flutter — это фреймворк от Google для разработки мультиплатформенных приложений с использованием языка Dart. Он предлагает высокую производительность, быструю разработку и консистентный пользовательский интерфейс благодаря своему собственному графическому движку, который рендерит интерфейс независимо от платформы. Однако Flutter имеет свои ограничения, в частности, он может не поддерживать некоторые специфичные платформенные возможности, и разработчику может потребоваться время на изучение языка Dart.
5. Kotlin Multiplatform — это технология от JetBrains, которая позволяет использовать одну кодовую базу на языке Kotlin для создания мультиплатформенных приложений. Этот подход обеспечивает хорошую производительность и доступ к платформенным возможностям, а также позволяет разработчикам использовать преимущества языка Kotlin для написания кросс-платформенного кода. Однако Kotlin Multiplatform является до-

вольно молодой технологией и может иметь ограниченную экосистему библиотек и инструментов по сравнению с другими технологиями.

В данной работе будет проведен подробный обзор каждой из этих технологий, с акцентом на их особенности, преимущества и недостатки. После изучения этих технологий, разработчики смогут принять обоснованное решение о выборе подходящей технологии для своего мультиплатформенного проекта.

Цель данной работы заключается в том, чтобы дать общее представление о современных подходах и технологиях кросс-платформенной разработки, а также помочь разработчикам определиться с наиболее подходящим инструментом для конкретных проектов, исходя из требований к производительности, доступности платформенных возможностей и других факторов.

1.2 Обзор технологий

В рамках данной части будут рассмотрены наиболее популярные инструменты для разработки мультиплатформенных приложений: нативная разработка (раздел 1.2.1), Xamarin (раздел 1.2.2), React Native (раздел 1.2.3), Flutter (раздел 1.2.4), Kotlin Multiplatform Mobile (раздел 1.2.5).

1.2.1 Нативная разработка

Нативная разработка предполагает создание приложений с использованием официальных инструментов, языков программирования и библиотек, предоставляемых разработчиками мобильных операционных систем, таких как iOS и Android. В этом контексте рассмотрим нативную разработку для каждой из этих платформ отдельно.

1. Разработка под Android [1]. Нативная разработка для Android включает использование языков программирования Java и Kotlin, а также инструментов, предоставляемых Google. Java был основным языком разработки для Android с момента его создания, но в последнее время Kotlin стал все более популярным благодаря своей лаконичности, современным функциям и обратной совместимости с Java. В 2017 году Google объявил Kotlin официальным языком разработки для Android.

Android Studio является официальной средой разработки (IDE) для создания нативных приложений под Android. Она предоставляет разработчикам доступ ко всем инструментам, необходимым для проектирования, разработки, тестирования и отладки приложений для Android-устройств.

2. Разработка под iOS [2]. Нативная разработка для iOS включает использование языков программирования Swift и Objective-C, а также инструментов, предоставляемых Apple. Swift является современным и мощным языком программирования, разработанным Apple специально для создания приложений под iOS, macOS, watchOS и tvOS. Objective-C — это более старый язык, который использовался для разработки приложений под iOS до появления Swift, и до сих пор поддерживается Apple.

   Xcode является официальной средой разработки (IDE) для создания нативных приложений под iOS. Она предоставляет разработчикам доступ ко всем инструментам, необходимым для проектирования, разработки, тестирования и отладки приложений для Apple-устройств.

Нативная разработка имеет ряд преимуществ, таких как высокая производительность, оптимальное использование платформенных возможностей и лучший пользовательский опыт благодаря применению стандартных элементов пользовательского интерфейса и поведения, специфичных для каждой платформы.

Однако нативная разработка также имеет некоторые недостатки, среди которых следует выделить следующие:

1. Дублирование кода. Разработка отдельных приложений для iOS и Android может привести к дублированию кода, особенно если приложения имеют схожую функциональность на обеих платформах. Это может увеличить время разработки и затраты на поддержку приложений.
2. Более высокая стоимость разработки и поддержки. Нативная разработка требует наличия разработчиков, специализирующихся на каждой платформе, что может привести к большему количеству затрат на зарплаты и обучение, а также увеличению времени на обновление и поддержку приложений.
3. Сложность синхронизации функций и исправления ошибок. Поскольку приложения разрабатываются на разных языках программирования и используют разные библиотеки, синхронизация новых функций и исправ-

ление ошибок может быть сложным процессом. Разработчикам нужно уделять больше времени на координацию между командами и проверку того, что изменения в одной версии приложения не вызывают проблем в другой версии.

4. Медленное внедрение новых технологий. В силу того, что нативные приложения тесно связаны с конкретной платформой, разработчики могут столкнуться с ограничениями, когда дело доходит до внедрения новых технологий или адаптации к изменениям на рынке. Это может замедлить инновационный процесс и снизить конкурентоспособность приложения.

В связи с вышеуказанными недостатками нативной разработки, многие разработчики и компании начали искать альтернативные решения для создания мультиплатформенных приложений. Одним из таких решений является кросс-платформенная разработка, которая предлагает разработчикам возможность использовать один и тот же код для создания приложений, работающих на разных платформах.

1.2.2 Xamarin

Xamarin[3, 4] — это платформа кросс-платформенной разработки, созданная компанией Xamarin, которую впоследствии приобрела Microsoft. Xamarin позволяет разработчикам создавать мобильные приложения для iOS, Android и Windows с использованием единой кодовой базы на языке программирования C# и .NET-фреймворка.

Основные особенности и преимущества Xamarin:

1. Общий код. Xamarin использует общую кодовую базу для бизнес-логики и частично для пользовательского интерфейса, что позволяет разработчикам снизить дублирование кода и упростить поддержку приложений на разных платформах.
2. Производительность. Xamarin обеспечивает близкую к нативной производительность, так как использует платформенно-специфичные элементы пользовательского интерфейса и обращается к нативным API для доступа к возможностям устройства.
3. Интеграция с Visual Studio. Xamarin тесно интегрирован с Visual Studio, популярной средой разработки от Microsoft, что позволяет разработчи-

кам использовать знакомые инструменты и рабочие процессы. Xamarin также доступен для пользователей Visual Studio for Mac, обеспечивая поддержку разработки на macOS.

4. Обширная библиотека компонентов. Xamarin предоставляет разработчикам доступ к богатой библиотеке компонентов, которые облегчают реализацию различных функций приложения и интеграцию с внешними сервисами.
5. Поддержка Xamarin.Forms [5]. Xamarin.Forms —- это дополнительный фреймворк для создания пользовательского интерфейса, который позволяет разработчикам создавать общий пользовательский интерфейс для iOS, Android и Windows с использованием XAML-разметки. Это упрощает разработку и сокращает время на создание интерфейса для каждой платформы.

Однако, есть и некоторые недостатки при использовании Xamarin:

1. Размер приложения. Приложения, созданные с использованием Xamarin, могут иметь больший размер по сравнению с нативными приложениями, так как они включают дополнительные библиотеки и среду выполнения Mono. Это может привести к дольше времени загрузки приложений и большему использованию ресурсов устройства.
2. Отставание в поддержке новых возможностей платформ. Xamarin может не сразу поддерживать новые возможности и API, представленные в новых версиях iOS или Android. Это может ограничить разработчиков в использовании самых актуальных функций операционных систем.
3. Зависимость от Microsoft и сообщества. Разработчики, использующие Xamarin, зависят от поддержки и обновлений со стороны Microsoft, а также от сообщества разработчиков. В случае возникновения проблем, разработчики могут столкнуться с задержками в решении проблем или ограниченной доступностью ресурсов для обучения.
4. Сложность в создании сложных пользовательских интерфейсов. Хотя Xamarin.Forms позволяет создавать общий пользовательский интерфейс для разных платформ, реализация сложных и высоко индивидуализированных пользовательских интерфейсов может быть более трудоем-

кой. В таких случаях разработчикам могут потребоваться платформо-специфические элементы и код, что увеличивает сложность проекта.

5. Необходимость знания платформо-специфических API и концепций. Несмотря на то, что Xamarin позволяет использовать единую кодовую базу на C#, разработчикам все равно нужно разбираться в платформо-специфических API и концепциях для реализации некоторых функций или для оптимизации производительности приложений.

Подводя итог, Xamarin является мощным инструментом для кросс-платформенной разработки, который подходит для относительно небольших проектов с общей бизнес-логикой и пользовательским интерфейсом на разных платформах. Однако, данный инструмент на текущий момент уже не является достаточно популярным, что может сказаться на поиске кадров для поддержки существующих приложений, так что его стоит использовать только при условии, что разрабатывается относиьельно небольшое приложение не для широкого использования.

1.2.3 React Native

React Native — это популярный кросс-платформенный фреймворк, который позволяет создавать мобильные приложения для iOS и Android, используя JavaScript и React. Он предоставляет разработчикам возможность писать одну кодовую базу, которая работает на обеих платформах, что значительно сокращает время разработки и упрощает поддержку приложений.

Преимущества React Native:

1. Один язык программирования. React Native позволяет использовать JavaScript, один из самых популярных и широко используемых языков программирования, что облегчает доступ к разработке приложений для многих разработчиков.
2. Фреймворк использует React, известный своей производительностью и модульностью, что позволяет разработчикам быстро создавать сложные и отзывчивые пользовательские интерфейсы с использованием компонентного подхода.
3. Горячая перезагрузка и быстрое обновление. React Native поддерживает возможность горячей перезагрузки [6], что позволяет разработчикам

видеть изменения в коде в реальном времени без необходимости перезагрузки приложения. Это значительно ускоряет процесс разработки и улучшает производительность.

4. Большое сообщество и экосистема. React Native имеет большое и активное сообщество разработчиков, что обеспечивает быстрое решение проблем, обширные ресурсы для обучения и большой выбор сторонних библиотек и плагинов для упрощения разработки.
5. Доступ к платформо-специфическим API [7]: React Native предоставляет доступ к платформо-специфическим API и нативным компонентам через модули, что позволяет разработчикам использовать функциональность и возможности конкретных платформ.

Недостатки React Native:

1. Производительность. Хотя React Native обеспечивает достаточно высокую производительность для большинства приложений, некоторые приложения с интенсивными графическими или вычислительными операциями могут столкнуться с проблемами производительности. В таких случаях нативная разработка может предоставить лучшие результаты [8].
2. Нативные модули и библиотеки. Не все платформо-специфические API и нативные библиотеки доступны "из коробки" в React Native, и иногда может потребоваться создавать собственные модули для их интеграции. Это может увеличить сложность проекта и время разработки.
3. Разработка и поддержка третьих компонентов. Использование сторонних библиотек и компонентов может привести к проблемам с обновлениями и поддержкой, особенно если эти компоненты зависят от нативного кода. Разработчикам приходится тщательно выбирать и проверять сторонние компоненты перед использованием их в проекте.
4. Изменения в платформах. Из-за быстрого развития мобильных платформ иногда могут возникать задержки в поддержке новых функций и изменений API в React Native. В таких случаях разработчики могут столкнуться с необходимостью самостоятельно реализовывать эти функции или ждать обновлений фреймворка.
5. Обучение и освоение. Несмотря на то что React Native использует популярный язык программирования JavaScript и библиотеку React, разработ-

чикам все равно потребуется время на освоение специфики фреймворка и его компонентов. Кроме того, знание платформо-специфических особенностей и API может быть необходимым для эффективной работы с React Native.

В целом, React Native является мощным и гибким инструментом для кросс-платформенной разработки, который подходит для создания множества разных типов приложений. Однако его эффективность и применимость в конкретных проектах зависят от ряда факторов, таких как сложность приложения, требования к производительности, доступность сторонних библиотек и компонентов, а также опыт и знания разработчиков. В некоторых случаях, использование React Native может быть оптимальным решением для разработки мультиплатформенного приложения, но в других ситуациях может быть предпочтительнее рассмотреть альтернативные технологии или нативную разработку.

1.2.4 Flutter

Flutter — это открытый фреймворк для разработки мультиплатформенных приложений, разработанный Google. Он позволяет создавать красивые и высокопроизводительные приложения для iOS, Android, Web и Desktop с использованием единого кодовой базы. Flutter использует язык программирования Dart, который также был разработан Google.

Преимущества Flutter:

1. Горячая перезагрузка [9]. Flutter также поддерживает горячую перезагрузку, что позволяет разработчикам видеть результат изменений кода в реальном времени, без необходимости перезапуска приложения. Это ускоряет процесс разработки и сокращает время на отладку.
2. Встроенные виджеты и дизайн. Flutter предлагает обширный набор встроенных виджетов, которые отлично адаптируются под разные платформы и экраны. Это позволяет создавать привлекательные и отзывчивые пользовательские интерфейсы с минимальными усилиями. Кроме того, Flutter поддерживает Material Design [10] и Cupertino [11] стили, что облегчает создание приложений, соответствующих стандартам дизайна каждой платформы.

3. Высокая производительность. Благодаря тому, что Flutter использует Dart и компилирует код в нативный ARM или x86, приложения на Flutter имеют высокую производительность, сопоставимую с нативными приложениями.
4. Поддержка разных платформ. Flutter поддерживает разработку приложений не только для iOS и Android, но и для веб-приложений и настольных приложений (Windows, macOS, Linux), что делает его еще более гибким и мощным инструментом для разработчиков.
5. Активное сообщество и поддержка: Сообщество разработчиков Flutter активно растет, и Google продолжает вкладывать ресурсы в его развитие. Это обеспечивает хорошую поддержку, доступность обучающих материалов и сторонних библиотек, а также регулярные обновления фреймворка.

Однако Flutter также имеет свои недостатки:

1. Зависимость от Dart. Flutter использует язык программирования Dart, который может быть незнакомым для многих разработчиков. Несмотря на то что Dart легко освоить, особенно для тех, кто знаком с JavaScript или Java, потребуется время на обучение и практику для эффективной работы с этим языком.
2. Размер приложений. Приложения, созданные с использованием Flutter, могут иметь больший размер по сравнению с нативными приложениями. Это связано с тем, что Flutter включает собственный движок для отрисовки, что увеличивает размер итогового пакета. Хотя это не всегда является критическим недостатком, в некоторых случаях это может повлиять на время загрузки и использование памяти на устройствах пользователей.
3. Доступность сторонних библиотек и плагинов. Несмотря на растущее сообщество и активное развитие, количество сторонних библиотек и плагинов для Flutter меньше, чем для некоторых других платформ, таких как React Native. В некоторых случаях, разработчикам придется создавать собственные решения или адаптировать существующие библиотеки для их нужд.

В целом, Flutter является мощным и гибким инструментом для разработки мультиплатформенных приложений. Он подходит для проектов различного масштаба и сложности, благодаря своим преимуществам, таким как высокая произво-

дительность, красивый и адаптивный дизайн, и поддержка различных платформ. Однако, перед началом работы с Flutter, разработчикам следует учесть потенциальные недостатки, такие как необходимость изучения языка программирования Dart, увеличенный размер приложений и ограниченное количество сторонних библиотек и плагинов.

1.2.5 Kotlin Multiplatform

Kotlin Multiplatform — это инновационный подход к разработке мультиплатформенных приложений, предложенный командой JetBrains, создателями языка программирования Kotlin. Kotlin Multiplatform позволяет использовать одну общую кодовую базу для разработки приложений под разные платформы, такие как Android, iOS, Web, Desktop (Windows, macOS, Linux), и даже серверные приложения.

Основные преимущества Kotlin Multiplatform:

1. Общий код. Kotlin Multiplatform позволяет разработчикам писать общий код для разных платформ, что сокращает время разработки, упрощает поддержку и обновление приложений, и обеспечивает единообразие функциональности на всех платформах.
2. Взаимодействие с нативными API. В отличие от некоторых других кроссплатформенных решений, Kotlin Multiplatform предоставляет возможность прямого взаимодействия с нативными API каждой платформы, что позволяет создавать высокопроизводительные и адаптированные приложения.
3. Гибкость. Kotlin Multiplatform позволяет разработчикам выбирать степень объединения кода между платформами. Разработчики могут решить, какие части кода будут общими, а какие останутся платформозависимыми. Это обеспечивает гибкость в выборе архитектуры приложения и позволяет сохранить преимущества нативной разработки.
4. Интеграция с существующими проектами. Kotlin Multiplatform может быть внедрен в уже существующие проекты, что позволяет разработчикам постепенно переходить на использование общего кода без необходимости переписывать приложение с нуля.

5. Поддержка сообщества и экосистема. Kotlin получил широкую популярность среди разработчиков и активно поддерживается сообществом. Это означает, что разработчики имеют доступ к большому количеству библиотек, инструментов и ресурсов, которые могут помочь в разработке мультиплатформенных приложений на Kotlin.
6. Мультиплатформенный пользовательский интерфейс. помимо инструмента для общего написания бизнес-логики приложения (Kotlin Multiplatform), JetBrains также разрабатывает инструмент для мультиплатформенного пользовательского интерфейса, Compose Multiplatform. Данная фреймворк разрабатывается на основе разрабатываемого Google фреймворка для пользовательского интерфейса для Android, Jetpack Compose, имеет совместимость с инструментом Google, и помимо Android может быть использован для Web, Desktop, и, с недавнего времени, iOS [12].

Несмотря на множество преимуществ, Kotlin Multiplatform также имеет свои недостатки:

1. Относительно новая технология. Kotlin Multiplatform является сравнительно новым решением на рынке, и его экосистема все еще развивается. Это может означать меньшее количество доступных ресурсов и библиотек по сравнению с более зрелыми кросс-платформенными решениями.
2. Более сложный процесс разработки графического пользовательского интерфейса. Несмотря на возможность писать мультиплатформенный графический пользовательский интерфейс, без глобальных модификаций под каждую платформу выглядеть в соответствиями с привычными интерфейсами под платформы он будет только на Android. Чтобы интерфейс выглядел "нативно" iOS, Web и Desktop потребуются значительные доработки или использование других способов написания пользовательского интерфейса
3. Библиотека для написания общего пользовательского интерфейса ещё не является стабильной для всех платформ. Compose Multiplatform официально считается стабильным только для Android и Desktop. Для Web данный инструмент еще в эксперементальной стадии разработки, для iOS на момент написания выпущена только Alpha версия библиотеки [12].

В целом, Kotlin Multiplatform является перспективным и гибким решением для разработки мультиплатформенных приложений, которое может облегчить процесс разработки, сократить затраты на поддержку и обновление, и улучшить качество продукта. Однако, как и любой другой инструмент, он имеет свои преимущества и недостатки, которые следует учитывать при выборе технологии для конкретного проекта.

1.3 Подведение итогов и выбор технологии

В результате анализа различных кросс-платформенных технологий и нативной разработки были выявлены их ключевые преимущества и недостатки. Нативная разработка предоставляет наилучшую производительность и интеграцию с платформами, но может быть ресурсоемкой и сложной в поддержке. Xamarin, React Native и Flutter предлагают разные подходы к разработке мультиплатформенных приложений, каждый со своими особенностями, возможностями и ограничениями.

Kotlin Multiplatform, в свою очередь, предлагает уникальный и гибкий подход, позволяющий разработчикам определить, какие части кода будут общими, а какие останутся платформо-зависимыми. Это обеспечивает возможность сохранить преимущества нативной разработки и одновременно сократить затраты на поддержку и обновление приложений.

Учитывая проведенный анализ, а также возможности, которые предоставляет Kotlin Multiplatform, было принято решение выбрать данную технологию для дальнейшего разбора и применения в разработке приложения. Kotlin Multiplatform обеспечивает поддержку разных платформ, включая Android, iOS, и Desktop, что делает его универсальным решением для создания современных приложений.

Основываясь на гибкости, перспективности и возможности совмещения с нативными платформами, Kotlin Multiplatform может быть оптимальным выбором для данного проекта. В следующих разделах работы будет рассмотрена детальная информация о Kotlin Multiplatform, его особенностях, архитектуре, возможностях и примерах применения в реальных проектах.

2 Разбор выбранной технологии

2.1 Введение в Kotlin Multiplatform

В данной главе будет рассмотрена технология Kotlin Multiplatform, которая была выбрана на основе сравнительного анализа в предыдущей главе. Kotlin Multiplatform представляет собой решение для разработки кросс-платформенных приложений, позволяющее использовать одну кодовую базу для создания приложений на разных платформах, таких как Android, iOS, Web и десктоп.

2.1.1 Преимущества Kotlin для разработки приложений

Kotlin предлагает ряд преимуществ для разработки приложений:

1. Обратная совместимость с Java. Kotlin полностью совместим с Java, что позволяет разработчикам интегрировать Kotlin в существующие проекты на Java или использовать Java-библиотеки в Kotlin-приложениях.
2. Поддержка многопоточности. Kotlin предлагает корутины работы с многопоточностью, которые позволяют разработчикам эффективно управлять параллелизмом и асинхронностью в своих приложениях.
3. Упрощение обслуживания и обновления. Kotlin Multiplatform упрощает обновление и поддержку приложений, так как разработчики могут использовать одну кодовую базу для всех платформ, что обеспечивает более быструю разработку и устранение ошибок.

2.2 Поддержка различных платформ

Как уже было сказано ранее, Kotlin Multiplatform предоставляет возможность использовать общий код для разработки приложений на разных платформах, таких как JVM/Android, Native (iOS, Desktop), WASM и JavaScript (Web). Это позволяет значительно сократить время разработки и обеспечивает удобство сопровождения приложений.

2.2.1 JVM/Android

Kotlin имеет сильную интеграцию с экосистемой Java и средой выполнения Java (JVM). Это обеспечивает разработчикам возможность использовать все преимущества Kotlin, такие как безопасность, совместимость и производительность, при создании приложений на платформе Android [13], а также серверных приложений [14]. Благодаря тесной интеграции с Android Studio и Intellij IDEA, разработчики могут комбинировать Kotlin и Java код в одном проекте и использовать обширную базу существующих Java-библиотек.

2.2.2 Native

Kotlin/Native [15] — это вариант компилятора Kotlin, который позволяет компилировать код на нативный исполняемый файл для различных платформ, таких как iOS, macOS, Linux, и Windows. Kotlin/Native основан на компиляторе LLVM и предоставляет доступ к нативным API платформы. Таким образом, разработчики могут создавать производительные и нативные приложения на Kotlin для мобильных и настольных платформ.

2.2.3 JS и WebAssembly (WASM)

Kotlin/JS [16] — это бэкенд компилятора Kotlin Multiplatform, который позволяет использовать Kotlin для разработки веб-приложений. С помощью Kotlin/JS разработчики могут писать код на языке Kotlin, который затем компилируется в JavaScript. Это обеспечивает совместимость с существующими библиотеками и инфраструктурой JavaScript, а также предоставляет возможность использовать преимущества Kotlin, такие как типобезопасность, расширения функций и более чистый синтаксис.

Kotlin/JS может быть скомпилирован для использования в браузер и Node.js (на данный момент находится в экспериментальной стадии разработки). Это позволяет разработчикам использовать Kotlin для создания как клиентских, так и серверных приложений, а также использовать одни и те же абстракции и библиотеки на обеих платформах.

Kotlin/WASM (WebAssembly) [17] — это еще одна возможность для разработки веб-приложений с использованием Kotlin. WebAssembly — это двоичный

формат инструкций для стековой виртуальной машины, предназначенный для выполнения кода на веб-страницах с высокой производительностью. Это позволяет разработчикам писать код на Kotlin и компилировать его в WebAssembly, который может быть выполнен в современных браузерах.

Kotlin/WASM находится на экспериментальной стадии разработки и пока что не так широко используется, как Kotlin/JS. Однако это может стать интересным направлением для будущих веб-проектов, так как WebAssembly предлагает преимущества в производительности и безопасности по сравнению с JavaScript.

В целом, Kotlin/JS и Kotlin/WASM предоставляют разработчикам гибкие инструменты для создания веб-приложений с использованием языка Kotlin, обеспечивая доступ к широкому спектру возможностей и совместимости с существующими технологиями веб-разработки.

2.3 Разбор компиляции под разные целевые платформы

Как уже упоминалось ранее, при разработке приложений с использованием Kotlin Multiplatform требуется написать общий код на языке Kotlin. Затем, в зависимости от выбранной конфигурации, этот код компилируется под нужную платформу. В данной секции будет кратко рассмотрен процесс компиляции под различные платформы.

2.3.1 Общий обзор компиляции в Kotlin Multiplatform

Процесс компиляции в Kotlin Multiplatform (KMP) основан на использовании Gradle, который обеспечивает автоматизацию сборки проектов. Для того чтобы лучше понять, как происходит компиляция под разные целевые платформы или таргеты с помощью Gradle, рассмотрим некоторые теоретические аспекты этого процесса.

Gradle использует систему инкрементальной компиляции, которая обеспечивает быструю сборку измененных частей проекта. Во время компиляции под разные таргеты, Gradle выполняет следующие основные этапы:

1. Анализ исходного кода и зависимостей. Gradle определяет структуру проекта, его модули и зависимости между ними. Исходный код может быть

разделен на общий код и платформозависимый код, который хранится в соответствующих модулях и исходных наборах (source sets).

2. Компиляция исходного кода. Gradle запускает компиляцию исходного кода для каждого таргета с использованием соответствующих компиляторов. Например, для компиляции под Android, Gradle использует Kotlin/JVM компилятор; для компиляции под iOS — Kotlin/Native компилятор; для компиляции под JavaScript — Kotlin/JS компилятор. Каждый из компиляторов генерирует код, оптимизированный для своей платформы.
3. Сборка и связывание. Gradle производит сборку и связывание всех компонентов проекта, таких как библиотеки, ресурсы, и исполняемые файлы, в соответствии с настройками проекта и таргета. В зависимости от платформы, могут использоваться разные инструменты и процессы для этого этапа.
4. Тестирование и проверка. Gradle позволяет запускать автоматические тесты и выполнять проверку кода с помощью статического анализа и других инструментов. Этот этап обеспечивает качество кода и корректное функционирование приложения на разных платформах.
5. Установка и публикация. После успешной компиляции и проверки, Gradle может автоматически развернуть приложение на устройствах или симуляторах для тестирования и отладки. Кроме того, Gradle поддерживает публикацию скомпилированных артефактов, таких как библиотеки и приложения, в удаленные репозитории или магазины приложений.

Процесс компиляции в Gradle для разных платформ основан на плагинах, которые обеспечивают поддержку соответствующих компиляторов и инструментов. Например, Kotlin Multiplatform Plugin добавляет поддержку Kotlin компиляторов для разных платформ и позволяет настроить проект для мультиплатформенной разработки.

2.3.2 Особенности компиляции для разных платформ

Компиляция для Android включает несколько шагов, таких как компиляция Kotlin кода в JVM байт-код, преобразование Java и Kotlin байт-кода в DEX-файлы, упаковка ресурсов, и создание APK или AAB файла.

Для компиляции кода под iOS, Kotlin/Native компилирует Kotlin код в нативные исполняемые файлы для конкретной платформы. В процессе сборки, Gradle выполняет задачи, связанные с компиляцией кода, созданием исполняемых файлов, и интеграцией с Xcode для сборки и установки приложения на устройство или симулятор.

Для компиляции Kotlin кода в JavaScript, Gradle использует Kotlin/JS, который транслирует Kotlin код в эквивалентный JavaScript код. В случае с браузерными приложениями, Gradle также может упаковывать ресурсы и создавать HTML-шаблон с подключением скомпилированного JS-файла.

Хотя поддержка компиляции Kotlin кода в WebAssembly находится на экспериментальной стадии, Gradle взаимодействует с Kotlin/WASM для генерации WASM-файлов. В процессе компиляции, Kotlin код транслируется в промежуточный LLVM-код, который затем компилируется в WebAssembly.

2.3.3 Использование механизма "ожидание-актуализация" (expect-actual)

В мультиплатформенных проектах на Kotlin Multiplatform, механизмы expect и actual позволяют реализовывать платформо-специфический код и обеспечивать его взаимодействие с общим кодом [18].

1. Объявление expect: в общем коде используются объявления expect для указания функций, классов, свойств или объектов, которые должны быть реализованы для каждой платформы. Объявления expect не содержат реализации и представляют собой контракт, который должен быть удовлетворен платформо-специфическим кодом. В листинге 1 представлен пример объявления такой функции.
2. Объявление actual: в платформо-специфических исходных наборах используются объявления actual для реализации expect-объявлений из общего кода. Объявление actual должно соответствовать сигнатуре expect-объявления и предоставлять платформо-специфическую реализацию. В листингах 2, 3 представлены примеры реализаций таких функций для Android и iOS
3. Использование expect и actual в коде: в общем коде можно вызывать функции, классы, свойства или объекты, объявленные с использовани-

ем expect, без знания о конкретной платформе. Во время компиляции и выполнения, платформо-специфические реализации actual будут использоваться вместо expect-объявлений, обеспечивая корректное поведение на каждой платформе.

Листинг 1: Пример объявления expect функции

```
expect fun getPlatformName(): String
```

Листинг 2: Пример объявления actual функции для Android

```
actual fun getPlatformName(): String {
    return "Android"
}
```

Листинг 3: Пример объявления actual функции для iOS

```
actual fun getPlatformName(): String {
    return "iOS"
}
```

Листинг 4: Пример использование expect функции из общего кода

```
fun printPlatformName() {
    println("Running on ${getPlatformName()}")
}
```

3 Разработка приложения

В третьей главе осуществляется детальное описание процесса разработки приложения, начиная от описания используемого API и этапа настройки проекта до написания платформенного кода.

В разделе 3.1 приводится краткое описание используемого API для реализации приложения.

В разделе 3.2 акцентируется внимание на вопросах настройки проекта. Здесь рассматриваются основные аспекты, связанные с подготовкой проекта к разработке, включая многомодульность, выбор и подключение библиотек, настройка модулей и инфраструктуры.

В разделе 3.3 описан код проекта. Будут расмотрены основные детали использования сетевых запросов, сохранения данных в локальную базу данных, сохранение простых данных, написание графического пользовательского интерфейса и архитектура приложения.

3.1 Используемое API

Для разработки приложения используется API базы данных «За Христа пострадавшие» — `http://api.nmbook.ru/docs`.

«За Христа пострадавшие» — это база данных, которая служит хранилищем информации о православных христианах, переживших репрессии за свою веру в период с 1917 по 1959 годы. Этот ресурс стал важным местом обращения для тех, кто искал информацию о людях, пропавших во время религиозных преследований. Данные о пострадавших представлены в виде биографических карточек, содержащих более ста различных характеристик. Все репрессированные за веру внесены в эту базу данных, независимо от их канонизации. База данных автоматически формирует справочник и способна содержать неограниченное количество информации.

База данных используется для научных исследований и публикаций ПСТГУ[1]. Материалы из базы данных послужили основой для создания иконы «Собор Новомучеников и Исповедников Российских XX века». База данных явля-

[1]Православный Свято-Тихоновский Гуманитарный университет

ется неоценимым инструментом для решения сложных вопросов, которые трудно решить без компьютерной обработки.

Были использованы методы для авторизации (`/login`), для проверки авторизации (`/whoami`), для получения списка новомученников (`/public/dela`) и для получения деталей по конкретному человеку (`/public/dela/number`).

3.2 Настройка проекта

В данном разделе подробно рассматривается процесс создания и настройки проекта. Приводится информация о создании многомодульного мультиплатформенного проекта, освещается вопрос подключения библиотек, раскрывается механизм композитных сборок и convention plugins. К тому же, детализируется процедура написания общего и платформенного кода, интеграции кода в iOS проект.

3.2.1 Многомодульность в контексте gradle

Gradle [19] — это система автоматизации сборки, широко используемая в Java и Android разработке. Одной из ключевых особенностей Gradle является поддержка многомодульности, что делает его мощным инструментом для больших и сложных проектов.

С ростом проекта и увеличением сложности кода, время его сборки становится гораздо дольше. Без использования многомодульности в проекте, основной модуль пересобирается полностью при любых изменениях, что может стать препятствием для эффективной и гибкой разработки. Для решения этой проблемы и ускорения процесса разработки используется подход, известный как многомодульность.

Многомодульность предполагает разделение проекта на отдельные модули, каждый из которых отвечает за выполнение определенных задач. В контексте Gradle, многомодульность позволяет эффективно управлять зависимостями, повторно использовать код и оптимизировать сборку проекта [20].

Gradle поддерживает многомодульные проекты путем создания структуры проекта, где каждый модуль представляет собой отдельный проект Gradle со сво-

им собственным файлом `build.gradle(.kts)`. Это позволяет индивидуально настраивать каждый модуль, управлять его зависимостями и параметрами сборки.

Благодаря многомодульности, код разделяется на логические блоки, что позволяет разрабатывать и тестировать каждый модуль независимо. Это не только ускоряет процесс разработки, но и делает его более управляемым, поскольку изменения в одном модуле могут быть проверены и протестированы без влияния на другие модули. Также Gradle может параллельно собирать независимые модули, значительно ускоряя процесс сборки.

Кроме того, многомодульность облегчает управление зависимостями. Вместо одного большого файла зависимостей, каждый модуль может иметь свой собственный, что обеспечивает большую гибкость и контроль.

Также в контексте многомодульных проектов стоит упомянуть процесс управления зависимостями. Поскольку различные модули могут требовать одни и те же внешние библиотеки, возникает необходимость следить за их версиями, чтобы в рамках проекта не были подключены одни и те де библиотеки с различными версиями. Для данного процесса у Gradle есть механизм version catalog [21], позволяющий указывать все зависимости в одном месте и ссылаться на них из файлов конфигурации. Все необходимые зависимости указываются в файле `libs.version.toml`, по умолчанию располагающемся в директории `gradle` в корне проекта, однако путь возможно поменять в файле `settings.gradle(.kts)`. В листинге 5 представлен пример объявления библиотек с версией, в листинге 6 представлен пример использования данного объявления внутри конфигурационного файла `build.gradle.kts`.

Листинг 5: Пример объявления библиотек в файле libs.version.toml

```
1  [versions]
2  kotlin-general = "1.8.20"
3
4  [libraries]
5  kotlin-gradle = { group = "org.jetbrains.kotlin", name =
   ↪  "kotlin-gradle-plugin", version.ref = "kotlin-general" }
6  kotlin-serialization = { group = "org.jetbrains.kotlin", name =
   ↪  "kotlin-serialization", version.ref = "kotlin-general" }
```

Листинг 6: Пример подключения библиотек в конфигурационном файле

```
dependencies {
    implementation(libs.kotlin.gradle)
    implementation(libs.kotlin.serialization)
}
```

3.2.2 Выбор и подключение библиотек

Базовые аспекты, которыми обладают современные приложения, включают в себя функциональность для сетевого обмена данными, сохранение информации в локальном хранилище и использование подходящих инструментов для структурирования архитектуры.

Для данных целей были разработаны библиотеки, поддерживающие работу на разных платформах, и являющиеся на данный момент наиболее популярными в сфере разработки мультиплатформенных приложений на Koltin. Данные библиотеки имеют общую часть для использования в общем коде и платформенные — для инициализации библиотек с учетом особенности платформы.

Один модуль `gradle` делится на несколько `sourceSets` — `commonMain`, `androidMain`, `desktopMain`, `iosMain` и так далее. Если библиотека реализована под функциональность, имеющую архитекрутные особенности реализации для каждой платформыы, она должна иметь реализации для каждой из платформ, на которой планируется её запуск. Библиотеки подключаются в `build.gradle(.kts)` файл соответствующего модуля в блок `dependencies`. В листинге 7 представлен пример подключения одной библиотеки с различными реализациями под платформы в модуль.

В примере приложения необходимо продемонстрировать следующую функциональность: сетевые запросы, сохранение данных в базе данных, сохранение простых данных в формате ключ-значение, также необходима навигация между экранами, Dependency Injection для обеспечения модульности, тестируемости и управления жизненным циклом компонентов.

Для сетевых запросов было решено использовать библиотеку `ktor` [22], которая является одним из наиболее популярных вариантов для сетевых запросов

Листинг 7: Пример подключения библиотек в конфигурационном файле

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                api(libs.ktor.core)
            }
        }
        androidMain {
            dependencies {
                implementation(libs.ktor.android)
            }
        }
        iosMain {
            dependencies {
                implementation(libs.ktor.ios)
            }
        }
        desktopMain {
            dependencies {
                implementation(libs.ktor.okhttp)
            }
        }
    }
}
```

на языке Kotlin и разрабатывается компанией JetBrains. Ktor совместима с Kotlin Coroutines, что обеспечивает асинхронную работу данного фреймворка.

Для сохранения в базе данных было решено использовать один из наиболее популярных вариантов, поддерживающих Kotlin Multiplatform, SQLDelight [23]. Она позволяет использовать SQL для описания схемы базы данных и запросов, после чего автоматически генерирует безопасные по типам Kotlin API для этих запросов. Благодаря интеграции с Gradle и Android Studio, SQLDelight предоставляет возможности для проверки схемы базы данных, управления миграциями базы данных, подсветки синтаксиса и автозаполнения. Кроме того, SQLDelight поддерживает тестирование, предоставляя сгенерированные интерфейсы, которые можно имитировать в ваших тестах.

Для сохранения простых данных была выбрана библиотека Multiplatform Settings (Settings) [24]. Данный инструмент, разработанный для Kotlin

Multiplatform, обеспечивает простой и удобный доступ к постоянным настройкам пользователей и данным на всех поддерживаемых платформах.

Она позволяет хранить простые данные, такие как строки, числа, булевы значения или списки, в постоянном хранилище, которое сохраняется между сеансами приложения. На Android эта библиотека использует SharedPreferences для хранения данных, а на iOS — NSUserDefaults. На Desktop эта библиотека использует подходящие механизмы хранения настроек, зависящие от операционной системы. Например, на Windows может быть использован Registry, на macOS — UserDefaults, а на Linux — возможны различные варианты, включая файлы конфигурации в домашнем каталоге пользователя.

Settings обеспечивает единый, платформонезависимый API, который значительно упрощает работу с настройками при разработке приложений с использованием Kotlin Multiplatform.

Для Dependency Injection была выбрана библиотека Kodein [25], для навигации — Odyssey [26], для реализации архитектурного паттерна MVI — библиотека KViewModel [27].

3.2.3 Convention plugins

В процессе реализации многомодульного приложения возникает проблема множественного объявления файлов конфигурации каждого модуля. Модули могут быть созданы для схожей логики, иметь одинаковые зависимости и идентичную настройку. Для решения данной проблемы система сборки Gradle представляет инструмент Convention Plugins, позволяющий переспользовать конфигурации для различных модулей. Для примера, в листинге 8 представлен вариант convention plugin'а для общего модуля, не имеющего пользовательского интерфейса и собирающегося для Android, iOS и Desktop.

В данном примере сначала применяются плагины, которые необходимы модулям. Модули с данным плагином будут представлять из себя библиотеку, который можно подключить в Android, desktop и iOS проект, всвязи с чем подключаются плагины com.android.library и kotlin.multiplatform в блоке plugins.

В блоке kotlin описаны платформы, на которых можно будет использовать даный модуль. В блоке sourceSets описаны наборы исходников, не заданных по

Листинг 8: Пример Convention Plugin'а

```
plugins {
    id("com.android.library")
    kotlin("multiplatform")
}

kotlin {
    jvm("desktop")
    android()
    ios()
    iosSimulatorArm64()

    sourceSets {
        val iosSimulatorArm64Main by getting
        val iosSimulatorArm64Test by getting
        val iosMain by getting {
            iosSimulatorArm64Main.dependsOn(this)
        }
        val iosTest by getting {
            iosSimulatorArm64Test.dependsOn(this)
        }
    }

    tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile> {
        kotlinOptions.jvmTarget = "11"
    }
}
```

умолчанию. В них указывается, что `iosSimulatorArm64Main` зависит от `iosMain`, и `iosSimulatorArm64Test` зависит от `iosText`.

Далее устанавливается версия целевой JVM для компиляции кода Kotlin: `kotlinOptions.jvmTarget = "11"` указывает, что код должен быть скомпилирован для JVM 11.

В листинге 9 представлен пример применения `convention plugin`'а в конфигурационном файле модуля. Данный плагин подключается как другие Gradle плагины в блоке `plugins` файла `build.gradle.kts`. При подключении модуль получает все свойства, описанные в плагине.

Листинг 9: Пример применения Convention Plugin'а

```
plugins {
    id("multiplatform-setup")
}
```

3.2.4 Общий и платформенный код

Как уже было сказано ранее, код в модулях проекта делится на общий и платформенные. Данное разделение происходит с помощью инструмента `sourceSets`, описанный в блоке 3.2.2. В общей части пишется основная логика, в платформенных происходит специфичная настройка для улучшения пользовательского опыта и использование платформенных API.

Общий код реализуется в `commonMain sourceSet`, платформенный — в `androidMain`, `desktopMain`, `iosMain` и так далее, в зависимости от поддерживаемых платформ. Если для метода или класса нужна специфичная платформенная реализация, в `commonMain` объявляется метод или класс с модификатором `expected`, и в других `sourceSet`'ах описываются `actual` реализации. Подробнее данных механизм был описан в блоке 2.3.3.

3.2.5 Подключение в iOS проект

Для подключения многомодульного проекта к iOS-проекту используется итоговый модуль (так называемый `umbrella`-модуль), в который подключаются все остальные модули и реализуется логика. Стоит отметить, что подключение отдельных модулей нежелательно, поскольку каждый из них содержит в себе среду выполнения Kotlin Multiplatform, что может существенно увеличить итоговый размер приложения.

Для подключения итогового мультиплатформенного модуля в iOS проект будет использоваться инструмент управления зависимостями `CocoaPods`. В дальнейшем описании будет использоваться термин «фреймворк» — в iOS разработке так называют библиотеки. Для сборки фреймфорка из Gradle модуля необходимо подключить плагин `native.cocoapods`. Подключение происходит вместе с остальными требуемыми плагинами и описано в листинге 10.

Листинг 10: Подключение плагина cocoapods в build.gradle.kts

```
plugins {
    id("multiplatform-setup")
    id("android-setup")
    kotlin("native.cocoapods")
}
```

Далее в блоке `cocoapods` внутри `kotlin` описывается фреймворк. В данном блоке есть описание фреймворка (`summary`), ссылка на страницу (`homepage`), целевая версия iOS для сборки (`ios.deploymentTarget`), путь к Podfile iOS проекта (`podfile`), а также настраиваются зависимости в блоке `framework`. Поле `transitiveExport` указывает, должны ли все зависимости, указанные в `export`, быть включены в финальную сборку. Если значение `false`, то в сборку включаются только непосредственные зависимости.

Параметр `isStatic` указывает, должна ли библиотека быть статической. Статические библиотеки включают в себя все свои зависимости, что облегчает их распространение, так как все необходимое для работы содержится внутри одного файла.

В параметре `baseName` указывается базовое имя для создаваемого фреймворка. В данном случае, финальная библиотека будет называться `shared_compose`.

Параметр `freeCompilerArgs` добавляет аргументы для компилятора. В данном случае указывается bundleId для финального фреймворка.

С помощью метода `export` указываются всё модули, которые используются для сборки фреймфорка

В листинге 11 представлена настройка фреймворка в итоговом `umbrella` модуле, который будет подключен в iOS проект.

В листинге 33 представлен полный код файла build.gradle.kts модуля, подключаемого в iOS проект.

Далее данный фреймворк должен быть подключен в проект. Для этого в файле Podfile в корне iOS проекта указывается фреймворк, который необходимо подключить. Код для подключения описан в листинге 12. В блоке указывается проект, в которые подключается модуль (`target`), целевая версия операционной

Листинг 11: Настройка фреймворка в файле build.gradle.kts

```
kotlin {
    cocoapods {
        summary = "MPP iOS SDK + Compose"
        homepage = "https://github.com/AndVl1/mpp-diploma-app"
        ios.deploymentTarget = "14.0"
        podfile = project.file("../../apps/iosApp/Podfile")

        framework {
            transitiveExport = false
            isStatic = true
            baseName = "shared_compose"
            freeCompilerArgs += "-Xbinary=bundleId=ru.andvl.mppapp"
            export(/*...*/)
        }
    }
    // ...
}
```

системы и сама операционная система, для которой происходит подключение (`platform`), название подключаемого фреймворка и путь к нему относительно расположения текущего файла (`pod, :path`).

Листинг 12: Настройка фреймворка в файле build.gradle.kts

```
# workspace 'iosApp'
target 'iosApp' do
  use_frameworks!
  platform :ios, '14.1'
  pod 'umbrella_compose', :path => '../../common/umbrella-compose'
end
```

3.3 Написание общего кода

В этом разделе будет рассмотрена ключевая составляющая процесса мультиплатформенной разработки — написанию общего кода. Именно в этом пункте уникальность подхода Kotlin Multiplatform проявляется наиболее ярко, позволяя разработчикам создавать код, который может быть использован на всех поддерживаемых платформах без значительных изменений.

Сначала будет подробно рассмотрена настройка внедрения зависимостей (Dependency Injection, DI) в проекте. Это важный этап, который поможет обеспечить модульность и повторное использование кода.

Затем будет описан процесс реализации сетевых запросов с помощью библиотеки ktor — одного из наиболее популярных инструментов в экосистеме Kotlin.

В следующем подразделе будет рассмотрена работа с базой данных, используя SQLDelight.

Также будет рассмотрено создание графического пользовательского интерфейса и навигации приложения. Здесь мы рассмотрим основные подходы и рекомендации, которые следует использовать при проектировании пользовательского интерфейса в мультиплатформенном приложении.

Также будет исследован процесс сохранения простых данных. Это важное дополнение к работе с базой данных и сетевыми запросами, которое может существенно улучшить пользовательский опыт.

Наконец, мы обсудим архитектуру приложения. В этом подразделе мы покажем, как можно строить структуру проекта, чтобы сделать его максимально читаемым, масштабируемым и поддерживаемым.

3.3.1 Настройка DI в проекте

Для управления зависимостями в проекте используется библиотека Kodein-DI. Данный инструмент позволяет создавать и управлять жизненным циклом объектов в мультиплатформенных проектах на языке Kotlin. Kodein-DI поддерживает концепцию модулей, которые представляют собой самостоятельные единицы конфигурации для биндинга зависимостей.

Модули Kodein-DI используются для группировки и инкапсуляции биндингов зависимостей, которые связаны по логике или функциональности. Каждый модуль представляет собой отдельный блок конфигурации, в котором определяются правила для создания и предоставления зависимых объектов.

Примеры применения Kodein будут приведены далее, в контексте настройки других библиотек.

3.3.2 Сетевые запросы — ktor

Как уже было сказано в блоке 3.2.2, для сетевых запросов в проекте используется библиотека `Ktor`. Для использования библиотеки необходимо подключить её в `build.gradle.kts` файле модуля. Данная библиотека имеет драйвера для каждой платформы, всвязи с этим необходимо подключать свой артефакт в каждый из `sourceSet`'ов. В листинге 7 уже был представлен код для подключения библиотек с необходимостью использования платформенной реализации.

Для настройки данной библиотеки пишется код в `commonMain sourceSet'e` и в платформенных, для инициализации с требуемым движком. В общей части необходимо реализовать DI модуль, чтобы предоставлять его в другие части приложения, где требуется взаимодействие с сетью. В листинге 13 представлен код инициализации `Http` клиента в общем коде. Внутри инициализации могут быть установлены разлчные модули для `Ktor` для логирования, сериализации, контроля таймаутов и так далее. Также в блоке `defaultRequest` могут быть установлены параметры запроса по умолчанию.

Листинг 13: Инициализация DI модуля для Ktor

```
internal val ktorModule = DI.Module("ktorModule") {
    bind<HttpClient>() with singleton {
        HttpClient(HttpEngineFactory().createEngine()) {
            // Установка модулей Ktor...

            defaultRequest { // установка параметров по умолчанию
                url("http://api.nmbook.ru/")
                header("accept", "application/json; charset=UTF-8")
                header("Content-Type",
                ↪  "application/x-www-form-urlencoded")
            }
        }
    }
}
```

Затем, в зависимости от платформы, создается объект `HttpClientEngineFactory`. Это сделано для того, чтобы в каждой платформе использовался свой `HttpClientEngine`, оптимизированный для данной платформы. В листинге 14 представлен код инициализации `HttpClientEngine` для `Android` (`Android`), `Desktop` (`OkHttp`) и `iOS` (`Darwin`).

Листинг 14: Инициализация движка Ktor для платформ

```
// commonMain
internal expect class HttpEngineFactory constructor() {
    fun createEngine(): HttpClientEngineFactory<HttpClientEngineConfig>
}
// androidMain
internal actual class HttpEngineFactory actual constructor() {
    actual fun createEngine():
    ↪ HttpClientEngineFactory<HttpClientEngineConfig> = Android
}
// desktopMain
internal actual class HttpEngineFactory actual constructor() {
    actual fun createEngine():
    ↪ HttpClientEngineFactory<HttpClientEngineConfig> = OkHttp
}
// iosMain
internal actual class HttpEngineFactory actual constructor() {
    actual fun createEngine():
    ↪ HttpClientEngineFactory<HttpClientEngineConfig> = Darwin
}
```

Для использования созданного клиента в модуле, в котором планируется использование, необходимо создать DI модуль. Использование `ktor` будет рассмотрено на примере модуля авторизации. В данном модуле создаётся `authModule`, в котором создаются экземпляры классов (в нашем случае — `RemoteAuthDataSource`), для которых нужен `HttpClient`. `HttpClient` предоставляется в конструктор класса с помощью вызова функции `instance()`. В листинге 15 представлен код создания `RemoteAuthDataSource`, использующего `Ktor`.

Листинг 15: Предоставление экземпляра Ktor в DataSource

```
val authModule = DI.Module("authModule") {
    bind<RemoteAuthDataSource>() with provider {
        RemoteAuthDataSource(instance())
    }
}
```

Далее, для осуществления запросов используются методы `get`, `post` и так далее. Функции выполняются асинхронно с использованием корутин, так что метод, в которым они выполняются, должен быть помечен модификатором `suspend`.

Полученное значение десериализуется с помощью метода `body()`, вызываемого для запроса. В листинге 16 представлен пример вызова `post` метода.

Листинг 16: Пример вызова метода post с использованием Ktor

```
class RemoteAuthDataSource(
    private val httpClient: HttpClient
) {
    suspend fun performLogin(request: AuthRequest): LoginRequestDto {
        return httpClient.post {
            url {
                path("login") // путь запроса
                setBody(FormDataContent(
                    Parameters.build {
                        append("grant_type", request.grantType)
                        // инициализация остальных параметров
                    }
                ))
            }
        }.body()
    }
    // ...
}
```

3.3.3 База данных — SQLDelight

Для сохранения в базе данных используется библиотека `SQLDelight`. Аналогично `Ktor`, для базы данных создаётся DI-модуль. В листинге 17 написана инициализация базы данных в общем модуле. Здесь `Kodein-DI` используется для связывания экземпляра `DbDriverFactory` и базы данных `Database` как синглтонов. `DbDriverFactory` отвечает за создание драйвера базы данных, который используется при создании экземпляра базы данных. В листинге

Примечательно, что `DbDriverFactory` определен как платформозависимый (`expect`) класс, что означает, что реализация этого класса может отличаться в зависимости от конкретной платформы.

Так, например, для платформы `Android` используется `AndroidSqliteDriver`, для настольных операционных систем — `JdbcSqliteDriver`, для `iOS` — `NativeSqliteDriver`. База данных инициализируется аналогично `HttpClient` с помощью механизма `expect-actual`.

Листинг 17: Инициализация DI модуля для SQLDelight

```kotlin
internal val databaseModule = DI.Module("databaseModule") {
    bind<DbDriverFactory>() with singleton {
        DbDriverFactory(instance())
    }
    bind<Database>() with singleton {
        val driverFactory: DbDriverFactory = instance()
        val driver = driverFactory.createDriver("cache.db")
        Database(driver)
    }
}
```

Также для работы базы данных необходимо инициализировать её в `build.gradle.kts` файле модуля, к котором описывается база данных. Для этого необходимо подключить плагин `app.cash.sqldelight`. Также необходимо в `commonMain sourceSet'e` модуля создать директорию `sqldelight`, в которой будет находиться `sql` скрипт базы данных, который при компиляции будет использоваться для генерации кода на `Kotlin`. После всего этого в блоке `sqldelight` конфигурационного скрипта `build.gradle.kts` необходимо описать создаваемую базу данных. В листинге 18 представлена настройка базы данных в скрипте конфигурации. В данном коде указывается имя класса базы данных, которое будет создано (`create("Database")`), имя пакета (`packageName.set()`), путь к схеме базы данных, которая будет сгенерирована (`schemaOutputDirectory.set()`), путь к файлам, которые будут сгенерированы при миграции базы данных (`migrationOutputDirectory.set(file())`) и параметр `linkSqlite`, который указывает на необходимость подключения стандартной библиотеки `SQLite` к проекту.

В директории `sqldelight`, упомянутой выше, создаётся файл с расширением `.sq`, в котором описываются скрипты для базы данных. В листинге 19 представлен пример подобного файла. В данном файле описывается таблица базы данных и различные скрипты. Перед определением скрипта указывается, как будет называться сгенерированный метод для соответствующего скрипта в коде `Kotlin`.

В результате компиляции создаётся интерфейс с названием, соответствующему названию, указанному при инициализации в `build.gradle.kts` (листинг

Листинг 18: Инициализация SQLDelight в build.gradle.kts

```
sqldelight {
    databases {
        create("Database") {
            packageName.set("")
            schemaOutputDirectory.set(file("relative-path"))
            migrationOutputDirectory.set(file("relative-path"))
            linkSqlite.set(true)
        }
    }
}
```

Листинг 19: Пример файла для базы данных SQLDelight

```
CREATE TABLE dela(
    key INTEGER NOT NULL PRIMARY KEY,
    fio TEXT NOT NULL
);

insert:
INSERT OR REPLACE INTO dela(key, fio)
VALUES (?, ?);

select:
SELECT * FROM dela;

update:
INSERT OR REPLACE INTO dela(key, fio)
VALUES (?, ?);
```

18). Далее через данный интерфейс происходит взаимодействие с базой данных, пример использования функций `insert` и `select` приведён в листинге 20.

3.3.4 Графический пользовательский интерфейс и навигация

Для реализации графического пользовательского интерфейса было решено использовать фреймворк `Compose Multiplatform`. Это новый инструментарий, предоставленный `JetBrains` и `Google`, для создания реактивных пользовательских интерфейсов в мультиплатформенных проектах `Kotlin`.

Листинг 20: Использование интерфейса Database в коде

```
class DelasLocalDataSource(
    private val database: Database
) {
    fun saveDelaList(dela: DelaDto) {
        dela.items
            ?.map { Pair(it.key, it.fio) }
            ?.filter {
                it.first != null && it.second != null
            }
            ?.forEach {
                database.itemsQueries.insert(it.first!!.toLong(),
                ↪  it.second!!)
            }
    }

    fun getLocalDela(): DelaDto {
        val delas = database.itemsQueries.select().executeAsList()
            .map { ItemDto(key = it.key.toInt(), fio = it.fio) }
            .toCollection(ArrayList())
        return DelaDto(items = delas)
    }
}
```

Compose Multiplatform —- это адаптация Jetpack Compose для мультиплатформенной разработки на Kotlin, позволяющая разрабатывать графической пользовательский интерфейс с использованием декларативного подхода. Compose Multiplatform поддерживает не только Android, но и другие платформы, такие как iOS и Desktop, в связи с чем он и был выбран для разработки.

Для использования Compose Multiplatform, в convention plugin, подключаемый в модули с графическим пользовательским интерфейсом должны быть подключены необходимые зависимости: плагин org.jebrains.compose и сами зависимости для использования Compose Multiplatform на использовании на платформе Desktop и iOS и Jetpack Compose для использования на Android. Код convention plugin'а с подключенным Compose Multiplatform представлен в листинге 34. Также необходимо добавить строку org.jetbrains.compose.experimental.uikit.enabled=true в файл gradle.properties.

Для использования графического пользовательского интерфейса в приложениях для каждой платформы необходимо создать точку входа. Для этого в каждом платформенном sourceSet создаётся файл, init.android.kt. В данном файле реализуется функция для инициализаци DI iOS и Desktop приложения (Android инициализируется из Application класса), а также настраивается пользовательский интерфейс. Для Android была написана extension функция для ComponentActivity. Пример подобной инициализации представлен в листинге 21.

Листинг 21: Инициализация Compose в Android приложении

```
fun ComponentActivity.setupThemedNavigation() {
    setContent {
        MaterialTheme {
            MainContent()
        }
    }
}

@Composable
private fun MainContent(rootController: RootController) {
    // Инициализация контента
    CompositionLocalProvider(
        LocalRootController provides rootController
    ) {
        // Контент
    }
}
```

В листинге 22 представлен код функции main Desktop-приложения и вызываемая функция setupThemedNavigation из модуля umbrella-compose, которая уже содержит в себе пользовательский интерфейс приложения.

Для отображения графического пользовательского интерфейса на Compose Multiplatform на iOS, необходимо создать «мост» между приложением и кодом на Kotlin. Далее будет описан процесс вызова Composable функций из Swift кода.

Для начала, @main указывает на точку входа в приложение, которая является структурой iOSApp. Внутри этой структуры используется Scene, который включает в себя одну или несколько WindowGroup. Они представляют собой группы окон приложения. Данный код представлен в листинге 23.

Листинг 22: Инициализация Compose в Desktop приложении

```
// apps/desktopApp/main.kt
fun main() = singleWindowApplication(
    state = WindowState(size = DpSize(500.dp, 500.dp))
) {
    setupThemedNavigation()
}
// umbrella-compose/desktopMain
@Composable
fun setupThemedNavigation() {
    PlatformSdk.init(PlatformConfiguration())

    MaterialTheme {
        // Контент
    }
}
```

Внутри группы окон находится `ContentView`, основное представление, которое отображается при запуске приложения. `ContentView` включает в себя `ComposeView`, которое отображает представление, созданное с помощью `Compose Multiplatform`.

`ComposeView` является реализацией протокола `UIViewControllerRepresentable`, который позволяет интегрировать `UIKit ViewController` в `SwiftUI View`. В методе `makeUIViewController`, `ComposeView` создает `MainViewController` из `shared_compose`. Код реализации представлен в листинге 24. Код реализации `MainViewController`, вызывающегося из `Swift` кода, представлен в листинге 25.

Листинг 23: Точка входа в iOS приложение

```
import SwiftUI

@main
struct iOSApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Листинг 24: Инициализация MainViewController в Kotlin коде

```
import UIKit
import SwiftUI
import shared_compose

struct ComposeView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UIViewController {
        Init_iosKt.MainViewController()
    }

    func updateUIViewController(_ uiViewController: UIViewController,
    ↪  context: Context) {}
}

struct ContentView: View {
    var body: some View {
        ComposeView()
                .ignoresSafeArea(.keyboard) // Compose has own keyboard
                ↪  handler
    }
}
```

Листинг 25: Функции вызова Compose кода из iOS проекта

```
fun MainViewController(): UIViewController =
    ComposeUIViewController {
        PlatformSdk.init(PlatformConfiguration())
        // контент
    }
```

Для навигации была выбрана библиотека `Odyssey` — декларативная мультиплатформенная библиотека навигации, специально разработанная для `Compose Multiplatform`. Для использования все экраны прописываются графе в модуле, подключаемом к приложениям (`umbrella-compose`), каждому экрану присваивается идентефикатор в виде строки. Идентефикаторы могут храниться в любом виде по желанию разработчика, однако наиболее удобно добавлять все экраны в один `object` в `core`-модуль, для навигации из других модулей. Пример объявления экранов представлен в листинге 26. В данном примере объявлены экраны `SplashScreen`, являющийся начальным экраном приложения, `DetailsScreen`,

являющийся экраном с деталями элемента списка и принимающем целое число как параметр.

Листинг 26: Инициализация навигационного графа приложения

```
@Composable
internal fun RootComposeBuilder.generateGraph(
    backgroundColor: Color,
    selectedColor: Color,
    unselectedColor: Color
) {
    screen(name = NavTree.Splash.SplashScreen.name) {
        SplashScreen()
    }
    screen(name = NavTree.Details.Details.name) {
        (it as? Int)?.let { id -> DetailsScreen(id) }
    }

    mainFlow(
        backgroundColor,
        selectedColor,
        unselectedColor,
    )
}
```

Также в данной функции объявляется `mainFlow`, внутри которого содержится главный экран с нижней навигацией. Листинг объявления данного экрана представлен в листинге 27.

3.3.5 Сохранение простых данных

Для выполнения запросов к используемому `API` необходимо иметь токен, который отправляется вместе с запросом и на серверной части проверяется его валидность. Данных токен сохраняется в памяти при помощи библиотеки `Settings`, которая использует платформенные хранилища для сохранения простых данных. Для использования данную библиотеку также надо предоставлять с помощью DI. Для этого создаётся `DI` модуль. Код создания предоставлен в листинге 28.

Для использования необходимо предоставить созданный экземпляр класса в места, где надо сохранять данные. Для этого, по аналогии с `Ktor`, вызывается функция `instance()` в конструкторе класса при создании синглтона данного класса. В листинге 29 представлен код создания `SettingsAuthDataSource`, использующего `Settings`.

Листинг 27: Создание нижней навигации приложения

```
@Composable
fun RootComposeBuilder.mainFlow(
    backgroundColor: Color,
    selectedColor: Color,
    unselectedColor: Color
) {
    bottomNavigation(
        name = NavTree.Main.Dashboard.name,
        colors = BottomBarDefaults.bottomColors(
            backgroundColor
        )
    ) {
        val colors = TabDefaults.tabColors(
            selectedTextColor = selectedColor,
            selectedIconColor = selectedColor,
            unselectedTextColor = unselectedColor,
            unselectedIconColor = unselectedColor
        )
        tab(MainTab().tab, colors) {
            screen(name = NavTree.Main.List.name) {
                ListScreen()
            }
        }
    }
}
```

Листинг 28: Инициализация DI модуля для Settings

```
internal val settingsModule = DI.Module("settings") {
    bind<Settings>() with singleton { Settings() }
}
```

Листинг 29: Предоставление экземпляра Settings в DataSource

```
val authModule = DI.Module("authModule") {
    bind<SettingsAuthDataSource>() with provider {
        SettingsAuthDataSource(instance())
    }
}
```

Как уже было сказано в 3.2.2, Settings используется для хранения простых данных. Данные хранятся как пары ключа и значения, где ключ — это строка, а значение — любой простой тип. Доступ к данным происходит с помощью методов get и put, в листинге 30 представлен пример использования данной библиотеки.

Листинг 30: Использование библиотеки Settings

```
class SettingsAuthDataSource(
    private val settings: Settings
) {
    fun saveToken(token: String) {
        settings.putString(TOKEN_KEY, token)
    }
    fun getToken(): String {
        return settings[TOKEN_KEY, ""]
    }
    companion object {
        private const val TOKEN_KEY = "auth_token"
    }
}
```

3.3.6 Архитектура приложения

Как уже было сказано ранее, для реализации приложения было выбрано использовать многомодульную архитектуру. Для реализации были выделены отдельные модули для "core"-функциональности (инициализация базы данных, сетевого слоя, DI и так далее), для каждого экрана (список, детали) или отдельной логики не имеющей экранов (авторизация), и общий модуль, подключаемый к приложениям ("umbrella"-модуль). На рисунке 4 представлена структура проекта, на рисунке 3 представлен граф зависимостей проекта для Android приложения, демонстрирующий зависимости между модулями проекта. Для Desktop граф выглядит аналогично, для iOS отсутствуют связи на модуль приложение, поскольку к нему подключается только финальный, umbrella-compose модуль в качестве единой библиотеки.

Модуль для пользовательских функциональностей разделены на 4 — api, data, presentation и compose. В модулях api описываются основные интерфейсы взаимодействия с функциональностью модуля, data — реализация интерфейсов и дополнительные классы для реализации, presentation — архитектурная

прослойка между пользовательским интерфейсом и api-слоем, о которой будет сказано далее, compose — реализация графического пользовательского интерфейса функциональности.

Для реализации функциональностей с пользовательским интерфейсом было выбрано использовать архитектурный паттерн MVI (Model-View-Intent). Данный подход — один из наиболее популярных UDF паттернов (unidirectional data flow — однонаправленный поток данных). Существует множество библиотек для реализации данного архитектурного паттерна, для реализации в текущем проекте была выбрана KViewModel [27], имеющая небольшой, но достаточный для реализации MVI набор абстракций.

Основная сущность данной библиотеки — абстрактный класс BaseViewModel, принимающий три типа параметров: State, Action, и Event, где State представляет текущее состояние View, Action — действия, которые могут быть выполнены, и Event — события, которые приходят от View. Также данный класс имеет публичные методы viewStates() и viewActions(), которые возвращают обёртки над StateFlow и SharedFlow соответственно, что позволяет другим классам подписываться на эти потоки данных, но не менять их и поля viewState и viewAction, которые позволяют получить текущее состояние и действие и установить их новые значения.

Для обработки событий, приходящих от View, данный класс имеет абстрактный метод obtainEvent(viewEvent: Event), который должен быть реализован в подклассах.

В листинге 35 представлен пример реализации ViewModel. В данном классе обрабатываются события, которые приходят с View слоя (DelaEvent), хранится состояние экрана (DelaViewState) и отправляются действия на слой View (DelaAction). DelaEvent и DelaAction представляют собой sealed структуры — интерфейсы с конечным числом наследников, их код представлен в листинге 31.

События DelaAction отправляются на View слой и обрабатываются там. В листинге 32 представден пример обработки данных событий во View. В данном примере из ViewModel приходит событие после нажатия на элемент списка, в результате которого необходимо перейти на экран деталей соответствующего события.

Листинг 31: Код DelaEvent и DelaAction.

```
sealed interface DelaAction {
    data class OpenDetails(val id: Int) : DelaAction
}

sealed interface DelaEvent {
    object Refresh : DelaEvent
    object DeloOpen : DelaEvent
    data class DeloClick(val id: Int) : DelaEvent
}
```

Листинг 32: Пример обработки Action в View

```
@Composable
fun ListScreen() {
    StoredViewModel(factory = { DelaListViewModel() }) { viewModel ->
        val state = viewModel.viewStates().observeAsState()
        val action = viewModel.viewActions().observeAsState()
        val rootController = LocalRootController.current

        ListView(
            state = state.value,
            onClick = { viewModel.obtainEvent(DelaEvent.DeloClick(it)) }
        )

        when(val action = action.value) {
            is DelaAction.OpenDetails -> {
                val rc = rootController.findRootController()
                rc.present(NavTree.Details.Details.name, params =
                ↪  action.id)
                viewModel.obtainEvent(DelaEvent.DeloOpen)
            }
            null -> {}
        }
    }
}
```

3.4 Итоговое описание приложения

В результате работы было разработано клиентское приложение, имеющее общий код на Kotlin, использующее HTTP и Rest Api для сетевого взаимодействия и сохраняющее данные локально.

Для сетевых запросов на предоставленном API используется Bearer аутентификация, в связи с чем необходимо сохранять токен для HTTP запросов. Однако, не реализован механизм для Refresh токенов, из-за чего перед каждым сетевым запросом необходимо проверять текущий сохранённый токен на валидность. В случае истечения срока действия токена, повторно отправляется запрос авторизации с сохранёнными данными для входа, вследствии чего обновляется токен.

На главном экране приложения выводится список Новомученников на текущий день. Все данные, получаемые по запросу на главном экране, сохраняются в базе данных. В случае отсутствия интернета выводятся список из всех сохранённых ранее данных, вне зависимости от даты.

При нажатии на элемент списка отправляется запрос на получение деталей про конкретного человека. Предварительно проверяется валидности токена, как уже было сказано ранее. На экране деталей выводятся фамилия, имя и отчество новомученника, дата смерти и ключевые записанные события из базы данных электронно биографического справочника собранного на основе базы данных «За Христа пострадавшие».

4 Тестирование приложения

В данной главе будет описано тестирование сборки приложений с использованием инструментов Github Actions. Также будет представлен результат запуска приложения на платформах iOS, Android и Desktop

4.1 Проверка сборки в Github Actions

Для управления версиями данного проекта используется сервис Github. Благодаря использованию этого сервиса, возможно автоматизировать некоторые процессы, связанные с проектом. Например, возможно использовать Github Actions для автоматического запуска определенных задач при отправке нового кода. Для проверки сборки и распространения сборок в Telegram было реализовано несколько пайплайнов.

Когда происходят изменения в рамках Pull Request или коммит в одну из главных веток (dev, master), происходит автоматическая сборка проекта Android и загрузка `.apk` файла в канал Telegram (debug-сборка для Pull Request, release — для dev и master). Также запускается сборка Desktop-приложения без отправки артефакта, для проверки сборки. Обе сборки запускаются с помощью gradle задачи. Однако, чтобы выполнить сборку iOS-проекта, необходимо выполнить дополнительные шаги, из-за чего сборка iOS на CI не производится.

В листинге 36 представлен код для сборки Android приложения на Pull Request в главные ветки (`dev` и `master`). Сборка Desktop-приложения производится по аналогии, однако из кода удалены части, отвечающие за создание версии и загрузку файла в Telegram. Также аналогичным образом реализована сборка и загрузка приложений из веток `dev` и `master`.

На рисунке 5 представлен пример загрузки приложения в Telegram-канал. По аналогии возможно настроить загрузку приложения в любой другой источник, включая загрузку сразу в магазины приложений для распространения новых версий.

4.2 Снимка экранов страниц под iOS Android Desktop

На рисунках 1, 2 представлен результат запуска приложения на платформах iOS, Android, Desktop.

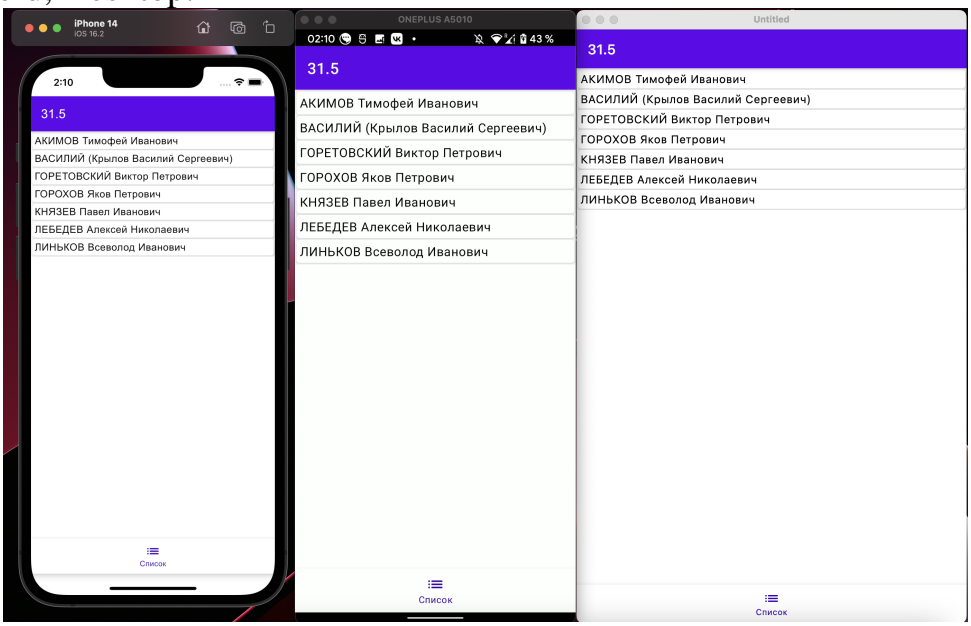

Рисунок 1 — Экран списка на iOS, Android, Desktop

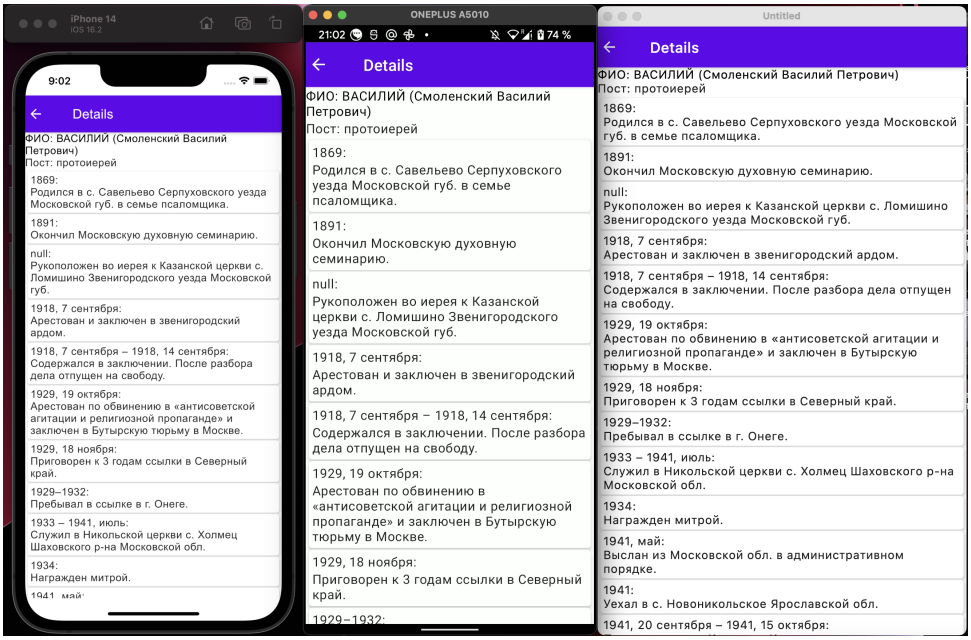

Рисунок 2 — Экран деталей на iOS, Android, Desktop

ЗАКЛЮЧЕНИЕ

В данной работе был произведён детальный обзор мультиплатформенной разработки и сравнение различных технологий в этой области, таких как нативная разработка, Xamarin, React Native, Flutter и Kotlin Multiplatform. В результате анализа в качестве наиболее подходящей технологии для данного исследования был выбран Kotlin Multiplatform.

Были раскрыты преимущества Kotlin Multiplatform, включая поддержку различных платформ (JVM/Android, Native, JS и WebAssembly) и особенности компиляции под разные целевые платформы. Данный подход позволяет максимально использовать преимущества разработки на одном языке и обеспечивает высокую производительность и кросс-платформенность приложений.

Был произведён разбор процесса разработки приложения, включая настройку проекта, выбор и подключение библиотек, использование convention plugins, организацию общего и платформенного кода. Важно отметить использование механизма "ожидание-актуализация"(expect-actual), который позволяет более гибко управлять разработкой для различных платформ.

На практике было демонстрировано написание общего кода, включая настройку DI в проекте, сетевые запросы с помощью Ktor, использование SQLDelight для работы с базой данных, создание графического пользовательского интерфейса с навигацией, сохранение простых данных, а также была описана архитектура приложения.

Итак, данная работа подтвердила эффективность Kotlin Multiplatform для разработки мультиплатформенных приложений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Sharma A. Android app development using Kotlin programming language //. T. 2427. — 02.2023. — С. 020099. — DOI: 10.1063/5.0130782.
2. Olaoye A. Basics of iOS Application Development //. — 02.2022. — С. 1—23. — DOI: 10.1007/978-1-4842-8023-2_1.
3. Что такое Xamarin? // Средства разработчика, техническая документация и примеры кода. [Электронный ресурс]. — URL: https://docs.microsoft.com/ru-ru/xamarin/get-started/what-is-xamarin.
4. Hermes D. Xamarin Mobile Application Development. — 2015. — Янв. — DOI: 10.1007/978-1-4842-0214-2.
5. Johnson P. Using Xamarin Forms //. — 01.2018. — С. 157—181. — ISBN 978-1-4842-2474-8. — DOI: 10.1007/978-1-4842-2475-5_8.
6. Introducing Hot Reloading // React Native. [Электронный ресурс]. — URL: https://reactnative.dev/blog/2016/03/24/introducing-hot-reloading.
7. Platform // React Native. [Электронный ресурс]. — URL: https://reactnative.dev/docs/platform.
8. Analyzing the Performance of Apps Developed by using Cross-Platform and Native Technologies / L. Barros [и др.] //. — 12.2020.
9. Hot reload // Flutter. [Электронный ресурс]. — URL: https://docs.flutter.dev/development/tools/hot-reload.
10. Material Design for Flutter. [Электронный ресурс]. — URL: https://docs.flutter.dev/development/ui/material.
11. Cupertino (iOS-style) widgets // Flutter. [Электронный ресурс]. — URL: https://docs.flutter.dev/development/ui/widgets/cupertino.
12. Compose Multiplatform for iOS [Электронный ресурс]. — URL: https://github.com/JetBrains/compose-multiplatform/blob/d44114d/README.md#ios.
13. Kotlin for Android [Электронный ресурс]. — URL: https://kotlinlang.org/docs/android-overview.html.
14. Kotlin for server side [Электронный ресурс]. — URL: https://kotlinlang.org/docs/server-overview.html.

15. Kotlin Native [Электронный ресурс]. — URL: https://kotlinlang.org/docs/native-overview.html.

16. Kotlin for JavaScript [Электронный ресурс]. — URL: https://kotlinlang.org/docs/js-overview.html.

17. Get started with Kotlin/Wasm in IntelliJ IDEA [Электронный ресурс]. — URL: https://kotlinlang.org/docs/wasm-get-started.html.

18. Connect to platform-specific APIs [Электронный ресурс]. — URL: https://kotlinlang.org/docs/multiplatform-connect-to-apis.html.

19. Система сборки Gradle [Электронный ресурс]. — URL: https://gradle.org.

20. Денисенко А. А. Организация многомодульной, слабосвязанной архитектуры приложения при работе с Gradle // Технические науки: теория и практика : материалы IV Междунар. науч. конф. (г. Казань, ноябрь 2018 г.) — Казань : Общество с ограниченной ответственностью Издательство Молодой ученый, 2018. — С. 7—10. — URL: https://moluch.ru/conf/tech/archive/312/14585/.

21. Gradle version catalog [Электронный ресурс]. — URL: https://docs.gradle.org/current/userguide/platforms.html.

22. Ktor framework [Электронный ресурс]. — URL: https://ktor.io.

23. SQLDelight Overview [Электронный ресурс]. — URL: https://cashapp.github.io/sqldelight/2.0.0-alpha05/.

24. Multiplatform Settings [Электронный ресурс]. — URL: https://github.com/russhwolf/multiplatform-settings.

25. Kodein [Электронный ресурс]. — URL: https://kosi-libs.org/kodein/7.19/index.html.

26. Odyssey [Электронный ресурс]. — URL: https://github.com/AlexGladkov/Odyssey.

27. KViewModel-mpp [Электронный ресурс]. — URL: https://github.com/adeo-opensource/kviewmodel--mpp.

ПРИЛОЖЕНИЕ А

Листинг 33: Файл build.gradle.kts umbrella модуля.

```
plugins {
    id("mp-compose-setup")
    id("android-setup")
    kotlin("native.cocoapods")
}
version = "0.0.1"
kotlin {
    cocoapods {
        summary = "MPP iOS SDK + Compose"
        homepage = "https://google.com"
        ios.deploymentTarget = "14.0"
        podfile = project.file("../../apps/iosApp/Podfile")
        framework {
            transitiveExport = false
            isStatic = true
            baseName = "shared_compose"
            freeCompilerArgs += "-Xbinary=bundleId=ru.andvl.mppapp"
            export(projects.common.core)
            export(projects.common.coreUtils)
            export(projects.common.coreCompose)
            export(projects.common.auth.data)
            export(projects.common.main.compose)
            export(projects.common.list.data)
            export(projects.common.list.compose)
            export(projects.common.umbrellaCore)
        }
    }
    sourceSets {
        commonMain {
            dependencies {
                implementation(projects.common.core)
                implementation(projects.common.coreUtils)
                implementation(projects.common.coreCompose)

                implementation(projects.common.auth.data)

```

```
                implementation(projects.common.main.compose)

                implementation(projects.common.list.data)
                implementation(projects.common.list.compose)

                implementation(projects.common.umbrellaCore)

                implementation(libs.kViewModel.core)
                implementation(libs.kViewModel.compose)
                implementation(libs.kViewModel.odyssey)

                implementation(libs.odyssey.core)
                implementation(libs.odyssey.compose)
            }
        }
        androidMain {
            dependencies {
                implementation(project.dependencies.platform(
                    libs.androidx.compose.bom
                ))
                implementation(libs.compose.activity)
            }
        }
        iosMain {
            dependencies {
                api(projects.common.core)
                api(projects.common.coreUtils)
                api(projects.common.coreCompose)
                api(projects.common.auth.data)
                api(projects.common.main.compose)
                api(projects.common.list.data)
                api(projects.common.list.compose)
                api(projects.common.umbrellaCore)
            }
        }
    }
}
```

ПРИЛОЖЕНИЕ Б

Листинг 34: Convention plugin для модулей с графическим пользовательским интерфейсом.

```
plugins {
    id("com.android.library")
    kotlin("multiplatform")
    id("org.jetbrains.compose")
}

kotlin {
    jvm("desktop")
    android()
    ios()
    iosSimulatorArm64()

    sourceSets {
        val iosSimulatorArm64Main by getting
        val iosSimulatorArm64Test by getting
        val commonMain by getting {
            dependencies {
                implementation(compose.runtime)
                implementation(compose.foundation)
                implementation(compose.material)
                implementation(compose.ui)
            }
        }

        named("desktopMain") {
            dependencies {
                implementation(compose.desktop.common)
            }
        }

        named("androidMain") {
            dependencies {
                implementation(project.dependencies.platform(
                    libs.androidx.compose.bom
                ))
```

```
36                  implementation(libs.compose.ui)
37                  implementation(libs.compose.material)
38                  implementation(libs.compose.tooling)
39                  implementation(libs.compose.icons)
40              }
41          }
42          val iosMain by getting {
43              dependsOn(commonMain)
44              iosSimulatorArm64Main.dependsOn(this)
45          }
46          val iosTest by getting  {
47              dependsOn(commonMain)
48              iosSimulatorArm64Test.dependsOn(this)
49          }
50      }
51
52      tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile> {
53          kotlinOptions.jvmTarget = "11"
54      }
55  }
```

ПРИЛОЖЕНИЕ В

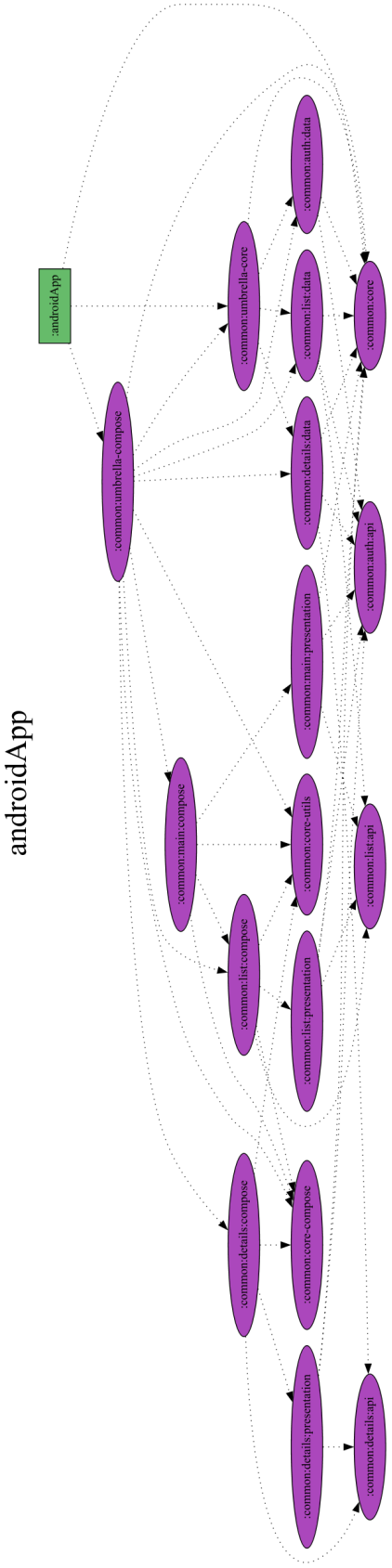

Рисунок 3 — Граф зависимостей проекта.

ПРИЛОЖЕНИЕ Г

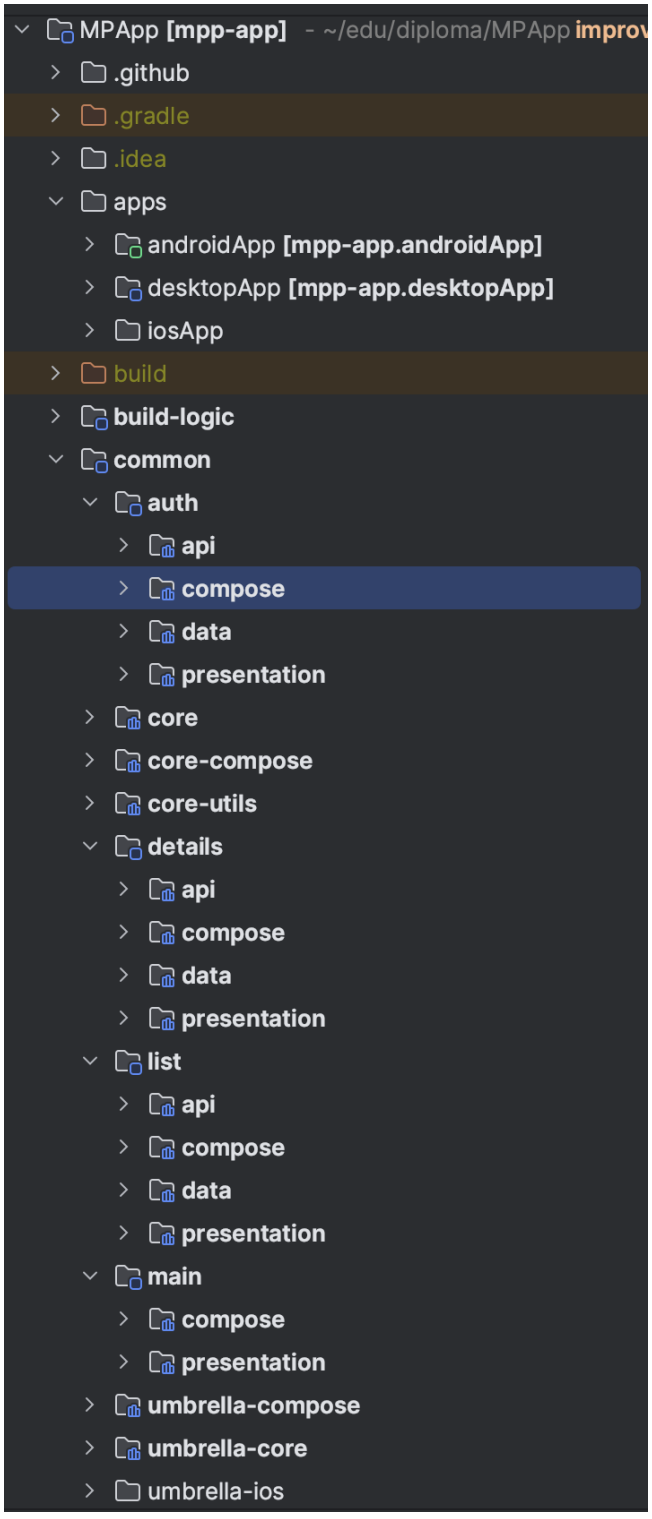

Рисунок 4 — Структура проекта.

ПРИЛОЖЕНИЕ Д

Листинг 35: Пример реализации ViewModel.

```
class DelaListViewModel : BaseSharedViewModel<DelaViewState, DelaAction,
↪  DelaEvent>(
    initialState = DelaViewState()
) {
    private val authRepository: AuthRepository = Inject.instance()
    private val delaRepository: DelaListRepository = Inject.instance()

    init {
        viewModelScope.launch {
            viewState = viewState.copy(isLoading = true)
            checkLogin()
            getDela()
        }
    }

    override fun obtainEvent(viewEvent: DelaEvent) {
        when (viewEvent) {
            is DelaEvent.Refresh -> {
                viewState = viewState.copy(isLoading = true, dela = null,
                ↪  isError = false)
                viewModelScope.launch {
                    getDela()
                }
            }
            is DelaEvent.DeloClick -> {
                viewAction = DelaAction.OpenDetails(viewEvent.id)
            }
            is DelaEvent.DeloOpen -> {
                viewAction = null
            }
        }
    }

    private suspend fun checkLogin() {
        // поход в authRepository для проверки авторизации пользователя
    }
```

```
35
36      private suspend fun getDela() {
37          val now = Clock.System.now()
38          val datetime =
            ↪  now.toLocalDateTime(TimeZone.currentSystemDefault())
39          val dela = delaRepository.getDelas(token =
            ↪  authRepository.getToken())
40          viewState = if (dela != null) {
41              viewState.copy(
42                  isLoading = false,
43                  dela = dela,
44                  date = "${datetime.dayOfMonth}.${datetime.monthNumber}"
45              )
46          } else {
47              viewState.copy(isLoading = false, isError = true)
48          }
49      }
50  }
```

ПРИЛОЖЕНИЕ Е

Листинг 36: Сборка Android приложения на Pull Request.

```yaml
name: Build Android Pull request

on:
  pull_request:
    branches:
      - 'dev'
      - 'master'

env:
  GITHUB_REF: github.ref

jobs:
  build:
    name: "Build apk"
    runs-on: ubuntu-latest
    outputs:
      build_number: ${{ steps.buildnumber.outputs.build_number }}
    steps:
      - uses: actions/checkout@v2
        with:
          submodules: 'recursive'
      - name: Set up JDK 1.11
        uses: actions/setup-java@v2
        with:
          distribution: 'temurin'
          java-version: '11'
      - name: Generate build number
        id: buildnumber
        uses: onyxmueller/build-tag-number@v1
        with:
          token: ${{secrets.github_token}}

      - name: Gradle Wrapper Validation
        uses: gradle/wrapper-validation-action@v1

      - uses: actions/cache@v2
```

```
      with:
        path: |
          ~/.gradle/caches
          ~/.gradle/wrapper
        key: ${{ runner.os }}-gradle-${{ hashFiles('**/*.gradle*',
↪  '**/gradle-wrapper.properties') }}
        restore-keys: |
          ${{ runner.os }}-gradle-
    - name: 'Set variables'
      id: vars
      run: |
        export $(cat .github/workflows/version.env | xargs)
        echo "::set-output name=major_version::${MAJOR_VERSION}"
        echo ${{ steps.buildnumber.outputs.build_number }} >
↪  version.txt
    - name: Build dev
      run: |
        ./gradlew generateCommonMainDatabaseInterface
↪  :androidApp:assembleDebug \
          -Dversion_code=${{ steps.buildnumber.outputs.build_number }}
↪  \
          -Dversion_name="${{ steps.vars.outputs.major_version }}" \
    - name: Copy artifacts
      id: artifacts_copy
      run: |
        mkdir artifacts
        cp version.txt artifacts/version.txt
        cp apps/androidApp/build/outputs/apk/debug/androidApp-debug.apk
↪  artifacts/app-debug-`cat version.txt`.apk
        echo "::set-output name=path::artifacts/"
    - name: Upload Artifacts
      uses: actions/upload-artifact@v2
      with:
        name: artifacts
        path: ${{ steps.artifacts_copy.outputs.path }}
  upload-to-tg:
    name: Upload to tg channel
    runs-on: ubuntu-latest
    needs: build
    steps:
```

```
72      - uses: actions/checkout@v2
73      - uses: actions/download-artifact@v2
74        id: download
75        with:
76          name: artifacts
77      - name: Get version number
78        id: version-num
79        run: |
80          echo "::set-output name=num::`cat
    ↪ ${{steps.download.outputs.download-path}}/version.txt`"
81      - name: Upload to Telegram
82        uses: appleboy/telegram-action@master
83        with:
84          to: ${{ secrets.TELEGRAM_TO }}
85          token: ${{ secrets.TELEGRAM_TOKEN }}
86          message: |
87            New commit in pull request #${{
    ↪ github.event.pull_request.number }}: ${{
    ↪ github.event.pull_request.title }}
88            View: https://github.com/AndVl1/mpp-diploma-app/pull/${{
    ↪ github.event.pull_request.number }}
89          document: ${{ steps.download.outputs.download-path
    ↪ }}/app-debug-${{ steps.version-num.outputs.num }}.apk
90          disable_notification: true
91
```

ПРИЛОЖЕНИЕ Ж

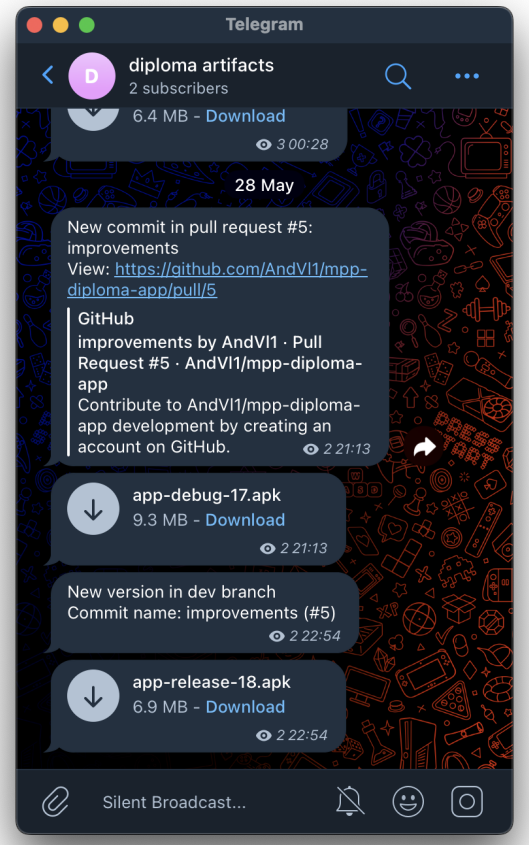

Рисунок 5 — Пример загруженных в Telegram Android приложений из Pull Request и из ветки dev.