МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой, д.ф.-м.н.,
профессор

_____ Л.Б. Соколинский

«___»_____ 2025 г.

Разработка кроссплатформенной библиотеки BDUI с использованием Kotlin Multiplatform

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 02.03.02.2025.308-002.ВКР

Научный руководитель,
ст. преподаватель кафедры СП
_____ Л.Н. Петрова

Автор работы,
студент группы КЭ-401
_____ М.В. Бабушкин

Ученый секретарь
(нормоконтролер)
_____ И.Д. Володченко
«___»_____ 2025 г.

Челябинск, 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

УТВЕРЖДАЮ
Зав. кафедрой СП
_____ Л.Б. Соколинский
27.01.2025 г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра
студенту группы КЭ-401
Бабушкину Михаилу Вадимовичу,
обучающемуся по направлению
02.03.02 «Фундаментальная информатика и информационные технологии»

1. **Тема работы** (утверждена приказом ректора от 21.04.2025 г. № 648-13/12) Разработка кроссплатформенной библиотеки BDUI с использованием Kotlin Multiplatform.
2. **Срок сдачи студентом законченной работы:** 02.06.2025 г.
3. **Исходные данные к работе**

3.1. Kotlin Multiplatform by Tutorials (Second Edition): Build Native Apps for iOS and Android with Kotlin Multiplatform. Пер. с англ. СПб.: Питер, 2029. 550 с. ISBN 978-5-4461-2910-2. ББК 32.973.26-018.

3.2. Исакова С., Жемеров Д. Kotlin в действии. Пер. с англ. СПб.: Питер, 2018. 624 с. ISBN 978-5-4461-0923-4. ББК 32.973.26-018.1.

3.3. Москала М. Effective Kotlin: Best practices. Пер. с англ. СПб.: Питер, 2024. 400 с. ISBN 978-5-4461-2155-7. ББК 32.973.26-018.1.

3.4. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. СПб.: Питер, 2018. – 352 с.: ил. – (Серия «Библиотека программиста»). ISBN 978-5-4461-0772-8.

4. **Перечень подлежащих разработке вопросов**

4.1. Провести анализ предметной области и обзор аналогов.

4.2. Определить требования к проектируемой системе, включая детализированные функциональные и нефункциональные требования.

4.3. Спроектировать архитектуру системы.

4.4. Реализовать систему, обеспечив кроссплатформенную поддержку с использованием Kotlin Multiplatform.

4.5. Провести тестирование системы.

5. **Дата выдачи задания:** 27.01.2025 г.

Научный руководитель,
ст. преподаватель кафедры СП Л.Н. Петрова

Задание принял к исполнению М.В. Бабушкин

ГЛОССАРИЙ

1. Backend-Driven UI (BDUI) [1] – это архитектурный подход к построению пользовательского интерфейса, при котором основное управление его структурой осуществляется на стороне сервера. В отличие от традиционных клиент-ориентированных интерфейсов, где UI жестко задан в коде приложения, в BDUI клиент получает структуру и содержимое интерфейса в виде конфигурационных данных, обычно представленных в формате JSON или XML. Это позволяет динамически изменять внешний вид и функциональность приложения без необходимости выпуска обновлений.

2. Kotlin Multiplatform (KMP) [2] – это фреймворк для разработки кроссплатформенных приложений, позволяющий использовать один код на различных платформах, таких как Android, iOS, Web и JVM. Основная цель KMP – предоставить разработчикам возможность делиться логикой приложения между различными платформами, что значительно ускоряет процесс разработки и уменьшает количество повторяющегося кода.

3. Compose Multiplatform (CMP) [3] – это фреймворк для создания пользовательских интерфейсов с использованием декларативного подхода, который позволяет разрабатывать UI на различных платформах, включая Android, iOS, Web и Desktop, с единым кодом. Он является частью экосистемы Kotlin Multiplatform и предоставляет инструменты для создания кроссплатформенных приложений с использованием одной кодовой базы.

4. Gradle [4] – это современная система автоматической сборки проектов, основанная на концепции декларативного управления зависимостями и конфигурациями сборки. Gradle широко используется в экосистеме Kotlin и Android-разработки, а также поддерживает проекты на различных языках программирования, включая Java, Groovy, Scala и C++.

5. Система контроля версий (VCS) [5] – это программный инструмент, предназначенный для управления изменениями в исходном коде, совместной работы разработчиков и отслеживания истории изменений в проекте. VCS позволяет восстанавливать предыдущие версии файлов, работать

с несколькими ветками разработки и эффективно интегрировать код в рамках командной разработки.

6. Continuous Integration (CI) [6] – это практика разработки программного обеспечения, при которой разработчики регулярно объединяют свои изменения кода в центральный репозиторий, после чего автоматически выполняются сборка и тестирование. Основная цель CI – быстрое обнаружение и исправление ошибок, улучшение качества программного обеспечения и сокращение времени, необходимого для проверки и выпуска новых обновлений.

7. Continuous Delivery (CD) [7] – это подход, при котором команды разрабатывают программное обеспечение таким образом, что оно может быть выпущено в любое время. При непрерывной доставке каждое изменение кода автоматически проходит через конвейер сборки и тестирования и готовится к выпуску в производственную среду, хотя фактическое развертывание в производство может требовать ручного одобрения.

8. Внедрение зависимостей (Dependency Injection) [8] – это программный шаблон проектирования, при котором объект получает свои зависимости извне, а не создаёт их самостоятельно. Данный подход способствует ослаблению связности компонентов системы, улучшая модульность, тестируемость и расширяемость программного кода.

9. Архитектура MVI (Model–View–Intent) [9] – это архитектурный шаблон проектирования пользовательского интерфейса, основанный на однонаправленном потоке данных и управлении состоянием.

10. DSL (Domain-Specific Language) [10] – это язык программирования или описание, специально разработанный для решения задач в конкретной предметной области. DSL может создаваться на базе существующего языка программирования с использованием его синтаксических и семантических возможностей.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность

С увеличением количества платформ для пользовательских приложений – таких как iOS, Android, Web и десктопные операционные системы – и ограниченными ресурсами на их поддержку, всё большее внимание привлекает концепция Backend-Driven UI. Это подход, при котором интерфейс приложения формируется на сервере и передаётся клиенту в виде описания, например, JSON-файла. Такой механизм позволяет значительно сократить время выхода обновлений, снизить необходимость публикации новых версий в магазинах приложений и добиться высокой гибкости в управлении интерфейсом.

Особенно актуален BDUI для бизнес-приложений, в которых интерфейс должен часто меняться: подстраиваться под новые требования, поддерживать A/B-тесты и персонализацию. В таких условиях достаточно просто внести изменения на серверной стороне – и клиентское приложение мгновенно отобразит новый интерфейс без необходимости обновления.

Следовательно, создание кроссплатформенной библиотеки для реализации концепта BDUI представляет собой значимую задачу, способствующую ускорению разработки приложений, уменьшению затрат на реализацию и улучшению адаптивности и персонализации пользовательских интерфейсов, что полностью соответствует актуальным требованиям к мобильным и десктопным приложениям**.**

Постановка задачи

Цель данной выпускной квалификационной работы заключается в создании кроссплатформенной библиотеки, которая реализует концепцию Backend-Driven UI с применением технологии Kotlin Multiplatform. Библиотека будет предназначена для предоставления инструментов, позволяющих рендерить пользовательский интерфейс на основании JSON-описания, и должна поддерживать функционирование на различных платформах: Web, Android и Desktop.

Для достижения поставленной цели необходимо решить ряд задач, описанных ниже.

1. Провести анализ предметной области. Необходимо изучить концепцию BDUI, ознакомиться с подходами к проектированию кроссплатформенных библиотек, рассмотреть существующие решения, реализующие похожую функциональность.

2. Определить требования к проектируемой системе, включая детализированные функциональные и нефункциональные требования. Необходимо спроектировать архитектуру библиотеки, описать ключевые компоненты системы и спроектировать механизм рендеринга интерфейсов на основе JSON-описания.

3. Реализовать библиотеку, обеспечив кроссплатформенную поддержку с использованием Kotlin Multiplatform. Необходимо настроить инструменты разработки, такие как систему контроля версий, систему сборки проекта, системы непрерывной интеграции и развертывания библиотеки.

4. Разработать тестовый стенд для проверки функциональности библиотеки. Необходимо провести функциональное, модульное и интеграционное тестирование решения.

5. Проанализировать результаты работы. Необходимо оценить процессы выполнения работы на каждом из этапов, а также рассмотреть потенциал для возможности дальнейшего использования, расширения и развития библиотеки BDUI.

Структура и содержание работы

Данная работа состоит из глоссария, введения, четырех глав, заключения, списка литературы и приложений. Объем работы составляет 60 страниц, объем списка литературы – 20 источников.

В первой главе проводится анализ предметной области. Рассматриваются концепции Backend-Driven UI и кроссплатформенной разработки, осуществляется обзор существующих решений в области динамического построения пользовательского интерфейса. Формулируются выводы и

обоснование выбора технологии Kotlin Multiplatform для реализации данной задачи.

Вторая глава включает формирование требований к системе, функциональных и нефункциональных свойств библиотеки. Разрабатывается архитектура библиотеки BDUI, включая описание механизма рендеринга интерфейсов на основе JSON-описания.

В третьей главе подробно описан процесс разработки кроссплатформенной библиотеки BDUI. Рассматриваются используемые программные средства, настройка Gradle, система контроля версий, непрерывной интеграции и развертывания. Дается описание этапов реализации механизма рендеринга экрана и взаимодействия с JSON-описанием.

В четвертой главе представлено описание процедур тестирования библиотеки. Рассматриваются функциональное, модульное и интеграционное тестирование, а также приводятся результаты их проведения.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Предметная область проекта

Современные пользовательские интерфейсы требуют высокой гибкости, способности быстро адаптироваться к постоянно меняющимся бизнес-требованиям, а также поддержки различных платформ – мобильных, десктопных и веб-приложений. Всё это делает необходимым создание эффективных инструментов, позволяющих легко и быстро изменять внешний вид приложения с минимальными затратами времени как для всей системы в целом, так и для каждой платформы в отдельности.

На этом фоне в последние годы активно развивается подход, известный как Backend-Driven UI. Суть его в том, что сервер отправляет клиенту структурированное описание пользовательского интерфейса – например, в формате JSON. Клиентское приложение затем интерпретирует это описание и отображает соответствующий интерфейс. Такой подход позволяет перенести управление интерфейсом на серверную часть, что даёт возможность мгновенно обновлять внешний вид приложения без необходимости повторной публикации в магазинах приложений.

Подход BDUI используется в различных сферах, описанных ниже.

1. Быстроразвивающийся бизнес. К примеру, в направлении e- commerce, где необходимо часто обновлять внешний вид продукта по запросу от бизнеса: добавлять баннеры, акции, изменять структуру страницы – при этом без выпуска новых версий приложения, поскольку процесс обновления приложения увеличивает метрику time-to-market, что не всегда нравится заказчику.

2. A/B-тестирование. Это метод маркетингового исследования, позволяющий на основе статистики оценить влияние изменения на метрики продукта. В рамках тестов пользователи разбиваются на две группы, и одной из этих групп показывается изменение в рамках контролируемого эксперимента. Для того чтобы проводить A/B тест необходимо иметь возможность

изменять структуру пользовательского интерфейса для разных групп пользователей «на лету».

3. Персонализированный контент. Это способ удержания интереса клиентов за счет создания уникального пользовательского опыта для каждого из них, посредством предоставления каждому пользователю уникального формата продукта с учётом пользовательских предпочтений, поведения, истории взаимодействия с брендом или других данных. Цель персонализации – сделать контент максимально релевантным и полезным для каждого клиента.

Таким образом, создание кроссплатформенной библиотеки, реализующей концепцию BDUI, позволит разработчикам унифицировать подход к реализации пользовательских интерфейсов и упростить управление приложениями.

1.2. Анализ аналогичных проектов и существующих решений

В рамках данного анализа будут рассмотрены как коммерческие, так и открытые решения для построения BDUI, их архитектурные особенности, поддерживаемые платформы, возможности расширения и кастомизации, а также производительность и удобство использования.

Результаты данного анализа послужат фундаментом для формирования требований к разрабатываемой библиотеке и выбора оптимальных архитектурных решений в последующих главах работы.

DivKit

DivKit [11] представляет собой открытую библиотеку для создания пользовательских интерфейсов с использованием подхода Backend-Driven UI, разработанную компанией Яндекс. Компания активно развивает собственную экосистему приложений и сервисов, для которых первоначально и была создана библиотека DivKit. Впоследствии решение было открыто для широкого использования и опубликовано как проект с открытым исходным кодом.

Функционал DivKit охватывает широкий спектр возможностей для построения динамических пользовательских интерфейсов. Библиотека позволяет описывать интерфейс в формате JSON, что обеспечивает полный контроль сервера над отображением контента. DivKit поддерживает основные платформы: iOS, Android и Web, используя единый формат описания интерфейса. Важной особенностью является возможность динамического обновления интерфейса без необходимости обновления самого приложения. Библиотека предоставляет механизмы для создания кастомных компонентов и расширения базовой функциональности, поддерживает описание и воспроизведение различных анимаций элементов интерфейса. В DivKit реализована система выражений, обеспечивающая условную логику и вычисления на стороне клиента, а также встроенные инструменты для тестирования и отладки интерфейсов. На рисунке 1 представлен функционал онлайн-редактора DivKit.

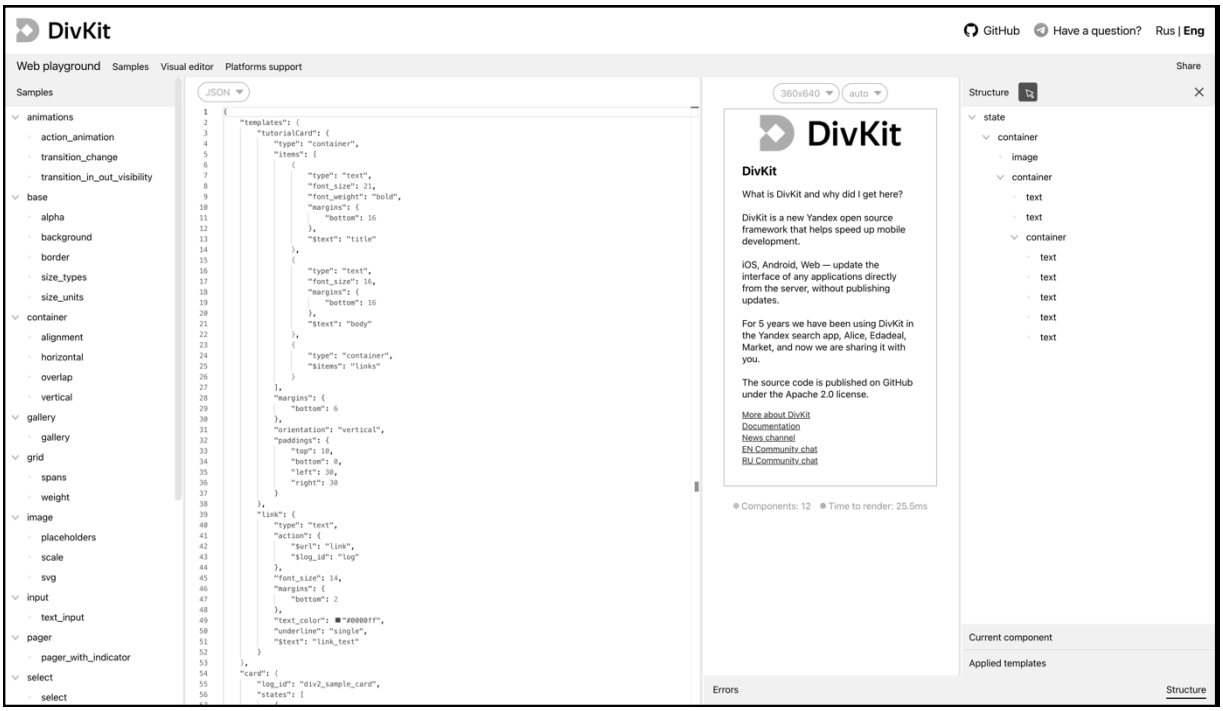

Рисунок 1 – Онлайн-редактор DivKit

Архитектура DivKit основана на нескольких ключевых принципах. В основе лежит компонентная архитектура, где интерфейс строится из набора предопределенных компонентов, таких как контейнеры, тексты,

изображения и другие элементы. Компоненты образуют иерархическое дерево, аналогичное DOM в веб-разработке. Важным архитектурным решением является разделение описания и отрисовки: JSON-описание преобразуется в промежуточное представление, которое затем отрисовывается нативными средствами платформы. В библиотеке реализована система стилей, обеспечивающая поддержку переиспользуемых стилей и тем для унификации внешнего вида. Также присутствует система состояний, позволяющая компонентам иметь различные состояния, влияющие на их отображение.

Механизм отрисовки экрана в DivKit представляет собой многоэтапный процесс. Сначала происходит парсинг входящего JSON-документа и преобразование его в объектную модель DivData. Затем выполняется валидация для проверки корректности структуры и параметров компонентов. На следующем этапе строится виртуальное дерево, создающее промежуточное представление интерфейса DivView. После этого производится расчет layout – вычисление размеров и позиций элементов с учетом ограничений экрана. Далее осуществляется рендеринг – преобразование виртуального дерева в нативные UI-компоненты соответствующей платформы.

Технологический стек DivKit включает различные технологии в зависимости от целевой платформы. Для Android используется система View из Android SDK. Для iOS применяются UIKit. Веб-версия базируется на TypeScript/JavaScript и DOM-элементах HTML/CSS. В отличие от разрабатываемой в рамках данной работы библиотеки, DivKit не использует Kotlin Multiplatform для обеспечения кроссплатформенности, а вместо этого имеет отдельные реализации для каждой платформы с общим форматом описания интерфейса, что создает определенные сложности при поддержке и расширении функциональности.

Redwood

Redwood [12] – это современная библиотека для создания пользовательских интерфейсов, разработанная командой Cash App. Сам Cash App, ранее известный как Square Cash – это популярное мобильное приложение для

денежных переводов и инвестиций, разработанное компанией Block, Inc. основанной Джеком Дорси, также известным как сооснователь Twitter. Cash App занимает одну из лидирующих позиций среди финтех-продуктов в США и позволяет пользователям отправлять и получать деньги, инвестировать в акции и криптовалюты, а также выполнять банковские операции. Команда, стоящая за приложением, славится высоким качеством кода и инновационными подходами к разработке.

Библиотека Redwood предназначена для создания декларативных пользовательских интерфейсов и построена на архитектурных принципах, близких к концепции Backend-Driven UI. Она предоставляет функционал описания интерфейсов в виде иерархической структуры, которую можно сериализовать и передавать между различными частями приложения, включая передачу с сервера на клиент.

Библиотека Redwood поддерживает создание сложных интерфейсов с вложенными компонентами, обработку пользовательских событий и состояний, а также предоставляет возможности для стилизации UI-элементов. Одним из ключевых преимуществ библиотеки является возможность использовать одно и то же описание интерфейса на разных платформах, что значительно упрощает кроссплатформенную разработку и поддержку. Кроме того, библиотека включает инструменты для тестирования и отладки компонентов пользовательского интерфейса.

Архитектура Redwood основана на ряде ключевых принципов. Центральной является модель «виджет – модификатор», в которой виджеты служат базовыми строительными блоками интерфейса, а модификаторы изменяют их поведение и внешний вид. Это похоже на подходы, используемые в SwiftUI и Jetpack Compose. Важным решением является разделение платформенно-независимого слоя описания UI и платформенно-специфичных рендереров. Также реализована концепция однонаправленного потока данных, при которой любые изменения в данных автоматически приводят к

обновлению интерфейса. Redwood эффективно обновляет только изменённые части UI, что положительно сказывается на производительности.

Процесс рендеринга в Redwood проходит в несколько этапов. Сначала создаётся виртуальное дерево виджетов, описывающее интерфейс. Далее происходит этап layout-компоновки, где рассчитываются размеры и позиции каждого элемента. Затем виртуальное дерево преобразуется в нативные UI-компоненты соответствующей платформы: на Android – это Jetpack Compose, на iOS – SwiftUI или UIKit. При изменении данных Redwood создаёт новое виртуальное дерево, сравнивает его с предыдущим и обновляет только те части интерфейса, которые действительно изменились. Это напоминает подход Virtual DOM, популярный в веб-разработке.

Технологический стек Redwood базируется на Kotlin Multiplatform, что позволяет использовать один и тот же код на разных платформах. На Android библиотека работает с Kotlin и Jetpack Compose [13], на iOS – с Kotlin/Native [14] и SwiftUI или UIKit. В коде активно применяются корутины для асинхронных операций, делегированные свойства для управления состоянием, а также DSL для описания интерфейсов. Для сериализации используется kotlinx.serialization. Сборка проекта организована с помощью Gradle с поддержкой мультиплатформенных проектов. Redwood делает акцент на максимальном использовании общего кода и минимизации платформенно-специфичных решений, что отличает её от многих других решений в этой области.

Lona

Lona [15] – это инструмент для создания и сопровождения дизайн-систем, разработанный компанией Airbnb. Airbnb – одна из крупнейших платформ краткосрочной аренды жилья в мире, основанная в 2008 году. Компания известна своим вниманием к дизайну и пользовательскому опыту, что побудило её команду разработать собственные инструменты, упрощающие процесс создания интерфейсов. Возникшая потребность в согласованности

дизайна на разных платформах и эффективном взаимодействии между дизайнерами и разработчиками стала основой для появления Lona.

Основная задача Lona – создание единой дизайн-системы, пригодной для использования на всех целевых платформах. Библиотека позволяет описывать UI-компоненты в декларативном формате, который затем автоматически преобразуется в нативный код. В Lona можно задать как визуальные характеристики компонентов, так и их поведение: состояния, переходы и взаимодействие. Важным преимуществом является наличие визуального редактора – Lona Studio, позволяющего дизайнерам создавать и редактировать компоненты без написания кода. Также в систему встроены инструменты генерации документации и поддержки тем и стилей, обеспечивающие визуальную целостность интерфейса на всех уровнях приложения.

Архитектура Lona строится на принципе «компонент как единый источник истины». Компоненты описываются в формате JSON, который выступает в роли универсального описания интерфейса и служит основой для всех платформ. Такое решение позволяет избежать расхождений между реализациями компонентов на Android, iOS и Web. Особенность архитектуры заключается в разделении описания и реализации: сначала задаются компоненты в JSON, а затем с помощью генераторов кода они трансформируются в нативные реализации для каждой платформы.

Отличительной чертой Lona является то, что она не интерпретирует описание интерфейса в реальном времени, как это делают многие Backend-Driven UI-системы. Вместо этого она генерирует нативный код на этапе разработки. Процесс начинается с создания JSON-описания компонента, содержащего структуру, стили и логику. Затем генераторы создают нативные компоненты для каждой из платформ. Для iOS – Swift-код на основе UIKit или SwiftUI. Для Android – код на Kotlin с использованием View System или Jetpack Compose. Для Web – JavaScript или TypeScript на базе React или других фреймворков.

Сгенерированные компоненты интегрируются в приложение как обычные нативные элементы. Такой подход обеспечивает высокую производительность, поскольку устраняет необходимость в дополнительном слое интерпретации во время выполнения. Однако это требует перегенерации кода при каждом изменении в дизайне.

Технологический стек Lona включает в себя несколько ключевых компонентов. JSON – основной формат описания компонентов. Lona Studio – десктопное приложение для macOS, написанное на Swift. Генераторы кода – в основном реализованы на TypeScript. Реализована поддержка платформ: Swift (iOS), Kotlin (Android), TypeScript/JavaScript (Web). Система сборки основана на Node.js и npm.

В отличие от других решений, Lona делает ставку на генерацию нативного кода из общего описания, что позволяет максимально приблизиться к платформенным стандартам и обеспечить высокую производительность. Однако этот подход требует отдельной поддержки генераторов для каждой целевой платформы.

Выводы по первой главе

Рассмотренные библиотеки – Lona от Airbnb, Redwood от Cash App и DivKit от Яндекса – представляют собой специализированные решения, каждое из которых предлагает собственный подход к реализации концепции Backend-Driven UI. Все они нацелены на решение общей задачи: обеспечение единообразного и динамически обновляемого пользовательского интерфейса на различных платформах. При этом каждая из библиотек использует разные архитектурные принципы и технологии.

Lona делает ставку на генерацию нативного кода во время разработки. Это обеспечивает высокую производительность интерфейсов, но ограничивает возможность их динамического обновления во время работы приложения.

Redwood использует возможности Kotlin Multiplatform, что позволяет максимально переиспользовать код между платформами. Библиотека

реализует механизм создания виртуального дерева компонентов, благодаря чему эффективно обновляет только изменившиеся части UI, однако этот алгоритм более затратен по памяти, чем другие решения.

DivKit опирается на единый JSON-формат описания интерфейсов и имеет отдельные нативные реализации для каждой платформы. Это предоставляет большую гибкость, но усложняет поддержку и развитие системы.

Важно понимать, что каждое из этих решений имеет свои сильные стороны, а также определённые ограничения. Например, подход Lona ограничен в плане динамики – любые изменения требуют пересборки. В то же время, платформенно-зависимые реализации в DivKit повышают гибкость, но создают дополнительную нагрузку на сопровождение.

Создаваемая библиотека должна учитывать эти компромиссы и стремиться к более сбалансированному решению, которое объединяет лучшие практики существующих подходов.

В следующих разделах будет представлено подробное рассмотрение архитектурных решений и технических аспектов реализации библиотеки, основанное на анализе представленных инструментов.

2. ПРОЕКТИРОВАНИЕ

2.1. Требования к проектируемой системе

Проектируемая библиотека BDUI направлена на реализацию концепции Backend-Driven UI с использованием Kotlin Multiplatform [16]. Разработка данной системы требует формулирования функциональных требований, а также нефункциональных требований, учитывающих особенности применения библиотеки. Кроме того, для разработки функционала библиотеки необходимо разработать варианты использования и сформулировать их спецификацию.

Функциональные требования

В ходе проектирования библиотеки были сформированы следующие функциональные требования, описанные ниже.

1. Обработка JSON-описания интерфейсов. Библиотека должна принимать JSON-описание пользовательского интерфейса, которое сервер генерирует динамически. Это описание должно включать структуру пользовательского интерфейса, свойства элементов, стили и действия.

2. Динамический рендеринг пользовательского интерфейса. Библиотека должна генерировать интерфейсы на основе JSON-описания, используя компоненты платформы или Compose Multiplatform.

3. Поддержка кастомизации компонентов. В библиотека должна быть возможность добавлять пользовательские компоненты. Настройка должна происходить через расширения или наследование существующих компонентов.

4. Управление стилями интерфейса. Библиотека должна поддерживать передачу и применение базовых стилей, таких как шрифты, цвета, отступы и выравнивание, через JSON-описание.

5. Поддержка кроссплатформенности. Библиотека должна быть совместимой с несколькими платформами: Android, iOS, Desktop. Это предполагает единую кодовую базу, использующую Kotlin Multiplatform.

6. Тестируемость. Библиотека должна содержать инструменты для тестирования JSON-описаний, рендеринга UI и ключевых функций.

Нефункциональные требования

В ходе проектирования библиотеки были сформированы следующие нефункциональные требования, описанные ниже.

1. Совместимость. Библиотека должна поддерживать Android (API 26+), WASM JS, macOS, Windows, Linux (на базе JVM) и опционально iOS (iOS 11+).

2. Масштабируемость. Архитектура библиотеки должна быть модульной и легко масштабируемой, чтобы разработчики могли добавлять новые компоненты или изменять существующие без значительных изменений в базе кода.

3. Удобство интеграции. Установка библиотеки должна быть простой. Должна быть возможность подключения посредством платформенных build-инструментов, таких как Gradle или Maven.

4. Надежность. Библиотека должна обрабатывать ошибки на всех этапах работы. Для них должны быть предусмотрены fallback-механизмы, к примеру, вывод placeholder’ов или предупреждений.

2.2. Проектирование функционала системы

Диаграмма вариантов использования на рисунке 2 представляет ключевые взаимодействия приложения-хоста с проектируемой библиотекой BDUI.

На диаграмме выделены два основных варианта использования, отражающих базовый функционал системы: «Переопределить зависимости компонентов библиотеки» и «Открыть экран по указанному запросу». Эти варианты использования представляют собой два самых важных аспекта работы библиотеки – гибкость настройки и динамическое построение интерфейсов. Спецификация этих вариантов использования представлена в таблицах 1 – 4 приложения А.

Рисунок 2 – Диаграмма вариантов использования

Первый вариант использования, «Переопределить зависимости компонентов библиотеки», предоставляет возможность настройки библиотеки под конкретные нужды проекта. Он позволяет разработчикам заменять стандартные реализации компонентов, механизмы обработки данных или стилизации, обеспечивая интеграцию с существующей архитектурой приложения. Данный функционал особенно важен для проектов, где требуется адаптация библиотеки под специфические бизнес-логики или дизайн-системы.

Второй вариант использования, «Открыть экран по указанному запросу», описывает основной сценарий взаимодействия с библиотекой – динамическое построение пользовательского интерфейса на основе JSON-описания, полученного от сервера. Данный вариант использования является олицетворением концепции Backend-Driven UI, позволяя осуществить обновление интерфейса без необходимости модификации клиентского кода.

2.3. Проектирование архитектуры системы

Проектируемая библиотека BDUI построена по модульной архитектуре, основанной на принципах чистой архитектуры и разделения ответственности. Диаграмма компонентов системы представлена на рисунке 3.

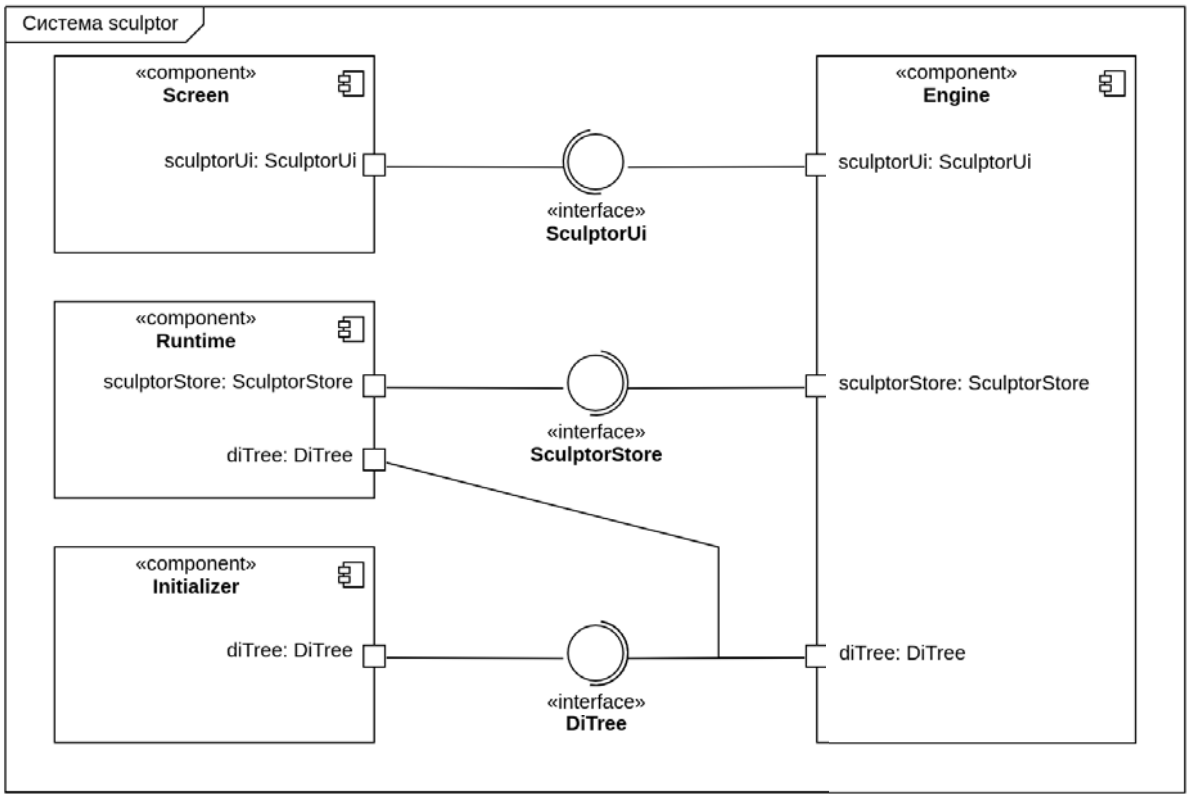

Рисунок 3 – Диаграмма компонентов системы

Основу системы составляют четыре ключевых компонента, взаимодействующих через четко определенные интерфейсы.

Для начальной настройки библиотеки используется компонент Initializer, построенный по паттерну «Строитель». Он предоставляет интерфейс, с помощью которого можно гибко конфигурировать библиотеку. Кроме того, Initializer гарантирует, что инициализация происходит атомарно, обеспечивая согласованное состояние всех подсистем до начала их использования.

Компонент Runtime представляет собой исполняющую среду библиотеки. Он служит для обработки данных и взаимодействия с платформо-специфичными слоями. Этот компонент, используя систему инициализации через DiTree, инкапсулирует свой функционал в ScultorStore, представляющим собой интерфейс для обращения к исполняющей среде библиотеки. Через SculptorStore возможно отслеживать текущее состояние работы библиотеки, а также делегировать исполнение различных событий и команд извне.

Ядром библиотеки является компонент Engine, который отвечает за организацию взаимодействия других компонентов библиотеки друг с другом. Этот модуль содержит в себе множество платформенно-специфичных интерфейсов и компонентов, которые позволяют интегрировать всю систему Sculptor в конкретную платформу хоста. Этот модуль предоставляет API для создания SculptorUi – прослойки, которую использует компонент Screen для общения с остальной системой Sculptor.

Взаимодействие с библиотекой происходит через компонент Screen. Этот модуль представляет конкретный способ интеграции системы Sculptor в приложение хоста. Для различных платформ созданы различные варианты Screen, которые поддерживают различные UI-системы платформ. К примеру, для Android этот компонент поставляется в рамках отдельной Activity или Fragment. Если же на Android используется подход с single-activity application, то становится возможным использовать реализацию Screen в виде @Composable функции, которая встраивается в дерево UI конкретного экрана и начинает отслеживать его жизненный цикл. Этот компонент предоставляет определенный и понятный API для работы с библиотекой и скрывает всю внутреннюю сложность реализации, предлагая разработчикам методы для открытия экранов Sculptor.

Проектирование компонента Initializer

Основная задача компонента Initializer, представленного на рисунке 4, заключается в подготовке окружения для работы библиотеки перед созданием экземпляров UI-компонентов. Компонент выполняет валидацию конфигурации, инициализирует граф зависимостей через систему DI и настраивает внутренние сервисы. Особенностью реализации является поддержка как глобальной конфигурации, так и экземплярной настройки для отдельных частей приложения.

Его архитектура построена вокруг принципов модульности и разделения ответственности, что позволяет легко настраивать библиотеку под конкретные требования проекта.

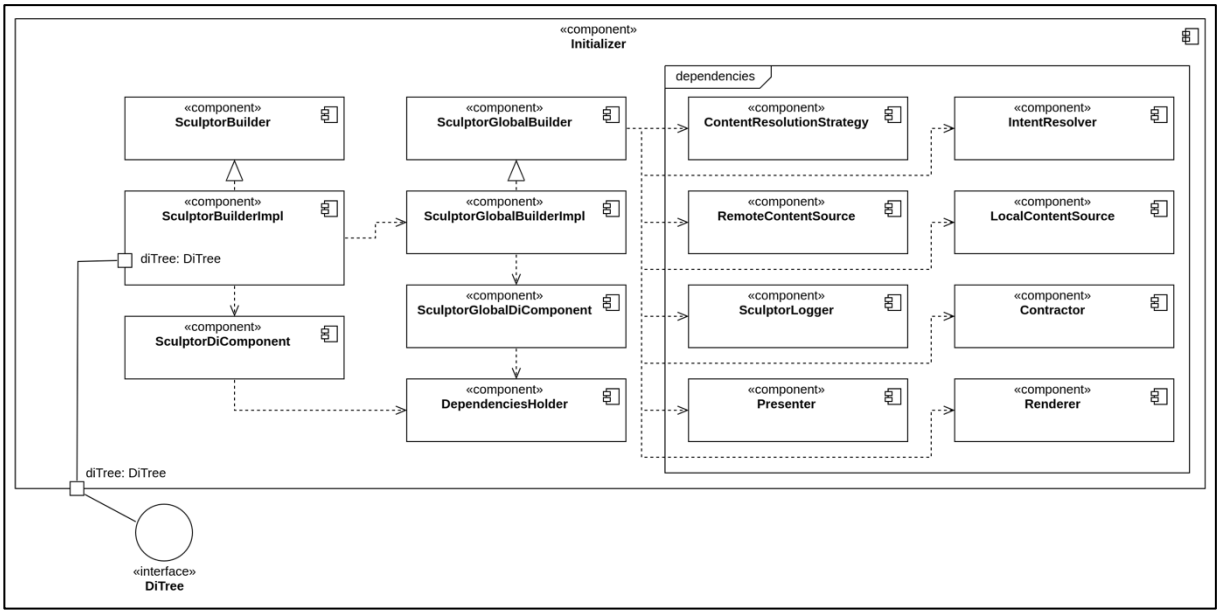

Рисунок 4 – Диаграмма компонента Initializer

Основу компонента составляют два строителя – SculptorBuilder для настройки экземпляров и SculptorGlobalBuilder для глобальной настройки системы зависимостей библиотеки. Каждый строитель имеет соответствующую реализацию в виде impl-классов, что обеспечивает соблюдение принципа инверсии зависимостей. Строители предоставляют dsl-интерфейс, позволяющий разработчику задавать параметры через отдельные ленивые вызовы методов, что делает код конфигурации читаемым и производительным. Кроме того, использование dsl-интерфейса позволяет сохранить обратную совместимость библиотеки при ее обновлении хостами.

Система управления зависимостями реализована через компоненты SculptorDiComponent и SculptorGlobalDiComponent. Эти компоненты тесно интегрируются с DiTree, предоставляя механизм внедрения зависимостей. Они отвечают за регистрацию сервисов, проверку графа зависимостей на циклические ссылки, а также поддерживают ленивую инициализацию ресурсоемких объектов.

Особое внимание уделено возможности переопределения компонентов библиотеки, что дает разработчикам свободу в кастомизации поведения системы при ее интеграции в приложение.

Проектирование компонента Runtime

Компонент Runtime на рисунке 5 представляет собой исполняющую среду библиотеки, отвечающую за обработку данных и взаимодействие с платформо-специфичными слоями.

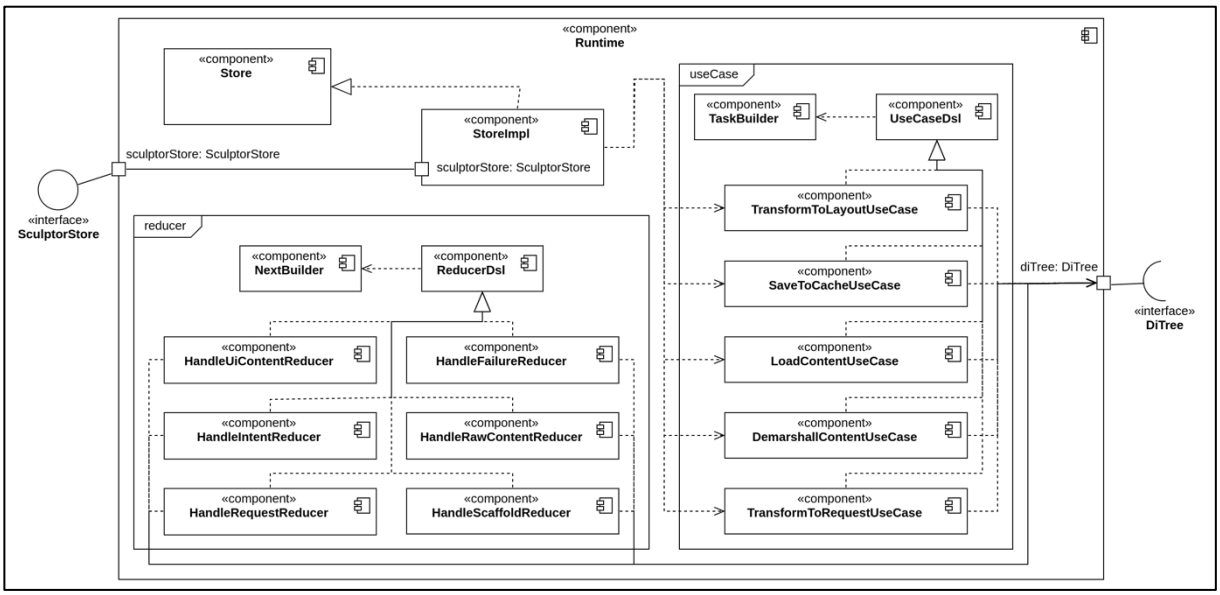

Рисунок 5 – Диаграмма компонента Runtime

Его архитектура построена вокруг двух ключевых подсистем: управления состоянием и обработки бизнес-логики, что обеспечивает эффективное выполнение операций в кроссплатформенном контексте.

В основе модуля Runtime находятся компоненты Store и StoreImpl, реализующие паттерн хранилища для управления состоянием приложения. Компонент Store выполняет роль централизованного источника истины: он хранит актуальное состояние пользовательского интерфейса и обрабатывает входящие события с помощью системы редьюсеров. Конкретная реализация этой логики находится в StoreImpl, который оптимизирован для работы с большими объёмами данных и поддерживает транзакционные обновления состояния.

Система редьюсеров включает в себя ряд специализированных компонентов, каждый из которых обрабатывает определённый тип событий. HandleIntentReducer преобразует диплинки в запросы к бизнес-логике. HandleRequestReducer обрабатывает запросы на загрузку экранов по URL.

HandleFailureReducer централизованно обрабатывает ошибки. HandleScaffoldReducer и HandleRawContentReducer выполняют предварительную обработку контента перед отображением, обеспечивая целостность и согласованность данных.

Бизнес-логика представлена в виде набора UseCase-компонентов, каждый из которых выполняет отдельную задачу. TransformToLayoutUseCase – преобразует абстрактное описание интерфейса в конкретную layout-структуру. SaveToCacheUseCase – отвечает за кэширование данных. LoadContentUseCase – управляет загрузкой контента. DemarshallContentUseCase – занимается десериализацией данных. TransformToRequestUseCase – формирует сетевые запросы в соответствии с бизнес-правилами.

Архитектура компонента Runtime отличается чётким разделением ответственности между редьюсерами и UseCase-компонентами, что делает код более чистым и удобным для тестирования. Поддержка транзакционных обновлений состояния обеспечивает целостность данных даже при параллельной обработке событий.

Компонент Runtime тесно связан с остальными частями системы. Он получает все необходимые зависимости от Initializer, собирает компонент Store, обогащает его нужными редьюсерами и UseCase-логикой, после чего передаёт в движок библиотеки. При этом сохраняется строгая изоляция: другие части системы не имеют доступа к внутреннему устройству Runtime и взаимодействуют с ним только через заранее определённый публичный API.

Этот компонент не доступен в полной мере внешнему хосту, который использует библиотеку. Вместо этого существует возможность добавлять кастомные редьюсеры и UseCase-компоненты, которые будут обрабатывать кастомные Event и Command, заранее известные хосту.

Проектирование компонента Engine

Компонент Engine, представленный на рисунке 6, выступает центральным компонентом библиотеки, обеспечивающим связь между бизнес-

логикой и UI-слоем. Его ключевая задача – создание и управление экземплярами SculptorUi, которые инкапсулируют всю логику работы с конкретным экраном. Движок действует как посредник, трансформирующий данные и события между различными частями системы.

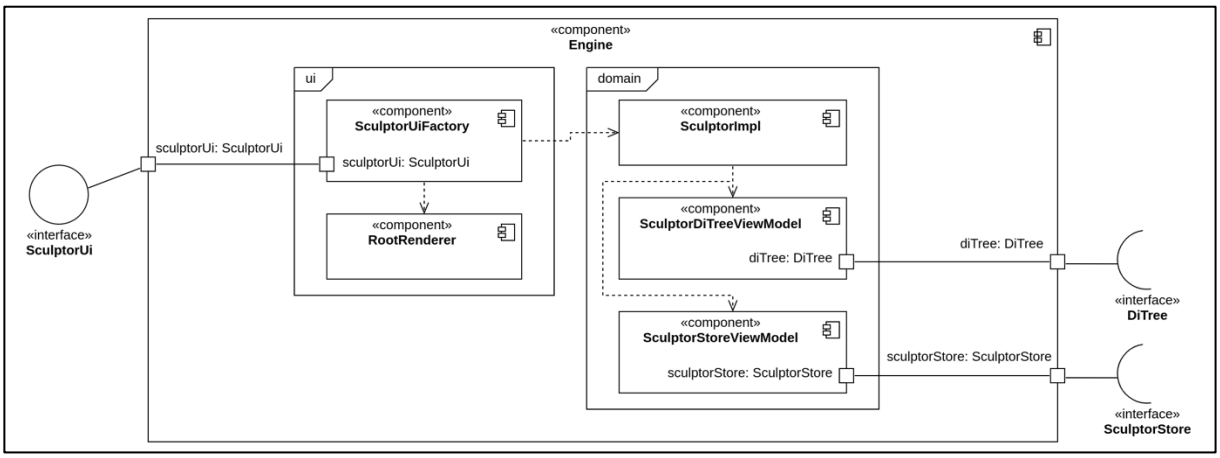

Рисунок 6 – Диаграмма компонента Engine

Основной механизм работы Engine строится вокруг SculptorUiFactory, которая отвечает за создание экземпляров SculptorUi. SculptorUi становится контейнером, объединяющим состояние, обработчики событий и способ визуального представления экрана Sculptor, что обеспечивает изолированную работу для каждого места интеграции системы в приложение.

Для интеграции с жизненным циклом платформы Engine использует SculptorStoreViewModel. Эта прослойка подписывается на изменения состояния экрана и синхронизирует их с SculptorUi. Сам механизм ViewModel выступает мостом между нативными механизмами жизненного цикла и абстрактной логикой библиотеки, гарантируя корректное освобождение ресурсов и обработку фоновых задач.

Аналогично, DiTree интегрируется с жизненным циклом платформы через DiTreeViewModel. При завершении жизненного цикла экрана дерево зависимостей очищается в целях недопущения утечек памяти. Сам компонент DiTree в этой архитектуре выполняет роль системы управления зависимостями. Движок запрашивает у него необходимые сервисы и компоненты, которые затем внедряются в SculptorUi. Это позволяет гибко настраивать

поведение UI-компонентов без изменения их внутренней логики. Например, подменить реализацию навигации или механизм кэширования для конкретного экрана.

Проектирование компонента Screen

Компонент Screen на рисунке 7 представляет собой конечную точку взаимодействия с библиотекой, предоставляя разработчикам простой API для работы с BDUI-экранами.

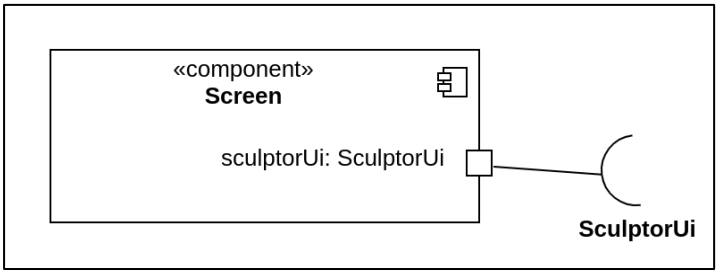

Рисунок 7 – Диаграмма компонента Screen

Основу компонента составляет сущность Screen, выступающая фасадом для доступа к функциональности библиотеки. Она инкапсулирует экземпляр SculptorUi, который содержит всю логику работы с конкретным экраном – от рендеринга компонентов до обработки пользовательских событий.

Главная задача Screen – обеспечить удобный интерфейс для базовых операций с экраном. Компонент предоставляет метод для открытия экрана по диплинку, скрывая за этим простым действиями сложную цепочку преобразований и проверок. Все вызовы к Screen транслируются в соответствующие операции SculptorUi, но разработчик работает только с высокоуровневым API, который предоставляет библиотека.

2.4. Проектирование структуры библиотеки

Библиотека организована по принципу модульности с четким разделением на публичные и внутренние компоненты. Такая архитектура позволяет предоставлять разработчикам стабильный API, одновременно сохраняя гибкость для внутренних модификаций.

Проектирование внешней структуры библиотеки

Публичная часть библиотеки, представленная на рисунке 8, организована как набор четко структурированных модулей, каждый из которых решает конкретную задачу интеграции с клиентскими приложениями.

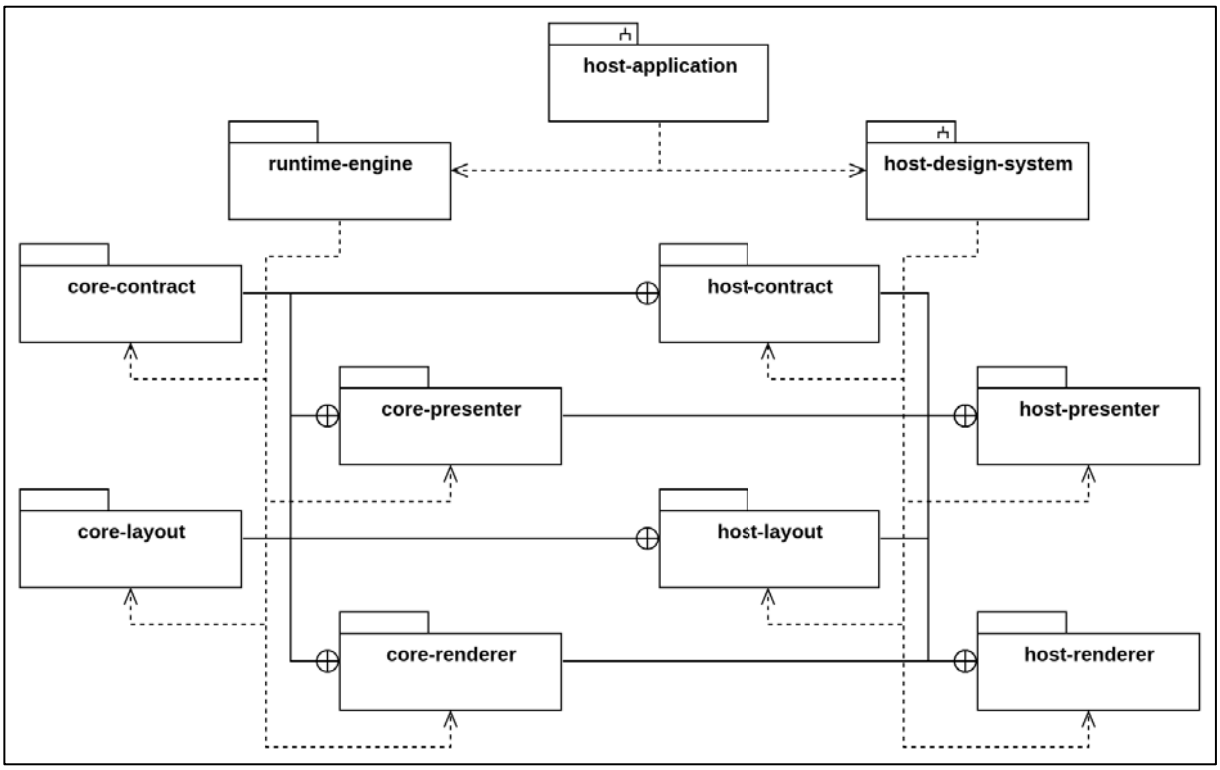

Рисунок 8 – Диаграмма публичных модулей

Модули с префиксом host на схеме представляют собой модули-расширения функционала библиотеки в приложении-интеграции. Они не входят в поставляемый библиотекой функционал.

Core-contract представляет собой фундаментальный модуль, определяющий базовые соглашения между сервером и клиентом. В нем содержатся интерфейсы для описания UI-компонентов, их свойств и возможных состояний. Разработчики могут расширять базовые контракты, добавляя кастомные типы компонентов и свойств через систему наследования.

Core-layout, аналогично core-contract, представляет собой описания UI-элементов в виде компонентов состояний. Разработчики могут добавлять свои UI-элементы посредством описания новых компонентов состояний.

Core-presenter отвечает за преобразование сериализуемых контрактов в готовые для отображения модели. Модуль предоставляет точки расширения для кастомизации логики трансформации состояний конкретных компонентов.

Core-renderer предоставляет абстракцию для фактического отображения компонентов на экране. Разработчики могут добавлять свои рендереры, если это необходимо. К примеру, в случае использования собственной дизайн-системы в приложении интеграции.

Runtime-engine выступает связующим звеном между всеми модулями, управляя жизненным циклом UI-компонентов и координируя их взаимодействие. Этот модуль предоставляет точки входа для инициализации библиотеки, создания экранов и обработки пользовательских событий. В его обязанности входит кэширование данных, управление состоянием и обработка ошибок.

Host-design-system это условный модуль, который может создать хост с целью объединения всего перопределенного функционала системы в одном месте для последующей интеграции. В этом модуле может находиться дизайн-система, включая кастомные контракты, описания UI-элементов, рендереров и презентеров. Эта дизайн-система может быть использована как для построения структуры экрана с помощью контрактов на серверной части, так и для рендеринга этих компонентов на стороне клиента.

Host-application это некоторый модуль-приложение хоста, которое интегрирует в себя как runtime-engine, так и host-design-system. Этот модуль представляет собой конечное место для интеграции системы Sculptor в приложение с возможностью использования кастомной дизайн-системы.

Проектирование внутренней структуры библиотеки

Внутренняя организация библиотеки на рисунке 9 представляет собой сложную систему взаимосвязанных модулей, спроектированных для максимальной эффективности и гибкости. Эта структура скрыта от конечных пользователей и служит основой для работы публичного API.

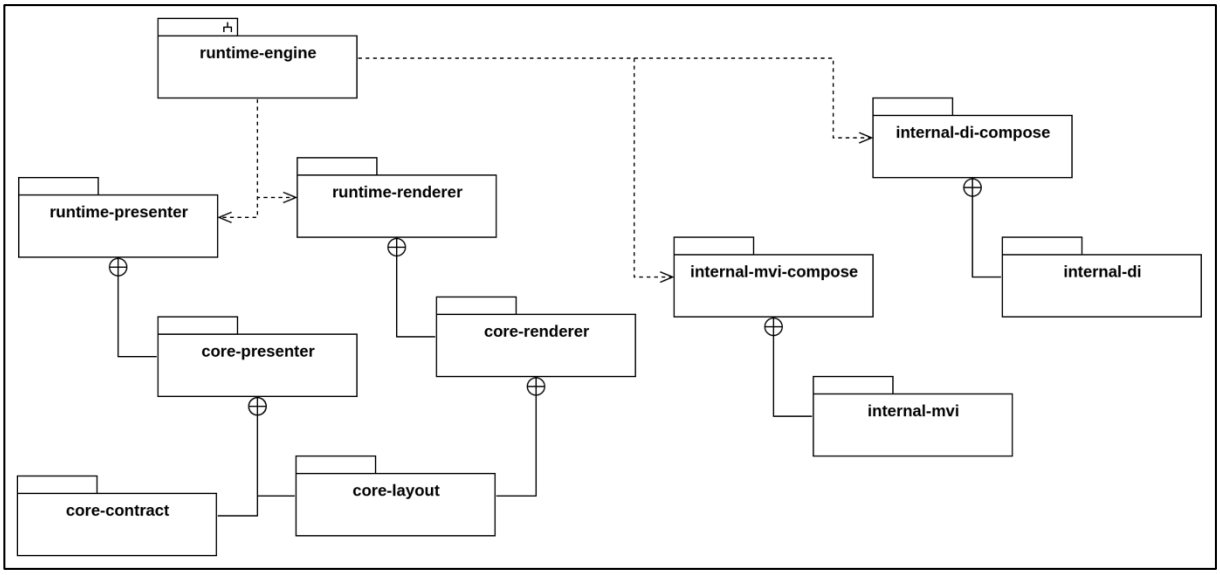

Рисунок 9 – Диаграмма внутренних модулей

Runtime-engine выступает центральным координатором внутренней архитектуры. Этот модуль интегрирует все компоненты библиотеки в единую рабочую систему, управляя жизненным циклом процессов и распределяя ресурсы.

Runtime-presenter содержит реализации абстракций, объявленных в core-presenter модуле. А runtime-renderer, аналогично, содержит реализации абстракций из core-renderer.

Internal-di представляет собой систему внедрения зависимостей, специально разработанную для нужд библиотеки. Модуль включает собственный DI-контейнер с поддержкой графа зависимостей, ленивой инициализации и кастомных scope. Internal-di-compose расширяет базовую DI-систему специализированными компонентами для работы с Compose Multiplatform.

Internal-mvi представляет собой ключевой компонент внутренней архитектуры, реализующий паттерн MVI для управления состоянием приложения. Этот модуль тесно интегрируется со всей экосистемой библиотеки, обеспечивая предсказуемое и декларативное управление потоком данных. Internal-mvi-compose расширяет базовую MVI-систему специализированными компонентами для работы с Compose Multiplatform.

Выводы по второй главе

В данном разделе была проведена работа по проектированию архитектуры кроссплатформенной библиотеки BDUI, основанной на концепции Backend-Driven UI. В результате были определены ключевые компоненты системы, изучены их взаимодействие и внутренняя структура, что позволило сформировать целостное представление о будущей реализации библиотеки.

Особое внимание было уделено разделению архитектуры на внешние и внутренние модули. Внешние публичные модули – такие как core-contract, core-presenter, core-layout и core-renderer – предоставляют стабильный API, предназначенный для работы разработчиков с динамическими интерфейсами. Внутренние модули – такие как runtime-presenter, runtime-renderer, internal-di, internal-mvi – реализуют низкоуровневую, оптимизированную логику, обеспечивающую высокую производительность и гибкость системы.

Архитектура строится на принципах модульности, инкапсуляции и однонаправленного потока данных. В центре системы выделен компонент Engine, выполняющий роль координатора: он управляет жизненным циклом экранов и связывает бизнес-логику с пользовательским интерфейсом.

Таким образом, на этапе проектирования удалось заложить архитектурную основу, определить структуру библиотеки и ключевые механизмы её работы.

3. РЕАЛИЗАЦИЯ

3.1. Программные средства реализации

Для разработки кроссплатформенной библиотеки BDUI используется язык программирования Kotlin [17], а именно его мультиплатформенная версия – Kotlin Multiplatform. Эта технология позволяет создать единую кодовую базу, которая работает на всех целевых платформах – Android [18], WasmJS и Desktop.

В качестве основной среды разработки была выбрана Android Studio [19], так как она предоставляет встроенную поддержку Kotlin Multiplatform и удобные инструменты для запуска и тестирования проектов на Android, WasmJS и Desktop.

Для создания пользовательского интерфейса на всех платформах используется фреймворк Compose Multiplatform. Он предлагает декларативный подход к построению UI-компонентов и выступает единым стандартом рендеринга для Android, WasmJS и Desktop. Это значительно упрощает процесс разработки и тестирования, а также снижает необходимость в написании платформозависимого кода.

Обработка JSON-описаний интерфейсов реализована с помощью библиотеки kotlinx.serialization, которая благодаря встроенной поддержке в Kotlin обеспечивает удобную и быструю работу с данными. Для сетевых операций – например, загрузки JSON-файлов или отправки пользовательских событий на сервер – используется мультиплатформенная библиотека Ktor [20].

3.2. Настройка CI/CD

Для автоматизации сборки, тестирования и развертывания проекта был настроен конвейер непрерывной интеграции и доставки (CI/CD) с использованием GitHub Actions. В репозитории проекта был создан конфигурационный файл .github/workflows/ci.yml, в котором описаны все этапы конвейера.

Конвейер CI/CD включает в себя следующие этапы, описанные ниже.

1. Сборка проекта. На данном этапе проверяется корректность компиляции исходного код проекта для каждой из платформ.

2. Запуск модульных тестов. На данном этапе проверяется корректность работы отдельных компонентов библиотеки.

3. Интеграционные и UI тесты. На данном этапе проверяется взаимодействие между модулями и компонентами проекта, в том числе UI-компонентами.

4. Сборка артефактов. На данном этапе создаются исполняемые файлы, библиотеки и другие компоненты для разных платформ с помощью Gradle.

5. Развертывание и публикация. На данном этапе готовые артефакты, полученные на предыдущем этапе, публикуются в хранилище GitHub Packages для дальнейшего использования в других проектах.

CI/CD-процесс автоматически запускается при каждом push в ветку master или при создании pull request, что обеспечивает своевременное внедрение изменений и быстрое выявление ошибок.

Также в проекте используется семантическое версионирование. Каждому релизу присваивается версия в формате MAJOR.MINOR.PATCH, где MAJOR – значительные изменения, MINOR – добавление новых функций, а PATCH – исправление ошибок.

К примеру, релиз библиотеки с версией 0.0.1, находящийся в репозитории GitHub Packages представлен на рисунке 10.

Рисунок 10 – Релиз библиотеки в репозитории GitHub Packages

Таким образом, настройка CI/CD обеспечила эффективное управление исходным кодом, автоматизацию ключевых процессов сборки и тестирования, а также надежный процесс релиза новых версий библиотеки.

3.3. Настройка системы сборки проекта

В проекте для сборки и управления зависимостями используется система Gradle. Главный файл конфигурации settings.gradle.kts расположен в корне проекта и содержит общие настройки: используемую версию Gradle, список репозиториев и базовые зависимости, применяемые ко всем модулям.

Каждый модуль имеет собственный файл build.gradle.kts, в котором указываются его уникальные зависимости и параметры сборки.

Чтобы стандартизировать и упростить настройку модулей, в проекте используются convention-плагины Gradle, находящиеся в директории build-logic. Эти плагины инкапсулируют общие конфигурации и позволяют избежать дублирования кода, обеспечивая единообразие настроек во всех частях проекта.

В проекте реализованы следующие convention-плагины.

1. Плагин convention.base.android.gradle.kts – базовая конфигурация для Android-модулей. Подключает плагин com.android.library и настраивает параметры вроде compileSdk, defaultConfig, compileOptions и другие.

2. Плагин convention.base.kmp.gradle.kts – конфигурация для Kotlin Multiplatform-модулей. Подключает плагин org.jetbrains.kotlin.multiplatform и настраивает kotlinOptions, sourceSets и прочие общие параметры.

3. Плагин convention.base.wasmJs.gradle.kts – конфигурация для модулей, использующих WebAssembly и JavaScript. Также применяет multiplatform-плагин и задаёт специфические настройки для платформы WasmJS.

4. Плагин convention.project.compose.library.gradle.kts – используется для модулей, в которых применяется Jetpack Compose. Подключает плагины

com.android.library и org.jetbrains.compose, а также настраивает необходимые зависимости.

5. Плагин convention.project.kotlin.library.gradle.kts – для модулей с библиотеками на чистом Kotlin без дополнительных компиляторов, к примеру, composeCompiler. Устанавливает kotlinOptions, jvmToolchain и другие параметры компиляции.

6. Плагин convention.project.showroom.application.gradle.kts – применяется к демонстрационному приложению Showroom. Этот плагин подключает плагины com.android.application и org.jetbrains.compose, а также задаёт параметры вроде applicationId, versionCode, versionName и др.

Использование convention плагинов помогает поддерживать порядок в проекте, ускоряет конфигурацию новых модулей, упрощает сопровождение и поддержку системы контроля зависимостей проекта и делает процесс сборки более предсказуемым и управляемым за счет указания всей конфигурации проекта в одном месте.

Таким образом, использование Gradle convention-плагинов в проекте обеспечивает эффективную сборку, централизованное управление зависимостями и консистентность настроек между всеми модулями, что значительно облегчает разработку, тестирование и развёртывание библиотеки.

3.4. Реализация системы

Библиотека BDUI разработана на языке Kotlin с использованием технологии Kotlin Multiplatform, что обеспечивает её работу на разных платформах. Для создания пользовательского интерфейса используется фреймворк Compose Multiplatform.

Каждый модуль библиотеки имеет собственную структуру и организацию кода. В основе архитектуры лежат принципы инверсии управления и внедрения зависимостей, что делает систему гибкой и легко расширяемой. Это позволяет без труда заменять отдельные компоненты и адаптировать библиотеку под специфические требования проекта.

Реализация механизма dependency injection

В библиотеке BDUI внедрение зависимостей реализовано через компонентную систему. Основная цель этой системы – централизованное и управляемое подключение зависимостей в проекте посредством их декларативного объявления и последующего разрешения.

Ключевым элементом системы является интерфейс Dependencies, который предоставляет методы для регистрации зависимостей двух видов: singleton и factory. Первый обеспечивает единичный экземпляр объекта для всей системы, в то время как второй создаёт новый экземпляр при каждом запросе. Методы singleton и factory принимают тип ключа, тип зависимости и функцию-фабрику для создания экземпляра.

DiComponentImpl реализует интерфейс DiComponent и содержит хранилище зависимостей в виде набора Declaration. Каждое объявление описывает тип зависимости, её ключ и функцию создания. Важной особенностью является поддержка ленивой инициализации – объект создаётся только при первом обращении через методы get, getOrNull или getAll.

Организацию зависимостей в логические группы упрощает интерфейс Module и его реализация DiModuleImpl. Модули позволяют сгруппировать объявления зависимостей, а затем добавить их в DiComponent одним вызовом метода addModule.

Также предусмотрена возможность клонирования компонентов – метод clone в DiComponent создаёт копию текущего компонента со всеми зарегистрированными зависимостями. Это позволяет изолировать контексты для различных частей приложения, сохраняя при этом единый граф зависимостей.

Ещё один важный элемент – DiTree, представляющий иерархическую структуру компонентов. Он позволяет выстраивать дерево, в котором каждый узел может иметь собственный DiComponent и наследовать зависимости от родительского. Такая организация позволяет создавать многоуровневую архитектуру с изолированными контекстами выполнения.

Механизм dependency injection наиболее полно используется в рамках варианта использования «Переопределить зависимости компонентов библиотеки. Диаграмма последовательности для этого варианта использования на рисунке 11 детализирует процесс переопределения зависимостей в библиотеке, раскрывая взаимодействие между ключевыми компонентами системы.

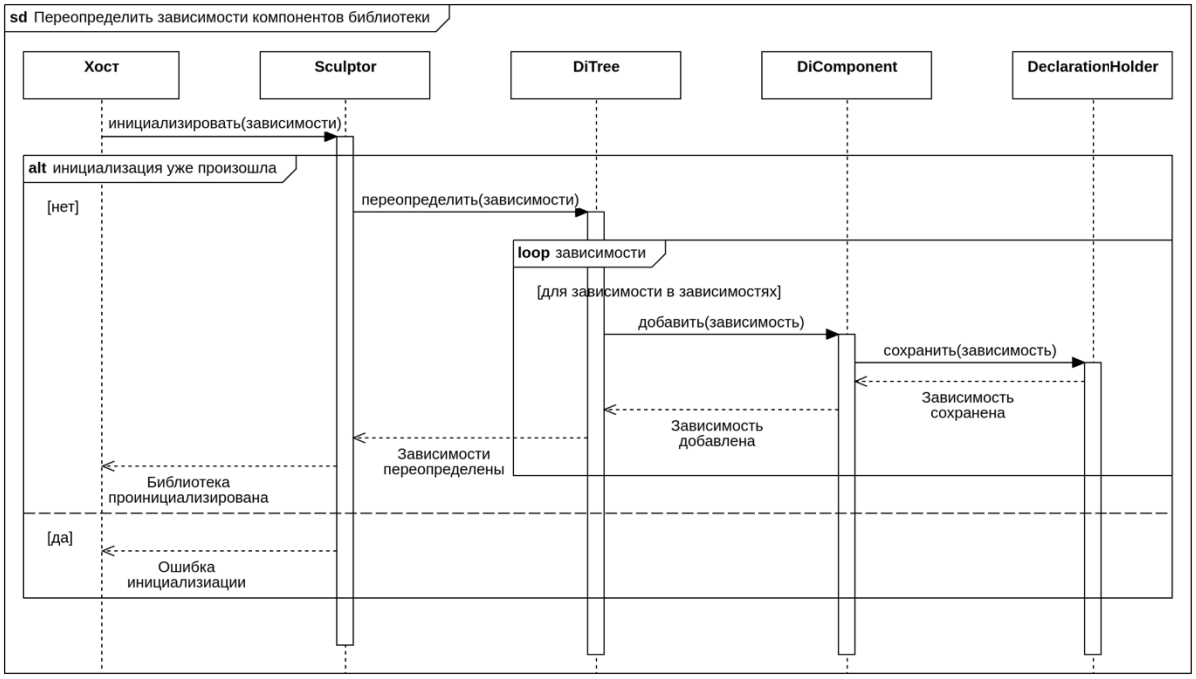

Рисунок 11 – Диаграмма последовательности

Основными участниками процесса являются Sculptor (ядро библиотеки), DiTree (контейнер зависимостей), DiComponent (базовый элемент для внедрения) и DeclarationHolder (хранилище объявлений).

Процесс начинается с инициализации зависимостей, где Sculptor отправляет запрос к DiTree. Если инициализация уже выполнена, система обрабатывает это как альтернативный поток, предотвращая дублирование операций. Далее DiTree последовательно обрабатывает каждую зависимость в цикле, взаимодействуя с DiComponent для валидации и регистрации.

При успешной обработке DiTree сохраняет зависимости в DeclarationHolder, что завершает основной поток с состоянием

«Зависимости переопределены». В случае ошибки, например, некорректная конфигурация, система возвращает статус «Ошибка инициализации».

Таким образом, механизм внедрения зависимостей в системе предлагает гибкую, модульную и расширяемую архитектуру, позволяющую разработчику точно управлять жизненным циклом объектов, использовать ленивую инициализацию, клонирование и структурирование компонентов.

Реализация механизма mvi

Библиотека BDUI реализует механизм управления состоянием на основе архитектурного паттерна MVI. Ключевые компоненты этого подхода – UI, Store, Reducer и UseCase – тесно взаимодействуют друг с другом для обработки состояний и событий. Диаграмма потока данных на рисунке 12 иллюстрирует, как данные и события перемещаются между этими элементами.

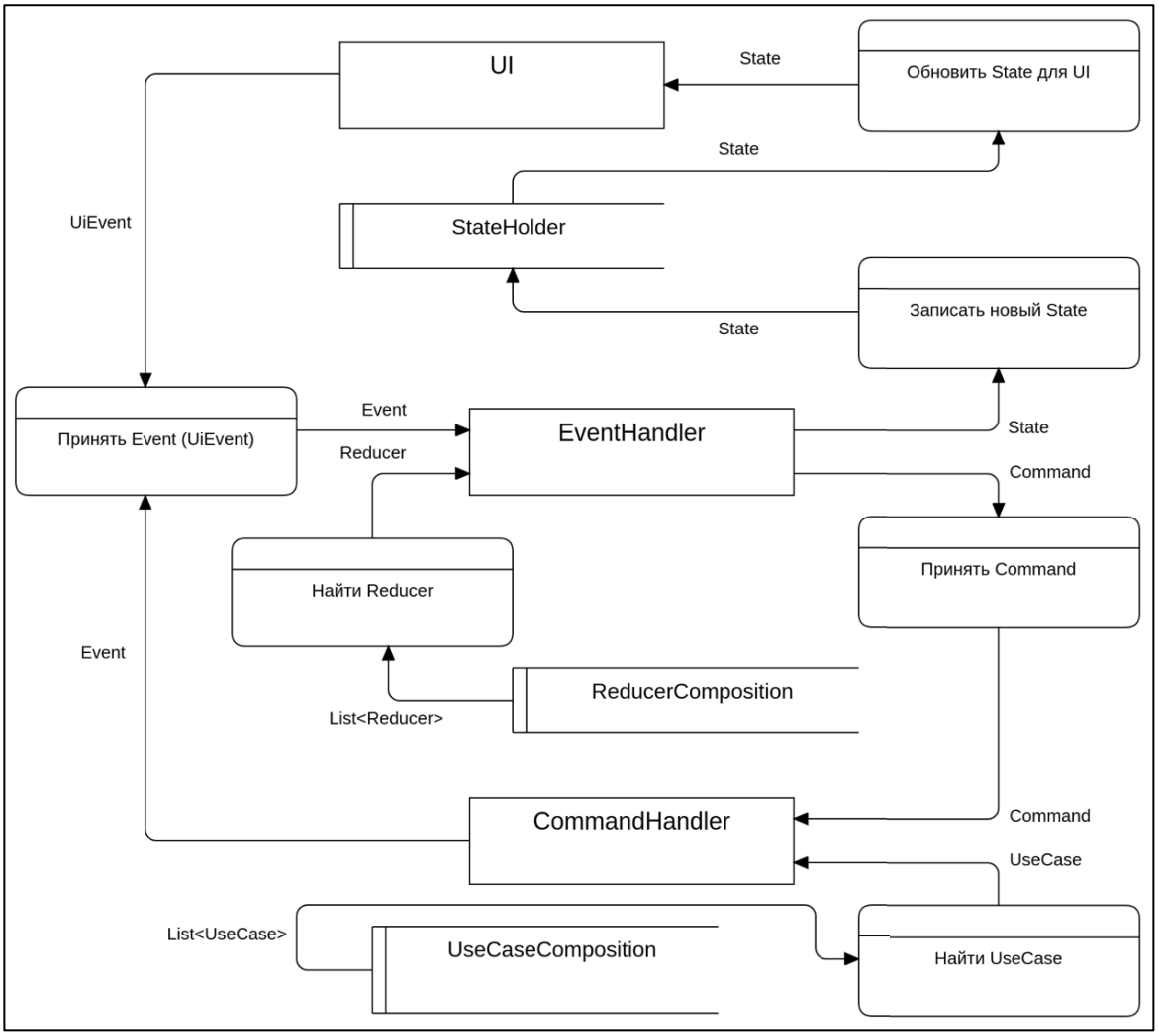

Рисунок 12 – Диаграмма потока данных MVI

Store – это центральное хранилище состояния приложения. Он реализован в виде интерфейса, включающего компонент StateHolder для отслеживания изменений и метод dispatch для обработки входящих событий. Основная реализация – StoreImpl, который управляет состоянием и обрабатывает события с помощью списка редьюсеров и UseCase-компонентов. Также Store поддерживает логирование с помощью StoreLogger, фиксируя все изменения состояния и отправленные команды.

Reducer отвечает за преобразование текущего состояния и события в новое состояние и список команд. Интерфейс Reducer определяет метод reduce, возвращающий объект Next, содержащий новое состояние и набор команд к выполнению. В проекте реализованы различные редьюсеры, включая HandleFailureReducer, HandleIntentReducer и HandleRequestReducer, каждый из которых обрабатывает определённый тип событий.

UseCase инкапсулирует бизнес-логику и реагирует на команды. Интерфейс UseCase определяет метод handle, принимающий команду и возвращающий список событий. Основная реализация работает с асинхронными операциями и потоками данных, обеспечивая изоляцию бизнес-логики от остальной части приложения.

Task представляет собой контейнер, в котором описываются события, возникающие в процессе выполнения команды или изменения состояния. С помощью интерфейса TaskDsl такие задачи можно описывать декларативно, используя методы dispatch и events.

Объект Next содержит новое состояние и список команд, сформированные редьюсерами или UseCase. Он возвращается обратно в Store и инициирует дальнейшие действия. Таким образом, Next играет роль связующего звена между Reducer, UseCase и Store, обеспечивая целостность потока обработки событий.

Цикл работы приложения с использованием этой архитектуры начинается с пользовательского интерфейса, отправляющего UiEvent в EventHandler. Затем событие передаётся в Reducer, который формирует

новое состояние и команду. Команда передаётся в CommandHandler, который находит подходящий UseCase, выполняет бизнес-логику и генерирует новые события. Эти события снова поступают в EventHandler, обеспечивая цикличную обработку. Компонент StateHolder хранит и обновляет текущее состояние, взаимодействуя с UI для отображения изменений.

Таким образом, механизм MVI обеспечивает централизованное управление состоянием через Store, декларативное описание бизнес-логики через UseCase и обработку событий через Reducer. Это обеспечивает единообразное управление потоком данных и предсказуемость поведения компонентов системы.

Реализация механизма рендеринга экрана

Механизм рендеринга экрана реализует функционал варианта использования «Открыть экран по указанному запросу». Диаграмма активности для этого варианта использования на рисунке 13 визуализирует процесс динамического построения интерфейса на основе серверного запроса, отражая ключевые этапы преобразования JSON-описания в готовый UI.

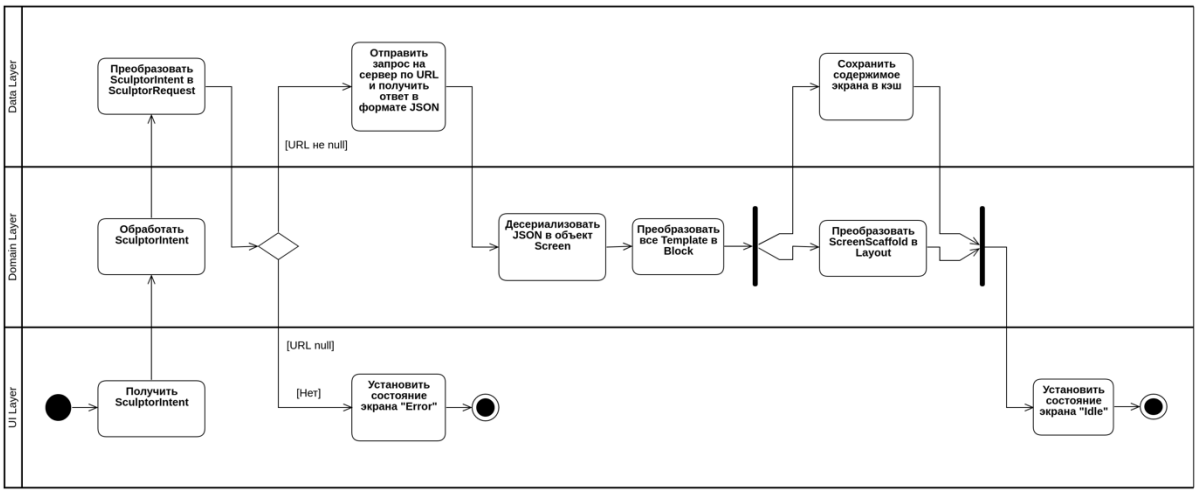

Рисунок 13 – Диаграмма активности

Всё начинается с получения SculptorIntent – диплинка, содержащего необходимые данные для запуска процесса. Этот Intent передаётся в SculptorUi, который запускает цепочку действий, ведущую к построению экрана. Первый шаг – взаимодействие с IntentResolver, задачей которого

является извлечение URL из полученного диплинка. Если извлечение URL оканчивается неудачей, то система переходит в состояние ошибки.

Далее происходит загрузка данных. За стратегию загрузки отвечает ContentResolutionStrategy: она определяет, откуда именно брать контент – из локального кэша или с удалённого сервера. Для локальных данных по умолчанию используется InMemoryLocalContentSource, а для сетевых запросов – RemoteContentSource. Если нужный URL не найден, генерируется исключение ContentNotFoundException, после чего система переключается на отображение экрана ошибки.

После загрузки данных в формате JSON начинается парсинг JSON-структуры в объектную модель Screen, где выполняется валидация обязательных полей и типов данных. Далее система последовательно обрабатывает шаблоны Template, преобразуя их в унифицированные блоки Block, которые содержат логику отображения и взаимодействия. Далее происходит обработка параметров блоков, которые трансформируются в объекты Layout с расчетом размеров, позиционирования и иерархии элементов. Параллельно происходит кэширование провалидированных к этому моменту данных для ускорения последующих загрузок экрана по этому запросу.

После формирования состояния компонентов экрана запускается этап валидации, который выполняет StateValidator. Он проверяет структуру и корректность полученной информации. Если данные проходят проверку, они преобразуются в SculptorState и сохраняются в Store – это и есть текущее состояние экрана. Если же при валидации возникли ошибки, например, структура данных не совпала с ожидаемой, библиотека активирует HandleFailureReducer. Этот компонент формирует команду на отображение экрана ошибки.

На этапе рендеринга SculptorUi определяет, какой рендерер использовать – это делает RendererProvider, который подбирает подходящий Renderer в зависимости от типа отрисовываемого компонента. После этого управление передаётся в RendererScope, где вызывается метод draw – именно он

отвечает за передачу данных в компоненты Compose и их отображение на экране.

Таким образом, механизм рендеринга экрана представляет собой многоэтапный процесс, включающий загрузку данных, валидацию, кэширование и обработку ошибок.

На рисунке 14 приведён пример верстки экрана в формате JSON. Структура экрана состоит из четырех текстовых блоков, расположенных друг за другом вертикально. Рисунки 15 – 18 демонстрируют как библиотека рендерит этот экран на различных платформах.

На текущий момент поддерживаются следующие платформы: Android, Desktop (Linux, Windows, macOS), WasmJS (в браузерах с поддержкой WebAssembly).

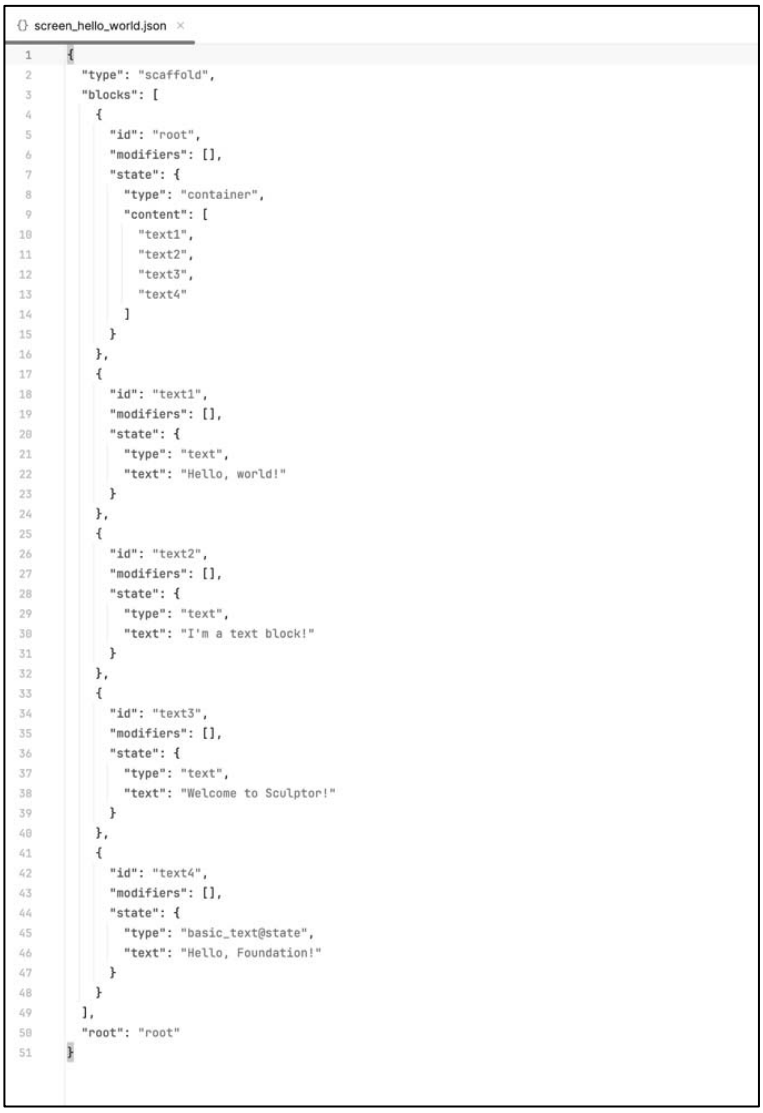

Рисунок 14 – Пример верстки экрана в формате JSON

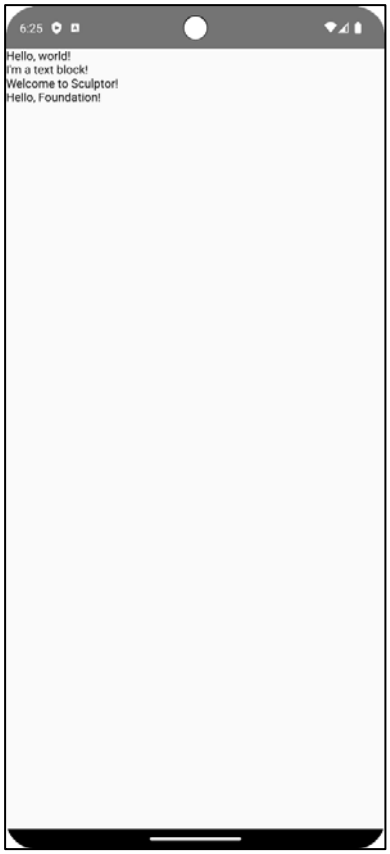

Рисунок 15 – Скриншот работы библиотеки на платформе Android (API 34)

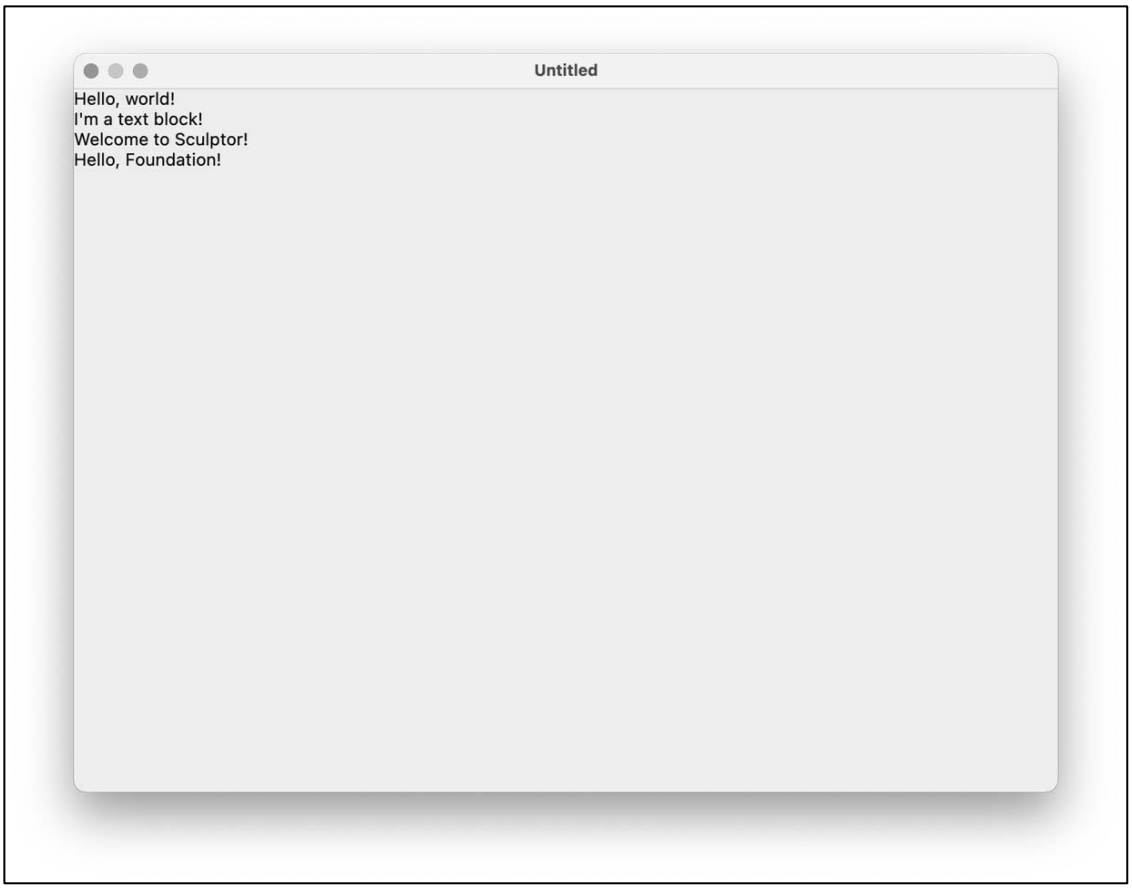

Рисунок 16 – Скриншот работы библиотеки на платформе Desktop (macOS)

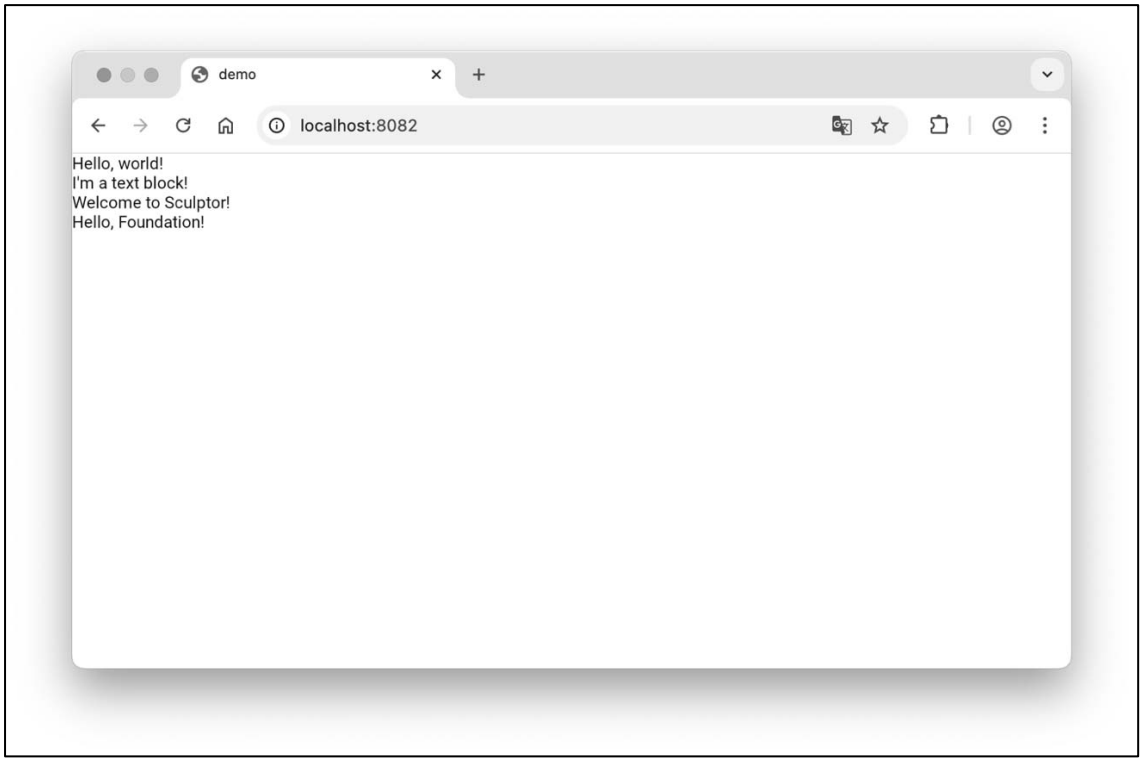

Рисунок 17 – Скриншот работы библиотеки на платформе WasmJS (Chrome)

Выводы по третьей главе

В этой главе была проделана работа по созданию архитектуры системы и реализации функциональности кроссплатформенной библиотеки BDUI с использованием Kotlin Multiplatform. Основное внимание сосредоточилось на трёх ключевых направлениях: внедрение зависимостей, реализация паттерна MVI и механизм рендеринга интерфейса.

Для внедрения зависимостей была использована связка DiTree и DiComponent, что позволило выстроить иерархию компонентов и обеспечить гибкое управление зависимостями. Такой подход помогает создавать изолированные контексты выполнения и сокращать число прямых связей между модулями – что особенно важно в условиях кроссплатформенной разработки.

Архитектурный шаблон MVI позволил добиться предсказуемого управления состояниями. Это реализовано через Store, обработку событий с

помощью Reducer и UseCase, что обеспечивает чёткое разделение ответственности и упрощает отладку.

Механизм рендеринга экрана реализован с использованием SculptorUi и RendererScope, которые обеспечивают прозрачный и управляемый поток данных: от получения SculptorIntent до отрисовки компонентов. Благодаря мемоизации состояний обновляются только те элементы интерфейса, которые действительно изменились. Это повышает производительность и снижает нагрузку на систему.

В итоге была реализована архитектура, сочетающая модульность, предсказуемость работы и высокий уровень изоляции компонентов. Такой подход позволил создать библиотеку BDUI, которая эффективно управляет состоянием, гибко обрабатывает события и адаптирована к различным платформам, оставаясь при этом в рамках единой кодовой базы.

4. ТЕСТИРОВАНИЕ

В разделе «Тестирование» рассматриваются процессы проверки корректности и надежности компонентов библиотеки BDUI, включая механизм внедрения зависимостей, управление состояниями через паттерн MVI и рендеринг экранов. Основная цель тестирования – обеспечить предсказуемое поведение системы, минимизировать количество ошибок и выявить возможные дефекты на ранних этапах разработки.

4.1. Модульное тестирование системы

Модульное тестирование в проекте BDUI организовано для проверки корректности работы отдельных компонентов системы, таких как система DI и система MVI. Основной задачей модульных тестов является изоляция логики компонентов и проверка их функциональности в независимых условиях.

Для написания тестов используются фреймворки Kotlin Test и JUnit, которые обеспечивают необходимые инструменты для создания и выполнения тестовых сценариев.

Значительное внимание уделяется тестированию механизма dependency. В модулях internal-di и internal-di-compose реализованы тесты на регистрацию singleton-зависимостей, обработку модулей и их переопределение. Написанные тесты представлены в таблице 5 приложения Б.

В контексте архитектуры MVI ключевым элементом для тестирования является Store. В модулях internal-mvi и internal-mvi-compose реализованы сценарии проверки начального состояния Store, его обновление через команды и события, а также корректное применение редьюсеров. В тестах моделируются различные сценарии передачи событий и команд, а также проверяется соответствие выходного состояния ожидаемому результату. Написанные тесты представлены в таблице 6 приложения Б.

4.2. Интеграционное тестирование системы

Интеграционное тестирование играет важную роль в обеспечении стабильной и предсказуемой работы системы при взаимодействии её ключевых компонентов. В отличие от модульного тестирования, которое проверяет отдельные части системы в изоляции, интеграционное тестирование сосредоточено на том, как эти компоненты работают вместе в рамках общего потока данных. Такой подход позволяет своевременно выявлять ошибки и несоответствия на стыках между компонентами и убедиться, что передача данных между ними происходит корректно.

Интеграционное тестирование реализовано с помощью фреймворков Kotlin Test и Mokkery. Основное внимание уделяется взаимодействию компонентов DI, MVI и Engine. Тесты охватывают ключевые сценарии передачи и обработки данных, а также проверяют правильность обновления состояния в ответ на изменения. Это позволяет убедиться, что все части системы синхронно работают друг с другом, без сбоев и неожиданных эффектов.

Например, в SculptorGlobalBuilderTest проверяется корректность инициализации глобального DiTree. В SculptorBuilderTest проверяется корректность инициализации локального DiTree. Кроме того, тесты охватывают сценарии загрузки данных, обработки ошибок и обновления состояния SculptorStore. В SculptorStoreIntegrationTest фокус смещается на проверку процесса формирования UiState и его передачи в SculptorUi. Написанные тесты представлены в таблице 7 приложения Б.

Значимость интеграционного тестирования заключается в том, что оно позволяет убедиться в целостности архитектуры и корректности взаимодействия компонентов при выполнении сложных сценариев. В случае изменений в одном из компонентов интеграционные тесты помогают быстро обнаружить возможные нарушения в работе системы и гарантировать стабильность всей библиотеки в целом.

4.3. UI-тестирование системы

UI-тестирование в библиотеке BDUI необходимо для проверки корректности отображения компонентов интерфейса и взаимодействия с ними. Основная цель UI-тестов – убедиться в том, что визуальные компоненты, их состояния и реакции на события соответствуют ожиданиям, обеспечивая единообразие пользовательского опыта на всех поддерживаемых платформах.

В проекте BDUI UI-тестирование организовано с использованием фреймворка Compose Multiplatform Test. Этот инструмент позволяет запускать UI-тесты на всех целевых платформах, включая Android, iOS, Web и Desktop.

UI-тесты фокусируются на следующих аспектах, описанных ниже.

1. Корректное отображение компонентов.

Проверяется структура экрана, правильность расположения компонентов и соответствие их визуального состояния заданным макетам.

2. Корректная обработка событий.

Моделируются пользовательские действия, такие как нажатия на кнопки, прокрутка списка и т.д., и проверяется корректность обработки этих событий.

3. Правильные состояния компонентов.

Тестируются различные сценарии изменения состояния UI, включая загрузку данных, отображение ошибок и успешное обновление контента.

Для реализации UI-тестов в проекте используется Showroom – отдельный модуль, который представляет собой демонстрационное приложение с набором экранов, реализующих типичные сценарии использования BDUI. Внутри Showroom компоненты библиотеки представлены в различных состояниях, что позволяет протестировать их взаимодействие в контролируемой среде. Написанные тесты представлены в таблице 8 приложения Б.

Выводы по четвертой главе

В этой главе была проведена проверка компонентов библиотеки BDUI на корректность работы, предсказуемость поведения и соответствие заявленной функциональности. Основное внимание было уделено трём уровням тестирования: модульному, интеграционному и UI-тестированию.

Модульное тестирование библиотеки охватило такие ключевые аспекты, как корректность регистрации и извлечения зависимостей, передача событий и команд в Store и формирование состояния на основе данных. Это позволило убедиться в надёжности базовой логики ключевых компонентов библиотеки.

Интеграционное тестирование библиотеки направлено на проверку взаимодействия между различными компонентами системы. В рамках интеграционных тестов проверялись такие сценарии, как инициализация библиотеки, загрузка данных из сети, кэширование запросов и преобразование состояния. Эти тесты позволили выявить возможные сбои и несогласованности в работе компонентов.

UI-тестирование сосредоточилось на проверке правильности отображения пользовательского интерфейса и его поведения при изменении состояния. Для этого использовалось специальное демонстрационное приложение – showroom, в котором были собраны примерные экраны, имитирующие реальные сценарии использования библиотеки. Во время тестов воспроизводились типичные пользовательские действия – такие как нажатия, прокрутка и обновление данных. Это позволило оценить стабильность, предсказуемость и корректность поведения интерфейса при взаимодействии с пользователем.

Таким образом, в ходе тестирования была подтверждена стабильность компонентов BDUI, их согласованность при взаимодействии и предсказуемость визуального поведения интерфейса на всех поддерживаемых платформах.

ЗАКЛЮЧЕНИЕ

В процессе выполнения выпускной квалификационной работы была разработана кроссплатформенная библиотека BDUI, реализующая концепцию Backend-Driven UI с использованием Kotlin Multiplatform. Основной целью работы являлось создание инструментария для динамического формирования интерфейсов на основе JSON-описаний и обеспечения их рендеринга на множестве платформ, включая Android, iOS, Web и Desktop.

В рамках проведенного исследования были проанализированы существующие решения в области BDUI, а также их архитектурные особенности. Анализ позволил определить ключевые требования к разрабатываемой библиотеке и выбрать оптимальные архитектурные решения, что легло в основу проектирования системы.

В результате проектирования системы была сформирована многоуровневая архитектура библиотеки, включающая компоненты для управления зависимостями, обработки событий и команд, рендеринга интерфейсов и интеграции с Compose Multiplatform.

На этапе реализации был создан механизм dependency injection на основе DiTree и DiComponent, обеспечивающий гибкость конфигурирования зависимостей и их иерархическое структурирование. Это позволило организовать изолированные контексты выполнения и минимизировать связанность компонентов. Для управления состояниями и событиями был реализован паттерн MVI с использованием Store, Reducer и UseCase, что обеспечило предсказуемое поведение системы и централизованное управление состояниями. Особое внимание было уделено разработке механизма рендеринга экранов. Процесс рендеринга включает в себя этапы извлечения данных, валидации JSON-структур, преобразования данных в UiState и динамической отрисовки интерфейсов с поддержкой кэширования и оптимизированного обновления только измененных компонентов.

Завершающим этапом работы стало тестирование библиотеки. В ходе тестирования были проверены основные компоненты системы на

корректность работы, предсказуемость поведения и согласованность взаимодействия. Модульные тесты охватили ключевые элементы MVI и DI-архитектуры, интеграционные тесты проверили передачу данных между компонентами, а UI-тестирование обеспечило визуальную целостность компонентов на различных платформах.

Таким образом, в результате выполнения работы была создана библиотека BDUI, способная динамически формировать пользовательские интерфейсы на основе JSON-описаний и обеспечивать их кроссплатформенное отображение. Реализованные механизмы DI и MVI гарантируют гибкость архитектуры, модульность компонентов и предсказуемость их поведения.

Внедрение подхода Backend-Driven UI позволяет централизовать управление интерфейсами и ускорить процессы обновления приложений без необходимости публикации новых версий. Полученные результаты могут быть использованы для дальнейшего расширения функциональности библиотеки и интеграции её в реальные проекты.

Исходный код проекта находится в репозитории по следующей ссылке: https://github.com/AlexCawl/sculptor.

ЛИТЕРАТУРА

1. Habr. BDUI: удовольствие или боль. [Электронный ресурс] URL: https://habr.com/ru/companies/cian/articles/840664 (дата обращения 27.01.2025 г.).

2. Документация фреймворка Kotlin Multiplatform [Электронный ресурс]. URL: https://kotlinlang.org/docs/multiplatform.html (дата обращения 27.01.2025 г.).

3. Документация фреймворка Jetpack Compose. [Электронный ресурс]. URL: https://developer.android.com/develop/ui/compose/ (дата обращения 27.01.2025 г.).

4. Документация системы сборки Gradle. [Электронный ресурс]. URL: https://docs.gradle.org/ (дата обращения 27.01.2025 г.).

5. Habr. Что такое VCS (система контроля версий). [Электронный ресурс]. URL: https://habr.com/ru/articles/552872 (дата обращения 27.01.2025 г.).

6. Habr. Что такое CI (непрерывное развертывание) [Электронный ресурс]. URL: https://habr.com/ru/articles/508216 (дата обращения 27.01.2025 г.).

7. Habr. Что такое CD (непрерывная доставка) [Электронный ресурс]. URL: https://habr.com/ru/companies/1cloud/articles/449364 (дата обращения 27.01.2025 г.).

8. AppTractor. Что такое DI (внедрение зависимостей) [Электронный ресурс]. URL: https://apptractor.ru/info/articles/dependency-injection.html (дата обращения 27.01.2025 г.).

9. AppTractor. Что такое архитектура MVI. [Электронный ресурс]. URL: https://apptractor.ru/info/articles/mvi.html (дата обращения 27.01.2025 г.).

10. DEV.TO. Что такое DSL. [Электронный ресурс]. URL: https://dev.to/surajvatsya/understanding-domain-specific-languages-dsls-2eee (дата обращения 27.01.2025 г.).

11. Документация DivKit. [Электронный ресурс]. URL: https://div-kit.tech/ru (дата обращения 27.01.2025 г.).

12. Репозиторий Redwood. [Электронный ресурс]. URL: https://github.com/cashapp/redwood (дата обращения 27.01.2025 г.).

13. Фацио М. Kotlin and Android Development featuring Jetpack: Build Better, Safer Android Apps. Пер. с англ. СПб.: Наука, 2026. 416 с. ISBN 978-5-02-039876-3. ББК 32.973.26-018.1.

14. Kotlin Multiplatform by Tutorials (Second Edition): Build Native Apps for iOS and Android with Kotlin Multiplatform. Пер. с англ. СПб.: Питер, 2029. 550 с. ISBN 978-5-4461-2910-2. ББК 32.973.26-018.1.

15. Репозиторий Lona. [Электронный ресурс]. URL: https://github.com/Lona/Lona (дата обращения 27.01.2025 г.).

16. Simplifying Application Development with Kotlin Multiplatform Mobile. Пер. с англ. М.: ДМК Пресс, 2030. 430 с. ISBN 978-5-97060-830-6. ББК 32.973.26-018.1.

17. Руководство по языку Kotlin. [Электронный ресурс] URL: https://kotlinlang.org/docs/home.html (дата обращения: 27.01.2025 г.).

18. Форрестер А., и др. How to Build Android Apps with Kotlin. Пер. с англ. М.: Альпина Паблишер, 2025. 360 с. ISBN 978-5-9614-1176-0. ББК 32.973.26-018.1.

19. Android Studio. [Электронный ресурс] URL: https://developer.android.com/studio/ (дата обращения: 27.01.2025 г.).

20. Kotlin Multiplatform Mobile with Ktor. Пер. с англ. СПб.: БХВ-Петербург, 2031. 390 с. ISBN 978-5-9775-7892-2. ББК 32.973.26-018.1.

ПРИЛОЖЕНИЯ

Приложение А. Спецификация вариантов использования

Таблица 1 – Спецификация ВИ «Переопределить зависимости компонентов библиотеки»

Прецедент: Переопределить зависимости компонентов библиотеки
ID: 1
Аннотация: Процесс настройки и замены зависимостей библиотеки для адаптации под конкретный проект
Главные акторы: Хост
Второстепенные акторы: Нет
Предусловия: Библиотека подключена к проекту
Основной поток: 1. Разработчик инициализирует процесс переопределения зависимостей. 2. Система проверяет состояние инициализации. 3. Для каждой зависимости выполняется валидация и регистрация. 4. Зависимости сохраняются в DeclarationHolder. 5. Система подтверждает успешное переопределение.
Постусловия: Все компоненты используют обновленную конфигурацию
Альтернативные потоки: Ошибка инициализации

Таблица 2 – Альтернативный поток 1.1

Альтернативный поток: Переопределить зависимости компонентов библиотеки: Ошибка инициализации
ID: 1.1
Аннотация: Ошибка во время инициализации
Главные акторы: Хост
Второстепенные акторы: Нет
Предусловия: Альтернативный поток начинается на шаге 2 основного потока 1.
Альтернативный поток: 1. Происходит ошибка при инициализации зависимостей библиотеки. 2. Система сигнализирует об ошибке.
Постусловия: Система инициализации возвращает сообщение об ошибке с указанием причины

Таблица 3 – Спецификация ВИ «Открыть экран по указанному запросу»

Прецедент: Открыть экран по указанному запросу
ID: 2
Аннотация: Процесс динамического построения пользовательского интерфейса на основе JSON-описания с сервера
Главные акторы: Хост
Второстепенные акторы: Нет
Предусловия: Библиотека подключена к проекту. Зависимости проинициализированы
Основной поток: 1. Система отправляет запрос на указанный URL через RemoteContentSource. 2. Получает ответ в формате JSON. 3. Сохраняет данные в кэше. 4. Парсит JSON в объектную модель Screen. 5. Преобразует шаблоны в блоки (Template → Block). 6. Формирует Layout через TransformToLayoutUseCase. 7. Рендерит интерфейс через RootRenderer. 8. Отображает готовый экран пользователю.
Постусловия: Пользовательский интерфейс отображен согласно серверному описанию
Альтернативные потоки: Невалидный JSON

Таблица 4 – Альтернативный поток 2.1

Альтернативный поток: Открыть экран по указанному запросу: Невалидный JSON
ID: 2.1
Аннотация: Алгоритм действий при открытии экрана по указанному запросу, если полученный JSON невалиден
Главные акторы: Хост
Второстепенные акторы: Нет
Предусловия: Альтернативный поток начинается на шаге 4 основного потока 2
Альтернативный поток: 1. Происходит ошибка во время парсинга JSON 2. Система сигнализирует об ошибке
Постусловия: Отображен пользовательский fallback-интерфейс

Приложение Б. Тестирование библиотеки

Таблица 5 – Модульное тестирование системы DI

№	Название теста	Шаги	Ожидаемый результат	Тест пройден?
1	Регистрация singleton-зависимости	1. Создать DiComponent; 2. Зарегистрировать singleton-зависимость.	Объект успешно создается и извлекается	Да
2	Переопределение singleton-зависимости	1. Создать DiComponent; 2. Зарегистрировать singleton-зависимость; 3. Зарегистрировать новую singleton-зависимость того же типа.	Последний зарегистрированный объект успешно создается и извлекается	Да
3	Регистрация factory-зависимости	1. Создать DiComponent; 2. Зарегистрировать factory-зависимость.	Объект успешно создается и извлекается	Да
4	Проверка корректности создания объекта, объявленного как factory-зависимость	1. Создать DiComponent; 2. Зарегистрировать factory-зависимость; 3. Извлечь объект, сохранить его хешкод; 4. Извлечь объект заново.	Хешкоды объектов должны отличаться	Да
5	Переопределение factory-зависимости	1. Создать DiComponent; 2. Зарегистрировать singleton-зависимость; 3. Зарегистрировать новую singleton-зависимость того же типа.	Последний зарегистрированный объект успешно создается и извлекается	Да
6	Безопасное извлечение незарегистрированной зависимости	1. Создать DiComponent; 2. Извлечь незарегистрированную зависимость с помощью метода getOrNull.	Возвращается null	Да
7	Извлечение незарегистрированной зависимости	1. Создать DiComponent; 2. Извлечь незарегистрированную зависимость с помощью метода get.	Система падает с ошибкой	Да
8	Подключение зависимостей через модуль	1. Создать DiComponent; 2. Подключить модуль с зависимостями.	Объекты успешно создается и извлекается	Да
9	Клонирование дерева зависимостей	1. Создать DiComponent; 2. Зарегистрировать singleton и factory зависимости; 3. Извлечь объекты и сохранить хешкод; 4. Клонировать компонент.	Объекты успешно создается и извлекается. Для объекта singleton хешкод тот же самый	Да

Таблица 6 – Модульное тестирование системы MVI

№	Название теста	Шаги	Ожидаемый результат	Тест пройден?
1	Создание пустого Store	1. Создать Store и указать начальное состояние.	У Store начальное состояние	Да
2	Создание Store с указанием начальных команд	1. Создать Store и указать начальное состояние и начальные команды.	У Store начальное состояние, поскольку event loop не запущен	Да
3	Запуск пустого Store	1. Создать Store и указать начальное состояние, тестовый reducer и тестовый usecase; 2. Активировать event loop в Store.	При подписке на Store, его состояние неизменно	Да
4	Запуск Store с указанием начальных команд	1. Создать Store и указать начальное состояние, начальные команды, тестовый reducer и тестовый usecase; 2. Активировать event loop в Store.	У Store в результате исполнения команд изменилось состояние	Да
5	Проверка Reducer	1. Создать Store и указать начальное состояние и тестовый reducer; 2. Активировать event loop в Store. 3. Отправить в Store тестовый event.	У Store в результате обработки event изменилось состояние.	Да
6	Проверка Reducer + UseCase	1. Создать Store и указать начальное состояние, начальные команды, тестовый reducer и тестовый usecase; 2. Активировать event loop в Store. 3. Отправить в Store тестовый event, который вызывает command.	У Store в результате обработки event и command изменилось состояние.	Да
7	Проверка отмены подписки на состояние Store	1. Создать Store и указать начальное состояние, начальные команды, тестовый reducer и тестовый usecase; 2. Активировать event loop в Store; 3. Подписаться на жизненный цикл тестового компонента; 4. «Убить» тестовый компонент.	Подписка на состояние Store отменяется, event loop прекращает свою работу.	Да

Таблица 7 – Интеграционное тестирование библиотеки

№	Название теста	Шаги	Ожидаемый результат	Тест пройден?
1	Загрузка контента из сети	1. Проинициализировать Sculptor; 2. Установить ContentResolutionPolicy как OnlyRemote; 3. Начать загрузку экрана по диплинку.	JSON-схема экрана успешно загружена и десериализована	Да
2	Загрузка контента из кэшей и асинхронная подгрузка из сети	1. Проинициализировать Sculptor; 2. Установить ContentResolutionPolicy как WithCache; 3. Начать загрузку экрана по диплинку.	JSON-схема экрана успешно загружена и десериализована	Да
3	Загрузка контента из кэшей если сеть недоступна	1. Проинициализировать Sculptor; 2. Установить ContentResolutionPolicy как Default; 3. Замокать отсутствие сети; 4. Начать загрузку экрана по диплинку.	JSON-схема экрана успешно загружена и десериализована	Да
4	Стейт ошибки если экран невозможно загрузить	1. Проинициализировать Sculptor; 2. Замокать отсутствие сети и пустой кэш; 3. Начать загрузку экрана по диплинку.	SculptorUi покажет SculptorState.Error	Да
5	Успешная загрузка экрана	1. Проинициализировать Sculptor; 2. Начать загрузку экрана по диплинку.	SculptorUi покажет SculptorState.Idle	Да
6	Стейт ошибки, если не найден Presenter для блока экрана	1. Проинициализировать Sculptor; 2. Замокать отсутствие Presenter для блока экрана; 3. Начать загрузку экрана по диплинку.	SculptorUi покажет SculptorState.Error	Да
7	Стейт ошибки, если не найден Renderer для блока экрана	1. Проинициализировать Sculptor; 2. Замокать отсутствие Renderer для блока экрана; 3. Начать загрузку экрана по диплинку.	SculptorUi покажет SculptorState.Error	Да

Таблица 8 – UI-тестирование библиотеки

№	Название теста	Шаги	Ожидаемый результат	Тест пройден?
1	Отображение Loading экрана во время загрузки контента	1. Проинициализировать Sculptor; 2. Создать инстанс SculptorUi; 3. Открыть экран приложения; 4. Отправить запрос на открытие Sculptor-экрана.	Будет показан Loading экран пока контент не загрузился	Да
2	Отображение Error экрана если произошла ошибка во время загрузки контента	1. Проинициализировать Sculptor; 2. Подменить метод загрузки экрана, чтобы он возвращал ошибку; 3. Создать инстанс SculptorUi; 4. Открыть экран приложения; 5. Отправить запрос на открытие Sculptor-экрана.	Будет показан Error экран	Да
3	Отображение структуры экрана при успешной загрузке	1. Проинициализировать Sculptor; 2. Создать инстанс SculptorUi; 3. Открыть экран приложения; 4. Отправить запрос на открытие Sculptor-экрана.	Будет показан экран со структурой такой же, как входящий JSON	Да