СОДЕРЖАНИЕ

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

API (Application Programming Interface) – набор способов и правил, по которым различные программы общаются между собой и обмениваются данными.

HTTP (HyperText Transfer Protocol) – широко распространённый протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов.

SSL/TLS (Secure Sockets Layer / Transport Layer Security) – криптографические протоколы, предназначенные для обеспечения безопасности передачи данных между сетевыми устройствами. TLS является более новой версией протокола SSL, и оба протокола часто используются для защиты данных, передаваемых через Интернет.

HTTPS (HyperText Transfer Protocol Secure) – расширенная версия протокола HTTP, которая использует шифрование на основе SSL/TLS для безопасного обмена данными в Интернете. HTTPS защищает целостность и конфиденциальность данных, передаваемых между пользователем и сайтом.

REST (Representational State Transfer) – стиль архитектуры программного обеспечения для распределенных систем, который, как правило, используется для построения веб-служб.

UI (User Interface) – пользовательский интерфейс: все кнопки, таблички, поля ввода текста и другие способы взаимодействия юзера с сайтом, приложением или иным IT-сервисом.

CSS (Cascading Style Sheets) – это язык стилей, используемый для описания внешнего вида и форматирования документов.

POM (Project Object Model) – концепция в Maven. Представляет собой XML файл, который содержит информацию о проекте и конфигурационные детали, используемые Maven для сборки проекта.

JSON (JavaScript Object Notation) – текстовый формат обмена данными.

JPA (Java Persistence API) – спецификация Java, описывающая систему управления сохранением java объектов в таблицы реляционных баз данных в удобном виде.

DTO (Data Transfer Object) – объект передачи данных, один из шаблонов проектирования, используется для передачи данных между подсистемами приложения.

JWT (JSON Web Token) – открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JSON. Как правило, используется для передачи данных для аутентификации в клиент-серверных приложениях.

ORM (Object-Relational Mapping) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования.

URL (Uniform Resource Locator) – адрес ресурса в сети Интернет.

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Стриминговые сервисы – это онлайн-платформы, обеспечивающие потоковую трансляцию различных данных в режиме реального времени.

Фронтенд – часть веб-приложения или веб-сайта, которая отвечает за взаимодействие с пользователем и отображение данных.

Бэкенд – часть веб-приложения или веб-сайта, которая отвечает за обработку данных и бизнес-логику.

Библиотека (в программировании) – сборник подпрограмм или объектов, используемых для разработки программного обеспечения (ПО).

Фреймворк – набор готовых инструментов, библиотек и стандартов, предназначенных для разработки программного обеспечения, который облегчает и ускоряет процесс создания приложений.

Паттерн (шаблон) – повторяемое решение для общей проблемы в проектировании программного обеспечения, которое описывает основные элементы решения и отношения между ними.

Парсинг – автоматизированный сбор и структурирование информации с сайтов при помощи программы или сервиса.

Реляционная база данных – тип базы данных, в которой данные организованы в таблицы, и эти таблицы связаны на основе данных, общих для каждой из них. Эта структура позволяет выполнять гибкие сложные запросы одновременно ко многим таблицам.

Нереляционная база данных (NoSQL) – термин, обозначающий технологии управления данными, отличных от SQL. К NoSQL относятся столбцовые, графовые, документно-ориентированные системы, а также системы по модели “ключ – значение”.

ВВЕДЕНИЕ

В современном мире музыка играет ключевую роль в жизни многих людей. Она сопровождает нас повсюду: дома, в пути, на работе и, конечно, на концертах, где фанаты собираются вместе, чтобы насладиться живым исполнением своих любимых артистов. Посещение концертов становится важным социальным событием, где фанаты имеют возможность вживую увидеть своих любимых исполнителей. Однако пользователи часто сталкиваются с трудностью отслеживания предстоящих музыкальных событий, что приводит к упущенной возможности участвовать в этих мероприятиях.

С развитием интернета и социальных сетей каждый исполнитель имеет возможность самостоятельно продвигать свои мероприятия, но не все артисты используют дорогостоящие рекламные кампании, что зачастую приводит к тому, что информация об их концертах теряется среди бесчисленного количества других новостей. Это проблема особенно актуальна для независимых исполнителей, у которых нет значительного бюджета на рекламу. В условиях информационного перенасыщения, когда пользователи подписаны на множество различных источников, легко пропустить анонсы интересующих событий. Таким образом, потенциальные поклонники рискуют не узнать о предстоящих концертах своих любимых артистов.

В связи с этим возникает необходимость в создании агрегатора концертных событий, который бы учитывал музыкальные предпочтения пользователя и предлагал бы ему персонализированный список предстоящих выступлений. Такое приложение могло бы интегрироваться с популярными музыкальными сервисами и афишами концертов, обеспечивая доступ к актуальной информации о мероприятиях в интересующем городе. Такой подход значительно упрощает процесс поиска и планирования культурного досуга, делая его более доступным и удобным для каждого пользователя.

1 ТЕОРЕТИЧЕСКИЙ ОБЗОР

1.1 Актуальность и практическая значимость разработки приложения для повышения вовлеченности пользователей в концертные события

Актуальность разработки приложения для повышения вовлеченности пользователей в концертные события обусловлена несколькими ключевыми факторами, которые отражают текущие тенденции в области музыкальной индустрии и технологий. Во-первых, современный темп жизни и перегруженность информацией затрудняют регулярное отслеживание музыкальных событий, особенно для тех, кто интересуется множеством артистов или жанров. Приложение, которое обеспечивает оперативное информирование о концертах, предлагая уведомления и обновления, будет востребовано среди таких пользователей. Во-вторых, увеличение числа музыкальных исполнителей и групп, которые предлагают широкий спектр жанров и стилей, делая рынок концертов более насыщенным и разнообразным. При этом пользователи ищут удобные и эффективные способы нахождения интересующих их мероприятий, что подчеркивает важность разработки инновационных решений в этой сфере. В-третьих, отсутствие аналогов такого приложения на российском рынке является значительным фактором, подчеркивающим актуальность разработки.

Практическая значимость предлагаемого приложения заключается в возможности улучшения пользовательского опыта при выборе музыкальных мероприятий. Персонализированный подход к предложению концертов, основанный на предпочтениях пользователя и его геолокации, позволяет существенно упростить поиск мероприятий, которые будут отвечать их интересам. Интеграция с популярными музыкальными сервисами обеспечивает автоматическое получение данных о новых предпочтениях пользователя, что способствует формированию точного и актуального списка предложений.

Предложенное приложение не только удовлетворяет потребности пользователей в удобном и эффективном доступе к концертным событиям, но и способствует развитию музыкальной индустрии, укрепляя связи между артистами и их аудиторией. Это, в свою очередь, может иметь положительное влияние на качество и разнообразие предлагаемых музыкальных мероприятий, а также на общую культурную атмосферу в регионе.

1.2 Обзор аналогов

При анализе рынка приложений для информирования о концертных мероприятиях можно выделить несколько ключевых аналогов, которые реализуют идеи схожие с предлагаемым проектом:

1. Bandsintown [1] – одно из наиболее известных приложений для отслеживания концертных событий. Приложение предлагает пользователю персонализированную информацию о концертах на основе его музыкальных исполнителей, которые определяются через интеграцию с музыкальными сервисами. Однако сервис ориентирован преимущественно на европейский и американский рынки, не поддерживая российские платформы афиш и музыкальных сервисов.
2. Songkick [2] – еще один популярный сервис, который позволяет пользователям отслеживать музыкальные события своих любимых исполнителей и покупать билеты на концерты. Подобно Bandsintown, Songkick интегрируется с музыкальными сервисами для предоставления персонализированных рекомендаций, и, так же в основном фокусируется на западном рынке и имеет сложный интерфейс, что снижает удобство использования.
3. Яндекс Афиша [3] – российский сервис для покупки билетов на различные мероприятия, включая концерты. Хотя он предлагает рекомендации по мероприятиям, основой для них служат исключительно данные из Яндекс Музыки и покупки билетов на самой платформе Яндекс Афиша. Это означает, что пользователи

получают персонализированные предложения только если используют Яндекс Музыку и, если на Яндекс Афише есть анонсы концертов их любимых исполнителей. Этот подход ограничивает возможности пользователей, предпочитающих другие музыкальные сервисы, и не позволяет полноценно использовать потенциал сервиса для охвата более широкой аудитории.

На основе сравнения существующих аналогов можно сделать вывод о наличии значительного потенциала для нового приложения, ориентированного на российский рынок. В то время как международные сервисы, такие как Bandsintown и Songkick, не предоставляют поддержку российских музыкальных платформ и фокусируются на западной аудитории, российские аналоги, например Яндекс Афиша, не обеспечивают достаточной персонализации и интеграции с широким спектром музыкальных сервисов. Это открывает возможности для разработки приложения, которое будет адаптировано под нужды и предпочтения российских пользователей, предлагая интеграцию с различными музыкальными сервисами и учитывая специфику местного рынка концертных событий.

1.3 Постановка задач

1.3.1 Проведение интеграции с афишами концертных событий

Одним из ключевых этапов разработки приложения для агрегации информации о концертных событиях является интеграция с афишами, на которых публикуется актуальная информация о предстоящих концертах. Взаимодействие с API позволит приложению собирать данные о предстоящих концертах, включая детали о времени, месте проведения и исполнителе.

Для реализации этой задачи планируется запросить доступ к API следующих платформ: Яндекс Афиша и KASSIR.RU [4]. Каждая из этих платформ обладает уникальным набором функций и покрывает различные аспекты рынка концертных мероприятий, что позволит приложению охватить широкий спектр предложений и предоставить пользователям полный каталог доступных событий.

1. Яндекс Афиша – Ведущий афишный сервис в России. Предлагает информацию не только о концертах, но и о других культурных событиях, таких как театры и кино. В контексте концертов сервис охватывает широкий спектр музыкальных жанров и предлагает данные о выступлениях как известных, так и менее популярных исполнителей. Концертные события, предоставляемые Яндекс Афишей отлично обогащены данными, что позволяет приложению обеспечить пользователей подробной информацией о мероприятиях, включая детали о месте проведения, времени начала и доступных билетах.
2. KASSIR.RU – Сервис известен своим акцентом на более традиционные и классические музыкальные события, включая концерты исполнителей “старой школы”. Он предлагает богатый выбор мероприятий, которые могут не всегда быть представлены на других платформах, что делает его ценным ресурсом для любителей классической и традиционной музыки. Интеграция с KASSIR.RU расширит возможности приложения по предоставлению разнообразных музыкальных предложений, обогащая пользовательский опыт и предоставляя доступ к широкому спектру культурных событий.

Таким образом, интеграция как минимум с этими платформами станет основой для создания надежного и эффективного сервиса, который будет способен предоставлять пользователям актуальные данные о концертах, учитывая их индивидуальные музыкальные предпочтения и географическое положение.

1.3.2 Интеграция с музыкальными сервисами

Вторым ключевым этапом разработки приложения является интеграция с популярными музыкальными стриминговыми сервисами. Эта интеграция необходима для того, чтобы пользователи моего приложения могли делиться информацией о своих музыкальных предпочтениях, что позволит приложению предлагать им персонализированные рекомендации по концертным событиям

мгновенно. Для достижения этой цели были выбраны два ведущих музыкальных стриминговых сервиса в России: Яндекс Музыка и VK Music.

1. Яндекс Музыка [5] – это один из крупнейших музыкальных стриминговых сервисов в России, предлагающий широкий спектр музыкальных жанров и исполнителей. Интеграция с Яндекс Музыкой позволит приложению привлечь к себе как можно больше аудитории.
2. VK Music [6] – часть социальной сети ВКонтакте, которая является одной из наиболее популярных платформ в России. Сервис предоставляет доступ к обширной библиотеке музыкальных треков и взаимодействует с большой аудиторией пользователей. Интеграция с VK Music дает приложению возможность достигать широкой социальной базы пользователей, что способствует ещё большему расширению возможностей для персонализации и социального взаимодействия в контексте музыкальных интересов.

Выбор этих сервисов обоснован их популярностью, масштабом доступной музыкальной базы и глубокой интеграцией с российской культурной средой. Интеграция с такими платформами позволяет обеспечить максимально релевантный и насыщенный контент для пользователей, способствуя созданию более глубокого и удовлетворительного опыта использования приложения.

1.3.3 Создание унифицированного интерфейса

Создание унифицированного интерфейса для работы с данными из разных источников является важным этапом в разработке. Это требование обусловлено необходимостью интегрировать и обрабатывать информацию из различных сервисов. Каждый из этих сервисов использует собственный формат данных и структуру API, что создаёт определённые вызовы при их агрегации и последующем использовании в едином приложении. Разработка унифицированного интерфейса позволит стандартизировать процесс получения данных в приложении и упростить добавление новых источников информации.

Для достижения этих целей можно использовать паттерны проектирования, которые помогут скрыть различия в API разных сервисов за единым интерфейсом. Это облегчит интеграцию новых сервисов в будущем и сделает систему более гибкой и адаптируемой к изменениям. Основная задача унифицированного интерфейса – обеспечить, чтобы приложение могло эффективно извлекать, трансформировать и использовать данные, оставаясь независимым от конкретных технических особенностей каждого источника.

1.3.4 Разработка пользовательского интерфейса

Разработка пользовательского интерфейса является критически важным аспектом в создании приложения. Чтобы максимально удовлетворить потребности широкого круга пользователей, важно, чтобы приложение было доступно на различных операционных системах, что позволит пользователям наслаждаться сервисом независимо от их устройств. Ключевым элементом дизайна является минимализм, который не только содействует эстетической привлекательности, но и способствует упрощению пользовательского интерфейса. Это делает приложение более интуитивно понятным и легким в использовании, что особенно важно в условиях, когда пользователь стремится быстро найти информацию о концертах и приобрести билеты.

Минималистичный дизайн также помогает сосредоточить внимание пользователя на самом важном – контенте. Использование чистых линий, ограниченной цветовой палитры и максимальной утилизации пространства экрана способствует созданию организованного и приятного визуального опыта. Такой подход позволяет избежать перегруженности интерфейса и делает взаимодействие с приложением более комфортным и менее стрессовым для пользователя. Центральное место в дизайне занимает удобство навигации, что обеспечивает быстрый доступ к необходимым функциям и информации, делая процесс поиска концертов максимально эффективным и приятным.

1.4 Выбор средств реализации

1.4.1 Ключевые технологии

Бэкенд с использованием Spring [7]: Выбор Spring Boot Framework как основы для серверной части приложения обусловлен его широкими возможностями и мощной экосистемой. В сравнении с другими фреймворками, такими как Django [8] или Express [9], Spring предлагает более обширную поддержку для создания сложных, высоконагруженных веб-приложений благодаря своей модульной структуре и интеграции с множеством библиотек и сервисов. Spring поддерживает полную интеграцию с базами данных и предоставляет мощные средства для работы с безопасностью, что очень важно для приложений, обрабатывающих пользовательские данные.

Фронтенд с использованием React Native [10]: Чтобы приложение поддерживало актуальные платформы и не приходилось вести две кодовые базы, имеет смысл обратиться к кроссплатформенным фреймворкам. Для фронтенда основными претендентами были React Native и Flutter [11]. Flutter – относительно новая технология от Google, способна обеспечивать высокую производительность за счет собственного рендеринга и компиляции в нативный код. Однако в контексте разработки данного приложения, где скорость не является решающим фактором, преимущества React Native оказываются более значимыми. React Native позволяет использовать JavaScript и React, что делает его идеальным выбором для разработчиков, уже знакомых с этими технологиями. Кроме того, технология React Native поддерживается крупным сообществом разработчиков и компаний, что обеспечивает регулярные обновления и поддержку большого количества плагинов и библиотек, расширяющих функциональные возможности приложений.

PostgreSQL [12] как база данных: При выборе системы управления базами данных было рассмотрено использование как реляционных (например, PostgreSQL), так и нереляционных (например, MongoDB [13]) систем управления базами данных. PostgreSQL была выбрана из-за её мощных возможностей работы со сложными запросами и надёжной транзакционной

модели, что важно для обеспечения целостности данных в приложении, где предусмотрена сложная логика обработки информации о мероприятиях и билетах. В отличие от нереляционных баз данных, таких как MongoDB, которые хорошо подходят для гибких данных и быстрых чтений, PostgreSQL предлагает более строгую схему данных и лучшую поддержку сложных операций соединения и агрегации, что необходимо для данного проекта.

Expo [14] для React Native: Expo представляет собой набор инструментов и сервисов, которые построены вокруг React Native и упрощают процесс разработки, тестирования и развертывания мобильных приложений. При сравнении с чистым React Native, Expo предлагает разработчикам готовый к использованию набор API, что значительно сокращает время на разработку и упрощает доступ к нативным функциям устройства, таким как камера, уведомления и другие, без необходимости вручную связывать нативные модули. Это делает Expo особенно ценным инструментом для ускорения разработки и снижения порога вхождения для новых разработчиков. Отказ от использования возможностей Expo и возвращение к чистому React Native увеличил бы сложность проекта и время разработки.

1.4.2 Библиотеки

NativeWind [15] для стилизирования компонентов: NativeWind обеспечивает разработчиков набором утилитарных классов, которые можно применять прямо в разметке, значительно ускоряя процесс разработки и предоставляя полный контроль над дизайном каждого элемента. В сравнении с компонентными CSS-фреймворками, такими как GluestackUI [16], которые предлагают набор предопределённых компонентов с ограниченными возможностями для кастомизации, NativeWind предоставляет гораздо большую гибкость. Такой подход позволяет точно настраивать внешний вид приложения без необходимости переопределять стили предопределенных компонентов, что часто ведет к повышению производительности кода. Тем более, в случае отсутствия требуемых компонентов, всё равно придется

создавать и стилизировать собственные, чтобы они соответствовали дизайну приложения.

Async Storage [17] для локального хранения: Async Storage представляет собой эффективное хранилище ключ-значение, которое предоставляет значительно более высокую производительность по сравнению с Local Storage, особенно в контексте мобильных приложений. В отличие от стандартного Local Storage, который синхронно взаимодействует с данными, что может замедлить выполнение скриптов и влиять на производительность интерфейса, Async Storage работает асинхронно. Это значит, что операции чтения и записи не блокируют основной поток выполнения приложения, позволяя пользовательскому интерфейсу оставаться отзывчивым даже при выполнении операций с данными. Async Storage, будучи частью экосистемы React Native, оптимизирован для работы в мобильных приложениях и предоставляет более удобный и эффективный способ локального хранения данных без ограничений, свойственных Local Storage, таких, как максимальный размер данных.

Эти выбранные библиотеки и фреймворки создают прочную основу для разработки приложения, максимально улучшая взаимодействие пользователя с интерфейсом и обеспечивая гибкость в реализации дизайнерских решений.

2 ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

В процессе разработки приложения я осознанно подошел к каждому аспекту проектирования, стремясь создать удобное, функциональное и надежное решение. Проектирование включало три основные составляющие: базу данных, клиентскую часть и серверную часть. Для каждого из этих элементов я тщательно анализировал возможные потребности пользователей и технические требования, чтобы обеспечить оптимальное взаимодействие между различными компонентами системы.

2.1 База данных

В основе успешного функционирования любого приложения лежит хорошо спроектированная база данных. Она должна быть способна не только эффективно хранить данные, но и обеспечивать быстрый доступ и возможность масштабирования под растущие нужды приложения. В проектируемом приложении база данных играет ключевую роль, так как именно здесь будет храниться информация о пользователях, их любимых исполнителях и концертных мероприятиях.

В процессе проектирования схемы базы данных я сосредоточился на следующих ключевых аспектах:

1. Связи и целостность данных: Определение связей между таблицами важно для поддержания логической структуры данных. Включение внешних ключей помогает поддерживать целостность данных при операциях вставки, обновления и удаления.
2. Индексация: Для ускорения поиска и извлечения данных используются индексы. Правильное применение индексации существенно улучшает производительность запросов, особенно в условиях больших объемов данных.

В проектировании базы данных для приложения, ориентированного на парсинг и представление данных о концертных событиях, ключевым является создание моделей, которые обеспечивают структурированное и эффективное хранение информации. Для визуализации связей между данными

использована схема, описывающая взаимодействие основных таблиц: event, artist, stage, и city (Рисунок 1).

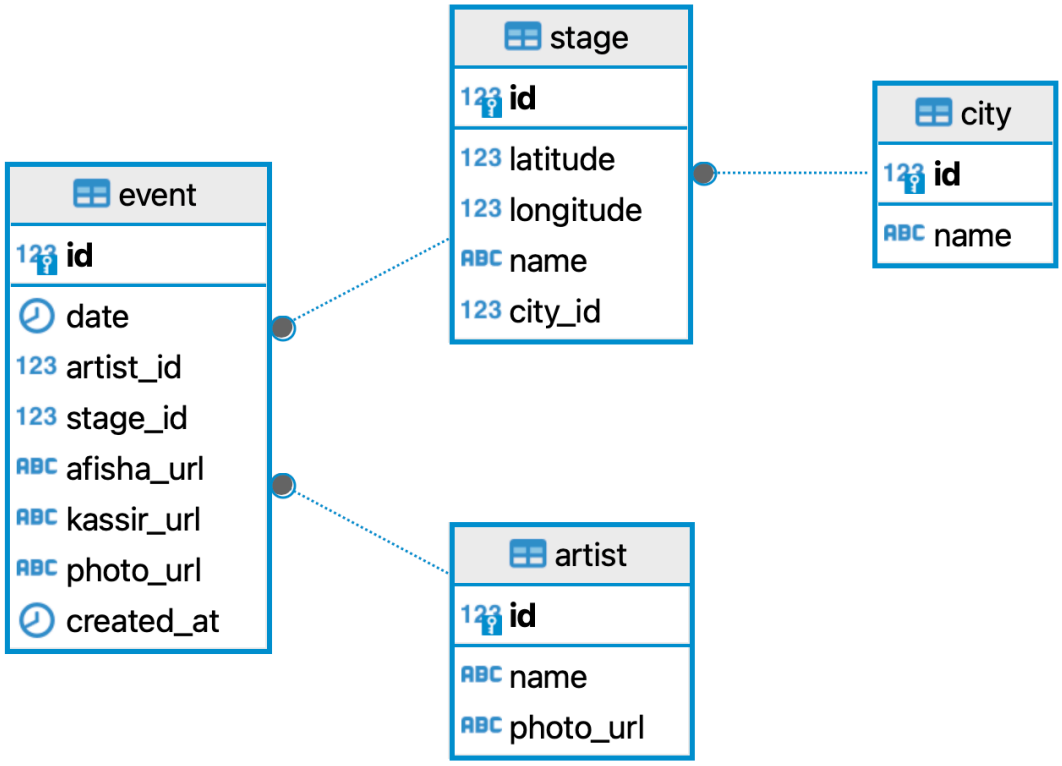

Рисунок 1 – Схема таблиц связанных с парсингом данных

1. Город (City): Каждый город в базе данных имеет уникальный идентификатор (id) и название (name).
2. Площадка (Stage): Площадка, где проходит событие, связана с городом и содержит информацию о своих географических координатах – широте (latitude) и долготе (longitude). Эти данные позволяют интегрировать функциональность карт в приложение, давая пользователю возможность открыть локацию площадки в приложениях картографических сервисов. Также каждая площадка имеет своё название (name), что помогает пользователю легко идентифицировать место проведения события.
3. Артист (Artist): В базе данных каждый артист имеет уникальный идентификатор (id), имя (name) и ссылку на фотографию (photo_url). Ссылка на изображение артиста используется для обогащения пользовательского интерфейса, делая его более визуально

привлекательным и информативным, а также помогает пользователю визуально распознать интересующего исполнителя.

4. Событие (Event): Событие является центральной таблицей базы данных, связывающей артиста, площадку и дату проведения (date). Каждое событие содержит ссылки на ресурсы для покупки билетов (afisha_url, kassir_url) и ссылку на изображение события (photo_url). Это позволяет пользователям перейти к покупке билетов непосредственно через приложение, упрощая процесс получения билетов на желаемое событие.

Для эффективного управления данными, полученными в результате парсинга афиш, важно обеспечить механизмы фильтрации и верификации информации, чтобы гарантировать её достоверность. В этом контексте были разработаны дополнительные таблицы (artist_filter, city_filter, и stage_filter). Эти таблицы служат в качестве промежуточного хранилища для данных, которые требуют дополнительной проверки перед их включением в основные таблицы базы данных.

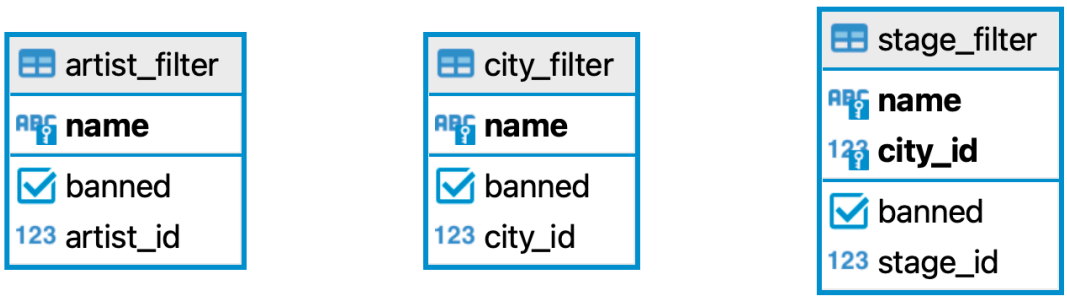

Рисунок 2 – Схема таблиц для верификации данных

Общая структура таблиц для фильтрации:

1. Наименование (name): Поле для хранения названия, которое может быть именем артиста, названием города или площадки, в зависимости от контекста. Это имя изначально извлекается из парсинга афиш и требует дополнительной проверки.
2. Исключено (banned): Булевый флаг, указывающий, следует ли исключить данную запись из возможности добавления в основные

таблицы. Это поле помогает отфильтровывать нежелательные или недостоверные данные.

3. Идентификатор (id): Ссылка на соответствующий идентификатор в основной таблице (artist_id, city_id, stage_id). Этот идентификатор устанавливается только после подтверждения соответствия данных стандартам и критериям верификации. В случае с идентификатором площадки используется компонентный ключ, где первый компонент – название площадки, а второй – идентификатор города. Это помогает различать площадки с одинаковым названием в разных городах, например «МТС Live Холл».

В рамках начального этапа разработки, города, в которых будет использоваться приложение, планируется одобрять вручную. Это решение позволяет контролировать и ограничивать область сервиса, фокусируясь на крупных городских центрах, таких как Санкт-Петербург и Москва. Информация об артистах, полученная через стриминговые сервисы, будет автоматически считаться надежной. Это решение основано на высокой достоверности данных, предоставляемых крупными и проверенными платформами. Артисты, идентифицированные через эти сервисы, будут напрямую добавляться в базу данных приложения и становиться доступными в поиске для пользователей.

В рамках базы данных приложения ключевое внимание уделяется не только управлению информацией о мероприятиях, но и интеграции с пользовательскими данными и их взаимодействием со стриминговыми сервисами.

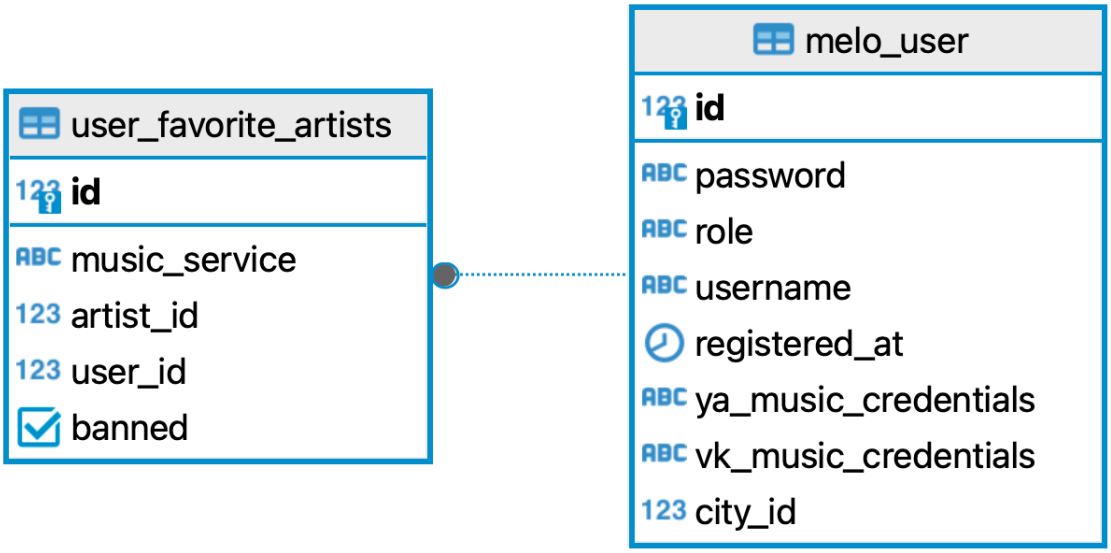

Рисунок 3 – Схема таблиц пользовательских данных

1. Таблица пользователей (melo_user): Эта таблица служит для хранения основной информации о пользователях приложения. Каждый пользователь идентифицируется уникальным “id”, и для входа использует “username” и зашифрованный “password”. Дополнительные поля включают “role”, который определяет уровень доступа пользователя и “registered_at”, указывающее на дату регистрации. Важной особенностью является наличие полей “ya_music_credentials” и “vk_music_credentials”, которые содержат данные для интеграции со стриминговыми сервисами, позволяя приложению получать информацию о музыкальных предпочтениях пользователя и персонализировать контент. Также имеется столбец “city_id” определяющий какой город выбран у пользователя в данный момент.
2. Таблица избранных артистов пользователя (user_favorite_artists): Таблица предназначена для сохранения связей между пользователями и их избранными артистами, отражая музыкальные предпочтения. “id” – это уникальный идентификатор каждой записи, “user_id” связывает запись с конкретным пользователем, а “artist_id” указывает на артиста из основной таблицы артистов. Поле “music_service”

отражает источник, откуда пользователь добавил артиста в избранное, что помогает отслеживать предпочтения пользователей в разрезе различных платформ. Поле “banned” позволяет определить, счел ли пользователь артиста, полученного через стриминговый сервис, вовсе ему неинтересным.

На рисунке 4 приведена полная схема получившейся базы данных со всеми связями:

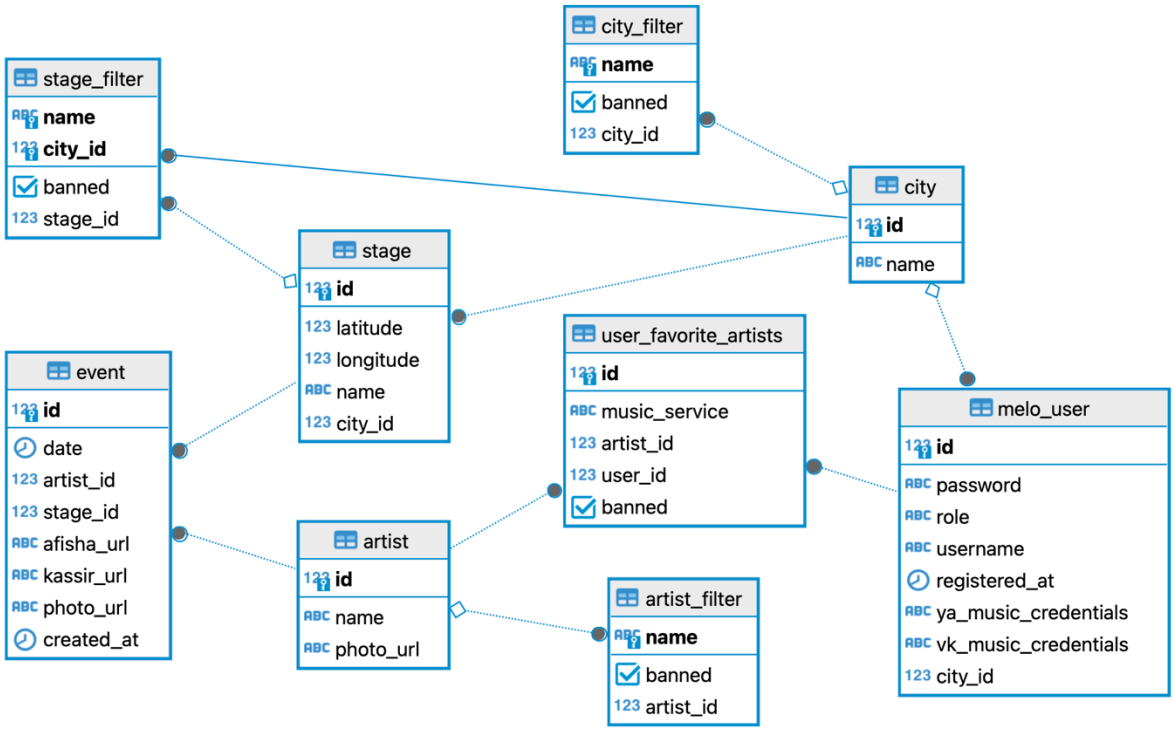

Рисунок 4 – Схема базы данных приложения

Для повышения производительности и удобства работы с базой данных были добавлены индексы в ключевые таблицы:

1. Индексация в таблице “user_favorite_artists”: Добавление индексов на “user_id” в таблице “user_favorite_artists” позволяет ускорить процессы извлечения данных о любимых артистах конкретного пользователя. Это особенно важно, когда пользователь захочет просмотреть или отредактировать свой список избранных артистов.
2. Индексация в таблице “melo_user”: В таблице “melo_user” был добавлен индекс на поле “username”, что способствует ускорению процесса аутентификации пользователей. Индекс по полю,

используемому для входа в систему, существенно сокращает время необходимое для поиска записи пользователя в базе данных, ускоряя таким образом процесс аутентификации.

Разработанная схема базы данных обеспечивает комплексное и масштабируемое решение для управления и хранения данных приложения, ориентированного на концертные события и музыкальные предпочтения пользователей. Структура данных включает как основные, так и фильтрующие таблицы, что позволяет не только аккуратно организовать информацию о мероприятиях, артистах и площадках, но и обеспечивает высокий уровень контроля над качеством входящих данных через промежуточные фильтры. Эффективное использование индексации и оптимизация запросов через интеграцию с внешними музыкальными сервисами позволяет приложению предлагать быстрый и персонализированный пользовательский опыт, делая его весьма адаптированным к будущим расширениям и обновлениям.

2.2 Архитектура приложения

В ходе разработки моего приложения, предназначенного для увеличения вовлеченности пользователей в концертные события, я обратил внимание на необходимость обеспечения гибкой и масштабируемой архитектуры. Основной задачей стало создание структуры, которая позволила бы эффективно обрабатывать пользовательские запросы и взаимодействовать с различными компонентами системы, такими как серверная часть, база данных и внешние API.

2.2.1 Высокоуровневая архитектура

Архитектура приложения включает в себя следующие ключевые элементы:

1. Фронтенд, построенный на базе React Native с использованием дополнительных инструментов, таких как Expo, для обеспечения универсальности и доступа к нативным функциям устройств.

2. Бэкенд, реализованный на Spring Framework, предоставляет мощные возможности для создания REST API, которые связывают фронтенд с базой данных и внешними сервисами.
3. База данных, настроенная на PostgreSQL, гарантирует надежное хранение и быстрый доступ к данным о пользователях, концертах и билетах.
4. Внешние сервисы, включающие афиши и музыкальные платформы, интегрируются через API для предоставления актуальной информации о музыкальных событиях и предпочтениях пользователей.

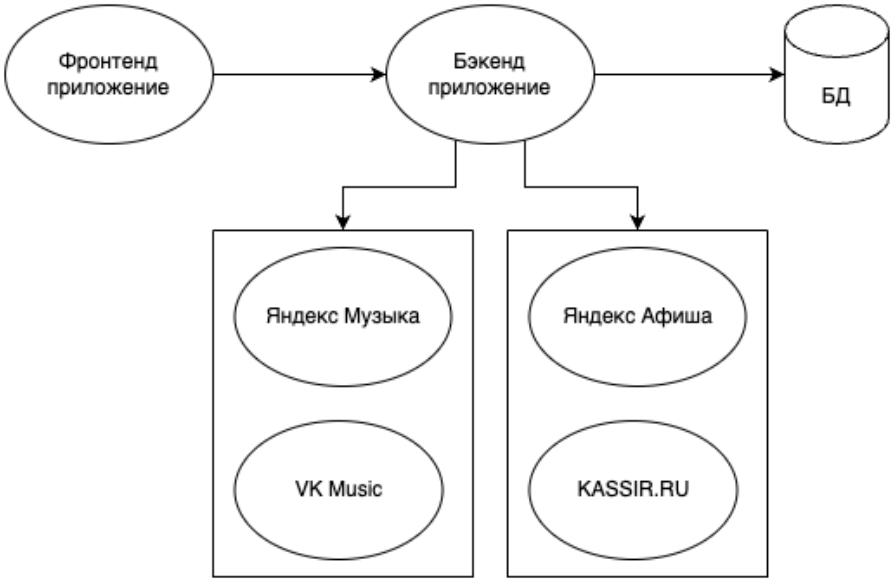

Рисунок 5 – Диаграмма компонентов приложения

Диаграмма компонентов приложения (Рисунок 5) иллюстрирует взаимосвязь и поток данных между вышеупомянутыми компонентами системы. Она поможет наглядно показать, как пользовательские запросы обрабатываются через различные слои приложения и как информация циркулирует в системе.

2.2.2 Организация модулей бэкенд приложения

В проекте используется Maven с родительским POM файлом, который устанавливает зависимость от “spring-boot-starter-parent” (родительский модуль, требующийся для разработки на Spring Boot), обеспечивая

совместимость конфигураций и версий библиотек. Также, для упрощения кода и его поддержки, в родительском файле применяется библиотека Lombok [18], автоматизирующая создание шаблонных методов через аннотации, что унифицирует структуру проекта и повышает эффективность разработки.

Проект разделен на три основных модуля, каждый из которых выполняет свои задачи и имеет четко выделенные зависимости:

1. Commons – Этот модуль содержит общие классы, такие как запросы и ответы, а также сущности, которые не требуют прямых зависимостей от основных библиотек фреймворка. Здесь используется “spring-boot-starter-validation” для валидации данных и Jackson [19] для сериализации и десериализации JSON. Это позволяет обеспечить корректное взаимодействие с клиентской частью и другими системами, а также поддерживать строгую типизацию и правильность данных на уровне API.
2. Domain – В этом модуле размещены сущности приложения, а также логика их взаимодействия с базой данных. Зависимость “spring-boot-starter-data-jpa” используется для интеграции с JPA, что позволяет эффективно управлять данными через высокоуровневые абстракции и операции. Кроме того, применяется MapStruct [20] для маппинга данных между DTO из Commons модуля и сущностями, что упрощает преобразование данных и уменьшает возможность ошибок в ручных преобразованиях.
3. Melo-Application – Этот модуль является ключевым в архитектуре серверной части и включает в себя настройки для веб-сервера, безопасности и управления сессиями. Зависимости, такие как “spring-boot-starter-web” для обработки HTTP-запросов, “spring-boot-starter-security” и “jjwt” для аутентификации и авторизации по JWT токену, а также “spring-boot-starter-webflux” для асинхронной обработки запросов на парсинг данных из источников. Использование WebFlux важно для

обработки нескольких параллельных потоков данных без блокировок, что повышает производительность приложения при высоких нагрузках.

Модульная архитектура серверной части, построенная с использованием Spring Boot, значительно упрощает разработку, тестирование и поддержку каждого компонента приложения отдельно. Эта структура обеспечивает гибкость и удобство в управлении изменениями, позволяя эффективно внедрять новые функции и обновления без риска нарушения работы остальной системы, что делает её идеальной для развития таких приложений.

2.2.3 Структура проекта клиентской части

В проектировании клиентской части приложения для повышения вовлеченности пользователей в концертные события я использовал целый ряд современных технологий и подходов, чтобы создать эффективный, удобный и масштабируемый фронтенд. Клиентская часть приложения базируется на React Native с использованием фреймворка Expo для облегчения разработки и развертывания на различных платформах. Для управления стилями я выбрал NativeWind, который предоставляет удобные утилиты для быстрой стилизации компонентов. Состояние приложения управляется с помощью Async Storage, что обеспечивает легкость и гибкость управления состоянием без больших затрат на настройку и поддержку.

Структура проекта:

1. package.json – этот файл играет ключевую роль в управлении зависимостями проекта, позволяя легко добавлять и обновлять библиотеки и фреймворки, используемые в приложении. Он содержит список всех внешних пакетов и их версий, которые необходимы для работы приложения.
2. assets/ – папка, предназначенная для хранения статических ресурсов, таких как изображения и шрифты. Она обеспечивает централизованное хранение файлов, которые могут быть легко использованы в разных частях приложения.
3. src/ – основная папка с исходным кодом приложения, включает:

4. api/ – содержит модули для выполнения HTTP-запросов к серверу и обработки ответов. Модули в этой папке отвечают за взаимодействие с бэкендом и преобразование полученных данных в объекты, которые можно использовать в компонентах.
5. app/ – в этой папке находятся компоненты высокого уровня, такие как страницы приложения, модальные окна и панели навигации. Это компоненты, которые напрямую взаимодействуют с пользователем.
6. components/ – папка для переиспользуемых компонентов, таких как кнопки, поля ввода и любые другие пользовательские интерфейсы, которые могут быть использованы в различных частях приложения. Разработка отдельных UI-компонентов обеспечивает унификацию внешнего вида и поведения элементов интерфейса по всему приложению.

Такая структура проекта не только облегчает навигацию по коду и его поддержку, но и способствует модульности и переиспользуемости компонентов, что является критически важным для эффективной работы и масштабирования приложения в будущем.

2.4 Безопасность приложения

Безопасность приложения является одним из ключевых аспектов, определяющих его надежность и доверие со стороны пользователей. В современном мире, где угрозы цифровой безопасности постоянно эволюционируют, важно использовать технологии и методики для защиты данных пользователей. Основное внимание в разделе безопасности уделяется механизмам, используемым на бэкенде, поскольку именно там обрабатывается большинство критически важных данных.

Для аутентификации и авторизации пользователей в приложении используется комбинация HTTP Basic и JWT. Процесс начинается с того, что пользователь вводит свои учетные данные, которые передаются на сервер в зашифрованном виде посредством HTTP Basic. Это первый уровень защиты, который обеспечивает конфиденциальность учетных данных при их передаче.

После проверки учетных данных на сервере, пользователю выдается JWT, который далее используется для доступа к защищенным ресурсам. JWT обладает рядом преимуществ, включая возможность хранения в токене необходимых для системы данных, что уменьшает количество обращений к серверу за дополнительной информацией о пользователе. Токены JWT настроены на автоматическое прекращение действия после определенного времени, что повышает безопасность системы.

Вся коммуникация между клиентом и сервером происходит через HTTPS, использующий протоколы SSL/TLS для шифрования данных. Это не только помогает защитить данные пользователя при передаче от перехвата третьими лицами, но и подтверждает, что пользователь действительно соединяется с верифицированным сервером. На сервере установлен сертификат SSL, который регулярно обновляется для предотвращения его компрометации. Этот сертификат подтверждает подлинность домена, что особенно важно для предотвращения фишинговых атак и атак на основе подмены домена. HTTPS является стандартом для любого приложения, которое обрабатывает конфиденциальную информацию, обеспечивая целостность и конфиденциальность передаваемых данных.

Благодаря этим мерам приложение обеспечивает высокий уровень безопасности как на уровне передачи данных, так и на уровне доступа к ресурсам. Это помогает минимизировать потенциальные угрозы безопасности и обеспечить пользователю безопасное и надежное использование сервиса.

2.5 Описание конечных точек сервиса

API приложения играет ключевую роль во взаимодействии клиентской части с сервером, позволяя осуществлять запросы и получать необходимые данные для функционирования приложения. Все запросы к API, за исключением начальной аутентификации, требуют подтверждения доступа через JWT, передаваемый в заголовке каждого запроса.

Описание конечных точек “User Controller”:

1. GET /api/user – Запрос на получение информации о пользователе, идентифицируемого по токену в заголовке. Возвращает данные пользователя, для отображения их в профиле.
2. PUT /api/user – Метод для обновления данных пользователя, например, изменение выбранного города.
3. GET /api/user/service-info/{service} – Получение информации о связи пользователя со стриминговыми сервисами. **{service}** в URL заменяется на название сервиса, информация о котором запрашивается, включая логин пользователя в этом сервисе и список его избранных исполнителей.

Описание конечных точек “Auth Controller”:

1. POST /api/auth/sign-up – Регистрация пользователя с получением JWT. Пользователь отправляет свои учетные данные (логин и пароль), и в ответ получает JWT для доступа к защищенным разделам приложения.
2. POST /api/auth/sign-in – Аутентификация пользователя и получение JWT. Процесс аналогичен регистрации, но предназначен для уже зарегистрированных пользователей.
3. POST /api/auth/yandex и /api/auth/vk – Эти точки используются для аутентификации через соответствующие социальные сети Яндекс и ВКонтакте. Пользователь перенаправляется на страницу авторизации соцсети, и после подтверждения доступа, приложение получает токены для работы от имени пользователя.

Описание конечных точек “Event Controller”:

1. GET /api/event/{id} – Предоставляет развернутую информацию о конкретном концертном мероприятии по идентификатору {id}. Включает в себя все связанные сущности, такие как исполнители, местоположение события и другие детали.
2. POST /api/event/search – Позволяет выполнить детализированный поиск событий по различным критериям. Функционал поиска включает

возможности сортировки, фильтрации по новизне событий, любимым исполнителям, дате начала концерта и другое.

Описание конечных точек “City Controller”:

1. GET /api/city – Запрос на получение списка доступных городов. Этот метод предоставляет пользователю возможность выбрать город из списка, что используется для уточнения поиска событий.

Описание конечных точек “Artist Controller”:

1. PUT /api/artist/{id} – Метод позволяет добавлять или удалять артиста из списка избранных. Эта функция предоставляет пользователям возможность персонализировать свои предпочтения.
2. GET /api/artist/search – Поиск артистов по именам. Запрос позволяет пользователям находить исполнителей, что облегчает открытие новых артистов для добавления их в любимые.

Также в приложении предусмотрена обработка ошибок. При возникновении ошибок на стороне сервера или при неправильных запросах клиента, система предоставляет информативные сообщения об ошибках. Каждый ответ об ошибке будет содержать поле “message”, в котором детально описывается, возникшая ошибка и средства как эту ошибку избежать, например правильно заполнить то или иное поле. Это позволяет клиентам приложения быстро понять причину возникновения ошибки и предпринять необходимые шаги для успешного выполнения запроса.

Стандартные коды HTTP состояния используются для указания типа ошибки (например, 400 для ошибок в запросе, 401 для проблем с аутентификацией, 404 для не найденных ресурсов и 500 для внутренних ошибок сервера). Такой подход помогает в быстром диагностировании и решении проблем.

3 РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ

В этом разделе описывается процесс создания функциональных элементов приложения. От бэкенда и парсеров, заканчивая фронтендом.

3.1 Реализация основной части бэкенда

3.1.1 Пример реализации сущностей и работа с ними

В моем приложении применяется подход объектно-реляционного отображения (ORM), который в Spring реализуется через аннотации JPA. Каждая сущность в системе аннотирована как @Entity, что позволяет Spring контексту распознавать классы как таблицы базы данных. В каждой сущности должен быть определен уникальный идентификатор с аннотацией @Id, что необходимо для корректной организации и управления записями в базе данных.

```
@Entity  26 usages  Grigory Yakovlev
@Getter
@Setter
@Builder
@ToString
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "stage")
public class Stage {

    @Id
    @Column(name = "id", nullable = false)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "stage_id_seq")
    @SequenceGenerator(sequenceName = "stage_id_seq", name = "stage_id_seq", allocationSize = 1)
    private Long id;

    @NonNull
    @Column(name = "name", nullable = false)
    private String name;

    @NonNull
    @ToString.Exclude
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "city_id", nullable = false)
    private City city;

    @Column(name = "latitude")
    private Double latitude;

    @Column(name = "longitude")
    private Double longitude;

}
```

Рисунок 6 – Пример реализации сущности “Stage”

Для примера на Рисунке 6 изображена одна из моих сущностей – “Stage” – сцена, на которой выступают исполнители. Для сопоставления точного названия таблиц и колонок используются аннотации @Table и @Column. Здесь также используется связь с другой сущностью “City” через аннотации @ManyToOne и @JoinColumn. Аннотация @JoinColumn указывает, что в таблице “Stage” будет колонка “city_id”, которая связана с первичным ключом сущности “City”. Параметры “nullable = false” и “optional = false” на примере с “City” указывают, что каждая сцена должна быть связана с городом, и наличие города является обязательным. @GeneratedValue и @SequenceGenerator позволяют мне точно подключиться к называемой последовательности.

Для упрощения кода, автоматизации генерации стандартных методов и более надежного создания объектов, к каждой сущности я добавляю аннотации из библиотеки Lombok:

1. @Getter и @Setter – Автоматически генерируют методы для доступа и изменения значений полей класса.
2. @Builder – Используется мной для более надежного создания объектов через паттерн “Строитель”. С комбинацией аннотации @NonNull это позволяет на этапе компиляции обнаружить что где-то сущность создается не со всеми требуемыми полями.
3. @NoArgsConstructor, @AllArgsConstructor – Генерируют конструкторы без параметров и с параметрами для всех полей класса, что необходимо для преобразования объекта библиотекой MapStruct в будущем.
4. @ToString – Переопределяет метод “toString()”, который возвращает строковое представление объекта с названиями и значениями его полей. Полезно для логирования и отладки.

Для работы с сущностями в коде приложения необходимы репозитории. Это компоненты, основанные на паттерне “Repository”, который предоставляет коллекцию методов для выполнения различных операций с данными без необходимости писать конкретный SQL-код.

```
@Repository  2 usages  Grigory Yakovlev
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);  2 usages  Grigory Yakovlev
    boolean existsByUsername(String username);  2 usages  Grigory Yakovlev
}
```

Рисунок 7 – Пример реализации репозитория “UserRepository”

На Рисунке 7 изображен пример собственного репозитория, который расширяет интерфейс JpaRepository из Spring Data, предоставляя все стандартные методы для сохранения, удаления и извлечения данных пользователя. Кроме того, в нём определены собственные методы, такие как:

1. Метод для поиска пользователя по имени, возвращая обернутый объект User. Это обеспечивает обработку случаев, когда пользователь не найден.
2. Метод для проверки существования пользователя с заданным именем. Нужен при проверке регистрации новых пользователей для предотвращения дублирования имен.

```
@Mapper(componentModel = "spring")  1 inheritor  Grigory Yakovlev
public abstract class UserMapper {

    @Mapping(target = "cityName", source = "city.name")  1 implementation  Grigory Yakovlev
    public abstract UserResponse map(User entity);

}
```

Рисунок 8 – Пример реализации преобразователя “UserMapper”

Для преобразования данных между различными слоями приложения, такими как база данных и клиентский интерфейс, используются преобразователи (маперы). На Рисунке 8 изображен такой, реализованный с помощью библиотеки MapStruct. Я использую маперы, чтобы упростить процесс трансформации данных, минимизируя необходимость в ручном написании кода для превращения одного представления в другое.

Библиотека MapStruct, автоматически генерирует код для маппинга данных между объектами Java на основе аннотаций. В случае с UserMapper, аннотация @Mapper(componentModel = "spring") указывает MapStruct

генерировать реализацию мапера как Spring бин, что позволяет внедрять его в другие компоненты Spring через автоматическое связывание зависимостей.

А аннотация @Mapping(target = "cityName", source = "city.name") используется для настройки маппинга отдельных свойств объекта. В данном случае она указывает MapStruct преобразовать значение “city.name” из сущности User в “cityName” в объекте UserResponse. Её я использую, чтобы предоставлять только необходимую информацию от сервера.

3.1.2 Реализация аутентификации и авторизации

Для реализации механизмов аутентификации и авторизации была использована библиотека Spring Security, которая предоставляет мощные и гибкие инструменты для защиты веб-приложений. Основа безопасности приложения строится на использовании HTTP Basic Authentication для первичной аутентификации пользователей и JSON Web Tokens (JWT) для поддержания сессии пользователя после его входа в систему. Использование JWT упрощает процесс управления сессиями, позволяя избежать повторной аутентификации при каждом запросе и уменьшая нагрузку на систему.

Для интеграции HTTP Basic Authentication, в конфигурацию безопасности был встроен сервис UserDetailsService, который позволяет загружать данные пользователя по его имени. Данные пользователя – это класс User в моем приложении, который наследуется от UserDetails, что позволяет UserDetailsService работать с ним, абстрагируясь от конкретной реализации. Это стандартный подход для работы с аутентификационными данными пользователей в Spring Security.

Для обеспечения безопасности на основе токенов был создан специализированный фильтр JwtAuthenticationFilter, который настраивается для работы перед UsernamePasswordAuthenticationFilter. Этот фильтр интерцептирует запросы, извлекает JWT из заголовка Authorization, и, если токен валиден, аутентифицирует пользователя в контексте Spring Security.

```java
@Component  Grigory Yakovlev *
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;
    private final UserService userService;

    @Override  no usages  Grigory Yakovlev *
    protected void doFilterInternal(@NonNull HttpServletRequest request,
                                    @NonNull HttpServletResponse response,
                                    @NonNull FilterChain filterChain) throws ServletException, IOException {
        //1. Получаем токен из заголовка
        var authHeader = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (StringUtils.isEmpty(authHeader) || !StringUtils.startsWith(authHeader, Constants.BEARER)) {
            filterChain.doFilter(request, response);
            return;
        }

        //2. Обрезаем префикс и получаем имя пользователя из токена
        var jwt = authHeader.substring(Constants.BEARER.length());
        var username = jwtUtil.extractUserName(jwt);

        //3. Проверка перед аутентификацией
        if (StringUtils.isNotEmpty(username) && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = userService.loadUserByUsername(username);

            //4. Если токен валиден, то аутентифицируем пользователя
            if (jwtUtil.isTokenValid(jwt, userDetails)) {
                SecurityContext context = SecurityContextHolder.createEmptyContext();

                //5. Установить аутентификацию
                UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
                        userDetails,  credentials: null, userDetails.getAuthorities());

                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                context.setAuthentication(authToken);
                SecurityContextHolder.setContext(context);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

Рисунок 9 – Фильтр аутентификации приложения

На Рисунке 9 представлена реализация JwtAuthenticationFilter. В этом классе происходит следующее:

1. Из заголовка Authorization извлекается JWT.
2. С помощью вспомогательного класса JwtUtil, который содержит методы для работы с JWT, из токена извлекается имя пользователя.
3. Если имя пользователя не пустое и пользователь еще не аутентифицирован, с помощью UserDetailsService загружаются данные пользователя.
4. Проверяется валидность токена с учетом данных пользователя.

5. Если токен валиден, создается аутентификационный токен (UsernamePasswordAuthenticationToken), который устанавливается в контекст безопасности, обеспечивая тем самым аутентификацию пользователя на уровне приложения.

Такой подход позволяет эффективно управлять аутентификацией пользователя, обеспечивая безопасность и удобство использования приложения.

3.1.3 Реализация поиска

В разработке модуля поиска в приложении я столкнулся с необходимостью создания гибкой и мощной системы поиска, способной поддерживать различные критерии фильтрации, сортировку и пагинацию. Вместо того чтобы создавать множество специфичных методов в репозитории для каждого потенциального запроса, я выбрал подход, основанный на использовании CriteriaQuery из JPA (Java Persistence API).

```
private CriteriaQuery<Event> createCriteriaQuery(SearchEventsRequest request,    1 usage  Grigory Yakovlev
                                                 User user) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<Event> cq = builder.createQuery(Event.class);
    cq.distinct(true);

    var predicates = new ArrayList<Predicate>();
    var root = cq.from(Event.class);
    var joinArtist = root.join(Event_.artist, JoinType.LEFT);
    var joinStage = root.join(Event_.stage, JoinType.LEFT);
    var joinCity = joinStage.join(Stage_.city, JoinType.LEFT);

    var artistName = request.getArtistName();
    if (artistName != null) {
        artistName = artistName.trim().toLowerCase();
        var predicateForArtist = builder.like(
                builder.lower(joinArtist.get(Artist_.name)),
                pattern: "%" + artistName + "%");
        predicates.add(predicateForArtist);
    }

    var dateFrom = request.getDateFrom();
    if (dateFrom != null) {...}

    var dateTo = request.getDateTo();
    if (dateTo != null) {...}
```

Рисунок 10 – Пример поиска через CriteriaQuery

CriteriaQuery позволяет динамически строить запросы к базе данных, основываясь на условиях, которые могут изменяться в зависимости входных данных. Это дает возможность единообразно обрабатывать различные типы фильтров, сортировок и поддерживать пагинацию без необходимости переписывания или добавления новых методов в коде.

Приведенный код на Рисунке 10 демонстрирует, как можно эффективно использовать CriteriaQuery для создания сложных запросов в базе данных. В данном примере:

1. CriteriaBuilder и CriteriaQuery используются для создания и формирования запроса. CriteriaBuilder служит для создания различных частей запроса, в то время как CriteriaQuery определяет тип возвращаемого результата и структуру запроса.
2. Root определяет основную сущность, по которой будет производиться запрос (в данном случае Event).
3. Join используется для соединения таблиц, что позволяет включать в запрос информацию из связанных сущностей, таких как артисты, сцены и города. Это обеспечивает возможность фильтрации и сортировки по полям, которые находятся не только в основной таблице.
4. Predicates формируют условия для фильтрации запросов. В примере создается предикат для фильтрации по имени артиста, исключая регистр символов. Подобные условия добавляются в запрос, чтобы отфильтровать результаты в соответствии с предоставленными пользователем значениями.

Использование CriteriaQuery предоставляет значительные преимущества в гибкости и масштабируемости приложения, позволяя адаптировать функциональность поиска под различные требования. Этот подход не только облегчает добавление новых типов фильтров и критериев поиска в будущем, но и значительно упрощает поддержку и развитие функционала.

3.1.4 Пример реализации контроллера

Контроллеры необходимы для получения данных с бэкенда в приложение. При реализации контроллеров я в первую очередь акцентирую внимание на простоте и безопасности.

```
@RestController  Grigory Yakovlev *
@RequiredArgsConstructor
@RequestMapping(value = "/api/event")
public class EventController {

    private final EventService eventService;
    private final EventSearchHandler eventSearchHandler;

    @GetMapping("/{id}")  Grigory Yakovlev *
    public ResponseEntity<ExtendedEventResponse> read(@PathVariable("id") Long id) {
        return new ResponseEntity<>(
                eventService.read(id),
                HttpStatus.OK
        );
    }

    @PostMapping("/search")  Grigory Yakovlev *
    public ResponseEntity<SearchEventsResponse> search(
            @Validated @RequestBody SearchEventsRequest request,
            Authentication authentication) {
        return new ResponseEntity<>(
                eventSearchHandler.search(request, authentication.getName()),
                HttpStatus.OK
        );
    }
```

Рисунок 11– Фрагмент реализации контроллера “EventController”

На Рисунке 11 изображен контроллер EventController, который является ключевым компонентом для обработки запросов, связанных с мероприятиями. Контроллер использует аннотации Spring MVC для определения маршрутов и обработки HTTP запросов:

1. @RestController – Указывает, что класс является контроллером в Spring MVC, и все методы возвращают данные в формате, подходящем для REST API, по умолчанию в JSON.

2. @RequestMapping("/api/event") – Устанавливает базовый URL для всех методов в этом контроллере, что упрощает навигацию и обеспечивает структурирование URL-адресов API.
3. @GetMapping("/{id}") – Определяет метод, который обрабатывает GET запросы на получение информации о событии по его идентификатору. Использование @PathVariable позволяет извлекать идентификатор события прямо из URL.
4. @PostMapping("/search") – Метод обрабатывает POST запросы для выполнения поиска событий на основе предоставленных критериев. Используется @RequestBody для получения данных запроса в виде объекта, и @Validated для обеспечения валидации этих данных перед обработкой.

В контексте безопасности и аутентификации, некоторые мои методы используют объект Authentication, чтобы получить, например, имя пользователя, выполнившего запрос. Это позволяет реализовать функционал, зависимый от пользователя. Например, в методе search имя пользователя используется для уточнения города, выбранного пользователем в контексте приложения.

Возвращаемый тип ResponseEntity в методах контроллера позволяет управлять HTTP ответом, включая код состояния и тело ответа. Например, успешный запрос на чтение данных о событии вернет статус 200 (OK), вместе с данными события в формате JSON.

В случае возникновения ошибки во время выполнения любого из запросов, специализированный контроллер для обработки ошибок перехватит исключение и автоматически сформирует ответ с информативным сообщением о произошедшей ошибке. Это позволяет пользователю понять причину проблемы и, при необходимости, предпринять соответствующие действия для её устранения.

3.2 Разработка парсера концертных событий

При попытке интеграции с частными API различных афишных сервисов, я столкнулся с рядом препятствий. Некоторые сервисы либо отказывали в предоставлении доступа к API, уверяя что у них самих нет такой технологии, либо пояснили, что не могут выдать доступ к API для внешних разработчиков. В результате мною было принято решение получать данные другим образом. Современные технологии веб-разработки часто подразумевают, что фронтенд страницы загружается со статического сервера, а затем активно используются методы для запроса данных через скрытые API. Именно ими я решил воспользоваться для получения информации о концертных событиях.

В рамках разработки приложения для повышения вовлеченности пользователей в концертные события, ключевым аспектом является сбор и обработка информации о концертах. Это требует создания гибкой системы, способной адаптироваться к разнообразным форматам данных различных источников.

```
public interface ParserStrategy {  4 usages  2 implementations  Grigory Yakovlev *

    Парсит данные из URL.

    void parse();  1 usage  1 implementation  Grigory Yakovlev

    Обрабатывает ошибку, возникшую во время парсинга.

    Params: e – Исключение, возникшее во время парсинга

    void handleError(@NonNull Exception e);  1 usage  1 implementation  Grigory Yakovlev
}
```

Рисунок 12 – Интерфейс парсера “ParserStrategy”

Для реализации данной задачи мною был разработан интерфейс ParserStrategy, который также используется для автоматического обнаружения наследующихся компонентов в контексте Spring. ParserStrategy имеет основной метод “parse()”, при вызове которого запустится процесс парсинга сервиса. Такой паттерн проектирования, когда предоставляется урезанный интерфейс, не имеющий 100% функциональности, называется “Фасад”. Таким

образом, ежедневные запланированные задачи на парсинг будут запускаться, не зная деталей их реализации.

Далее, как реализовывается конкретный парсер рассмотрим на примере парсера Яндекс Афиши. Для интеграции с сервисом необходимо создать класс, который будет наследоваться от интерфейса ParserStrategy и реализовывать наследуемые методы. Метод parse() будет активироваться ежедневной задачей, предназначенной для автоматического сбора данных. Он представлен в приложении А

Логика работы парсеров концертных событий начинается с извлечения данных о городах, в которых действует афишный сервис. После получения списка городов система производит запросы для каждого города отдельно, собирая информацию о всех предстоящих концертах.

Важно соблюдать все требований сервиса к запросам. Например, для Яндекс Афиши необходимо включить в заголовки запроса специальный параметр X-Parent-Request-Id, который может быть любым значением, но его наличие обязательно. Эти заголовки, вместе с адресами для парсинга, задаются в файле applications.properties, который используется для конфигурации Spring приложения.

Ограничения API Яндекс Афиши не позволяют запрашивать информацию обо всех мероприятиях одновременно, поэтому запрашиваю их пачками по 20 штук. Как только данные успешно получены, они приходят в формате JSON, который затем трансформируется в объекты, специфичные для каждого афишного сервиса, например AfishaEvents. В нем помимо информации о концертных событиях я также получаю данные о пагинации, и продолжаю запрашивать новые концерты, пока не придет флаг hasNext = false. Далее специфичный объект преобразовывается в универсальный формат, понятный приложению. Это позволяет отправить информацию о концерте дальше в приложение, где впоследствии эти данные будут обработаны и сохранены.

3.3 Разработка парсера данных со стриминговых сервисов

Стриминговые сервисы такие как ВК Музыка и Яндекс Музыка, не предоставляют специализированных API для доступа к данным о любимых треках пользователей. Однако я выяснил что данные о любимых треках можно сделать публичными по настройкам приватности, таким образом, я собираюсь получать эту информацию снова по скрытым API.

Однако, чтобы гарантировать, что пользователь действительно является владельцем учетной записи на музыкальном сервисе, было решено ввести авторизацию пользователя через соответствующий музыкальный сервис с использованием протокола OAuth 2.0.

Процесс настройки OAuth 2.0 является стандартным для большинства сервисов и включает несколько ключевых шагов:

1. Регистрация приложения: Сначала необходимо зарегистрировать своё приложение в панели разработчика, чтобы получить уникальные идентификаторы client_id и client_secret, необходимые для дальнейшей авторизации.
2. Получение кода доступа: Пользователь перенаправляется на страницу сервиса для входа в систему и разрешения доступа приложению к своим данным. После успешного входа сервис перенаправляет пользователя обратно в приложение с кодом доступа в URL.
3. Обмен кода на токен: Приложение обменивает полученный код на токен доступа с помощью запроса к сервису, включая в запрос client_id, client_secret, и код доступа.
4. Использование токена: Полученный токен используется для запросов к API.

На бэкенде я реализовал логику, которая позволяет обменивать код на токен и использовать этот токен для выполнения единственного запроса – получения логина пользователя. Это ключевой шаг, позволяющий связать пользователя в приложении с его учетной записью в Яндекс Музыке.

Процесс парсинга данных со стриминговых сервисов реализован аналогично сервисам афиш, с тем отличием, что вместо событий извлекаются данные о треках. Так же, как и в случае с афишами, ежедневные задачи на парсинг необходимы для обновления информации о любимых исполнителях пользователей. На примере Яндекс Музыки я расскажу, как это реализовано. В классе, созданном для парсинга данных со стримингового сервиса, я создаю новый метод parseUser(User user) который будет вызываться внутри метода parse() и будет собирать информацию о треках каждого пользователя отдельно. Метод parseUser(User user) прикреплен в приложении Б.

В рамках нового метода начальным шагом является получение библиотеки пользователя, в виде идентификаторов треков, которые пользователь отметил как любимые. Затем, по другому адресу Яндекс Музыки, осуществляется отправка запросов для получения дополнительной информации о каждом треке по его идентификатору.

В процессе анализа каждого трека система изучает артистов, участвующих в создании трека, и добавляет их в список любимых артистов пользователя, если они еще не были включены ранее.

3.4 Реализация фронтенда

Особенностью моего проекта является использование React Native, что позволяет мне разрабатывать приложение не только для мобильных платформ, таких как Android и iOS, но и для веба одновременно. Я решил, что буду поддерживать сразу все эти платформы, потому что раньше не имел опыта подобной разработки.

Фронтенд приложение является ключевой частью, которая напрямую взаимодействует с пользователем. Разработка клиентской стороны включает в себя создание интерфейса, реализацию пользовательских взаимодействий, а также обеспечение надежной связи с серверной частью для обработки и отображения данных.

3.4.1 Реализация функций запросов клиента

Важной задачей фронтенда является взаимодействие с сервером для получения и отправки данных. Это достигается через различные API вызовы, которые должны быть эффективно реализованы для обеспечения стабильности и безопасности приложения.

```
import { API_URL } from "@env";

export interface FetchUserInfoResponse {  Show usages   Grigory Yakovlev *
  username?: string;
  registeredAt?: string;
  role?: string;
  cityName?: string;
  message?: string;
}

export const fetchUserInfo = async (token: string) : Promise<FetchUserInfoResponse>  => {
  try {
    const response : Response  = await fetch( input: `${API_URL}/api/user`,  init: {
      method: "GET",
      headers: {
        Authorization: `Bearer ${token}`,
        "Content-Type": "application/json",
      },
    });

    const data: FetchUserInfoResponse = await response.json();
    if (!response.ok) {
      throw new Error( message: data.message || "Unknown error");
    }

    return data;
  } catch (error) {
    throw new Error((error as Error).message);
  }
};
```

Рисунок 13 – Запрос на получении данных о пользователе

Для организации запросов к серверу используется Fetch API, позволяющий отправлять HTTP-запросы асинхронно. URL сервера и другие конфигурационные параметры хранятся в переменных окружения, что упрощает управление конфигурацией при развертывании приложения в различных средах.

Пример функции получения данных о пользователе (Рисунок 13) демонстрирует использование TypeScript для типизации ответа, что помогает при последующей обработке приложением. Также запрос включает в себя токен аутентификации, передаваемый в заголовке, позволяющий идентифицировать пользователя, отправившего запрос.

В случае возникновения ошибок система генерирует исключение с сообщением, которые возвращаются пользователю. Это позволяет пользователю понять причину проблемы и применить меры для устранения ошибки.

Все остальные функции взаимодействия с сервером в приложении реализованы аналогичным образом. Такой подход обеспечивает консистентность в обработке запросов и ответов на всех уровнях приложения. Использование переменных окружения для хранения конфигурационных параметров, типизация через TypeScript для улучшения поддержки и предотвращения ошибок на этапе компиляции, а также стандартизированная система обработки ошибок с информативными сообщениями – все это является общими чертами для всех функций обмена данными с бэкендом.

3.4.2 Реализация компонентов приложения

В процессе разработки приложения я акцентировал внимание на модульности кода, что критически важно для обеспечения его масштабируемости и удобства поддержки. Использование компонентной архитектуры в React Native позволяет мне разделять интерфейс на мелкие, независимые блоки, что существенно упрощает процесс разработки и тестирования.

Я старался разрабатывать компоненты так, чтобы они были максимально переиспользуемыми. Это позволяет мне значительно ускорить разработку новых функций и интерфейсов, так как многие элементы могут быть легко адаптированы под новые задачи без дополнительных изменений кода. Каждый компонент в моём приложении принимает параметры, что делает его гибким и легко адаптируемым к разным сценариям использования.

```tsx
interface ProfileInfoProps {  Show usages  Grigory Yakovlev *
  userInfo: FetchUserInfoResponse;
}

const ProfileInfo: React.FC<ProfileInfoProps> = ({ userInfo : FetchUserInfoResponse  }) => {
  return (
    <View className="my-2 p-2 rounded-xl bg-white dark:bg-neutral-900">
      <Text className="text-xl font-bold text-neutral-900 dark:text-neutral-200">
        {userInfo.username}
      </Text>
      <Link href="/choose-city" asChild>
        <Pressable>
          {({ pressed : boolean  }) => (
            <Text
              className={`${pressed ? "opacity-50" : "opacity-100"} text-base text-blue-500`}
            >
              {userInfo.cityName} {">"}
            </Text>
          )}
        </Pressable>
      </Link>
      <Text className="text-sm text-neutral-600 dark:text-neutral-400">
        Зарегистрирован с {userInfo.registeredAt}
      </Text>
    </View>
  );
};

export default ProfileInfo;  Show usages  Grigory Yakovlev
```

Рисунок 14 – Реализация компонента “ProfileInfo”

На примере компонента ProfileInfo, который изображен на Рисунке 14, можно увидеть, как компоненты могут быть эффективно настроены для отображения данных. Компонент принимает объект userInfo, содержащий данные пользователя и отображает различные данные, такие как имя пользователя, город и дату регистрации. Также реализована навигация на экран с изменением города.

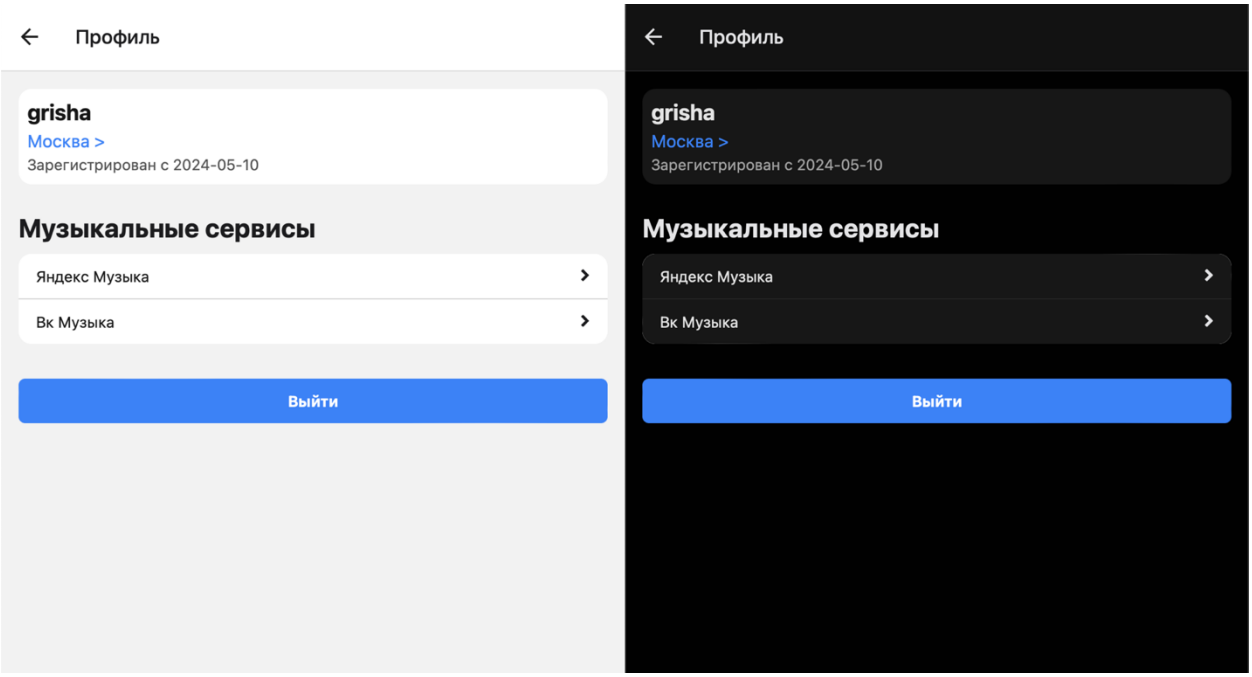

Рисунок 15 – Пример стилизации компонентов с использованием NativeWind

Для стилизации компонентов я выбрал NativeWind. Стили описываются в поле “className”, что позволяет мне быстро и эффективно управлять ими на месте. NativeWind также поддерживает условное форматирование стилей, что упростило внедрение таких функций, как поддержка темной темы.

Такая стратегия разработки компонентов в React Native значительно упрощает разработку и повышает качество интерфейса приложения.

3.4.3 Реализация основной панели приложения

Для настройки маршрутизации в приложении я использую Expo Router. Именно он обеспечивает навигацию между различными экранами приложения. Маршрутизация конфигурируется с помощью компонентов Stack и Stack.Screen, где каждый экран приложения ассоциируется с уникальным путем. Это позволяет автоматически обрабатывать переходы пользователей между разделами приложения, используя стандартные URL-пути.

Файл index.tsx представляет собой начальную страницу приложения и настраивается как корневой путь “/”. На этой странице будет предоставляться информация о концертах, включая ближайшие мероприятия, мероприятия любимых исполнителей и новые события, которые недавно были добавлены в сервис.

```
export default function Index() {  no usages  Grigory Yakovlev *
  const [jwtToken, setJwtToken] = useState<string | undefined>();
  const [loading, setLoading] = useState<boolean>( initialState: true);
  const [userEvents, setUserEvents] = useState<EventData[] | undefined>();
  const [nearestEvents, setNearestEvents] = useState<EventData[] | undefined>();
  const [newestEvents, setNewestEvents] = useState<EventData[] | undefined>();

  useEffect( effect: () => {
    const init = async () => {  Show usages  new *
      const token = await storage.get("jwtToken");
      setJwtToken( value: token ? token : undefined);

      if (token) {
        try {
          await loadEvents(token);
        } finally {
          setLoading( value: false);
        }
      } else {
        setLoading( value: false);
      }
    };

    init();
  }, deps: []);

  const loadEvents = async (token: string) => {...};
```

Рисунок 16 – использование React Hooks в компоненте Index

На Рисунке 16 представлен код, который иллюстрирует использование React Hooks, таких, как useState и useEffect, для управления состоянием в компоненте React:

1. useState используется для создания локального состояния в компоненте. В данном примере я создаю несколько состояний: jwtToken для хранения токена аутентификации, loading для отслеживания состояния загрузки данных, userEvents, nearestEvents, newestEvents для хранения различных категорий концертных событий.
2. useEffect описывает побочные эффекты, которые выполняются после рендеринга компонента. В данном случае useEffect используется для

инициализации – извлечения токена из локального хранилища и загрузки событий, если токен доступен. Это обеспечивает, что данные о событиях загружаются сразу после того, как пользователь входит в приложение.

```
return (
  <ScrollView className="flex-1">
    <View className="mx-4 my-2">
      {loading ? (
        <ActivityIndicator size="large" color="#0000ff" />
      ) : jwtToken ? (
        <>
          {userEvents && userEvents.length > 0 && (
            <EventStack events={userEvents} headerText="Ваши концерты" />
          )}
          {nearestEvents && nearestEvents.length > 0 && (
            <EventStack
              events={nearestEvents}
              headerText="Ближайшие концерты"
            />
          )}
          {newestEvents && newestEvents.length > 0 && (
            <EventStack events={newestEvents} headerText="Новые концерты" />
          )}
        </>
      ) : (
        <Text className="text-lg text-center px-4">
          Авторизуйтесь в приложении.
        </Text>
      )}
    </View>
  </ScrollView>
);
```

Рисунок 17 – код рендеринга начальной страницы

На Рисунке 17 демонстрируется рендеринг начальной страницы приложения, который использует условные рендеринги и компоненты для динамического отображения содержимого в зависимости от состояния приложения.

Если данные ещё загружаются (состояние loading равно true), то в интерфейсе отображается индикатор загрузки (ActivityIndicator). Это позволяет пользователю видеть, что процесс получения данных находится в активной фазе.

Как только процесс загрузки завершился, и переменная loading устанавливается в false, выполняется проверка наличия токена JWT. Если токен присутствует, приложение показывает различные категории концертных событий, разделённые на блоки.

Для каждой категории используется компонент EventStack, который принимает массив событий и заголовок, отображая список соответствующих концертов.

Если пользователь не аутентифицирован, то есть отсутствует токен JWT, приложение предлагает авторизоваться.

Получившийся компонент использует современные подходы React, такие, как условный рендеринг и хуки. Это позволяет приложению динамично реагировать на изменения состояния и предоставлять разнообразный контент, зависящий от данных пользователя и его статуса в системе.

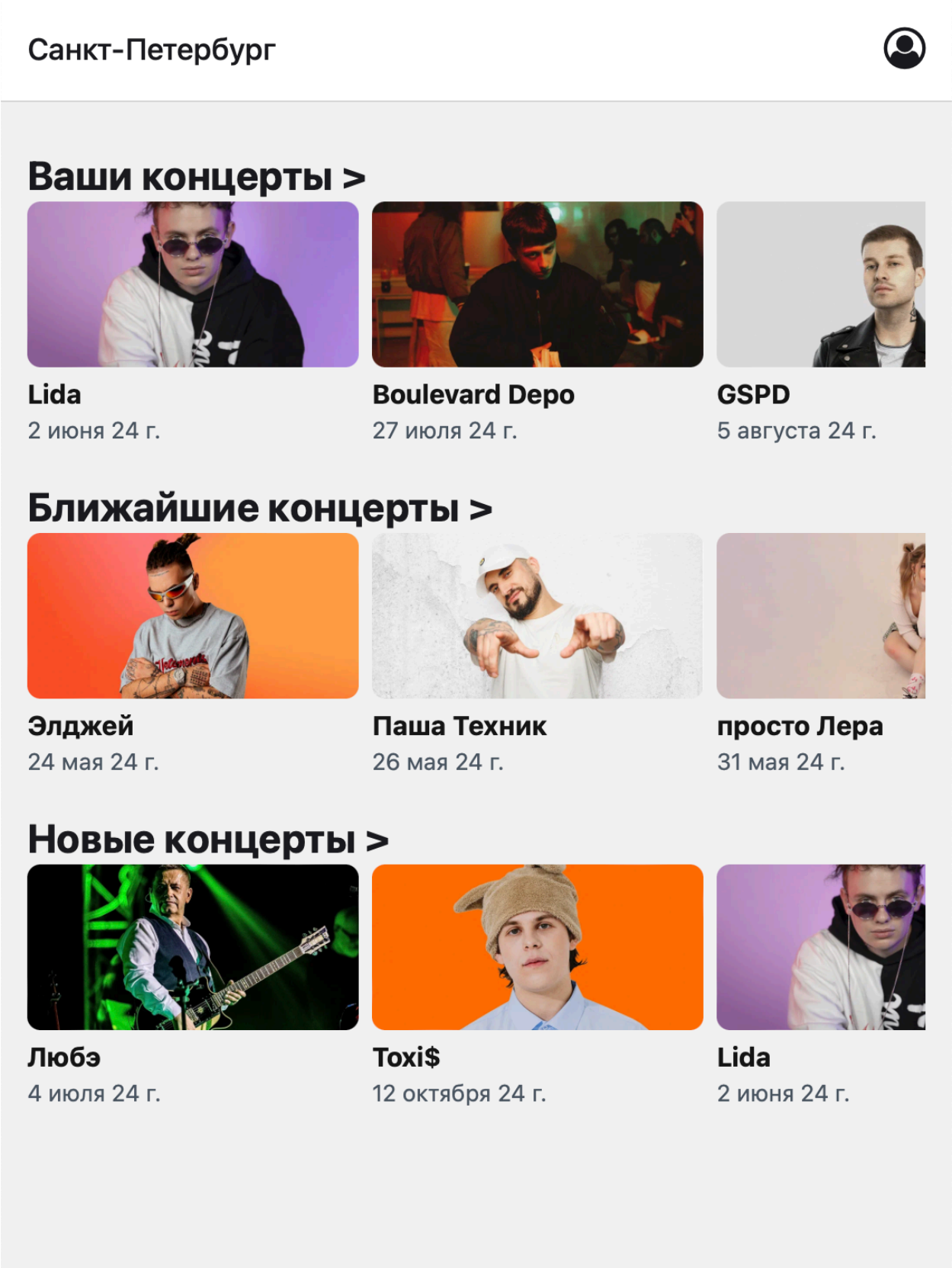

Рисунок 18 – Скриншот главной страницы приложения

На Рисунке 18 демонстрируется получившийся интерфейс главной страницы приложения.

4 РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТАЛЬНОЙ ПРОВЕРКИ ПРИЛОЖЕНИЯ

Для проверки достижения цели я собрал группу из 8 человек и предложил им воспользоваться моим приложением, чтобы выяснить, способно ли оно заинтересовать пользователя поучаствовать в каком-либо концерте его любимого исполнителя.

В ходе эксперимента каждый участник исследовал приложение в течение 15 минут, чтобы оценить его функциональность и удобство использования.

После регистрации в приложении и аутентификации через стриминговые сервисы приложение предложило информацию о предстоящих концертах любимых исполнителей, фильтруя их по выбранному городу и сортируя по дате проведения.

Количественные результаты:

1. Уровень интереса: 7 из 8 участников выразили интерес к посещению как минимум одного из предложенных концертов после использования приложения.
2. Обнаружение новых событий: 5 из 8 участников сообщили, что приложение помогло им узнать о предстоящем выступлении, о котором они ранее не знали.
3. Количество заинтересованных исполнителей: В среднем, каждый участник проявил интерес к концертам трех различных исполнителей.

Качественные результаты:

1. Пользовательский отзыв: Большинство участников отметили, что приложение имеет интуитивно понятный интерфейс, а также полезные функции, которые упрощают процесс поиска концертных событий.
2. Функциональность: Участники высоко оценили возможность фильтрации концертов по дате и местоположению, а также интеграцию с музыкальными стриминговыми сервисами.

ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы мною было разработано приложение, изначально заявленная цель которого была достигнута – приложение способно предоставлять пользователям персонализированный список концертов, что увеличивает их вовлеченность в концертные события.

Для достижения поставленной цели были решены следующие задачи:

1. Интеграция с афишами концертных событий – приложение автоматически собирает и агрегирует информацию о ближайших концертах.
2. Интеграция с музыкальными сервисами – приложение получает данные любимых исполнителей пользователей для предложения концертов.
3. Разработка унифицированного интерфейса для работы с данными из разных источников – была создана система, позволяющая работать с данными различного представления через единый интерфейс.
4. Разработка пользовательского интерфейса – приложение было успешно оснащено интуитивно понятным интерфейсом, который делает процесс поиска концертных событий максимально удобным.

Ожидаемые результаты были достигнуты, однако у приложения остаются потенциальные пути для улучшения. К примеру, добавление новых музыкальных сервисов и афиш, что расширит выбор для пользователей. Кроме того, можно разработать собственную рекомендательную систему, основанную на анализе предпочтений пользователей со схожими музыкальными вкусами. Такая система сможет предлагать мероприятия, опираясь на реальный опыт и выборы пользователей, что позволит приложению предоставлять более точные рекомендации.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Приложение Bandsintown [Электронный ресурс]. – Режим доступа. – https://bandsintown.com/, свободный (дата обращения 24.04.2024)
2. Приложение Songkick [Электронный ресурс]. – Режим доступа. – https://www.songkick.com/, свободный (дата обращения 24.04.2024)
3. Сервис продажи билетов Яндекс Афиша [Электронный ресурс]. – Режим доступа. – https://afisha.yandex.ru/, свободный (дата обращения 24.04.2024)
4. Сервис продажи билетов KASSIR.RU [Электронный ресурс]. – Режим доступа. – https://kassir.ru/, свободный (дата обращения 24.04.2024)
5. Стриминговый сервис Яндекс Музыка [Электронный ресурс]. – Режим доступа. – https://music.yandex.ru/, свободный (дата обращения 24.04.2024)
6. Стриминговый сервис VK Music [Электронный ресурс]. – Режим доступа. – https://music.vk.com/, свободный (дата обращения 24.04.2024)
7. Документация для разработки Spring Boot приложений [Электронный ресурс]. – Режим доступа. – https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/, свободный (дата обращения 24.04.2024)
8. Фреймворк Django [Электронный ресурс]. – Режим доступа. – https://djangoproject.com/, свободный (дата обращения 24.04.2024)
9. Фреймворк Express [Электронный ресурс]. – Режим доступа. – https://expressjs.com/, свободный (дата обращения 24.04.2024)
10. React Native [Электронный ресурс]. – Режим доступа. – https://reactnative.dev/, свободный (дата обращения 24.04.2024)
11. Flutter [Электронный ресурс]. – Режим доступа. – https://flutter.dev/, свободный (дата обращения 24.04.2024)
12. Документация PostgreSQL [Электронный ресурс]. – Режим доступа. – https://www.postgresql.org/docs/, свободный (дата обращения 24.04.2024)

13. MongoDB [Электронный ресурс]. – Режим доступа. – https://mongodb.com/, свободный (дата обращения 24.04.2024)
14. Документация Expo [Электронный ресурс]. – Режим доступа. – https://docs.expo.dev/, свободный (дата обращения 24.04.2024)
15. Документация NativeWind [Электронный ресурс]. – Режим доступа. – https://nativewind.dev/, свободный (дата обращения 24.04.2024)
16. Компоненты GluestackUI [Электронный ресурс]. – Режим доступа. – https://gluestack.io/, свободный (дата обращения 24.04.2024)
17. Документация Async Storage [Электронный ресурс]. – Режим доступа. – https://docs.expo.dev/versions/latest/sdk/async-storage/, свободный (дата обращения 24.04.2024)
18. Документация Lombok [Электронный ресурс]. – Режим доступа. – https://projectlombok.org/, свободный (дата обращения 24.04.2024)
19. Документация Jackson [Электронный ресурс]. – Режим доступа. – https://github.com/FasterXML/jackson-docs/, свободный (дата обращения 24.04.2024)
20. Документация MapStruct [Электронный ресурс]. – Режим доступа. – https://mapstruct.org/documentation/stable/reference/html/, свободный (дата обращения 24.04.2024)

ПРИЛОЖЕНИЕ А

```
@Override
public void parse() {
    MultiValueMap<String, String> params = new LinkedMultiValueMap<>();

    // Получаем данные о существующих городах
    AfishaCities cities = webClient.get()
            .uri(parserProperties.getCitiesUrl())
            .retrieve()
            .bodyToMono(AfishaCities.class)
            .block();

    // Парсим события каждого города
    for (AfishaCities.AfishaCity city : cities.getData()) {
        int offset = 0;

        // Устанавливаем параметры в запрос
        params.set("city", city.getId());
        params.set("hasMixed", String.valueOf(0));
        params.set("limit", String.valueOf(LIMIT));
        params.set("offset", String.valueOf(offset));

        AfishaEvents eventsData;
        do {
            // Отправляем запросы пачками по 20
            eventsData = webClient.get()
                    .uri(parserProperties.getEventsUrl(), uriBuilder -> uriBu
ilder.queryParams(params).build())
                    .retrieve()
                    .bodyToMono(AfishaEvents.class)
                    .block();

            // Преобразуем полученные данные в понятные приложению
            List<EventData> data = eventsData.getData().stream().map(parserMa
pper::map).toList();

            // Сохраняем данные через фильтрационный сервис
            eventFilterService.saveData(data);

            params.set("offset", String.valueOf(offset += LIMIT));
        } while (eventsData.hasNext());
    }
}
```

ПРИЛОЖЕНИЕ Б

```
public void parseUser(@NonNull User user) {
    String libraryUrl = parserProperties.getLibraryUrl().formatted(user.getYa
MusicCredentials());

    // Получаем библиотеку пользователя (его треки)
    YaMusicLibrary yaMusicLibrary = webClient.post()
            .uri(libraryUrl)
            .retrieve()
            .bodyToMono(YaMusicLibrary.class)
            .block();

    // Делим все треки пользователя на части
    List<List<String>> batches = Lists.partition(yaMusicLibrary.getTrackIds()
, BATCH_SIZE);

    for (List<String> batch : batches) {
        // Запрос необходимо выполнять пачками по 49 треков максимум
        List<YaMusicTrack> yaMusicTracks = webClient.post()
                .uri(parserProperties.getTracksUrl())
                .body(BodyInserters.fromFormData("MIME Type", "application/x-
www-form-urlencoded; charset=UTF-8")
                        .with("entries", String.join(",", batch))
                        .with("strict", "true")
                        .with("removeDuplicates", "true")
                        .with("lang", "ru")
                        .with("overembed", "false"))
                .retrieve()
                .bodyToMono(new ParameterizedTypeReference<List<YaMusicTrack>
>() {})
                .block();

        // Добавление каждого артиста в избранное
        for (YaMusicTrack yaMusicTrack : yaMusicTracks) {
            // Получение всех артистов из трека
            for (YaMusicTrack.Artist artist : yaMusicTrack.getArtists()) {
                var coverUri = transformCoverUri(artist.getCover());
                // Добавление конкретного артиста в избранное
                userFavoriteArtistsService.addFavoriteArtist(user, MusicServi
ce.YANDEX_MUSIC, artist.getName(), coverUri);
            }
        }
    }
}
```