

Федеральное государственное образовательное бюджетное учреждение
высшего образования
**«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ
РОССИЙСКОЙ ФЕДЕРАЦИИ»**

Факультет информационных технологий и анализа больших данных
Кафедра информационных технологий

Выпускная квалификационная работа
на тему: «Разработка мобильного приложения «Путешествия по России»»

Направление подготовки: 09.03.03 Прикладная информатика
Профиль: ИТ-сервисы и технологии обработки данных в экономике и
финансах

Выполнили студенты учебной группы
ПИ21-4

Иванов Дмитрий Михайлович 

Трушкова Мария Владимировна 

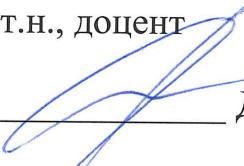
Дранникова Елена Михайловна 

Научный руководитель работы

к.т.н., доцент

Будаев Евгений Сергеевич 

**ВКР соответствует предъявляемым
требованиям:**

Заведующий Кафедрой информационных
технологий, к.т.н., доцент 

Д. А. Петров

« » 2025 г.

Москва 2025

Содержание

ВВЕДЕНИЕ	3
ГЛАВА 1 ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ	7
1.1 Исследование целевой аудитории	7
1.2 Приоритизация функциональности	18
1.3 Оценка рынка	22
1.4 Исследование конкурентной среды	26
1.5 Бизнес-модель стартапа	42
1.6 Экосистема монетизации	47
1.7 Финансовая модель	49
ГЛАВА 2 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ	56
2.1 Описание требований и ограничений	56
2.2 Архитектурное решение	63
2.3 Диаграммы проектирования	70
2.4 Компоненты инфраструктуры	97
2.5 Меры обеспечения безопасности	101
Выводы по главе	104
ГЛАВА 3 РЕАЛИЗАЦИЯ СЕРВИСНОЙ ЧАСТИ	106
3.1 Технологический стек	106
3.2 Реализация микросервисов	116
3.3 Организация DevOps-инфраструктуры и CI/CD	136
3.4 Тестирование микросервисной системы	140

Выводы по главе	143
ГЛАВА 4 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ И ВЫБОР ТЕХНОЛОГИЧЕСКОГО СТЕКА	145
4.1 Проектирование архитектуры мобильного приложения	147
4.2 Основные пути пользователя	150
4.3 Архитектура мобильного приложения	156
ГЛАВА 5 РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ	181
5.1 Базовая структура приложения	181
5.2 Реализация слоя данных (взаимодействия с бекеном)	183
ГЛАВА 6 ПРОЕКТИРОВАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА МОБИЛЬНОГО ПРИЛОЖЕНИЯ	190
6.1 Принципы и подходы к проектированию интерфейса	191
6.2. Разработка макетов интерфейса	195
6.3. Проектирование структуры экранов и пользовательских сценариев	196
6.4. Технические требования к интерфейсу	207
ЗАКЛЮЧЕНИЕ	211
ИСТОЧНИКИ	213

ВВЕДЕНИЕ

В последние десятилетия туризм претерпел значительную трансформацию и стал неотъемлемым компонентом глобальных экономических процессов, активно способствуя не только экономическому развитию, но и обеспечению устойчивого социального и культурного взаимодействия между различными регионами и странами мира. В современной экономике туризм играет существенную роль в формировании ВВП, создании рабочих мест, стимулировании малого и среднего предпринимательства, а также в повышении уровня жизни населения и решении социально значимых задач, таких как преодоление социального неравенства и интеграция различных слоев общества.

Актуальность исследования внутреннего туризма обусловлена не только его экономическими, но и социокультурными аспектами, которые активно обсуждаются в научных кругах. Туризм как форма социокультурной коммуникации позволяет странам и регионам обмениваться опытом, традициями, инновациями и знаниями, что способствует укреплению взаимопонимания и стабильности на национальном и международном уровнях. В условиях глобализации туристические потоки выступают в роли своеобразных культурных мостов, способствующих формированию толерантности и взаимоуважения среди различных сообществ. Внутренний туризм в Российской Федерации в последние годы особенно актуализировался под воздействием глобальных экономических и социальных изменений, таких как пандемия COVID-19 и международные санкции. Если ранее внутренний туризм воспринимался преимущественно как второстепенная сфера по отношению к международному туризму, то сегодня он занимает стратег-

гическое место в экономике и активно поддерживается государством через различные программы и инициативы. Пандемия COVID-19, ограничившая международные путешествия, стала катализатором переориентации туристических потоков на внутренние направления. Это привело к возникновению новых туристических маршрутов, развитию местных достопримечательностей и инфраструктуры, а также формированию новой модели поведения путешественников, стремящихся получать качественные услуги внутри страны.

По данным Федеральной службы государственной статистики (Росстат), вклад туризма в ВВП России, несмотря на значительный спад в 2020 году, обусловленный пандемией, продемонстрировал устойчивую тенденцию к восстановлению и росту. Так, если в 2018 году туристическая индустрия составляла 2,7% от ВВП, то к 2023 году показатель восстановился до 2,8%. Важную роль в этом процессе сыграли не только государственные меры поддержки, но и инициативы на региональном уровне, направленные на повышение туристической привлекательности территорий, развитие инфраструктуры и продвижение локальных туристических продуктов.

Рост спроса на внутренний туризм подтверждается также увеличением общего объёма услуг, предоставляемых туристическими агентствами, туроператорами и другими профильными организациями. Согласно данным Росстата, за последние пять лет этот показатель вырос почти на 66%, достигнув 285,9 млрд рублей в 2023 году по сравнению с 172,1 млрд рублей в 2018 году. Такая динамика свидетельствует о высокой востребованности качественных туристических продуктов, что усиливает конкуренцию на рынке и подталкивает компании к внедрению инновационных решений и повышению профессионального уровня сотрудников.

Одним из наиболее заметных трендов в современном туризме становится растущая потребность путешественников в индивидуализации и персонализации туристического опыта [1]. Современные туристы предпочитают уникальные впечатления, ориентированные на их персональные предпочтения, физические возможности и социальные особенности. В связи с этим возрастаёт важность внедрения инновационных технологий, таких как мобильные приложения, которые позволяют легко планировать путешествия, учитывать индивидуальные потребности пользователей и обеспечивать высокое качество сервиса на всех этапах туристического опыта. Именно на удовлетворение таких индивидуализированных потребностей ориентировано разрабатываемое в рамках дипломного проекта мобильное приложение «Путешествия по России». Данное приложение предназначено для широкого спектра целевых аудиторий, включая маломобильных граждан, семьи с детьми, пожилых людей, студентов и других категорий путешественников, и предлагает им персонализированный подход к организации и планированию поездок. Приложение будет способствовать формированию активного туристического сообщества, члены которого смогут обмениваться опытом, рекомендациями и создавать уникальный контент, повышая привлекательность и узнаваемость местных туристических направлений.

Целью настоящей работы является всестороннее обоснование и практическая реализация цифровой платформы «Путешествия по России», способной предложить персонализированный туристический опыт для широкого круга внутренних путешественников и тем самым стимулировать социально-экономическое развитие регионов страны. Для достижения поставленной цели сформулированы четыре взаимосвязанные задачи, отражающие рас-

пределение ролей в проектной команде:

- 1. Бизнес-аналитика и стратегия монетизации** – исследовать рынок travel-tech, определить целевые сегменты, построить финансовую модель и разработать долгосрочную стратегию доходов на основе партнёрских комиссий и нативной рекламы.
- 2. Продуктовая концепция и UX/UI-дизайн** – сформировать пользовательские сценарии, разработать инклюзивный интерфейс и визуальную айдентику, обеспечивающие высокую вовлечённость и удобство взаимодействия.
- 3. Кроссплатформенная мобильная разработка** – создать полноценное приложение на Flutter для Android, iOS и RuStore, реализовав модули авторизации, построения маршрутов, офлайн-карт и социального обмена контентом.
- 4. Разработка масштабируемого серверного ядра** – спроектировать микросервисную архитектуру на Go, внедрить безопасные механизмы аутентификации и горизонтального масштабирования, настроить CI/CD-конвейер и систему мониторинга для бесперебойной работы платформы.

Комплексное выполнение перечисленных задач позволит вывести на рынок технологически совершенный и коммерчески жизнеспособный продукт, отвечающий современным запросам российских путешественников и способный к дальнейшему масштабированию.

ГЛАВА 1 ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ

1.1 Исследование целевой аудитории

Для максимально эффективной реализации и соответствия разрабатываемого продукта запросам целевой аудитории был проведен комплексный анализ потребностей потенциальных пользователей мобильных туристических приложений. Данный этап исследования включал в себя количественный онлайн-опрос и качественные персональные интервью, направленные на глубокое понимание ожиданий, предпочтений и требований потенциальных пользователей.

Первый этап исследования был осуществлён посредством онлайн-опро

-са на платформе Яндекс.Формы, который охватил 120 участников, представляющих различные возрастные и социальные группы: студенты, сотрудники университетов, коллеги-разработчики и родственники авторов проекта. Такая диверсификация выборки позволила получить репрезентативные данные, учитывающие разнообразие взглядов и потребностей различных социальных групп, и обеспечила надёжность и валидность полученных результатов.

На основе собранных данных были выделены ключевые потребности и ожидания от мобильных туристических приложений:

- Возможность выбора и создания маршрутов с учётом различных сложностей и ограничений (отметили более 85% респондентов);
- Подробная навигация и удобство использования приложения (93%)

опрошенных);

- Доступ к информации о малоизвестных и оригинальных точках по России от других путешественников (74% респондентов);
- Возможность быстрого обмена маршрутами и опытом между путешественниками непосредственно в приложении (67% респондентов);
- Индивидуальное планирование маршрутов по протяженности и продолжительности путешествий (81% участников опроса).

Второй этап исследования представлял собой качественные персональные интервью с 10 респондентами, охватывающими различные возрастные и социальные группы. Цель интервью состояла в уточнении и дополнении количественных данных, полученных в ходе онлайн-опроса. В процессе бесед респонденты выразили пожелания относительно создания удобного и интуитивно понятного интерфейса приложения, что позволило бы им быстро адаптировать маршруты под собственные нужды. Участники интервью подчеркнули также значимость эмоционального удовлетворения от путешествий, возможность обмена впечатлениями с другими пользователями и интерес к открытию новых, малоизвестных локаций.

Также анализируя собранные данные, команда проекта сегментировала целевую аудиторию – пользователей сервиса – на несколько групп, объединенных сходными характеристиками и потребностями. Сегментация важна для более точной настройки продукта под разные категории клиентов. Ниже приведены основные сегменты и их особенности:

- Семьи с детьми ценят безопасность и комфорт. Предпочитают маршруты умеренной сложности, с наличием инфраструктуры (туалеты, ме-

ста для отдыха, детские развлечения). Им важно заранее знать, подходит ли маршрут для детей, есть ли детские комнаты в музеях и пр. Также семьи заинтересованы в особых местах, интересных для разных поколений, и в сокращении времени на логистику, чтобы дети не утомлялись.

- Люди с ограниченной подвижностью или инвалиды, а также пожилые с проблемами здоровья. Для них критически важна доступность: наличие пандусов, лифтов, отсутствие длинных пеших переходов по пересечённой местности. Они нуждаются в маршрутах небольшой протяженности, с подробной информацией о рельефе, покрытиях дорог, наличии скамеек для отдыха. Сегмент требует тщательной фильтрации объектов по параметру доступности и ищет сервис, который избавит от необходимости самостоятельно уточнять эти условия.
- Путешественники пенсионного и предпенсионного возраста. Обычно они предпочитают более спокойный темп маршрута и умеренную нагрузку. Ценят подробное описание и историческую информацию о местах (культурно-познавательный аспект), а также простоту интерфейса приложения. Им важно избегать чрезмерно сложных для физической формы активностей (например, долгих горных походов). Этот сегмент также может нуждаться в увеличенном шрифте или голосовых подсказках в навигации, учитывая возможные возрастные ограничения зрения/слуха.
- В группу активных туристов входят в основном молодые или опытные путешественники, нацеленные на активный отдых и новые впечатления. Они готовы к маршрутам высокой сложности: длительным

пешим прогулкам, походам, нестандартным видам активности (например, треккинг, велосипедные туры). Для них ключевым является наличие уникальных мест в маршруте – они стремятся уйти от массовых туристических троп и исследовать что-то новое. Активные туристы также более склонны сами делиться своими маршрутами и отзывами, выступая генераторами контента сообщества. Технически подкованы, поэтому с энтузиазмом используют продвинутые функции приложения (навигацию, социальные фичи).

Результаты проведённых исследований подчёркивают высокую востребованность и значимость ключевых функций разрабатываемого приложения. В частности, выделены такие приоритетные функции, как персонализированный подбор маршрутов, фильтрация по уровню сложности, доступности инфраструктуры и временным рамкам путешествия, а также обеспечение активного социального обмена опытом и маршрутами между пользователями. Таким образом, реализация этих возможностей в мобильном приложении «Путешествия по России» позволит не только эффективно удовлетворить выявленные потребности целевой аудитории, но и обеспечить конкурентные преимущества на рынке туристических приложений.

User Story № 1

Как молодой человек в возрасте 20–25 лет, начинающий изучать туристический потенциал своей страны, я стремлюсь получить детализированную и структурированную информацию о достопримечательностях, как широко известных, так и менее популярных маршрутах. Мне необходимо, чтобы представленные сведения сопровождались высококачественны-

ми фотографиями и интерактивными картами, что позволит формировать чёткое представление о будущих путешествиях, минимизировать неопределённость и обеспечивать максимально комфортное планирование поездок.

Критерии приемки:

- Доступ к карточке маршрута, содержащей развернутое текстовое описание ключевых объектов;
- Визуализация маршрутов посредством качественных фотографий, позволяющих оценить эстетику и особенности местности;
- Интерактивная карта с точной геолокацией точек интереса, представляющая возможность предварительного анализа маршрута и его логистической доступности.

User Story № 2

Как родитель, путешествующий с детьми младшего возраста (до 10 лет), я заинтересован в функционале, который позволит мне находить и планировать маршруты, адаптированные под особенности семейного отдыха. Для меня критически важно, чтобы система обеспечивала фильтрацию маршрутов по уровню сложности и наличию соответствующей инфраструктуры, включая зоны отдыха, объекты общественного питания, санитарные узлы и детские площадки. Данный функционал позволит минимизировать потенциальные неудобства в поездке, повысить уровень комфорта передвижения и способствовать формированию благоприятного эмоционального опыта путешествия для всей семьи.

Критерии приемки:

- Возможность выбора уровня сложности маршрута (например, лёгкий, средний, повышенной сложности);
- Наличие фильтрации маршрутов с учетом семейной инфраструктуры и ключевых удобств;
- Детализированное описание маршрутов, включающее информацию о наличии зон отдыха, пунктов питания и объектов, ориентированных на потребности семей с детьми.

User Story № 3

Как пользователь с ограниченными физическими возможностями (например, человек, передвигающийся на инвалидной коляске), я нуждаюсь в доступной и детализированной информации о маршрутах с адаптированной инфраструктурой. Для меня критически важно заранее получать сведения о наличии пандусов, ровных дорожек, отсутствии крутых спусков и подъёмов, а также возможности безопасного передвижения. Дополнительно необходимо учитывать возможность задания максимальной дистанции маршрута, что позволит мне эффективно планировать поездку с учётом физических возможностей и комфортного передвижения.

Критерии приемки:

- Наличие специализированного фильтра для поиска маршрутов, адаптированных для маломобильных граждан;
- Возможность установки предельной дистанции маршрута в километрах в соответствии с индивидуальными потребностями пользователя;

- Подробное описание доступности инфраструктуры, включая информацию о наличии пандусов, ширине дорожек, доступности общественного транспорта и наличии санитарных зон, адаптированных для маломобильных граждан.

User Story № 4

Как опытный турист и тревел-блогер, занимающийся изучением малоизвестных регионов России, я стремлюсь к созданию индивидуальных туристических маршрутов с возможностью их последующего распространения среди сообщества единомышленников. Для меня важно иметь инструменты, позволяющие формировать маршруты с детализированными текстовыми описаниями, качественным визуальным сопровождением в виде фотографий и точной географической привязкой к картографическим данным. В дополнение к этому мне необходимо взаимодействовать с другими пользователями платформы, получать обратную связь в формате оценок и комментариев, а также привлекать внимание к уникальным местам, представляющим культурную, историческую и природную ценность.

Критерии приемки:

- Разработка интуитивно понятного и функционального инструмента для создания пользовательских маршрутов;
- Возможность загрузки высококачественных фотографий и детализированных описаний маршрутов и достопримечательностей;
- Интеграция механизмов взаимодействия пользователей через систему комментариев, рейтингов и оценок, обеспечивающая активное вовлечение сообщества в процесс обмена туристическим опытом.

User Story № 5

Как молодой пользователь социальных сетей (возраст от 18 до 30 лет), я стремлюсь к активному взаимодействию с туристическим сообществом и обмену опытом в цифровом формате. Для меня важно иметь доступ к отзывам и оценкам других пользователей, чтобы принимать информированные решения при выборе маршрутов. Я также хочу иметь возможность оставлять собственные комментарии, делиться впечатлениями, присваивать рейтинги маршрутам и быстро находить самые популярные маршруты, основанные на пользовательских рекомендациях. Этот функционал позволит не только улучшить качество туристического опыта, но и создать динамичное цифровое сообщество путешественников.

Критерии приемки:

- Возможность оставлять комментарии и оценки маршрутов, обеспечивая двустороннюю коммуникацию между пользователями;
- Доступ к отзывам и оценкам других путешественников, что позволит принимать более осознанные решения при выборе маршрутов;
- Быстрый просмотр наиболее популярных маршрутов, ранжированных на основе пользовательских оценок и отзывов.

Данный анализ позволил выявить ключевые тренды и определить наиболее перспективные направления для разработки мобильного приложения туристической тематики.

User Story № 6

Как пользователь, работающий полный рабочий день и имеющий ограниченное количество свободного времени для планирования путешествий, я нуждаюсь в функционале, который позволит мне оперативно получать уведомления о появлении новых маршрутов и туристических мероприятий, доступных в выходные дни и расположенных в непосредственной близости от моего местоположения. Это позволит мне быстро принимать решения о коротких поездках на 1–2 дня, избегая длительных поисков информации и повышая эффективность планирования отдыха.

Критерии приемки:

- Интеллектуальная система уведомлений, анализирующая предпочтения пользователя и его геолокацию для предоставления актуальных предложений;
- Автоматическое отображение дат проведения мероприятий и туристических маршрутов, доступных в ближайшие выходные;
- Возможность персонализации уведомлений в настройках профиля, позволяя пользователю выбирать их частоту, тематику и метод получения.

User Story № 7

Как пожилой пользователь (возраст 60 лет и старше), я заинтересован в получении персонализированных рекомендаций по маршрутам, которые соответствуют моим культурным интересам (например, экскурсионные программы, посещение исторических и природных достопримечательностей).

тельностей) и физическим возможностям (маршруты с минимальной сложностью, адаптированные для комфорtnого передвижения). Мне важно избегать чрезмерных физических нагрузок, чтобы сосредоточиться на получении положительных впечатлений от путешествий без дополнительных затруднений.

Критерии приемки:

- Автоматическая система персональных рекомендаций, основанная на истории путешествий пользователя и его предпочтениях;
- Возможность фильтрации маршрутов по уровню сложности и общей протяжённости, обеспечивающая комфортные условия передвижения;
- Доступ к рекомендованным маршрутам в отдельном разделе приложения, упрощающий навигацию и планирование поездок.

User Story № 8

Как студент, часто путешествующий в компании друзей-рөвесников, я стремлюсь к удобной системе хранения и группировки выбранных маршрутов в своём профиле приложения. Это позволит мне быстро возвращаться к планированию путешествий, сохранять наиболее интересные маршруты и эффективно делиться ими с друзьями. Такой функционал обеспечит гибкость в организации поездок и облегчит совместное обсуждение туристических планов.

Критерии приемки:

- Возможность добавления маршрутов в избранное для быстрого доступа;
- Функция группировки маршрутов в тематические коллекции для систематизации информации;
- Инструменты удобного обмена избранными маршрутами с друзьями, способствующие коллективному планированию путешествий.

User Story № 9

Как администратор/менеджер туристического сообщества или организатор мероприятий, я нуждаюсь в доступе к аналитическим данным, отражающим популярность туристических маршрутов, отзывы пользователей, количество просмотров и посещений. Данный функционал позволит мне выявлять наиболее востребованные направления, анализировать предпочтения аудитории и на основе этих данных принимать обоснованные решения по разработке новых туристических программ и событий, ориентированных на актуальные интересы пользователей.

Критерии приемки:

- Генерация наглядных отчётов и статистики по популярности маршрутов;
- Доступ к данным об отзывах, рейтингах и оценках маршрутов;
- Возможность анализа количества просмотров, посещений и пользовательских предпочтений в удобной веб-панели администратора, обеспечивающей эффективное управление туристическим контентом.

User Story № 10

Как пользователь, совершающий путешествия в регионы с нестабильным подключением к интернету (например, малонаселённые или труднодоступные территории России), я нуждаюсь в функционале оффлайн-доступа к маршрутам и картам. Для меня критически важно иметь возможность заранее загружать необходимую информацию, включая детализированные описания маршрутов и навигационные данные, чтобы не зависеть от мобильного интернета во время поездки. Это обеспечит уверенность в ориентировании на местности и позволит избежать возможных сложностей, связанных с отсутствием связи. Критерии приемки:

- Функция предварительной загрузки маршрутов и карт для автономного использования;
- Беспрепятственный доступ к контенту приложения в оффлайн-режиме без потери функциональности;
- Интуитивно понятный интерфейс для управления загрузкой и доступом к сохранённому оффлайн-контенту.

1.2 Приоритизация функциональности

В результате детализированного анализа пользовательских историй команда разработчиков сформировала многоуровневую систему приоритизации функциональности приложения. Цель данного процесса заключается в оптимальном распределении ресурсов разработки, последовательном внедрении наиболее значимых функций и формировании стратегического видения эволюции продукта. Такой подход обеспечивает долг-

срочную конкурентоспособность, соответствие запросам различных групп пользователей и адаптацию к изменяющимся требованиям туристического рынка.

Высокий приоритет (MVP – ключевой функционал первой версии)

Функциональные возможности, относящиеся к данной категории, формируют базовую основу мобильного приложения. Их наличие является критически важным для первичного релиза, так как именно они обеспечивают удовлетворение базовых потребностей пользователей и определяют ценность продукта.

Основные элементы MVP:

- Каталог маршрутов с подробными описаниями, визуальными материалами и картографической интеграцией. Этот элемент представляет собой ключевой механизм взаимодействия пользователей с сервисом, предоставляя исчерпывающую информацию о доступных маршрутах и способствуя информированному выбору туристических направлений.
- Гибкая система фильтрации маршрутов по уровню сложности, протяжённости и наличию инфраструктуры. Этот функционал ориентирован на удовлетворение потребностей широкого круга пользователей, включая семьи с детьми, пожилых путешественников и маломобильных граждан, для которых вопросы доступности играют решающую роль.
- Создание и публикация пользовательских маршрутов с интерактивными элементами. Данный функционал направлен на стимулирование

ние пользовательского контента, вовлечение тревел-блогеров и опытных туристов, предоставляя им платформу для обмена знаниями и популяризации новых направлений.

User Stories: № 1 (каталог маршрутов), № 2 и № 3 (фильтрация маршрутов), № 4 (создание пользовательского контента).

Средний приоритет (улучшение пользовательского опыта после запуска MVP)

Функции данного уровня приоритета направлены на повышение удобства взаимодействия с приложением, увеличение вовлечённости аудитории и укрепление пользовательской лояльности. Функции среднего приоритета:

- Механизмы социальной вовлечённости. Возможность оставлять комментарии, выставлять рейтинги и просматривать отзывы способствует формированию сообщества пользователей и усилению доверия к информации в приложении.
- Система персонализированных уведомлений. Этот функционал позволяет пользователям оперативно получать актуальные сведения о новых маршрутах, туристических мероприятиях и событиях на основе их предпочтений и геолокации.
- Сохранение избранных маршрутов и функция их обмена. Данная возможность особенно востребована среди пользователей, путешествующих в группах, а также среди тех, кто планирует поездки заранее и хочет иметь быстрый доступ к сохранённым маршрутам.

User Stories: № 5 (социальные взаимодействия), № 6 (уведомления), № 8 (избранные маршруты).

Низкий приоритет (перспективные функции для долгосрочного развития)

Функциональность этой категории включает элементы, которые могут быть внедрены на более поздних стадиях развития продукта в зависимости от пользовательского спроса и общей стратегии масштабирования.

Ключевые направления:

- Персонализированные рекомендации маршрутов, основанные на анализе пользовательского поведения и истории поездок.
- Оффлайн-доступ к маршрутам и картам для автономного использования в зонах с низким уровнем интернет-покрытия.
- Расширенные инструменты аналитики для организаторов туристических мероприятий и владельцев бизнеса.
- Продвинутая социальная интеграция, включая расширенные профили, возможность добавления друзей и интеграцию с внешними туристическими сервисами.

User Stories: № 7 (персонализация маршрутов), № 9 (аналитика), № 10 (оффлайн-доступ).

Разработанная стратегия приоритизации функционала позволяет команде сосредоточиться на реализации наиболее значимых для пользователей возможностей, обеспечивающих успешный запуск продукта. Средне- и низкоприоритетные функции планируются к внедрению по мере роста

пользовательской базы и накопления аналитических данных. Такой подход способствует устойчивому развитию приложения, повышению его востребованности и закреплению позиций на рынке цифровых туристических решений.

1.3 Оценка рынка

В последние годы внутренний туризм в России демонстрирует значительный рост. В 2024 году количество туристических поездок по России составило около 92 миллионов, что является рекордным показателем за всю историю внутреннего туризма. По данным Российского союза туриндустрии, количество поездок по стране в 2024 году оценивается примерно в 96 миллионов[8], что на четверть больше, чем в 2023 году. Аналогичные данные приводит другой источник, указывая на 96 миллионов поездок в 2024 году, что на 25% превышает показатель в 78 миллионов поездок в 2023 году. Согласно исследованию "СберАналитики", внутренний турпоток в России в 2023 году составил 152 миллиона человек, что является максимальным значением за последние пять лет. Другие данные указывают, что россияне совершили 160 миллионов поездок в 2023 году, и прогнозируется увеличение до 170 миллионов поездок в 2024 году. По данным другого источника, количество внутренних туристских перемещений достигло 108 миллионов в 2023 году, показав годовой прирост на уровне 11%. Предварительные данные Российского союза туриндустрии (РСТ) также говорят о росте внутреннего турпотока на 20% до 78 миллионов поездок в 2023 году. Таким образом, наблюдается устойчивая тенденция к увеличению количества внутренних туристических поездок в России. Разница в абсолютных

значениях, вероятно, связана с различиями в методиках подсчета, включая определение понятия "турист" и "поездка" (например, учет краткосрочных поездок, посещений вторых домов и т.д.). Тем не менее, все источники сходятся во мнении о значительном росте рынка.

Помимо количества поездок, важным показателем является объем рынка в денежном выражении. В 2023 году туристическая отрасль России оказала услуг более чем на 4,3 триллиона рублей, что в 1,5 раза больше, чем в допандемийном 2019 году (2,8 триллиона рублей), причем основной прирост обеспечил внутренний туризм. Оборот внутреннего туризма за лето 2024 года вырос на 24% и составил 917 миллиардов рублей[9], при этом прогнозируется, что годовой оборот может достичь 2-2,2 триллиона рублей. За первые девять месяцев 2023 года выручка туристического бизнеса в стране увеличилась примерно до 8,5 миллиардов рублей, что также в 1,5 раза больше показателей 2019 года. По данным Росстата, в 2023 году туристическая отрасль произвела товаров и услуг на 4304,7 миллиарда рублей по сравнению с 2789,9 миллиарда рублей в 2019 году. Доходы коллективных средств размещения за 11 месяцев 2024 года превысили 1 триллион рублей, что на 30% больше, чем за аналогичный период 2023 года (256 миллиардов рублей). Эти данные свидетельствуют о значительном объеме рынка внутреннего туризма в России и его динамичном росте в денежном выражении. Увеличение расходов туристов говорит о готовности потребителей тратить на путешествия внутри страны.

Вклад туризма в экономику России также является важным показателем. В 2023 году доля валовой добавленной стоимости туристской индустрии в валовом внутреннем продукте Российской Федерации составила 2,8%, вернувшись к уровню 2019 года. По другим данным, вклад туризма в

экономику РФ составляет 3,47% ВВП, или 3 триллиона рублей. Президент Российской Федерации поставил задачу увеличить долю туризма в ВВП до 5%. Эти данные подчеркивают экономическую значимость туристического сектора для страны и потенциал для его дальнейшего развития и увеличения вклада в экономику.

По оценкам, около 3,3 миллиона человек с инвалидностью путешествуют в России. Общее количество людей с инвалидностью в России на начало 2023 года составляло 10,9 миллиона человек, или 7,5% от общей численности населения. Возможность фильтрации маршрутов по сложности и протяженности делает приложение привлекательным для этого значительного и часто недостаточно обслуживаемого сегмента рынка. Семьи с детьми составляют значительную долю туристического рынка. В летний сезон 2023 года на них приходилось 39% всех продаж туроператоров, а зимой 2023–2024 годов - 26% [11]. По другим данным, около 30% туристов путешествуют с детьми, а в 2024 году эта доля превысила 36%. Учитывая, что летний сезон является пиковым для туроператоров и составляет 60–75% годовых продаж, семьи с детьми являются ключевым целевым сегментом.

Функциональность приложения, позволяющая выбирать маршруты с учетом потребностей детей (продолжительность, сложность), делает его ценным инструментом для этой аудитории. Доля населения старше 55 лет в России достигла рекордных 30%. Наблюдается растущая тенденция путешествий среди пенсионеров внутри страны, где в первой половине 2018 года 71% их авиаперелетов приходилось на внутренние направления. В 2018 году 19% пенсионеров в возрасте 60–75 лет выражали желание совершить туристическую поездку по России или за границу, и доля путешествующих внутри страны растет. В 2023 году 63% респондентов старше 55

лет совершили поездки. Возможность выбора маршрутов с учетом физических возможностей делает приложение привлекательным для пожилых путешественников. Почти каждый третий житель России (27%) интересуется активным отдыхом. Наблюдается рост предпочтений к активному отдыху с 23% до 29%, особенно среди мужчин (39%) и молодых людей до 34 лет (47%). Приложение, позволяющее планировать пешие, велосипедные и другие активные маршруты, может привлечь эту значительную часть рынка.

Рынок внутреннего туризма в России демонстрирует устойчивые темпы роста в последние годы. Количество поездок выросло на 10–25% в 2023 и 2024 годах [10], а доходы также показали значительный рост. Позитивная динамика наблюдается как в количестве поездок, так и в доходах, что свидетельствует о здоровом и расширяющемся рынке, создавая благоприятные условия для приложения "Путешествия по России".

Правительственные инициативы и инвестиции, направленные на развитие туристической инфраструктуры, а также прогнозируемый рост внутреннего туризма на 2025 год, указывают на то, что эта положительная тенденция, вероятно, сохранится. Цель президента по значительному увеличению вклада туризма в ВВП и количества туристических поездок еще больше укрепляет этот прогноз. Сочетание сильного исторического роста и позитивных будущих прогнозов, обусловленных как рыночным спросом, так и государственной поддержкой, указывает на высокий темп роста рынка в обозримом будущем, что делает запуск соответствующего туристического приложения своевременным. Итого, закладывая целью в течение 3 трёх лет достичь 3% охвата целевого рынка, то мы получаем аудиторию в 500 тыс. пользователей.

Итого, закладывая целью в течение 3 трёх лет достичь 3% охвата центрального рынка, то мы получаем аудиторию в 500 тыс. пользователей.

Показатель	2023 (оценка)	2024 (оценка)	Рост (YoY)
Количество внутренних туристических поездок	78-160 млн	92-170 млн	11-25%
Оборот внутреннего туризма (лето)	740 млрд руб.	917 млрд руб.	24%
Общий оборот туристической отрасли	4,3 трлн руб.	2-2,2 трлн руб. (прогноз)	10%
Доходы средств размещения (11 мес.)	256 млрд руб.	1 трлн руб.	30%

Таблица 1. Показатели туристической отрасли

1.4 Исследование конкурентной среды

Исследование конкурентной среды является важным этапом подготовки проекта, поскольку позволяет выявить ключевые тенденции и лучшие практики, уже реализованные на рынке мобильных туристических приложений. Анализ существующих решений предоставляет возможность определить их сильные и слабые стороны, а также обозначить ниши и перспективы, которые могут быть эффективно заняты новым приложением.

Анализ конкурентного сервиса Яндекс.Карты

Яндекс.Карты представляют собой ведущий российский сервис картографических и навигационных услуг, разработанный и активно поддерживаемый компанией «Яндекс». С момента запуска данный сервис завоевал широкую популярность среди пользователей не только в Российской Федерации, но и в странах СНГ, благодаря высокой точности предоставляемых данных, удобству использования и продуманной интеграции с другими цифровыми продуктами компании.

Преимущества Яндекс.Карт:

1. Высокая детализация и точность картографических данных;
2. Развитая многофункциональность и интеграция с общественным транспортом;
3. Постоянное обновление и актуализация информации;
4. Активное участие пользователей в создании контента.

Недостатки Яндекс.Карт:

1. Ограниченные возможности персонализации туристических маршрутов;
2. Слабо развитые социальные функции;
3. Ограниченнное покрытие малоизвестных и удаленных туристических направлений;
4. Недостаточные инструменты расширенной фильтрации и поиска.

Несмотря на выявленные ограничения, Яндекс.Карты остаются одним из ведущих навигационных решений в России и странах СНГ, эффективно удовлетворяя основные потребности пользователей в навигации и планировании стандартных туристических поездок. Вместе с тем, существующие пробелы в функционале по персонализации, социальному взаимодействию и охвату локальных нишевых направлений открывают перспективы для нового продукта – приложения «Путешествия по России». Данний продукт может дополнить существующие возможности Яндекс.Карт, предоставив путешественникам продвинутые инструменты для индивидуального и специализированного планирования маршрутов, а также улучшить покрытие и качество информации о менее популярных и удалённых туристических объектах России.

Анализ конкурентного сервиса TripAdvisor

TripAdvisor представляет собой одну из наиболее авторитетных и востребованных платформ в сфере международного туризма, специализирующуюся на обмене опытом, рекомендациями и объективными оценками пользователей. Основанная в 2000 году, компания заняла лидирующие позиции на глобальном туристическом рынке, сформировав крупнейшую в мире базу пользовательского контента, посвящённого гостиницам, ресторанам, достопримечательностям и другим туристическим объектам.

Преимущества TripAdvisor:

1. Обширная и постоянно пополняемая база данных. Платформа акумулирует миллионы отзывов, комментариев и фотографий, предоставленных путешественниками со всего мира, что позволяет пользо-

вательям быстро ориентироваться в особенностях и качестве предоставляемых услуг и объектов туристической инфраструктуры.

2. Международное присутствие и широкая географическая доступность. TripAdvisor доступен на множестве языков и охватывает практически все туристически значимые регионы и страны мира. Это обеспечивает пользователям доступ к разнообразной и разносторонней информации, необходимой как для внутреннего, так и международного туризма.
3. Высокая степень вовлечённости аудитории. Благодаря активной пользовательской базе, регулярно публикующей отзывы и мультимедийные материалы, платформа отличается высокой актуальностью контента и оперативностью обновления информации, что существенно облегчает принятие туристических решений.
4. Комплексная интеграция с сервисами бронирования. Платформа обеспечивает возможность прямого бронирования гостиниц, ресторанов, экскурсий и мероприятий, упрощая тем самым процесс планирования и организации путешествий, увеличивая пользовательский комфорт и удобство.

Недостатки TripAdvisor:

1. Неоднородность качества и низкая модерация контента. В связи с большим объемом генерируемой пользователями информации возникают проблемы в обеспечении её достоверности, объективности и структурированности. Часто пользователи сталкиваются с субъективными или эмоционально окрашенными отзывами, которые могут

создавать искажённое восприятие действительности.

2. Недостаточные возможности персонализации маршрутов. Несмотря на обширность информационной базы, TripAdvisor ограничен в инструментах, позволяющих формировать и адаптировать маршруты с учетом специфических потребностей отдельных групп пользователей, таких как маломобильные граждане, семьи с детьми, пожилые туристы или лица с ограниченными возможностями.
3. Ограниченност контента о локальных и малоизвестных направлениях. Особенно выражено это в отношении регионов, не обладающих широкой популярностью или удаленных от традиционных туристических центров. Это приводит к недостаточной информированности путешественников и сложности планирования поездок в подобные места.
4. Ограниченные возможности фильтрации по специализированным критериям. TripAdvisor предлагает недостаточно гибкие инструменты фильтрации и поиска туристических объектов и услуг по специализированным параметрам, таким как уровень доступности для лиц с ограниченными возможностями, наличие семейной инфраструктуры или образовательный потенциал места.
5. Избыточная коммерциализация и наличие рекламного контента. Активная коммерческая политика платформы, связанная с продвижением платных услуг и объектов, зачастую приводит к предвзятой расстановке акцентов и искажению результатов поиска и ранжирования, что может создать неудобства пользователям и снижать объектив-

ность оценки.

Заключение и перспективы:

TripAdvisor является важным игроком на мировом туристическом рынке, предоставляя обширный и востребованный функционал, который охватывает основные потребности путешественников. Вместе с тем, выявленные ограничения, такие как недостаток персонализированного подхода и слабое покрытие локальных туристических направлений, создают значительные перспективы для разработки и продвижения специализированных решений. Приложение «Путешествия по России» способно занять свою нишу на рынке, предложив пользователям качественно новые возможности персонализированного планирования маршрутов, детализированной фильтрации по специфическим критериям и улучшенного контента по региональным российским туристическим направлениям.

Анализ конкурентного сервиса Russpass

Russpass является специализированной цифровой туристической платформой, реализуемой при поддержке Правительства Москвы с целью стимулирования развития и популяризации внутреннего туризма в Российской Федерации. Запущенная в 2020 году, платформа представляет собой значимый шаг в области цифровизации туристических услуг, предоставляя пользователям широкий спектр возможностей для удобного и эффективного планирования туристических поездок.

Преимущества Russpass:

1. Комплексность и интегрированность предлагаемых услуг. Russpass предлагает пользователям удобные инструменты для организации

полного цикла путешествия, включая бронирование гостиниц и других мест размещения, приобретение билетов на различные виды транспорта, заказ экскурсий, а также планирование развлекательных и культурных мероприятий. Такая комплексная интеграция различных услуг позволяет минимизировать затраты времени на организационные вопросы и повысить удобство путешествия.

2. Государственная поддержка и сотрудничество с региональными администрациями. Благодаря тому, что платформа Russpass является проектом, инициированным Правительством Москвы, она активно взаимодействует с органами власти регионов, местными туристическими организациями и учреждениями культуры. Это взаимодействие обеспечивает высокий уровень достоверности и актуальности предоставляемой информации, способствует развитию региональной туристической инфраструктуры, привлекая дополнительные инвестиции и стимулируя развитие локальных туристических продуктов.
3. Многоязычность и международная доступность. Russpass представлен на нескольких языках: русском, английском, испанском и арабском, что значительно расширяет целевую аудиторию и делает платформу привлекательной как для российских туристов, так и для гостей из-за рубежа. Такая многоязычность также способствует продвижению Российской Федерации как привлекательного туристического направления на международном уровне.
4. Интуитивность интерфейса и удобство эксплуатации. Платформа Russpass характеризуется высоким уровнем удобства и простоты пользовательского интерфейса, что позволяет эффективно использовать её

как опытным, так и начинающим путешественникам. Сервис регулярно улучшается и дорабатывается, обеспечивая простые и доступные инструменты для различных категорий пользователей.

Недостатки Russpass:

1. Ограничность персонализации туристических продуктов. Несмотря на достаточно широкий ассортимент услуг, Russpass не обладает достаточным уровнем персонализации маршрутов, особенно когда речь идет о специальных потребностях отдельных категорий туристов, таких как маломобильные граждане, семьи с детьми, пожилые путешественники и лица с особыми потребностями здоровья.
2. Региональная неравномерность качества и наполненности контента. Качество и разнообразие предложений платформы во многом определяются уровнем активности и заинтересованности региональных партнёров, что приводит к значительным различиям в представлении информации о разных регионах. Это может существенно ограничивать привлекательность платформы для путешествий в менее развитые туристические направления.
3. Слабое развитие социального взаимодействия и коммуникационных функций. В отличие от более специализированных туристических сервисов, Russpass не обладает достаточными инструментами для активного взаимодействия пользователей между собой, такими как обмен рекомендациями, формирование сообщества путешественников и совместная организация поездок.
4. Невысокая узнаваемость и необходимость усиленного маркетинга

вого продвижения. Несмотря на поддержку на государственном уровне, узнаваемость Russpass среди широкой аудитории остаётся на относительно невысоком уровне, что требует дополнительных усилий и вложений в маркетинг и продвижение бренда.

Russpass представляет собой важный проект, способствующий развитию внутреннего туризма в России посредством цифровизации и интеграции туристических услуг. Вместе с тем, выявленные недостатки и ограничения платформы открывают пространство для развития дополнительных специализированных решений. В этом контексте мобильное приложение «Путешествия по России» способно дополнить Russpass, предлагая туристам более глубокие возможности для персонализации маршрутов, увеличенное покрытие региональных направлений и развитые механизмы социального взаимодействия, что в итоге повысит привлекательность и конкурентоспособность туристических предложений.

Анализ конкурентного сервиса izi.TRAVEL

izi.TRAVEL представляет собой глобальную цифровую платформу, предназначенную для распространения интерактивных аудиогидов и виртуальных экскурсионных туров по городам, музеям и культурно-историческим объектам. Запущенная в 2011 году международной командой разработчиков из Нидерландов и России, платформа ставит перед собой амбициозную задачу по расширению доступности культурного и исторического наследия через использование передовых цифровых технологий и инновационных решений.

Преимущества izi.TRAVEL:

1. Глобальный охват и богатство контента. На начало 2023 года платформа включает более 1600 музеиных аудиогидов и свыше 4000 городских туроров, созданных тысячами авторов по всему миру. Контент izi.TRAVEL представлен в более чем 90 странах на 70 языках, что существенно расширяет потенциальную аудиторию и делает его доступным для туристов с самыми разнообразными культурными и языковыми особенностями.
2. Доступность и демократичность создания контента. Открытая модель платформы позволяет участвовать в процессе создания аудиогидов и виртуальных экскурсий не только профессиональным учреждениям (музеям, туристическим офисам и культурным организациям), но и независимым авторам, блогерам и энтузиастам. Это способствует многообразию контента, регулярному обновлению и появлению уникальных и нишевых экскурсий, отражающих различные интересы и запросы пользователей.
3. Инновационный подход к использованию технологий. izi.TRAVEL активно внедряет и использует современные цифровые технологии, такие как QR-коды, iBeacon, NFC, геолокационные сервисы и фотонавигация. Это обеспечивает удобный и оперативный доступ к необходимому контенту, облегчает взаимодействие пользователей с платформой и повышает качество пользовательского опыта в музеиных пространствах и городских условиях.
4. Простота и доступность интерфейса. Платформа отличается простым, интуитивно понятным интерфейсом, доступным для широкой пользовательской аудитории, включая людей разного возраста и уровня

владения современными технологиями. Широкий выбор доступных языков дополнительно увеличивает удобство и делает сервис вос требованным как среди локальных туристов, так и среди междуна родных путешественников.

Недостатки izi.TRAVEL:

1. Вариативность качества контента. Открытость платформы и отсутствие строгой модерации приводят к тому, что качество представленных аудиогидов значительно варьируется – от профессионально созданных материалов до любительских и недостаточно качественных экскурсий, что может негативно повлиять на общий пользовательский опыт.
2. Проблемы устойчивости и монетизации. Бизнес-модель платформы, основанная преимущественно на рекламных доходах, требует привлечения большой пользовательской базы и регулярного обновления качественного контента. Это создает финансовые и стратегические вызовы в условиях возрастающей конкуренции на туристическом рынке и неопределенности доходов от рекламы.
3. Ограничения технологической доступности. Используемые платформой технологии, такие как QR-коды, iBeacon и NFC, требуют от пользователей наличия современных мобильных устройств и определенных технических навыков. Это может ограничивать охват платформы, исключая из числа пользователей тех, кто не обладает необходимым оборудованием или навыками для взаимодействия с предлагаемыми технологиями.

4. Конкуренция с локальными туристическими решениями. В различных странах и регионах активно развиваются собственные специализированные сервисы, предлагающие контент, который зачастую более детализирован и актуален для местной аудитории. Это может ограничивать конкурентные преимущества и привлекательность izi.TRAVEL на отдельных рынках.

izi.TRAVEL занимает значимую позицию на глобальном туристическом рынке, предоставляя пользователям обширный и разнообразный выбор интерактивных аудиоэкскурсий и виртуальных туров. Вместе с тем, существующие проблемы с качеством контента, устойчивостью финансовой модели и ограничениями технологической доступности требуют стратегических шагов по совершенствованию платформы. Для долгосрочного развития и укрепления конкурентных позиций izi.TRAVEL необходимо улучшать механизмы модерации контента, расширять источники монетизации, активнее привлекать профессиональные организации и укреплять партнерства с региональными и локальными туристическими и культурными учреждениями.

Анализ конкурентного сервиса Komoot

Komoot – это специализированная цифровая платформа для планирования маршрутов и навигации, ориентированная на активный отдых и туризм, включая пешие, велосипедные и туристические маршруты. Сервис был основан в 2010 году в Германии и активно развивается, привлекая аудиторию, увлечённую outdoor-активностями.

Преимущества Komoot:

1. Подробное и интерактивное планирование маршрутов. Платформа предоставляет возможность составлять маршруты, учитывая различные критерии: тип активности (велосипед, пеший туризм, бег и другие), физическую подготовку пользователей, уровень сложности маршрута, а также интересы и предпочтения.
2. Широкая интеграция с устройствами. Комоот совместим с большинством популярных GPS-устройств, включая велокомпьютеры и умные часы (Garmin, Wahoo, Apple Watch и другие). Это позволяет пользователям легко и удобно использовать платформу как во время планирования маршрута, так и в процессе его прохождения.
3. Социальные функции и пользовательский контент. Пользователи имеют возможность делиться созданными маршрутами, комментариями и фотографиями, образуя сообщество единомышленников, что значительно увеличивает вовлечённость и привлекательность сервиса.
4. Возможность использования в оффлайн-режиме. Пользователи могут заранее скачивать маршруты и карты, что особенно актуально для путешествий в регионы с нестабильным интернет-соединением.

Недостатки Komoot:

1. Ограниченностю бесплатной версии. Для полного доступа к функционалу пользователям необходимо приобретать доступ к отдельным регионам или оформлять подписку, что ограничивает возможности использования сервиса для некоторых категорий путешественников.
2. Разное качество картографической информации по регионам. В отдалённых и менее популярных туристических зонах карты и марш-

руты могут быть менее детализированы, что снижает качество пользовательского опыта.

3. Зависимость от пользовательского контента. Качество представленных маршрутов и рекомендаций может значительно варьироваться, так как зависит от активности и компетентности самих пользователей.

Статистические данные и показатели:

- Согласно данным за 2022 год, платформа Komooot насчитывает более 30 миллионов пользователей по всему миру.
- Пользователями было создано более 100 миллионов индивидуальных маршрутов, что подчеркивает популярность сервиса и высокий уровень вовлечённости.
- Средняя оценка приложения в магазинах App Store и Google Play составляет 4,6 балла из 5 возможных, что свидетельствует о высоком уровне удовлетворенности пользователей.

Komooot занимает важную нишу на рынке туристических и навигационных решений для активного отдыха, предоставляя пользователям удобные инструменты для планирования и прохождения маршрутов. Несмотря на существующие ограничения, такие как необходимость оплаты дополнительных регионов и вариативность качества контента, платформа продолжает расти и привлекать новых пользователей. Для дальнейшего укрепления своих позиций Komooot необходимо работать над повышением качества картографической информации в удалённых регионах, а также развивать механизмы проверки и модерации пользовательского контента.

Анализ конкурентного сервиса Find Penguins

Find Penguins представляет собой цифровую платформу, ориентированную на автоматическое ведение и документирование путешествий. Созданная в 2014 году, платформа быстро набрала популярность среди пользователей, ценящих возможность удобного и автоматизированного сбора впечатлений и создания красочных воспоминаний о поездках.

Преимущества Find Penguins:

1. Автоматическое отслеживание и документирование маршрутов. Платформа позволяет пользователям автоматически фиксировать передвижения, создавая детализированный журнал путешествий без необходимости вручную вносить информацию.
2. Создание персонализированных фотоальбомов и отчётов. Сервис предоставляет удобные инструменты для генерации визуально привлекательных отчётов, объединяющих фотографии, комментарии и заметки о путешествиях, позволяя легко сохранять и делиться впечатлениями.
3. Интерактивные карты путешествий. Пользователи имеют возможность отображать маршруты своих поездок на интерактивной карте, что значительно улучшает наглядность и удобство восприятия информации.
4. Социальная интеграция и формирование сообщества. Find Penguins предлагает инструменты для взаимодействия между пользователями: можно следить за путешествиями друзей, оставлять комментарии

и оценки, способствуя формированию активного и вовлечённого туристического сообщества.

Недостатки Find Penguins:

1. Зависимость от точности GPS-сигнала. Автоматическое отслеживание передвижений может давать сбои или неточности в местах с плохим покрытием GPS-сигнала, снижая качество итоговых отчётов.
2. Ограничения бесплатного функционала. Полный доступ ко всем возможностям платформы, таким как расширенные настройки и некоторые функции, доступен только после оформления платной подписки.
3. Вопросы конфиденциальности. Автоматическое документирование местоположения пользователя вызывает у части аудитории опасения, связанные с защитой личной информации и приватности данных.

Статистические данные и показатели:

- На данный момент платформа не предоставляет публичных данных о точном количестве пользователей и активности. Однако отзывы и рейтинги в магазинах приложений указывают на позитивный пользовательский опыт.
- В Google Play и App Store приложение получает высокие оценки, преимущественно выше 4 баллов из 5 возможных, что подтверждает его популярность среди целевой аудитории.

Find Penguins эффективно занимает нишу автоматизированного ведения путешествий и создания интерактивных отчётов, что делает его привлекательным инструментом для многих путешественников. В то же время сервису необходимо продолжать работу по улучшению точности и надёжности

отслеживания маршрутов, расширять возможности бесплатного функционала и совершенствовать механизмы защиты данных пользователей для повышения конкурентоспособности и увеличения пользовательской базы.

1.5 Бизнес-модель стартапа

В этом разделе будет разобрана бизнес-модель стартапа по шаблону Александра Остервальдера.

Сегменты клиентов

Основной целевой сегмент проекта «Путешествия по России» – туристы, планирующие поездки по территории страны. В эту группу входят различные демографические и поведенческие подгруппы: молодые путешественники и студенты, семейные туристы, пары и пенсионеры, интересующиеся экскурсиями и отдыхом внутри России. Кроме того, выделяются сегменты по типу интересов (любители природы, культурно-познавательного туризма, активных развлечений) и по формату поездки (самостоятельное планирование, групповые туры, корпоративный туризм). Привлечение клиентов будет осуществляться как среди постоянных отпускников, так и среди тех, кто раньше преимущественно путешествовал за рубеж: текущие тенденции свидетельствуют о росте внутреннего туризма в России. Так, по оценке «Яндекс.Путешествий», объём российского рынка гостиничных услуг в 2023 г. составит порядка 785 млрд руб., и при развитии внутреннего туризма он может вырасти до 1 трлн руб. к 2027 г. Это подтверждает перспективность платформы, ориентированной именно на российских путешественников.

Ценностные предложения

Проект «Путешествия по России» предлагает пользователям единый онлайн-сервис для планирования и бронирования поездок внутри страны. В качестве ценностного предложения выделяются: широкий ассортимент туристических продуктов (отелей, транспорта, экскурсий) с упором на региональные достопримечательности России; качественный контент (маршруты, путеводители, инсайдерские советы), который помогает пользователям открывать новые места; удобство и экономия времени за счёт объединения поиска, сравнения и бронирования в одном приложении. Сервис также обеспечивает систему поиска и фильтраций (учёт предпочтений путешественника) и поддержку на всех этапах путешествия. Наконец, платформа ориентирована на демонстрацию преимуществ российских регионов и малознакомых туристических направлений, что выгодно отличает её от международных ОТА и стимулирует развитие внутреннего туризма.

Каналы

Доступ к клиентам планируется через цифровые каналы. Основным каналом дистрибуции будет официальное мобильное приложение старта, оптимизированное для поиска маршрутов. Для привлечения аудитории будут использоваться социальные сети (Instagram, ВКонтакте, «Одноклассники», Telegram-каналы) с публикацией вдохновляющего контента и экскурсий, а также e-mail-рассылки с персональными предложениями. Важным каналом станут партнёрства с блогерами и медиа в туристической сфере: публикации и рекламные материалы в популярных туристических онлайн-изданиях и на порталах, посвящённых путешествиям по России.

В оффлайне проект может представлять себя на туристических выставках и сотрудничать с региональными туристскими организациями, размещая информацию на сайтах региональных турфирм.

Взаимоотношения с клиентами

Стартап стремится выстраивать доверительные и долгосрочные отношения с путешественниками. На платформе реализуется система самообслуживания: пользователю доступен личный кабинет, где он может сохранять избранные маршруты, оформлять повторные заявки и отслеживать историю бронирований. Для поддержания контакта используются автоматические уведомления и социальное взаимодействие между пользователя в виде реакций и комментариев.

Потоки доходов

Основными источниками дохода проекта являются партнёрские комиссии и нативная реклама. Во-первых, платформа взимает комиссию с партнёров за каждое совершённое через сервис бронирование (отелей, билетов, туров и т.д.). Это соответствует модели агентства (*agency model*) в OTA-бизнесе: компания не закупает инвентарь, а получает процент от продаж партнёров. Так, по опыту крупных онлайн-тревел-агентств, средний размер комиссии составляет примерно 15–30% от стоимости услуги. Во-вторых, проект продаёт рекламные места в виде интегрированных рекламных материалов (нативной рекламы) на сайте и в приложении. Таким образом, доход формируется за счёт комиссий с бронирований и платы за размещение нативных промо-материалов от партнёров, а также за счёт

возможных платных интеграций с внешними сервисами (модель CPA/CPC при подключении сторонних офферов).

Ключевые ресурсы

Ключевым ресурсом проекта является собственная техническая платформа (мобильное приложение) с метапоиском и системой управления контентом. Не менее важны накопленные базы данных по отелям, транспорту, экскурсиям, а также контент (фотографии, описания маршрутов, видео) о туристических направлениях России. Ключевыми ресурсами также являются разработчики проекта, которые поддерживают работоспособность сервиса.

Ключевые виды деятельности

Основные виды деятельности проекта включают разработку и поддержку цифровой инфраструктуры (разработка новых функций, поддержка сайта и приложения), а также постоянное обновление и расширение контента (добавление новых маршрутов и путеводителей). Ежедневная работа включает поиск и подключение партнёров (автоматы бронирований гостиниц, железных дорог, туроператоров), ведение переговоров о сотрудничестве и условиях комиссий. Маркетинг и продвижение – ещё одна ключевая задача: настройка онлайн-рекламы, ведение соцсетей, создание рекламных кампаний и промо-акций. Проект также уделяет внимание аналитике пользовательского поведения (для улучшения рекомендаций) и обслуживанию клиентов (ответы на запросы, разрешение спорных ситуаций при бронировании).

Ключевые партнёры

Ключевые партнёрские категории и форматы взаимодействия:

- Отели и средства размещения: заключение договоров на интеграцию системы бронирования (через API или канал-менеджеры) и взимание комиссии с каждого подтверждённого бронирования. Проект предлагает отелям дополнительный поток клиентов, а за счёт гарантированной загрузки снимает риски недозагрузки.
- Транспортные компании: авиакомпании, железные дороги, автобусные перевозчики. Интеграция с их системами бронирования (или подключение через GDS/агрегаторы) позволяет пользователям искать билеты в одном интерфейсе. С транспортными партнёрами также обсуждается комиссия с продажи билетов.
- Экскурсионные и туроператоры: поставщики экскурсионных программ, активного отдыха, туров (как массовых, так и индивидуальных). С ними реализуется механизм размещения туров на платформе и получения комиссии с продаж турпакетов. Проект может совместно с туроператорами формировать пакетные предложения «отель + экскурсия» или «маршрут + транспорт».
- Региональные туристические организации и медиа: взаимодействие с туристическими ведомствами и региональными турагентствами для продвижения направлений. Это может быть обмен контентом (спонсируемые статьи, совместные маркетинговые кампании) и размещение спонсорского материала о регионах. Такие партнёры помогают расширить аудиторию проекта и обогащают его контентную часть.

Основные статьи затрат включают:

- Разработка и поддержка ИТ-платформы (включая разработку, хостинг, закупку облачных ресурсов и лицензий).
- Расходы на маркетинг и продвижение (онлайн-реклама, SEO-продвижение, SMM, участие в выставках, партнерские программы).
- Фонд оплаты труда команды (разработчики, маркетологи, специалисты по работе с клиентами и партнёрами, контент-менеджеры).
- Операционные и административные расходы (офисные затраты, бухгалтерия, юридическое сопровождение, плата за платежные системы).
- Расходы на привлечение и удержание партнёров (например, бонусы за высокие показатели продаж, участие в совместных акциях).

1.6 Экосистема монетизации

Проект «Путешествия по России» формирует доходы через две взаимосвязанные составляющие: комиссионные с партнёров и нативную рекламу. Комиссионные платежи являются основой бизнес-модели: стартап действует по принципу OTA-агентства (agency model), то есть получает процент от стоимости каждой успешно совершённой транзакции.

Например, при бронировании отеля или экскурсии сервис удерживает установленную долю (часто в диапазоне 10–30%) от суммы сделки. Такой подход снижает финансовые риски для проекта: оплата производится только при фактической продаже, партнёр не несёт дополнительных

издержек, а платформа мотивирована привлекать больше бронирований для увеличения дохода. В роли партнёров могут выступать отели, туроператоры, транспортные компании, страховые провайдеры и т.д.; с каждым из них оговариваются условия комиссии и формат взаимодействия (API-подключение или офферы по модели CPA/CPC).

Второй компонент монетизации – нативная реклама. Под нативной рекламой понимаются рекламные материалы, органично встроенные в пользовательский опыт платформы (например, спонсируемые статьи, рекомендованные туры или отели в списках контента, интерактивные карты достопримечательностей с брендированными подсказками). Такой формат позволяет рекламодателям (отелям, турфирмам, туристическим регионам) продвигать свои предложения без агрессивных баннеров. Исследования показывают, что пользователи предпочитают нативную рекламу стандартным баннерам: например, по данным FreakOut, путешественники охотнее взаимодействуют с органично встроенными объявлениями и уделяют им больше внимания.

При этом нативные объявления в совокупности получают более высокий CTR, чем обычные баннеры. По данным Outbrain/Econsultancy, размещённая на премиальных площадках нативная реклама вызывает на 44% больше доверия со стороны аудитории и обеспечивает на 21% более высокую кликабельность, чем традиционные форматы [13]; конверсия (покупки) в таких объявлениях также выше примерно на 24%. Кроме того, нативные объявления воспринимаются как менее навязчивые: по их словам, 95% пользователей негативно реагируют на прерывающую рекламу, тогда как нативные форматы не разрывают поток контента. Таким образом, сочетание партнёрских комиссий и нативной рекламы обеспечивает муль-

тиканальный поток дохода. Комиссионная модель гарантирует, что платформа зарабатывает напрямую при росте продаж партнёров, а нативные объявления повышают доход за счёт платных промо-блоков и спонсорского контента. Преимуществами выбранной схемы являются низкий риск для стартапа (оплата от партнёров только за результат) и высокая эффективность рекламы: нативные форматы органично дополняют сервис и лучше вовлекают туристов, что выгодно и площадке, и рекламодателям.

1.7 Финансовая модель

Распределение инвестиций. Основные статьи начальных вложений – оплата труда команды, маркетинг и инфраструктура. Предлагаемая смета вложений (примерно):

Статья расходов	Сумма, руб
Разработка MVP (3 мес. зарпл.)	1 800 000
Налоги и взносы ($\approx 30\%$)	540 000
Маркетинг (реклама запуска)	1 000 000
Серверы и инфраструктура (год)	300 000
Публикация в App Store/GP	10 000
Поддержка/администрирование	100 000
Итого	3 750 000

Таблица 2. Смета начальных вложений

За счёт всех статей требуется примерно 3,5–4,0 млн руб. стартовых инвестиций. Поддержка включает первые месяцы работы (например, техподдержку, модерацию). Затраты на серверы зависят от нагрузки: для неболь-

шого приложения достаточно порядка 20–30 тыс. руб./мес (используя облачные VM и СУБД), то есть 300 тыс. в год. Монетизация. Приложение бесплатное, доход формируется за счёт нативной рекламы и партнёрских комиссий. Реклама: предлагаются местные объявления (отели, транспорт, экскурсии) как нативный контент в маршрутном приложении. Допустимая ставка CPM (стоимость 1000 показов) на российском рынке сейчас в среднем высокая – порядка 150 руб. CPM (данные об общих ставках CPM в соцсетях за 2024 год – 138–182 руб.). При этом средний пользователь может видеть, например, несколько десятков показов рекламы в месяц. Если считать, что пользователь генерирует 30 показов рекламы в месяц, при CPM 150 руб. это 4,5 рубля/мес, или 54 руб./год от рекламы на одного MAU. Комиссии партнёров: за бронирование жилья и туров обычно берут около 5–15% от суммы заказа. В РФ, например, агрегаторы брони отелей платят партнёрам ≈8–10% (Яндекс.Путешествия – 8%, «Островок» – 6%, брендовые программы – 5–8%). Авиабилеты приносят низкую комиссию (1–3%), поезда и автобусы – ≈5–10%. Пусть в среднем при умелом продвижении в приложении на одного активного пользователя приходится комиссия 100–200 руб. в год (зависит от конверсии пользователей, доли бронирующих и среднего чека). Например, если 20% MAU совершают бронирование отеля на 20 000 руб., при комиссии 10% получаем ≈400 руб. на этих 20% пользователей, т.е. 80 руб. в среднем на пользователя. Добавляя туры/перевозки – может дойти до 150–200 руб. с пользователя в год. Таким образом средний годовой доход на пользователя (ARPU) будем оценивать примерно в 200–300 руб./год (≈17–25 руб./мес), включающий 50–100 руб. от рекламы и 150–200 руб. от комиссий. С такими допущениями финансовая модель будет выглядеть следующим образом. Прогноз доходов. Оценим доходы

по годам. Предполагаем постепенный рост MAU и доли монетизирующихся пользователей:

Год	Среднее MAU	Доход рекламы, млн руб	Доход комиссий, млн руб	Всего, млн руб
1	30 000	(30 000×70 руб)≈2.1	(30 000×200 руб)≈6.0	8.1
2	70 000	(70 000×180 руб)≈12.6	(70 000×240 руб)≈16.8	29.4
3	120 000	(120 000×210 руб)≈25.2	(120 000×250 руб)≈30.0	55.2

Таблица 3. Прогноз доходов

Примечание: в модели Year1 консервативно взяты ARPU рекламы ≈70 руб/год (54 руб от СРМ + неточисленная прочая реклама), ARPU комиссии ≈200 руб/год. К 3-му году ARPU можно ожидать рост до 300–400 руб./год (увеличится конверсия и средний чек): так доход от рекламы и комиссий существенно вырастет. Эти цифры иллюстративны, демонстрируют порядок величин доходов: уже во 2–3 год при хорошей динамике MAU доходы могут существенно превышать расходы. Точка безубыточности и окупаемость. Постоянные затраты после запуска – преимущественно зарплаты (≈600 000 руб./мес) + инфраструктура (30 000) + маркетинг (например, 100–200 тыс. руб.) ≈800 000 руб/мес. При ARPU ≈200–250 руб/год (≈16–21 руб/мес) для безубыточности нужно приблизительно 40–50 тыс. MAU. Например, $800\ 000 \div 20 \approx 40\ 000$ человек. Это MAU, при котором выручка покрывает расходы. С точки окупаемости: если первоначально вложено 3,75 млн руб, а после выхода MAU растёт до точки безубыточности за 9–12 ме-

сяцев, то при дальнейшем росте выручки проект может выйти в прибыль к концу 2-го года. При наших допущениях (ARPU 300 руб/год, MAU стабильно растёт до 70–100 тыс.) инвестиции вернутся за ≈20–24 месяца. Оценка рынка и пользовательский потенциал

- Объём рынка. Ежегодно россияне совершают сотни миллионов поездок по стране. Даже в кризисных условиях внутренний туризм резко растёт – 78 млн турпоездок в 2023, ожидается более 90–96 млн в 2024. Отпуск с ночёвкой в отеле практикуют десятки миллионов человек. Учитывая, что проникновение смартфонов – ≈73% и распространение мобильных сервисов, потенциальная аудитория приложения измеряется десятками миллионов. Даже 1% от поездок (≈0,8–1,0 млн поездок) может приносить сотни тысяч активных пользователей.
- Конкуренция. Рынок travel-приложений насыщен: крупнейшие сервисы («Яндекс.Путешествия» – 47% пользователей, «Суточно.ру» – 40%, «Туту.ру» – 39% опрошенных) ужеочно вошли в жизнь путешественников. При этом нет массовых приложений, специализирующихся именно на создании/обмене маршрутов с учётом доступности. Реально притянуть в первый год можно несколько десятков тысяч MAU (консервативная оценка) – при благоприятном маркетинге и фокусе на нишевые сегменты. Например, даже 0,5% от потенциальной базы (здесь 1% внутреннего турпотока – 1 млн поездок в год) даст порядка 5–10 тыс. пользователей; агрессивные маркетинговые кампании могут увеличить эту цифру до 50–100 тыс. MAU в год.
- Привлечение пользователей. Стоимость привлечения клиента в сегменте «Туризм» по рынку – около 1800 руб., что говорит о высоком

ких расходах на рекламу. С учётом этого и высокой конкуренции важно оптимизировать продвижение (таргет в соцсетях, коллаборации с региональными турофициами, контент-маркетинг). Цену клика в VK/Instagram сейчас можно оценивать 10–20 руб., а CPM – 150–200 руб. При разумном маркетинге первые месяцы можно заложить бюджет 500–1000 тыс. руб. на привлечение MAU.

Например, если годовые расходы составляют 1,5–2 млн руб, а ARPU ≈ 100 руб/год, то безубыточный MAU будет порядка 15–20 тыс. пользователей в месяц. Если же учесть всю инвестицию 3,75 млн руб как расходы за период, то при ARPU 80 руб/год точка безубыточности достигается примерно при MAU ≈ 45–50 тыс. пользователей. Таким образом, с учётом наших допущений проект выйдет на безубыточность уже в конце первого года или начале второго. В дальнейшем, при росте MAU до 100 тыс. к третьему году и сохранении ARPU ≈ 100 руб/год, чистая прибыль существенно превышает постоянные расходы. По нашим прогнозам, маржинальность бизнеса к концу третьего года может достигать 30–40% (при учёте только переменных и операционных затрат). Таким образом, вложенные 3,75 млн руб полностью окупятся, а внутренняя норма рентабельности инвестиций (ROI) к концу прогнозного периода будет на уровне 100% и выше.

Выводы по главе

В первой главе была выполнена всесторонняя подготовительная работа, заложившая фундамент для дальнейшей разработки мобильного приложения «Путешествия по России».

- Проведён сравнительный анализ шести ключевых сервисов (Яндекс.Карты,

TripAdvisor, Russpass, izi.TRAVEL, Komoot, Find Penguins). Выявлены их сильные стороны — высокая детальность карт, развитая социальная база, комплексные услуги бронирования — и ограничения — неоднородность качества контента, недостаток персонализации маршрутов, слабая работа с нишевыми регионами. Полученные результаты показали, что на рынке остаётся незаполненная ниша приложения, ориентированного именно на внутренний туризм по России, с акцентом на персонализированные маршруты, социальное взаимодействие и доступность малоизвестных локаций.

- Комбинация количественного онлайн опроса (120 респондентов) и качественных интервью позволила выявить основные потребности разных сегментов пользователей: молодых путешественников, семей с детьми, лиц с ограниченной мобильностью, пожилых туристов, тревел блогеров и организаторов мероприятий. На этой основе сформулированы десять подробных user story, охватывающих функциональные сценарии от базового поиска маршрутов до аналитики для администраторов.
- Также с помощью модели MVP установлены три уровня приоритета:
 - Высокий (MVP) – каталог маршрутов с картами и фото, фильтрация по сложности и инфраструктуре, создание пользовательских маршрутов.
 - Средний – социальные функции (комментарии, рейтинги), персонализированные уведомления, «избранное» и обмен маршрутами.

- Низкий – оффлайн доступ, углублённая персонализация, расширенная аналитика, продвинутая социальная интеграция.
- Такая градация обеспечивает фокус команды на критически значимых функциях для первого релиза, сохраняя при этом стратегический план эволюции продукта.

Таким образом, в главе 1 сформулированы четкие функциональные и нефункциональные требования к будущей системе, определены целевые сегменты пользователей и их ключевые сценарии, а также обоснованы конкурентные преимущества разрабатываемого приложения. Полученные выводы послужат методологической основой для проектирования архитектуры и реализации микросервисной платформы, обеспечив соответствие конечного продукта реальным потребностям рынка внутреннего туризма в России.

ГЛАВА 2 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ

Во второй главе будет рассмотрено архитектурное устройство серверной части приложения «Путешествия по России»: от обоснования выбора микросервисной модели и описания ключевых нефункциональных требований до детальной демонстрации диаграмм (контекстной, контейнерной, компонентных и ER моделей баз данных) и принципов организации DevOps инфраструктуры с мерами безопасности.

2.1 Описание требований и ограничений

Функциональные требования

Бэкенд мобильного приложения «Путешествия по России» реализован на языке Go (Golang) с использованием микросервисной архитектуры. Это означает, что различные функции системы разделены по независимым сервисам. Основные функциональные требования к серверной части приложения группируются по этим сервисам следующим образом:

- Сервис Auth (Регистрация и авторизация пользователей): Бэкенд должен предоставлять возможность регистрации новых пользователей и последующей авторизации (входа в систему). Это включает создание учетной записи с уникальными учетными данными, сохранение учетной информации пользователя и проверку введённых при входе данных. Сервис аутентификации должен обеспечивать выдачу средств доступа (например, токена сессии) для подтверждённых пользователей.

лей, так чтобы последующие запросы к другим сервисам выполнялись от имени авторизованного пользователя. Таким образом, Auth отвечает за то, чтобы только зарегистрированные и корректно вошедшие в систему пользователи могли пользоваться защищёнными функциями приложения.

- Сервис Profile (Управление профилем пользователя): Функционал профилей позволяет пользователям просматривать и изменять сведения своего профиля. Бэкенд должен обеспечивать получение и обновление персональных данных (например, имени, контактной информации и других настроек аккаунта). Отдельно подчёркивается возможность загрузки аватара – пользователь может загрузить изображение профиля, которое будет сохраняться в системе файлового хранилища. Кроме того, сервис Profile поддерживает систему подписок: пользователь может подписываться на обновления других пользователей или контента. Это означает, что бэкенд должен предоставлять операции для оформления подписки и отмены подписки, а также хранить список подписок каждого пользователя. В совокупности, Profile отвечает за персонализацию и социальное взаимодействие, связанное с учетной записью пользователя.
- Сервисы Content и Activity (Создание и взаимодействие с контентом): Бэкенд должен поддерживать функции создания, хранения и получения основного контента приложения, связанного с путешествиями. В приложении предусмотрены такие сущности, как маршруты путешествий, интересные места и активности. Сервис Content предоставляет интерфейсы для создания новых маршрутов и мест: пользова-

тели могут добавлять описание маршрута, координаты или адреса мест, фотографии достопримечательностей и другую сопутствующую информацию. Сервис Activity дополняет функциональность контента, отвечая за взаимодействие пользователей с этими данными. Например, Activity может обрабатывать действия пользователей, связанные с контентом: просмотр маршрута или места, отметки о посещении, возможное комментирование или оценивание (если такие возможности предусмотрены архитектурой приложения). Оба сервиса совместно обеспечивают, что пользователи могут наполнять систему информацией о путешествиях (создавать контент) и взаимодействовать с ней (получать, просматривать и отмечать контент). Запросы к данным маршрутов, мест и активностей обрабатываются этими сервисами независимо, что упрощает масштабирование и модификацию функционала для каждой категории контента.

- Сервис FileStorage (Хранение изображений и медиафайлов): для поддержки медийного контента бэкенд должен предоставлять надежное хранение фотографий и других файлов, связанных с профилями и контентом. Решение основано на S3-совместимом хранилище, что означает использование удалённого облачного хранилища по типу Amazon S3 (либо его аналога) для сохранения файлов. Сервис FileStorage принимает файлы (например, загруженные пользователем изображения аватаров или фотографий мест) и сохраняет их в системе хранения. Ключевым требованием здесь является предоставление доступа к файлам через временные ссылки с ограниченным сроком действия (TTL – Time To Live). При запросе клиента на получение изображения бэкенд генерирует уникальную URL-ссылку,

действительную лишь ограниченное время, по которой можно скачать или просмотреть файл. После истечения TTL ссылка становится недействительной. Таким образом, FileStorage реализует функциональность загрузки файлов и последующего безопасного доступа к ним, что обеспечивает как удобство (нет необходимости непосредственно хранить медиафайлы на устройстве клиента), так и контроль доступа к приватным ресурсам.

- Сервис Notification (Отправка email-уведомлений): Система должна уведомлять пользователей о различных событиях посредством электронной почты. В настоящее время реализована ключевая функция в этой категории – верификация email-адреса пользователя. После регистрации нового пользователя бэкенд с помощью сервиса Notification генерирует письмо с подтверждением (например, содержащим специальную ссылку или код активации) и отправляет его на указанный пользователем адрес электронной почты. Это функциональное требование гарантирует, что пользователь подтвердит владение указанным email, что повышает доверие к учетной записи и позволяет активировать профиль. Хотя на текущем этапе проектирования Notification сосредоточен на верификации почты, архитектура предусматривает возможность расширения данного сервиса для отправки и других типов уведомлений (например, оповещений о новых маршрутах, сообщениях от других пользователей и т.д.) при развитии проекта.

Нефункциональные требования

Помимо перечисленных функций, к бэкенду предъявляются нефункциональные требования, определяющие качественные характеристики системы. В случае данного проекта основными нефункциональными требованиями являются масштабируемость, отказоустойчивость и безопасность системы. Также учитываются общие соображения производительности, хотя строгих количественных метрик не задавалось ввиду учебного характера проекта.

- **Масштабируемость и отказоустойчивость:** Архитектура системы должна обеспечивать высокую способность к масштабированию и устойчивость к сбоям. Достигается это за счёт использования микросервисного подхода, контейнеризации сервисов и балансировки нагрузки. Микросервисная архитектура позволяет горизонтально масштабировать каждый сервис независимо: при увеличении нагрузки можно запустить дополнительные экземпляры наиболее загруженных сервисов, не затрагивая остальные. Контейнеризация (например, с использованием Docker) обеспечивает портативность и единообразие окружения для каждого микросервиса, облегчая развертывание новых инстансов и их автоматический перезапуск в случае сбоя. В качестве балансировщика нагрузки используется Envoy – современный прокси-сервер и балансировщик, распределяющий входящие запросы от клиента между экземплярами микросервисов. Envoy не только равномерно рассеивает нагрузку, но и выполняет функции обратного прокси, маршрутизируя запросы к нужному сервису (на основе URL или других критериев), а также проводит проверки работоспособно-

сти сервисов. Благодаря этому, если один из экземпляров сервиса выходит из строя, Envoy перестает направлять трафик, перенаправляя запросы к другим экземплярам. Таким образом, сочетание микросервисной архитектуры, контейнеров и балансировщика Envoy обеспечивает системе высокую масштабируемость (способность обслуживать растущее количество пользователей и запросов путем добавления ресурсов) и отказоустойчивость (способность продолжать работу при выходе из строя отдельных компонентов).

- Безопасность: Важным нефункциональным требованием является обеспечение безопасности данных и операций. В серверной части реализованы следующие меры защиты:
 - Безопасное хранение паролей: Все пользовательские пароли хранятся только в зашифрованном (хэшированном) виде. Для хэширования применяется алгоритм bcrypt, который генерирует крипостойкое хэш-значение пароля. Алгоритм bcrypt подразумевает использование соли (случайной добавки) и множественной итерации хэш-функции, что делает полученные хэши устойчивыми к попыткам подбора и радикально усложняет обратное восстановление исходного пароля. Таким образом, даже в случае утечки базы данных злоумышленник не сможет напрямую получить пароли пользователей.
 - Защита от SQL-инъекций: для всех операций, которые взаимодействуют с базой данных, используются параметризованные SQL-запросы (prepared statements). Это означает, что данные, вводимые пользователями, не включаются непосредственно в

текст SQL-запро

-са, а передаются через параметры запроса. Такой подход исключает возможность внедрения чужого SQL-кода и предотвращает атаки типа SQL-инъекция, поскольку СУБД воспринимает пользовательские данные не как часть исполняемой команды, а как параметры. Данное решение повышает общую безопасность приложения на уровне взаимодействия с базой данных.

- Контролируемый доступ к файлам: как упомянуто в функциональных требованиях, для доступа к загруженным медиафайлам используются временные ссылки с ограниченным сроком действия. Генерация временных (TTL) ссылок на файлы в хранилище обеспечивает то, что прямой доступ к файлам возможен только по ссылке, выданной сервером, и лишь в течение заданного короткого периода. После истечения времени жизни ссылки попытка повторного доступа по ней будет отклонена. Это защищает контент пользователей от несанкционированного доступа: даже если кто-то получит ссылку, она скоро станет недействительной. Кроме того, доступ к генерации таких ссылок, естественно, имеет только авторизованный сервис, что предотвращает обход проверок доступа. В совокупности, данный механизм гарантирует конфиденциальность пользовательских фотографий и других медиа, хранящихся в системе.
- Производительность и прочие характеристики: Явных количественных требований по производительности системы (таких как максимальное число обрабатываемых запросов в секунду или поддержи-

ваемое количество одновременных пользователей) на этапе проектирования не устанавливалось, поскольку проект является учебным и не предназначен для нагрузочного промышленного использования. Тем не менее, выбранная архитектура изначально нацелена на достаточно высокую эффективность: язык Golang известен своей производительностью и эффективной работой с параллелизмом, а микросервисный подход и балансировка нагрузки позволяют при необходимости наращивать пропускную способность системы. Таким образом, даже без строго заданных метрик, система спроектирована так, чтобы удовлетворительно работать при разумных объёмах данных и пользователей, соответствующих типичному сценарию использования приложения. Основной упор сделан на корректность и надёжность выполнения требуемых функций, а при возникновении необходимости масштабирования или оптимизации производительности архитектура позволяет реализовать эти улучшения без кардинальной переработки системы.

2.2 Архитектурное решение

Обоснование выбора микросервисной архитектуры

Архитектура командного мобильного приложения «Путешествия по России» основана на микросервисном подходе. Основными причинами выбора микросервисной архитектуры стали:

- Независимое масштабирование сервисов. Дробление приложения на отдельные микросервисы позволяет горизонтально масштабировать

именно те компоненты, на которые приходится наибольшая нагрузка. Каждый сервис разворачивается в достаточном числе экземпляров (контейнеров) по мере роста количества пользователей или запросов к соответствующей части системы. Такой подход обеспечивает гибкость и высокую пропускную способность при росте проекта, поскольку ресурсы можно направлять точечно на нужные модули, не затрагивая работу остальных.

- Повышенная отказоустойчивость системы. Микросервисы работают автономно, поэтому сбой одного из них не вызывает остановку всего приложения [2]. Это означает, что при выходе из строя, например, сервиса отправки уведомлений, остальные модули (поиск туроров, профили пользователей и т. д.) продолжат функционировать в штатном режиме. Локализация сбоев внутри отдельных сервисов минимизирует простоту и повышает общую доступность приложения.
- Удобство поддержки и развития. Каждый микросервис отвечает за ограниченную область функциональности, имеет меньшую кодовую базу и чёткие границы ответственности, что упрощает его понимание, тестирование и сопровождение. Команда разработки может вносить изменения в отдельный сервис без риска нарушить работу других компонентов: требуемые доработки локализованы и не затрагивают всё приложение целиком simpleone.ru. Благодаря этому ускоряется выпуск новых версий – обновлять и развёртывать можно каждый сервис по отдельности без простоев для всей системы.

На практике перечисленные факторы реализуются с помощью современных технологий контейнеризации и оркестрации. Каждый микросервис упаковывается в контейнер.

кован в контейнер (Docker) и управляетя оркестратором (например, Kubernetes), что обеспечивает изоляцию и независимость его развёртывания. Это позволяет гибко масштабировать сервисы под нагрузку и обновлять их независимо друг от друга, удовлетворяя требования проекта по нагрузке, надёжности и скорости разработки.

Преимущества использования микросервисов в проекте

Микросервисная архитектура даёт следующие ключевые преимущества для данного проекта:

- Независимое развертывание и обновление: Каждый сервис можно деплоить и обновлять автономно, без остановки всего приложения [3]. Это сокращает время вывода изменений в продакшн и снижает риски, так как при выпуске новой версии одного модуля остальные продолжают работу. При необходимости откат затрагивает лишь конкретный сервис.
- Изоляция доменных областей и ответственности: Система разбита на отдельные сервисы, каждый из которых реализует конкретную бизнес-логику (свой «домен») и строго отвечает за неё. Такое организационное разделение вокруг бизнес-функций делает разработку новых возможностей или модификацию существующих более гибкой. Команды могут фокусироваться на узких областях, а изменения в одной доменной области не влияют напрямую на другие.
- Возможность расширения системы: благодаря слабой связности компонентов добавить новую функцию можно, просто разработав новый

микросервис [4]. Новые сервисы интегрируются в общую экосистему без существенных изменений существующих модулей. Этот подход упрощает эволюцию продукта – по мере роста требований можно наращивать систему, подключая дополнительные сервисы (например, сервис аналитики путешествий или чат-поддержку), не рефакторя монолитное ядро.

- Разделение хранилищ данных: Каждый микросервис имеет свою собственную базу данных (или иной источник данных) и не разделяет её напрямую с другими. Этот принцип «база данных на сервис» устраняет глобальную точку отказа и снижает взаимозависимость модулей по данным. Каждая служба может выбирать оптимальную технологию хранения под свои задачи и управлять схемой данных независимо. В результате улучшается целостность данных в пределах домена сервиса и упрощается эволюция структуры данных – изменения в одной базе не оказывают побочного эффекта на другие сервисы.

Взаимодействие сервисов и выбор механизмов интеграции

Все взаимодействия между компонентами системы реализованы в виде удалённых вызовов через gRPC – как межсервисные коммуникации, так и обмен данными между мобильным клиентом и backend-сервисами происходит по gRPC. Данный фреймворк удалённого вызова процедур был выбран из соображений производительности и надёжности интеграции [5]. Во-первых, gRPC существенно эффективнее REST при интенсивном трафике: благодаря использованию HTTP/2 и бинарного протокола Protocol Buffers он обеспечивает низкие задержки и высокую пропускную способ-

ность передачи данных. Сообщения в формате protobuf компактны, что особенно важно для мобильного клиента (уменьшается объем передаваемых данных и ускоряется обмен). Во-вторых, gRPC строго типизирован: интерфейсы сервисов описываются в виде контракта (файлы .proto), на основе которого генерируется код клиента и сервера. Это гарантирует согласованность API и обнаружение многих ошибок на этапе компиляции, упрощая интеграцию компонентов. В-третьих, фреймворк поддерживает потоковую передачу данных (*streaming*) в обе стороны, включая двунаправленные стримы. Это позволяет при необходимости реализовать в приложении функции реального времени (например, обновление информации о геолокации или чате) поверх устойчивого соединения. Наконец, gRPC изначально рассчитан на многоплатформенность: для него доступны библиотеки на различных языках, что облегчает поддержку клиентов под Android/iOS и развитие бэкенда на разных технологиях. Следует отметить, что все gRPC-вызовы проходят через специализированный прокси-сервер Envoy, выполняющий роль обратного прокси и балансировщика нагрузки для микросервисов. Вместо прямого обращения к адресу каждого сервиса, мобильное приложение устанавливает единственное соединение с Envoy, а тот маршрутизирует запросы к нужному сервису и обратно. Таким образом достигается единообразие точки входа и эффективное распределение запросов между сервисами.

Использование брокера сообщений NATS JetStream

Помимо синхронных gRPC-взаимодействий, в системе реализована асинхронная обработка отдельных задач через брокер сообщений NATS JetStream. Данный механизм применяется для действий, не критичных к

моментальному выполнению или мгновенному ответу, например: отправка email-уведомлений, рассылка проверочного кода регистрации, логирование и другие фоновые задачи. Вместо того чтобы выполняться в рамках пользовательского запроса, такие действия помещаются в очередь сообщений и обрабатываются отдельно. Это означает, что инициирующий сервис (например, сервис регистрации пользователей) отправляет сообщение в брокер и сразу продолжает работу, не ожидая результата. Благодаря этому пользователь получает отклик от основного сервиса быстрее, система остаётся отзывчивой даже при нагрузке, а длительные операции выполняются параллельно. Например, после регистрации нового пользователя сервис аутентификации публикует задачу на отправку письма с подтверждением, которая будет обработана специальным сервисом-рассыльщиком в фоновом режиме. Такой подход разгружает основной сервис и изолирует вспомогательную логику уведомлений от основной бизнес-функции регистрации. Для реализации очередей сообщений выбран NATS с модулем JetStream, который обеспечивает хранение и доставку сообщений в асинхронном режиме. В отличие от простого режима NATS (где непрочитанные сообщения теряются при отсутствии подписчика), JetStream предоставляет persistent-очереди: сообщения сохраняются на сервере и могут быть обработаны даже если получатель подключится с задержкой. Консюмеры (потребители/службы-обработчики) подтверждают получение, что даёт гарантии доставки по крайней мере один раз. Это повышает надёжность фоновой обработки – в случае временного сбоя сервиса-получателя задачи не потеряются, а будут выполнены после восстановления. При этом система сохраняет устойчивость даже в экстремальном случае потери сообщения: подобные операции не влияют на целостность основных дан-

ных и могут быть повторены (пользователь всегда может запросить новый код подтверждения, если не получил email). Таким образом, применение асинхронного брокера снижает нагрузку с основного потока запросов, повышает масштабируемость системы (за счёт неблокирующей постановки задач) и локализует сбои вспомогательных сервисов, не нарушая работу ключевых компонентов приложения.

Обратный прокси

В архитектуре проекта используется Envoy Proxy, выступающий в качестве интеллектуального обратного прокси-сервера для всех сервисов. Он обеспечивает маршрутизацию трафика и балансировку нагрузки между микросервисами, а также служит единой точкой входа для внешних (мобильных) клиентов. Все вызовы от приложения поступают сперва на Envoy, который проверяет запрос и перенаправляет его к нужному сервису по внутреннему адресу. Это позволяет скрыть внутреннюю структуру системы от клиента и защитить сервисы от прямого доступа из вне, устранив возможность их неконтролируемого использования. Одновременно Envoy распределяет запросы между несколькими экземплярами каждого сервиса, тем самым предотвращая перегрузку отдельного узла и повышая отказоустойчивость. Envoy изначально разрабатывался как высокопроизводительный прокси для облачных микросервисных систем. Будучи написанным на C++, он обладает малыми задержками и поддерживает масштабирование под большие объёмы трафика. Важное преимущество – первоклассная поддержка HTTP/2 и gRPC: Envoy способен проксировать HTTP/2 соединения и полностью совместим с протоколом gRPC как на входящих, так и на исходящих соединениях. Это позволяет без труда интегрировать

его в нашу gRPC-коммуникацию. Кроме того, Envoy предоставляет расширенные возможности балансировки нагрузки и управления трафиком, унаследованные от корпоративных решений: автоматический повтор запросов при сбоях, отключение «нездоровых» экземпляров сервисов, ограничение скорости запросов, трассировка распределённых запросов и другие функции. Таким образом, наличие уровня Envoy упрощает масштабирование сервисов, повышает наблюдаемость системы и обеспечивает гибкое управление сетевыми аспектами без изменения кода самих микросервисов.

2.3 Диаграммы проектирования

Этот посвящен диаграммам проектирования архитектуры серверной части приложения, иллюстрирующим организацию системы на разных уровнях абстракции и служащим основой для дальнейшего анализа и проектирования архитектуры системы.

Контекстная диаграмма системы

Контекстная диаграмма системы отражает взаимодействие внешних акторов и подсистем с рассматриваемой системой. В приложении «Путешествия по России» рассматриваются следующие компоненты: конечный пользователь, мобильное клиентское приложение, серверная часть, S3-совместимое хранилище (MinIO) и внешний почтовый сервис (Brevo). Пользователь использует мобильное приложение как основной интерфейс; приложение отправляет gRPC-запросы к серверной части для выполнения операций регистрации, авторизации и получения данных о маршрутах и т.д. Кроме того,

для передачи больших файлов (например, фотографий) приложение может напрямую загружать или скачивать файлы из хранилища MinIO по предварительно полученным безопасным TTL-ссылкам. Серверная часть взаимодействует с MinIO для хранения и извлечения файлов, а также с внешним почтовым сервисом для отправки уведомлений пользователям.

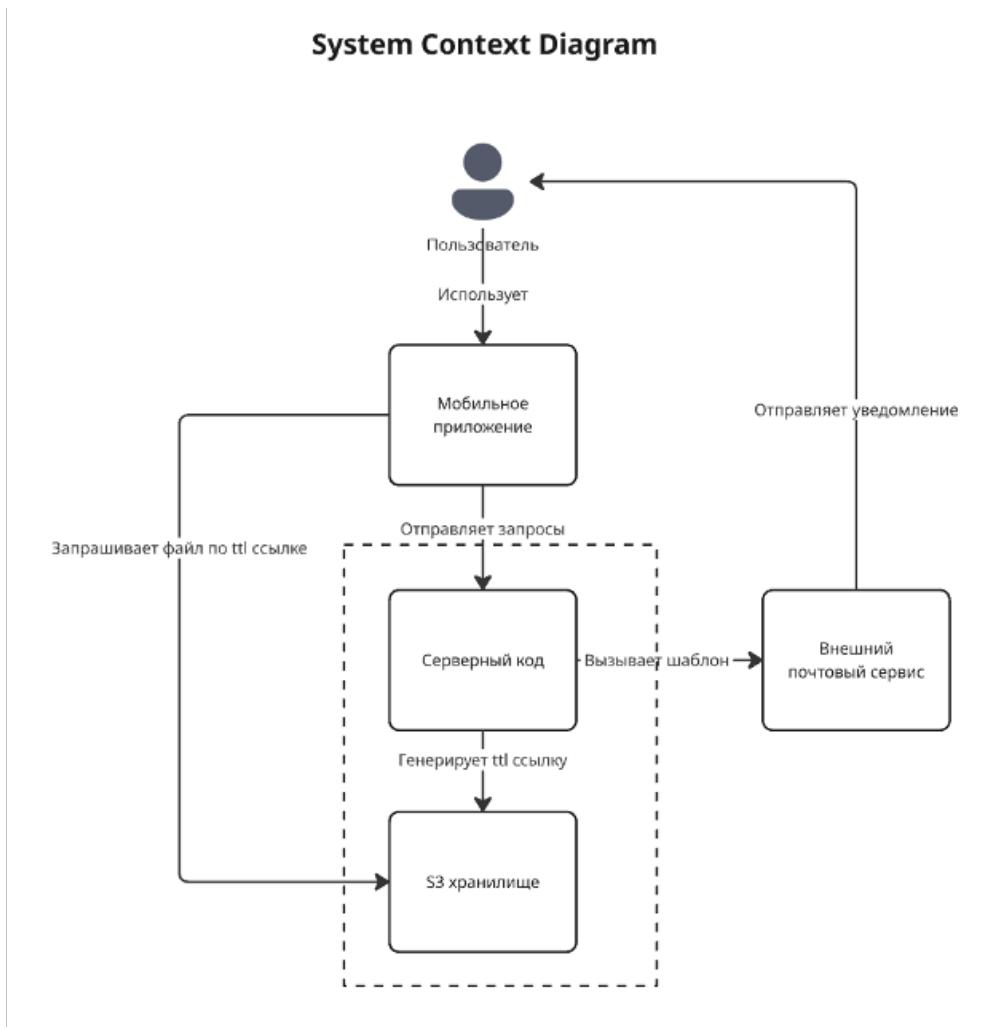


Рис. 1. Системная контекстная диаграмма

Контейнерная диаграмма

Контейнерная диаграмма демонстрирует ключевые контейнеры системы и связи между ними. В архитектуре приложения выделены следую-

щие компоненты:

- Envoy – обратный прокси/API-шлюз, принимающий все входящие gRPC-запросы и выполняющий маршрутизацию и балансировку нагрузки между сервисами;
- Auth Service – сервис аутентификации и авторизации пользователей;
- Profile Service – сервис управления профилями пользователей и подписками;
- Content Service – сервис управления маршрутами и достопримечательностями;
- Activity Service – сервис обработки пользовательской активности (лайки и комментарии);
- Notifications Service – сервис отправки уведомлений (email);
- FileStorage Service – сервис управления загрузкой и выдачей файлов, генерации временных ссылок (TTL);
- Broker Service – служба для маршрутизации и логирования событий между сервисами;
- NATS JetStream – брокер сообщений, обеспечивающий асинхронную коммуникацию и передачу событий между сервисами;
- PostgreSQL – реляционные базы данных для хранения постоянных данных сервисов;
- Redis – кэш и хранилище сессий/токенов;

- MinIO (S3-совместимое хранилище) – внешнее файловое хранилище для сохранения медиафайлов.

Входящие запросы от мобильного клиента сначала обрабатываются Envoy, который перенаправляет gRPC-вызовы к соответствующим микросервисам и обеспечивает балансировку нагрузки между их инстансами. Микросервисы взаимодействуют между собой синхронно через gRPC-вызовы и асинхронно посредством публикации и подписки на события в NATS JetStream. Таким образом обмен данными организован через комбинацию прямых RPC-вызовов и системы обмена сообщениями.

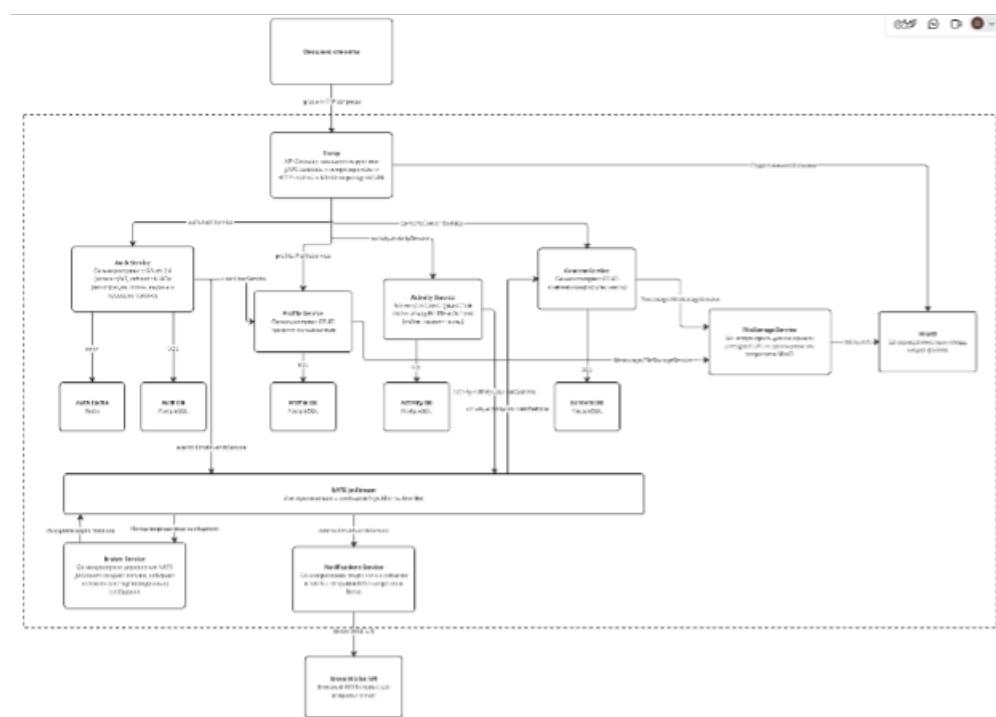


Рис. 2. Контейнерная диаграмма

Компонентные диаграммы

Для каждого микросервиса приведены компонентные диаграммы, описывающие их внутреннюю структуру и основные модули:

Auth Service отвечает за регистрацию и аутентификацию пользователей. В состав сервиса входят следующие компоненты:

- Transport (gRPC) – слой транспорта, принимающий входящие gRPC-запросы от Envoy (например, команды регистрации и логина);
- User Service – модуль бизнес-логики, управляющий данными пользователей (создание и обновление учетных записей);
- Token Service – модуль управления токенами (генерация и проверка JWT или других токенов доступа);
- PostgreSQL – реляционная база данных, содержащая таблицу users с полями user_id (PK, UUID), email, пароль и другими атрибутами учетных записей;
- Redis – кэш для хранения активных сессий и токенов доступа (для быстрой проверки валидности токенов);
- NATS JetStream – брокер сообщений для публикации событий (например, оповещения других сервисов о регистрации нового пользователя).

Сервис выполняет логику авторизации и выдачи токенов, обеспечивает хранение информации о пользователях и токенах, а также публикует события об изменениях в учетных записях.

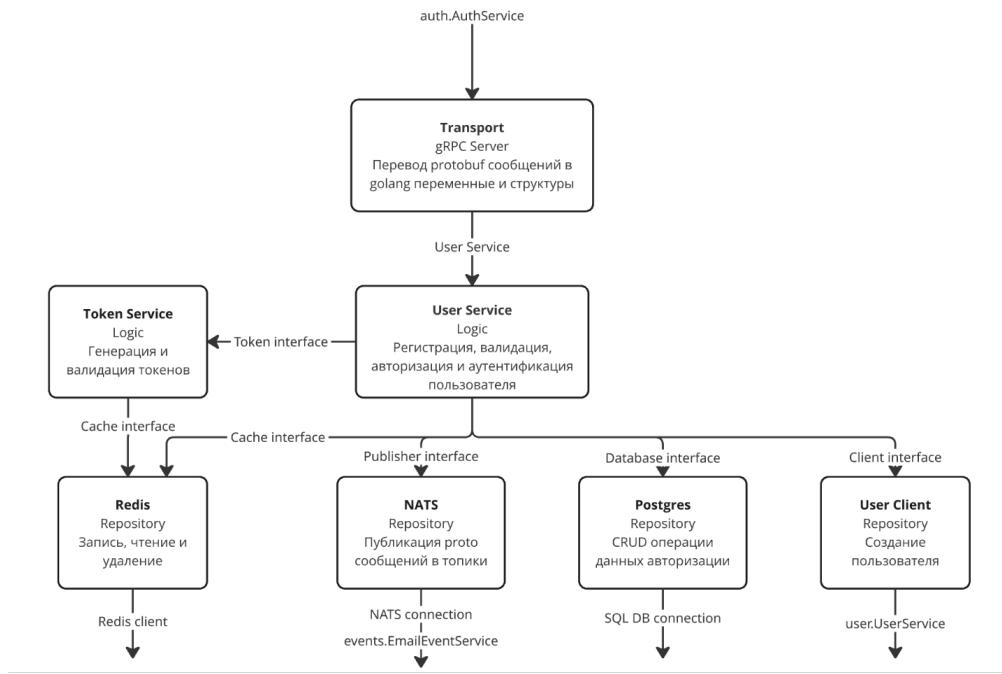


Рис. 3. Компонентная диаграмма Auth сервиса

Profile Service управляет информацией профиля пользователя и его социальными связями. В состав микросервиса входят:

- **Transport (gRPC)** – интерфейс приема gRPC-запросов для создания и редактирования профиля, а также управления подписками;
- **Profile Logic** – бизнес-логика сервиса, обрабатывающая операции создания/обновления профиля и управления отношениями «подписка–подписчик»;
- **PostgreSQL** – база данных с таблицами `profiles` (профили пользователей) и `follows` (таблица отношений «подписчик–подписка»);
- **FileStorage Client** – клиентский модуль для взаимодействия с FileStorage Service при загрузке аватаров и других файлов профиля.

Profile Service позволяет пользователям редактировать свои профили (имя, аватар, информацию о себе) и управлять списками подписок на других пользователей. Компоненты сервиса обеспечивают проверку и запись данных в базу и загрузку файлов аватаров в файловое хранилище.

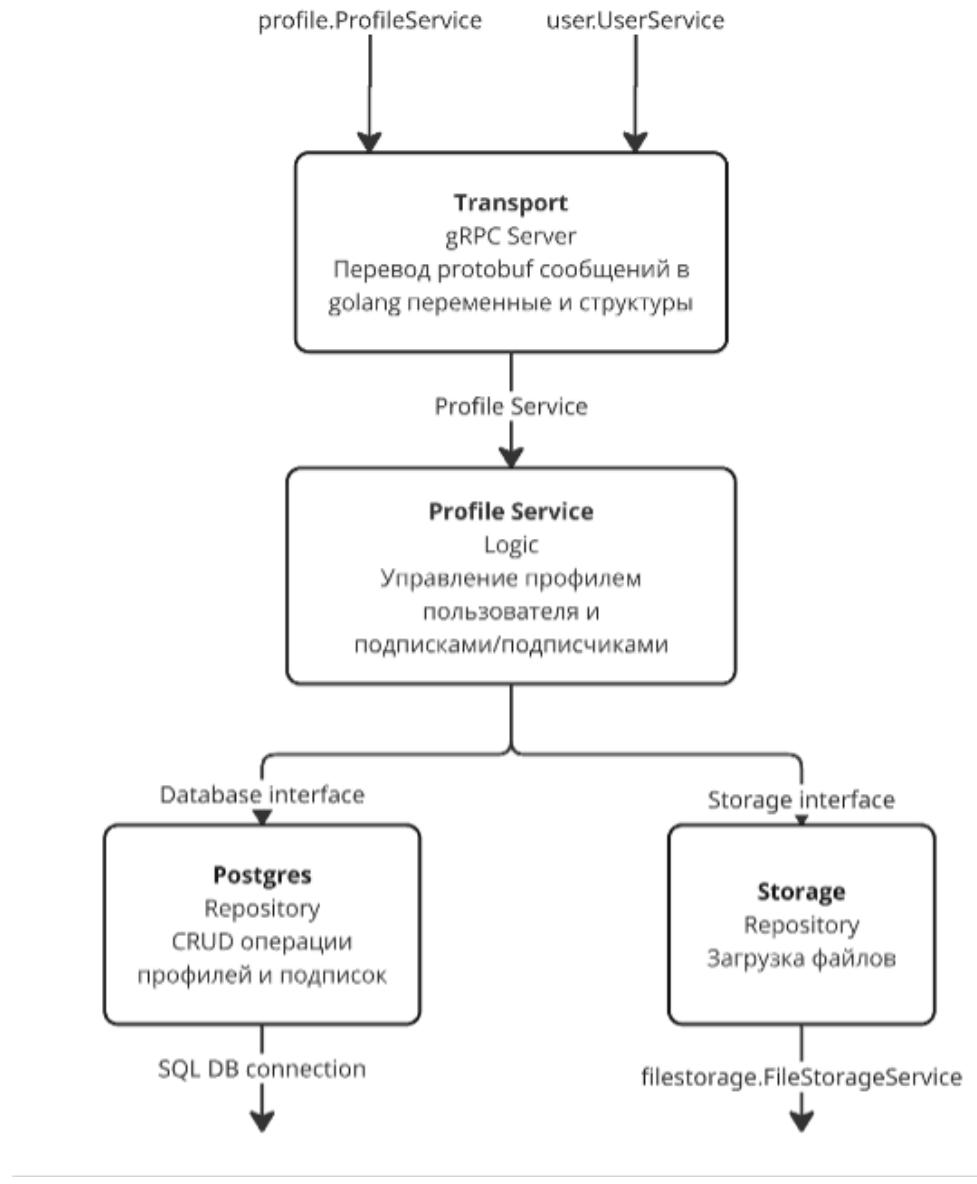


Рис. 4. Компонентная диаграмма Profile сервиса

Activity Service отвечает за обработку действий пользователей, связанных с активностью (лайки и комментарии). В составе сервиса выделя-

ются:

- Transport (gRPC) – прием и обработка gRPC-запросов типа «добавить лайк» или «оставить комментарий»;
- Activity Logic – бизнес-логика, реализующая операции добавления/удаления лайков и комментариев к маршрутам;
- PostgreSQL – база данных, содержащая таблицы route_likes (связка «пользователь–лайк– маршрут») и route_comments (комментарии к маршрутам);
- NATS JetStream – брокер сообщений для публикации событий активности (например, публикация нового комментария может инициировать отправку уведомления другим сервисам).

Сервис Activity обрабатывает сохранение лайков и комментариев в базе данных и передает события об этой активности в систему через NATS для дальнейшей обработки (уведомления, статистика и др.).

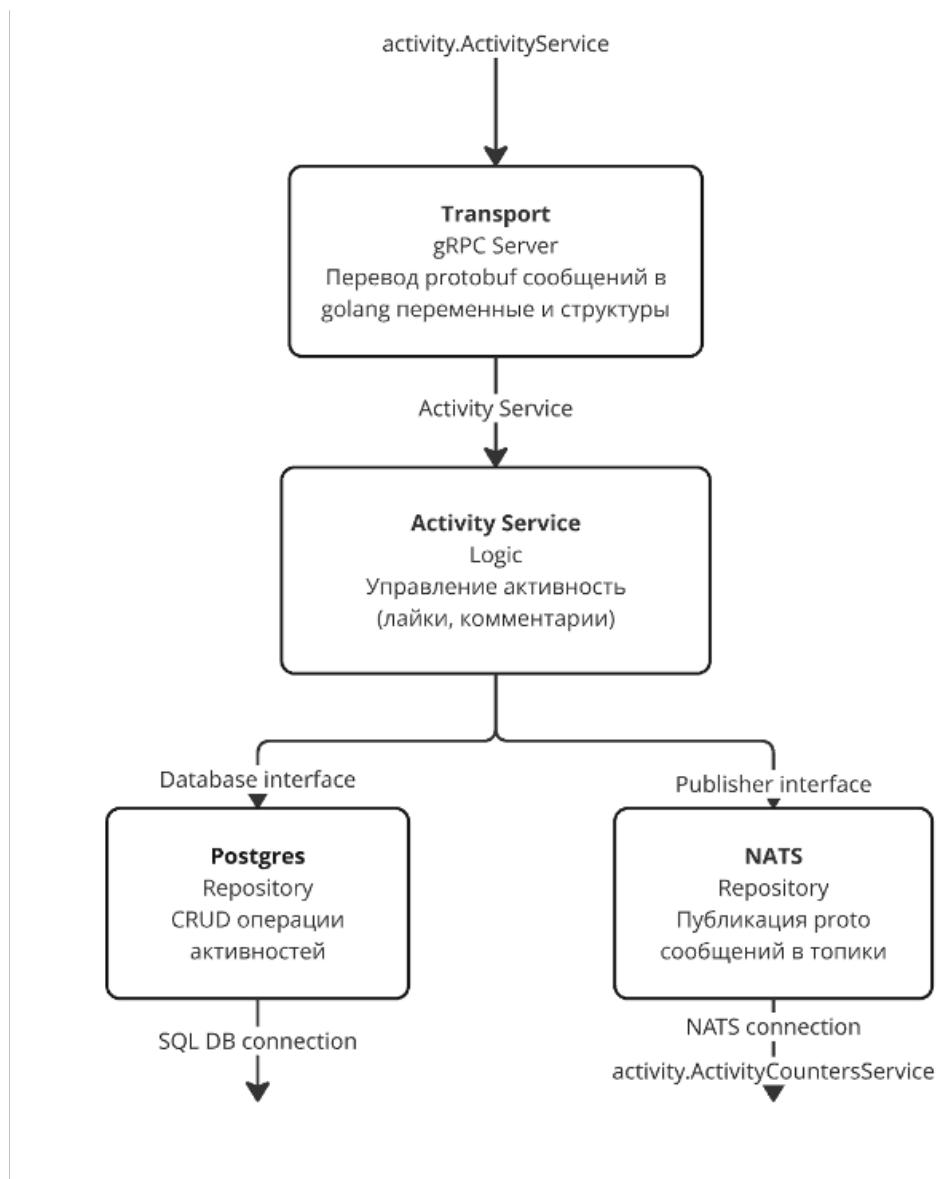


Рис. 5. Компонентная диаграмма Activity сервиса

Content Service отвечает за управление основным контентом приложения: маршрутами и достопримечательностями (местами). Компоненты микросервиса включают:

- **Transport (gRPC)** – интерфейс для приема запросов на создание/чтение/обновление/удаление маршрутов и мест;
- **Content Logic** – бизнес-логика для создания, редактирования и уда-

ления маршрутов (routes) и мест (places), а также управления связями между ними;

- PostgreSQL – база данных с таблицами routes (маршруты), places (достопримечательности), route_places (связующая таблица между routes и places), place_files (связи между places и files).
- FileStorage Client – клиент для взаимодействия с FileStorage Service при загрузке файлов (например, изображений) для мест и маршрутов;
- NATS Listener – слушатель событий из NATS (например, для отслеживания удалений пользователей или обновлений другого контента).

Content Service управляет информацией о туристических маршрутах и местах. Он обеспечивает сохранение метаданных о маршрутах и местах в базе данных и обращение к файловому хранилищу при работе с медиаконтентом.

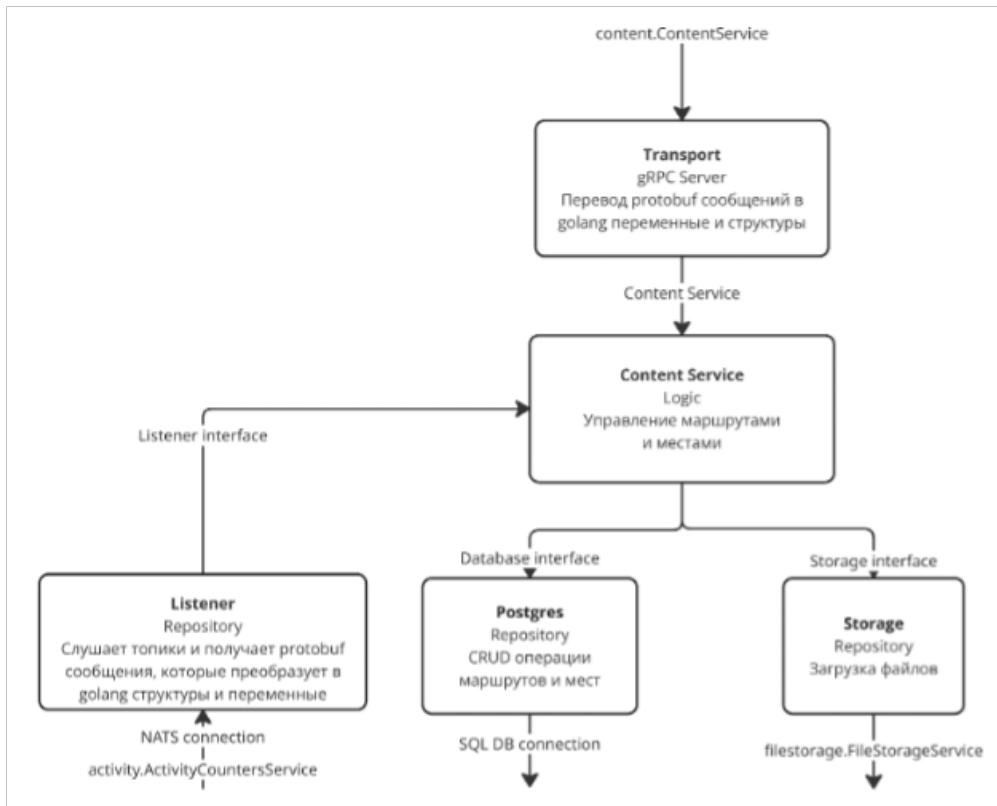


Рис. 6. Компонентная диаграмма Content сервиса

Notifications Service занимается отправкой email-уведомлений пользователям по различным событиям системы. Компоненты сервиса включают:

- Mailer Service – модуль, формирующий и отправляющий электронные письма;
- NATS Listener – подписчик на события из NATS JetStream (например, события о регистрации нового пользователя, новом комментарии и др.), инициирующий отправку уведомления;
- Внешний почтовый сервис (Brevo API) – сторонний сервис доставки писем.

При поступлении события Notifications Service формирует письмо с контентом события и через API внешнего почтового сервиса Brevo отправляет его пользователю.

Таким образом реализована асинхронная отправка email-уведомлений по событиям системы.

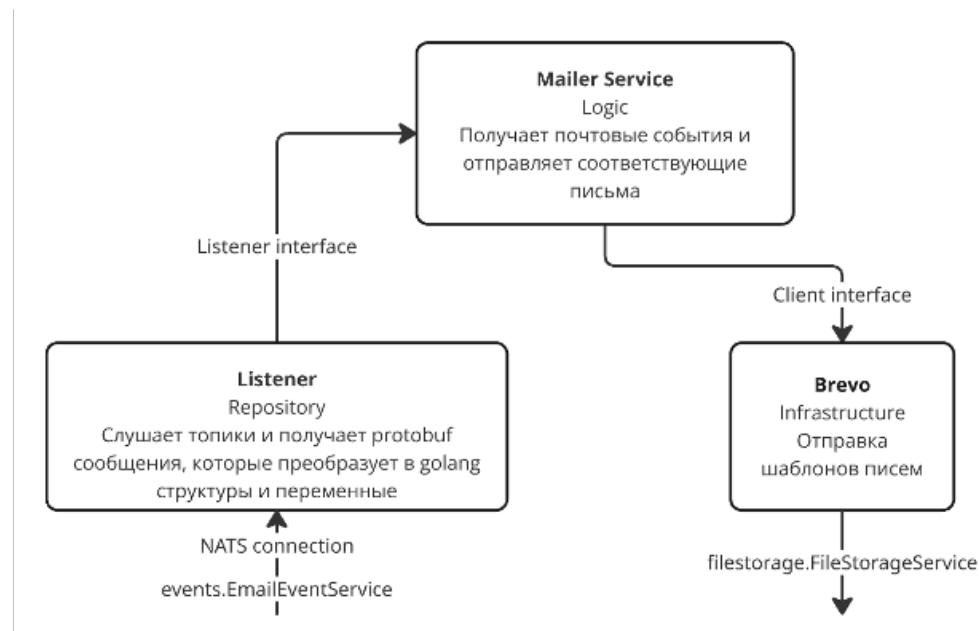


Рис. 7. Компонентная диаграмма Notifications сервиса

FileStorage Service управляет загрузкой и выдачей файлов (например, изображений и других медиа). Его компоненты включают:

- Transport (gRPC) – интерфейс приема запросов на загрузку/скачивание файлов;
- File Logic – бизнес-логика, генерирующая защищенные временные URL (TTL-ссылки) для операций с файлами, а также обрабатывающая метаданные файлов;
- MinIO SDK – библиотека для взаимодействия с S3-совместимым хра-

нилищем MinIO.

При получении запроса на загрузку FileStorage Service создает временную ссылку с ограниченным сроком действия, по которой клиент может напрямую передать файл в MinIO. Сервис также может управлять метаданными файлов (имя, тип, размер и др.), сохраняемыми в базе данных. FileStorage Service обеспечивает безопасное и эффективное хранение любых типов файлов, используемых приложением.

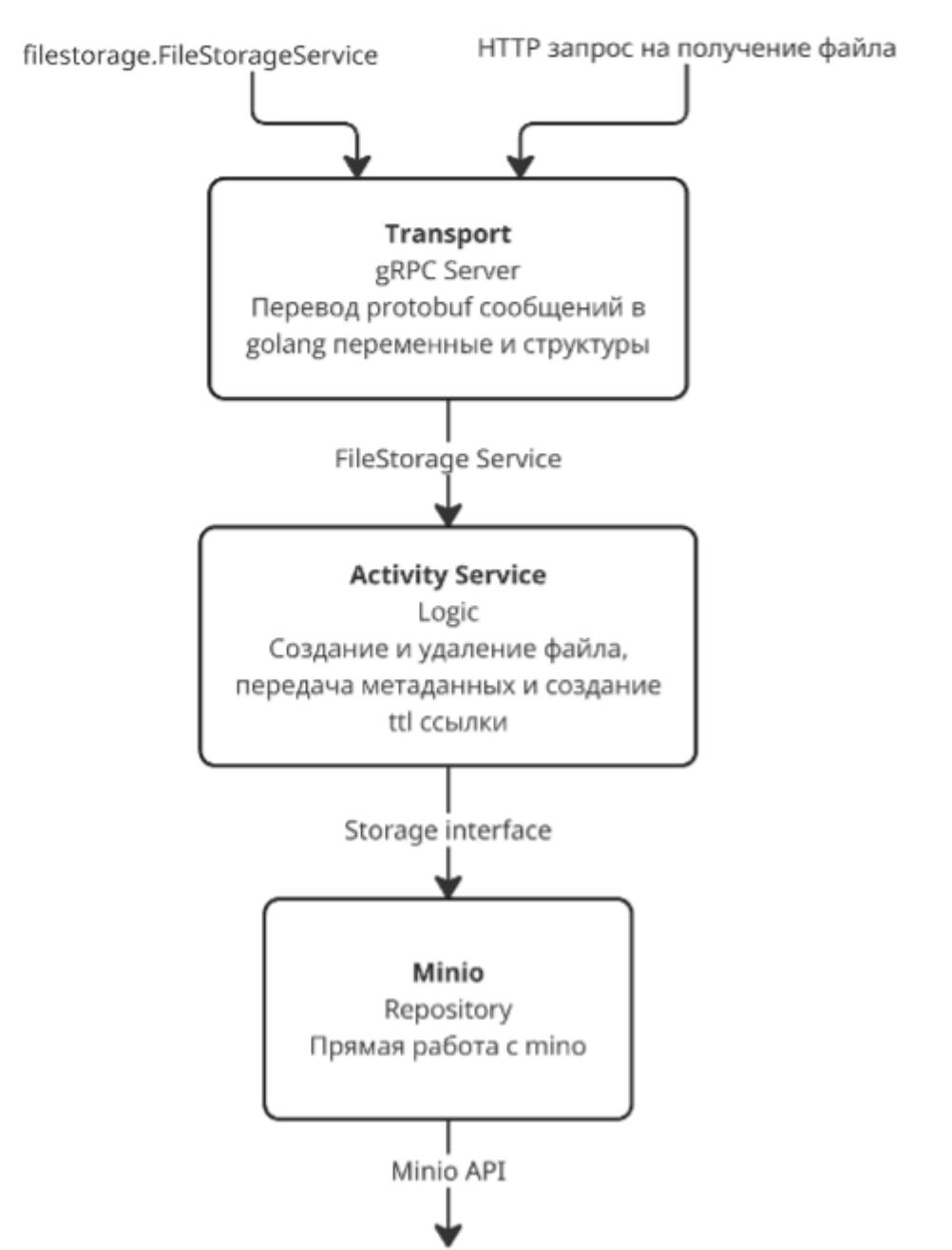


Рис. 8. Компонентная диаграмма FileStorage сервиса

Broker Service служит для организации обмена сообщениями между микросервисами. Его компоненты включают:

- Broker Logic – модуль логики инициализации обмена событиями и логирования;

- Listener (NATS) – слушатель, подписанный на необходимые топики в NATS JetStream;
- NATS JetStream – система обмена сообщениями, где определены темы (topics) для сервиса.

Broker Service отвечает за настройку публикации и маршрутизацию сообщений между сервисами, а также за сбор и логирование событий в системе.

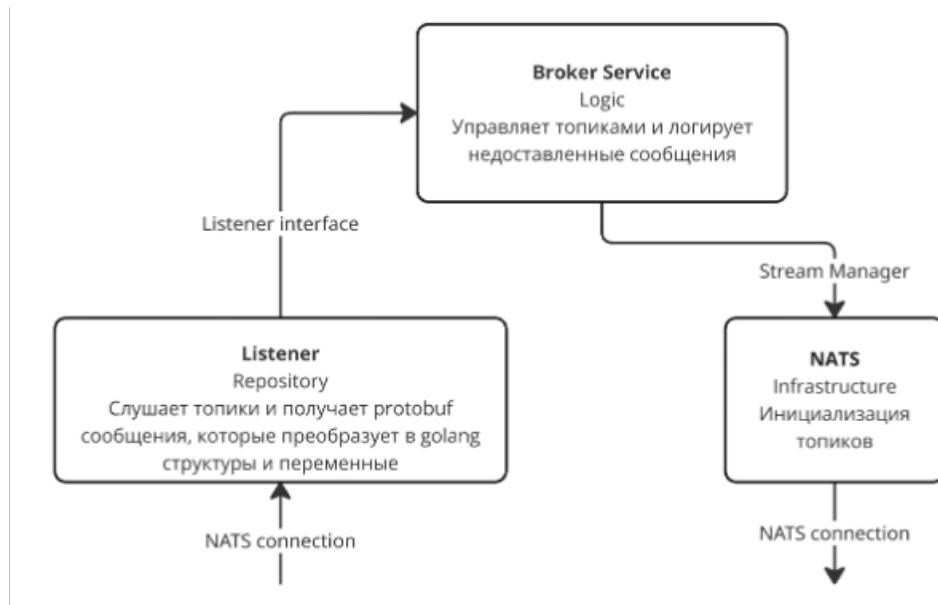


Рис. 9. Компонентная диаграмма Broker сервиса

ERD-диаграммы баз данных

Auth Service: Сервис аутентификации управляет учётными записями пользователей, обеспечивая регистрацию и авторизацию. В ERD этой системы имеется одна таблица `users` для хранения базовой информации об учётных записях:

- `email` (text) – адрес электронной почты пользователя; используется для идентификации учётной записи при авторизации. Обычно это

логин пользователя в системе.

- `password` (text) – хеш пароля пользователя; хранится в зашифрованном виде для последующей проверки.
- `status` (`user_status`) – текущий статус учётной записи; специальный тип (перечисление), задающий состояние пользователя (например, `ACTIVE`, `SUSPENDED` и т. п.). Позволяет пометить аккаунт как активный, заблокированный или ожидающий подтверждения.
- `created_at` (`timestamp with time zone`) – дата и время создания учётной записи; заполняется автоматически при регистрации пользователя в системе.
- `updated_at` (`timestamp with time zone`) – дата и время последнего обновления записи; обновляется при изменении данных пользователя.
- `id` (`uuid`, PRIMARY KEY) – глобально уникальный идентификатор пользователя; первичный ключ таблицы. Используется для однозначной связи пользователя между всеми сервисами системы.

Стоит отметить, что `uuid` применяется во всех таблицах нашей системы в качестве глобального уникального идентификатора. Например, поле `id` в таблице `users` имеет тип `uuid` и связывает данные из этого сервиса с другими микросервисами через единый идентификатор пользователя.

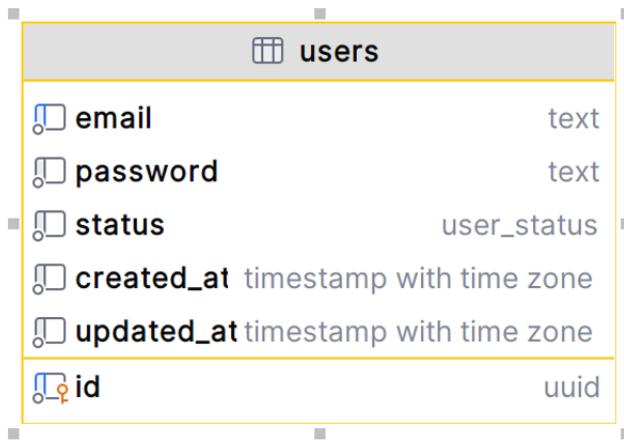


Рис. 10. ERD Auth сервиса

Profile Service: Сервис профилей хранит дополнительную информацию о пользователе, не связанную непосредственно с учётными данными, а также информацию о связях «подписчиков-подписок». ERD этого сервиса включает следующие таблицы:

Таблица profiles содержит персональные данные пользователя и его статистику подписок:

- username (text) – уникальное имя пользователя в системе (никнейм); служит для отображения в интерфейсе и может использоваться при поиске.
- first_name (text) – имя пользователя; хранится для отображения полного имени в профиле.
- last_name (text) – фамилия пользователя; аналогично отображается в профиле.

- `date_of_birth` (`date`) – дата рождения; может использоваться для определения возраста и персонализации.
- `followers_count` (`integer`) – количество подписчиков (`followers`) данного профиля; поле для быстрого доступа к этой статистике.
- `image_id` (`integer, FOREIGN KEY`) – ссылка на файл с аватаром пользователя; внешний ключ, ссылающийся на поле `id` таблицы `files` (сервиса `FileStorage`). Хранит идентификатор файла-аватара в файловом хранилище.
- `created_at` (`timestamp with time zone`) – дата и время создания профиля (обычно совпадает с регистрацией); автоматически задаётся при инициации профиля.
- `updated_at` (`timestamp with time zone`) – дата и время последнего обновления профиля (например, после изменения имени или фото).
- `following_count` (`integer`) – количество пользователей, на которых подписан данный пользователь; поле для быстрого отображения статистики «ПОДПИСОК».
- `user_id` (`uuid, PRIMARY KEY, FOREIGN KEY`) – глобально уникальный идентификатор пользователя; первичный ключ таблицы `profiles`. Кроме того, это внешний ключ на таблицу `users` сервиса `Auth` и указывает на соответствующую учётную запись пользователя.

Таблица `follows` фиксирует отношения подписки между пользователями («кто на кого подписался»):

- `created_at` (`timestamp with time zone`) – дата и время установления подписки; автоматически задаётся при создании записи подписки.
- `followed_id` (`uuid`, PRIMARY KEY (совместно с `follower_id`), FOREIGN KEY) – идентификатор пользователя, на которого осуществлена подписка. Является частью составного первичного ключа и внешним ключом на поле `user_id` таблицы `profiles` (или `users` сервиса Auth).
- `follower_id` (`uuid`, PRIMARY KEY (совместно с `followed_id`), FOREIGN KEY) – идентификатор пользователя, совершившего подписку (подписчика). Также часть составного первичного ключа и внешний ключ на `user_id` таблицы `profiles`.

Таким образом, составной ключ (`follower_id`, `followed_id`) гарантирует уникальность каждой пары «подписчик–подписка», а оба поля ссылаются на записи пользователей. Данная таблица не содержит отдельного идентификатора (ID), так как уникальность обеспечивается комбинацией этих двух полей.

Основные поля таблицы `files` следующие:

- `file_name` (`text`) – исходное имя файла при загрузке. Сохраняется для информационных целей (например, отображение названия).
- `storage_bucket` (`text`) – имя хранилища или «бакета» в облачном хранилище (например, S3), где находится файл.
- `storage_id` (`text`) – внутренний идентификатор файла в хранилище (например, ключ объекта); используется для извлечения файла из облака.

- `internal_url` (text) – сгенерированный URL или путь для доступа к файлу внутри системы (например, через CDN или сервер файлов); может использоваться для получения файла.
- `placeholder` (text) – данные-заполнитель или URL для превью/миниатюры файла; может содержать ссылку на уменьшенное изображение или другой вспомогательный контент.
- `created_at` (timestamp with time zone) – дата и время добавления файла в систему; автоматически фиксируется при загрузке файла.
- `from_public` (boolean) – флаг, указывающий, был ли файл получен из публичного источника. Например, `true` – файл взят из публичной ссылки, `false` – загружен через приложение.
- `size` (bigint) – размер файла в байтах; хранится для контроля и статистики использования пространства.
- `id` (integer, PRIMARY KEY) – уникальный идентификатор файла; первичный ключ таблицы `files`.

Таблица `files` используется не только для хранения изображений (аватаров, фото мест и т. д.), но и для любых других файлов, необходимых системе.

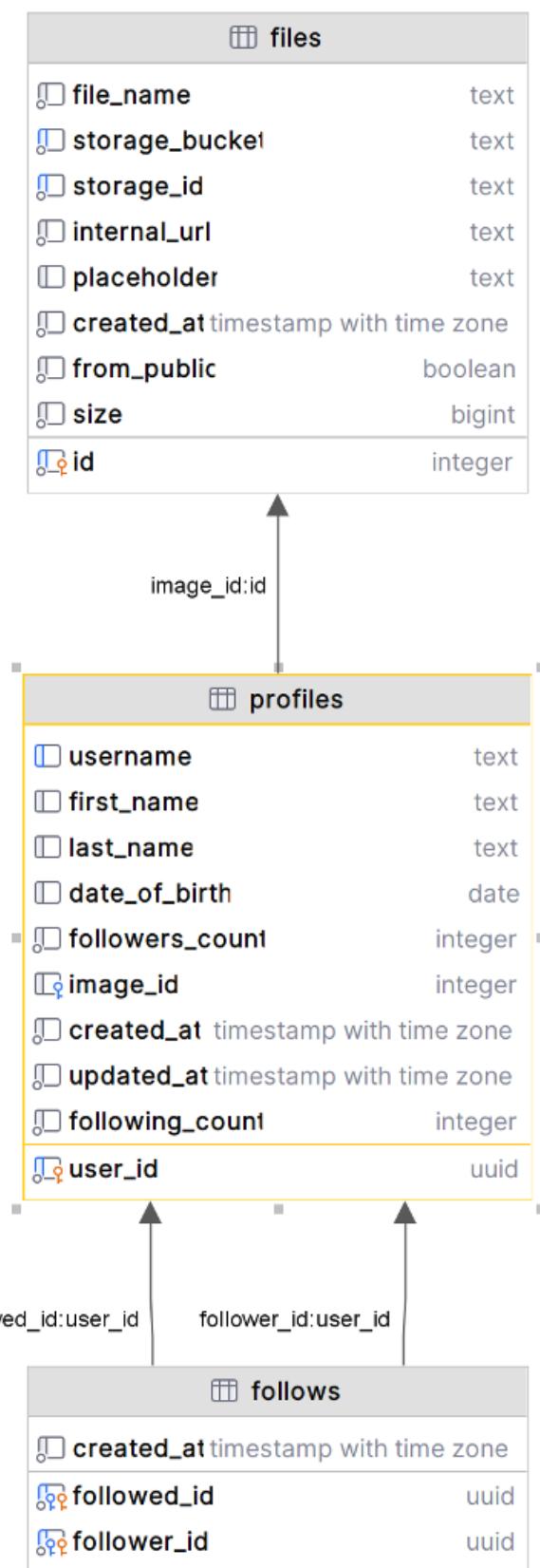


Рис. 11. ERD Profile сервиса
91

Content Service: Сервис контента управляет информацией о туристических маршрутах и местах. ERD этого сервиса состоит из следующих таблиц:

Таблица routes содержит описание путешествий-туров:

- name (text) – название маршрута; краткое описание, которое видит пользователь.
- difficulty (difficulty_level) – уровень сложности маршрута; специальный перечислимый тип (enum), задающий категорию сложности (например, EASY, MODERATE, HARD).
- distance (real) – протяжённость маршрута (например, в километрах); числовое поле с плавающей точкой.
- user_id (uuid, FOREIGN KEY) – идентификатор пользователя, создавшего маршрут;
внешний ключ на таблицу users сервиса Auth (или profiles.user_id).
Позволяет связать маршрут с автором.
- latitude (double precision[]) – массив значений широт точек маршрута; используется для хранения многоугольника пути (географические координаты всех точек маршрута).
- longitude (double precision[]) – массив долготы точек маршрута; параллельно latitude описывает путь.
- created_at (timestamp with time zone) – дата и время создания маршрута; автоматически присваивается при добавлении нового маршрута.

- `updated_at` (timestamp with time zone) – дата и время последнего изменения маршрута (например, после редактирования описания или пути).
- `description` (text) – подробное текстовое описание маршрута; может включать рекомендации, обзор мест.
- `id` (uuid, PRIMARY KEY) – глобально уникальный идентификатор маршрута; первичный ключ таблицы `routes`.

Таблица `places` содержит информацию о конкретных местах (точках интереса):

- `name` (text) – название места (например, название достопримечательности или географического объекта).
- `address` (text) – адрес или описание местоположения места (улица, город и т. п.).
- `description` (text) – детальное описание места; история, примечания, другая справочная информация.
- `latitude` (double precision) – широта географического расположения места (одиночная координата).
- `longitude` (double precision) – долгота географического расположения места.
- `created_at` (timestamp with time zone) – дата и время добавления записи о месте; автоматически заполняется при создании.

- `updated_at` (`timestamp with time zone`) – дата и время последнего обновления данных места.
- `id` (`uuid`, PRIMARY KEY) – глобально уникальный идентификатор места; первичный ключ таблицы `places`.

Таблица `route_places` связывает маршруты и места, формируя последовательность точек маршрута:

- `route_id` (`uuid`, FOREIGN KEY) – идентификатор маршрута; внешний ключ на `id` таблицы `routes`.
- `place_id` (`uuid`, FOREIGN KEY) – идентификатор места; внешний ключ на `id` таблицы `places`.
- `ordering` (`integer`) – порядок (позиция) места в данном маршруте; указывает, в каком порядке пользователь посещает эти места.
- `id` (`integer`, PRIMARY KEY) – уникальный идентификатор записи; первичный ключ таблицы.

Данная таблица реализует связь «многие-ко-многим» между маршрутами и местами, причём через поле `ordering` поддерживается упорядоченность. Каждая запись указывает, что конкретное место входит в маршрут.

Таблица `place_files` хранит связь между местами и файлами (например, фотографиями мест):

- `ordering` (`integer`) – порядок файла в списке файлов для данного места; помогает упорядочить изображения.
- `place_id` (`uuid`, FOREIGN KEY) – идентификатор места; внешний ключ на `id` таблицы `places`.

- file_id (integer, FOREIGN KEY) – идентификатор файла; внешний ключ на id таблицы files сервиса FileStorage.
- id (integer, PRIMARY KEY) – уникальный идентификатор записи; первичный ключ таблицы place_files.

Таким образом, таблица place_files связывает множество файлов с каждым местом, позволяя прикреплять к месту изображения и другие медиафайлы. Для каждого связанного файла хранится его сортировочный индекс (ordering) и ссылки на место и файл. Таблица files аналогична одноименной таблице из Profile сервиса.

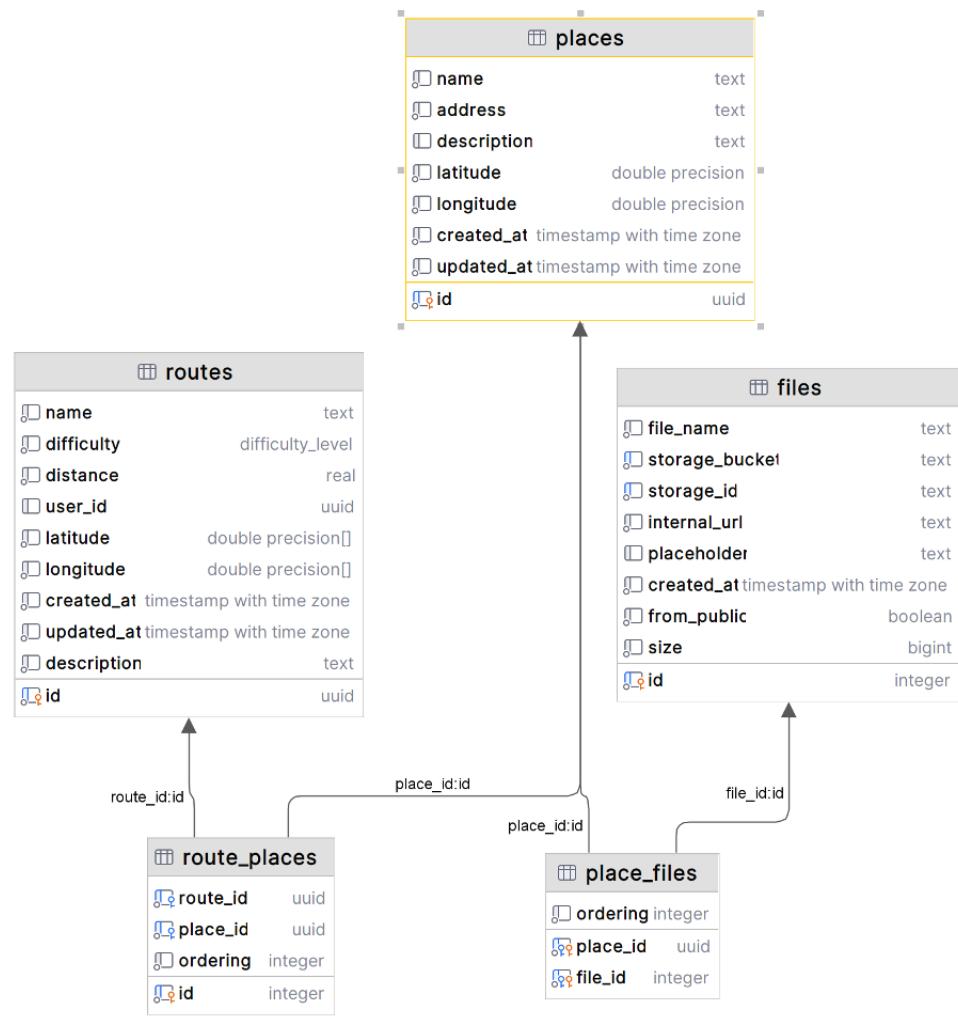


Рис. 12. ERD Content сервиса

Activity Service: Сервис активности отвечает за пользовательские взаимодействия с маршрутами – лайки и комментарии. ERD этого сервиса включает следующие таблицы: Таблица route_likes фиксирует факт постановки лайка маршруту:

- created_at (timestamp with time zone) – дата и время проставления лайка; автоматически устанавливается при создании записи.
- route_id (uuid, PRIMARY KEY (вместе с user_id), FOREIGN KEY) – идентификатор маршрута, которому поставлен лайк; внешний ключ на id

таблицы routes. Является частью составного первичного ключа вместе с user_id.

- user_id (uuid, PRIMARY KEY (вместе с route_id), FOREIGN KEY) – идентификатор пользователя, поставившего лайк; внешний ключ на таблицу users (или profiles.user_id). Является частью составного первичного ключа.

Комбинация полей (route_id, user_id) образует составной первичный ключ, что гарантирует, что каждый пользователь может поставить лайк заданному маршруту не более одного раза. Таким образом эта таблица не имеет отдельного ID.

Таблица route_comments хранит комментарии пользователей к маршрутам:

- route_id (uuid, FOREIGN KEY) – идентификатор маршрута, к которому оставлен комментарий; внешний ключ на id таблицы routes.
- user_id (uuid, FOREIGN KEY) – идентификатор пользователя, написавшего комментарий; внешний ключ на таблицу users.
- text (text) – текст комментария.
- created_at (timestamp with time zone) – дата и время публикации комментария.
- comment_id (bigint, PRIMARY KEY) – уникальный идентификатор комментария; первичный ключ таблицы route_comments.

Поле `comment_id` является автоинкрементным или генерируется последовательно и гарантирует уникальность комментария. Остальные поля связывают комментарий с маршрутом и пользователем.

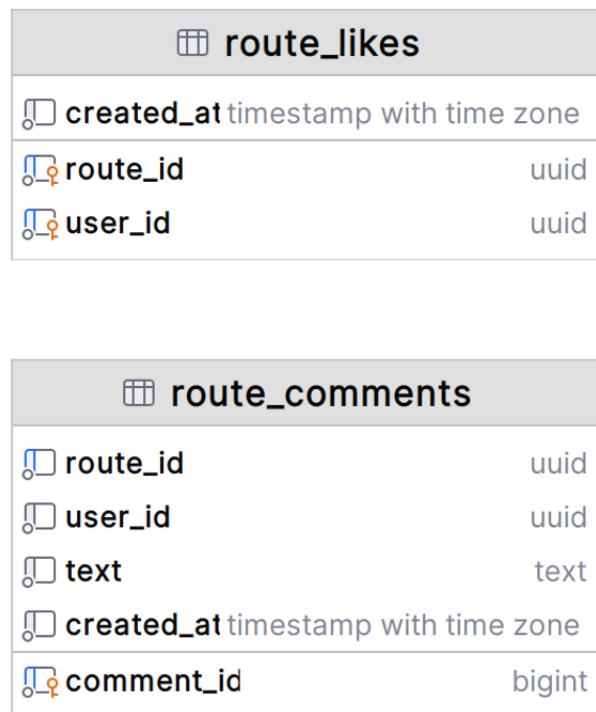


Рис. 13. ERD Activity сервиса

2.4 Компоненты инфраструктуры

Организация инфраструктуры и DevOps-процессов является важной частью проектирования микросервисной системы приложения «Путешествия по России». На этапе проектирования проекта принимаются решения о средствах автоматизации сборки, тестирования и развертывания, а также о способах управления артефактами и конфигурациями. В данном разделе описываются основные компоненты инфраструктуры: процесс CI/CD на основе GitHub Actions, использование GitHub Container Registry для хранения

нения артефактов, система Helm-чартов для управления развертыванием, балансировка нагрузки через Envoy и подходы к хранению конфигураций и секретов.

CI/CD-процесс на основе GitHub Actions

Для обеспечения непрерывной интеграции и доставки (CI/CD) в микросервисной архитектуре приложения «Путешествия по России» на этапе проектирования предусматривается использование платформы GitHub Actions. Проект состоит из множества независимых репозиториев, соответствующих отдельным микросервисам (например, vkr-auth, vkr-notifications, vkr-profile и др.). В каждом репозитории настраивается собственный workflow в GitHub Actions, реализующий этапы сборки и доставки сервисов. При каждом пуше в ветку main запускается автоматизированный процесс сборки: если изменяется код микросервиса, запускается создание Docker-образа этого сервиса и его публикация в GitHub Container Registry (GCR); если меняются компоненты платформы, инициируется сборка обновлённого Helm-чарта. Предусматривается отдельный триггер для миграций базы данных: при изменении файлов миграции запускается выполнение соответствующего workflow для применения новых изменений в структуре БД. Таким образом, внедрение GitHub Actions обеспечивает автоматическое прохождение всего цикла разработки – от фиксации изменений в репозитории до развертывания обновлённой версии приложения. Данный подход исключает необходимость ручного развёртывания и ускоряет выпуск новых версий. В частности, после успешного пуша в главную ветку репозитория vkr-platform, содержащего интегрированный Helm-чарт всех сервисов, в продакшн-окружении автоматически запускается обновление при-

ложений до новой версии.

GitHub Container Registry (GHCR)

В качестве централизованного хранилища контейнерных образов и Helm-чартов в проекте предполагается использовать GitHub Container Registry (GHCR) под личной учётной записью pedrecho. Все образы сервисов и сформированные чарты публикуются в этот регистр, что позволяет удобно управлять версиями артефактов и обеспечивать их доступность при развертывании. Каждый образ или чарт получает уникальный тег, отражающий версию и дату сборки (например, для чарта используется префикс chart-), при этом предыдущие версии не удаляются из регистра во избежание потери истории и возможности отката. Для аутентификации при публикации и загрузке артефактов в GitHub Container Registry в CI/CD-пайплайнах используются защищённые переменные окружения: GHCR_USERNAME, GHCR_TOKEN (токен доступа к регистру) и GOPRIVATE_PAT (личный токен доступа для приватных Go-модулей). Использование централизованного регистра упрощает управление зависимостями и обеспечивает постоянный доступ к нужным версиям контейнеров и чартов в процессе развертывания.

Helm Charts для управления развертываниями

Для каждого микросервиса разрабатывается собственный Helm-чарт, описывающий его шаблоны развёртывания, сервисы и другие необходимые Kubernetes-ресурсов. В проекте vkr-platform создаётся объединённый («собирательный») Helm-чарт, в котором в качестве зависимостей перечислены все чарты микросервисов и шаблон Envoy, выступающий в роли

маршрутизатора. Такой подход позволяет централизовать описание всей системы и обеспечить согласованность версий сервисов при развертывании. В процессе CI/CD каждый Helm-чарт упаковывается в архив (файл .tgz), затем включается в контейнерный образ или публикуется в реестр артефактов с помощью GitHub Actions. Публикация чарта позволяет Kubernetes-кластеру автоматически получать актуальные сведения о конфигурации сервисов. Это обеспечивает удобную дистрибуцию чартов и автоматическую доставку обновлённых версий конфигураций кластеру при каждой сборке проекта.

Балансировка нагрузки и маршрутизация трафика

В качестве внешнего прокси-сервера и балансировщика нагрузки для gRPC-трафика от мобильного приложения к микросервисам используется Envoy. Все входящие запросы от клиента маршрутизируются через единый контейнер Envoy, что упрощает управление коммуникацией между сервисами и позволяет централизованно настраивать правила маршрутизации. Envoy развёртывается как часть платформенного Helm-чарта и размещается в том же

namespace, что и остальные сервисы приложения. Конфигурация Envoy задаётся с помощью шаблонов Helm и генерируется на основе значений из файлов values.yaml. Такой подход позволяет адаптировать параметры прокси-сервера под различные окружения и случаи использования без изменения исходного кода. При этом система изначально не предполагает использование TLS для межсервисного взаимодействия, поэтому соответствующие настройки шифрования в данном разделе не рассматриваются.

Хранение конфигураций и секретов

Управление конфигурацией системы осуществляется через файлы Helm-чартов `values.yaml` и `values.secret.yaml`. В файле `values.yaml` задаются общие параметры для каждого сервиса, включая настройки подключения, порты и т.д. Файл `values.secret.yaml` предназначен для конфиденциальных данных (пароли, токены и т.п.), поэтому он не хранится в системе контроля версий. Вместо этого в репозитории сохраняется шаблон `values.secret.example.yaml`, который иллюстрирует необходимые поля для секретной конфигурации без реальных значений. На этапе деплоя значения из `values.yaml` и `values.secret.yaml` компонуются, а соответствующие параметры записываются в объекты `ConfigMap` и `Secret` Kubernetes [6][7]. В Go-приложениях используется единый YAML-файл конфигурации, который формируется из этих значений: переменные окружения подставляются в шаблон Go-конфига на основе собранных данных Helm-чартов. Таким образом, каждый сервис получает все необходимые настройки из одного объединённого YAML-файла, обеспечивая унифицированный и прозрачный механизм конфигурации приложений.

2.5 Меры обеспечения безопасности

OAuth 2.0 и Auth Gateway

Принята модель авторизации OAuth 2.0. При этом выдаётся два вида токенов:

- Access-токен (JWT) [8] с коротким временем жизни (около 15 минут).

- Refresh-токен (UUID) со значительно более длительным временем жизни (около 7 дней).

Для повышения безопасности при выдаче токенов учитывается уникальный «отпечаток» (fingerprint) устройства или браузера пользователя, что привязывает токен к конкретному клиенту. Проверка валидности токенов и их обновление централизованы в сервисе AuthService: у него имеется специальный gRPC-эндпоинт для проверки токенов. Такой подход позволяет гарантировать консистентность управления токенами в системе и повышает надёжность авторизации.

Безопасность межсервисного взаимодействия

Коммуникация между микросервисами осуществляется по внутренним защищённым каналам (через протокол gRPC), проксируемым Envoy. Envoy выступает в роли «data plane» для микросервисов, централизованно управляя трафиком и повышая безопасность сетевого взаимодействия. Кроме того, используются механизмы сегментации сети: сервисы развернуты в отдельных namespace Kubernetes, а маршруты между ними ограничены сетевыми политиками (Network Policy). Это минимизирует «боковое» распространение атаки и снижает зону потенциального поражения в случае компрометации отдельного компонента.

Шифрование и безопасное хранение данных

При работе с учётными данными пользователей пароли никогда не сохраняются в открытом виде: перед записью в базу они хэшируются с помощью алгоритма bcrypt. Остальные чувствительные данные пользователе-

лей не шифруются явно на уровне приложения, но передаются по защищённым внутренним каналам (TLS внутри кластера) и хранятся в приватных базах данных, доступ к которым строго ограничен. Защита данных на уровне хранилища обеспечивается использованием защищённых томов (PVC). Современные облачные платформы обычно поддерживают автоматическое шифрование томов на уровне хоста или инфраструктуры: например, в GKE on Azure тома данных по умолчанию шифруются платформенными ключами (Azure Key Vault). Это гарантирует, что даже при физическом доступе к носителю данные останутся зашифрованными.

Безопасность файлового хранилища

Хранилище объектов MinIO настроено с разграничением доступа: один публичный бакет используется для общедоступных ресурсов (аватары пользователей, изображения маршрутов и т.п.), тогда как все прочие файлы хранятся в приватных бакетах. Доступ к приватным файлам возможен только по заранее сгенерированным временным URL (pre-signed URL) с ограниченным сроком действия. Такие ссылки действуют строго ограниченное время – после истечения TTL доступ по ним автоматически прекращается. Таким образом, даже при потенциальной утечке ссылки её сила быстро аннулируется, что значительно повышает безопасность файлового хранилища.

Резервное копирование и аудит

Предусмотрено регулярное резервное копирование конфигураций и данных. В частности, современные облачные СУБД и сервисы поддержива-

ют автоматическое ежедневное создание резервных копий – например, в Cloud SQL для MySQL ежедневно выполняется автоматическая резервная копия данных. Хотя конкретный провайдер на этапе проектирования ещё не выбран, в большинстве решений можно настроить автоматическое хранение истории бэкапов с возможностью восстановления. Аудит безопасности реализуется через сбор системных логов на уровне Kubernetes-подов и сервисов. В текущей архитектуре отдельного агрегатора логов нет, но логи формируются последовательно и структурировано, что позволяет в будущем легко подключить централизованные системы мониторинга и аудита (Grafana Loki, ELK и т.п.) для глубокого анализа событий и инцидентов.

Выводы по главе

Проведённый во второй главе анализ и проектирование подтвердили обоснованность выбора микросервисной архитектуры как базового подхода к построению серверной части приложения «Путешествия по России». Предложенная структура – разделение функциональности на независимые сервисы Auth, Profile, Content, Activity, FileStorage и Notifications, взаимодействующие через gRPC и асинхронный брокер NATS JetStream, – обеспечивает требуемые нефункциональные характеристики: масштабируемость (горизонтальное масштабирование отдельных узких мест), отказоустойчивость (локализация сбоев и автоматический перезапуск контейнеров) и гибкость развития (возможность добавления новых доменных сервисов без переработки существующего кода). Детализированные контекстная, контейнерная и компонентные диаграммы визуально подтвердили согласованность слоёв, а ER модели баз данных показали целостность схем хра-

нения, поддерживающих концепцию «база на сервис» и разграничение ответственности за данные. Важной частью проекта стало описание DevOps инфраструктуры и мер безопасности: автоматизированные CI/CD конвейеры на GitHub Actions, хранение артефактов в GHCR, единый Helm чарт «platform» для развёртывания, балансировка и маршрутизация трафика через Envoy, а также комплекс защитных механизмов (OAuth 2.0, bcrypt хеширование паролей, параметризованные SQL запросы, TTL ссылки на файлы, сегментация сети внутри кластера). Принятые решения гарантируют последовательную доставку изменений в Kubernetes кластер, безопасное хранение чувствительных данных и строгий контроль доступа к ресурсам. Таким образом, во второй главе сформирована целостная и технически обоснованная архитектура, которая служит прочным фундаментом для реализации и последующей эксплуатации системы, соответствующая как функциональным, так и нефункциональным требованиям, выявленным на предыдущем этапе работы.

ГЛАВА 3 РЕАЛИЗАЦИЯ СЕРВИСНОЙ ЧАСТИ

3.1 Технологический стек

Go 1.24.0 единообразие в микросервисах

Go 1.24.0 выбран в качестве основного языка разработки для всех микросервисов системы. Это решение обеспечивает простоту сопровождения: Go имеет лаконичный и выразительный синтаксис, строгую статическую типизацию и стандартные инструменты форматирования (например, `gofmt`), что упрощает чтение и поддержку кода. Кроме того, Go предоставляет эффективное управление памятью и масштабируемую модель конкурентности. Легковесные горутины и оптимизированный сборщик мусора позволяют сервисам обрабатывать множество одновременных запросов без существенного роста потребления ресурсов. Единый язык и стек библиотек упрощают обучение новой команды и стандартизацию практик разработки.

- Детерминированность сборки: Механизм модулей Go (`go.mod` и `go.sum`) строго фиксирует версии всех зависимостей и их контрольные суммы. Это гарантирует, что сборки на локальных машинах, в CI/CD и на продакшн-среде будут идентичными, исключая «эффект снежинки». Благодаря этому на каждом этапе конвейера мы получаем точно известный набор библиотек, что повышает воспроизводимость и стабильность системы.
- Контроль качества кода: для поддержки единообразия стиля и обнаружения ошибок применяется статический анализатор `golangci-lint`.

`lint`. Этот инструмент запускает множество линтеров параллельно (проверка форматирования, выявление забытых проверок ошибок, поиск потенциальных утечек и др.) и интегрируется в процесс CI. Интеграция `golangci-lint` позволяет на раннем этапе обнаруживать дефекты и поддерживать единый стандарт качества кода.

gRPC и отказ от REST

Межсервисное взаимодействие организовано с использованием gRPC, что обеспечивает бинарную сериализацию данных (*Protocol Buffers*) и строгое определение контрактов. В отличие от REST, gRPC передаёт сообщения в компактном бинарном формате по протоколу HTTP/2, что снижает сетевые задержки и уменьшает объём передаваемых данных. Определение интерфейсов и типов сообщений в файлах `.proto` задаёт жёсткую схему запросов и ответов, благодаря чему генерируемый Go-код точно соответствует контракту. Это позволяет компилятору выявлять несоответствия на этапе сборки и предотвращает ошибки интерфейса между сервисами.

- Бинарная сериализация и эффективность: gRPC использует HTTP/2 и *Protocol Buffers*, что даёт более компактные сообщения и эффективную сериализацию по сравнению с текстовым JSON. Меньший объём данных и мультиплексирование запросов на одном соединении приводят к низкой задержке и высокой пропускной способности, особенно при большом числе мелких вызовов.
- Строгие контракты и стабильность API: Определение сервисов в `.proto`-файлах задаёт чёткие схемы запросов и ответов. Для Go применяют-

ся плагины `protoc-gen-go` и `protoc-gen-go-grpc`, автоматически генерирующие типобезопасные структуры, сервисные интерфейсы и клиентские/серверные заглушки. При внесении изменений компилятор быстро выявляет возможные несовместимости, что повышает надёжность и контролируемость версий API.

- Двунаправленные стримы: gRPC поддерживает как односторонние, так и двунаправленные потоки данных. Это позволяет серверам и клиентам обмениваться сообщениями в режиме реального времени и передавать данные по мере их готовности, без необходимости дожидаться завершения каждого запроса. Такой подход полезен для реализации непрерывных событийных подписок и долгоживающих потоковых операций.
- Отказ от REST: gRPC выбран вместо REST ввиду строгого подхода к описанию интерфейсов и высокой производительности. REST с JSON-представлением создаёт дополнительные расходы на сериализацию и десериализацию и требует ручной валидации данных, что увеличивает задержки при большом числе запросов. В тесно интегрированной микросервисной среде эти накладные расходы могут стать узким местом, тогда как строго типизированный и генерируемый gRPC обеспечивает масштабируемость и упрощает сопровождение системы.

Логирование и конфигурация

Интерфейс логгера основан на библиотеке `zap` (Uber) и обёрнут в собственный модуль. Такой подход обеспечивает высокую производитель-

ность и структурированный вывод логов (каждое сообщение содержит уровень, метку времени и дополнительные поля). Ротация логов организована с помощью пакета `lumberjack`: при достижении заданного размера или срока хранения текущий файл переименовывается и создаётся новый, что предотвращает бесконтрольный рост логов. Логи записываются одновременно в стандартный поток `stdout` (важно для контейнеризированной среды) и в файлы, что упрощает как оперативный мониторинг (например, сбор логов через оркестратор), так и долговременное хранение журналов.

- Обёртка над `zap`: высокопроизводительное и структурированное логирование с возможностью добавления метаданных (уровень, модуль, временная метка и т. д.).
- `Lumberjack`: пакет для ротации лог-файлов по размеру и времени (архивирование устаревших файлов).
- Одновременный вывод: запись логов как в `stdout` (для сбора логов контейнеризатором), так и в файл (для архивации и анализа).

Конфигурация сервисов хранится в YAML-файлах и загружается с помощью `gorkg.in/yaml.v3`. Формат YAML выбран из-за его читабельности и поддержки вложенных структур и комментариев. Библиотека `yaml.v3` десериализует данные конфигурации (например, параметры баз данных, порты, флаги) прямо в структуры Go (через теги `yaml`), что обеспечивает строгую типизацию и валидацию. Благодаря единому YAML-файлу все параметры сосредоточены в одном месте, легко читаются и изменяются без сложной сериализации или форматирования.

- Формат YAML: человекочитаемый, поддерживает вложенность и комментарии, упрощает ручную правку конфигов.

- `gopkg.in/yaml.v3`: преобразует YAML в Go-структуры по тегам `yaml`, обеспечивая строгую проверку и удобный доступ к параметрам.

PostgreSQL и Redis

Для работы с PostgreSQL используется стандартный пакет `database/sql` вместе с драйвером (`pgx` или `pq`). Такой подход даёт полный контроль над SQL-запросами и высокую прозрачность операций: все SQL-команды пишутся вручную, что позволяет точно понимать, какие запросы выполняются, и оптимизировать их при необходимости. Отсутствие ORM снижает «магический» код и упрощает отладку. Транзакции и подготовленные выражения (`PreparedStatement`) позволяют эффективно управлять соединениями и защищать от SQL-инъекций.

- Контроль и прозрачность: `database/sql` предоставляет гибкость и прямой контроль над запросами (сложные JOIN, CTE, индексы и т.д.), упрощая оптимизацию и диагностику производительности.
- Гибкость: без ограничений ORM можно использовать все возможности PostgreSQL; результаты запросовчитываются в типизированные структуры Go.

Схема базы данных поддерживается через миграции: каждая версия представлена парой файлов `*.up.sql` и `*.down.sql`. Файлы `up` содержат команды для применения изменений (создание или изменение таблиц и др.), а `down` – инструкции для отката этих изменений. Миграции нумеруются или метятся по времени, обеспечивая последовательность. Для применения миграций используется утилита `migrate`, часто запускаемая че-

рез Docker. Команда `migrate up` последовательно применяет все новые скрипты, а `migrate down` откатывает последние изменения. Такой механизм гарантирует синхронизацию структуры БД во всех средах и позволяет безопасно возвращаться к предыдущей версии.

`Redis` выполняет роль *in-memory* хранилища для временных данных и кэша. Для ключей устанавливается время жизни (TTL), что удобно для одноразовых или временных объектов: например, SMS-коды верификации, одноразовые токены или кэшированные результаты. По истечении TTL ключ автоматически удаляется, что упрощает управление памятью. Также ключи разделяются префиксами (например, `verification:`, `cache:`, `session:`) для группировки по функционалу и предотвращения коллизий имён. `Redis` отлично подходит для сценариев с высокой частотой чтения и короткоживущих данных, например, хранения кодов верификации или кэширования результатов интенсивных запросов.

- TTL: ключи с ограниченным сроком жизни; подходит для временных данных (верификационных кодов, сессий), которые автоматически удаляются.
- Префиксы ключей: группировка ключей по префиксам (`otp:`, `cache:` и др.) упрощает организацию пространства имён и избегает пересечений.
- Сценарии использования: кэширование часто запрашиваемых данных и хранение одноразовых верификационных кодов или токенов с коротким сроком действия, обеспечивая быстрый доступ и автоматическое истечение.

NATS JetStream и брокер

Обычный NATS – это лёгкий брокер сообщений с поддержкой pub/sub, но без долговременного хранения: сообщения доставляются активным подписчикам мгновенно и затем удаляются (at-most-once). JetStream расширяет возможности NATS, добавляя постоянное хранилище и подтверждение доставки (at-least-once). Сообщения сохраняются на диске в потоках (streams), и потребители должны отправлять подтверждение (Ack). Если сообщение не подтверждено, оно остаётся в потоке и будет доставлено повторно. Благодаря JetStream новые или временно отключённые потребители могут получать пропущенные события, а политика хранения сообщений (сколько хранить и как долго) настраивается централизованно.

- Гарантии доставки: JetStream требует явного подтверждения (Ack) от потребителя, обеспечивая хотя бы однократную доставку каждого сообщения.
- Хранение сообщений: сообщения сохраняются в потоках (Streams) на диске, что позволяет позднее повторно их считывать и добавлять новых подписчиков.
- Durable-подписчики: при использовании долговременных потребителей (Durable Consumers) позиция чтения сохраняется, и при переподключении потребитель продолжает с последнего непрочитанного сообщения.

В проекте для работы с JetStream используется официальный клиент nats.go. Отдельный сервис «Broker» отвечает за настройку потоков и подписок. Этот сервис создаёт необходимые JetStream-стримы с задан-

ными политиками хранения и регистрирует durable-консьюмеры для микросервисов. При публикации Broker отправляет сообщения в JetStream и ожидает подтверждения, а при приёме он отдаёт сообщения подписчикам и фиксирует получение. Такой подход обеспечивает надёжный обмен событиями между микросервисами: даже при сбоях или перезапусках ничего не теряется.

JWT и безопасность

JWT (JSON Web Token) состоит из трёх частей: `header`, `payload` и `signature`. `Header` определяет метаданные токена (например, алгоритм HS256 и тип), `payload` содержит утверждения (`claims`) – стандартные (`iss`, `exp`, `sub`, `iat` и др.) и пользовательские (например, идентификатор пользователя). `Signature` формируется с помощью HMAC SHA-256: это результат применения HMAC к base64-закодированным `header` и `payload` с использованием секретного ключа. Такой механизм обеспечивает целостность: при любой подмене данных подпись не пройдёт проверку.

- `Header`: задаёт алгоритм подписи (HS256) и тип токена (JWT).
- `Payload`: содержит информацию о пользователе и сроке действия (например, поле `exp` с временем истечения).
- `Signature`: HMAC SHA-256 по содержимому `header` и `payload` с секретом, гарантирующий неизменность токена.

При обработке запроса AuthService извлекает JWT из заголовка (обычно `Authorization: Bearer`). Сначала декодируются `header` и `payload` (из base64),

затем проверяется подпись с использованием известного секретного ключа. Если подпись верна и не истёк срок действия (claim `exp`), токен считается валидным. Так как используется HMAC (HS256), для проверки подписи не требуется обращение к внешним сервисам – достаточно владения ключом.

- HS256: симметричный алгоритм HMAC SHA-256 с секретным ключом для подписи и проверки.
- Валидация: сервис проверяет подпись, а также проверяет временные поля (`exp`, `nbf`), принимая или отвергая токен на основе этих данных.

В токен дополнительно включается так называемый `fingerprint` – уникальная метка клиента (например, хэш браузерного отпечатка или случайный идентификатор сессии). При валидации AuthService сверяет этот `fingerprint` с данными запроса (например, с текущим User-Agent). Это повышает устойчивость к перехвату: даже если JWT украдут, без совпадающего `fingerprint`-а (то есть без соответствующего клиентского окружения) токен будет считаться недействительным.

- Привязка к `fingerprint`: в токен добавляется метка устройства или сессии; при проверке сервис сверяет её с фактической информацией клиента, препятствуя использованию токена на другом устройстве.

bcrypt

Для хранения паролей используется алгоритм `bcrypt`. `Bcrypt` применяет соль и настраиваемый параметр сложности (`cost`). При создании пользователя `bcrypt` генерирует случайную соль и комбинирует её с паролем, а затем многократно вычисляет хеш. Количество раундов (`cost`)

задаёт, насколько медленным будет вычисление: чем выше значение, тем дольше генерируется хеш. В результате одна и та же комбинация пароля и соли всегда даёт один и тот же хеш, но каждый раз `bcrypt` генерирует новую соль и новый хеш, что исключает повторяемость.

- Соль: автоматическая случайная соль для каждого пароля предотвращает атаки по заранее скомпилированным словарям (радужным таблицам).
- Количество раундов: параметр сложности (`cost`) определяет число итераций; увеличенный `cost` значительно замедляет процесс хеширования, затрудняя подбор пароля.
- Невозможность обратного вычисления: `bcrypt` – односторонняя функция. Имея только хеш, невозможно извлечь исходный пароль, что защищает данные в случае компрометации хешей.

`Bcrypt` предпочтительнее быстрых хешей (MD5, SHA-256) для паролей, поскольку специально разработан для медленного хеширования. Быстрые алгоритмы (MD5, SHA-256) без соли уязвимы к современным атакам, так как их вычисление очень быстрое на современных GPU. `Bcrypt` же с солью и регулируемым `cost` устойчив к таким атакам, поэтому считается стандартом для безопасного хранения паролей.

Использование единого технологического стека (Go, gRPC, PostgreSQL, Redis, NATS JetStream и др.) существенно снижает накладные расходы на разработку и сопровождение. Единый язык и инструментарий упрощают обучение команды, позволяют переиспользовать компоненты (логгеры, генерацию кода, шаблоны конфигураций) и стандартизировать процессы CI/CD.

Благодаря согласованной архитектуре микросервисы легко масштабируются: все сервисы работают по одинаковым принципам и протоколам, поэтому можно быстро развертывать новые экземпляры и интегрировать дополнительную функциональность. В итоге однородный стек технологий делает систему «Путешествия по России» более предсказуемой, надёжной и удобной в сопровождении.

3.2 Реализация микросервисов

Система спроектирована по принципу слабосвязанной, модульной архитектуры: каждый сервис отвечает за отдельную доменную область и может масштабироваться независимо. Обмен сообщениями между сервисами осуществляется через gRPC-вызовы (все gRPC-запросы маршрутизируются через API-Gateway Envoy), а асинхронные события обрабатываются посредством брокера сообщений NATS JetStream. Хранилищем данных выступают PostgreSQL (для большинства сервисов), Redis (для кэша и хранения токенов), MinIO (S3-совместимое хранилище файлов); внешним почтовым сервисом служит Brevo (SendinBlue) для отправки электронной почты.

Auth Service

Назначение: Сервис аутентификации и авторизации отвечает за регистрацию пользователей, проверку их учётных данных и управление JWT-токенами (OAuth2.0). Он обеспечивает создание новых учётных записей, выдачу и обновление токенов доступа и обновления (refresh tokens), а также валидацию токенов при запросах к другим сервисам.

Основные бизнес-задачи: регистрация (`SignUp`), вход (`SignIn`), выход (`Logout`), обновление токенов (`RefreshToken`), проверка валидности токена (`ValidateToken`). При регистрации выполняется проверка уникальности `email`, хеширование пароля (`bcrypt/Argon2`), сохранение пользователя в БД и установка статуса учётной записи. При успешной регистрации сервис публикует событие о новом пользователе в шину NATS `JetStream` (топик `EmailEventsService`) для последующей отправки приветственного письма через сервис уведомлений. При логине проверяются `email` и пароль пользователя, после чего генерируются JWT (доступа) и токен обновления. Сами `refresh`-токены сохраняются в `Redis` (`Auth Cache`) с привязкой к пользователю для возможности проверки/отзыва. При запросе на обновление токена сервис проверяет валидность переданного `refresh`-токена (существует ли он в `Redis` и не истёк ли), генерирует новые токены. При `logout` – удаляет `refresh`-токен из `Redis` или помечает его аннулированным. Для повышения безопасности сервис ограничивает длину и формат входных данных, возвращая `INVALID_ARGUMENT`, если аргументы некорректны.

gRPC-методы и логика:

- `RegisterUser` (регистрация): принимает `email` и пароль. Сначала проверяет, что `email` имеет корректный формат и в БД пока нет пользователя с таким `email`. Если дублирование обнаружено, возвращает ошибку `ALREADY_EXISTS`. В случае отсутствия конфликта создаёт запись в таблице `users` (см. рис. 3.3–2) и устанавливает статус пользователя «`active`». После сохранения отправляет событие в NATS для рассылки `email`. Возвращает статус `OK` или `INTERNAL_ERROR`.

при ошибке БД.

- `LoginUser` (вход): получает `email` и пароль. Ищет пользователя в БД; если не найден – возвращает `NOT_FOUND`; если пароль неверен – возвращает `UNAUTHENTICATED`. При успехе генерирует `JWT` и `refresh`-токен, сохраняет `refresh`-токен в `Redis` (TTL согласно настройкам) и возвращает токены.
- `RefreshToken`: на вход получает старый `refresh`-токен. Проверяет его наличие в `Redis`: если отсутствует или истёк – возвращает `UNAUTHENTICATED`, иначе удаляет старый и выдаёт новую пару токенов (новый `refresh` также сохраняется в `Redis`).
- `ValidateToken`: принимает `JWT`, декодирует и проверяет подпись и срок. Если токен невалиден или истёк – возвращает `UNAUTHENTICATED`, иначе возвращает успех.
- При необходимости могут быть методы `Logout` (удаление `refresh`-тока) и `RevokeToken`. Все методы обрабатывают ошибки невалидных аргументов (`INVALID_ARGUMENT`), ошибки БД (`INTERNAL`) и нарушения бизнес-правил (дублирование – `ALREADY_EXISTS`, отсутствие ресурса – `NOT_FOUND`).

Архитектура и организация кода: Сервис реализован на Go по принципам «чистой архитектуры». Слои разделены следующим образом: уровень `Domain` описывает сущность `User` и её бизнес-логику, уровень `Repository` (интерфейс и реализация) взаимодействует с `PostgreSQL` и `Redis`, уровень `Usecase` реализует сценарии (регистрация, логин и др.), слой `Delivery` содержит gRPC-сервер и переводит входящие протоколы `Protobuf` в до-

менные структуры. В корне пакета определены интерфейсы (файлы `repository`, `usecase.go`), а в подпакетах (`/repository`, `/usecase`, `/delivery`) находятся их реализации и обработчики. При запуске сервиса происходит конфигурация соединений с БД и Redis, настройка gRPC-сервера (регистрация методов `AuthServiceServer` из `protobuf`), а также настройка клиента NATS для публикации событий. Все зависимости (доступ к БД, брокеру сообщений и прочему) внедряются через параметры конструкторов (`Dependency Injection`) для слабой связанности и удобства тестирования.

Работа с базой и кэшем: `AuthService` использует PostgreSQL для хранения учётных записей и статус пользователей. Для хранения активных `refresh`-токенов применяется Redis, позволяющий быстро валидировать токены и осуществлять их отзыв (удаление при `logout`). Таблица `users` имеет уникальный индекс по полю `email` – при попытке регистрации существующего `email` будет выброшена ошибка `ALREADY_EXISTS` (код 6).

Взаимодействие с другими сервисами: после создания учётной записи `AuthService` генерирует событие о регистрации и публикует его в NATS JetStream (топик `EmailEventsService`). Сервис `Notifications` подписан на этот топик и при получении события отправляет пользователю письмо через Brevo. Также сервис `Auth` может выполнять вызов `CreateProfile` в `ProfileService` (gRPC-вызов `profile.ProfileService`) чтобы автоматически создать связанную запись профиля при регистрации (например, с начальным именем и аватаром по умолчанию). Такой вызов производится синхронно по gRPC и при возникновении ошибок взаимодействия возвращает `FAILED_PRECONDITION` или `UNAVAILABLE`. При

аутентификации запросы клиентов требуют действительного JWT, который проверяется во всех сервисах через встроенную логику AuthService (или встроенным middleware), генерируя UNAUTHENTICATED для невалидного/просроченного токена.

Profile Service

Назначение: Сервис профилей отвечает за хранение и управление пользовательскими профилями, включая информацию о имени (username), ФИО, дате рождения, аватаре, а также за реализацию подписок (follow/unfollow) между пользователями.

Основные бизнес-задачи: хранение данных профиля, поддержка уникальности username, обработка действий «подписаться/отписаться» между пользователями. Сюда входят операции создания/чтения/обновления/удаления профиля (CRUD), изменение аватара, поиск профилей, а также ведение таблицы подписчиков. При создании нового профиля сервис проверяет, что username уникален (уникальный индекс), и сохраняет новую запись, связывая её с user_id из AuthService. При обновлении профиля возможно изменение поля username (тоже с проверкой уникальности) и загрузка нового аватара. Для загрузки/получения файла аватара сервис обращается к FileStorageService через gRPC. Например, метод UploadAvatar (реализованный во FileStorage) вызывается из ProfileService: он возвращает ID файла, который затем сохраняется в поле profiles.image_id. Поля followers_count и following_count автоматически обновляются при операциях подписки.

В качестве базы данных используется PostgreSQL с двумя таблицами: таблица profiles (поля user_id (PK, UUID), username, first_name,

`last_name,`
`date_of_birth, image_id`, счетчики подписок, метки времени) и связная таблица `follows` (`follower_id, followed_id`). Поле `username` имеет уникальное ограничение, поэтому в случае конфликта возвращаетсѧ `ALREADY_EXISTS`.

gRPC-методы и логика:

- `CreateProfile`: получает `user_id` (из `Auth`), `username`, ФИО и др. Проверяет заполнение обязательных полей; если `username` занят – возвращает `ALREADY_EXISTS`. Далее создает запись в `profiles`. Ошибки БД обрабатываются как `INTERNAL`.
- `GetProfile`: по `user_id` или `username` извлекает данные профиля. Если профиль не найден – возвращает `NOT_FOUND`. Может возвращать публичную информацию (без паролей).
- `UpdateProfile`: обновляет имя, фамилию, дату рождения; при смене `username` повторяет проверку уникальности (`ALREADY_EXISTS` при дублировании). Если указана загрузка нового аватара, получает от клиента поток байтов и передает его во `FileStorageService` (см. ниже). При успехе возвращает обновленный профиль. Ошибки валидации аргументов дают `INVALID_ARGUMENT`.
- `DeleteProfile`: удаляет запись профиля и связанные данные (например, обнуляет подписки). Если профиль не найден – `NOT_FOUND`.
- `FollowUser`: принимает `follower_id` и `followed_id`. Проверяет, что оба профиля существуют (вызов `GetProfile`). Нельзя подписаться на себя – приводит к `FAILED_PRECONDITION`. Если подпис-

ка уже существует, возвращает `ALREADY_EXISTS`. Иначе создает запись в таблице `follows` и инкрементирует счетчики `followers_count/following_count` в таблице `profiles`.

- `UnfollowUser`: обратная операция; если записи нет – возвращает `NOT_FOUND`, иначе удаляет подписку и декрементирует счетчики.
- `ListFollowers>ListFollowing`: возвращают список пользователей по запросу; возвращают пустой список, если подписчиков нет.
- `UploadAvatar` (в сотрудничестве с `FileStorageService`): клиент передаёт бинарный поток файла; `ProfileService` может выступать прокси или просить `FileStorageService` сгенерировать `presigned URL`. После загрузки файл регистрируется в таблице `files` `FileStorage`, а ID сохраняется в `profiles.image_id`. При некорректных файлах (слишком большой размер, неподдерживаемый формат) возвращается `INVALID_ARGUMENT` или `RESOURCE_EXHAUSTED`.

Сервис разбит по «чистой архитектуре» на слои: доменная модель `Profile` (`domain`), интерфейсы репозитория и `usecase` (`catalog of business logic`), реализации (`persistence` с использованием `SQL`), а также слой доставки (`gRPC handlers`). Например, в корне модуля определён интерфейс `ProfileRepository` (CRUD-операции над `profiles/follows`) и интерфейс `ProfileUsecase` с методами `CreateProfile`, `FollowUser` и т. д., а соответствующие имплементации находятся в подпакетах `repository` и `usecase`. Слой `Delivery` реализует `gRPC`-сервер (`generated из protobuf`) и преобразует входящие запросы в вызовы `usecase`. Такой подход гарантирует независимость бизнес-логики от конкретного хранилища и протокола, упрощает тестирование и замену компонентов.

Взаимодействие: ProfileService получает user_id от AuthService (обычно через метаданные gRPC/токен). При создании профиля AuthService инициирует gRPC-вызов CreateProfile в ProfileService для добавления записи.

ProfileService обращается к FileStorageService (`filestorage.File`) для работы с аватаром: при загрузке пользователя на уровне usecase вызывается метод для генерации PresignedURL или непосредственной передачи файла. Ответы от FileStorage возвращают либо ID файла, либо ошибку (NotFound, если хранилище недоступно). ProfileService не публикует событий в NATS напрямую (логика подписок обрабатывается синхронно), однако при необходимости можно расширить систему на события (например, оповещения через Notifications при новых подписках).

Activity Service

Назначение: Сервис активности реализует функциональность «лайки», «комментарии» и подсчета статистики по контенту. Он связывает пользователей и туристический контент (маршруты) в терминах социального взаимодействия.

Основные бизнес-задачи: добавление/удаление лайков к маршруту, публикация комментариев к маршруту, асинхронное обновление счётчиков лайков/комментариев. Например, пользователь может поставить лайк маршруту – ActivityService сохраняет соответствующую запись в БД. Аналогично создаются записи комментариев. Сервис также обеспечивает получение списка комментариев и информации о лайках. В системе реализовано асинхронное обновление агрегированных счётчиков: после изме-

нения лайка или комментария сервис публикует событие в NATS, которое обрабатывает вспомогательный компонент (`ActivityCountersService`) для обновления общих счетчиков в БД `ContentService`. Это позволяет снизить нагрузку и обеспечить `eventual consistency`.

База данных: PostgreSQL с таблицами: `route_likes` (поля `route_id`, `user_id`, метка и времени) и `route_comments` (`comment_id` PK, `route_id`, `user_id`, `text`, `created_at`). Каждая запись лайка уникальна по паре (`route_id`, `user_id`). При попытке поставить повторный лайк должна сработать проверка уникальности.

gRPC-методы и логика:

- `AddLike`: получает `route_id` и `user_id`. Сервис сначала проверяет, что маршрут существует (вызывая метод `GetRoute` из `ContentService`), если нет – возвращает `NOT_FOUND`. Затем пытается вставить запись в `route_likes`. Если такая запись уже есть (пользователь уже лайкал) – возвращает `ALREADY_EXISTS`. Иначе сохраняет лайк и публикует событие `LikeAdded` в NATS JetStream (содержит `route_id`), чтобы обновить счётчик лайков.
- `RemoveLike`: удаляет запись лайка. Если записи нет – `NOT_FOUND`. После удаления публикуется событие `LikeRemoved`.
- `AddComment`: принимает `route_id`, `user_id`, текст. Проверяет существование пользователя (`ProfileService`) и маршрута (`ContentService`). Вставляет запись в `route_comments` с новым `comment_id`. Если текст пустой или слишком длинный – `INVALID_ARGUMENT`. При успехе публикует событие `CommentAdded`.

- `DeleteComment`: удаляет свой комментарий (по `comment_id`). Если не найден или не принадлежит пользователю – `NOT_FOUND` или `PERMISSION_DENIED`.
- `ListComments`: возвращает список комментариев по маршруту. Если маршрута нет – `NOT_FOUND`.
- `GetLikeCount/GetCommentCount`: может возвращать текущие счётики. Эти данные можно либо поддерживать в памяти (посчитать при запросе), либо получать из агрегированной таблицы/кэша, обновляемой с помощью NATS-событий.

В упомянутых методах используются стандартные коды ошибок gRPC: `NOT_FOUND` при отсутствии ресурса, `ALREADY_EXISTS` при попытке дублирования (например, лайка), `INVALID_ARGUMENT` при ошибках валидации, если пользователь не авторизован – `UNAUTHENTICATED`, и `PERMISSION_DENIED` при попытке удалить чужой комментарий.

Архитектура: Чистая архитектура: доменные сущности «Like» и «Comment», репозиторий для CRUD-операций над таблицами (PostgreSQL), usecase-слой для бизнес-логики (проверка существования маршрута/пользователя, транзакции вставки/удаления, публикация событий), слой доставки gRPC. В модуле определены интерфейсы `ActivityRepository` и `ActivityUsecase` их реализации и gRPC-хендлеры. Очистка кода и тестируемость обеспечиваются тем, что работа с БД и внешними сервисами (`ContentService`, `ProfileService`, NATS) инвертирована через интерфейсы.

Работа с NATS: Каждый раз при изменении лайков или комментариев сервис публикует событие в JetStream. Например, после успешной

вставки лайка отправляется сообщение «LikeAdded» (с JSON- или Protobuf-данными). Другой поток (`ActivityCountersService`) подписан на эти события и обновляет агрегированные счётчики в таблице маршрутов или отдельном хранилище. Такой подход гарантирует, что тяжёлые операции обновления статистики выполняются асинхронно, а базовая транзакция обработки лайка/комментария остаётся простой и быстрой.

Взаимодействие: при добавлении лайка или комментария `ActivityService` обращается к `ContentService` (`content.ContentService`) для проверки существования маршрута, а к `ProfileService` (`profile.ProfileService`) для проверки пользователя (определение его прав). Эти вызовы gRPC инкапсулированы в `usecase` (например, интерфейсы `ContentClient`, `ProfileClient`). Обмен сообщениями с `FileStorageService` или `Notifications` здесь не происходит напрямую (если только при расширении системы делать уведомления об активности).

Content Service

Назначение: Сервис контента отвечает за хранение и управление туристическим контентом: маршрутами (`routes`) и достопримечательностями/местами (`places`). Он предоставляет CRUD-операции над маршрутами (название, сложность, дистанция, координаты точек) и местами (имя, адрес, описание, координаты). Кроме того, организована поддержка связывания мест с маршрутами в заданном порядке и прикрепления мультимедийных файлов (изображений) к этим сущностям.

Основные бизнес-задачи: добавление, обновление, удаление и поиск маршрутов и мест, фильтрация маршрутов по параметрам (например, сложности), изменение последовательности точек пути. Каждому маршру-

ту может соответствовать несколько мест (через промежуточную таблицу `route_places`), и у каждого места может быть несколько изображений (таблица `place_files`). Сервис обеспечивает консистентное хранение геоданных (`latitude/longitude`) и поддерживает транзакции при изменении маршрутов и связанных объектов. Также реализован gRPC метод потоковой передачи (`stream`) для загрузки/выгрузки изображений: например, клиент может вызывать метод

`UploadPlaceImage` как поток данных, который на стороне сервиса перенаправляется в

`FileStorageService` для сохранения в `MinIO`.

База данных: PostgreSQL с таблицами (см. рис. 3.3–5):

- `routes` (`id UUID, name, difficulty, distance`, массивы координат `latitudes/longitudes`, `user_id` владельца, метки времени, описание).
- `places` (`id UUID, name, address, description, latitude, longitude`, метки времени).
- Связующая таблица `route_places` (`route_id, place_id, ordering` для порядка следования).
- Таблица `files` (используется `FileStorageService`) и `place_files` (`place_id, file_id, ordering`) для привязки файлов-изображений к месту.

Для каждой таблицы определены первичные ключи и необходимые индексы. В таблице `routes` поле `name` может иметь составной уникальный

индекс (например, уникальность имени маршрута для одного пользователя). Схема организации данных позволяет эффективно фильтровать и выполнять геозапросы (через SQL или PostGIS расширения).

gRPC-методы и логика:

Маршруты:

- `CreateRoute`: принимает параметры маршрута (имя, сложность, дистанцию, координаты, описание, `user_id`). Проверяет обязательные поля (имя, хотя бы одну координату и пр.); если имя уже занято у данного пользователя – возвращает `ALREADY_EXISTS`. Создаёт новую запись в `routes`, сохраняет массивы координат. При добавлении начальных мест может выполнять дополнительные вставки в `route_places`.
- `GetRoute`: по `route_id` возвращает структуру маршрута со всеми полями и списком связанных мест (с учётом поля `ordering`). Если маршрут не найден – `NOT_FOUND`.
- `UpdateRoute`: обновляет свойства маршрута (может менять имя, сложность, описание и т. д.); проверяет наличие записи; если маршрут с новым именем конфликтует – `ALREADY_EXISTS`. Для изменения путевых точек может реализовываться несколько методов (например, `SetRoutePlaces`), которые обновляют `route_places` (добавление/удаление мест).
- `DeleteRoute`: удаляет маршрут и все связанные записи (места к маршруту, файлы). При отсутствии записи – `NOT_FOUND`.

- `ListRoutes`: возвращает список маршрутов с возможными фильтрами (например, по сложности, по географическому диапазону). Если фильтр указан некорректно – `INVALID_ARGUMENT`.

Места (Places):

- `CreatePlace/GetPlace/UpdatePlace/DeletePlace`: аналогичные CRUD-операции для объектов `places`. При создании указывается имя, адрес, координаты.
- `AddPlaceToRoute/RemovePlaceFromRoute`: связывают место с маршрутом. Если `route_id` или `place_id` не найдены – `NOT_FOUND`. При добавлении проверяется, что такая связь ещё не существует (иначе `ALREADY_EXISTS`), и задаётся порядок следования.

Изображения (Media):

- `UploadPlaceImage (stream)`: потоковое получение бинарных данных изображения от клиента. `ContentService` принимает потоки (`Chunked gRPC`) и переадресует данные во `FileStorageService`: либо получает от него `Presigned URL` (и сам загружает через `HTTP`), либо напрямую вызывает `UploadFile gRPC`. После успешной загрузки сервис получает идентификатор файла (из `FileStorage`) и создаёт запись в `place_files` с `place_id` и `file_id`. Если файл по размеру превышает лимит – `RESOURCE_EXHAUSTED`.

- `GetPlaceImage (stream)`: загрузка изображения с помощью gRPC-стрима из `FileStorageService` по `file_id` и транслирование клиенту. Если файл отсутствует – `NOT_FOUND`.

Таким образом, большинство gRPC-методов `ContentService` оборачивает стандартные CRUD-операции над SQL, дополняя их необходимыми проверками и вызовами внешнего сервиса для работы с файлами. При этом используется унифицированный паттерн обработки ошибок: `NOT_FOUND` для отсутствующих записей, `ALREADY_EXISTS` для дубликатов, `INVALID_ARGUMENT` при некорректных входных данных.

Архитектура: как и в других сервисах, применена «чистая архитектура»: в пакете `content` определены доменные модели `Route` и `Place`, интерфейсы репозитория и `usecase` (файлы `repository.go`, `usecase.go` на верхнем уровне) и их реализации в подпакетах. Layer `usecase` инкапсулирует основную логику (создание маршрутов, валидация координат, связывание с файлами), слой `repository` (например, `PostgresRepository`) выполняет SQL-запросы к таблицам. Слой `Delivery` реализован gRPC-сервер: все обработчики принимают `Protobuf`-запросы и вызывают методы `usecase`. Такой подход отделяет бизнес-логику от инфраструктуры (БД, протокол коммуникации).

Работа с файловым сервисом и хранением: для работы с изображениями `ContentService` взаимодействует с `FileStorageService`. В момент загрузки изображения сервис запрашивает у `FileStorage` `presigned URL` или использует gRPC-метод загрузки. Ответ от `FileStorage` возвращает `file_id`, который связывается с `place_id` в таблице `place_files`. `MinIO` настроен как S3-совместимое хранилище; `FileStorage` контролирует TTL для файлов и шифрование. API-Gateway (Envoy) перенаправля-

ет HTTP-запросы по `presigned URL` напрямую в MinIO, что разгружает сервис от передачи больших бинарных файлов.

Взаимодействие с другими сервисами: ContentService может вызывать методы ProfileService (например, для получения информации о владельце маршрута) и ActivityService (для получения количества лайков/комментариев), но в минимальной конфигурации такого нет. В основном другие сервисы обращаются к ContentService: ActivityService – для проверки существования маршрута, ProfileService – при необходимости искать маршруты по `user_id`, AuthService – при проверке прав владельца, и FileStorageService – для загрузки файлов. После модификации контента сервис может публиковать события (например, «RouteCreated» или «PlaceAdded») в NATS для дальнейшей обработки или обновления кэшей, однако базовые операции сконцентрированы на синхронной CRUD-логике.

FileStorage Service

Назначение: Сервис управления файлами обеспечивает надёжную загрузку, хранение и выдачу медиафайлов (изображений, аватаров и т. д.) в систему. Он выступает как прослойка между микросервисами и объектным хранилищем MinIO, реализуя политику безопасности, ограничения по размеру, TTL (время жизни) и шифрование файлов.

Основные бизнес-задачи: приём файлов от клиентов или других сервисов, сохранение их в MinIO, хранение метаданных в БД, выдача файлов по запросу, управление сроком жизни (TTL) и правами доступа. Также FileStorage генерирует `presigned URL`, позволяющие клиентам напрямую загружать или скачивать файлы через MinIO по HTTP (через API

Gateway).

База данных: PostgreSQL с таблицей `files` (пример на рис. 3.3–3 и 3.3–5). Поля: `id` (`integer`, PK), `file_name`, `storage_bucket`, `storage_id` (идентификатор в MinIO), `internal_url`, `placeholder` (для превью), `size`, флаги доступа (`from_public`), метки времени. Эта таблица хранит метаданные загруженных файлов. Первичный ключ – целочисленный `id`.

gRPC-методы и логика:

- `UploadFile (stream)`: клиент передаёт байтовый поток файла. Сервис накапливает полученные данные, проверяет, чтобы общий размер не превышал предельный (например, 10 MB); если превышает, прерывает с кодом `RESOURCE_EXHAUSTED`. Данные шифруются на лету (используется серверная сторона MinIO или KMS) и отправляются в заданный бакет MinIO. После завершения загрузки генерируется запись в таблице `files` с метаданными (имя, размер, `internal_url` и пр.) и возвращается `id` файла. Если запись с таким именем уже есть – `ALREADY_EXISTS`. При любых внутренних ошибках возвращается `INTERNAL`.
- `DownloadFile (stream)`: клиент запрашивает файл по `id`. Сервис находит запись в БД; если нет – `NOT_FOUND`. Далее запрашивает файл из MinIO и передаёт данные клиенту по gRPC-стриму. Если файл удалён или повреждён – возвращает `NOT_FOUND` или `INTERNAL`.
- `GetPresignedUploadURL`: по запросу с названием файла возвращает клиенту подписанный URL для загрузки (HTTP PUT) непосред-

ственno в MinIO. Срок действия URL ограничен TTL (например, 15 минут). Это позволяет разгрузить сервис от передачи больших файлов через gRPC.

- `GetPresignedDownloadURL`: аналогично выдаёт URL для скачивания (HTTP GET).
- `DeleteFile`: удаляет запись из БД и файл из MinIO. Если файла нет – NOT_FOUND.

Архитектура: Сервис организован по слоям: доменная модель `File` с методами (например, метод проверки размера), репозиторий для работы с PostgreSQL, usecase-слой, и слой доставки gRPC. Используются интерфейсы для абстракции работы с MinIO (например, `FileStorageClient`) и для доступа к БД. Шифрование файлов осуществляется через встроенные механизмы MinIO (Server-Side Encryption) или путем шифрования байтов перед отправкой. Политика TTL задаётся либо на стороне MinIO (правила удаления после X дней), либо самим сервисом (отмечая метку времени в БД и периодически удаляя устаревшие файлы, используя внутренний `scheduler` или события NATS).

Интеграция с другими сервисами: `ProfileService` и `ContentService` обращаются к `FileStorageService` при необходимости загрузки или получения файлов: они могут использовать gRPC-методы загрузки либо `presigned` URL. Например, при смене аватара `ProfileService` запрашивает `GetPresignedUploadURL`, получает ссылку и отправляет файл напрямую в MinIO. Либо вызывает `UploadFile` с передачей потока. После получения `file_id` записывает его в профиль.

`ContentService` поступает аналогично для изображений мест. API Gateway

(Envoy) перенаправляет запросы по presigned URL напрямую в MinIO. FileStorageService также публикует события об удалении старых файлов в NATS (для логирования или обратного оповещения других компонентов). Ошибки взаимодействия (например, NOT_FOUND при запросе несуществующего файла) обрабатываются по стандарту gRPC.

Notifications Service

Назначение: Сервис уведомлений занимается отправкой электронных сообщений пользователям системы через сторонний SMTP/API сервис Brevo (SendinBlue). Он реагирует на события, публикуемые другими микросервисами в NATS JetStream, и превращает их в email-уведомления (регистрация, смена пароля, действия в приложении и т. д.).

Основные бизнес-задачи: подписка на топики событий NATS (например, UserRegistered, PasswordReset, CommentAdded), формирование писем по заранее заданным шаблонам и отправка их через REST API Brevo. При получении события сервис десериализует данные, проверяет их корректность (если формат события неверен – INVALID_ARGUMENT) и, в случае необходимости, обогащает информацией из ProfileService (например, подставляет имя пользователя в письмо). После подготовки вызывает Brevo API; при проблемах (недоступность сети, неверные настройки) логирует ошибку и может инициировать повторную отправку (алгоритм повторов настраивается отдельно). Код ответа внешнего сервиса не влияет на gRPC-интерфейс, так как Notifications работает по модели Pub/Sub, а не по gRPC-запросам.

Архитектура: внутри NotificationsService также используется концепция чистой архитектуры: есть слой получателей сообщений (Subscriber)

от NATS, слой бизнес-логики (формирование тела письма, выбор шаблона) и слой доставки (REST-клиент к Brevo). Хранилища данных как такового нет (все шаблоны и конфигурации хранятся в конфиге или внутрипамяти). Каждый `subscriber` запускается как отдельный поток, подписывается на соответствующий `topic JetStream` и вызывает `usecase` при получении сообщения. Такая модульная организация упрощает добавление новых типов уведомлений.

Взаимодействие: `NotificationsService` не предоставляет собственных gRPC-методов клиентам; его задача – слушать события от других сервисов. Например, после регистрации `AuthService` публикует событие «`UserRegistered`» (с `email` и именем), `Notifications` его получает и вызывает `Brevo API`. Подобным образом могут обрабатываться события из `ActivityService` (например, «`CommentAdded`» для уведомления автора маршрута) или любые другие. Благодаря использованию NATS и `JetStream` достигается надёжная асинхронная доставка: если сервис упадёт, после перезапуска он продолжит получать неполученные сообщения.

Коды ошибок и устойчивость: `NotificationsService` возвращает в лог или мониторинг статусы успеха/ошибок API `Brevo` (например, `502 Bad Gateway` при недоступности `Brevo` трактуется как `UNAVAILABLE`). В gRPC-слое также можно определить метод `SendEmail` (например, для ручной отправки), который на вход принимает структуру `EmailRequest` и возвращает статус. В таком случае возвращаемые коды будут аналогичны: `INVALID_ARGUMENT` на неправильный формат адреса или тела письма, `INTERNAL` на ошибки сервера. Однако основная работа – через асин-

хронные события и REST, а не через gRPC.

Таким образом, каждый микросервис спроектирован с учётом принципов модульности, слабой связанности и расширяемости. AuthService обеспечивает безопасность и единую авторизацию, ProfileService – персональные данные пользователей, ActivityService – социальные функции, ContentService – хранение маршрутов, FileStorageService – управление файлами, а NotificationsService – оповещения. Между ними устанавливаются чёткие интерфейсы (gRPC и события NATS), что упрощает сопровождение и масштабирование системы.

3.3 Организация DevOps-инфраструктуры и CI/CD

Структура Helm-чартов

В системе «Путешествия по России» каждый микросервис (`auth`, `profile`, `activity`, `content`, `filestorage`, `notifications`, `broker`) оформлен отдельным Helm-чартом. Такой подход обеспечивает модульность и независимость сервисов: каждый чарт описывает Kubernetes-ресурсы для одного приложения. Все эти чарты объединены в один родительский чарт `platform`, в котором с помощью секции `dependencies` задаются зависимости на каждый микросервис. В блоке `Chart.yaml` родительского чарта указываются пары `name`, `version` и `repository`, причём в качестве репозитория используется адрес OCI-реестра (например, GHCR).

При локальной отладке каждый микросервис может запускаться со своим собственным файлом `values.yaml`, содержащим настройки окружения для разработки. Для деплоя в `production` используется централизо-

зованный набор конфигураций: общий `values.yaml` и разделённый на обычные и секретные параметры `values.secret.yaml`. В этих файлах определяются порты, URL сервисов, переменные окружения, параметры подключения к БД и т. д., а чувствительные данные (пароли, токены) выносятся в `values.secret.yaml` для безопасности. Кроме того, в чартах сервисов, использующих PostgreSQL, настраивается Kubernetes Job с аннотацией `helm.sh/hook: pre-install,pre-upgrade` для применения миграций схемы базы данных перед установкой или обновлением приложения. Такой job загружает контейнер с миграциями и выполняет SQL-скрипты, гарантируя, что схема БД актуализируется до запуска основного пода.

CI/CD (GitHub Actions)

В каждом репозитории микросервиса организован конвейер CI/CD на базе GitHub Actions с несколькими `workflow`-файлами. Обычно используются три типа `workflow`: `app.yaml` для сборки приложения и создания Docker-образа, `publish-*-chart.yaml` для упаковки и публикации Helm-чарта, а при наличии PostgreSQL – `migrations.yaml` для сборки контейнера с миграциями. Все эти действия выполняются в среде GitHub Actions с использованием секретов (например, `GHCR_TOKEN`, `GHCR_USERNAME`, `GOPRIVATE_PAT`) для аутентификации в GitHub Container Registry и доступа к приватным модулям.

Сборка приложения ведётся с помощью многостадийного `Dockerfile`. На этапе `build` используется образ `golang:1.24`, в котором скачиваются зависимости (через `go mod`) и компилируется исполняемый файл. Затем на финальной стадии берётся минимальный образ `alpine:3.19`, в кото-

рый копируется только готовый бинарник. В результате итоговый образ получается очень компактным и безопасным: в нём отсутствуют инструменты сборки и прочие ненужные слои. Такой подход значительно уменьшает размер контейнера и поверхность потенциальных уязвимостей.

Настройки триггеров `workflow` соответствуют следующей логике:

- Push в ветку `main` запускается сборка и публикация Docker-образа приложения.
- Push тега вида `chart-*` запускается упаковка и публикация Helm-чарта в GHCR.
- Изменения в папке `migrations` инициируется сборка и публикация контейнера миграций.

Все собранные Docker-образы и чарты публикуются в GitHub Container Registry (GHCR), где хранятся как пакеты OCI. Использование GHCR позволяет легко управлять версиями и правами доступа. Каждый релиз получает семантический тег (например, `v0.1.12`), что обеспечивает трассируемость версий. При необходимости образы и чарты могут быть удалены вручную, иначе GHCR сохраняет их бессрочно, что гарантирует повторяемость окружения.

Автоматический деплой (Helm)

Для автоматического деплоя в репозитории `vkr-platform` настроен отдельный `workflow GitHub Actions`. При пуше в ветку `main` этот `workflow` выполняет команду `helm upgrade --install platform`, используя заранее подготовленный `kubeconfig` (он хранится в GitHub

`Secrets`). Поскольку родительский чарт `platform` содержит все сервисы в секции `dependencies`, единственный вызов `helm` разворачивает все микросервисы сразу. При вызове `Helm` передаются файлы `values.yaml` и `values.secret.yaml` из общего хранилища, в которых заданы порты, URL, переменные окружения, параметры подключения к базе данных и другие настройки. Такой централизованный подход упрощает управление конфигурацией: однократная правка значений в `values.yaml` отражается на всех сервисах.

Перед запуском каждого сервиса из чарта также выполняются миграции схемы (см. раздел 3.4.1). Например, для сервиса `auth` определяется Kubernetes Job `auth-migrations` с аннотацией `Helm hook`, который применяет SQL-миграции до старта контейнера приложения. Таким образом, структура базы данных гарантированно соответствует версии сервиса. В целом этот pipeline позволяет оперативно и согласованно обновлять всю платформу одной операцией `Helm` – что повышает надёжность и упрощает откат при необходимости.

Логирование

Во всех микросервисах реализовано структурированное логирование на базе библиотеки `zap` от Uber. `Zap` изначально проектировался как очень высокопроизводительный логгер: он обеспечивает «blazing fast» скорость записи и избегает лишних аллокаций при сериализации полей. Например, API `SugaredLogger` из `zap` в 4–10 раз быстрее многих аналогичных библиотек. В нашем случае `zap` обёрнут во внутренний интерфейс логгера приложения. Такая обёртка позволяет при необходимости менять реализацию логирования или дополнять её (например, добавлять новые

уровни или форматы) без изменения бизнес-логики.

Логи сервисов пишутся одновременно в консоль (`stdout`), и в файл (напр., во внутренний том контейнера). Вывод в `stdout` соответствует рекомендациям Kubernetes: системный агент `kubelet` читает эти сообщения и агрегирует их, а разработчики могут просматривать логи через `kubectl logs`. Запись в файл (например, в монтируемый `PersistentVolume`) позволяет сохранять полную историю логов и при необходимости передавать её в сторонние системы. Выбранная архитектура лога легко расширяется: в будущем можно подключить отправку логов в ELK/Loki, S3 или другие решения, не меняя код микросервисов.

Серверное окружение

Для развертывания микросервисов проекта «Путешествия по России» была использована арендованная виртуальная машина. Параметры виртуального сервера составляют 4 виртуальных ядра процессора (3,3 ГГц), 8 ГБ оперативной памяти и 80 ГБ NVMe SSD-диска. Для доступа к кластеру был выделен статический IP-адрес. Виртуальная машина выступает хост-нодой локального Kubernetes-кластера, в котором развернуты все микросервисы. При запуске всех компонентов в режиме ожидания (без нагрузки) потребление оперативной памяти составляет около 1,5 ГБ.

3.4 Тестирование микросервисной системы

Юнит-тестирование

В Go для написания юнит-тестов используется стандартный пакет `testing`. Командой `go test ./...` можно автоматически обнаруживать и выпол-

нять все тесты в проекте.

Дополнительно применяются расширения из библиотеки `Testify`: пакеты `assert` и `require` позволяют делать проверки более выразительными и удобными. Например, `assert.Equal(t, ожидаемое, фактическое)` упрощает формирование читаемых сообщений об ошибках.

В юнит-тестах проверяются изолированные логические функции микросервисов, не зависящие от внешних компонентов. Сюда относятся задачи валидации данных, генерации и проверки токенов, базовая бизнес-логика «чистых» сервисов и пр. Такие тесты пишутся без обращения к базе данных или сторонним сервисам.

Основная выгода юнит-тестирования – быстрая отладка логики без поднятия всей системы. Это помогает обнаруживать и устранять ошибки на ранних этапах разработки, когда локализовать проблему проще и дешевле. Юнит-тесты гарантируют надёжность каждого компонента как отдельного блока, что повышает устойчивость всей микросервисной архитектуры.

Интеграционное тестирование

Интеграционные тесты проверяют совместную работу сервисов в условиях, близких к реальным. Они направлены на верификацию взаимодействия компонентов и потоков данных между ними, чтобы убедиться, что система функционирует как единое целое. В контексте микросервисов это означает отправку реальных запросов между сервисами через API-шлюз (`Envoy`) и проверку обработки сообщений через шину (`NATS JetStream`).

- Для тестирования взаимодействия используется `Postman Collection`, включающая все gRPC- и REST-эндпоинты микросервисов (напри-

мер, запросы к сервисам Auth, Profile, Content, Notifications).

Это позволяет задать последовательность сценариев и проследить, как ответы одного сервиса поступают в следующий.

- Запуск коллекции автоматизирован через Postman CLI (Newman) или встроенный Runner. Newman – это инструмент командной строки для выполнения Postman коллекций, который легко интегрируется в CI/CD-пайплайн. К примеру, в Kubernetes можно запустить Postman-контейнер с Newman внутри кластера, чтобы тестировать внутренние сервисы без их внешней экспозиции.
- Само тестовое окружение развертывается с помощью Helm-чарта (например, в dev-кластере). Команда `helm test` позволяет запускать произвольные Kubernetes-ресурсы, включая контейнеры с Newman и тестовыми скриптами. Например, можно использовать официальный образ `postman/newman` и передавать в него JSON-файлы коллекции и окружения.
- Такой подход гарантирует, что система протестирована в условиях, максимально приближенных к продакшенну. Проверяется не только логика отдельных API, но и сетевое взаимодействие через Envoy, а также асинхронный обмен сообщениями через NATS. Это позволяет убедиться в корректной работе всего цикла запросов и ответов между сервисами.

Выводы по главе

Таким образом, в третьей главе продемонстрирована целостная реализация серверной части приложения «Путешествия по России» – от выбора и обоснования единого технологического стека (Go 1.24, gRPC, PostgreSQL, Redis, NATS JetStream, MinIO) до детального описания микросервисов и DevOps потока, подтверждённого комплексным тестированием. Все сервисы построены на принципах чистой архитектуры, чётко разделяют доменные области (`Auth`, `Profile`, `Content`, `Activity`, `FileStorage`, `Notifications`) и общаются строго типизированными gRPC контрактами, а асинхронные задачи надёжно обрабатываются через JetStream с гарантией хотя бы однократной доставки. Контейнеризация и единая система Helm чартов обеспечивают воспроизводимое развёртывание всей платформы в локальном Kubernetes кластере на выделенной виртуальной машине (4 vCPU 3,3 ГГц, 8 ГБ RAM, 80 ГБ NVMe), где при штатной работе потребление памяти не превышает 1,5 ГБ. Автоматизированные CI/CD конвейеры GitHub Actions формируют минимальные Docker образы, публикуют их вместе с чартами в GHCR и запускают Helm upgrade, гарантируя непрерывную доставку и миграцию схем БД без простоев; структурированное логирование на базе zap и ротация lumberjack предусматривают последующее подключение централизованной системы сбора логов. Юнит тесты (`testing + testify`) подтверждают корректность ключевой бизнес логики в изоляции, а интеграционные сценарии Postman/Newman, исполняемые в `dev` кластере через `helm test`, удостоверяют согласованность микросервисного взаимодействия и работу критических путей через Envoy и JetStream в среде, максимально приближённой к `production`.

Совокупность изложенных решений демонстрирует, что реализованная архитектура не только удовлетворяет функциональным требованиям, но и обладает заложенными во второй главе нефункциональными характеристиками — масштабируемостью, отказоустойчивостью, безопасностью и готовностью к дальнейшему расширению.

Такая структура проекта обеспечивает хорошую организацию кода и четкое разделение ответственности между компонентами. Такая структура каталогов соответствует принципам clean architecture: слой инфраструктуры отделен от бизнес-логики, а конфигурация выделена в отдельный модуль для упрощения управления настройками приложения.

В данной главе представлена подробная реализация серверной части мобильного приложения «Путешествия по России» на языке Go с использованием микросервисной архитектуры. Разработанная система включает 6 основных микросервисов: Auth для аутентификации, Profile для управления профилями, Content для работы с маршрутами, Activity для социальных взаимодействий, FileStorage для работы с файлами и Notifications для отправки уведомлений.

ГЛАВА 4 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ И ВЫБОР ТЕХНОЛОГИЧЕСКОГО СТЕКА

Введение

В предыдущей главе были определены ключевые функциональные требования и приоритеты для мобильного приложения, направленного на удовлетворение потребностей различных групп пользователей в сфере туризма и путешествий. Для успешной реализации задуманного продукта необходимо тщательно продумать архитектуру приложения и механизмы взаимодействия между его компонентами.

Данная глава посвящена детальному проектированию технической инфраструктуры приложения, включая архитектуру мобильного клиента и API для взаимодействия с бэкендом. Проработка этих аспектов позволит обеспечить эффективное функционирование приложения, его масштабируемость и возможность интеграции с внешними системами.

Основные задачи, решаемые в ходе работы:

Цели и задачи главы

- Проектирование архитектуры мобильного приложения с упором на принципы SOLID¹. Например, KISS² и DRY³.
- Разработка спецификации API для взаимодействия мобильного клиента с бэкенд-сервисами, обеспечивающие обмен необходимыми данными и функциональностью.
- Выявление потенциальные сложности и предложить пути их решения на этапе проектирования архитектуры и API.

Структура главы

- **4.1. Проектирование архитектуры мобильного приложения** – в этом разделе будет рассмотрена выбор технологической стопки, структура приложения, подходы к организации кода и данные, необходимые для корректной работы приложения.
- **4.2. Проработка API для взаимодействия с бэкендом** – здесь будут описаны требования к API, структура запросов и ответов, механизмы аутентификации и авторизации, а также примеры использования API

¹SOLID – акроним для пяти принципов объектно-ориентированного проектирования: Single Responsibility Principle (Принцип единственной ответственности), Open/Closed Principle (Принцип открытости/закрытости), Liskov Substitution Principle (Принцип подстановки Барбары Лисков), Interface Segregation Principle (Принцип разделения интерфейса) и Dependency Inversion Principle (Принцип инверсии зависимостей)

²KISS (Keep It Simple, Stupid) – принцип проектирования, который призывает к простоте и отказу от излишней сложности в системе.

³DRY (Dont Repeat Yourself) – принцип разработки программного обеспечения, направленный на снижение повторения информации различного рода, особенно в системах со множеством слоев абстрагирования.

для основных сценариев взаимодействия между клиентом и сервером.

4.1 Проектирование архитектуры мобильного приложения

Технические требования (SRS)

Для проектирования эффективной архитектуры необходимо четко понимать задачи, которые должно решать приложение, и как пользователи будут с ним взаимодействовать. В процессе разработки мобильного приложения для сферы туризма и путешествий особое внимание уделяется техническим аспектам, которые обеспечивают его функциональность, удобство использования и безопасность. Технические требования (SRS) играют ключевую роль в определении основных характеристик приложения и его взаимодействия с пользователем и внешними системами.

На данном этапе разработки важно детально проработать технические аспекты, чтобы обеспечить соответствие приложения ожиданиям пользователей и требованиям рынка. Это включает в себя выбор платформы, разработку пользовательского интерфейса, обеспечение безопасности данных и интеграцию с внешними сервисами.

На основе анализа, проведенного в Главе 1 и общих целей проекта, были сформулированы следующие ключевые аспекты следующие технические требования:

- **Поддержка iOS и Android:** приложение должно быть доступно для пользователей устройств на базе операционных систем iOS и Android.

Это обеспечит широкий охват аудитории и возможность использования приложения на различных устройствах. Разработка под обе платформы позволяет охватить максимальное количество потенциальных пользователей и обеспечивает гибкость в выборе устройства для доступа к приложению. Выбор кроссплатформенного фреймворка Flutter позволяет поддерживать обе эти платформы из коробки с наименьшей затратой усилий по сравнению с нативной разработкой под каждую из этих платформ.

- **Интуитивная понятность:** интерфейс приложения должен быть разработан таким образом, чтобы пользователи могли легко ориентироваться и выполнять необходимые действия без дополнительных инструкций. Это достигается за счёт логичной структуры меню, понятных иконок и чётких инструкций.
- **Адаптивность:** дизайн интерфейса должен быть адаптирован под различные размеры экранов мобильных устройств, обеспечивая оптимальное отображение и удобство использования. Адаптивный дизайн позволяет приложению корректно отображаться на смартфонах и планшетах с разными диагоналями экранов.
- **Визуальная привлекательность:** интерфейс должен иметь привлекательный и современный дизайн, который будет соответствовать ожиданиям пользователей и способствовать положительному восприятию приложения. Визуальная составляющая играет важную роль в создании первого впечатления и удержании интереса пользователей.
- **Защита персональных данных:** приложение должно обеспечивать на-

дёжную защиту персональных данных пользователей, включая их имя, адреса, номера телефонов и другую конфиденциальную информацию. Это достигается путём использования современных криптографических методов и соблюдения требований законодательства в области защиты данных. Защита данных повышает доверие пользователей к приложению и соответствует юридическим нормам.

- **Аутентификация и авторизация:** реализация механизмов аутентификации и авторизации пользователей для обеспечения доступа к персональным данным и функциональности приложения только для авторизованных пользователей. Это предотвращает несанкционированный доступ к чувствительной информации и обеспечивает безопасность пользовательских данных.
- **Внешние сервисы:** приложение должно предоставлять возможность интеграции с картографическими сервисами, такими как Яндекс Карты и 2GIS, для просмотра маршрутов. Это позволит пользователям получать дополнительные данные о местах, строить маршруты и просматривать информацию о достопримечательностях прямо из приложения. Интеграция с внешними сервисами расширяет функциональность приложения и улучшает пользовательский опыт. Для безопасного открытия этих приложений, можно завести диплинки ⁰.

Первые три пункта помогают снизить количество ошибок в продакшене и сделать их более предсказуемыми. Последний пункт важен для обеспечения отзывчивости пользовательского интерфейса, так как долгие операции

⁰диплинк - гиперссылка, которая открывает конкретную страницу в приложении

не должны блокировать основной поток выполнения, ответственный за отрисовку интерфейса.

4.2 Основные пути пользователя

Путь пользователя (user journey) - это последовательность шагов, которую предпринимает пользователь для того, чтобы завершить конкретную задачу на сайте или в приложении. Для реализации максимально удобного функционала необходимо понимать самые частые пути пользователя и делать их наиболее простыми для него.

В нашем приложении можно выделить следующие основные пути пользователя:

- **Поиск маршрута** - это главная цель пользователя, найти тот самый маршрут, который он захочет пройти. Эта функция позволяет пользователям быстро и удобно находить маршруты, соответствующие их интересам и потребностям. Они могут использовать различные критерии для поиска, такие как местоположение, тип маршрута, длина и сложность. После выбора маршрута пользователи могут просмотреть его детали, включая карту, описание и фотографии, чтобы лучше понять, что их ждёт во время путешествия. Посмотреть путь, который придётся пройти пользователю, чтобы найти маршрут, можноувидеть на рисунке 2.1.

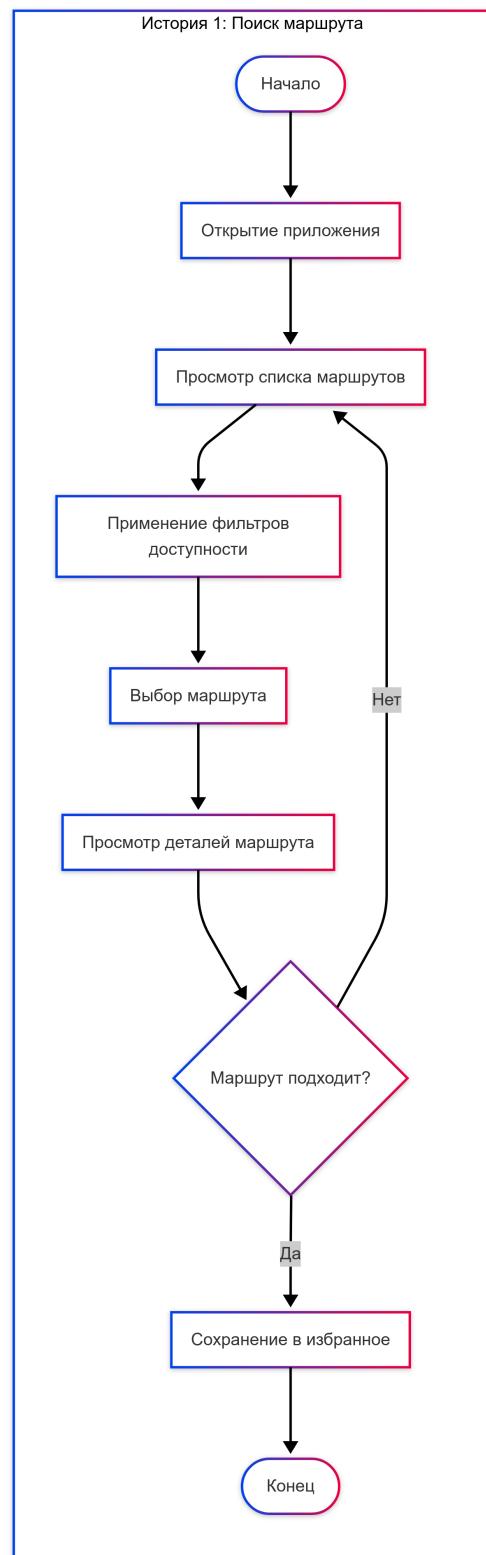


Рис. 14. Диаграмма «Поиск маршрута»

- **Создание маршрута** - создание маршрута также важная часть функционала, которая позволяет пользователям делиться своими любимыми маршрутами с другими. Пользователи могут создавать маршруты, добавляя точки интереса, описания и фотографии. Это даёт возможность не только планировать свои путешествия, но и помогать другим пользователям находить интересные места и маршруты. Чтобы узнать, что должен сделать пользователь, чтобы создать маршрут, нужно открыть рисунок 2.2.

Диаграмма вариантов использования

Как было замечено ранее, у нашего приложения будут два основных пользователя: создатель и потребитель маршрута. Диаграмма вариантов использования является важным инструментом для визуализации взаимодействия между пользователями и системой. В контексте нашего приложения можно выделить два основных типа пользователей: создатель маршрута и потребитель маршрута. Наш сервис выступает в роли посредника, обеспечивая взаимодействие между этими двумя сторонами.

Создатель маршрута – это пользователь, который создаёт новые маршруты, добавляя в них информацию о точках интереса, описания, фотографии и другие данные. Он использует наш сервис для публикации маршрутов, которые затем могут быть найдены и использованы другими пользователями – потребителями маршрутов.

Потребитель маршрута – это пользователь, который ищет интересные маршруты для своих путешествий. Он использует наш сервис для поиска, просмотра и сохранения маршрутов, созданных другими пользователями. Потребитель может оценивать маршруты, оставлять комментарии



Рис. 15. Диаграмма «Создание маршрута»

и делиться ими с друзьями.

Наш сервис предоставляет функциональность, необходимую для создания, управления и просмотра маршрутов. Он обеспечивает безопасное и удобное взаимодействие между создателями и потребителями маршрутов, а также предоставляет дополнительные возможности, такие как фильтрация маршрутов по различным критериям, сохранение маршрутов в избранное и получение уведомлений о новых маршрутах.

Так как наш сервис будет проводником между этими двумя сторонами, в следующей диаграмме вариантов использования фигурируют три стороны: наш сервис, создатель и потребитель маршрута. Как именно они связаны, можно увидеть на рисунке 2.3.

Диаграмма вариантов использования демонстрирует следующие взаимодействия:

1. **Просмотр маршрутов:** потребитель маршрутов взаимодействует с нашим сервисом для просмотра списка доступных маршрутов.
2. **Редактирование маршрута:** потребитель может редактировать маршрут, и наш сервис отображает отредактированный маршрут.
3. **Фильтрация маршрутов:** потребитель использует функционал фильтрации для поиска маршрутов по определённым критериям, и сервис отображает отфильтрованные маршруты.
4. **Сохранение маршрута в избранное:** потребитель может сохранять маршруты в избранное, и сервис подтверждает сохранение.
5. **Комментирование маршрута:** потребитель может оставлять комментарии к маршрутам, и сервис отображает комментарии.

- 6. Создание нового маршрута:** создатель маршрута взаимодействует с сервисом для создания нового маршрута, и сервис подтверждает создание.
- 7. Редактирование маршрута:** создатель может редактировать маршрут, и сервис отображает отредактированный маршрут.

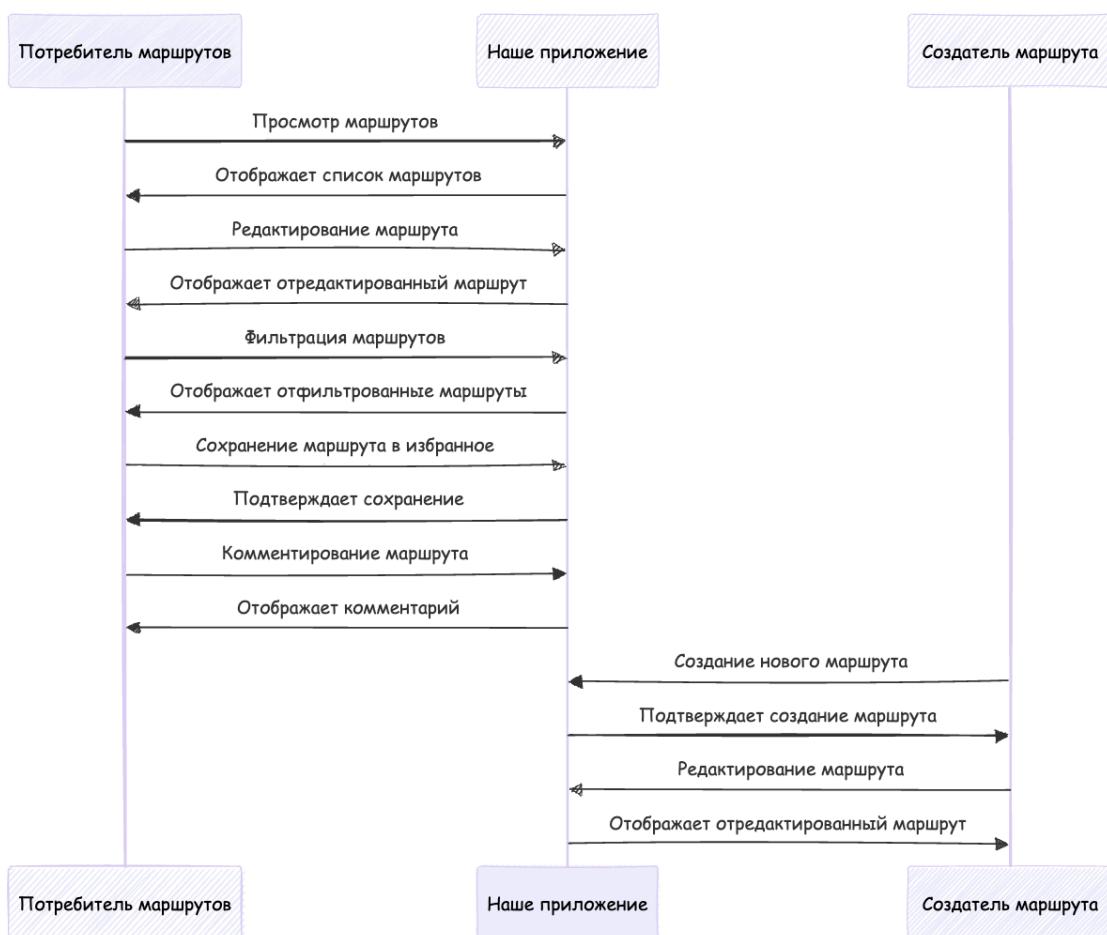


Рис. 16. Диаграмма вариантов использования приложения

4.3 Архитектура мобильного приложения

После проработки требований и сценариев можно приступить к выбору архитектуры мобильного приложения. Но перед этим необходимо дать определение термину архитектура.

Выбор архитектуры является критически важным решением, влияющим на все аспекты жизненного цикла программного продукта. **Архитектура программного обеспечения** – это фундаментальная организация системы, воплощенная в ее компонентах, их взаимоотношениях друг с другом и окружением, а также принципы, направляющие ее проектирование и эволюцию.

При разработке мобильного приложения для путешествий на Flutter одним из ключевых решений является выбор архитектуры. Архитектура определяет структуру приложения, распределение ответственности между компонентами и влияет на такие аспекты, как масштабируемость, тестируемость и сопровождаемость кода. Для начала важно выбрать архитектуру для управления состоянием приложения, для этого рассмотрим основные архитектуры, применяемые при разработке Flutter-приложений, и определим наиболее подходящую для нашего проекта по внутреннему туризму в России.

MVC (Model-View-Controller)

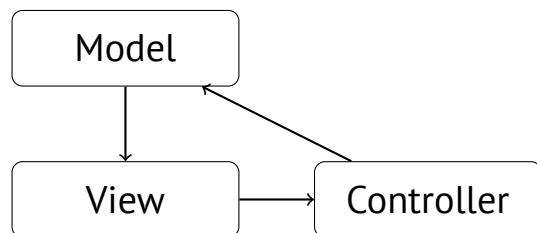


Рис. 17. Схема архитектуры MVC

MVC (Model-View-Controller) – одна из старейших и наиболее известных архитектурных парадигм. В этой архитектуре:

- **Model** – представляет данные и бизнес-логику приложения. В контексте нашего туристического приложения это будут классы, представляющие туристические маршруты, достопримечательности, отзывы пользователей и другие бизнес-сущности.
- **View** – отвечает за отображение данных пользователю. В Flutter это будут виджеты, отображающие карты, списки маршрутов, профили пользователей.
- **Controller** – обрабатывает пользовательский ввод, взаимодействует с моделью и обновляет представление. Здесь будут находиться обработчики нажатий кнопок, логика фильтрации маршрутов и т.д.

Плюсы MVC для нашего приложения:

- Простота внедрения и понимания – для небольшой команды разработчиков это упростит координацию.

- Хорошо подходит для простых экранов приложения, таких как страницы с описанием достопримечательностей или профили пользователей.
- Легко адаптировать существующие знания о MVC из других платформ.

Минусы MVC для нашего приложения:

- Controller может стать «массивным» (так называемый "Massive View Controller"), особенно в экранах с большим количеством интерактивных элементов, например, при создании и редактировании маршрута.
- Тестируемость Controller затруднена из-за тесной связи с View.
- Масштабируемость ограничена – при добавлении новых функций (например, офлайн-режим для маршрутов) архитектура может стать запутанной.

Применимость в приложении для путешествий: MVC может хорошо работать на начальных этапах разработки, но по мере добавления функциональности (например, интеграции геолокации, офлайн-режима, социальных функций для обмена маршрутами) структура может стать громоздкой. Для простых экранов, таких как просмотр информации о достопримечательностях, эта архитектура вполне подойдет, но для более сложных взаимодействий, таких как конструктор маршрутов, будут заметны ограничения.

MVP (Model-View-Presenter)

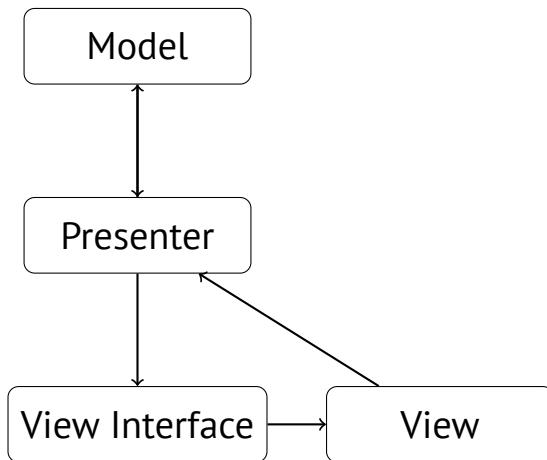


Рис. 18. Схема архитектуры MVP

MVP (Model-View-Presenter) – эволюция MVC, направленная на улучшение тестируемости и разделения ответственности:

- **Model** – аналогично MVC, содержит данные и бизнес-логику. В нашем приложении это репозитории маршрутов, сервисы геолокации, хранилище избранных мест.
- **View** – пассивный элемент, который только отображает данные и передает пользовательские события в Presenter. В Flutter это StatelessWidget или StatelessWidget с коллбэками.
- **Presenter** – содержит логику представления, реагирует на пользовательские действия, запрашивает данные от Model и обновляет View. Это будет класс, который обрабатывает действия, связанные с маршрутами, фильтрацией объектов и т.д.

Плюсы MVP для нашего приложения:

- Лучшая тестируемость – Presenter не зависит от Flutter-виджетов, что делает его легко тестируемым с помощью unit-тестов.
- Чёткое разделение ответственности – облегчает разработку сложных экранов, таких как интерактивные карты маршрутов или поиск достопримечательностей.
- Улучшенная возможность повторного использования – Presenter может использоваться с разными реализациями View при необходимости (например, для разных размеров экрана).

Минусы MVP для нашего приложения:

- Увеличение количества кода – требуется создание дополнительных интерфейсов между View и Presenter.
- Presenter всё ещё может становиться громоздким, особенно для сложных экранов с множеством функций (например, экран редактирования маршрута с добавлением фотографий, описаний, геолокаций).
- Двунаправленная зависимость между View и Presenter – не всегда идеальна для реактивных приложений.

Применимость в приложении для путешествий: MVP обеспечивает лучшую структуру, чем MVC, для реализации сложных экранов приложения. Эта архитектура хорошо подойдет для экранов с активным взаимодействием, таких как создание маршрута, поиск и фильтрация мест. Разделение логики в Presenter позволит легко реализовать различные функции, связанные с планированием путешествий, и делать код более модульным.

Однако для реактивных обновлений, например, при отслеживании текущей локации пользователя или отображении обновлений маршрута в реальном времени, могут потребоваться дополнительные паттерны.

MVVM (Model-View-ViewModel)

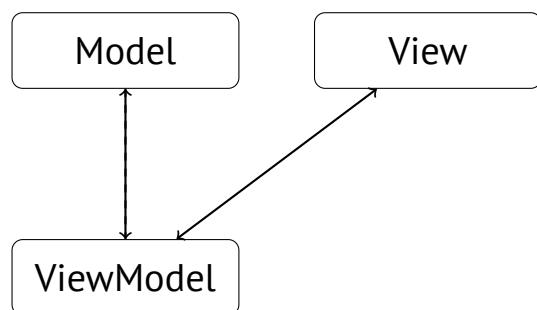


Рис. 19. Схема архитектуры MVVM

Применимость в приложении для путешествий

В нашем приложении для внутреннего туризма:

- **Model** включает классы для хранения информации о российских туристических местах, маршрутах, пользовательских данных.
- **View** представляет Flutter-виджеты для отображения списка маршрутов, карты достопримечательностей, экрана создания маршрута и т.д.
- **ViewModel** обрабатывает логику представления, такую как фильтрация маршрутов по регионам России, форматирование данных о достопримечательностях, обработка поиска и т.д.

Преимущества MVVM для нашего приложения

- **Двусторонняя привязка данных** – позволяет автоматически синхронизировать View и ViewModel, что удобно при создании интерактивных элементов для редактирования маршрутов.
- **Разделение обязанностей** – четкое разделение между UI и бизнес-логикой.
- **Хорошая тестируемость** – ViewModel можно тестировать независимо от View.
- **Переиспользуемость** – ViewModel не зависит от конкретного View, можно повторно использовать для разных представлений.
- **Управление состоянием** – хорошо подходит для сложных состояний приложения, например, для отслеживания прогресса создания маршрута пользователем.

Недостатки MVVM для нашего приложения

- **Сложность реализации для простых задач** – для простых экранов, таких как просмотр деталей достопримечательности, архитектура может быть избыточной.
- **Избыточное связывание** – для небольшого приложения может привести к излишней сложности.
- **Требует использования специальных инструментов** – для полноценной двухсторонней привязки во Flutter может потребоваться дополнительные библиотеки.

- **Кривая обучения** – освоение MVVM может занять больше времени для новых разработчиков проекта.

VIPER (View, Interactor, Presenter, Entity, Router)

Описание архитектуры

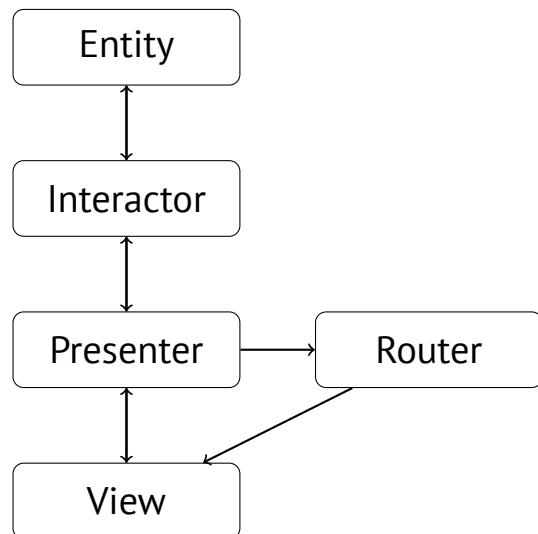


Рис. 20. Схема архитектуры VIPER

VIPER – архитектурный шаблон, который расширяет принцип разделения обязанностей до пяти компонентов:

- **View** – отвечает за отображение данных и передачу действий пользователя Presenter.
- **Interactor** – содержит бизнес-логику, связанную с манипуляцией данными.
- **Presenter** – получает действия от View, запрашивает данные от Interactor и определяет, как эти данные будут представлены в View.

- **Entity** – модели данных, используемые Interactor.
- **Router** – отвечает за навигацию между модулями приложения.

Применимость в приложении для путешествий

В контексте нашего приложения для внутреннего туризма:

- **View** – Flutter-виджеты для отображения экранов приложения: списка маршрутов, карты регионов России, деталей достопримечательностей.
- **Interactor** – содержит логику для работы с данными туристических маршрутов, обработки пользовательских запросов, загрузки информации о местах.
- **Presenter** – форматирует данные для отображения и обрабатывает пользовательские действия (например, сохранение маршрута, добавление новой точки в маршрут).
- **Entity** – модели данных для маршрутов, мест, пользовательских профилей и т.д.
- **Router** – управляет навигацией между различными частями приложения (например, переход от списка маршрутов к детальному просмотру).

Преимущества VIPER для нашего приложения

- **Высокая модульность** – каждый модуль (экран) приложения является независимым, что упрощает разработку командой.

- **Отличная тестируемость** – компоненты разделены и могут тестироваться изолированно.
- **Четкое разделение обязанностей** – улучшает понимание кода и его поддержку.
- **Масштабируемость** – хорошо подходит для управления большими, сложными приложениями.
- **Структурированная навигация** – Router обеспечивает чистую организацию навигации между экранами, что важно для приложения с множеством маршрутов и мест.

Недостатки VIPER для нашего приложения

- **Избыточность для небольших приложений** – создает значительные накладные расходы для простых функций.
- **Большое количество классов** – может привести к фрагментации кода.
- **Высокая сложность** – сложно внедрять и поддерживать для небольшой команды.
- **Затраты на обучение** – требует значительного времени на обучение новых разработчиков.
- **Избыточность для Flutter** – некоторые концепции VIPER могут дублировать встроенные механизмы Flutter.

Model-View-Intent (MVI)

Описание архитектуры

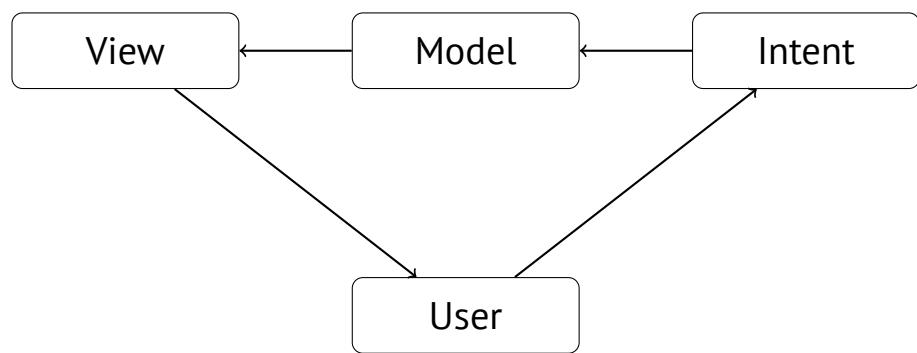


Рис. 21. Схема архитектуры MVI

MVI – односторонний и циклический архитектурный шаблон:

- **Model** – представляет состояние приложения, является неизменяемым (immutable).
- **View** – отображает состояние и отправляет намерения пользователя.
- **Intent** – представляет намерение пользователя изменить состояние.

Основная концепция MVI – циклический поток данных: View генерирует Intent, Intent преобразуется в новое состояние Model, Model отображается View.

Применимость в приложении для путешествий

В нашем приложении для внутреннего туризма:

- **Model (State)** – представляет текущее состояние приложения, включая список маршрутов, выбранные фильтры, текущее местоположение пользователя.
- **View** – отображает текущее состояние (карту с маршрутами, список достопримечательностей) и отправляет намерения пользователя (создать новый маршрут, добавить место в избранное).
- **Intent** – представляет действия пользователя, такие как выбор региона России, создание маршрута, поиск достопримечательностей.

Преимущества MVI для нашего приложения

- **Предсказуемое управление состоянием** – особенно полезно для интерактивных функций приложения, таких как фильтрация маршрутов или создание новых маршрутов.
- **Отслеживаемость изменений** – каждое изменение состояния можно отследить, что полезно для отладки и журналирования.
- **Единое направление потока данных** – упрощает понимание того, как данные перемещаются в приложении.
- **Хорошая тестируемость** – поскольку состояние неизменяемо, тестирование становится более предсказуемым.
- **Устойчивость к параллельным операциям** – хорошо подходит для обработки асинхронных операций, таких как загрузка данных о туристических местах.

Недостатки MVVI для нашего приложения

- **Излишнее создание объектов** – из-за неизменяемости состояния создаётся много новых объектов, что может влиять на производительность.
- **Сложность для простых случаев** – для простых экранов, таких как страница "О приложении", архитектура может быть избыточной.
- **Кривая обучения** – может быть сложно освоить для разработчиков, не знакомых с реактивным программированием.
- **Избыточное количество boilerplate-кода** – требуется написание большого количества шаблонного кода.

Business Logic Component (BLoC)

Описание архитектуры

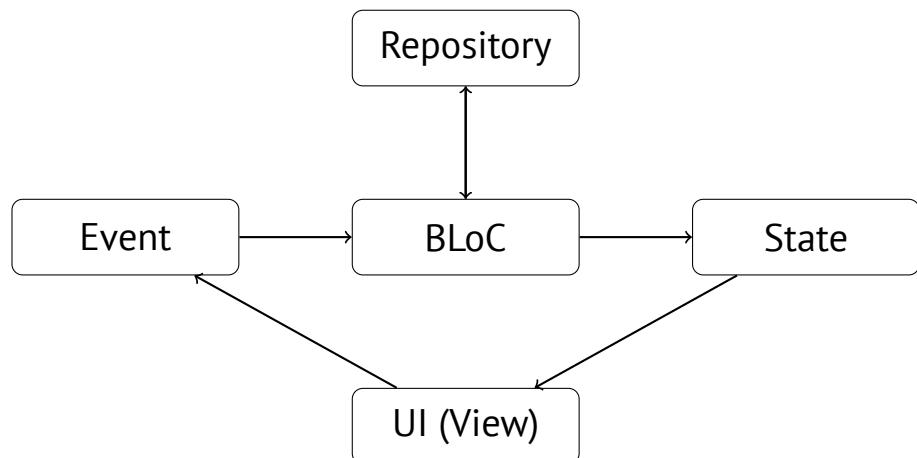


Рис. 22. Схема архитектуры BLoC

BLoC – архитектурный шаблон, специально разработанный для Flutter, основанный на реактивном программировании с использованием потоков (streams):

- **BLoC (Business Logic Component)** – компонент, который принимает события от пользовательского интерфейса, обрабатывает их и выдает новое состояние через потоки.
- **Events (События)** – действия пользователя или системные события, которые инициируют изменения в приложении.
- **States (Состояния)** – различные состояния, которые может принимать приложение в ответ на события.
- **UI (Пользовательский интерфейс)** – реактивно обновляется в зависимости от состояния, полученного от BLoC.

Применимость в приложении для путешествий

В нашем приложении для внутреннего туризма по России:

- **BLoC** – обрабатывает бизнес-логику, такую как загрузка маршрутов, фильтрация по регионам России, поиск мест и создание новых маршрутов.
- **Events** – такие события, как "Загрузить маршруты", "Применить фильтр по региону", "Сохранить новый маршрут".
- **States** – состояния, такие как "Загрузка", "Данные загружены", "Ошибка загрузки", "Фильтр применен".

- **UI** – Flutter-виджеты, отображающие карту с маршрутами, список мест, экран создания маршрута.

Преимущества BLoC для нашего приложения

- **Идеально подходит для Flutter** – разработан специально для Flutter и хорошо интегрируется с его реактивным подходом.
- **Разделение UI и бизнес-логики** – UI становится декларативным и зависит только от состояния.
- **Простота тестирования** – бизнес-логика изолирована и может быть легко протестирована.
- **Реактивное программирование** – естественным образом поддерживает асинхронные операции, которые важны для туристического приложения (загрузка данных о местах, маршрутах).
- **Управление состоянием** – обеспечивает централизованное и предсказуемое управление состоянием.
- **Большое сообщество и поддержка** – многочисленные библиотеки и документация.
- **Масштабируемость** – хорошо работает как с небольшими, так и с крупными приложениями.

Недостатки BLoC для нашего приложения

- **Избыточный код для простых случаев** – для простых экранов, таких как информационные страницы о региональных особенностях,

может быть избыточным.

- **Кривая обучения** – новичкам во Flutter может потребоваться время для освоения концепций потоков и реактивного программирования.
- **Сложность отладки** – асинхронный характер потоков может усложнить отладку.
- **Поддержка состояния** – для очень сложных состояний может потребоваться дополнительная работа.

Таким образом, для управления состоянием нами был выбран паттерн **BLoC** (Business Logic Component). BLoC помогает отделить бизнес-логику от UI, делая код более структурированным, тестируемым и предсказуемым. Подробное описание архитектуры BLoC (события, состояния, потоки) было приведено в исходной версии данной главы.

Ключевые преимущества BLoC для данного проекта:

- **Реактивность:** Идеально подходит для Flutter, позволяя UI декларативно реагировать на изменения состояния.
- **Разделение ответственности:** Четкое разделение UI от бизнес-логики. UI отправляет события в BLoC и слушает поток состояний.
- **Тестируемость:** BLoC можно легко тестировать изолированно, проверяя преобразование событий в состояния.
- **Масштабируемость:** Подходит для управления как простыми, так и сложными состояниями.

Библиотека *flutter_bloc* предоставляет удобные инструменты для реализации этого паттерна.

Ключевые архитектурные принципы и подходы

Кроме того, при всем процессе разработки важно придерживаться общепризнанных принципов проектирования программного обеспечения, таких как SOLID, KISS, DRY, и концепциях объектно-ориентированного программирования.

Ключевые архитектурные принципы

Принципы SOLID. Акроним SOLID обозначает пять основных принципов объектно-ориентированного проектирования, предложенных Робертом Мартином:

- **S – Принцип единственной ответственности (Single Responsibility Principle – SRP):** Каждый класс или модуль должен иметь одну и только одну причину для изменения. В контексте приложения это означает, например, что BLoC отвечает за управление состоянием конкретного экрана или фичи, репозиторий – за доступ к данным определенного типа, а виджет – за отображение части пользовательского интерфейса.
- **O – Принцип открытости/закрытости (Open/Closed Principle – OCP):** Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для модификации. Это достигается за счет использования абстракций и полиморфизма, позволяя добавлять новую функциональность без изменения существующего рабочего кода.
- **L – Принцип подстановки Барбары Лисков (Liskov Substitution Principle – LSP):** Если один объект заменен другим, то не должно произойти нарушения логики программы.

- **LSP**): Объекты в программе должны быть заменяемы экземплярами их подтипов без изменения правильности выполнения программы. Это важно при работе с наследованием и интерфейсами, гарантируя, что производные классы могут использоваться там, где ожидаются базовые.
- **I – Принцип разделения интерфейса (Interface Segregation Principle**
 - **ISP**): Клиенты не должны быть вынуждены зависеть от интерфейсов, которые они не используют. Предпочтительнее создавать небольшие, специфичные интерфейсы, чем один большой универсальный. Это актуально при определении контрактов для репозиториев или сервисов.

- **D – Принцип инверсии зависимостей (Dependency Inversion Principle**
 - **DIP**): Модули верхних уровней не должны зависеть от модулей нижних уровней. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Этот принцип является ключевым для построения слабо связанных систем. В приложении он реализуется через использование интерфейсов (абстракций) для репозиториев, которые внедряются в BLoC или другие бизнес-компоненты.

KISS (Keep It Simple, Stupid). Этот принцип призывает к простоте дизайна и реализации. Следует избегать излишней сложности и выбирать наиболее прямолинейное решение, которое удовлетворяет требованиям. Простой код легче понимать, тестировать и поддерживать.

DRY (Don't Repeat Yourself). Принцип "Не повторяйся" нацелен на избежание дублирования кода. Повторяющиеся участки кода следует выно-

сить в отдельные функции, классы или компоненты, чтобы обеспечить их многократное использование. Это повышает поддерживаемость и снижает вероятность ошибок при внесении изменений.

Многоуровневое разделение ответственности (Layering). Приложение структурировано с использованием многоуровневого подхода, где каждый слой имеет четко определенную зону ответственности. Это способствует лучшей организации кода, повышает его тестируемость и позволяет независимую разработку и модификацию отдельных частей системы. Основные логические слои включают:

- **Слой представления (Presentation Layer):** Отвечает за отображение информации пользователю и обработку его ввода. Включает виджеты Flutter и компоненты управления состоянием (BLoC), которые адаптируют данные для отображения и передают действия пользователя в нижележащие слои.
- **Слой бизнес-логики (Business Logic/Domain Layer):** Содержит основные бизнес-правила, сущности предметной области (модели данных) и логику их взаимодействия. Этот слой не зависит от деталей UI или источников данных. Включает модели (например, *RouteModel*, *PlaceModel*) и, при необходимости, сервисы или интеракторы, инкапсулирующие сложные операции.
- **Слой доступа к данным (Data Layer):** Обеспечивает взаимодействие с источниками данных, такими как удаленный сервер (через gRPC) или локальное хранилище. Включает реализации репозиториев, которые абстрагируют детали получения и сохранения данных от слоя бизнес-логики.

Правило зависимостей (Dependency Rule). Важным аспектом многоуровневой архитектуры является направление зависимостей. Зависимости должны быть направлены "внутрь" – от слоев, отвечающих за внешние детали (UI, база данных), к слоям, содержащим основную бизнес-логику. Слой бизнес-логики не должен знать о конкретных реализациях UI или источников данных. Это достигается за счет использования абстракций (интерфейсов) и принципа инверсии зависимостей (DIP). Например, слой бизнес-логики (или BLoC в слое представления) зависит от интерфейса репозитория, а не от его конкретной реализации, которая предоставляется через механизм внедрения зависимостей.

Эта комбинация принципов и подходов позволяет создать гибкую, масштабируемую и поддерживаемую архитектуру для мобильного приложения "Путешествия по России".

Применение Чистой архитектуры способствует независимости от фреймворков, тестируемости бизнес-логики без UI и базы данных, а также лучшей организации кода.

Подход "Feature-First" к организации кода

Для структурирования проекта был выбран подход **Feature-First**. Это означает, что код группируется не по техническим слоям (например, все виджеты в одной папке, все модели в другой), а по функциональным модулям (фичам). Каждая фича (например, "просмотр списка маршрутов", "создание маршрута") инкапсулирует все необходимые для ее работы компоненты: UI, управление состоянием (BLoC), бизнес-логику (если она специфична для фичи и не вынесена в общие Use Cases), и модели данных.

Такой подход улучшает локализацию изменений: при модификации

фичи изменения, как правило, затрагивают только ее папку. Это также облегчает навигацию по проекту и параллельную работу нескольких разработчиков над разными фичами. Внутри каждой фичи код может быть дополнительно структурирован по слоям (например, *ui*, *bloc*, *domain*, *data*).

Внедрение зависимостей (DI) с `yx_scope`

Для управления зависимостями (Dependency Injection, DI) в проекте был выбран фреймворк `yx_scope`. Разработанный в Яндексе, `yx_scope` представляет собой DI-фреймворк, нацеленный на упорядочивание работы со скоупами⁴. в больших и многомодульных проектах.

Ключевая задача, которую решает `yx_scope` – унификация подхода к управлению зависимостями в условиях, когда разные команды или модули могут использовать различные DI-решения (например, `getIt`, `injectable`, `Riverpod`), что усложняет взаимодействие и поддержку проекта. `yx_scope` был создан с целью удовлетворить следующий набор требований: чистый Dart⁵, DI-подобный подход (не статика и не `ServiceLocator`), отсутствие кодогенерации для основной функциональности, возможность встраивания DI-контейнера в виджеты⁶, нереактивное и декларативное дерево зависимостей, однозначное поведение и жизненный цикл зависимостей, поддержка вложенных скоупов любой глубины, асинхронные зависимости и

⁴Скоуп (scope) в контексте `yx_scope` – это контейнер с набором зависимостей, который существует только определённое время. Создание и удаление отдельного скоупа происходит по заранее описанным условиям в процессе работы приложения. В приложении может существовать несколько скоупов с разными жизненными циклами.

⁵Dart – язык программирования, созданный Google, оптимизированный для клиентской разработки веб- и мобильных приложений. Является основным языком для фреймворка Flutter.

⁶Виджет (Widget) в Flutter – основной строительный блок пользовательского интерфейса. Все, что отображается на экране, является виджетом, от простого текста до сложных компоновок.

их инициализация, а также compile-safe⁷ доступ к зависимостям и защита от циклических зависимостей.

Фреймворк состоит из трёх основных библиотек:

- **yx_scope**: ядро реализации, основной «движок».
- **yx_scope_flutter**: библиотека-адаптер, позволяющая интегрировать контейнеры **yx_scope** в дерево виджетов Flutter⁸.
- **yx_scope_linter**: набор кастомных правил статического анализа (lint-правил) для дополнительной защиты от ошибок при работе с фреймворком.

Ключевые особенности и преимущества **yx_scope**:

- **Безопасность на этапе компиляции (Compile-safety)**: Гарантирует, что если код компилируется, он будет корректно работать в части DI.
- **Простота**: В базовых сценариях синтаксис и поведение схожи с другими известными DI-решениями.
- **Масштабируемость**: Скоупы легко создаются, связываются и изменяются, позволяя скрывать сложность DI за абстракциями.
- **Чистый Dart, но Flutter-friendly**: DI-контейнер не зависит от UI, но легко интегрируется с Flutter, используя привычные паттерны.

⁷Compile-safe (безопасность на этапе компиляции) – свойство системы типов или инструмента разработки, которое позволяет обнаруживать определённые классы ошибок во время компиляции программы, а не в рантайме.

⁸Flutter – это UI SDK (набор инструментов разработки пользовательского интерфейса) с открытым исходным кодом, созданный Google. Используется для создания нативных кроссплатформенных приложений из единой кодовой базы для мобильных устройств (iOS, Android), веба и десктопа.

Основные понятия в **yx_scope**:

- **Dep (зависимость)**: Контейнер для одного конкретного экземпляра зависимости.
- **Scope (скоуп)**: Контейнер, который определяет, какие зависимости (Deps) будут в нём созданы и как они будут связаны друг с другом. Это декларативное описание набора зависимостей.
- **ScopeHolder**: Сущность, которая «держит» активный экземпляр Scope. ScopeHolder отвечает за создание, существование и удаление инстанса скоупа. Важно, что ScopeHolder не является статическим, что обеспечивает локализацию скоупов.
- **ScopeProvider** и **ScopeBuilder**: Виджеты из библиотеки **yx_scope_flutter**, предназначенные для интеграции скоупов в дерево виджетов Flutter, обеспечивая доступ к зависимостям из UI.

В данном приложении **yx_scope** используется для определения нескольких уровней скоупов: глобальный *AppScope* для общеприложеческих зависимостей (например, gRPC⁹ клиент, главный репозиторий¹⁰, сервис логирования), *ContentScope* для фич, связанных с отображением основного контента, и более локальные скоупы, такие как *CreateRouteScope* для фичи создания маршрута. Эта иерархия позволяет эффективно управлять жизненным циклом зависимостей, соотнося его с жизненным циклом фич, что хорошо согласуется с Feature-First подходом к организации проекта.

⁹gRPC (gRPC Remote Procedure Calls) – это современный высокопроизводительный фреймворк RPC (удаленного вызова процедур) с открытым исходным кодом, который может работать в любой среде.

¹⁰Репозиторий (Repository) – паттерн проектирования, который разделяет логику доступа к данным от бизнес-логики, предоставляя интерфейс для работы с сущностями как с коллекцией объектов.

Взаимодействие с бэкендом: gRPC

Для сетевого взаимодействия между мобильным приложением и серверной частью был выбран протокол **gRPC**. gRPC (Google Remote Procedure Call) – это современный, высокопроизводительный фреймворк RPC с открытым исходным кодом, который может работать в любой среде.

Ключевые причины выбора gRPC:

- **Производительность:** gRPC использует HTTP/2 для передачи данных и Protocol Buffers (Protobuf) в качестве языка описания интерфейсов и формата сериализации сообщений. Protobuf обеспечивает эффективную бинарную сериализацию, что приводит к меньшему объему передаваемых данных и более низкой задержке по сравнению с текстовыми форматами, такими как JSON, используемыми в REST API.
- **Строгая типизация и контракты:** Описание сервисов и сообщений с помощью .proto файлов обеспечивает строгую типизацию данных и четкие контракты между клиентом и сервером. Это снижает вероятность ошибок интеграции и упрощает эволюцию API.
- **Кодогенерация:** Инструменты gRPC автоматически генерируют клиентский и серверный код на различных языках (включая Dart для Flutter) на основе .proto файлов. Это избавляет от необходимости писать boilerplate-код для сетевых вызовов и парсинга данных.
- **Потоковая передача (Streaming):** gRPC нативно поддерживает различные типы потоковой передачи данных (серверный, клиентский, двунаправленный), что может быть полезно для реализации интерактивных функций и передачи больших объемов данных.

Контракты взаимодействия с бэкендом описываются в `.proto` файлах, размещенных в директории `lib/src/proto/`. На их основе генерируется Dart-код для gRPC клиента, который используется в слое данных приложения.

ГЛАВА 5 РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ

После определения архитектур принципов был выполнен плавный переход к разработке мобильного приложения следуя определенным нами принципам. Первой частью было создание базовой структуры приложения.

5.1 Базовая структура приложения

Следуя выбранному нами подходу feature-first, сначала создавалась фича и только внутри нее определялись слои. Таким образом, получилась древовидная структура, где одна фича могла содержать в себе другие подфичи, но благодаря продуманному делению на фичи в проекте сложно было запутаться. Основной фичей нашего приложения, конечно, является фича с контентом, где мы получаем, фильтруем и создаем маршруты. На примере этой фичи можно рассмотреть удобство использования feature-first подхода. На рисунке 3.1 видно взаимосвязь фичей контента в мобильном приложении, а именно, что фича контента содержит фичи просмотра списка маршрутов, подробной информации о конкретном маршруте, а также создания маршрута, есть общая папка, названная *shared*, которая содержит общий код для всех фичей, такой как например слой данных, общий сконуп контента и также часть общих виджетов, которые переиспользуются в нескольких фичах.

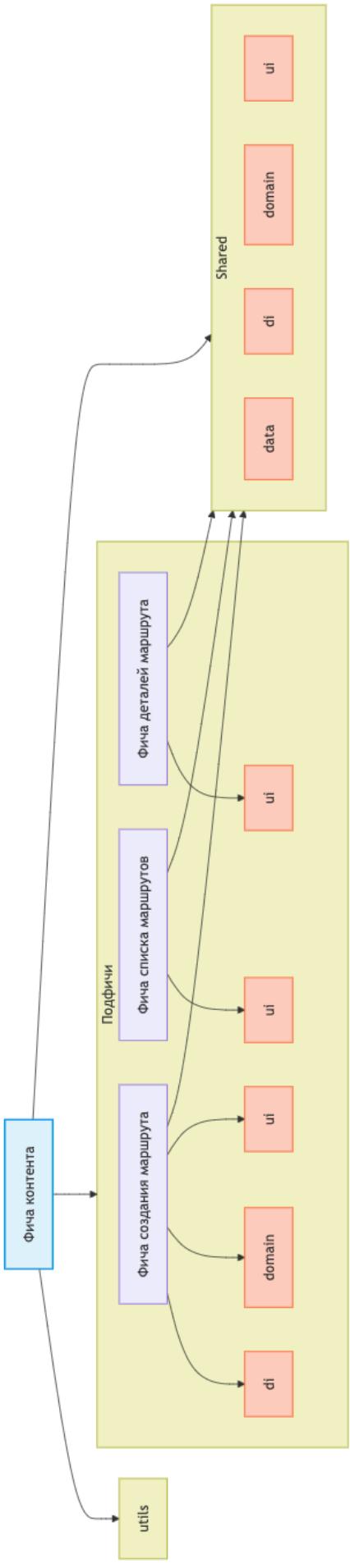


Рис. 23. Схема фичи контента приложения

5.2 Реализация слоя данных (взаимодействия с бекен-дом)

Взаимодействие с бекеном было организовано в отдельном слое - слое данных, который у каждой фичи свой, что следует feature-first подходу. Кроме того, сами запросы отправляются через отдельные классы, которые были названы ApiClient, следуя логике, что эти классы отвечают за взаимодействие с API. А обработка исключений из этих классов была вынесена в классы, которые мы назвали, также согласно чистой архитектуре Repositories (репозиториями). Конкретную реализацию слоя фичи контента можно рассмотреть на примере классов ContentApiClient и ContentApiClientImpl. Эти классы отвечают за взаимодействие с API бекенда, предоставляя методы для получения маршрутов, фильтров, загрузки изображений, а также создания мест и маршрутов. Подробный код представлен в Приложении A, файл content_api_client.dart.

Так был создан ContentApiClient, который отвечает за взаимодействие с сервисом контента бекенда. Кроме того, следуя принципам ООП была сделана абстракция и имплементация клиента апи. Апи контента имеет 5 основных методов:

1. **getRoutes** отвечает за получение списка маршрутов согласно выбранным пользователем фильтрам
2. **getRoutesFilter** возвращает фильтров для выбора пользователем
3. **uploadImages** позволяет загружать изображения пользователем на сервер

4. **createPlace** отправляет запрос на создание пользователем места маршрута
5. **createRoute** позволяет создать маршрут

Важно отметить, что апи слой не обрабатывает исключения, а отвечает только за передачу данных на следующий слой. Этим слоем в нашем приложении является репозиторий, а конкретно в случае фичи контента ContentRep и его реализация ContentRepositoryImpl. Репозиторий инкапсулирует логику получения данных от API клиента, их преобразования в модели приложения и обработку исключений. Реализацию можно увидеть в Приложении А, файл content_repository.dart.

Также была сделана абстракция репозитория, следуя принципам чистой архитектуры, также уже в репозитории происходит обработка исключений, которые может возвратить grpc. Для обработки исключений были созданы кастомные исключения (AlreadyExistsGrpcException, InvalidAr InternalServerGrpcException, UnknownGrpcException), наследуемые от BaseGrpcException. Маппинг стандартных GrpcError на эти кастомные исключения происходит с помощью расширения GrpcErrorToExcep Код данных исключений и расширения доступен в Приложении А, файл grpc_exceptions.dart.

Дальше полученные данные обрабатываются в соответствующих BLoCах.

Получение списка маршрутов и применение фильтров

Получение маршрута происходит через RoutesBloc. У него есть четыре состояния: RoutesLoadSuccess (успешная загрузка), RoutesInitial

(начальное), `RoutesLoadInProgress` (загрузка) и `RoutesLoadFailure` (ошибка). Описание этих состояний находится в Приложении А, файл `routes_state.dart`.

Этот блок обрабатывает одно событие `RoutesFetched`, сигнализирующее о необходимости загрузить список маршрутов. Код события представлен в Приложении А, файл `routes_event.dart`.

Сама логика обработки события `RoutesFetched` и преобразования состояний происходит в `RoutesBloc`. Он взаимодействует с `ContentRepository` для получения данных. Полный код блока доступен в Приложении А, файл `routes_bloc.dart`.

Так как событие всего одно, то для его обработки нужен только один метод `_onGetRoutes`

Для получения фильтра маршрутов был создан `FilterRoutesBloc`, у которого четыре состояния: `FilterRoutesInitial`, `FilterRoutesLoadInProgress`, `FilterRoutesLoadSuccess` (успешная загрузка доступных и пользовательских фильтров) и `FilterRoutesLoadFailure`. Код состояний можно найти в Приложении А, файл `filter_routes_state.dart`.

Этот блок обрабатывает уже два события: `AvailableFilterRoutesFetched` (запрос доступных фильтров) и `UserFilterRoutesChanged` (изменение пользовательских фильтров). Код событий представлен в Приложении А, файл `filter_routes_event.dart`.

Логика обновления состояния для `FilterRoutesBloc` описана в соответствующем файле. Блок обрабатывает события получения доступных фильтров и изменения пользовательских фильтров, взаимодействуя с `ContentRepository`. Код блока находится в Приложении А, файл `filter_routes_bloc.dart`.

При этом обновление списка маршрутов происходит при изменении пользователем фильтров с помощью механизма `BlocListener` в видже-

те ContentPage. При изменении фильтров в FilterRoutesBloc, инициируется событие RoutesFetched в RoutesBloc. Код данного слушателя представлен в Приложении A, файл content_page_listener.dart.

Создание маршрута

Фича создания маршрутов является более сложной фичой, чем фичи получения и фильтрацию, потому что в ней сосредоточено сразу несколько других фичей:

- Взаимодействие с плагином Яндекс.Карт, чтобы пользователь мог выбрать точку маршрута на карте
- Валидация всех полей при заполнении маршрута
- Возможность изменять порядок мест маршрута перетаскиванием на странице создания
- Правильная последовательность отправки запросов на бекенд. Потому что сначала нужно отправить запрос на создание места, потом на создание фотографий и только потом создать маршрут

Для реализации этой сложной части нами было выделено 3 блока и 1 интерактор для связи этих блоков между собой.

Так как у нас есть две страницы для создания маршрута, было решено для каждой страницы сделать свой блок, чтобы перейти на следующую страницу можно было только при правильном заполнении данных на предыдущей.

Так для заполнения общей информации о маршруте был создан CreateRouteForm. У него есть два основных состояния: CreateRouteFormEdited (форма

редактируется) и `CreateRouteFormFilled` (форма заполнена корректно). Эти состояния содержат поля для общей информации о маршруте. Код состояний представлен в Приложении А, файл `create_route_form_state.dart`.

`CreateRouteFormBloc` обрабатывает события для обновления каждого поля формы (название, описание, тип транспорта и т.д.), а также событие для сброса формы. Код событий представлен в Приложении А, файл `create_route_form_event.dart`.

и сама логика обработки событий и обновления состояния определена в `CreateRouteFormBloc`. Блок управляет состоянием формы создания общей информации о маршруте и определяет, заполнена ли форма корректно. Полный код представлен в Приложении А, файл `create_route_form_bloc.dart`.

За создание и изменение порядка точек на маршруте отвечает `CreatePointsFormBloc`. У него есть одно состояние `CreatePointsFormState`, содержащее список моделей точек маршрута (`CreatePointFormModel`). Код состояния представлен в Приложении А, файл `create_points_form_state.dart`.

И этот блок обрабатывает события добавления (`PlaceFormAddPoint`), обновления (`PlaceFormUpdatePoint`), удаления (`PlaceFormDeletePoint`) точки, изменения порядка точек (`PlaceFormReorderPoints`), а также сброса формы (`PlaceFormReset`). Код событий представлен в Приложении А, файл `create_points_form_event.dart`.

Сама логика обновления состояния и обработки событий описана в `CreatePointsFormBloc`. Блок управляет списком точек маршрута. Полный код представлен в Приложении А, файл `create_points_form_bloc.dart`.

Кроме того, есть блок `CreatePointFormBloc`, который отвечает за редактирование точки маршрута на самой странице. Он имеет состояния для управления формой редактирования отдельной точки маршрута, такие

как `CreatePathPointEditedFormModel`, `CreatePathPointFilledFormModel`,
`CreatePlacePointEditedFormModel`, и `CreatePlacePointFilledFormModel`.
Код состояний представлен в Приложении А, файл `create_point_form_state.dart`.

А также он содержит следующие события на изменения этого состояния. Код событий представлен в Приложении А, файл `create_point_form_event.dart`.

Сама логика обработки происходит в блоке `CreatePointFormBloc`. Он управляет состоянием формы редактирования отдельной точки маршрута, обрабатывает события обновления полей, управляет изображениями и определяет, заполнена ли форма точки корректно. Полный код представлен в Приложении А, файл `create_point_form_bloc.dart`.

Объединение состояния блоков `CreatePointsFormBloc` и `CreateRouteFormBloc` происходит в методе `createRoute` интерактора `CreateRouteInteractor`. Интерактор координирует процесс создания маршрута, взаимодействуя с `ContentRepository`. Полный код представлен в Приложении А, файл `create_route_interactor.dart`.

В рамках главы 4 была детально описана реализация слоя логики мобильного приложения "Путешествия по России" с применением архитектурного паттерна BLoC (Business Logic Component). Были разработаны отдельные BLoC'и для управления состоянием различных функциональных модулей, таких как получение и фильтрация списка маршрутов (`RoutesBloc`, `FilterRoutesBloc`), а также для сложной фичи создания маршрутов.

Для реализации функционала создания маршрутов были выделены блоки `CreateRouteFormBloc` для управления общей информацией о маршруте и `CreatePointsFormBloc` для работы со списком точек маршрута, включая изменение их порядка. Отдельный блок `CreatePointFormBloc` был разработан для редактирования информации по конкретной точке маршрута,

учитывая различные типы точек (опорная точка или место).

Взаимодействие между блоками `CreatePointsFormBloc` и `CreateRouteFormBloc` а также координация отправки запросов к слою данных, реализованы в интеракторе `CreateRouteInteractor`, который обеспечивает правильную последовательность операций, включая создание мест и загрузку изображений перед созданием самого маршрута.

Таким образом, в главе 4 продемонстрирован подход к разработке мобильного приложения с четким разделением ответственности между компонентами с использованием BLoC и интеракторов, что способствует модульности, тестируемости и поддерживаемости кода.

ГЛАВА 6 ПРОЕКТИРОВАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА МОБИЛЬНОГО ПРИЛОЖЕНИЯ

На основе анализа требований, проведённого в предыдущей главе, сформированы функциональные приоритеты и целевые пользовательские сценарии. Учитывая разнообразие целевой аудитории – от студентов до пожилых пользователей и людей с ограниченными возможностями – разработка интуитивно понятного, доступного и адаптивного пользовательского интерфейса приобретает критически важное значение. В данной главе рассматриваются основные этапы проектирования интерфейсов, используемые подходы, технические требования и методология, применённая при создании макетов экранов мобильного приложения. Целью главы является формирование целостного представления о структуре и логике интерфейсов до начала этапа разработки, что позволяет заранее выявить потенциальные проблемы, повысить удобство взаимодействия с приложением и сократить время на доработки в будущем.

Цели и задачи главы

- Определить основные принципы проектирования пользовательского интерфейса с учётом пользовательских историй и сценариев;
- Разработать макеты ключевых экранов мобильного приложения;
- Зафиксировать требования к элементам управления, визуальному стилю и адаптивности;
- Представить план работ по реализации интерфейсов;

- Построить диаграммы вариантов использования (use case diagrams) и активности (activity diagrams) для ключевых функций приложения.

Структура главы

- **2.1. Принципы и подходы к проектированию интерфейса** – описание используемых принципов UX/UI-дизайна, подходов к проектированию и обоснование выбора Flutter как инструмента реализации.
- **2.2. Разработка макетов интерфейса** – представление макетов ключевых экранов с комментариями к функциональным элементам, цветовой палитре и навигации.
- **2.3. Технические требования к интерфейсу** – спецификация требований к адаптивности, доступности, скорости отклика и другим техническим характеристикам UI.
- **2.4. Диаграммы вариантов использования и активности** – визуализация сценариев взаимодействия пользователей с приложением.
- **2.5. План реализации интерфейсов** – последовательность этапов разработки интерфейсов, включая создание компонентов, тестирование и возможные улучшения.

6.1. Принципы и подходы к проектированию интерфейса

Проектирование пользовательского интерфейса (UI) мобильного приложения осуществляется на основе современных принципов UX/UI-дизайна

с учётом функциональных требований, определённых в результате анализа пользовательских историй. Основной задачей данного этапа является формирование логичной, интуитивной и визуально согласованной структуры взаимодействия пользователя с функциональными возможностями приложения.

Методология проектирования

Для проектирования интерфейса использовалась итеративная модель разработки, предполагающая последовательное уточнение требований и постепенное усложнение макетов. Каждый экран и компонент интерфейса проектировался с учётом обратной связи от потенциальных пользователей и возможных ограничений платформы. Акцент был сделан на Mobile-First подход, что предполагает оптимизацию интерфейса в первую очередь под смартфоны как основное устройство использования.

Подход к организации интерфейса

Архитектура интерфейса базируется на компонентной модели. Каждый экран приложения разбит на независимые компоненты, такие как панели фильтрации, карточки маршрутов, блоки навигации и другие элементы. Компонентный подход позволяет повторно использовать элементы интерфейса в разных частях приложения, упрощает тестирование и ускоряет разработку. Организация экранов построена с применением навигационной модели «нижнее таб-меню + стековая навигация». Основные разделы приложения (поиск маршрутов, избранное, карта, профиль) доступны через нижнюю панель навигации, а переход между экранами внутри каждого

раздела осуществляется по стековой модели. Это обеспечивает понятную логику перемещения между экранами и предсказуемость поведения приложения для пользователя.

Принципы UX/UI-дизайна

Проектирование интерфейса базируется на следующих ключевых принципах:

- Консистентность (consistency) – элементы управления, цвета, шрифты и отступы унифицированы по всему приложению, что способствует снижению когнитивной нагрузки.
- Принцип визуальной иерархии – важные элементы визуально выделяются с помощью цвета, размера и расположения. Это позволяет пользователю быстро ориентироваться в интерфейсе.
- Доступность (accessibility) – реализуемые интерфейсы адаптированы для пользователей с ограниченными возможностями. Используются крупные кликабельные области, читаемые шрифты, высококонтрастные элементы. В дальнейшем планируется реализация поддержки скринридеров.
- Адаптивность – интерфейс корректно отображается на различных размерах экранов (от малых смартфонов до планшетов) за счёт применения гибких сеток и пропорционального масштабирования элементов.

Выбор технологии реализации

Для разработки пользовательского интерфейса выбрана кроссплатформенная технология Flutter, позволяющая создавать приложения под Android и iOS с единой кодовой базой. Основные причины выбора Flutter:

- Высокая производительность благодаря нативной компиляции;
- Поддержка гибкой и настраиваемой системы пользовательских интерфейсов;
- Развитая экосистема готовых виджетов и библиотек;
- Возможность реализации сложной анимации и пользовательских сценариев с минимальными затратами ресурсов.

Использование Flutter позволяет ускорить процесс прототипирования и обеспечить визуальное соответствие макетов конечной реализации.

Прототипирование

Прототипирование экранов проводилось с использованием Figma, что позволило визуализировать структуру будущего интерфейса, протестировать основные сценарии взаимодействия пользователей и согласовать макеты с участниками проектной команды до начала кодирования. На данном этапе были выявлены и скорректированы избыточные и неочевидные элементы навигации, обеспечив более чистую и логичную структуру приложения.

6.2. Разработка макетов интерфейса

Основываясь на выявленных пользовательских историях и функциональных требованиях, представленных в первой главе, настоящая глава посвящена проектированию интерфейсной части мобильного приложения для путешествий. Основной задачей данного этапа стало создание интуитивно понятного, функционального и универсального интерфейса, способного удовлетворить потребности различных категорий пользователей, включая молодёжь, семьи с детьми, пожилых людей, пользователей с ограниченными возможностями, тревел-блогеров и администраторов туристических сообществ. Проектирование охватывает следующие ключевые аспекты: определение структуры экранов, визуальных макетов, технических требований к пользовательскому интерфейсу, а также анализ пользовательских сценариев с применением диаграмм прецедентов и активностей.

Требования к пользовательскому интерфейсу

В процессе проектирования пользовательского интерфейса были выделены следующие технические и функциональные требования:

- Кроссплатформенность: Приложение должно работать как на Android, так и на iOS-устройствах. В качестве фреймворка выбран Flutter, обеспечивающий единую кодовую базу.
- Доступность: Интерфейс должен учитывать потребности пользователей с ограниченными возможностями (поддержка экранных читалок, контрастные цвета, крупные элементы навигации).

- Производительность: Быстрая загрузка экранов, оптимизированные анимации, минимальное потребление ресурсов устройства.
- Масштабируемость: Интерфейс должен адаптироваться под экраны различных размеров, поддерживать горизонтальную и вертикальную ориентацию.
- Оффлайн-режим: Предусмотрена возможность загрузки маршрутов и карты для последующего использования без подключения к интернету.
- Персонализация: Пользовательский интерфейс должен предлагать индивидуальные рекомендации и учитывать предпочтения пользователя.
- Безопасность: Все действия, связанные с персональными данными (профиль, история маршрутов, комментарии), должны быть защищены механизмами аутентификации и авторизации.

6.3. Проектирование структуры экранов и пользовательских сценариев

Проектирование структуры экранов мобильного приложения выполнялось на основе предварительно собранных требований и пользовательских историй, проанализированных в первой главе. Основной целью данного этапа стало определение ключевых пользовательских сценариев, их последовательности и логики переходов между экранами. Разработка структуры началась с выделения функциональных блоков, необходимых для ре-

ализации всех целевых сценариев использования. Были определены следующие основные группы экранов:

- Главный экран: Шапка экрана содержит быстрый поиск маршрутов, персональные предложения, рекламные интеграции, категории маршрутов. Нижний прокручиваемый блок – персонально подобранные маршруты подобранные маршруты в виде ленты постов. (см. Рис.2.2.1.)

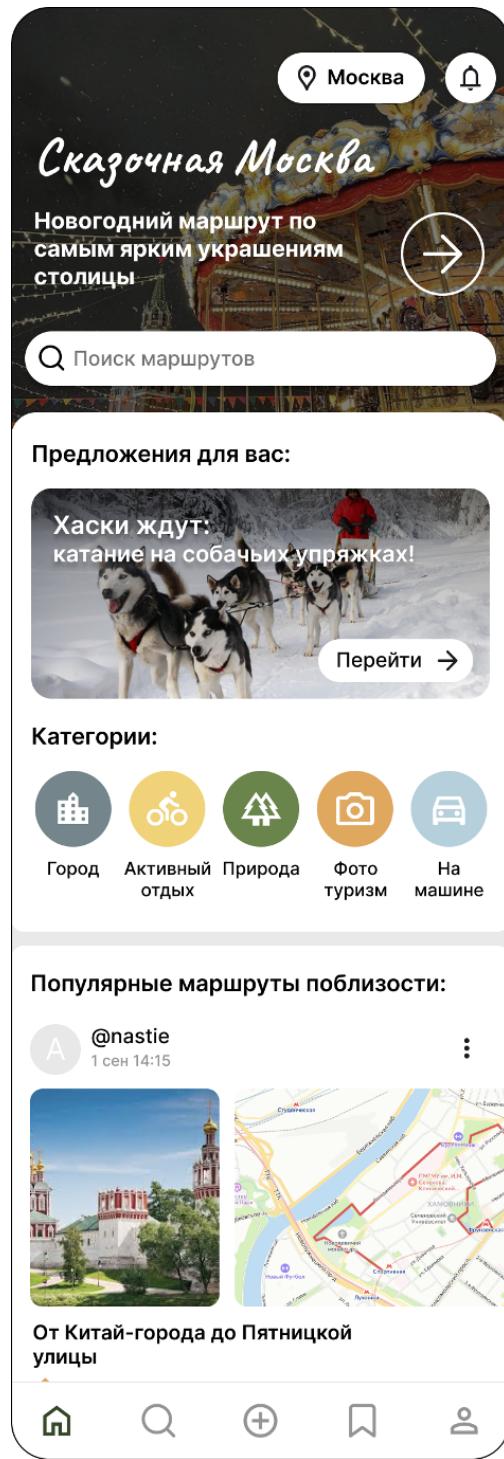


Рис. 24. Главный экран

- Экран поиска – предоставляет расширенные фильтры для подбора маршрутов по таким параметрам как расстояние и сложность. Также

содержит быстрый поиск по категориям и местам поблизости с пользователем. (см. Рис. 2.2.2.-2.2.3.)

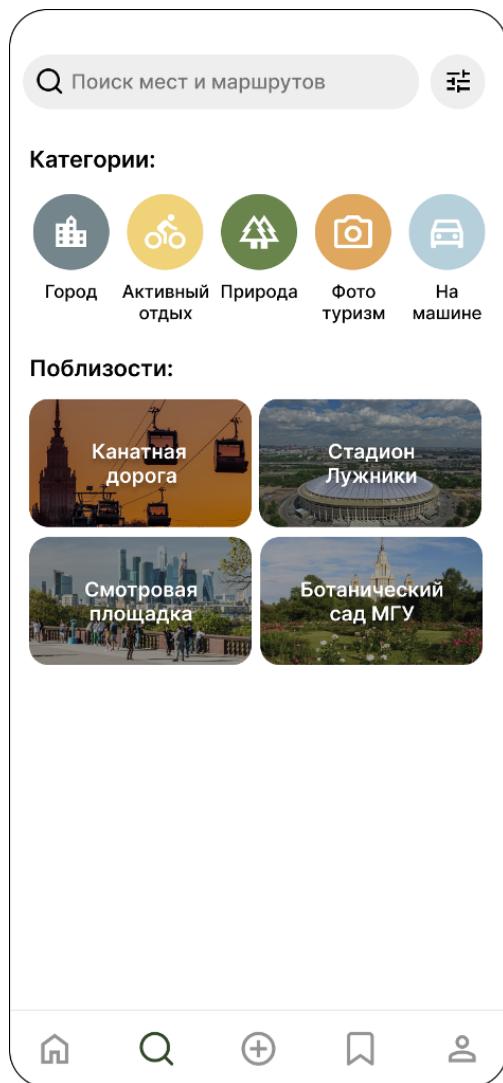


Рис. 25. Экран поиска

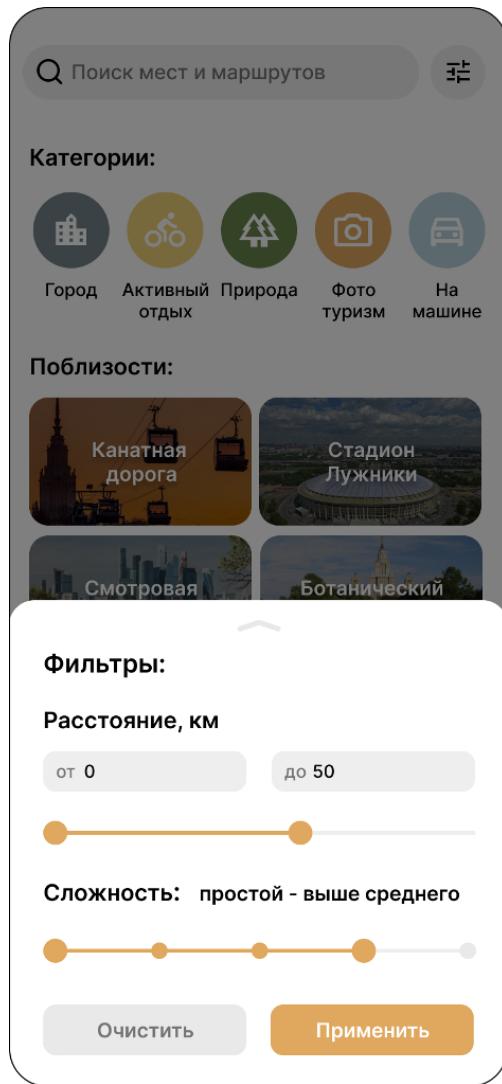


Рис. 26. Окно с фильтрами на экране поиска

- Экран маршрута – содержит карту и подробное описание маршрута. Имеет свернутый вид – с акцентом на карту и минимумом информации, и развернутый – содержит подробную информацию обо всех точках маршрута с описанием. (см. Рис. 2.2.4.-2.2.5.)

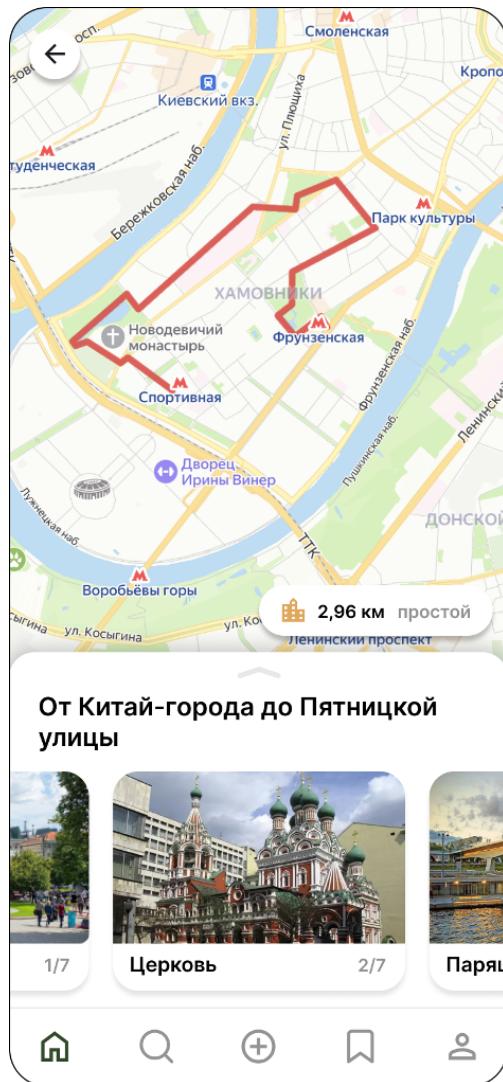


Рис. 27. Экран просмотра маршрута

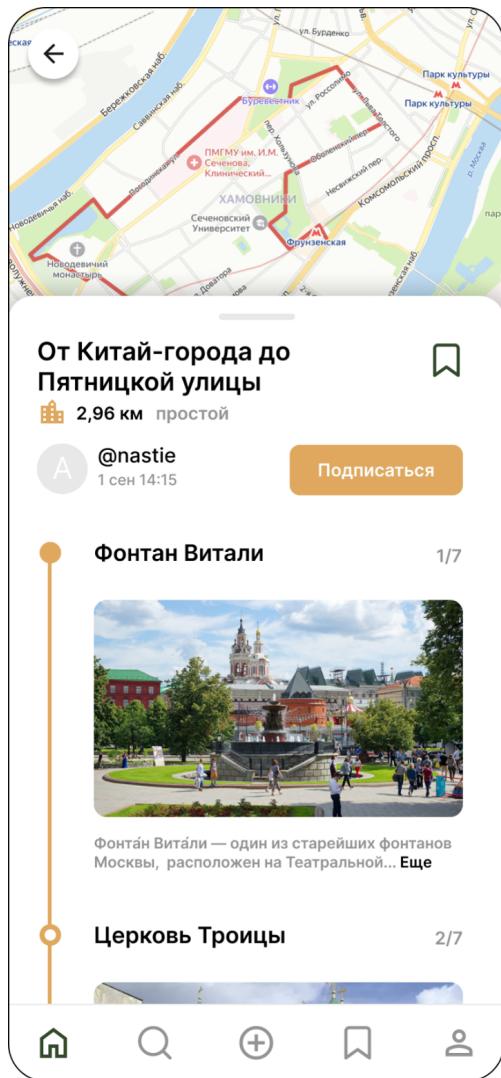


Рис. 28. Развернутый вид экрана создания маршрута

- Экраны создания маршрута. Процесс создания маршрута поделен на три экрана, чтобы не перегружать интерфейс. На первом экране пользователь задает название и выбирает категории маршрута. На втором экране доступен созданный обзор маршрута целиком. Третий экран дает возможность посмотреть маршрут на карте. (см. Рис. 2.2.6.-2.2.8.)

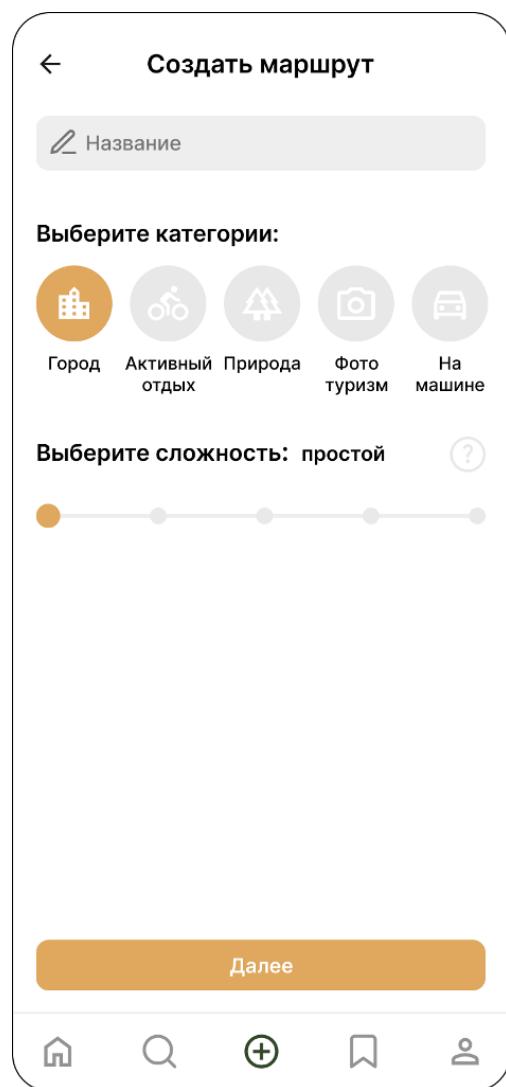


Рис. 29. Экран создания маршрута. Первый этап

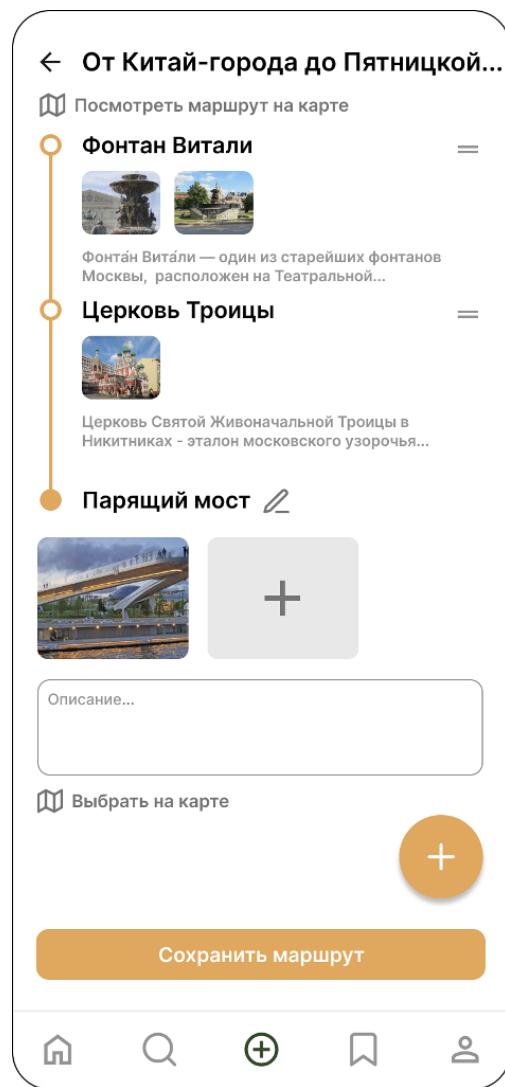


Рис. 30. Экран создания маршрута. Второй этап

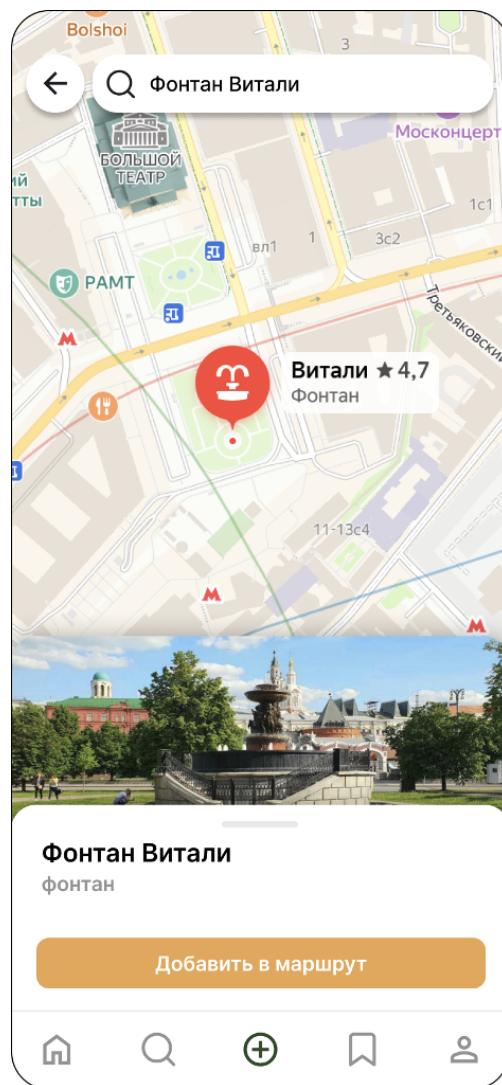


Рис. 31. Экран выбора точки маршрута на карте

- Профиль пользователя – включает историю активности, список избранных и созданных маршрутов. (см. Рис. 2.2.9.)

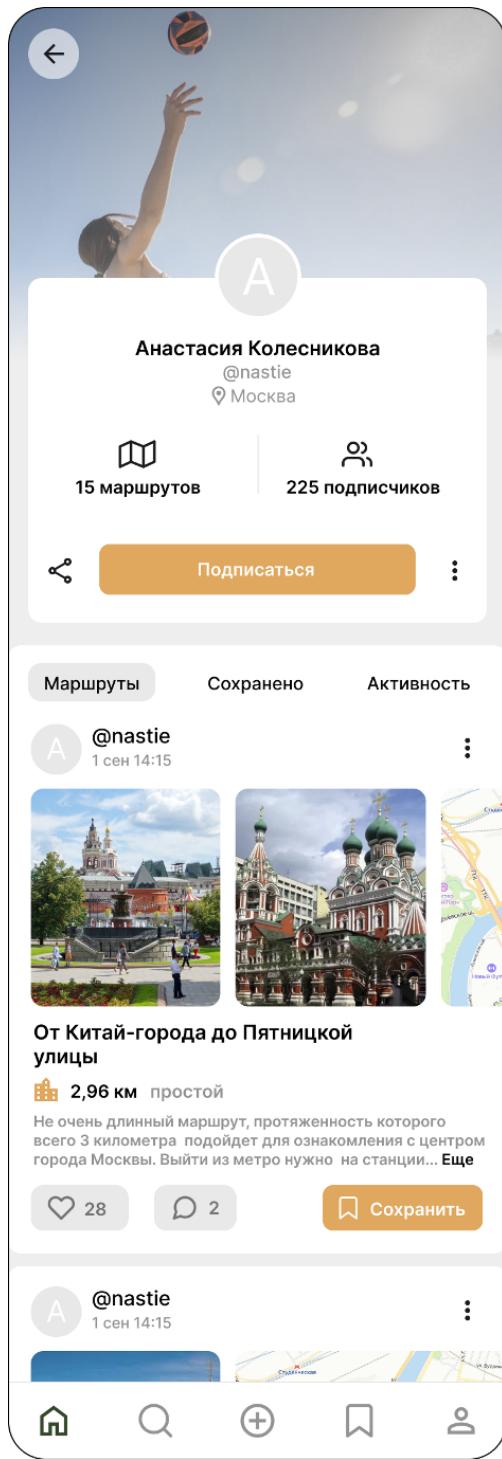


Рис. 32. Экран профиля пользователя

Особое внимание при проектировании былоделено доступности интерфейса для различных категорий пользователей: людей с ограниченными

возможностями, семей с детьми, пожилых пользователей и других. Это выразилось в адаптивной навигации, масштабируемости шрифтов, использовании контрастных цветов и возможности фильтрации контента по критериям доступности. Каждый из экранов был детально проработан с учётом следующих аспектов:

- последовательность пользовательских действий для достижения цели (например, планирование маршрута, добавление в избранное, просмотр отзывов);
- взаимодействие между экранами и логика переходов;
- минимизация числа шагов при выполнении частых сценариев;
- интуитивная организация интерфейса, соответствующая привычным шаблонам мобильных приложений.

6.4. Технические требования к интерфейсу

При проектировании пользовательского интерфейса были определены ключевые технические характеристики, необходимые для обеспечения стабильной, удобной и универсальной работы мобильного приложения. Эти требования охватывают аспекты адаптивности, доступности, отклика интерфейса, кроссплатформенности и других параметров, влияющих на пользовательский опыт.

Адаптивность и масштабируемость

- Интерфейс должен корректно отображаться на устройствах с различными размерами экранов (от смартфонов с диагональю 4 дюйма до

планшетов с диагональю 12 дюймов).

- Используется система адаптивной вёрстки с применением MediaQuery, LayoutBuilder и Flexible/Expanded компонентов во Flutter.
- Все элементы интерфейса масштабируются в зависимости от плотности пикселей (pixel ratio), что обеспечивает корректную визуализацию на устройствах с высоким разрешением (Retina, AMOLED и т.д.).

Доступность (Accessibility)

- Приложение должно быть совместимо с системами экранного озвучивания (например, TalkBack и VoiceOver).
- Элементы управления снабжаются семантическими метками (Semantics, Tooltip, aria-label).
- Цветовая палитра и контрастность соответствуют рекомендациям WCAG 2.1 (уровень AA и выше).
- Минимальный размер интерактивных элементов – 48x48 dp, согласно рекомендациям Google.
- Навигация должна быть возможна как с помощью сенсорного ввода, так и с помощью вспомогательных технологий (переключатели, клавиатуры).

Производительность и скорость отклика

- Интерфейс должен обеспечивать отклик на действия пользователя в пределах 100 мс, приоритет на отсутствие «задержек» между взаи-

модействиями.

- Используется lazy-загрузка компонентов (`ListView.builder`, `FutureBuilder`) для экономии памяти и ускорения загрузки.
- Изображения и карты кэшируются и масштабируются под размер экрана для минимизации объёма загружаемых данных.
- Предусмотрена предварительная загрузка данных и отображение `placeholder` компонентов для плавности восприятия.

Работа в офлайн-режиме

- Интерфейс должен предоставлять доступ к маршрутам, загруженным ранее, без подключения к интернету.
- Предусмотрены состояния `online`, `offline` и `reconnecting`, влияющие на доступность определённых функций.
- Компоненты синхронизируются с сервером при восстановлении подключения, с отображением индикаторов статуса.

Интерактивность и отклик на действия

- Все действия пользователя сопровождаются визуальной и/или тактильной обратной связью (нажатия, свайпы, ошибки).
- Предусмотрены состояния загрузки (`loading`), ошибки (`error`), успешного завершения действий (`success`) для всех ключевых операций.

Кроссплатформенность

- Приложение должно иметь идентичный набор функций и поведение на Android и iOS.
- Используются только те компоненты Flutter, которые обеспечивают корректную работу на обеих платформах.
- Сторонние библиотеки и плагины подбираются с учётом их поддержки в обеих системах.

Безопасность взаимодействия

- Обработка ошибок и недоступности функций реализована на уровне интерфейса – пользователь получает понятные уведомления и возможные пути решения (например, перезагрузить экран, проверить подключение).

ЗАКЛЮЧЕНИЕ

Завершённый проект «Путешествия по России» убедительно продемонстрировал, что грамотно выстроенная бизнес-модель способна превратить идею персонализированных внутренних путешествий в финансово устойчивый продукт. На основе анализа ТАМ, SAM и SOM сформирована трёхлетняя финансовая перспектива, учитывающая реальные ставки СРМ-/СРА-монетизации российского travel-рынка; расчёт показал выход на безубыточность при 50 000 MAU и рост совокупной выручки до ≈ 9 млн ₽ к третьему году. Разработанная стратегия партнёрских комиссий и нативной рекламы подтверждена классическими ОТА-метриками и подкреплена маркетинговым планом, ориентированным на семьи, маломобильных туристов, аудиторию 55+ и активных путешественников, что обеспечивает проекту коммерческую жизнеспособность и потенциал масштабирования по регионам страны.

Ключевым катализатором пользовательской вовлечённости стал тщательно проработанный дизайн: серия глубинных интервью и CJM-сессий позволила создать инклюзивный UX, где любые сегменты аудитории – от молодых бэкпекеров до пожилых туристов – интуитивно находят релевантный контент. Визуальная айдентика, разработанная в Figma, опирается на принципы Material 3 и соблюдает WCAG-стандарты доступности; масштабируемая дизайн-система с атомарными компонентами упорядочивает интерфейсы, повышая консистентность и сокращая время вывода новых функций. Такой подход не только минимизирует когнитивную нагрузку, но и формирует эмоциональную ценность бренда, подтверждая, что качественный дизайн является неотъемлемой частью конкурентного пре-

имущества.

Мобильное приложение реализовано на Flutter с архитектурой BLoC + Clean Architecture, что обеспечивает чёткое разделение слоёв и лёгкую расширяемость функционала. Ключевые модули – авторизация, построитель маршрутов, интеграция с картографическими SDK (для интерактивного отображения путевых точек), push-уведомления и социальные взаимодействия (лайки, комментарии, подписки) – объединены единым кодбейсом и покрыты модульными тестами. Асинхронное состояние управляется потоками Events/States, что гарантирует стабильные 60 fps даже на устройствах среднего сегмента и снижает тэхдолг при дальнейших обновлениях. Благодаря этому приложение уже готово к публикации в магазинах и быстрому тиражированию на новые платформы.

Надёжность всей экосистемы поддерживает микросервисное серверное ядро на Go: сервисы Auth, Profile, Content, Activity, FileStorage и Notifications взаимодействуют по строго типизированным gRPC-контрактам, а событийная шина NATS JetStream обеспечивает горизонтальную масштабируемость и реактивную обработку данных. Безопасность гарантируется OAuth 2.0, JWT-токенами и сегментацией сети; данные хранятся в PostgreSQL, Redis и MinIO, что обеспечивает баланс между консистентностью и производительностью. Полностью автоматизированный CI/CD-конвейер GitHub Actions и Helm-оркестрация в Kubernetes позволяют выкатывать новые релизы за минуты, а нагрузочные тесты подтвердили стабильную работу кластера при 1 000 rps и потреблении ≤ 1,5 ГБ RAM. Тем самым проект продемонстрировал техническую и коммерческую готовность к дальнейшему росту, открывая перед внутренним туризмом России новые горизонты цифровой трансформации.

ИСТОЧНИКИ

- [1] Яндекс.Путешествия. Аналитика ранних бронирований гостиниц: отчёт 2025. [Электронный ресурс], 2025. URL <https://travel.yandex.ru/analytics>. Дата обращения: 17 мая 2025.
- [2] С. Ньюман. *Создание микросервисов*. Питер / O'Reilly, СПб., 2-е изд. edition, 2020.
- [3] Р. Митра and И. Надараишвили. *Микросервисы: от архитектуры до релиза. Пошаговое руководство*. Питер / O'Reilly, СПб., 2021.
- [4] А. Беллемар. *Создание событийно-управляемых микросервисов: масштабирование организационных данных*. Питер / O'Reilly, СПб., 2022.
- [5] К. Индрасири and Д. Куруппу. *gRPC: запуск и эксплуатация облачных приложений*. Питер / O'Reilly, СПб., 2021.
- [6] Дж. Рocco, Р. Ландер, А. Бранд, and Дж. Харрис. *Kubernetes на практике*. Питер / O'Reilly, СПб., 2022.
- [7] Дж. Стронг and В. Лэнси. *Kubernetes в сети*. Питер / O'Reilly, СПб., 2022.
- [8] JWT.io – JSON Web Tokens. [Электронный ресурс], 2025. URL <https://jwt.io>. Дата обращения: 17 мая 2025.

- [9] Российский союз туриндустрии. Внутренний турпоток в России в 2024 году достиг исторического максимума. [Электронный ресурс], 2024. URL <https://russiatourism.ru/news/15770>. Дата обращения: 17 мая 2025.
- [10] Федеральная служба государственной статистики (Росстат). Туризм в Российской Федерации: основные показатели 2023 года. [Электронный ресурс], 2023. URL <https://rosstat.gov.ru>. Дата обращения: 17 мая 2025.
- [11] Внутренний туризм в России вырос на 10 % // Ведомости. [Электронный ресурс], 2024. URL <https://www.vedomosti.ru/tourism>. Опубликовано: 23.12.2024. Дата обращения: 17 мая 2025.
- [12] АТОР. Средние траты семей с детьми на летний отдых составляют 282 тыс. руб. [Электронный ресурс], 2025. URL <https://atorus.ru/article/>. Дата обращения: 17 мая 2025.
- [13] Outbrain. Native Advertising in Travel Industry Report 2024. [Электронный ресурс], 2024. URL <https://www.outbrain.com/native-travel-2024>. Дата обращения: 17 мая 2025.
- [14] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.
- [15] Роберт С. Мартин. Чистая архитектура, 2012. URL <https://habr.com/ru/>

[post/158955/](#). Дата обращения: 6 апреля 2025.

[16] Сергей Кольцов. Встречаем ух_scope: Di-фреймворк для работы со скопами в открытом доступе, 2025. URL <https://habr.com/ru/companies/yandex/articles/852278/>. Дата обращения: 9 апреля 2025.

Репозиторий github.com/pedrecho/vkr-activity

.github/workflows/activity-chart.yaml

```
name: Publish activity Helm Chart

on:
  push:
    tags:
      - 'chart-*'

jobs:
  helm:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Helm
        uses: azure/setup-helm@v3

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ secrets.GHCR_USERNAME }}
          password: ${{ secrets.GHCR_TOKEN }}

      - name: Extract Chart Version
        id: chart
        run:
          echo "VERSION=$(grep '^version:' activity-chart/Chart.yaml | awk '{print $2}')" >> $GITHUB_OUTPUT

      - name: Package Helm Chart
        run: helm package activity-chart

      - name: Push Helm Chart to GHCR
        run:
          helm push activity-chart-${{ steps.chart.outputs.VERSION }}.tgz
          oci://ghcr.io/${{ secrets.GHCR_USERNAME }}
```

.github/workflows/app.yaml

```
name: Build & Push activity-app to GHCR

on:
  push:
    branches: [master]
  workflow_dispatch:
```

```

jobs:
  docker:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ secrets.GHCR_USERNAME }}
          password: ${{ secrets.GHCR_TOKEN }}

      - name: Build and Push Docker Image
        uses: docker/build-push-action@v5
        with:
          context: .
          file: docker/activity-app/Dockerfile
          push: true
          tags: ghcr.io/${{ secrets.GHCR_USERNAME }}/activity-app:latest
          build-args: |
            GITHUB_TOKEN=${{ secrets.GOPRIVATE_PAT }}

```

.github/workflows/migrations.yaml

```

name: Build & Push activity-migrations to GHCR

on:
  push:
    branches: [master]
    paths:
      - 'migrations/**'
      - 'docker/activity-migrations/**'
  workflow_dispatch:

jobs:
  docker:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ secrets.GHCR_USERNAME }}
          password: ${{ secrets.GHCR_TOKEN }}

      - name: Build and Push Docker Image
        uses: docker/build-push-action@v5
        with:

```

```

    context: .
    file: docker/activity-migrations/Dockerfile
    push: true
    tags: ghcr.io/${{ secrets.GHCR_USERNAME }}/activity-
migrations:latest

Makefile

# ===== Docker image =====
IMAGE_NAME := ghcr.io/pedrecho/activity-app
DOCKER_TAG := latest

docker-build:
    docker build -t $(IMAGE_NAME):$(DOCKER_TAG) .

docker-push:
    docker push $(IMAGE_NAME):$(DOCKER_TAG)

# ===== Helm chart =====
CHART_NAME := activity-chart
CHART_VERSION := $($shell grep "^\>version:" $(CHART_NAME)/Chart.yaml | awk
"\{print $$2\}")
RELEASE_NAME := activity

helm-tag:
    git tag chart-$(CHART_VERSION)
    @echo "Created local tag: chart-$(CHART_VERSION)"
    @echo "To push it: git push origin chart-$(CHART_VERSION)"

helm-install:
    helm upgrade --install $(RELEASE_NAME) $(CHART_NAME) \
        -f ./activity-chart/values.yaml \
        -f ./activity-chart/values.secret.yaml

helm-uninstall:
    helm uninstall $(RELEASE_NAME) || true

helm-clean:
    kubectl delete all -l app=$(RELEASE_NAME) || true

```

activity-chart/Chart.yaml

```

apiVersion: v2
name: activity-chart
description: A Helm chart for deploying activity-app with PostgreSQL
version: 0.1.2

```

activity-chart/files/config.yaml

```

server:
  port: {{ .Values.activityApp.containerPort }}

logger:

```

```

level: debug

nats:
  connection:
    host: {{ .Values.nats.connection.host }}
    port: {{ .Values.nats.connection.port }}
    ssl: {{ .Values.nats.connection.ssl }}
    durable_name: {{ .Values.nats.connection.durableName }}
  topics:

postgres:
  host: {{ .Values.database.host }}
  port: {{ .Values.database.port }}
  user: {{ .Values.database.user }}
  password: {{ .Values.database.password }}
  dbname: {{ .Values.database.name }}
  ssl: false

```

activity-chart/templates/activity-app-deployment.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: activity-app
spec:
  replicas: {{ .Values.activityApp.replicaCount }}
  selector:
    matchLabels:
      app: activity-app
  template:
    metadata:
      labels:
        app: activity-app
    spec:
      serviceAccountName: {{ .Values.serviceAccount.name }}
      containers:
        - name: activity-app
          image: {{ .Values.activityApp.image }}
          imagePullPolicy: {{ .Values.activityApp.imagePullPolicy }}
          ports:
            - containerPort: {{ .Values.activityApp.containerPort }}
          command: ["./activity-app"]
          args: ["--config", "{{ .Values.activityApp.configPath }}"]
      volumeMounts:
        - name: config-volume
          mountPath: {{ .Values.activityApp.configPath }}
          subPath: config.yaml
  volumes:
    - name: config-volume
      configMap:
        name: activity-app-config

```

activity-chart/templates/activity-app-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: activity-app
spec:
  selector:
    app: activity-app
  ports:
    - port: {{ .Values.activityApp.externalPort }}
      targetPort: {{ .Values.activityApp.containerPort }}
  type: ClusterIP

```

activity-chart/templates/configmap.yaml

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: activity-app-config
  labels:
    app: activity-app

data:
  config.yaml: |-
{{ tpl (.Files.Get "files/config.yaml") . | indent 4 }}

```

activity-chart/templates/migration-job.yaml

```

apiVersion: batch/v1
kind: Job
metadata:
  name: activity-db-migrate
  labels:
    app: activity-db
  annotations:
    "helm.sh/hook": post-install,post-upgrade
    # "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    spec:
      serviceAccountName: {{ .Values.serviceAccount.name }}
      restartPolicy: OnFailure
      containers:
        - name: migrate
          image: "{{ .Values.migrations.image.repository }}:{{ .Values.migrations.image.tag }}"
          imagePullPolicy: {{ .Values.migrations.imagePullPolicy }}
          args:
            - "-source=file:///migrations"
            - "-database=$(DB_URL)"
            - "up"
          env:
            - name: DB_URL
              value: "postgres://{{ .Values.database.user }}:{{ .Values.database.password }}@{{ .Values.database.host }}:{{ .Values.database.port }}/{{ .Values.database.name }}?sslmode=disable"

```

activity-chart/templates/postgres-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: activity-db
spec:
  replicas: {{ .Values.database.replicaCount }}
  selector:
    matchLabels:
      app: activity-db
  template:
    metadata:
      labels:
        app: activity-db
    spec:
      serviceAccountName: {{ .Values.serviceAccount.name }}
      containers:
        - name: postgres
          image: {{ .Values.database.image }}
          ports:
            - containerPort: {{ .Values.database.containerPort }}
      env:
        - name: POSTGRES_PASSWORD
          value: "{{ .Values.database.password }}"
        - name: POSTGRES_DB
          value: "{{ .Values.database.name }}"
        - name: POSTGRES_USER
          value: "{{ .Values.database.user }}"
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata/data
      volumeMounts:
        - mountPath: /var/lib/postgresql/data/pgdata
          name: pgdata
  volumes:
    - name: pgdata
  persistentVolumeClaim:
    claimName: activity-db-pvc
```

activity-chart/templates/postgres-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: activity-db-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: {{ .Values.database.storage }}
```

activity-chart/templates/postgres-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: activity-db
spec:
  selector:
    app: activity-db
  ports:
    - port: {{ .Values.database.containerPort }}
```

activity-chart/values.secret.example.yaml

```
database:
  password: "<db-password>"
```

activity-chart/values.yaml

```
activityApp:
  image: ghcr.io/pedrecho/activity-app:latest
  imagePullPolicy: Always
  replicaCount: 1
  host: activity-app
  externalPort: 13804
  containerPort: 50051
  configPath: /app/config.yaml

database:
  image: ghcr.io/pedrecho/postgres:15
  replicaCount: 1
  containerPort: 5432
  host: activity-db
  port: 5432
  user: postgres
  name: activity
  storage: 1Gi

nats:
  connection:
    host: "nats"
    port: 4222
    ssl: false
    durableName: "activity-service"
  topics:

serviceAccount:
  name: vkr-ghcr-access

migrations:
  image:
    repository: ghcr.io/pedrecho/activity-migrations
    tag: latest
    imagePullPolicy: Always
```

cmd/main.go

```

package main

import (
    "flag"
    "fmt"
    "github.com/pedrecho/vkr-activity/internal/app"
)

var (
    configPath = flag.String("config", "", "config path")
)

func main() {
    flag.Parse()

    app, err := app.New(*configPath)
    if err != nil {
        panic(fmt.Errorf("init app: %w", err))
    }

    if err = app.Run(); err != nil {
        panic(err)
    }
}

```

docker/activity-app/Dockerfile

```

# Стадия сборки
FROM golang:1.24 AS builder

ARG GITHUB_TOKEN
ENV GOPRIVATE=github.com/pedrecho
ENV CGO_ENABLED=0

RUN git config --global url."https://${GITHUB_TOKEN}:x-oauth-basic@github.com/".insteadOf "https://github.com/"

WORKDIR /app
COPY ../../go.mod go.sum .
RUN go mod download

COPY ../../..
# Сборка бинарника activity-app вместо profile-app
RUN go build -o activity-app ./cmd/main.go

# Финальный образ
FROM alpine:3.19
WORKDIR /app
COPY --from=builder /app/activity-app .
RUN chmod +x ./activity-app
CMD ["/activity-app"]

```

docker/activity-migrations/Dockerfile

```
FROM migrate/migrate:v4.15.2

COPY ./migrations /migrations

ENTRYPOINT ["/migrate"]

go.mod

module github.com/pedrecho/vkr-activity

go 1.24.0

require (
    github.com/golang-jwt/jwt/v5 v5.2.2
    github.com/google/uuid v1.6.0
    github.com/pedrecho/vkr-pkg v0.0.0-20250508155451-5484a53270fe
    google.golang.org/grpc v1.70.0
    gopkg.in/yaml.v3 v3.0.1
)

require (
    github.com/klauspost/compress v1.18.0 // indirect
    github.com/lib/pq v1.10.9 // indirect
    github.com/nats-io/nats.go v1.39.1 // indirect
    github.com/nats-io/nkeys v0.4.9 // indirect
    github.com/nats-io/nuid v1.0.1 // indirect
    go.uber.org/multierr v1.10.0 // indirect
    go.uber.org/zap v1.27.0 // indirect
    golang.org/x/crypto v0.36.0 // indirect
    golang.org/x/net v0.38.0 // indirect
    golang.org/x/sys v0.31.0 // indirect
    golang.org/x/text v0.23.0 // indirect
    google.golang.org/genproto/googleapis/rpc v0.0.0-20241202173237-
19429a94021a // indirect
    google.golang.org/protobuf v1.36.5 // indirect
    gopkg.in/natefinch/lumberjack.v2 v2.2.1 // indirect
)
```

internal/app/app.go

```
package app

import (
    "fmt"
    "github.com/pedrecho/vkr-activity/internal/config"
    "github.com/pedrecho/vkr-activity/internal/grpctransport"
    "github.com/pedrecho/vkr-activity/internal/repository/postgres"
    "github.com/pedrecho/vkr-activity/internal/service"
    "github.com/pedrecho/vkr-pkg/logger"
    pb "github.com/pedrecho/vkr-pkg/pb/activity"
    "google.golang.org/grpc"
    "net"
)

type App struct {
```

```

    cfg *config.Config
}

func New(configPath string) (*App, error) {
    cfg, err := config.Load(configPath)
    if err != nil {
        return nil, fmt.Errorf("config init: %w", err)
    }

    return &App{
        cfg: cfg,
    }, nil
}

func (a *App) Run() error {
    zapSLogger, err := logger.NewZapSLogger(a.cfg.Logger)
    if err != nil {
        return fmt.Errorf("zaplogger init: %w", err)
    }

    zapSLogger.Info("content-app started")

    postgresRep, err := postgres.New(a.cfg.Postgres)
    if err != nil {
        return fmt.Errorf("init postgres: %w", err)
    }

    activitySvc := service.New(postgresRep)

    grpcSrv := grpc.NewServer()
    activityTransport := grpctransport.NewServer(zapSLogger, activitySvc)
    pb.RegisterActivityServiceServer(grpcSrv, activityTransport)

    listener, err := net.Listen("tcp", fmt.Sprintf(":%d", a.cfg.Server.Port))
    if err != nil {
        return fmt.Errorf("listen: %w", err)
    }

    zapSLogger.Infof("starting gRPC server on %d", a.cfg.Server.Port)
    return grpcSrv.Serve(listener)
}

```

internal/config/config.go

```

package config

import (
    "fmt"
    "github.com/pedrecho/vkr-pkg/db"
    "github.com/pedrecho/vkr-pkg/logger"
    "github.com/pedrecho/vkr-pkg/messaging"
    "gopkg.in/yaml.v3"
    "os"
)

type Config struct {

```

```

Server    ServerConfig      `yaml:"server"`
Logger    logger.ZapConfig  `yaml:"logger"`
Postgres  db.PostgresConfig `yaml:"postgres"`
//todo client
Nats      NatsConfig       `yaml:"nats"`
}

func Load(filename string) (*Config, error) {
    file, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("read config file: %w", err)
    }
    cfg := Config{}
    err = yaml.Unmarshal(file, &cfg)
    if err != nil {
        return nil, fmt.Errorf("unmarshal config file: %w", err)
    }

    return &cfg, err
}

type ServerConfig struct {
    Port int `yaml:"port"`
}

type NatsConfig struct {
    Connection messaging.NatsConfig `yaml:"connection"`
    Topics     TopicsConfig         `yaml:"topics"`
}

type TopicsConfig struct {
}

```

internal/grpctransport/route-comment.go

```

package grpctransport

import (
    "context"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-activity/internal/token"
    pb "github.com/pedrecho/vkr-pkg/pb/activity"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "google.golang.org/protobuf/types/known/timestamppb"
)

func (s *Server) CreateRouteComment(ctx context.Context, req
*pb.CreateRouteCommentRequest) (*pb.CreateRouteCommentResponse, error) {
    userID, err := token.ExtractUserIDFromToken(ctx)
    if err != nil {
        s.log.Error("CreateRouteComment: extract user id", "error", err)
        return nil, status.Errorf(codes.Unauthenticated, "%v", err)
    }

    routeUUID, err := uuid.Parse(req.GetRouteId())

```

```

    if err != nil {
        s.log.Error("CreateRouteComment: invalid route id", "route_id",
req.GetRouteId(), "error", err)
        return nil, status.Errorf(codes.InvalidArgument, "invalid route_id:
%v", err)
    }

    commentModel, err := s.service.CreateRouteComment(ctx, routeUUID, userID,
req.GetCommentText())
    if err != nil {
        s.log.Error("CreateRouteComment: service error", "error", err)
        return nil, status.Errorf(codes.Internal, "could not create comment:
%v", err)
    }

    pbComment := &pb.Comment{
        CommentID: commentModel.CommentID,
        RouteID:   commentModel.RouteID.String(),
        UserID:    commentModel.UserID.String(),
        Text:      commentModel.Text,
        CreatedAt: timestamppb.New(commentModel.CreatedAt),
    }

    return &pb.CreateRouteCommentResponse{Comment: pbComment}, nil
}

func (s *Server) GetRouteComments(ctx context.Context, req
*pb.GetRouteCommentsRequest) (*pb.GetRouteCommentsResponse, error) {
    routeUUID, err := uuid.Parse(req.GetRouteId())
    if err != nil {
        s.log.Error("GetRouteComments: invalid route id", "route_id",
req.GetRouteId(), "error", err)
        return nil, status.Errorf(codes.InvalidArgument, "invalid route_id:
%v", err)
    }

    comments, err := s.service.GetRouteComments(ctx, routeUUID)
    if err != nil {
        s.log.Error("GetRouteComments: service error", "route_id", routeUUID,
"error", err)
        return nil, status.Errorf(codes.Internal, "could not get comments:
%v", err)
    }

    pbComments := make([]*pb.Comment, 0, len(comments))
    for _, c := range comments {
        pbComments = append(pbComments, &pb.Comment{
            CommentID: c.CommentID,
            RouteID:   c.RouteID.String(),
            UserID:    c.UserID.String(),
            Text:      c.Text,
            CreatedAt: timestamppb.New(c.CreatedAt),
        })
    }

    return &pb.GetRouteCommentsResponse{Comments: pbComments}, nil
}

```

internal/grpctransport/route-like.go

```
package grpctransport

import (
    "context"
    "github.com/google/uuid"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "google.golang.org/protobuf/types/known/emptypb"

    "github.com/pedrecho/vkr-activity/internal/token"
    pb "github.com/pedrecho/vkr-pkg/pb/activity"
)

func (s *Server) LikeRoute(ctx context.Context, req *pb.LikeRouteRequest) (*emptypb.Empty, error) {
    userID, err := token.ExtractUserIDFromToken(ctx)
    if err != nil {
        s.log.Error("LikeRoute: extract user id", "error", err)
        return nil, status.Errorf(codes.Unauthenticated, "%v", err)
    }

    routeUUID, err := uuid.Parse(req.GetRouteId())
    if err != nil {
        s.log.Error("LikeRoute: invalid route id", "route_id",
            req.GetRouteId(), "error", err)
        return nil, status.Errorf(codes.InvalidArgument, "invalid route_id:
%v", err)
    }

    err = s.service.LikeRoute(ctx, routeUUID, userID)
    if err != nil {
        s.log.Error("LikeRoute: service error", "route_id", routeUUID,
            "user_id", userID, "error", err)
        return nil, status.Errorf(codes.Internal, "could not like route: %v",
            err)
    }

    return &emptypb.Empty{}, nil
}

func (s *Server) UnlikeRoute(ctx context.Context, req *pb.UnlikeRouteRequest) (*emptypb.Empty, error) {
    userID, err := token.ExtractUserIDFromToken(ctx)
    if err != nil {
        s.log.Error("UnlikeRoute: extract user id", "error", err)
        return nil, status.Errorf(codes.Unauthenticated, "%v", err)
    }

    routeUUID, err := uuid.Parse(req.GetRouteId())
    if err != nil {
        s.log.Error("UnlikeRoute: invalid route id", "route_id",
            req.GetRouteId(), "error", err)
        return nil, status.Errorf(codes.InvalidArgument, "invalid route_id:
%v", err)
    }
}
```

```

    }

    err = s.service.UnlikeRoute(ctx, routeUUID, userID)
    if err != nil {
        s.log.Error("UnlikeRoute: service error", "route_id", routeUUID,
"user_id", userID, "error", err)
        return nil, status.Errorf(codes.Internal, "could not unlike route:
%v", err)
    }

    return &emptypb.Empty{}, nil
}

func (s *Server) IsRouteLiked(ctx context.Context, req
*pb.IsRouteLikedRequest) (*pb.IsRouteLikedResponse, error) {
    userID, err := token.ExtractUserIDFromToken(ctx)
    if err != nil {
        s.log.Error("IsRouteLiked: extract user id", "error", err)
        return nil, status.Errorf(codes.Unauthenticated, "%v", err)
    }

    routeUUID, err := uuid.Parse(req.GetRouteId())
    if err != nil {
        s.log.Error("IsRouteLiked: invalid route id", "route_id",
req.GetRouteId(), "error", err)
        return nil, status.Errorf(codes.InvalidArgument, "invalid route_id:
%v", err)
    }

    liked, err := s.service.IsRouteLiked(ctx, routeUUID, userID)
    if err != nil {
        s.log.Error("IsRouteLiked: service error", "route_id", routeUUID,
"user_id", userID, "error", err)
        return nil, status.Errorf(codes.Internal, "could not check like
status: %v", err)
    }

    return &pb.IsRouteLikedResponse{Liked: liked}, nil
}

```

internal/grpctransport/service.go

```

package grpctransport

import (
    "github.com/pedrecho/vkr-activity/internal/service"
    "github.com/pedrecho/vkr-pkg/logger"
    pb "github.com/pedrecho/vkr-pkg/pb/activity"
)

type Server struct {
    pb.UnimplementedActivityServiceServer
    log logger.Logger
    service *service.Service
}

```

```
func NewServer(log logger.Logger, service *service.Service) *Server {
    return &Server{
        log:      log,
        service: service,
    }
}
```

internal/models/route-comment.go

```
package models

import (
    "github.com/google/uuid"
    "time"
)

type RouteComment struct {
    CommentID int64
    RouteID   uuid.UUID
    UserID    uuid.UUID
    Text      string
    CreatedAt time.Time
}
```

internal/models/route-like.go

```
package models

import (
    "time"

    "github.com/google/uuid"
)

type RouteLike struct {
    RouteID   uuid.UUID
    UserID    uuid.UUID
    CreatedAt time.Time
}
```

internal/models/token.go

```
package models

import "github.com/golang-jwt/jwt/v5"

type AccessClaims struct {
    jwt.RegisteredClaims
}
```

internal/repository/postgres/postgres.go

```

package postgres

import (
    "database/sql"
    "fmt"
    "github.com/pedrecho/vkr-pkg/db"
)

type Postgres struct {
    db *sql.DB
}

func New(cfg db.PostgresConfig) (*Postgres, error) {
    sqlDB, err := db.PostgresConnect(cfg)
    if err != nil {
        return nil, fmt.Errorf("postgres connect: %w", err)
    }
    return &Postgres{
        db: sqlDB,
    }, nil
}

```

internal/repository/postgres/route-comments.go

```

package postgres

import (
    "context"
    "fmt"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-activity/internal/models"
)

func (p *Postgres) CreateRouteComment(ctx context.Context, routeID, userID
uuid.UUID, text string) (*models.RouteComment, error) {
    const query = `

        INSERT INTO route_comments (route_id, user_id, text)
        VALUES ($1, $2, $3)
        RETURNING comment_id, created_at
    `

    var comment models.RouteComment
    comment.RouteID = routeID
    comment.UserID = userID
    comment.Text = text
    // textscan returns generated ID and timestamp
    if err := p.db.QueryRowContext(ctx, query, routeID, userID,
text).Scan(&comment.CommentID, &comment.CreatedAt); err != nil {
        return nil, fmt.Errorf("CreateRouteComment exec: %w", err)
    }
    return &comment, nil
}

func (p *Postgres) GetRouteComments(ctx context.Context, routeID uuid.UUID)
([]*models.RouteComment, error) {
    const query = `
```

```

        SELECT comment_id, route_id, user_id, text, created_at
        FROM route_comments
      WHERE route_id = $1
      ORDER BY created_at ASC
    `

rows, err := p.db.QueryContext(ctx, query, routeID)
if err != nil {
    return nil, fmt.Errorf("GetRouteComments query: %w", err)
}
defer rows.Close()

var comments []*models.RouteComment
for rows.Next() {
    var c models.RouteComment
    if err := rows.Scan(&c.CommentID, &c.RouteID, &c.UserID, &c.Text,
&c.CreatedAt); err != nil {
        return nil, fmt.Errorf("GetRouteComments scan: %w", err)
    }
    comments = append(comments, &c)
}
if err := rows.Err(); err != nil {
    return nil, fmt.Errorf("GetRouteComments rows: %w", err)
}
return comments, nil
}

```

internal/repository/postgres/route-likes.go

```

package postgres

import (
    "context"
    "database/sql"
    "errors"
    "fmt"

    "github.com/google/uuid"
)

func (p *Postgres) LikeRoute(ctx context.Context, routeID, userID uuid.UUID) error {
    const query = `
        INSERT INTO route_likes (route_id, user_id)
        VALUES ($1, $2)
        ON CONFLICT DO NOTHING
    `

    if _, err := p.db.ExecContext(ctx, query, routeID, userID); err != nil {
        return fmt.Errorf("LikeRoute exec: %w", err)
    }
    return nil
}

func (p *Postgres) UnlikeRoute(ctx context.Context, routeID, userID
uuid.UUID) error {

```

```

const query = `
    DELETE FROM route_likes
    WHERE route_id = $1 AND user_id = $2
`

if _, err := p.db.ExecContext(ctx, query, routeID, userID); err != nil {
    return fmt.Errorf("UnlikeRoute exec: %w", err)
}
return nil
}

func (p *Postgres) IsRouteLiked(ctx context.Context, routeID, userID
uuid.UUID) (bool, error) {
    const query = `
        SELECT EXISTS(
            SELECT 1 FROM route_likes WHERE route_id = $1 AND user_id = $2
        )
    `

    var exists bool
    if err := p.db.QueryRowContext(ctx, query, routeID,
userID).Scan(&exists); err != nil {
        if errors.Is(err, sql.ErrNoRows) {
            return false, nil
        }
        return false, fmt.Errorf("IsRouteLiked query: %w", err)
    }
    return exists, nil
}

```

internal/service/route-comment.go

```

package service

import (
    "context"
    "fmt"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-activity/internal/models"
)

func (s *Service) CreateRouteComment(ctx context.Context, routeID, userID
uuid.UUID, text string) (*models.RouteComment, error) {
    comment, err := s.db.CreateRouteComment(ctx, routeID, userID, text)
    if err != nil {
        return nil, fmt.Errorf("service: create route comment failed: %w",
err)
    }
    return comment, nil
}

func (s *Service) GetRouteComments(ctx context.Context, routeID uuid.UUID)
([]*models.RouteComment, error) {
    comments, err := s.db.GetRouteComments(ctx, routeID)
    if err != nil {
        return nil, fmt.Errorf("service: get route comments failed: %w", err)
    }
    return comments, nil
}

```

```
    }
    return comments, nil
}
```

internal/service/route-like.go

```
package service

import (
    "context"
    "fmt"

    "github.com/google/uuid"
)

func (s *Service) LikeRoute(ctx context.Context, routeID, userID uuid.UUID) error {
    if err := s.db.LikeRoute(ctx, routeID, userID); err != nil {
        return fmt.Errorf("service: like route failed: %w", err)
    }
    return nil
}

func (s *Service) UnlikeRoute(ctx context.Context, routeID, userID uuid.UUID) error {
    if err := s.db.UnlikeRoute(ctx, routeID, userID); err != nil {
        return fmt.Errorf("service: unlike route failed: %w", err)
    }
    return nil
}

func (s *Service) IsRouteLiked(ctx context.Context, routeID, userID uuid.UUID) (bool, error) {
    liked, err := s.db.IsRouteLiked(ctx, routeID, userID)
    if err != nil {
        return false, fmt.Errorf("service: check route liked failed: %w", err)
    }
    return liked, nil
}
```

internal/service/service.go

```
package service

import (
    "context"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-activity/internal/models"
    _ "image/jpeg"
    _ "image/png"
)

type Database interface {
    LikeRoute(ctx context.Context, routeID, userID uuid.UUID) error
    UnlikeRoute(ctx context.Context, routeID, userID uuid.UUID) error
}
```

```

    IsRouteLiked(ctx context.Context, routeID uuid.UUID) (bool,
error)
    CreateRouteComment(ctx context.Context, routeID, userID uuid.UUID, text
string) (*models.RouteComment, error)
    GetRouteComments(ctx context.Context, routeID uuid.UUID)
([]*models.RouteComment, error)
}

type Service struct {
    db Database
}

func New(db Database) *Service {
    return &Service{db: db}
}

```

internal/token/access.go

```

package token

import (
    "context"
    "fmt"
    "github.com/golang-jwt/jwt/v5"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-activity/internal/models"
    "google.golang.org/grpc/metadata"
)

// ParseAccessToken парсит access token и возвращает claims без верификации
// подписи.
func ParseAccessToken(tokenString string) (*models.AccessClaims, error) {
    claims := &models.AccessClaims{}

    _, _, err := jwt.NewParser().ParseUnverified(tokenString, claims)
    if err != nil {
        return nil, err
    }

    return claims, nil
}

func ExtractUserIDFromToken(ctx context.Context) (uuid.UUID, error) {
    md, ok := metadata.FromIncomingContext(ctx)
    if !ok {
        return uuid.Nil, fmt.Errorf("no metadata found")
    }

    authHeaders := md.Get("authorization")
    if len(authHeaders) == 0 {
        return uuid.Nil, fmt.Errorf("authorization header not found")
    }

    const bearerPrefix = "Bearer "
    tokenStr := authHeaders[0]
    if len(tokenStr) <= len(bearerPrefix) || tokenStr[:len(bearerPrefix)] !=
```

```

bearerPrefix {
    return uuid.Nil, fmt.Errorf("invalid bearer token format")
}

claims, err := ParseAccessToken(tokenStr[len(bearerPrefix):])
if err != nil {
    return uuid.Nil, fmt.Errorf("parse token: %w", err)
}

uid, err := uuid.Parse(claims.Subject)
if err != nil {
    return uuid.Nil, fmt.Errorf("invalid subject in token: %w", err)
}

return uid, nil
}

```

migrations/20250508182700_route_likes.down.sql

```
DROP TABLE IF EXISTS route_likes;
```

migrations/20250508182700_route_likes.up.sql

```

CREATE TABLE route_likes (
    route_id      UUID      NOT NULL,
    user_id       UUID      NOT NULL,
    created_at    TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    PRIMARY KEY (route_id, user_id)
);

CREATE INDEX idx_route_likes_route_id ON route_likes(route_id);

```

migrations/20250508182800_route_comments.down.sql

```
DROP TABLE IF EXISTS route_comments;
```

migrations/20250508182800_route_comments.up.sql

```

CREATE TABLE route_comments (
    comment_id    BIGSERIAL   PRIMARY KEY,
    route_id      UUID        NOT NULL,
    user_id       UUID        NOT NULL,
    text          TEXT        NOT NULL,
    created_at    TIMESTAMP WITH TIME ZONE DEFAULT
CURRENT_TIMESTAMP NOT NULL
);

CREATE INDEX idx_route_comments_route_id ON route_comments(route_id);

```

Репозиторий github.com/pedrecho/vkr-auth

.github/workflows/publish-auth-app.yaml

```
name: Build & Push auth-app to GHCR

on:
  push:
    branches: [master]
  workflow_dispatch:

jobs:
  docker:
    runs-on: ubuntu-latest

  steps:
    - name: Checkout
      uses: actions/checkout@v3

    - name: Log in to GHCR
      uses: docker/login-action@v3
      with:
        registry: ghcr.io
        username: ${{ secrets.GHCR_USERNAME }}
        password: ${{ secrets.GHCR_TOKEN }}

    - name: Build and Push Docker Image
      uses: docker/build-push-action@v5
      with:
        context: .
        file: docker/auth-app/Dockerfile
        push: true
        tags: ghcr.io/${{ secrets.GHCR_USERNAME }}/auth-app:latest
        build-args:
          GITHUB_TOKEN=${{ secrets.GOPRIVATE_PAT }}
```

.github/workflows/publish-auth-chart.yaml

```
name: Publish auth Helm Chart

on:
  push:
    tags:
      - 'chart-*'

jobs:
  helm:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Helm
```

```

uses: azure/setup-helm@v3

- name: Log in to GHCR
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${{ secrets.GHCR_USERNAME }}
    password: ${{ secrets.GHCR_TOKEN }}

- name: Extract Chart Version
  id: chart
  run: echo "VERSION=$(grep '^version:' auth-chart/Chart.yaml | awk '{print $2}')" >> $GITHUB_OUTPUT

- name: Package Helm Chart
  run: helm package auth-chart

- name: Push Helm Chart to GHCR
  run: helm push auth-chart-${{ steps.chart.outputs.VERSION }}.tgz
oci://ghcr.io/${{ secrets.GHCR_USERNAME }}

```

.github/workflows/publish-auth-migrations.yaml

```

name: Build & Push auth-migrations to GHCR

on:
  push:
    branches: [master]
    paths:
      - 'migrations/**'
      - 'docker/auth-migrations/**'
  workflow_dispatch:

jobs:
  docker:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ secrets.GHCR_USERNAME }}
          password: ${{ secrets.GHCR_TOKEN }}

      - name: Build and Push Docker Image
        uses: docker/build-push-action@v5
        with:
          context: .
          file: docker/auth-migrations/Dockerfile
          push: true
          tags: ghcr.io/${{ secrets.GHCR_USERNAME }}/auth-migrations:latest

```

Makefile

```
# ===== Docker image =====
IMAGE_NAME := ghcr.io/pedrecho/auth-app
DOCKER_TAG := latest

docker-build:
    docker build -t $(IMAGE_NAME):$(DOCKER_TAG) .

docker-push:
    docker push $(IMAGE_NAME):$(DOCKER_TAG)

# ===== Helm chart =====
CHART_NAME := auth-chart
CHART_VERSION := $($shell grep "version:" $(CHART_NAME)/Chart.yaml | awk
"{print $2}")
RELEASE_NAME := auth

helm-tag:
    git tag chart-$(CHART_VERSION)
    @echo "Created local tag: chart-$(CHART_VERSION)"
    @echo "To push it: git push origin chart-$(CHART_VERSION)"

helm-install:
    helm upgrade --install $(RELEASE_NAME) $(CHART_NAME) \
        -f ./auth-chart/values.yaml \
        -f ./auth-chart/values.secret.yaml

helm-uninstall:
    helm uninstall $(RELEASE_NAME) || true

helm-clean:
    kubectl delete all -l app=$(RELEASE_NAME) || true

# ===== DB =====
db-clean:
    kubectl delete pvc auth-db-pvc || true
```

auth-chart/Chart.yaml

```
apiVersion: v2
name: auth-chart
description: A Helm chart for deploying auth-app with PostgreSQL and Redis
version: 0.1.29
```

auth-chart/files/config.yaml

```
server:
    port: {{ .Values.authApp.containerPort }}

logger:
    level: debug
```

```

nats:
  connection:
    host: {{ .Values.nats.connection.host }}
    port: {{ .Values.nats.connection.port }}
    ssl: {{ .Values.nats.connection.ssl }}
    durable_name: {{ .Values.nats.connection.durableName }}
  topics:
    send_email_verification: {{ .Values.nats.topics.sendEmailVerification }}

postgres:
  host: {{ .Values.database.host }}
  port: {{ .Values.database.port }}
  user: {{ .Values.database.user }}
  password: {{ .Values.database.password }}
  dbname: {{ .Values.database.name }}
  ssl: false

redis:
  host: {{ .Values.cache.host }}
  port: {{ .Values.cache.port }}
  password: {{ .Values.cache.password }}
  db: 0
  ssl: false

user_client:
  host: {{ .Values.userService.host }}

```

auth-chart/templates/auth-app-deployment.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-app
spec:
  replicas: {{ .Values.authApp.replicaCount }}
  selector:
    matchLabels:
      app: auth-app
  template:
    metadata:
      labels:
        app: auth-app
    spec:
      serviceAccountName: {{ .Values.serviceAccount.name }}
      containers:
        - name: auth-app
          image: {{ .Values.authApp.image }}
          imagePullPolicy: {{ .Values.authApp.imagePullPolicy }}
          ports:
            - containerPort: {{ .Values.authApp.containerPort }}
          command: ["./auth-app"]
          args: [--config, "{{ .Values.authApp.configPath }}"]
      volumeMounts:
        - name: config-volume
          mountPath: {{ .Values.authApp.configPath }}

```

```

        subPath: config.yaml
volumes:
- name: config-volume
  configMap:
    name: auth-app-config

```

auth-chart/templates/auth-app-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: auth-app
spec:
  selector:
    app: auth-app
  ports:
    - port: {{ .Values.authApp.externalPort }}
      targetPort: {{ .Values.authApp.containerPort }}
  type: ClusterIP

```

auth-chart/templates/configmap.yaml

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: auth-app-config
  labels:
    app: auth-app

data:
  config.yaml: |-
{{ tpl (.Files.Get "files/config.yaml") . | indent 4 }}

```

auth-chart/templates/migration-job.yaml

```

apiVersion: batch/v1
kind: Job
metadata:
  name: auth-db-migrate
  labels:
    app: auth-db
  annotations:
    "helm.sh/hook": post-install
    "helm.sh/hook-delete-policy": before-hook-creation
spec:
  template:
    spec:
      serviceAccountName: {{ .Values.serviceAccount.name }}
      restartPolicy: OnFailure
      containers:
        - name: migrate
          image: "{{ .Values.migrations.image.repository }}:{{ .Values.migrations.image.tag }}"
          imagePullPolicy: {{ .Values.migrations.imagePullPolicy }}

```

```

args:
  - "-source=file:///migrations"
  - "-database=$(DB_URL)"
  - "up"
env:
  - name: DB_URL
    value: "postgres://{{ .Values.database.user }}:{{ .Values.database.password }}@{{ .Values.database.host }}:{{ .Values.database.port }}/{{ .Values.database.name }}?sslmode=disable"

```

auth-chart/templates/postgres-deployment.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-db
spec:
  replicas: {{ .Values.database.replicaCount }}
  selector:
    matchLabels:
      app: auth-db
  template:
    metadata:
      labels:
        app: auth-db
    spec:
      serviceAccountName: {{ .Values.serviceAccount.name }}
      containers:
        - name: postgres
          image: {{ .Values.database.image }}
          ports:
            - containerPort: {{ .Values.database.containerPort }}
      env:
        - name: POSTGRES_PASSWORD
          value: "{{ .Values.database.password }}"
        - name: POSTGRES_DB
          value: "{{ .Values.database.name }}"
        - name: POSTGRES_USER
          value: "{{ .Values.database.user }}"
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata/data
      volumeMounts:
        - mountPath: /var/lib/postgresql/data/pgdata
          name: pgdata
    volumes:
      - name: pgdata
      persistentVolumeClaim:
        claimName: auth-db-pvc

```

auth-chart/templates/postgres-pvc.yaml

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: auth-db-pvc

```

```
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: {{ .Values.database.storage }}
```

auth-chart/templates/postgres-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: auth-db
spec:
  selector:
    app: auth-db
  ports:
    - port: {{ .Values.database.containerPort }}
```

auth-chart/templates/redis-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-cache
spec:
  replicas: {{ .Values.cache.replicaCount }}
  selector:
    matchLabels:
      app: auth-cache
  template:
    metadata:
      labels:
        app: auth-cache
  spec:
    serviceAccountName: {{ .Values.serviceAccount.name }}
    containers:
      - name: redis
        image: {{ .Values.cache.image }}
      ports:
        - containerPort: {{ .Values.cache.containerPort }}
```

auth-chart/templates/redis-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: auth-cache
spec:
  selector:
    app: auth-cache
  ports:
    - port: {{ .Values.cache.containerPort }}
```

auth-chart/values.secret.example.yaml

```
database:  
  password: "<db-password>"  
  
cache:  
  password: "<redis-password>"
```

auth-chart/values.yaml

```
authApp:  
  image: ghcr.io/pedrecho/auth-app:latest  
  imagePullPolicy: Always  
  replicaCount: 1  
  host: auth-app  
  externalPort: 13801  
  containerPort: 50051  
  configPath: /app/config.yaml  
  
database:  
  image: ghcr.io/pedrecho/postgres:15  
  replicaCount: 1  
  containerPort: 5432  
  host: auth-db  
  port: 5432  
  user: postgres  
  name: auth  
  storage: 1Gi  
  
cache:  
  image: redis:7  
  replicaCount: 1  
  containerPort: 6379  
  host: auth-cache  
  port: 6379  
  db: 0  
  ssl: false  
  
nats:  
  connection:  
    host: "nats"  
    port: 4222  
    ssl: false  
    durableName: "auth-service"  
  topics:  
    sendEmailVerification: "notifications.emailverification"  
  
serviceAccount:  
  name: vkr-ghcr-access  
  
migrations:  
  image:  
    repository: ghcr.io/pedrecho/auth-migrations  
    tag: latest  
    imagePullPolicy: Always
```

```
userService:  
  host: "profile-app:13804"
```

cmd/main.go

```
package main  
  
import (  
    "flag"  
    "fmt"  
    "github.com/pedrecho/vkr-auth/internal/app"  
)  
  
var (  
    configPath = flag.String("config", "", "config path")  
)  
  
func main() {  
    flag.Parse()  
  
    app, err := app.New(*configPath)  
    if err != nil {  
        panic(fmt.Errorf("init app: %w", err))  
    }  
  
    if err = app.Run(); err != nil {  
        panic(err)  
    }  
}
```

docker/auth-app/Dockerfile

```
# Стадия сборки  
FROM golang:1.24 AS builder  
  
ARG GITHUB_TOKEN  
ENV GOPRIVATE=github.com/pedrecho  
ENV CGO_ENABLED=0  
  
RUN git config --global url."https://${GITHUB_TOKEN}:x-oauth-  
basic@github.com/".insteadOf "https://github.com/"  
  
WORKDIR /app  
COPY ../../go.mod go.sum ./  
RUN go mod download  
  
COPY ../../ .  
RUN go build -o auth-app ./cmd/main.go  
  
# Финальный образ  
FROM alpine:3.19
```

```
WORKDIR /app
COPY --from=builder /app/auth-app .
RUN chmod +x ./auth-app
CMD ["../auth-app"]
```

docker/auth-migrations/Dockerfile

```
FROM migrate/migrate:v4.15.2
COPY ./migrations /migrations
ENTRYPOINT ["/migrate"]
```

go.mod

```
module github.com/pedrecho/vkr-auth

go 1.24

require (
    github.com/golang-jwt/jwt/v5 v5.2.2
    github.com/google/uuid v1.6.0
    github.com/lib/pq v1.10.9
    github.com/nats-io/nats.go v1.39.1
    github.com/pedrecho/vkr-pkg v0.0.0-20250505190913-de00c4a305c8
    github.com/redis/go-redis/v9 v9.7.1
    golang.org/x/crypto v0.36.0
    google.golang.org/grpc v1.70.0
    google.golang.org/protobuf v1.36.5
    gopkg.in/yaml.v3 v3.0.1
)

require (
    github.com/cespare/xxhash/v2 v2.3.0 // indirect
    github.com/dgryski/go-rendezvous v0.0.0-20200823014737-9f7001d12a5f // indirect
    github.com/klauspost/compress v1.18.0 // indirect
    github.com/nats-io/nkeys v0.4.9 // indirect
    github.com/nats-io/nuid v1.0.1 // indirect
    go.uber.org/multierr v1.10.0 // indirect
    go.uber.org/zap v1.27.0 // indirect
    golang.org/x/net v0.38.0 // indirect
    golang.org/x/sys v0.31.0 // indirect
    golang.org/x/text v0.23.0 // indirect
    google.golang.org/genproto/googleapis/rpc v0.0.0-20241202173237-19429a94021a // indirect
    gopkg.in/natefinch/lumberjack.v2 v2.2.1 // indirect
)
```

internal/app/app.go

```
package app

import (
```

```

"fmt"
"github.com/pedrecho/vkr-auth/internal/config"
"github.com/pedrecho/vkr-auth/internal/grpctransport"
"github.com/pedrecho/vkr-auth/internal/repository/nats"
"github.com/pedrecho/vkr-auth/internal/repository/postgres"
"github.com/pedrecho/vkr-auth/internal/repository/redis"
uc "github.com/pedrecho/vkr-auth/internal/repository/user"
"github.com/pedrecho/vkr-auth/internal/service/token"
"github.com/pedrecho/vkr-auth/internal/service/user"
"github.com/pedrecho/vkr-pkg/logger"
pb "github.com/pedrecho/vkr-pkg/pb/auth"
"google.golang.org/grpc"
"net"
)

type App struct {
    cfg *config.Config
}

func New(configPath string) (*App, error) {
    cfg, err := config.Load(configPath)
    if err != nil {
        return nil, fmt.Errorf("config init: %w", err)
    }

    return &App{
        cfg: cfg,
    }, nil
}

func (a *App) Run() error {
    zapSLogger, err := logger.NewZapSLogger(a.cfg.Logger)
    if err != nil {
        return fmt.Errorf("zaplogger init: %w", err)
    }

    zapSLogger.Info("auth-app started")

    db, err := postgres.New(a.cfg.Postgres)
    if err != nil {
        return fmt.Errorf("init postgres: %w", err)
    }

    redisClient, err := redis.New(a.cfg.Redis)
    if err != nil {
        return fmt.Errorf("init redis: %w", err)
    }

    natsClient, err := nats.New(a.cfg.Nats)
    if err != nil {
        return fmt.Errorf("init nats: %w", err)
    }

    client, err := uc.New(a.cfg.UserClient)
    if err != nil {
        return fmt.Errorf("init user: %w", err)
    }
}

```

```

tokenService := token.New(a.cfg.Token, redisClient)

userService := user.New(zapSLogger, db, redisClient, natsClient,
tokenService, client)

grpcSrv := grpc.NewServer()
authTransport := grpctransport.NewAuthService(userService, zapSLogger)
pb.RegisterAuthServiceServer(grpcSrv, authTransport)

listener, err := net.Listen("tcp", fmt.Sprintf(":%d", a.cfg.Server.Port))
if err != nil {
    return fmt.Errorf("listen: %w", err)
}

zapSLogger.Infof("starting gRPC server on %d", a.cfg.Server.Port)
return grpcSrv.Serve(listener)
}

```

internal/config/config.go

```

package config

import (
    "fmt"
    "github.com/pedrecho/vkr-pkg/cache"
    "github.com/pedrecho/vkr-pkg/db"
    "github.com/pedrecho/vkr-pkg/logger"
    "github.com/pedrecho/vkr-pkg/messaging"
    "gopkg.in/yaml.v3"
    "os"
)

type Config struct {
    Server      ServerConfig      `yaml:"server"`
    Logger      logger.ZapConfig `yaml:"logger"`
    Postgres    db.PostgresConfig `yaml:"postgres"`
    Redis       cache.RedisConfig `yaml:"redis"`
    Nats        NatsConfig        `yaml:"nats"`
    Token       TokenConfig       `yaml:"token"`
    UserClient  UserServiceConfig `yaml:"user_client"`
}

func Load(filename string) (*Config, error) {
    file, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("read config file: %w", err)
    }
    cfg := Config{}
    err = yaml.Unmarshal(file, &cfg)
    if err != nil {
        return nil, fmt.Errorf("unmarshal config file: %w", err)
    }

    return &cfg, err
}

```

```

type ServerConfig struct {
    Port int `yaml:"port"`
}

type NatsConfig struct {
    Connection messaging.NatsConfig `yaml:"connection"`
    Topics     TopicsConfig        `yaml:"topics"`
}

type TopicsConfig struct {
    SendEmailVerification string `yaml:"send_email_verification"`
}

type TokenConfig struct {
    //AccessTokenTTL time.Duration `yaml:"access_duration"`
    //RefreshTokenTTL time.Duration `yaml:"refresh_duration"`
    SecretKey string `yaml:"secret_key"`
}

type UserServiceConfig struct {
    Host string `yaml:"host"`
}

```

internal/dto/token.go

```

package dto

import (
    "github.com/golang-jwt/jwt/v5"
    "time"
)

type TokenPair struct {
    AccessToken     string
    RefreshToken   string
    RefreshTokenTTL time.Duration
}

type AccessClaims struct {
    jwt.RegisteredClaims
}

type RefreshTokenData struct {
    UserID     string `json:"user_id"`
    Email      string `json:"email"`
    Fingerprint string `json:"fingerprint"`
}

```

internal/dto/user.go

```

package dto

import (
    "github.com/google/uuid"

```

```

        "time"
)

type UserStatus string

const (
    UserStatusPending    UserStatus = "pending"
    UserStatusConfirmed UserStatus = "confirmed"
)

type User struct {
    ID      uuid.UUID `db:"id"`
    Email   string    `db:"email"`
    Password string    `db:"password"`
    Status   UserStatus `db:"status"`
    CreatedAt time.Time `db:"created_at"`
    UpdatedAt time.Time `db:"updated_at"`
}

```

internal/grpctransport/auth-service.go

```

package grpctransport

import (
    "github.com/pedrecho/vkr-auth/internal/service/user"
    "github.com/pedrecho/vkr-pkg/logger"
    pb "github.com/pedrecho/vkr-pkg/pb/auth"
)

type AuthService struct {
    pb.UnimplementedAuthServiceServer
    userService *user.Service
    log         logger.Logger
}

func NewAuthService(userSvc *user.Service, log logger.Logger) *AuthService {
    return &AuthService{
        userService: userSvc,
        log:         log,
    }
}

```

internal/grpctransport/login.go

```

package grpctransport

import (
    "context"
    "errors"
    "strings"

    "github.com/pedrecho/vkr-auth/internal/service/user"
    pb "github.com/pedrecho/vkr-pkg/pb/auth"

    "google.golang.org/grpc/codes"

```

```

    "google.golang.org/grpc/status"
)

func (s *AuthService) Login(
    ctx context.Context,
    req *pb.LoginRequest,
) (*pb.LoginResponse, error) {
    if req == nil {
        return nil, status.Error(codes.InvalidArgument, "request must not be
nil")
    }
    email := strings.TrimSpace(req.Email)
    password := req.Password
    fingerprint := req.Fingerprint

    if email == "" || password == "" || fingerprint == "" {
        return nil, status.Error(codes.InvalidArgument, "email, password and
fingerprint are required")
    }

    tokenPair, err := s.userService.Login(ctx, email, password, fingerprint)
    if err != nil {
        switch {
        case errors.Is(err, user.ErrUserNotFound),
            errors.Is(err, user.ErrWrongPassword):
            return nil, status.Error(codes.Unauthenticated, "invalid email or
password")
        default:
            s.log.Errorf("login: %v", err)
            return nil, status.Error(codes.Internal, "internal server error")
        }
    }

    return &pb.LoginResponse{
        AccessToken: tokenPair.AccessToken,
        RefreshToken: tokenPair.RefreshToken,
    }, nil
}

```

internal/grpctransport/refresh-token.go

```

package grpctransport

import (
    "context"
    "errors"
    "strings"

    "github.com/pedrecho/vkr-auth/internal/service/user"
    pb "github.com/pedrecho/vkr-pkg/pb/auth"

    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

```

```

func (s *AuthService) RefreshToken(
    ctx context.Context,
    req *pb.RefreshTokenRequest,
) (*pb.RefreshTokenResponse, error) {
    if req == nil {
        return nil, status.Error(codes.InvalidArgument, "request must not be
nil")
    }
    refreshToken := strings.TrimSpace(req.RefreshToken)
    fingerprint := req.Fingerprint

    if refreshToken == "" || fingerprint == "" {
        return nil, status.Error(codes.InvalidArgument, "refresh_token and
fingerprint are required")
    }

    tokens, err := s.userService.RefreshTokens(ctx, refreshToken,
fingerprint)
    if err != nil {
        switch {
        case errors.Is(err, user.ErrRefreshTokenNotFound):
            return nil, status.Error(codes.Unauthenticated, "refresh token not
found or expired") // 401
        case errors.Is(err, user.ErrRefreshTokenCorrupted):
            return nil, status.Error(codes.Unauthenticated, "refresh token data
corrupted") // 401
        case errors.Is(err, user.ErrFingerprintMismatch):
            return nil, status.Error(codes.PermissionDenied, "fingerprint
mismatch") // 403
        default:
            s.log.Errorf("refresh token error: %v", err)
            return nil, status.Error(codes.Internal, "internal server error")
// 500
        }
    }
}

return &pb.RefreshTokenResponse{
    AccessToken: tokens.AccessToken,
    RefreshToken: tokens.RefreshToken,
}, nil
}

```

internal/grpctransport/register-step-one.go

```

package grpctransport

import (
    "context"
    "errors"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "github.com/pedrecho/vkr-auth/internal/service/user"
    pb "github.com/pedrecho/vkr-pkg/pb/auth"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

```

```

)

func (s *AuthService) RegisterStepOne(ctx context.Context, req
*pb.RegisterStepOneRequest) (*pb.RegisterStepOneResponse, error) {
    if req.GetEmail() == "" || req.GetPassword() == "" {
        s.log.Warn("missing email or password in RegisterStepOne")
        return nil, status.Error(codes.InvalidArgument, "email and password
are required")
    }

    _, err := s.userService.CreateUser(ctx, req.Email, req.Password,
dto.UserStatusPending)
    if err != nil {
        switch {
        case errors.Is(err, user.ErrUserAlreadyExists):
            s.log.Infof("user already exists: %s", req.Email)
            return nil, status.Error(codes.AlreadyExists, "user already
exists")

        case errors.Is(err, user.ErrInvalidEmail):
            s.log.Warnf("invalid email format: %s", req.Email)
            return nil, status.Error(codes.InvalidArgument, "invalid email
format")

        case errors.Is(err, user.ErrInvalidPassword):
            s.log.Warnf("invalid password for: %s", req.Email)
            return nil, status.Error(codes.InvalidArgument, "invalid password
format")

        default:
            s.log.Errorf("failed to create user: %v", err)
            return nil, status.Error(codes.Internal, "failed to create user")
        }
    }
}

return &pb.RegisterStepOneResponse{
    Message: "verification code sent",
}, nil
}

```

internal/grpctransport/resend-verification.go

```

package grpctransport

import (
    "context"
    "errors"
    "github.com/pedrecho/vkr-auth/internal/service/user"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"

    pb "github.com/pedrecho/vkr-pkg/pb/auth"
)

func (s *AuthService) ResendVerification(ctx context.Context, req
*pb.ResendVerificationRequest) (*pb.ResendVerificationResponse, error) {

```

```

email := req.GetEmail()
if email == "" {
    s.log.Warn("missing email in ResendVerification")
    return nil, status.Error(codes.InvalidArgument, "email is required")
}

err := s.userService.ResendVerificationCode(ctx, email)
if err != nil {
    switch {
    case errors.Is(err, user.ErrInvalidEmail):
        s.log.Warnf("invalid email format: %s", email)
        return nil, status.Error(codes.InvalidArgument, "invalid email
format")

    case errors.Is(err, user.ErrUserNotFound):
        s.log.Infof("user not found: %s", email)
        return nil, status.Error(codes.NotFound, "user not found")

    case errors.Is(err, user.ErrUserNotPending):
        s.log.Infof("user already confirmed: %s", email)
        return nil, status.Error(codes.FailedPrecondition, "user already
confirmed")

    default:
        s.log.Errorf("resend verification failed: %v", err)
        return nil, status.Error(codes.Internal, "failed to resend
verification")
    }
}

return &pb.ResendVerificationResponse{
    Message: "verification code resent",
}, nil
}

```

internal/grpctransport/validate-access-token.go

```

package grpctransport

import (
    "context"
    "errors"
    "strings"

    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/metadata"
    "google.golang.org/grpc/status"

    "github.com/pedrecho/vkr-auth/internal/service/user"
    pb "github.com/pedrecho/vkr-pkg/pb/auth"
)

func (s *AuthService) ValidateAccessToken(
    ctx context.Context,
    _ *pb.ValidateAccessTokenRequest,
) (*pb.ValidateAccessTokenResponse, error) {

```

```

        md, ok := metadata.FromIncomingContext(ctx)
        if !ok {
            return nil, status.Error(codes.Unauthenticated, "missing metadata") // 401
        }

        auth := md.Get("authorization")
        if len(auth) == 0 {
            return nil, status.Error(codes.Unauthenticated, "authorization header is required") // 401
        }

        parts := strings.Fields(auth[0])
        if len(parts) != 2 || !strings.EqualFold(parts[0], "Bearer") {
            return nil, status.Error(
                codes.InvalidArgument,
                `authorization header must be in format "Bearer <token>"`, // 400
            )
        }
        tokenStr := parts[1]

        if err := s.userService.ValidateAccessToken(ctx, tokenStr); err != nil {
            switch {
            case errors.Is(err, user.ErrInvalidAccessToken):
                return nil, status.Error(codes.Unauthenticated, "invalid access token") // 401
            case errors.Is(err, user.ErrTokenExpired):
                return nil, status.Error(codes.Unauthenticated, "access token expired") // 401
            default:
                s.log.Errorf("validate access token: %v", err)
                return nil, status.Error(codes.Internal, "internal server error")
            }
        }
    }

    return &pb.ValidateAccessTokenResponse{
        Message: "token is valid",
    }, nil
}

```

internal/grpctransport/verify-registration.go

```

package grpctransport

import (
    "context"
    "errors"
    "github.com/pedrecho/vkr-auth/internal/service/user"
    pb "github.com/pedrecho/vkr-pkg/pb/auth"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

func (s *AuthService) VerifyRegistration(ctx context.Context, req

```

```

*pb.VerifyRegistrationRequest) (*pb.VerifyRegistrationResponse, error) {
    email := req.GetEmail()
    code := req.GetVerificationCode()
    fingerprint := req.GetFingerprint()

    if email == "" || code == "" || fingerprint == "" {
        s.log.Warn("missing email, code or fingerprint in VerifyRegistration")
        return nil, status.Error(codes.InvalidArgument, "email, code and
fingerprint are required")
    }

    tokens, err := s.userService.VerifyUser(ctx, email, code, fingerprint)
    if err != nil {
        switch {
        case errors.Is(err, user.ErrUserNotFound):
            s.log.Infof("user not found: %s", req.GetEmail())
            return nil, status.Error(codes.NotFound, "user not found")

        case errors.Is(err, user.ErrUserNotPending):
            s.log.Infof("user already confirmed: %s", req.GetEmail())
            return nil, status.Error(codes.FailedPrecondition, "user already
confirmed")

        case errors.Is(err, user.ErrVerificationCodeNotFound):
            s.log.Infof("verification code not found for: %s", req.GetEmail())
            return nil, status.Error(codes.NotFound, "verification code not
found")

        case errors.Is(err, user.ErrInvalidVerificationCode),
            errors.Is(err, user.ErrInvalidEmail),
            errors.Is(err, user.ErrInvalidFingerprint):
            s.log.Infof("invalid input for: %s", req.GetEmail())
            return nil, status.Error(codes.InvalidArgument, "invalid input")

        default:
            s.log.Errorf("verify user failed: %v", err)
            return nil, status.Error(codes.Internal, "failed to verify user")
        }
    }

    return &pb.VerifyRegistrationResponse{
        AccessToken: tokens.AccessToken,
        RefreshToken: tokens.RefreshToken,
    }, nil
}

```

internal/repository/nats/nats.go

```

package nats

import (
    "fmt"

    "github.com/nats-io/nats.go"
    "github.com/pedrecho/vkr-auth/internal/config"
    "github.com/pedrecho/vkr-pkg/messaging"

```

```

)
type Nats struct {
    conn    *nats.Conn
    js      nats.JetStreamContext
    topics config.TopicsConfig
}

func New(cfg config.NatsConfig) (*Nats, error) {
    conn, js, err := messaging.NatsJetStreamConnect(cfg.Connection)
    if err != nil {
        return nil, fmt.Errorf("connect to nats: %w", err)
    }

    return &Nats{
        conn:    conn,
        js:      js,
        topics: cfg.Topics,
    }, nil
}

```

internal/repository/nats/verification.go

```

package nats

import (
    "context"
    "fmt"
    "github.com/pedrecho/vkr-pkg/pb/events"
    "google.golang.org/protobuf/proto"
)

func (n *Nats) PublishEmailVerification(ctx context.Context, email string,
code int32) error {
    msg := &events.SendEmailVerificationRequest{
        Email:           email,
        ConfirmationCode: code,
    }

    data, err := proto.Marshal(msg)
    if err != nil {
        return fmt.Errorf("marshal proto: %w", err)
    }

    _, err = n.js.Publish(n.topics.SendEmailVerification, data)
    if err != nil {
        return fmt.Errorf("publish message: %w", err)
    }

    return nil
}

```

internal/repository/postgres/errors.go

```

package postgres

import "errors"

var (
    ErrDuplicateEmail = errors.New("duplicate email")
    ErrUserNotFound   = errors.New("user not found")
)

```

internal/repository/postgres/postgres.go

```

package postgres

import (
    "database/sql"
    "fmt"
    "github.com/pedrecho/vkr-pkg/db"
)

type Postgres struct {
    db *sql.DB
}

func New(cfg db.PostgresConfig) (*Postgres, error) {
    sqlDB, err := db.PostgresConnect(cfg)
    if err != nil {
        return nil, fmt.Errorf("postgres connect: %w", err)
    }
    return &Postgres{
        db: sqlDB,
    }, nil
}

```

internal/repository/postgres/user-status.go

```

package postgres

import (
    "context"
    "fmt"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-auth/internal/dto"
)

func (p *Postgres) UpdateUserStatus(ctx context.Context, id uuid.UUID, status
dto.UserStatus) error {
    const query = `
        UPDATE users
        SET status = $1, updated_at = NOW()
        WHERE id = $2
    `

    res, err := p.db.ExecContext(ctx, query, status, id)
    if err != nil {
        return fmt.Errorf("update user status: %w", err)
    }
}

```

```

    }

affected, err := res.RowsAffected()
if err != nil {
    return fmt.Errorf("check update rows: %w", err)
}
if affected == 0 {
    return ErrUserNotFound
}

return nil
}

```

internal/repository/postgres/user.go

```

package postgres

import (
    "context"
    "database/sql"
    "database/sql"
    "errors"
    "fmt"
    "github.com/google/uuid"
    "github.com/lib/pq"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "time"
)

func (p *Postgres) CreateUser(ctx context.Context, email, password string,
status dto.UserStatus) (uuid.UUID, error) {
    id := uuid.New()
    now := time.Now()

    const query = `
        INSERT INTO users (id, email, password, status, created_at,
updated_at)
        VALUES ($1, $2, $3, $4, $5, $6)
    `

    _, err := p.db.ExecContext(ctx, query, id, email, password, status, now,
now)
    if err != nil {
        if isUniqueViolation(err) {
            return uuid.Nil, ErrDuplicateEmail
        }
        return uuid.Nil, fmt.Errorf("create user: %w", err)
    }

    return id, nil
}

func (p *Postgres) GetUserByEmail(ctx context.Context, email string)
(*dto.User, error) {
    const query = `
        SELECT id, email, password, status, created_at, updated_at
    `

```

```

    FROM users
    WHERE email = $1
`
```

```

row := p.db.QueryRowContext(ctx, query, email)

var user dto.User
err := row.Scan(
    &user.ID,
    &user.Email,
    &user.Password,
    &user.Status,
    &user.CreatedAt,
    &user.UpdatedAt,
)

if err != nil {
    if errors.Is(err, sql.ErrNoRows) {
        return nil, ErrUserNotFound
    }
    return nil, fmt.Errorf("get user by email: %w", err)
}

return &user, nil
}

func isUniqueViolation(err error) bool {
    var pqErr *pq.Error
    if errors.As(err, &pqErr) {
        return pqErr.Code == "23505" // uniqueViolation
    }
    return false
}

```

internal/repository/redis/errors.go

```

package redis

import (
    "errors"
)

var ErrKeyNotFound = errors.New("key not found")

```

internal/repository/redis/redis.go

```

package redis

import (
    "context"
    "errors"
    "fmt"
    "time"

    "github.com/pedrecho/vkr-pkg/cache"

```

```

    "github.com/redis/go-redis/v9"
)

type Redis struct {
    client *redis.Client
}

func New(cfg cache.RedisConfig) (*Redis, error) {
    client, err := cache.RedisConnect(cfg)
    if err != nil {
        return nil, fmt.Errorf("connect redis: %w", err)
    }

    return &Redis{client: client}, nil
}

func (r *Redis) Set(ctx context.Context, key, value string, ttl time.Duration) error {
    return r.client.Set(ctx, key, value, ttl).Err()
}

func (r *Redis) Get(ctx context.Context, key string) (string, error) {
    val, err := r.client.Get(ctx, key).Result()
    if err != nil {
        if errors.Is(err, redis.Nil) {
            return "", ErrKeyNotFound
        }
        return "", fmt.Errorf("get key from redis: %w", err)
    }

    return val, nil
}

func (r *Redis) Delete(ctx context.Context, key string) error {
    n, err := r.client.Del(ctx, key).Result()
    if err != nil {
        return fmt.Errorf("delete redis key: %w", err)
    }
    if n == 0 {
        return ErrKeyNotFound
    }
    return nil
}

```

internal/repository/user/client.go

```

package user

import (
    "context"
    "fmt"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/connectivity"
    "google.golang.org/grpc/credentials/insecure"

```

```

"github.com/pedrecho/vkr-auth/internal/config"
pb "github.com/pedrecho/vkr-pkg/pb/user"
)

const (
    dialTimeout      = 5 * time.Second
    requestTimeout   = 5 * time.Second
    maxMsgSize       = 1 << 20 // 1 MB
)

type Client struct {
    cfg  config.UserServiceConfig
    conn *grpc.ClientConn
    svc  pb.UserServiceClient
}

// New создаёт и возвращает готовый к работе gRPC-клиент UserService
func New(cfg config.UserServiceConfig) (*Client, error) {
    ctx, cancel := context.WithTimeout(context.Background(), dialTimeout)
    defer cancel()

    conn, err := grpc.Dial(
        cfg.Host,
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        // небезопасный канал, как в вашем примере
        grpc.WithDefaultCallOptions(grpc.MaxCallRecvMsgSize(maxMsgSize)),
        // по необходимости
        grpc.WithDefaultCallOptions(grpc.MaxCallSendMsgSize(maxMsgSize)),
        // по необходимости
        grpc.WithConnectParams(grpc.ConnectParams{MinConnectTimeout:
dialTimeout}), // блокировать до готовности
    )
    if err != nil {
        return nil, fmt.Errorf("user service dial: %w", err)
    }

    // Эмулируем старый WithBlock – ждём, пока канал не перейдёт в READY
    if !conn.WaitForStateChange(ctx, connectivity.Idle) && conn.GetState() !=
connectivity.Ready {
        conn.Close()
        return nil, fmt.Errorf("user service dial: connection not ready")
    }

    return &Client{
        cfg:  cfg,
        conn: conn,
        svc:  pb.NewUserServiceClient(conn),
    }, nil
}

// Close закрывает соединение
func (c *Client) Close() error {
    return c.conn.Close()
}

// CreateUser вызывает rpc CreateUser и возвращает ошибку, если что-то пошло

```

```

Но так
func (c *Client) CreateUser(ctx context.Context, userID string) error {
    // ограничиваем время выполнения RPC
    ctx, cancel := context.WithTimeout(ctx, requestTimeout)
    defer cancel()

    _, err := c.svc.CreateUser(ctx, &pb.CreateUserRequest{
        UserId: userID,
    })
    if err != nil {
        return fmt.Errorf("create user: %w", err)
    }
    return nil
}

```

internal/service/token/errors.go

```

package token

import (
    "errors"
    "fmt"
)

var (
    ErrInvalidAccessToken = fmt.Errorf("invalid access token")
    ErrTokenExpired      = fmt.Errorf("access token is expired")

    ErrRefreshTokenNotFound = errors.New("refresh token not found")
    ErrRefreshTokenCorrupted = errors.New("refresh token data corrupted")
)

```

internal/service/token/generate.go

```

package token

import (
    "context"
    "fmt"
    "github.com/golang-jwt/jwt/v5"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "time"
)

func (s *Service) GenerateTokens(
    ctx context.Context,
    userID, email, fingerprint, refreshPrefix string,
    accessTTL, refreshTTL time.Duration,
) (*dto.TokenPair, error) {
    now := time.Now()

    claims := &dto.AccessClaims{
        RegisteredClaims: jwt.RegisteredClaims{

```

```

        Subject: userID,
        ExpiresAt: jwt.NewNumericDate(now.Add(accessTTL)),
        IssuedAt: jwt.NewNumericDate(now),
    },
}

accessToken, err := jwt.NewWithClaims(jwt.SigningMethodHS256, claims).
    SignedString([]byte(s.cfg.SecretKey))
if err != nil {
    return nil, fmt.Errorf("sign access token: %w", err)
}

refreshToken := uuid.NewString()
if err := s.StoreRefreshToken(
    ctx, refreshPrefix, refreshToken, userID, email, fingerprint,
refreshTTL,
); err != nil {
    return nil, fmt.Errorf("store refresh token: %w", err)
}

return &dto.TokenPair{
    AccessToken: accessToken,
    RefreshToken: refreshToken,
}, nil
}

```

internal/service/token/refresh-token.go

```

package token

import (
    "context"
    "encoding/json"
    "errors"
    "fmt"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "github.com/redis/go-redis/v9"
    "time"
)

func (s *Service) StoreRefreshToken(
    ctx context.Context,
    refreshPrefix, refreshToken, userID, email, fingerprint string,
    ttl time.Duration,
) error {
    key := fmt.Sprintf("%s:%s", refreshPrefix, refreshToken)

    data := dto.RefreshTokenData{
        UserID:     userID,
        Email:      email,
        Fingerprint: fingerprint,
    }

    encoded, err := json.Marshal(data)
    if err != nil {
        return fmt.Errorf("marshal refresh token data: %w", err)
    }
}

```

```

    }

    return s.cache.Set(ctx, key, string(encoded), ttl)
}

func (s *Service) GetRefreshToken(
    ctx context.Context,
    refreshPrefix, refreshToken string,
) (*dto.RefreshTokenData, error) {

    key := fmt.Sprintf("%s:%s", refreshPrefix, refreshToken)

    raw, err := s.cache.Get(ctx, key)
    if err != nil {
        // redis.Nil = ключ не найден
        if errors.Is(err, redis.Nil) {
            return nil, ErrRefreshTokenNotFound
        }
        return nil, fmt.Errorf("get refresh token: %w", err)
    }

    var data dto.RefreshTokenData
    if err := json.Unmarshal([]byte(raw), &data); err != nil {
        // Данные повреждены (невалидный JSON, битый тип и т.п.)
        return nil, fmt.Errorf("%w: %v", ErrRefreshTokenCorrupted, err)
    }

    return &data, nil
}

func (s *Service) DeleteRefreshToken(
    ctx context.Context,
    refreshPrefix, refreshToken string,
) error {
    key := fmt.Sprintf("%s:%s", refreshPrefix, refreshToken)

    if err := s.cache.Delete(ctx, key); err != nil {
        return fmt.Errorf("delete refresh token from cache: %w", err)
    }

    return nil
}

```

internal/service/token/service.go

```

package token

import (
    "context"
    "github.com/pedrecho/vkr-auth/internal/config"
    "time"
)

type Cache interface {
    Set(ctx context.Context, key, value string, ttl time.Duration) error
    Get(ctx context.Context, key string) (string, error)
}

```

```

        Delete(ctx context.Context, key string) error
    }

type Service struct {
    cache Cache
    cfg   config.TokenConfig
}

func New(cfg config.TokenConfig, cache Cache) *Service {
    return &Service{
        cfg:   cfg,
        cache: cache,
    }
}

```

internal/service/token/validate.go

```

package token

import (
    "context"
    "errors"
    "github.com/golang-jwt/jwt/v5"
    "github.com/pedrecho/vkr-auth/internal/dto"
)

func (s *Service) ValidateAccessToken(
    ctx context.Context,
    tokenStr string,
) (*dto.AccessClaims, error) {
    claims := &dto.AccessClaims{}

    token, err := jwt.ParseWithClaims(tokenStr, claims, func(token
*jwt.Token) (interface{}, error) {
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, ErrInvalidAccessToken
        }
        return []byte(s.cfg.SecretKey), nil
    })
    //todo other error cases?
    if err != nil || !token.Valid {
        if errors.Is(err, jwt.ErrTokenExpired) {
            return nil, ErrTokenExpired
        }
        return nil, ErrInvalidAccessToken
    }

    return claims, nil
}

```

internal/service/user/cache.go

```

package user

import (
    "context"
    "fmt"
)

func (s *Service) StoreVerificationCode(ctx context.Context, email, code string) error {
    key := fmt.Sprintf("%s:%s", VerifyPrefix, email)
    return s.cache.Set(ctx, key, code, VerificationTTL)
}

func (s *Service) GetVerificationCode(ctx context.Context, email string) (string, error) {
    key := fmt.Sprintf("%s:%s", VerifyPrefix, email)
    return s.cache.Get(ctx, key)
}

```

internal/service/user/errors.go

```

package user

import (
    "errors"
    "fmt"
)

var (
    ErrUserAlreadyExists      = errors.New("user already exists")
    ErrInvalidEmail           = errors.New("invalid email")
    ErrInvalidPassword        = errors.New("invalid password")
    ErrUserNotFound            = errors.New("user not found")
    ErrUserNotPending          = errors.New("user is not pending")
    ErrVerificationCodeNotFound = errors.New("verification code not found")
    ErrInvalidVerificationCode = errors.New("invalid verification code")
    ErrInvalidFingerprint      = errors.New("invalid fingerprint")
    ErrWrongPassword           = errors.New("wrong password")

    // token
    ErrInvalidAccessToken      = fmt.Errorf("invalid access token")
    ErrTokenExpired             = fmt.Errorf("access token is expired")

    ErrRefreshTokenNotFound    = errors.New("refresh token not found")
    ErrRefreshTokenCorrupted   = errors.New("refresh token corrupted")
    ErrFingerprintMismatch     = errors.New("fingerprint mismatch")
)

```

internal/service/user/login.go

```

package user

import (
    "context"
    "errors"

```

```

    "fmt"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "github.com/pedrecho/vkr-auth/internal/repository/postgres"
)

func (s *Service) Login(ctx context.Context, email, password, fingerprint
string) (*dto.TokenPair, error) {
    user, err := s.storage.GetUserByEmail(ctx, email)
    if err != nil {
        if errors.Is(err, postgres.ErrUserNotFound) {
            return nil, fmt.Errorf("get user by email: %w %w", err,
ErrUserNotFound)
        }
        return nil, fmt.Errorf("get user by email: %w", err)
    }

    if err = s.CheckPassword(password, user.Password); err != nil {
        return nil, fmt.Errorf("check password: %w %w", err, ErrWrongPassword)
    }

    tokenPair, err := s.token.GenerateTokens(ctx, user.ID.String(), email,
fingerprint, RefreshPrefix, AccessTokenTTL, RefreshTokenTTL)
    if err != nil {
        return nil, fmt.Errorf("generate tokens: %w", err)
    }

    return tokenPair, nil
}

```

internal/service/user/password.go

```

package user

import (
    "fmt"
    "golang.org/x/crypto/bcrypt"
)

// TODO improve
func (s *Service) HashPassword(password string) ([]byte, error) {
    hashed, err := bcrypt.GenerateFromPassword([]byte(password),
bcrypt.DefaultCost)
    if err != nil {
        return nil, fmt.Errorf("generate from password: %w", err)
    }

    return hashed, nil
}

func (s *Service) CheckPassword(password, hashedPassword string) error {
    return bcrypt.CompareHashAndPassword([]byte(hashedPassword),
[]byte(password))
}

```

internal/service/user/refresh.go

```

package user

import (
    "context"
    "errors"
    "fmt"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "github.com/pedrecho/vkr-auth/internal/service/token"
)

func (s *Service) RefreshTokens(
    ctx context.Context,
    refreshToken string,
    fingerprint string,
) (*dto.TokenPair, error) {

    rtData, err := s.token.GetRefreshToken(ctx, RefreshPrefix, refreshToken)
    if err != nil {
        switch {
        case errors.Is(err, token.ErrRefreshTokenNotFound):
            return nil, ErrRefreshTokenNotFound
        case errors.Is(err, token.ErrRefreshTokenCorrupted):
            return nil, ErrRefreshTokenCorrupted
        default:
            return nil, fmt.Errorf("get refresh token: %w", err)
        }
    }

    if rtData.Fingerprint != fingerprint {
        return nil, ErrFingerprintMismatch
    }

    if err := s.token.DeleteRefreshToken(ctx, RefreshPrefix, refreshToken);
    err != nil {
        s.log.Warnf("delete refresh token: %v", err)
    }

    newPair, err := s.token.GenerateTokens(
        ctx,
        rtData.UserID,
        rtData.Email,
        fingerprint,
        RefreshPrefix,
        AccessTokenTTL,
        RefreshTokenTTL,
    )
    if err != nil {
        return nil, fmt.Errorf("generate tokens: %w", err)
    }

    return newPair, nil
}

```

internal/service/user/registration.go

```
package user

import (
    "context"
    "errors"
    "fmt"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "github.com/pedrecho/vkr-auth/internal/repository/postgres"
    "math/rand"
    "regexp"
    "strconv"
)

var emailRegex = regexp.MustCompile(`^[^@\s]+@[^\s]+\.[^\s]+$`)
var passwordRegex = regexp.MustCompile(`^[\p{L}\p{N}\p{P}]{8,}$`) // поддерживает латиницу, кириллицу, цифры и знаки

func (s *Service) CreateUser(ctx context.Context, email, password string,
status dto.UserStatus) (string, error) {
    if !emailRegex.MatchString(email) {
        return "", ErrInvalidEmail
    }

    if len(password) < MinPasswordLength ||
!passwordRegex.MatchString(password) {
        return "", ErrInvalidPassword
    }

    hashed, err := s.HashPassword(password)
    if err != nil {
        return "", fmt.Errorf("hash password: %w", err)
    }

    id, err := s.storage.CreateUser(ctx, email, string(hashed), status)
    if err != nil {
        if errors.Is(err, postgres.ErrDuplicateEmail) {
            return "", ErrUserAlreadyExists
        }
        return "", err
    }

    code := generateVerificationCode()
    if err := s.StoreVerificationCode(ctx, email, strconv.Itoa(int(code)));
err != nil {
        return "", fmt.Errorf("store verification code: %w", err)
    }

    if err := s.publisher.PublishEmailVerification(ctx, email, code); err !=
nil {
        return "", fmt.Errorf("publish email verification: %w", err)
    }

    return id.String(), nil
}

func (s *Service) ResendVerificationCode(ctx context.Context, email string)
error {
```

```

user, err := s.storage.GetUserByEmail(ctx, email)
if err != nil {
    if errors.Is(err, postgres.ErrUserNotFound) {
        return ErrUserNotFound
    }
    return fmt.Errorf("get user: %w", err)
}

if user.Status != dto.UserStatusPending {
    return ErrUserNotPending
}

code := generateVerificationCode()

if err := s.StoreVerificationCode(ctx, email, strconv.Itoa(int(code)));
err != nil {
    return fmt.Errorf("store verification code: %w", err)
}

if err := s.publisher.PublishEmailVerification(ctx, email, code); err != nil {
    return fmt.Errorf("publish email verification: %w", err)
}

return nil
}

func generateVerificationCode() int32 {
    return int32(100000 + rand.Intn(900000))
}

```

internal/service/user/service.go

```

package user

import (
    "context"
    "github.com/google/uuid"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "github.com/pedrecho/vkr-pkg/logger"
    "time"
)

const (
    VerifyPrefix = "verify"
    RefreshPrefix = "refresh"
)

const (
    MinPasswordLength = 8
)

const (
    VerificationTTL = 24 * time.Hour
    RefreshTokenTTL = 7 * 24 * time.Hour
    AccessTokenTTL = 15 * time.Minute
)

```

```

)

type Database interface {
    CreateUser(ctx context.Context, email, password string, status
    dto.UserStatus) (uuid.UUID, error)
    GetUserByEmail(ctx context.Context, email string) (*dto.User, error)
    UpdateUserStatus(ctx context.Context, id uuid.UUID, status
    dto.UserStatus) error
}

type Cache interface {
    Set(ctx context.Context, key, value string, ttl time.Duration) error
    Get(ctx context.Context, key string) (string, error)
    Delete(ctx context.Context, key string) error
}

type Publisher interface {
    PublishEmailVerification(ctx context.Context, email string, code int32)
    error
}

type TokenService interface {
    GenerateTokens(ctx context.Context, userID, email, fingerprint,
    refreshPrefix string, accessTTL, refreshTTL time.Duration) (*dto.TokenPair,
    error)
    ValidateAccessToken(ctx context.Context, tokenStr string)
    (*dto.AccessClaims, error)
    GetRefreshToken(ctx context.Context, refreshPrefix, refreshToken string)
    (*dto.RefreshTokenData, error)
    DeleteRefreshToken(ctx context.Context, refreshPrefix, refreshToken
    string) error
}

type Client interface {
    CreateUser(ctx context.Context, userID string) error
}

type Service struct {
    log      logger.Logger
    storage  Database
    cache    Cache
    publisher Publisher
    token    TokenService
    client   Client
}

func New(log logger.Logger, storage Database, cache Cache, publisher
Publisher, tokenService TokenService, client Client) *Service {
    return &Service{
        storage:  storage,
        cache:    cache,
        publisher: publisher,
        token:    tokenService,
        client:   client,
    }
}

```

internal/service/user/validate.go

```
package user

import (
    "context"
    "errors"
    "fmt"
    "github.com/pedrecho/vkr-auth/internal/service/token"
)

func (s *Service) ValidateAccessToken(
    ctx context.Context,
    tokenStr string,
) error {
    _, err := s.token.ValidateAccessToken(ctx, tokenStr)
    switch {
    case errors.Is(err, token.ErrInvalidAccessToken):
        return fmt.Errorf("validate access token: %w %w", err,
ErrInvalidAccessToken)
    case errors.Is(err, token.ErrTokenExpired):
        return fmt.Errorf("validate access token: %w %w", err,
ErrTokenExpired)
    case err != nil:
        return fmt.Errorf("validate access token: %w", err)
    }

    return nil
}
```

internal/service/user/verify.go

```
package user

import (
    "context"
    "errors"
    "fmt"
    "github.com/pedrecho/vkr-auth/internal/dto"
    "github.com/pedrecho/vkr-auth/internal/repository/postgres"
    "github.com/pedrecho/vkr-auth/internal/repository/redis"
)

func (s *Service) VerifyUser(ctx context.Context, email, code, fingerprint
string) (*dto.TokenPair, error) {
    if fingerprint == "" {
        return nil, ErrInvalidFingerprint
    }

    user, err := s.storage.GetUserByEmail(ctx, email)
    if err != nil {
        if errors.Is(err, postgres.ErrUserNotFound) {
            return nil, ErrUserNotFound
        }
        return nil, fmt.Errorf("get user: %w", err)
    }
```

```

}

if user.Status != dto.UserStatusPending {
    return nil, ErrUserNotPending
}

storedCode, err := s.GetVerificationCode(ctx, email)
if err != nil {
    if errors.Is(err, redis.ErrKeyNotFound) {
        return nil, ErrVerificationCodeNotFound
    }
    return nil, fmt.Errorf("get code: %w", err)
}

if storedCode != code {
    return nil, ErrInvalidVerificationCode
}

//if err := s.cache.Delete(ctx, fmt.Sprintf("%s:%s", VerifyPrefix,
email)); err != nil {
//    return nil, fmt.Errorf("delete code: %w", err)
//}

if err := s.storage.UpdateUserStatus(ctx, user.ID,
dto.UserStatusConfirmed); err != nil {
    return nil, fmt.Errorf("update user status: %w", err)
}

if err := s.client.CreateUser(ctx, user.ID.String()); err != nil {
    return nil, fmt.Errorf("create user in other services: %w", err)
}

tokens, err := s.token.GenerateTokens(ctx, user.ID.String(), email,
fingerprint, RefreshPrefix, AccessTokenTTL, RefreshTokenTTL)
if err != nil {
    return nil, fmt.Errorf("generate tokens: %w", err)
}

return tokens, nil
}

```

migrations/20250322181700_users.down.sql

```

DROP TABLE IF EXISTS users;

DROP TYPE IF EXISTS user_status;

```

migrations/20250322181700_users.up.sql

```

DROP TABLE IF EXISTS users;

DROP TYPE IF EXISTS user_status;

```

Репозиторий github.com/pedrecho/vkr-broker

.github/workflows/publish-broker-app.yaml

```
name: Build & Push broker-app to GHCR

on:
  push:
    branches: [master]
  workflow_dispatch:

jobs:
  docker:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ secrets.GHCR_USERNAME }}
          password: ${{ secrets.GHCR_TOKEN }}

      - name: Build and Push Docker Image
        uses: docker/build-push-action@v5
        with:
          context: .
          file: docker/broker-app/Dockerfile
          push: true
          tags: ghcr.io/${{ secrets.GHCR_USERNAME }}/broker-app:latest
          build-args:
            GITHUB_TOKEN=${{ secrets.GOPRIVATE_PAT }}
```

.github/workflows/publish-broker-chart.yaml

```
name: Publish broker Helm Chart
```

```
on:
  push:
    tags:
      - 'chart-*'

jobs:
  helm:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout code
```

```

uses: actions/checkout@v3

- name: Set up Helm
  uses: azure/setup-helm@v3

- name: Log in to GHCR
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${{ secrets.GHCR_USERNAME }}
    password: ${{ secrets.GHCR_TOKEN }}

- name: Extract Chart Version
  id: chart
  run: echo "VERSION=$(grep '^version:' broker-chart/Chart.yaml | awk '{print $2}')" >> $GITHUB_OUTPUT

- name: Package Helm Chart
  run: helm package broker-chart

- name: Push Helm Chart to GHCR
  run: helm push broker-chart-${{ steps.chart.outputs.VERSION }}.tgz
oci://ghcr.io/${{ secrets.GHCR_USERNAME }}

```

Makefile

```

# ===== Docker image =====
IMAGE_NAME := ghcr.io/pedrecho/broker-app
DOCKER_TAG := latest

docker-build:
    docker build -t $(IMAGE_NAME):$(DOCKER_TAG) .

docker-push:
    docker push $(IMAGE_NAME):$(DOCKER_TAG)

# ===== Helm chart =====
CHART_DIR := broker-chart
CHART_NAME := $(CHART_DIR)
CHART_VERSION := $(shell grep "^version:" $(CHART_DIR)/Chart.yaml | awk
"{print $$2}")
RELEASE_NAME := broker

helm-tag:
    git tag chart-$(CHART_VERSION)
    @echo "Created local tag: chart-$(CHART_VERSION)"
    @echo "To push it: git push origin chart-$(CHART_VERSION)"

helm-install:
    helm upgrade --install $(RELEASE_NAME) $(CHART_NAME) \
    -f $(CHART_DIR)/values.yaml
    # -f $(CHART_DIR)/values.secret.yaml # ← если появится secrets,
    # раскомментируй

helm-uninstall:

```

```
helm uninstall $(RELEASE_NAME) || true

helm-clean:
  kubectl delete all -l app=$RELEASE_NAME || true
```

broker-chart/Chart.yaml

```
apiVersion: v2
name: broker-chart
description: A Helm chart for deploying broker-app with NATS
version: 0.1.14
```

broker-chart/files/config.yaml

```
logger:
  level: debug

nats:
  connection:
    host: {{ .Values.nats.host }}
    port: {{ .Values.nats.port }}
    ssl: {{ .Values.nats.ssl }}

  streams:
    {{ toYaml .Values.nats.streams | nindent 4 }}
```

broker-chart/templates/broker-init-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: broker-init
  annotations:
    "helm.sh/hook": post-install,post-upgrade
    "helm.sh/hook-delete-policy": before-hook-creation,hook-succeeded
spec:
  template:
    metadata:
      name: broker-init
    spec:
      serviceAccountName: {{ .Values.serviceAccount.name }}
      restartPolicy: Never
      initContainers:
        - name: wait-for-nats
          image: busybox:1.36
          command:
            - sh
            - -c
            - >
              until nc -z {{ .Values.nats.host }} {{ .Values.nats.port }};
              do echo waiting for nats...
              sleep 1;
              done
      containers:
```

```

- name: broker-app
  image: {{ .Values.brokerApp.image }}
  imagePullPolicy: {{ .Values.brokerApp.imagePullPolicy }}
  command: ["/broker-app"]
  args: ["--config", "{{ .Values.brokerApp.configPath }}", "--up"]
  volumeMounts:
    - name: config-volume
      mountPath: {{ .Values.brokerApp.configPath }}
      subPath: config.yaml
  volumes:
    - name: config-volume
      configMap:
        name: broker-app-config

```

broker-chart/templates/configmap.yaml

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: broker-app-config
  labels:
    app: broker-app

data:
  config.yaml: |-
{{ tpl (.Files.Get "files/config.yaml") . | indent 4 }}

```

broker-chart/templates/nats-deployment.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nats
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nats
  template:
    metadata:
      labels:
        app: nats
  spec:
    containers:
      - name: nats
        image: {{ .Values.nats.image }}
        args:
          - "-js"
          - "--store_dir=/data/jetstream"
        ports:
          - containerPort: {{ .Values.nats.port }}
        volumeMounts:
          - name: js-store
            mountPath: /data/jetstream
    volumes:

```

```
- name: js-store
  emptyDir: {}
```

broker-chart/templates/nats-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nats
spec:
  selector:
    app: nats
  ports:
    - port: {{ .Values.nats.port }}
      targetPort: {{ .Values.nats.port }}
  type: ClusterIP
```

broker-chart/values.yaml

```
brokerApp:
  image: ghcr.io/pedrecho/broker-app:latest
  imagePullPolicy: Always
  replicaCount: 1
  containerPort: 8080
  configPath: /app/config.yaml

nats:
  image: nats:2.10.7
  host: "nats"
  port: 4222
  ssl: false

streams:
  - name: "NOTIFICATIONS"
    subjects:
      - "notifications.emailverification"

serviceAccount:
  name: vkr-ghcr-access
```

cmd/main.go

```
package main

import (
  "flag"
  "fmt"
  "github.com/pedrecho/vkr-broker/internal/app"
)

var (
  configPath = flag.String("config", "", "config path")
  upFlag     = flag.Bool("up", false, "run broker in up mode")
  downFlag   = flag.Bool("down", false, "run broker in down mode")
```

```

)
func main() {
    flag.Parse()

    app, err := app.New(*configPath)
    if err != nil {
        panic(fmt.Errorf("init app: %w", err))
    }

    if err = app.Run(true, false); err != nil {
        panic(err)
    }
}

```

docker/broker-app/Dockerfile

```

# Стадия сборки
FROM golang:1.24 AS builder

ARG GITHUB_TOKEN
ENV GOPRIVATE=github.com/your-user
ENV CGO_ENABLED=0

RUN git config --global url."https://$GITHUB_TOKEN:x-oauth-basic@github.com/".insteadOf "https://github.com/"

WORKDIR /app
COPY ../../go.mod go.sum .
RUN go mod download

COPY ../../.
RUN go build -o broker-app ./cmd/main.go

# Финальный образ
FROM alpine:3.19
WORKDIR /app
COPY --from=builder /app/broker-app .
RUN chmod +x ./broker-app
CMD ["./broker-app"]

```

go.mod

```

module github.com/pedrecho/vkr-broker

go 1.24

require (
    github.com/nats-io/nats.go v1.39.1
    github.com/pedrecho/vkr-pkg v0.0.0-20250225111939-85ee6427a470
    go.uber.org/zap v1.27.0
    gopkg.in/yaml.v3 v3.0.1
)

require (

```

```
github.com/klauspost/compress v1.17.9 // indirect
github.com/nats-io/nkeys v0.4.9 // indirect
github.com/nats-io/nuid v1.0.1 // indirect
go.uber.org/multierr v1.10.0 // indirect
golang.org/x/crypto v0.31.0 // indirect
golang.org/x/sys v0.28.0 // indirect
gopkg.in/natefinch/lumberjack.v2 v2.2.1 // indirect
)
```

internal/app/app.go

```
package app

import (
    "fmt"
    "github.com/pedrecho/vkr-broker/internal/config"
    "github.com/pedrecho/vkr-broker/internal/infrastructure/nats"
    "github.com/pedrecho/vkr-broker/internal/service/broker"
    "github.com/pedrecho/vkr-pkg/zaplogger"
)

type App struct {
    cfg *config.Config
}

func New(configPath string) (*App, error) {
    cfg, err := config.Load(configPath)
    if err != nil {
        return nil, fmt.Errorf("config init: %w", err)
    }

    return &App{
        cfg: cfg,
    }, nil
}

func (a *App) Run(init bool, cleanup bool) error {
    zapsync, err := zaplogger.ReplaceZap(a.cfg.Logger)
    if err != nil {
        return fmt.Errorf("zaplogger init: %w", err)
    }
    defer zapsync()

    natsService, err := nats.New(a.cfg.Nats.Connection)
    if err != nil {
        return fmt.Errorf("init nats: %w", err)
    }

    brokerService := broker.New(a.cfg.Nats, natsService)

    if cleanup {
        if err = brokerService.CleanupStreams(); err != nil {
            return fmt.Errorf("cleanup streams: %w", err)
        }
    }
}
```

```

    if init {
        if err = brokerService.InitStreams(); err != nil {
            return fmt.Errorf("init streams: %w", err)
        }
    }

    return nil
}

```

internal/config/config.go

```

package config

import (
    "fmt"
    "github.com/pedrecho/vkr-pkg/messaging"
    "github.com/pedrecho/vkr-pkg/zaplogger"
    "gopkg.in/yaml.v3"
    "os"
)

type Config struct {
    Logger zaplogger.Config `yaml:"logger"`
    Nats   NatsConfig       `yaml:"nats"`
}

func Load(filename string) (*Config, error) {
    //TODO remove
    data, err := os.ReadFile(filename)
    if err != nil {
        fmt.Println("Ошибка чтения файла:", err)
        os.Exit(1)
    }
    fmt.Println(string(data))

    file, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("read config file: %w", err)
    }
    cfg := Config{}
    err = yaml.Unmarshal(file, &cfg)
    if err != nil {
        return nil, fmt.Errorf("unmarshal config file: %w", err)
    }

    return &cfg, err
}

type NatsConfig struct {
    Connection messaging.NatsConfig `yaml:"connection"`
    Streams    []StreamConfig      `yaml:"streams"`
}

type StreamConfig struct {
    Name     string   `yaml:"name"`
}

```

```
    Subjects []string `yaml:"subjects"`
}
```

internal/infrastructure/nats/nats.go

```
package nats

import (
    "fmt"
    "github.com/nats-io/nats.go"
    "github.com/pedrecho/vkr-pkg/messaging"
)

type Service struct {
    cfg  messaging.NatsConfig
    conn *nats.Conn
    js   nats.JetStreamContext
}

func New(cfg messaging.NatsConfig) (*Service, error) {
    conn, js, err := messaging.NatsJetStreamConnect(cfg)
    if err != nil {
        return nil, fmt.Errorf("nats jetstream connection: %w", err)
    }

    return &Service{
        cfg:  cfg,
        conn: conn,
        js:   js,
    }, nil
}
```

internal/infrastructure/natsstreams.go

```
package nats

import (
    "errors"
    "fmt"
    "github.com/nats-io/nats.go"
    "go.uber.org/zap"
    "log"
)

func (s *Service) AddOrUpdateStream(streamName string, subjects []string) error {
    _, err := s.js.StreamInfo(streamName)
    if err != nil {
        if errors.Is(err, nats.ErrStreamNotFound) {
            _, err = s.js.AddStream(&nats.StreamConfig{
                Name:      streamName,
                Subjects: subjects,
            })
            if err != nil {
                return fmt.Errorf("add stream: %w", err)
            }
        }
    }
}
```

```

        }

        zap.S().Infof("New stream created: %s", streamName)
        return nil
    }

    return fmt.Errorf("get stream info: %w", err)
}

_, err = s.js.UpdateStream(&nats.StreamConfig{
    Name:      streamName,
    Subjects: subjects,
})
if err != nil {
    return fmt.Errorf("update stream: %w", err)
}

log.Printf("Stream updated: %s", streamName)
return nil
}

func (s *Service) DeleteStream(streamName string) error {
err := s.js.DeleteStream(streamName)
if err != nil {
    if errors.Is(err, nats.ErrStreamNotFound) {
        zap.S().Infof("Stream not found: %s", streamName)
        return nil
    }

    return fmt.Errorf("delete stream: %w", err)
}

zap.S().Infof("Stream deleted: %s", streamName)
return nil
}

```

internal/service/broker/broker.go

```

package broker

import "github.com/pedrecho/vkr-broker/internal/config"

type StreamManager interface {
    AddOrUpdateStream(streamName string, subjects []string) error
    DeleteStream(streamName string) error
}

type Service struct {
    streamManager StreamManager
    cfg          config.NatsConfig
}

func New(cfg config.NatsConfig, streamManager StreamManager) *Service {
    return &Service{
        streamManager: streamManager,
        cfg:           cfg,
    }
}

```

```
    }
```

internal/service/broker/streams.go

```
package broker

import (
    "fmt"
)

func (s *Service) InitStreams() error {
    for _, stream := range s.cfg.Streams {
        err := s.streamManager.AddOrUpdateStream(stream.Name, stream.Subjects)
        if err != nil {
            return fmt.Errorf("init stream %s: %w", stream.Name, err)
        }
    }
    return nil
}

func (s *Service) CleanupStreams() error {
    for _, stream := range s.cfg.Streams {
        err := s.streamManager.DeleteStream(stream.Name)
        if err != nil {
            return fmt.Errorf("cleanup stream %s: %w", stream.Name, err)
        }
    }
    return nil
}
```