

МАГИСТЕРСКАЯ ПРОГРАММА 09.04.01/05 Современные интеллектуальные  
программно-аппаратные комплексы

# Система поиска курьеров

2024 з.

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

---

УТВЕРЖДАЮ

Заведующий кафедрой ИУ6  
(Индекс)

А.В. Пролетарский  
(И.О.Фамилия)

«     »                      2024 г.

**ЗАДАНИЕ  
на выполнение курсового проекта**

по дисциплине **Современные технологии разработки программного обеспечения**

Студенты группы ИУ6-31М

Марчук Иван Сергеевич, Голуябтников Дмитрий Сергеевич, Джабри Абделькадер  
Шакер  
(Фамилия, имя, отчество)

Тема курсового проекта, реферата (нужное подчеркнуть):

Система поиска курьеров

Магистерская программа Современные интеллектуальные программно-аппаратные комплексы

Направленность КП (учебная, исследовательская, практическая, производственная, др.)  
учебная

Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения КП: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

**Основные требования к курсовому проекту:**

1. Система должна предусматривать аутентификацию пользователя для разграничения доступа к данным;
2. Система должна обеспечить для пользователя возможность создавать учетную запись и редактировать её;
3. Система должна обеспечить для пользователя возможность зарегистрировать посылку;
4. Система должна обеспечить для пользователя возможность посмотреть статус отправленной посылки, а также отредактировать её данные;

5. Система должна обеспечить для пользователя возможность поиска маршрутов других пользователей;
6. Система должна обеспечить для пользователя возможность сделать заявку на доставку посылки курьером;
7. Система должна для библиотекаря обеспечить возможность получить рейтинг пользователя.
8. Система должна состоять из сервиса бронирования, сервиса библиотек, сервиса рейтинга пользователя и gateway-сервиса.
9. Обращение к системе должно происходить только через gateway-сервис.

**Оформление курсового проекта:**

1. РПЗ, реферат (нужное подчеркнуть) на 40-60 листах формата А4.
2. Все материалы и исходный текст программы записать в репозиторий исходного кода, документы – на страницу дисциплины на сайте кафедры.

Дата выдачи задания « 2 » сентября 2024 г.

**Руководитель курсового проекта**

\_\_\_\_\_  
(Подпись, дата) М.В.Фетисов  
(И.О.Фамилия)

**Студент**

\_\_\_\_\_  
(Подпись, дата) И.С. Марчук  
(И.О.Фамилия)

**Студент**

\_\_\_\_\_  
(Подпись, дата) Д.С. Голубятников  
(И.О.Фамилия)

**Студент**

\_\_\_\_\_  
(Подпись, дата) А.Ш.Джабри  
(И.О.Фамилия)

## РЕФЕРАТ

Расчетно-пояснительная записка 52 страниц, 28 рисунков, 4 таблица, 12 источников.

ДОСТАВКА, ПОСЫЛКИ КУРЬЕР, ПОЧТА,  
МОБИЛЬНОЕ\_ПРИЛОЖЕНИЕ.

Целью данного курсового проекта является разработка программной системы поиска курьеров с серверной частью, реализованной на технологии микросервисов и клиентской частью в виде Android приложения.

В ходе выполнения курсового проекта были выполнены следующие задачи:

- созданы все необходимые схемы, диаграммы и проектная документация;
- разработана микросервисная архитектура;
- разработано мобильное приложение;
- разработаны тесты.

## ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ .....	9
1 Анализ задания, изучение предметной области и описание модели .....	10
1.1 Выбор технологии, фреймворка, языка и среды разработки .....	11
1.2 Разработка концептуальной модели предметной области.....	12
1.4 Взаимодействие изолированных контекстов предметной области ....	14
2 Проектирование структуры и компонентов программного продукта .....	16
2.1 Межпроцессное взаимодействие (выбор протоколов, форматов и регламентов) .....	16
2.2 Проектирование бизнес-логики.....	17
2.3 Базы данных.....	20
2.4 Описание gateway API .....	25
3 Разработка и развертывание микросервисов с использованием конвейера CI/CD .....	31
3.1 Обзор технологий и инструментов .....	31
3.2 Структура и конфигурация микросервисов .....	32
3.3.1 Структура папок.....	32
3.3.2 Конфигурация окружения .....	34
3.3.3 Dockerfiles .....	34
3.3.4 Docker-Compose Конфигурация.....	35
3.3.5 Особенности каждого сервиса.....	37
4 Валидация процесса CI/CD для микросервисов .....	38
4.1 Успешное выполнение конвейера CI/CD .....	38
4.2 Проверка наличия образов в Docker Hub .....	39
4.3 Проверка локального развертывания .....	39
5 Разработка Android приложения.....	41
5.1 Обзор технологий и инструментов .....	41
5.1.1 Android SDK .....	41
5.1.2 Язык программирования .....	41
5.1.3 Связь с BackEnd приложением.....	41

5.1.4 Material Design 3.....	42
5.2 Проектирование и разработка приложения.....	42
5.2.1 Архитектура приложения.....	42
5.2.2 Компоненты системы .....	43
5.2.4 Создание пользовательского интерфейса .....	43
5.2.5 Навигация .....	46
5.2.6 Работа с REST API .....	48
СПИСОК ЛИТЕРАТУРЫ.....	51

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

**Activity** — компонент приложения, который является одним из его фундаментальных строительных блоков.

**Alpine Linux** — минималистичная версия Linux, часто используемая в Docker для создания компактных образов.

**Android** — Операционная система для смартфонов, планшетов, электронных книг, и т.д.

**API (Application Programming Interface)** — интерфейс программирования приложений, обеспечивающий взаимодействие между микросервисами.

**Buildx** — расширение Docker для создания мультиархитектурных образов.

**CI/CD (Continuous Integration / Continuous Deployment)** — процесс непрерывной интеграции и развертывания, используемый для автоматизации сборки, тестирования и развёртывания приложений.

**Docker** — программное обеспечение для контейнеризации, позволяющее создавать, развёртывать и управлять контейнерами приложений.

**Docker Hub** — облачная платформа для хранения и распространения образов Docker.

**Environment Variables (Переменные окружения)** — параметры, используемые для передачи данных конфигурации в приложения.

**Fragment** — это часть activity, которая обеспечивает более модульный дизайн activity.

**JWT (JSON Web Token)** — стандарт для передачи информации между сторонами в виде JSON-объектов с цифровой подписью.

**Microservices (Микросервисы)** — архитектурный подход к созданию приложений, где каждая функциональная часть является независимым сервисом, взаимодействующим с другими через API.

**MVVM** – шаблон проектирования архитектуры приложения. Отвечает за представление данных и бизнес-логику приложения. Модель может включать в себя операции с данными, хранение информации и управление состоянием приложения.

**NavigationGraph** – ресурс XML, который содержит всю связанную с навигацией информацию в одном централизованном месте. Это включает в себя все отдельные области содержимого в приложении, называемые destinations (пункты назначения), а также возможные пути, которые пользователь может пройти через приложение.

**Pipeline (Конвейер)** — автоматизированный рабочий процесс для сборки, тестирования и развёртывания программного обеспечения.

**REST (Representational State Transfer)** – архитектурный стиль взаимодействия компонентов распределённого приложения в сети.

**View (Представление)** – Отвечает за отображение данных и взаимодействие с пользователем. Он визуализирует данные, предоставляемые ViewModel.

**ViewModel (Модель-представления)** – преобразует данные из Model в формат, который может быть легко отображен в View.

**XML** – расширяемый язык разметки.

**YAML (YAML Ain't Markup Language)** — человеко-читаемый формат для записи конфигурационных файлов, часто используемый в CI/CD и Docker.



## **ВВЕДЕНИЕ**

Студенты иностранцы, осваивающие жизнь в России, часто ощущают тоску по дому, родным, друзьям. Однако расстояние, разделяющее их с родиной, ограничивает возможность общения. Кроме того, часто возникают проблемы в международных службах доставки из-за разных геополитических событий и санкций, задача отправки и получения посылок становится все более трудной и дорогой.

Решением этой проблемы может быть система, позволяющая людям путешествующим за границу узнать друг о друге. В таком случае можно было бы попросить путешественника взять с собой на родину открытку или сувенир для того, чтобы передать их семье студента иностранца. Это приложение призвано стать мостом между студентами и их семьями, облегчая обмен вещами и поддерживая культурные связи.

Основная идея нашего проекта заключается в создании удобной платформы, на которой пользователи могут без проблем находить путешественников с пустым местом в багаже. Это не только решает практическую проблему доставки, но и способствует налаживанию новых знакомств и культурному обмену.

В этой работе мы рассмотрим каждый аспект, касающийся концепции этого приложения, стремясь обеспечить всестороннее понимание его разработки и назначения.

## **1 Анализ задания, изучение предметной области и описание модели**

Разрабатываемая система должна предоставлять пользователям возможность найти путешественника, который собирается пройти по маршруту, и который может взять дополнительный вес с собой. Интерфейс должен позволить пользователям добавить описание своей посылки и сделать заявку на отправку посылки по маршруту. А также возможность отслеживания статуса почтового отправления.

Среди аналогов разрабатываемого проекта можно отметить сервисы доставки почтовых отправлений, которые, однако, делают это платно и по заказу клиента. Систем, которые бы позволяли сгруппироваться людям для обмена посылками в качестве дополнительного багажа нет. Но похожая система есть в сервисе бронирования такси, у Яндекс, тариф “Вместе” для совместного заказа такси[<https://taxi.yandex.ru/blog/chto-takoe-tarif-vmeste-i-kak-on-rabotaet>].

Приложение должно упрощать два основных типа операций:

*Публикация маршрута:*

Когда пользователь планирует отправиться в определенный пункт назначения, скажем, в город X, он может создать пост с указанием своих планов поездки.

Это пост служит объявлением для других пользователей, которым могут понадобиться товары, доставленные в город X.

Если есть посты от пользователей, которым требуется доставка посылок в город X, путешественник может связаться с ними, чтобы договориться о доставке посылки во время поездки.

*Запрос на доставку посылки:*

Если пользователю необходимо отправить посылку в город X, он может выполнить поиск по существующим постам от путешественников, планирующих отправиться туда.

Путешественники, которые увидят эти посты, могут предложить доставить посылку во время поездки.

По сути, приложение выступает в качестве платформы для связи путешественников с людьми, которым требуется доставка товаров в определенные пункты назначения, что способствует созданию одноранговой сети доставки.

### **1.1 Выбор технологии, фреймворка, языка и среды разработки**

В качестве языка программирования был выбран Go (Golang) версии 1.22.1. Его преимущества представлены ниже:

1) Производительность: Go компилируется в машинный код, что обеспечивает высокую производительность выполнения программ. Это особенно важно для серверных приложений и микросервисов.

2) Простота и лаконичность: Go имеет простую и лаконичную синтаксис, что облегчает чтение и поддержку кода.

3) Параллелизм: Встроенная поддержка конкурентного программирования с помощью горутин и каналов позволяет эффективно использовать многопоточность.

4) Статическая типизация: Статическая типизация помогает избежать множества ошибок на этапе компиляции, повышая надежность кода.

5) Стандартная библиотека: Go имеет мощную стандартную библиотеку, особенно для работы с сетью и HTTP.

Для написания кода на Go была выбрана среда разработки Goland в силу удобства её использования из коробки.

## **1.2 Разработка концептуальной модели предметной области**

Концептуальная модель предметной области — это модель, представленная множеством понятий и связей между ними, определяющих смысловую структуру рассматриваемой предметной области или её конкретного объекта.

Модель предметной области включает в себя различные сущности, их атрибуты и отношения, а также ограничения, определяющие концептуальную целостность элементов структурной модели, составляющих эту область.

Рассматривая выбранную предметную область «Система поиска курьеров» удалось выделить 4 сущности:

- аутентификация;
- пользователь;
- посылка;
- поездка.

В итоге разделения предметной области на части удалось выявить концептуальную схему представленную на рисунке 1.

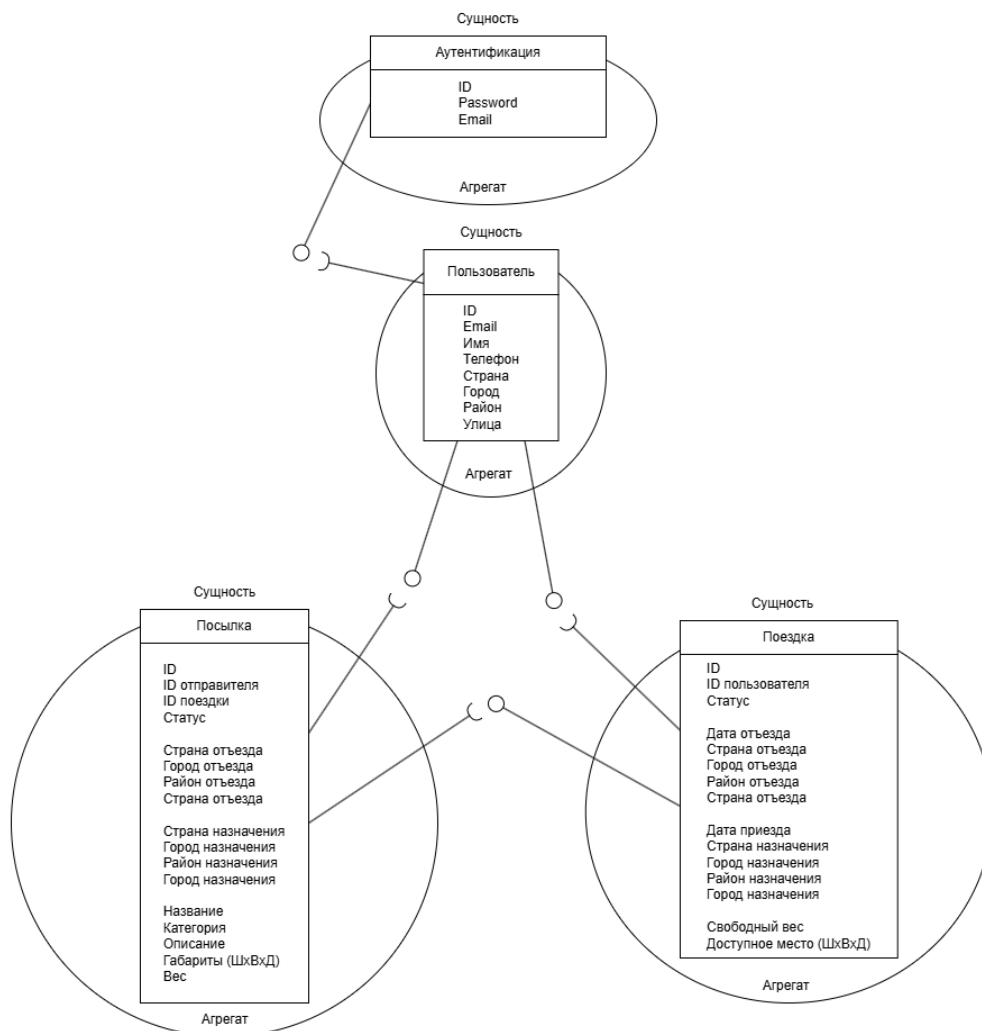


Рисунок 1 – Концептуальная модель предметной области

Аутентификация отвечает за хранение минимальной информации о пользователе, необходимой для регистрации и входа в систему, с дальнейшим предоставлением доступа в систему.

Пользователь отвечает за хранение и предоставление информации о пользователе.

Письмо хранит, предоставляет и обрабатывает информацию о посылках.

Поездка хранит, предоставляет и обрабатывает информацию о поездках.

## 1.4 Взаимодействие изолированных контекстов предметной области

Ограниченные контексты в предметно-ориентированном проектировании (DDD) — это явные границы внутри программной системы, к которой применяется конкретная модель. Они помогают предотвратить двусмысленность в различных частях системы, где могут использоваться схожие термины, но с разным значением.

Карта контекстов — это диаграмма, которая даёт полное представление о разрабатываемой системе. На схеме каждый элемент представляет ограниченный контекст, а связи между элементами отображают отношения, существующие между ограниченными контекстами.

Карта контекстов, представленная на рисунке 2, помогает выявить зависимости, сотрудничество и потенциальные конфликты между контекстами, а также обеспечивает глобальный обзор архитектуры системы с точки зрения предметной области.

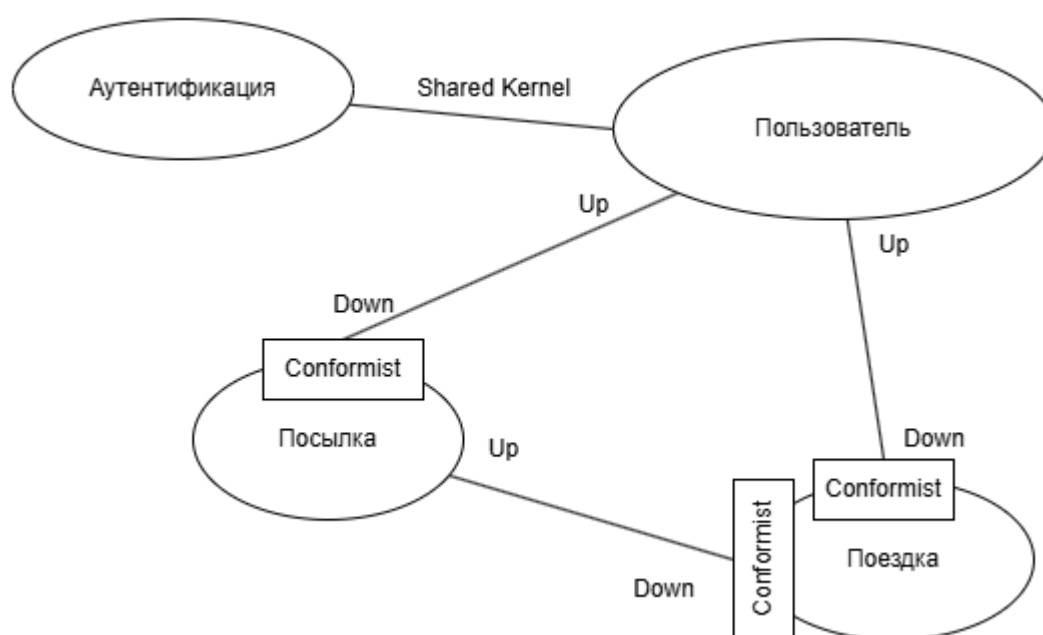


Рисунок 2 – Карта контекстов

Как видно из рисунка аутентификация и пользователь делят общее ядро. Т.к. имеют общие понятия и делят их между собой.

В свою очередь пользователь является источником контекста для посылки и поездки. И для корректного взаимодействия между ними следует использовать «Конформист» (Conformist), т.к. нижестоящие микросервисы адаптируются под правила другого сервиса, не меняя взаимодействие.

Также посылка определяет является источником контекста для поездок. И для корректного взаимодействия так же стоит использовать «Конформист».

## **2 Проектирование структуры и компонентов программного продукта**

### **2.1 Межпроцессное взаимодействие (выбор протоколов, форматов и регламентов)**

Межпроцессное взаимодействие будет осуществляться с помощью предметно-ориентированного проектирования (Domain Driven Design, DDD). DDD помогает сосредоточиться на бизнес-требованиях и логике, что приводит к более качественному и соответствующему решению, помогает разбить сложные системы на более управляемые части (агрегаты, сущности и т.д.), что упрощает разработку и поддержку, а также способствует созданию модульных приложений, что облегчает тестирование, поддержку и масштабирование.

Был выбран протокол HTTP как протокол для взаимодействия с системой и между сервисами. HTTP является стандартом для веб-приложений, и его поддержка в большинстве языков и фреймворков делает его универсальным выбором, а также его легко использовать и отлаживать, что упрощает взаимодействие между сервисами.

В том числе использовался REST (архитектурный стиль взаимодействия компонентов распределённого приложения в сети). Архитектурный стиль – это набор согласованных ограничений и принципов проектирования, позволяющий добиться определённых свойств системы).

Выбор указанных технологий и подходов обеспечивает высокую производительность, простоту разработки и поддержку, а также способствует созданию надежных и масштабируемых приложений. Эти



технологии и методы позволяют эффективно справляться с задачами, связанными с проектированием и реализацией сложных систем.

## 2.2 Проектирование бизнес-логики

В этом разделе мы рассмотрим тонкости наших бизнес-процессов, чтобы охватить поток действий, взаимодействий и данных в нашей системе. Визуализируя эти процессы, мы можем определить области для оптимизации, упростить рабочие процессы и обеспечить эффективность нашего решения.

Благодаря тщательному анализу и продуманному дизайну мы заложим основу для разработки нашего приложения, гарантируя его полное соответствие требованиям и ожиданиям наших заинтересованных сторон. Теперь давайте перейдем к изучению бизнес-процессов с помощью подробных диаграмм.

Начиная со схемы бизнес-процесса регистрации:

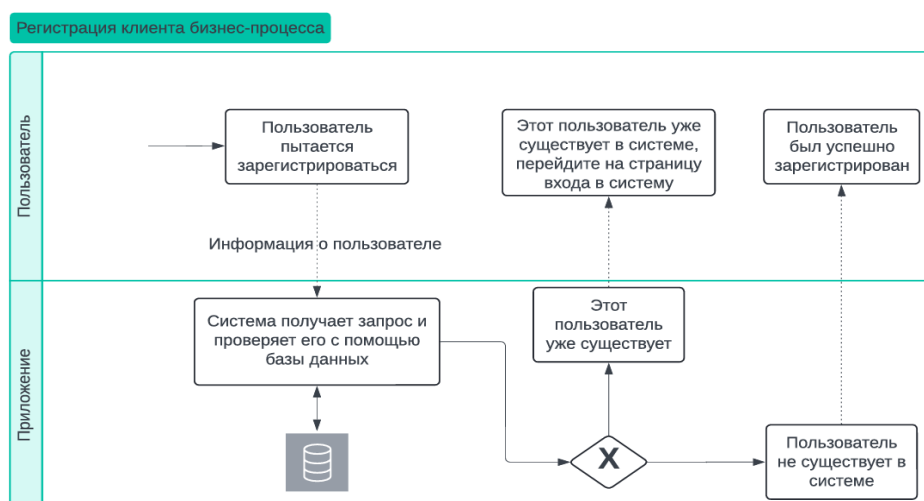


Рисунок 3 - Схема бизнес-процесса «Регистрация клиента».

Вторая схема бизнес-процесса — это схема входа в систему, где каждый пользователь должен иметь учетную запись и входить в нее на случай, если он захочет что-то опубликовать.

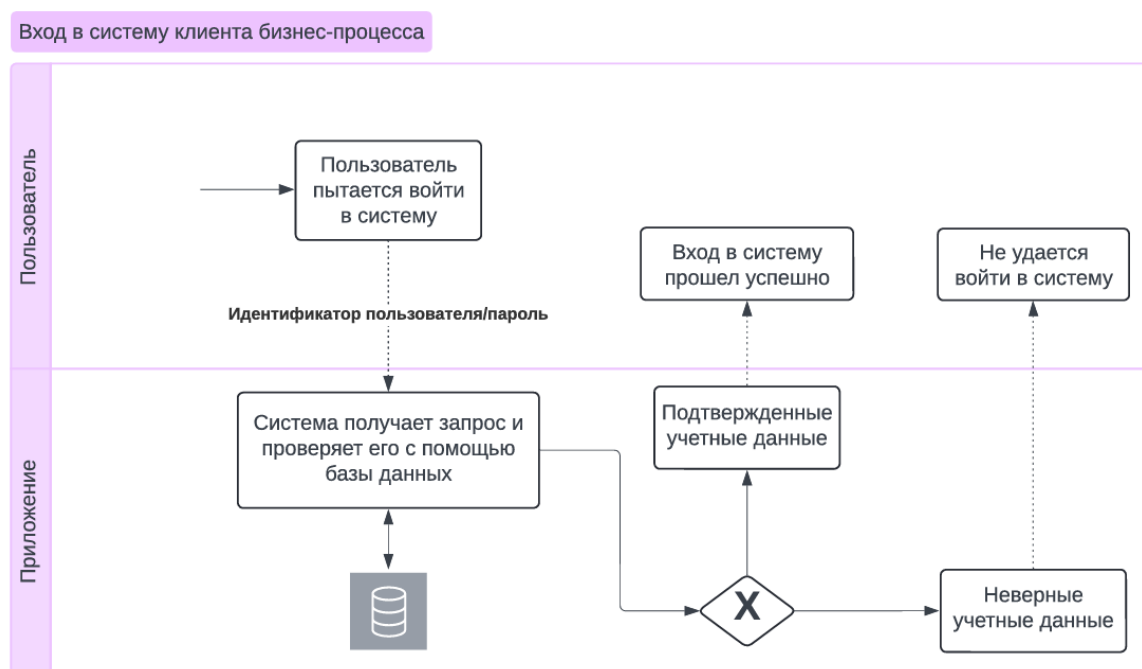


Рисунок 4 - Схема бизнес-процесса «Войдите в систему».

Третья схема бизнес-процесса выглядит следующим образом: когда пользователь (путешественник) собирается куда-то в случае командировочной почты и ему есть куда взять посылку с собой, или когда пользователю нужно что-то куда-то отправить и он ищет кого-то, кто туда направляется.

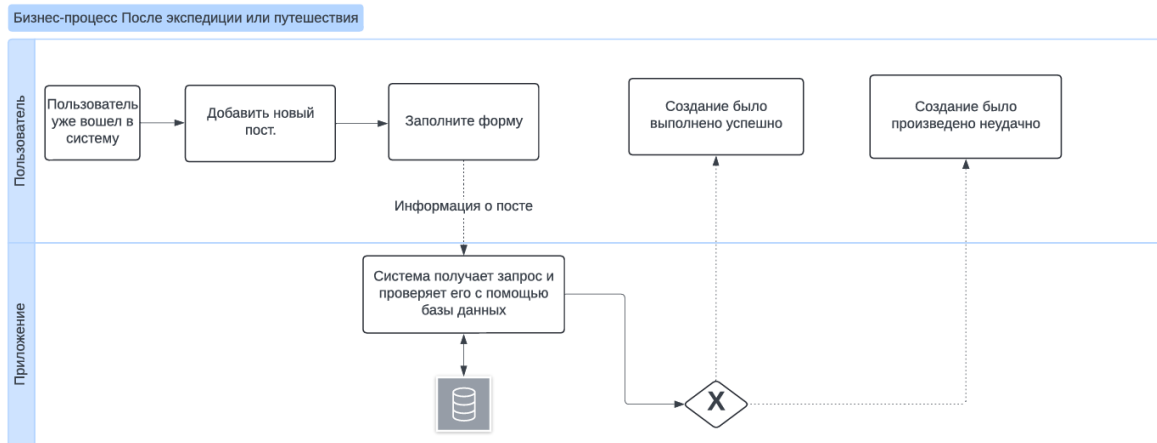


Рисунок 5 - Схема бизнес-процесса «новый пост».

Четвертая схема бизнес-процесса — это когда пользователь ищет конкретную запись, и в основном это происходит путем фильтрации адресов с места на место.

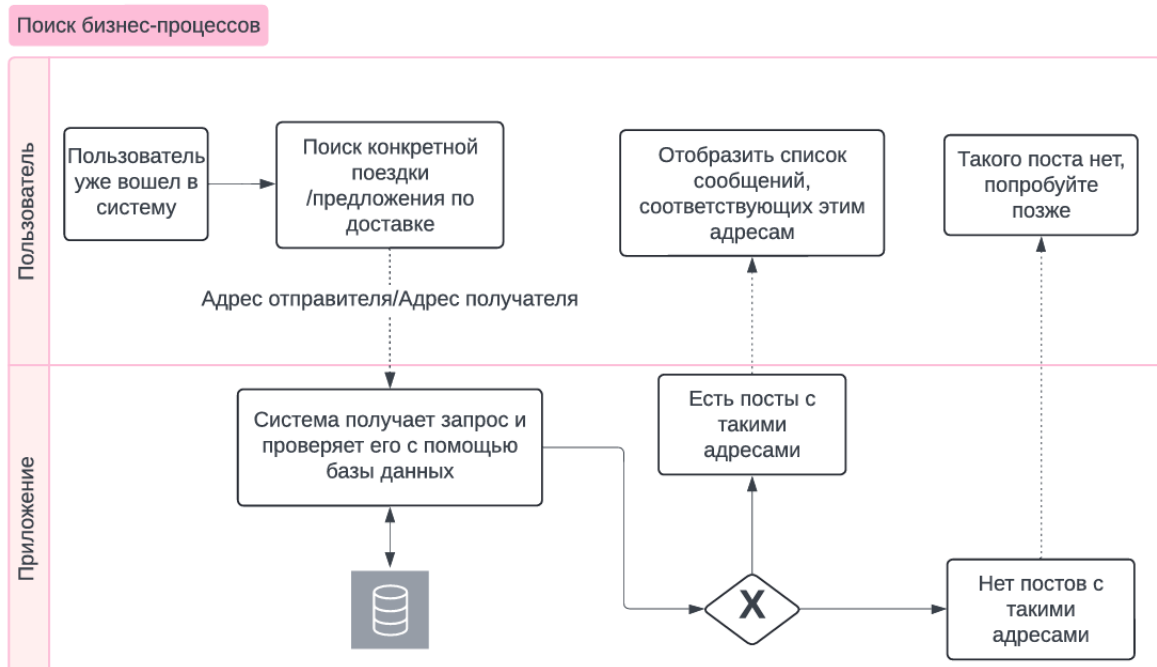


Рисунок 6 - Схема бизнес-процесса «Поиск».

Пятый и последний бизнес-процесс — это редактирование существующей записи.

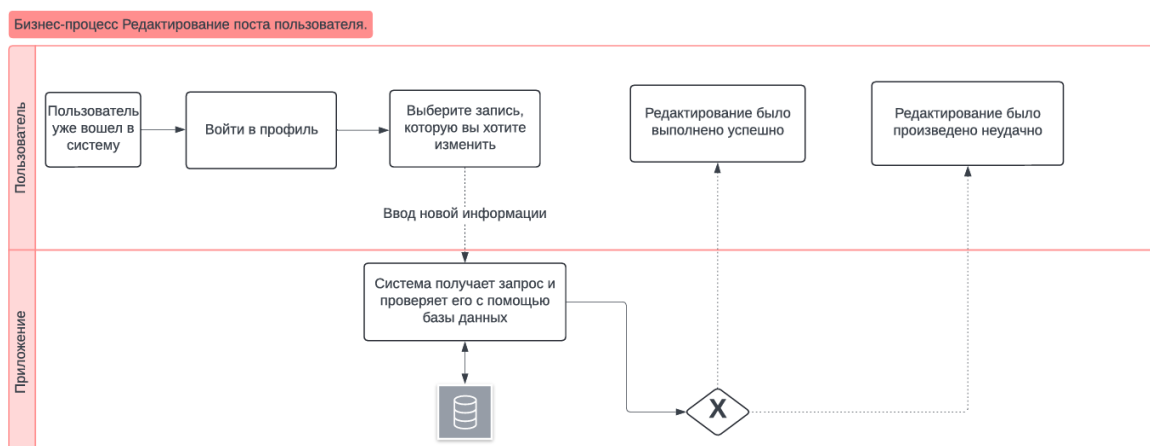


Рисунок 7 - Схема бизнес-процесса «Редактировать пост».

## 2.3 Базы данных

База данных — это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе.

Базы данных необходимы для хранения больших объёмов информации. Базы данных позволяют обрабатывать, хранить и структурировать информацию, предоставляют возможность множественного доступа. Удалённый доступ и система запросов позволяет множеству людей одновременно использовать базы данных. БД облегчает управление данными. Базы данных позволяют легко управлять данными: изменять, обновлять, контролировать и упорядочивать.

В данной системе бронирования книг в библиотеках используется СУБД PostgreSQL. Вот некоторые его преимущества:

1. Гибкость. Пользователи могут создавать функции, операторы, типы данных и индексные методы. Это позволяет расширять функционал базы данных под свои нужды.

2. Соответствие ACID. PostgreSQL выполняет все четыре свойства ACID — набора требований к транзакциям. Это гарантирует надёжность транзакций и целостность данных.

3. Поддержка JSON и XML. В PostgreSQL можно хранить данные и управлять ими в форматах JSON и XML, поэтому её часто используют для работы с неструктурированными данными.

4. Масштабируемость. PostgreSQL поддерживает горизонтальное и вертикальное масштабирование — можно работать с большими объёмами данных и с большим числом пользователей.

5. Надёжность. PostgreSQL быстро восстанавливается после сбоев благодаря журналу изменений транзакций (WAL).

6. Свободное распространение и открытый исходный код. Проект распространяется под лицензией BSD, что позволяет бесплатно его использовать, модифицировать и распространять.

Для работы с базой данных была использован пакет GORM языка GO, являющийся ORM. ORM — это технология, которая позволяет разработчикам взаимодействовать с реляционными базами данных, используя объектно-ориентированный подход. Она предоставляет слой абстракции между программным кодом и базой данных, что упрощает работу с данными.

Основными функциями ORM является:

1. Автоматизация преобразований.

ORM автоматически преобразует данные из реляционных таблиц в объекты программного кода и наоборот. Пример: Строка из таблицы базы данных становится объектом класса.

2. Упрощённый доступ к данным.

Вместо написания SQL-запросов вы работаете с данными, используя методы и свойства объектов.

### 3. Управление схемой базы данных.

ORM поддерживает миграции, позволяя изменять структуру базы данных через код.

### 4. Управление связями.

ORM упрощает работу с отношениями между таблицами, автоматически связывая объекты.

В таблице 1 представлена таблица `users` базы данных сервиса аутентификации созданная при помощи GORM.

Таблица 1 - база данных сервиса аутентификации.

Название поля	Тип поля
ID	<code>uint `gorm:"primaryKey"</code>
Email	<code>string `gorm:"unique;not null"</code>
Password	<code>string `gorm:"not null"</code>

В таблице 2 представлена таблица `users` базы данных сервиса пользователей созданная при помощи GORM.

Таблица 2 - база данных сервиса пользователей.

Название поля	Тип поля
ID	<code>uint `gorm:"primaryKey"</code>
Email	<code>string `gorm:"unique;not null"</code>
Name	<code>string `gorm:"not null"</code>
Phone	<code>string `gorm:"not null"</code>

Продолжение таблицы 2.

Country	string `gorm:"not null"``
City	string `gorm:"not null"``
District	string `gorm:"not null"``
Street	string `gorm:"not null"``

В таблице 3 представлена таблица packages базы данных сервиса посылок созданная при помощи GORM.

Таблица 3 - база данных сервиса посылок.

Название поля	Тип поля
ID	uint `gorm:"primaryKey"``
SenderID	uint `gorm:"not null"``
TripID	*uint
Status	*string
DepCountry	string `gorm:"not null"``
DepCity	string `gorm:"not null"``
DepDistrict	string `gorm:"not null"``
DepStreet	string `gorm:"not null"``
DestCountry	string `gorm:"not null"``
DestCity	string `gorm:"not null"``
DestDistrict	string `gorm:"not null"``

Продолжение таблицы 3.

DestStreet	string `gorm:"not null"``
Name	string `gorm:"not null"``
Category	string `gorm:"not null"``
Description	string `gorm:"not null"``
Width	float64 `gorm:"not null"``
Height	float64 `gorm:"not null"``
Length	float64 `gorm:"not null"``
Weight	float64 `gorm:"not null"``

В таблице 4 представлена таблица trips базы данных сервиса поездок созданная при помощи GORM.

Таблица 4 - база данных сервиса поездок.

Название поля	Тип поля
ID	uint `gorm:"primarykey"``
UserID	uint `gorm:"not null"``
IsBusy	bool
DepDate	time.Time `gorm:"not null"``
DepCountry	string `gorm:"not null"``
DepCity	string `gorm:"not null"``
DepCity	string `gorm:"not null"``



Продолжение таблицы 4.

DepDistrict	string `gorm:"not null"``
DepStreet	string `gorm:"not null"``
DestDate	time.Time `gorm:"not null"``
DestCountry	string `gorm:"not null"``
DestCity	string `gorm:"not null"``
DestDistrict	string `gorm:"not null"``
DestStreet	string `gorm:"not null"``
FreeWidth	float64 `gorm:"not null"``
FreeHeight	float64 `gorm:"not null"``
FreeLength	float64 `gorm:"not null"``
FreeWeight	float64 `gorm:"not null"``

## 2.4 Описание gateway API

Сервис gateway - сервис, объединяющий остальные сервисы системы, с помощью которого взаимодействует клиент. В данном случае, gateway объединяет сервис библиотек, сервис бронирования и сервис рейтинга.

В gateway реализован middleware для проверки валидности jwt-токена, возвращаемого пользователю при регистрации и входе в систему.

JWT (JSON Web Token) — это компактный, самодостаточный токен, используемый для передачи информации между участниками в безопасном виде. Токен состоит из трех частей (заголовок, полезная нагрузка, подпись),

соединенных точками, и позволяет удостоверять аутентификацию и авторизацию, обеспечивая целостность данных с помощью криптографической подписи.

При проверке jwt-токена из полезной нагрузки извлекается время жизни токена, т.е. время, до которого он считается валидным. При истечении этого времени пользователю отказывается в доступе к сервисам. Чтобы продолжить работу, пользователю необходимо заново войти в систему. Также в полезной нагрузке в нашей системе сохраняется id пользователя. Это облегчает работу, т.к. для каждого из запросов требующего идентификатор пользователя он извлекается из токена и передается в заголовок «X-User-ID», который затем читается в принимающих запросы сервисах.

Сервис gateway реализует следующие api-методы:

1) Создание пользователя

Создание пользователя происходит запросом POST по адресу localhost:8080/create\_user.

В теле запроса передаются данные необходимые для регистрации пользователя такие как: электронная почта, пароль, имя, номер телефона, страна, город, район и улица проживания.

Запрос переадресуется сервисам аутентификации и пользователей и создаёт в них новые записи. При успешном создании пользователя возвращается сообщение о успешной регистрации и jwt-токен для дальнейшей работы пользователя в системе.

2) Вход в систему

Вход в систему осуществляется запросом POST по адресу localhost:8080/login\_user.

В теле запроса передаются данные необходимые для регистрации пользователя такие как: электронная почта и пароль.

Запрос переадресуется в сервис аутентификации, где происходит проверка правильности указанных почты и пароля. При успешном входе в систему возвращается сообщение о успешном входе в систему и jwt-токен для дальнейшей работы пользователя в системе.

### 3) Получение данных о пользователе

Получение данных о пользователе происходит запросом GET по адресу localhost:8080/get\_user.

При этом запросе передается только «Bearer token» в котором содержится jwt-токен.

Запрос переадресуется в сервис пользователей, где считывается id пользователя из «X-User-ID» и возвращаются данные о указанном в данном заголовке пользователе.

### 4) Редактирование профиля пользователя

Редактирование профиля пользователя происходит запросом POST по адресу localhost:8080/update\_user.

В запросе передается «Bearer token» и тело запроса, содержащее поля, которые должны быть обновлены.

Запрос отправляется в сервис пользователей, который в свою очередь при успешном изменении данных пользователя возвращает сообщение об этом.

#### 5) Загрузка посылок пользователя

Загрузка посылок пользователя осуществляется запросом GET по адресу localhost:8080/package\_by\_sender\_id.

При этом запросе передается только «Bearer token» в котором содержится jwt-токен.

Запрос переадресуется в сервис посылок. При успешной работе сервиса пользователю возвращается массив json-объектов, содержащих информацию о посылках, принадлежащих пользователю.

#### 6) Создание новой посылки

Создание новой посылки происходит запросом POST по адресу localhost:8080/create\_package.

В запросе передается «Bearer token» и тело запроса, содержащее все необходимые поля для создания посылки.

Gateway перенаправляет этот запрос в сервис посылок. При успешной обработке запроса пользователю возвращается сообщение о успешном создании посылки.

#### 7) Редактирование посылки

Редактирование посылки осуществляется запросом POST по адресу localhost:8080/update\_package.

В запросе передается «Bearer token» и тело запроса, содержащее все необходимые поля для редактирования посылки.

Gateway перенаправляет этот запрос в сервис посылок. При успешной обработке запроса пользователю возвращается сообщение о успешном изменении посылки.

#### 8) Создание поездки

Создание поездки происходит запросом POST по адресу localhost:8080/create\_trip.

В запросе передается «Bearer token» и тело запроса, содержащее все необходимые поля для создания поездки.

Gateway перенаправляет этот запрос в сервис посылок. При успешной обработке запроса пользователю возвращается сообщение о успешном создании поездки.

#### 9) Редактирование поездки

Редактирование поездки осуществляется запросом POST по адресу localhost:8080/update\_trip.

В запросе передается «Bearer token» и тело запроса, содержащее все необходимые поля для редактирования поездки.

Gateway перенаправляет этот запрос в сервис посылок. При успешной обработке запроса пользователю возвращается сообщение о успешном редактировании поездки.

#### 10) Фильтрация поездок

Фильтрация поездок происходит запросом POST по адресу localhost:8080/filtered\_trips.

В запросе передается «Bearer token» и тело запроса, содержащее такие поля как: страна и город отбытия, страна и город прибытия, дата выезда и вес посылки, которую с собой могут взять.

Запрос перенаправляет в сервис поездок, где выбираются и возвращаются все посылки, удовлетворяющие фильтру. В качестве ответа передается массив json-объектов, удовлетворяющих параметрам запроса.

#### 11) Связать посылку с поездкой

Связь посылки с поездкой осуществляется запросом POST по адресу localhost:8080/link\_with\_trip.

В запросе передается «Bearer token» и тело запроса, содержащее идентификаторы посылки и поездки.

Запрос перенаправляется в сервис посылок, где в свою очередь формируется запрос в сервис поездок на изменение статуса поездки как занятой. При успешном изменении поездки в поле trip\_id посылки заносится id поездки, с которой она будет отправлена. При прохождении всех этапов возвращается сообщение о успешной связи посылки и поездки.

#### 12) Выбор поездки

Выбор поездки осуществляется путём запроса POST по адресу localhost:8080/select\_trip.

В запросе передается «Bearer token» и тело запроса, содержащее идентификатор поездки.

Запрос перенаправляется в сервис посылок. В результате работы при успешной работе возвращается информация о заданной поездке.

### 3 Разработка и развертывание микросервисов с использованием конвейера CI/CD

В современной разработке программного обеспечения архитектура микросервисов стала одной из наиболее популярных благодаря своей гибкости, масштабируемости и удобству поддержки. Эта глава посвящена практической реализации разработки и развертывания микросервисов с использованием конвейера CI/CD (непрерывная интеграция/непрерывное развертывание). Автоматизируя ключевые процессы, конвейер CI/CD обеспечивает более быструю доставку, минимизирует человеческие ошибки и упрощает процесс развертывания сложных систем. В этой главе мы рассмотрим пошаговый путь создания, настройки, тестирования и развертывания нескольких микросервисов, акцентируя внимание на используемых инструментах и методологиях.

#### 3.1 Обзор технологий и инструментов

Для реализации архитектуры микросервисов и конвейера CI/CD была использована комбинация технологий и инструментов. Каждый из них играл важную роль в обеспечении бесперебойного выполнения процесса: от сборки отдельных микросервисов до их развертывания в контейнерной среде. Ниже представлен обзор ключевых технологий и инструментов:

1. **Docker:** использовался для контейнеризации микросервисов, что гарантировало их стабильную работу в различных средах. Docker Compose упрощал оркестрацию сервисов для локального тестирования.
2. **Go (Golang):** основной язык программирования, использовавшийся для разработки микросервисов благодаря своей производительности, простоте и поддержке конкурентности.

3. **PostgreSQL:** надежная и масштабируемая система управления базами данных для хранения и извлечения данных каждого микросервера.
4. **GitHub Actions:** использовался для автоматизации конвейера CI/CD, включая сборку, тестирование и развертывание Docker-образов всех микросервисов.
5. **Docker Hub:** служил в качестве реестра контейнеров для хранения и управления Docker-образами, делая их доступными для развертывания.
6. **Alpine Linux:** легковесное дистрибутивное ядро Linux, использовавшееся как базовый образ для контейнеров Docker, что уменьшало размер образов и повышало их безопасность.
7. **Ubuntu (GitHub Runners):** использовался как виртуальная среда для выполнения конвейеров CI/CD в GitHub Actions.

### 3.2 Структура и конфигурация микросервисов

Этот раздел описывает структуру и конфигурацию микросервисов, разработанных для проекта. Каждый микросервис работает как независимый модуль, взаимодействуя с другими через определенные интерфейсы. Микросервисы включают auth, user, trip, package и gateway, каждый из которых имеет свою функциональность и индивидуальную настройку.

#### 3.3.1 Структура папок

Проект организован в виде четко структурированной иерархии для обеспечения модульности и ясности. Пример структуры:



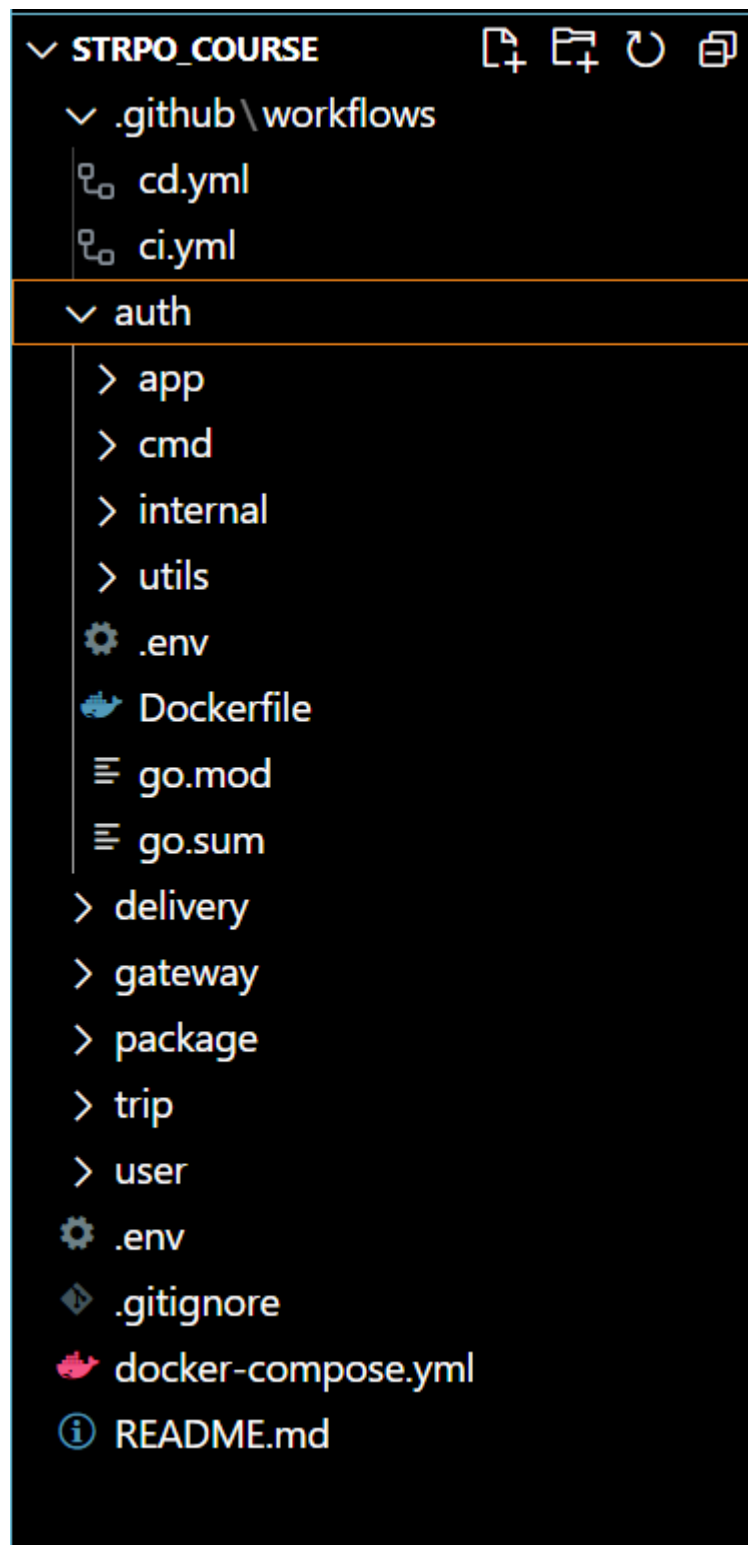


Рисунок 8 - Структура папок.

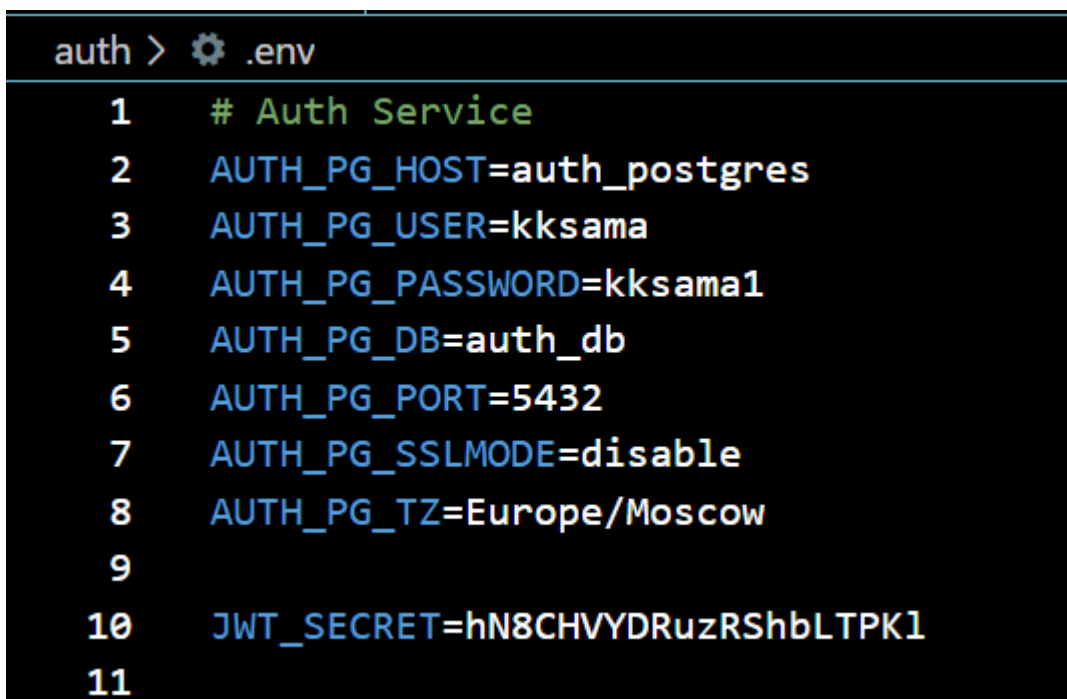
Каждый микросервис находится в собственной папке и включает:

- Dockerfile: определяет контейнерное окружение сервиса.

- go.mod и go.sum: управление зависимостями Go.
- cmd/: точка входа в приложение.
- internal/: включает конфигурации, доступ к базе данных, обработчики и другие внутренние модули.

### 3.3.2 Конфигурация окружения

Каждому сервису требуются переменные окружения для настройки. Файлы .env используются для определения таких переменных, как учетные данные базы данных и настройки хоста. Пример конфигурации .env для сервиса auth:



```
auth > ⚙ .env
1  # Auth Service
2  AUTH_PG_HOST=auth_postgres
3  AUTH_PG_USER=kksama
4  AUTH_PG_PASSWORD=kksama1
5  AUTH_PG_DB=auth_db
6  AUTH_PG_PORT=5432
7  AUTH_PG_SSLMODE=disable
8  AUTH_PG_TZ=Europe/Moscow
9
10 JWT_SECRET=hN8CHVYDRuzRShbLTPK1
11
```

Рисунок 9 - Конфигурация окружения.

### 3.3.3 Dockerfiles

Для каждого микросервиса определен Dockerfile, который описывает его контейнерное окружение. Например, для *auth* используется многослойная сборка:

```

auth > Dockerfile > ...
1  # Use the Go image for building the binary
2  FROM golang:1.22-alpine as builder
3  # Set the working directory inside the container
4  WORKDIR /usr/local/src
5  # Copy Go modules files and download dependencies
6  COPY go.mod go.sum ./
7  RUN go mod download
8  # Copy the application source code
9  COPY . .
10 # Build the Go application
11 RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o ./bin/app ./cmd/main.go
12 # Use a minimal image for the runtime environment
13 FROM alpine:latest
14 # Set the working directory
15 WORKDIR /root
16 # Copy the built application from the builder stage
17 COPY --from=builder /usr/local/src/bin/app .
18 # Copy the .env file into the container
19 COPY .env ./
20 # Expose the necessary port
21 EXPOSE 8080
22 # Command to run the application
23 CMD ["/app"]
24

```

Рисунок 10 - Конфигурация файла Dockerfile для микросервиса *auth*.

### 3.3.4 Docker-Compose Конфигурация

Файл *docker-compose.yml* координирует микросервисы, задавая их сети, зависимости и ресурсы. Пример:

```

version: "3.8"

services:
  # _____ Auth Service _____
  auth_postgres:
    image: postgres:latest
    container_name: auth_postgres
    environment:
      POSTGRES_USER: kksama
      POSTGRES_PASSWORD: kksama1
      POSTGRES_DB: auth_db
    ports:
      - "5432:5432"
    networks:
      - auth_network
    volumes:
      - postgres_data:/var/lib/postgresql/data

  auth:
    image: strpo_course-auth
    container_name: auth
    depends_on:
      - auth_postgres
    ports:
      - "8081:8080"
    environment:
      - AUTH_PG_HOST=auth_postgres
      - AUTH_PG_USER=kksama
      - AUTH_PG_PASSWORD=kksama1
      - AUTH_PG_DB=auth_db
      - AUTH_PG_PORT=5432
      - AUTH_PG_SSLMODE=disable
      - AUTH_PG_TZ=Europe/Moscow
      - JWT_SECRET=hN8CHVYDRuzRShbLTPK1
    networks:
      - auth_network

```

Рисунок 11 - docker-compose Конфигурация.

### 3.3.5 Особенности каждого сервиса

- *auth*: управляет аутентификацией и генерацией JWT токенов.
- *user*: обрабатывает данные пользователей.
- *trip*: сохраняет записи о поездках.
- *package*: управляет данными о посылках.
- *gateway*: API-шлюз, маршрутизирующий запросы.

Эта структура поддерживает масштабируемость, упрощает обслуживание и повышает отказоустойчивость.

## 4 Валидация процесса CI/CD для микросервисов

### 4.1 Успешное выполнение конвейера CI/CD

Этот раздел подчеркивает успешное выполнение конвейера CI/CD. Скриншот из раздела GitHub Actions показывает, что все шаги, включая сборку и публикацию Docker-образов для всех микросервисов, были успешно выполнены без ошибок. Это демонстрирует, что автоматизация конвейера, которую мы внедрили, полностью функционирует.

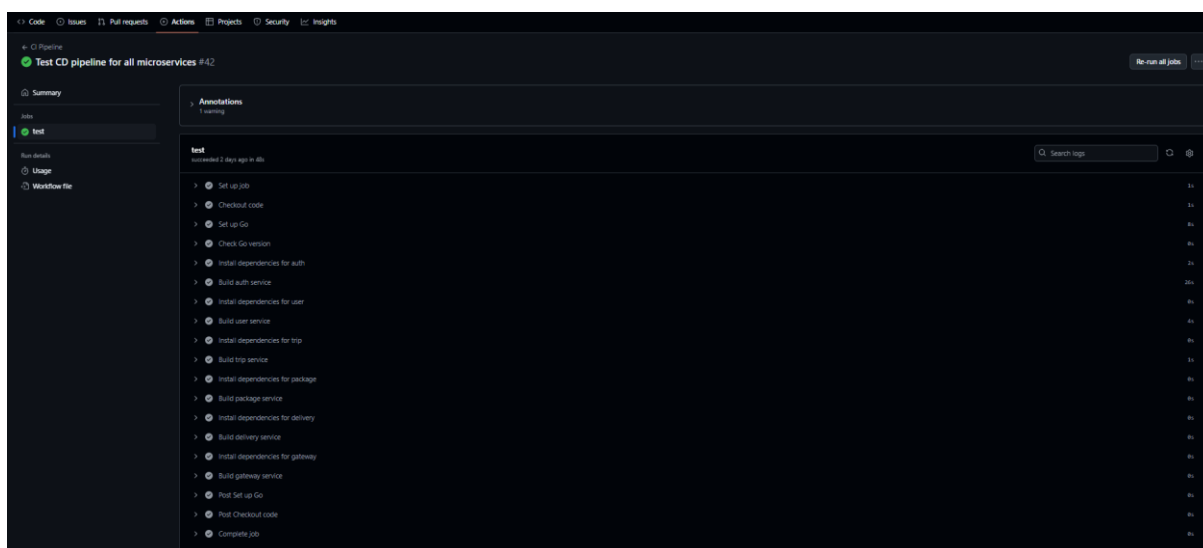


Рисунок 12 - Валидация процесса создания CI для микросервисов

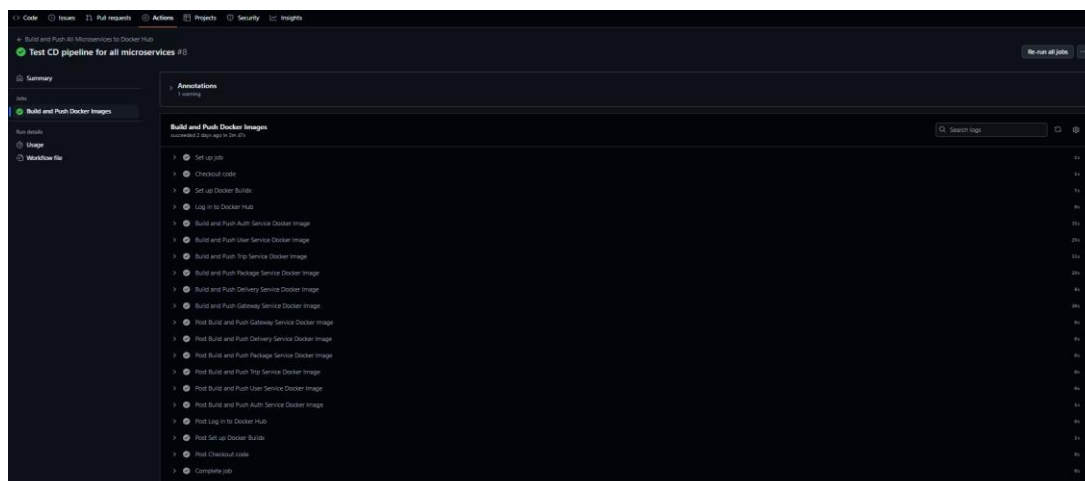


Рисунок 13 - Валидация процесса создания CD для микросервисов

## 4.2 Проверка наличия образов в Docker Hub

Этот раздел демонстрирует, что все микросервисы были успешно собраны и отправлены в Docker Hub. Скриншот из репозитория Docker Hub показывает образы каждого микросервиса, подтверждая, что процессы сборки и отправки в конвейере CI/CD работают как задумано.

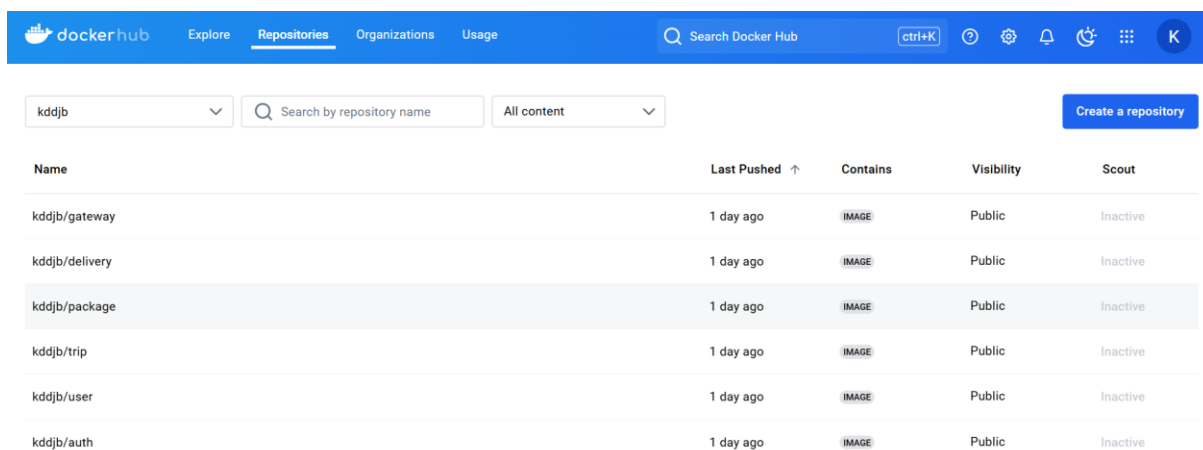


Рисунок 14 - Репозиторий Docker Hub со всеми образами микросервисов.

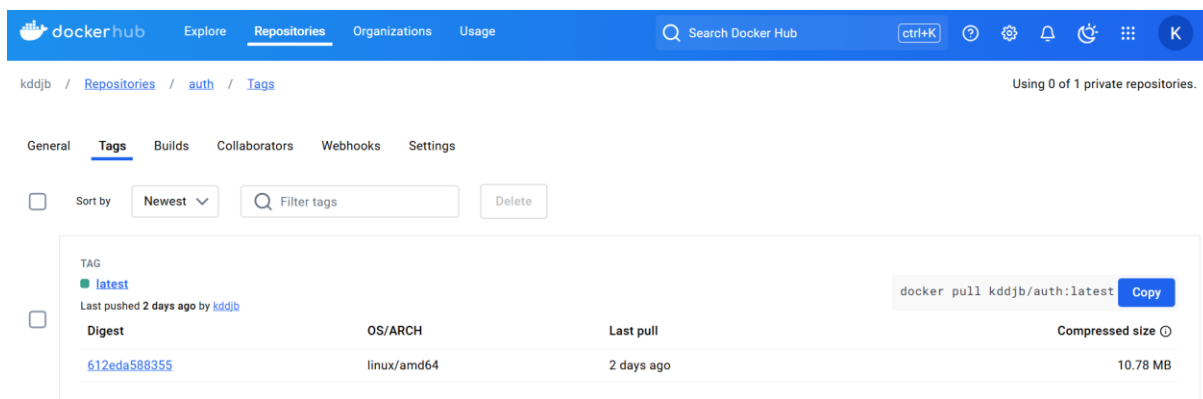


Рисунок 15 - *auth* изображения на Dockerhub.

## 4.3 Проверка локального развертывания

Этот раздел подчеркивает, что все микросервисы успешно запущены на локальном компьютере. Скриншот из приложения Docker Desktop

показывает, что все микросервисы развернуты и работают, подтверждая, что образы, загруженные из Docker Hub, функциональны.

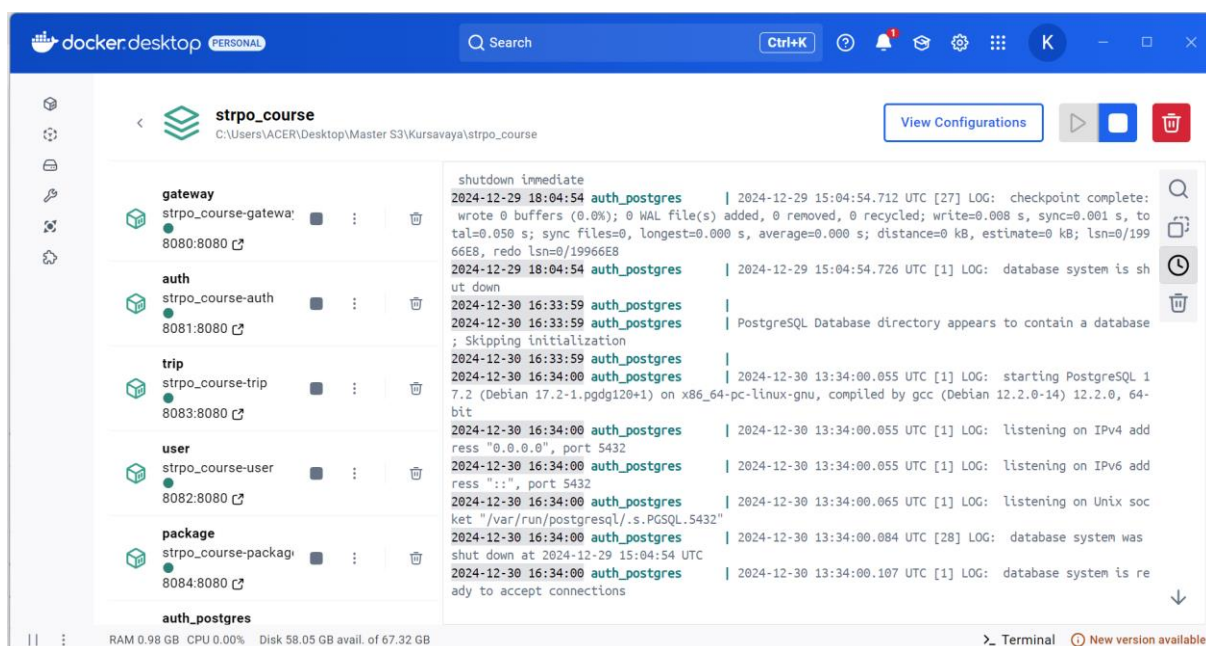


Рисунок 16 - Desktop приложение Docker, отображающее все запущенные контейнеры для микросервисов.

Вместе эти три раздела иллюстрируют весь процесс реализации и проверки конвейера CI/CD для микросервисов. Успешное выполнение конвейера, доступность образов в Docker Hub и правильное функционирование микросервисов локально демонстрируют, что мы успешно достигли нашей цели по автоматизированной и надежной сборке, развертыванию и запуску микросервисов.



## **5 Разработка Android приложения**

### **5.1 Обзор технологий и инструментов**

В этом разделе рассмотрим используемые технологии и инструменты, которые были задействованы при разработке Android приложения в том числе необходимые для взаимодействия с REST API.

#### **5.1.1 Android SDK**

Для разработки приложения была использована платформа Android SDK версии 34. Android SDK предоставляет набор инструментов и библиотек, необходимых для создания приложений под операционную систему Android. В SDK включены компиляторы, инструменты для отладки, эмуляторы и множество библиотек, упрощающих разработку.

#### **5.1.2 Язык программирования**

Приложение было разработано с использованием языка программирования Kotlin. Kotlin является передовым для разработки под Android, предоставляя разработчикам мощные инструменты для создания надежных и масштабируемых приложений. Более того, язык Kotlin является основным развитием языка программирования под платформу Android от компании Google.

#### **5.1.3 Связь с BackEnd приложением**

Диалог с серверной частью проекта был построен на базе REST. Для работы с ним была использована библиотека OkHttp [12], обладающая всем необходимым функционалом для работы с POST и GET запросами в асинхронном режиме. Это позволит хранить все данные приложения на сервере, и выполнять все операции с базой данных в асинхронном режиме, что обеспечивает отзывчивость пользовательского интерфейса и предотвращает блокировку главного потока.

### **5.1.4 Material Design 3**

Для создания современного и интуитивно понятного пользовательского интерфейса была использована библиотека Material Design 3. Material Design – это язык дизайна, разработанный Google, который предоставляет набор инструментов и компонентов для создания привлекательных и удобных интерфейсов. Использование Material Design 3 позволяет приложению соответствовать современным стандартам дизайна и ожиданиям пользователей.

## **5.2 Проектирование и разработка приложения**

### **5.2.1 Архитектура приложения**

В качестве архитектурного паттерна была выбрана MVVM + Single activity. Такая архитектура, помимо того, что считается самой актуальной и желательной при разработке новых приложений, позволяет:

- Сократить количество межмодульных вызовов, и использование памяти, так как для работы приложения используется только одна Activity. И при переключении экранов приложения просто подменяются элементы интерфейса, а не перестраиваются большие куски приложения;
- Необходимо отслеживать жизненный цикл только одной Activity, и все данные инициализируются в ней;
- Создается класс ViewModel, который хранит все данные приложения и работает с моделью (сохраненными данными и REST API). Более того, обычно такие события как переворот экрана или смена цветовой темы вызывает перестройку всей Activity, однако ViewModel лишена этих проблем так как существует отдельно от жизненного цикла Activity, и очищается только после закрытия приложения.

### **5.2.2 Компоненты системы**

Приложение было разделено на следующие компоненты:

- MainActivity, главная страница интерфейса, служит контейнером для фрагментов;
- MainViewModel, находится в памяти пока приложение работает, хранит все данные взаимодействует с API.
- RegisterFragment, экран авторизации пользователя
- LoginFragment, экран регистрации пользователя
- UserFragment, экран профиля пользователя, с возможностью просмотра информации о пользователе и просмотра входящих уведомлений
- ExpeditionsFragment, экран отображения отправленных пользователем посылок;
- TrajectoriesFragment, экран на котором можно посмотреть путешествия других пользователей, сделать заявку на отправку своей посылки и написать о своем планирующемся путешествии.

### **5.2.4 Создание пользовательского интерфейса**

Прежде чем разрабатывать само приложение я сделал макет пользовательского интерфейса в редакторе Figma.

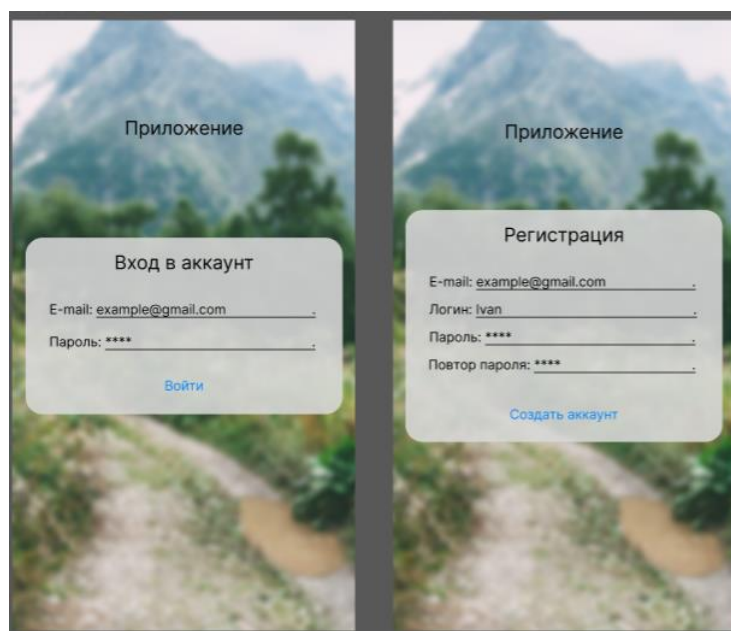


Рисунок 17 - Экраны входа пользователя в аккаунт и его регистрации, созданные в программе figma

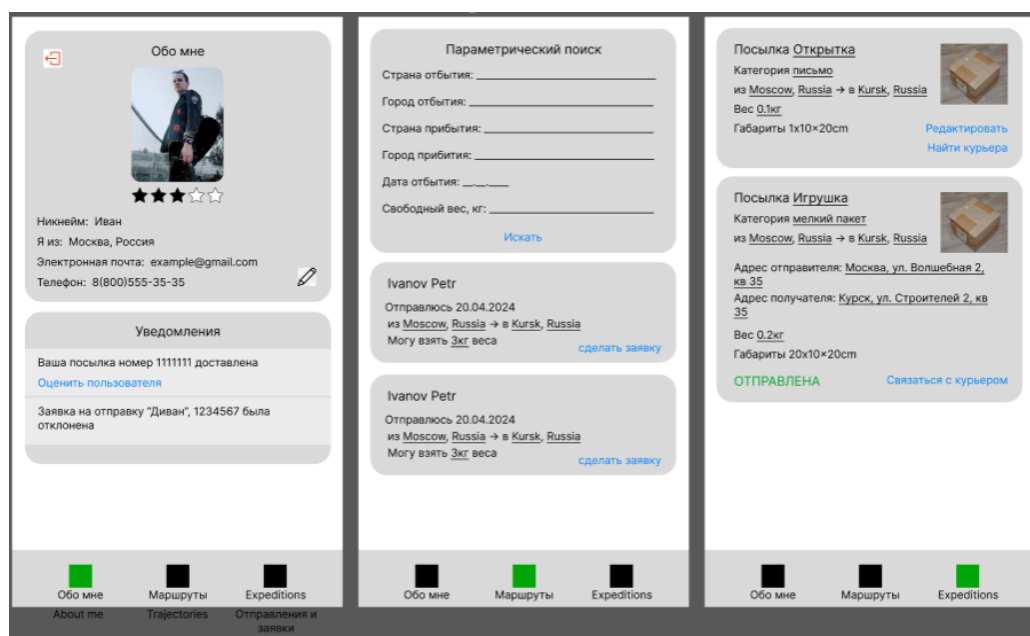


Рисунок 18 - Экран просмотра данных пользователя, экран просмотра маршрутов других пользователей и экран просмотра посылок пользователя созданные в программе figma

Подобное моделирование будущего интерфейса помогает не только правильно распределить элементы управления, но и продумать и утвердить варианты использования пользователем приложения.

После проработки всех аспектов интерфейса я приступил к созданию разметки в Android Studio.

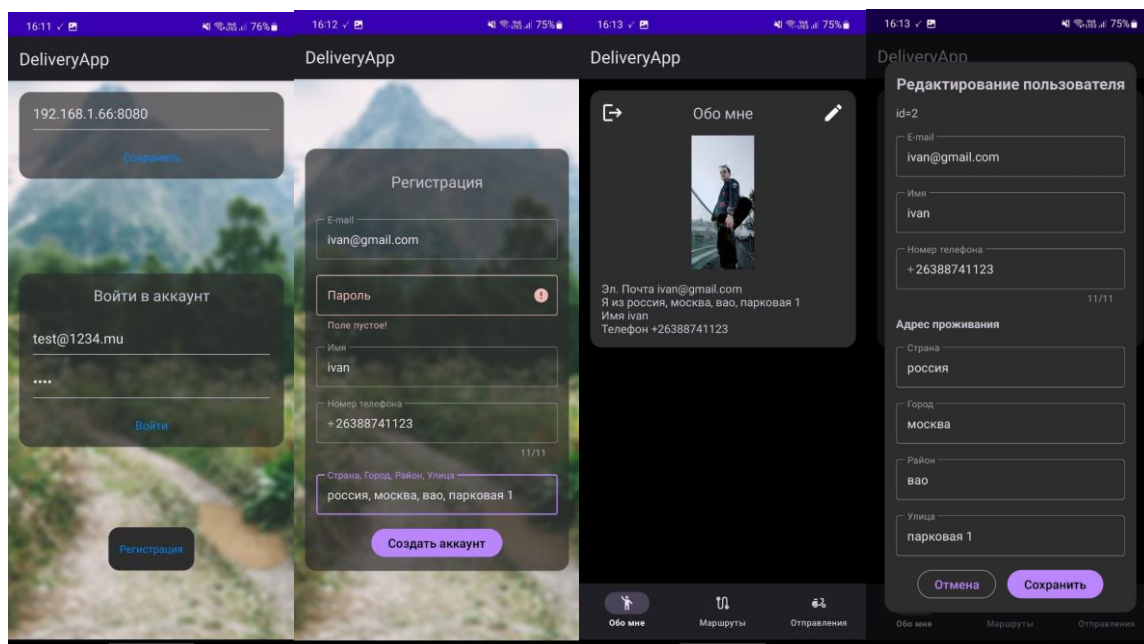


Рисунок 19 - Экраны входа пользователя в аккаунт и его регистрации слева, просмотр данных о пользователе и окно редактирования пользователя справа

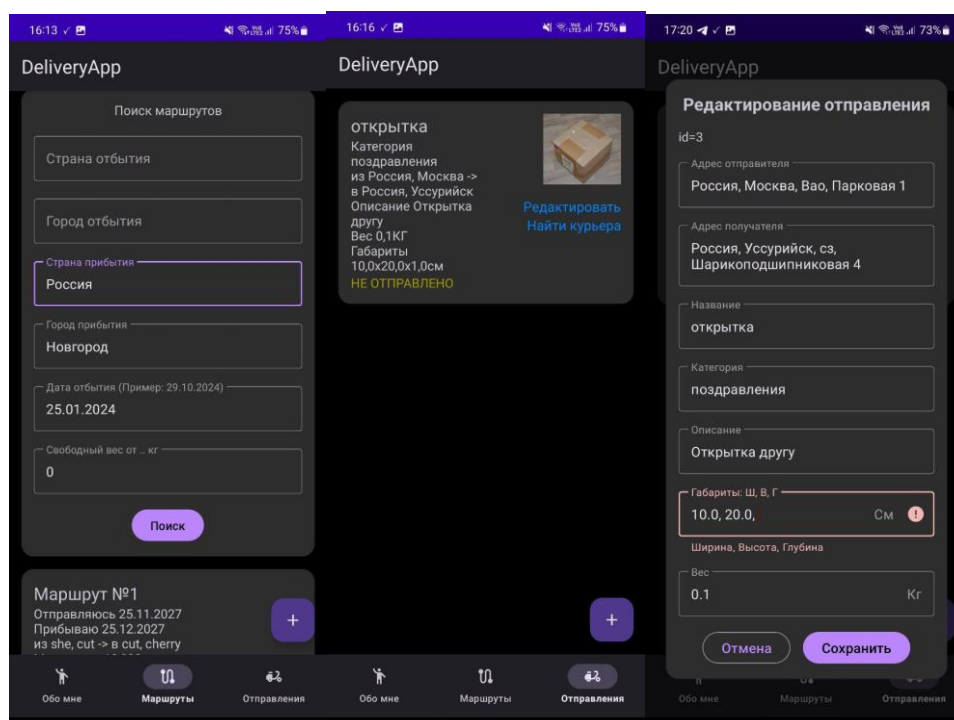


Рисунок 20 - Экран поиска подходящего маршрута слева, просмотра посылок пользователя в центре и диалог редактирования посылок справа

### 5.2.5 Навигация

Совместно с архитектурой MVVM в Android нередко используют схему навигации `NavigationGraph`. Это новейший подход проектирования связей и переходов между экранами приложения применимый только в `SingleActivity` приложениях.

Принцип его состоит в том, что программист с помощью языка разметки XML, описывает все переходы (action) между экранами (Fragments).

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/register_navigation"
    app:startDestination="@id/loginFragment">

    <!-- Регистрация -->
    <fragment
        android:id="@+id/loginFragment"
        android:name="com.texnar13.deliveryapp.ui.LoginFragment"
        android:label="fragment_login"
        tools:layout="@layout/fragment_login">
        <action
            android:id="@+id/action_loginFragment_to_userFragment"
            app:destination="@id/fragment_user"
            app:enterAnim="@anim/nav_default_enter_anim"
            app:exitAnim="@anim/nav_default_exit_anim"
            app:popEnterAnim="@anim/nav_default_pop_enter_anim"
            app:popExitAnim="@anim/nav_default_pop_exit_anim" />
        <action
            android:id="@+id/action_loginFragment_to_registerFragment"
            app:destination="@id/registerFragment" />
    </fragment>
    <fragment
```

Рисунок 21 - Пример описания переходов (Action) для экрана (Fragment) аутентификации пользователя.

На рисунке 13 первый переход описывает переход на страницу пользователя после аутентификации, а второй описывает переход на страницу регистрации нового пользователя.

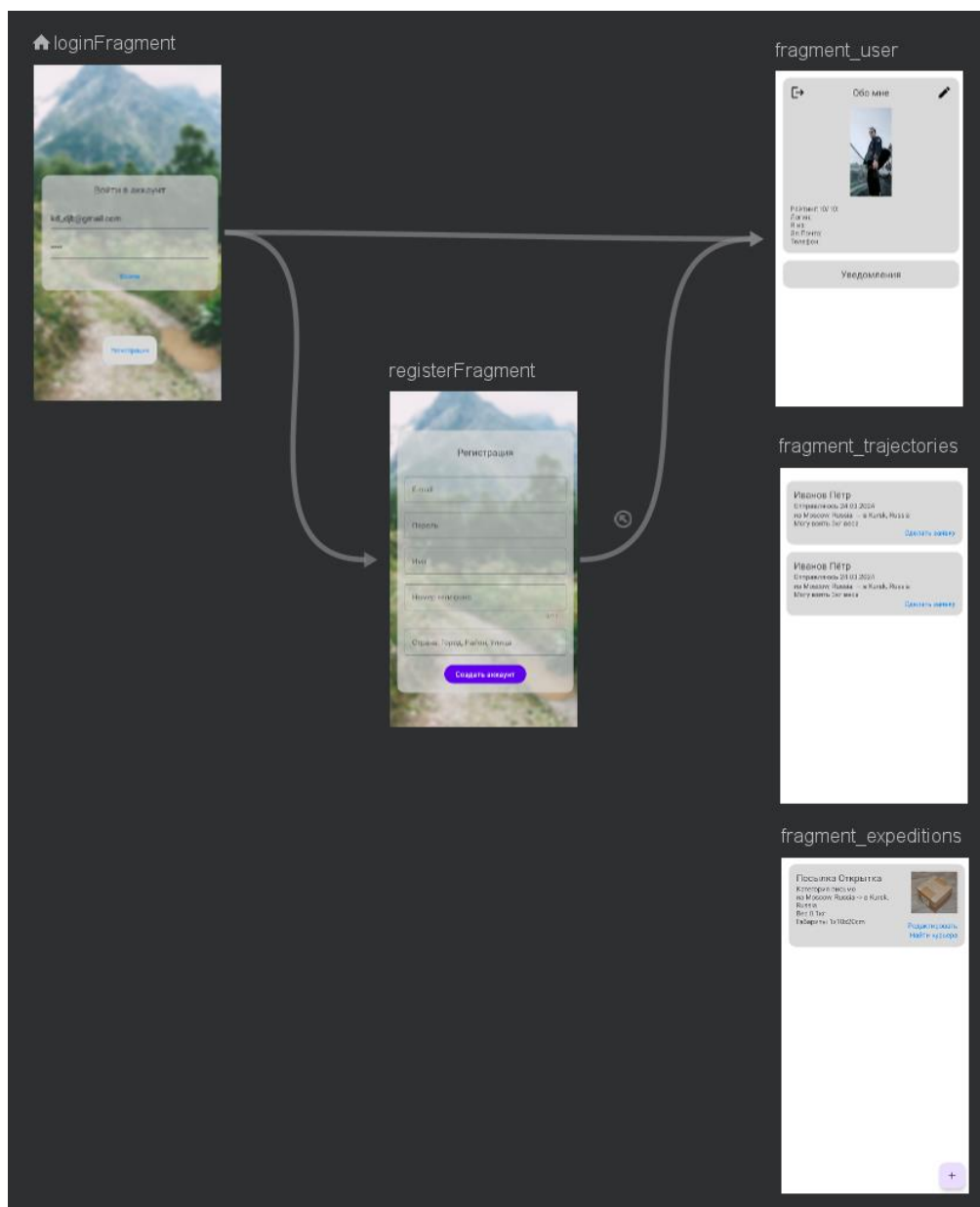


Рисунок 22 - Визуализация схемы переходов между экранами Navigation.

Однако как видно из рисунка выше, экраны пользователя, маршрутов и отправлений никак друг с другом не связаны. Сделано это по тому, что за навигацию между ними отвечает специальный компонент Navigation View, у которого есть готовая логика для реализации Navigation Graph.

### 5.2.6 Работа с REST API

Как писалось выше для работы с Rest API я использовал библиотеку OKHttp. В основе работы с ней стоят принципы асинхронной работы поскольку загрузка данных с сервера может быть достаточно долгой. В потоке отрисовки интерфейса HTTP запросы отправлять нельзя. За отправку запросов из разметки к API и модели отвечает ViewModel которая следит за состоянием соединения и уведомляет разметку.

Для примера рассмотрим механизм загрузки пользовательских отправлений.

В ExpeditionsFragment происходило обращение к ViewModel:

```
// получаем отправления пользователя
mainViewModel.loadUserExpeditions()
```

Рисунок 23 - Обращение к viewModel.

Во ViewModel происходило асинхронное обращение к API:

```
// ----- Страницка отправлений -----
// -----
// загрузить отправления пользователя
fun loadUserExpeditions() {
    // Марчук Иван Сергеевич +1
    // обнуляем
    currentUserExpeditions.postValue(value: null)

    val token = token.value
    if (token != null) {
        // Загружаем отправления пользователя
        viewModelScope.launch {
            httpClient.loadUserExpeditions(token)
        }
    }
}

// ответ Http загрузчика
override fun httpLoadUserExpeditionsFailure(
    errorCode: HttpApi.Companion.ErrorCode,
    status: String
) {
    when (errorCode) {
        HttpApi.Companion.ErrorCode.TOKEN_EXPIRED -> {
            sendToast("Сессия истекла: $status")
            // Выходим из текущего пользователя
            logout()
        }
        else -> sendToast("Ошибка: $status")
    }
}
```

Рисунок 24 - Обращение viewModel к API и метод обратной связи в случае неудачной попытки



При успешном или неудачном обращении HTTP клиент вызывал соответствующий метод во ViewModel, которая в свою очередь уведомляла разметку через LiveData.

```
// ответ Http загрузчика
override fun httpLoadUserExpeditionsSuccess(expeditions: List<EntityExpedition>) {
    currentUserExpeditions.postValue(expeditions)
}
```

Рисунок 25 - Метод обратной связи в случае удачной попытки вызывающий обновление переменной содержащей список посылок пользователя, на которую подписана разметка

```
// Создаём запрос
val request = Request.Builder()
    .url("$serverAddress/package_by_sender_id") // URL
    .addHeader( name: "Authorization", value: "Bearer $token") // JWT токен
    .get() // GET запрос
    .build()
```

Рисунок 26 - Пример инициализации GET запроса на сервер (получение посылок пользователя)

Пример инициализации POST запроса на сервер (редактирования пользователя)

```
// Создаём тело запроса с медиатипом JSON
val requestBody = RequestBody.create(
    "application/json; charset=utf-8".toMediaType(),
    // Тело запроса в формате JSON
    JSONObject().apply {
        put( name: "email", editedUserData.email)
        put( name: "name", editedUserData.name)
        put( name: "phone", editedUserData.phoneNumber)
        put( name: "country", editedUserData.address.getCountry())
        put( name: "city", editedUserData.address.getCity())
        put( name: "district", editedUserData.address.getDistrict())
        put( name: "street", editedUserData.address.getStreet())
    }.toString()
)

// Создаём запрос
val request = Request.Builder()
    .url("$serverAddress/update_user")
    .addHeader( name: "Authorization", value: "Bearer $token") // Добавляем заголовок с JWT токеном
    .post(requestBody)
    .build()
```

Рисунок 27 - Пример инициализации POST запроса на сервер (редактирования пользователя)

Пример вызова обратной связи у подписчика (подписчиком API является ViewModel) после того, как HTTP клиент получил (или не получил) ответ сервера.

```
// Отправляем запрос асинхронно
client.newCall(request).enqueue(object : Callback {
    override fun onFailure(call: Call, e: IOException) {
        // Обработка ошибки подключения
        httpResultAndStatusListener?.httpEditUserDataFailure(
            ErrorCode.UNDEFINED_ERROR,
            status: "${e.message}"
        )
        // Выставляем статус
        updateStatus(HttpClientState.NO_WORK)
    }

    override fun onResponse(call: Call, response: Response) {
        // Обработка ответа сервера
        if (response.isSuccessful) {
            // Получаем JSON ответ
            val jsonObject = JSONObject(response.body!!.string())
            httpResultAndStatusListener?.httpEditUserDataSuccess()
        } else {
            when (response.code) {
                400 -> {
                    httpResultAndStatusListener?.httpEditUserDataFailure(
                        ErrorCode.BAD_REQUEST,
                        status: "[${response.code}] некорректный запрос"
                    )
                }
                401 -> {
                    httpResultAndStatusListener?.httpEditUserDataFailure(
```

Рисунок 28 - Методы обратной связи в API которые вызывает HTTP клиент

## СПИСОК ЛИТЕРАТУРЫ

1. Документация Docker. Официальный сайт [Электронный ресурс] – Режим доступа: <https://docs.docker.com> (дата обращения: 17.08.2024)
2. Документация GitHub Actions. Официальный сайт [Электронный ресурс] – Режим доступа: <https://docs.github.com/actions> (дата обращения: 17.08.2024)
3. Docker Hub. Официальный сайт [Электронный ресурс] – Режим доступа <https://hub.docker.com> (дата обращения: 17.08.2024)
4. Язык программирования Go. Официальный сайт [Электронный ресурс] – Режим доступа <https://golang.org> (дата обращения: 17.08.2024)
5. Спецификация формата YAML. Официальная документация [Электронный ресурс] – Режим доступа <https://yaml.org> (дата обращения: 17.08.2024)
7. Мартина Фаулера. [Электронный ресурс] – Режим доступа <https://martinfowler.com/articles/continuous-integration.html> (дата обращения: 17.08.2024)
8. Alpine Linux. Официальный сайт. [Электронный ресурс] – Режим доступа: <https://alpinelinux.org> (дата обращения: 17.08.2024)
9. Введение в JSON Web Token (JWT). Официальный сайт. [Электронный ресурс] – Режим доступа: <https://jwt.io> (дата обращения: 17.08.2024)
10. Документация Docker Buildx. Официальный сайт. [Электронный ресурс] – Режим доступа: <https://docs.docker.com/buildx/> (дата обращения: 17.08.2024)
11. Шаблоны архитектуры микросервисов. Блог Сэма Ньюмана. [Электронный ресурс] – Режим доступа: <https://samnewman.io/books/building-microservices> (дата обращения: 17.08.2024)

12. Официальный сайт библиотеки OkHttp. [Электронный ресурс] –  
Режим доступа: <https://square.github.io/okhttp> (дата обращения:  
17.08.2024)