

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

**Информационная система организации
грузовых перевозок**

Студент _____ Бичинов Константин Сергеевич
(подпись, дата) (ФИО полностью)

Руководитель _____ Генералова А.А.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от 03.06.2026 № 12)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой

А.Ю. Козлов
« » _____ 2026 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Бичинов Константин Сергеевич гр. 22ВП2 факультета ВТ
<Фамилия, имя, отчество полностью>
2. Тема работы Информационная система организации грузовых перевозок

Тема утверждена приказом ПГУ № 1583/о от "30" декабря 2025 г.

3. Руководитель работы к.т.н., доцент кафедры «МО и ПЭВМ» Генералова А. А.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Назначение: программное средство для обеспечения взаимодействия клиентов с сервисом транспортной компании, в виде IDE с возможностью компиляции и запуска.

2) Среда выполнения: операционная система Windows

Языковые средства разработки: Python, JavaScript, SQL, UML

Инструментальные средства: PyCharm, PgAdmin 4, Visual Paradigm

Технологии разработки: микросервисная архитектура, ООП.

3) Основные функции приложения для клиента:

а) просмотр справочных данных;

б) создание заказов;

в) просмотр истории заказов;

Основные функции приложения для администратора:

а) операции создания, чтения, редактирования и удаления над типами грузов, маршрутами и транспортом;

б) отслеживание и изменение статусов заказов

в) просмотр списка пользователей.

5. Перечень подлежащих разработке вопросов

- 1) Анализ предметной области и требований к системе;*
- 2) Проектирование программных средств;*
- 3) Реализация программных средств;*
- 4) Тестирование программных средств;*
- 5) Оформление пояснительной записки.*

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	08.04.2026 - 14.04.2026	
2	Проектирование программных средств	20%	15.04.2026 - 25.04.2026	
3	Реализация программных средств	30%	26.04.2026 - 12.05.2026	
4	Тестирование программных средств	20%	13.05.2026 - 22.05.2026	
5	Оформление пояснительной записки	20%	23.05.2026 - 03.06.2026	

Дата выдачи задания «26» декабря 2025г.

Руководитель бакалаврской работы

26.12.2025 г.

(подпись, дата)

Задание к исполнению принял студент

26.12.2025 г.

(подпись, дата)

Работу к защите допустить
Декан ФВТ

(подпись, дата)

Л.Р. Фионова

Реферат

Пояснительная записка содержит 105 листов, 26 рисунков, 12 таблиц, 20 использованных источников и 2 приложения.

ИНФОРМАЦИОННАЯ СИСТЕМА, ТРАНСПОРТНАЯ КОМПАНИЯ, МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, PYTHON, FASTAPI, REACT, POSTGRESQL, REST API, API GATEWAY, ГРУЗОПЕРЕВОЗКИ, УПРАВЛЕНИЕ ЗАКАЗАМИ, ВЕБ-ПРИЛОЖЕНИЕ.

Объект исследования – процессы организации и автоматизации деятельности транспортной компании в сфере грузоперевозок.

Цель выпускной квалификационной работы – спроектировать и разработать информационную систему транспортной компании, обеспечивающую поддержку основных логистических процессов и взаимодействие пользователей с сервисом.

Технологии разработки – реализация серверной части системы выполнена в среде разработки PyCharm 2025.2.3 на языке программирования Python с использованием фреймворка FastAPI. Для хранения данных используется система управления базами данных PostgreSQL. Клиентская часть разработана с использованием библиотеки React. Развертывание приложения осуществляется посредством контейнеризации с использованием Docker на операционной системе Windows.

Среда исполнения – персональный компьютер пользователя.

Результаты работы – в ходе выполнения выпускной квалификационной работы была разработана информационная система транспортной компании, реализующая функции управления грузами, маршрутами, транспортом и заказами пользователей.

					ПГУ 09.03.04 – 08BP222.04 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Информационная система организации грузовых перевозок Пояснительная записка	Лит.	Лист	Листов
Разраб.		Бичинов К. С.					4	105
Пров.		Генералова А.А.				Группа 22ВП2		
Реценз.								
Н. Контр.		Такташкин Д.В						
Утв.		Козлов А.Ю.						

Оглавление

Введение	7
1 Анализ предметной области и требований на разработку информационной системы организации грузовых перевозок	9
1.1 Анализ предметной области и постановка задачи.....	9
1.2 Анализ аналогов.....	10
1.2.1 ATI.SU	10
1.2.2 Яндекс Доставка	11
1.2.3 TRAFIT-Online	12
1.2.4 Результаты сравнительного анализа аналогов.....	14
2 Анализ требований	16
2.1 Анализ функциональных требований.....	16
2.2 Анализ нефункциональных требований	19
3 Проектирование приложения	20
3.1 Высокоуровневое проектирование	20
3.2 Проектирование серверной части	21
3.2.1 Проектирование микросервиса справочника грузов, маршрутов и транспорта (cargo_service).....	21
3.2.2 Проектирование микросервиса заказов (order_service)	26
3.2.3 Проектирование микросервиса аутентификации (auth_service).....	28
3.2.4 Проектирование микросервиса профилей пользователей (user_profile_service).....	30
3.3 Проектирование клиентской части	33
4 Реализация приложения.....	36
4.1 Реализация серверной части приложения	36

4.2	Реализация клиентской части приложения	39
4.3	Контейнеризация и развертывание системы	40
5	Тестирование	45
5.1	Тестирование методом чёрного ящика	45
5.1.1	Функциональная диаграмма и таблица решений	45
5.1.2	Автоматизированное тестирование	49
5.1.3	Анализ сложности кода модуля регистрации	52
6	Планирование процесса разработки	58
	Заключение	62
	Список использованных источников	64
	Приложение А (обязательное) Листинг приложения	67
	Приложение Б (необязательное) Результаты тестирования программы	97

Введение

Современный этап цифровой трансформации бизнеса характеризуется активным внедрением информационных систем, направленных на автоматизацию ключевых бизнес-процессов и повышение эффективности управления. В сфере транспортной логистики особое значение приобретает использование информационных систем, обеспечивающих централизованное управление данными о грузах, маршрутах, транспорте и заказах, а также предоставляющих удобные инструменты взаимодействия между клиентами и администраторами компании. При этом многие существующие решения перегружены сложной семантикой интерфейса, что затрудняет работу пользователей. Актуальность данной дипломной работы обусловлена необходимостью разработки современной, удобной для пользователя информационной системы с простым и интуитивно понятным интерфейсом, способной одновременно обеспечить прозрачность процессов, удобство оформления и отслеживания заказов, а также предоставить административные инструменты для управления справочными данными и контроля бизнес-процессов.

Объектом исследования в данной работе являются процессы проектирования и разработки распределенных информационных систем.

Предметом исследования является разработка информационной системы транспортной компании с использованием современных технологий клиент-серверной веб-разработки.

Целью дипломной работы является разработка и практическая реализация информационной системы транспортной компании, обеспечивающего управление грузами, маршрутами, транспортом и заказами пользователей, с применением механизмов аутентификации и ролевой модели доступа.

Для достижения поставленной цели в работе необходимо решить следующие задачи:

- провести анализ предметной области транспортной компании и определить основные требования к разрабатываемой системе;
- сформулировать функциональные и нефункциональные требования к информационной системе;
- спроектировать архитектуру системы;
- реализовать серверную часть приложения, включая аутентификацию и управление пользователями, грузами, маршрутами, транспортом, заказами, а также API-шлюза;
- реализовать клиентскую часть приложения и обеспечить ее взаимодействие с серверной частью через единый API;
- разработать механизмы безопасности, включая аутентификацию на основе JWT-токенов и разграничение прав доступа между пользователями и администраторами;
- организовать тестирование основных функциональных сценариев работы системы для подтверждения ее корректности и работоспособности.

Методология и инструменты исследования основаны на использовании современных технологий веб-разработки. В качестве языка программирования серверной части используется Python, а в качестве веб-фреймворка – FastAPI. Для хранения данных применяется реляционная система управления базами данных PostgreSQL. Клиентская часть приложения реализована с использованием библиотеки React и сборщика Vite, а для оформления пользовательского интерфейса применяется Tailwind CSS. Все компоненты системы контейнеризированы с помощью Docker.

1 Анализ предметной области и требований на разработку информационной системы организации грузовых перевозок

1.1 Анализ предметной области и постановка задачи

В рамках выполнения выпускной квалификационной работы необходимо разработать информационную систему организации грузовых перевозок, предназначенную для автоматизации процессов оформления, сопровождения и администрирования заказов на перевозку грузов.

В качестве предметной области рассматривается деятельность транспортной компании, осуществляющей организацию грузоперевозок. В разрабатываемой системе выделяются две основные категории пользователей: клиенты и администраторы.

Клиенты являются основной категорией пользователей системы и формируют заказы на перевозку грузов. Им предоставляется возможность просматривать доступные виды грузов, маршруты перевозок и транспортные средства, создавать заказы, отслеживать их текущее состояние, а также управлять собственными заказами и просматривать историю перевозок.

Администраторы осуществляют управление информационной системой и контролируют процессы организации перевозок. В их задачи входит управление данными о грузах, маршрутах и транспортных средствах, просмотр информации о пользователях и заказах, изменение статусов заказов и контроль выполнения логистических процессов.

Особенностью рассматриваемой предметной области является необходимость обработки большого количества информации, связанной с перевозками, включая сведения о грузах, транспортных средствах, маршрутах и оформленных заказах. При этом система должна обеспечивать удобный механизм взаимодействия пользователей с сервисом, а также предоставлять инструменты централизованного управления данными.

Основной функционал управления перевозками будет реализован в серверной части приложения. Для обеспечения взаимодействия пользователей с

системой также требуется разработка клиентской части, включающей средства просмотра данных, оформления заказов и администрирования элементов системы.

Таким образом, в результате выполнения работы должна быть разработана информационная система организации грузовых перевозок, обеспечивающая автоматизацию процессов оформления заказов, управления логистическими данными и сопровождения перевозок.

1.2 Анализ аналогов

1.2.1 ATI.SU

ATI.SU – цифровая платформа для организации грузоперевозок, объединяющая участников транспортно-логистической отрасли. Система предоставляет инструменты поиска заказов, обмена документами, мониторинга перевозок и взаимодействия между участниками перевозочного процесса [1].

К преимуществам ATI.SU можно отнести:

- наличие инструментов поиска и размещения грузов;
- поддержку мониторинга грузов;
- наличие средств электронного документооборота;
- возможность взаимодействия между участниками перевозок.

Ниже на рисунке 1 приведен интерфейс системы ATI.SU с отображением списка грузов и перевозок.

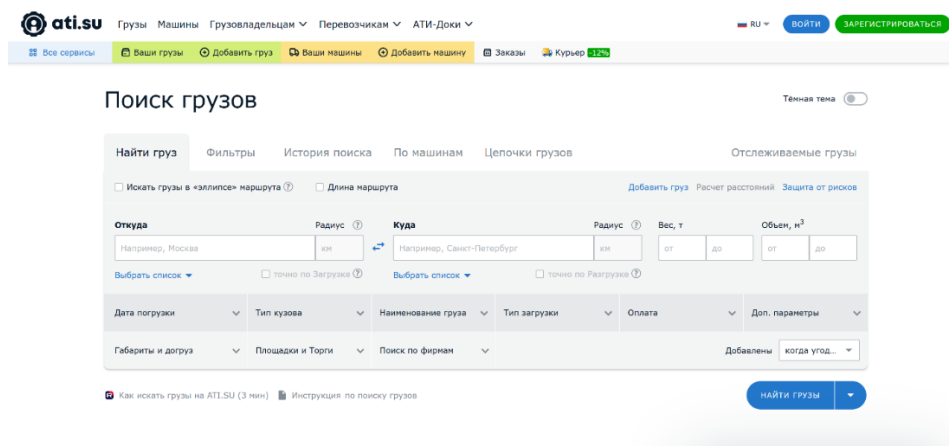


Рисунок 1 – Просмотр грузов и маршрутов в ATI.SU

К недостаткам системы относятся:

- сложный интерфейс для пользователей без опыта работы в логистике;
- ориентация преимущественно на взаимодействие между перевозчиками;
- отсутствие специализированного интерфейса для внутреннего управления транспортной компанией.

1.2.2 Яндекс Доставка

Яндекс Доставка – сервис организации доставки и перевозок, предоставляющий пользователям возможность оформления заказов и отслеживания процесса доставки через веб-интерфейс [2].

К преимуществам сервиса относятся:

- удобный интерфейс оформления заказа;
- возможность отслеживания статуса перевозки;
- наличие информации о маршруте и стоимости перевозки;
- простота взаимодействия пользователей с системой.

Ниже на рисунке 2 приведен интерфейс оформления заказа в сервисе Яндекс Доставка.

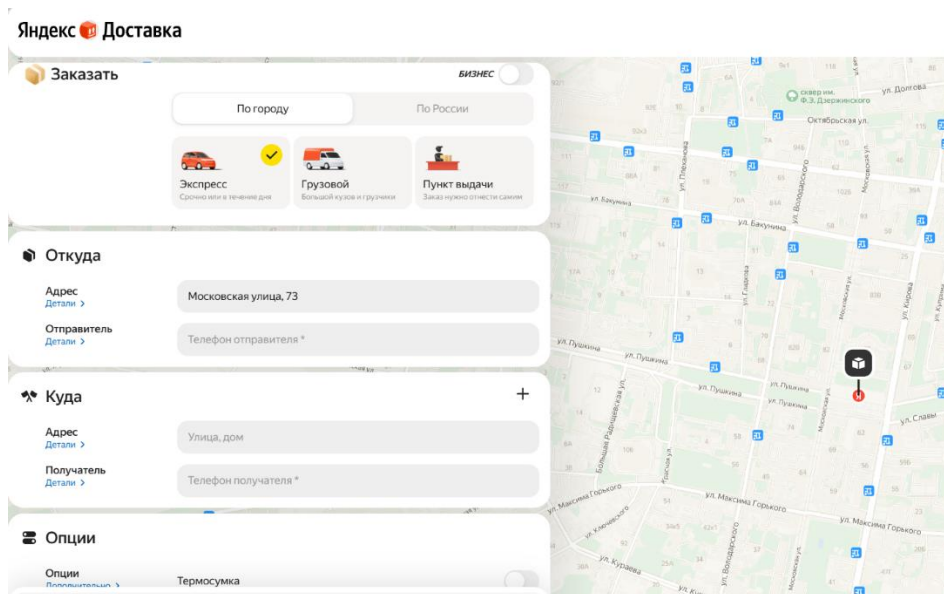


Рисунок 2 – Интерфейс оформления перевозки в Яндекс Доставке

Также сервис поддерживает отслеживание текущего состояния заказа и информации о перевозке.

Ниже на рисунке 3 приведен интерфейс отслеживания заказа.

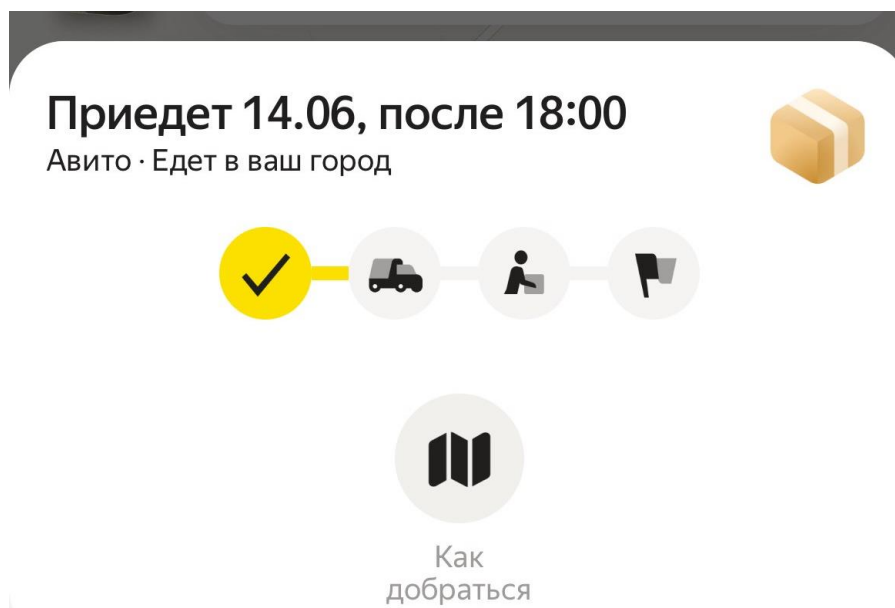


Рисунок 3 – Интерфейс отслеживания перевозки в Яндекс Доставке

К недостаткам системы относятся:

- ориентация преимущественно на доставку, а не управление транспортной компанией (выступает в роли агрегатора);
- отсутствие административных функций управления транспортом и грузами;
- ограниченные возможности администрирования.

1.2.3 TRAFТ-Online

TRAFТ-OnLine – российская информационная система управления грузоперевозками, предназначенная для автоматизации логистических процессов и управления транспортными операциями. Система ориентирована преимущественно на транспортные и логистические компании и предоставляет инструменты управления заказами, мониторинга перевозок и контроля логистических процессов [3].

К преимуществам системы относятся:

- возможность централизованного управления заказами и перевозками;
- наличие инструментов мониторинга транспортных операций;
- поддержка электронного документооборота;
- наличие средств аналитики и контроля логистических процессов;
- поддержка интеграции с корпоративными системами.

Ниже на рисунке 4 приведен интерфейс личного кабинета системы TRAFT-OnLine.



Рисунок 4 – Интерфейс личного кабинета системы TRAFT-OnLine

Также система поддерживает функционал мониторинга перевозок и контроля транспортных операций. Пользователям доступны инструменты отслеживания маршрутов, просмотра статусов перевозок и управления логистическими процессами.

Ниже на рисунке 5 приведен интерфейс мониторинга перевозок в системе TRAFT-OnLine.

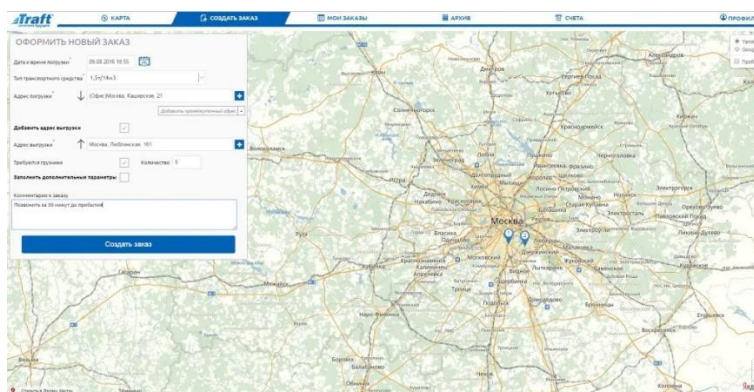


Рисунок 5 – Интерфейс мониторинга перевозок в системе TRAFT-OnLine

К недостаткам системы относятся:

- ориентация преимущественно на крупные транспортные и логистические компании;
- высокая сложность интерфейса;
- наличие избыточного функционала для небольших организаций и частных пользователей;
- необходимость адаптации системы под бизнес-процессы конкретной компании.

1.2.4 Результаты сравнительного анализа аналогов

На основе проведенного анализа аналогов была сформирована таблица 1 со сравнением функционала программных средств [1 – 3].

Таблица 1 – Сравнение систем распределения обращений

Критерий	ATI.SU	Яндекс Доставка	TRAFT- OnLine
Возможность создания заказов	Да	Да	Да
Возможность отслеживания статуса перевозок	Да	Да	Да
Критерий	ATI.SU	Яндекс Доставка	TRAFT- OnLine
Возможность работы с перевозчиками	Да	Нет	Да
Наличие административного функционала	Ограничено	Нет	Да
Поддержка ЭДО	Да	Нет	Да
Поддержка аналитики и мониторинга	Ограничено	Нет	Да
Наличие API / возможности интеграции	Да	Нет	Да
Сложность интерфейса	Высокая	Низкая	Высокая

Продолжение таблицы 1

Критерий	ATI.SU	Яндекс Доставка	TRAFT- OnLine
Доступность без корпоративного подключения	Да	Да	Нет

По результатам сравнения программных средств аналогичного назначения не было найдено ни одной системы, которая удовлетворяла бы всем требованиям. Поэтому было решено разработать собственное программное средство, которое будет удовлетворять всем рассмотренным критериям.

2 Анализ требований

2.1 Анализ функциональных требований

Согласно заданию, разрабатываемое приложение должно соответствовать следующим функциональным требованиям:

- возможность регистрации и авторизации пользователей и администратора;
- просмотр информации о грузах, маршрутах и транспорте;
- создание заказов пользователями;
- управление справочными данными (грузы, маршруты, транспорт) администраторами;
- просмотр своих заказов пользователями и всех заказов администраторами;
- изменение статуса заказов администраторами.

По заданным требованиям была составлена диаграмма вариантов использования [4], представленная на рисунке 6.

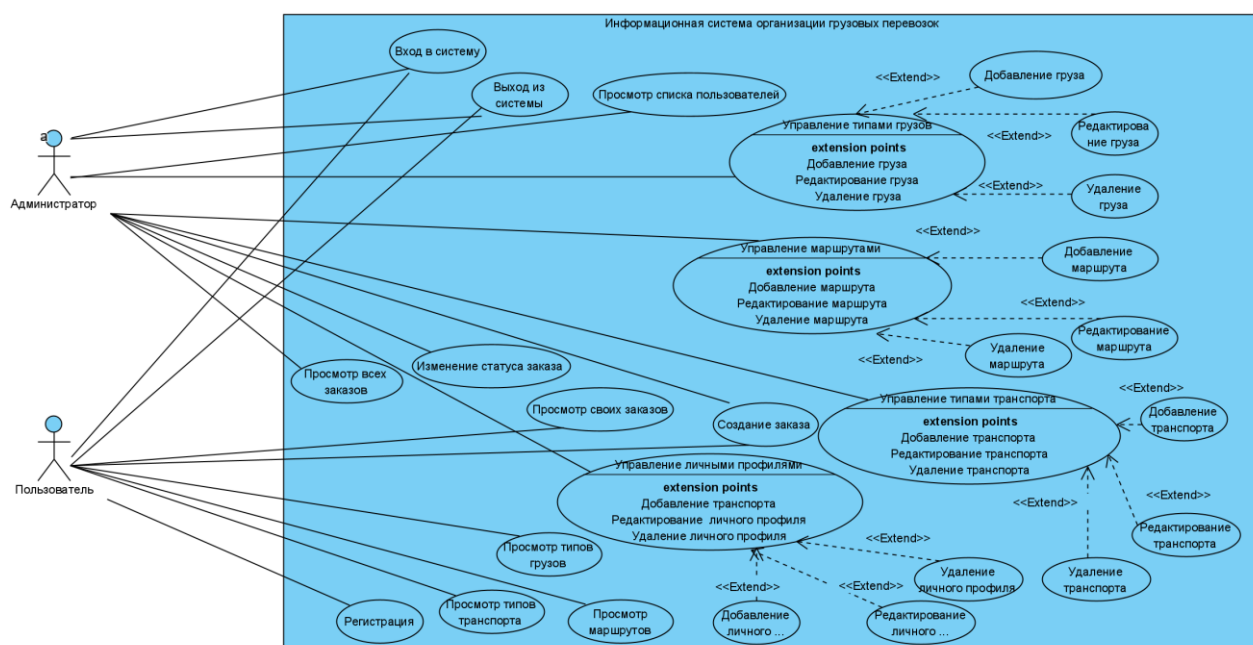


Рисунок 6 – Диаграмма вариантов использования приложения

В таблицах 2 – 4 описаны спецификации некоторых вариантов использования:

Таблица 2 – Спецификация варианта использования «Вход в систему»

Поле	Значение
Название	Вход в систему
Цель	Получить доступ к функционалу системы, авторизовавшись под своей учетной записью
Основной поток	<ul style="list-style-type: none"> – пользователь открывает страницу входа – пользователь вводит логин и пароль – система проверяет корректность учетных данных – система формирует сессию или JWT-токен – пользователь получает доступ к функционалу системы
Дополнительные действия	– система блокирует доступ при превышении количества неверных попыток – пользователь может восстановить пароль

Таблица 3 – Спецификация варианта использования «Управление грузами (CRUD)» (администратор)

Поле	Значение
Название	Управление грузами (CRUD)
Цель	Добавлять, изменять, удалять и просматривать данные о грузах
Основной поток	<ul style="list-style-type: none"> – администратор открывает страницу управления грузами – система отображает список грузов – администратор выбирает действие: добавить, изменить, удалить – система выполняет операцию через cargo_service – система подтверждает успешное выполнение действия

Продолжение таблицы 3

Поле	Значение
Дополнительные действия	– система проверяет корректность данных при создании/изменении- система отображает уведомление о возможной ошибке

Таблица 4 – Спецификация варианта использования «Создание заказа»

Поле	Значение
Название	Создание заказа
Цель	Создать новый заказ на перевозку груза
Основной поток	<ul style="list-style-type: none"> – пользователь или администратор открывает страницу создания заказа – система отображает форму для ввода данных заказа – пользователь/администратор вводит необходимые параметры (груз, маршрут, транспорт) – система сохраняет заказ через order_service – система подтверждает успешное создание заказа
Дополнительные действия	– возможность отменить создание заказа до отправки данных

Таким образом, разработанная система обладает четко определённой функциональной структурой, включающей основные пользовательские сценарии и административные функции. Рассмотренные варианты использования охватывают ключевые процессы взаимодействия пользователей с системой, такие как авторизация, управление справочными данными и оформление заказов на перевозку. Описанные спецификации позволяют формализовать поведение системы и служат основой для дальнейшей реализации серверной и клиентской части приложения.

2.2 Анализ нефункциональных требований

При разработке приложения необходимо провести систематическую идентификацию и анализ ключевых нефункциональных требований, которые лягут в основу архитектурного проектирования системы. Среди основных требований можно выделить следующие [5]:

- безопасность – доступ к административным функциям и API должен быть защищён от несанкционированного доступа. Для обеспечения безопасности используется JWT-аутентификация [10] и проверка прав доступа на серверной стороне;
- кроссплатформенность – приложение должно корректно работать на различных устройствах и браузерах. Достигается использованием web-технологий (React, Vite, Tailwind) и адаптивной версткой страниц;
- производительность – система должна обеспечивать быстрый отклик при работе с данными пользователей и заказов, а также при отображении списков грузов, маршрутов и транспорта;
- масштабируемость – архитектура приложения на базе микросервисов [8] позволяет расширять функционал и увеличивать нагрузку без значительного изменения существующих компонентов;
- удобство использования – интерфейс должен быть интуитивно понятен пользователям и администраторам, обеспечивая user-friendly взаимодействие с функционалом системы.

3 Проектирование приложения

3.1 Высокоуровневое проектирование

Для удовлетворения требований к разрабатываемому приложению была выбрана микросервисная архитектура. Микросервисная архитектура (MSA, microservices architecture) – это архитектура программного обеспечения, которая состоит из комплекса отдельных модулей (микросервисов), обеспечивающих выполнение независимых функций. Такая распределённая система отличается гибкостью в разработке и широкими возможностями масштабирования [6]. Для отображения выделяемых в приложении сервисов была составлена диаграмма компонентов (рисунок 7).

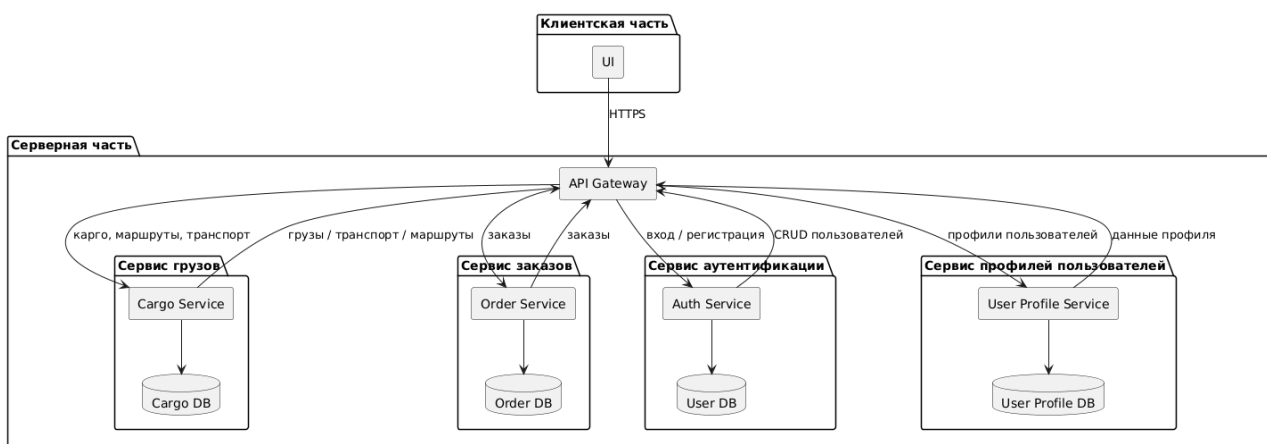


Рисунок 7 – Диаграмма компонентов информационной системы грузоперевозок

Согласно диаграмме, в приложении выделяются следующие компоненты:

- клиентскую часть приложения представляет веб-приложение на React с использованием Vite и Tailwind, через которое пользователь и администратор имеют возможность просматривать грузы, маршруты, транспорт, создавать и управлять заказами;
- взаимодействие между клиентской частью и серверной осуществляется через API-Gateway, который предоставляет унифицированный интерфейс для запросов к микросервисам;

- микросервис аутентификации (Auth Service) управляет регистрацией, входом в систему, обновлением и валидацией JWT-токенов [10];
- микросервис грузов (Cargo Service) управляет справочником грузов, маршрутов и транспорта, включая возможность добавления, редактирования, удаления и просмотра записей;
- микросервис заказов (Order Service) обрабатывает создание заказов, хранение и изменение их статусов, а также предоставляет пользователю и администратору соответствующую информацию о заказах;
- микросервис профилей пользователей (User Profile Service) позволяет работать с необходимыми для оформления заказа личными данными пользователя.

Выбор данной архитектуры позволяет обеспечить модульность, удобство сопровождения и масштабируемость приложения, а также удовлетворяет всем функциональным и нефункциональным требованиям, предъявленным к системе [7].

3.2 Проектирование серверной части

3.2.1 Проектирование микросервиса справочника грузов, маршрутов и транспорта (cargo_service)

При разработке микросервиса была выбрана многослойная архитектура для четкой структуры компонентов и поддержки высокой модульности программы. Были выделены следующие слои:

- Data Layer (Слой данных) – слой, отвечающий за работу с базой данных (тут хранятся структуры, соответствующие таблицам в БД, репозитории, утилиты, а также точки подключения к БД);
- Presentation Layer (Слой представления) – слой, ответственный за работу с API (хранятся хендлеры для обработки запросов и модели запросов и ответов API).

Структура микросервиса представлена на диаграмме классов (рисунок 8).

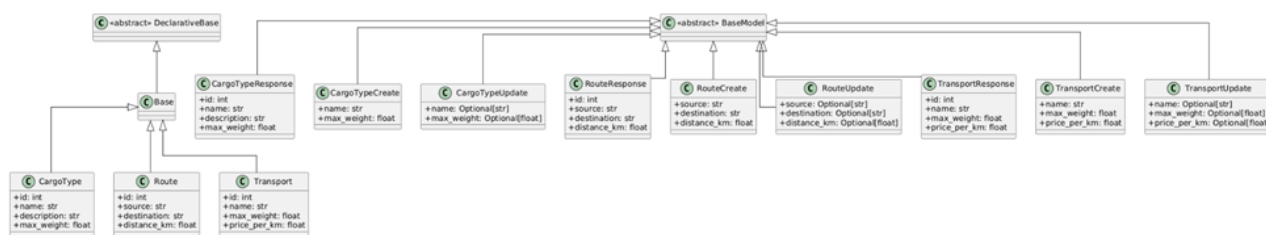


Рисунок 8 – Диаграмма классов микросервиса справочника грузов, маршрута и транспорта (cargo_service)

Согласно данной схеме можно выделить следующие составные элементы:

- DeclarativeBase: Абстрактный базовый класс для всех SQLAlchemy моделей. Задаёт общий интерфейс и обеспечивает интеграцию с базой данных.
- Base: Класс, наследуемый от DeclarativeBase, хранит метаданные всех таблицы БД, позволяя проще описывать модели таблиц.
- BaseModel: Абстрактный базовый класс для всех Pydantic моделей, предоставляющий функционал валидации данных и конфигурацию ORM.
- CargoType: Основная SQLAlchemy модель типа груза, содержит идентификатор, наименование, описание и максимальный вес.
- Route: Модель маршрута, хранит исходный пункт, пункт назначения и расстояние в километрах.
- Transport: Модель транспорта, включает наименование, максимальный вес и цену за километр.
- CargoTypeResponse / RouteResponse / TransportResponse: Pydantic модели для сериализации данных при возврате ответа API, включают все необходимые поля соответствующих SQLAlchemy моделей.
- CargoTypeCreate / RouteCreate / TransportCreate: Pydantic модели для создания новых сущностей в системе, содержат обязательные поля для инициализации объектов.
- CargoTypeUpdate / RouteUpdate / TransportUpdate: Pydantic модели для обновления существующих сущностей, позволяют изменять отдельные поля через опциональные атрибуты.

Все SQLAlchemy модели наследуются от абстрактного класса Base, что обеспечивает единый интерфейс работы с базой данных. Pydantic модели наследуются от BaseModel и используются для валидации и сериализации данных при взаимодействии с API. Такой подход позволяет разделить внутреннее представление данных и модели обмена информацией, что повышает надежность и поддерживаемость программного кода.

Описание основных запросов микросервиса описано в таблице 5.

Таблица 5 – Методы микросервиса справочника грузов, маршрута и транспорта

Метод	Точка доступа	Описание	Параметры
GET	/cargo	Получить список всех грузов	query: limit (опционально)
GET	/cargo/{cargo_id}	Получить информацию о конкретном грузе	path: cargo_id
POST	/cargo	Создать новый груз	body: CargoTypeCreate
PUT	/cargo/{cargo_id}	Обновить данные существующего груза	path: cargo_id, body: CargoTypeCreate
DELETE	/cargo/{cargo_id}	Удалить груз	path: cargo_id
GET	/cargo/{cargo_id}/image	Получить изображение груза	path: cargo_id

Продолжение таблицы 5

Метод	Точка доступа	Описание	Параметры
PUT	/cargo/{cargo_id}/image	Загрузить изображение груза	path: cargo_id, file: image
DELETE	/cargo/{cargo_id}/image	Удалить изображение груза	path: cargo_id
GET	/cargo/{cargo_id}/image/exists	Проверить наличие изображения	path: cargo_id
GET	/route	Получить список всех маршрутов	query: limit (опционально)
GET	/route/{route_id}	Получить информацию о конкретном маршруте	path: route_id
POST	/route	Создать новый маршрут	body: RouteCreate
PUT	/route/{route_id}	Обновить маршрут	path: route_id, body: RouteCreate
DELETE	/route/{route_id}	Удалить маршрут	path: route_id
GET	/transport	Получить список всех транспортов	query: limit (опционально)

Окончание таблицы 5

Метод	Точка доступа	Описание	Параметры
GET	/transport/{transport_id}	Получить информацию о конкретном транспорте	path: transport_id
POST	/transport	Создать новый транспорт	body: TransportCreate
PUT	/transport/{transport_id}	Обновить данные транспорта	path: transport_id, body: TransportCreate
DELETE	/transport/{transport_id}	Удалить транспорт	path: transport_id
GET	/transport/{transport_id}/image	Получить изображение транспорта	path: transport_id
PUT	/transport/{transport_id}/image	Загрузить изображение транспорта	path: transport_id, file: image
DELETE	/transport/{transport_id}/image	Удалить изображение транспорта	path: transport_id
GET	/transport/{transport_id}/image/exists	Проверить наличие изображения	path: transport_id

Указанные запросы обеспечивают удобные точки доступа для работы с справочником грузов, маршрутов и транспорта.

3.2.2 Проектирование микросервиса заказов (order_service)

При проектировании данного микросервиса, так же поддерживался стиль многослойной архитектуры для организации кода. Слои, выделенные в данном микросервисе полностью идентичны слоям в микросервисе справочника грузов, маршрутов и транспорта.

Структура микросервиса представлена на рисунке 9.

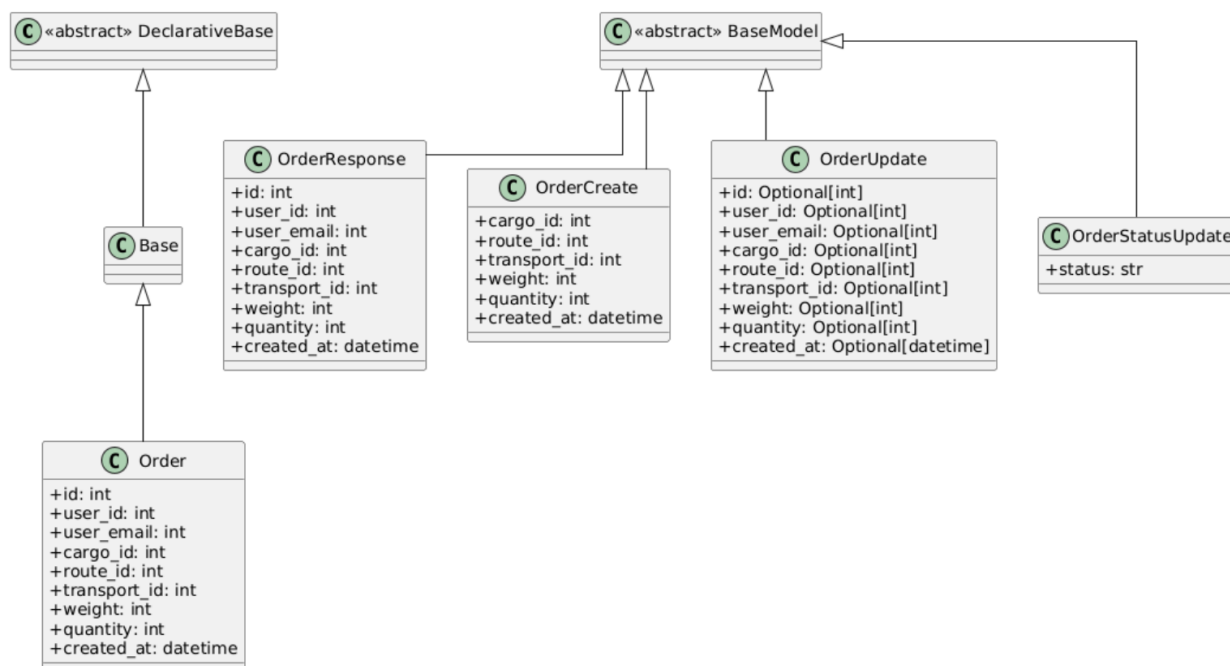


Рисунок 9 – Диаграмма классов микросервиса заказов (order_service)

Описание основных компонентов микросервиса приведены ниже.

`DeclarativeBase` – абстрактный базовый класс SQLAlchemy, обеспечивающий хранение метаданных всех моделей и поддержку ORM-функционала; от него наследуется локальный `Base`.

`Base` – локальный базовый класс микросервиса, от которого наследуются все SQLAlchemy модели (`Order`).

`Order` – основная SQLAlchemy модель для хранения информации о заказах: включает данные о пользователе, грузе, маршруте, транспорте, весе, количестве и дате создания. Все операции CRUD выполняются через этот класс в сочетании с сессией SQLAlchemy.

BaseModel – абстрактный базовый класс для всех Pydantic моделей, предоставляющий функционал валидации данных и конфигурацию ORM.

Pydantic модели:

- OrderResponse – модель для формирования ответа API с полной информацией о заказе;
- OrderCreate – модель запроса на создание нового заказа;
- OrderUpdate – модель запроса на обновление существующего заказа;
- OrderStatusUpdate – модель запроса для изменения статуса заказа.

Методы микросервиса представлены в таблице 6.

Таблица 6 – Методы микросервиса заказов

Метод	Точка доступа	Описание	Параметры
GET	/orders	Получение списка заказов. Для админа возвращаются все заказы, для пользователя – только его собственные.	Заголовки: X-User-Role, X-User-Id
GET	/order/{order_id}	Получение информации о конкретном заказе. Доступ проверяется по роли и владельцу заказа.	Path: order_id; Заголовки: X-User-Role, X-User-Id
POST	/order	Создание нового заказа пользователем.	Body: OrderCreate; Заголовки: X-User-Id, X-User-Email

Продолжение таблицы 6

Метод	Точка доступа	Описание	Параметры
PUT	/order/{order_id}	Обновление существующего заказа. Доступ проверяется по роли и владельцу заказа.	Path: order_id; Body: OrderUpdate; Заголовки: X-User-Role, X-User-Id
DELETE	/order/{order_id}	Удаление заказа. Доступ проверяется по роли и владельцу заказа.	Path: order_id; Заголовки: X-User-Role, X-User-Id
PUT	/order/{order_id}/status	Изменение статуса заказа. Доступно только администратору.	Path: order_id; Body: OrderStatusUpdate; Заголовки: X-User-Role

Данными методами обеспечиваются все основные методы для работы с заказами.

3.2.3 Проектирование микросервиса аутентификации (auth_service)

Данный микросервис так же спроектирован в слоистой архитектуре с слоями, идентичными предыдущим микросервисам. Структура микросервиса представлена на рисунке 10.

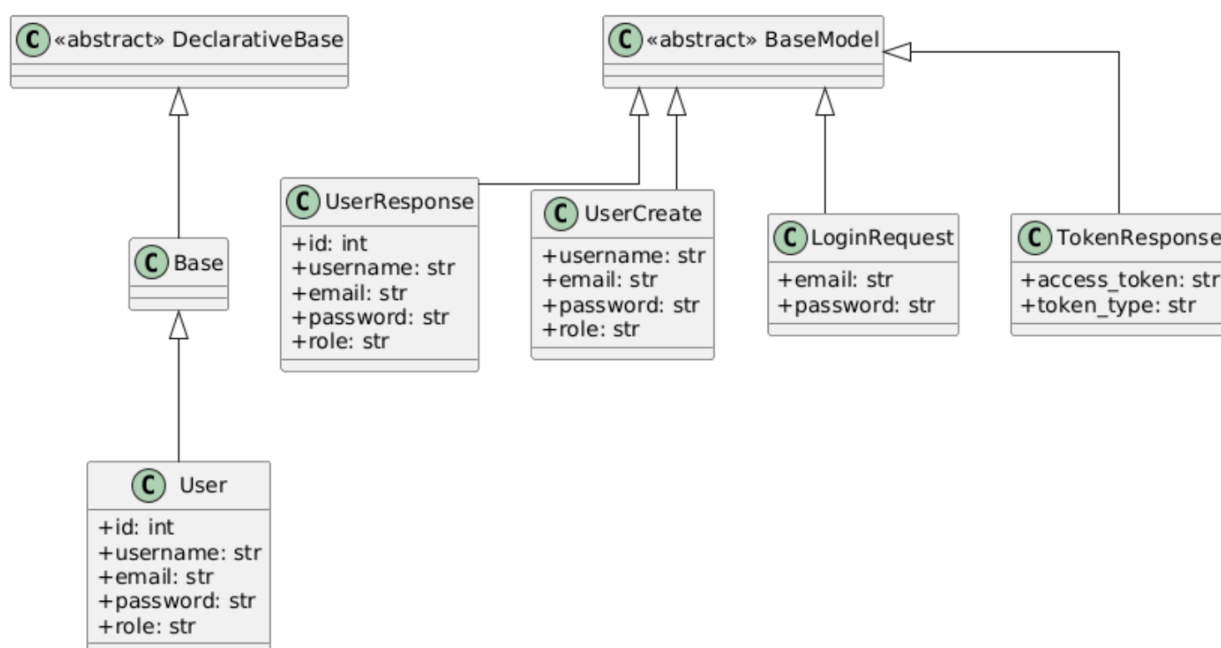


Рисунок 10 – Диаграмма классов микросервиса пользователей (user)

Ниже представлено описание компонентов микросервиса:

- DeclarativeBase – абстрактный базовый класс SQLAlchemy, обеспечивающий хранение метаданных всех моделей и поддержку ORM-функционала; от него наследуется локальный Base;
- Base – локальный базовый класс микросервиса, от которого наследуются все SQLAlchemy-модели (UserProfile);
- User – основная модель пользователя для хранения информации в базе данных (id, username, email, password, role);
- BaseModel – абстрактный базовый класс для всех Pydantic моделей, предоставляющий функционал валидации данных и конфигурацию ORM.

Pydantic-модели:

- UserResponse – модель для ответа API при получении данных о пользователе.
- UserCreate – модель для создания нового пользователя (регистрация).

- LoginRequest – модель запроса для авторизации пользователя по email и паролю.
- TokenResponse – модель ответа при успешной аутентификации, содержащая JWT-токен.

Методы, используемые в данном микросервисе представлены в таблице 7.

Таблица 7 – Методы микросервиса пользователей

Метод	Точка доступа	Описание	Параметры
GET	/users	Получение списка всех пользователей	Query: limit (необязательный)
GET	/user/{user_id}	Получение информации о конкретном пользователе	Path: user_id
POST	/register	Регистрация нового пользователя	Body: UserCreate
POST	/login	Авторизация пользователя и получение JWT-токена	Body: LoginRequest

Указанные методы предоставляют необходимые методы для работы с пользователями системы.

3.2.4 Проектирование микросервиса профилей пользователей (user_profile_service)

При проектировании данного микросервиса, так же поддерживался стиль многослойной архитектуры для организации кода. Структура микросервиса представлена на рисунке 11.

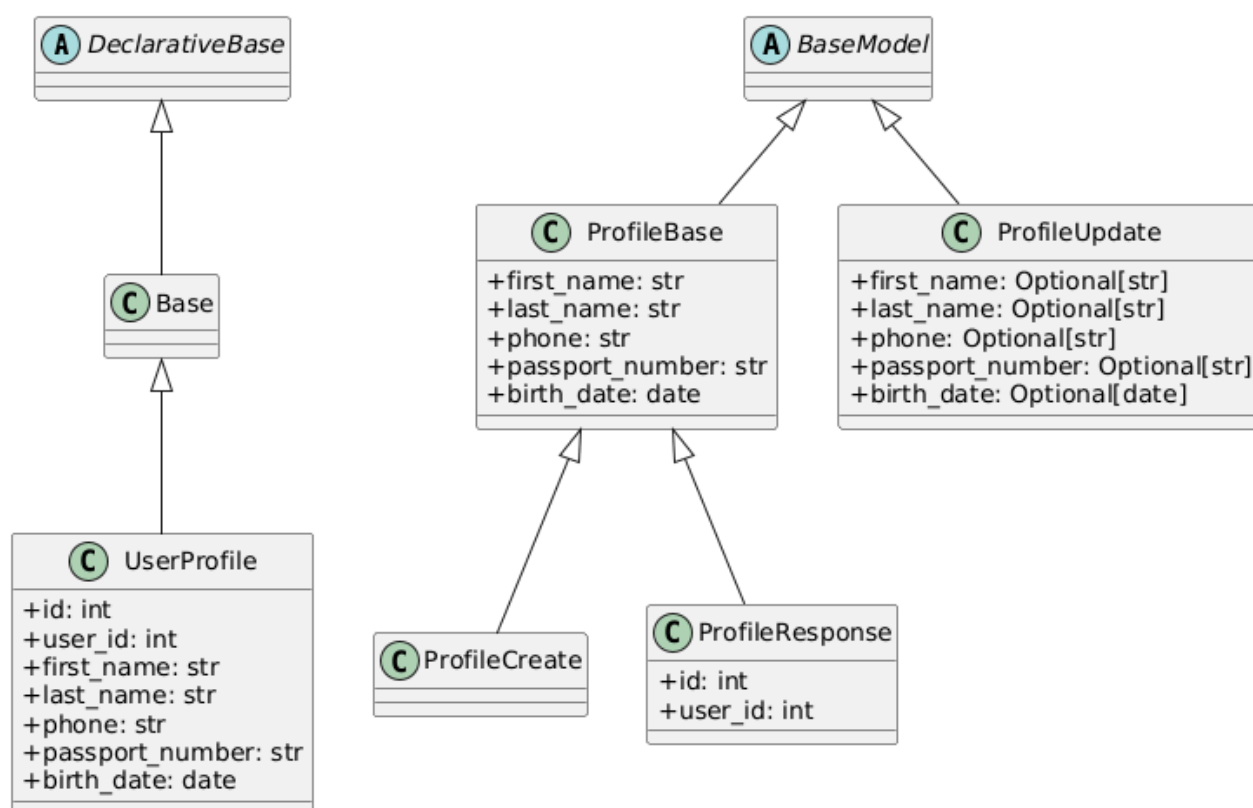


Рисунок 11 – Диаграмма классов микросервиса профилей пользователей (user_profile_service)

Описание основных компонентов микросервиса приведены ниже:

- DeclarativeBase – абстрактный базовый класс SQLAlchemy, обеспечивающий хранение метаданных всех моделей и поддержку ORM-функционала; от него наследуется локальный Base.
- Base – локальный базовый класс микросервиса, от которого наследуются все SQLAlchemy-модели (UserProfile).
- UserProfile – основная SQLAlchemy-модель для хранения информации о профиле пользователя. Содержит идентификатор пользователя из сервиса аутентификации, имя, фамилию, номер телефона, номер паспорта и дату рождения. Все операции CRUD выполняются через данный класс в сочетании с сессией SQLAlchemy.
- BaseModel – абстрактный базовый класс для всех Pydantic моделей, предоставляющий функционал валидации данных и конфигурацию ORM.

Pydantic-модели:

- ProfileBase – базовая модель профиля пользователя, содержащая общие поля и правила валидации;
- ProfileCreate – модель запроса на создание профиля пользователя;
- ProfileUpdate – модель запроса на обновление существующего профиля пользователя;
- ProfileResponse – модель ответа API с полной информацией о профиле пользователя.

Методы микросервиса представлены в таблице 8.

Таблица 8 – Методы микросервиса задач

Метод	Точка доступа	Описание	Параметры
GET	/profile	Получение профиля текущего пользователя.	Заголовок: X-User-Id
POST	/profile	Создание профиля пользователя. Перед созданием выполняется проверка на существование профиля и уникальность номера паспорта.	Body: ProfileCreate; Заголовок: X-User-Id
PUT	/profile	Обновление профиля текущего пользователя. Изменяются только переданные поля профиля.	Body: ProfileUpdate; Заголовок: X-User-Id
DELETE	/profile	Удаление профиля текущего пользователя.	Заголовок: X-User-Id

Данные методы реализуют базовые операции по управлению профилями пользователей, включая создание, получение, изменение и удаление персональных данных.

3.3 Проектирование клиентской части

В качестве клиентской части приложения было решено разрабатывать web-приложение на JavaScript. Для удобства разработки были использованы следующие инструменты:

- библиотека React – для создания продвинутых пользовательских интерфейсов;
- react-router-dom – расширение для управления навигацией внутри приложения – позволяет, в том числе, блокировать доступ к функционалу неавторизованным пользователям;
- tailwind – для удобной стилизации компонентов.

Взаимодействие с серверной частью приложение происходит с помощью REST API через API-Gateway.

Страница создания заказа – страница, отражающая детальную форму заказа для пользователя с возможностью подсчета стоимости. Интерфейс страницы создания заказа приведен на рисунке 10.

Рисунок 10 – Интерфейс главной страницы

Страница управления заказами – страница, на которой пользователь может просматривать свои заказы и отслеживать их, а администратор может

просматривать все заказы, изменять их статусы и управлять ими. Интерфейс страницы приведен на рисунке 11.

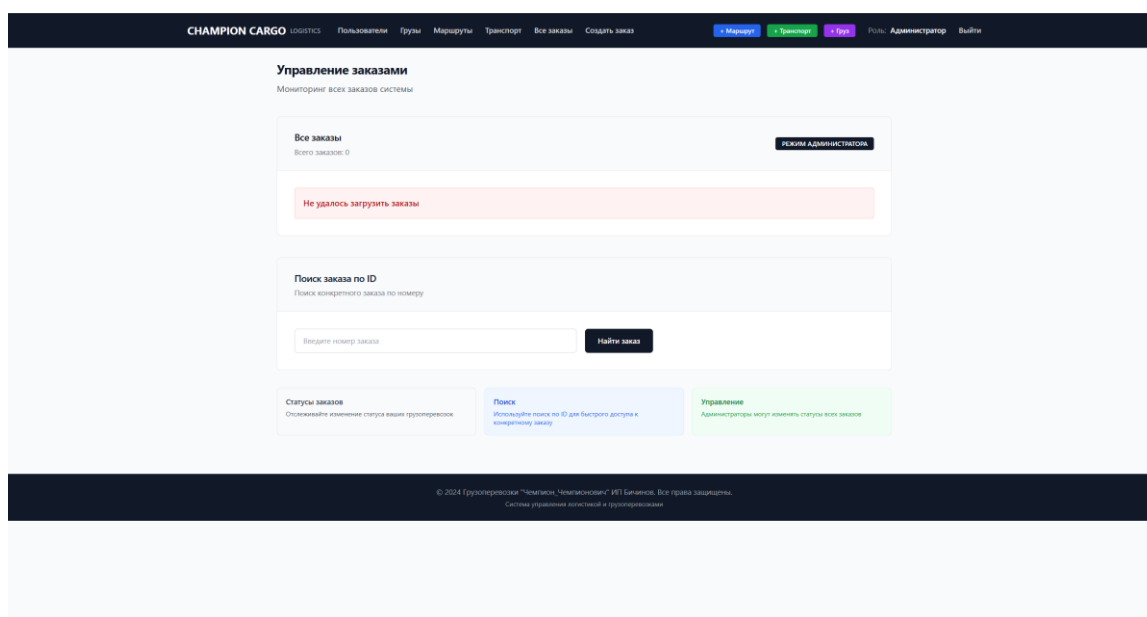


Рисунок 11 – Страница управления заказами

Страница грузов – на данной странице пользователь может просматривать доступные грузы и прочесть информацию о них, а администратору доступны CRUD-операции над грузами. Интерфейс страница представлен на рисунке 12.

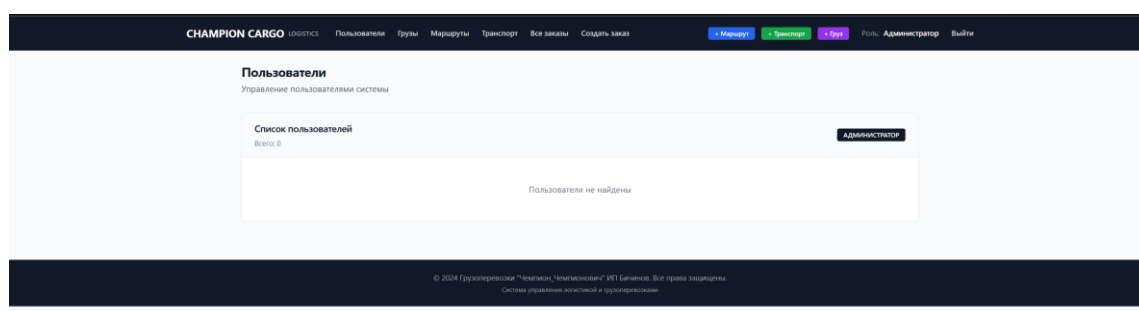


Рисунок 12 – Интерфейс страницы жалоб

Страница авторизации – точка входа в приложение, разграничивает функционал для администратора и пользователя. Интерфейс страницы представлен на рисунке 13.

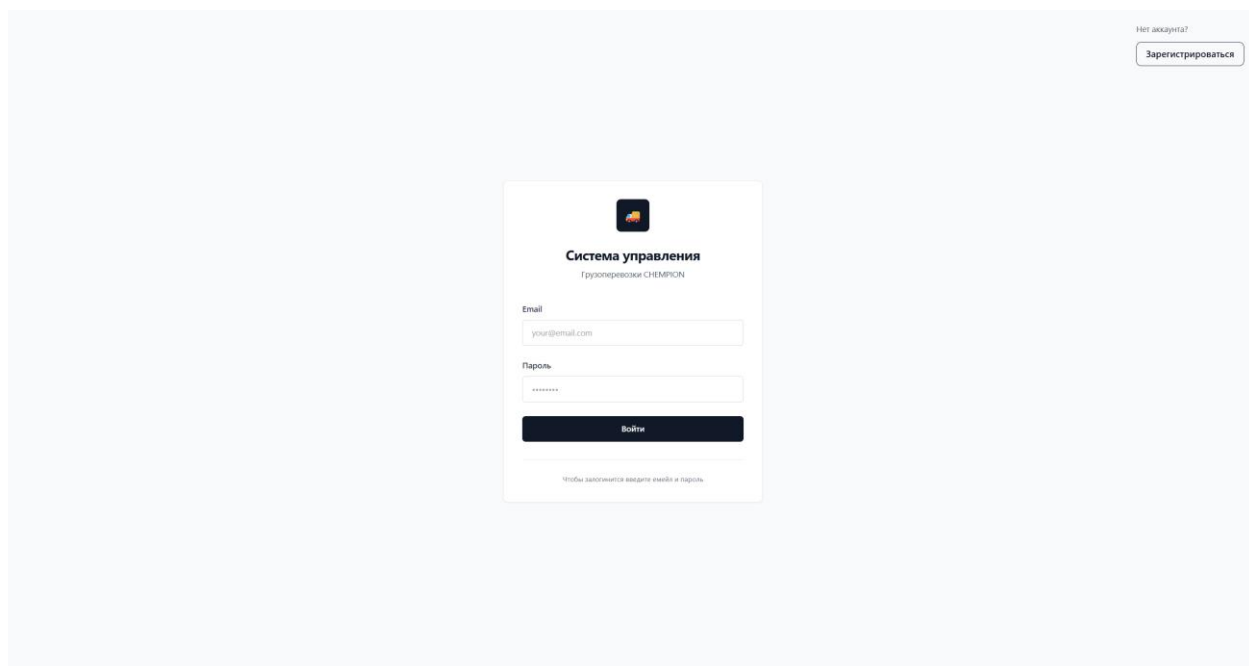


Рисунок 13 – Интерфейс страницы авторизации

В итоге клиентская часть приложения реализована как SPA и покрывает весь основной пользовательский функционал: оформление заказов, их управление, работу с грузами и авторизацию [11].

4 Реализация приложения

4.1 Реализация серверной части приложения

Серверная часть системы построена на основе микросервисной архитектуры, что обеспечивает масштабируемость, отказоустойчивость и возможность независимого развертывания компонентов. В качестве основного языка программирования выбран фреймворк FastAPI на языке Python, что позволяет быстро разрабатывать функционал и легко интегрироваться с современными библиотеками для web-разработки, баз данных и API.

Архитектурно система состоит из нескольких взаимосвязанных микросервисов, каждый из которых отвечает за определенную предметную область. API Gateway служит единой точкой входа для всех клиентов, осуществляя маршрутизацию запросов к соответствующим микросервисам.

Основные микросервисы включают:

- Auth Service – управляет регистрацией, входом в систему, аутентификацией пользователей и выдачей JWT-токенов;
- Cargo Service – отвечает за справочники грузов, маршрутов и транспорта, включая CRUD операции и работу с изображениями;
- Order Service – управляет заказами пользователей, позволяет создавать, редактировать и отслеживать их статус, связывая заказы с пользователями, грузами, маршрутами и транспортом;
- User Profile Service – отвечает за личную информацию пользователей, необходимую для создания заказа.

Все микросервисы взаимодействуют с данными через слой доступа к базе данных на основе SQLAlchemy, что обеспечивает целостность информации и единый подход к работе с PostgreSQL.

Система предоставляет REST API для синхронных операций. API включает все необходимые методы для работы с пользователями, заказами и справочниками грузов/транспорта/маршрутов, с полной документацией по

каждому эндпоинту, включая форматы запросов и ответов, коды ошибок и примеры использования [9].

Структура одного из сервисов `cargo_service` представлена на рисунке 14.

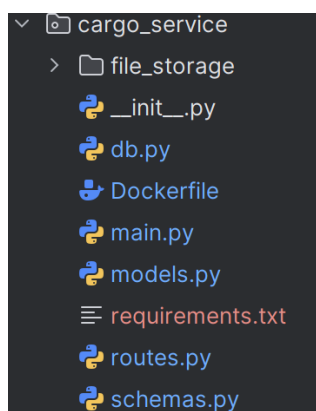


Рисунок 14 – Структура микросервиса `cargo_service`

На данном рисунке можно выделить следующие компоненты:

- `db.py` – файл, отвечающий за соединение с БД;
- `models.py` – файл, в котором реализованы модели сущностей БД с их связями;
- `schemas.py` – файл, в котором реализованы модели в таком виде, в каком их будет валидировать и работать с ними фреймворк FastAPI;
- `routes.py` – файл, в котором содержатся все виды запросов и собираются в один роутер;
- `main.py` – точка входа в сервис;
- `Dockerfile` – файл конфигурации контейнера;
- `requirements.txt` – текстовый файл, в котором собраны используемые микросервисом библиотеки (используется при сборке Docker-контейнера).

На примере микросервиса `cargo_service` будет показана реализация компонентов (листинг 1–5)[9].

Листинг 1 – Реализация подключения к БД Postgres в `db.py`

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, DeclarativeBase

DATABASE_URL =
```

```

"postgresql://postgres:postgres@cargo_db:5432/ivanchuk_cargo_db"

engine = create_engine(DATABASE_URL, echo=True, connect_args={
    "options": "-c client_encoding=UTF8"
})
SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)

class Base(DeclarativeBase):
    pass

```

Листинг 2 – Пример модели из models.py

```

class CargoType(Base):
    __tablename__ = "cargo_type"

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    description = Column(String)
    max_weight = Column(Float)

```

Листинг 3 – Пример pydentic модели из schemas.py

```

class CargoTypeResponse(BaseModel):
    id: int
    name: str
    description: str
    max_weight: float
    class Config:
        orm_mode = True

```

Листинг 4 – Пример эндпоинта (в данном случае get-запроса) из routes.py

```

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@router.get("/cargo", response_model=List[CargoTypeResponse])
def get_all_cargo(limit: int = 100, db: Session =
Depends(get_db)):
    return db.query(CargoType).limit(limit).all()

```

Листинг 5 – Точка входа в микросервис (main.py)

```

from fastapi import FastAPI
from routes import router
from db import Base, engine
app = FastAPI()

app.include_router(router)

```

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", host="0.0.0.0", port=8001,
reload=True)
```

Таким образом, разработанная информационная система транспортной компании реализует принципы микросервисной архитектуры, обеспечивая разделение функциональности, масштабируемость и удобство сопровождения. Применённый стек технологий и архитектурные решения позволяют эффективно автоматизировать основные процессы логистики и создают основу для дальнейшего развития и расширения системы.

4.2 Реализация клиентской части приложения

Клиентская часть реализована как SPA веб-приложение, написанное на JavaScript [12] с использованием библиотек React, Tailwind и react-router-dom[13]. В архитектуре приложения были задействованы некоторые элементы из Feature-Sliced Design [14], что позволило эффективно разделить функционал приложения в соответствующих компонентах.

В общем виде структура приложения представлена рисунке 15.

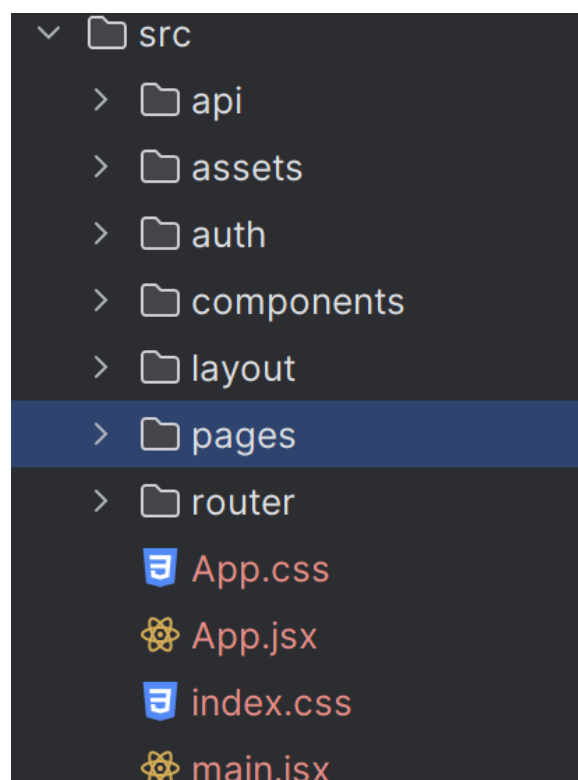


Рисунок 15 – Общая структура приложения

На данном рисунке можно выделить следующие компоненты:

- `assets` – директория с используемыми объектами, в частности изображениями;
- `pages` – директория со всеми страницами приложения;
- `auth` – содержит логику аутентификации и работы с токенами, включая проверку доступа и хранение состояния авторизации;
- `components` – отдельные повторно используемые компоненты интерфейса;
- `layout` – компоненты для построения общей структуры страниц;
- `router` – конфигурация маршрутизации внутри приложения, включая защиту маршрутов для авторизованных пользователей;
- `App.tsx` – центральный компонент приложения, отвечающий за навигацию и подключение основных провайдеров контекста;
- `main.tsx` – стартовая точка системы, где инициализируются контексты приложения и рендерится корневой компонент.

Таким образом, разработанная структура клиентского приложения обеспечивает разделение функциональности на независимые модули, что упрощает разработку, тестирование и дальнейшее сопровождение системы.

4.3 Контейнеризация и развертывание системы

Для упрощения развертывания и обеспечения независимости программных компонентов от особенностей операционной системы была использована технология контейнеризации `Docker`. Каждый микросервис системы, а также клиентское приложение и базы данных разворачиваются в отдельных контейнерах [15], а взаимодействие между контейнерами осуществляется через внутреннюю сеть `Docker`, что обеспечивает изоляцию сервисов, упрощает управление зависимостями и повышает переносимость системы между различными средами выполнения.

В общем виде структура развертывания показана на рисунке 16.

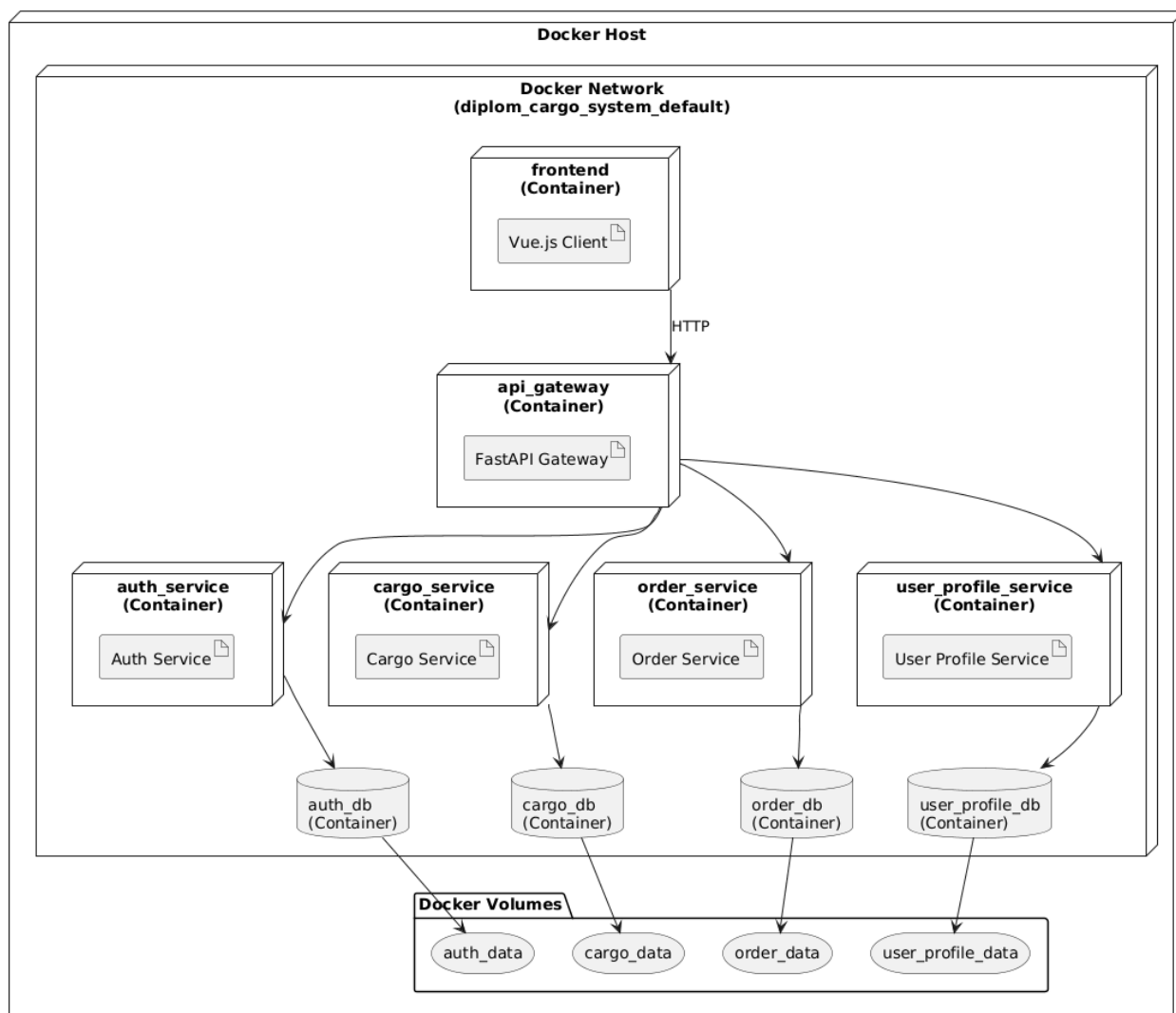


Рисунок 16 – Диаграмма развертывания информационной системы

На представленном рисунке можно выделить следующие компоненты:

- **frontend** – контейнер клиентского веб-приложения, обеспечивающего взаимодействие пользователя с системой через графический интерфейс;
- **api_gateway** – контейнер шлюза API, осуществляющего маршрутизацию запросов между клиентским приложением и микросервисами системы;
- **auth_service** – контейнер микросервиса аутентификации и авторизации пользователей, отвечающий за регистрацию, вход в систему и управление учетными записями пользователей;

- cargo_service – контейнер микросервиса справочника грузов, маршрутов и транспортных средств;
- order_service – контейнер микросервиса управления заказами, обеспечивающий создание, изменение, удаление и получение информации о заказах;
- user_profile_service – контейнер микросервиса управления профилями пользователей, предназначенный для хранения и обработки персональных данных пользователей;
- auth_db – контейнер базы данных микросервиса аутентификации;
- cargo_db – контейнер базы данных микросервиса справочника грузов, маршрутов и транспортных средств;
- order_db – контейнер базы данных микросервиса заказов;
- user_profile_db – контейнер базы данных микросервиса профилей пользователей;
- auth_data – том Docker Volume, предназначенный для постоянного хранения данных базы данных микросервиса аутентификации;
- cargo_data – том Docker Volume для хранения данных базы данных микросервиса справочника грузов;
- order_data – том Docker Volume для хранения данных базы данных заказов;
- user_profile_data – том Docker Volume для хранения данных базы данных профилей пользователей;
- diplom_cargo_system_default – виртуальная сеть Docker, обеспечивающая взаимодействие всех контейнеров системы между собой.

Для автоматизации развертывания всех компонентов системы использовался инструмент Docker Compose. Конфигурация контейнеров, сетевого взаимодействия и зависимостей между сервисами описана в файле docker-compose.yml.

В листинге 6 представлен фрагмент конфигурации микросервиса заказов.

Листинг 6 – Конфигурация контейнера микросервиса заказов

```
order_service:
  build: ./services/order_service
  container_name: order_service
  expose:
    - "8004"
  depends_on:
    - db_seed
  environment:
    DATABASE_URL:
postgres://postgres:postgres@order_db:5432/ivanchuk_order_service_db
```

Помимо основных микросервисов и баз данных, при развертывании системы используется вспомогательный контейнер `db_seed`. Он предназначен для автоматической загрузки подготовленных данных в базы данных из SQL-дампов. После запуска контейнер ожидает готовности всех экземпляров PostgreSQL, выполняет импорт данных и завершает работу. Такой подход позволяет автоматизировать подготовку системы к работе и избежать необходимости ручного заполнения баз данных [8].

После инициализации баз данных запускаются контейнеры микросервисов, API-шлюза и клиентского приложения. Результат успешного развертывания всех компонентов системы с использованием Docker Compose [14] представлен на рисунке 17.

☐		Name	Container ID	Image	Port(s)	CPU (%)	Memory usag...	Actions
☐	▼	diplom_cargo_system	-	-	-	0.6%	495.67MB / 76.2	⚙️ 🔴 ⋮ 🗑️
☐	●	order_db	1a48b81e70f6	postgres:1		0%	25.31MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	●	cargo_db	58c8e21441b9	postgres:1		0%	24.13MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	●	auth_db	ac20e20e9813	postgres:1		0%	40.61MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	●	user_profile_db	f516835b92b4	postgres:1		0%	23.71MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	○	db_seed	f3aca20f0a89	postgres:1		0%	0B / 0B	⚙️ ▶️ ⋮ 🗑️
☐	●	user_profile_service	b25c16cb199a	diplom_car		0.12%	52.48MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	●	cargo_service	9fa47049d6e8	diplom_car		0.11%	53.98MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	●	order_service	07bece4ca2b6	diplom_car		0.09%	52.5MB / 7.63GB	⚙️ 🔴 ⋮ 🗑️
☐	●	auth_service	511b827f7928	diplom_car		0.09%	56.13MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	●	api_gateway	e5db82bf4a18	diplom_car	8000:8000	0.12%	39.12MB / 7.63G	⚙️ 🔴 ⋮ 🗑️
☐	●	frontend	ebc558341114	diplom_car	5173:5173	0.07%	127.7MB / 7.63G	⚙️ 🔴 ⋮ 🗑️

Showing 12 items

Рисунок 17 – Результат развертывания информационной системы

Таким образом, использование технологии Docker и инструмента Docker Compose позволило упростить процесс развертывания программного средства, обеспечить изоляцию всех компонентов системы в отдельных контейнерах и организовать их взаимодействие в рамках единой сети.

5 Тестирование

При тестировании функционала разрабатываемой системы использовались методы как ручного, так и автоматизированного тестирования. Для ручного тестирования был применен метод черного ящика. Для автоматизированного – фреймворк `pytest` для создания `unit`-тестов.

5.1 Тестирование методом чёрного ящика

Тестирование методом черного ящика – это подход к тестированию программного обеспечения, при котором тестировщик оценивает функциональность приложения без знания его внутренней структуры или кода. При применении данного метода основное внимание уделяется входным данным и ожидаемым выходным результатам. Тестировщик создает тест-кейсы на основе требований и спецификаций, проверяя, соответствует ли поведение программы заданным критериям. Этот метод позволяет сосредоточиться на том, как система работает с точки зрения пользователя, и выявлять ошибки, которые могут быть незаметны при анализе внутренней работы программы [16].

5.1.1 Функциональная диаграмма и таблица решений

Для выявления потенциальных проблем при разработке системы были использованы такие инструменты, как функциональная диаграмма и таблица решений.

Функциональная диаграмма – это графическое представление функциональных требований системы. Она помогает визуализировать, как различные компоненты системы взаимодействуют между собой и какие функции выполняют. Диаграмма разбивает систему на функциональные компоненты для улучшения понимания структуры программы, помогая таким образом тестировщикам быстрее идентифицировать области для тестирования.

Таблица решений помогает структурировать сложные условия и варианты действий на основе различных входных данных. Она позволяет визуализировать

все возможные комбинации входных и выходных данных, что упрощает восприятие информации [17].

Эти методы позволяют тестировщику составлять тест-кейсы, охватывающие все возможные сценарии, что способствует более полному покрытию тестами.

При создании нового заказа пользователь должен указать параметры заказа, включая выбранный груз, маршрут, транспорт, а также статус заказа. Далее будут приведены требования к формату данных каждого поля.

Требования к данным поля «Груз»:

- пользователь должен выбрать груз из списка доступных;
- поле должно быть заполнено.

Требования к данным поля «Маршрут»:

- пользователь должен выбрать маршрут из списка доступных;
- поле должно быть заполнено.

Требования к данным поля «Транспорт»:

- пользователь должен выбрать транспорт из списка доступных;
- поле должно быть заполнено.

Требования к данным полей «Вес единицы груза(кг)» и «Количество товара»:

- значение поля должно быть вещественным числом;
- значение поля не может быть меньше нуля;
- значение веса единицы груза не может превышать грузоподъемность

выбранного типа груза и транспорта.

Требования к данным пользователя:

- пользователь должен быть авторизован в системе;
- данные пользователя должны быть корректными.

На основе данных требований была составлена таблица причин и следствий, представленная в таблице 9.

Таблица 9 – Таблица причин и следствий

П1	Тип груза выбран
П2	Маршрут выбран
П3	Тип транспорта выбран
П4	Вес товара введен
П5	Вес товара не превышает грузоподъемность выбранного типа груза
П6	Вес товара не превышает грузоподъемность выбранного транспорта
П7	Количество товара введено
П8	Профиль пользователя заполнен
С1	Ошибка: «Груз не выбран»
С2	Ошибка: «Маршрут не выбран»
С3	Ошибка: «Транспорт не выбран»
С4	Ошибка: «Вес товара некорректен»
С5	Ошибка: «Вес товара превышает грузоподъемность выбранного транспорта»
С6	Ошибка: «Количество товара некорректно»
С7	Ошибка: «Профиль пользователя не заполнен»
С8	Заказ успешно создан

Графическое представление причин и следствий в виде функциональной диаграммы представлено на рисунке 18.

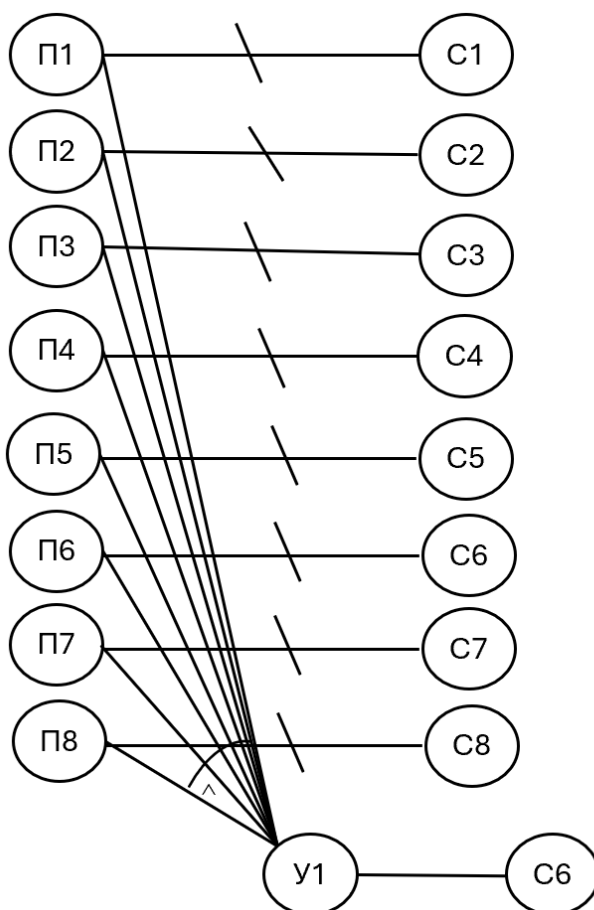


Рисунок 18 – Функциональная диаграмма причин и следствий

Для перебора всех сценариев был выбран инструмент Pairwise. Все варианты тест-кейсов представлены на рисунке 19.

	Тип груза	Тип транспорта	Маршрут	Вес товара	Количество товара	Профиль пользователя
1	Выбран	Не выбран	Не выбран	<=0	<=0	Не заполнен
2	Не выбран	Выбран	Выбран	>=Максимального веса типа транспорта	>0	Заполнен
3	Выбран	Выбран	Не выбран	>=Максимального веса типа груза	<=0	Заполнен
4	Выбран	Выбран	Выбран	<=0	<=0	Заполнен
5	Выбран	Не выбран	Выбран	>=Максимального веса типа груза	>0	Не заполнен
6	Выбран	Выбран	Не выбран	>=Максимального веса типа транспорта	>0	Заполнен
7	Выбран	Не выбран	Не выбран	>0	>0	Заполнен
8	Не выбран	Выбран	Выбран	>0	<=0	Не заполнен
9	Не выбран	Выбран	Не выбран	<=0	>0	Заполнен
10	Выбран	Не выбран	Не выбран	>=Максимального веса типа груза	>0	Не заполнен
11	Не выбран	Выбран	Выбран	>=Максимального веса типа транспорта	<=0	Заполнен
12	Выбран	Выбран	Не выбран	>=Максимального веса типа груза	<=0	Заполнен
13	Выбран	Выбран	Выбран	>=Максимального веса типа транспорта	>0	Не заполнен
14	Выбран	Не выбран	Выбран	>0	<=0	Заполнен
15	Выбран	Выбран	Не выбран	>0	>0	Не заполнен
16	Выбран	Не выбран	Выбран	<=0	>0	Заполнен

Рисунок 19 – Варианты тест-кейсов тестирования

Результаты тестирования представлены в приложении В.

Тестирование системы с использованием приведенных тест-кейсов поможет выявить потенциальные проблемы в работе функционала при создании заказа.

5.1.2 Автоматизированное тестирование

Для проведения автоматизированного тестирования был использована библиотека `pytest`, предоставляющий инструменты для написания и выполнения модульных тестов на языке Python. Было принято решение протестировать реализованную логику микросервисов моего приложения, а именно:

- регистрация пользователя и вход в аккаунт;
- функционал взаимодействия с грузами, маршрутами и транспортом.

В листинге 7 приведен код фикстур, необходимых для проведения тест-кейсов[18].

Листинг 7 – Фикстуры

```
import pytest_asyncio
import httpx

BASE_URL = "http://localhost:8000"

@pytest_asyncio.fixture
async def client():
    async with httpx.AsyncClient(base_url=BASE_URL) as c:
        yield c

@pytest_asyncio.fixture
async def create_user(client):
    async def _create_user(email, password, username):
        r = await client.post("/register", json={
            "email": email,
            "password": password,
            "username": username
        })
        if r.status_code == 400 and "Email уже занят" in r.text:
            return None
        elif r.status_code != 200:
            raise Exception(f"Ошибка создания пользователя {email}: {r.text}")
        return r.json()
    return _create_user

@pytest_asyncio.fixture
async def get_token(client):
```

```

async def _get_token(email, password):
    r = await client.post("/login", json={
        "email": email,
        "password": password
    })
    if r.status_code != 200:
        raise Exception(f"Не удалось получить токен для
{email}: {r.text}")
    return r.json()["access_token"]
return _get_token

@pytest_asyncio.fixture
async def user_token(create_user, get_token):
    email = "user@test.com"
    password = "123456"
    await create_user(email, password, "user")
    token = await get_token(email, password)
    return token

@pytest_asyncio.fixture
async def admin_token(get_token):
    email = "admin@example.com"
    password = "admin"
    token = await get_token(email, password)
    return token

```

Также в таблице 10 представлено функциональное описание автоматизируемых тестов.

Таблица 10 – Функциональное описание автоматизируемых тестов

№	Модуль	Тестируемая функция	Входные данные	Ожидаемый результат
1	Аутентификация	Регистрация пользователя	email, password, username	200 ОК или 400 (если email занят)
2	Аутентификация	Авторизация пользователя	email, password	200 ОК, возвращается access_token
3	Тип груза (Cargo)	Создание типа груза (admin)	name, description, max_weight	200 ОК, возвращается id
4	Тип груза	Получение типа груза	cargo_id	200 ОК

Продолжение таблицы 10

№	Модуль	Тестируемая функция	Входные данные	Ожидаемый результат
5	Тип груза	Обновление типа груза	cargo_id, новые данные	200 OK, данные обновлены
6	Тип груза	Удаление типа груза	cargo_id	200 OK
7	Тип груза	Создание типа груза (user)	данные груза	403 Forbidden
8	Заказы (Order)	Создание заказа	cargo_id, route_id, transport_id, weight, quantity	200 OK, возвращается id
9	Заказы	Получение заказа	order_id	200 OK
10	Заказы	Обновление статуса заказа	status = DELIVERED	200 OK, статус обновлён
11	Заказы	Удаление заказа	order_id	200 OK
12	Маршруты (Route)	Создание маршрута	source, destination, distance_km	200 OK
13	Маршруты	Получение маршрута	route_id	200 OK
14	Маршруты	Обновление маршрута	distance_km	200 OK
15	Маршруты	Удаление маршрута	route_id	200 OK
16	Тип транспорта	Создание типа транспорта	name, max_weight, price_per_km	200 OK
17	Тип транспорта	Получение транспорта	transport_id	200 OK
18	Тип транспорта	Обновление транспорта	price_per_km	200 OK
19	Тип транспорта	Удаление транспорта	transport_id	200 OK
20	Пользователи	Создание пользователя	email, password, username	200 OK или 400
21	Пользователи	Авторизация пользователя	email, password	200 OK, токен

Окончание таблицы 10

№	Модуль	Тестируемая функция	Входные данные	Ожидаемый результат
22	Пользователи	Получение списка пользователей (admin)	—	200 OK
23	Профиль	Создание профиля	first_name, last_name, phone и др.	200/201 OK
24	Профиль	Получение профиля	—	200 OK
25	Профиль	Обновление профиля	phone	200 OK
26	Профиль	Удаление профиля	—	200 OK
27	Профиль	Получение удалённого профиля	—	404 Not Found

На рисунке 20 приведены результаты запуска автоматизированного тестирования.

```
(.venv) PS C:\Users\Konstantin\PycharmProjects\FastAPIProject\ivanchuk_cargo\tests> pytest -v
===== test session starts =====
platform win32 -- Python 3.12.10, pytest-9.0.2, pluggy-1.6.0 -- C:\Users\Konstantin\PycharmProjects\FastAPIProject\.venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Konstantin\PycharmProjects\FastAPIProject\ivanchuk_cargo\tests
plugins: anyio-4.12.0, asyncio-1.3.0
asyncio: mode=Mode.STRICT, debug=False, asyncio_default_fixture_loop_scope=None, asyncio_default_test_loop_scope=function
collected 7 items

test_auth.py::test_register_and_login PASSED [ 14%]
test_cargo.py::test_cargo_crud PASSED [ 28%]
test_orders.py::test_order_crud PASSED [ 42%]
test_routes.py::test_route_crud PASSED [ 57%]
test_transport.py::test_transport_crud PASSED [ 71%]
test_user_profile.py::test_user_profile_crud PASSED [ 85%]
test_users.py::test_user_crud PASSED [100%]

===== 7 passed in 5.11s =====
(.venv) PS C:\Users\Konstantin\PycharmProjects\FastAPIProject\ivanchuk_cargo\tests>
```

Рисунок 20 – Результаты автоматизированного тестирования

Успешно выполненное автоматизированное тестирование свидетельствует о том, что рассматриваемый блок кода работает корректно.

5.1.3 Анализ сложности кода модуля регистрации

Для измерения сложности кода были использованы метрики Холстеда и метрики размера программы.

В листинге 8 приведён метод обновления информации о типе груза через API, который обрабатывает как multipart/form-data (с изображением), так и JSON-запросы.

Метрики размера программы представляют собой подсчет количества строк программного кода, а также числа строк с комментариями.

Листинг 8 – Метод валидации данных нового сотрудника и метод для добавления новой записи в базу данных

```
@router.put("/cargo/{cargo_id}")
async def update_cargo(
    cargo_id: int,
    request: Request,
    user=Depends(require_admin),
    image: Optional[UploadFile] = None
):
    headers = build_auth_headers(request, user)

    # Проверяем, multipart или json
    content_type = request.headers.get("content-type", "")
    if content_type.startswith("multipart/form-data"):
        # Приходит форма (картинка + поля)
        form_data = await request.form()
        data = {k: form_data[k] for k in form_data if k !=
"image"}
        files = None
        if image:
            files = {"image": (image.filename, await image.read(),
image.content_type)}
        else:
            # Приходит JSON (только текстовые поля)
            data = await request.json()
            files = None

    async with httpx.AsyncClient() as client:
        if files:
            response = await client.put(
                f"{CARGO_SERVICE_URL}/cargo/{cargo_id}",
                data=data,
                files=files,
                headers=headers
            )
        else:
            response = await client.put(
                f"{CARGO_SERVICE_URL}/cargo/{cargo_id}",
                json=data,
                headers=headers
            )
```

```

if response.status_code != 200:
    raise HTTPException(status_code=response.status_code,
detail=response.text)
return response.json()

```

Физическая строка – это каждая строка кода, включая пустые строки и строки с комментариями; логические строки – это строки, которые содержат исполняемый код или выражения, игнорируя пустые и строки с комментариями. Далее будет приведено вычисление метрик размера программы для приведенного выше программного кода.

В данном фрагменте кода имеется:

- 34 физических строки;
- 25 логических строк;
- 3 строк-комментариев.

Метрики Холстеда представляют собой набор количественных показателей, разработанных Норманом Холстедом в 1977 году для оценки сложности программного кода. Эти метрики помогают разработчикам и понять качество и сложность программного обеспечения. Для вычисления метрик Холстеда необходимо измерить четыре характеристики программы:

- число уникальных операторов;
- число уникальных операндов;
- общее число операторов;
- общее число операндов [19].

В таблице 11 приведен словарь операторов для рассматриваемого фрагмента кода.

Таблица 11 – Словарь операторов

Оператор	Вхождения
=	9
.	15
()	16

Продолжение таблицы 11

Оператор	Вхождения
{ }	2
:	6
,	10
!=	2
if	3
else	1
for	1
in	1
async	2
await	4
with	1
as	1
return	1
raise	1

В таблице 11 приведен словарь операторов и словарь операндов для рассматриваемого фрагмента кода.

Таблица 12 – Словарь операндов

Операнд	Вхождения
cargo_id	4
request	4
user	2
image	3
headers	3
content_type	2
form_data	3

Продолжение таблицы 12

Операнд	Вхождения
Data	5
files	5
Response	4
client	2
CARGO_SERVICE_URL	2
«content-type»	1
«multipart/form-data»	1
«image»	2
None	2
200	1

Таким образом, число уникальных операторов программы $n_1 = 17$, число уникальных операндов программы $n_2 = 17$, общее число операторов в программе $N_1 = 76$, общее число операндов в программе $N_2 = 46$.

На основе метрик Холстеда можно вычислить ряд характеристик программы. Далее будут рассмотрены некоторые из них[19].

Расчёт словаря программы производится по формуле:

$$n = n_1 + n_2 = 17 + 17 = 34,$$

где n_1 – количество операторов в программе;

n_2 – количество операндов программы.

Расчёт длины программы производится по формуле:

$$N = N_1 + N_2 = 76 + 46 = 122,$$

где N_1 – число вхождений всех операторов в программе;

N_2 – число вхождений всех операндов программы.

Расчёт объёма программы производится по формуле:

$$V = N \cdot \log_2 n = 122 \cdot \log_2 34 = 620,98,$$

где N – длина программы;

n – словарь программы.

Расчёт сложности программы производится по формуле:

$$D = \left(\frac{n_1}{2}\right) \cdot \left(\frac{N_2}{n_2}\right) = \left(\frac{17}{2}\right) \cdot \left(\frac{46}{17}\right) = 23,04 ,$$

где n_1 – количество операторов в программе;

n_2 – количество операндов программе;

N_2 – число вхождений всех операндов программе.

Расчёт оценки интеллектуальных усилий производится по формуле:

$$E = D \cdot V = 23,04 \cdot 620,98 = 14306,6 ,$$

где D – сложность программы;

V – объем программы.

По результатам расчета метрик Холстеда получены потенциальные значения длины программы – 122, объема – 620,98, сложности 23,04 и оценки интеллектуальных усилий – 14306,6.

6 Планирование процесса разработки

Для эффективной разработки информационной системы транспортной компании необходимо выполнить предварительное планирование процесса разработки. Данный этап является важной частью реализации проекта, поскольку позволяет точно определить состав задач, распределить этапы разработки и снизить возможные риски при создании системы.

Для планирования процесса разработки использовался Microsoft Project. Были определены все ресурсы проекта, лист ресурсов представлен на рисунке 21.

	Название ресурса	Тип	Единицы измерения материалов	Краткое название	Группа	Макс. единиц	Стандартная ставка	Ставка сверхурочны	Затраты на использ.	Начисление	Базовый календарь
1	Backend-разработчик	Трудовой		B		100%	450,00 Р/ч	0,00 Р/ч	0,00 Р	Пропорциональн	Стандартный
2	Frontend-разработчик	Трудовой		F		100%	400,00 Р/ч	0,00 Р/ч	0,00 Р	Пропорциональн	Стандартный
3	Системный аналитик	Трудовой		C		100%	300,00 Р/ч	0,00 Р/ч	0,00 Р	Пропорциональн	Стандартный
4	QA-тестировщик	Трудовой		Q		100%	350,00 Р/ч	0,00 Р/ч	0,00 Р	Пропорциональн	Стандартный

Рисунок 21 – Лист ресурсов проекта

Был составлен список всех задач проекта, для которых были выбраны ресурсы, необходимые для выполнения задач, а также была определена длительность каждой из задач. Задачи проекта демонстрируются на рисунках 22 – 23.

Режим задачи	Название задачи	Длительность	Названия ресурсов	Начало	Окончание	Предшественник
1	Начало проекта	0 дней		Ср 08.04.26	Ср 08.04.26	
2	Анализ предметной области и постановка задачи	8 дней		Ср 08.04.26	Пт 17.04.26	1
3	Анализ аналогов	3 дней	Системный аналитик	Ср 08.04.26	Пт 10.04.26	
4	Анализ предметной области	2 дней	Системный аналитик	Пн 13.04.26	Вт 14.04.26	3
5	Анализ функциональных требований	2 дней	Системный аналитик	Ср 15.04.26	Чт 16.04.26	4
6	Анализ нефункциональных требований	1 день	Системный аналитик	Пт 17.04.26	Пт 17.04.26	5
7	Анализ предметной области и постановка задачи завершены	0 дней		Пт 17.04.26	Пт 17.04.26	6
8	Проектирование системы	17 дней		Пн 20.04.26	Вт 12.05.26	2
9	Выбор технологий	2 дней	Системный аналитик; Backend-раз	Пн 20.04.26	Вт 21.04.26	
10	Проектирование структуры приложения	4 дней	Системный аналитик	Ср 22.04.26	Пн 27.04.26	9
11	Проектирование баз данных	3 дней	Backend-разработчик	Вт 28.04.26	Чт 30.04.26	10
12	Проектирование серверной части приложения	4 дней	Backend-разработчик	Пт 01.05.26	Ср 06.05.26	11
13	Проектирование клиентской части приложения	4 дней	Frontend-разработчик	Чт 07.05.26	Вт 12.05.26	12
14	Проектирование программных средств завершено	0 дней		Вт 12.05.26	Вт 12.05.26	13
15	Подготовка среды разработки и развертывания	7 дней		Ср 13.05.26	Чт 21.05.26	8
16	Подготовка окружения разработки	1 день	Backend-разработчик	Ср 13.05.26	Ср 13.05.26	
17	Настройка backend проекта	2 дней	Backend-разработчик	Чт 14.05.26	Пт 15.05.26	16
18	Настройка frontend проекта	1 день	Frontend-разработчик	Пн 18.05.26	Пн 18.05.26	17
19	Настройка взаимодействия frontend и backend	1 день	Backend-разработчик; Frontend-раз	Вт 19.05.26	Вт 19.05.26	18
20	Выбор метода развертывания	1 день	Backend-разработчик	Ср 20.05.26	Ср 20.05.26	19
21	Реализация механизма развертывания	1 день	Backend-разработчик	Чт 21.05.26	Чт 21.05.26	20
22	Подготовка механизма развертывания приложения завершена	0 дней		Чт 21.05.26	Чт 21.05.26	21
23	Реализация серверной части приложения	28 дней		Пт 22.05.26	Вт 30.06.26	15
24	Реализация баз данных и моделей	4 дней	Backend-разработчик	Пт 22.05.26	Ср 27.05.26	
25	Реализация авторизации и аутентификации пользователей	4 дней	Backend-разработчик	Чт 28.05.26	Вт 02.06.26	24
26	Реализация ролей администратора и пользователя	2 дней	Backend-разработчик	Ср 03.06.26	Чт 04.06.26	25
27	Реализация управления глущами	4 дней	Backend-разработчик	Пт 05.06.26	Сб 10.06.26	26

Рисунок 22 – Задачи проекта (1 часть)

	Режим задачи	Название задачи	Длительность	Названия ресурсов	Начало	Окончание	Предшественник	A
26		Реализация ролей администратора и пользователя	2 дней	Backend-разработчик	Ср 03.06.26	Чт 04.06.26	25	
27		Реализация управления грузами	4 дней	Backend-разработчик	Пт 05.06.26	Ср 10.06.26	26	
28		Реализация управления маршрутами	3 дней	Backend-разработчик	Чт 11.06.26	Пн 15.06.26	27	
29		Реализация управления транспортом	3 дней	Backend-разработчик	Вт 16.06.26	Чт 18.06.26	28	
30		Реализация создания и обработки заказов	5 дней	Backend-разработчик	Пт 19.06.26	Чт 25.06.26	29	
31		Реализация API взаимодействия с клиентской частью	3 дней	Backend-разработчик	Пт 26.06.26	Вт 30.06.26	30	
32		Реализация серверной части приложения завершена	0 дней		Вт 30.06.26	Вт 30.06.26	31	
33		Реализация клиентской части приложения	22 дней		Ср 01.07.26	Чт 30.07.26	23	
34		Реализация интерфейса авторизации и регистрации	3 дней	Frontend-разработчик	Ср 01.07.26	Пт 03.07.26		
35		Реализация интерфейса просмотра грузов, маршрутов и транспорта	4 дней	Frontend-разработчик	Пн 06.07.26	Чт 09.07.26	34	
36		Реализация интерфейса создания заказов	3 дней	Frontend-разработчик	Пт 10.07.26	Вт 14.07.26	35	
37		Реализация личного кабинета пользователя	4 дней	Frontend-разработчик	Ср 15.07.26	Пн 20.07.26	36	
38		Реализация административной панели	5 дней	Frontend-разработчик	Вт 21.07.26	Пн 27.07.26	37	
39		Интеграция с backend API	3 дней	Frontend-разработчик; Backend-разработчик	Вт 28.07.26	Чт 30.07.26	38	
40		Реализация клиентской части завершена	0 дней		Чт 30.07.26	Чт 30.07.26	39	
41		Тестирование программных средств	10 дней		Пт 31.07.26	Чт 13.08.26	33	
42		Функциональное тестирование API	2 дней	QA-тестировщик	Пт 31.07.26	Пн 03.08.26		
43		Тестирование авторизации и ролей	2 дней	QA-тестировщик	Вт 04.08.26	Ср 05.08.26	42	
44		Тестирование CRUD операций	2 дней	QA-тестировщик	Чт 06.08.26	Пт 07.08.26	43	
45		Тестирование пользовательского интерфейса	2 дней	QA-тестировщик	Пн 10.08.26	Вт 11.08.26	44	
46		Исправление выявленных ошибок	2 дней	Backend-разработчик; Frontend-разработчик	Ср 12.08.26	Чт 13.08.26	45	
47		Тестирование программных средств завершено	0 дней		Чт 13.08.26	Чт 13.08.26	46	
48		Завершение проекта	6 дней		Пт 14.08.26	Пт 21.08.26	41	
49		Составление документации	5 дней	Системный аналитик	Пт 14.08.26	Чт 20.08.26		
50		Сдача проекта	1 день	Системный аналитик	Пт 21.08.26	Пт 21.08.26	49	
51		Работа над проектом завершена	0 дней		Пт 21.08.26	Пт 21.08.26	50	
52		Конец проекта	0 дней		Пт 21.08.26	Пт 21.08.26	48	

Рисунок 23 – Задачи проекта (2 часть)

Для более наглядного отображения длительности выполнения проекта, а также связей между задачами была составлена диаграмма Ганта [20], которая демонстрируется на рисунке 24 – 25.

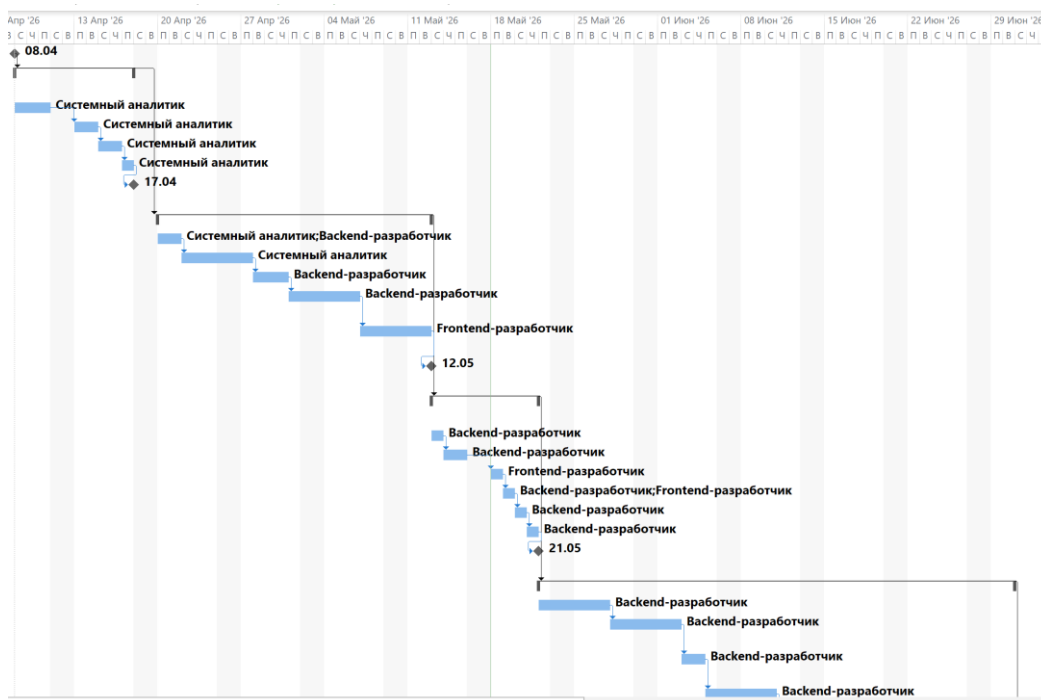


Рисунок 24 – Диаграмма Ганта (1 часть)

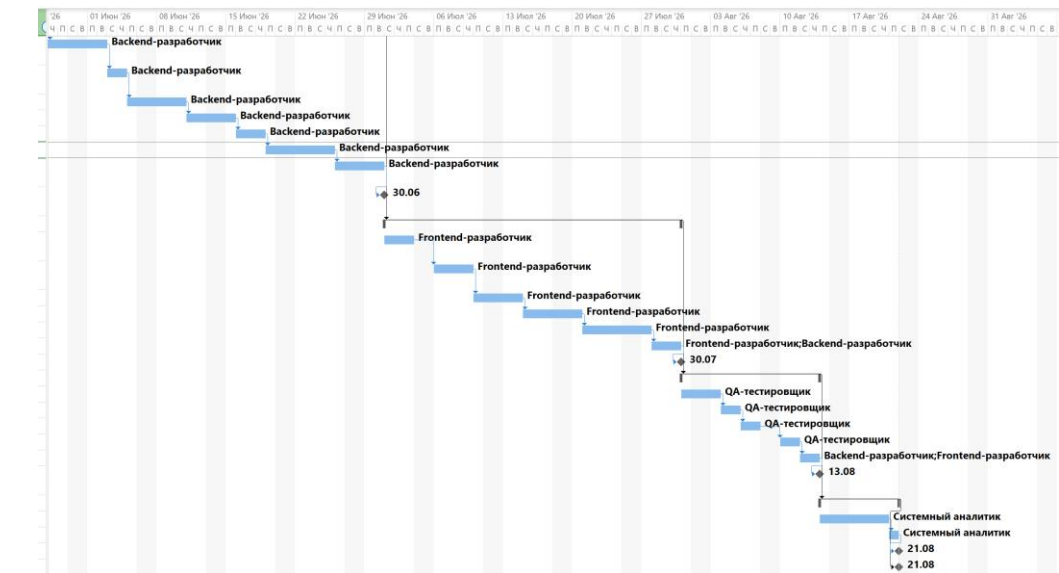


Рисунок 25 – Диаграмма Ганта (2 часть)

Таким образом, общая длительность проекта составляет 98 дней.

Используя встроенный отчет «Обзор затрат» из Microsoft Project был посчитан общий объем затрат для выполнения проекта. График затрат приведен на рисунке 26.

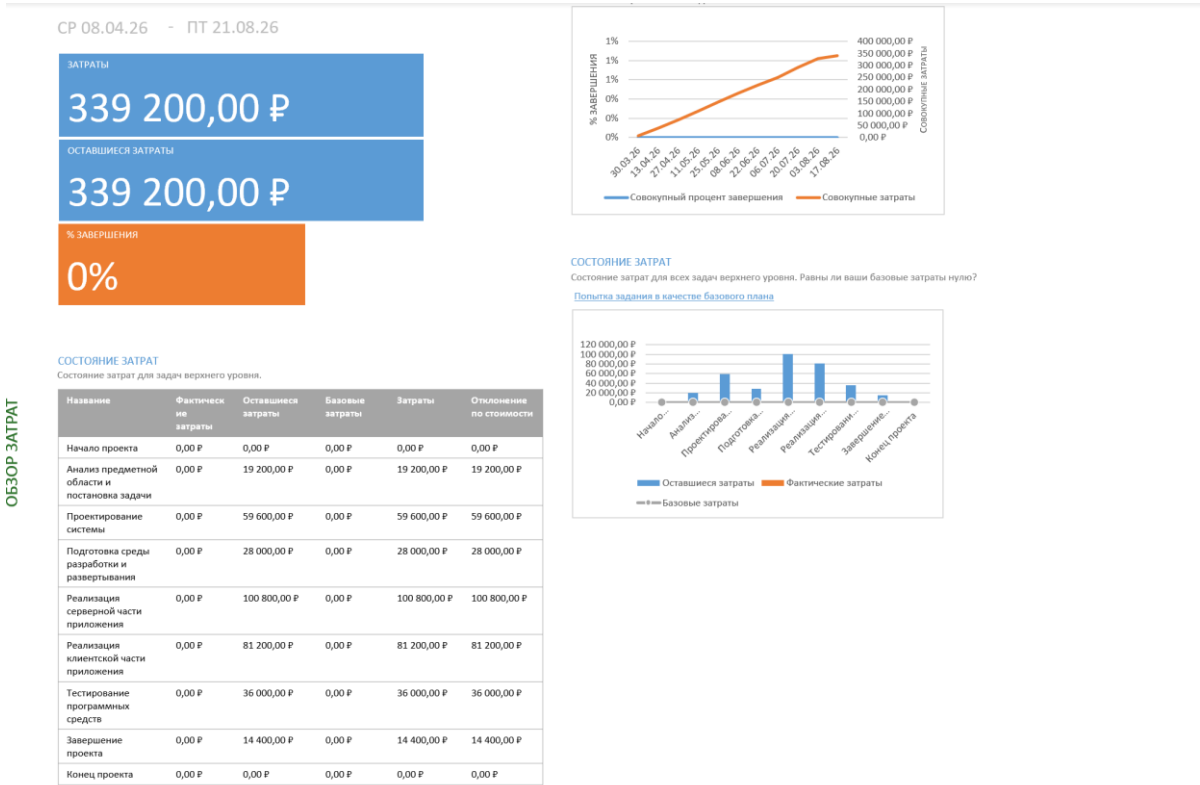


Рисунок 26 – Общие затраты на выполнение проекта

Итоговая сумма на выполнение проекта составляет 339 200.00 рублей.

По итогам проведенного планирования можно сделать вывод, что разработка программных средств распределения обращений клиентов между сотрудниками клиентской поддержки является экономически целесообразной.

Заключение

В рамках настоящей дипломной работы была разработана полнофункциональная информационная система корпоративного сайта компании грузоперевозок, включающая серверную и клиентскую части с расширенными возможностями управления грузами, заказами и транспортными средствами. Поставленная цель, заключающаяся в создании надежной, масштабируемой и удобной системы для автоматизации логистических процессов, была достигнута в полном объеме.

В ходе выполнения работы была спроектирована и реализована микросервисная архитектура, основанная на использовании языка программирования Python. Данный подход обеспечил логическое разделение системы на независимые компоненты, отвечающие за отдельные предметные области, включая управление грузами, обработку заказов, аутентификацию пользователей и маршрутизацию запросов через API Gateway. Использование REST-интерфейсов для взаимодействия между сервисами позволило обеспечить гибкость архитектуры, повысить степень отказоустойчивости системы, а также упростить её сопровождение и дальнейшее развитие.

Отдельное внимание в работе было уделено проектированию структуры данных и организации хранения информации. В качестве системы управления базами данных была выбрана PostgreSQL, для которой была разработана нормализованная схема, обеспечивающая целостность данных и поддержку сложных взаимосвязей между сущностями предметной области. Это позволило реализовать эффективную обработку операций, связанных с управлением логистическими процессами.

Клиентская часть системы была реализована с использованием React и Tailwind CSS, что обеспечило создание современного, адаптивного и интуитивно понятного пользовательского интерфейса. Интерфейс поддерживает основные сценарии работы пользователей, связанные с управлением грузами, заказами и транспортом, обеспечивая удобство взаимодействия с системой.

Результаты проведенного тестирования подтвердили корректность функционирования всех ключевых модулей системы, включая операции создания, редактирования и удаления сущностей, а также обработку заказов и управление маршрутами. Полученные результаты свидетельствуют о том, что разработанная система является работоспособным и устойчивым решением для автоматизации процессов грузоперевозок и может быть использована как основа для дальнейшего расширения функциональности и интеграции с внешними информационными сервисами.

Также в рамках дипломной работы также было выполнено планирование бюджета разработки информационной системы. Были определены основные статьи затрат, связанные с проектированием, разработкой, тестированием и внедрением программного продукта, а также произведена оценка трудозатрат на выполнение работ. Проведённые расчёты показали экономическую обоснованность разработки и позволили оценить необходимые ресурсы для реализации проекта.

Таким образом, поставленная цель дипломной работы была достигнута, а все поставленные задачи успешно выполнены. Разработанная информационная система обеспечивает автоматизацию основных процессов транспортной компании и может быть использована в качестве основы для дальнейшего развития и внедрения дополнительных функциональных возможностей.

Список использованных источников

1. ATI.SU – биржа грузоперевозок и логистическая платформа // ATI.SU – 2026 [Электронный ресурс]. URL: <https://ati.su> (дата обращения: 17.05.2026)
2. Курьерская служба доставки–Яндекс Доставка // Яндекс – 2026 [Электронный ресурс]. URL: <https://dostavka.yandex.ru> (дата обращения: 17.05.2026)
3. TRAFТ – интегратор логистического ПО // TRAFТ – 2026 [Электронный ресурс]. URL: <https://traft.ru/online> (дата обращения: 17.05.2026)
4. Джим Арлоу. UML2 и Унифицированный процесс. Практический объектно ориентированный анализ и проектирование/Джим Арлоу, Айла Нейштадт. – Санкт Петербург, Издательство Символ-Плюс, 2007. – 624с.
5. Как правильно формулировать нефункциональные требования // Habr [Электронный ресурс]. URL: <https://habr.com/ru/articles/948506/> (дата обращения: 22.04.2026).
6. Ричардс, М. Фундаментальный подход к программной архитектуре: паттерны, свойства, проверенные методы / М. Ричардс, Н. Форд. – СПб. : Питер, 2023. – 448 с. – (Серия «Для профессионалов»). – ISBN 978-5-4461-1842-7.
7. Микросервисы. Паттерны разработки и рефакторинга. – СПб. : Питер, 2019. – 544 с. : ил. – (Серия «Библиотека программиста»). – ISBN 978-5-4461-0996-8.
8. Микросервисная архитектура: как работает и каким компаниям нужна // Сбер Про URL: <https://sber.pro/publication/avtonomiya-i-universalnost-kak-rabotaet-mikroservisnaya-arhitektura/?ysclid=mj62dgbm6a340818575> (дата обращения: 14.12.2025).
9. FastAPI. Руководство по работе с фреймворком FastAPI [Электронный ресурс]. URL: <https://fastapi.tiangolo.com/ru/tutorial/> (дата обращения: 04.05.2026).

10. JSON Web Token (JWT) // Yandex Cloud URL: https://yandex.cloud/ru/docs/glossary/jwt?ysclid=mj6w4hn9m0403214032&utm_referrer=https%3A%2F%2Fyandex.ru%2F (дата обращения: 13.12.2025).

11. Single Page Application (SPA) [Электронный ресурс]. – URL: <https://blog.skillfactory.ru/glossary/single-page-application/> (дата обращения: 11.05.2026).

12. Крокфорд, Д. Как устроен JavaScript / Д. Крокфорд ; пер. с англ. Н. Вильчинского. – СПб. : Питер, 2019. – 304 с. – (Серия «Для профессионалов»).

13. Tailwind CSS. Руководство по установке с использованием фреймворков [Электронный ресурс]. URL: <https://tailwindcss.ru/docs/installation/framework-guides/> (дата обращения: 13.05.2026).

14. Обзор // FSD URL: <https://feature-sliced.github.io/documentation/ru/docs/get-started/overview> (дата обращения: 14.12.2025).

15. Полное практическое руководство по Docker: с нуля до кластера на AWS // Habr [Электронный ресурс]. URL: <https://habr.com/ru/post/310460/> (дата обращения: 05.05.2026).

16. Бейзер, Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем / Б. Бейзер. – СПб. : Питер, 2004. – 320 с.

17. Decision Table – что это и как применять // Habr [Электронный ресурс]. URL: <https://habr.com/ru/articles/546432/> (дата обращения: 22.04.2026).

18. Pytest: документация [Электронный ресурс]. URL: <https://pytest-docs.ru.readthedocs.io/ru/latest/contents.html> (дата обращения: 10.05.2026).

19. Что делает код трудным для чтения? Визуальные паттерны сложности // Habr [Электронный ресурс]. URL: <https://habr.com/ru/articles/893820/> (дата обращения: 11.05.2026).

20. Что такое Диаграмма Ганта – // Яндекс Практикум – 2026 [Электронный ресурс]. URL: <https://practicum.yandex.ru/blog/diagramma-ganta/>

(дата обращения: 07.05.2026)

Приложение А
(обязательное)
Листинг приложения

Листинг А.1 – Настройка подключения к базе данных PostgreSQL и создание базового класса SQLAlchemy

```

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, DeclarativeBase

DATABASE_URL =
"postgresql://postgres:postgres@cargo_db:5432/ivanchuk_cargo_db"

engine = create_engine(DATABASE_URL, echo=True, connect_args={
    "options": "-c client_encoding=UTF8"
})
SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)

class Base(DeclarativeBase):
    pass

```

Листинг А.2 – ORM-модели предметной области: типы груза, маршруты и транспорт с ограничениями целостности

```

from sqlalchemy import Column, Integer, String, Float,
CheckConstraint
from db import Base

class CargoType(Base):
    __tablename__ = "cargo_type"

    id = Column(Integer, primary_key=True)

    name = Column(String(100), nullable=False)          #
стройматериалы
    description = Column(String(255), nullable=False)
    max_weight = Column(Float, nullable=False)          # кг

    __table_args__ = (
        CheckConstraint("max_weight > 0",
name="check_cargo_max_weight_positive"),
    )

class Route(Base):
    __tablename__ = "route"

    id = Column(Integer, primary_key=True)

    source = Column(String(100), nullable=False)        # Москва
    destination = Column(String(100), nullable=False)

```

```

distance_km = Column(Float, nullable=False)

__table_args__ = (
    CheckConstraint("distance_km > 0",
name="check_distance_positive"),
)

class Transport(Base):
    __tablename__ = "transport"

    id = Column(Integer, primary_key=True)

    name = Column(String(100), nullable=False)          # Газель,
    # Фурга
    max_weight = Column(Float, nullable=False)
    price_per_km = Column(Float, nullable=False)

    __table_args__ = (
        CheckConstraint("max_weight > 0",
name="check_transport_weight_positive"),
        CheckConstraint("price_per_km > 0",
name="check_price_positive"),
    )

```

Листинг А.3 – Pydantic-схемы валидации и сериализации данных API (Cargo, Route, Transport)

```

from pydantic import BaseModel, Field, field_validator
from typing import Optional

# ----- Cargos -----

class CargoTypeResponse(BaseModel):
    id: int
    name: str
    description: str
    max_weight: float

    class Config:
        from_attributes = True

class CargoTypeCreate(BaseModel):
    name: str = Field(..., min_length=2, max_length=100)
    description: str = Field(..., min_length=2, max_length=255)
    max_weight: float = Field(..., gt=0, le=100000) # до 100 тонн

    @field_validator("name", "description")
    @classmethod
    def strip_strings(cls, v):

```

```

        return v.strip()

class CargoTypeUpdate(BaseModel):
    name: Optional[str] = Field(None, min_length=2,
max_length=100)
    description: Optional[str] = Field(None, min_length=2,
max_length=255)
    max_weight: Optional[float] = Field(None, gt=0, le=100000)

# ----- Route -----

class RouteResponse(BaseModel):
    id: int
    source: str
    destination: str
    distance_km: float

    class Config:
        from_attributes = True

class RouteCreate(BaseModel):
    source: str = Field(..., min_length=2, max_length=100)
    destination: str = Field(..., min_length=2, max_length=100)
    distance_km: float = Field(..., gt=0, le=50000)

class RouteUpdate(BaseModel):
    source: Optional[str] = Field(None, min_length=2,
max_length=100)
    destination: Optional[str] = Field(None, min_length=2,
max_length=100)
    distance_km: Optional[float] = Field(None, gt=0, le=50000)

# ----- Transport -----

class TransportResponse(BaseModel):
    id: int
    name: str
    max_weight: float
    price_per_km: float

    class Config:
        from_attributes = True

class TransportCreate(BaseModel):
    name: str = Field(..., min_length=2, max_length=100)
    max_weight: float = Field(..., gt=0, le=100000)
    price_per_km: float = Field(..., gt=0, le=10000)

```

```

    @field_validator("name")
    @classmethod
    def strip_name(cls, v):
        return v.strip()

class TransportUpdate(BaseModel):
    name: Optional[str] = Field(None, min_length=2,
max_length=100)
    max_weight: Optional[float] = Field(None, gt=0, le=100000)
    price_per_km: Optional[float] = Field(None, gt=0, le=10000)

```

Листинг А.4 – REST API маршруты для управления грузами, транспортом и маршрутами, включая загрузку и удаление изображений

```

from fastapi import Depends, APIRouter, HTTPException, File,
UploadFile, Request, Response
from fastapi.responses import FileResponse
from typing import List, Optional
from pathlib import Path
from sqlalchemy.orm import Session
from db import SessionLocal
from schemas import TransportUpdate, RouteUpdate, CargoTypeUpdate
from models import CargoType, Transport, Route
from schemas import CargoTypeCreate, CargoTypeResponse,
TransportCreate, TransportResponse, RouteCreate, RouteResponse

router = APIRouter()

CARGO_IMAGE_DIR = Path("file_storage/cargos")
TRANSPORT_IMAGE_DIR = Path("file_storage/transport")

CARGO_IMAGE_DIR.mkdir(parents=True, exist_ok=True)
TRANSPORT_IMAGE_DIR.mkdir(parents=True, exist_ok=True)

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# -----
# Cargo CRUD + Images
# -----
@router.get("/cargo", response_model=List[CargoTypeResponse])
def get_all_cargo(limit: int = 100, db: Session =
Depends(get_db)):
    return db.query(CargoType).limit(limit).all()

@router.get("/cargo/{cargo_id}", response_model=CargoTypeResponse)

```

```

def get_cargo(cargo_id: int, db: Session = Depends(get_db)):
    db_cargo = db.query(CargoType).filter(CargoType.id ==
cargo_id).first()
    if not db_cargo:
        raise HTTPException(status_code=404, detail="Cargo not
found")
    return db_cargo

@router.post("/cargo", response_model=CargoTypeResponse)
def create_cargo(cargo: CargoTypeCreate, db: Session =
Depends(get_db)):
    db_cargo = CargoType(**cargo.dict())
    db.add(db_cargo)
    db.commit()
    db.refresh(db_cargo)
    return db_cargo

@router.put("/cargo/{cargo_id}", response_model=CargoTypeResponse)
def update_cargo(cargo_id: int, cargo: CargoTypeUpdate, db:
Session = Depends(get_db)):
    db_cargo = db.query(CargoType).filter(CargoType.id ==
cargo_id).first()
    if not db_cargo:
        raise HTTPException(status_code=404, detail="Cargo not
found")
    for key, value in cargo.dict(exclude_unset=True).items():
        setattr(db_cargo, key, value)
    db.commit()
    db.refresh(db_cargo)
    return db_cargo

@router.delete("/cargo/{cargo_id}")
def delete_cargo(cargo_id: int, db: Session = Depends(get_db)):
    db_cargo = db.query(CargoType).filter(CargoType.id ==
cargo_id).first()
    if not db_cargo:
        raise HTTPException(status_code=404, detail="Cargo not
found")
    db.delete(db_cargo)
    db.commit()
    return {"detail": "Deleted successfully"}

# Cargo Images
@router.get("/cargo/{cargo_id}/image")
async def get_cargo_image(cargo_id: int):
    image_path = CARGO_IMAGE_DIR / f"{cargo_id}.jpg"
    if not image_path.exists():
        default_path = CARGO_IMAGE_DIR / "default.jpg"
        if default_path.exists():
            return FileResponse(default_path)
        raise HTTPException(status_code=404, detail="Image not
found")

```

```

        return FileResponse(image_path)

@router.put("/cargo/{cargo_id}/image")
async def upload_cargo_image(cargo_id: int, image: UploadFile =
File(...)):
    if not image.content_type.startswith("image/"):
        raise HTTPException(status_code=400, detail="File must be
an image")
    content = await image.read()
    if len(content) > 5 * 1024 * 1024:
        raise HTTPException(status_code=400, detail="File too
large (max 5MB)")
    file_path = CARGO_IMAGE_DIR / f"{cargo_id}.jpg"
    with open(file_path, "wb") as f:
        f.write(content)
    return {"message": "Image uploaded successfully", "filename":
f"{cargo_id}.jpg"}

@router.delete("/cargo/{cargo_id}/image")
async def delete_cargo_image(cargo_id: int):
    file_path = CARGO_IMAGE_DIR / f"{cargo_id}.jpg"
    if file_path.exists():
        file_path.unlink()
        return {"message": "Image deleted successfully"}
    raise HTTPException(status_code=404, detail="Image not found")

# -----
# Transport CRUD + Images
# -----
@router.get("/transport", response_model=List[TransportResponse])
def get_all_transport(limit: int = 100, db: Session =
Depends(get_db)):
    return db.query(Transport).limit(limit).all()

@router.get("/transport/{transport_id}",
response_model=TransportResponse)
def get_transport(transport_id: int, db: Session =
Depends(get_db)):
    db_transport = db.query(Transport).filter(Transport.id ==
transport_id).first()
    if not db_transport:
        raise HTTPException(status_code=404, detail="Transport not
found")
    return db_transport

@router.post("/transport", response_model=TransportResponse)
def create_transport(transport: TransportCreate, db: Session =
Depends(get_db)):
    db_transport = Transport(**transport.dict())
    db.add(db_transport)
    db.commit()
    db.refresh(db_transport)

```

```

        return db_transport

@router.put("/transport/{transport_id}",
            response_model=TransportResponse)
def update_transport(transport_id: int, transport:
TransportUpdate, db: Session = Depends(get_db)):
    db_transport = db.query(Transport).filter(Transport.id ==
transport_id).first()
    if not db_transport:
        raise HTTPException(status_code=404, detail="Transport not
found")
    for key, value in transport.dict(exclude_unset=True).items():
        setattr(db_transport, key, value)
    db.commit()
    db.refresh(db_transport)
    return db_transport

@router.delete("/transport/{transport_id}")
def delete_transport(transport_id: int, db: Session =
Depends(get_db)):
    db_transport = db.query(Transport).filter(Transport.id ==
transport_id).first()
    if not db_transport:
        raise HTTPException(status_code=404, detail="Transport not
found")
    db.delete(db_transport)
    db.commit()
    return {"detail": "Deleted successfully"}

# Transport Images
@router.get("/transport/{transport_id}/image")
async def get_transport_image(transport_id: int):
    image_path = TRANSPORT_IMAGE_DIR / f"{transport_id}.jpg"
    if not image_path.exists():
        default_path = TRANSPORT_IMAGE_DIR / "default.jpg"
        if default_path.exists():
            return FileResponse(default_path)
        raise HTTPException(status_code=404, detail="Image not
found")
    return FileResponse(image_path)

@router.put("/transport/{transport_id}/image")
async def upload_transport_image(transport_id: int, image:
UploadFile = File(...)):
    if not image.content_type.startswith("image/"):
        raise HTTPException(status_code=400, detail="File must be
an image")
    content = await image.read()
    if len(content) > 5 * 1024 * 1024:
        raise HTTPException(status_code=400, detail="File too
large (max 5MB)")
    file_path = TRANSPORT_IMAGE_DIR / f"{transport_id}.jpg"

```

```

        with open(file_path, "wb") as f:
            f.write(content)
        return {"message": "Image uploaded successfully", "filename":
f"{transport_id}.jpg"}

@router.delete("/transport/{transport_id}/image")
async def delete_transport_image(transport_id: int):
    file_path = TRANSPORT_IMAGE_DIR / f"{transport_id}.jpg"
    if file_path.exists():
        file_path.unlink()
        return {"message": "Image deleted successfully"}
    raise HTTPException(status_code=404, detail="Image not found")

# -----
# Route CRUD
# -----
@router.get("/route", response_model=List[RouteResponse])
def get_all_routes(limit: int = 100, db: Session =
Depends(get_db)):
    return db.query(Route).limit(limit).all()

@router.get("/route/{route_id}", response_model=RouteResponse)
def get_route(route_id: int, db: Session = Depends(get_db)):
    db_route = db.query(Route).filter(Route.id ==
route_id).first()
    if not db_route:
        raise HTTPException(status_code=404, detail="Route not
found")
    return db_route

from sqlalchemy.exc import SQLAlchemyError
from fastapi import HTTPException

@router.post("/route", response_model=RouteResponse)
def create_route(route: RouteCreate, db: Session =
Depends(get_db)):
    try:
        db_route = Route(**route.dict())
        db.add(db_route)
        db.commit()
        db.refresh(db_route)
        return db_route
    except SQLAlchemyError as e:
        db.rollback() # откат транзакции
        print("SQLAlchemy ERROR:", e) # <-- теперь увидим точную
ошибку
        raise HTTPException(status_code=500, detail="Ошибка при
создании маршрута")

@router.put("/route/{route_id}", response_model=RouteResponse)

```

```

def update_route(route_id: int, route: RouteUpdate, db: Session = Depends(get_db)):
    db_route = db.query(Route).filter(Route.id == route_id).first()
    if not db_route:
        raise HTTPException(status_code=404, detail="Route not found")
    for key, value in route.dict(exclude_unset=True).items():
        setattr(db_route, key, value)
    db.commit()
    db.refresh(db_route)
    return db_route

@router.delete("/route/{route_id}")
def delete_route(route_id: int, db: Session = Depends(get_db)):
    db_route = db.query(Route).filter(Route.id == route_id).first()
    if not db_route:
        raise HTTPException(status_code=404, detail="Route not found")
    db.delete(db_route)
    db.commit()
    return {"detail": "Deleted successfully"}

```

Листинг А.5 – Dockerfile для контейнеризации FastAPI-сервиса и подготовки файлового хранилища

```

FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Копируем ВСЕ файлы включая file_storage
COPY . .

# Создаем папки если их нет
RUN mkdir -p /app/file_storage/cargos /app/file_storage/transport

# Копируем дефолтные фото если они есть
COPY file_storage/ /app/file_storage/

# Или создаем дефолтные если их нет
RUN if [ ! -f /app/file_storage/cargos/default.jpg ]; then \
    echo -ne \
    '\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01' > \
    /app/file_storage/cargos/default.jpg; \
    fi

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8001"]

```

Листинг А.6 – API Gateway: проксирование запросов между микросервисами (auth, cargo, order, profile) с JWT-аутентификацией и ролями

```

from fastapi import APIRouter, Request, HTTPException, Depends,
File, UploadFile, Response
from deps import require_user, require_admin
from typing import Optional
import httpx

router = APIRouter()

CARGO_SERVICE_URL = "http://cargo_service:8001"
ORDER_SERVICE_URL = "http://order_service:8004"
AUTH_SERVICE_URL = "http://auth_service:8005"
PROFILE_SERVICE_URL = "http://user_profile_service:8006"

# -----
# Helpers
# -----
def build_auth_headers(request: Request, user: dict, json: bool =
True):
    user_id = user.get("sub") or user.get("id")
    if not user_id:
        raise HTTPException(status_code=500, detail="User id not
found in auth context")

    headers = {
        "Authorization": request.headers.get("authorization"),
        "X-User-Id": str(user_id),
        "X-User-Role": user.get("role"),
    }
    if user.get("email"):
        headers["X-User-Email"] = user["email"]
    if json:
        headers["Content-Type"] = "application/json"
    return headers

# -----
# Auth
# -----
@router.post("/login")
async def login_gateway(request: Request):
    data = await request.json()
    async with httpx.AsyncClient() as client:
        response = await client.post(f"{AUTH_SERVICE_URL}/login",
json=data)
    return Response(content=response.content,
status_code=response.status_code, media_type="application/json")

```

```

@router.post("/register")
async def register_gateway(request: Request):
    data = await request.json()
    async with httpx.AsyncClient() as client:
        response = await
client.post(f"{AUTH_SERVICE_URL}/register", json=data)
        return Response(content=response.content,
status_code=response.status_code, media_type="application/json")

# -----
# Cargo
# -----
@router.get("/cargos")
async def get_all_cargo(user=Depends(require_user)):
    async with httpx.AsyncClient() as client:
        response = await client.get(f"{CARGO_SERVICE_URL}/cargo")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.get("/cargo/{cargo_id}")
async def get_cargo(cargo_id: int, user=Depends(require_user)):
    async with httpx.AsyncClient() as client:
        response = await
client.get(f"{CARGO_SERVICE_URL}/cargo/{cargo_id}")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.post("/cargo")
async def create_cargo(request: Request,
user=Depends(require_admin)):
    data = await request.json()
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await client.post(f"{CARGO_SERVICE_URL}/cargo",
json=data, headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.put("/cargo/{cargo_id}")
async def update_cargo(
    cargo_id: int,
    request: Request,

```

```

        user=Depends(require_admin),
        image: Optional[UploadFile] = None
    ):
        headers = build_auth_headers(request, user)

        # Проверяем, multipart или json
        content_type = request.headers.get("content-type", "")
        if content_type.startswith("multipart/form-data"):
            # Приходит форма (картинка + поля)
            form_data = await request.form()
            data = {k: form_data[k] for k in form_data if k !=
"image"}
            files = None
            if image:
                files = {"image": (image.filename, await image.read(),
image.content_type)}
            else:
                # Приходит JSON (только текстовые поля)
                data = await request.json()
                files = None

        async with httpx.AsyncClient() as client:
            if files:
                response = await client.put(
                    f"{CARGO_SERVICE_URL}/cargo/{cargo_id}",
                    data=data,
                    files=files,
                    headers=headers
                )
            else:
                response = await client.put(
                    f"{CARGO_SERVICE_URL}/cargo/{cargo_id}",
                    json=data,
                    headers=headers
                )

            if response.status_code != 200:
                raise HTTPException(status_code=response.status_code,
detail=response.text)
            return response.json()

@router.delete("/cargo/{cargo_id}")
async def delete_cargo(cargo_id: int,
user=Depends(require_admin)):
    async with httpx.AsyncClient() as client:
        response = await
client.delete(f"{CARGO_SERVICE_URL}/cargo/{cargo_id}")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

```

```

# -----
# Cargo / Transport Images
# -----
@router.put("/cargo/{cargo_id}/image")
async def upload_cargo_image(cargo_id: int, request: Request,
user=Depends(require_admin), image: UploadFile = File(...)):
    headers = build_auth_headers(request, user, json=False)
    async with httpx.AsyncClient() as client:
        files = {"image": (image.filename, await image.read(),
image.content_type)}
        response = await
client.put(f"{CARGO_SERVICE_URL}/cargo/{cargo_id}/image",
files=files, headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.get("/cargo/{cargo_id}/image")
async def get_cargo_image(cargo_id: int):
    async with httpx.AsyncClient() as client:
        response = await
client.get(f"{CARGO_SERVICE_URL}/cargo/{cargo_id}/image")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return Response(content=response.content,
media_type=response.headers.get("content-type", "image/jpeg"))

@router.put("/transport/{transport_id}/image")
async def upload_transport_image(transport_id: int, request:
Request, user=Depends(require_admin), image: UploadFile =
File(...)):
    headers = build_auth_headers(request, user, json=False)
    async with httpx.AsyncClient() as client:
        files = {"image": (image.filename, await image.read(),
image.content_type)}
        response = await
client.put(f"{CARGO_SERVICE_URL}/transport/{transport_id}/image",
files=files, headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.get("/transport/{transport_id}/image")
async def get_transport_image(transport_id: int):

```

```

        async with httpx.AsyncClient() as client:
            response = await
client.get(f"{CARGO_SERVICE_URL}/transport/{transport_id}/image")
            if response.status_code != 200:
                raise HTTPException(status_code=response.status_code,
detail=response.text)
            return Response(content=response.content,
media_type=response.headers.get("content-type", "image/jpeg"))

@router.get("/transports")
async def get_all_transport(user=Depends(require_user)):
    async with httpx.AsyncClient() as client:
        response = await
client.get(f"{CARGO_SERVICE_URL}/transport")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.get("/transport/{transport_id}")
async def get_transport(transport_id: int,
user=Depends(require_user)):
    async with httpx.AsyncClient() as client:
        response = await
client.get(f"{CARGO_SERVICE_URL}/transport/{transport_id}")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.post("/transport")
async def create_transport(request: Request,
user=Depends(require_admin)):
    data = await request.json()
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await
client.post(f"{CARGO_SERVICE_URL}/transport", json=data,
headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.put("/transport/{transport_id}")
async def update_transport(
    transport_id: int,
    request: Request,
    user=Depends(require_admin),

```

```

        image: Optional[UploadFile] = None
    ):
        headers = build_auth_headers(request, user)

        # Определяем, что пришло: multipart (форма) или JSON
        content_type = request.headers.get("content-type", "")
        if content_type.startswith("multipart/form-data"):
            # Форма с картинкой
            form_data = await request.form()
            data = {k: form_data[k] for k in form_data if k !=
"image"}
            files = None
            if image:
                files = {"image": (image.filename, await image.read(),
image.content_type)}
            else:
                # JSON только для текстовых полей
                data = await request.json()
                files = None

        async with httpx.AsyncClient() as client:
            if files:
                response = await client.put(
                    f"{CARGO_SERVICE_URL}/transport/{transport_id}",
                    data=data,
                    files=files,
                    headers=headers
                )
            else:
                response = await client.put(
                    f"{CARGO_SERVICE_URL}/transport/{transport_id}",
                    json=data,
                    headers=headers
                )

            if response.status_code != 200:
                raise HTTPException(status_code=response.status_code,
detail=response.text)
            return response.json()

@router.delete("/transport/{transport_id}")
async def delete_transport(transport_id: int,
user=Depends(require_admin)):
    async with httpx.AsyncClient() as client:
        response = await
client.delete(f"{CARGO_SERVICE_URL}/transport/{transport_id}")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

```

```

# -----
# Routes
# -----
@router.get("/routes")
async def get_all_routes(user=Depends(require_user)):
    async with httpx.AsyncClient() as client:
        response = await client.get(f"{CARGO_SERVICE_URL}/route")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.get("/route/{route_id}")
async def get_route(route_id: int, user=Depends(require_user)):
    async with httpx.AsyncClient() as client:
        response = await
client.get(f"{CARGO_SERVICE_URL}/route/{route_id}")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.post("/route")
async def create_route(request: Request,
user=Depends(require_admin)):
    data = await request.json()
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await client.post(f"{CARGO_SERVICE_URL}/route",
json=data, headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.put("/route/{route_id}")
async def update_route(route_id: int, request: Request,
user=Depends(require_admin)):
    data = await request.json()
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await
client.put(f"{CARGO_SERVICE_URL}/route/{route_id}", json=data,
headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

```

```

@router.delete("/route/{route_id}")
async def delete_route(route_id: int,
user=Depends(require_admin)):
    async with httpx.AsyncClient() as client:
        response = await
client.delete(f"{CARGO_SERVICE_URL}/route/{route_id}")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

# -----
# Orders
# -----
@router.post("/order")
async def create_order(request: Request,
user=Depends(require_user)):
    data = await request.json()
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await client.post(f"{ORDER_SERVICE_URL}/order",
json=data, headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.get("/orders")
async def get_all_orders(request: Request,
user=Depends(require_user)):
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await client.get(f"{ORDER_SERVICE_URL}/orders",
headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

@router.get("/order/{order_id}")
async def get_order(order_id: int, request: Request,
user=Depends(require_user)):
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await
client.get(f"{ORDER_SERVICE_URL}/order/{order_id}",
headers=headers)

```

```

        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
                                detail=response.text)
        return response.json()

@router.put("/order/{order_id}/status")
async def update_order_status(order_id: int, request: Request,
                               user=Depends(require_admin)):
    data = await request.json()
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await
client.put(f"{ORDER_SERVICE_URL}/order/{order_id}/status",
           json=data, headers=headers)
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
                                detail=response.text)
        return response.json()

@router.delete("/order/{order_id}")
async def delete_order(order_id: int, request: Request,
                        user=Depends(require_admin)):
    headers = build_auth_headers(request, user)
    async with httpx.AsyncClient() as client:
        response = await
client.delete(f"{ORDER_SERVICE_URL}/order/{order_id}",
              headers=headers)

        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
                                detail=response.text)

        return response.json()

# -----
# Users
# -----
@router.get("/users")
async def get_all_users(user=Depends(require_admin)):
    async with httpx.AsyncClient() as client:
        response = await client.get(f"{AUTH_SERVICE_URL}/users")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
                                detail=response.text)
        return response.json()

@router.get("/user/{user_id}")
async def get_user(user_id: int, user=Depends(require_admin)):
    async with httpx.AsyncClient() as client:

```

```

        response = await
client.get(f"{AUTH_SERVICE_URL}/user/{user_id}")
        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code,
detail=response.text)
        return response.json()

```

```

@router.get("/debug/user")
def debug_user(user=Depends(require_user)):
    return user

```

```

# -----
# Profile
# -----

```

```

@router.get("/profile")
async def get_profile(request: Request,
user=Depends(require_user)):
    headers = build_auth_headers(request, user)

    async with httpx.AsyncClient() as client:
        response = await client.get(
            f"{PROFILE_SERVICE_URL}/profile",
            headers=headers
        )

    return Response(
        content=response.content,
        status_code=response.status_code,
        media_type="application/json"
    )

```

```

@router.post("/profile")
async def create_profile(request: Request,
user=Depends(require_user)):
    data = await request.json()
    headers = build_auth_headers(request, user)

    async with httpx.AsyncClient() as client:
        response = await client.post(
            f"{PROFILE_SERVICE_URL}/profile",
            json=data,
            headers=headers
        )

    return Response(
        content=response.content,
        status_code=response.status_code,
        media_type="application/json"
    )

```

```

@router.put("/profile")
async def update_profile(request: Request,
user=Depends(require_user)):
    data = await request.json()
    headers = build_auth_headers(request, user)

    async with httpx.AsyncClient() as client:
        response = await client.put(
            f"{PROFILE_SERVICE_URL}/profile",
            json=data,
            headers=headers
        )

    return Response(
        content=response.content,
        status_code=response.status_code,
        media_type="application/json"
    )

@router.delete("/profile")
async def delete_profile(request: Request,
user=Depends(require_user)):
    headers = build_auth_headers(request, user)

    async with httpx.AsyncClient() as client:
        response = await client.delete(
            f"{PROFILE_SERVICE_URL}/profile",
            headers=headers
        )

    return Response(
        content=response.content,
        status_code=response.status_code,
        media_type="application/json"
    )

```

Листинг А.7 – API-методы для работы с сущностью «Груз» на клиентской стороне (CRUD и загрузка изображений)

```

import { apiFetch } from "../client"

export async function getAllCargo() {
    return apiFetch("http://localhost:8000/cargos")
}

export async function getCargoById(id) {
    return apiFetch(`http://localhost:8000/cargo/${id}`)
}

```

```

export async function createCargo(data) {
  return apiFetch("http://localhost:8000/cargo", {
    method: "POST",
    body: JSON.stringify(data)
  })
}

export async function updateCargo(id, data) {
  return apiFetch(`http://localhost:8000/cargo/${id}`, {
    method: "PUT",
    body: JSON.stringify(data)
  })
}

export async function deleteCargo(id) {
  return apiFetch(`http://localhost:8000/cargo/${id}`, {
    method: "DELETE"
  })
}

export async function uploadCargoImage(cargoId, imageFile) {
  const formData = new FormData()
  formData.append("image", imageFile)

  const token = localStorage.getItem("token")

  const response = await
fetch(`http://localhost:8000/cargo/${cargoId}/image`, {
  method: "PUT",
  headers: {
    "Authorization": `Bearer ${token}`
  },
  body: formData
})

  if (!response.ok) {
    const error = await response.json()
    throw new Error(error.detail || "Ошибка загрузки фото")
  }

  return response.json()
}

export async function getCargoImageUrl(cargoId) {
  return
`http://localhost:8000/cargo/${cargoId}/image?t=${Date.now()}`
}

```

Листинг A.8 – Инициализация API Gateway FastAPI с подключением роутеров и настройкой CORS для фронтенда

```

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from routes import router

app = FastAPI(title="API Gateway")
app.include_router(router)

origins = [
    "http://localhost:5173",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],      # GET, POST, OPTIONS
    allow_headers=["*"],      # Authorization, Content-Type и др.
)

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", host="0.0.0.0", port=8000,
        reload=True)

```

Листинг А.9 – Унифицированная функция HTTP-запросов к API с автоматическим добавлением JWT-токена

```

import { getToken } from "../auth/token"

export async function apiFetch(url, options = {}) {
    const token = getToken()

    const headers = {
        "Content-Type": "application/json",
        ...options.headers,
    }

    if (token) {
        headers.Authorization = `Bearer ${token}`
    }

    const response = await fetch(url, {
        ...options,
        headers,
    })

    if (!response.ok) {
        const text = await response.json()
        throw new Error(text.detail || "Network error")
    }

    return response.json()
}

```

```
}
```

Листинг А.10 – Компонент защиты маршрутов с проверкой JWT-токена

```
import { Navigate } from "react-router-dom"
import { getToken } from "../token"

export default function RequireAuth({ children }) {
  const token = getToken()

  if (!token) {
    return <Navigate to="/login" replace />
  }

  return children
}
```

Листинг А.11 – Административная форма создания типа груза с валидацией, загрузкой изображения и отправкой данных на API

```
import { useEffect, useState } from "react"
import { getAllCargo, deleteCargo } from "../../api/cargos.js"
import { getUserRole } from "../../auth/user"
import { useNavigate } from "react-router-dom"

export default function Cargos() {
  const [cargoList, setCargoList] = useState([])
  const [loading, setLoading] = useState(true)
  const [error, setError] = useState(null)
  const navigate = useNavigate()
  const role = getUserRole()
  const isAdmin = role === "admin"

  useEffect(() => {
    loadCargos()
  }, [])

  const loadCargos = async () => {
    try {
      const data = await getAllCargo()
      setCargoList(Array.isArray(data) ? data : [])
    } catch (e) {
      console.error(e)
      setError("Failed to load cargo")
    } finally {
      setLoading(false)
    }
  }

  const handleDelete = async (cargoId, cargoName) => {
    if (!window.confirm(`Вы уверены, что хотите удалить груз "${cargoName}"?`)) {

```

```

    return
  }

  try {
    await deleteCargo(cargoId)
    alert("Груз успешно удален")
    loadCargos()
  } catch (e) {
    alert("Ошибка при удалении груза: " + e.message)
  }
}

const handleEdit = (cargoId) => {
  navigate(`/admin/edit-cargo/${cargoId}`)
}

const handleAddNew = () => {
  navigate("/admin/add-cargo")
}

const getCargoImage = (cargoId) => {
  return
`http://localhost:8000/cargo/${cargoId}/image?t=${Date.now()}`
}

return (
  <div className="p-6">
    <div className="max-w-7xl mx-auto">
      {/* Заголовок с кнопками */}
      <div className="mb-10">
        <div className="flex justify-between items-start">
          <div>
            <h1 className="text-2xl font-bold text-gray-
900">Типы грузов</h1>
            <p className="text-gray-600 mt-2">Каталог доступных
типов грузов для перевозки</p>
          </div>

          {isAdmin && (
            <button
              onClick={handleAddNew}
              className="px-4 py-2 bg-gray-900 hover:bg-black
text-white text-sm font-medium rounded-md transition-colors"
            >
              + Добавить груз
            </button>
          )}
        </div>
      </div>

      {/* Ошибки */}
      {error && (

```

```
<div className="mb-6 p-4 bg-red-50 border border-red-200 rounded">  
    <p className="text-red-700 font-medium">{error}</p>  
</div>  
)}  
  
{/* Сетка с карточками грузов */}  
{loading ? (  
    <div className="text-center py-16">  
        <div className="inline-block animate-spin rounded-full h-8 w-8 border-b-2 border-gray-900"></div>  
        <p className="mt-4 text-gray-600">Загрузка грузов...</p>  
    </div>  
    ) : cargoList.length === 0 ? (  
        <div className="text-center py-16">  
            <p className="text-gray-500 text-lg mb-4">Грузы не найдены</p>  
            {isAdmin && (  
                <button  
                    onClick={handleAddNew}  
                    className="px-4 py-2 bg-gray-900 hover:bg-black text-white text-sm font-medium rounded-md transition-colors">  
                        Создать первый тип груза  
                </button>  
            )}  
        </div>  
    ) : (  
        <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">  
            {cargoList.map(c => (  
                <div key={c.id} className="bg-white border border-gray-200 rounded-lg overflow-hidden hover:shadow-md transition-shadow">  
                    {/* Фото груза */}  
                    <div className="h-48 bg-gray-100 overflow-hidden">  
                        <img  
                            src={getCargoImage(c.id)}  
                            alt={c.name}  
                            className="w-full h-full"  
                            style={{  
                                maxWidth: '100%',  
                                maxHeight: '100%',  
                                objectFit: 'contain',  
                                objectPosition: 'center'  
                            }} />  
                        <br/>  
                        <p>Груз: {c.name}</p>  
                        <p>Вес: {c.weight}</p>  
                        <p>Объем: {c.volume}</p>  
                        <p>Статус: {c.status}</p>  
                    </div>  
                </div>  
            )})  
        </div>  
    )  
}
```

```

        e.target.src =
"http://localhost:8000/cargo/default/image"
        // Или локальное фото
        // e.target.src =
"/images/cargos/default.jpg"
    }}
    />
</div>

    { /* Информация о грузе */ }
    <div className="p-6">
        <div className="flex justify-between items-start
mb-4">
            <div>
                <h3 className="text-lg font-medium text-
gray-900">{c.name}</h3>
                <p className="text-sm text-gray-500 mt-
1">ID: {c.id}</p>
            </div>
            <div className="flex items-center space-x-2">
                <span className="px-2 py-1 bg-gray-900 text-
white text-xs font-medium rounded">
                    Груз
                </span>
                {isAdmin && (
                    <>
                        <button
                            onClick={() => handleEdit(c.id)}
                            className="p-1 text-gray-500
hover:text-gray-700 transition-colors"
                            title="Редактировать"
                        >
                            
                        </button>
                        <button
                            onClick={() => handleDelete(c.id,
c.name)}
                            className="p-1 text-gray-500
hover:text-red-600 transition-colors"
                            title="Удалить"
                        >
                            
                        </button>
                    </>
                ) }
            </div>
        </div>

    { /* Характеристики */ }
    <div className="space-y-3">
        <div className="flex justify-between items-
center py-2 border-b border-gray-100">

```

```

        <span className="text-sm text-gray-
600">Максимальный вес</span>
        <span className="font-medium text-gray-
900">{c.max_weight} кг</span>
    </div>

    {c.dimensions && (
        <div className="flex justify-between items-
center py-2 border-b border-gray-100">
            <span className="text-sm text-gray-
600">Габариты</span>
            <span className="font-medium text-gray-
900">{c.dimensions}</span>
        </div>
    )}

    {c.price_per_kg && (
        <div className="flex justify-between items-
center py-2 border-b border-gray-100">
            <span className="text-sm text-gray-
600">Цена за кг</span>
            <span className="font-medium text-green-
600">{c.price_per_kg} ₺</span>
        </div>
    )}

    {c.description && (
        <div className="pt-2">
            <p className="text-sm text-gray-
600">{c.description}</p>
        </div>
    )}
</div>
</div>
</div>
    )}
</div>
</div>
)
}

```

Листинг А.12 – Маршрутизация клиентской части

```

import { BrowserRouter, Routes, Route, Navigate } from "react-
router-dom"
import Login from "../pages/Login"
import Users from "../pages/admin/Users"
import Cargos from "../pages/admin/Cargos"
import RoutesPage from "../pages/admin/RoutesPage"
import Transports from "../pages/admin/Transports"
import Orders from "../pages/user/Orders"

```

```

import CreateOrder from "../pages/user/CreateOrder"
// Добавляем новые страницы
import AddRoute from "../pages/admin/AddRoute"
import EditRoute from "../pages/admin/EditRoute"
import AddTransport from "../pages/admin/AddTransport"
import EditTransport from "../pages/admin/EditTransport"
import AddCargo from "../pages/admin/AddCargo"
import EditCargo from "../pages/admin/EditCargo"
import RegisterPage from "../pages/RegisterPage.jsx"
import ProfilePage from "../pages/user/Profile.jsx";

import RequireAuth from "../auth/RequireAuth"
import Layout from "../layout/Layout"

export default function AppRouter() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Navigate to="/login" replace />} />
      />
      <Route path="/register" element={<RegisterPage />} />
      <Route path="/login" element={<Login />} />

      <Route
        element={
          <RequireAuth>
            <Layout />
          </RequireAuth>
        }
      >
        <Route path="/profile" element={<ProfilePage />} />
        <Route path="/users" element={<Users />} />
        <Route path="/cargos" element={<Cargos />} />
        <Route path="/routes" element={<RoutesPage />} />
        <Route path="/transport" element={<Transports />} />
        <Route path="/orders" element={<Orders />} />
        <Route path="/create_order" element={<CreateOrder />} />

        {/* Админские страницы для CRUD операций */}
        <Route path="/admin/add-route" element={<AddRoute />} />
        <Route path="/admin/edit-route/:id" element={<EditRoute
      />} />
        <Route path="/admin/add-transport"
      element={<AddTransport />} />
        <Route path="/admin/edit-transport/:id"
      element={<EditTransport />} />
        <Route path="/admin/add-cargo" element={<AddCargo />} />
        <Route path="/admin/edit-cargo/:id" element={<EditCargo
      />} />
      </Route>
    </BrowserRouter>
  )
}

```

```

        <Route path="*" element={<Navigate to="/login" replace />}
/>
    </Routes>
  </BrowserRouter>
)
}

```

Листинг Б.13 – Корневой компонент React-приложения с подключением системы маршрутизации

```

import AppRouter from "../router";

function App() {
  return <AppRouter />
}

export default App

```

Приложение Б
(необязательное)
Результаты тестирования программы

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Стройматериалы (макс. 5000 кг)

Описание: Бетон, кирпич, цемент

Маршрут

Выберите маршрут

Выберите один из пунктов списка.

Транспорт

Выберите транспорт

Вес единицы груза (кг)

0 кг

Количество

0 шт.

Сводка заказа

Тип груза

Стройматериалы

Маршрут

Не выбран

Транспорт

Не выбран

Параметры груза

Вес

0 кг

Количество

0 шт.

Расчет стоимости

Расстояние:

0 км

Тариф:

0 Р/км

Количество:

0 шт.

Итоговая стоимость

0 Р

Без учета НДС

Создать заказ

Рисунок Б.1 – Результат тест-кейса 1

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Выберите тип груза

Выберите один из пунктов списка.

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

КамАЗ (макс. 12000 кг, 60 Р/км)

Грузоподъемность: 12000 кг

Вес единицы груза (кг)

13000 кг

Количество

1 шт.

Сводка заказа

Тип груза

Не выбран

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

КамАЗ

60 Р/км

Параметры груза

Вес

13000 кг

Количество

1 шт.

Расчет стоимости

Расстояние:

715 км

Тариф:

60 Р/км

Количество:

1 шт.

Итоговая стоимость

42 900 Р

Без учета НДС (715км × 60Р × 1шт.)

Создать заказ

Рисунок Б.2 – Результат тест-кейса 2

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Стройматериалы (макс. 5000 кг)

Описание: Бетон, кирпич, цемент

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

Выберите транспорт

Вес единицы груза (кг)

6000 кг

Количество

-1 шт.

Сводка заказа

Тип груза

Стройматериалы

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

Не выбран

Параметры груза

Вес

6000 кг

Количество

-1 шт.

Расчет стоимости

Расстояние:

715 км

Тариф:

0 Р/км

Количество:

-1 шт.

Итоговая стоимость

Без учета НДС

0 Р

Создать заказ

Рисунок Б.3 – Результат тест-кейса 3

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Стройматериалы (макс. 5000 кг)

Описание: Бетон, кирпич, цемент

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

Фура (макс. 8000 кг, 50 Р/км)

Грузоподъемность: 8000 кг

Вес единицы груза (кг)

0 кг

Количество

0 шт.

Сводка заказа

Тип груза

Стройматериалы

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

Фура

50 Р/км

Параметры груза

Вес

0 кг

Количество

0 шт.

Расчет стоимости

Расстояние:

715 км

Тариф:

50 Р/км

Количество:

0 шт.

Итоговая стоимость

Без учета НДС

0 Р

Создать заказ

Рисунок Б.4 – Результат тест-кейса 4

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Стройматериалы (макс. 5000 кг)

Описание: Бетон, кирпич, цемент

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

Выберите транспорт

Вес единицы груза (кг)

6000 кг

10 шт.

Сводка заказа

Тип груза

Стройматериалы

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

Не выбран

Параметры груза

Вес

6000 кг

Количество

10 шт.

Расчет стоимости

Расстояние:

715 км

Тариф:

0 ₽/км

Количество:

10 шт.

Итоговая стоимость

Без учета НДС

0 ₽

Создать заказ

Рисунок Б.5 – Результат тест-кейса 5

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Мебель (макс. 3000 кг)

Описание: Шкафы, диваны, столы

Маршрут

Выберите маршрут

Транспорт

Газель (макс. 1500 кг, 25 ₽/км)

Грузоподъемность: 1500 кг

Вес единицы груза (кг)

1600 кг

Количество

1 шт.

Сводка заказа

Тип груза

Мебель

Маршрут

Не выбран

Транспорт

Газель

25 ₽/км

Параметры груза

Вес

1600 кг

Количество

1 шт.

Расчет стоимости

Расстояние:

0 км

Тариф:

25 ₽/км

Количество:

1 шт.

Итоговая стоимость

Без учета НДС

0 ₽

Создать заказ

Рисунок Б.6 – Результат тест-кейса 6

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Стройматериалы (макс. 5000 кг)

Описание: Бетон, кирпич, цемент

Маршрут

Выберите маршрут

Транспорт

Выберите транспорт

Вес единицы груза (кг)

10

кг

Количество

10

шт.

Сводка заказа

Тип груза

Стройматериалы

Маршрут

Не выбран

Транспорт

Не выбран

Параметры груза

Вес

10 кг

Количество

10 шт.

Расчет стоимости

Расстояние:

0 км

Тариф:

0 Р/км

Количество:

10 шт.

Итоговая стоимость

0 Р

Без учета НДС

Создать заказ

Рисунок Б.7 – Результат тест-кейса 7

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Выберите тип груза

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

Газель (макс. 1500 кг, 25 Р/км)

Грузоподъемность: 1500 кг

Вес единицы груза (кг)

10

кг

Количество

1

шт.

Сводка заказа

Тип груза

Не выбран

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

Газель

25 Р/км

Параметры груза

Вес

10 кг

Количество

0 шт.

Расчет стоимости

Расстояние:

715 км

Тариф:

25 Р/км

Количество:

0 шт.

Итоговая стоимость

0 Р

Без учета НДС

Создать заказ

Рисунок Б.8 – Результат тест-кейса 8

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Выберите тип груза

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

Выберите транспорт

Вес единицы груза (кг)

-1

кг

Количество

10

шт.

Сводка заказа

Тип груза

Не выбран

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

Не выбран

Параметры груза

Вес

-1 кг

Количество

10 шт.

Расчет стоимости

Расстояние:

715 км

Тариф:

0 Р/км

Количество:

10 шт.

Итоговая стоимость

0 Р

Без учета НДС

Создать заказ

Рисунок Б.9 – Результат тест-кейса 9

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Продукты (макс. 2000 кг)

Описание: Пищевые товары

Маршрут

Выберите маршрут

Транспорт

Выберите транспорт

Вес единицы груза (кг)

2100

кг

Количество

1

шт.

Сводка заказа

Тип груза

Продукты

Маршрут

Не выбран

Транспорт

Не выбран

Параметры груза

Вес

2100 кг

Количество

1 шт.

Расчет стоимости

Расстояние:

0 км

Тариф:

0 Р/км

Количество:

1 шт.

Итоговая стоимость

0 Р

Без учета НДС

Создать заказ

Рисунок Б.10 – Результат тест-кейса 10

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Выберите тип груза

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

Газель (макс. 1500 кг, 25 Р/км)

Грузоподъемность: 1500 кг

Вес единицы груза (кг)

2000 кг

Количество

0 шт.

Сводка заказа

Тип груза

Не выбран

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

Газель

25 Р/км

Параметры груза

Вес	Количество
2000 кг	0 шт.

Расчет стоимости

Расстояние:	715 км
Тариф:	25 Р/км
Количество:	0 шт.

Итоговая стоимость

Без учета НДС

0 Р

Создать заказ

Рисунок Б.11 – Результат тест-кейса 11

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Стройматериалы (макс. 5000 кг)

Описание: Бетон, кирпич, цемент

Маршрут

Выберите маршрут

Транспорт

Газель (макс. 1500 кг, 25 Р/км)

Грузоподъемность: 1500 кг

Вес единицы груза (кг)

2000 кг

Количество

-1 шт.

Сводка заказа

Тип груза

Стройматериалы

Маршрут

Не выбран

Транспорт

Газель

25 Р/км

Параметры груза

Вес	Количество
2000 кг	-1 шт.

Расчет стоимости

Расстояние:	0 км
Тариф:	25 Р/км
Количество:	-1 шт.

Итоговая стоимость

Без учета НДС

0 Р

Создать заказ

Рисунок Б.12 – Результат тест-кейса 12

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Все поля обязательны для заполнения

Тип груза

Стройматериалы (макс. 5000 кг)

Описание: Бетон, кирпич, цемент

Маршрут

Москва → Санкт-Петербург (715 км)

Расстояние: 715 км

Транспорт

Газель (макс. 1500 кг, 25 Р/км)

Грузоподъемность: 1500 кг

Вес единицы груза (кг)

2000

кг

Количество

1

шт.

Сводка заказа

Тип груза

Стройматериалы

Маршрут

Москва → Санкт-Петербург

Расстояние: 715 км

Транспорт

Газель

25 Р/км

Параметры груза

Вес

2000 кг

Количество

1 шт.

Расчет стоимости

Расстояние:

715 км

Тариф:

25 Р/км

Количество:

1 шт.

Итоговая стоимость

17 875 Р

(715км × 25Р × 1шт.)

Без учета НДС

Создать заказ

⚠ Сначала заполните профиль, чтобы создать заказ

⚠ Общий вес груза (2000 кг) превышает грузоподъемность транспорта (1500 кг)

Рисунок Б.13 – Результат тест-кейса 13

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Все поля обязательны для заполнения

Тип груза

Химия (макс. 1500 кг)

Описание: Бытовая и промышленная химия

Маршрут

Санкт-Петербург → Казань (1260 км)

Расстояние: 1260 км

Транспорт

Выберите транспорт

Выберите один из пунктов списка.

Вес единицы груза (кг)

10

кг

Количество

0

шт.

Сводка заказа

Тип груза

Химия

Маршрут

Санкт-Петербург → Казань

Расстояние: 1260 км

Транспорт

Не выбран

Параметры груза

Вес

10 кг

Количество

0 шт.

Расчет стоимости

Расстояние:

1260 км

Тариф:

0 Р/км

Количество:

0 шт.

Итоговая стоимость

0 Р

Без учета НДС

Создать заказ

Рисунок Б.14 – Результат тест-кейса 14

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Химия (макс. 1500 кг)

Описание: Бытовая и промышленная химия

Маршрут

Выберите маршрут

Транспорт

Газель (макс. 1500 кг, 25 Р/км)

Грузоподъемность: 1500 кг

Вес единицы груза (кг)

100

кг

Количество

10

шт.

Сводка заказа

Тип груза

Химия

Маршрут

Не выбран

Транспорт

Газель

25 Р/км

Параметры груза

Вес

100 кг

Количество

10 шт.

Расчет стоимости

Расстояние:

0 км

Тариф:

25 Р/км

Количество:

10 шт.

Итоговая стоимость

Без учета НДС:

0 Р

Создать заказ

Рисунок Б.15 – Результат тест-кейса 15

Создание заказа

Оформление новой грузоперевозки

Данные заказа

Заполните все обязательные поля

Тип груза

Химия (макс. 1500 кг)

Описание: Бытовая и промышленная химия

Маршрут

Москва → Владивосток (9300 км)

Расстояние: 9300 км

Транспорт

Выберите транспорт

Вес единицы груза (кг)

0.0

кг

Количество

5

шт.

Сводка заказа

Тип груза

Химия

Маршрут

Москва → Владивосток

Расстояние: 9300 км

Транспорт

Не выбран

Параметры груза

Вес

0 кг

Количество

5 шт.

Расчет стоимости

Расстояние:

9300 км

Тариф:

0 Р/км

Количество:

5 шт.

Итоговая стоимость

Без учета НДС:

0 Р

Создать заказ

Рисунок Б.16 – Результат тест-кейса 16