Министерство образования Республики Беларусь

Учреждение образования

БелорусскиЙ государственный университет

информатики и радиоэлектроники

Факультет информационных технологий и управления

Кафедра информационных технологий автоматизированных систем

Дисциплина: Системы управления базами данных

|  |
| --- |
|  |
|  |
|  |

Пояснительная записка

к курсовому проекту

™

**проектирование программного обеспечения АРЕНДЫ РЕКЛАМЫ В МЕТРО**

БГУИР КР 6*–*05*–*0612*–*03 149 ПЗ

|  |  |  |
| --- | --- | --- |
| Студент |  | П. М. Ясинский |
| Руководитель |  | Е. В. Протченко |
|  |  |  |
|  |  |  |
|  |  |  |

Минск 2025

Учреждение образования

«Белорусский государственный университет информатики   
и радиоэлектроники»

Факультет информационных технологий и управления

УТВЕРЖДАЮ

Заведующий кафедрой

*––––––––––––––––––––––––*

(подпись)

–––––––––––––––2025 г.

ЗАДАНИЕ

по курсовому проектированию

Студенту    *Ясинскому Павлу Михайловичу ––––––––––––––––––*

1. Тема проекта   Проектирование программного обеспечения аренды рекламы в метро.

2. Срок сдачи студентом законченного проекта – ––––*?–– –*

3. Исходные данные к проекту *Сайт аренды рекламы метро имеет систему заказа рекламы, которая позволяет заказчикам загрузить свою рекламу на сайт, отправить на проверку, а также разместить рекламу в метро. Система характеризуется удобством для заказчиков и имеет функции, дающая возможность им легко взаимодействовать с системой.*

*Авторизованные пользователи имеют возможность воспользоваться личным кабинетом, где расположена информация об аккаунте, а также созданные и арендованные рекламы и информация о её эффективности.*

*При создании рекламы заказчик имеет возможность выбрать размер рекламного баннера, изображение, которое будет размещено на баннере, а также место, где этот баннер будет располагаться.*

4. Содержание расчетно*–*пояснительной записки (перечень вопросов, которые подлежат разработке):

*Введение*

1. *Обзор исследуемой области*
2. *Проектирование автоматизированной информационной системы аренды рекламы в метро*
3. *Разработка автоматизированной информационной системы аренды рекламы в метро*

*Заключение*

*Список использованных источников*

*Приложения*

*Ведомость курсового проекта*

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

1. *Логическая схема базы данных (ПЛ, формат А1)?*
2. *Диаграмма базы данных в нотации Питера Чена*
3. *Диаграмма классов*
4. *Диаграмма вариантов использования*
5. *Схема алгоритма*
6. *Физическая схема базы данных*

6. Консультант по проекту *Е. В. Протченко–*

7. Дата выдачи задания –––– –*15 сентября 2025 г.–––––––––––––––––––   –*

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и трудоемкости отдельных этапов):

*раздел 1 к 15.03 – 15 %; ––––––––––––––––– ––––––––––––––––        –*

*раздел 2 к 15.04 – 50 %; –– –––––––––––––– ––––––––––––––––––       ––––*

*раздел 3 к 15.05 – 15 %; ––––––––––––––––––––– –––––––––––––––––*

*оформление пояснительной записки и графического   
материала к 20. 05 – 20 %*

*Защита курсового проекта с* ***02.06*** *по* ***07.06.2025*** *г.–––––––––––––––––––*

Руководитель*– Е. В. Протченко*

(подпись)

Задание принял к исполнению *–––––––\_\_\_\_\_\_\_\_\_\_\_\_ П. М. Ясинский*

(дата и подпись студента)

**СОДЕРЖАНИЕ**

[Введение 6](#_Toc214879098)

[1. Обзор исследуемой области 7](#_Toc214879099)

[1.1 Описание предметной области 7](#_Toc214879100)

[1.2 Обзор аналогов 11](#_Toc214879101)

[1.3 Постановка задачи 16](#_Toc214879102)

[2. Проектирование системы 17](#_Toc214879103)

[2.1 Проектирование базы данных 17](#_Toc214879104)

[2.2 Проектирование сайта 22](#_Toc214879105)

[3. Разработка информационной системы авиакомпании 28](#_Toc214879106)

[3.1 Выбор средств реализации 28](#_Toc214879107)

[3.2 Реализация базы данных 32](#_Toc214879108)

[3.3 Программная реализация приложения 37](#_Toc214879109)

[3.4 Инструкция пользователя 42](#_Toc214879110)

[Заключение 51](#_Toc214879111)

[*C*писок использованных источников 52](#_Toc214879112)

[Приложение А 53](#_Toc214879113)

[Приложение Б 54](#_Toc214879114)

[Ведомость курсового проекта 59](#_Toc214879115)

**Введение**

Курсовой проект по дисциплине «Системы управления базами данных» посвящён разработке программного обеспечения, предназначенного для автоматизации процесса аренды рекламных площадей в метрополитене. Основная цель работы — закрепление и практическое применение знаний, полученных при изучении принципов проектирования, создания и эксплуатации баз данных, а также освоение инструментов для реализации реальных информационных систем.

В ходе выполнения проекта рассматриваются ключевые этапы разработки: от построения логической структуры базы данных и нормализации таблиц до определения связей между сущностями и выбора ключей. Особое внимание уделено применению языка *SQL* для решения задач различной сложности — от базовых операций выборки до сложных аналитических запросов.

Предметная область аренды рекламы в метро представляет собой актуальное направление, связанное с управлением большим объёмом данных: сведениями о клиентах, баннерах, заявках, станциях метро, оплате и эффективности баннеров. Система должна обеспечивать регистрацию пользователей, возможность подачи заявок на размещение рекламы, прохождение модерации, а также мониторинг статистики и результатов рекламных размещений.

В процессе разработки показано, как реляционная модель данных позволяет поддерживать целостность информации, предотвращать противоречия и обеспечивать стабильную работу системы. Дополнительно рассматриваются методы оптимизации структуры хранения данных и ускорения выполнения запросов для повышения общей производительности приложения.

Таким образом, данный курсовой проект объединяет теоретические знания в области проектирования и администрирования баз данных с практическими навыками создания веб-приложений. Итогом работы является функционирующая система, позволяющая пользователям удобно арендовать рекламные места в метро, а администраторам — эффективно управлять размещениями.

1. **обзор исследуемой области**
   1. **Описание предметной области**

Развитие информационных систем для управления арендой рекламных площадей в метрополитене уходит корнями в начало *XX* века, когда метро только начинало становиться массовым видом транспорта в крупных городах, таких как Нью-Йорк и Лондон . В те времена системы общественного транспорта переживали быстрый подъем, что вызвало необходимость в более точном контроле за размещением рекламы и оперативном обмене данными между арендаторами и операторами. Основные операции, включая оформление договоров на баннеры, учет доступных мест и координацию графиков размещений, велись в ручном режиме, приводя к частым неточностям, повторению записей и замедлению обработки запросов. Стремление к упорядочиванию этих процессов послужило одним из импульсов для эволюции технологий в сфере транспортной рекламы.

Важным шагом вперед стало внедрение первых централизованных платформ для координации аренды рекламных поверхностей. Одним из пионеров здесь можно считать систему, аналогичную *Metro* *Vision*, появившуюся в конце 1980-х годов в Нью-Йорке в сотрудничестве с *Metropolitan* *Transportation* *Authority* (*MTA*) и технологическими партнерами . Эта платформа явилась ранним образцом масштабной информационной инфраструктуры, функционирующей в режиме близком к реальному времени, и позволяла рекламным агентствам обращаться к общей базе с актуальными сведениями о станциях, свободных слотах и расценках.

С технической стороны *Metro* *Vision* опиралась на сеть мониторов и терминалов, связанных с центральными вычислительными устройствами, размещенными в специализированных центрах обработки данных. Запросы обрабатывались с впечатляющей для того периода скоростью — всего за считанные секунды. Благодаря этому удалось впервые воплотить идею «дистанционного доступа» к ресурсам, хотя на практике это было организовано через периодические обновления в пакетном формате.

После *MTA* подобные подходы взяли на вооружение и другие крупные метрополитены: в Лондоне запустили систему для *Underground* с акцентом на цифровые экраны, в Париже — платформу для *RATP* под названием *Systeme* *d*'*Information* *Publicitaire*, а в Токио — *J*-*Ads* для координации рекламных кампаний. Эти решения не ограничивались простой арендой, но также собирали данные для анализа эффективности размещений и оптимизации распределения площадей по линиям.

В Советском Союзе развитие информационных технологий для управления рекламой в метрополитене происходило в условиях плановой экономики и строгой централизации. Главный упор делался на точный учёт рекламных площадей, составление графиков размещений и контроль выполнения договоров с государственными предприятиями. В 1970–1980-е годы для этих задач использовались вычислительные комплексы ЕС ЭВМ и СМ ЭВМ, к которым подключались терминалы в административных зданиях метрополитенов, отделах рекламы и на крупных станциях.

Несмотря на технологическое отставание от западных решений, уже в тот период начали формироваться первые централизованные базы данных, в которых собирались сведения о станциях, типах рекламных конструкций, сроках аренды, клиентах и финансовых расчётах. Это позволило частично автоматизировать документооборот и снизить количество ошибок при ручном ведении учёта.

Особое значение придавалось системам планирования и диспетчеризации рекламных кампаний. В московском, ленинградском и киевском метрополитенах создавались собственные вычислительные центры, где информация фиксировалась на магнитных лентах, дисках и перфокартах. Отсутствие полноценной сети между городами компенсировалось регулярным обменом данными на физических носителях, что всё же заложило фундамент для будущей единой информационной инфраструктуры метро.

Однако, несмотря на прогрессивность для своего времени, ранние системы управления рекламными площадями обладали существенными архитектурными, техническими и организационными ограничениями, которые заметно снижали их надёжность и возможности масштабирования.

Ключевая проблема систем того периода — как зарубежных (*Metro* *Vision*, *J*-*Ads*), так и отечественных (ЕС ЭВМ, СМ ЭВМ) — заключалась в крайней степени централизации. Все данные концентрировались и обрабатывались в одном вычислительном центре, который одновременно выполнял роли сервера, хранилища и вычислительного узла.

В те годы, когда глобальные сети ещё не существовали, обмен данными между станциями и центральными узлами осуществлялся по выделенным телефонным линиям или с помощью физической доставки магнитных лент и дисков. Единых стандартов протоколов не было — каждый метрополитен применял собственные форматы записи и кодирования информации. В Советском Союзе, к примеру, сведения из региональных управлений в центральный вычислительный центр часто передавались по телетайпным каналам, после чего операторы вручную вводили их в ЭВМ. Всё это резко ограничивало взаимодействие между метрополитенами разных городов и стран, особенно при организации трансферных рекламных кампаний и синхронизации графиков размещений. Информация о новых заявках, изменениях тарифов или освобождении площадей могла поступать с задержкой в несколько часов, а иногда и суток.

Ранние системы создавались обособленно под узкие задачи: оформление аренды, учёт рекламных конструкций, планирование кампаний, контроль оплаты. Каждая подсистема работала с собственными структурами данных, что исключало их автоматическое взаимодействие. В результате данные, внесённые в модуль подачи заявок, не попадали напрямую в финансовый учёт, а сведения о занятых местах на станциях не синхронизировались с графиками демонтажа. Это порождало дублирование записей, расхождения в отчётах и постоянную необходимость ручного переноса информации между отделами.

Отсутствие автоматических средств проверки целостности и синхронизации делало надёжность всей системы зависимой от внимательности сотрудников. Одна неверно введённая цифра в номере договора или коде станции могла исказить данные по всей цепочке. Регулярно возникала потребность в ручном сверке бумажных и электронных документов, что замедляло процессы и увеличивало нагрузку на персонал.

Таким образом, хотя первые информационные системы управления рекламными площадями в метро и стали важным шагом вперёд по сравнению с полностью ручным учётом, они оставались ограниченными по возможностям и надёжности. Они требовали постоянного контроля со стороны человека и не могли обеспечить полноценное взаимодействие между подразделениями, городами и тем более международными партнёрами.

Конец 1980-х и особенно 1990-е годы ознаменовали переломный этап в эволюции информационных систем управления рекламными площадями в метрополитене.

Распространение персональных компьютеров, появление локальных сетей и бурный рост рынка транспортной рекламы потребовали качественно нового уровня интеграции и оперативности работы с данными. Если в 1970–1980-е годы автоматизация касалась лишь отдельных процессов, то в 1990-е начался переход к объединению информации, созданию единых корпоративных хранилищ и стандартизации архитектурных решений.

Именно в этот период стали внедряться системы, построенные на корпоративных базах данных, где все сведения о станциях, клиентах, договорах и платежах хранились централизованно и были доступны всем задействованным службам одновременно.

Основой для создания интегрированных платформ стали корпоративные сети. Ранее доступ к информации осуществлялся исключительно через терминалы, жёстко привязанные к центральным ЭВМ, но теперь появилась возможность связать в единую внутреннюю сеть головной офис метрополитена, региональные управления, рекламные агентства, подрядчиков и партнёров. Сети, построенные на протоколах *TCP*/*IP* и *Ethernet*, обеспечили надёжный и защищённый обмен данными: от актуальных графиков размещений и статусов заявок до статистики охвата аудитории и финансовых отчётов в электронном формате.

Технологическим фундаментом этих преобразований стало массовое внедрение реляционных систем управления базами данных. Такие платформы, как *Oracle* *Database*, *IBM* *DB*2, *Informix* и *Microsoft* *SQL* *Server*, позволили хранить огромные массивы информации в строго структурированном виде, быстро выполнять поиск, фильтрацию и совместный доступ к записям.

Язык *SQL* открыл возможности для выполнения сложных аналитических запросов с объединением данных из нескольких таблиц — например, формирование отчётов о загрузке станций по линиям и времени суток или анализ эффективности рекламных конструкций за выбранный период. При этом реляционные СУБД гарантировали целостность информации благодаря использованию первичных и внешних ключей, ограничений уникальности и триггеров, что существенно повысило надёжность и достоверность всей системы управления рекламными площадями в метро.

С появлением Интернета в начале 1990-х годов открылась совершенно новая глава в истории информационных систем управления рекламными площадями в метрополитене. Всемирная сеть дала толчок к созданию первых веб-сайтов метрополитенов и рекламных операторов, которые поначалу выполняли роль информационных витрин, а затем эволюционировали в полноценные онлайн-платформы для подачи заявок, бронирования мест и заключения договоров на размещение рекламы. Благодаря универсальным протоколам обмена данными (*HTTP*, *FTP*, *SMTP*) и стандартам представления информации (*XML*, а позже *JSON*) стало возможным автоматическое взаимодействие между различными участниками рынка — рекламными агентствами, платформами цифровой рекламы (например, *JCDecaux*, *Clear* *Channel* *Outdoor*, *Exterion* *Media*), глобальными системами дистрибуции и внутренними базами данных метрополитенов.

Таким образом, массовое распространение Интернета выступило мощным катализатором цифровой трансформации всей сферы транспортной рекламы. Оно не только резко ускорило и упростило обмен данными, но и кардинально изменило подход к построению информационных систем: из закрытых локальных комплексов они превратились в открытые, взаимосвязанные сетевые инфраструктуры, доступные из любой точки мира.

* 1. **Обзор аналогов**

В наше время онлайн-платформы метрополитенов выполняют важную функцию не только в организации аренды рекламных поверхностей, но и в оказании информационной помощи клиентам, а также интеграции с сервисами партнёров. Проанализируем три примера систем размещения рекламы в метрополитенах — *NEBO* *Digital*, Реклама в метро Санкт-Петербург и Оскар — чтобы оценить их возможности, сильные стороны и слабые места с позиции рекламодателей.

Платформа *NEBO* *Digital* предлагает современный и удобный интерфейс с акцентом на автоматизацию, что позволяет рекламодателям быстро ориентироваться в каталоге без лишних сложностей. Сервис преимущественно ориентирован на русский язык, с элементами английской версии для международных клиентов (рисунок 1).

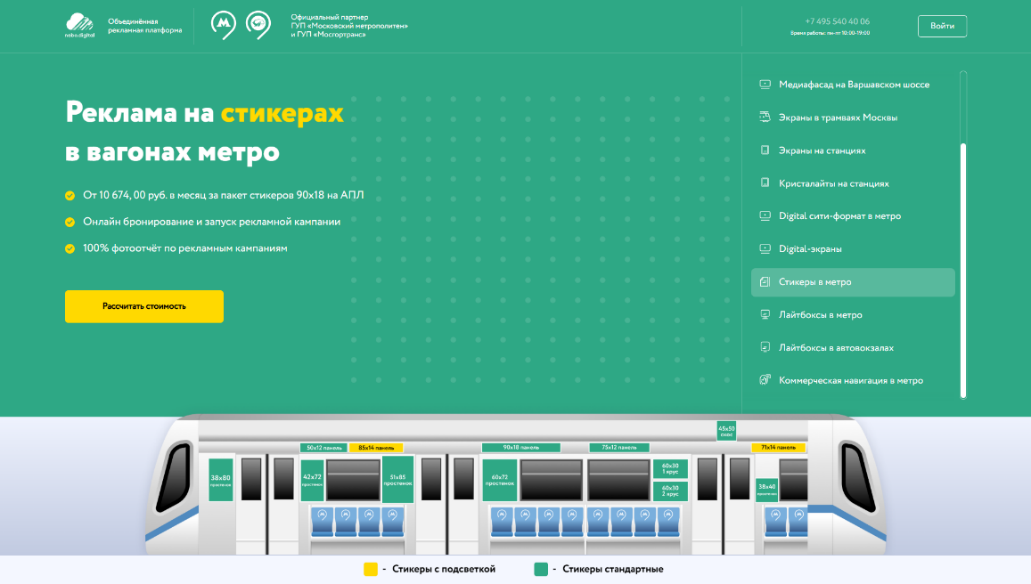


Рисунок 1 – Главная страница *NEBO* *Digital*

Процесс бронирования построен по принципу "одного окна": пользователь выбирает станцию, формат рекламы (стикеры, лайтбоксы, цифровые экраны, брендирование), указывает даты, загружает макет на премодерацию метрополитеном, выбирает типографию, оплачивает онлайн и отслеживает статус в личном кабинете

Для лояльных рекламодателей действует партнерская программа с "Город" (на базе карты "Тройка"), предоставляющая накопительные баллы, скидки, доступ к обезличенной аналитике аудитории и отчетам по эффективности кампаний (рисунок 2).

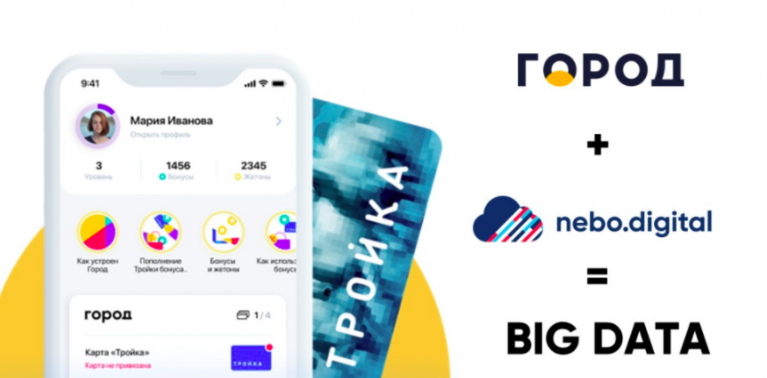


Рисунок 2 – Раздел аналитики и программы лояльности *NEBO* *Digital*

В то же время платформа имеет ограничения: нет полной интеграции с внешними медиапланерами или *programmatic*-системами, фотоотчеты о размещении и детальная статистика доступны только после запуска и оплаты кампании, а в прошлом отмечались жалобы агентств на жесткие условия контрактов и низкую гибкость тарифов.

Сайт «Реклама в метро Санкт-Петербург» предоставляет более широкий спектр возможностей для заказа и размещения рекламных материалов. Он обеспечивает интуитивно понятный процесс выбора рекламных форматов и локаций, а также позволяет подключать дополнительные услуги — разработку креативов, медиапланирование и аналитику эффективности кампаний (рисунок 3).

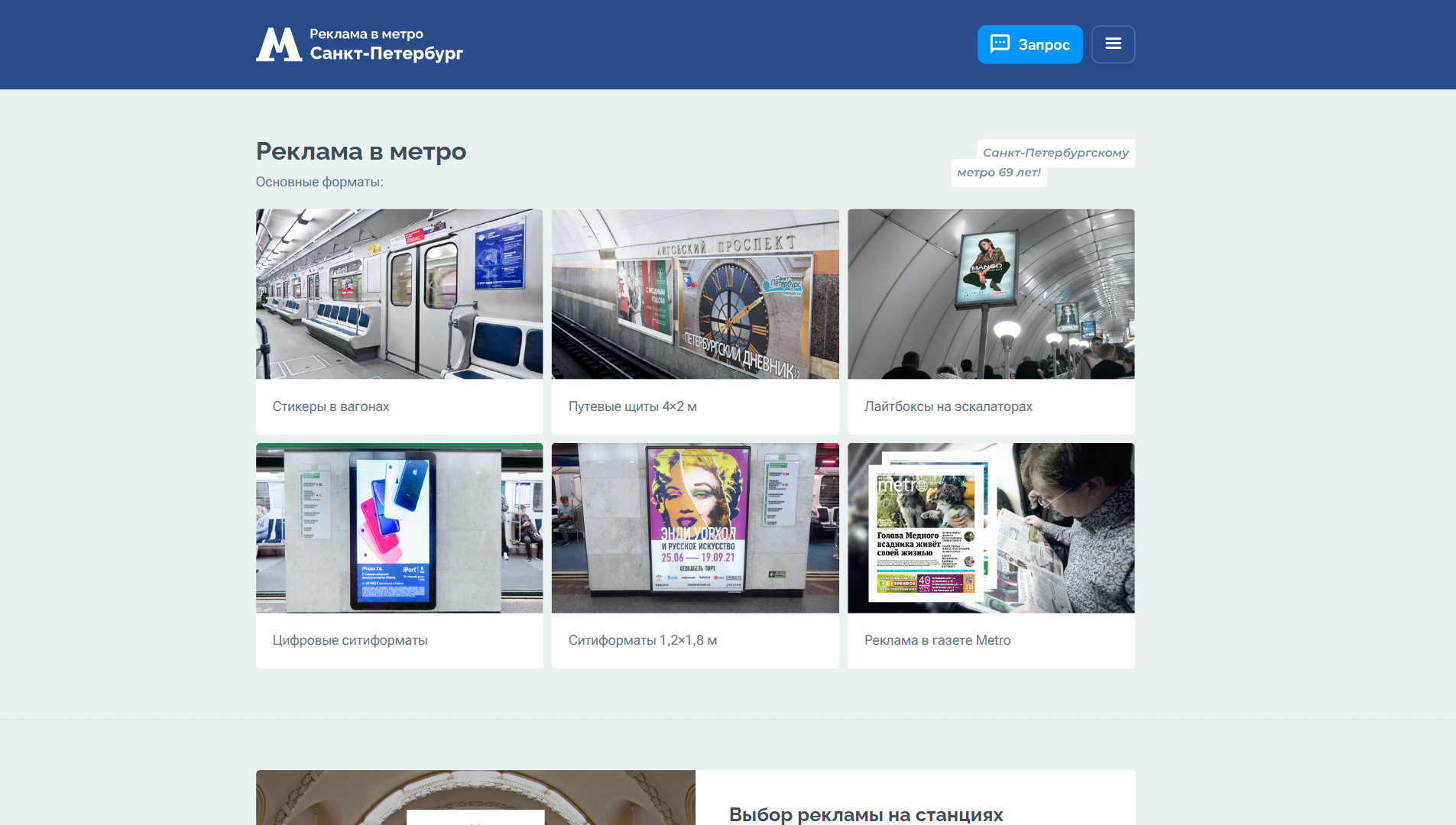


Рисунок 3 – Главная страница *Aeroflot*

При этом сайт «Реклама в метро Санкт-Петербург» имеет ряд недостатков: интерфейс может показаться перегруженным и не всегда удобным для быстрого поиска нужных форматов, мобильная версия ограничена по функционалу, а также периодически фиксируются технические сбои, влияющие на корректность отображения данных о доступности рекламных мест.

Сайт Оскар выделяется современным и профессиональным дизайном, что делает навигацию лёгкой и понятной для клиентов. На ресурсе представлена информация о полном спектре рекламных услуг: размещение рекламы в вагонах и на станциях Минского метро, полиграфия, изготовление макетов и брендинг (рисунок 4).

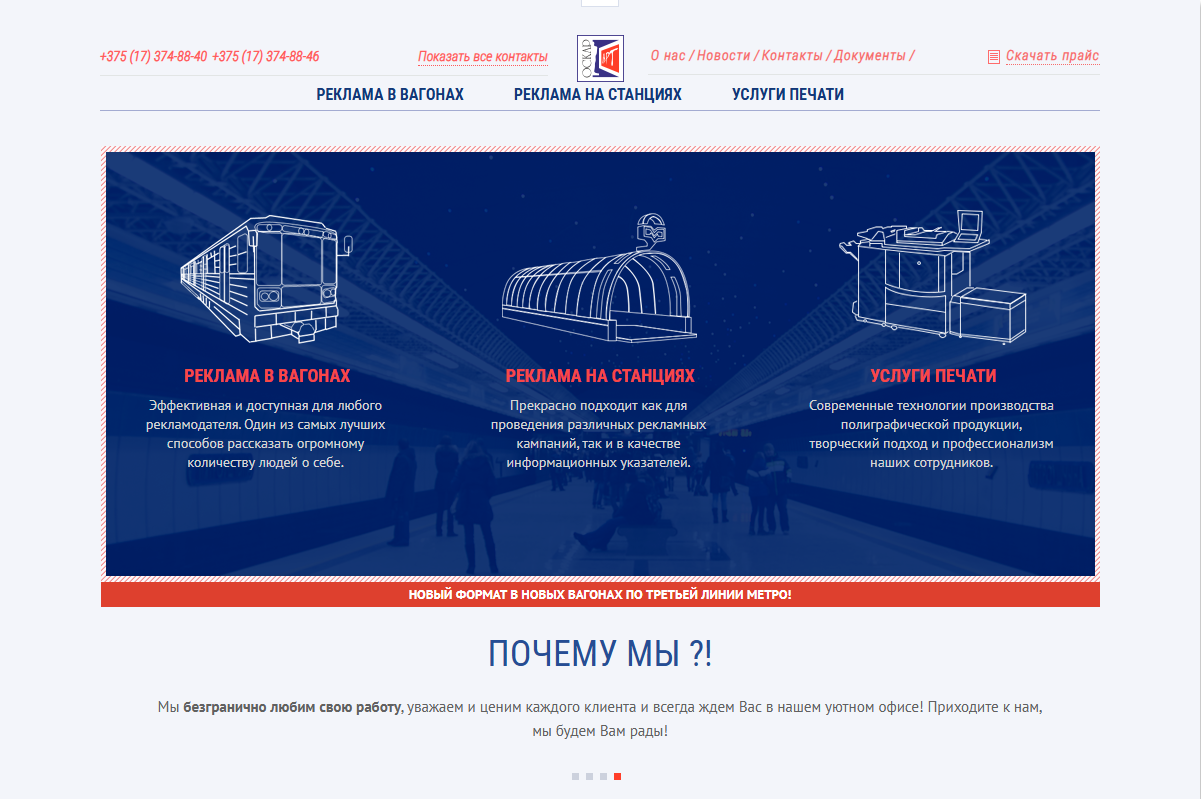


Рисунок 4 – Главная страница Оскар

Однако у компании есть и ограничения: поскольку она специализируется на метро, её охват ограничен этой транспортной системой; в дополнение, недостаточно автоматизированных онлайн-инструментов для самозаказа — часто необходимо взаимодействие с менеджером, что может затруднить оперативную работу для крупных рекламодателей.

Современные сайты Nebo.Digital, Метро Санкт-Петербург и Оскар, несмотря на различия в оформлении, функционале и региональной специфике, во многом схожи по своей структуре и принципам работы. Эти три платформы отражают современный уровень развития информационных систем в сфере наружной и транспортной рекламы и демонстрируют, как интернет стал ключевым инструментом взаимодействия рекламных операторов с клиентами. В основе каждой из систем лежит общая концепция: обеспечение полного цикла обслуживания — от выбора рекламных поверхностей до оформления заявки и получения аналитики по кампании.

Одной из ключевых общих характеристик всех трёх сайтов является наличие личных кабинетов и систем клиентского взаимодействия, которые формируют основу персонализированного обслуживания. Через личный кабинет пользователь получает доступ к своим заявкам, медиапланам, доступным рекламным форматам, договорам и отчетам. На сайтах реализована возможность повторить размещение, запросить исправление макета, продлить рекламную кампанию, а также получить уведомления о сроках публикации и технических требованиях. Все три платформы опираются на современные принципы интерфейсов — структурированность, разделение данных по тематическим блокам и интеграцию с другими цифровыми сервисами, включая мобильные уведомления и электронный документооборот.

Другим важным сходством является наличие систем лояльности и партнёрских предложений. Nebo.Digital предоставляет расширенные возможности для постоянных клиентов в сфере цифровой рекламы, Метро Санкт-Петербург предлагает специальные условия на размещение в разных форматах метрополитена, а Оскар-Арт включает пакетные предложения и скидки на услуги печати и установки. Несмотря на отличия в содержании программ, принцип их работы остаётся аналогичным: клиенты получают преимущества при долгосрочном сотрудничестве, а также доступ к партнерским сервисам и дополнительным услугам — от дизайна макетов до комплексного медиапланирования. Все сайты предоставляют информацию о тарифах, спецпредложениях и пакетных решениях, расширяя возможности рекламодателей.

Особое внимание на всех трёх платформах уделено процессу выбора рекламных поверхностей и оформления заявки. Это реализовано в виде удобных каталогов и интерактивных инструментов, позволяющих выбрать формат, станцию метро, цифровой экран или печатную поверхность. Пользователь может фильтровать варианты по локации, периоду размещения, стоимости и техническим параметрам. Сайты используют современные формы подачи информации — карты, схемы метро, галереи изображений и примеры размещения. После выбора поверхности доступны дополнительные услуги: изготовление макета, печать, монтаж и аналитика эффективности. Все три ресурса обеспечивают оперативную отправку заявки менеджеру и предоставление подтверждающей документации.

Не менее значимым направлением развития этих систем стала служба поддержки клиентов. На сайтах Nebo.Digital, Метро Санкт-Петербург и Оскар-Арт реализованы формы обратной связи, телефонные линии, чаты или системы заявок. Пользователь может быстро уточнить технические требования, согласовать макет или получить консультацию о стоимости размещения. Многие ресурсы внедряют автоматизированные инструменты, такие как чат-боты и формы стандартизированных запросов, что позволяет ускорить обработку обращений и снизить нагрузку на операторов. Кроме того, клиенты получают доступ к актуальной информации о тарифах, регламентах и условиях размещения рекламы.

Таким образом, сайты Nebo.Digital, Метро Санкт-Петербург и Оскар-Арт обладают общим функциональным ядром, отражающим современные тенденции развития цифровых рекламных платформ и систем наружной рекламы. Их сходство объясняется стремлением операторов к унификации процессов, повышению удобства для клиентов и обеспечению прозрачности всех этапов рекламных кампаний. Все три платформы предоставляют пользователям комплексный набор услуг — от выбора рекламных поверхностей до управления заявками, документами и аналитикой.

* 1. **Постановка задачи**

Для создания платформы аренды рекламных поверхностей в метро необходимо использовать современную веб-платформу, обеспечивающую высокую производительность, масштабируемость и безопасность обработки данных пользователей. Основной функционал системы должен включать возможность регистрации и авторизации рекламодателей и администраторов, поиск и бронирование рекламных поверхностей, оплату аренды, а также административный модуль с аналитикой и экспортом данных. Пользовательская часть сайта должна обеспечивать интерактивный интерфейс для выбора станций метро, типов носителей (щиты, световые панели, наклейки), отображения доступности.

В административной панели требуется реализовать функции управления клиентами, бронированиями и платежами, а также систему обработки заявок и модуль формирования аналитических отчётов (включая экспорт в DOCX).

В качестве программной платформы рекомендуется использовать стек технологий Node.js + Express.js для серверной логики, PostgreSQL в качестве реляционной системы управления базами данных с поддержкой транзакций, индексов и внешних ключей, а также EJS (Embedded JavaScript Templates) или React.js для рендеринга динамических страниц. Для безопасного хранения паролей применяется библиотека bcrypt.js, для генерации отчётов – пакет docx. Подключение к базе данных осуществляется через модуль pg, с использованием пула подключений и SSL-шифрования при необходимости. Обеспечивается модульная структура проекта: отдельные модули для работы с клиентами, рекламными поверхностями, бронированиями, оплатами и аналитикой.

Таким образом, создаётся надёжная, функциональная и масштабируемая веб-платформа для управления арендой рекламных поверхностей в метро и клиентскими бронированиями.

# **ПРОЕКТИРОВАНИЕ СИСТЕМЫ**

* 1. **Проектирование базы данных**

Разрабатываемая база данных авиакомпании предназначена для хранения информации о пользователях, станциях, линиях метро, поездов,

Логическая структура построена на основе модели «сущность-связь» (*ER*-модели) [6].

Каждая таблица базы данных представляет собой отдельную сущность, содержащую атрибуты – характеристики объекта, а между сущностями устанавливаются отношения (связи), описывающие их взаимодействие [7].

Связи между объектами описаны с помощью *ER*-диаграммы в нотации Питера Чена (см. рисунок 8).

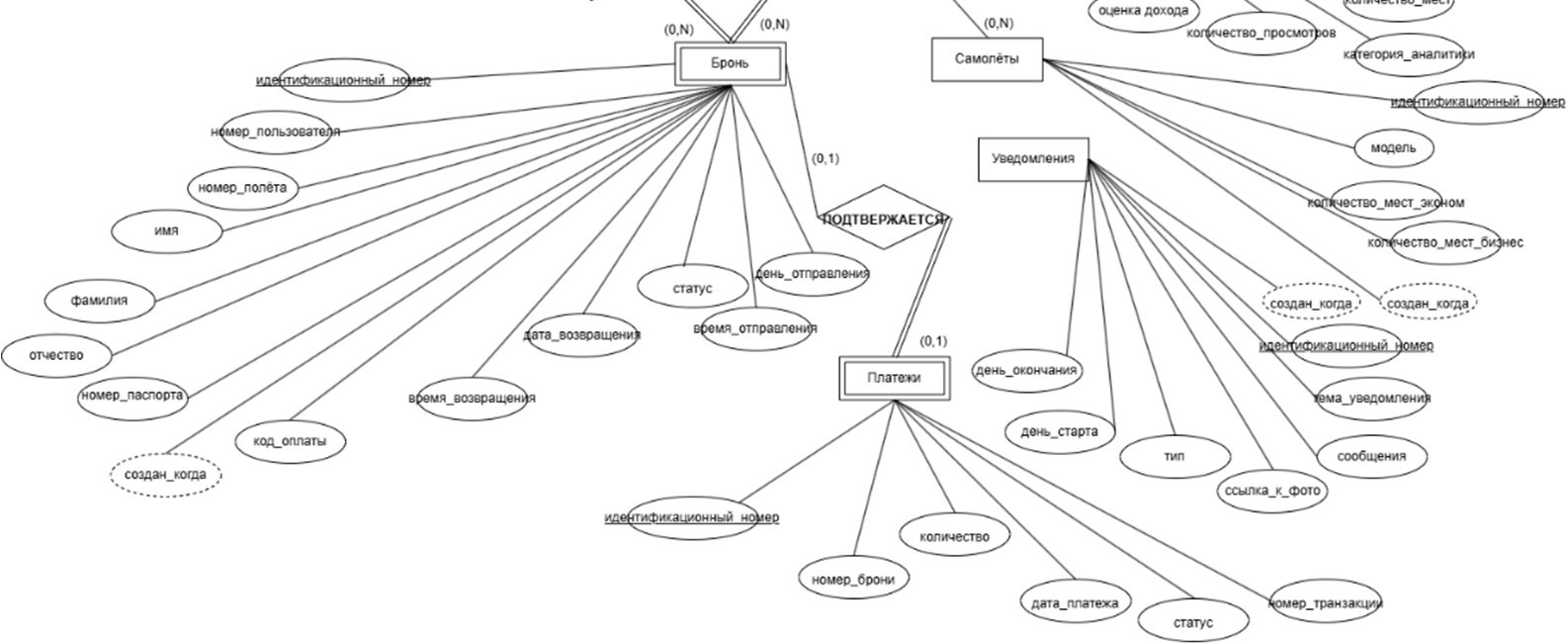


Рисунок 8 – Диаграмма в нотации Питера Чена

Сущность «пользователи» содержит основную информацию о зарегистрированных клиентах системы, которые совершают бронирование билетов, оплачивают полёты и взаимодействуют с системой поддержки.

Атрибуты таблицы «пользователи»:

* идентификационный номер – уникальный ключ (первичный

идентификатор) пользователя;

* имя\_пользователя – уникальное имя (логин), используемое для входа в систему;
* хэш\_пароля – зашифрованный пароль пользователя;
* электронная\_почта (*email*) – адрес электронной почты, служащий для связи и восстановления доступа;
* токен\_повторный – уникальный токен, генерируемый при сбросе пароля;
* истечение\_входа – дата и время окончания действия сессии или токена;
* создан\_когда – дата регистрации пользователя в системе; – обновлён\_когда – дата последнего изменения данных.
* Связи:
* Один пользователь может создавать множество бронирований (1:*N*);
* Один пользователь может отправлять множество сообщений в поддержку (1:*N*).
* Сущность «полёты» отражает информацию обо всех рейсах авиакомпании – как внутренних, так и международных.
* Атрибуты:
* идентификационный\_номер – уникальный номер рейса;
* номер\_самолёта – ссылка на конкретный самолёт, выполняющий
* рейс;
* город\_отправления – пункт вылета;
* город\_прибытия – пункт назначения;
* дата\_возвращения – дата обратного рейса;
* время\_отправления – время вылета;
* время\_возвращения – время обратного вылета;
* класс\_полёта – категория обслуживания (эконом, бизнес);
* цена – стоимость билета на данный рейс;
* количество\_мест – общее количество мест на рейсе;
* количество\_просмотров – число обращений к рейсу; – оценка\_дохода – прогнозируемая прибыль от рейса; – создан\_когда – дата внесения рейса в систему.

Связи:

* Один полёт выполняется одним самолётом (1:1);
* Один полёт может иметь множество бронирований (1:*N*).

*C*ущность «самолёты» используется для быстрого создания рейса, указания мест как мест бизнес класса, так и эконома.

Атрибуты:

* идентификационный\_номер – уникальный номер самолёта;
* модель – название модели (например, *Boeing* 737, *Airbus* *A*320);
* количество\_мест\_эконом – число мест в экономическом классе; – количество\_мест\_бизнес – число мест в бизнес-классе; – создан\_когда – дата добавления самолёта в систему.
* Связи
* Один самолёт может выполнять множество полётов (1:*N*).

Сущность «бронь» – одна из центральных таблиц системы, связывающая пользователей и рейсы. Содержит информацию о бронировании билета, времени, статусе и оплате.

Атрибуты:

* идентификационный\_номер – уникальный номер брони;
* номер\_пользователя – ссылка на пользователя, совершившего бронирование;
* номер\_полёта – ссылка на рейс, для которого оформлено бронирование;
* имя, фамилия, отчество – личные данные пассажира;
* номер\_паспорта – документ, удостоверяющий личность;
* дата\_отправления, время\_отправления – планируемая дата и время вылета;
* дата\_возвращения, время\_возвращения – данные о дате и времени обратного рейса;
* код\_оплаты – внутренний код, связывающий бронь с платёжной системой;
* статус – состояние бронирования (активно, ожидает оплаты, отменено);
* создан\_когда – дата оформления брони.
* Связи:
* Одна бронь принадлежит одному пользователю и одному полёту
* (*M*:*N* через связь «бронирует»);
* Одна бронь может иметь одну запись оплаты (1:1).

Сущность «платежи» отражает процесс финансовых транзакций, связанных с оплатой бронирований, взаимодействие банковской системы происходит на уровне этой сущности. Имеет наибольшую важность при непосредственно покупке билетов.

Атрибуты:

* идентификационный\_номер – уникальный номер платежа;
* номер\_брони – ссылка на бронь, которую подтверждает платёж;
* количество – сумма платежа;
* дата\_платежа – дата и время проведения операции; – статус – текущее состояние (ожидает, успешно, отменён); – номер\_транзакции – уникальный код платёжной операции.
* Связи:
* Один платёж подтверждает одно бронирование (1:1).

Сущность «поддержка\_билеты» (обращения в службу поддержки) содержит сведения о сообщениях и запросах пользователей, направленных в службу поддержки авиакомпании.

Атрибуты:

* идентификационный\_номер – уникальный номер обращения;
* номер\_пользователя – ссылка на пользователя, отправившего сообщение;
* имя – имя заявителя;
* электронная\_почта – контактный *e*-*mail*;
* сообщение – текст обращения;
* статус – состояние обработки запроса (новое, решено, закрыто);
* создан\_когда – дата отправки сообщения;
* ответ – текст ответа службы поддержки;
* время\_ответа – дата и время предоставления ответа.
* Связи:
* Один пользователь может отправить множество сообщений в поддержку (1:*N*).

Сущность «уведомления» используется для публикации информационных и рекламных сообщений на сайте авиакомпании.

Атрибуты:

* идентификационный\_номер – уникальный номер уведомления;
* тема\_уведомления – заголовок сообщения; – сообщение – основной текст уведомления; – тип – категория уведомления.
* ссылка\_к\_фото – путь к изображению уведомления.
* день\_старта – дата начала публикации;
* день\_окончания – дата завершения показа уведомления;
* статус – активно или нет;
* создан\_когда – дата добавления уведомления.

Сущность не связана напрямую с пользователями, но используется в интерфейсе сайта для информирования всех клиентов.

База данных проектируется с учетом принципов нормализации, что позволяет избежать избыточности и аномалий при обновлениях:

* первая нормальная форма – все атрибуты атомарны, отсутствуют повторяющиеся группы.
* вторая нормальная форма – каждый неключевой атрибут полностью зависит от первичного ключа.
* третья нормальная форма – отсутствует транзитивная зависимость между неключевыми полями (например, цена не зависит от имени пользователя).

Логическая схема базы данных авиакомпании (см. рисунок 9) отражает взаимосвязь между основными сущностями, участвующими в функционировании информационной системы [8]. Схема построена на принципах реляционной модели данных и демонстрирует логические связи между таблицами, обеспечивающими хранение информации о пользователях, рейсах, самолётах, бронированиях, оплатах, уведомлениях и обращениях в службу поддержки. Каждая сущность на схеме представляет собой отдельную таблицу, обладающую собственным набором атрибутов и уникальным идентификатором.

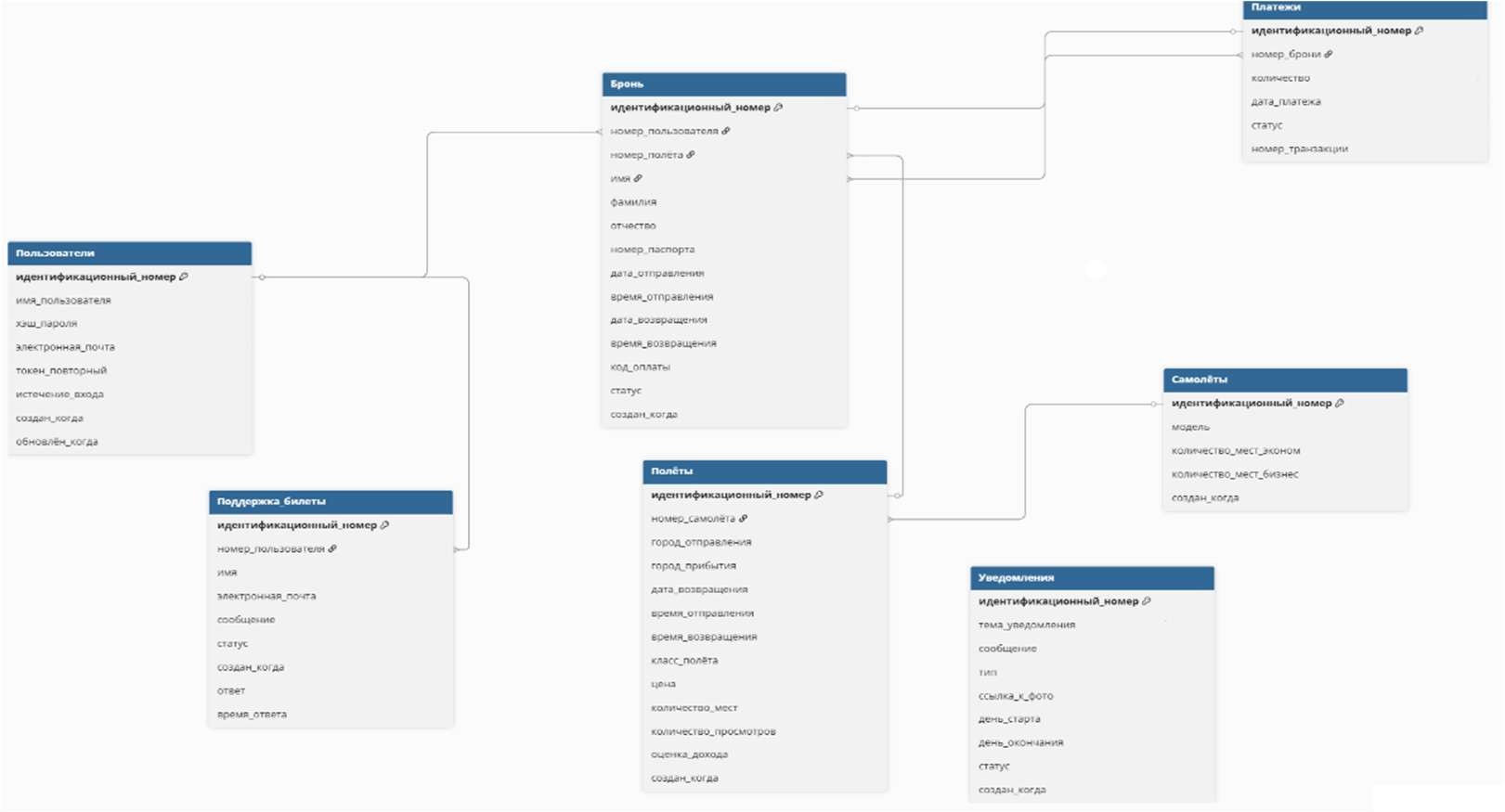


Рисунок 9 – Логическая схема базы данных

Представленная логическая модель обеспечивает основу для построения реляционной базы данных авиакомпании.

* 1. **Проектирование сайта**

Диаграмма использования (*Use* *Case* *Diagram*) предназначена для визуализации функциональных требований к разрабатываемой системе с точки зрения взаимодействия пользователей (актеров) с системой. Она отображает ключевые сценарии использования, определяя, какие функции доступны различным категориям пользователей, и позволяет выявить основные процессы, поддерживаемые системой. Диаграмма использования обеспечивает наглядное представление границ системы, взаимосвязей между актерами и прецедентами, а также служит инструментом для согласования требований между заказчиком и разработчиком, способствуя минимизации недопониманий и ошибок на этапах проектирования и анализа.

Создадим диаграмму использования для вебсайта (см. рисунок 10), а именно процедуру покупки билета.

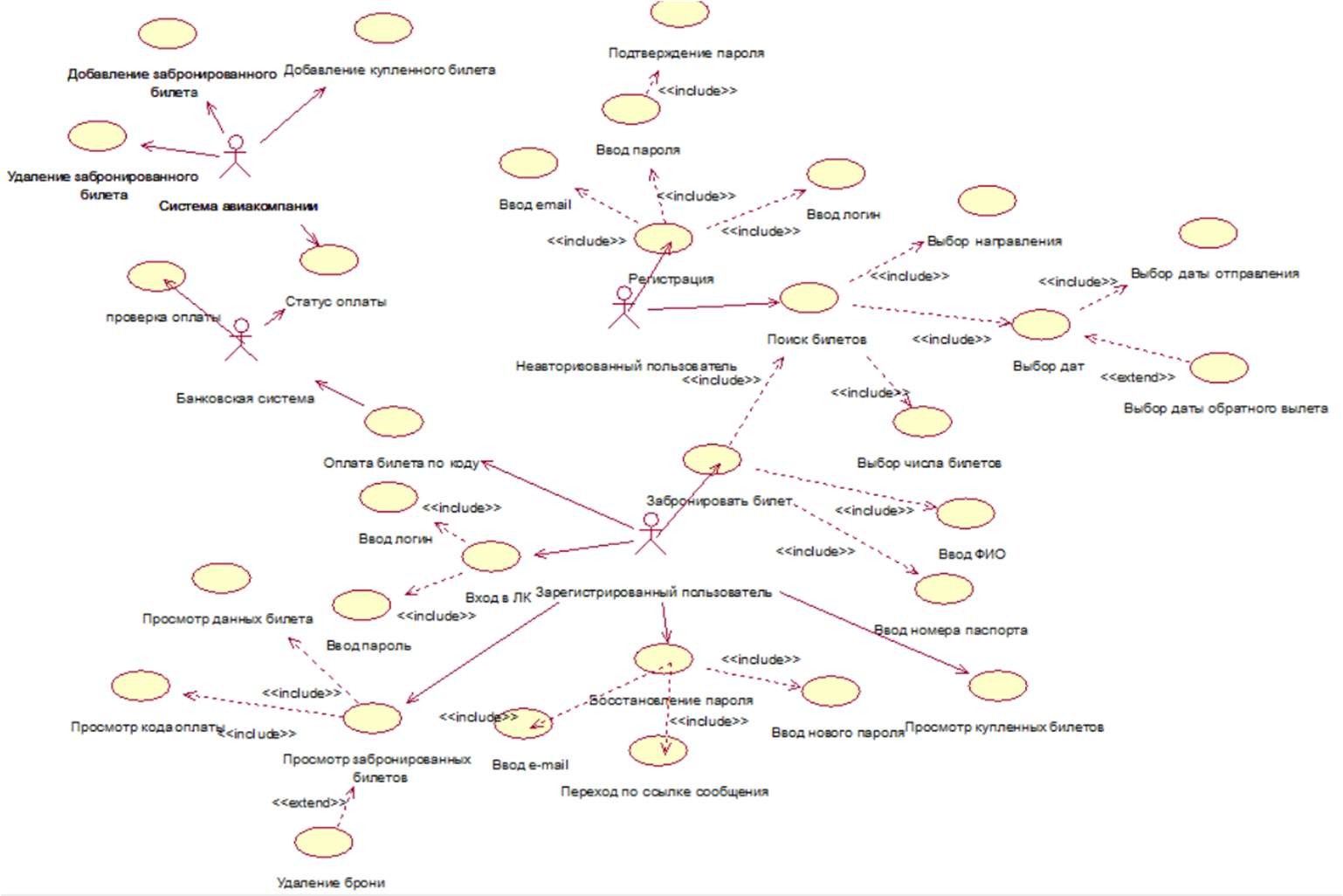


Рисунок 10 – Диаграмма варианта использования

Распишем вариант использования по диаграмме.

Вариант использования «Регистрация».

Краткое описание: данный вариант использования позволяет неаутентифицированному пользователю создать новую учетную запись в системе.

Основной поток событий:

* пользователь инициирует процесс регистрации;
* система запрашивает логин, пароль и его подтверждение;
* пользователь вводит логин, пароль и подтверждает пароль;
* система проверяет корректность данных и создает новую учетную запись;
* система подтверждает успешную регистрацию.

Альтернативный поток событий: если введенные данные некорректны (например, логин уже существует или пароли не совпадают), система выводит сообщение об ошибке и предлагает повторить ввод.

Предусловия: пользователь не аутентифицирован в системе.

Постусловия: создана новая учетная запись, пользователь становится зарегистрированным.

Вариант использования «Вход в личный кабинет».

Краткое описание: данный вариант использования позволяет зарегистрированному пользователю аутентифицироваться в системе для доступа к личному кабинету.

Основной поток событий:

* пользователь инициирует процесс входа в личный кабинет;
* система запрашивает логин и пароль;
* пользователь вводит логин и пароль;
* система проверяет корректность учетных данных; – система предоставляет доступ к личному кабинету.

Альтернативный поток событий: если введенные учетные данные неверны, система выводит сообщение об ошибке и предлагает повторить попытку.

Предусловия: пользователь имеет зарегистрированную учетную запись.

Постусловия: пользователь аутентифицирован и имеет доступ к функциям личного кабинета.

Вариант использования «Поиск билетов».

Краткое описание: данный вариант использования позволяет пользователю найти подходящие авиабилеты по заданным критериям.

Основной поток событий:

* пользователь инициирует поиск билетов;
* система запрашивает критерии поиска: направление, дату отправления;
* пользователь вводит направление, выбирает дату отправления и при необходимости возврата;
* система выполняет поиск в системе авиакомпании и отображает найденные варианты.

Альтернативный поток событий: если по заданным критериям билеты не найдены, система информирует пользователя об этом.

Предусловия: отсутствуют.

Постусловия: пользователь получил список доступных рейсов.

Вариант использования «Забронировать билет».

Краткое описание: данный вариант использования позволяет пользователю забронировать выбранный билет.

Основной поток событий:

* пользователь выбирает опцию «забронировать билет» для
* выбранного рейса;
* система запрашивает необходимые данные: имя, фамилия, отчество и номер паспорта;
* пользователь вводит фамилию, имя, отчество и номер паспорта;
* система добавляет билет в статусе «забронированный» и подтверждает бронирование.

Предусловия: пользователь выполнил поиск билетов и выбрал конкретный рейс.

Постусловия: билет добавлен в систему со статусом

«забронированный».

Вариант использования «Просмотр забронированных билетов».

Краткое описание: данный вариант использования позволяет аутентифицированному пользователю просмотреть список своих забронированных билетов.

Основной поток событий:

* пользователь входит в личный кабинет;
* пользователь выбирает опцию «просмотр забронированных билетов»; – система отображает список всех бронирований пользователя.

Предусловия: пользователь аутентифицирован в системе.

Постусловия: пользователь видит список своих забронированных билетов.

Вариант использования «Просмотр кода оплаты».

Краткое описание: данный вариант использования позволяет аутентифицированному пользователю получить код для оплаты забронированного билета.

Основной поток событий:

* пользователь просматривает детали забронированного билета;
* пользователь выбирает опцию «просмотр кода оплаты»;
* система отображает уникальный код для проведения платежа.

Предусловия: пользователь аутентифицирован и имеет активные бронирования.

Постусловия: пользователь получил код для оплаты билета.

Вариант использования «Просмотр данных билета».

Краткое описание: данный вариант использования позволяет аутентифицированному пользователю просмотреть детальную информацию о билете.

Основной поток событий:

* пользователь просматривает список своих билетов (забронированных или купленных);
* пользователь выбирает конкретный билет для просмотра деталей; – система отображает подробную информацию о выбранном билете.

Предусловия: пользователь аутентифицирован в системе.

Постусловия: пользователь получил детальную информацию о билете.

Вариант использования «Оплата билета по коду».

Краткое описание: данный вариант использования позволяет пользователю произвести оплату забронированного билета через банковскую систему.

Основной поток событий:

* пользователь инициирует оплату билета по коду;
* система перенаправляет запрос в банковскую систему;
* банковская система проверяет платеж и возвращает статус оплаты; – система, получив подтверждение оплаты, изменяет статус билета с «забронированного» на «купленный» и добавляет купленный билет.

Альтернативный поток событий: если оплата не прошла, банковская система возвращает статус ошибки, и билет остается в статусе «забронированный».

Предусловия: пользователь имеет код оплаты для забронированного билета.

Постусловия: билет оплачен и добавлен как купленный.

Далее создадим диаграмму классов. Диаграмма классов является ключевым объектом *UML*-моделирования, который описывает статическую структуру системы. Она служит для визуализации основных сущностей (классов), их атрибутов (свойств данных), операций (методов) и отношений между ними (ассоциации, композиции, агрегации).

Диаграмма построена на основе логической схемы базы данных, где таблицы преобразованы в классы, столбцы в атрибуты, а внешние ключи в ассоциации (см. рисунок 11).

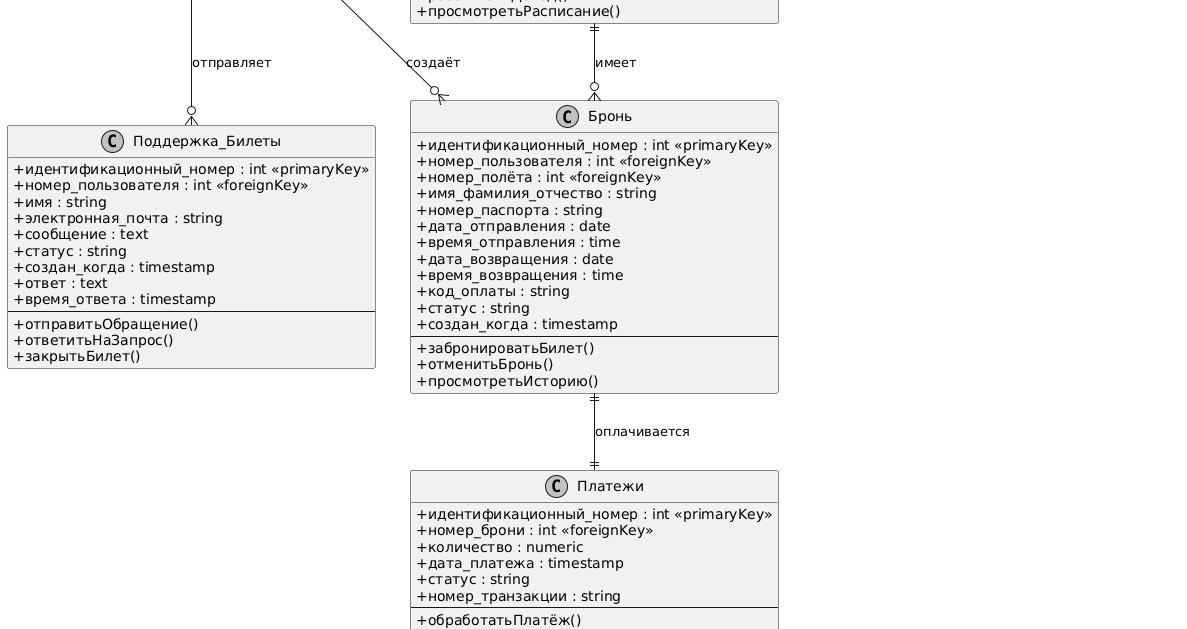
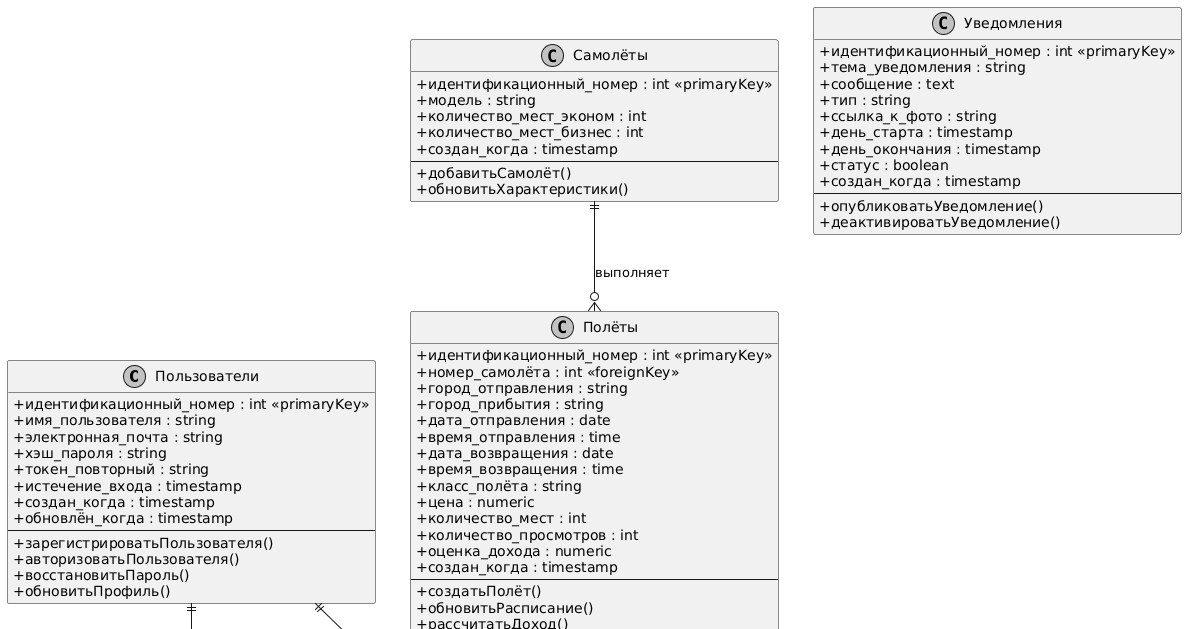


Рисунок 11 – Диаграмма классов базы данных

Подробно разберём все операции, определённые в диаграмме классов.

Класс «пользователи»: операции для аутентификации и профиля. зарегистрироватьПользователя() создаёт учётную запись по *username*, *email* и паролю; результат: *ID* пользователя или ошибка дубликата; авторизоватьПользователя() проверяет логин/пароль, выдаёт *JWT*-токен; результат: токен или отказ; восстановитьПароль() генерирует токен по *email*, отправляет ссылку; результат: подтверждение; обновитьПрофиль() меняет данные (*email*, телефон); результат: обновлённый профиль.

Класс «Полёты»: управление рейсами. создатьПолёт() добавляет рейс по маршруту, датам, цене; результат: *ID* рейса; обновитьРасписание() корректирует даты/время; результат: подтверждение; рассчитатьДоход() оценивает прибыль по броням; результат: сумма; аналитика. просмотретьРасписание() ищет рейсы по фильтрам; результат: список рейсов.

Класс «Самолёты»: каталог воздушных судов. добавитьСамолёт() регистрирует модель и количество мест; результат: *ID* самолёта; обновитьХарактеристики() корректирует места в классах; результат: подтверждение изменений.

Класс «Бронь»: бронирование билетов. забронироватьБилет() создаёт бронь по пользователю, рейсу и данным пассажира; результат: код брони; отменитьБронь() аннулирует, освобождает места; результат: статус отмены; просмотретьИсторию() выдаёт список броней по фильтрам; результат: массив записей.

Класс «Платежи»: обработка транзакций. обработатьПлатёж() подтверждает оплату по брони и сумме; результат: электронный билет;

Класс «Поддержка\_Билеты»: клиентская поддержка. отправитьОбращение() формирует билет по *email* и сообщению; результат: номер обращения; ответитьНаЗапрос() добавляет ответ от агента; результат: обновлённый статус; закрытьБилет() завершает обработку; результат: статус *closed*.

Класс «Уведомления»: отображение всех системных сообщений. опубликоватьУведомление() активирует по теме, тексту и периоду; результат: *ID* уведомления; деактивироватьУведомление() скрывает по *ID*; результат: статус неактивен.

Схема алгоритма в приложение А представляет собой компактную визуализацию основного сценария взаимодействия пользователя с вебсистемой бронирования авиабилетов. Это ветвящаяся последовательность шагов, ориентированная сверху вниз, где прямоугольники обозначают действия, а ромбы – ключевые решения. Дизайн подчёркивает удобство: синие тона для инициации, зелёные для успеха, красные для ошибок, с жирным шрифтом и переносами строк для читаемости. Алгоритм интегрирует *use* *cases* проекта, обеспечивая плавный пользовательский путь от входа до подтверждения транзакции, с акцентом на безопасность и обработку отказов.

Процесс стартует с момента, когда пользователь заходит на сайт, где система сразу проверяет наличие регистрации. Если пользователь уже зарегистрирован, следует быстрая авторизация по логину и паролю с генерацией сессионного токена. В противном случае активируется комбинированный шаг регистрации и авторизации, где вводятся базовые данные (*email*, пароль), за которыми следует верификация. Оба варианта сливаются в поиск билета, где указываются тип рейса, направление, класс обслуживания и дата – это запускает запрос к базе данных для отбора доступных вариантов.

Далее пользователь вводит данные (фамилия, имя, отчество, номер документа) для идентификации пассажира, после чего переходит в личный кабинет для обзора профиля и истории. Здесь отображается сгенерированный код платежа, связанный с предварительной бронью: если оплата успешна, статус обновляется на «Оплачен», генерируется электронный билет. При неудаче статус меняется на «Не оплачен».

# **РАЗРАБОТКА ИНФОРМАЦИОННОЙ СИСТЕМЫ АВИАКОМПАНИИ**

* 1. **Выбор средств реализации**

В рамках курсового проекта по разработке системы авиакомпании выбор средств реализации играет ключевую роль в обеспечении эффективности, масштабируемости и удобства разработки. Проанализируем конкурентные решения для систем управления базами данных (СУБД) и языков программирования (ЯП), с последующим обоснованием выбранных технологий.

Начнём с анализа альтернатив для СУБД. Реляционные базы данных предпочтительны для проектов с предопределённой схемой, как в нашем случае, где сущности взаимосвязаны: пользователи связаны с бронями, рейсы с самолётами, а уведомления – с общими настройками. *MySQL*, один из наиболее распространённых вариантов, выделяется скоростью выполнения простых запросов и удобством администрирования, что делает его подходящим для веб-приложений начального уровня. В проекте *MySQL* мог бы хранить данные пользователей с уникальными индексами на *email* и *username*, а также обеспечивать внешние ключи для автоматического управления связями между бронями и рейсами.

Однако его ограничения в работе с расширенными типами данных, такими как временные метки с часовыми поясами для расписаний или массивы для категорий уведомлений, потребовали бы дополнительных настроек, усложняя процесс миграций.

*SQLite* представляет собой лёгкую альтернативу для этапа прототипирования, с нулевыми требованиями к серверу и возможностью быстрого тестирования схем таблиц локально. Это позволило бы проверить реляционные связи между рейсами и бронями без внешней инфраструктуры, но отсутствие полноценной поддержки параллельного доступа сделало бы его неподходящим для симуляции множественных поисков, где важна одновременная обработка запросов от нескольких пользователей.

В категории *NoSQL* *MongoDB* привлекает гибкостью для динамических данных, таких как записи в поддержку или списки уведомлений, где документы могут эволюционировать без строгой схемы. Это упростило бы добавление новых полей для описания рейсов, например, категории аналитики, без полной перестройки базы. Тем не менее, для операций, требующих соединений между пользователями и их бронями в отчётах личного кабинета, *MongoDB* опирался бы на агрегации, что увеличило бы сложность запросов и время выполнения по сравнению с классическим *SQL*.

На фоне конкурентов выбрана *PostgreSQL* как реляционная СУБД, реализованная через платформу *Supabase* для оптимизации развертывания и обслуживания. *PostgreSQL* полностью соответствует задачам проекта благодаря строгому соблюдению *SQL*-стандартов, что гарантирует целостность связей: внешние ключи между таблицами пользователей и броней обеспечивают автоматическое обновление данных, минимизируя риски несогласованности. Расширенные типы, такие как *NUMERIC* для расчёта стоимости рейсов и *TIMESTAMP* *WITH* *TIME* *ZONE* для точного фиксации дат создания записей, поддерживают глобальную актуальность информации. Индексация ключевых полей, например, по маршрутам и датам в таблице рейсов, ускоряет поисковые операции, обеспечивая отклик в миллисекунды даже при росте объёма.

*Supabase*, как облачная платформа на базе *PostgreSQL*, усиливает преимущества: встроенная система аутентификации с *JWT*-токенами упрощает управление входами пользователей. Функция *Realtime* для подписок на обновления в таблицах поддержки обеспечивает мгновенные оповещения о новых записях, а *Edge* *Functions* позволяют запускать кастомные скрипты, такие как автоматический подсчёт просмотров рейсов, без необходимости в отдельной серверной среде. По сравнению с самостоятельным *PostgreSQL*, *Supabase* снижает нагрузку на администрирование, позволяя сосредоточиться на логике приложения, например, на фильтрации данных прямо в личном кабинете администратора (рисунок 12).

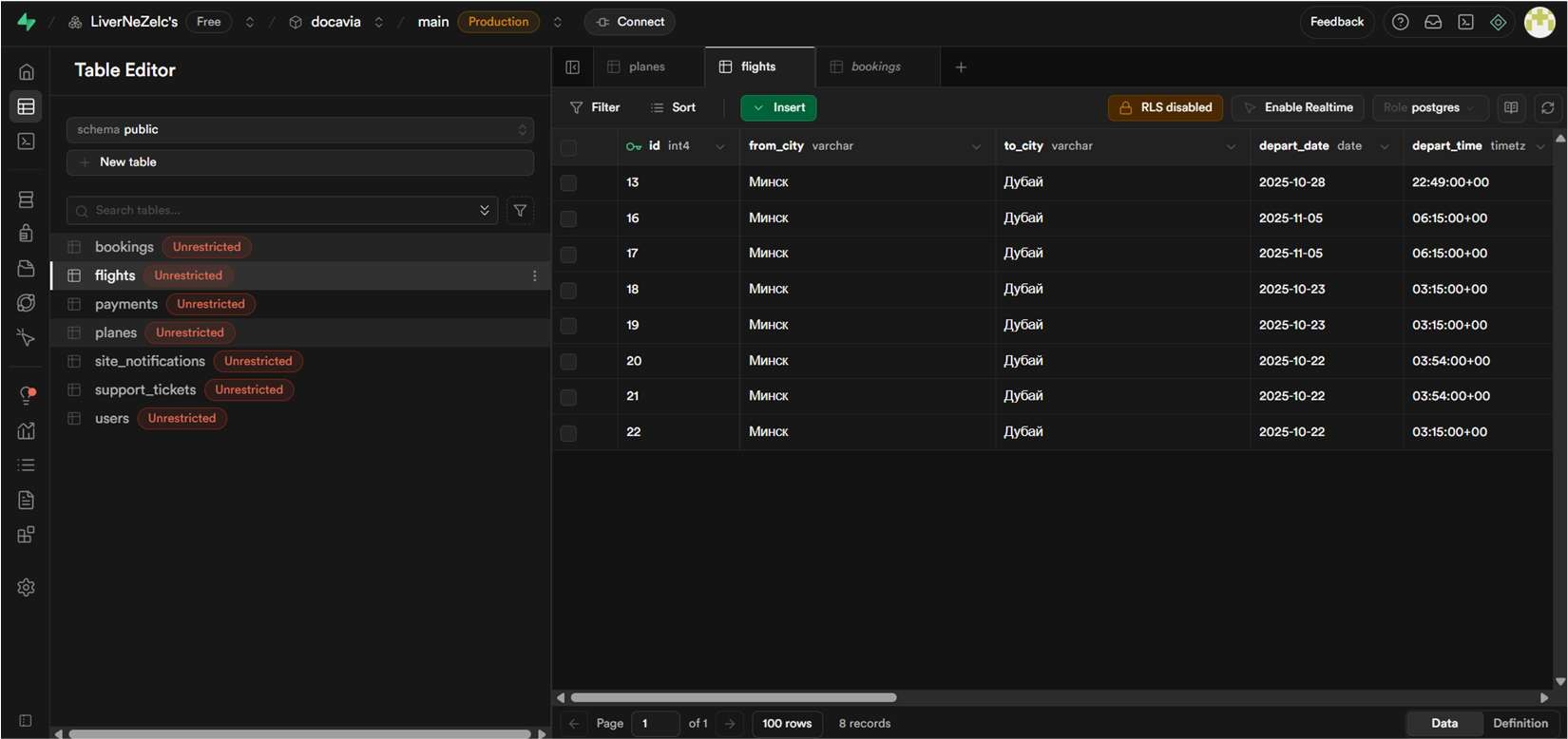


Рисунок 12 – Работа с таблицами

Для аналитических задач, таких как агрегация данных о популярности рейсов или категоризация пользователей, *PostgreSQL* предоставляет мощные агрегатные функции и возможность подключения расширений, что открывает путь к будущим улучшениям, как временные ряды для тенденций бронирований.

Теперь перейдём к альтернативам языков программирования.

*Python* остаётся одним из самых доступных вариантов для разработки благодаря своей выразительности и обширной экосистеме инструментов для обработки данных. В проекте *Python* мог бы эффективно справляться с задачами поиска и фильтрации рейсов, используя встроенные структуры для манипуляции списками результатов и асинхронные возможности для параллельного выполнения запросов. Однако его общая ориентация на последовательные операции могла бы привести к задержкам в сценариях с высокой интенсивностью ввода-вывода, таких как одновременный просмотр нескольких профилей или обновлений уведомлений, требуя дополнительных усилий для оптимизации.

*PHP*, в свою очередь, традиционно используется для серверной логики веб приложений, предлагая простой синтаксис для обработки форм и сессий. Это сделало бы его подходящим для базового управления данными пользователей и броней, с лёгкостью в реализации циклов для проверки доступности рейсов. Тем не менее, *PHP* часто сталкивается с ограничениями в асинхронной обработке, что усложнило бы реальное время обновлений в личном кабинете, где нужно мгновенно отображать изменения в статусе броней.

*Java*, как объектно-ориентированный язык, обеспечивает строгую типизацию и мощные инструменты для модульного проектирования, что позволило бы структурировать код вокруг классов для сущностей вроде рейсов и пользователей. *Java* подходит для сложных расчётов, таких как сортировка результатов поиска по цене или дате, с гарантией стабильности в многопоточной среде. Однако его необходимость в компиляции могли бы усложнить быстрый деплой и тестирование на разных окружениях.

Предпочтение отдано *JavaScript* с *Node*.*js* как средой выполнения, что обеспечивает единый язык для клиентской и серверной разработки, упрощая общий объём работы.

*Node*.*js*, основанный на высокопроизводительном ядре, что идеально для задач с интенсивным обменом данными: параллельный поиск рейсов по различным параметрам или обновление списка уведомлений происходит без задержек. Это позволяет создавать лёгкие *API* для обработки запросов на просмотр броней, где асинхронные вызовы последовательно загружают связанные данные о рейсах и пользователях.

Экосистема *JavaScript* предлагает обширный набор готовых решений для типичных нужд проекта: от валидации входных данных при регистрации до генерации уникальных кодов для записей в поддержке (рисунок 13). *Node*.*js* легко расширяется для симуляции нагрузок, что полезно для тестирования поиска по популярным маршрутам.

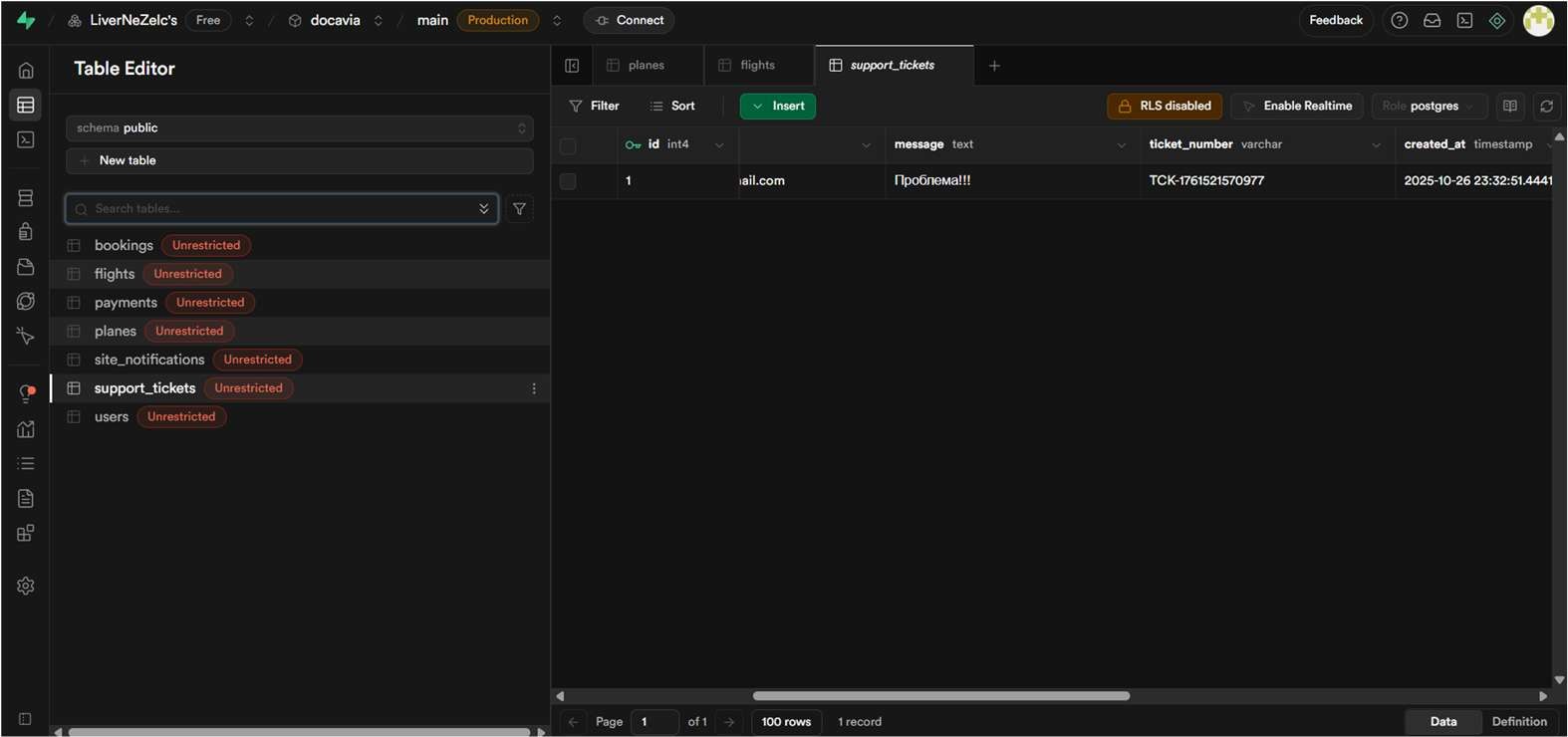


Рисунок 13 – Отображение реального времени

Обоснование выбора: по сравнению с *Python*, фокусирующимся на анализе данных, или *PHP* с его традиционным серверным уклоном, *JavaScript* с *Node*.*js* повышает скорость разработки за счёт знакомого синтаксиса для веб-разработчиков, обеспечивая лучшую отзывчивость в сценариях с множественными запросами.

Для динамических элементов, таких как реальное время отображения изменений в таблице поддержки, *Node*.*js* позволяет реализовать подписки без постоянных проверок, интегрируясь напрямую с механизмами *Supabase*. В проекте это проявляется в личном кабинете, где запросы возвращают объединённые данные о бронях с фильтрацией по пользователю, делая интерфейс более интерактивным.

Для операций поиска *Node*.*js*-логика фильтрует результаты по индексам базы, минимизируя время ожидания. Масштабируемость заложена в дизайн: *Supabase* автоматически адаптирует ресурсы под нагрузку, а *Node*.*js* поддерживает горизонтальное расширение через простые конфигурации.

Комбинация *Node*.*js* и *PostgreSQL* демонстрирует повышенную пропускную способность для типичных *API*-запросов по сравнению с традиционными альтернативами. В рамках курсового проекта это даёт возможность продемонстрировать полный цикл от проектирования схемы данных до создания конечного продукта, с акцентом на проверку покрытия кода. Перспективы развития включают миграцию на расширенные облачные сервисы для *PostgreSQL* при необходимости или внедрение дополнительных протоколов обмена данными в *Node*.*js* для более гибких запросов.

Итоговый стек оптимален по критериям удобства, производительности и адаптивности, обеспечивая эффективную реализацию системы бронирования авиабилетов.

Выбор средств реализации для системы бронирования авиабилетов в рамках курсового проекта стал определяющим фактором, позволив гармонично объединить реляционную мощь *PostgreSQL* через *Supabase* с асинхронной гибкостью *JavaScript* и *Node*.*js*. Этот стек не только удовлетворил ключевые требования к хранению и обработке данных – от целостных связей между пользователями и бронями до динамических обновлений уведомлений, – но и обеспечил баланс между производительностью и простотой внедрения. *PostgreSQL* с его расширенными возможностями индексации и типизацией данных гарантирует быстрый поиск рейсов и точное управление расписаниями, в то время как *Supabase* добавляет слой удобства: встроенная аутентификация и реальное время обновлений делают систему готовой к интерактивным сценариям, таким как просмотр личного кабинета без перезагрузок. *Node*.*js*, в свою очередь, усиливает фронтенд-бэкенд единством, где асинхронные *API* минимизируют задержки в запросах, а обширная экосистема упрощает интеграцию готовых модулей для валидации и генерации контента.

Такой подход не только повышает эффективность реализации, но и закладывает основу для будущих расширений, таких как добавление нового функционала для персонала или мультиязычной поддержки. В итоге, выбранные технологии обеспечивают не только техническую надёжность, но и образовательную ценность, иллюстрируя современные практики *full*-*stack* разработки в реальном проекте. Это позволяет перейти к следующим этапам с полученной программной базой, где акцент сместится на детальную реализацию.

* 1. **Реализация базы данных**

Опишем базу данных информационной системы, используя физическую схему.

Физическая схема – это описание структуры базы данных с указанием всех таблиц, атрибутов, типов данных, индексов, внешних ключей и ограничений целостности. Она определяет реальную организацию хранения данных в СУБД и отражает конкретные технические характеристики полей.

Физическая схема базы данных авиакомпании состоит из семи взаимосвязанных таблиц: «пользователи», «полёты», «самолёты», «бронь», «платежи», «поддержка\_билеты» и «уведомления» (рисунок 14).

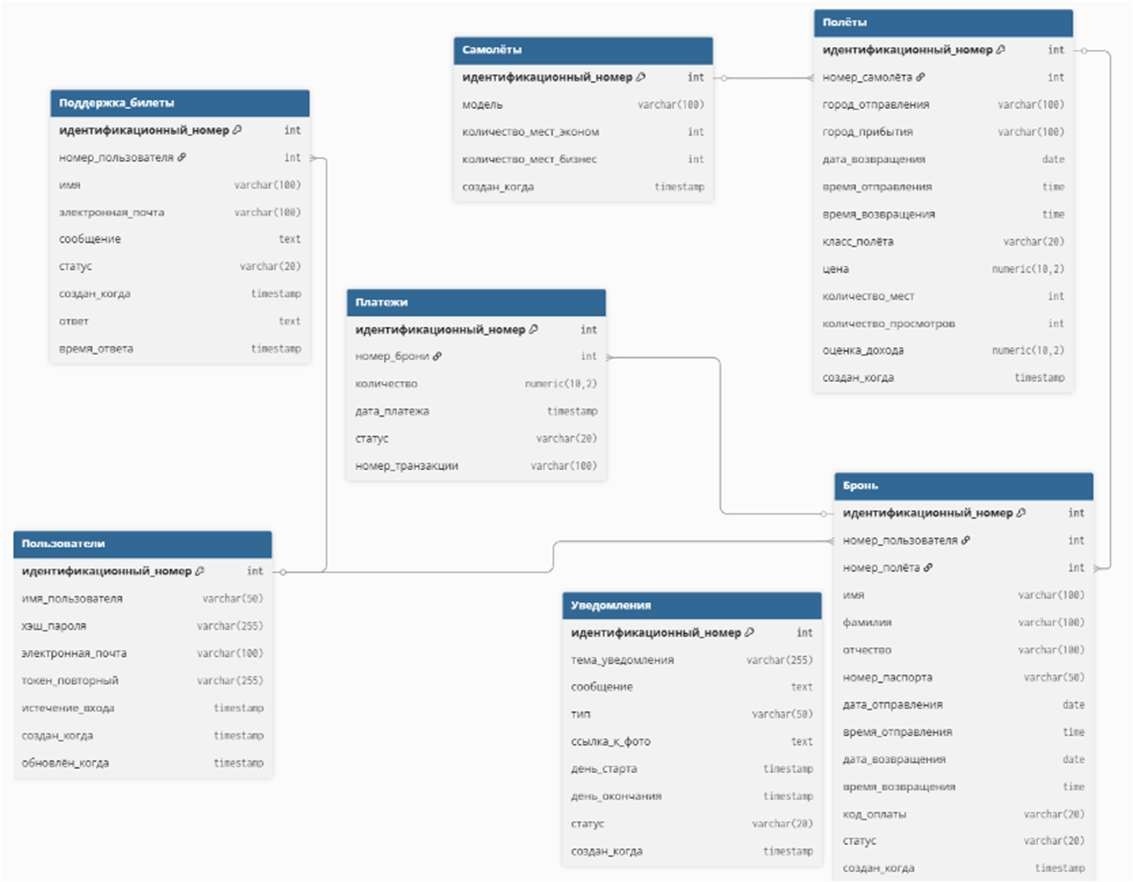


Рисунок 14 – Физическая схема базы данных

Опишем все типы полей каждой из таблиц базы данных, а также диапазон их значений [9].

Поля таблицы «полёты»:

* идентификационный\_номер – *INT*, первичный ключ, автоинкремент;
* номер\_самолёта – *INT*, внешний ключ, ссылка на таблицу «Самолёты», поле идентификационный\_номер;
* город\_отправления – *VARCHAR*(100), не допускает *NULL*;
* город\_прибытия – *VARCHAR*(100), не допускает *NULL*;
* дата\_возвращения – *DATE*, допускает *NULL*;
* время\_отправления – *TIME* *WITH* *TIME* *ZONE*, не допускает *NULL*;
* время\_возвращения – *TIME* *WITH* *TIME* *ZONE*, допускает *NULL*;
* класс\_полёта – *VARCHAR*(20), по умолчанию *economy*;
* цена – *NUMERIC*(10,2), не допускает *NULL*;
* количество\_мест – *INT*, не допускает *NULL*, значение по умолчанию равняется 1;
* количество\_просмотров – *INT*, по умолчанию 0;
* оценка\_дохода – *NUMERIC*(10,2), по умолчанию 0;
* создан\_когда – *TIMESTAMP* *WITH* *TIME* *ZONE*, по умолчанию *CURRENT*\_*TIMESTAMP*.

Обоснование выбора типов и ограничений.

Тип *INT* выбран для идентификаторов и ссылок, обеспечивая эффективное хранение и индексацию. Поля город\_отправления и город\_прибытия ограничены длиной 100 символов – этого достаточно для хранения географических названий любого уровня. Поля даты (*DATE*) и времени (*TIME* *WITH* *TIME* *ZONE*) разделены, чтобы обеспечить гибкость при формировании расписаний, особенно для обратных рейсов. Поле класс\_полёта хранит категорию обслуживания – эконом, бизнес, премиум; тип *VARCHAR*(20) достаточен для текстовых значений. Поля цена и оценка\_дохода используют тип *NUMERIC*(10,2) для точного хранения денежных значений без ошибок округления. Поля количество\_мест и количество\_просмотров являются целыми числами, ограничение по умолчанию позволяет избежать пустых значений. Поле создан\_когда фиксирует момент добавления рейса с учётом часового пояса.

Поля таблицы «самолёты»:

* идентификационный\_номер – *INT*, первичный ключ, автоинкремент;
* модель – *VARCHAR*(100), не допускает *NULL*;
* количество\_мест\_эконом – *INT*, не допускает *NULL*;
* количество\_мест\_бизнес – *INT*, не допускает *NULL*; – создан\_когда – *TIMESTAMP* *WITH* *TIME* *ZONE*; Обоснование выбора типов и ограничений.

Поле модель содержит название самолёта (например, *Airbus* *A*320, *Boeing* 737), поэтому *VARCHAR*(100) – оптимальная длина для описания моделей и модификаций. Поля количество\_мест\_эконом и количество\_мест\_бизнес целочисленные, так как представляют физическое количество посадочных мест. Тип *INT* обеспечивает хранение до нескольких сотен или тысяч значений, что достаточно для любых типов лайнеров. Поле создан\_когда фиксирует момент регистрации самолёта в базе данных.

Поля таблицы «бронь»:

* идентификационный\_номер – *INT*, первичный ключ, автоинкремент;
* номер\_пользователя – *INT*, внешний ключ, ссылка на таблицу
* «пользователи», поле идентификационный\_номер;
* номер\_полёта – *INT*, внешний ключ, ссылка на таблицу «Полёты», поле идентификационный\_номер;
* имя – *VARCHAR*(100), не допускает *NULL*;
* фамилия – *VARCHAR*(100), не допускает *NULL*;
* отчество – *VARCHAR*(100), допускает *NULL*;
* номер\_паспорта – *VARCHAR*(50), не допускает *NULL*;
* дата\_отправления – *DATE*, не допускает *NULL*;
* время\_отправления – *TIME* *WITH* *TIME* *ZONE*, не допускает *NULL*;
* дата\_возвращения – *DATE*, допускает *NULL*;
* время\_возвращения – *TIME* *WITH* *TIME* *ZONE*, допускает *NULL*;
* код\_оплаты – *VARCHAR*(20), не допускает *NULL*;
* статус – *VARCHAR*(20), по умолчанию *pending*; – создан\_когда – *TIMESTAMP* *WITH* *TIME* *ZONE*; Обоснование выбора типов и ограничений.

Поле идентификационный\_номер определено как *INT* для обеспечения уникальности записей. Поля номер\_пользователя и номер\_полёта установлены как внешние ключи для сохранения ссылочной целостности между таблицами. Поля имя, фамилия и отчество имеют длину 100 символов, что позволяет хранить полные ФИО без сокращений. Поле номер\_паспорта хранится в формате *VARCHAR*(50) для поддержки паспортов разных стран. Поля даты и времени позволяют хранить как основную, так и обратную часть маршрута. Поле код\_оплаты используется для связи с системой платежей, а статус хранит текущее состояние бронирования. Поле создан\_когда фиксирует момент оформления брони.

Поля таблицы «платежи»:

* идентификационный\_номер – *INT*, первичный ключ, автоинкремент;
* номер\_брони – *INT*, внешний ключ, ссылка на таблицу «бронь», поле идентификационный\_номер;
* количество – *NUMERIC*(10,2), не допускает *NULL*;
* дата\_платежа – *TIMESTAMP* *WITH* *TIME* *ZONE*;
* статус – *VARCHAR*(20), по умолчанию *pending*;
* номер\_транзакции – *VARCHAR*(100), уникальное значение; Обоснование выбора типов и ограничений.

Тип *NUMERIC*(10,2) используется для точного хранения денежных значений без округлений. Поле номер\_транзакции ограничено 100 символами, что позволяет хранить идентификаторы из внешних платёжных систем. Статус (*VARCHAR*(20)) отражает текущее состояние транзакции: «успешно», «ошибка» или «ожидает». Поле дата\_платежа фиксирует время проведения операции, а связь через номер\_брони обеспечивает полное соответствие платежа конкретному бронированию.

Поля таблицы «поддержка\_билеты»:

* идентификационный\_номер – *INT*, первичный ключ, автоинкремент;
* номер\_пользователя – *INT*, внешний ключ;
* имя – *VARCHAR*(100), не допускает *NULL*;
* электронная\_почта – *VARCHAR*(100), не допускает *NULL*;
* сообщение – *TEXT*, не допускает *NULL*;
* статус – *VARCHAR*(20), по умолчанию '*new*';
* создан\_когда – *TIMESTAMP*;
* ответ – *TEXT*, допускает *NULL*;
* время\_ответа – *TIMESTAMP*, допускает *NULL*.

Обоснование выбора типов и ограничений.

Текстовые поля сообщение и ответ имеют тип *TEXT*, так как длина текста сообщений не ограничена заранее. Поле статус необходимо для отслеживания этапа обработки запроса (например, «новое», «решено», «в ожидании»). Поле электронная\_почта хранит контактный адрес пользователя, а связь по внешнему ключу номер\_пользователя обеспечивает идентификацию автора обращения. Поля времени фиксируют момент создания и ответа, что позволяет контролировать сроки выполнения обращений.

Поля таблицы «пользователи»:

* идентификационный\_номер – *INT*, первичный ключ, автоинкремент;
* имя\_пользователя – *VARCHAR*(50), уникальное;
* хэш\_пароля – *VARCHAR*(255), не допускает *NULL*;
* электронная\_почта – *VARCHAR*(100), уникальное;
* токен\_повторный – *VARCHAR*(255), допускает *NULL*;
* истечение\_входа – *TIMESTAMP*, допускает *NULL*;
* создан\_когда – *TIMESTAMP* *WITH* *TIME* *ZONE*; – обновлён\_когда – *TIMESTAMP* *WITH* *TIME* *ZONE*; Обоснование выбора типов и ограничений.

Поле идентификационный\_номер выбрано типа *INT*, так как идентификаторы пользователей представляют собой последовательные числовые значения, удобные для индексирования и быстрого поиска; диапазона *INT* достаточно для хранения миллионов записей. Поле имя\_пользователя имеет ограничение длины *VARCHAR*(50) – этого диапазона достаточно для ввода логина любого разумного размера, а уникальность и *NOT* *NULL* обеспечивают однозначную идентификацию пользователя в системе. Поле хэш\_пароля требует длину до 255 символов, так как хэширование (например, с помощью алгоритмов *SHA*-256, *bcrypt* или *Argon*2) генерирует строки длиной 60-255 символов. Поле электронная\_почта ограничено 100 символами – этого достаточно для всех реальных адресов, уникальность необходима, чтобы избежать дублирования и использовать почту для восстановления доступа. Поле токен\_повторный допускает *NULL*, так как токен присутствует только во время сброса пароля. Поле истечение\_входа хранит временную метку истечения сессии, имеет тип *TIMESTAMP* для точности до секунд. Поля создан\_когда и обновлён\_когда имеют тип *TIMESTAMP* *WITH* *TIME* *ZONE*, что позволяет фиксировать дату и время создания и изменения с учётом часовых поясов, с автоматическим присвоением текущего времени при вставке или обновлении данных.

Поля таблицы «уведомления»:

* идентификационный\_номер – *INT*, первичный ключ, автоинкремент;
* тема\_уведомления – *VARCHAR*(255), не допускает *NULL*;
* сообщение – *TEXT*, не допускает *NULL*;
* тип – *VARCHAR*(50), значение по умолчанию *info*;
* ссылка\_к\_фото – *TEXT*, допускает *NULL*;
* день\_старта – *TIMESTAMP* *WITH* *TIME* *ZONE*;
* день\_окончания – *TIMESTAMP* *WITH* *TIME* *ZONE*; – статус – *VARCHAR*(20), значение по умолчанию *active*; – создан\_когда – *TIMESTAMP* *WITH* *TIME* *ZONE*.

Обоснование выбора типов и ограничений.

Поле тема\_уведомления ограничено длиной 255 символов, что позволяет формировать информативные заголовки сообщений. Поле сообщение хранит текст уведомления произвольной длины. Поле тип определяет категорию уведомления – информационное, рекламное или предупреждение. ссылка\_к\_фото позволяет хранить *URL*-адрес изображений или рекламных баннеров. Поля начала и окончания публикации реализованы через *TIMESTAMP* *WITH* *TIME* *ZONE*, что обеспечивает точное управление временем отображения уведомлений независимо от региона. Поле статус регулирует активность уведомления, а поле создан\_когда фиксирует момент его добавления в систему.

Таким образом, рассмотрели типы данных полей всех семи таблиц базы данных информационной системы авиакомпании, а также ограничения и их отношения.

Перейдём к рассмотрению функций для работы с полученной базой данных.

* 1. **Программная реализация приложения**

Для описания программной реализации приложения разберём функции веб сайта, а именно как они реализованы в операцих с базой данных.

Отдельные методы для создания таблиц опустим, так как они уже были разобраны ранее.

Начнём разбор реализации сайта с функции поисков билета по фильтру. Функция *findFlights* реализует поиск рейсов по заданным фильтрам. В качестве параметров она принимает города отправления и назначения, дату вылета, дату возврата и класс перелёта.

*const* *findFlights* = *async* ({ *from*, *to*, *departDate*, *returnDate*, *flightClass* }) => {

Создаётся асинхронная функция *findFlights*, принимающая объект с параметрами поиска: *from* – город отправления, *to* – город назначения, *departDate* – дата вылета, *returnDate* – дата обратного рейса, а *flightClass* является классом рейса.

*let* *conditions* = ['*from*\_*city* = $1', '*to*\_*city* = $2'];

Инициализируется массив *conditions*, содержащий базовые условия для *SQL*-запроса: *from*\_*city* и *to*\_*city* должны соответствовать первым двум параметрам.

*let* *params* = [*from*, *to*];

Создаётся массив *params*, который будет содержать значения для подстановки в *SQL*-запрос. Первые два элемента – это *from* и *to*.

*let* *index* = *params*.*length* + 1;

Создаётся переменная *index*, чтобы нумеровать параметры для *SQL*плейсхолдеров ($3, $4 и т.д.). Начальное значение = 3, потому что первые два параметра уже заняты.

*if* (*departDate*) { *conditions*.*push*(`*depart*\_*date* = $${*index*}`); *params*.*push*(*departDate*); *index*++;

}

Если указана дата вылета (*departDate*), добавляем условие *depart*\_*date* = $*index* в массив условий и значение даты в *params*. Затем увеличиваем *index* для следующего параметра.

*const* *sql* = `*SELECT* \* *FROM* *flights*

*WHERE* ${*conditions*.*join*(' *AND* ')}

*ORDER* *BY* *depart*\_*date* *ASC*`;

Формируем *SQL*-запрос: выбираем все колонки из таблицы *flights*, объединяем условия через *AND*, сортируем результаты по дате вылета в порядке возрастания.

Продолжим обзор функций с метода создания брони, которая инициирована пользователем. Она в свою очередь напрямую связана с таблицей полётов – получение данных о выбранном рейсе. Разберём функцию *createBooking*.

*const* *createBooking* = *async* ({ *user*\_*id*, *flight*\_*id*, *fullname*, *passport*,

*depart*\_*date*, *depart*\_*time*, *return*\_*date*, *return*\_*time*, *passengers* = 1 }) => {

Создаётся асинхронная функция *createBooking*, принимающая объект с параметрами бронирования.

*const* *expiresAt* = *new* *Date*(*Date*.*now*() + 30 \* 60 \* 1000); // 30 минут

Создаётся переменная *expiresAt*, которая хранит время истечения бронирования – текущее время 30 минут.

*const* *res* = *await* *query*(

`*INSERT* *INTO* *bookings*

(*user*\_*id*, *flight*\_*id*, *fullname*, *passport*, *depart*\_*date*, *depart*\_*time*, *return*\_*date*, *return*\_*time*, *passengers*, *expires*\_*at*)

*VALUES* ($1,$2,$3,$4,$5,$6,$7,$8,$9,$10)

*RETURNING* \*`,

[*user*\_*id*, *flight*\_*id*, *fullname*, *passport*, *depart*\_*date*, *depart*\_*time*, *return*\_*date* || *null*, *return*\_*time* || *null*, *passengers*, *expiresAt*]

);

Выполняется *SQL*-запрос на вставку новой записи в таблицу *bookings*.

Вставляются все переданные значения, если *return*\_*date* или *return*\_*time* отсутствуют – используется *null*.

*RETURNING* \* возвращает созданную запись, результат сохраняется в *res*.

*await* *query*(

`*UPDATE* *flights* *SET* *seats* = *seats* – $1 *WHERE* *id* = $2 *AND* *seats*

>= $1`,

[*passengers*, *flight*\_*id*]

);

Обновляется таблица *flights*, уменьшая количество доступных мест на число пассажиров. Условие *seats* >= $1 гарантирует, что количество мест не станет отрицательным.

*return* *res*.*rows*[0];};

Функция возвращает объект с данными созданного бронирования

(*res*.*rows*[0]).

После покупки билетов и осуществления рейса билеты перемещаются в историю. Рассмотрим функцию *moveOldTicketsToHistory* отвечающую за то. *async* *function* *moveOldTicketsToHistory*() {

Создаётся асинхронная функция *moveOldTicketsToHistory*, которая не принимает аргументы. Её задача – переместить старые билеты в историю, обновляя их статус.

*try* {

Начало блока *try* для обработки возможных ошибок при выполнении *SQL*-запроса.

*const* *query* = ` *UPDATE* *bookings*

*SET* *status* = '*history*'

*WHERE* *status* = '*paid*'

*AND* (

(*return*\_*date* *IS* *NOT* *NULL* *AND* *return*\_*date* < *CURRENT*\_*DATE*)

*OR* (*return*\_*date* *IS* *NULL* *AND* *depart*\_*date* < *CURRENT*\_*DATE*)

); `;

Создаётся *SQL*-запрос для обновления таблицы *bookings*.

Обновляется поле *status* на '*history*' для записей, где текущий статус '*paid*'. Условие: если есть дата обратного рейса (*return*\_*date* *IS* *NOT* *NULL*) и она меньше текущей даты, или если билеты в один конец (*return*\_*date* *IS* *NULL*) и дата вылета меньше текущей даты.

Таким образом, только завершённые рейсы переводятся в историю.

*await* *pool*.*query*(*query*);

Выполняется *SQL*-запрос через пул подключений *pool*. Ожидается завершение операции. Вместо того чтобы создавать новое подключение для каждого запроса, пул поддерживает несколько готовых соединений, которые переиспользуются.

} *catch* (*err*) { *console*.*error*('Ошибка при обновлении истории билетов:', *err*);

}

}

Блок *catch* перехватывает ошибки, возникшие при выполнении запроса, и выводит их в консоль.

Функция завершена.

Теперь рассмотрим некоторые функции, используемые в панели администратора системы.

Первой такой функций будет *updateFlight*. Она в свою очередь обнавляет общие сведения конкретного рейса.

*const* *updateFlight* = *async* (*id*, *updates*) => {

Создаётся асинхронная функция *updateFlight*, которая принимает два параметра: идентификатор рейса, который нужно обновить, *updates* – объект с полями и новыми значениями для обновления (например { *seats*: 100, *depart*\_*time*: '12:00' }).

*const* *fields* = []; *const* *values* = []; *let* *idx* = 1;

Инициализируются три переменные: *fields* – массив для хранения *SQL*выражений вида *column* = $1, *values* – массив значений для подстановки в *SQL*–запрос, *idx* – счётчик параметров для плейсхолдеров $1, $2 и так далее.

*for* (*const* [*key*, *value*] *of* *Object*.*entries*(*updates*)) { *fields*.*push*(`${*key*} = $${*idx*}`); *values*.*push*(*value*); *idx*++;

}

Проходим по всем парам ключ-значение из объекта *updates*. Для каждого поля формируем строку ключ = $индекс и добавляем её в массив *fields*. Значение добавляется в массив *values*. Индекс *idx* увеличивается для следующего параметра.

*values*.*push*(*id*);

Добавляем *id* рейса в массив *values* для использования в условии *WHERE*.

*const* *sql* = `*UPDATE* *flights* *SET* ${*fields*.*join*(', ')} *WHERE* *id*=$${*idx*} *RETURNING* \*`;

Формируем *SQL*–запрос: *SET* содержит все поля для обновления, соединённые через запятую, *WHERE* *id*=$индекс – обновляем только запись с заданным идентификатором, *RETURNING* – возвращает обновлённую запись после выполнения запроса.

*const* *res* = *await* *query*(*sql*, *values*);

Выполняем *SQL*-запрос с подстановкой массива *values*. Ожидаем завершение операции.

*return* *res*.*rows*[0];

};

Возвращаем объект с обновлённой записью рейса.

Разберём функцию, отвечающую за получение активных уведомлений, которые отображаются на сайте пользователю. Данная функция выбирает из базы данных только те уведомления, у которых установлен активный статус и текущая дата находится в пределах времени их действия.

*const* *getActiveNotifications* = *async* () => {

Объявляется асинхронная функция *getActiveNotifications*, не принимающая аргументов. Её задача – получить список актуальных уведомлений, которые сейчас должны быть показаны пользователю.

*const* *now* = *new* *Date*();

Создаётся переменная *now*, содержащая текущие дату и время. Это значение используется для проверки, попадает ли уведомление в период своей активности.

*const* *res* = *await* *query*(`

*SELECT* *id*, *title*, *message*, *image*\_*url*, *type*

*FROM* *site*\_*notifications*

*WHERE* *is*\_*active* = *true*

*AND* *start*\_*date* <= $1

*AND* (*end*\_*date* >= $1 *OR* *end*\_*date* *IS* *NULL*)

*ORDER* *BY* *created*\_*at* *DESC*

*LIMIT* 5

`, [*now*]);

Выполняется *SQL*-запрос к таблице *site*\_*notifications*. Из таблицы выбираются поля: *id*, *title* (заголовок), *message* (текст уведомления), *image*\_*url* (ссылка на изображение), *type* (тип уведомления). Условия выборки: *is*\_*active* = *true* – уведомление активно; *start*\_*date* <= $1 – дата начала активности уже наступила; (*end*\_*date* >= $1 *OR* *end*\_*date* *IS* *NULL*) – дата окончания активности ещё не прошла или не указана вовсе; $1 заменяется текущей датой *now*. Результаты сортируются по дате создания (*created*\_*at* *DESC*), чтобы первыми шли самые новые уведомления. Ограничение *LIMIT* 5 возвращает не более пяти актуальных уведомлений.

*return* *res*.*rows*;};

Возвращается массив найденных записей (*res*.*rows*), где каждая строка представляет одно активное уведомление.

Рассмотренные функции обеспечивают все операции веб-приложения при работе с базой данных – поиск рейсов, бронирование билетов, обновление и архивирование информации. Более подробная реализация остальных функций представлена в приложении Б. В совокупности они обеспечивают корректное функционирование сайта.

* 1. **Инструкция пользователя**

Сайт авиакомпании *DocAvia* предназначен для организации и автоматизации процессов, связанных с бронированием, покупкой и управлением авиабилетами. Веб-приложение обеспечивает пользователей удобным интерфейсом для поиска рейсов, выбора маршрута, оформления билета и взаимодействия с системой поддержки.

Система реализована в виде клиент-серверного веб-приложения. Клиентская часть доступна через любой современный браузер, а серверная часть работает на платформе *Node*.*js* с подключением к базе данных *PostgreSQL* [10].

Интерфейс разработан с использованием современных технологий (*HTML*5, *CSS*3, *JavaScript*) и оптимизирован для корректного отображения на устройствах.

Как и на любом сайте работа начинается с главной страницы, на которой размещение основные функции информационной системы: личный кабинет, поиск билетов, поддержка. Кроме этого, на странице размещена информация о самой авиакомпании, а также уведомления о акционных предложениях (рисунок 15).

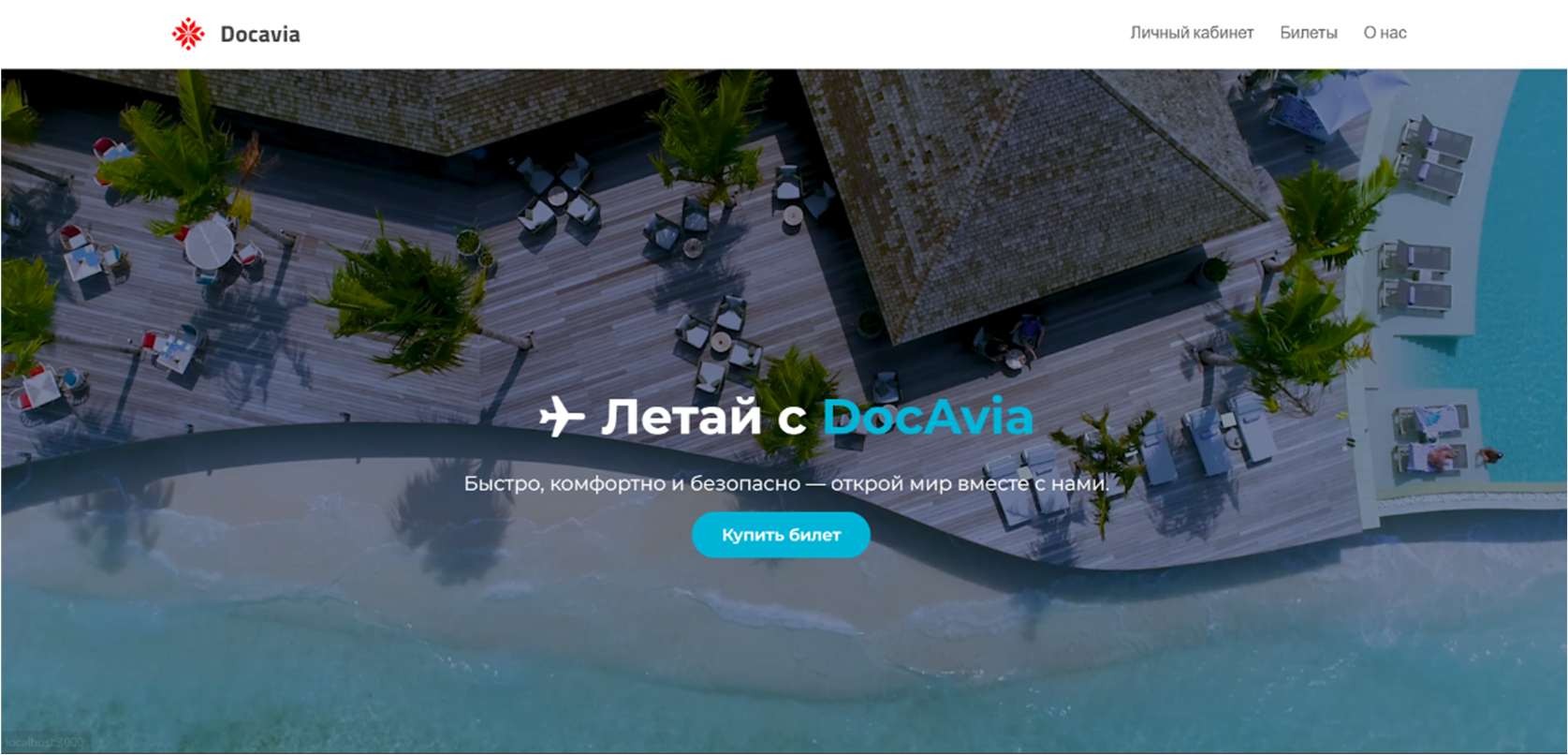


Рисунок 15 – Главная страница сайта авиакомпании

Для полноценной работы с сайтом пользователю необходимо войти и личный кабинет или зарегистрироваться. Для входа необходим только логин или *email*, а также пароля от личного кабинета, поэтому рассмотрим подробнее операцию регистрации нового пользователя. Для этого переходим на вкладку регистрации (рисунок 16).

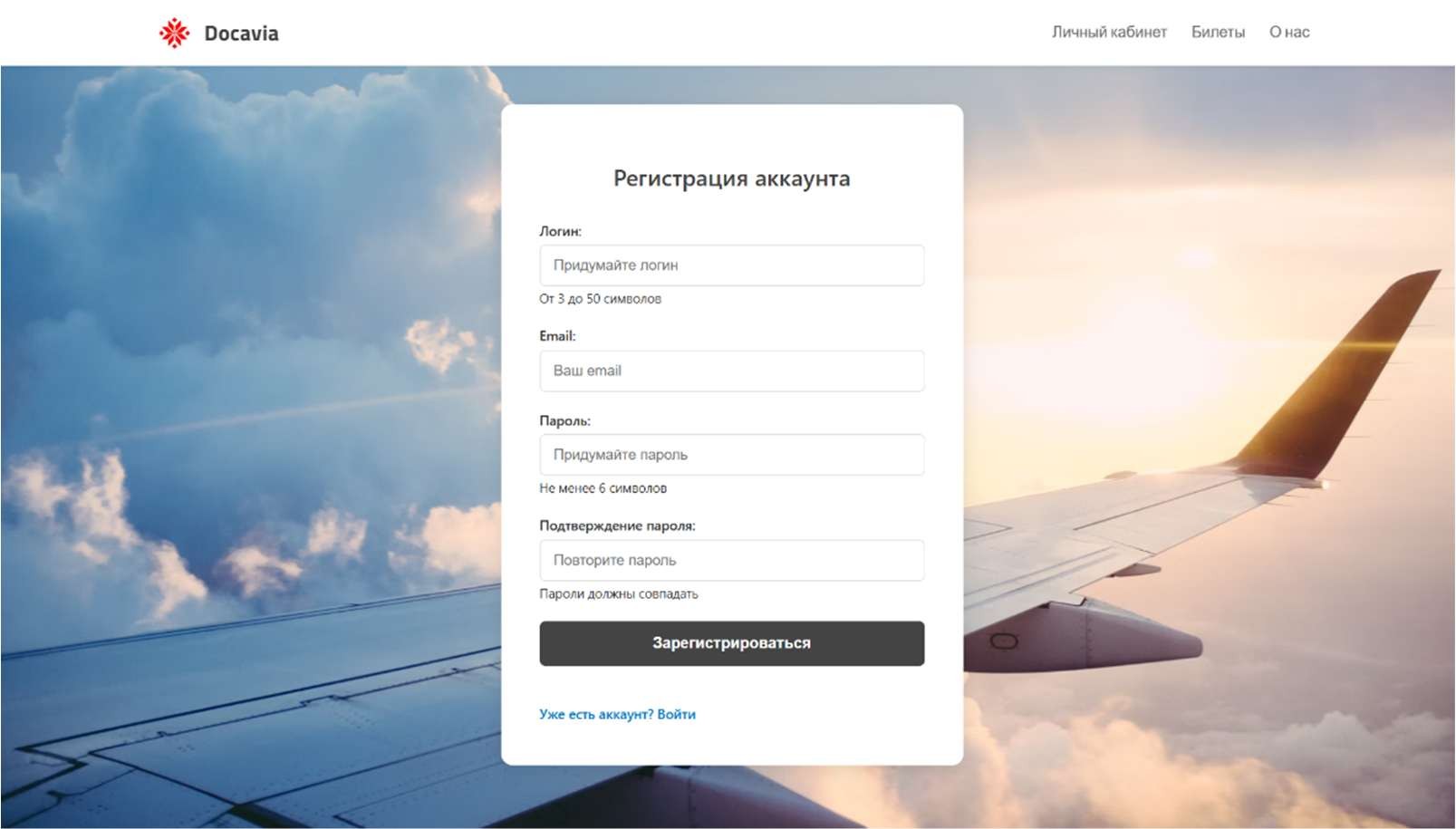


Рисунок 16 – Страница регистрации

Для страницы регистрации, нужны те же самые поля, что и для авторизации, а также подтверждение пароля. *Email* необходим для восстановления пароля. При необходимости пользователь получит на почту ссылку, содержащую токен для восстановления (рисунок 17).

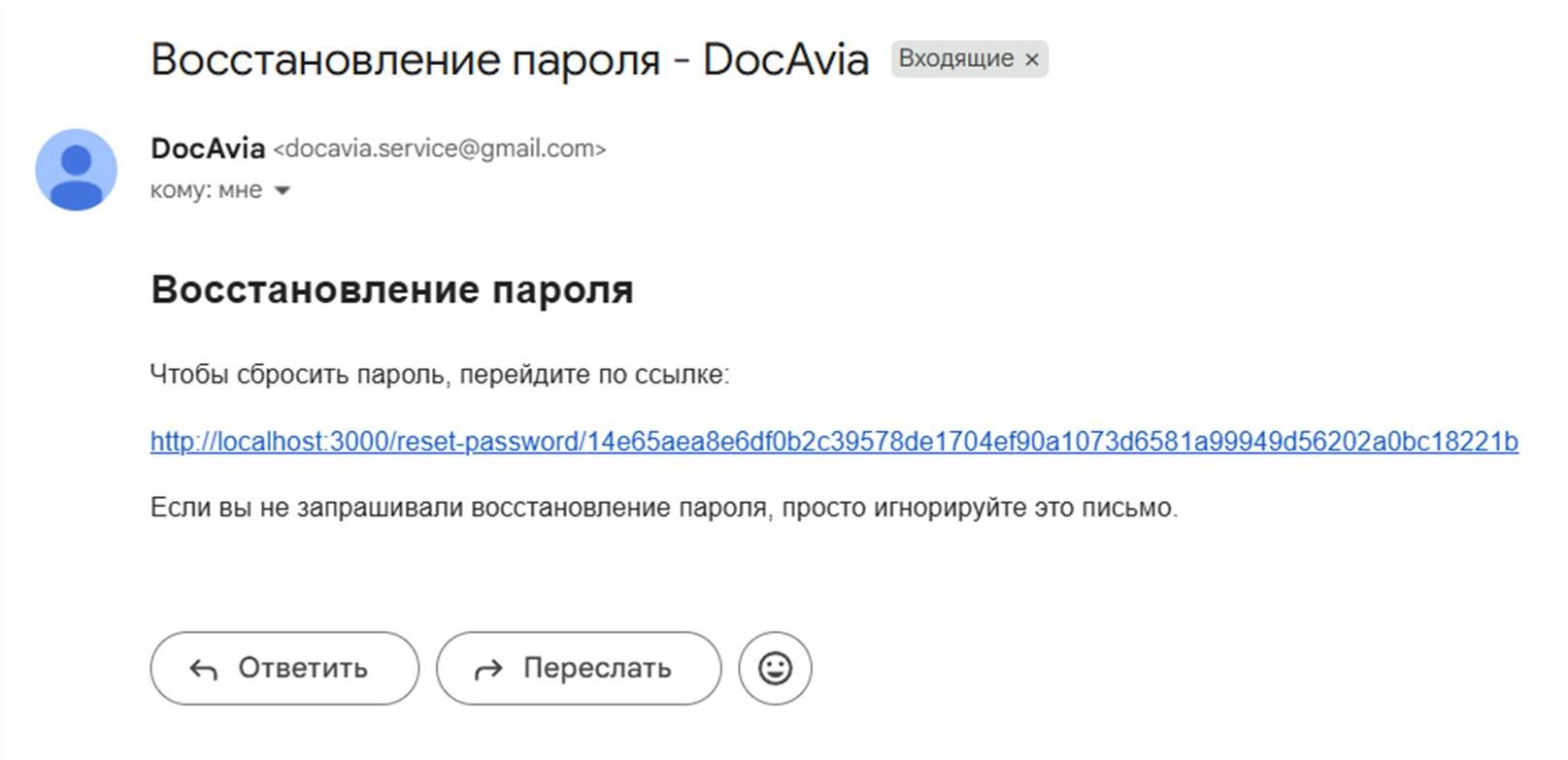


Рисунок 17 – Письмо восстановления пароля

После успешного входа в аккаунт станет доступна возможность войти в личный кабинет пользователя.

В личном кабинете имеется возможность просмотреть данные пользователя, а также забронированные билеты, купленные билеты, а также билеты уже прошедших рейсов – история билетов. Личный кабинет администратора имеет дополнительные функции (рисунок 18).

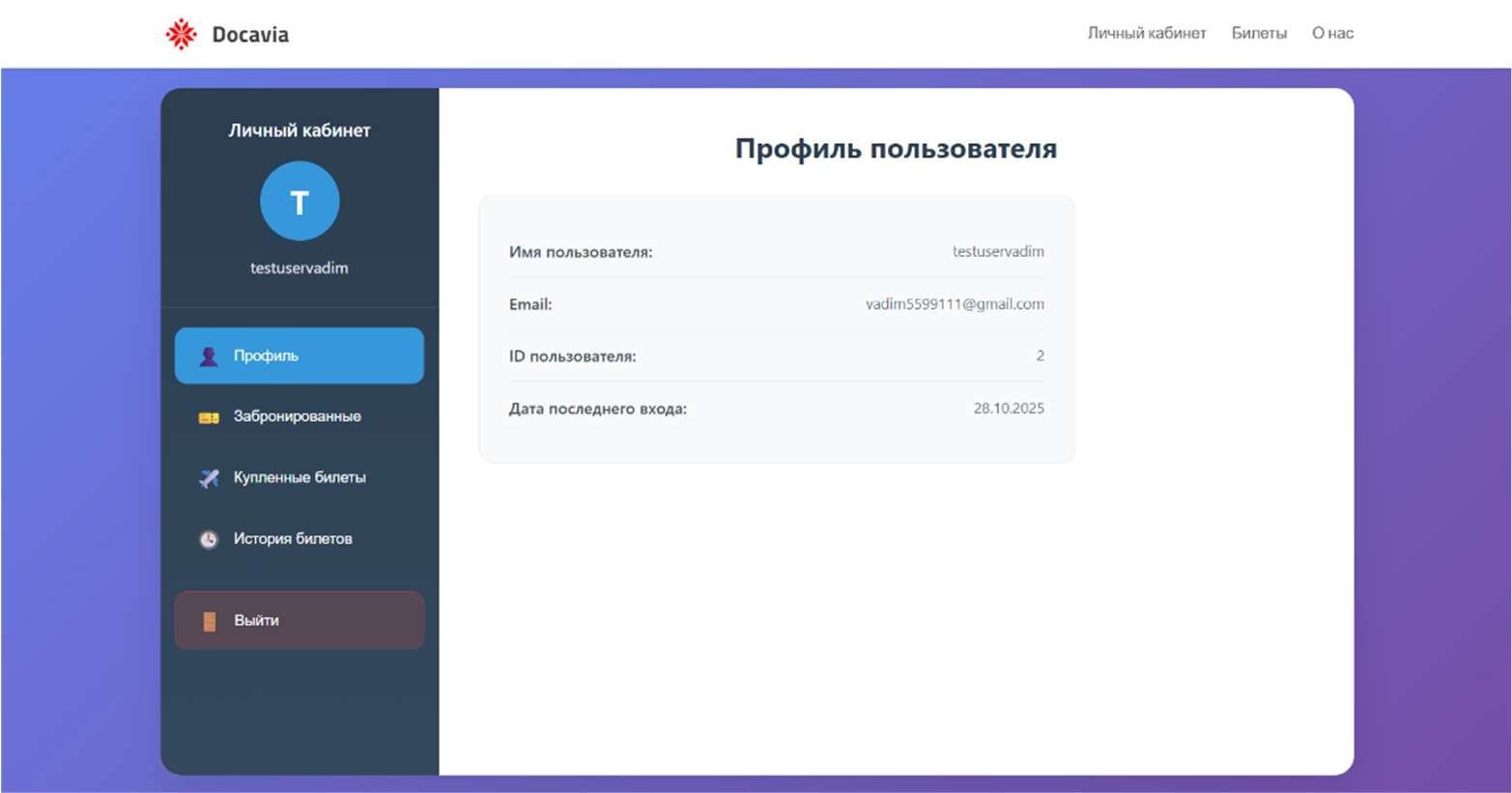


Рисунок 18 – Личный кабинет пользователя

Опция поиска билета доступна и неавторизованному пользователю. Для этого необходимо ввести города вылета и прибытия, а также тип, класс билета и дату отправления. Если тип билета указан с возвратом, необходимо также указать дату обратного вылета (рисунок 19).

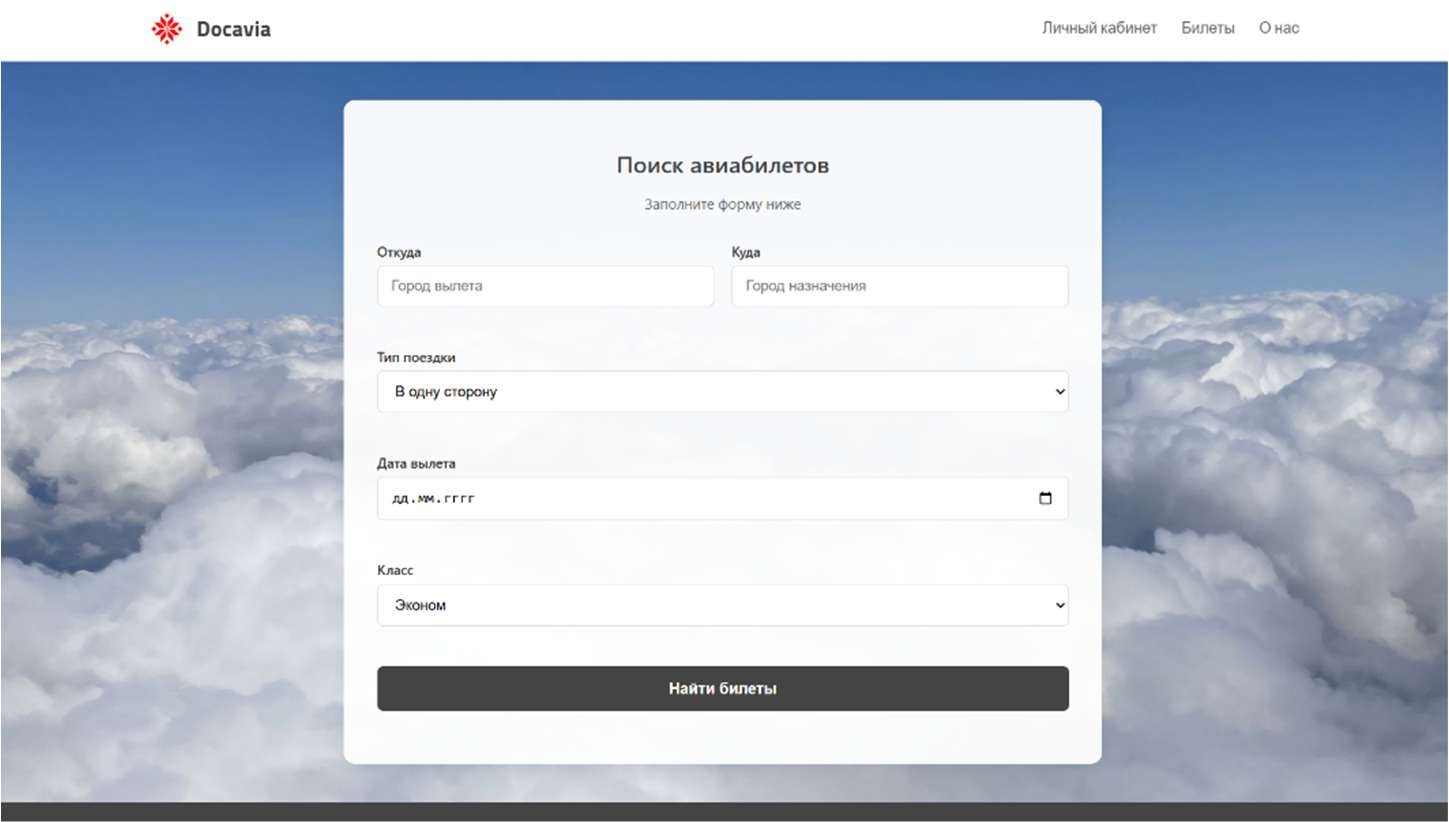


Рисунок 19 – Страница поиска билета

Если билеты найдены по параметрам пользователя, то они отобразятся на странице найденные рейсы. Сами билеты содержат основную информацию рейса: класс билета, количество мест, цену билета, а также дату вылета и возврата (если выбран билет в обе стороны) (рисунок 20). Если билеты не найдены отобразиться соответствующее окно, после чего пользователь сможет произвести поиск заново.

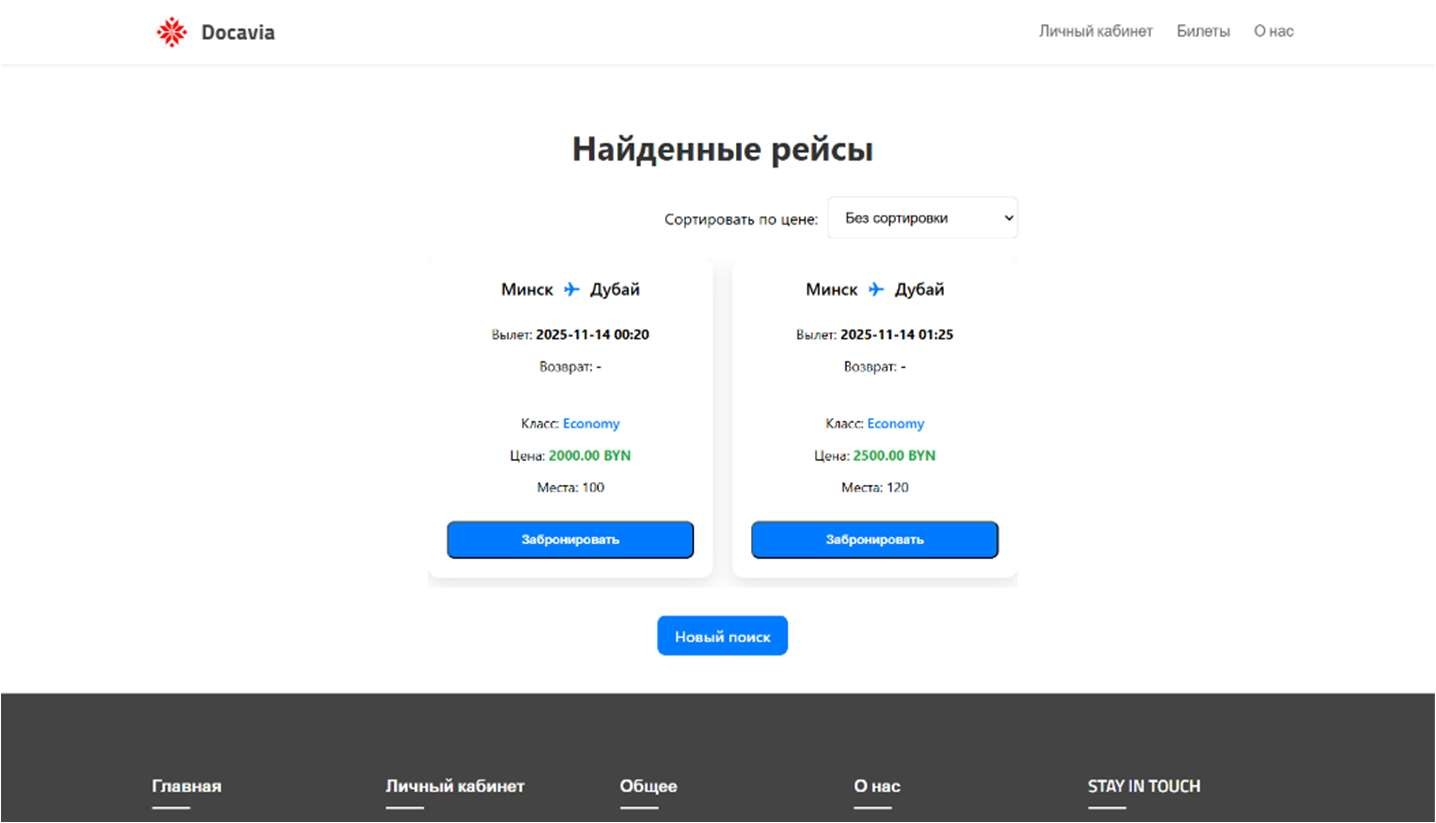


Рисунок 20 – Страница результатов поиска билета

Добавление рейсов в базу данных может осуществляться через личный кабинет администратора как с помощью ручного ввода данных рейса, так и с помощью загрузки *json* файла с данными.

После бронирования билет попадает в вкладку забронированные личного кабинет, где имеется возможность отменить бронь, а также увидеть основную информацию рейса, время, отведённое на оплату билета и сам код для оплаты (рисунок 21).

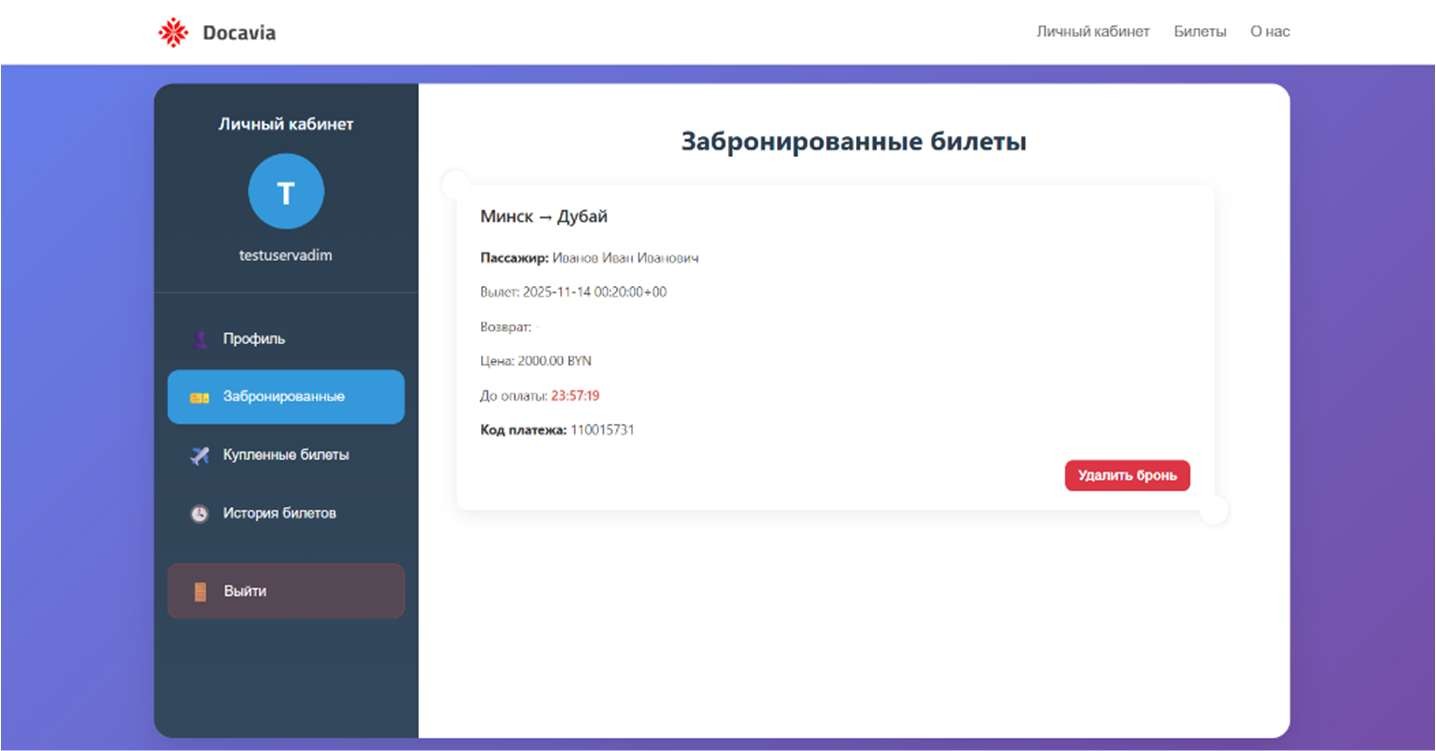


Рисунок 21 – Отображение забронированного билета

В последствии, после оплаты по коду, билет попадает в вкладку купленные билет, где точно также отображается основная информация рейса, а также статус билета оплачен, так как был оплачен (рисунок 22). После того, как билет станет не актуальным, он попадает в вкладку история билетов.

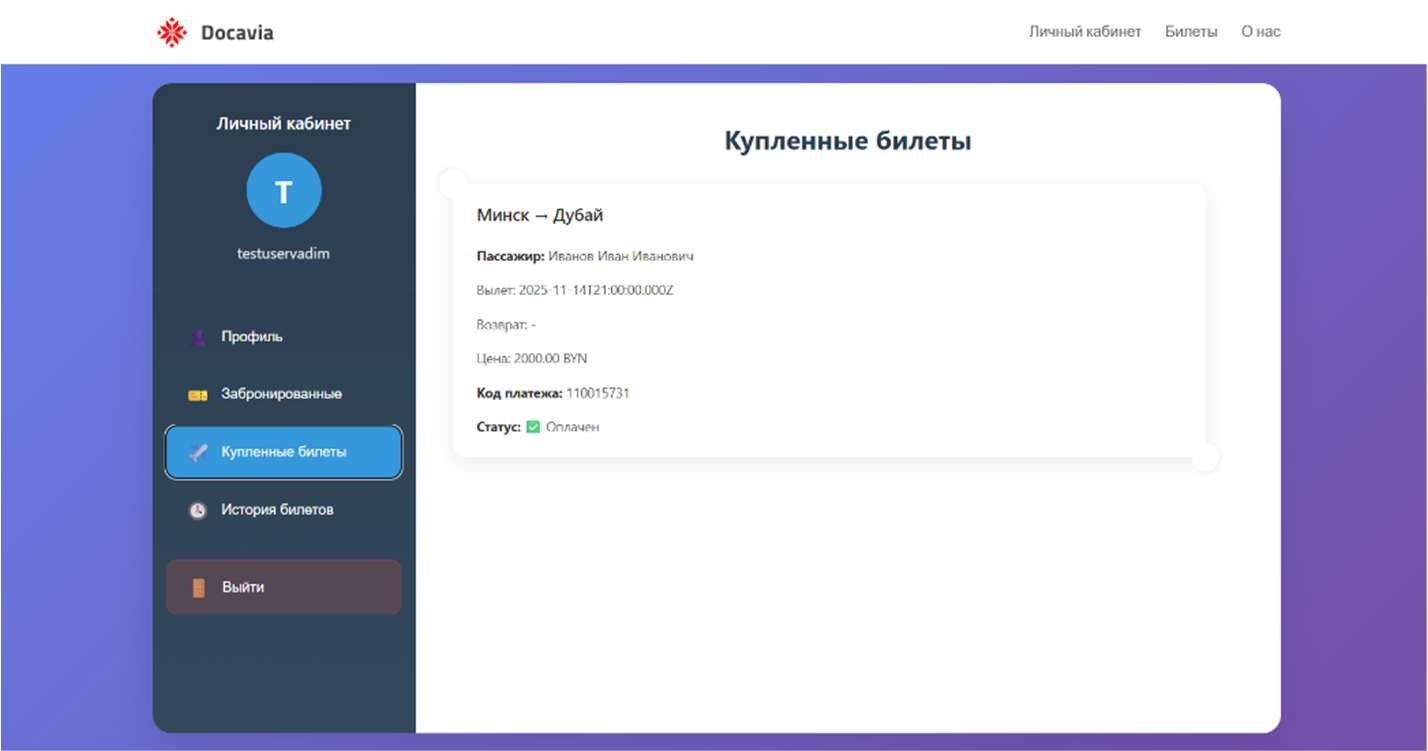


Рисунок 22 – Отображение купленного билета

В личном кабинете администратора имеется возможно отменять бронирование при его некорректных данных, а также менять данные конкретного рейса, при этом изменённые данные фиксируются только в билетах при поиске.

При возникновении проблемы пользователь имеет возможность обратиться в службу поддержки, для этого необходимо перейти на вкладку поддержки, указать имя, фамилию, *email*, а также проблему, с которой столкнулся пользователь. После ввода необходимых данных пользователь получит код обращения, по которому сможет получить ответ (рисунок 23).

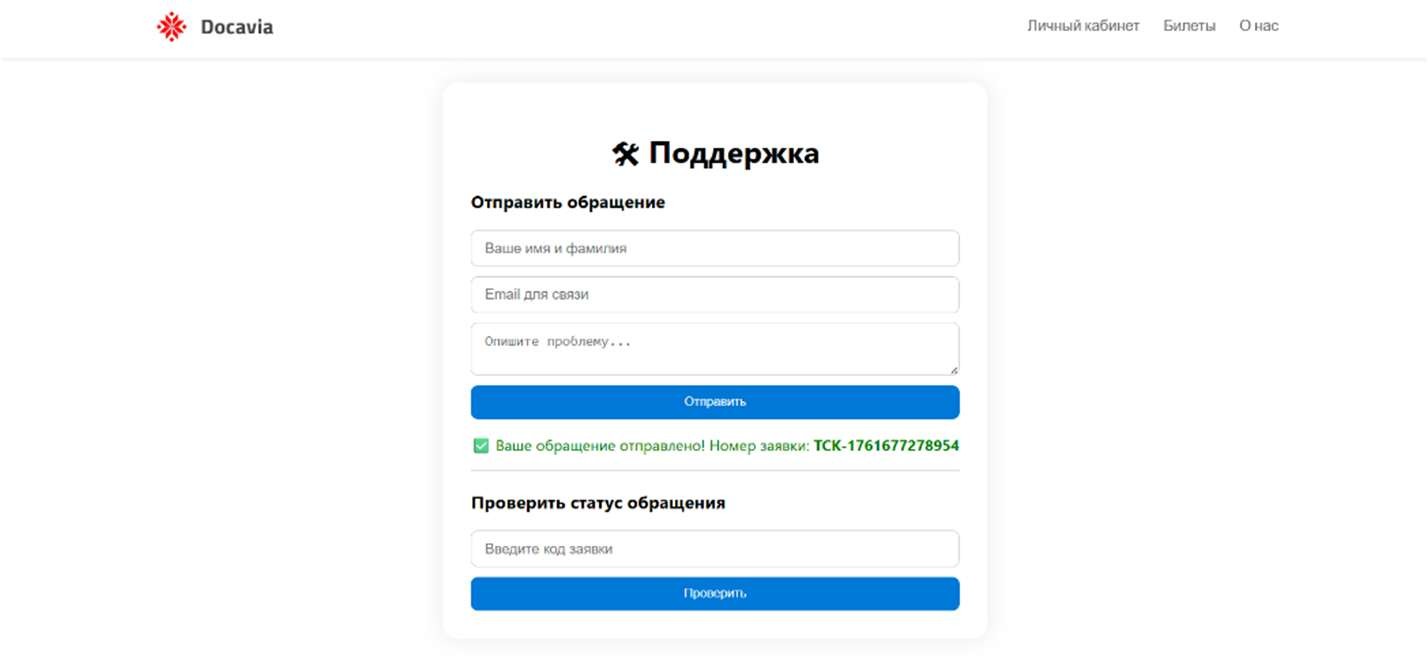


Рисунок 23 – Страница службы поддержки

Ответ может предоставить администратор в своём личном кабинете. На этом функции обычного пользователя завершаются, однако имеется аккаунт администратора упоминаемый выше, который имеет ещё несколько функций для комфортной работы с базой данных сайта авиакомпании (рисунок 24).

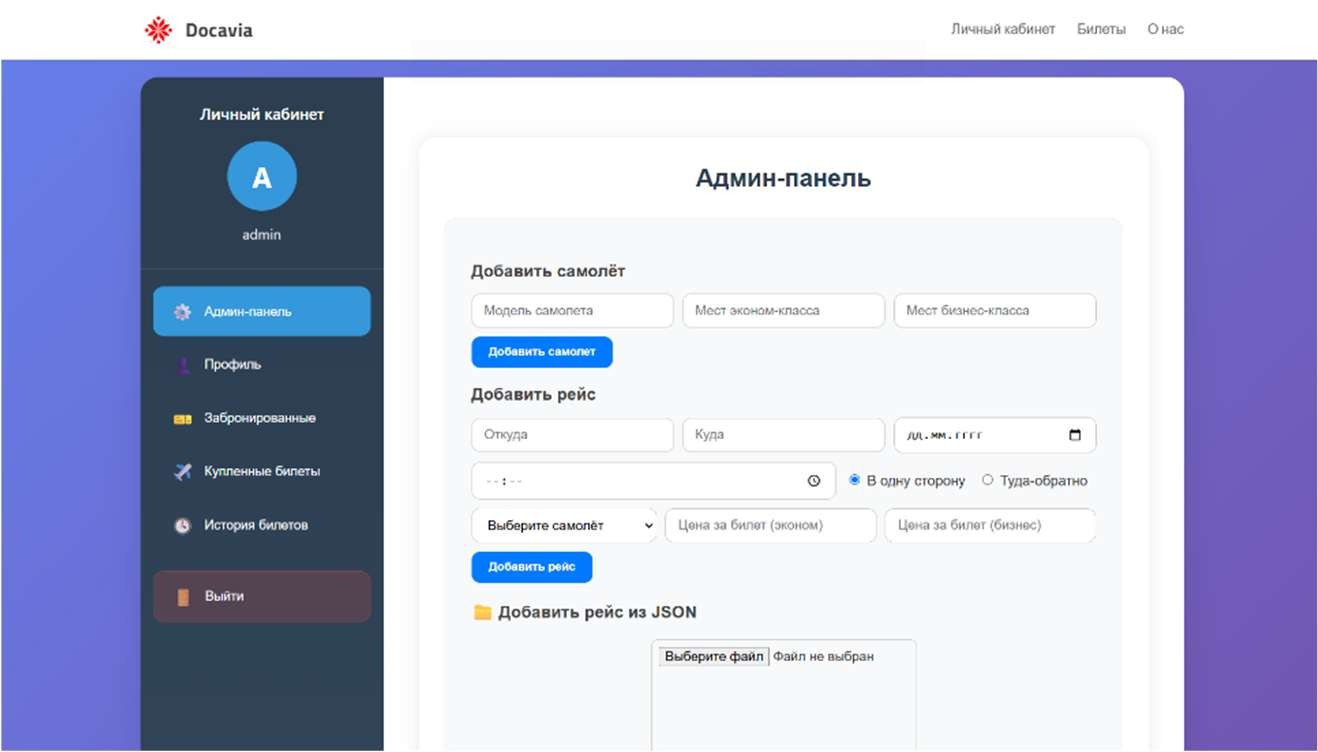


Рисунок 24 – Панель администратора

Ранее не была рассмотрена аналитическая функция информационной системы авикомпании. Она включает в себя данные о бронировании и продажах билетов, количествах рейсов, дохода компании, обращений пользователей за определённые промежутки времени: год, неделя, месяц, день, а также столбчатую диаграмму за эти же промежутки (рисунок 25). Диаграмма отображает тенденцию проведения рейсов, что явно указывает на количество клиентов авиакомпании.

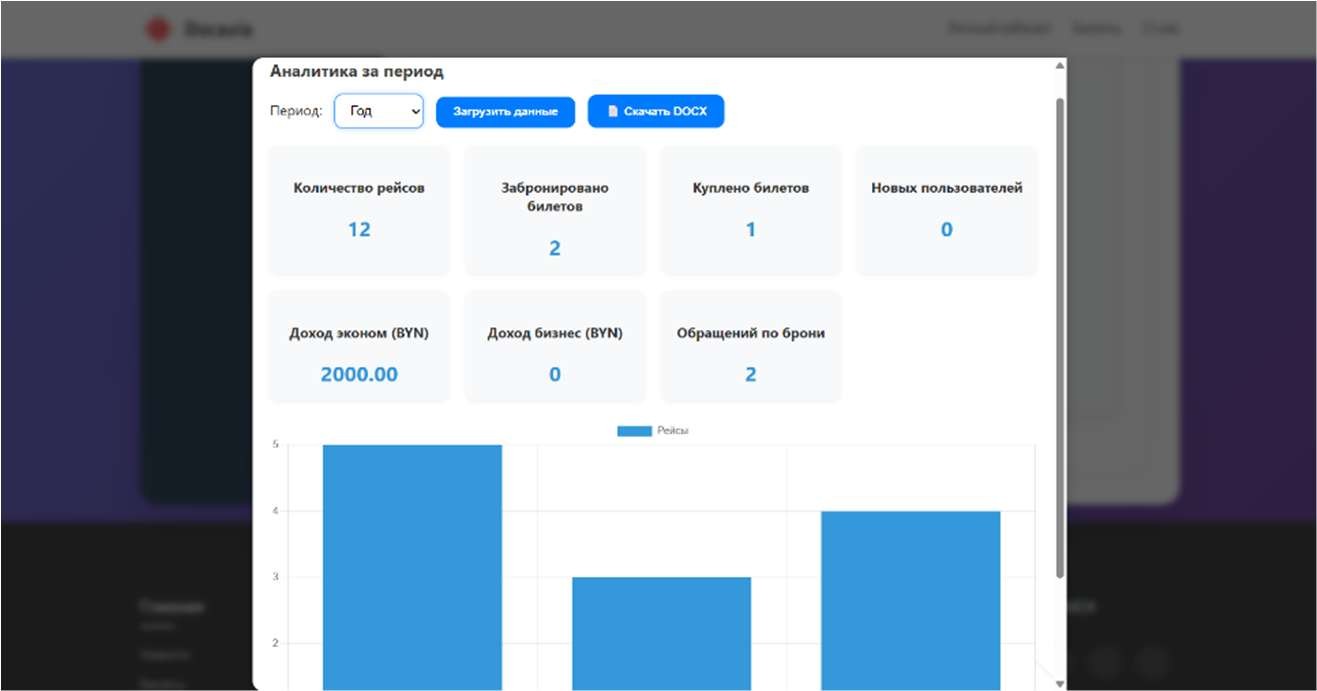


Рисунок 25 – Функция аналитики данных

Также имеется возможность вывести этот отчёт по аналитике в отдельный файл расширения *docx*. Отчёт в свою очередь имеет ту же информацию, что и аналитика на сайте, а также график рейсов и бронирований по дням (бронирования делятся на эконом и бизнес класс). Отчёт берётся по тому же промежутку времени, что и аналитика на сайте (рисунок 26).

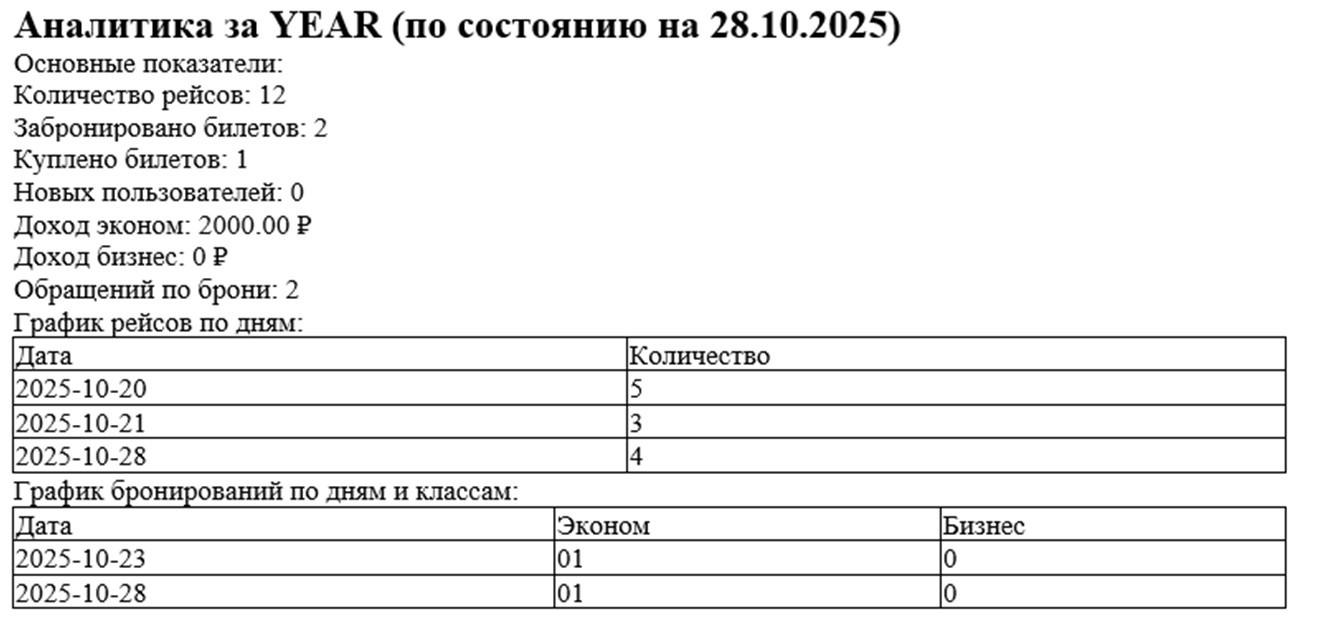


Рисунок 26 – Пример файла отчёта

Таким образом, функция сочетает в себе аналитический и информационный аспект панели администратора.

В результате разработки веб-приложения авиакомпании *DocAvia* удалось создать современную, удобную и функциональную систему, обеспечивающую полный цикл взаимодействия пользователя с сервисом авиаперевозок. Реализованный сайт объединяет в себе инструменты для поиска, бронирования, оплаты и анализа авиарейсов, что позволяет существенно упростить процесс покупки билетов и управление ими.

Благодаря адаптивному интерфейсу и использованию актуальных веб технологий обеспечивается корректная работа на различных устройствах и платформах.

Внедрение личного кабинета пользователя и администратора позволило организовать гибкое распределение прав доступа и обеспечить безопасную работу с персональными данными и базой рейсов.

Особое значение имеет аналитический модуль, предоставляющий наглядное представление о деятельности авиакомпании, динамике продаж и загруженности рейсов, что способствует повышению эффективности управленческих решений.

Возможность формирования отчётов и экспорта данных в документальный формат делает систему удобной не только для пользователей, но и для внутреннего анализа компании.

В целом, проект демонстрирует комплексный подход к автоматизации бизнес-процессов авиаперевозок. Он обеспечивает удобство всех этапов взаимодействия пользователя с системой, от поиска рейсов до оплаты билетов и получения поддержки. Встроенные аналитические инструменты позволяют эффективно контролировать работу авиакомпании и принимать управленческие решения.

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения курсового проекта была спроектирована и разработана автоматизированная информационная система авиакомпании, обеспечивающая поддержку основных процессов бронирования и управления рейсами. В ходе работы проведён детальный анализ предметной области, включающий изучение исторического развития авиационных информационных систем и современных веб-платформ авиаперевозчиков. На основе полученных данных сформулированы функциональные и технические требования к создаваемой системе.

В процессе проектирования была построена логическая и физическая модель базы данных. Для описания структуры использовалась модель «сущность-связь» в нотации Питера Чена, что позволило определить основные сущности, их атрибуты и взаимосвязи. На основании разработанной логической модели реализована физическая схема базы данных, включающая таблицы пользователей, рейсов, самолётов, бронирований, платежей, уведомлений и обращений в службу поддержки. Это обеспечило целостность данных и исключило их дублирование.

Далее выполнено проектирование функциональной части сайта. Были построены диаграммы вариантов использования, классов и алгоритмов, описывающие основные сценарии взаимодействия пользователей с системой: регистрацию, поиск и бронирование авиабилетов, оплату заказов, работу с личным кабинетом и обращениями в поддержку. *UML*-диаграммы позволили формализовать архитектуру программного обеспечения и отразить ключевые связи между компонентами системы.

В процессе разработки реализована база данных на платформе *PostgreSQL* с использованием облачного сервиса *Supabase*, а также серверная часть веб-приложения на языке *JavaScript* с применением *Node*.*js* и *Express*.*js*. Выбранный технологический стек обеспечил высокую производительность, гибкость и масштабируемость решения.

В результате проделанной работы создана функциональная модель вебсайта авиакомпании, объединяющая возможности современных систем бронирования и администрирования данных. Разработанная система демонстрирует применение принципов объектно-ориентированного анализа и проектирования, методов нормализации данных и приёмов построения архитектуры клиент-серверных приложений. Реализация проекта позволила закрепить теоретические знания и получить практические навыки разработки информационных систем с использованием *UML*-моделирования, *SQL* и современных веб-технологий.

# **Список использованных источников**

[1] Шильдс, У. *SQL*: быстрое погружение : учебное пособие / У. Шильдс. – Санкт–Петербург : Питер, 2022. – 224 с.

[2] Сытин, Л. Е. Легендарная гражданская авиация мира : учеб.– метод. пособие / Л. Е. Сытин. – Санкт–Петербург : Полигон, 2012. – 160 с.

[3] Веб-сайт авиакомпания «Белавиа» [Электронный ресурс]. – Режим доступа: *https*://*www*.*belavia*.*by*. – Дата доступа: 04.11.2025.

[4] Веб-сайт авиакомпания «Аэрофлот» [Электронный ресурс]. – Режим доступа: *https*://*www*.*aeroflot*.*ru*. – Дата доступа: 04.11.2025.

[5] Веб-сайт авиакомпания «Аэрофлот» [Электронный ресурс]. – Режим доступа: *https*://*www*.*aeroflot*.*ru*. – Дата доступа: 04.11.2025.

[6] Веб-сайт авиакомпания «Аэрофлот» [Электронный ресурс]. – Режим доступа: *https*://*www*.*aeroflot*.*ru*. – Дата доступа: 04.11.2025. Якобсон, А. Унифицированный процесс разработки программного обеспечения / А. Якобсон, Г. Буч. – Санкт–Петербург : Питер, 2002. – 496 *c*.

[7] Кузин, А. В. Базы данных : учеб.-метод. пособие / А. В. Кузин, С. В. Левонисова. – Москва : Академия, 2005. – 320 с.

[8] Орлов, С. А. Технологии разработки программного обеспечения: разработка сложных программных систем : учеб. Пособие / С. А. Орлов. – 3-е изд. – Санкт-Петербург : Питер, 2004. – 527 с.

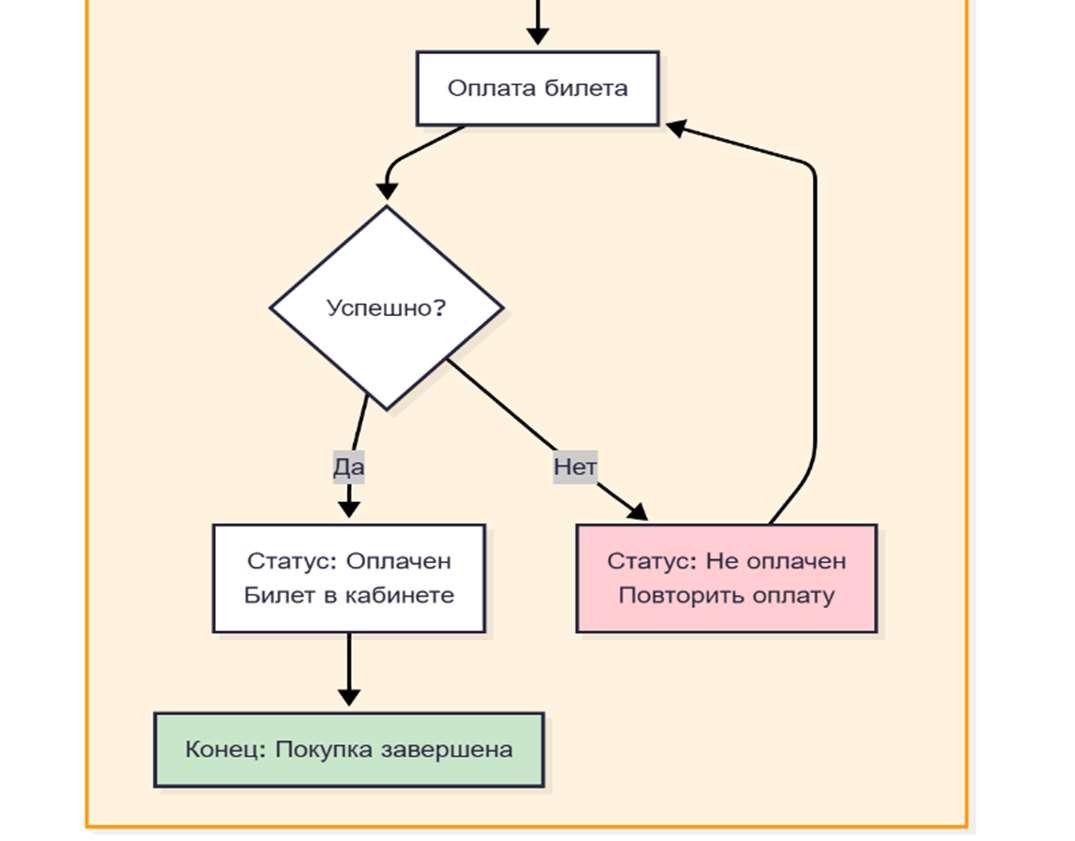
[9] Ульман, Л. *MySQL*: руководство по изучению языка / Л. Ульман. – Москва : ДМК Пресс, 2004. – 352 с.

[10] Рамбо, Дж. *UML* 2.0. Объектно-ориентированное моделирование и разработка / Дж. Рамбо, М. Блаха. – 2-у изд. – Санкт-Петербург : Питер, 2007. – 544 с.

# **ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Схема алгоритма покупки билета**



# **ПРИЛОЖЕНИЕ Б**

**(обязательное)**

**Листинг кода аналитической функции**

*const getAnalytics = async (period) => { try { const intervalMap = { day: 'day', week: 'week', month: 'month', year:*

*'year' }; const intervalUnit = intervalMap[period] || 'year'; const interval = `1 ${intervalUnit}`;*

*// Основные метрики (booked – общее бронирований за период, кроме cancelled) const metricsQuery = `*

*SELECT*

*(SELECT COUNT(\*) FROM flights WHERE created\_at >= NOW() - INTERVAL '${interval}') as flights,*

*(SELECT COUNT(\*) FROM bookings WHERE created\_at >= NOW() - INTERVAL '${interval}' AND status IN ('pending', 'paid', 'history')) as booked, -- общее бронирований*

*(SELECT COUNT(\*) FROM bookings WHERE status = 'paid' AND created\_at >= NOW() - INTERVAL '${interval}') as purchased,*

*(SELECT COUNT(\*) FROM users WHERE date\_trunc('day', created\_at) >= date\_trunc('day', NOW()) - INTERVAL '1 ${intervalUnit}') as newUsers,*

*-- Доход эконом (SUM(f.price \* b.passengers) для paid)*

*(SELECT COALESCE(SUM(f.price \* b.passengers), 0) FROM*

*bookings b*

*JOIN flights f ON b.flight\_id = f.id*

*WHERE b.status = 'paid' AND b.created\_at >= NOW() - INTERVAL*

*'${interval}'*

*AND f.flight\_class = 'economy') as economy\_revenue,*

*-- Доход бизнес (аналогично)*

*(SELECT COALESCE(SUM(f.price \* b.passengers), 0) FROM*

*bookings b*

*JOIN flights f ON b.flight\_id = f.id*

*WHERE b.status = 'paid' AND b.created\_at >= NOW() - INTERVAL*

*'${interval}'*

*AND f.flight\_class = 'business') as business\_revenue,*

*-- Обращения (если support\_tickets есть)*

*(SELECT COUNT(\*) FROM support\_tickets WHERE created\_at >= NOW() - INTERVAL '${interval}') as support\_booking\_issues*

*`;*

*const metrics = await query(metricsQuery); const metricsData = metrics.rows[0] || { flights: 0, booked: 0, purchased: 0, newUsers: 0, economy\_revenue: 0, business\_revenue: 0, support\_booking\_issues: 0*

*};*

*// График рейсов (без изменений) const flightsChartQuery = `*

*SELECT date\_trunc('day', created\_at) as day, COUNT(\*) as count*

*FROM flights WHERE created\_at >= NOW() - INTERVAL '${interval}'*

*GROUP BY day ORDER BY day*

*`;*

*const flightsChart = await query(flightsChartQuery);*

*const flightsLabels = flightsChart.rows.map(r => r.day.toISOString().split('T')[0]); const flightsData = flightsChart.rows.map(r => r.count);*

*// График бронирований (все брони за период, с классом из f.flight\_class) const bookingsChartQuery = `*

*SELECT date\_trunc('day', b.created\_at) as day, f.flight\_class as class,*

*COUNT(\*) as count*

*FROM bookings b JOIN flights f ON b.flight\_id = f.id*

*WHERE b.created\_at >= NOW() - INTERVAL '${interval}' AND*

*b.status IN ('pending', 'paid', 'history')*

*GROUP BY day, class ORDER BY day*

*`;*

*const bookingsChart = await query(bookingsChartQuery); const bookingsByDay = {}; bookingsChart.rows.forEach(r => {*

*const dayKey = r.day.toISOString().split('T')[0]; if (!bookingsByDay[dayKey]) bookingsByDay[dayKey] = { economy: 0,*

*business: 0 }; const classKey = r.class.toLowerCase(); bookingsByDay[dayKey][classKey] =*

*(bookingsByDay[dayKey][classKey] || 0) + r.count;*

*});*

*const sortedDays = Object.keys(bookingsByDay).sort(); const bookingsLabels = sortedDays;*

*const bookingsEconomyData = sortedDays.map(d =>*

*bookingsByDay[d].economy || 0);*

*const bookingsBusinessData = sortedDays.map(d =>*

*bookingsByDay[d].business || 0);*

*return { success: true, metrics: metricsData,*

*charts: { flightsLabels, flightsData, bookingsLabels, bookingsData: {*

*economy: bookingsEconomyData, business: bookingsBusinessData } }*

*};*

*} catch (error) { console.error('Ошибка в getAnalytics:', error); return { success: false, message: error.message };*

*}*

*};*

*// Экспорт в DOCX (обновлено: добавлены таблицы, графики как текст, новые метрики; требует npm i docx) const exportAnalyticsToDocx = async (period) => { try {*

*const data = await getAnalytics(period); if (!data.success) { throw new Error(data.message || 'Ошибка получения данных');*

*}*

*const { Document, Packer, Paragraph, TextRun, Table, TableRow, TableCell, WidthType, AlignmentType } = require('docx');*

*const doc = new Document({ sections: [{ properties: {}, children: [*

*// Заголовок new Paragraph({ children: [new TextRun({ text: `Аналитика за ${period.toUpperCase()} (по состоянию на*

*${new Date().toLocaleDateString('ru-RU')})`,*

*bold: true, size: 28*

*})]*

*}),*

*// Основные метрики new Paragraph({ children: [new TextRun('Основные показатели:')]*

*}),*

*new Paragraph({ children: [new TextRun(`Количество рейсов:*

*${data.metrics.flights}`)] }), new Paragraph({ children: [new TextRun(`Забронировано билетов:*

*${data.metrics.booked}`)] }), new Paragraph({ children: [new TextRun(`Куплено билетов:*

*${data.metrics.purchased}`)] }), new Paragraph({ children: [new TextRun(`Новых пользователей:*

*${data.metrics.newUsers}`)] }), new Paragraph({ children: [new TextRun(`Доход эконом:*

*${data.metrics.economy\_revenue} ₽`)] }), new Paragraph({ children: [new TextRun(`Доход бизнес:*

*${data.metrics.business\_revenue} ₽`)] }), new Paragraph({ children: [new TextRun(`Обращений по брони: ${data.metrics.support\_booking\_issues}`)] }),*

*// Таблица для графика рейсов (как текст) new Paragraph({ children: [new TextRun('График рейсов по*

*дням:')] }), new Table({*

*width: { size: 100, type: WidthType.PERCENTAGE }, alignment: AlignmentType.CENTER, rows: [ new TableRow({ children: [ new TableCell({ children: [new Paragraph('Дата')] }), new TableCell({ children: [new Paragraph('Количество')] })*

*]*

*}),*

*...data.charts.flightsLabels.map((label, i) => new TableRow({ children: [ new TableCell({ children: [new Paragraph(label)] }),*

*new TableCell({ children: [new*

*Paragraph(data.charts.flightsData[i].toString())] })*

*]*

*})*

*)*

*]*

*}),*

*// Таблица для графика бронирований по классам new Paragraph({ children: [new TextRun('График бронирований по*

*дням и классам:')] }), new Table({*

*width: { size: 100, type: WidthType.PERCENTAGE }, alignment: AlignmentType.CENTER, rows: [ new TableRow({ children: [*

*new TableCell({ children: [new Paragraph('Дата')] }), new TableCell({ children: [new Paragraph('Эконом')] }), new TableCell({ children: [new Paragraph('Бизнес')] })*

*]*

*}),*

*...data.charts.bookingsLabels.map((label, i) => new TableRow({ children: [*

*new TableCell({ children: [new Paragraph(label)] }),*

*new TableCell({ children: [new*

*Paragraph((data.charts.bookingsData.economy[i] || 0).toString())] }),*

*new TableCell({ children: [new*

*Paragraph((data.charts.bookingsData.business[i] || 0).toString())] })*

*]*

*})*

*)*

*]*

*})*

*]*

*}]*

*});*

*const buffer = await Packer.toBuffer(doc); return buffer; } catch (error) {*

*console.error('Ошибка в exportAnalyticsToDocx:', error); throw error; // Перебрасываем для обработки в роуте*

*}*

*};*

# **Ведомость курсового проекта**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | | | | | *Наименование* | *Дополнительные сведения* | | | | |
|  | | | | | *Текстовые документы* |  | | | | |
|  | | | | |  |  | | | | |
| *БГУИР КР 6-05-0612-03 211 ПЗ* | | | | | *Пояснительная записка* | *33 с.* | | | | |
|  | | | | |  |  | | | | |
|  | | | | | *Графические документы* |  | | | | |
|  | | | | |  |  | | | | |
| *ГУИР 320604.001 ПД* | | | | | *Диаграмма* | *Формат А1* | | | | |
|  | | | | | *последовательности* |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  |  |  |  |  | *БГУИР КР 6-05-0612-03 211 ПЗ* | | | | | |
|  |  |  |  |  |
|  |  |  |  |  |
| *Изм.* | *Л* | *№ докум.* | *Подп.* | *Дата* | *Проектирование программного обеспечения аренды рекламы в метро*  *Ведомость курсового проекта* | *Лит* | | | *Лист* | *Листов* |
| *Разраб.* | | *Ясинский* |  |  |  | *Т* |  | *33* | *33* |
| *Провер.* | | *Протченко* |  |  | *Кафедра ИТАС*  *гр. 320604* | | | | |
| *Т.контр.* | |  |  |  |
| *Н.контр.* | |  |  |  |
| *Утв.* | |  |  |  |
| *Реценз.* | |  |  |  |