Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Объектно-ориентированное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему

СОЗДАНИЕ API ДЛЯ РАЗРАБОТКИ ДВУМЕРНЫХ ИГР ДЛЯ ОПЕРАЦИОННЫХ СИСТЕМ Windows И Linux

БГУИР КП 1-40 04 01 01 009

Студент Тимофеев К.А.

Руководитель Рогов М.Г.

Минск 2023

СОДЕРЖАНИЕ

Введение 4

1 Постановка задачи 5

1.1 Анализ рынка и пользователей 5

1.2 Анализ существующих аналогов 5

1.3 Функциональные требования 6

2 Теоретическое обоснование разработки 7

2.1 Технологии программирования 7

2.2 Паттерны программирования 8

3 Архитектура разработанной системы 9

3.1 Структура и архитектура серверной части 9

3.2 Структура и архитектура тестовой игры….................................................. 11

3.3 Структура классов ............................................................................................. 13

4 Функциональные возможности программы 14

4.1 Описание функций программного обеспечения 14

4.2 Дополнительные функции приложения 23

Заключение 24

Список использованных источников 25

Приложение А (обязательное) Исходный код ...................................................... 26

**ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ**

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Паттерн проектирования – повторяемая архитектурная конструкция в сфере проектирования программного обеспечения, предлагающая решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Архитектура программного средства – это структурная организация и компоненты программного средства, их взаимосвязи и взаимодействие, которые определяют его общую форму и функциональность. Она описывает различные аспекты программного средства, такие как его компоненты, модули, слои, интерфейсы, взаимодействие между ними, принципы организации кода и данные.

Архитектура разработанной системы – это описание структуры, организации и компонентов системы, которая была разработана на основе определенной архитектурной концепции или дизайна. Она представляет собой реализацию архитектурных решений, учитывающих функциональные и нефункциональные требования системы.

Simple DirectMedia Layer (или коротко SDL) — свободная кроссплатформенная мультимедийная библиотека, реализующая единый программный интерфейс к графической подсистеме, звуковым устройствам и средствам ввода для широкого спектра платформ. SDL API доступны для языков программирования C, C++, C#, Java и многих других.

Use-case диаграмма – диаграмма, отражающая отношения между группами пользователей и являющаяся составной частью модели прецедентов, позволяющей описать систему на концептуальном уровне.

UML-диаграмма – это схема, нарисованная с применением символов UML. Она может содержать множество элементов и соединений между ними.

**ВВЕДЕНИЕ**

### Игровая индустрия активно развивается, поэтому актуален вопрос в разработке специализированных API для разработки игр – игровых движков.

### Игровой движок – это центральный программный компонент компьютерных и видео игр или других интерактивных приложений с графикой, обрабатываемой в реальном времени. Он обеспечивает основные технологии, упрощает разработку и часто даёт игре возможность запускаться на нескольких платформах, таких как игровые консоли и настольные операционные системы, например, Linux, Mac OS X и Microsoft Windows. Основную функциональность обычно обеспечивает игровой движок, включающий движок рендеринга («визуализатор») для 2D или 3D графики, систему скриптов, управление памятью и графические сцены.

### Игровые движки помогают упростить разработку игр по многим направлениям. Они позволяют неоднократно использовать код, разделить разработку игры по направлениям (разработка графической части, программирование скриптов, разработка и модифицирование самого движка).

### 2D игры всегда были популярны. Во времена начала игровой индустрии все игры были двумерными, но в конце 90-ых с развитием 3D-графики они на время потеряли популярность. Однако с развитием игровой индустрии и появлением свободно распространяемых игровых движков они вновь обрели популярность, перейдя из категории крупнобюджетных проектов в сегмент игр от независимых разработчиков с малым бюджетом. Еще один скачок популярности 2D игр пришелся на начало 10-ых, когда технологии позволили запускать игры в браузерах и в телефонах.

**1 ПОСТАНОВКА ЗАДАЧИ**

* 1. **Анализ рынка и пользователей**

Основную долю рынка свободно распространяемых игровых движков занимают Unity и Unreal Engine. Они имеют большое количество встроенного функционала, активно поддерживаются разработчиками и регулярно получают обновления, добавляющие передовые технологии в области графики. Они оба имеют бесплатные версии с немного ограниченным функционалом, что существенно облегчает обучение новых специалистов за счет широкого сообщества. Так же они поддерживают разработку 2D игр, хоть и в основе всех технологий лежит работа с трехмерной графикой.

Серьезную долю занимают движки Source и CryEngine. Они общедоступны, но менее популярны и используются в основном для разработки модификаций к уже вышедшим на этих движках играм. Оба не поддерживают разработку 2D игр.

Набирает популярность многоцелевой игровой движок Godot. К его достоинствам можно отнести открытый исходный код, возможность разработки 2D игр, активную поддержку и быстро развивающееся сообщество.

Так же существуют узконаправленные движки для разработки конкретного жанра игр – RPG Maker(jRpg) , Ren'Py (визуальные новеллы), jMonkeyEngine (платформеры) и многие другие.

* 1. **Анализ существующих аналогов**

Самым популярным движком в разработке 2D игр считается Unity. В качестве языка для написания скриптов в нем используется C#. Для работы с Unity необходимо установить официальный редактор. Система скриптов вместе с возможностями языка C# позволяют легко настраивать поведение игровых объектов и настраивать связи между объектами. В специальном режиме есть возможность запуска игры без отключения большинства возможностей редактора, что дает возможность корректировать поведение объектов для лучшего игрового опыта. По умолчанию представлена широкая библиотека компонентов для работы с физикой и светом. Редактор позволяет проводить диагностику используемой памяти, легко добавлять и сохранять игровые объекты. Однако Unity является в первую очередь 3D-движком, поэтому ресурсы компьютера используются не самым эффективным образом, некоторые встроенные модули не адаптированы по работу в 2D. Разрабатываемый в данном проекте движок будет приспособлен для работы только с 2D играми, что позволит увеличить производительность и упростить разработку.

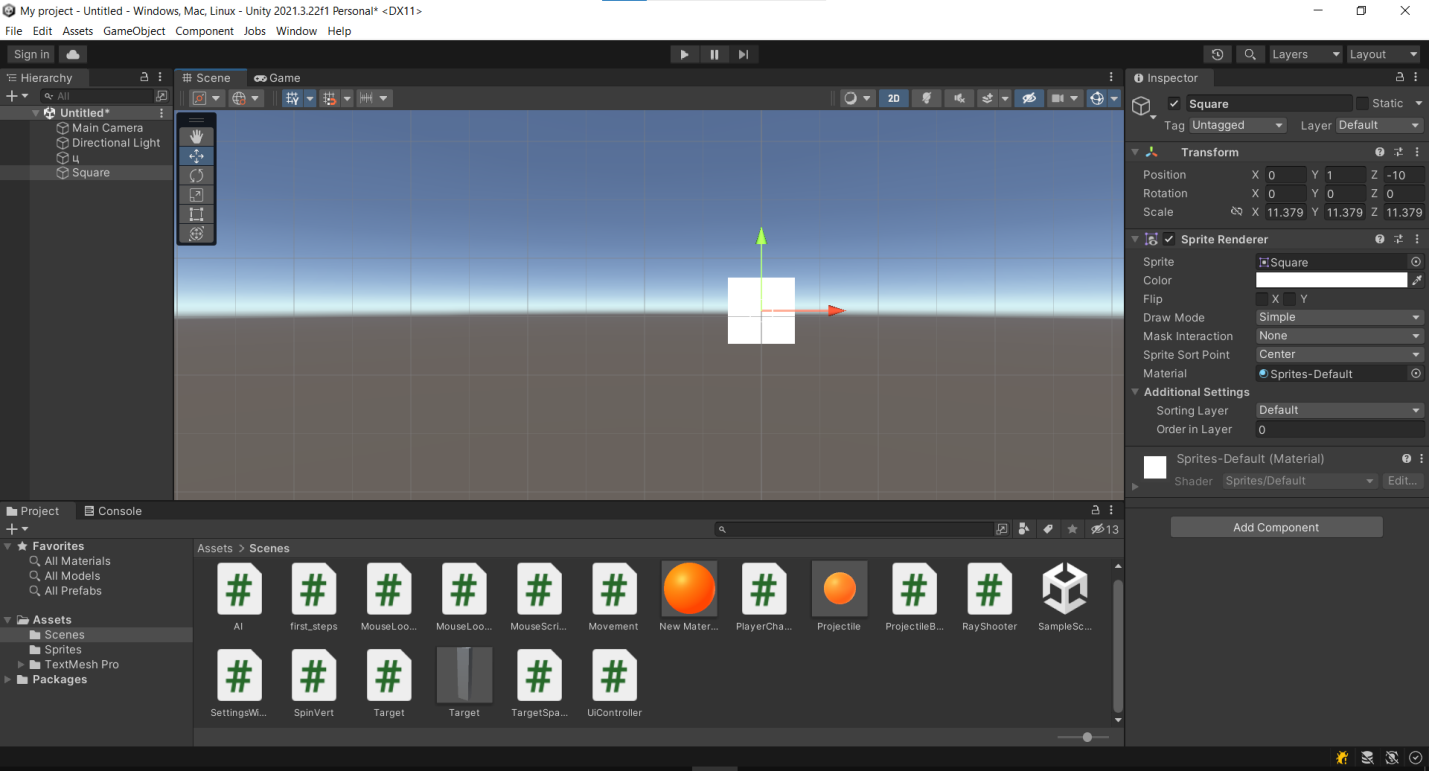


Рисунок 1.1 –Интерфейс редактора Unity

* 1. **Функциональные требования**

Движок должен позволять создавать игровые сцены как в ручную через код, так и при выгрузке из файла. Должны быть реализованы система скриптов, позволяющая менять поведение игровых объектов во время выполнения, система контроля используемых ресурсов, система родитель-потомок для связей между игровыми объектами. Необходима возможность настройки реагирования программы на нажатие клавиш. Техническое изложение предоставляемого функционала можно увидеть на Use-case диаграмме (см. рисунок 1.2):

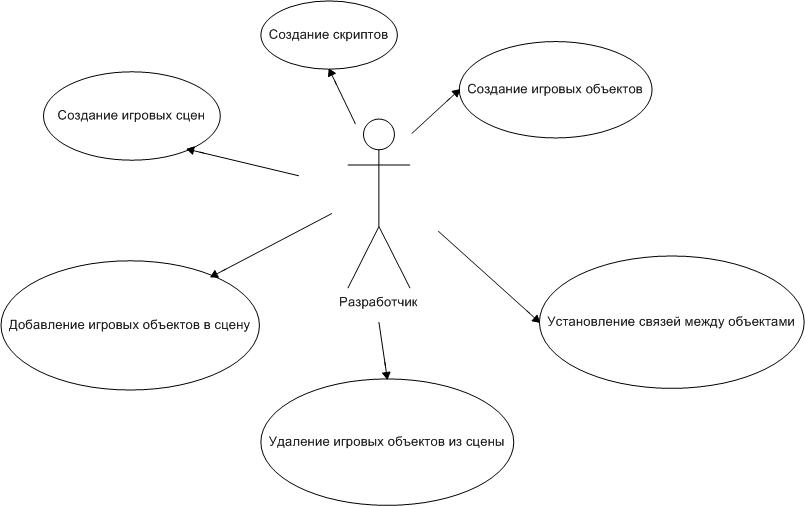


Рисунок 1.2 – Use-case диаграмма приложения

**2 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ**

**2.1 Технологии программирования**

Для решения поставленных задач был выбран язык программирования С++. Данный выбор позволит добиться требуемых результатов в производительности. Так же это дает возможность использовать огромное количество библиотек, написанных для языка программирования С.

Для вывода изображения на экран была использована кроссплатформенная библиотека SDL. Данная библиотека позволяет не только выводить изображение на экран, но и считывать нажатия клавиш, выводить звуковые файлы и видео.

Будут использованы различные технологии программирования, основанные на принципах ООП и SOLID.

Классы используют инкапсуляцию для скрытия внутренней реализации и данных от внешнего доступа. Они определяют приватные поля и методы для обеспечения контролируемого доступа к данным и выполнения операций.

Использование принципа абстракции позволило скрыть детали реализации объектов и предоставить только необходимый интерфейс для взаимодействия с ними. Внутренние механизмы работы объектов остаются скрытыми и недоступными для внешнего кода. Это позволяет изменять или улучшать реализацию объектов без влияния на клиентский код, который взаимодействует только с их абстракцией.

Помимо этого, в проекте был соблюден принцип единственной ответственности (Single Responsibility Principle, SRP), поскольку каждый класс имеет четко определенную задачу и отвечает только за свою функциональность.

Соблюдается принцип открытости/закрытости (Open/Closed Principle, OCP), который позволяет добавлять новую функциональность без изменения существующего кода.

Соблюдается принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP). Этот принцип устанавливает, что объекты в программе могут быть заменены их наследниками без изменения свойств программы. Другими словами, объекты наследников могут использоваться вместо объектов базового класса без изменения поведения программы.

Таким образом, применение принципов инкапсуляции, полиморфизма, наследования и абстракции в сочетании с принципами SOLID в проекте позволяет создать гибкую и расширяемую архитектуру, где каждый класс имеет четко определенные обязанности и может быть легко модифицирован или заменен при необходимости.

**2.2 Паттерны программирования**

В основе программного продукта были использован паттерн Strategy.

Strategy – это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы. Его суть состоит в использовании ссылок на объекты, реализующие некоторый интерфейс, обеспечивающий необходимую функциональность.

**3 АРХИТЕКТУРА РАЗРАБОТАННОЙ СИСТЕМЫ**

**3.1 Структура и архитектура серверной части**

Главным объектом является объект класса Game. Он хранит в себе текущую игровую сцену, ресурсы необходимые для отрисовки и объект специального класса ResourceManager, необходимого для регистрации загруженных ресурсов. Двумя главными методами класса Game являются Run() и setCurrentScene(). Run() запускает основной цикл игры, а setCurrentScene() устанавливает игровую сцену.

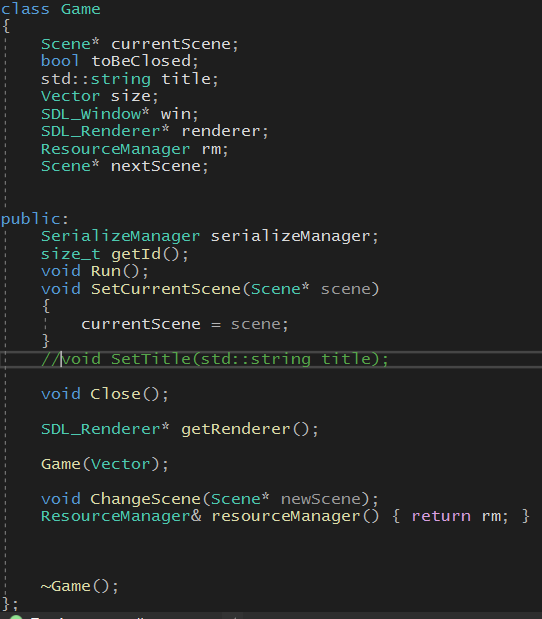


Рисунок 3.1 – Заголовочный файл класса Game

Объекты класса Scene представляют собой игровую сцену и является в сути коллекцией объектов класса GameObject. Объекты класса Scene отвечают за их отрисовку и удаление из памяти. Имеются методы доступа к конкретным объектам через уникальный идентификатор и метод доступа к копии коллекции игровых объектов. В качестве коллекции для хранения ссылок на игровые объекты был выбран класс std::list из стандартной библиотеки шаблонов языка C++. Выбор обусловлен регулярными вставками, удалениями и отсутствием необходимости быстрого доступа по индексу.

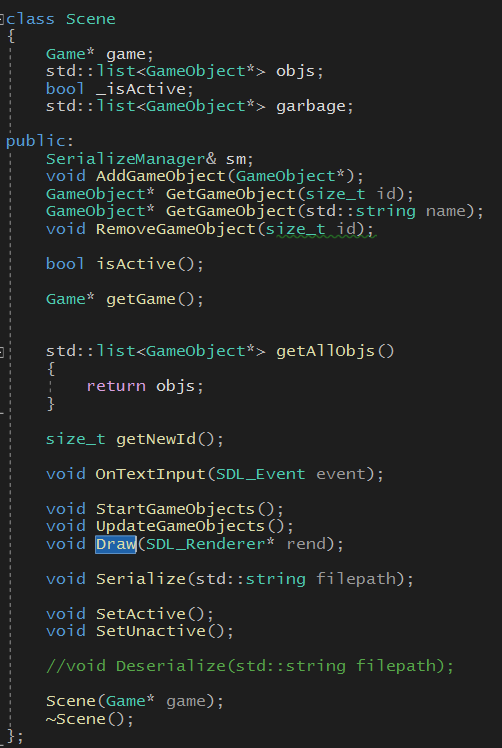


Рисунок 3.2 – Заголовочный файл класса Scene

Объекты класса GameObject хранят в себе коллекцию скриптов, объект специального класса Texture для хранения текстур, маркеры активности и видимости объекта, коллекцию потомков, ссылку на родительский игровой объект, ссылку на текущую сцену и объект структуры Transform, хранящей данные о положении игрового объекта в сцене.



Рисунок 3.3 – Заголовочный файл класса GameObject

Скрипты представляют собой классы, унаследованные от абстрактного класса Component. В этом классе имеются два виртуальных метода: Start() и Update(), которые необходимо переопределить в классе скрипта. Метод Start() вызывается при появлении компонента в игровом объекте, а метод Update() вызывается игровым объектом в каждом кадре прорисовки.

В игровом объекте хранится коллекция ссылок на объекты классов, унаследованных от класса Component. Однако есть необходимость в доступе к конкретному копоненту игрового объекта со всеми добавленными полями и методами, а не только к тем, чьё наличие гарантировано классом Component. Поэтому выбор для коллекции ссылок на компоненты пал на std::unordered\_map и стандартной библиотеки. Данный класс является реализацией хеш-таблицы, которая идеально подходит под поставленную задачу хранения элементов разных типов. Благодаря встроенной функции языка C++ typeid мы можем вычислить хеш некоторого типа данных. Таким образом при добавлении скрипта объекту мы используем шаблонную функцию AddComponent<T>(T\*), где T – тип, унаследованный от класса Component. В качестве хеша берется хеш типа T, благодаря чему имеется возможность хранить в одной коллекции объекты разных типов, не теряя возможности обращаться ко всем их полям и методам.

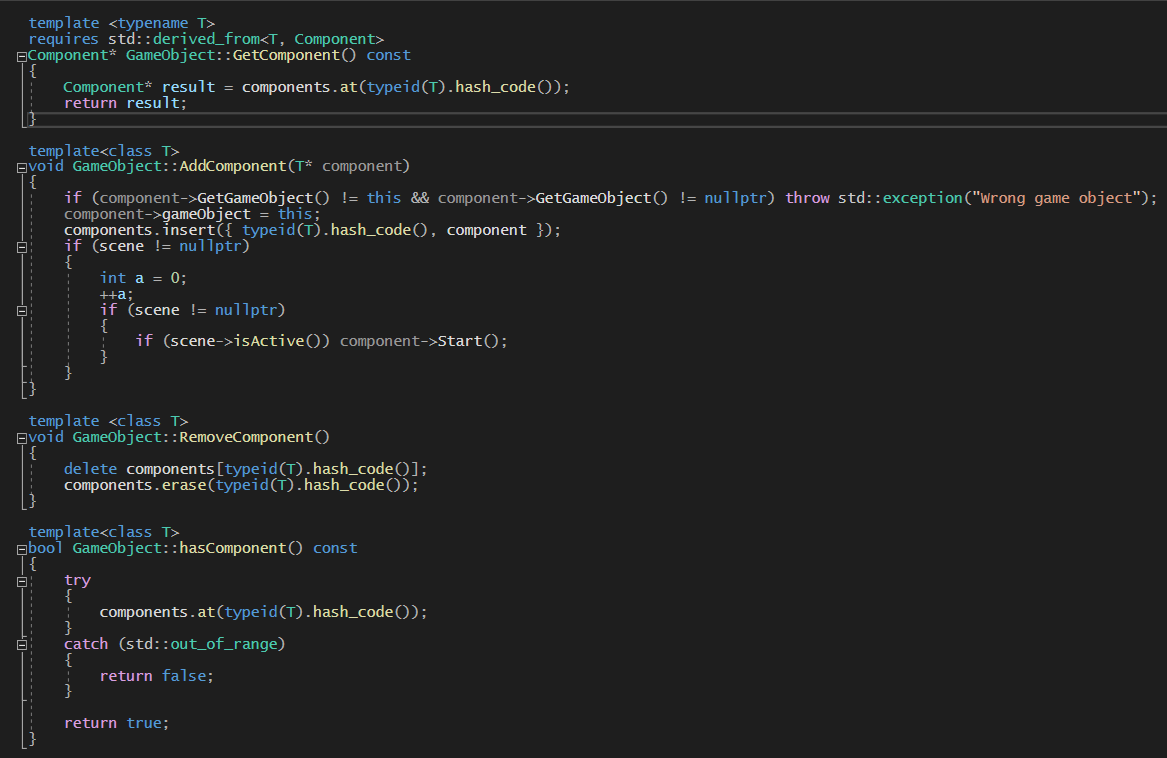


Рисунок 3.4 – Реализация работы с компонентами

Для сохранения объектов и сцен в файлах в соответствующие классы был добавлен метод Serialize(std::fstream). Аналогично с выгрузкой из файлов при помощи метода Deserialize(std::fstream). Однако при сохранении игровых объектов стоит вопрос о сохранении объектов скриптов в памяти. Если загрузку в память можно обеспечить добавлением в родительский класс Component виртуального метода для сериализации, то десериализовать объект не зная базовый тип из-за особенностей языка мы не сможем. Для этого был создан специальный класс SerializeManager. Он отвечает за сериализацию базовых типов данных и имеет в себе специальную таблицу функций, отвечающих за сериализацию и десериализацию объектов и скриптов. Данная таблица представляет собой объект std::unordered\_map. Хранение пары функций организовано по тому же принципу что и хранение компонентов (через хеш типа). Поэтому при необходимости сериализации объектов разработчику необходимо вручную зарегистрировать все используемые в сцене компоненты вызовом метода register\_component<T>().

Так же уделено внимание сериализации ссылок на другие объекты или компоненты. Напрямую сохранять объекты не имеет смысла, поэтому проблема была решена по-другому. При сериализации в файл сохраняется идентификатор объекта, на который делается ссылка. При десериализации в специальную коллекцию добавляются пара значений идентификаторов, представляющие пару держателя ссылки и объекта, на который ссылаются. Возможности языка С++ позволяют обращаться к участкам памяти программы, даже если они не являются доступными для текущего объекта. Поэтому мы можем хранить держателя ссылки в виде указателя на указатель на объект или компонент, что позволит потом их заполнить актуальными данными. Так как идентификаторы при десериализации будут изменены, пара значений старого и нового идентификаторов заносится в другую таблицу для дальнейшего использования. В конце десериализации, в держателей ссылок заносятся актуальные ссылки на компоненты и игровые объекты.

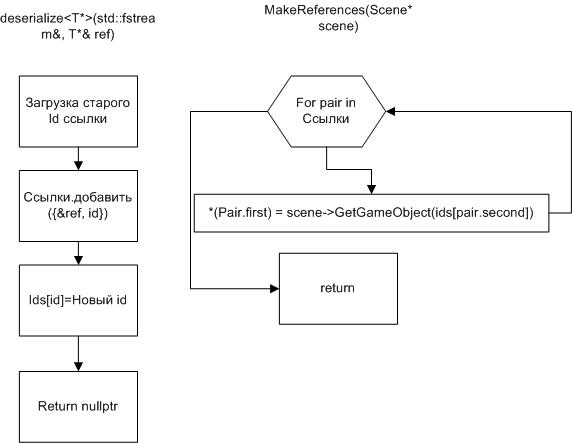


Рисунок 3.5 – Схема алгоритма десериализации ссылок

## **3.2 Структура и архитектура тестовой игры**

В качестве демонстрационного проекта была разработана игра, являющаяся аналогом «Arcanoid».

Представления пишутся на языке разметки HTML.

Также для стилизации был использован CSS. Он используется для зада-ния цветов, шрифтов, стилей, расположения отдельных блоков и других ас-пектов представления внешнего вида страниц. А также он помогает создавать достаточно простую анимацию, что делает взаимодействие пользователя с интерфейсом программы более приятным.

Структура и архитектура клиентской части системы обеспечивают разделение логики и представления, удобство управления пользовательским интерфейсом и возможность легкого изменения или добавления новых компонентов интерфейса. Благодаря такому подходу была достигнута высокая гибкость и масштабируемость приложения.

**3.3 Структура классов**

При разработке программного решения для приложения для воспроизведения музыки была применена архитектура, основанная на объектно-ориентированной парадигме программирования. Данная парадигма позволяет создать систему, где реальные сущности и их свойства могут быть точно описаны в виде объектов.

Детальнее с построением зависимостей между классами можно ознакомиться на UML-схеме (см. рисунок 3.10).

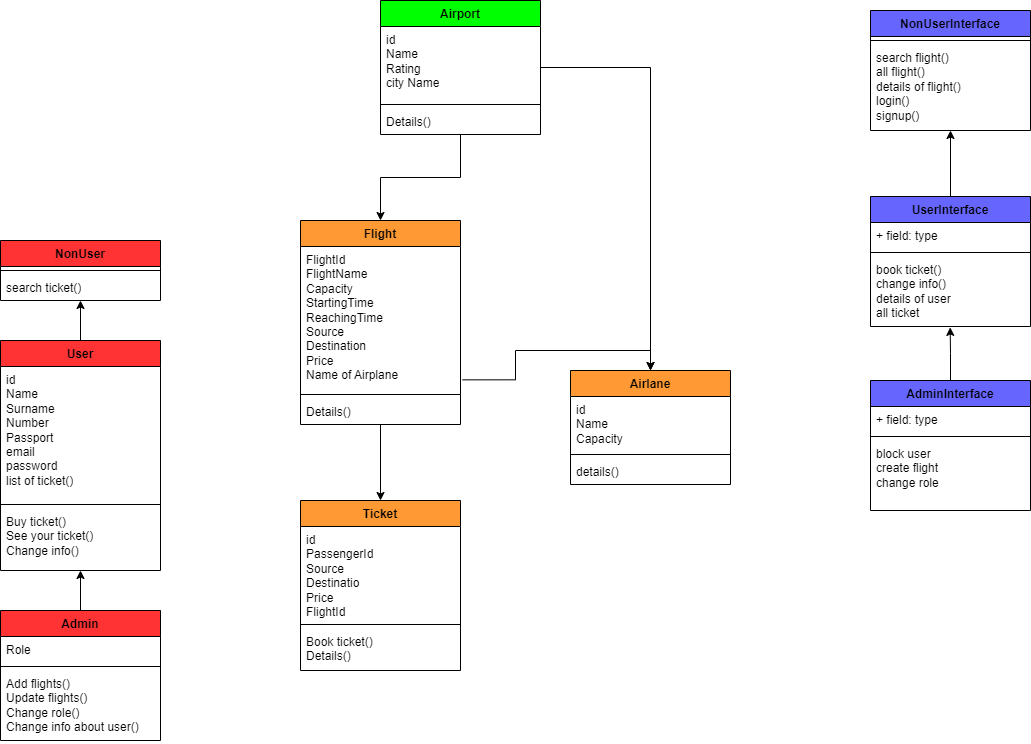


Рисунок 3.10 – UML-диаграмма классов

**4 ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ**

**4.1** **Описание функций программного средства**

Основной функционал программного средства включает в себя возможность аутентификации, регистрации пользователя, оформления заказа, просмотр доступных ритуальных урн и залов, свободных дат конкретного зала без необходимости аутентификации. Работник крематория имеет доступ к заказам всех пользователей, имеет возможность выполнить заказ. Администратор имеет доступ к добавлению, удалению и изменению ритуальных урн и залов, к списку всех пользователей приложения, к изменению роли пользователя. Так же основной функционал включает в себя хранение пользователей, заказов и товаров в базе данных.

При старте программы нас встречает начальная страница, на которой находится панель с поиском рейсов, а также выводом всех доступных рейсов. Поиск рейсов происходит по городу вылета и городу прилета (см. рисунок 4.1).

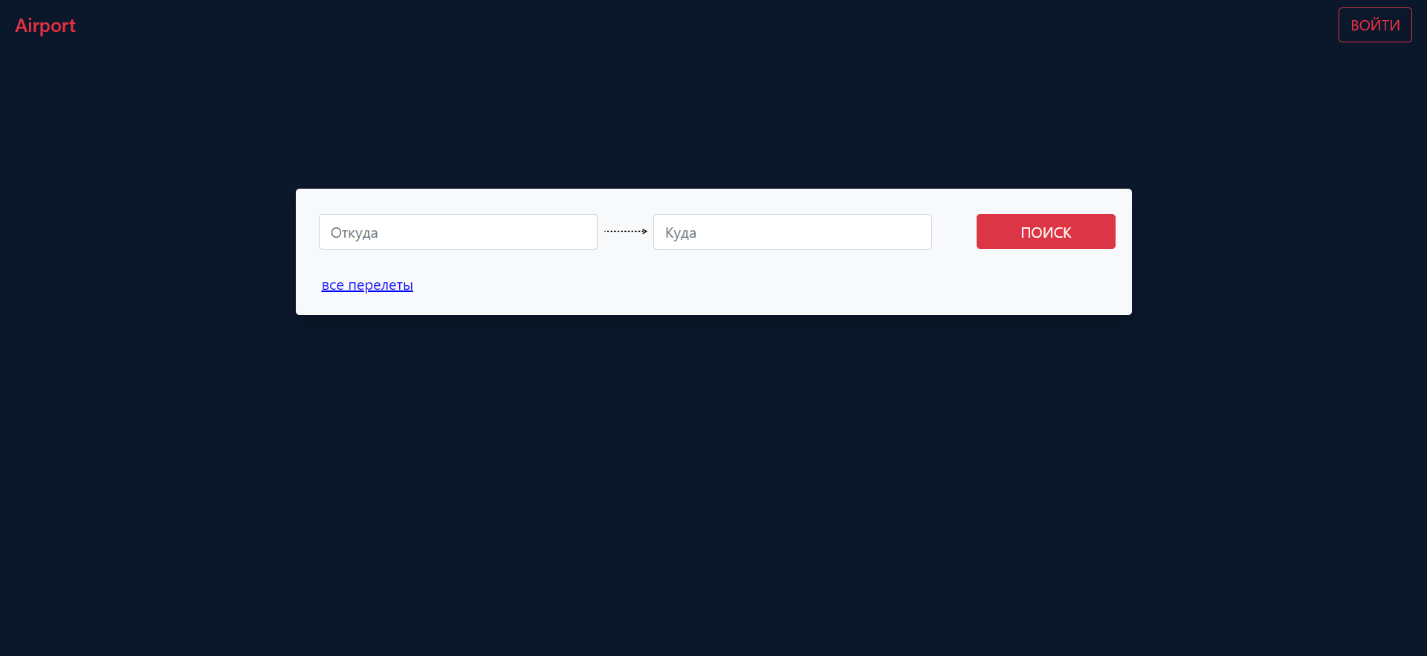


Рисунок 4.1 – Начальная страница

При нажатии кнопки аутентификации, пользователю открывается страница входа (см. рисунок 4.2), в котором пользователь может ввести данные своего аккаунта, отменить вход в аккаунт и вернуться на главную страницу, зарегистрироваться.

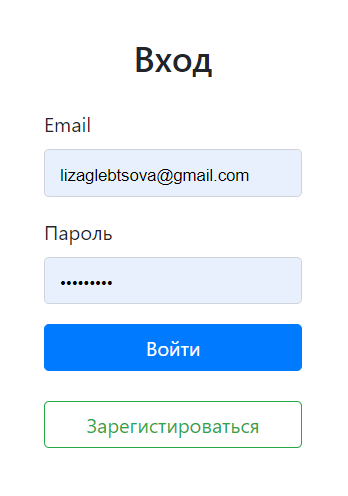


Рисунок 4.2 – Панель аутентификации

В случае регистрации открывается новая веб-страница для ввода полной информации об аккаунте (см. рисунок 4.3).

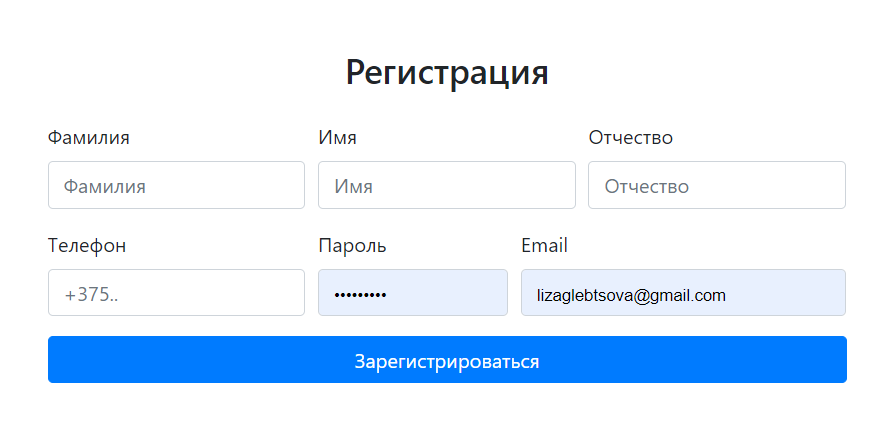


Рисунок 4.3 – Панель регистрации

В случае успешной аутентификации или регистрации нового пользователя, пользователь автоматически возвращается на начальную страницу (см. рисунок 4.4). Зарегистрированному пользователю открываются новые функции такие как профиль, изменение профиля и выход из аккаунта.

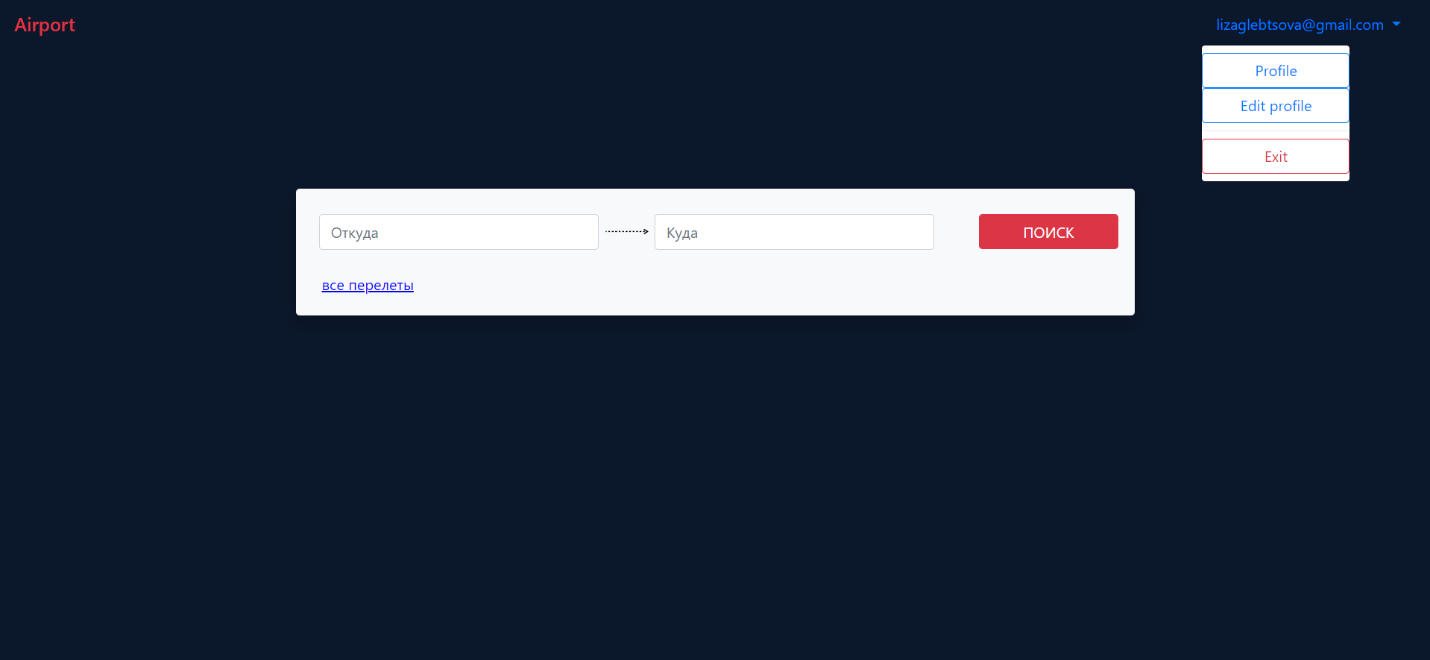


Рисунок 4.4 – Начальная страница авторизированного пользователя

Панель рейсов (см. рисунок 4.5) представляет собой список, в котором отображается краткая информация о рейсах.

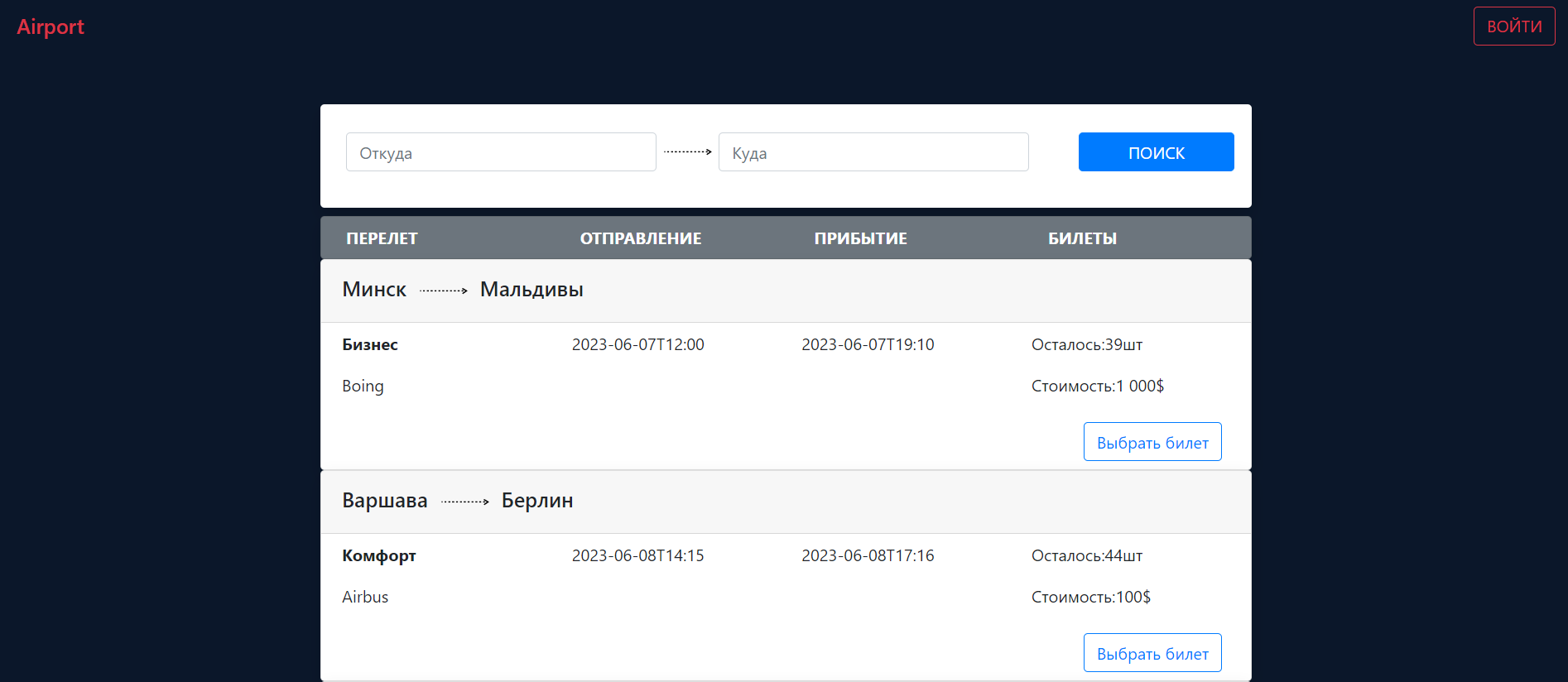


Рисунок 4.5 – Вывод всех рейсов

Для просмотра более подробной информации о рейсе необходимо нажать на кнопку выбрать билет после этого пользователь перейдет на веб-страницу с подробной информацией и с возможностью выбора места (см. рисунок 4.6).

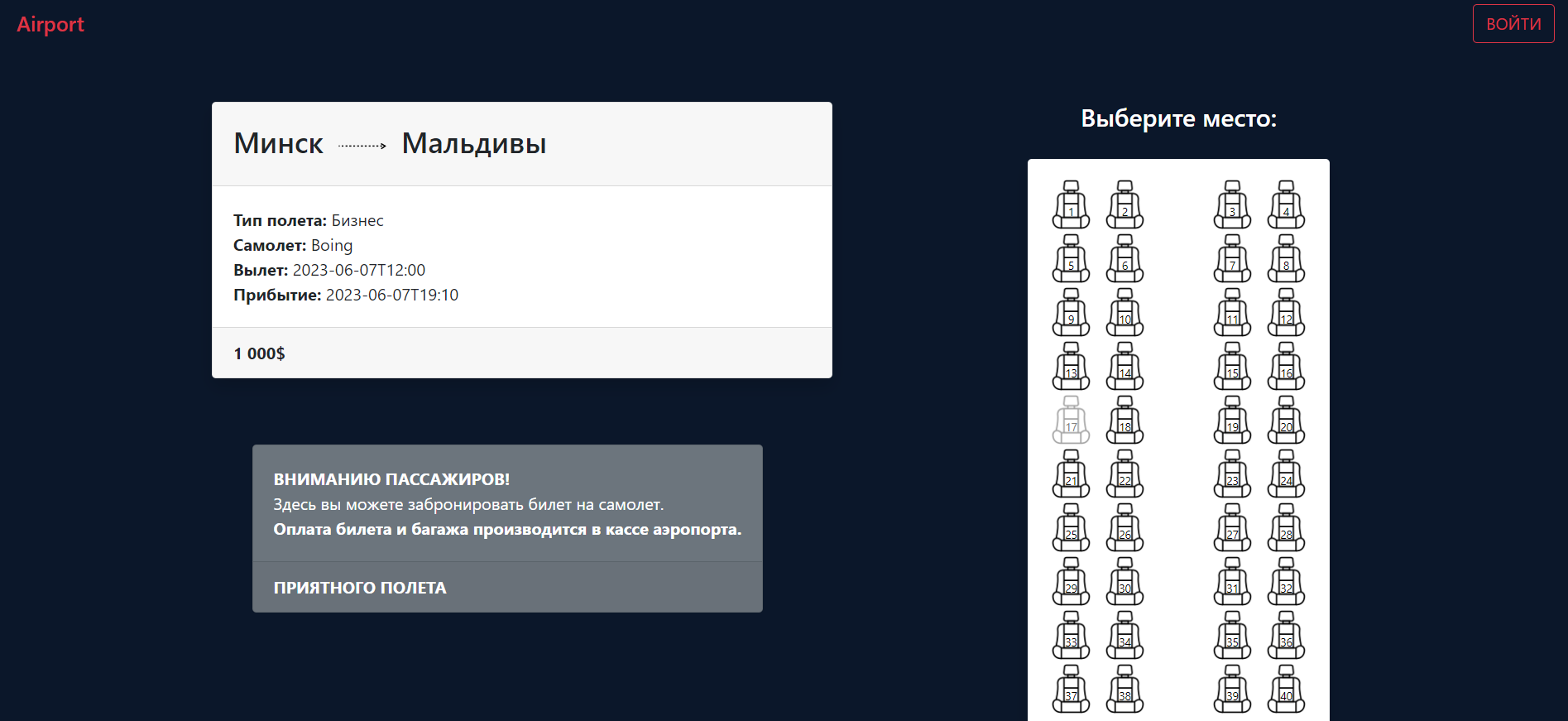


Рисунок 4.6 – Страница заказов пользователя

При нажатии на доступное место для не авторизированного пользователя откроется веб-страница с входом в аккаунт, для авторизированного пользователя откроется панель с подтверждением заказа.

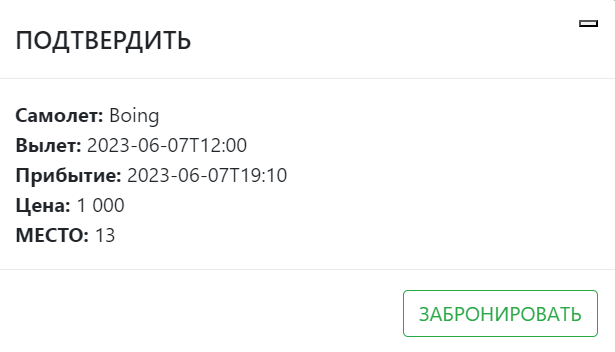


Рисунок 4.7 – Панель подтверждения заказа

На странице пользователя представлена вся личная информация (см. рисунок 4.8). А также предоставлена информация о запланированных рейсов с возможностью отменить заказ и история заказов.

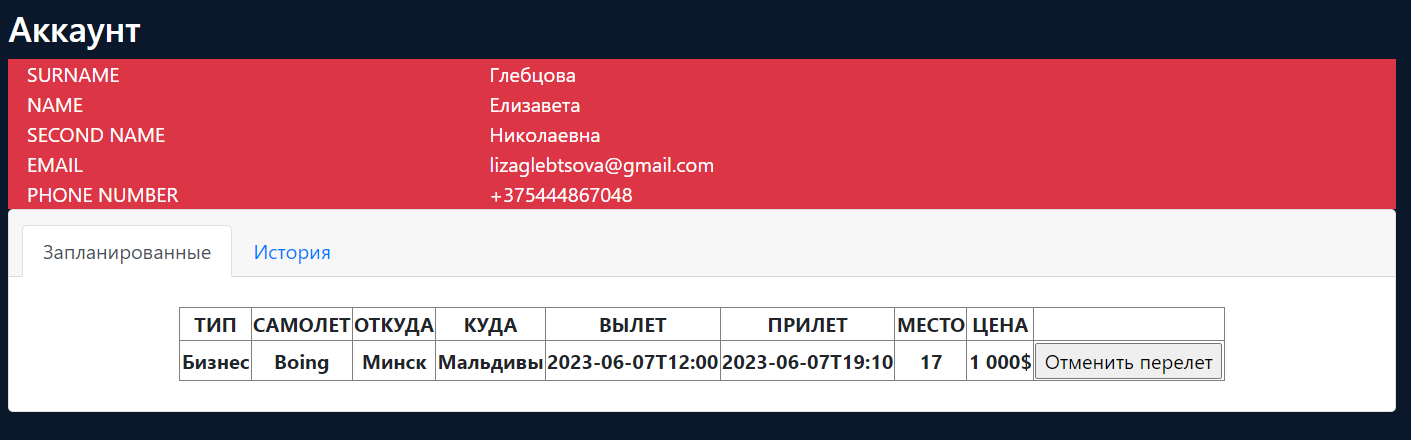


Рисунок 4.8 – Страница пользователя

Пользователь может изменить свои данные, для подтверждения необходимо будет ввести пароль.

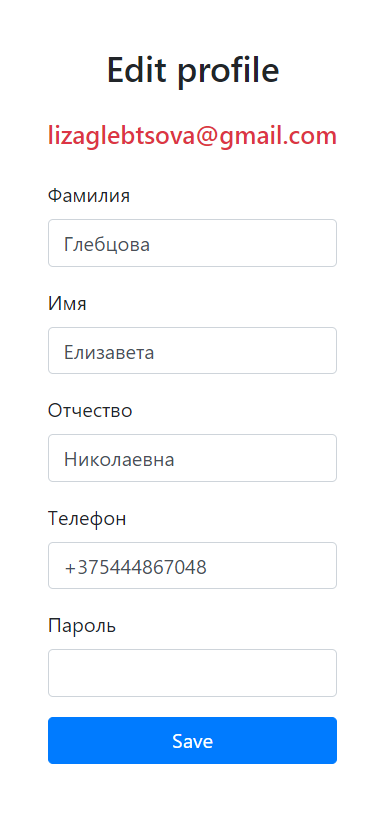


Рисунок 4.9 – Редактирование данных о пользователе

Администратор помимо общего интерфейса также имеет панель администратора (см. рисунок 4.10).

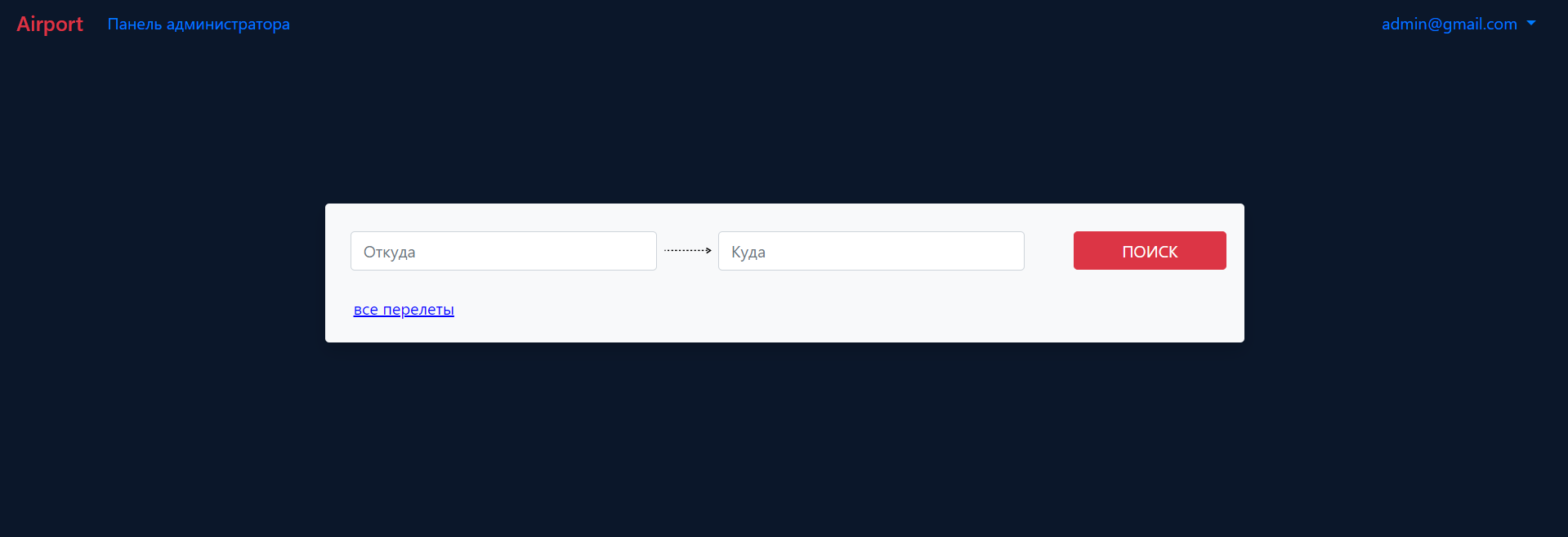


Рисунок 4.10 – Главная страница с панелью администратора

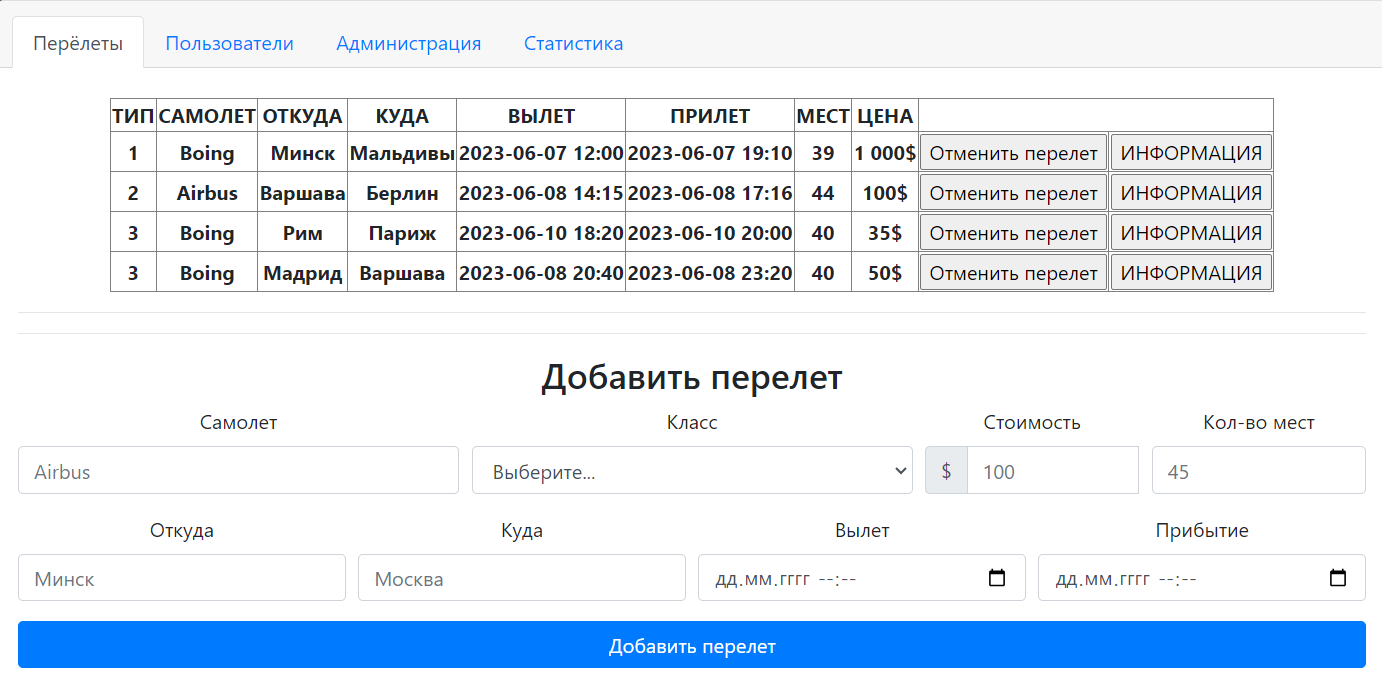


Рисунок 4.11 – Панель администратора

Панель администратора представлена в виде навигационной панели. Навигационная панель состоит из пунктов: перелеты, пользователи, администрация, статистика.

Администратор может добавлять и отменять рейсы. При добавлении рейсов администратор вводит информацию об названии самолета, тип посадочных мест, стоимости и количестве мест, а также город отправления, город назначения, дату и время вылета и прилета.

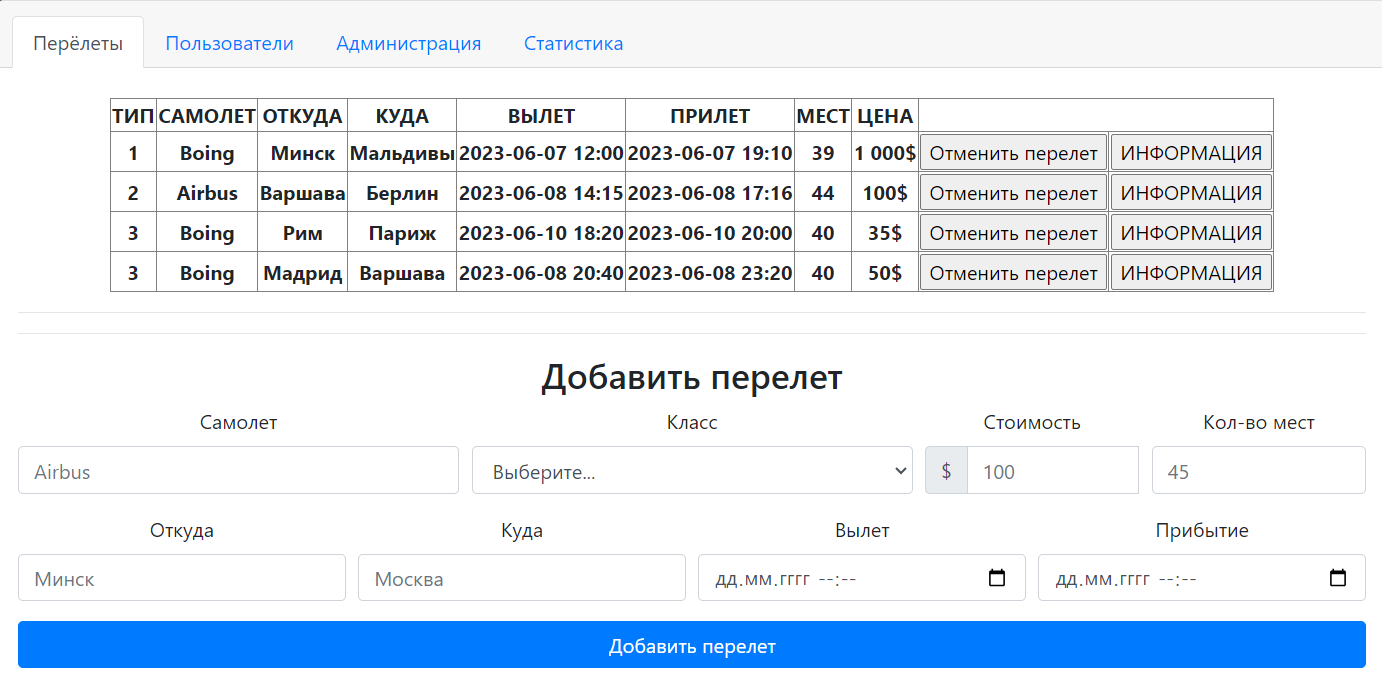


Рисунок 4.12 – Панель администратора. Перелеты

Также администратор может увидеть информацию о конкретном рейсе и отменить регистрацию на рейс определенного человека.

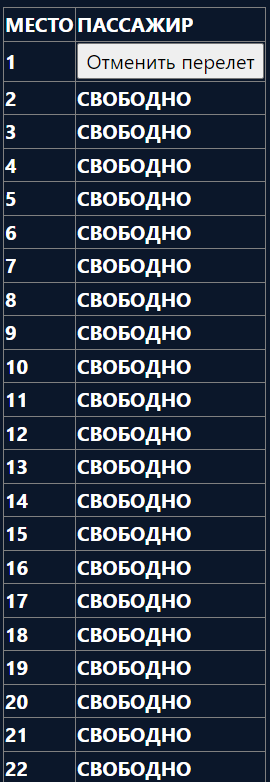


Рисунок 4.13 – Информация

Администратор может видеть и редактировать информацию о пользователях. А также администратор может блокировать пользователей, в таком случае пользователь не сможет зайти в аккаунт.

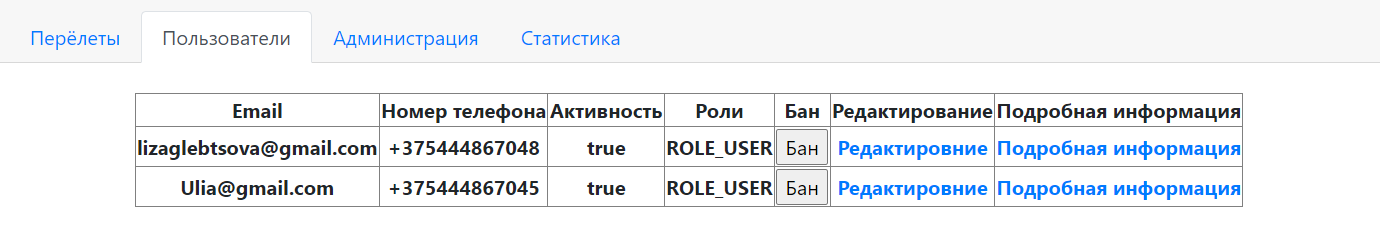


Рисунок 4.14 – Панель администратора. Пользователи

В пункте администрация видна информация об пользователях с правами администратора.

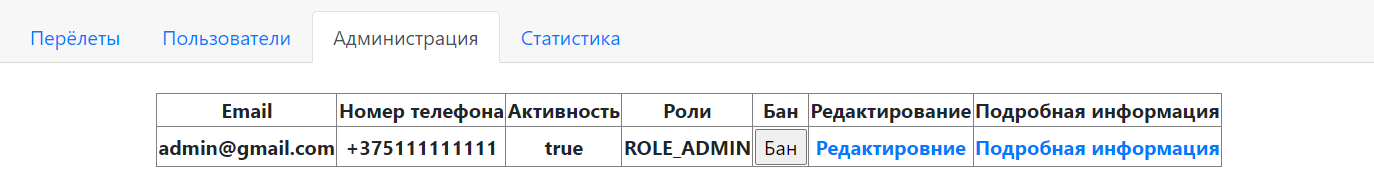


Рисунок 4.15 – Панель администратора. Администрация

В пункте статистика показывается статистика всех рейсов в виде таблицы. Статистика рейсов включает в себя: количество всех рейсов, выполненных рейсов; количество всех билетов, забронированных билетов; количество активных броней, а также общую прибыль.

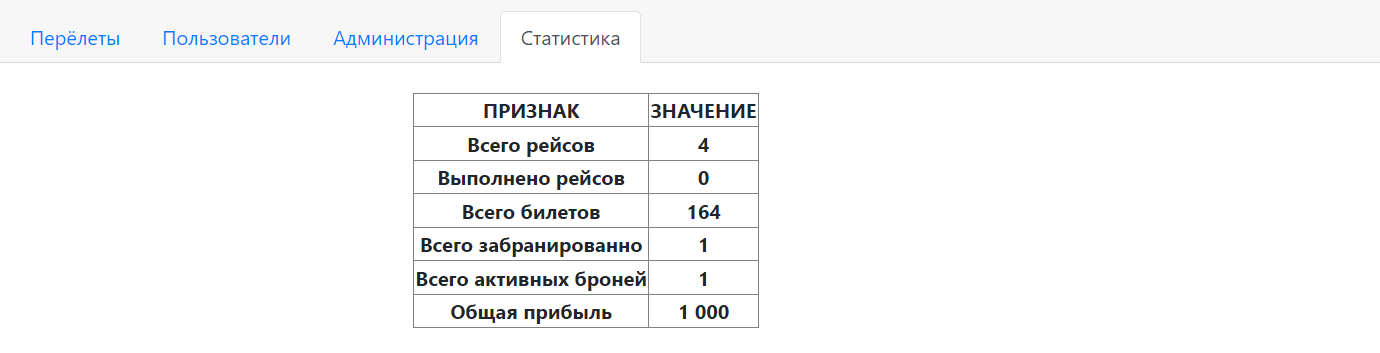


Рисунок 4.16 – Панель администратора. Статистика

**4.2 Дополнительные функции приложения**

Помимо основного функционала в приложении присутствует полная валидация данных при изменении/добавлении объектов. А именно проверка электронной почты (см. рисунок 4.16), проверка текстовых данных таких, как фамилия, имя, отчество пользователя, а также проверка номера телефона по стандарту Республики Беларусь.

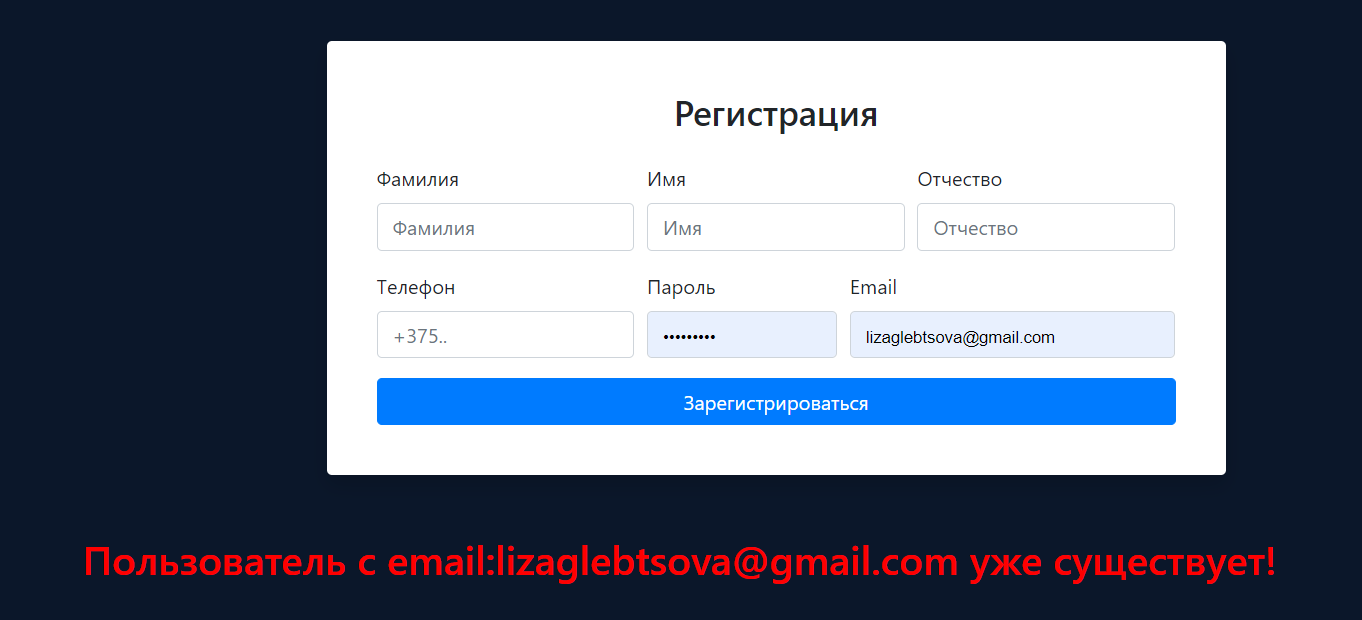


Рисунок 4.17 – Обработка ошибок ввода

**ЗАКЛЮЧЕНИЕ**

В данной курсовой работе было разработано приложение для бронирования рейсов на языке программирования Java с применением концепций объектно-ориентированного программирования. Были использованы принципы ООП (инкапсуляция, наследование, полиморфизм, абстракция), а также принципы SOLID и паттерны программирования.

У данного проекта есть дальнейшие перспективы модификации и развития. Например, интеграция с другими платформами, создания десктопного приложения для различных платформ таких, как Windows Android, IOS для бронирования билетов на рейсы, например, с использованием среды разработки Android Studio. Можно расширить функционал для упрощения работы с данными.

Итоговый продукт удовлетворяет не только техническим, но и функциональным требованиям. Благодаря соблюдению принципов ООП и SOLID, в приложение можно легко добавлять новые функциональные модули и расширять его возможности без необходимости переписывать или усложнять код. Новые модули могут быть реализованы путем разработки новых классов, наследующихся от базовых, что позволит использовать уже существующие методы и функции программного средства.

Таким образом, данная курсовая работа демонстрирует возможности и преимущества использования ООП-подходов при разработке программного средства для воспроизведения музыки.

**CПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Вайсфельд М. Объектно-ориентированное мышление, 2014. – 462 с.

[2] Java Documentation [Электронный ресурс]. – Режим доступа: <https://docs.oracle.com/en/java/>

[3] Spring Documentation [Электронный ресурс]. – Режим доступа: <https://spring.io/>

[4] HashDork [Электронный ресурс]. – Режим доступа: <https://hashdork.com/ru/onion-architecture/>.

[5] Bootstrap [Электронный ресурс]. – Режим доступа: <https://getbootstrap.com/>

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Исходный код**

**Файл User.java**

@Entity

@Table(name = "users")

@Data

public class User implements UserDetails {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "email", unique = true)

private String email;

@Column(name = "phonenumber")

private String phoneNumber;

@Column(name = "lastname")

private String lastName;

@Column(name = "secondname")

private String secondName;

@Column(name = "name")

private String name;

@Column(name = "active")

private boolean active;

@Column(name = "password", length = 1000)

private String password;

@ElementCollection(targetClass = Role.class, fetch = FetchType.EAGER)

@CollectionTable(name = "user\_role", joinColumns = @JoinColumn(name = "user\_id"))

@Enumerated(EnumType.STRING)

private Set<Role> roles = new HashSet<>();

@OneToMany( fetch = FetchType.EAGER,orphanRemoval = true)

private List<Ticket> ticketList = new ArrayList<>();

private LocalDateTime dateOfCreated;

@PrePersist

private void init() {

dateOfCreated = LocalDateTime.now();

}

public boolean isAdmin (){ return roles.contains(Role.ROLE\_ADMIN);}

//security

@Override

public Collection<? extends GrantedAuthority> getAuthorities() {

return roles;

}

@Override

public String getUsername() {

return email;

}

@Override

public boolean isAccountNonExpired() {

return true;

}

@Override

public boolean isAccountNonLocked() {

return true;

}

@Override

public boolean isCredentialsNonExpired() {

return true;

}

@Override

public boolean isEnabled() {

return active;

}

@Override

public String toString() {

return "User{" +

"id=" + id +

", email='" + email + '\'' +

", phoneNumber='" + phoneNumber + '\'' +

", name='" + name + '\'' +

", active=" + active +

", password='" + password + '\'' +

", roles=" + roles +

", dateOfCreated=" + dateOfCreated +

'}';

}

}

**Файл UserService.java**

package com.example.airport.services;

import java.util.stream.Collectors;

import static com.fasterxml.jackson.databind.cfg.CoercionInputShape.Array;

@Service

@Slf4j

@RequiredArgsConstructor

public class UserService {

private final UserRepository userRepository;

private final FlightRepository flightRepository;

private final PasswordEncoder passwordEncoder;

private final TicketRepository ticketRepository;

public boolean createUser(User user) {

String userEmail = user.getEmail();

if (userRepository.findByEmail(userEmail) != null) return false;

user.setActive(true);

user.getRoles().add(Role.ROLE\_USER);

user.setPassword(passwordEncoder.encode(user.getPassword()));

log.info("Saving new User with email: {}", userEmail);

userRepository.save(user);

return true;

}

public List<User> list() {

return userRepository.findAll();

}

public void addFlight(Principal principal, Long id) {

User u = userRepository.findByEmail(principal.getName());

List<Ticket> ticketList = ticketRepository.findAllByUserAndFlight(null, flightRepository.findById(id).orElse(null));

Ticket t = ticketRepository.findById(id).orElse(null);

t.setUser(u);

ticketRepository.save(t);

u.getTicketList().add(t);

userRepository.save(u);

Flight f = t.getFlight();

f.setAmount(f.getAmount() - 1);

flightRepository.save(f);

}

public List<Ticket> getTicketsBeforeNow(User user) {

List<Ticket> list = user.getTicketList();

List<Ticket> result = new ArrayList<>();

for (Ticket t : list) {

if (isBeforeNow(t.getFlight().getDeparture())) result.add(t);

}

return result;

}

public List<Ticket> getTicketsAfterNow(User user) {

List<Ticket> list = user.getTicketList();

List<Ticket> result = new ArrayList<>();

for (Ticket t : list) {

if (isAfterNow(t.getFlight().getDeparture())) result.add(t);

}

return result;

}

private boolean isAfterNow(LocalDateTime localDateTime) {

return localDateTime.isAfter(LocalDateTime.now());

}

private boolean isBeforeNow(LocalDateTime localDateTime) {

return localDateTime.isBefore(LocalDateTime.now());

}

public void banUser(Long id) {

User user = userRepository.findById(id).orElse(null);

if (user != null) {

if (user.isActive()) {

user.setActive(false);

log.info("Ban user with id={}; email={}", user.getId(), user.getEmail());

} else {

user.setActive(true);

log.info("UNBan user with id={}; email={}", user.getId(), user.getEmail());

}

userRepository.save(user);

}

}

public void changeUserRoles(User user, Map<String, String> form) {

Set<String> roles = Arrays.stream(Role.values())

.map(Role::name)

.collect(Collectors.toSet());

user.getRoles().clear();

for (String key : form.keySet()) {

if (roles.contains(key)) {

user.getRoles().add(Role.valueOf(key));

}

}

userRepository.save(user);

}

public void updateUser(Long id, String newName, String newLastName, String newSecondName, String newPhoneNumber) {

User user = userRepository.findById(id).orElse(null);

if (user != null) {

user.setName(newName);

user.setLastName(newLastName);

user.setSecondName(newSecondName);

user.setPhoneNumber(newPhoneNumber);

userRepository.save(user);

}

}

}

**Файл Ticket.java**

package com.example.airport.models;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.\*;

@Entity

@Table(name="tickets")

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Ticket {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "place")

private int place;

@ManyToOne(fetch = FetchType.LAZY)

private Flight flight;

@ManyToOne(fetch = FetchType.LAZY)

private User user;

}**Файл Flight.java**

package com.example.airport.models;

@Entity

@Table(name="airport")

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Flight {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "desc")

private String description;

@Column(name = "price")

private int price;

@Column(name = "from")

private String cityFrom;

@Column(name = "dest")

private String cityDest;

@Column(name = "airplane")

private String airplane;

@DateTimeFormat(iso = DateTimeFormat.ISO.DATE\_TIME)

@Column(name = "arrival")

private LocalDateTime arrival;

@DateTimeFormat(iso = DateTimeFormat.ISO.DATE\_TIME)

@Column(name = "departure")

private LocalDateTime departure;

@Column(name ="amount")

private int amount;

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,orphanRemoval = true)

private List<Ticket> ticketList = new ArrayList<>();

public String getFormattedArrival() {

return arrival.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));

}

public String getFormattedDeparture() {

return departure.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));

}

@Override

public String toString() {

return "Flight{" +

"id=" + id +

", description='" + description + '\'' +

", price=" + price +

", cityFrom='" + cityFrom + '\'' +

", cityDest='" + cityDest + '\'' +

", airplane='" + airplane + '\'' +

", date='" + arrival + '\'' +

'}';

}

public String getInfo() {

return

"$TICKET: " + cityFrom + "->" + cityDest + ", price=" + price +

", airplane='" + airplane +

", date='" + departure ;

}

}

**Файл Role.java**

package com.example.airport.models.enums;

import org.springframework.security.core.GrantedAuthority;

public enum Role implements GrantedAuthority {

ROLE\_USER,ROLE\_ADMIN;

@Override

public String getAuthority() {

return name();

}

}