**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Специальность

1-40 04 01 «Информатика и технологии программирования»

Кафедра «Информатика»

РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Объектно-ориентированное программирование»

на тему: «**Разработка архитектуры бизнес-слоя и слоя доступа к данным автоматизированного рабочего места услуг морских пассажирских**

**перевозок»**

Исполнитель: студент гр. ИП-31

В. О. Фещенко

Руководитель: ст. преподаватель

В. Н. Шибеко

Дата проверки:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата допуска к защите:­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка работы:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписи членов комиссии  
по защите курсового проекта:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2024

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc185300033)

[1 АНАЛИТИЧЕСКИЙ ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ И СРЕДСТВ РЕШЕНИЯ 4](#_Toc185300034)

[1.1 Обзор существующих моделей данных 4](#_Toc185300035)

[1.2 Обзор средств автоматизаций 5](#_Toc185300036)

[1.3 Анализ задания на проектирование 5](#_Toc185300037)

[1.4 Требования к проектируемому программному обеспечению 6](#_Toc185300038)

[1.5 Используемые средства 7](#_Toc185300039)

[2 АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 9](#_Toc185300040)

[2.1 Пользовательские роли и функции 9](#_Toc185300041)

[2.2 Реляционная модель базы данных 10](#_Toc185300042)

[2.3 Документная модель данных 16](#_Toc185300043)

[3 СТРУКТУРА И ОСНОВНЫЕ АЛГОРИТМЫ СОЗДАВАЕМОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 18](#_Toc185300044)

[3.1 Общая схема приложения и ее описание 18](#_Toc185300045)

[3.2 Интерфейсы и классы доступа к данным 19](#_Toc185300046)

[3.3 Интерфейсы и классы предметной области 20](#_Toc185300047)

[3.4 Пользовательский интерфейс 23](#_Toc185300048)

[4 ТЕСТИРОВАНИЕ 31](#_Toc185300049)

[4.1 Тестирование интерфейсов и классов доступа к данным и предметной области 31](#_Toc185300050)

[4.2 Тестирование пользовательского интерфейса 34](#_Toc185300051)

[ЗАКЛЮЧЕНИЕ 36](#_Toc185300052)

[Список использованных источников 37](#_Toc185300053)

[Приложение А – Класс *TransportMongoDBProfile* 38](#_Toc185300054)

[Приложение Б – Класс *UserRepository* 39](#_Toc185300055)

[Приложение B – Класс *UserRepositoryMongoDB* 43](#_Toc185300056)

[Приложение Г – Класс *UserService* 44](#_Toc185300057)

[Приложение Д – Класс *UserDTO* 45](#_Toc185300058)

[Приложение Е – Класс *UserServiceTests* 46](#_Toc185300059)

[Приложение Ж – Инъекции зависимостей 50](#_Toc185300060)

# **ВВЕДЕНИЕ**

Морские перевозки пассажиров представляют собой сложный процесс, включающий в себя организацию расписания, управление потоком пассажиров, оптимизацию использования судов и маршрутов, а также обеспечение безопасности и комфорта пассажиров.

Целью данного курсового проекта является разработка модели, которая позволит анализировать и оптимизировать морские пассажирские перевозки. Моделирование позволит лучше понять сложные взаимосвязи и динамику, существующую в этой отрасли, и предоставит инструменты для принятия обоснованных решений.

Ожидается, что результаты данного курсового проекта принесут практическую пользу. Они могут быть использованы для оптимизации расписания, повышения эффективности и комфорта пассажиров, а также для принятия обоснованных решений в области морских пассажирских перевозок. Это особенно актуально в наше время, когда международный туризм и глобальная торговля играют важную роль в экономике и связывают различные страны и регионы.

# **1 АНАЛИТИЧЕСКИЙ ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ И СРЕДСТВ РЕШЕНИЯ**

## **Обзор существующих моделей данных**

В рамках проектирования системы для учета свободных кают и управления заказами, важно выбрать подходящую модель данных, которая обеспечит эффективное хранение, обработку и управление информацией. Рассмотрим подробнее несколько основных моделей данных, которые могут быть использованы в данном проекте.

Реляционная модель данных является одной из самых распространенных и широко используемых моделей. Она организует данные в виде таблиц (отношений), где каждая таблица состоит из строк и столбцов. Основные характеристики реляционной модели — это структурированность, данные хранятся в четко определенных таблицах, что упрощает их организацию и доступ. Возможность устанавливать связи между различными таблицами с помощью первичных и внешних ключей. Например, таблица "Заказы" может ссылаться на таблицу "Клиенты" через внешний ключ. Для работы с реляционными базами данных используется язык структурированных запросов (*SQL*), который позволяет выполнять операции выборки, вставки, обновления и удаления данных. Также, наиболее полезной особенностью реляционной базы данных является обеспечение механизма для поддержания целостности данных, такие как ограничения уникальности и проверки.

Объектно-ориентированная модель данных основана на концепциях объектно-ориентированного программирования. В этой модели данные представляются в виде объектов, которые содержат как атрибуты (данные), так и методы (функции). Данные и методы, работающие с ними, объединены в одном объекте, что упрощает управление сложными структурами данных. Позволяет создавать новые объекты на основе существующих, что способствует повторному использованию кода. Объекты могут быть обработаны как экземпляры их базового класса, что упрощает работу с различными типами данных.

Документо-ориентированная модель данных используется в *NoSQL* базах данных и позволяет хранить данные в формате документов, таких как *JSON* или *XML*. Структура документов может быть изменена без необходимости изменения схемы базы данных, что позволяет легко адаптироваться к изменениям в требованиях. Позволяет хранить вложенные структуры данных, что может быть полезно для хранения информации о заказах с множеством атрибутов. *NoSQL* базы данных, как правило, лучше масштабируются, что позволяет обрабатывать большие объемы данных.

Сетевые и иерархические модели данных являются более старыми подходами к организации данных. В сетевой модели данные организованы в виде графа, где узлы представляют собой записи, а ребра — связи между ними. В иерархической модели данные организованы в виде дерева, где каждая запись имеет родительскую и дочерние записи.

## **Обзор средств автоматизаций**

Существуют различные системы управления расписанием, которые помогают оптимизировать распределение судов по маршрутам, учитывая потребности пассажиров и эффективное использование ресурсов. Эти системы учитывают такие факторы, как пропускная способность портов, время путешествия, частота рейсов и другие параметры для создания оптимального расписания.

Современные морские пассажирские перевозки предлагают пассажирам различные информационные системы, которые облегчают процесс покупки билетов, бронирования, отслеживания расписания, получения уведомлений и других сервисов. Эти системы могут быть представлены в виде веб-платформ, мобильных приложений или автоматизированных терминалов в портах.

Важным аспектом оптимизации морских пассажирских перевозок является интеграция с другими видами транспорта, такими как железнодорожный, автомобильный и воздушный транспорт. Это позволяет создать сеть межмодальных перевозок и облегчить перемещение пассажиров от одного транспортного узла кдругому.

На данный момент существует множество туристических агенств, осуществляющие морские туры на круизных лайнерах, где предлагаются различного рода услуги. В Беларуси такие услуги осуществляет компания *MalibuTravel.* Данная компания предлагает широкий выбор морских круизов по всему миру. Данный приммер показывает, что морские перевозки до сих пор остаются востребованы на данный момент.

В целом, существует широкий спектр методов и средств решения, применяемых в области морских пассажирских перевозок. Они включают системы управления расписанием, информационные системы для пассажиров, технологии бронирования и оплаты, аналитические инструменты и бизнес-интеллект, технологии автоматизации и роботизации, а также интеграцию с другими видами транспорта. Эти методы и средства направлены на оптимизацию процессов, повышение удобства для пассажиров, улучшение эффективности и повышение качества предоставляемых услуг.

Важно отметить, что с развитием технологий и инноваций в области транспорта и информационных технологий, появляются новые методы и средства решения, которые могут значительно изменить и усовершенствовать морские пассажирские перевозки. Поэтому важно следить за последними тенденциями и инновациями в данной области.

* 1. **Анализ задания на проектирование**

Для разработки трехслойного приложения (*3-tier architecture*) для автоматизированного рабочего места услуг морских пассажирских перевозок, необходимо четко определить структуру каждого слоя: слоя доступа к данным (*Data Access Layer, DAL*), бизнес-слоя (*Business Logic Layer, BLL*) и пользовательского интерфейса (*User Interface, UI*).

Слой доступа к данным отвечает за взаимодействие с базой данных. Он инкапсулирует все операции, связанные с получением, сохранением и обновлением данных. Слой доступа к данным будет включать классы и методы для выполнения *SQL*-запросов напрямую.

Основные компоненты *DAL*:

– модели данных, определяющие структуру данных, которые будут храниться в базе данных;

– репозитории: каждый репозиторий будет отвечать за операции *CRUD* (*Create, Read, Update, Delete*) для конкретной сущности;

– файл для конфигурации зависимостей (*dependecnies injection*);

Бизнес-слой отвечает за реализацию бизнес-логики приложения. Он обрабатывает данные, полученные из слоя доступа к данным, и применяет к ним бизнес-правила. Основные компоненты *BLL*:

– сервисы: каждый сервис реализует определенную бизнес-логику и взаимодействует с репозиториями для получения и сохранения данных.

– управление процессами: слой бизнес-логики управляет бизнес-процессами, такими как создание и обработка заказов, управление запасами, логистика доставки и взаимодействие с пользователями;

– валидация данных: в *BLL* происходит проверка входных данных на соответствие бизнес-правилам, это помогает предотвратить ошибки и обеспечивает целостность данных;

– файл конфигурации для инъекции зависимостей.

Пользовательский интерфейс отвечает за взаимодействие с пользователем. Он отображает данные и предоставляет пользователю возможность взаимодействовать с приложением. Слой *UI* получает данные из слоя бизнес-логики и представляет их пользователю в удобном для восприятия формате. Также слой пользовательского интерфейса обеспечивает валидацию ввода данных, где происходит первичная проверка корректности введенных данных, что позволяет предотвратить ошибки.

* 1. **Требования к проектируемому программному обеспечению**

В курсовом проекте будет реализовано 3-х слойное приложение для автоматизированного рабочего места услуг морских пассажирских перевозок.

В программе будут реализованы справочники: пассажиры, корабли, маршруты, прейскурант, виды кают.

Функции: ведение списка пассажиров и кораблей. Создание и удаление маршрутов. Покупка билетов. Отмена билетов.

Отчеты: статистика по оказанным услугам. Данные о пассажирах. Данные о кораблях. Данные о маршрутах.

Задача состоит в том, чтобы реализовать описанные ранее возможности, которые будут предоставляться пользователю, а некоторые из них предоставляются сотруднику, у которого будет своя роль и свои возможности. А значит нужно уделить должное внимание дружескому пользовательскому интерфейсу, чтобы он был максимально интуитивно понятен большинству потенциальных клиентов.

Начнем с того, что в программе будут реализованы роли. И в зависимости от типа авторизованного пользователя (его роль) будет реализован различный функционал.

Возможности пользователя:

− регистрация в системе;

− поиск подходящего маршрута с выводом всей соответсвующей информацией (откуда, куда, расстояние, время отправления, время прибытия, класс каюты, итоговая стоимость билета);

− покупка билета;

− отмена билета;

− добавление пассажира, для дальнейшей возможности купить на него билет;

− редактирование пассажира;

− удаление пассажира.

Возможности администратора:

− добавление маршрутов;

− удаление маршрутов;

− редактирование маршрутов;

− добавить транспорт;

− удалить транспорт;

− редактировать транспорт;

− доступ к отчетам и статистике;

− просмотр данных клиентов.

Возможности экономиста:

− добавить новый прайс-лист;

− добавление новых цен на кабины и цену за милю;

− удаление прайс-листа;

− редактирование прайс-листа;

− доступ к отчетам и статистике.

* 1. **Используемые средства**

В данном проекте для разработки автоматизированного рабочего места услуг морских пассажирских перевозок будут использоваться следующие средства и технологии:

*Microsoft SQL Server —* это реляционная система управления базами данных (СУБД), разработанная компанией Microsoft. Она предоставляет мощные инструменты для хранения, обработки и управления данными. SQL Server будет использоваться для хранения структурированных данных, таких как информация о клиентах, заказах, каютах и прейскуранте. Система будет обеспечивать надежное хранение и быстрый доступ к данным.

*MongoDB —* это документо-ориентированная NoSQL база данных, которая позволяет хранить данные в формате JSON. Она обеспечивает гибкость в структуре данных и масштабируемость. MongoDB будет использоваться для хранения тех же данных, что и в *MS SQL Server*, только в другом виде.

В качестве среды разработки была выбрана *Microsoft Visual Studio 2022.* Проект был написан на языке С# на платформе *.NET 9*

1. **АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

## **Пользовательские роли и функции**

В программе реализованы 3 роли: администратор, пользователь и экономист. У каждой роли есть свои функции.

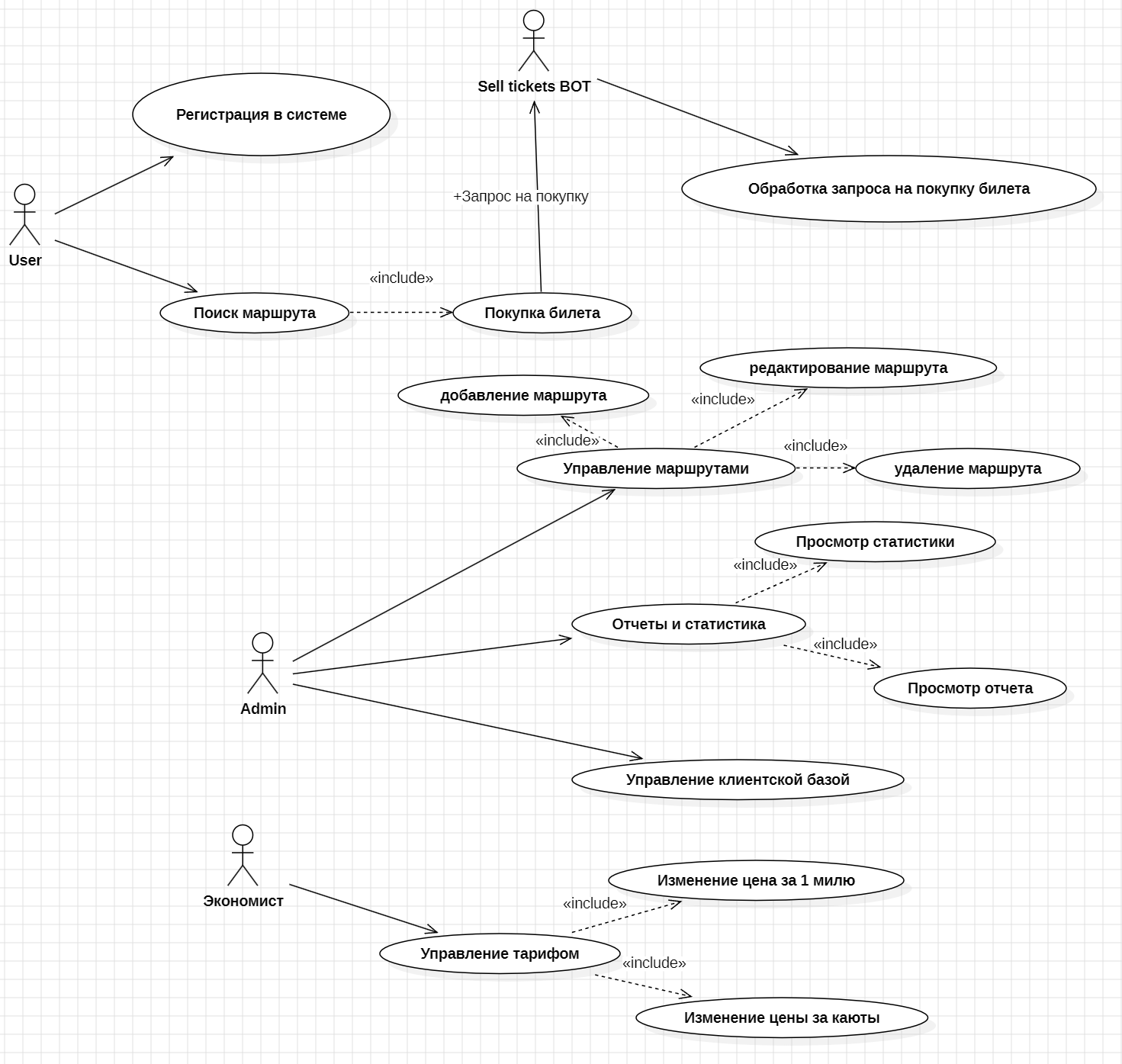


Рисунок 2.1 – роли и их возможности

На рисунке 2.1 представлены роли и их возможности в системе. Администратор может добавлять, удалять, редактировать маршруты. Экономист может изменять прайс-лист. Тот и другой могут создавать и просматривать отчеты.

Пользователь в свою очередь, имеет возможность регистрации в системе. После успешной регистрации пользователь может купить билет на интересующий его маршрут с любым уровнем комфорта и соответсвующей ценой.

Стоимость билета расчитывается исходя из вида транспорта, типа каюты и расстояния, которое будет необходимо пройти.

Сценарий прецедента “Покупка билета”:

* пользователь выбирает маршрут, на который хочет приобрести билет;
* выбирает, какому пассажиру он желает приобрести билет;
* оплата.

Сценарий прецедента “Регистрация пассажира”:

* пользователь выбирает пункт меню для добавления пассажира, вводит необходимые данные;
* подтверждение введенных данных по нажатию кнопки;
* занесение нового пассажира в таблицу базы данных.

Сценарий прецедента “Регистрация в системе”:

* пользователь вводит адрес электронной почты;
* пользователь вводит пароль;
* подтверждение введенных данных по нажатию кнопки;
* занесение данных пользователя в таблицу пользователей.

Далее будет преведен пример добавления нового объекта в таблицу, а именно добавление нового маршрута, транспорта, прайс-листа. Все эти операции производятся схожим образом, потому будет приведен один случай на примере маршрута.

Сценарий прецедента “Добавление нового маршрута”:

* сотрудник авторизовывается в системе;
* выбирает нужный пункт меню;

Сценарий прецедента “Удаление маршрута”:

* сотрудник авторизовывается в системе;
* выбирает нужный пункт меню;
* запись удалена.

Сценарий прецедента “Редактирование маршрута”:

* сотрудник авторизовывается в системе;
* выбирает нужный пункт меню;
* выбирает какие поля он хочет изменить;
* изменят;
* нажимает подтвердить.

## **Реляционная модель базы данных**

В качестве базы данных выбрана *Microsoft SQL Server*. В этой базе данных будет находится вся информация о клиентах, пассажирах, маршрутах, прайс-листе, видах кораблей. На рисунке 2.2 представлена схема базы данных, полученная в *Microsoft SQL Server Management Studio*. Далее будет представлена эта же схема, только построенная с помозью *StarUML*, где будут показаны связи между сущностями.

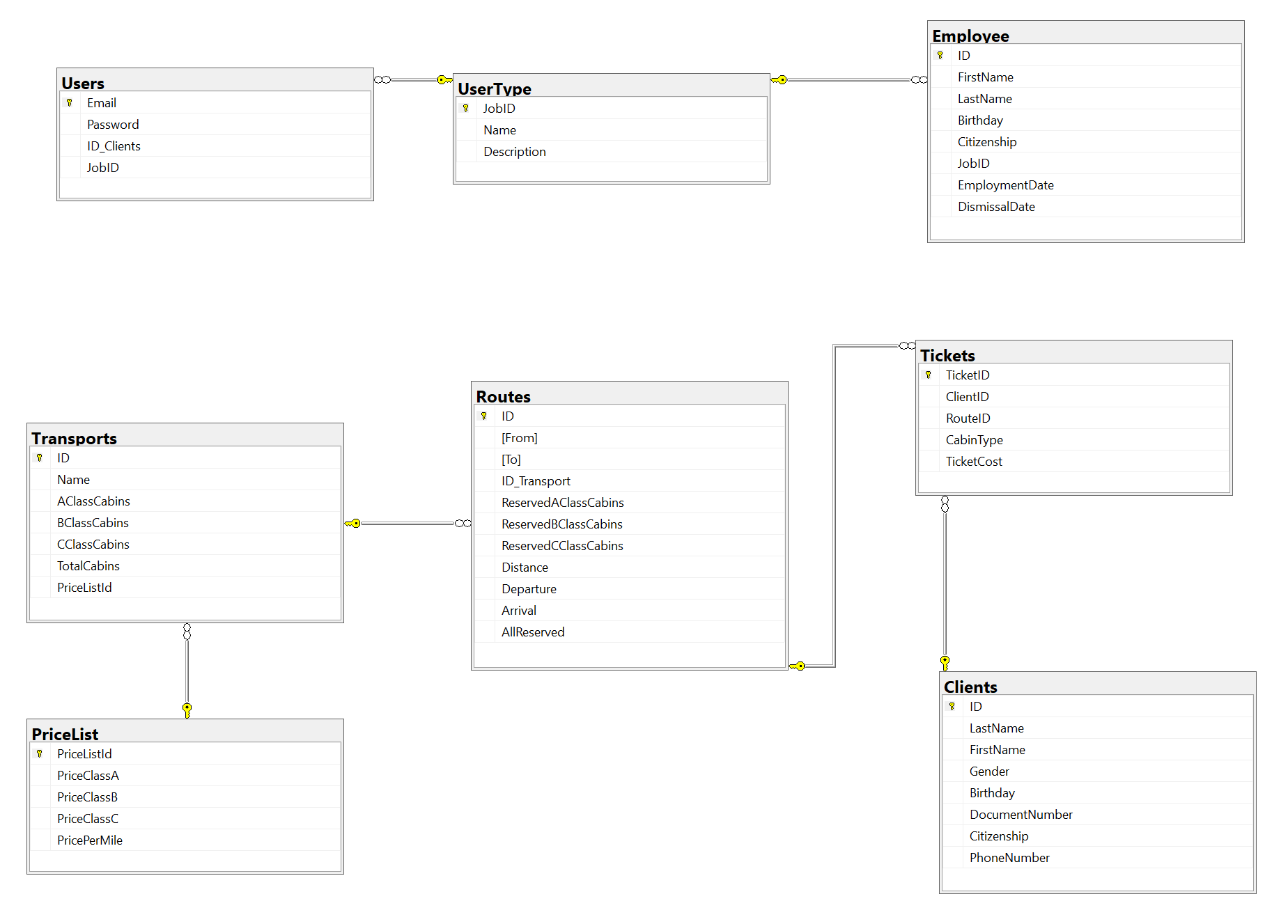


Рисунок 2.2 – схема базы данных *Microsoft SQL Server Management Studio*

На рисунке 2.3 представлена та же схема базы данных, но со связями сущностей.

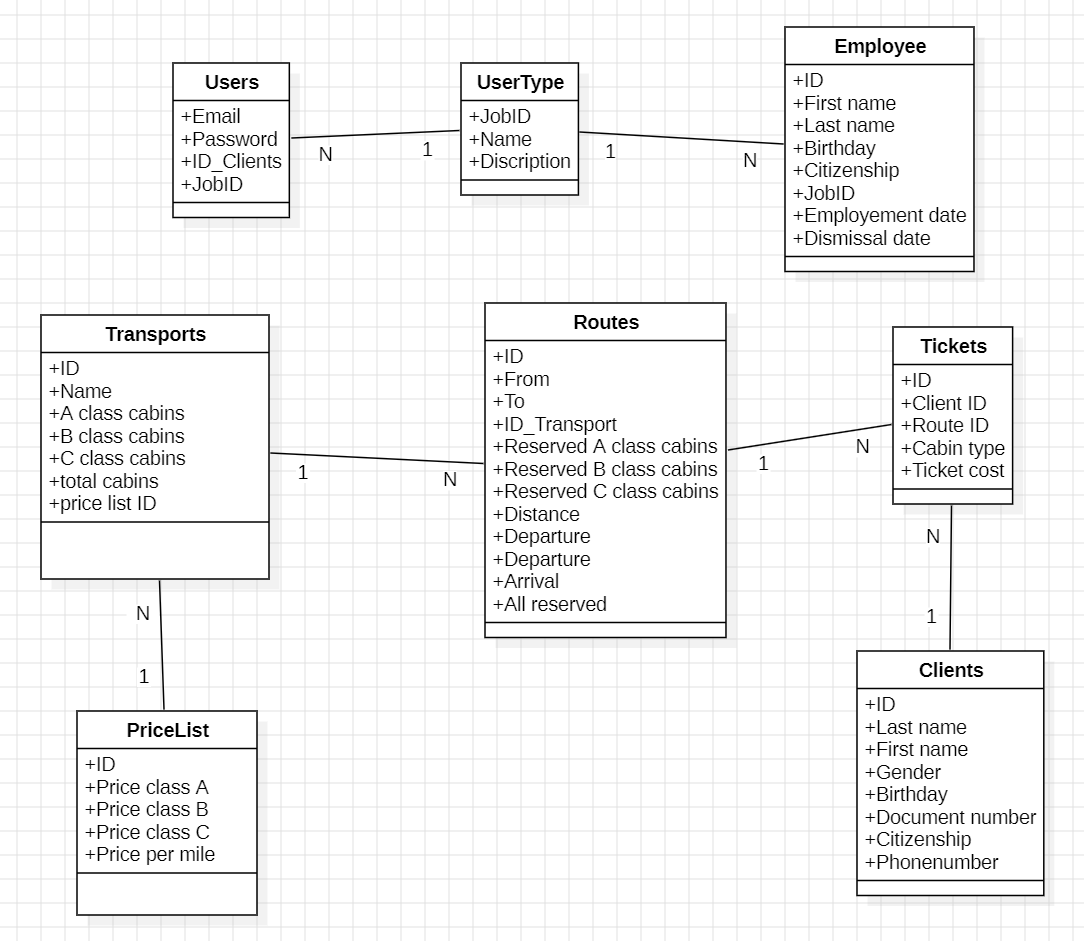


Рисунок 2.3 – схема базы данных *StarUML*

Таблица 2.1 — Таблица ограничений целостности

|  |  |
| --- | --- |
| **Связь** | **Ограничение целостности** |
| Пассажир – Билет | Запрет каскадного удаления |
| Билет – Маршрут | Запрет каскадного удаления |
| Маршрут – Транспорт | Запрет каскадного удаления |
| Транспорт – Прайс-лист | Запрет каскадного удаления |
| Тип пользователя – Сотрудник | Запрет каскадного удаления |
| Пользователь – Тип пользователя | Запрет каскадного удаления |

Описание атрибутов таблицы *Users* описаны в таблице 2.2.

Таблица 2.2 — Описание атрибутов таблицы *Users*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |
| **1** | **2** | **3** |
| Email (уникальный текст) | электронная почта пользователя, является уникальным идентификатором пользователя | Текст |
| Password | пароль пользователя | Текст |
| ID\_Clients | представляет собой строку с уникальными идентификаторами пассажиров, которым пользователь может купить билет | Текст в формате csv |
| UserType | Тип пользователя | int |

Описание атрибутов таблицы *Clients* описаны в таблице 2.3.

Таблица 2.3 — Описание атрибутов таблицы *Clients*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |

|  |  |  |
| --- | --- | --- |
| ID | уникальный идентификатор пассажира | Число |
| LastName | фамилия пассажира. Хранит текстовое значение, представляющее фамилию клиента. | Текст |
| FirstName | имя пассажира. Хранит текстовое значение, представляющее имя клиента. | Текст |
| Gender | пол клиента | Текст |
| Birthday | дата рождения пассажира | Дата |
| DocumentNumber | хранит информацию о номере документа, например, паспорта или удостоверения личности | Текст |

Продолжение таблицы 2.3

|  |  |  |
| --- | --- | --- |
| Citizenship | хранит информацию о гражданстве пассажира | Текст |
| PhoneNumber | хранит информацию о номере телефона пассажира, который может использоваться для связи | Текст |

Описание атрибутов таблицы *Tickets* описаны в таблице 2.4.

Таблица 2.4 — Описание атрибутов таблицы *Tickets*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |
| TicketID | уникальный идентификатор билета | Число |
| ClientID | идентификатор клиента. Связывает билет с определенным клиентом в системе. | Число |
| RouteID | идентификатор маршрута. Связывает билет с определенным маршрутом. | Число |
| CabinType | хранит информацию о типе каюты | Текст |
| TicketCost | стоимость билета. Хранит информацию о стоимости билета на данный маршрут и тип каюты | Число |

Описание атрибутов таблицы *Routes* описаны в таблице 2.5.

Таблица 2.5 — Описание атрибутов таблицы *Routes*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |
| **1** | **2** | **3** |
| ID | уникальный идентификатор маршрута | Число |
| From | хранит информацию о месте, откуда начинается маршрут | Текст |

Продолжение таблицы 2.5

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| To | хранит информацию о месте, куда направляется маршрут | Текст |
| ID\_Transport | идентификатор транспорта. Связывает маршрут с определенным транспортным средством, которое будет использоваться на этом маршруте. | Число |
| ReservedAClassCabins | количество зарезервированных кают класса A. | Число |
| ReservedBClassCabins | количество зарезервированных кают класса B | Число |
| ReservedCClassCabins | количество зарезервированных кают класса C | Число |
| Distance | расстояние между местом отправления и местом прибытия. | Число |
| Departure | дата и время отправления | Дата |
| Arrival | дата и время прибытия | Дата |
| AllReserved | общее количество зарезервированных мест | Число |

Описание атрибутов таблицы *Transports* описаны в таблице 2.6.

Таблица 2.6 — Описание атрибутов таблицы *Transports*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |
| **1** | **2** | **3** |
| ID | уникальный идентификатор транспорта | Число |
| Name | название транспорта | Текст |

Продолжение таблицы 2.6

|  |  |  |
| --- | --- | --- |
| AClassCabins | количество кают класса A. | Число |
| BClassCabins | количество кают класса B. | Число |
| CClassCabins | количество кают класса C | Число |
| TotalCabins | общее количество кают на данном транспорте | Число |
| PriceListId | идентификатор прайс-листа | Число |

Описание атрибутов таблицы *PriceList* описаны в таблице 2.7.

Таблица 2.7 — Описание атрибутов таблицы *PriceList*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |
| PriceListId | уникальный идентификатор прайс-листа | Число |
| PriceClassA | цена для каюты класса A | Число |
| PriceClassB | цена для каюты класса B | Число |
| PriceClassC | Цена для каюты класса C | Число |
| PricePerMile | цена за милю | Число |

Описание атрибутов таблицы *UserType* описаны в таблице 2.8.

Таблица 2.8 — Описание атрибутов таблицы *UserType*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |
| JobID | уникальный идентификатор должности | Число |
| Name | название должности | Текст |
| Description | описание должности | Текст |

Описание атрибутов таблицы *Employee* описаны в таблице 2.9.

Таблица 2.9 — Описание атрибутов таблицы *Employee*

|  |  |  |
| --- | --- | --- |
| **Атрибуты** | **Описание домена** | **Тип данных** |
| **1** | **2** | **3** |
| ID | уникальный идентификатор сотрудника | Число |

Продолжение таблицы 2.9

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| FirstName | Имя сотрудника | Текст |
| LastName | Фамилия сотрудника | Текст |
| Birthday | Дата рождения сотрудника | дата |
| Citizenship | Гражданство сотрудника | текст |
| JobID | Уникальный идентификатор должности | Число |
| EmploymentDate | Дата принятия на работу | Дата |
| DismissalDate | Дата увольнения | дата |

Контроль целостности соблюдается в программе на уровне *BLL*, путем простой проверки, не ссылается какой-то объект на сущность, которую пытаются удалить. Конечно, реляционная база данных не позволила бы по умолчанию удалять объект, на который кто-то ссылается. Документная база данных в лице *MongoDB* таким похвастаться не может и поэтому было необходимо реализовать даннуб проверку на слое *BLL.*

## **Документная модель данных**

Документная база данных слегка отличается от реляционной, а именно в сущности *Transports*, где присутствуют вложенные типы *Cabins* и *Prices.* На рисунке 2.3 показан пример сущности *Transports* в *MongoDB.*

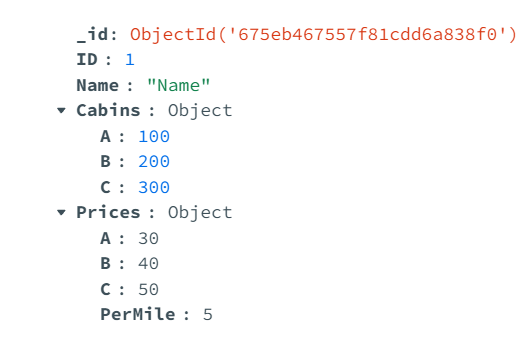


Рисунок 2.3 – пример сущности *Transports*

Также, сущность *Routes* содержит в себе встроенный тип в лице *ReservedCabins,* это показано на рисунке 2.4.



Рисунок 2.4 – пример сущности *Routes*

Также была исключена сущность *PriceList*, которая присутствовала в модели реляционной. Остальные сущности остались без изменений.

На рисунке 2.5 представлена схема документной базы данных.

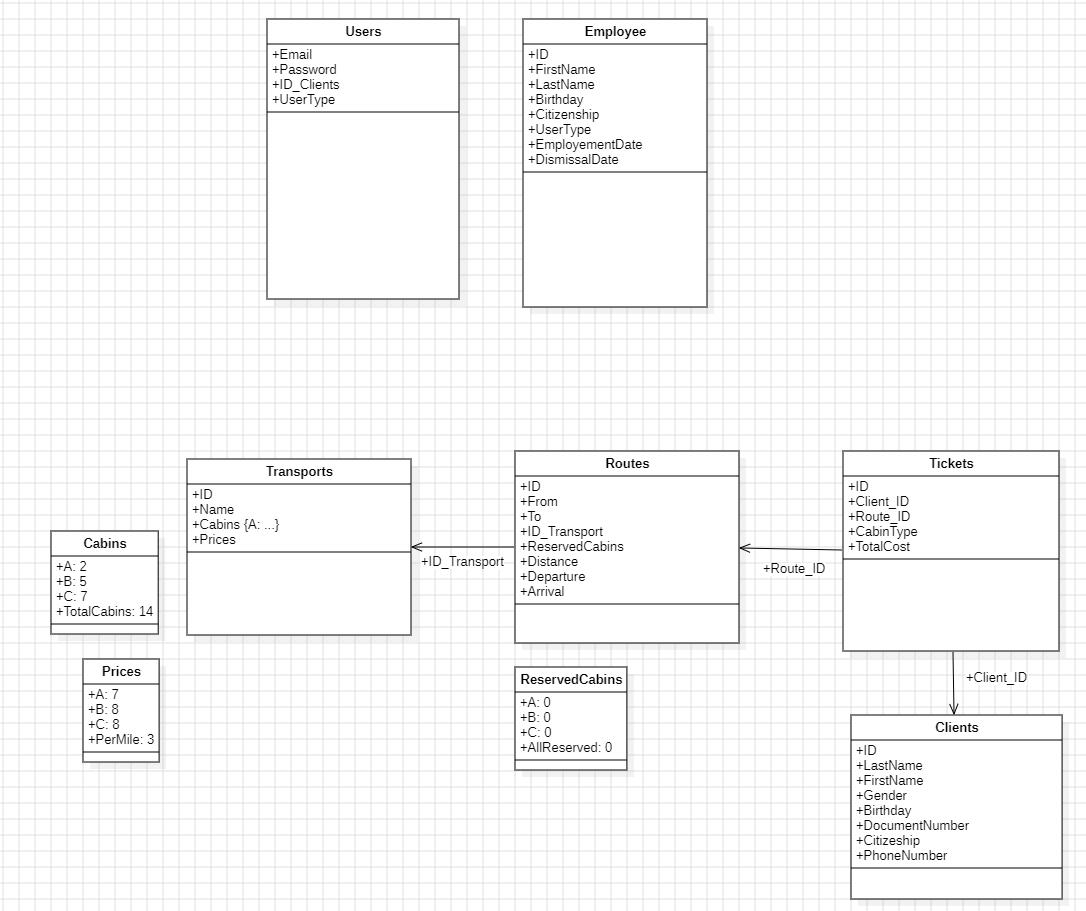


Рисунок 2.5 – Схема документной базы данных

Контроль целостности данных соблюдается на уровне *BLL.* Это значит, что при вызове метода для удаления записи сервис обнаружит, что на этот объект есть ссылка, то удаление будет отменено.

# **3 СТРУКТУРА И ОСНОВНЫЕ АЛГОРИТМЫ СОЗДАВАЕМОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

## **3.1 Общая схема приложения и ее описание**

Слой пользовательского интерфейса (*UI*): этот слой отвечает за взаимодействие с пользователем. Он включает в себя меню простого выбора, которое отображают информацию и позволяют пользователям вводить данные.

Бизнес-слой (*BLL*): бизнес-слой реализует бизнес-логику приложения. Он обрабатывает данные, полученные из слоя доступа к данным, и применяет к ним бизнес-правила. Включает в себя: сервисы для управления клиентами, заказами, оплатами и каютами и логику для обработки бизнес-процессов, таких как создание заказов, управление статусами и генерация отчетов. Также отвечает за валидацию данных и контроль целостности.

Слой доступа к данным (*DAL*): слой доступа к данным отвечает за взаимодействие с базой данных. Он инкапсулирует все операции, связанные с получением, сохранением и обновлением данных. Включает в себя репозитории для работы с сущностями, такими как клиенты, заказы, каюты и оплаты, а также класс подключения к базе данных, который управляет соединениями и выполняет SQL-запросы.

Также реализовано внедрение зависимостей, тоесть будут использоваться реализации сервисов, которые написаны для работы с той или иной базой данных.

Общая схема работы приложения показана на рисунке 3.1.

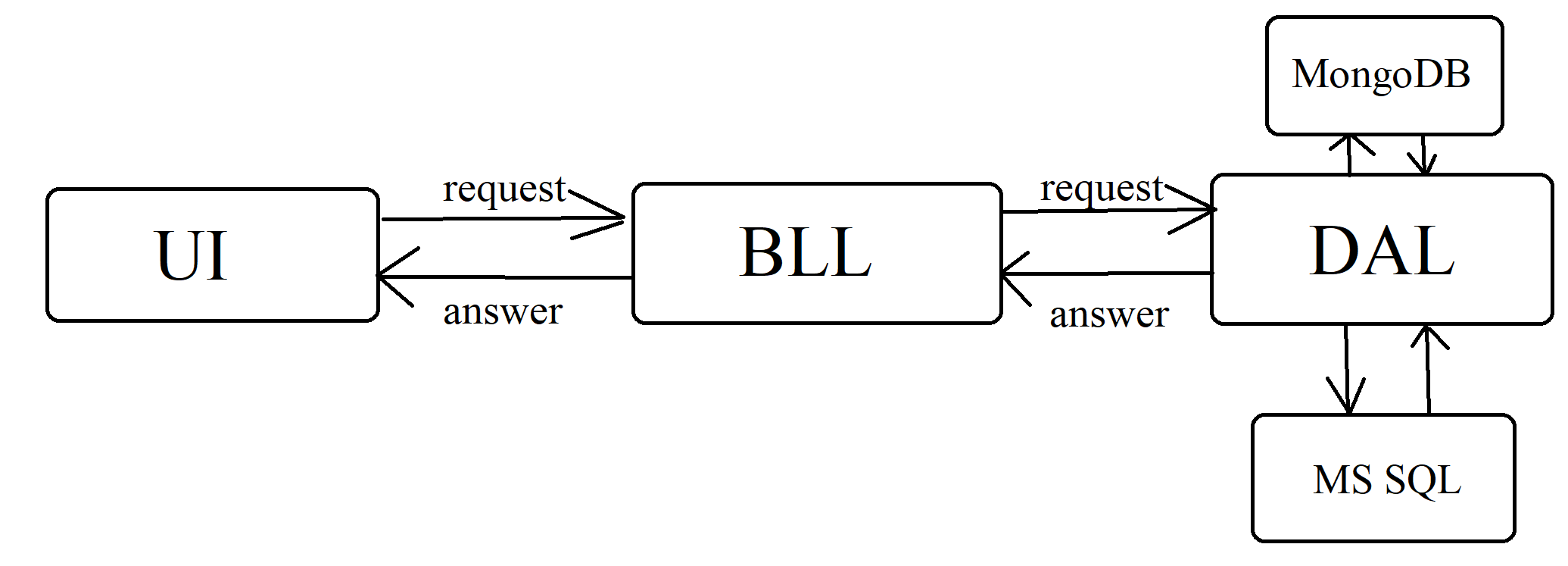


Рисунок 3.1 – Общая схема работы приложения

В процессе своей работы *BLL* получает на вход объект *DTO,* преобразует его к объекту базы данных, посылает на слой *DAL,* получает ответ, преобразует ответ обратно к *DTO* и возвращает на *UI.*

## **3.2 Интерфейсы и классы доступа к данным**

Для реализации паттерна «Репозиторий» необходимо было создать интерфейс *IRepository<T, TKey> where T : class* со следующими методами:  
 *– void Add(T entity) –* добавление новой сущности

*– void Update(T entity) –* обновление сущности

*– void Delete(T entity) –* удаление сущности

*– T Get(TKey id) –* получение сущности по ключу типа *TKey*

*– IEnumerable<T> GetAll()* – получение списка сущностей

Для последующих обновлений приложения и для лучшей масштабируемости были созданы интерфейсы для каждой модели, который реализует *IRepository<T, TKey>.* А каждый репозиторий в свою очередь реализует соответствующий ему интерфейс.

К примеру взять *IUserRepository,* который реализует *IRepository<User, string>,* в этом интерфейсе прописан дополнительный метод, который отсутствует в *IRepository: User CheckEmailAndPassword(string email, string password);*

Аналогично сделано со всеми репозиториями.

Также, из-за того что модели в реляционной базе данных и в документной немного отличаются, отличаются и их модели (классы) в программе, что в свою очередь делает необходимым создавать новый репозитории для работы с *MongoDB,* такое разделение дает четкое структуру проекту, что облегчит обслуживание данного программного обчеспечения в дальнейшем. Также из-за того что разные базы данных, у них разные модели, а значит нужно производить правильное преобразование моделей из *DTO* в объект базы данных и наоборот. Пример маппера показан в приложении А.

В таблице 3.2 показаны интерфейсы и репозиторий для работы с реляционной базой данных, который их реализует на слое *DAL.*

Таблица 3.2 — Интерфейсы и их реализации в *MS SQL*

|  |  |
| --- | --- |
| **Interface** | **Repository** |
| *IClientRepository* | *ClientRepository* |
| *IEmployeeRepository* | *EmployeeRepository* |
| *IPriceListRepository* | *PriceListRepository* |
| *IRouteRepository* | *RouteRepository* |
| *ITicketRepository* | *TicketRepository* |
| *ITransportRepository* | *TransportRepository* |
| *IUserRepository* | *UserRepository* |
| *IUserTypeRepository* | *UserTypeRepository* |

В таблице 3.3 показаны интерфейсы и репозиторий для работы с документной базой данных, который их реализует на слое *DAL.*

Таблица 3.3 — Интерфейсы и их реализации в *MongoDB*

|  |  |
| --- | --- |
| **Interface** | **Repository** |
| *IClientRepositoryMongoDB* | *ClientRepositoryMongoDB* |
| *IEmployeeRepositoryMongoDB* | *EmployeeRepositoryMongoDB* |
| *IRouteRepositoryMongoDB* | *RouteRepositoryMongoDB* |
| *ITicketRepositoryMongoDB* | *TicketRepositoryMongoDB* |
| *ITransportRepositoryMongoDB* | *TransportRepositoryMongoDB* |
| *IUserRepositoryMongoDB* | *UserRepositoryMongoDB* |
| *IUserTypeRepositoryMongoDB* | *UserTypeRepositoryMongoDB* |
| *IUserTypeRepository* | *UserTypeRepository* |

Примеры реализации интерфейсов репозиториями показаны в Приложении Б, B.

## **3.3 Интерфейсы и классы предметной области**

Общий для всех сервисов интерфейс является:

*public interface IService<T, TKey> where T : class*

*{*

*void Add(T item);*

*void Update(T item);*

*void DeleteById(TKey id);*

*T FindById(TKey id);*

*List<T> GetAll();*

*}*

Аналогично, как и в случае с репозиториями, каждый сервис реализует свой собственный интерфейс.

К примеру можно взять *IUserService:*

*public interface IUserService : IService<UserDTO, string>*

*{*

*public UserDTO CheckEmailAndPasword(string email, string password);*

*}*

Этот сервис реализует оба сервиса (который работает с *MS SQL* и c *MongoDB).* Примеры реализации показаны в приложении Г.

Далее будет приведено описание методов каждого сервиса, который работает с реляционной базой данных. Сервисы, которые работают с документной базой ничем не отличаются в плане функции от реляционных, кроме своего названия.

*ClientService -* сервис для управления клиентами*.*

– *Add(ClientDTO item):* Добавляет нового клиента в систему, преобразуя объект ClientDTO в объект Client и сохраняя его в репозитории.

– *DeleteById(int id):* Удаляет клиента по его идентификатору. Если клиент не найден, выбрасывает исключение ArgumentException.

– *FindById(int id):* Находит клиента по его идентификатору и возвращает его в виде объекта ClientDTO.

– *GetAll():* Возвращает список всех клиентов в виде списка объектов ClientDTO.

– *GetAll(string id\_clients):* Возвращает список клиентов, идентификаторы которых указаны в строке id\_clients. Идентификаторы разделяются запятыми.

– *GetAll(int[] id\_clients):* Возвращает список клиентов, идентификаторы которых указаны в массиве id\_clients.

– *Update(ClientDTO item):* Обновляет информацию о клиенте, преобразуя объект ClientDTO в объект Client и сохраняя изменения в репозитории.

*EmployeeService* – сервис для управления сотрудниками.

– *Add(EmployeeDTO item)*: Добавляет нового сотрудника в систему, преобразуя объект *EmployeeDTO* в объект *Employee*.

– *DeleteById(int id):* Удаляет сотрудника по его идентификатору. Если сотрудник не найден, выбрасывает исключение *ArgumentException*.

– *FindById(int id):* Находит сотрудника по его идентификатору и возвращает его в виде объекта *EmployeeDTO*.

– *GetAll():* Возвращает список всех сотрудников в виде списка объектов EmployeeDTO.

– *Update(EmployeeDTO item):* Обновляет информацию о сотруднике, преобразуя объект *EmployeeDTO* в объект *Employee*.

*PriceListService* – сервис для управления прайс-листами.

– *Add(PriceListDTO item):* Добавляет новый прайс-лист в систему, преобразуя объект *PriceListDTO* в объект *PriceList*.

– *DeleteById(int id*): Удаляет прайс-лист по его идентификатору. Если прайс-лист не найден, выбрасывает исключение *ArgumentException*.

– *FindById*(*int id*): Находит прайс-лист по его идентификатору и возвращает его в виде объекта *PriceListDTO*.

– *GetAll()*: Возвращает список всех прайс-листов в виде списка объектов PriceListDTO.

–*Update(PriceListDTO item*): Обновляет информацию о прайс-листе, преобразуя объект *PriceListDTO* в объект *PriceList*.

*RouteService* – сервис для управления маршрутами.

– *Add(RouteDTO item)*: Добавляет новый маршрут в систему, преобразуя объект *RouteDTO* в объект *Route*.

– *DeleteById(int id):* Удаляет маршрут по его идентификатору. Если маршрут не найден, выбрасывает исключение *ArgumentException*.

– *FindById(int id*): Находит маршрут по его идентификатору и возвращает его в виде объекта RouteDTO.

– *GetAll():* Возвращает список всех маршрутов в виде списка объектов RouteDTO.

– *Update(RouteDTO item):* Обновляет информацию о маршруте, преобразуя объект *RouteDTO* в объект *Route*.

*TicketService* – сервис для управления билетами.

– *Add(TicketDTO item):* Добавляет новый билет в систему, преобразуя объект TicketDTO в объект *Ticket*.

– *DeleteById(int id):* Удаляет билет по его идентификатору. Если билет не найден, выбрасывает исключение *ArgumentException*.

*– FindById(int id)*: Находит билет по его идентификатору и возвращает его в виде объекта *TicketDTO.*

– *GetAll():* Возвращает список всех билетов в виде списка объектов *TicketDTO*.

– *GetAvailableCabins(int routeId):* Возвращает доступные места (кабины) для указанного маршрута, а также цену за милю. Если маршрут или транспорт не найдены, выбрасывает исключение *ArgumentException*.

– *Update(TicketDTO item)*: Обновляет информацию о билете, преобразуя объект *TicketDTO* в объект *Ticket*.

*TransportService* – сервис для управления транспортом.

– *Add(TransportDTO item):* Добавляет новый транспорт в систему, преобразуя объект *TransportDTO* в объект *Transport*.

– *DeleteById(int id):* Удаляет транспорт по его идентификатору. Если транспорт не найден, выбрасывает исключение *ArgumentException*.

– *FindById(int id):* Находит транспорт по его идентификатору и возвращает его в виде объекта TransportDTO.

–*GetAll():* Возвращает список всех транспортных средств в виде списка объектов *TransportDTO*.

– *Update(TransportDTO item):* Обновляет информацию о транспорте, преобразуя объект *TransportDTO* в объект *Transport*.

*UserService* – сервис для управления пользователями.

– *Add(UserDTO item)*: Добавляет нового пользователя в систему, преобразуя объект *UserDTO* в объект *User*.

– *DeleteById(string id):* Удаляет пользователя по его идентификатору. Если пользователь не найден, выбрасывает исключение *ArgumentException*.

*– FindById(string id):* Находит пользователя по его идентификатору и возвращает его в виде объекта *UserDTO*.

– *GetAll():* Возвращает список всех пользователей в виде списка объектов UserDTO.

– *Update(UserDTO item):* Обновляет информацию о пользователе, преобразуя объект *UserDTO* в объект *User*.

*– CheckEmailAndPassword(string email, string password):* Проверяет наличие пользователя с указанным адресом электронной почты и паролем. Возвращает объект *UserDTO*, если пользователь найден, или null, если нет.

*UserTypeService* *–* сервис для управления типами пользователей.

*– Add(UserTypeDTO item):* Добавляет новый тип пользователя в систему, преобразуя объект *UserTypeDTO* в объект *UserType*.

*– DeleteById(int id):* Удаляет тип пользователя по его идентификатору. Если тип пользователя не найден, выбрасывает исключение *ArgumentException*.

– *FindById*(*int id)*: Находит тип пользователя по его идентификатору и возвращает его в виде объекта *UserTypeDTO.*

*– GetAll*(): Возвращает список всех типов пользователей в виде списка объектов *UserTypeDTO*.

– *Update(UserTypeDTO item):* Обновляет информацию о типе пользователя, преобразуя объект *UserTypeDTO* в объект *UserType*.

## **3.4 Пользовательский интерфейс**

При входе в программу перед пользователем появится меню, показанное на рисунке 3.2.

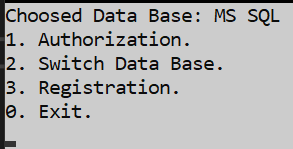


Рисунок 3.2 – Меню при запуске программы

После успешной авторизации в системе обычного пользователя увидит перед собой меню, показаное на рисунке 3.3.

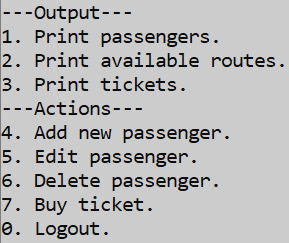


Рисунок 3.3 – Меню обычного пользователя

Пользователь может просмотреть список своих пассажирова, список доступныз маршуртов, список его билетов. Также он может добавить/отредактировать/удалить своего пассажира и, наконец, купить билет. Для отмены своего билета нужно будет выбрать пункт меню для отображения списка его билетов и далее выбрать хочет ли он отменить билет или нет. Процесс отмены билета показан на рисунке 3.4.

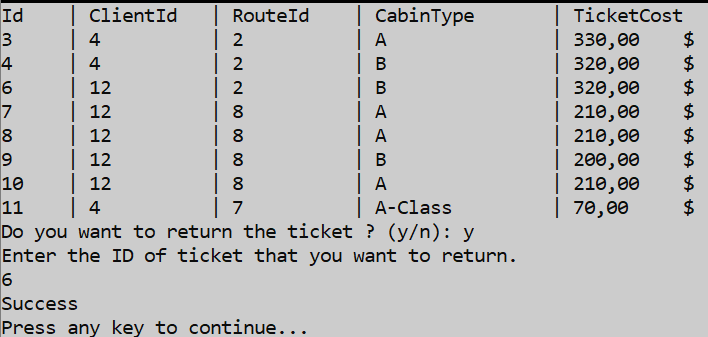


Рисунок 3.4 – Процесс отмены билета

Процесс покупки билета показан на рисунке 3.5.

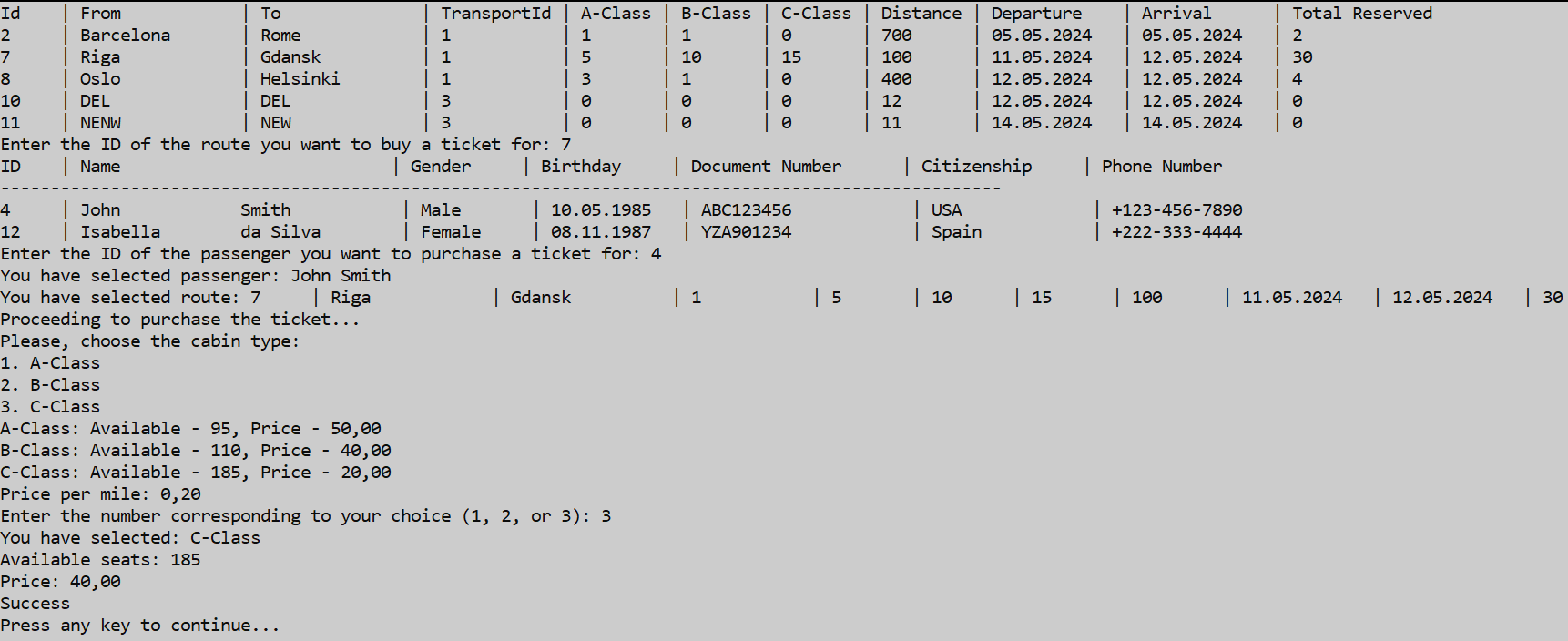


Рисунок 3.5 – Процесс покупки билета

На рисунке 3.6 показано, как пользователь может вывести данные по своим билетам.

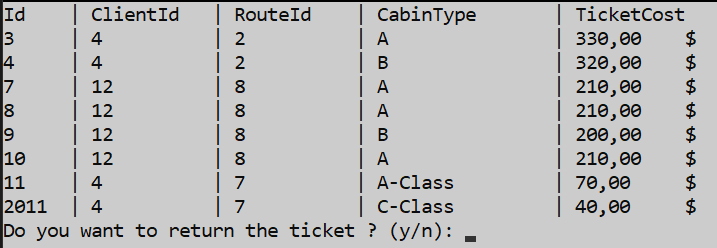


Рисунок 3.6 – Данные по билетам

Процесс добавления нового пассажира показан на рисунке 3.7.

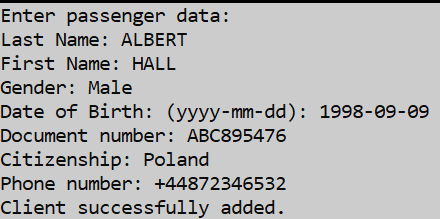


Рисунок 3.7 – процесс добавления пассажира

Процесс вывода списка пассажирова опказан на рисунке 3.8

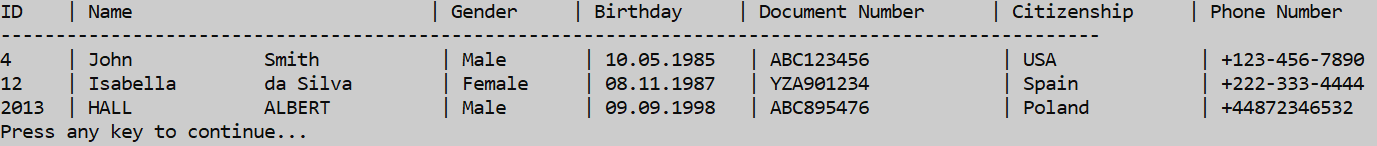


Рисунок 3.8 – процесс вывода списка пассажиров

Процесс редактирования данных пассажира показан на рисунке 3.9

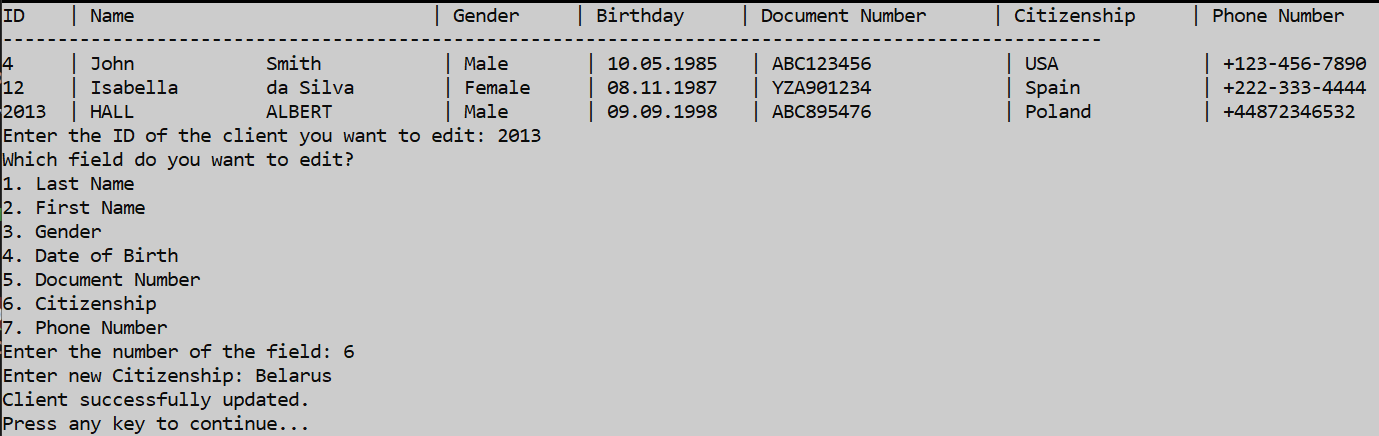


Рисунок 3.9 – процесс редактирования пассажира

Меню администратора в реляционной базе показано на рисунке 3.10.

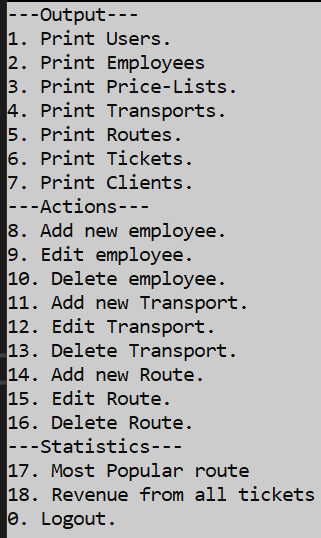


Рисунок 3.10 – меню администратора

Здесь админитсратор может вывести информацию о пользователях системы, работниках, прайс-листах, транспортах, маршрутах, билетов и клиентов.

Вывод информации аналогичен примеру показаному ранее (рисунок 3.8).

Пример добавления нового транспорта в реляционной базе показан на рисунке 3.11.

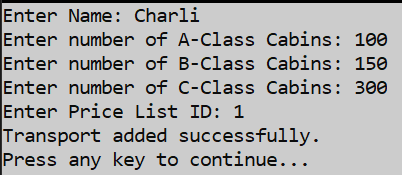


Рисунок 3.11 – добавление нового транспорта в реляционной базе

Процесс удаления транспорта показан на рисунке 3.12.

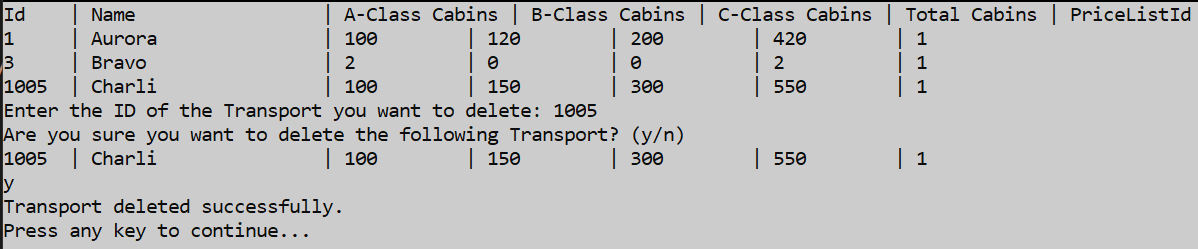


Рисунок 3.12 – удаление транспорта в реляционной базе

Администратор может посмотреть статистику по маршрутам, это показано на рисунке 3.13.

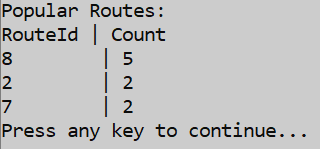


Рисунок 3.13 – статистика по популярности маршрутов

Общий доход от продажи билетов показан на рисунке 3.14.



Рисунок 3.14 – общий доход от продажи билетов

Остальной функционал добавления/редактирования/удаления данных идентичен тому, как показано ранее.

Далее для сравнения будет показано меню администратора при использовании документной базы, ведь там нет прайс-листов. На рисунке 3.15 показано меню администратора в документной базе.

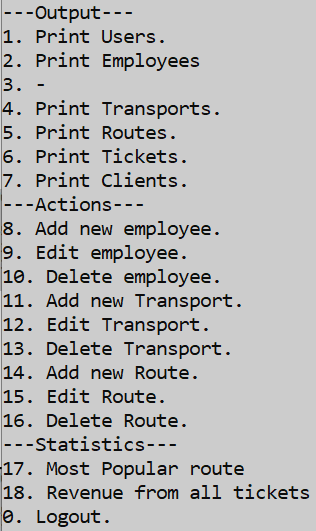


Рисунок 3.15 – меню администратора в документной базе

Как видно, здесь просто пропущен пункт меню для отображения прайс-листов, а функционал работы с ними принадлежит экономисту.

На рисунке 3.16 показан процесс добавления нового транспорта при использовании документной базы данных.

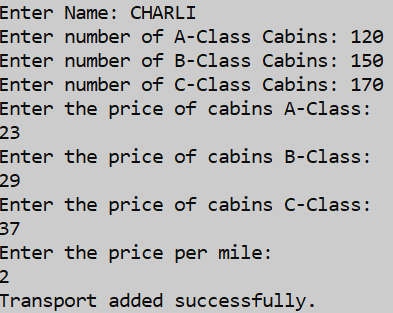


Рисунок 3.16 – процесс добавления транспорта в документной базе

Как видно, здесь нет ссылки на прайс-лист, так как тут используется вложенный тип. Остальной функционал идентичен функционалу администратора в реляционной базе.

На рисунке 3.17 показано меню экономиста в реляционной базе

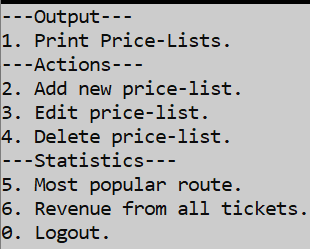


Рисунок 3.17 – меню экономиста в реляционной базе

На рисунке 3.18 показан процесс добавления нового прайс-листа.

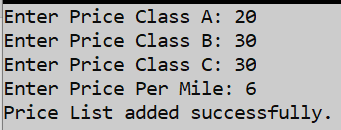


Рисунок 3.18 – процесс добавления нового прайс-листа

Остальной функционал работает аналогично, как было показано ранее на других примерах.

В документной базе данных отсутствует сущность с прайс-листом, поэтому меню экономиста также будет отличаться, а при изменении цен он будет менять цены напрямую к конкретному транспорту. На рисунке 3.19 показано меню экономиста в документной базе.

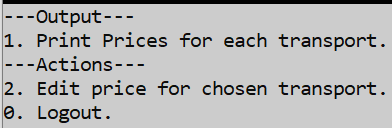


Рисунок 3.19 – меню экономиста в документной базе

Процесс изменения цены показан на рисунке 3.19

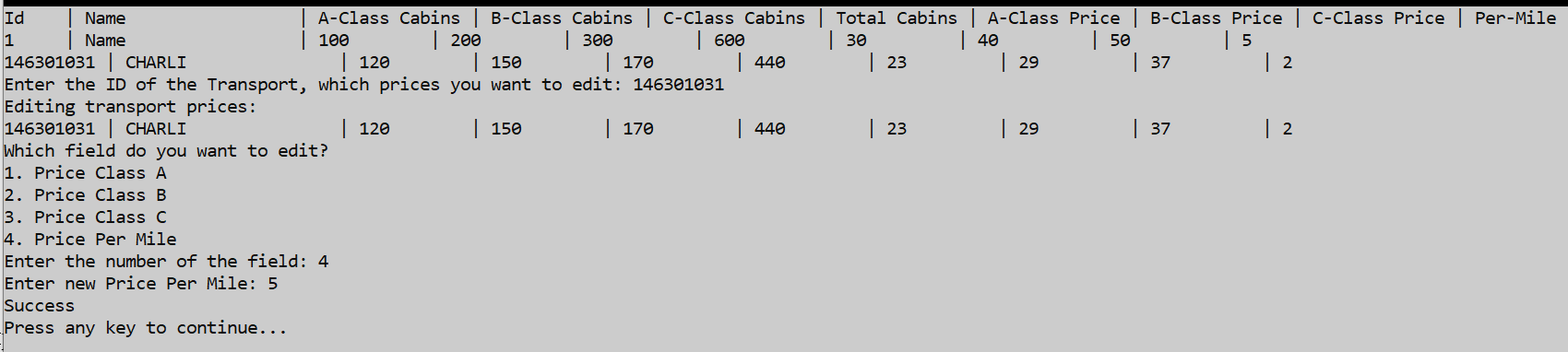


Рисунок 3.19 – Процесс изменения цены

**4 ТЕСТИРОВАНИЕ**

**4.1 Тестирование интерфейсов и классов доступа к данным и предметной области**

Были написаны тесты для проверки работоспособности методов и их корректность. Тесты написаны таким образом, что проверяется логика сервиса и взаимодействие с репозиторием, только вместореальной базы данных используется мок-объект. На рисунке 4.1 представлен результат выполнения тестов для *ClientService.*

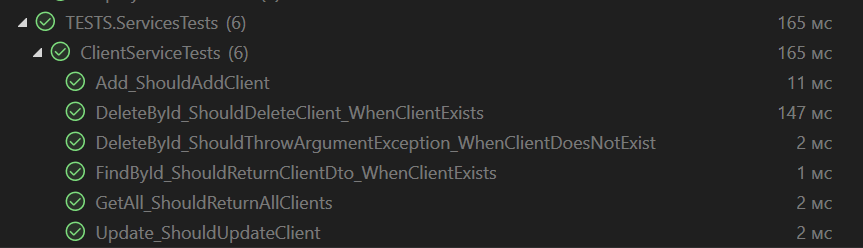


Рисунок 4.1 – Результат тестов для *ClientService*

Все тесты успешно пройдены, что говорит, что ожидаемый результат совпадает с фактическим. Здесь было проверено добавление, удаление, если клент существует, удаление, где вызывается ошибка если клиент не существует. Поиск по айди если клиент существует, получение списка объектов и обновление также пройдено.

На рисунке 4.2 представлен результат выполнения тестов для *EmployeeService.*

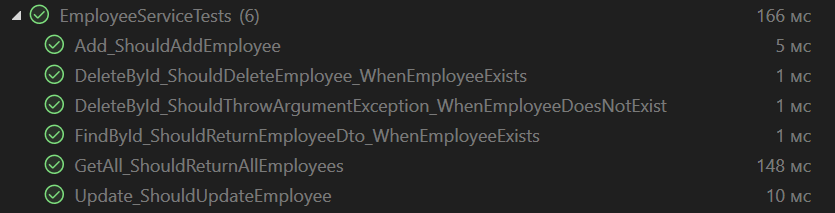


Рисунок 4.2 – Результат тестов для *EmployeeService*

Аналогично, все тесты на *CRUD* операции пройдены.

Тесты для сервиса *UserTypeService* показаны на рисунке 4.3.

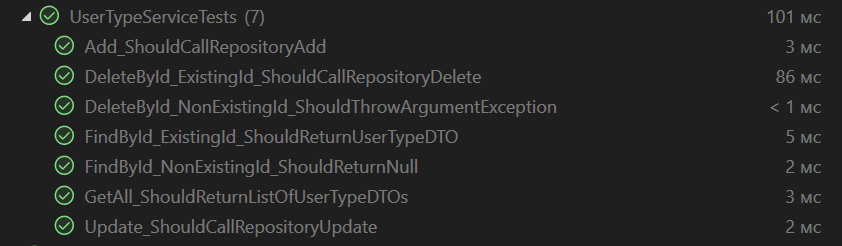


Рисунок 4.3 – Результат тестов для *UserTypeService*

Тесты для сервиса *UserService* показаны на рисунке 4.4.

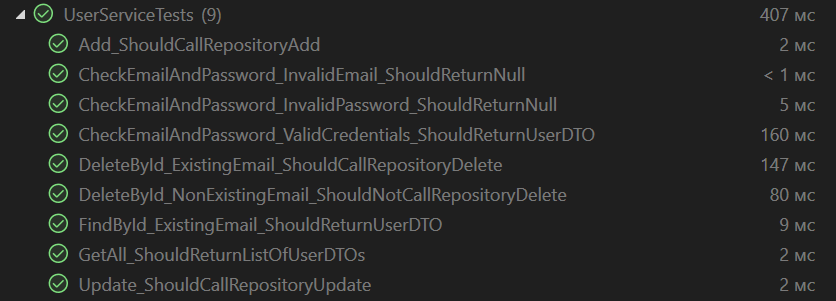


Рисунок 4.4 – Результат тестов для *UserService*

Тесты для сервиса *TransportService* показаны на рисунке 4.5.

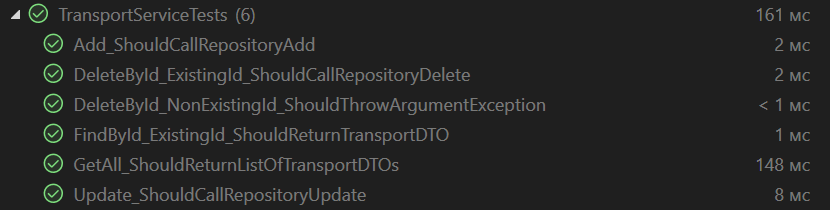


Рисунок 4.5 – Результат тестов для *TransportService*

Тесты для сервиса *TicketService* показаны на рисунке 4.6.

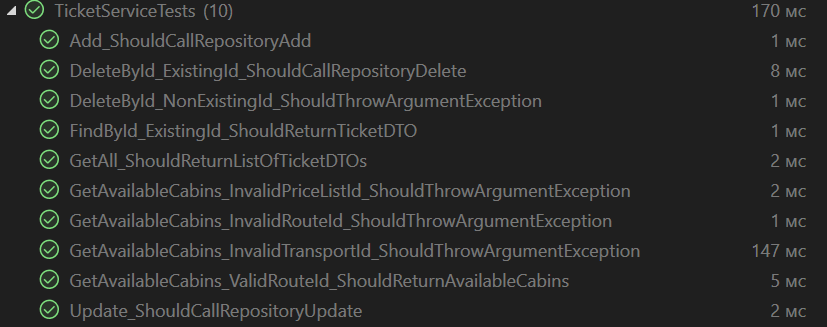


Рисунок 4.6 – Результат тестов для *TicketService*

Тесты для сервиса *RouteService* показаны на рисунке 4.7.

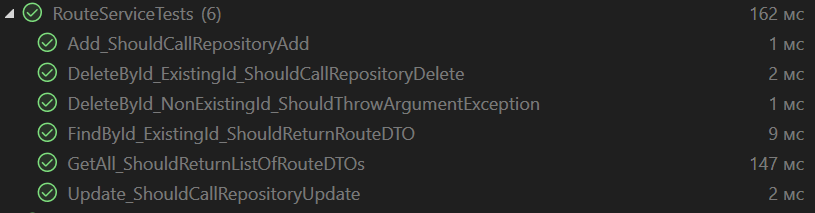


Рисунок 4.7 – Результат тестов для *RouteService*

Тесты для сервиса *PriceListService* показаны на рисунке 4.8.

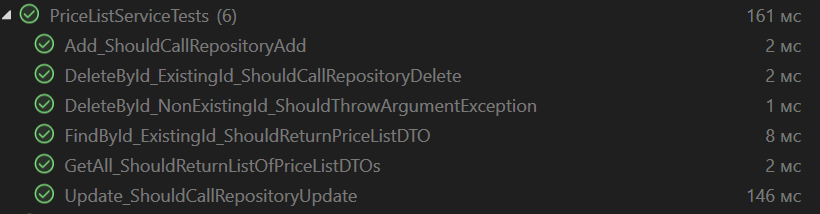


Рисунок 4.8 – Результат тестов для *PriceListService*

В каждом написаном тесте проверялись *CRUD* операции, а именно случаи удачные и случаи, где должны были поялвяться ошибки.

Отдельно стоит отметить случаи где происходит попытка удаления объекта, на который есть ссылка в другом объекте. Эти случаи покрывает соответсвующий сервис.

На рисунке 4.9 показана попытка удаления транспорта, который указан в маршруте.

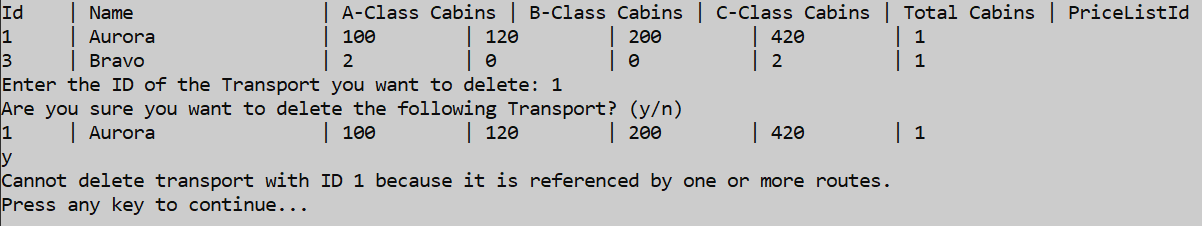


Рисунок 4.9 – попытка удаления транспорта, который указан в маршруте

Стоит отметить, что ошибка ловиться на слое *BLL* в соответсвующем сервисе.

* 1. **Тестирование пользовательского интерфейса**

В процессе авторизации могут возникнуть различные ошибки. На рисунке 4.10 показан случай, когда пользователя с введенным адресом электронной почты не существует.

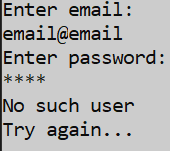


Рисунок 4.10 – Обработка ошибки при отсутствии пользователя в базе данных с таким адресом электронной почты

При неверном вводе пароля будет ваыводиться соответствующая ошибка. На рисунке 4.11 показано данное событие.

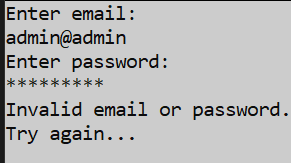


Рисунок 4.11 – Обработка ошибки при неверном вводе пароля

Процесс обработки ошибки при попытки ввести некорректный формат даты при показан на рисунке 4.12.

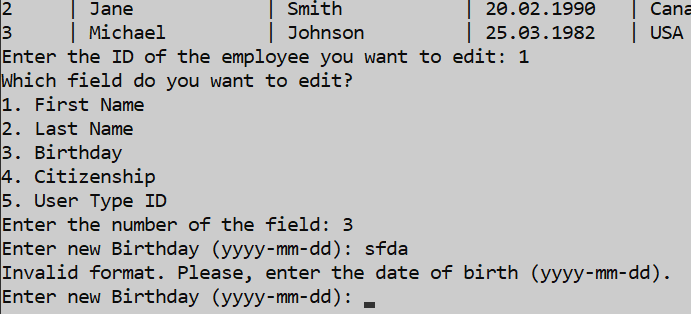


Рисунок 4.12 – Обработка ошибки при неверном вводе формата даты

Процесс обработки ошибки при неваерном вводе показан на рисунке 4.13.

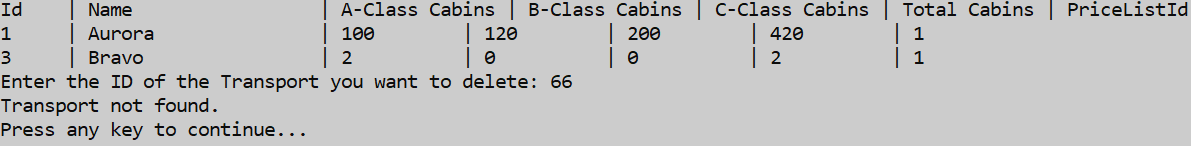


Рисунок 4.13 – Обработка ошибки при неверном вводе формата даты

**ЗАКЛЮЧЕНИЕ**

В рамках курсового проекта была разработана архитектура бизнес-слоя и слоя доступа к данным для автоматизированного рабочего места услуг морских пассажирских перевозок. Проект охватывает все ключевые аспекты, начиная с аналитического обзора существующих методов и средств решения, и заканчивая тестированием разработанного программного обеспечения.

В ходе работы был проведен обзор существующих моделей данных, что позволило выбрать наиболее подходящие подходы для реализации реляционной и документной моделей данных.

Обзор средств автоматизации выявил необходимые инструменты и технологии, такие как *Microsoft SQL Server* и *MongoDB* для хранения данных, а также *Visual Studio* и *.NET 9* для разработки. Эти средства обеспечили надежную платформу для создания приложения, соответствующего современным требованиям.

Анализ задания на проектирование и формулирование требований к проектируемому программному обеспечению позволили четко определить функциональные и нефункциональные аспекты системы. Это включало ведение справочников, учет свободных кают, прием заказов, учет оплаты и генерацию отчетов, что является основой для успешной работы приложения.

Архитектура программного обеспечения была спроектирована с учетом пользовательских ролей и функций, что обеспечило удобство и интуитивность интерфейса. Реализация реляционной и документной моделей данных позволила эффективно организовать хранение информации о городах, расписаниях, видах кают и прейскуранте.

В процессе разработки была создана общая схема приложения, описывающая взаимодействие между слоями (*DAL, BLL, UI*) и их компонентами. Интерфейсы и классы доступа к данным были реализованы для обеих моделей, что обеспечило четкую структуру и возможность дальнейшего расширения функционала.

Тестирование интерфейсов и классов доступа к данным, а также классов предметной области подтвердило корректность работы системы и ее соответствие заданным требованиям. Написанные тесты обеспечили надежность и стабильность приложения, что является важным аспектом для его дальнейшей эксплуатации.

Таким образом, в результате выполнения курсового проекта была успешно разработана и протестирована система, которая отвечает всем требованиям и обеспечивает эффективное управление услугами морских пассажирских перевозок. Полученные знания и опыт в области проектирования, разработки и тестирования программного обеспечения будут полезны для дальнейшей профессиональной деятельности.

# **Список использованных источников**

1. Прайс, Марк. C# 10 и .NET 6. Современная кросс-платформенная разработка: Освойте Python и сразу приступайте к решению прикладных задач, связанных с анализом данных. — Россия, Питер, 2022. — 256 с.
2. Вишневский А.В. Microsoft SQL Server. Эффективная работа. – СПб.: Питер, 2009. – 541 с.
3. Арораа, Гаурав, Чилберто, Джеффри. Паттерны проектирования для C# и платформы .NET Core. — Россия, Питер, 2023. — 352 с.
4. Онокой Л.С. Проектирование информационных систем: Учебное пособие / Л.С. Онокой, О.А. Морозова, Т.Е. Точилкина. – М.:Прометей, 2024 – 352 с.

# **Приложение А – Класс *TransportMongoDBProfile***

public class TransportMongoDBProfile : Profile

{

public TransportMongoDBProfile()

{

CreateMap<TransportMongoDB, TransportDTO>()

.ForMember(dest => dest.Id, opt => opt.MapFrom(src => src.ID))

.ForMember(dest => dest.ObjectId, opt => opt.MapFrom(src => src.Id))

.ForMember(dest => dest.AClassCabins, opt => opt.MapFrom(src => src.Cabins != null ? src.Cabins.A : 0))

.ForMember(dest => dest.BClassCabins, opt => opt.MapFrom(src => src.Cabins != null ? src.Cabins.B : 0))

.ForMember(dest => dest.CClassCabins, opt => opt.MapFrom(src => src.Cabins != null ? src.Cabins.C : 0))

.ForMember(dest => dest.AClassPrice, opt => opt.MapFrom(src => src.Prices != null ? src.Prices.A : 0))

.ForMember(dest => dest.BClassPrice, opt => opt.MapFrom(src => src.Prices != null ? src.Prices.B : 0))

.ForMember(dest => dest.CClassPrice, opt => opt.MapFrom(src => src.Prices != null ? src.Prices.C : 0))

.ForMember(dest => dest.PerMile, opt => opt.MapFrom(src => src.Prices != null ? src.Prices.PerMile : 0));

CreateMap<TransportDTO, TransportMongoDB>()

.ForMember(dest => dest.Id, opt => opt.MapFrom(src => src.ObjectId))

.ForMember(dest => dest.ID, opt => opt.MapFrom(src => src.Id))

.ForMember(dest => dest.Name, opt => opt.MapFrom(src => src.Name))

.ForMember(dest => dest.Cabins, opt => opt.MapFrom(src => new Cabins

{

A = src.AClassCabins,

B = src.BClassCabins,

C = src.CClassCabins

}))

.ForMember(dest => dest.Prices, opt => opt.MapFrom(src => new Prices

{

A = src.AClassPrice,

B = src.BClassPrice,

C = src.CClassPrice,

PerMile = src.PerMile

}));

}

}

# **Приложение Б – Класс *UserRepository***

internal class UserRepository : IUserRepository

{

readonly string \_connectionString;

public UserRepository(string connectionString)

{

\_connectionString = connectionString;

}

public void Add(User entity)

{

using (var sqlConnection = new SqlConnection(\_connectionString))

{

sqlConnection.Open();

string query = "INSERT INTO Users (Email, Password, ID\_Clients, JobID) VALUES (@Email, @Password, @ID\_Clients, @JobID)";

SqlCommand command = new SqlCommand(query, sqlConnection);

command.Parameters.AddWithValue("@Email", entity.Email);

command.Parameters.AddWithValue("@Password", entity.Password);

if (entity.ID\_Clients != null && entity.ID\_Clients.Length > 0)

{

command.Parameters.AddWithValue("@ID\_Clients", string.Join(";", entity.ID\_Clients));

}

else

{

command.Parameters.AddWithValue("@ID\_Clients", DBNull.Value);

}

command.Parameters.AddWithValue("@JobID", entity.JobID);

command.ExecuteNonQuery();

}

}

public User CheckEmailAndPassword(string email, string password)

{

using (var sqlConnection = new SqlConnection(\_connectionString))

{

sqlConnection.Open();

string query = "SELECT \* FROM Users WHERE Email = @Email AND Password = @Password";

SqlCommand command = new SqlCommand(query, sqlConnection);

command.Parameters.AddWithValue("@Email", email);

command.Parameters.AddWithValue("@Password", password);

SqlDataReader reader = command.ExecuteReader();

var user = new User();

if (reader.Read())

{

int i = 0;

user.Email = reader.GetString(i++);

user.Password = reader.GetString(i++);

int[] idClients\_i;

if (!reader.IsDBNull(i))

{

string idClients\_s = reader.GetString(i);

idClients\_i = idClients\_s

.Split(';')

.Select(int.Parse)

.ToArray() ?? [0];

}

else idClients\_i = [0];

user.ID\_Clients = idClients\_i;

user.JobID = reader.GetInt32(++i);

return user;

}

else

{

throw new Exception("Invalid email or password.");

}

}

}

public void Delete(User entity)

{

using (var sqlConnection = new SqlConnection(\_connectionString))

{

sqlConnection.Open();

string query = "DELETE FROM Users WHERE Email = @Email";

SqlCommand command = new SqlCommand(query, sqlConnection);

command.Parameters.AddWithValue("@Email", entity.Email);

command.ExecuteNonQuery();

}

}

public User Get(string email)

{

using var sqlConnection = new SqlConnection(\_connectionString);

sqlConnection.Open();

string query = "SELECT \* FROM Users WHERE Email = @Email";

SqlCommand command = new SqlCommand(query, sqlConnection);

command.Parameters.AddWithValue("@Email", email);

SqlDataReader reader = command.ExecuteReader();

var user = new User();

if (reader.Read())

{

user.JobID = reader.GetInt32(reader.GetOrdinal("JobID"));

user.Email = reader.GetString(reader.GetOrdinal("Email"));

user.Password = reader.GetString(reader.GetOrdinal("Password"));

if (!reader.IsDBNull(reader.GetOrdinal("ID\_Clients")))

{

string idClients\_s = reader.GetString(reader.GetOrdinal("ID\_Clients"));

user.ID\_Clients = idClients\_s

.Split(';')

.Select(int.Parse)

.ToArray();

}

else

{

user.ID\_Clients = new int[0];

}

return user;

}

else

{

throw new Exception("No such user");

}

}

public IEnumerable<User> GetAll()

{

var users = new List<User>();

using (var sqlConnection = new SqlConnection(\_connectionString))

{

sqlConnection.Open();

string query = "SELECT \* FROM Users";

SqlCommand command = new SqlCommand(query, sqlConnection);

using (SqlDataReader reader = command.ExecuteReader())

{

while (reader.Read())

{

var user = new User

{

Email = reader.GetString(reader.GetOrdinal("Email")),

Password = reader.GetString(reader.GetOrdinal("Password")),

JobID = reader.GetInt32(reader.GetOrdinal("JobID")),

ID\_Clients = reader.IsDBNull(reader.GetOrdinal("ID\_Clients"))

? new int[0]

: reader.GetString(reader.GetOrdinal("ID\_Clients"))

.Split(';')

.Select(int.Parse)

.ToArray()

};

users.Add(user);

}

}

}

return users;

}

public void Update(User entity)

{

using (var sqlConnection = new SqlConnection(\_connectionString))

{

sqlConnection.Open();

string query = "UPDATE Users SET Password = @Password, ID\_Clients = @ID\_Clients, JobID = @JobID WHERE Email = @Email";

SqlCommand command = new SqlCommand(query, sqlConnection);

command.Parameters.AddWithValue("@Email", entity.Email);

command.Parameters.AddWithValue("@Password", entity.Password);

if (entity.ID\_Clients != null && entity.ID\_Clients.Length > 0)

{

command.Parameters.AddWithValue("@ID\_Clients", string.Join(";", entity.ID\_Clients));

}

else

{

command.Parameters.AddWithValue("@ID\_Clients", DBNull.Value);

}

command.Parameters.AddWithValue("@JobID", entity.JobID);

command.ExecuteNonQuery();

}

}

}

# **Приложение B – Класс *UserRepositoryMongoDB***

internal class UserRepositoryMongoDB : IUserRepositoryMongoDB

{

private readonly IMongoCollection<UserMongoDB> \_collection;

public UserRepositoryMongoDB(IMongoDatabase database)

{

\_collection = database.GetCollection<UserMongoDB>("Users");

}

public void Add(UserMongoDB entity)

{

\_collection.InsertOne(entity);

}

public UserMongoDB CheckEmailAndPassword(string email, string password)

{

var users = \_collection.Find(u => true).ToList();

var user = \_collection.Find(u => u.Email == email && u.Password == password).FirstOrDefault();

if (user == null)

{

throw new Exception("Invalid email or password.");

}

return user;

}

public void Delete(UserMongoDB entity)

{

\_collection.DeleteOne(u => u.Id == entity.Id);

}

public UserMongoDB Get(string id)

{

return \_collection.Find(u => u.Email == id).FirstOrDefault();

}

public IEnumerable<UserMongoDB> GetAll()

{

return \_collection.Find(u => true).ToList();

}

public void Update(UserMongoDB entity)

{

\_collection.ReplaceOne(u => u.Email == entity.Email, entity);

}

}

# **Приложение Г – Класс *UserService***

internal class UserService : IUserService

{

private IUserRepository \_userRepository;

private IMapper \_mapper;

public UserService(IUserRepository userRepository, IMapper mapper)

{

\_userRepository = userRepository;

\_mapper = mapper;

}

public void Add(UserDTO item)

{

var user = \_mapper.Map<User>(item);

\_userRepository.Add(user);

}

public void DeleteById(string id)

{

var user = \_userRepository.Get(id);

if (user != null)

{

\_userRepository.Delete(user);

}

}

public UserDTO FindById(string id)

{

var user = \_userRepository.Get(id);

return \_mapper.Map<UserDTO>(user);

}

public List<UserDTO> GetAll()

{

return \_userRepository.GetAll()

.Select(u => \_mapper.Map<UserDTO>(u))

.ToList();

}

public void Update(UserDTO item)

{

var user = \_mapper.Map<User>(item);

\_userRepository.Update(user);

}

public UserDTO? CheckEmailAndPasword(string email, string password)

{

var user = \_userRepository.CheckEmailAndPassword(email, password);

if (user != null)

{

return \_mapper.Map<UserDTO>(user);

}

return null;

}

}

# **Приложение Д – Класс *UserDTO***

public class UserDTO

{

public string Email { get; set; }

public string Password { get; set; }

public int[] ID\_Clients { get; set; }

public int JobID { get; set; }

public override string ToString()

{

string clientsIdString = ID\_Clients != null ? string.Join(", ", ID\_Clients) : "N/A";

return $"{Email,-25} | {Password,-15} | {clientsIdString,-20} | {JobID,-10}";

}

public static string GetHeader()

{

return $"{"Email",-25} | {"Password",-15} | {"ClientsId",-20} | {"JobID",-10}";

}

}

# **Приложение Е – Класс *UserServiceTests***

public class UserServiceTests : IDisposable

{

private readonly Mock<IUserRepository> \_mockRepository;

private readonly IMapper \_mapper;

private readonly UserService \_service;

public UserServiceTests()

{

\_mockRepository = new Mock<IUserRepository>();

var config = new MapperConfiguration(cfg =>

{

cfg.CreateMap<UserDTO, User>();

cfg.CreateMap<User, UserDTO>();

});

\_mapper = config.CreateMapper();

\_service = new UserService(\_mockRepository.Object, \_mapper);

}

public void Dispose()

{

// Clean up resources if needed

}

[Fact]

public void Add\_ShouldCallRepositoryAdd()

{

// Arrange

var userDto = new UserDTO

{

Email = "test@example.com",

Password = "password123",

ID\_Clients = new int[] { 1, 2, 3 },

JobID = 1

};

// Act

\_service.Add(userDto);

// Assert

\_mockRepository.Verify(repo => repo.Add(It.IsAny<User>()), Times.Once);

}

[Fact]

public void DeleteById\_ExistingEmail\_ShouldCallRepositoryDelete()

{

// Arrange

var user = new User { Email = "test@example.com", Password = "password123" };

\_mockRepository.Setup(repo => repo.Get("test@example.com")).Returns(user);

// Act

\_service.DeleteById("test@example.com");

// Assert

\_mockRepository.Verify(repo => repo.Delete(user), Times.Once);

}

[Fact]

public void DeleteById\_NonExistingEmail\_ShouldNotCallRepositoryDelete()

{

// Arrange

\_mockRepository.Setup(repo => repo.Get("test@example.com")).Returns((User)null);

// Act

\_service.DeleteById("test@example.com");

// Assert

\_mockRepository.Verify(repo => repo.Delete(It.IsAny<User>()), Times.Never);

}

[Fact]

public void FindById\_ExistingEmail\_ShouldReturnUserDTO()

{

// Arrange

var user = new User { Email = "test@example.com", Password = "password123", ID\_Clients = new int[] { 1, 2, 3 }, JobID = 1 };

\_mockRepository.Setup(repo => repo.Get("test@example.com")).Returns(user);

// Act

var result = \_service.FindById("test@example.com");

// Assert

Assert.NotNull(result);

Assert.Equal("test@example.com", result.Email);

Assert.Equal("password123", result.Password);

Assert.Equal(new int[] { 1, 2, 3 }, result.ID\_Clients);

Assert.Equal(1, result.JobID);

}

[Fact]

public void GetAll\_ShouldReturnListOfUserDTOs()

{

// Arrange

var users = new List<User>

{

new User { Email = "test1@example.com", Password = "password1", ID\_Clients = new int[] { 1 }, JobID = 1 },

new User { Email = "test2@example.com", Password = "password2", ID\_Clients = new int[] { 2 }, JobID = 2 }

};

\_mockRepository.Setup(repo => repo.GetAll()).Returns(users);

// Act

var result = \_service.GetAll();

// Assert

Assert.Equal(2, result.Count);

Assert.Equal("test1@example.com", result[0].Email);

Assert.Equal("test2@example.com", result[1].Email);

}

[Fact]

public void Update\_ShouldCallRepositoryUpdate()

{

// Arrange

var userDto = new UserDTO

{

Email = "test@example.com",

Password = "newpassword",

ID\_Clients = new int[] { 1, 2 },

JobID = 2

};

// Act

\_service.Update(userDto);

// Assert

\_mockRepository.Verify(repo => repo.Update(It.IsAny<User>()), Times.Once);

}

[Fact]

public void CheckEmailAndPassword\_ValidCredentials\_ShouldReturnUserDTO()

{

// Arrange

var user = new User

{

Email = "test@example.com",

Password = "password123",

ID\_Clients = new int[] { 1, 2, 3 },

JobID = 1

};

\_mockRepository.Setup(repo => repo.CheckEmailAndPassword("test@example.com", "password123")).Returns(user);

// Act

var result = \_service.CheckEmailAndPasword("test@example.com", "password123");

// Assert

Assert.NotNull(result);

Assert.Equal("test@example.com", result.Email);

Assert.Equal("password123", result.Password);

Assert.Equal(new int[] { 1, 2, 3 }, result.ID\_Clients);

Assert.Equal(1, result.JobID);

}

[Fact]

public void CheckEmailAndPassword\_InvalidEmail\_ShouldReturnNull()

{

// Arrange

\_mockRepository.Setup(repo => repo.Get("invalid@example.com")).Returns((User)null);

// Act

var result = \_service.CheckEmailAndPasword("invalid@example.com", "password123");

// Assert

Assert.Null(result);

}

[Fact]

public void CheckEmailAndPassword\_InvalidPassword\_ShouldReturnNull()

{

// Arrange

var user = new User

{

Email = "test@example.com",

Password = "password123"

};

\_mockRepository.Setup(repo => repo.Get("test@example.com")).Returns(user);

// Act

var result = \_service.CheckEmailAndPasword("test@example.com", "wrongpassword");

// Assert

Assert.Null(result);

}

}

# **Приложение Ж – Инъекции зависимостей**

public static class ConfigurationExtensions // DAL

{

public static void ConfigureDAL(this IServiceCollection services, string conn, string db\_type)

{

switch (db\_type)

{

case "MS SQL":

services.AddScoped<SqlConnection>(provider =>

{

var connection = new SqlConnection(conn);

connection.Open();

return connection;

});

services.AddSingleton<string>(conn);

services.AddTransient<IClientRepository, ClientRepository>();

services.AddTransient<IEmployeeRepository, EmployeeRepository>();

services.AddTransient<IPriceListRepository, PriceListRepository>();

services.AddTransient<IRouteRepository, RouteRepository>();

services.AddTransient<ITicketRepository, TicketRepository>();

services.AddTransient<ITransportRepository, TransportRepository>();

services.AddTransient<IUserRepository, UserRepository>();

services.AddTransient<IUserTypeRepository, UserTypeRepository>();

break;

case "Mongo DB":

var mongoUrl = MongoUrl.Create(conn);

var mongoClient = new MongoClient(mongoUrl);

var mongoDatabase = mongoClient.GetDatabase(mongoUrl.DatabaseName);

services.AddSingleton<IMongoDatabase>(mongoDatabase);

services.AddTransient<IClientRepositoryMongoDB, ClientRepositoryMongoDB>();

services.AddTransient<IEmployeeRepositoryMongoDB, EmployeeRepositoryMongoDB>();

services.AddTransient<IRouteRepositoryMongoDB, RouteRepositoryMongoDB>();

services.AddTransient<ITicketRepositoryMongoDB, TicketRepositoryMongoDB>();

services.AddTransient<ITransportRepositoryMongoDB, TransportRepositoryMongoDB>();

services.AddTransient<IUserRepositoryMongoDB, UserRepositoryMongoDB>();

services.AddTransient<IUserTypeRepositoryMongoDB, UserTypeRepositoryMongoDB>();

break;

default:

throw new ArgumentException("Unsupported databse type.");

}

}

}

public static class ConfigurationExtensions // BLL

{

public static void ConfigureBLL(this IServiceCollection services, string connection, string db\_type)

{

services.ConfigureDAL(connection, db\_type);

switch (db\_type)

{

case "MS SQL":

services.AddAutoMapper(

typeof(ClientProfile),

typeof(EmployeeProfile),

typeof(PriceListProfile),

typeof(RouteProfile),

typeof(TicketProfile),

typeof(TransportProfile),

typeof(UserProfile),

typeof(UserTypeProfile));

services.AddTransient<IClientService, ClientService>();

services.AddTransient<IEmployeeService, EmployeeService>();

services.AddTransient<IPriceListService, PriceListService>();

services.AddTransient<IRouteService, RouteService>();

services.AddTransient<ITicketService, TicketService>();

services.AddTransient<ITransportService, TransportService>();

services.AddTransient<IUserService, UserService>();

services.AddTransient<IUserTypeService, UserTypeService>();

break;

case "Mongo DB":

services.AddAutoMapper(

typeof(ClientProfile),

typeof(EmployeeProfile),

typeof(PricesProfile),

typeof(RouteMongoDBProfile),

typeof(TicketProfile),

typeof(TransportMongoDBProfile),

typeof(UserProfileMongoDB),

typeof(UserTypeProfile),

typeof(ReservedCabinsProfile),

typeof(ClientMongoDBProfile),

typeof(TicketMongoDBProfile),

typeof(EmployeeMongoDBProfile));

services.AddTransient<IUserService, UserServiceMongoDB>();

services.AddTransient<IClientService, ClientServiceMongoDB>();

services.AddTransient<IRouteService, RouteServiceMongoDB>();

services.AddTransient<ITicketService, TicketServiceMongoDB>();

services.AddTransient<ITransportService, TransportServiceMongoDB>();

services.AddTransient<IEmployeeService, EmployeeServiceMongoDB>();

break;

default:

throw new ArgumentException("Unsupported databse type.");

}

}

}

private void InitializeDependencies(string db\_type) // UI

{

var serviceCollection = new ServiceCollection();

switch (db\_type)

{

case "MS SQL":

serviceCollection.ConfigureBLL(ConfigurationManager.ConnectionStrings["DBConnectionMSSQL"].ConnectionString, "MS SQL");

break;

case "Mongo DB":

serviceCollection.ConfigureBLL(ConfigurationManager.ConnectionStrings["DBConnectionMongoDB"].ConnectionString, "Mongo DB");

break;

}

var serviceProvider = serviceCollection.BuildServiceProvider();

\_userService = serviceProvider.GetRequiredService<IUserService>();

if (db\_type == "MS SQL") \_priceListService = serviceProvider.GetRequiredService<IPriceListService>();

\_clientService = serviceProvider.GetRequiredService<IClientService>();

\_routeService = serviceProvider.GetRequiredService<IRouteService>();

\_ticketService = serviceProvider.GetRequiredService<ITicketService>();

\_employeeService = serviceProvider.GetRequiredService<IEmployeeService>();

\_transportService = serviceProvider.GetRequiredService<ITransportService>();

}