**Министерство цифрового развития, связи и массовых коммуникаций РФ**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Сибирский государственный университет телекоммуникаций и информатики»**

**(СибГУТИ)**

Кафедра ПМиК

Зав. кафедрой

\_\_\_\_\_\_\_\_\_\_\_ /И.В. Нечта/

### **ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

по направлению 09.03.01 «Информатика и вычислительная техника»

профиль Программное обеспечение средств вычислительной техники и автоматизированных систем

Разработка программного обеспечения для создания сервиса перевозки попутных грузов.

**Студент**  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Минатулаев Мухаммад Русланович /

**Институт информатики и вычислительной техники**

**Группа -ИП-011**

**Руководитель** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Приставка Павел Анатольевич /

Новосибирск 2024 г.

**Министерство цифрового развития, связи и массовых коммуникаций РФ**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Сибирский государственный университет телекоммуникаций и информатики»**

**(СибГУТИ)**

**КАФЕДРА Прикладной математики и кибернетики**

**ЗАДАНИЕ**

**НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

|  |  |
| --- | --- |
| СТУДЕНТА Минатулаева Мухаммада Руслановича | ГРУППЫ ИП-011 |

УТВЕРЖДАЮ

зав. кафедрой ПМиК

**/** Нечта И.В./

Новосибирск 2024 г.

**1. Тема выпускной квалификационной работы бакалавра**

Разработка программного обеспечения для создания сервиса

перевозки попутных грузов

утверждена приказом СибГУТИ от «26» января 2024 г. №4/113о-24

**2.Срок сдачи студентом законченной работы** «24» июня 2024 г.

**3.Исходные данные к работе**

1 Руководство по ASP.NET Core MVC [Электронный ресурс] https://metanit.com/sharp/aspnetmvc/

2 Документация по языку C# [Электронный ресурс] https://learn.microsoft.com/ru-ru/dotnet/csharp/

|  |  |
| --- | --- |
| **4. Перечень подлежащих разработке вопросов (Содержание пояснительной записки)** | **Сроки выполнения**  **по разделам** |
| 1. Утверждение темы ВКР | 16.10.2023 |
| 2. Постановка задачи | 22.04.2024 – 25.04.2024 |
| 3. Работа с библиотечными фондами, сбор и анализ материалов по теме практики | 26.04.2024 – 29.04.2024 |
| 4. Изучение современного состояния предметной области | 30.04.2024 – 07.05.2024 |
| 5. Формирование требований к конечному продукту | 08.05.2024 – 14.05.2024 |
| 6. Выбор технологического стека | 15.05.2024 – 21.05.2024 |
| 7. Реализация программного продукта | 22.05.2024 – 10.06.2024 |
| 8. Заключение | 11.06.2024 – 17.06.2024 |

Задание выдано: Задание принял к исполнению

«16» октября 2023 г. «22» апреля 2024 г.

\_\_\_\_\_\_\_\_\_\_\_\_\_ /Приставка П.А./ \_\_\_\_\_\_\_\_\_\_\_ /Минатулаев М.Р/

*подпись (ФИО руководителя) подпись (ФИО студента)*

**Министерство цифрового развития, связи и массовых коммуникаций РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Сибирский государственный университет телекоммуникаций и информатики»**

**(СибГУТИ)**

направление подготовки 09.03.01 Информатика и вычислительная техника

направленность (профиль): «Программное обеспечение средств вычислительной техники и автоматизированных систем»

**АННОТАЦИЯ**

Выпускной квалификационной работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_Минатулаева М.Р.\_\_\_\_\_\_\_\_\_\_\_\_

(Фамилия,И.О.)

по теме «Разработка программного обеспечения для создания сервиса перевозки попутных грузов»

Объём работы – 45 страниц, на которых размещены 17рисунков. При написании работы использовалось 5источников.

Ключевые слова: веб-приложение, Visual Studio, C#, ASP.NET Core, MVC, SQL Server Management Studio.

Работа выполнена СибГУТИ, кафедра прикладной математики и кибернетики (ПМиК)

(название предприятия, подразделения)

Руководитель доцент каф. ПМиК, к.т.н., Приставка Павел Анатольевич

(должность, уч.степень, звание, Фамилия Имя Отчество*)*

Цель работы

Разработка программного обеспечения для создания сервиса перевозки попутных грузов

Решаемые задачи

Анализ существующих решений, формирование требований к приложению, разработка

макета, проектирование архитектуры, реализация серверной и клиентской части

Основные результаты

В результате было реализовано веб-приложение, являющееся связующим звеном между

заказчиком и исполнителем, в процессе доставки попутных грузов.

Содержание

[Введение 3](#_Toc170056353)

[1 Обзор предметной области 5](#_Toc170056354)

[2 задачи 7](#_Toc170056355)

[3 ВЫБОР СРЕДСТВ РАЗРАБОТКИ 8](#_Toc170056356)

[3.1 Платформа 8](#_Toc170056357)

[3.2 Языки программирования 9](#_Toc170056358)

[3.3 Визуализация базы данных 10](#_Toc170056359)

[3.4 Структура архитектуры приложения 11](#_Toc170056360)

[3.5 Инструменты для работы с запросами 14](#_Toc170056361)

[4 разработка приложения 17](#_Toc170056362)

[5 Руководство пользователя 37](#_Toc170056363)

[Заключение 44](#_Toc170056364)

[список использованных источников и литературы 45](#_Toc170056365)

[Приложение А 46](#_Toc170056366)

Введение

Современное развитие технологий и цифровизация общества приводят к возникновению новых видов сервисов и платформ, направленных на оптимизацию и улучшение процессов в различных сферах жизни. Одним из таких направлений является создание сервисов перевозки попутных грузов, которые позволяют эффективно использовать ресурсы транспортных средств и снизить затраты на доставку грузов.

На сегодняшний день, взаимодействие людей, находящихся в разных городах, распространилось повсеместно, вместе с этим повысился и спрос на передачу товаров от одного человека к другому. С развитием цифровых технологий и изменением взаимодействия между людьми возникают новые возможности оптимизации и упрощения этой задачи. Создание веб-приложения для попутной перевозки грузов - надёжного и удобного сервиса - актуальное и важное предложение на рынке транспортных услуг.

Данный сервис может способствовать оптимизации использования имеющихся транспортных ресурсов. Путем заполнения свободных мест в грузовых транспортных средствах эти сервисы снижают издержки для владельцев транспорта и отправителей грузов. Это особенно важно в условиях растущей конкуренции и стремления компаний к сокращению операционных расходов. Использование свободных мест в транспортных средствах помогает снизить количество необходимых рейсов и, следовательно, выбросы вредных веществ в атмосферу. Это соответствует стремлению многих стран и организаций к уменьшению углеродного следа и улучшению экологической устойчивости. Разработка программного обеспечения для таких сервисов требует внедрения инновационных технологий, таких как алгоритмы маршрутизации, системы управления заказами, онлайн-платформы для взаимодействия между пользователями и владельцами транспорта. Это способствует развитию технологической сферы и повышению конкурентоспособности компаний.

Для решения, выше описанных, задач, как раз и подходит данное веб-приложение, которое может иметь широкий спектр привлеченной аудитории, от обычных людей, с их повседневными потребностями, до мелких или даже крупных компаний. Пользователи могут с легкостью найти удобный для себя способ перевозки грузов и сэкономить на транспортных затратах, а также получить дополнительный заработок, предлагая свободное место в автомобиле для груза. Разработка веб-приложения для попутной перевозки грузов представляет собой не только бизнес-возможность, но и социально значимый шаг в направлении улучшения эффективности использования транспортных ресурсов.

Благодаря использованию современных информационных технологий, такие сервисы обеспечивают прозрачность и удобство взаимодействия между перевозчиками и отправителями, а также повышают безопасность и надежность процесса перевозки. Это делает данную тему не только актуальной, но и перспективной для дальнейшего развития и внедрения в практику.

В рамках данной работы будут рассмотрены основные технологии и методы, используемые для создания подобных сервисов, а также проведен анализ потребностей пользователей и особенностей организации попутных перевозок грузов. Результаты работы могут быть полезны как для предпринимателей, заинтересованных в создании подобных сервисов, так и для конечных пользователей, ищущих эффективные решения для транспортировки грузов.

# Обзор предметной области

Текущее состояние рынка перевозок попутных грузов характеризуется рядом основных тенденций и характеристик, которые важно учитывать при разработке программного обеспечения для создания соответствующего сервиса. С увеличением объемов грузоперевозок растет и спрос на более эффективные и удобные способы доставки грузов. Пользователи все чаще обращают внимание на возможность перевозки грузов попутными транспортными средствами в целях снижения затрат и экологической эффективности. В связи с усилением внимания к проблемам окружающей среды и устойчивому развитию, сервисы, способствующие уменьшению числа пустых рейсов и оптимизации использования транспортных средств, становятся все более востребованными.

В основном на рынке присутствует ряд крупных компаний, предоставляющих услуги по перевозке грузов, как в международном, так и внутреннем масштабе. Это может включать логистические компании, транспортные сети и частных перевозчиков. Перед разработкой программного обеспечения необходимо изучить и проанализировать их стратегии и условия предоставления услуг.

Одними из представителей крупных логистических компаний, являются "Деловые линии", "ПЭК", "Байкал-Сервис" и другие, которые предоставляют широкий спектр услуг по перевозке грузов как по России, так и за ее пределами. Эти компании обычно обладают обширной транспортной сетью и высоким уровнем технической оснащенности.

Такие крупные компании, в основном, заинтересованы в доставке большого количества грузов за раз, это в свою очередь приводит к тому, что осуществление доставки происходит чаще всего не в день регистрации груза, а либо в какой-то нормированный день, либо по достижении полной загруженности грузового автомобиля. Все это приводит к тому, что сроки доставки могут затянутся.

Конечно, для таких случаев данные компании предоставляет услугу по срочной доставке товара, которая будет осуществляться в кратчайшие сроки, но и стоит это будет существенно дороже.

Так же существует большое количество небольших транспортных компаний и перевозчиков, осуществляющих перевозки грузов на региональном и местном уровне. Эти компании могут предлагать более гибкие условия и цены, особенно для небольших заказов.

Данный вид предложений можно встретить на множестве онлайн-рынков и платформ, специализирующихся на предоставлении услуг перевозки грузов. Это включает в себя такие платформы, как Avito, Юла, Доска объявлений и т.д., где частные лица могут размещать свои объявления о перевозке грузов. Но одним из минусов таких платформ в том, что множество объявлений на таких платформах может сделать поиск подходящего перевозчика трудоемким и затруднительным.

Также существуют специализированные онлайн-платформы и приложения, созданные специально для организации перевозки грузов. Некоторые из них могут предоставлять возможность поиска и заказа грузоперевозок, а также оценку и отзывы о перевозчиках. Примеры таких сервисов в России могут включать Яндекс.Такси, Gett, и другие. Но и у таких платформ есть свои минусы, например, ограниченная география действия: некоторые специализированные сервисы могут быть доступны только в крупных городах или ограниченных регионах, что может ограничить выбор перевозчиков для грузоперевозок в отдаленных или малонаселенных районах. Так же не стоит забывать, что некоторые специализированные сервисы могут взимать высокие комиссии с перевозчиков, что может привести к повышению стоимости услуг для клиентов.

Для того чтобы конкурировать с уже существующими компаниями, необходимо иметь отличительные особенности. Например, осуществлять доставку по межгороду при помощи Яндекс сервисов может обойтись очень дорого, а такая крупная компания как Blablacar в принципе не позволяют осуществлять доставку грузов или животных. Таким образом нашей задачей будет являться реализация такого приложения, которое позволило бы за заранее оговоренную приемлемую сумму денег, осуществлять доставку любых грузов по всей стране.

Исходя из нашего анализа, можно сделать вывод, что существует потребность в создании узкоспециализированной платформы, которая будет полностью удовлетворять всем требованиям, как со стороны клиентов, которые хотят отправить посылку, так и со стороны тех, кто эту посылку будет доставлять.

# задачи

Цель: разработать программное обеспечение для создания сервиса перевозки попутных грузов, в виде веб-приложения.

Задачи:

- Реализовать хранение необходимой информации с использованием БД:

1) Добавление в БД информации о регистрирующемся в нашем приложении пользователе. Его адресе проживания, контактных данных.

2) Добавление в БД таблицы для хранения комментариев с оценками для конкретного пользователя, оставленные другими людьми после осуществления доставки необходимого товара.

3) Добавление в БД информации о заказе, которой конкретный пользователь решил разместить на сайте. Его физические показатели (вес, габариты и по желанию краткое описание товара). А также конкретный адрес доставки. Запросы в БД для получения той или иной необходимой информации.

- Создание регистрации и авторизации на сайте:

Реализация функционала регистрации и авторизации пользователей.

- Создание пользовательского интерфейса:

Разработка удобного и интуитивно понятного интерфейса для пользователей.

- Разработка функционала формирования маршрутов:

Использование API картографических сервисов для визуализации маршрутов доставки.

- Внедрение социальной функциональности:

1) Реализация страницы с информацией о пользователях в ролях "Доставщик" и "Отправитель".

2) Внедрение системы оценки качества доставки груза.

- Тестирование и оптимизация:

1) Проведение тестирования функционала приложения на различных устройствах и в различных браузерах.

2) Оптимизация производительности приложения для быстрой загрузки и отзывчивости интерфейса.

# ВЫБОР СРЕДСТВ РАЗРАБОТКИ

## Платформа

### Visual Studio

Для разработки программного обеспечения была выбрана платформа «Visual Studio».

«Visual Studio» — это интегрированная среда разработки (IDE) от компании Microsoft, предоставляющая широкий спектр инструментов для разработки, отладки и развертывания программного обеспечения на различных языках программирования.

Эта платформа была выбрана, так как она обладает требуемыми критериями для выполнения проекта.

Ключевыми преимуществами данной платформы стали:

* Множество обучающих материалов, гайдов и видеоуроков, которые можно получить бесплатно.
* Широкая поддержка различных языков программирования, включая C#, C++, Python, JavaScript и многие другие.
* Интеграция с платформой Azure для разработки облачных приложений.
* Встроенные инструменты для тестирования и отладки, включая профилировщик производительности и возможность пошаговой отладки.
* Поддержка разработки под различные платформы, включая Windows, macOS, iOS и Android.
* «Visual Studio» позволяет разработчикам создавать мощные приложения, используя облачные технологии, и интегрировать их с различными сервисами, что делает разработку более гибкой и эффективной.

### SQL Server Management Studio (SSMS)

Для работы с базами данных была выбрана платформа «SQL Server Management Studio» (SSMS).

«SQL Server Management Studio» — это интегрированная среда от компании Microsoft, предназначенная для управления и администрирования Microsoft SQL Server. Она предоставляет набор инструментов для настройки, управления и администрирования баз данных.

Эта платформа была выбрана, так как она обладает требуемыми критериями для выполнения проекта.

Ключевыми преимуществами данной платформы стали:

* Обширная документация и обучающие материалы, доступные бесплатно.
* Мощные инструменты для управления базами данных, включая графический интерфейс для написания запросов и выполнения скриптов.
* Возможность подключения и управления множественными серверами и базами данных из одного интерфейса.
* Интеграция с инструментами аналитики и отчетности, такими как SQL Server Reporting Services (SSRS) и SQL Server Analysis Services (SSAS).
* Поддержка различных типов данных и возможность работы с большими объемами данных.
* SSMS позволяет администраторам и разработчикам баз данных эффективно управлять данными, оптимизировать производительность запросов и обеспечивать безопасность и целостность данных в облачных и локальных средах.

## Языки программирования

1. Язык программирования: для разработки используется C# с ASP.NET Core. C# представляет собой мощный язык программирования с обширным набором инструментов для веб-разработки. ASP.NET Core, в свою очередь, является современным и гибким фреймворком для создания веб-приложений, обеспечивая высокую производительность и масштабируемость.
2. Интегрированная среда разработки (IDE): В качестве IDE используется Visual Studio. Эта среда разработки предоставляет широкий набор инструментов для работы с C# и ASP.NET Core, включая возможности отладки, автоматической сборки и развертывания приложений.
3. База данных: для хранения данных используется SQL Server с помощью SQL Server Management Studio. SQL Server обладает высокой производительностью и масштабируемостью, предоставляя широкий набор функций для управления данными. SQL Server Management Studio обеспечивает удобное администрирование баз данных.
4. Дополнительные инструменты и библиотеки:
   * Entity Framework Core: используется для работы с данными и объектно-реляционного отображения (ORM).
   * ASP.NET Core Identity: применяется для реализации аутентификации и авторизации пользователей.
   * ASP.NET Core MVC: используется для построения веб-интерфейса и управления запросами.
   * Bootstrap: Набор инструментов для создания пользовательского интерфейса.

Яндекс карты API и Bing maps API: предоставляют инструменты для отображения интерактивных карт на веб-страницах.

## Визуализация базы данных

SQL Server Management Studio (SSMS) предоставляет мощный инструмент для визуализации структуры базы данных — "Диаграммы базы данных". Этот инструмент позволяет создавать диаграммы, отображающие зависимости и взаимосвязи между таблицами в одной базе данных.

#### Основные возможности и преимущества:

* **Визуализация структуры базы данных:** Диаграммы на рисунке 3.1 позволяют наглядно увидеть, как таблицы в базе данных связаны между собой. Это особенно полезно для сложных баз данных с множеством взаимосвязанных таблиц.
* **Автоматическое обнаружение связей:** SSMS автоматически отображает внешние ключи и связи между таблицами, что позволяет быстро понять, как данные организованы и взаимодействуют друг с другом.
* **Удобное редактирование:** Пользователи могут добавлять и удалять таблицы с диаграммы, а также изменять связи между таблицами прямо в интерфейсе диаграммы. Изменения автоматически применяются к структуре базы данных.
* **Создание новых связей и изменений структуры:** Помимо отображения существующих связей, диаграммы позволяют добавлять новые связи (например, определять новые внешние ключи) и изменять структуру таблиц.
* **Документирование базы данных:** Диаграммы можно сохранить и использовать для документации структуры базы данных, что облегчает передачу знаний между разработчиками и администраторами.
* **Печать и экспорт диаграмм:** Диаграммы можно распечатать или экспортировать в различные форматы, такие как изображение или PDF, что удобно для презентаций и отчетов.

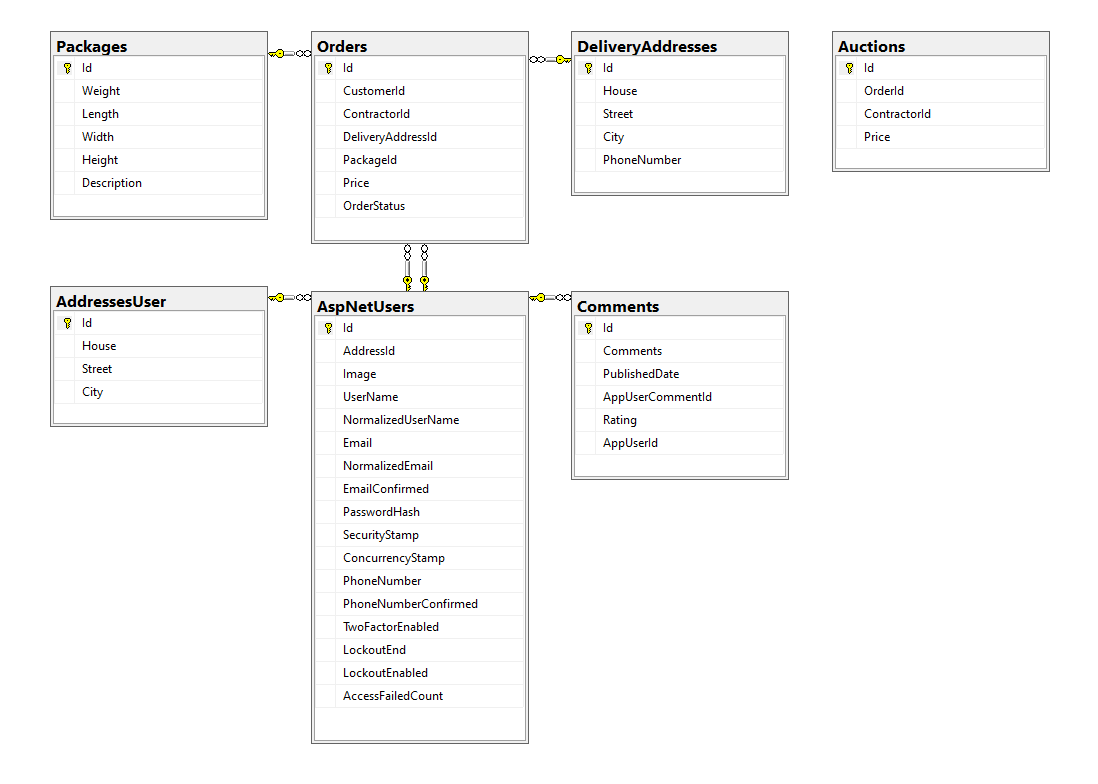


Рисунок 3.1 – Схема базы данных

## Структура архитектуры приложения

Для реализации структуры приложения использовался шаблон MVC (Model-View-Controller) в ASP.NET Core, который представляет собой архитектурный паттерн, разделяющий приложение на три основных компонента: модели, представления и контроллеры. Это способствует упрощению разработки, тестирования и сопровождения приложений за счет четкого разделения ответственности.

На рисунке 3.2 представлена схема шаблона MVC.

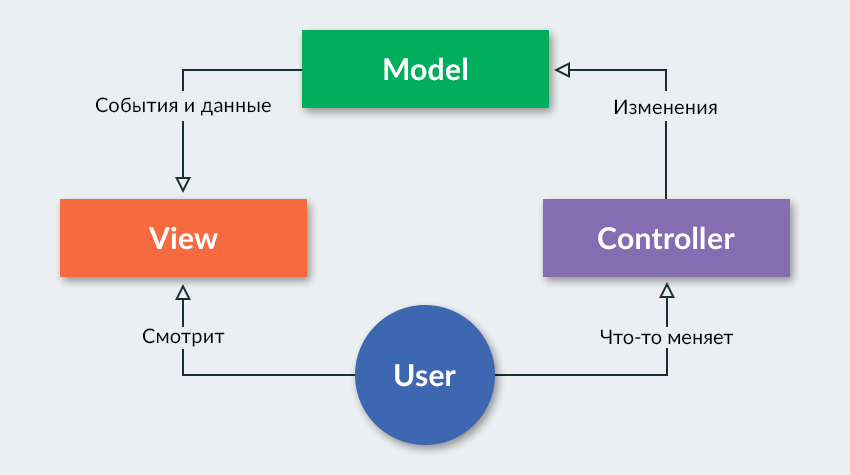


Рисунок 3.2 – Схема модели MVC

### Основные компоненты MVC:

1. **Model (Модель)**:
   * **Описание данных и бизнес-логики**: Модель отвечает за представление данных, бизнес-логики и правил валидации. Она определяет, как данные хранятся, изменяются и обрабатываются.
   * **Взаимодействие с базой данных**: В ASP.NET Core модели часто используются вместе с Entity Framework для взаимодействия с базой данных. Модели могут представлять таблицы базы данных и содержат свойства, соответствующие колонкам этих таблиц.
   * **Примеры**: Класс, представляющий пользователя, продукт, заказ и т.д.
2. **View (Представление)**:
   * **Отображение данных**: Представление отвечает за отображение данных пользователю. Оно получает данные от контроллера и рендерит их в HTML для отображения в браузере.
   * **Шаблоны**: Представления обычно создаются с использованием Razor синтаксиса, который позволяет смешивать HTML и C# код.
   * **Отделение логики отображения**: В представлении не должна находиться бизнес-логика; его основная задача – отображение данных, полученных от контроллера.
   * **Примеры**: Страницы, показывающие списки продуктов, формы для ввода данных и т.д.
3. **Controller (Контроллер)**:
   * **Обработка пользовательских запросов**: Контроллер принимает входящие HTTP-запросы, обрабатывает их и возвращает соответствующий ответ (представление или данные в формате JSON).
   * **Взаимодействие с моделью и представлением**: Контроллер запрашивает данные у модели, обрабатывает их (если необходимо) и передает результат представлению для отображения.
   * **Управление потоками данных**: Контроллер управляет потоками данных между моделью и представлением, обеспечивая логическое разделение процессов обработки данных и их отображения.
   * **Примеры**: Контроллеры для управления пользователями, заказами, продуктами и т.д.

### Преимущества использования шаблона MVC:

1. **Разделение обязанностей**: Четкое разделение обязанностей между моделью, представлением и контроллером облегчает разработку и сопровождение кода. Это позволяет разработчикам работать параллельно над различными компонентами приложения.
2. **Тестируемость**: Благодаря отделению логики обработки данных (контроллеры и модели) от логики отображения (представления), тестировать приложение становится проще. Это способствует созданию модульных тестов для бизнес-логики и контроллеров.
3. **Повторное использование кода**: Модели и контроллеры могут использоваться повторно в различных представлениях, что снижает количество дублирующегося кода и облегчает сопровождение.
4. **Четкая структура**: Шаблон MVC способствует созданию четкой и организованной структуры проекта, что упрощает навигацию и понимание кода для новых разработчиков.

### Принцип работы в контексте ASP.NET Core:

1. **Запрос пользователя**: Пользователь отправляет HTTP-запрос (например, при переходе по URL или отправке формы).
2. **Роутинг**: ASP.NET Core использует систему маршрутизации для определения того, какой контроллер и какое действие этого контроллера должны обработать данный запрос.
3. **Обработка контроллером**: Контроллер получает запрос, взаимодействует с моделью для получения или обработки данных, и затем выбирает соответствующее представление для возврата пользователю.
4. **Рендеринг представления**: Представление получает данные от контроллера и рендерит их в HTML для отображения в браузере пользователя.
5. **Ответ пользователю**: Сформированный HTML-код возвращается пользователю в виде HTTP-ответа, который отображается в браузере.

Таким образом, шаблон MVC в ASP.NET Core обеспечивает структурированную и гибкую архитектуру для построения веб-приложений, что упрощает разработку, тестирование и поддержку.

## Инструменты для работы с запросами

Для удобной работы с SQL-запросами была технология Entity Framework Core. Entity Framework Core (EF Core) — это объектно-реляционный диспетчер (ORM) для .NET, который упрощает взаимодействие между приложениями и базами данных. EF Core предоставляет разработчикам инструмент для работы с данными в виде объектов .NET, абстрагируя сложности взаимодействия с реляционными базами данных:

1. ORM (Object-Relational Mapping):
   * EF Core позволяет разработчикам работать с базами данных, используя объекты .NET. Это устраняет необходимость писать большую часть SQL-кода вручную, заменяя его на манипуляции с объектами.
2. Кроссплатформенность:
   * EF Core является кроссплатформенным инструментом и поддерживает работу на различных операционных системах, включая Windows, macOS и Linux.
3. Поддержка различных баз данных:
   * EF Core поддерживает множество провайдеров баз данных, таких как Microsoft SQL Server, SQLite, PostgreSQL, MySQL и другие. Это позволяет разработчикам работать с различными системами управления базами данных (СУБД).
4. Линейный запрос (LINQ):
   * EF Core интегрируется с LINQ, предоставляя возможность писать запросы к базе данных на языке C#. Эти запросы затем переводятся в SQL и выполняются на сервере базы данных.
5. Модели данных:
   * Разработчики могут определять модели данных, которые представляют структуру таблиц базы данных. Эти модели описываются с помощью классов .NET и могут быть настроены с использованием Fluent API или атрибутов.
6. Миграции:
   * EF Core поддерживает механизм миграций, который позволяет управлять изменениями схемы базы данных. Миграции помогают создавать, обновлять и откатывать изменения в структуре базы данных, сохраняя целостность данных.
7. Трекер изменений:
   * EF Core отслеживает изменения, сделанные в объектах, загруженных из базы данных. Это позволяет автоматически генерировать SQL-команды для обновления базы данных в соответствии с этими изменениями.
8. Загрузка связанных данных:
   * EF Core поддерживает ленивую, явную и жадную загрузку связанных данных, что позволяет управлять тем, какие данные загружаются из базы данных и когда.
9. Работа с транзакциями:
   * EF Core позволяет выполнять несколько операций с базой данных в рамках транзакции, обеспечивая целостность и согласованность данных.
10. Конфигурация и расширяемость:
    * EF Core предоставляет множество возможностей для конфигурации и расширяемости, включая настраиваемые соглашения, перехватчики запросов и другие расширения.
11. Поддержка не-реляционных баз данных:
    * Помимо реляционных баз данных, EF Core может работать с не-реляционными источниками данных, например, с базами данных типа Azure Cosmos DB.

Entity Framework Core был на необходим, для того чтобы мы могли использовать входящий в нее DBContext, который является центральной частью работы с базой данных в приложениях ASP.NET Core, обеспечивая удобный и мощный API для выполнения большинства задач, связанных с доступом к данным. DBContext в ASP.NET Core используется для управления взаимодействием с базой данных через Entity Framework Core, предоставляя набор функциональных возможностей для выполнения основных операций CRUD (создание, чтение, обновление и удаление) с данными:

1. Управление подключением к базе данных:
   * DBContext инкапсулирует логики работы с подключением к базе данных, управляя открытием и закрытием соединений.
2. Отслеживание изменений:
   * DBContext отслеживает изменения, сделанные в объектах, загруженных из базы данных, и синхронизирует эти изменения при сохранении.
3. Выполнение запросов:
   * С помощью DBContext можно выполнять LINQ-запросы для выборки данных из базы данных. Он преобразует LINQ-запросы в SQL-запросы и выполняет их на сервере базы данных.
4. Сохранение изменений:
   * DBContext предоставляет метод для сохранения изменений, сделанных в объектах, обратно в базу данных, генерируя и выполняя соответствующие SQL-команды.
5. Управление транзакциями:
   * Он может управлять транзакциями, что позволяет выполнить несколько операций как одну атомарную операцию, обеспечивая целостность данных.
6. Конфигурация моделей:
   * В DBContext можно настраивать модели данных с помощью Fluent API или атрибутов, задавая правила для таблиц, столбцов, связей и ограничений.
7. Миграции:
   * DBContext используется для применения миграций, которые обновляют схему базы данных в соответствии с изменениями моделей данных.
8. Работа с наборами данных (DbSet):
   * Он предоставляет коллекции объектов, представляющих таблицы базы данных, через свойства типа DbSet. Это позволяет легко управлять записями в таблицах.
9. Инъекция зависимостей:
   * DBContext легко интегрируется с системой внедрения зависимостей ASP.NET Core, что позволяет удобно управлять его жизненным циклом и зависимостями.

# 4 разработка приложения

4.1 Модели

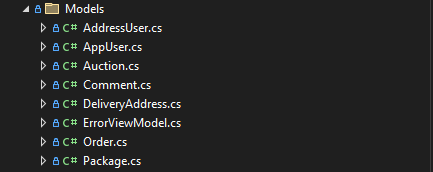


Рисунок 4.1 – Список моделей

### 1. AppUser

**Описание:** Модель AppUser представляет собой пользователя в системе, наследующего от IdentityUser (что подразумевает наличие базовых полей, таких как UserName, Email, PasswordHash и т.д.).

**Поля:**

* **AddressId**: Внешний ключ, ссылающийся на адрес пользователя (AddressUser).
* **Address**: Объект адреса пользователя, связанный через внешний ключ.
* **Image**: Необязательная строка для хранения URL или пути к изображению пользователя.
* **Comments**: Коллекция комментариев, оставленных пользователем.

**Зависимости:** Зависит от модели AddressUser и коллекции Comment.

### 2. Order

**Описание:** Модель Order представляет заказ в системе.

**Поля:**

* **Id**: Первичный ключ для идентификации заказа.
* **CustomerId**: Внешний ключ, ссылающийся на пользователя (заказчика).
* **Customer**: Объект пользователя (заказчика), связанный через внешний ключ.
* **ContractorId**: Внешний ключ, ссылающийся на пользователя (исполнителя).
* **Contractor**: Объект пользователя (исполнителя), связанный через внешний ключ.
* **DeliveryAddressId**: Внешний ключ, ссылающийся на адрес доставки.
* **DeliveryAddress**: Объект адреса доставки, связанный через внешний ключ.
* **PackageId**: Внешний ключ, ссылающийся на пакет.
* **Package**: Объект пакета, связанный через внешний ключ.
* **Price**: Цена заказа.
* **OrderStatus**: Статус заказа.

**Зависимости:** Зависит от моделей AppUser, DeliveryAddress, и Package.

### 3. AddressUser

**Описание:** Модель AddressUser представляет адрес пользователя.

**Поля:**

* **Id**: Первичный ключ для идентификации адреса.
* **House**: Номер дома.
* **Street**: Улица.
* **City**: Город.

**Зависимости:** Не имеет явных зависимостей, но используется в модели AppUser.

### 4. DeliveryAddress

**Описание:** Модель DeliveryAddress представляет адрес доставки заказа.

**Поля:**

* **Id**: Первичный ключ для идентификации адреса доставки.
* **House**: Номер дома.
* **Street**: Улица.
* **City**: Город.
* **PhoneNumber**: Номер телефона для связи.

**Зависимости:** Не имеет явных зависимостей, но используется в модели Order.

### 5. Package

**Описание:** Модель Package представляет посылку в заказе.

**Поля:**

* **Id**: Первичный ключ для идентификации пакета.
* **Weight**: Вес пакета.
* **Length**: Длина пакета.
* **Width**: Ширина пакета.
* **Height**: Высота пакета.
* **Description**: Необязательное описание пакета.

**Зависимости:** Не имеет явных зависимостей, но используется в модели Order.

### 6. Comment

**Описание:** Модель Comment представляет комментарий, оставленный пользователем.

**Поля:**

* **Id**: Первичный ключ для идентификации комментария.
* **Comments**: Текст комментария.
* **PublishedDate**: Дата публикации комментария.
* **AppUserCommentId**: Идентификатор пользователя, оставившего комментарий.
* **Rating**: Рейтинг (оценка).

**Зависимости:** Не имеет явных зависимостей, но используется в модели AppUser.

### 7. Auction

**Описание:** Модель Auction представляет аукцион на выполнение заказа.

**Поля:**

* **Id**: Первичный ключ для идентификации аукциона.
* **OrderId**: Идентификатор заказа.
* **ContractorId**: Идентификатор исполнителя.
* **Price**: Предложенная цена.

**Зависимости:** Не имеет явных зависимостей, но используется для связывания с моделями Order и AppUser.

### Общие зависимости

Модели зависят друг от друга через внешний ключ, что создает реляционные связи между ними. Например, модель Order зависит от моделей AppUser, DeliveryAddress и Package, создавая тем самым взаимосвязь между заказами, пользователями, адресами и посылками.

4.2 Контроллеры

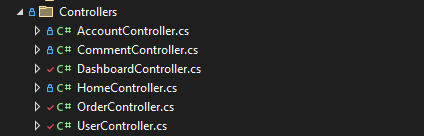


Рисунок 4.2 – Список контроллеров

### 4.2.1 Описание контроллера UserController

Контроллер UserController предназначен для управления операциями, связанными с пользователями в веб-приложении для перевозки грузов. Он взаимодействует с базой данных через ApplicationDbContext и репозиторием IUserRepository.

**Конструктор UserController(ApplicationDbContext context, IUserRepository userRepository):**

* **Назначение:** Инициализирует контроллер с контекстом базы данных и репозиторием пользователей.
* **Параметры:**
  + context: Объект типа ApplicationDbContext, который предоставляет доступ к базе данных.
  + userRepository: Интерфейс IUserRepository, предоставляющий методы для работы с данными пользователей.

#### Методы

**1. [HttpGet("users")] public async Task<IActionResult> Index()**

* **Назначение:** Получает список всех пользователей и отображает их на странице.
* **Что вызывает:**
  + Использует метод GetAllUsers из IUserRepository для получения всех пользователей.
  + Запрашивает комментарии каждого пользователя из контекста базы данных.
* **Что передает:**
  + Создает список объектов UserViewModel для представления пользователей.
  + Включает информацию о количестве комментариев и среднем рейтинге пользователя.
  + Возвращает представление со списком пользователей (View(result)).

**2. public async Task<IActionResult> Detail(string id)**

* **Назначение:** Получает подробную информацию о пользователе по его идентификатору и отображает её на странице.
* **Что вызывает:**
  + Использует метод GetUserById из IUserRepository для получения данных конкретного пользователя по его идентификатору.
* **Что передает:**
  + Создает объект UserDetailViewModel для представления подробной информации о пользователе.
  + Включает идентификатор, имя пользователя, адрес и изображение пользователя.
  + Возвращает представление с подробной информацией о пользователе (View(userDetailViewModel)).

### Взаимодействие с базой данных и репозиторием

* **ApplicationDbContext:** Используется для прямого взаимодействия с таблицей комментариев в методе Index.
* **IUserRepository:** Используется для абстрагированного взаимодействия с данными пользователей. Методы репозитория вызываются для получения списка всех пользователей (GetAllUsers) и конкретного пользователя по идентификатору (GetUserById).

### 4.2.2 Описание контроллера OrderController

Контроллер OrderController предназначен для управления заказами в веб-приложении для перевозки грузов. Он взаимодействует с базой данных через ApplicationDbContext, а также с репозиториями IOrderRepository и IUserRepository.

**Конструктор OrderController(ApplicationDbContext context, IOrderRepository orderRepository, IHttpContextAccessor httpContextAccessor, IUserRepository userRepository):**

* **Назначение:** Инициализирует контроллер с контекстом базы данных, репозиториями заказов и пользователей, а также аксессором HTTP-контекста.
* **Параметры:**
  + context: Объект типа ApplicationDbContext, предоставляющий доступ к базе данных.
  + orderRepository: Интерфейс IOrderRepository, предоставляющий методы для работы с данными заказов.
  + httpContextAccessor: Интерфейс IHttpContextAccessor для доступа к текущему HTTP-контексту.
  + userRepository: Интерфейс IUserRepository, предоставляющий методы для работы с данными пользователей.

#### Методы

**1. public async Task<IActionResult> Index()**

* **Назначение:** Получает список всех заказов и отображает их на странице.
* **Что вызывает:**
  + Использует метод GetAll из IOrderRepository для получения всех заказов.
* **Что передает:**
  + Возвращает представление со списком заказов (View(orders)).

**2. public async Task<IActionResult> Detail(int id)**

* **Назначение:** Получает подробную информацию о заказе по его идентификатору и отображает её на странице.
* **Что вызывает:**
  + Использует метод GetByIdAsync из IOrderRepository для получения данных конкретного заказа по его идентификатору.
* **Что передает:**
  + Возвращает представление с подробной информацией о заказе (View(order)).

**3. [HttpGet] public IActionResult Create()**

* **Назначение:** Отображает форму для создания нового заказа.
* **Что вызывает:**
  + Получает идентификатор текущего пользователя из HTTP-контекста.
* **Что передает:**
  + Создает объект OrderViewModel с заполненным идентификатором заказчика и возвращает представление с формой для создания заказа (View(createOrderViewModel)).

**4. [HttpPost] public async Task<IActionResult> Create(Order orderVM)**

* **Назначение:** Обрабатывает отправку формы для создания нового заказа.
* **Что вызывает:**
  + Создает новый объект Order на основе данных из orderVM.
  + Вызывает метод Add из IOrderRepository для добавления нового заказа в базу данных.
* **Что передает:**
  + Перенаправляет на страницу со списком заказов (RedirectToAction("Index")).
  + В случае ошибки добавляет сообщение об ошибке в модель состояния и возвращает представление с формой.

**5. [HttpGet] public async Task<IActionResult> Edit(int id)**

* **Назначение:** Отображает форму для редактирования существующего заказа.
* **Что вызывает:**
  + Использует метод GetByIdAsync из IOrderRepository для получения данных конкретного заказа по его идентификатору.
* **Что передает:**
  + Создает объект OrderViewModel на основе данных заказа и возвращает представление с формой для редактирования (View(orderEdit)).

**6. [HttpPost] public async Task<IActionResult> Edit(int id, OrderViewModel orderVM)**

* **Назначение:** Обрабатывает отправку формы для редактирования существующего заказа.
* **Что вызывает:**
  + Использует метод GetByIdAsyncNoTracking из IOrderRepository для получения данных конкретного заказа по его идентификатору.
  + Создает новый объект Order на основе данных из orderVM и обновляет его в базе данных через метод Update из IOrderRepository.
* **Что передает:**
  + Перенаправляет на страницу со списком заказов (RedirectToAction("Index")).
  + В случае ошибки возвращает представление с формой.

**7. [HttpGet] public async Task<IActionResult> Delete(int id)**

* **Назначение:** Отображает подтверждение для удаления заказа.
* **Что вызывает:**
  + Использует метод GetByIdAsync из IOrderRepository для получения данных конкретного заказа по его идентификатору.
* **Что передает:**
  + Возвращает представление с информацией о заказе для подтверждения удаления (View(orderDetails)).

**8. [HttpPost, ActionName("Delete")] public async Task<IActionResult> DeleteOrder(int id)**

* **Назначение:** Обрабатывает удаление заказа.
* **Что вызывает:**
  + Использует метод GetByIdAsync из IOrderRepository для получения данных конкретного заказа по его идентификатору.
  + Вызывает метод Delete из IOrderRepository для удаления заказа, адреса доставки и пакета из базы данных.
* **Что передает:**
  + Перенаправляет на страницу со списком заказов (RedirectToAction("Index")).

**9. [HttpGet] public async Task<IActionResult> SubmitApplication(int id)**

* **Назначение:** Отображает форму для подачи заявки на выполнение заказа.
* **Что вызывает:**
  + Проверяет, существует ли уже заявка от текущего пользователя на данный заказ.
  + Использует метод GetByIdAsync из IOrderRepository для получения данных конкретного заказа по его идентификатору.
* **Что передает:**
  + Возвращает представление с информацией о заказе (View(order)).

**10. [HttpPost] public async Task<IActionResult> SubmitApplication(int id, int Price)**

* **Назначение:** Обрабатывает подачу заявки на выполнение заказа.
* **Что вызывает:**
  + Создает новый объект Auction на основе данных из формы.
  + Добавляет заявку в базу данных через контекст ApplicationDbContext.
* **Что передает:**
  + Перенаправляет на страницу со списком заказов (RedirectToAction("Index")).

**11. [HttpGet] public async Task<IActionResult> ViewApplications(int id)**

* **Назначение:** Отображает список всех заявок на выполнение конкретного заказа.
* **Что вызывает:**
  + Получает список всех заявок на заказ из контекста базы данных.
  + Для каждой заявки получает данные исполнителя и его комментарии.
* **Что передает:**
  + Создает список объектов AuctionViewModel и возвращает представление с этим списком (View(applicationsInfo)).

**12. [HttpGet] public async Task<IActionResult> ChooseContractor(int id)**

* **Назначение:** Выбирает исполнителя для заказа и обновляет статус заказа.
* **Что вызывает:**
  + Использует метод GetByIdAsync из IOrderRepository для получения данных заказа.
  + Обновляет заказ с указанием исполнителя, статусом и ценой.
  + Удаляет все заявки на заказ.
* **Что передает:**
  + Перенаправляет на страницу панели управления (RedirectToAction("Index","Dashboard")).

**13. [HttpGet] public async Task<IActionResult> Complete(int id)**

* **Назначение:** Завершает выполнение заказа и обновляет его статус.
* **Что вызывает:**
  + Использует метод GetByIdAsync из IOrderRepository для получения данных заказа.
  + Обновляет статус заказа на "Finished".
* **Что передает:**
  + Перенаправляет на страницу создания комментария с идентификатором исполнителя (RedirectToAction("Create", "Comment", new { id = order.ContractorId })).

### 4.2.3 Описание контроллера DashboardController

Контроллер DashboardController предназначен для управления данными пользовательского профиля и заказами в системе перевозки грузов. Он взаимодействует с репозиториями и предоставляет пользователям интерфейс для просмотра и редактирования их информации, а также для управления заказами.

**Конструктор DashboardController(IDashboardRepository dashboardRepository, IHttpContextAccessor httpContextAccessor, IWebHostEnvironment webHostEnvironment, IUserRepository userRepository):**

* **Назначение:** Инициализирует контроллер с необходимыми сервисами и репозиториями.
* **Параметры:**
  + dashboardRepository: Интерфейс IDashboardRepository, предоставляющий методы для работы с данными пользовательского профиля и заказов.
  + httpContextAccessor: Интерфейс IHttpContextAccessor для доступа к текущему HTTP-контексту.
  + webHostEnvironment: Интерфейс IWebHostEnvironment для работы с окружением веб-хоста.
  + userRepository: Интерфейс IUserRepository, предоставляющий методы для работы с данными пользователей.

#### Методы

**1. public async void UserEdit(AppUser appUser, EditUserDashboardViewModel editVM)**

* **Назначение:** Обновляет информацию пользователя, включая изображение профиля.
* **Что вызывает:**
  + Проверяет и удаляет старое изображение, если оно существует.
  + Сохраняет новое изображение и обновляет информацию пользователя.
* **Что передает:**
  + Не возвращает результат (void).

**2. public async Task<IActionResult> Index()**

* **Назначение:** Получает все заказы пользователя и отображает их на странице.
* **Что вызывает:**
  + Использует метод GetAllUserOrders из IDashboardRepository для получения всех заказов пользователя.
* **Что передает:**
  + Создает объект DashboardViewModel с заказами пользователя и возвращает представление с этим объектом (View(userViewModel)).

**3. [HttpGet] public async Task<IActionResult> Profile()**

* **Назначение:** Получает информацию о текущем пользователе и отображает её на странице профиля.
* **Что вызывает:**
  + Использует метод GetUserById из IUserRepository для получения данных пользователя по его идентификатору.
* **Что передает:**
  + Создает объект UserDetailViewModel с данными пользователя и возвращает представление с этим объектом (View(userDetailViewModel)).

**4. [HttpGet] public async Task<IActionResult> EditUserProfile()**

* **Назначение:** Отображает форму для редактирования профиля пользователя.
* **Что вызывает:**
  + Получает идентификатор текущего пользователя из HTTP-контекста.
  + Использует метод GetUserById из IDashboardRepository для получения данных пользователя по его идентификатору.
* **Что передает:**
  + Создает объект EditUserDashboardViewModel с данными пользователя и возвращает представление с формой для редактирования (View(editUserViewModel)).

**5. [HttpPost] public async Task<IActionResult> EditUserProfile(EditUserDashboardViewModel editVM)**

* **Назначение:** Обрабатывает отправку формы для редактирования профиля пользователя.
* **Что вызывает:**
  + Проверяет валидность модели.
  + Использует метод GetUserByIdNoTracking из IDashboardRepository для получения данных пользователя без отслеживания изменений.
  + Вызывает метод UserEdit для обновления информации пользователя.
  + Обновляет данные пользователя в базе данных через метод Update из IDashboardRepository.
* **Что передает:**
  + В случае успешного редактирования перенаправляет на страницу с заказами (RedirectToAction("Index")).
  + В случае ошибки возвращает представление с формой и сообщением об ошибке.

**6. public async Task<IActionResult> AcceptedOrders()**

* **Назначение:** Получает все принятые заказы и заявки пользователя и отображает их на странице.
* **Что вызывает:**
  + Использует методы GetAllContractorOrders и GetAllContractorAuctions из IDashboardRepository для получения данных о принятых заказах и заявках.
* **Что передает:**
  + Создает объект DashboardViewModel с заказами и заявками пользователя и возвращает представление с этим объектом (View(userViewModel)).

### 4.2.4 Описание контроллера CommentController

Контроллер CommentController предназначен для управления комментариями пользователей в системе перевозки грузов. Он взаимодействует с базой данных для создания, получения и отображения комментариев.

**Конструктор CommentController(ApplicationDbContext context):**

* **Назначение:** Инициализирует контроллер с контекстом базы данных.
* **Параметры:**
  + context: Экземпляр ApplicationDbContext, представляющий собой контекст базы данных, используемый для взаимодействия с таблицами базы данных.

#### Методы

**1. public IActionResult Index(string id)**

* **Назначение:** Получает и отображает список комментариев для пользователя с указанным идентификатором.
* **Что вызывает:**
  + Извлекает из базы данных все комментарии, где AppUserCommentId соответствует заданному id.
* **Что передает:**
  + Возвращает представление с списком комментариев для указанного пользователя (View(\_context.Comments.Where(a => a.AppUserCommentId == id).ToList())).

**2. public IActionResult Create(string id)**

* **Назначение:** Отображает форму для создания нового комментария для пользователя с указанным идентификатором.
* **Что вызывает:**
  + Устанавливает ViewBag.ContractorId для передачи идентификатора пользователя в представление.
  + Извлекает из базы данных все комментарии, где AppUserCommentId соответствует заданному id.
* **Что передает:**
  + Возвращает представление с формой для создания нового комментария и текущими комментариями пользователя (View(\_context.Comments.Where(a => a.AppUserCommentId == id).ToList())).

**3. [HttpPost] public IActionResult AddComment(string contractorId, int rating, string comment)**

* **Назначение:** Обрабатывает отправку формы для добавления нового комментария.
* **Что вызывает:**
  + Создает новый объект Comment с заданными параметрами (contractorId, rating, comment) и текущей датой публикации.
  + Добавляет новый комментарий в базу данных.
  + Сохраняет изменения в базе данных.
* **Что передает:**
  + После успешного добавления комментария перенаправляет на страницу со списком пользователей (RedirectToAction("Index", "User")).

### 4.2.5 Описание контроллера AccountController

Контроллер AccountController отвечает за управление учетными записями пользователей, включая вход в систему, регистрацию и выход.

**Конструктор AccountController(UserManager<AppUser> userManager, SignInManager<AppUser> signInManager, ApplicationDbContext context, IWebHostEnvironment webHostEnvironment):**

* **Назначение:** Инициализирует контроллер с необходимыми сервисами и контекстом базы данных.
* **Параметры:**
  + userManager: Менеджер пользователей для выполнения операций с пользователями.
  + signInManager: Менеджер входа для выполнения операций входа и выхода.
  + context: Контекст базы данных для взаимодействия с базой данных.
  + webHostEnvironment: Объект для работы с файловой системой в веб-хостинговой среде.

#### Методы

**1. [HttpGet] public IActionResult Login()**

* **Назначение:** Отображает форму входа в систему.
* **Что вызывает:**
  + Создает новый объект LoginViewModel.
* **Что передает:**
  + Возвращает представление с формой входа (View(repsonse)).

**2. [HttpPost] public async Task<IActionResult> Login(LoginViewModel loginViewModel)**

* **Назначение:** Обрабатывает данные формы входа.
* **Что вызывает:**
  + Проверяет валидность модели.
  + Ищет пользователя по электронной почте.
  + Проверяет правильность пароля.
  + Входит в систему при успешной проверке.
* **Что передает:**
  + При успешном входе перенаправляет на главную страницу заказов (RedirectToAction("Index", "Order")).
  + При ошибках возвращает сообщение об ошибке и представление с формой входа.

**3. [HttpGet] public IActionResult Register()**

* **Назначение:** Отображает форму регистрации.
* **Что вызывает:**
  + Создает новый объект RegisterViewModel.
* **Что передает:**
  + Возвращает представление с формой регистрации (View(repsonse)).

**4. [HttpPost] public async Task<IActionResult> Register(RegisterViewModel registerViewModel)**

* **Назначение:** Обрабатывает данные формы регистрации.
* **Что вызывает:**
  + Проверяет валидность модели.
  + Ищет пользователя по электронной почте.
  + Загружает и сохраняет изображение пользователя.
  + Создает нового пользователя.
  + Добавляет пользователя в роль "User".
* **Что передает:**
  + При успешной регистрации перенаправляет на страницу входа (RedirectToAction("Login", "Account")).
  + При ошибках возвращает сообщение об ошибке и представление с формой регистрации.

**5. [HttpPost] public async Task<IActionResult> Logout()**

* **Назначение:** Выполняет выход пользователя из системы.
* **Что вызывает:**
  + Выполняет выход пользователя.
* **Что передает:**
  + Перенаправляет на страницу входа (RedirectToAction("Login", "Account")).

4.3 Репозитории

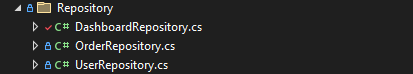


Рисунок 4.3 – Список репозиториев

### 4.3.1 Описание репозитория UserRepository

Репозиторий UserRepository отвечает за выполнение операций с данными, связанными с пользователями, в базе данных. Он реализует интерфейс IUserRepository.

**Конструктор UserRepository(ApplicationDbContext context):**

* **Назначение:** Инициализирует репозиторий с контекстом базы данных.
* **Параметры:**
  + context: Контекст базы данных для взаимодействия с базой данных.

#### Методы

**1. public async Task<IEnumerable<AppUser>> GetAllUsers()**

* **Назначение:** Получает всех пользователей из базы данных.
* **Что вызывает:**
  + Выполняет асинхронный запрос для получения всех пользователей из таблицы Users.
* **Что передает:**
  + Возвращает коллекцию всех пользователей (IEnumerable<AppUser>).

**2. public async Task<AppUser> GetUserById(string id)**

* **Назначение:** Получает пользователя по его идентификатору.
* **Что вызывает:**
  + Выполняет асинхронный запрос для получения пользователя с указанным идентификатором из таблицы Users.
  + Включает связанные данные из таблицы Address.
* **Что передает:**
  + Возвращает пользователя (AppUser), соответствующего указанному идентификатору.

**3. public bool Save()**

* **Назначение:** Сохраняет изменения в базе данных.
* **Что вызывает:**
  + Выполняет метод SaveChanges контекста базы данных.
* **Что передает:**
  + Возвращает true, если изменения успешно сохранены (количество измененных записей больше 0), иначе false.

**4. public bool Update(AppUser user)**

* **Назначение:** Обновляет информацию о пользователе в базе данных.
* **Что вызывает:**
  + Обновляет данные пользователя в контексте базы данных.
  + Вызывает метод Save для сохранения изменений.
* **Что передает:**
  + Возвращает true, если обновление прошло успешно, иначе false.

### 4.3.2 Описание репозитория OrderRepository

Репозиторий OrderRepository отвечает за выполнение операций с данными, связанными с заказами, в базе данных. Он реализует интерфейс IOrderRepository.

**Конструктор OrderRepository(ApplicationDbContext context):**

* **Назначение:** Инициализирует репозиторий с контекстом базы данных.
* **Параметры:**
  + context: Контекст базы данных для взаимодействия с базой данных.

#### Методы

**1. public bool Add(Order order)**

* **Назначение:** Добавляет новый заказ в базу данных.
* **Что вызывает:**
  + Метод \_context.Add(order) для добавления заказа в контекст.
  + Метод Save для сохранения изменений в базе данных.
* **Что передает:**
  + Возвращает true, если заказ успешно добавлен и изменения сохранены, иначе false.

**2. public bool Delete(Order order, DeliveryAddress deliveryAddress, Package package)**

* **Назначение:** Удаляет заказ, адрес доставки и пакет из базы данных.
* **Что вызывает:**
  + Метод \_context.Remove(order) для удаления заказа из контекста.
  + Метод \_context.Remove(deliveryAddress) для удаления адреса доставки из контекста.
  + Метод \_context.Remove(package) для удаления пакета из контекста.
  + Метод Save для сохранения изменений в базе данных.
* **Что передает:**
  + Возвращает true, если все объекты успешно удалены и изменения сохранены, иначе false.

**3. public async Task<IEnumerable<Order>> GetAll()**

* **Назначение:** Получает все заказы из базы данных.
* **Что вызывает:**
  + Асинхронный метод для получения всех заказов из таблицы Orders, включая связанные данные о пакетах, адресах доставки и адресах клиентов.
* **Что передает:**
  + Возвращает коллекцию всех заказов (IEnumerable<Order>).

**4. public async Task<Order> GetByIdAsync(int id)**

* **Назначение:** Получает заказ по его идентификатору.
* **Что вызывает:**
  + Асинхронный метод для получения заказа с указанным идентификатором из таблицы Orders, включая связанные данные о клиенте, адресе доставки и пакете.
* **Что передает:**
  + Возвращает заказ (Order), соответствующий указанному идентификатору.

**5. public async Task<Order> GetByIdAsyncNoTracking(int id)**

* **Назначение:** Получает заказ по его идентификатору без отслеживания изменений.
* **Что вызывает:**
  + Асинхронный метод для получения заказа с указанным идентификатором из таблицы Orders, включая связанные данные о клиенте, адресе доставки и пакете, с отключенным отслеживанием изменений.
* **Что передает:**
  + Возвращает заказ (Order), соответствующий указанному идентификатору, без отслеживания изменений.

**6. public async Task<IEnumerable<Order>> GetOrderByCity(string city)**

* **Назначение:** Получает заказы по названию города.
* **Что вызывает:**
  + Асинхронный метод для получения всех заказов, где город клиента содержит указанное название города.
* **Что передает:**
  + Возвращает коллекцию заказов (IEnumerable<Order>) для указанного города.

**7. public bool Save()**

* **Назначение:** Сохраняет изменения в базе данных.
* **Что вызывает:**
  + Метод \_context.SaveChanges() для сохранения изменений в контексте базы данных.
* **Что передает:**
  + Возвращает true, если изменения успешно сохранены (количество измененных записей больше 0), иначе false.

**8. public bool Update(Order order)**

* **Назначение:** Обновляет информацию о заказе в базе данных.
* **Что вызывает:**
  + Метод \_context.Update(order) для обновления данных заказа в контексте базы данных.
  + Метод Save для сохранения изменений.
* **Что передает:**
  + Возвращает true, если обновление прошло успешно и изменения сохранены, иначе false.

### 4.3.3 Описание репозитория DashboardRepository

Репозиторий DashboardRepository отвечает за выполнение операций с данными, связанными с панелью управления пользователя, в базе данных. Он реализует интерфейс IDashboardRepository.

**Конструктор DashboardRepository(ApplicationDbContext context, IHttpContextAccessor httpContextAccessor):**

* **Назначение:** Инициализирует репозиторий с контекстом базы данных и доступом к HTTP-контексту.
* **Параметры:**
  + context: Контекст базы данных для взаимодействия с базой данных.
  + httpContextAccessor: Объект для доступа к HTTP-контексту.

#### Методы

**1. public async Task<List<Order>> GetAllUserOrders()**

* **Назначение:** Получает все заказы текущего пользователя.
* **Что вызывает:**
  + Получает идентификатор текущего пользователя.
  + Асинхронный метод для получения всех заказов из таблицы Orders, где клиентом является текущий пользователь, включая связанные данные о пакетах.
* **Что передает:**
  + Возвращает список заказов (List<Order>) текущего пользователя.

**2. public async Task<List<Order>> GetAllContractorOrders()**

* **Назначение:** Получает все заказы текущего подрядчика.
* **Что вызывает:**
  + Получает идентификатор текущего пользователя.
  + Асинхронный метод для получения всех заказов из таблицы Orders, где подрядчиком является текущий пользователь, включая связанные данные о пакетах.
* **Что передает:**
  + Возвращает список заказов (List<Order>) текущего подрядчика.

**3. public async Task<List<Order>> GetAllContractorAuctions()**

* **Назначение:** Получает все аукционы текущего подрядчика.
* **Что вызывает:**
  + Получает идентификатор текущего пользователя.
  + Асинхронный метод для получения всех аукционов из таблицы Auctions, где подрядчиком является текущий пользователь.
  + Для каждого аукциона получает связанный заказ из таблицы Orders, включая данные о пакетах.
* **Что передает:**
  + Возвращает список заказов (List<Order>) для текущего подрядчика на аукционах.

**4. public async Task<AppUser> GetUserById(string id)**

* **Назначение:** Получает пользователя по его идентификатору.
* **Что вызывает:**
  + Асинхронный метод для получения пользователя с указанным идентификатором из таблицы Users, включая связанные данные об адресах.
* **Что передает:**
  + Возвращает пользователя (AppUser), соответствующего указанному идентификатору.

**5. public async Task<AppUser> GetUserByIdNoTracking(string id)**

* **Назначение:** Получает пользователя по его идентификатору без отслеживания изменений.
* **Что вызывает:**
  + Асинхронный метод для получения пользователя с указанным идентификатором из таблицы Users, с отключенным отслеживанием изменений.
* **Что передает:**
  + Возвращает пользователя (AppUser), соответствующего указанному идентификатору, без отслеживания изменений.

**6. public bool Update(AppUser appUser)**

* **Назначение:** Обновляет информацию о пользователе в базе данных.
* **Что вызывает:**
  + Метод \_context.Users.Update(appUser) для обновления данных пользователя в контексте базы данных.
  + Метод Save для сохранения изменений.
* **Что передает:**
  + Возвращает true, если обновление прошло успешно и изменения сохранены, иначе false.

**7. public bool Save()**

* **Назначение:** Сохраняет изменения в базе данных.
* **Что вызывает:**
  + Метод \_context.SaveChanges() для сохранения изменений в контексте базы данных.
* **Что передает:**
  + Возвращает true, если изменения успешно сохранены (количество измененных записей больше 0), иначе false.

# 5 Руководство пользователя

При первом заходе на сайт пользователя будет ждать главная страница, на которой будет краткая информация о сайте. На рисунке 5.1 можно увидеть внешний вид данной страницы.

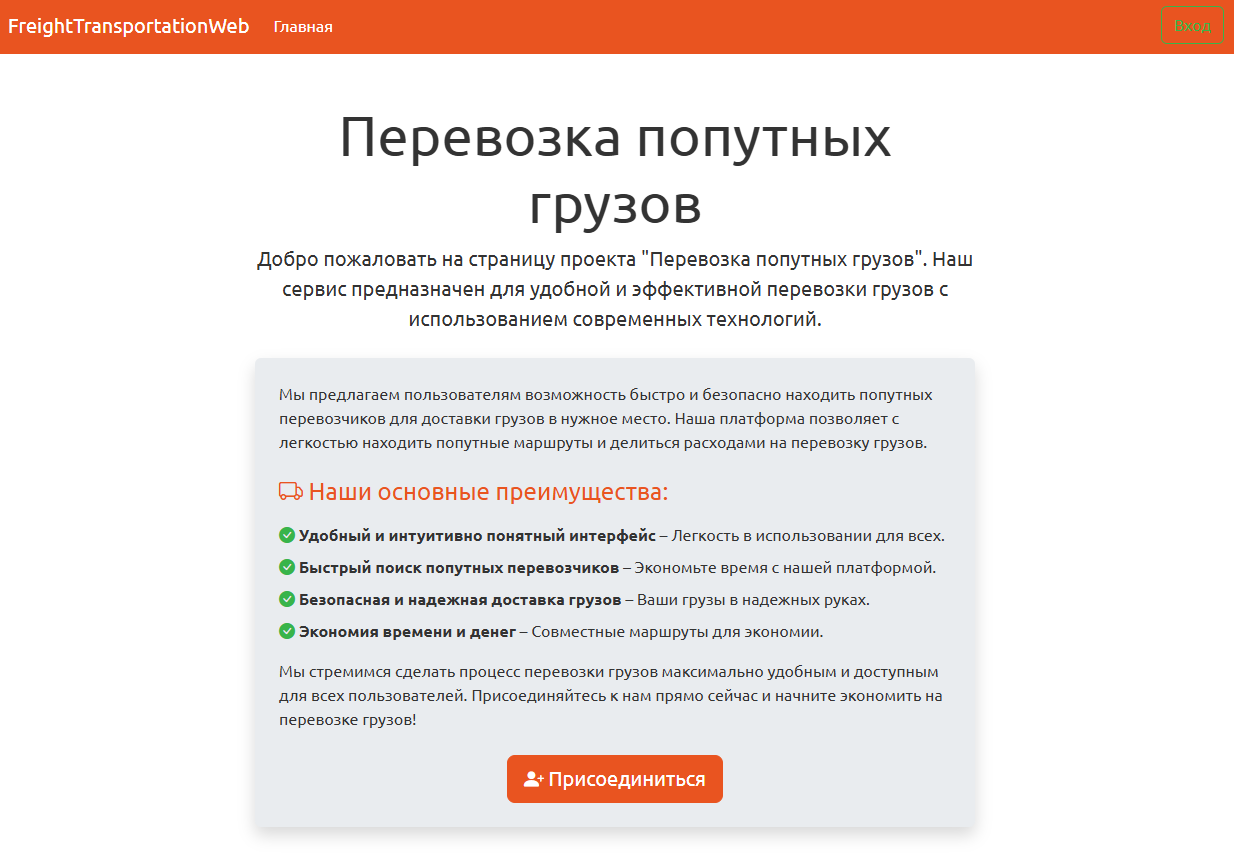


Рисунок 5.1 – Главная страница

После ознакомления с этой страницей, пользователю будет необходимо войти в свою учетную запись. На рисунке 5.2 можно ознакомиться с внешним видом страницы входа в аккаунт.

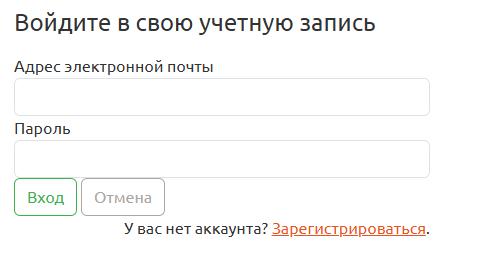


Рисунок 5.2 – Страница входа в аккаунт

Если это первое посещения пользователем сайта, он будет должен зарегистрироваться на сайте. Регистрация показана на рисунке 5.3

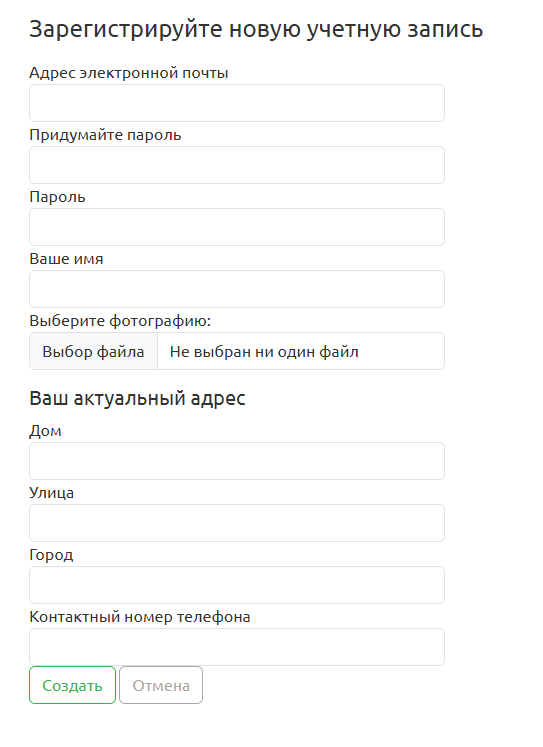


Рисунок 5.3 – Страница регистрации

После регистрации пользователь может принять на себя одну из двух ролей: «Заказчик» или «Исполнитель».

В роле «Заказчика» пользователь может создать заказ на доставку своего груза, предварительно заполнив о нем некоторые данные, как показано на рисунке 5.4

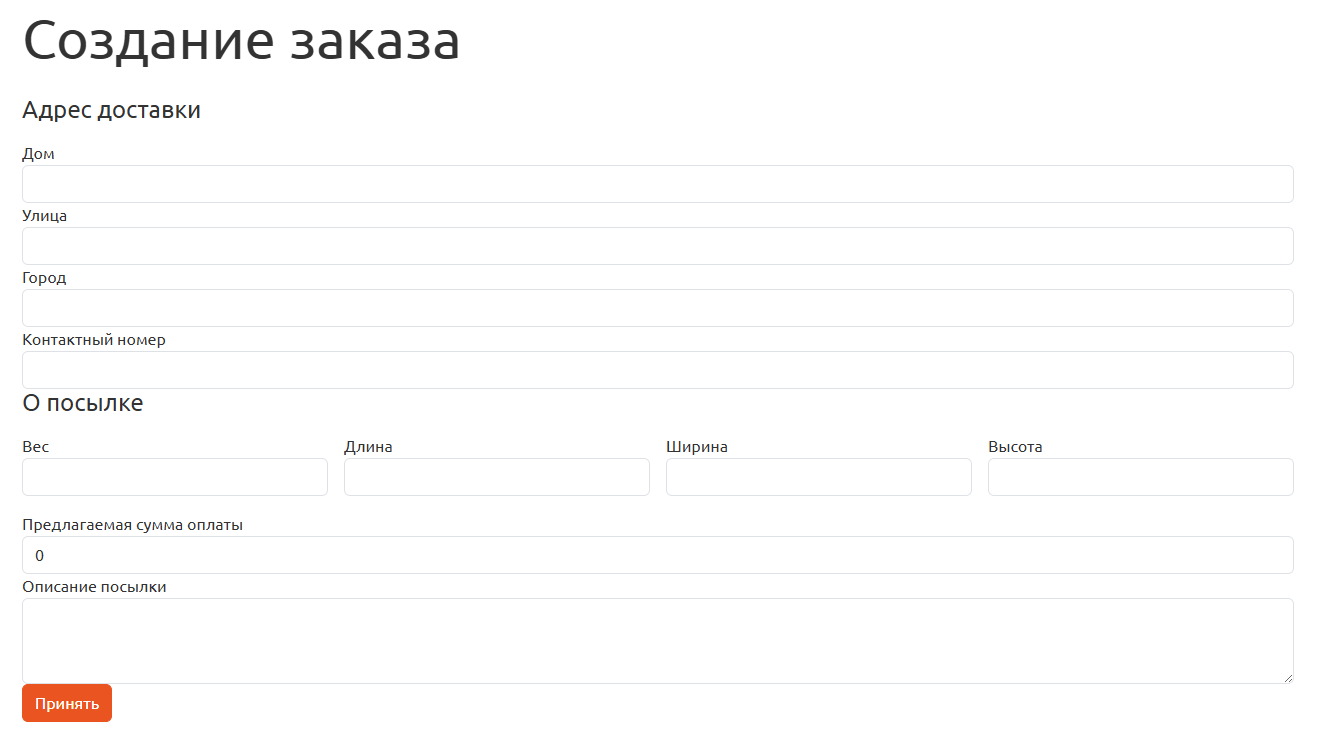


Рисунок 5.4 – Страница создания заказа

До момента осуществления доставки, пользователь может в любой момент редактировать данные этого заказа. После этого он будет ожидать пока потенциальные доставщики будут оставлять свои заявки на доставку этого груза, после чего он сможет ознакомиться с каждым из претендентов и просмотреть запрашиваемую ими сумму за выполнение заказа. Пример данного этапа доставки товара показан на рисунке 5.5

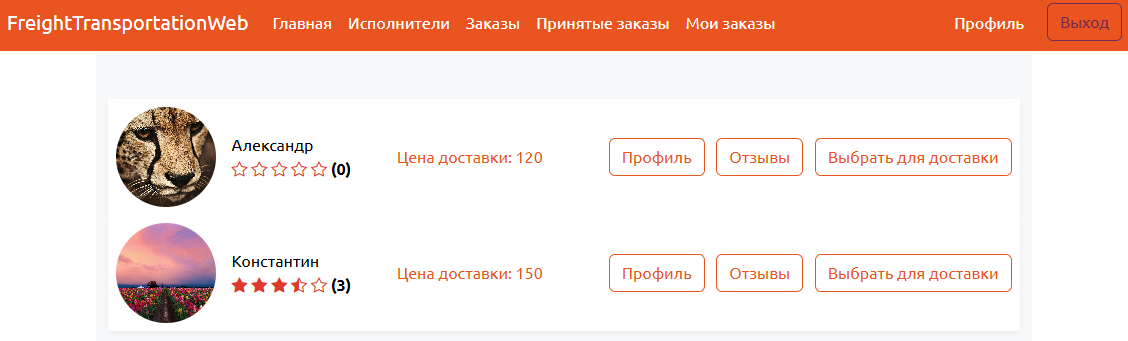


Рисунок 5.5 – Страница выбора исполнителя заказа

После выбора исполнителя заказ переходит в раздел «Заказы в процессе». И после того, как заказ был доставлен, заказчик завершает заказ, после чего ему будет предложено оставить отзыв и оценку исполнителю, а доставленный заказ перейдет в раздел «Завершенные», как продемонстрированно на рисунке 5.6.

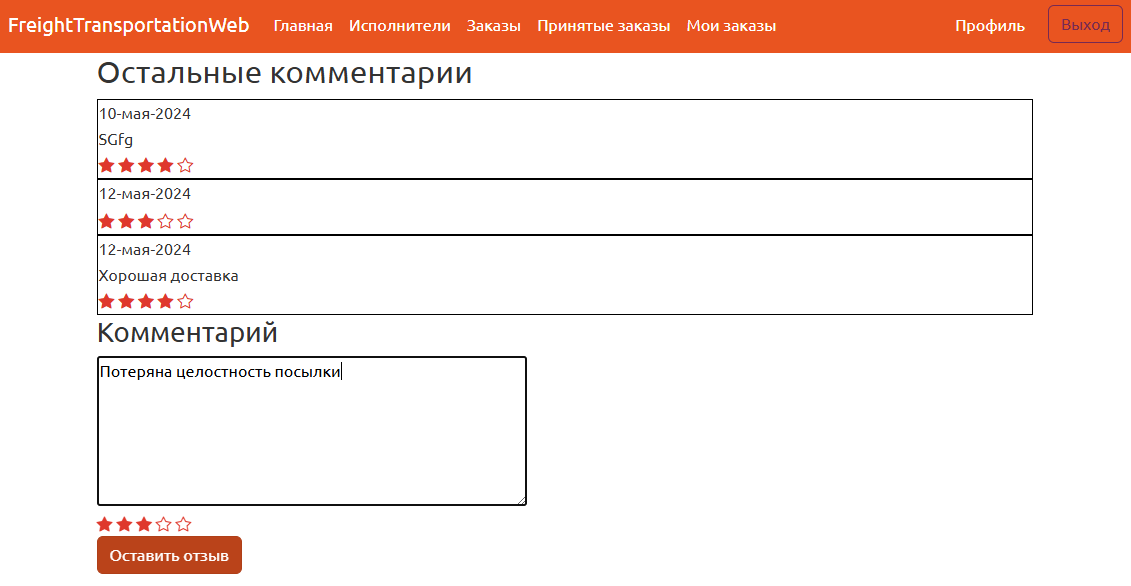


Рисунок 5.6 – Страница для оценки доставщика

На начальном этапе работы веб-приложения подразумевается, что оплата доставки будет производиться напрямую, между заказчиком и исполнителем.

В роле «Исполнителя» пользователь может зайти на вкладку со всеми имеющимися на данный момент заказами, для выбора подходящего для него заказа. Пример можно увидеть на рисунке 5.7

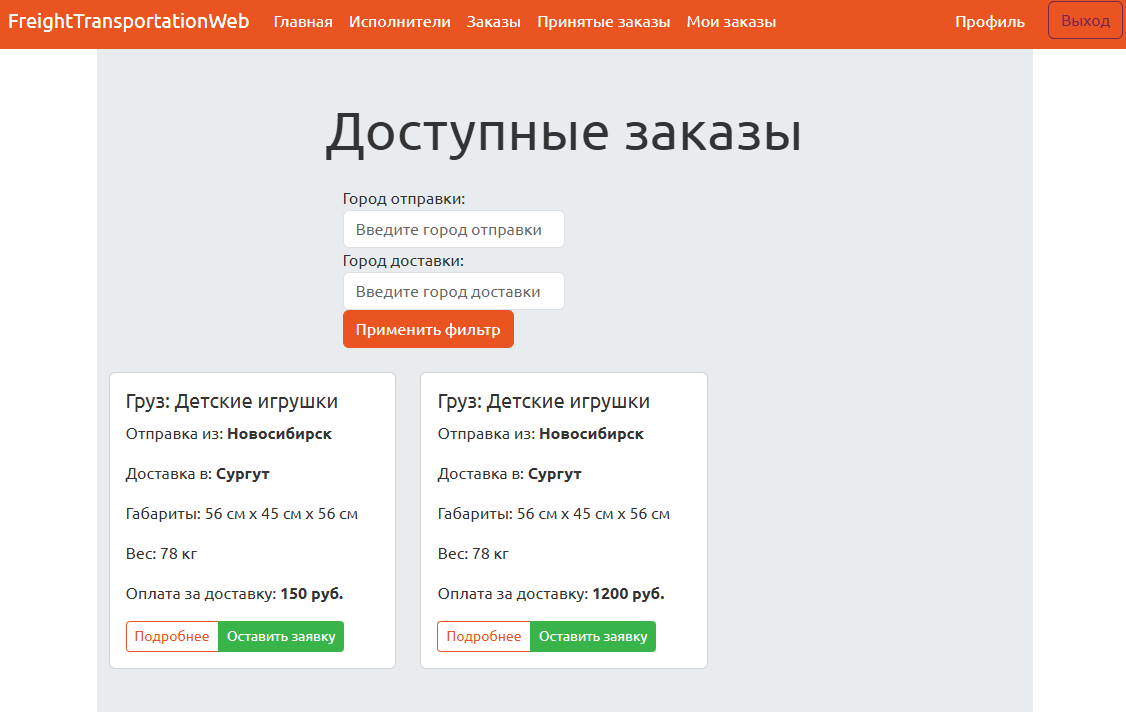
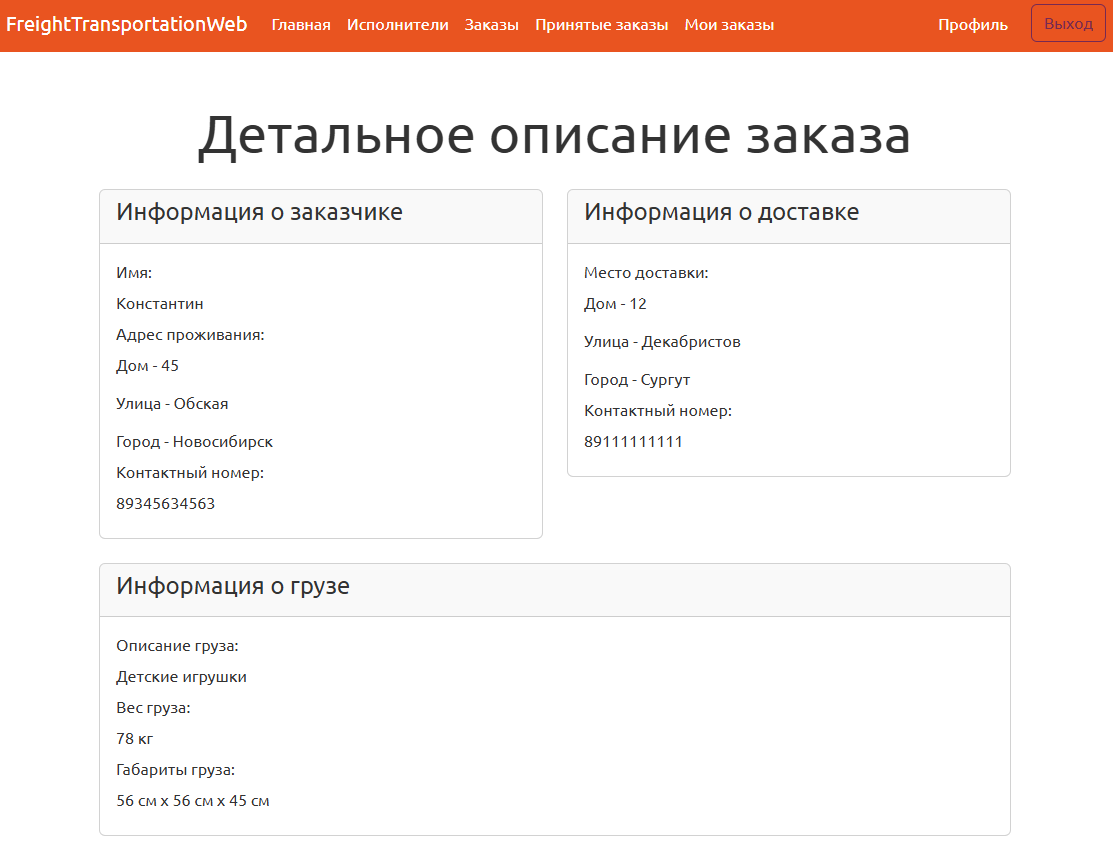


Рисунок 5.7 – Страница выбора заказа

При помощи интуитивно понятно фильтра он может отфильтровать заказы, по городу отправки и конечной точки доставки. Также он может просмотреть подробную информацию о посылке, пример детального описания заказа представлен на рисунке 5.8



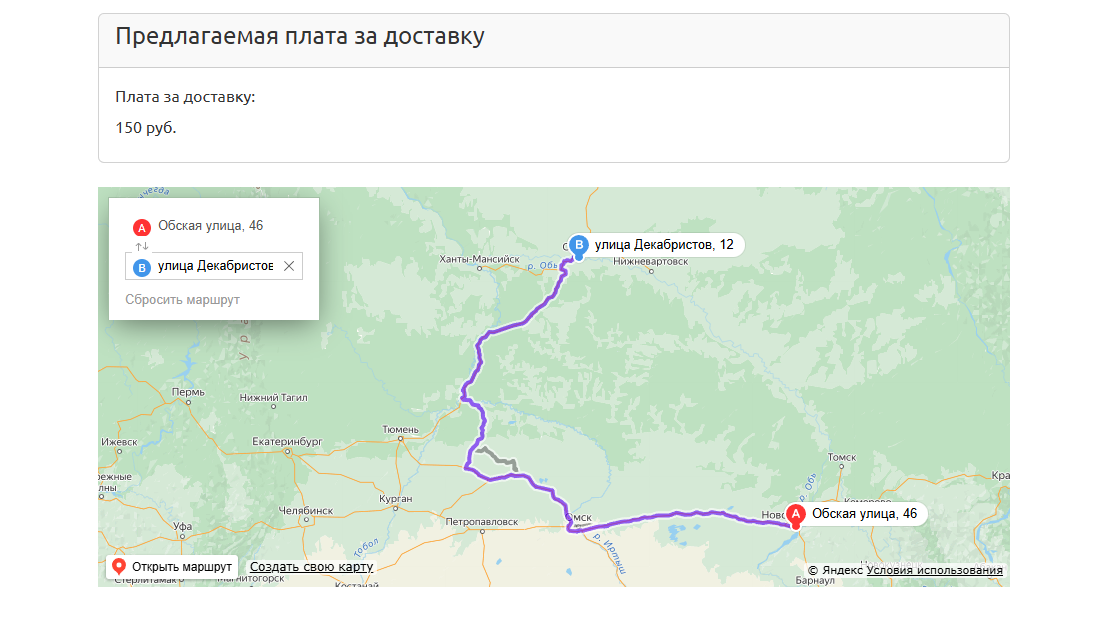


Рисунок 5.8 – Страница подробной информации заказа

После чего он может предложить свою цену, за которую он будет готов принять этот заказ, как показано на рисунке 5.9

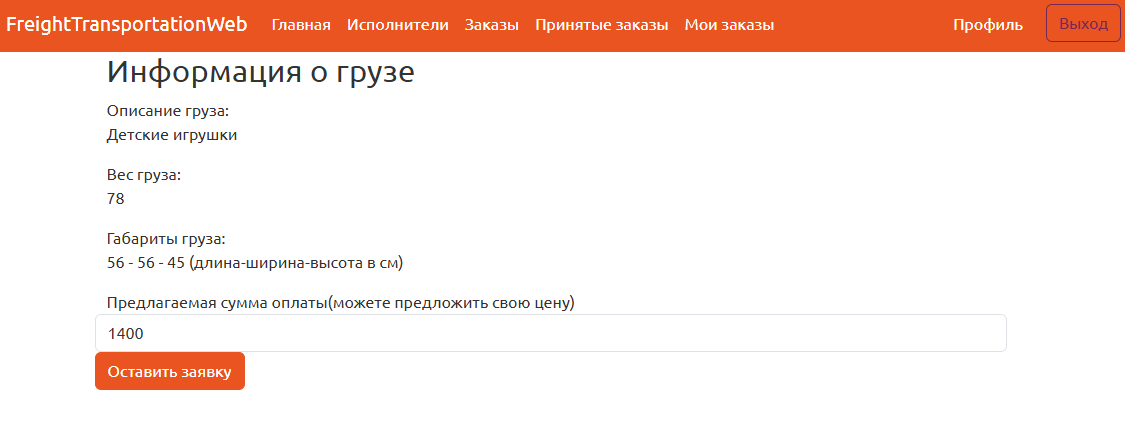


Рисунок 5.9 – Страница оставления заявки на исполнение заказа

Этот заказ будет отображаться в соответствующей вкладке. Если заказчик примет его предложение, этот заказ перейдет в стадию доставки и будет находится в разделе «Заказы в процессе». После того как заказ будет доставлен, исполнитель получит оценку с комментарием от заказчика, после чего заказ перейдет в раздел «Завершенные». Пример можно увидеть на рисунке 5.10

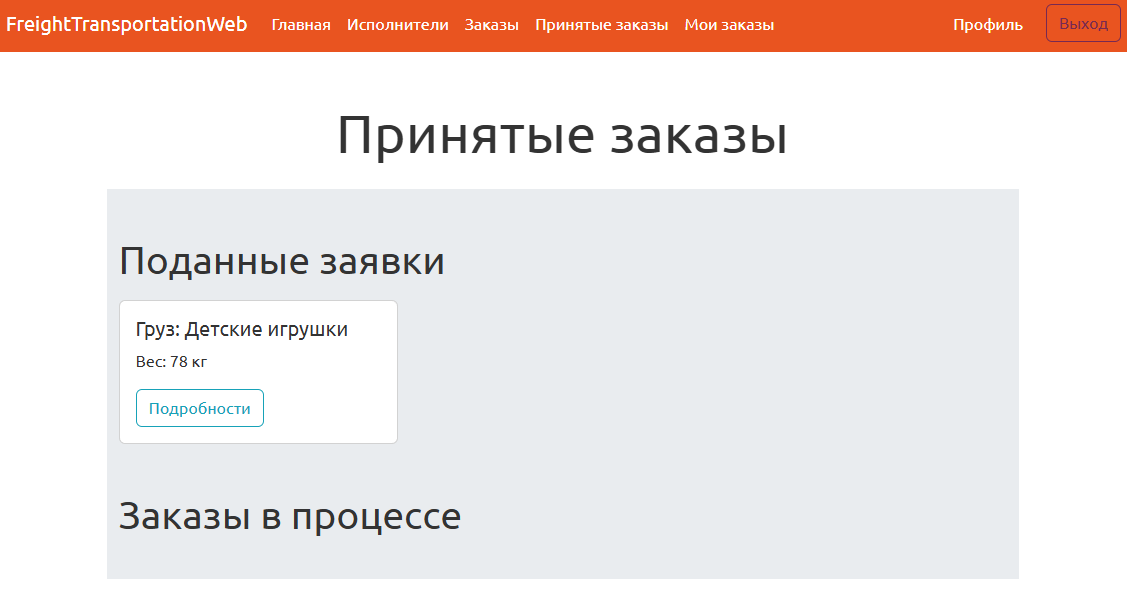


Рисунок 5.10 – Страница работы с заказами

Пользователь в любой момент может зайти и проверить свой профиль, пример внешнего вида профиля пользователя можно увидеть на рисунке 5.11

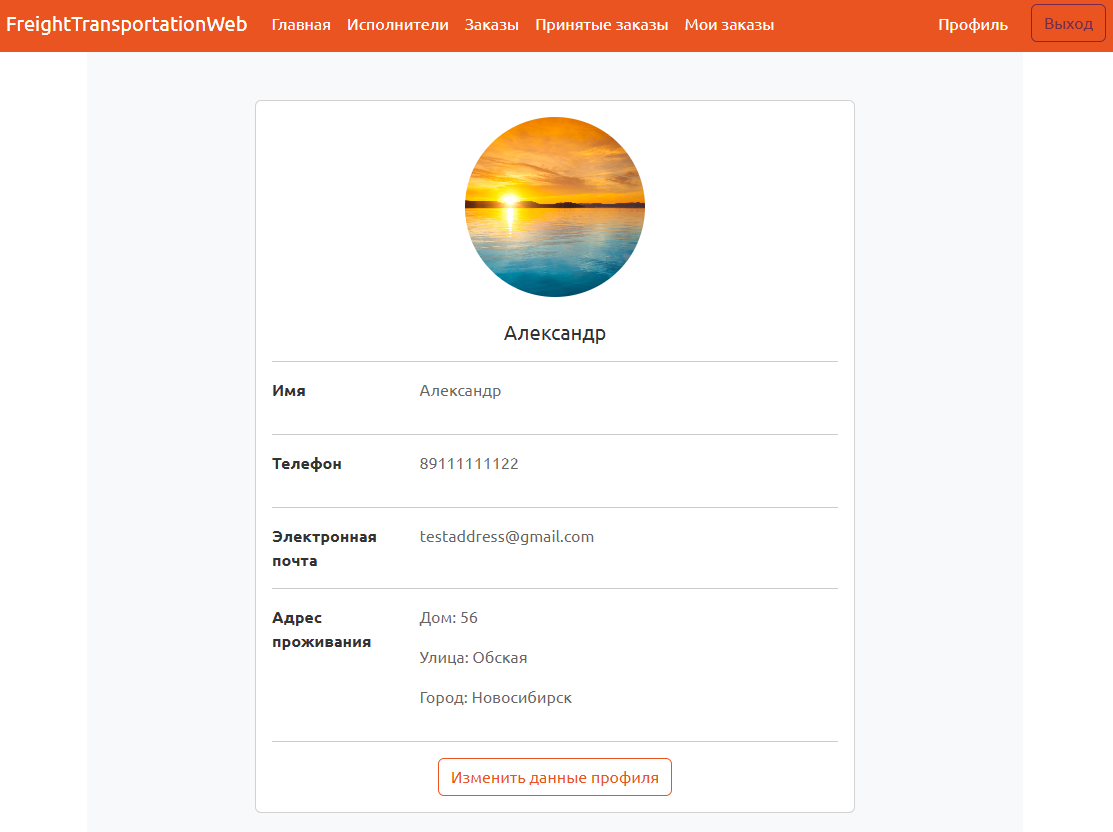


Рисунок 5.11 – Страница профиля

Также для всех пользователей есть возможность редактирования своих данных, как продемонстрированно на рисунке 5.12

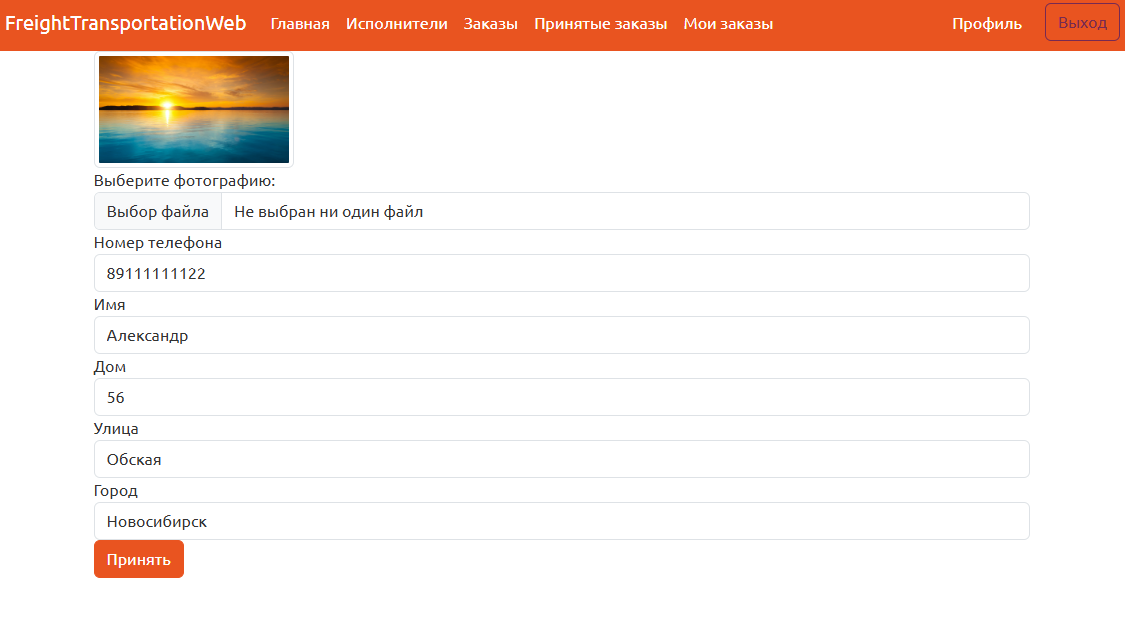


Рисунок 5.12 – Страница редактирования профиля

# Заключение

В процессе разработки веб-приложения для доставки попутных грузов были применены передовые технологии и инструменты с целью обеспечить высокую эффективность, удобство использования и безопасность для пользователей. Использование языка C# с ASP.NET Core в качестве основы приложения и фреймворка обеспечивает быстрое создание и возможность масштабирования, в то время как SQL Server, как выбранная база данных, гарантирует надежное хранение и управление данными.

Решение о выборе интегрированной среды разработки Visual Studio было обусловлено ее широкими функциональными возможностями и удобством работы с проектом. Помимо этого, дополнительные инструменты и библиотеки, такие как Entity Framework Core и ASP.NET Core Identity, значительно упростили процесс разработки и обеспечили высокую функциональность приложения.

Принимая во внимание требования к проекту и потребности пользователей, использование указанных технологий и инструментов является оптимальным решением. Этот выбранный набор технологий не только способствует эффективному развитию приложения, но и обеспечивает его будущую поддержку и расширение.

# список использованных источников и литературы

1. Документация по языку C# [Электронный ресурс].

URL: <https://learn.microsoft.com/ru-ru/dotnet/csharp/>

1. Руководство по ASP.NET Core MVC [Электронный ресурс].

URL: [https://metanit.com/sharp/aspnetmvc/](https://metanit.com/sharp/aspnetmvc/%20а)

1. SQL Server Management Studio [Электронный ресурс].

URL: [https://learn.microsoft.com/](https://learn.microsoft.com/%20)

1. Руководство по настройке Bootstrap [Электронный ресурс].

URL: https://getbootstrap.com/docs/

1. Bootstrap шаблоны [Электронный ресурс].

URL:  [https://bootswatch.com/morph/](%20https://bootswatch.com/morph/)

# Приложение А

Листинг программы

1. Модели

**AppUser**

using FreightTransportationWeb.Data.Enum;

using Microsoft.AspNetCore.Identity;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace FreightTransportationWeb.Models

{

public class AppUser: IdentityUser

{

[ForeignKey ("AddressUser")]

public int AddressId { get; set; }

public AddressUser Address { get; set; }

public string? Image { get; set; }

public ICollection<Comment> Comments { get; set; }

}

**AddressUser**

using System.ComponentModel.DataAnnotations;

namespace FreightTransportationWeb.Models

{

public class AddressUser

{

[Key]

public int Id { get; set; }

public string House { get; set; }

public string Street { get; set; }

public string City { get; set; }

}

}

**Order**

using FreightTransportationWeb.Data.Enum;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace FreightTransportationWeb.Models

{

public class Order

{

[Key]

public int Id { get; set; }

[ForeignKey("AppUser")]

public string? CustomerId { get; set; }

public AppUser? Customer { get; set; }

[ForeignKey("AppUser")]

public string? ContractorId { get; set; }

public AppUser? Contractor { get; set; }

[ForeignKey("DeliveryAddress")]

public int DeliveryAddressId { get; set; }

public DeliveryAddress DeliveryAddress { get; set; }

[ForeignKey("Package")]

public int PackageId { get; set; }

public Package Package { get; set; }

public int Price { get; set; }

public OrderStatus OrderStatus { get; set; }

}

}

**Package**

using System.ComponentModel.DataAnnotations;

namespace FreightTransportationWeb.Models

{

public class Package

{

[Key]

public int Id { get; set; }

public int Weight { get; set; }

public int Length { get; set; }

public int Width { get; set; }

public int Height { get; set; }

public string? Description { get; set; }

}

}

**Comment**

using System.ComponentModel.DataAnnotations;

using System.Data;

namespace FreightTransportationWeb.Models

{

public class Comment

{

public int Id { get; set; }

public string? Comments { get; set; }

public DateTime PublishedDate { get; set; }

public string AppUserCommentId { get; set; }

public int Rating { get; set; }

}

}

**Auction**

using System.ComponentModel.DataAnnotations;

namespace FreightTransportationWeb.Models

{

public class Auction

{

[Key]

public int Id { get; set; }

public int OrderId { get; set; }

public string ContractorId { get; set; }

public int Price { get; set; }

}}

**DeliveryAddress**

using System.ComponentModel.DataAnnotations;

namespace FreightTransportationWeb.Models

{

public class DeliveryAddress

{

[Key]

public int Id { get; set; }

public string House { get; set; }

public string Street { get; set; }

public string City { get; set; }

public string PhoneNumber { get; set; }

}

1. Контроллеры

**UserController**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Interfaces;

using FreightTransportationWeb.ViewModels;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace FreightTransportationWeb.Controllers

{

public class UserController : Controller

{

private readonly ApplicationDbContext \_context;

private readonly IUserRepository \_userRepository;

public UserController(ApplicationDbContext context,IUserRepository userRepository)

{

\_context = context;

\_userRepository = userRepository;

}

[HttpGet("users")]

public async Task<IActionResult> Index()

{

var users = await \_userRepository.GetAllUsers();

List<UserViewModel> result = new List<UserViewModel>();

foreach (var user in users)

{

var commentsUser = \_context.Comments.Where(a => a.AppUserCommentId == user.Id).ToList();

var countComments = commentsUser.Count();

double averageRating = 0;

if (commentsUser.Count != 0)

{

averageRating = commentsUser.Sum(r => r.Rating) / (double)commentsUser.Count;

}

var userViewModel = new UserViewModel()

{

Id = user.Id,

Image = user.Image,

UserName = user.UserName,

AddressUser = user.Address,

AverageRating = averageRating,

CountComments = countComments

};

result.Add(userViewModel);

}

return View(result);

}

public async Task<IActionResult> Detail(string id)

{

var user = await \_userRepository.GetUserById(id);

var userDetailViewModel = new UserDetailViewModel()

{

Id = user.Id,

UserName = user.UserName,

AddressUser = user.Address,

Image = user.Image

};

return View(userDetailViewModel);

}

}

}

**OrderController**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Data.Enum;

using FreightTransportationWeb.Interfaces;

using FreightTransportationWeb.Models;

using FreightTransportationWeb.ViewModels;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace FreightTransportationWeb.Controllers

{

public class OrderController : Controller

{

private readonly ApplicationDbContext \_context;

private readonly IOrderRepository \_orderRepository;

private readonly IHttpContextAccessor \_httpContextAccessor;

private readonly IUserRepository \_userRepository;

public OrderController(ApplicationDbContext context, IOrderRepository orderRepository, IHttpContextAccessor httpContextAccessor, IUserRepository userRepository)

{

\_context = context;

\_orderRepository = orderRepository;

\_httpContextAccessor = httpContextAccessor;

\_userRepository = userRepository;

}

public async Task<IActionResult> Index()

{

IEnumerable<Order> orders = await \_orderRepository.GetAll();

return View(orders);

}

public async Task<IActionResult> Detail(int id)

{

Order order = await \_orderRepository.GetByIdAsync(id);

return View(order);

}

[HttpGet]

public IActionResult Create()

{

var curUserId = \_httpContextAccessor.HttpContext.User.GetUserId();

var createOrderViewModel = new OrderViewModel { CustomerId = curUserId};

return View(createOrderViewModel);

}

[HttpPost]

public async Task<IActionResult> Create(Order orderVM)

{

try

{

var order = new Order

{

CustomerId = orderVM.CustomerId,

DeliveryAddress = new DeliveryAddress

{

House = orderVM.DeliveryAddress.House,

Street = orderVM.DeliveryAddress.Street,

City = orderVM.DeliveryAddress.City,

PhoneNumber = orderVM.DeliveryAddress.PhoneNumber

},

Package = new Package

{

Weight = orderVM.Package.Weight,

Length = orderVM.Package.Length,

Width = orderVM.Package.Width,

Height = orderVM.Package.Height,

Description = orderVM.Package.Description,

},

Price = orderVM.Price,

OrderStatus = OrderStatus.Created

};

\_orderRepository.Add(order);

return RedirectToAction("Index");

}

catch

{

ModelState.AddModelError("", "Error creating");

}

return View();

}

[HttpGet]

public async Task<IActionResult> Edit(int id)

{

var order = await \_orderRepository.GetByIdAsync(id);

if (order == null) return View("Error");

var orderEdit = new OrderViewModel

{

Customer = order.Customer,

CustomerId = order.CustomerId,

DeliveryAddressId = order.DeliveryAddressId,

DeliveryAddress = order.DeliveryAddress,

PackageId = order.PackageId,

Package = order.Package,

Price = order.Price,

OrderStatus = order.OrderStatus

};

return View(orderEdit);

}

[HttpPost]

public async Task<IActionResult> Edit(int id, OrderViewModel orderVM)

{

var orderEdit = await \_orderRepository.GetByIdAsyncNoTracking(id);

if(orderEdit != null)

{

var order = new Order

{

Id = id,

CustomerId = orderEdit.CustomerId,

Customer = orderEdit.Customer,

DeliveryAddressId = orderEdit.DeliveryAddressId,

DeliveryAddress = orderEdit.DeliveryAddress,

PackageId = orderEdit.PackageId,

Package = orderEdit.Package,

Price = orderVM.Price,

OrderStatus = orderEdit.OrderStatus

};

order.DeliveryAddress.PhoneNumber = orderVM.DeliveryAddress.PhoneNumber;

order.DeliveryAddress.House = orderVM.DeliveryAddress.House;

order.DeliveryAddress.Street = orderVM.DeliveryAddress.Street;

order.DeliveryAddress.City = orderVM.DeliveryAddress.City;

order.Package.Height = orderVM.Package.Height;

order.Package.Length = orderVM.Package.Length;

order.Package.Weight = orderVM.Package.Weight;

order.Package.Width = orderVM.Package.Width;

order.Package.Description = orderVM.Package.Description;

\_orderRepository.Update(order);

return RedirectToAction("Index");

}

else

{

return View(orderVM);

}

}

[HttpGet]

public async Task<IActionResult> Delete(int id)

{

var orderDetails = await \_orderRepository.GetByIdAsync(id);

if(orderDetails == null) return View("Error");

return View(orderDetails);

}

[HttpPost,ActionName("Delete")]

public async Task<IActionResult> DeleteOrder(int id)

{

var orderDetail = await \_orderRepository.GetByIdAsync(id);

if( orderDetail == null) return View("Error");

var deliveryAdress = orderDetail.DeliveryAddress;

var package = orderDetail.Package;

if (deliveryAdress == null || package == null) return View("Error");

\_orderRepository.Delete(orderDetail, deliveryAdress, package);

return RedirectToAction("Index");

}

[HttpGet]

public async Task<IActionResult> SubmitApplication(int id)

{

var curUserId = \_httpContextAccessor.HttpContext.User.GetUserId();

if (\_context.Auctions.FirstOrDefault(a => a.OrderId == id && a.ContractorId == \_httpContextAccessor.HttpContext.User.GetUserId()) != null)

{

TempData["ErrorMessage"] = "Ошибка: Вы уже оставили заявку на данный заказ.";

return RedirectToAction("Index"); // Перенаправление на страниц

}

Order order = await \_orderRepository.GetByIdAsync(id);

return View(order);

}

[HttpPost]

public async Task<IActionResult> SubmitApplication(int id, int Price)

{

Order order = await \_orderRepository.GetByIdAsync(id);

Auction auction = new Auction

{

OrderId = id,

Price = Price,

ContractorId = \_httpContextAccessor.HttpContext.User.GetUserId()

};

\_context.Add(auction);

\_context.SaveChanges();

return RedirectToAction("Index");

}

[HttpGet]

public async Task<IActionResult> ViewApplications(int id)

{

List <AuctionViewModel> applicationsInfo = new List<AuctionViewModel>();

var applications = await \_context.Auctions.Where(c => c.OrderId == id).ToListAsync();

foreach(var app in applications)

{

var contractor = await \_userRepository.GetUserById(app.ContractorId);

var commentsUser = \_context.Comments.Where(a => a.AppUserCommentId == app.ContractorId).ToList();

var countComments = commentsUser.Count();

double averageRating = 0;

if (commentsUser.Count != 0) averageRating = commentsUser.Sum(r => r.Rating) / (double)commentsUser.Count;

var applicationInfo = new AuctionViewModel

{

Id = app.Id,

AverageRating = averageRating,

Price = app.Price,

Contractor = contractor,

CountComments = countComments

};

applicationsInfo.Add(applicationInfo);

}

return View(applicationsInfo);

}

[HttpGet]

public async Task<IActionResult> ChoosContractor(int id)

{

var application = \_context.Auctions.FirstOrDefault(c => c.Id == id);

Order order = await \_orderRepository.GetByIdAsync(application.OrderId);

order.ContractorId = application.ContractorId;

order.OrderStatus = OrderStatus.InProgress;

order.Price = application.Price;

\_orderRepository.Update(order);

var applications = \_context.Auctions.Where(a => a.OrderId == order.Id).ToList();

foreach(var app in applications)

{

\_context.Auctions.Remove(app);

\_context.SaveChanges();

}

return RedirectToAction("Index","Dashboard");

}

[HttpGet]

public async Task<IActionResult> Complete(int id)

{

Order order = await \_orderRepository.GetByIdAsync(id);

if (order == null ) return View("Error");

order.OrderStatus = OrderStatus.Finished;

\_orderRepository.Update(order);

return RedirectToAction("Create", "Comment", new {id = order.ContractorId});

}

}

}

**HomeController**

using FreightTransportationWeb.Models;

using Microsoft.AspNetCore.Mvc;

using System.Diagnostics;

namespace FreightTransportationWeb.Controllers

{

public class HomeController : Controller

{

private readonly ILogger<HomeController> \_logger;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger;

}

public IActionResult Index()

{

return View();

}

public IActionResult Privacy()

{

return View();

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

}

}

**DashboardController**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Interfaces;

using FreightTransportationWeb.Models;

using FreightTransportationWeb.Repository;

using FreightTransportationWeb.ViewModels;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using System;

namespace FreightTransportationWeb.Controllers

{

public class DashboardController : Controller

{

private readonly IDashboardRepository \_dashboardRepository;

private readonly IHttpContextAccessor \_httpContextAccessor;

private readonly IWebHostEnvironment \_webHostEnvironment;

private readonly IUserRepository \_userRepository;

public DashboardController(IDashboardRepository dashboardRepository, IHttpContextAccessor httpContextAccessor, IWebHostEnvironment webHostEnvironment, IUserRepository userRepository)

{

\_dashboardRepository = dashboardRepository;

\_httpContextAccessor = httpContextAccessor;

\_webHostEnvironment = webHostEnvironment;

\_userRepository = userRepository;

}

public async void UserEdit(AppUser appUser, EditUserDashboardViewModel editVM)

{

if (editVM.Image != null)

{

if(appUser.Image != null)

{

string oldPhotoPath = Path.Combine(\_webHostEnvironment.WebRootPath, "images", appUser.Image);

if (System.IO.File.Exists(oldPhotoPath))

{

System.IO.File.Delete(oldPhotoPath);

}

}

string newUniqueFileName = Guid.NewGuid().ToString() + "\_" + Path.GetFileName(editVM.Image.FileName);

string newFilePath = Path.Combine(\_webHostEnvironment.WebRootPath, "images", newUniqueFileName);

using (var fileStream = new FileStream(newFilePath, FileMode.Create))

{

await editVM.Image.CopyToAsync(fileStream);

}

appUser.Image = newUniqueFileName;

}

appUser.Id = editVM.Id;

appUser.Address = editVM.AddressUser;

appUser.UserName = editVM.UserName;

appUser.PhoneNumber = editVM.PhoneNumber;

}

public async Task<IActionResult> Index()

{

var userOrders = await \_dashboardRepository.GetAllUserOrders();

var userViewModel = new DashboardViewModel()

{

Orders = userOrders

};

return View(userViewModel);

}

[HttpGet]

public async Task<IActionResult> Profile()

{

var user = await \_userRepository.GetUserById(\_httpContextAccessor.HttpContext.User.GetUserId());

var userDetailViewModel = new UserDetailViewModel()

{

Id = user.Id,

UserName = user.UserName,

AddressUser = user.Address,

Image = user.Image,

PhoneNumber = user.PhoneNumber,

Email = user.Email

};

return View(userDetailViewModel);

}

[HttpGet]

public async Task<IActionResult> EditUserProfile()

{

var curUserId = \_httpContextAccessor.HttpContext.User.GetUserId();

var user = await \_dashboardRepository.GetUserById(curUserId);

if(user == null)

{

return View("Error");

}

var editUserViewModel = new EditUserDashboardViewModel()

{

Id = curUserId,

UserName = user.UserName,

AddressUser = user.Address,

SaveImage = user.Image,

PhoneNumber = user.PhoneNumber

};

return View(editUserViewModel);

}

[HttpPost]

public async Task<IActionResult> EditUserProfile(EditUserDashboardViewModel editVM)

{

if (!ModelState.IsValid)

{

ModelState.AddModelError("", "Failed to edit profile");

return View("EditUserProfile",editVM);

}

var user = await \_dashboardRepository.GetUserByIdNoTracking(editVM.Id);

if(user != null )

{

UserEdit(user, editVM);

\_dashboardRepository.Update(user);

return RedirectToAction("Index");

}

else

{

return View(editVM);

}

}

public async Task<IActionResult> AcceptedOrders()

{

var contractorOrders = await \_dashboardRepository.GetAllContractorOrders();

var contractorAunctions = await \_dashboardRepository.GetAllContractorAuctions();

var userViewModel = new DashboardViewModel()

{

Orders = contractorOrders,

Auctions = contractorAunctions

};

return View(userViewModel);

}

}

}

**CommentController**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Models;

using Microsoft.AspNetCore.Mvc;

using System.Linq;

using System.Net;

using System.Xml.Linq;

namespace FreightTransportationWeb.Controllers

{

public class CommentController : Controller

{

private readonly ApplicationDbContext \_context;

public CommentController(ApplicationDbContext context)

{

\_context = context;

}

public IActionResult Index(string id)

{

return View(\_context.Comments.Where(a => a.AppUserCommentId == id).ToList());

}

// GET: ArticlesComments/Create

public IActionResult Create(string id)

{

ViewBag.ContractorId = id;

return View(\_context.Comments.Where(a => a.AppUserCommentId == id).ToList());

}

[HttpPost]

public IActionResult AddComment(string contractorId, int rating, string comment)

{

Comment newComment = new Comment()

{

AppUserCommentId = contractorId,

PublishedDate = DateTime.Now,

Comments = comment,

Rating = rating

};

\_context.Add(newComment);

\_context.SaveChanges();

return RedirectToAction("Index","User");

}

}

}

**AccountController**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Models;

using FreightTransportationWeb.ViewModels;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

namespace FreightTransportationWeb.Controllers

{

public class AccountController : Controller

{

private readonly UserManager<AppUser> \_userManager;

private readonly SignInManager<AppUser> \_signInManager;

private readonly ApplicationDbContext \_context;

private readonly IWebHostEnvironment \_webHostEnvironment;

public AccountController(UserManager<AppUser> userManager, SignInManager<AppUser> signInManager,ApplicationDbContext context, IWebHostEnvironment webHostEnvironment)

{

\_userManager=userManager;

\_signInManager=signInManager;

\_context=context;

\_webHostEnvironment = webHostEnvironment;

}

[HttpGet]

public IActionResult Login()

{

var repsonse = new LoginViewModel();

return View(repsonse);

}

[HttpPost]

public async Task<IActionResult> Login(LoginViewModel loginViewModel)

{

if(!ModelState.IsValid) return View(loginViewModel);

var user = await \_userManager.FindByEmailAsync(loginViewModel.EmailAddress);

if (user != null)

{

var passwordCheck = await \_userManager.CheckPasswordAsync(user, loginViewModel.Password);

if (passwordCheck)

{

var result = await \_signInManager.PasswordSignInAsync(user, loginViewModel.Password, false, false);

if(result.Succeeded)

{

return RedirectToAction("Index", "Order");

}

}

TempData["Error"] = "Введен неправильный пароль. Пожалуйста, попробуйте снова!";

return View(loginViewModel);

}

TempData["Error"] = "Данный пользователь не обнаружен. Пожалуйста, попробуйте снова!";

return View(loginViewModel);

}

[HttpGet]

public IActionResult Register()

{

var repsonse = new RegisterViewModel();

return View(repsonse);

}

[HttpPost]

public async Task<IActionResult> Register(RegisterViewModel registerViewModel)

{

if (!ModelState.IsValid) return View(registerViewModel);

var user = await \_userManager.FindByEmailAsync(registerViewModel.EmailAddress);

if (user != null)

{

TempData["Error"] = "This email address is already in use";

return View(registerViewModel);

}

string uniqueFileName = null;

if (registerViewModel.Image != null)

{

uniqueFileName = Guid.NewGuid().ToString() + "\_" + registerViewModel.Image.FileName;

string filePath = Path.Combine(\_webHostEnvironment.WebRootPath, "images", uniqueFileName);

using (var fileStream = new FileStream(filePath, FileMode.Create))

{

await registerViewModel.Image.CopyToAsync(fileStream);

}

}

var newUser = new AppUser()

{

Email = registerViewModel.EmailAddress,

UserName = registerViewModel.UserName,

Address = registerViewModel.AddressUser,

Image = uniqueFileName,

PhoneNumber = registerViewModel.PhoneNumber

};

var newUserResponse = await \_userManager.CreateAsync(newUser, registerViewModel.Password);

if(newUserResponse.Succeeded)

{

await \_userManager.AddToRoleAsync(newUser, UserRoles.User);

}

return RedirectToAction("Login", "Account");

}

[HttpPost]

public async Task<IActionResult> Logout()

{

await \_signInManager.SignOutAsync();

return RedirectToAction("Login", "Account");

}

}

}

1. Репозитории

**UserRepository**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Interfaces;

using FreightTransportationWeb.Models;

using Microsoft.EntityFrameworkCore;

namespace FreightTransportationWeb.Repository

{

public class UserRepository : IUserRepository

{

private readonly ApplicationDbContext \_context;

public UserRepository(ApplicationDbContext context)

{

\_context = context;

}

public bool Add(AppUser user)

{

throw new NotImplementedException();

}

public bool Delete(AppUser user)

{

throw new NotImplementedException();

}

public async Task<IEnumerable<AppUser>> GetAllUsers()

{

return await \_context.Users.ToListAsync();

}

public async Task<AppUser> GetUserById(string id)

{

return await \_context.Users.Include(a => a.Address)

.FirstOrDefaultAsync(a => a.Id == id);

}

public bool Save()

{

var saved = \_context.SaveChanges();

return saved > 0 ? true:false;

}

public bool Update(AppUser user)

{

\_context.Update(user);

return Save();

}

}

}

**OrderRepository**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Interfaces;

using FreightTransportationWeb.Models;

using Microsoft.EntityFrameworkCore;

namespace FreightTransportationWeb.Repository

{

public class OrderRepository : IOrderRepository

{

private readonly ApplicationDbContext \_context;

public OrderRepository(ApplicationDbContext context)

{

\_context = context;

}

public bool Add(Order order)

{

\_context.Add(order);

return Save();

}

public bool Delete(Order order, DeliveryAddress deliveryAddress, Package package)

{

\_context.Remove(order);

\_context.Remove(deliveryAddress);

\_context.Remove(package);

return Save();

}

public async Task<IEnumerable<Order>> GetAll()

{

return await \_context.Orders.Include(c => c.Package)

.Include(d => d.DeliveryAddress)

.Include(c => c.Customer.Address)

.ToListAsync();

}

public async Task<Order> GetByIdAsync(int id)

{

return await \_context.Orders.Include(c => c.Customer)

.Include(c => c.Customer.Address)

.Include(a => a.DeliveryAddress)

.Include(p => p.Package)

.FirstOrDefaultAsync(i => i.Id == id);

}

public async Task<Order> GetByIdAsyncNoTracking(int id)

{

return await \_context.Orders.Include(c => c.Customer)

.Include(c => c.Customer.Address)

.Include(a => a.DeliveryAddress)

.Include(p => p.Package)

.AsNoTracking()

.FirstOrDefaultAsync(i => i.Id == id);

}

public async Task<IEnumerable<Order>> GetOrderByCity(string city)

{

return await \_context.Orders.Where(c=>c.Customer.Address.City.Contains(city)).ToListAsync();

}

public bool Save()

{

var saved = \_context.SaveChanges();

return saved > 0 ? true: false;

}

public bool Update(Order order)

{

\_context.Update(order);

return Save();

}

}

}

**DashboardRepository**

using FreightTransportationWeb.Data;

using FreightTransportationWeb.Interfaces;

using FreightTransportationWeb.Models;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.ChangeTracking;

namespace FreightTransportationWeb.Repository

{

public class DashboardRepository : IDashboardRepository

{

private readonly ApplicationDbContext \_context;

private readonly IHttpContextAccessor \_httpContextAccessor;

public DashboardRepository(ApplicationDbContext context, IHttpContextAccessor httpContextAccessor)

{

\_context = context;

\_httpContextAccessor = httpContextAccessor;

}

public async Task<List<Order>> GetAllUserOrders()

{

var curUser = \_httpContextAccessor.HttpContext?.User.GetUserId();

var userOrders = \_context.Orders.Where(i => i.Customer.Id == curUser).Include(p=>p.Package);

return userOrders.ToList();

}

public async Task<List<Order>> GetAllContractorOrders()

{

var curUser = \_httpContextAccessor.HttpContext?.User.GetUserId();

var userOrders = \_context.Orders.Where(i => i.Contractor.Id == curUser).Include(p => p.Package);

return userOrders.ToList();

}

public async Task<List<Order>> GetAllContractorAuctions()

{

List<Order> orders = new List<Order>();

var curUser = \_httpContextAccessor.HttpContext?.User.GetUserId();

var userAuctions = \_context.Auctions.Where(i => i.ContractorId == curUser).ToList();

foreach(var item in userAuctions)

{

var order = \_context.Orders.Include(p => p.Package).FirstOrDefault(a => a.Id == item.OrderId);

orders.Add(order);

}

return orders;

}

public async Task<AppUser> GetUserById(string id)

{

return await \_context.Users.Include(a => a.Address)

.FirstOrDefaultAsync(a => a.Id == id);

}

public async Task<AppUser> GetUserByIdNoTracking(string id)

{

return await \_context.Users.Where(i => i.Id == id).AsNoTracking().FirstOrDefaultAsync();

}

public bool Update(AppUser appUser)

{

\_context.Users.Update(appUser);

return Save();

}

public bool Save()

{

var saved = \_context.SaveChanges();

return saved > 0 ? true: false;

}

}

}