МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

Специализация 1-40 05 01 «Информационные системы и технологии»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**курсового проекта**

по дисциплине «Программирование в Internet»

Тема: Web-приложение «Парковка»

Исполнитель

Студент(ка) 4 курса 1 группы Куликов Дмитрий Николаевич

(Ф.И.О.)

Руководитель проекта асс. Дубовик М.В.

(подпись, учен. степень, звание, должность, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов В.В.

(подпись, учен. степень, звание, должность, Ф.И.О.)

Консультанты асс. Дубовик М.В.

(подпись, учен. степень, звание, должность, Ф.И.О.)

Нормоконтролер асс. Дубовик М.В.

(подпись, учен. степень, звание, должность, Ф.И.О.)

Курсовой проект защищен с оценкой

**Минск 2022**

**Оглавление**

[Введение 4](#_Toc121861373)

[1 Постановка задачи 5](#_Toc121861374)

[1.1 Обзор аналогов 5](#_Toc121861375)

[1.1.1 Parkopedia 5](#_Toc121861376)

[1.1.2 ParkApp 6](#_Toc121861377)

[1.1.3 Парковки Москвы 2.0 7](#_Toc121861378)

[1.1.4 ParkMan 8](#_Toc121861379)

[1.2 Постановка цели 9](#_Toc121861380)

[2 Проектирование 10](#_Toc121861381)

[2.1 Выбор платформы 10](#_Toc121861382)

[2.1.1 Серверная часть 10](#_Toc121861383)

[2.1.2 Клиентская часть 10](#_Toc121861384)

[2.1.3 Централизованная база данных 11](#_Toc121861385)

[2.2 Диаграмма взаимодействия 11](#_Toc121861386)

[2.3 Логическая схема базы данных 12](#_Toc121861387)

[2.4 Описание базы данных 12](#_Toc121861388)

[3 Разработка 14](#_Toc121861389)

[3.1 Разработка серверной части 14](#_Toc121861390)

[3.2 Разработка клиентской части 16](#_Toc121861391)

[3.3 Создание базы данных 20](#_Toc121861392)

[4 Тестирование 22](#_Toc121861393)

[5 Руководство программиста 26](#_Toc121861394)

[5.1 Функционал приложения 26](#_Toc121861395)

[5.1.1 Client 27](#_Toc121861396)

[5.1.2 Administrator 30](#_Toc121861397)

[5.2 Создание конфигурационных файлов Docker 32](#_Toc121861398)

[5.3 Развертывание приложения в контейнере Docker 35](#_Toc121861399)

[5.4 Развертка приложения на ОС. 35](#_Toc121861400)

[Заключение 36](#_Toc121861401)

[Список литературы 37](#_Toc121861402)

[Приложение A 38](#_Toc121861403)

[Приложение Б 40](#_Toc121861404)

[Приложение В 44](#_Toc121861405)

# Введение

Объектом курсовой работы является программное средство «Парковка», подобные средства помогают многим людям во всем мире быстро и удобно найти парковочное место и, при необходимости, заранее забронировать его.

Для начала разберемся с термином парковки. Парковка представляет собой определённо отведённое место для безопасной стоянки транспортных средств в нерабочем состоянии. Также может означать сам процесс постановки транспортного средства на стояночное место. Наиболее распространенным транспортным средством является автомобиль.

Стоит различать понятия парковки и стоянки. Стоянкой называется специально обозначенное и оборудованное место, являющееся, в том числе частью автомобильной дороги или примыкает к ней, и предназначенное для организационной стоянки транспортных средств на платной основе или без платы. А стоянкой называется специальная открытая площадка, предназначенная для хранения автомототранспортных средств. Исходя из этих понятий следует, что стоянка предназначена для хранения автомототранспортных средств, а парковка является частью автомобильной дороги и выполняет вспомогательную роль по организационной стоянке транспортных средств.

Количество автомобилей в мире с каждым годом постоянно растет, в отличие от числа парковочных мест, которое не изменяется уже в течение нескольких лет. Нормально припарковаться в нужном месте не всегда и не всем удается. В таких случаях им помогают различные приложения парковки, которые помогают найти подходящее место, предоставляют возможность мониторинга и контроль ситуации в целом.

Эти средства, на данный момент, существует большое количество и достаточно большое количество вариаций этих средств, из-за чего каждый новый производитель программы пытается улучшить функционал и удобность своего приложения, и как следствие данная область разработки достаточно продвинута и полна своих тонкостей. Каждое приложение парковки реализует свою задумку, такие как, увеличение количества возможностей пользователя, уменьшение количества действий для конкретного действия, реализация более понятной структуры приложения и т.д.

Целью курсового проекта является разработка приложения, которое позволит пользователям бронировать парковку на определённое время, вводить данные своего автомобиля в базу данных и позволит просматривать статус парковки и статус своего автомобиля на данной парковке.

# 1 Постановка задачи

1.1 Обзор аналогов

1.1.1 Parkopedia

*Parkopedia* – это одно из самых популярных приложений парковки. Parkopedia использует в своей работе базу парковочных мест и дает возможность построить маршрут до любого из них. Отличием данного сервиса является ее охват. Дело в том, что приложение работает по всему миру и, по заверениям разработчиков, поможет найти парковочные места даже в экзотических странах.

Плюсы:

* найти парковку по адресу или местоположению;
* получить маршрут проезда прямо до въезда на парковку;
* получить информацию о количестве свободных мест на парковке на данный момент (там, где эта информация доступна);
* найти информацию о часах работы парковки, актуальных ценах, способах оплаты и т.д.;
* с помощью фильтров, таких как "бесплатно", "принимаются кредитные карты" и т.д., быстро найти нужную вам парковку.

Минусы:

* сложный интерфейс;
* медленная скорость работы и зависания.

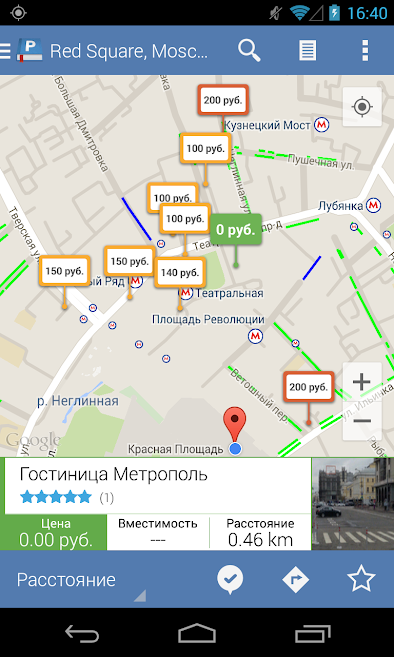
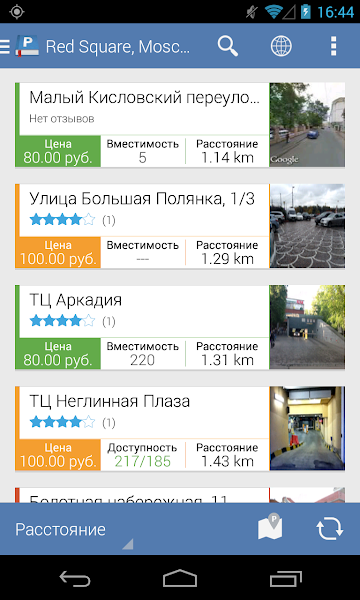
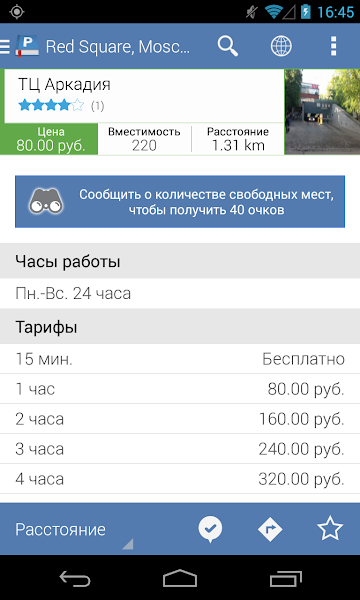
  

Рисунок 1.1 – Интерфейс приложения «Parkopedia»

### 1.1.2 ParkApp

*ParkApp* – приложение парковки, которое, в отличие конкурентов, не только находит ближайшие бесплатные парковки (в базе приложения их более девяти тысяч), но и позволяет заранее узнать, кто из пользователей собирается освободить парковочное место.

Это стало возможным благодаря «умному» алгоритму, который узнает о намерении водителя покинуть стоянку и сообщает об этом другим пользователям, находящимся поблизости. Также приложение предупреждает о приближающихся эвакуаторах и в любой непредвиденной ситуации позволяет вызвать службу дорожной помощи.

Плюсы:

* покажет вам освобождающиеся и свободные парковки в реальном времени;
* подберет ближайшую парковку и построит до нее маршрут;
* определяет зону парковки и составляет платежную СМС.

Минусы:

* нет функции оплатить парковку;
* плохой интерфейс;
* плохая оптимизация приложения;
* потребляет большое количество энергии аккумулятора.

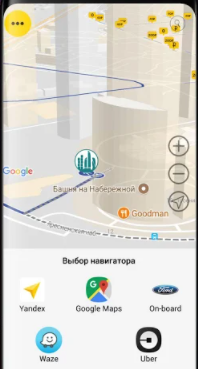
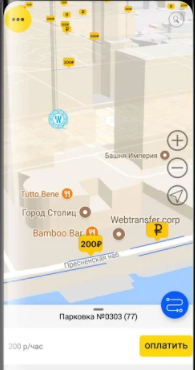
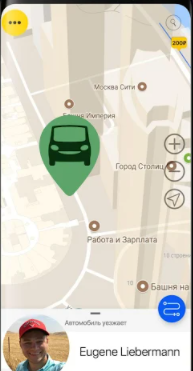
  

Рисунок 1.2 – Интерфейс приложения «ParkApp»

### 1.1.3 Парковки Москвы 2.0

*Парковки Москвы 2.0* – приложение содержит базу парковок при торговых центрах, вокзалах и аэропортах, а также платных и бесплатных стоянок столицы. В общей сложности сервис знает о 600 парковках по всему городу. Приложение помогает проложить маршрут до ближайшей свободной стоянки, учитывая график ее работы (на карте отображаются парковки, которые работают в момент обращения). Также с помощью данного сервиса пользователь может оплатить свое пребывание на парковочном месте.

Плюсы:

* легко определить место парковки;
* оплатить его с помощью SMS;
* следить за оставшимся временем и получать уведомления о его окончании;
* продлевать оплаченное время.

Минусы:

* нет оплаты картой;
* приложение только формирует SMS для оплаты парковки;
* частые сбои и неполадки приложения.

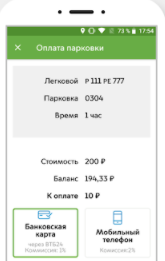
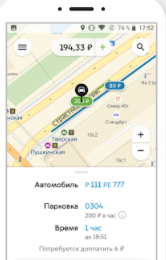
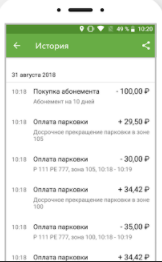
 

Рисунок 1.3 – Интерфейс приложения «Парковки Москвы 2.0»

### 1.1.4 ParkMan

*ParkMan* – приложение парковки получившее широкое распространение в Европе и США. Приложение доступно пользователям смартфонов как на операционной системе Андроид, так и для iOS устройств. Оплачивать услуги можно как кредитной и конечно дебетовой картой, так и по счёту заказа. Стоимость парковки за предыдущий месяц списывается с кредитной карты в первый день следующего месяца.

Плюсы:

* просмотр доступных вариантов парковки поблизости;
* продлить или закончить парковку в соответствии с потребностями;
* оплата только во время парковки;
* просмотр маршрута на припаркованный автомобиль.

Минусы:

* отсутствие русского языка;
* приложение является достаточно дорогим.

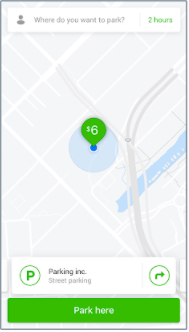
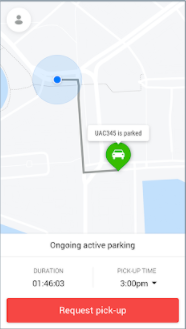
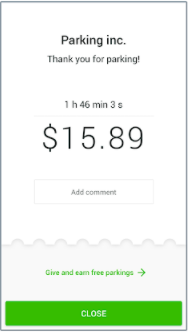
  

Рисунок 1.4 – Интерфейс приложения «ParkMan»

1.2 Постановка цели

Исходя из проанализированных аналогов были выявлены основные задачи и требования к разрабатываемому приложению.

Основной задачей моего курсового проекта является разработка веб приложения, позволяющего пользователю находить ближайшее парковочное место, возможность бронирования парковочного места, возможность просмотра свободных парковочных мест и общую сумму заказов, совершенных пользователем в течении всего периода, с момента регистрации человека на сайте веб приложения. Программное средство должно содержать профиль авторизованного пользователя с его контактными данными. Интерфейс должен быть простым и удобным для использования.

Так же не плохо было бы сделать адаптацию под разные формы экрана. Интерфейс должен быть простым и понятным.

Приложение будет разработано на базе ASP.NET Core, в качестве серверной части, и основного обработчика запросов от клиентов с клиентской части веб приложения. Роль клиентского приложения будет занимать открытая и свободная платформа для разработки [веб-приложений](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5), написанная на языке [TypeScript](https://ru.wikipedia.org/wiki/TypeScript), разрабатываемая командой из компании [Google](https://ru.wikipedia.org/wiki/Google_(%D0%BA%D0%BE%D0%BC%D0%BF%D0%B0%D0%BD%D0%B8%D1%8F)), а также сообществом разработчиков из различных компаний Angular.

В качестве хранилища данных будет выступать[система управления реляционными базами данных (РСУБД)](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94), разработанная корпорацией [Microsoft](https://ru.wikipedia.org/wiki/Microsoft), Microsoft SQL Server. Основной используемый язык запросов — [Transact-SQL](https://ru.wikipedia.org/wiki/Transact-SQL), создан совместно Microsoft и [Sybase](https://ru.wikipedia.org/wiki/Sybase). Transact-SQL является реализацией стандарта [ANSI](https://ru.wikipedia.org/wiki/%D0%90%D0%BC%D0%B5%D1%80%D0%B8%D0%BA%D0%B0%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D0%BD%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%81%D1%82%D0%B8%D1%82%D1%83%D1%82_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BE%D0%B2)/[ISO](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%B4%D1%83%D0%BD%D0%B0%D1%80%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BE%D1%80%D0%B3%D0%B0%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%BE_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8) по структурированному языку запросов ([SQL](https://ru.wikipedia.org/wiki/SQL)) с расширениями. Используется для работы с базами данных размером от персональных до крупных баз данных масштаба предприятия; конкурирует с другими СУБД в этом сегменте рынка.

# 2 Проектирование

* 1. Выбор платформы

Для реализации проекта необходимо, первым делом, построить структуру проекта и выбрать платформу разработки.

Структура проекта стоит из трех компонентов:

* серверная часть (web-api);
* клиентская часть (UI);
* централизованная база данных.

### 2.1.1 Серверная часть

Для реализации серверной части я выбрал платформу .NET Core. Платформа ASP.NET Core представляет технологию от компании Microsoft, предназначенную для создания различного рода веб-приложений: от небольших веб-сайтов до крупных веб-порталов и веб-сервисов.

С одной стороны, ASP.NET Core является продолжением развития платформы ASP.NET. Но, с другой стороны, это не просто очередной релиз. Выход ASP.NET Core фактически означает революцию всей платформы, ее качественное изменение.

ASP.NET Core может работать поверх кроссплатформенной среды .NET Core, которая может быть развернута на основных популярных операционных системах: Windows, Mac OS, Linux. И таким образом, с помощью ASP.NET Core мы можем создавать кроссплатформенные приложения. И хотя Windows в качестве среды для разработки и развертывания приложения до сих пор превалирует, но теперь уже мы не ограничены только этой операционной системой. То есть мы можем запускать веб-приложения не только на ОС Windows, но и на Linux и Mac OS. А для развертывания веб-приложения можно использовать традиционный IIS, либо кроссплатформенный веб-сервер Kestrel.

### 2.1.2 Клиентская часть

Для реализации клиентской части я выбрал прогрессивный фреймворк Angular. Angular представляет фреймворк от компании Google для создания клиентских приложений. Прежде всего он нацелен на разработку SPA-решений (Single Page Application), то есть одностраничных приложений. В этом плане Angular является наследником другого фреймворка AngularJS. В то же время Angular это не новая версия AngularJS, а принципиально новый фреймворк.

Angular предоставляет такую функциональность, как двустороннее связывание, позволяющее динамически изменять данные в одном месте интерфейса при изменении данных модели в другом, шаблоны, маршрутизация и так далее.

Одной из ключевых особенностей Angular является то, что он использует в качестве языка программирования TypeScript.

Но мы не ограничены языком TypeScript. При желании можем писать приложения на Angular с помощью таких языков как Dart или JavaScript. Однако TypeScript все-таки является основным языком для Angular.

### 2.1.3 Централизованная база данных

Для хранения данных я использовал Microsoft SQL Server. Microsoft SQL Server является одним из наиболее популярных систем управления базами данных. Данная СУБД подходит для самых различных проектов: от небольших приложений до больших высоконагруженных проектов.

SQL Server был создан компанией Microsoft. Первая версия вышла в 1987 году. А текущей версией является версия 2022, которая вышла в ноябре 2022 году и которая будет использоваться в текущем руководстве.

SQL Server долгое время был исключительно системой управления базами данных для Windows, однако начиная с версии 16 эта система доступна и на Linux.

SQL Server характеризуется несколькими ключевыми особенностями.

1. Производительность, SQL Server работает очень быстро;
2. Надёжность и безопасность, SQL Server предоставляет шифрование данных;
3. Простота, с данной СУБД относительно легко работать и вести администрирование.

2.2 Диаграмма взаимодействия

Первым делом пользователь взаимодействует с визуальным интерфейсом. Для отображения какой-либо информации или данных на пользовательском интерфейсе, клиентская часть делает запрос на сервер к конкретному контроллеру. Контроллер в свою очередь принимает запрос и, в зависимости от характеристик запроса и реализации контроллера, делает запрос на базу данных, получает данные, обрабатывает их и результат обработки посылает в виде ответа клиенту. Клиент получает данные и отображает их на экране.

На рисунке 2.1 показана схема взаимодействия всех трех компонентов веб-приложения.

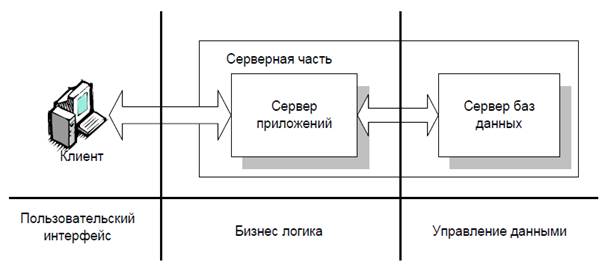


Рисунок 2.1 – Диаграмма взаимодействия компонентов веб-приложения

* 1. Логическая схема базы данных

На рисунке 2.2 отображена схема связей таблиц базы данных.

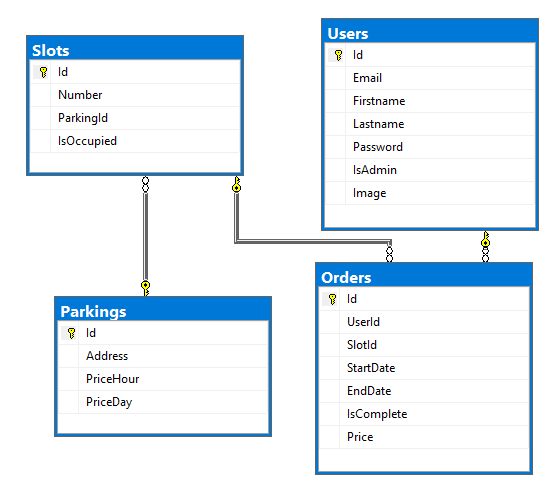


Рисунок 2.2 – Схема связей таблиц базы данных

* 1. Описание базы данных

В базе данных реализовано четыре таблицы:

* Users;
* Slots:
* Parkings;
* Orders.

Таблица **Users** предназначена для хранения информации о зарегистрированных пользователях, состоит из 7 столбцов:

* Id – ключевое поле таблицы, которое содержит уникальный индекс для каждого пользователя;
* Email ­– почта, уникальна для каждого пользователя;
* Firstname – фамилия пользователя;
* Lastname – имя пользователя;
* Password ­– пароль пользователя;
* IsAdmin – логическое поле, которое показывает, является ли пользователь администратором;
* Image – изображение пользователя.

Таблица **Slots** предназначена для хранения мест стоянок и состоит из 4 основных столбцов:

* Id ­– ключевое поле таблицы, которое содержит уникальный индекс для каждого места;
* Number – порядковый номер места на парковке;
* ParkingId – уникальный индекс парковки, которой принадлежит данное место;
* IsOccupied – логическое поле, показывающее статус места (занято, свободно).

Таблица **Parkings** предназначена для хранения парковок и состоит из 4 основных столбцов:

* Id – ключевое поле таблицы, которое содержит уникальный индекс для каждой парковки;
* Address – расположение парковки, её название;
* PriceHour – цена парковочного места за час;
* PriceDay – цена парковочного места за день.

Таблица **Orders** предназначена для хранения всех заказов и состоит из 7 столбцов:

* Id – ключевое поле таблицы, которое содержит уникальный индекс заказа;
* UserId – уникальный индекс пользователя сделавшего заказ;
* SlotId – уникальный индекс места на которое сделали заказ;
* StartDate – время начала заказа;
* EndDate – время окончания заказа;
* IsComplete – логическое поле, которое показывает статус заказа (активен, завершен);
* Price – цена заказа.

# 3 Разработка

Как говорилось ранее, для реализации проекта необходимо, первым делом, построить структуру проекта и выбрать платформу разработки.

Структура проекта стоит из трех компонентов:

* серверная часть (web-api);
* клиентская часть (UI);
* централизованная база данных.

3.1 Разработка серверной части

Для реализации серверной части использовалась кроссплатформенная технология ASP.NET Core. Язык программирования C#. Первым делом был создаем проект типа «web-api», который предназначен для создания серверной части. Структура проекта состоит из папок «Properties», «wwwroot», «ClientApp», «Context», «Controllers», «Helpers», «Models», «Migrations», «Pages» и классов «Program», «Startup», изображенных на рисунке 3.1.

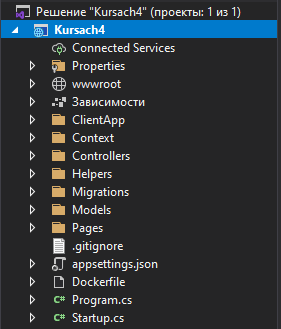


Рисунок 3.1 – Структура сервера web-api

Папка «wwwroot» хранит статически данные.

Папка «Models» хранит модели сущностей таблиц из базы данных. Именно с помощью их сервер может взаимодействовать с базой данных. Каждая сущность полностью соответствует таблице, листинг, одной из которых, размещен ниже.

|  |
| --- |
| public class User  {  [Key]  public int Id {get; set;}  [Column (TypeName = "nvarchar (MAX)")]  public string Email {get; set;}  [Column (TypeName = "nvarchar (MAX)")]  public string Firstname {get; set;}  [Column (TypeName = "nvarchar (MAX)")]  public string Lastname {get; set;}  [Column (TypeName = "varbinary (MAX)")]  public byte [] Image {get; set;}    [Column (TypeName = "nvarchar (MAX)")]  public string Password {get; set;}  [Column (TypeName = "nvarchar (MAX)")]  public string Role {get; set;}  [JsonIgnore ()]  public ICollection<Order> Orders {get; set;}  public User ()  {  Orders = new List<Order> ();  }  } |

Листинг 3.1 – Структура сущности «User»

Структуры всех сущностей можно просмотреть в приложении [A](#_Листинг_A).

Папка «Controllers» предназначена для обработки запросов от клиента и работы с базой данных. В ней находятся контроллеры для работы с каждой сущностью. Одним из условий проекта было построения приложения, который работает в стиле REST.

REST-архитектура предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером:

* GET;
* POST;
* PUT;
* DELETE.

Структура контроллера изображена на рисунке 3.2.

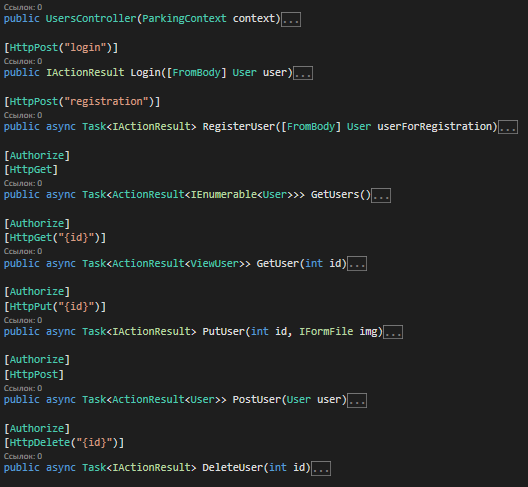


Рисунок 3.2 – Структура контроллера

Папка «Migrations» предназначена для сохранения миграций в базу данных. Сами по себе миграции используются для того, чтобы создать таблицы в базе данных на основании существующих сущностей.

С класса «Program», по сути, начинается выполнение приложения.

Класс «Startup» является входной точкой в приложение ASP.NET Core. По сути, это уровень конфигурации приложения, определяющий промежуточный слой обработки запросов, регистрирующий зависимости сторонних сервисов. Проект был разработан с помощью среды разработки Microsoft Visual Studio 2019.

3.2 Разработка клиентской части

Для реализации клиентской части я выбрал прогрессивный фреймворк Angular. Прежде всего он нацелен на разработку SPA-решений (Single Page Application), то есть одностраничных приложений.

Angular предоставляет такую функциональность, как двустороннее связывание, позволяющее динамически изменять данные в одном месте интерфейса при изменении данных модели в другом, шаблоны, маршрутизация и так далее.

Одной из ключевых особенностей Angular является то, что он использует в качестве языка программирования TypeScript. TypeScript имеет расширение \*.ts Структуру проекта можно увидеть на рисунке 3.3.

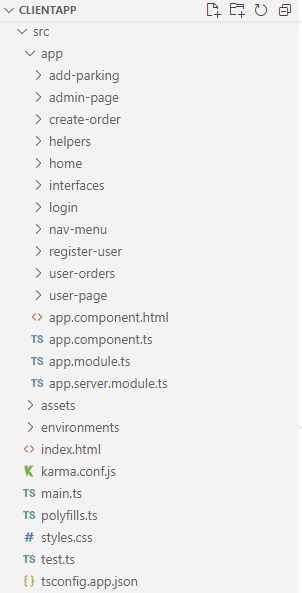


Рисунок 3.3 – Структура проекта пользовательского интерфейса

Сам по себе компонент состоит из трех секций:

* template;
* scripts;
* style.

Секция «Template» содержит в себе html разметку, которая отвечает за представление, то есть за то, что видит пользователь. Так же в этой секции используются компоненты. Сами компоненты необходимо импортировать из папки и объявить в секции «components». Структура секции «Template» изображена на рисунке 3.4



Рисунок 3.4– Секция «template»

В секции «scripts» находится вся логика взаимодействия интерфейса с серверной частью. В данной секции находится множества частей, опишу главные из них.

Часть «variables» содержит в себе переменные, с помощью которых можно взаимодействовать с представлением и серверной частью. В переменных хранятся данные, полученные от сервера. Так же эти переменные можно связать с html-разметкой.

Часть «components» позволяет зарегистрировать компоненты. После регистрации компонента, его можно использовать как часть html-разметки.

Часть «class», по сути, является ключевой, так как именно там находятся методы, которые делают запросы к серверной части и сохраняют полученные данные в виде ответа от сервера в переменных.

Пример данной секции можно увидеть на рисунке 3.5.



Рисунок 3.5 – Секция «scripts»

Секция «style» позволяет создать локальные стили для каждого компонента. Она не является обязательной.

Запросы к серверу выполняются с помощью Angular-библиотеки «http». Что бы сделать запрос, необходимо указать тип запроса (get, post, put, delete), в параметрах указать url на сервер, заголовки и при необходимости, в зависимости от запроса, можно передать данные на сам сервер. В обработчике «subscribe» в качестве параметра используются переменная ответа, куда приходит сам ответ от сервера. Что бы получить сами данные, необходимо выбрать переменную «response». Структура одного из методов размещена на рисунке 3.6.

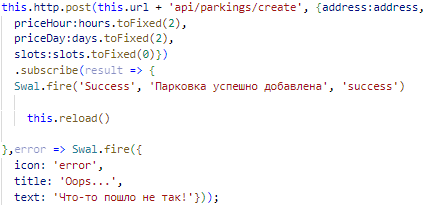


Рисунок 3.6 – Структура метода, использующего http-запрос

В случае ошибки, используются обработчик «error». В качестве параметра находится переменная, которая сохраняет в себе ответ сервера в результате ошибки.

3.3 Создание базы данных

Решил использовать базу данных Microsoft SQL Server, так как она является одной из самых популярных платформ. Как ранее говорилось, таблицы в базе данных создаются с помощью миграций. По-другому можно сказать, что используется подход «Entity Framework Code First». Для создания скрипты нужно прописать команду «Add-Migration nameMigration». После чего система автоматически создаст скрипт создания таблиц. Пример части скрипта находится в листинге 3.3 ниже:

|  |
| --- |
| migrationBuilder.CreateTable(  name: "Users",  columns: table => new  {  Id = table.Column<int>(type: "int", nullable: false)  .Annotation("SqlServer:Identity", "1, 1"),  Email = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Firstname = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Lastname = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Image = table.Column<byte[]>(type: "varbinary(MAX)", nullable: true),  Password = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Role = table.Column<string>(type: "nvarchar(MAX)", nullable: true)  },  constraints: table =>  {  table.PrimaryKey("PK\_Users", x => x.Id);  }); |

Рисунок 3.3 – Скрипт создания таблицы «Users»

Полный листинг скрипта находится в приложении [В](#_Приложение_С).

# 4 Тестирование

Тестирование веб-приложений – это комплекс услуг, который может включать в себя различные виды тестирования ПО.

Основная цель любого тестирования, в том числе и тестирования веб-приложений, – обнаружить все ошибки в программном обеспечении и разработать рекомендации по их предотвращению в будущем.

Для того, чтобы начать тестирование приложения, необходимо первым делом запустить сервер и потом проект с пользовательским интерфейсом. После запуска мы попадаем на приветственную страницу. Затем, чтобы получить доступ к функционалу приложения необходимо пройти на страницу входа, нажав на кнопку «Вход». Пример проверки на вход пользователя, который не зарегистрирован, изображен на рисунке 4.1.

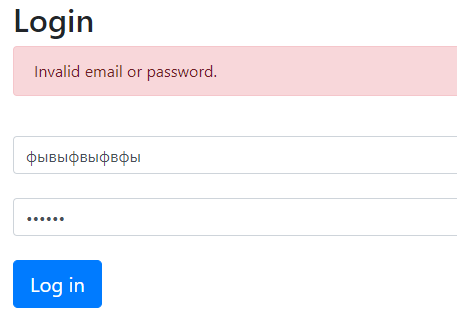


Рисунок 4.1 – Тестирование окна входа

Далее протестируем форму регистрации. При не полном заполнении полей, выдаст сообщение, что модель не валидна, Пример изображен на рисунке 4.2.

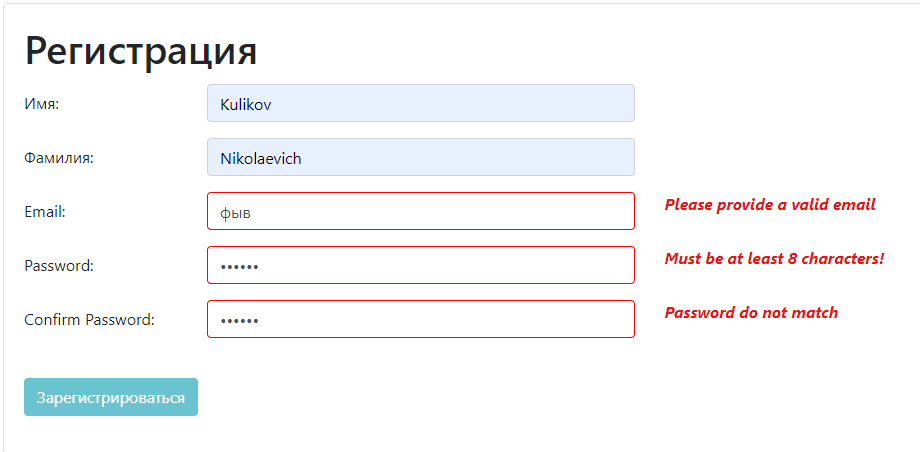


Рисунок 4.2 – Тестирование окна регистрации

Далее протестируем модель полностью валидной формы регистрации, в этом случае кнопка регистрации будет доспутна для нажатия пользователем. Пример изображен на рисунке 4.3.

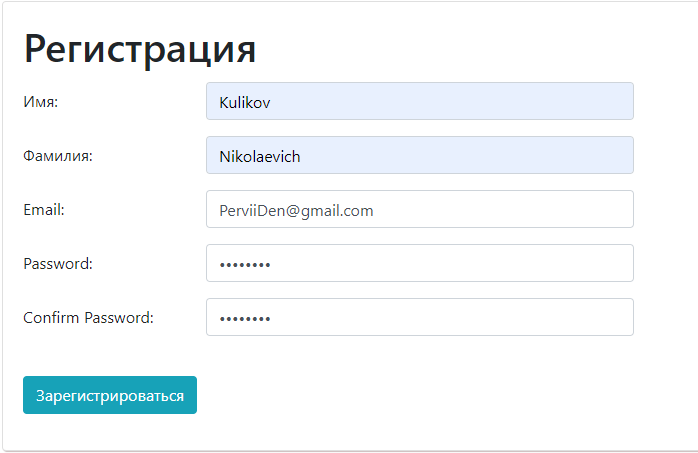


Рисунок 4.3 – Тестирование валидного окна регистрации

При добавлении данных в таблицы происходит проверка на допустимые значения. В данном случае при в вводе несуществующей парковки, появляется сообщение о неправильности данных, изображенное на рисунке 4.4.

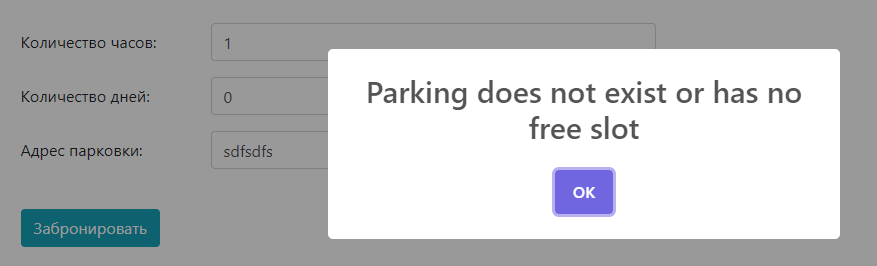


Рисунок 4.4 – Тестирование валидности значений при добавлении

Итак, представим, что у нас есть администратор, который хочет добавить парковку. Он её добавляет и уходит на перерыв за чаем. Затем он покупает чебупелли и возвращается к добавлению парковки. Он забывает, что в ввел парковку и пытается её добавить повторно. Что бы не было коллизий, в случае добавления новой парковки с тем же адресом, пользователю отображается сообщение об невозможности создания запроса, пример которого можно увидеть на рисунке 4.5 ниже.

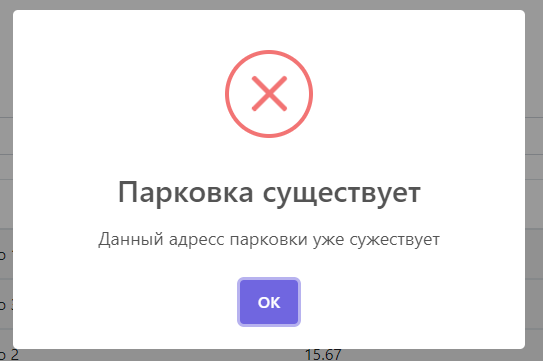


Рисунок 4.5 – Тестирование добавления заявки повторно

Протестируем адаптивность приложения. Как видим, при изменении размера экрана, элементы формы регистрации, изображенные на рисунке 4.6, сдвигаются в определенное место.

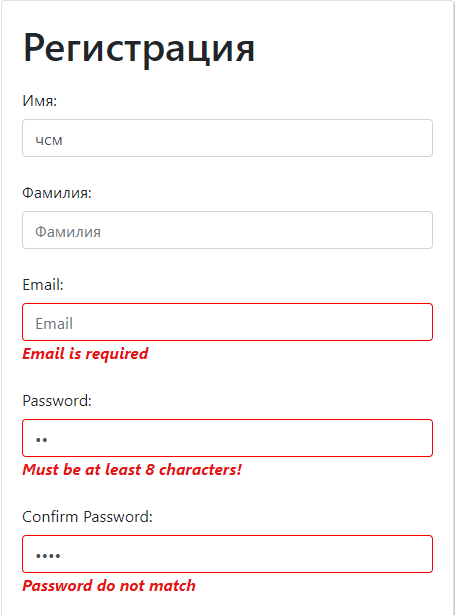


Рисунок 4.6 – Тестирование адаптивности приложения

После тестирования можно сказать, что приложение функционирует и успешно выполняет заданные цели.

# 5 Руководство программиста

5.1 Функционал приложения

После запуска приложения пользователь попадает на главная. При нажатии на кнопку «Вход», пользователь попадает на страницу, изображенной на рисунке 5.1.

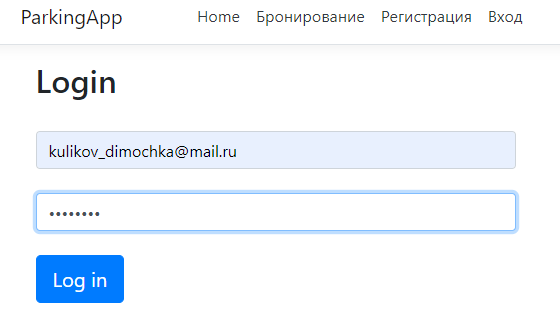


Рисунок 5.1 – Страница входа в приложение

Если пользователь прошел регистрацию, и администратор одобрил его аккаунт, то он может войти. В случае, если пользователь еще не зарегистрирован, он может перейти на страницу регистрации, нажав кнопку «Register». Пользователь попадает на страницу регистрации, изображенную на рисунке 5.2.

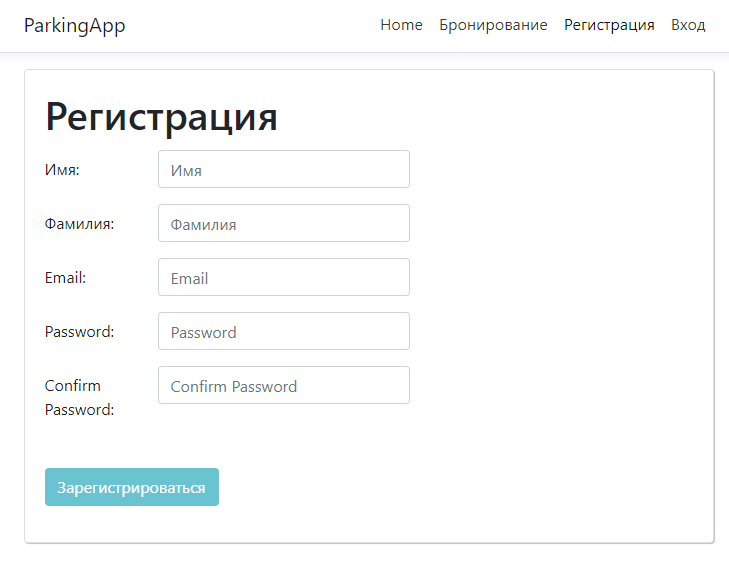


Рисунок 5.2 – Форма регистрации

Что бы зарегистрировать, нужно заполнить корректно все поля и отправить форму с данными на сервер с помощью кнопки «Зарегистрироваться». После регистрации, пользователю нужно подождать, когда администратор примет или отклонит его заявку.

5.1.1 Client

Войдя в учетную запись, пользователь попадает на главную страницу, на которой находится меню с доступными функциями:

* бронирование места
* просмотр всех заказов данного пользователя;
* просмотр личной страницы пользователя.

При бронировании места на парковке необходимо заполнить поля:

* количество часов;
* количество дней;
* адрес парковки;

Так же можно осуществить поиск интересующей клиента парковки и фильтрации по одному из столбцов таблицы, после чего клиент может нажать на строку парковки и, если количество свободных мест на выбранной парковке больше, чем одно то можно совершить заказ как показано на рисунке 5.3.

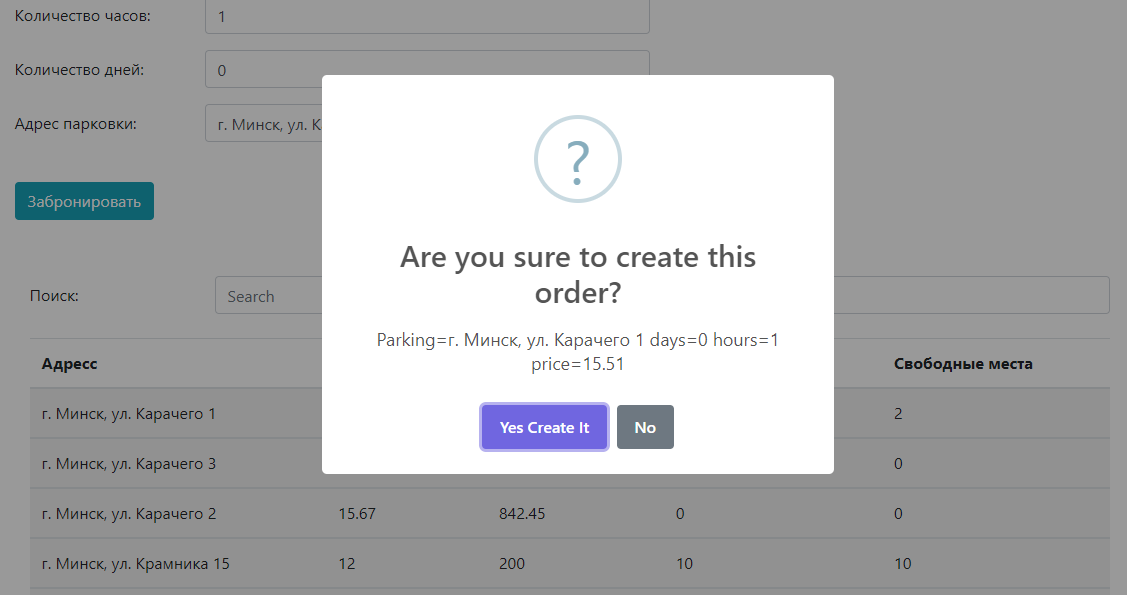


Рисунок 5.3 – Страница «Бронирование»

На странице «Ваши заказы» находится список всех заказов, совершенных пользователем, которые можно увидеть на рисунке 5.4.

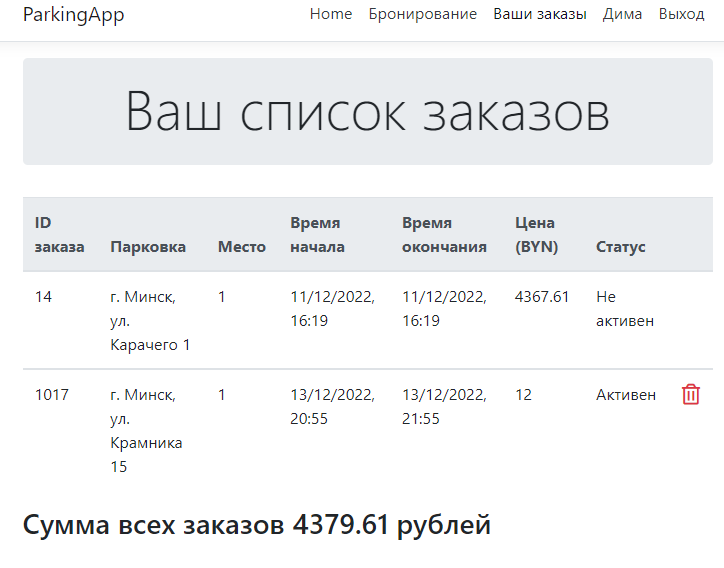


Рисунок 5.4 – Страница «Ваши заказы»

Бронирование можно преждевременно закрыть, для этого нужно нажать на значок урны справа от заказа.

При нажатии на значок урны появляется диалоговое окно, которое спрашивает подтверждения действия, как изображено на рисунке 5.5.

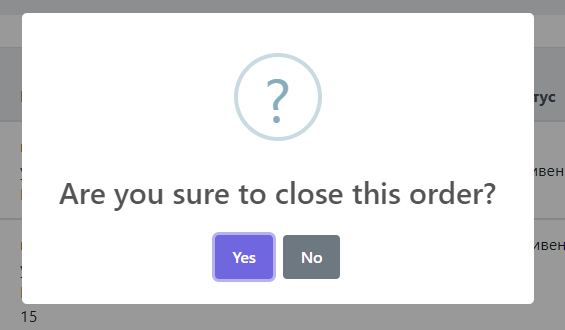


Рисунок 5.5 – Окно подтверждения закрытия заказа

Также есть страница пользователя, в которой отображена информация пользователя и его активные заказы, как изображено на рисунке 5.6.

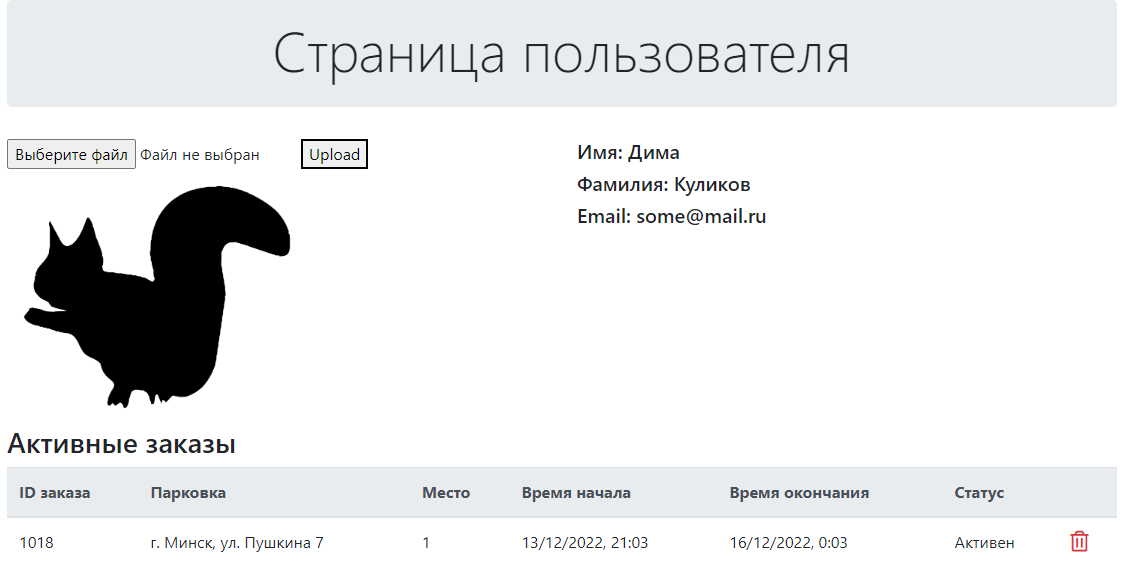


Рисунок 5.6 – Страница пользователя

На странице пользователя можно изменить изображения, для этого необходимо нажать сначала на кнопку «выберите файл» и выбрать нужное изображение, а затем на кнопку «Upload», после чего изображение измениться как показано на рисунке 5.7.

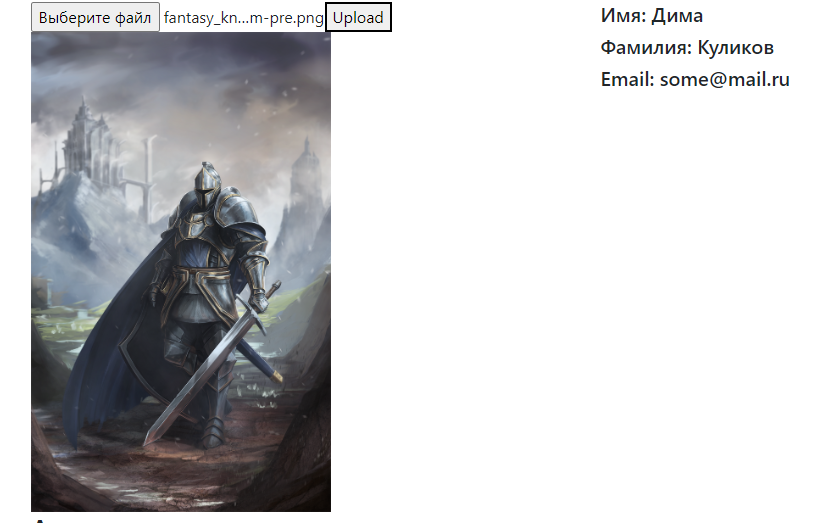


Рисунок 5.7 – Обновление изображения пользователя

При необходимости можно заменить картинку тем же методом.

5.1.2 Administrator

Администратором является пользователь с ролью «admin». Данному пользователю доступен весь функционал приложения. Все ссылки размещены в шапке меню:

* Home;
* Бронирование;
* Админ;
* Добавить парковку;
* Ваши заказы;
* Страница пользователя;
* Выход.

На страницах «Home», «Бронирование», «Ваши заказы», «Страница пользователя» и «Выход» функционал у администратора точно такой же как и у обычного пользователя.

Страница «Админ» представляет собой сборник информации для администратора о всем, что происходит на сайте. Пример страницы администратора изображен на рисунке 5.8.

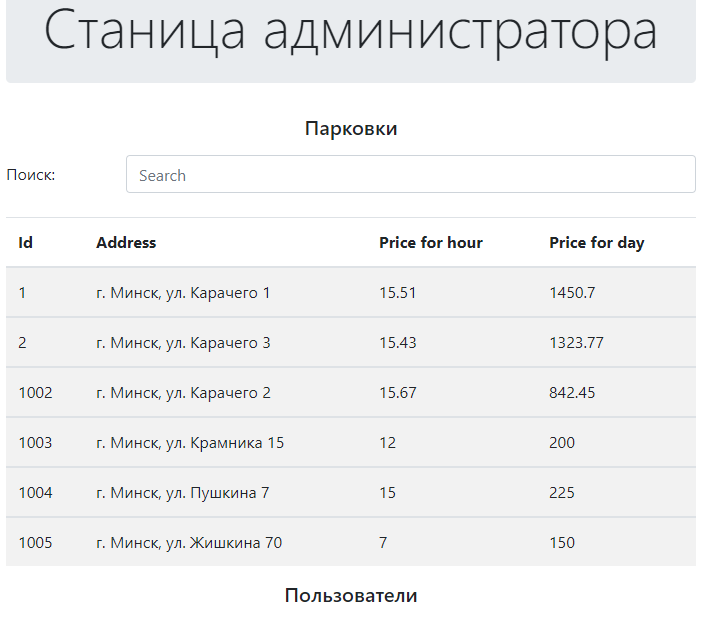


Рисунок 5.8 – Страница «Админ»

На данной странице администратор может выполнять поиск необходимой ему на данный момент информации либо о клиенте, либо о парковке, либо о конкретном заказе.

Страница «Добавить парковку» представляет собой форму для добавления новых парковок, пример страницы изображен на рисунке 5.9.

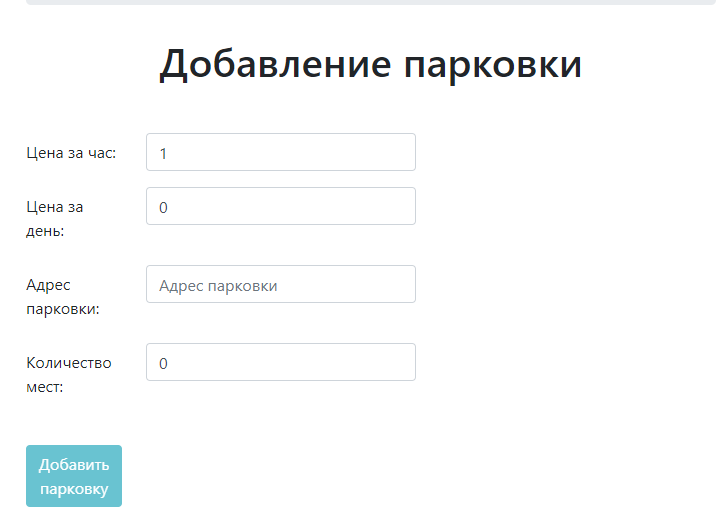


Рисунок 5.9 – Страница «Добавить парковку»

Страница необходима для добавления новых парковок. Если администратор внезапно введет существующую парковку, система заботливо ему сообщит об это и предостережет от ошибки и коллизий.

5.2 Создание конфигурационных файлов Docker

В современном мире существует большое количество сред и оперативных систем, которые имею свою инфраструктуру. Приложения, созданные на одной ОС, вряд ли заработают на другой ОС. Так же, чтобы запустить приложение на той же ОС, на какой создавалось приложение, требуется установить вспомогательные среды разработки, базы данных. Это не эффективно. На мой взгляд, наиболее быстрый и практичный способ является использование Docker.

Docker в настоящее время является одним из самых популярных средств для автоматизации развертывания и управления приложениями в различных средах, например, в операционных системах MacOS, Windows и Linux. Docker одинаково работает независимо от окружающей его среды. Что бы корректно развернуть приложение в контейнере Docker, необходимо написать скрипты развертывания.

Само веб-приложение разработано на базе операционной системы Windows 10. На ней же и будет продолжен процесс настройки администрирования веб-приложения.

Самым удобным способом, благодаря которому можно осуществить поддержку «контейнеризации» приложения, является возможность создания докер-файлов, содержащих в себе базовую конфигурацию. В случае серверной части приложения, Visual Studio имеет встроенную поддержку Docker. Для создания Dockerfile, нужно выбрать «Решение», затем «Добавить» и «Поддержка Docker». По окончанию будет создан Dockerfile со следующим содержимым, приведенным на листинге 5.1.

|  |
| --- |
| FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base  WORKDIR /app  EXPOSE 80  FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build  WORKDIR /src  COPY ["Kursach4/Kursach4.csproj", "Kursach4/"]  RUN dotnet restore "Kursach4/Kursach4.csproj"  COPY . .  WORKDIR "/src/Kursach4"  RUN dotnet build "Kursach4.csproj" -c Release -o /app/build  FROM build AS publish  RUN dotnet publish "Kursach4.csproj" -c Release -o /app/publish  FROM base AS final  WORKDIR /app  COPY --from=publish /app/publish .  ENTRYPOINT ["dotnet", "Kursach4.dll"] |

Листинг 5.1 – Dockerfile серверной части проекта

Здесь можно выделить следующий порядок действий:

* в начале мы качаем образ (image) из DockerHub самой платформы ASP.Net и делаем публикацию папки app на порт 80 и 443 соответственно (для поддержки протоколов HTTP и HTTPS);
* затем мы берем все содержимое из наших исходных файлов, копируем csproj-файл для восстановления зависимостей в папку src/Kursach4. После чего запускается сам процесс восстановления зависимостей командой RUN dotnet restore…;
* после того, как были загружены все пакеты из Nuget-репозитория, происходит публикация приложения в путь /app/publish;
* перед самим этапом запуска приложения мы копируем содержимое в папку app для публикации, работа приложения будет осуществляться на раннее описанных портах;
* и последним этапом остается запуск корневой DLL-библиотеки при помощи .Net CLI.

Следующим шагом будет создание файла Dockerfile для развертки клиентской части приложения. Cтриктуру данного файла, отображена в листинге 5.2.

FROM node:latest as node

RUN npm install -g http-server

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build

EXPOSE 8080

CMD [ "http-server", "dist"]

Листинг 5.2 – Dockerfile клиентской части проекта

Порядок действий, следующий:

* в начале мы загружаем образ из DockerHub платформы NodeJS.
* далее устанавливаем простой HTTP-сервер для статистики;
* делаем каталог «app» текущим рабочим каталогом;
* копируем оба файла «package. json» и «package-lock. json» если он есть;
* устанавливаем зависимости проекта;
* копируем файлы и каталоги проекта в текущий рабочий каталог;
* собираем приложение.

На данном этапе были созданы все необходимые docker-файлы, которые нужны для создания и запуска контейнеров Docker.

5.3 Развертывание приложения в контейнере Docker

Для работы с контейнером Docker нужна сама среда Docker. Для этого нужно с официального сайта скачать установщик Docker Desktop. Процесс установки показан на рисунке 5.18.

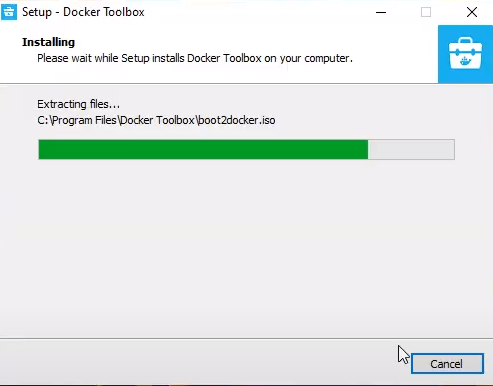


Рисунок 5.18 – Установка Docker Desktop

После установки запускаем приложение. Что бы приложение работало корректно, необходимо убедится в свободном месте на жестком диске и исправном подключении интернета. По умолчанию свободного места должно быть не менее 10гб. Следующим шагом будет зайти в папку с файлом «Dockerfile», открыть консоль или PoweShell в этой папке и прописать команду «docker run --build». Через некоторое время приложение развернется на докере. После чего можно начинать работу с приложением.

5.4 Развертка приложения на ОС.

Если же понадобиться развертывать приложение не в среде Docker, а на реальной операционной системе, то придется установить следующее программное обеспечение для корректной работы самого приложения:

* база данных – сервер Microsoft SQL Server;
* серверная часть – размещение данного приложения на таких серверах, как IIS, Nginx не составит проблем за счет кроссплатформенности и гибкости самой ASP.Net Core платформы;

клиентская часть – размещение не составит также никаких трудностей за счет кроссплатформенности NodeJS. В качестве базового сервера для размещения можно использовать тот же самый Nginx.

# Заключение

В ходе выполнения работы были проанализированы аналоги веб-приложения для бронирования парковочных мест. Выявлены их положительные и отрицательные стороны. Что было учтено при проектировании функционала и дизайна приложения.

В ходе выполнения курсовой работы было разработано и реализованное полноценное, логически завершенное веб-приложение, с учётом всех предоставленных требований, как со стороны используемого инструментария, так и со стороны требований предметной области. Благодаря глубокой проработке доменной области приложения, была спроектированная оптимальная структура данных, решающая поставленную задачу, а также отвечающая как реляционным требованиям хранилища данных, так и бизнес процессам. Также обработаны различные исключительные ситуации; пользовательский интерфейс выполнен в интуитивно понятной форме. В конечном итоге, веб-приложение было развёрнуто в Docker контейнере и показало полную функциональную и структурную готовность обслуживать реальных пользователей.

Объективно важным моментом в разработке подобного рода систем является и обработка различных возможных ошибочных сценариев, которые могут возникнуть при взаимодействии пользователя с системой. В задачу разработчика также входит не только предвидение и обработка этих ситуаций, но и разумное ограничение действий, которые может выполнять пользователь. Эти концепции были также соблюдены и внедрены в ходе разработки представленной курсовой работы.

Для реализации серверной части использовалась платформа ASP.NET Core. ASP.NET Core может работать поверх кроссплатформенной среды .NET Core, которая может быть развернута на основных популярных операционных системах: Windows, Mac OS, Linux. В результате была разработан веб-сервис по архитектуре REST, который обрабатывает клиентские запросы и делает запросы к базе данных.

Была разработана реляционная база данных для веб-приложения, использующая экземпляр Microsoft SQL Server, имеющая в своем наборе 4 таблиц.

С помощью фреймворка Angular была разработана клиентская часть приложения.

Были созданы дополнительные конфигурационные файлы, позволяющие быстро и легко администрировать, расширять функционал приложения.

Заключение: создание веб-приложений для больших систем является крайне комплексной задачей и ресурсами одного человека обойтись здесь крайне сложно, так как он должен сочетать в себе и прагматизм разработчика баз данных и слоя бизнес-логики, так гибкость понимания юзабилити интерфейсов пользователя, так и глубокое понимание процессов, протекающих в системе, с точки зрения конечного пользователя.

# Список литературы

1. Свободная энциклопедия Википедия [Электронный ресурс] /CRUD – Режим доступа: https://ru.wikipedia.org/wiki/CRUD ­– Дата доступа: 10.12.2022.
2. ASP .Net Core [Электронный ресурс] – Режим доступа: https://metanit.com/sharp/aspnet5/ ­– Дата доступа: 10.12.2022.
3. Microsoft SQL Server [Электронный ресурс] / Microsoft SQL Server – Режим доступа: https://www.microsoft.com/ru-ru/sql-server/sql-server-2019 ­– Дата доступа: 10.12.2022.
4. Angular [Электронный ресурс] / Angular – Режим доступа: https://angular.io/ – Дата доступа: 10.12.2022.
5. Angular [Электронный ресурс] / Введение в Angular – Режим доступа: https://metanit.com/web/angular2/1.1.php – Дата доступа: 10.12.2022.

# Приложение A

|  |
| --- |
| public class User  {  [Key]  public int Id { get; set; }  [Column(TypeName = "nvarchar(MAX)")]  public string Email { get; set; }  [Column(TypeName = "nvarchar(MAX)")]  public string Firstname { get; set; }  [Column(TypeName = "nvarchar(MAX)")]  public string Lastname { get; set; }  [Column(TypeName = "varbinary(MAX)")]  public byte[] Image { get; set; }    [Column(TypeName = "nvarchar(MAX)")]  public string Password { get; set; }  [Column(TypeName = "nvarchar(MAX)")]  public string Role { get; set; }  [JsonIgnore()]  public ICollection<Order> Orders { get; set; }  public User()  {  Orders = new List<Order>();  }  }  public class Slot  {  [Key]  public int Id { get; set; }  [Column(TypeName = "int")]  public int Number { get; set; }  [Column(TypeName = "int")]  public int? ParkingId { get; set; }    public Parking Parking { get; set; }  [Column(TypeName = "bit")]  public bool IsOccupied { get; set; }  [JsonIgnore()]  public ICollection<Order> Orders { get; set; }  public Slot()  {  Orders = new List<Order>();  }  }  public class Order  {  [Key]  public int Id { get; set; }  [Column(TypeName = "int")]  public int? UserId { get; set; }  public User User { get; set; }  [Column(TypeName = "int")]  public int? SlotId { get; set; }  public Slot Slot { get; set; }  [Column(TypeName = "datetime")]  public DateTime StartDate { get; set; }  [Column(TypeName = "datetime")]  public DateTime EndDate { get; set; }  [Column(TypeName = "float")]  public double Price { get; set; }  [Column(TypeName = "bit")]  public bool IsComplete { get; set; }  }  public class Parking  {  [Key]  public int Id { get; set; }  [Column(TypeName = "nvarchar(MAX)")]  public string Address { get; set; }  [Column(TypeName = "float")]  public double PriceHour { get; set; }  [Column(TypeName = "float")]  public double PriceDay { get; set; }  [JsonIgnore()]  public ICollection<Slot> Slots { get; set; }  public Parking()  {  Slots = new List<Slot>();  }  } |

# Приложение Б

|  |
| --- |
| [Route("api/[controller]")]  [ApiController]  public class UsersController : ControllerBase  {  private readonly ParkingContext \_context;  public UsersController(ParkingContext context)  {  \_context = context;  }  [HttpPost("login")]  public IActionResult Login([FromBody] User user)  {  if (user is null)  {  return BadRequest("Invalid client request");  }  var dbUser = \_context.Users.FirstOrDefault(x => x.Email == user.Email.ToLower());  var pas = PasswordHelper.GetMD5Hash(user.Password);  if (dbUser != null && pas == dbUser.Password)  {  var secretKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("superSecretKey@345"));  var signinCredentials = new SigningCredentials(secretKey, SecurityAlgorithms.HmacSha256);  var tokeOptions = new JwtSecurityToken(  issuer: "https://localhost:30923",  audience: "https://localhost:30923",  // claims: new List<>(),  expires: DateTime.Now.AddHours(2),  signingCredentials: signinCredentials  );  var tokenString = new JwtSecurityTokenHandler().WriteToken(tokeOptions);  return Ok(new AuthenticatedResponse { Token = tokenString, Id = dbUser.Id, User = dbUser.Firstname, Role = dbUser.Role });  }  return Unauthorized();  }  [HttpPost("registration")]  public async Task<IActionResult> RegisterUser([FromBody] User userForRegistration)  {  var dbUser = \_context.Users.FirstOrDefault(x => x.Email == userForRegistration.Email.ToLower() );  if (dbUser == null)  {  userForRegistration.Role = "Client";  userForRegistration.Password = PasswordHelper.GetMD5Hash(userForRegistration.Password);  \_context.Users.Add(userForRegistration);  await \_context.SaveChangesAsync();  return StatusCode(201);  }  return BadRequest("Email exist");  }  [Authorize]  [HttpGet]  public async Task<ActionResult<IEnumerable<User>>> GetUsers()  {  return await \_context.Users.ToListAsync();  }  [Authorize]  [HttpGet("{id}")]  public async Task<ActionResult<ViewUser>> GetUser(int id)  {  var user = await \_context.Users.FindAsync(id);  if (user == null)  {  return NotFound();  }  ViewUser viewUser = new ViewUser();  viewUser.Email = user.Email;  viewUser.Firstname = user.Firstname;  viewUser.Lastname = user.Lastname;  viewUser.Image = user.Image;  return viewUser;  }  [Authorize]  [HttpPut("{id}")]  public async Task<IActionResult> PutUser(int id, IFormFile img)  {  var user = await \_context.Users.FindAsync(id);  if (user == null)  {  return BadRequest();  }  var file = HttpContext.Request.Form.Files.Count > 0 ? HttpContext.Request.Form.Files[0] : null;  if (file != null && file.Length > 0)  {  using (var ms = new MemoryStream())  {  await file.CopyToAsync(ms);  var fileBytes = ms.ToArray();  user.Image = fileBytes;  \_context.Entry(user).State = EntityState.Modified;  try  {  await \_context.SaveChangesAsync();  }  catch (DbUpdateConcurrencyException)  {  if (!UserExists(id))  {  return NotFound();  }  else  {  throw;  }  }  }  return Ok(user.Image);  }  try  {  await \_context.SaveChangesAsync();  }  catch (DbUpdateConcurrencyException)  {  if (!UserExists(id))  {  return NotFound();  }  else  {  throw;  }  }  return NoContent();  }  [Authorize]  [HttpPost]  public async Task<ActionResult<User>> PostUser(User user)  {  \_context.Users.Add(user);  await \_context.SaveChangesAsync();  return CreatedAtAction("GetUser", new { id = user.Id }, user);  }  [Authorize]  [HttpDelete("{id}")]  public async Task<IActionResult> DeleteUser(int id)  {  var user = await \_context.Users.FindAsync(id);  if (user == null)  {  return NotFound();  }  \_context.Users.Remove(user);  await \_context.SaveChangesAsync();  return NoContent();  }  private bool UserExists(int id)  {  return \_context.Users.Any(e => e.Id == id);  }  } |

# Приложение В

|  |
| --- |
| migrationBuilder.CreateTable(  name: "Parking",  columns: table => new  {  Id = table.Column<int>(type: "int", nullable: false)  .Annotation("SqlServer:Identity", "1, 1"),  Address = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  PriceHour = table.Column<double>(type: "float", nullable: false),  PriceDay = table.Column<double>(type: "float", nullable: false)  },  constraints: table =>  {  table.PrimaryKey("PK\_Parking", x => x.Id);  });  migrationBuilder.CreateTable(  name: "Users",  columns: table => new  {  Id = table.Column<int>(type: "int", nullable: false)  .Annotation("SqlServer:Identity", "1, 1"),  Email = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Firstname = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Lastname = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Image = table.Column<byte[]>(type: "varbinary(MAX)", nullable: true),  Password = table.Column<string>(type: "nvarchar(MAX)", nullable: true),  Role = table.Column<string>(type: "nvarchar(MAX)", nullable: true)  },  constraints: table =>  {  table.PrimaryKey("PK\_Users", x => x.Id);  });  migrationBuilder.CreateTable(  name: "Slots",  columns: table => new  {  Id = table.Column<int>(type: "int", nullable: false)  .Annotation("SqlServer:Identity", "1, 1"),  Number = table.Column<int>(type: "int", nullable: false),  ParkingId = table.Column<int>(type: "int", nullable: true),  IsOccupied = table.Column<bool>(type: "bit", nullable: false)  },  constraints: table =>  {  table.PrimaryKey("PK\_Slots", x => x.Id);  table.ForeignKey(  name: "FK\_Slots\_Parking\_ParkingId",  column: x => x.ParkingId,  principalTable: "Parking",  principalColumn: "Id",  onDelete: ReferentialAction.Restrict);  });  migrationBuilder.CreateTable(  name: "Orders",  columns: table => new  {  Id = table.Column<int>(type: "int", nullable: false)  .Annotation("SqlServer:Identity", "1, 1"),  UserId = table.Column<int>(type: "int", nullable: true),  SlotId = table.Column<int>(type: "int", nullable: true),  StartDate = table.Column<DateTime>(type: "datetime", nullable: false),  EndDate = table.Column<DateTime>(type: "datetime", nullable: false),  Price = table.Column<double>(type: "float", nullable: false),  IsComplete = table.Column<bool>(type: "bit", nullable: false)  },  constraints: table =>  {  table.PrimaryKey("PK\_Orders", x => x.Id);  table.ForeignKey(  name: "FK\_Orders\_Slots\_SlotId",  column: x => x.SlotId,  principalTable: "Slots",  principalColumn: "Id",  onDelete: ReferentialAction.Restrict);  table.ForeignKey(  name: "FK\_Orders\_Users\_UserId",  column: x => x.UserId,  principalTable: "Users",  principalColumn: "Id",  onDelete: ReferentialAction.Restrict);  });  migrationBuilder.CreateIndex(  name: "IX\_Orders\_SlotId",  table: "Orders",  column: "SlotId");  migrationBuilder.CreateIndex(  name: "IX\_Orders\_UserId",  table: "Orders",  column: "UserId");  migrationBuilder.CreateIndex(  name: "IX\_Slots\_ParkingId",  table: "Slots",  column: "ParkingId"); |