МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

Гомельский государственный технический университет имени П.О.Сухого

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

специальность 1-40 05 01-01 Информационные системы и технологии

(в проектировании и производстве)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

по дисциплине «Разработка приложений баз данных для информационных систем»

на тему «Разработка *web*-приложения баз данных «Мероприятия города»»

Исполнитель: студент гр. ИТП-31

Шинкарук В.Г.

Руководитель: ассистент

Малиновский И.Л.

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

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

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

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

Подписи членов комиссии

по защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2024

**СОДЕРЖАНИЕ**

Введение 3

1 Логическая и физическая структура базы данных *web*-приложения   
мероприятий города 5

1.1 Информационно-логическая модель и её структура 5

1.2 Связи между сущностями и атрибуты базы данных 6

1.3 Соответствие нормализации и физическая структура базы данных 8

2 Аппаратное и программное обеспечение информационной системы   
приложения мероприятий города 11

2.1 Основные характеристики компьютеров для работы с системой 11

2.2 Требования к программному обеспечению и настройка приложения 12

3 Структура приложения баз данных мероприятий города 14

3.1 Архитектура программного комплекса 14

3.2 Инфраструктура проекта 16

3.3 *Web*-интерфейс и модульные тесты 17

4 Руководство пользователя 21

4.1 Вход в приложение 21

4.2 Перечень и описание всех выполняемых функций обработки данных 22

4.3 Рекомендации по освоению и эксплуатации приложения 25

5 Руководство программиста 27

Заключение 29

Список использованных источников 30

Приложение А Чертёж структуры *web*-приложения 31

Приложение Б Листинг программы 32

Приложение В Листинг модульных тестов 79

**ВВЕДЕНИЕ**

В эру глобального развития информационных технологий, когда цифровизация проникает во все аспекты жизни, программные решения становятся неотъемлемой частью эффективного функционирования любой сферы деятельности. Современные города характеризуются активной культурной и общественной жизнью, предполагающей организацию множества мероприятий: от массовых городских праздников до небольших тематических событий.

Заказ билетов на городские мероприятия является неотъемлемой частью современных городов, обеспечивая удобство и скорость взаимодействия между организаторами событий и теми, кто напрямую связаны с посещением их. Процесс проведения мероприятия в городе требует строгого планирования и нуждается в точном учёте количества участников, удобном бронировании мест и лёгком просмотре всей информации о множестве событий в городе.

Современный пользователь, привыкший к доступности информации в реальном времени, ожидает, что приложение позволит быстро находить, просматривать и бронировать мероприятия, минуя длительные телефонные звонки или посещение точек продажи, таких как касса. Для организаторов такая платформа позволяет оптимизировать внутренние процессы, сократить затраты времени и избежать лишних административных расходов. Полный доступ администратора к базе данных обеспечивает возможность не только быстро и эффективно заносить данные о новых событиях в базу данных, но и, в случае резкой необходимости, без дополнительных трат отредактировать уже занесенную ранее информацию о всех мероприятиях города.

Создание специализированных информационных систем актуально за счёт удобного способа администрирования городской событийной программы, позволяя структурировать информацию и быстро получать доступ к актуальным данным о самих мероприятиях, местах их проведения, а также различную дополнительную, но в то же время крайне важную информацию о клиентах и организаторах. Это позволяет решать важные в современном мире задачи, такие как регистрация пользователей, бронирование билетов, управление информацией о событиях и организаторах в одном приложении, без дополнительного оборудования.

Проблема автоматизации управления данными о мероприятиях и взаимодействия с клиентами часто связана с необходимостью создания единой системы с различной информацией. Уже существующие системы могут быть ограничены по функционалу или требовать значительных ресурсов на внедрение и поддержку. Данное приложение направлено на устранение данных проблем за счёт использования современных технологий – мощного языка программирования *C#* и его фреймворков.

Цель проекта заключается в создании интуитивно понятного и функционального веб-приложения, которое не только позволяет пользователям удобно находить мероприятия и заказывать на них билеты, но и представляет широкий инструментарий для администраторов, включающий управление всеми событиями и билетными заказами. В результате внедрения такой системы в рабочую среду жизни организаторы смогут экономить время и ресурсы, а пользователи – сократить затраты на поиск и заказ билетов.

При разработке системы была выбрана архитектура *ASP.NET MVC (Model-View-Contoller)* на платформе *.NET Core*. Такая архитектура обеспечивает чёткое и логичное разделение всех компонентов программы, что особенно важно для сложных систем с большим количество сущностей и серьёзной заполненностью базы данных. Кроме того, использование *.NET Core* позволяет запускать систему как на собственных серверах, так и на удалённом хостинге.

В качестве системы управления базами данными была выбрана платформа *MS SQL Server*, совместимая с *Entity Framework*, что упрощает взаимодействие между приложением и базой данных. Она удовлетворяет всем требованиям современной информационной безопасности, так как обеспечивает целостность данных и защиту персональной информации заказчиков и организаторов.

За счёт удобного разделения данных в приложении на различные вкладки, представляющие собой таблицы из базы данных, обеспечена эффективная навигация. Помимо этого, реализованы фильтрации, делающие нахождение краткой информации в большом объеме данных значительно проще и эффективнее. Реляционная база данных хранит информацию в стандартном табличном виде и не требует дополнительных функций для получения данных.

Приложение разработано с учётом возможностей расширения и модернизации в будущем. Открытый код, написанный на платформе *.NET Core*, позволяет адаптировать приложение под потребности пользователей и интегрировать новые модули, включая различный функционал аналитики, интеграции с внешними системами, оплатами и другими возможностями, которые могут потребоваться для расширения приложения в будущем.

Таким образом, разработка специализированного программного продукта для управления базой данных мероприятий города является не только актуальной, но и необходимой задачей, решение которой позволит повысить качество управления информацией и обеспечить безопасное использование её без потери данных.

**1 ЛОГИЧЕСКАЯ И ФИЗИЧЕСКАЯ СТРУКТУРА БАЗЫ ДАННЫХ**

***WEB*-ПРИЛОЖЕНИЯ МЕРОПРИЯТИЙ ГОРОДА**

* 1. **Информационно-логическая модель и её структура**

Информационно-логическая модель представляет собой абстрактное представление данных, описывающих объекты и взаимосвязи в рамках конкретной области и является одних из ключевых элементов проектирования информационной системы. Она делает данные предметной области структурируемыми, используя поддержку и оптимизацию процессов обработки информации. Помимо основных действий, информационно-логическая модель обеспечивает целостность и эффективность взаимодействия между разными частями системы. Она охватывает отношения между объектами управления мероприятиями, такими как место проведения, заказчики, организаторы, события и заказы билетов.

Использование информационно-логической модели имеет ряд преимуществ, которые проявляются в реализации сущностей и связей между ними:

– целостность данных обеспечивается, используя обязательные связи между объектами, например каждое мероприятие должно иметь организатора и место проведения. Тем самым ограничивается вероятность появления некорректных или недостаточно информативных данных [1];

– удобный анализ данных получается с помощью особого отражения предметной области. Информация легко извлекается о всех мероприятиях, отслеживать популярные места проведения, какие мероприятия являются наиболее предпочтительными для клиентов и ещё много различной информации;

– высокая производительность благодаря нормализации данных. Нормализация позволяет избегать избыточные и дублирующиеся данные. В проекте это реализовано с помощью выделения отдельных таблиц для информации, например *Places*.

– безопасность данных. Разделение данных по сущностям подкрепляется различными правами доступа. Администратор может контролировать доступ к каждой таблице, а обычные пользователи не имеют доступ к редактированию пользователей.

Сущности (*Entities*) – ключевые объекты, характеризующие основную информацию в предметной области. Основными сущности проекта являются:

– *Places* (Места проведения мероприятий) – сущность, представляющая информацию о местах проведения различных мероприятий. Начиная от самых простых, заканчивая главными мероприятиями города. Каждое место имеет уникальные идентификаторы и атрибуты, описывающие название и геолокацию;

– *Organizers* (Организаторы) – отвечает за хранение информации об организаторах мероприятий. Это могут быть отдельные физические лица или организации, которые управляют проведением событий;

– *Customers* (Заказчики) – сущность, представляющая клиентов. Клиенты покупают билеты на мероприятия, что делает их одной из ключевых частей всей системы. Содержит атрибуты, позволяющие однозначно идентифицировать заказчиков, включая их паспортные данные;

– *Events* (Мероприятия) – описывает проводимые мероприятия. Имеет серьезную структуру, так как напрямую связана с другими таблицами базы данных. Каждый объект «Мероприятие» содержит информацию о месте, времени, стоимости и количестве доступных билетов, имея связь с таблицей организаторов;

– *TicketOrders* (Заказы билетов) – сущность, представляющая заказы билетов, оформляемые заказчиками. Также, как и *Events*, она связана с другими таблицами базы данных. Хранит данные о количестве заказов, заказанных на определённое мероприятие, и дату оформления заказа.

На рисунке 1.1 представлена общая схема базы данных.

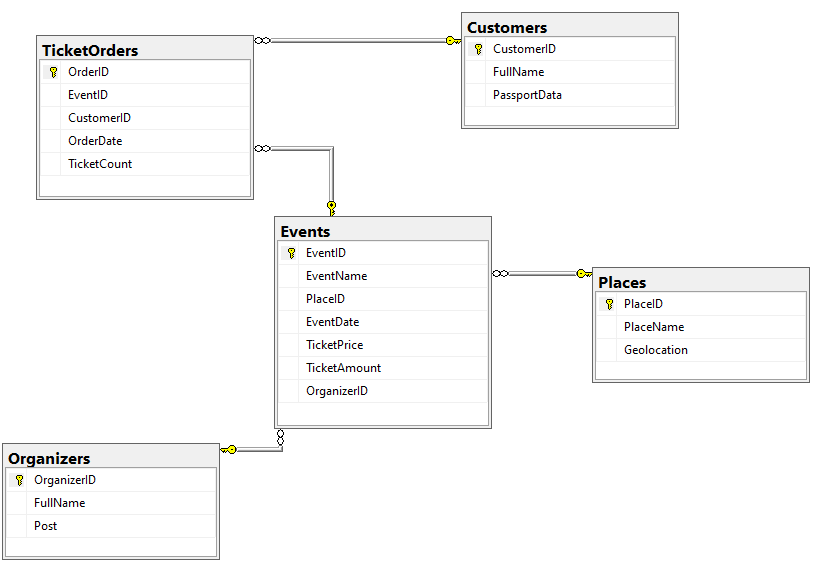


Рисунок 1.1 – Общая схема базы данных

**1.2 Связи между сущностями и атрибуты базы данных**

Связи между сущностями (*Relationships*) – определяют взаимосвязь между объектами модели с помощью ключевых полей. Они позволяют эффективно хранить и обрабатывать связанные данные.

– связь между *Places* и *Events*: каждое мероприятия связано с конкретным местом проведения. Это позволяет организовать расписание событий в различным местах города и эффективно управлять ими. Связь «многие к одному» оптимизирует процесс управления местами и распределения мероприятий по ним за счёт того, что много мероприятий могут проводиться в одном месте;

– связь между *Organizes* и *Events*: каждое мероприятие имеет организатора, ответственного за его проведение. Тут также реализована связь «многие к одному», где много мероприятий могут быть организованы одним организатором;

– связь между *Customers* и *TickerOrders*: все заказы билетов привязаны к конкретному заказчику, что упрощает анализ проведения клиентов, отслеживать активность заказчиков и формировать персонализированные приложения;

– связь между *Events* и *TickerOrders*: заказ билетов связан с определённым мероприятием. Это позволяет системе управлять доступностью билетов и отслеживать сколько билетов заказано на определённое мероприятие. Благодаря этому, приложение эффективно в современном мире, где нужно точно знать сколько человек может посетить мероприятие.

Каждая таблица имеет свои поля. В каждом из них есть свое ключевое поле, представляющее уникальный идентификатор.

Таблица мест (*Places*) имеет такие поля:

– *PlaceID* – целочисленный тип (*int*), представляющий уникальное место;

– *PlaceName* – представляет название места, которое может содержать не более 50 символов. Имеет тип *nvarchar(50)*;

– *Geolocation* – геолокационные данные, указывающие, где находится место. Геолокация может быть длинной, поэтому имеет ограничение 100. Имеет, соответственно, тип *nvarchar(100).*

Таблица организаторов (*Organizers*) содержит:

– *OrganizerID* – уникальный идентификатор организатора. Целочисленный(int);

– *FullName* – полное имя организатора, так называемое ФИО. Тип *nvarchar(100)*;

– *Post* – должность организатора. Тип данных: *nvarchar(50)*.

Таблица заказчиков (*Customers*) схожа с другими таблицами на стороне отношения «один» и содержит следующие поля:

– *CustomerID* – идентификатор каждого заказчика. Тип: *int*;

– *FullName* – полное имя заказчика, так называемое ФИО. Тип *nvarchar(50);*

– *PassportData* – паспортные данные заказчика. Тип данных: *nvarchar(100)*.

Первой из таблиц, получающей данные из других таблиц, является Events и имеет следующую структуру:

– *EventID* – уникальный идентификатор мероприятия. Целочисленный (int);

– *EventName* – название мероприятия. Ограничение на 100 символов *nvarchar(100)*;

– *PlaceID* – внешний ключ, указывающий на место проведения мероприятия. Образует связь «многие к одному»;

– *EventDate* – дата проведения мероприятия. Имеет тип *date*;

– *TicketPrice* – стоимость одного билета на мероприятие. Тип данных: *real*, характерный для чисел с плавающей запятой;

– *TicketAmount* – общее количество билетов, доступных для продажи. Целочисленное (*int*);

– *OrganizerID* – внешний ключ, указывающий на организатора мероприятия. Образует связь «многие к одному».

Таблица *TicketOrders* также получает данные с других таблиц и содержит следующие атрибуты:

– *OrderID* – уникальное значение для каждого заказа билетов;

– *EventID* – внешний ключ, указывающий на мероприятие. Образует связь «многие к одному»;

– *CustomerID* – внешний ключ, указывающий на заказчика;

– *OrderDate* – дата покупки билетов. Тип данных: *date*;

– *TicketCount* – количество приобретенных билетов. Целочисленный.

* 1. **Соответствие нормализации и физическая структура базы данных**

Для обеспечения целостности и согласованности данных в проекте база данных спроектирована с учётом нормализации. Она минимизирует избыточность данных и защищает он неожиданных результатов при обновлении данных. Все таблицы удовлетворяют требованиям до третьей нормальной формы [2]. Каждая из них имеет свои требования, выполнение которых описано ниже.

Первая нормальная форма (1НФ).

Требует, чтобы все атрибуты были атомарными. Это значит, что каждое поле в таблице должно содержать неделимое значение, и в каждой записи должен быть уникальный набор атрибутов.

Например, в таблице *Places* поле *PlaceName* является атомарным, поскольку хранит только название места, а *Geolocation* является неделимы атрибутом, отвечающим только за геолокацию места.

В таблице *Customers* также нет повторяющихся групп данных. Каждое поле содержит только одно значение, например *FullName* и *PassportData*. Это же характерно и для всех остальных таблиц базы данных.

Первая нормальная форма обеспечивает удобство в работе с данными и позволяет избежать неоднозначности и избыточности. Каждое поле представляет собой одно значение и каждая таблица имеет уникальный идентификатор (первичный ключ).

Вторая нормальная форма (2НФ).

Требует, чтобы база данных находилась в первой нормальной форме и все неключевые атрибуты были функционально зависимы от первичного ключа. Каждый неключевой атрибут должен зависеть от всего ключа и только от него, исключая частичные зависимости.

Например, в таблице Events первичный ключ *EventID* однозначно идентифицирует каждое мероприятие. А также другие атрибуты, такие как *EventName*, *PlaceID*, *EventDate*, *TicketPrice*, *TicketAmount*, *OrganizerID*. Все они зависят именно от *EventID*.

В таблице *TicketOrders* поле *OrderID* является первичным ключом и идентифицирует каждую запись заказа билета. Атрибуты *EventID*, *CustomerID*, *TicketCount* зависят только от *OrderID*.

Поле *PlaceID* является тем, от кого зависят *PlaceName* и *Geolocation*.

Каждая таблица проекта удовлетворяет второй нормальной форме, так как все атрибуты функционально зависят от первичного ключа, без частичных зависимостей.

Третья нормальная форма (3НФ).

Требует, чтобы таблица находилась во второй нормальной форме, и чтобы все неключевые атрибуты зависели только от первичного ключа и не имели зависимостей между неключевыми полями, которые обычно называют транзитивными [3].

Например, в таблице Events атрибуты *EventName*, *EventDate*, *TicketPrice*, *TicketAmount*, *OrganizerID*, *PlaceID* зависят от *EventID* и не зависят друг от друга. *TicketPrice* и *TicketAmount* не зависят от *EventDate* или *OrganizerID*, и не один из этих атрибутов не может быть вычислен но основе других неключевых.

В таблице Customers не существует транзитивных зависимостей, так как *FullName* и *PassportData* зависят исключительно от *CustomerID*.

Это же характерно и для всех остальных таблиц базы данных. Каждая таблица содержит только те атрибуты, которые описывают именно эту сущность.

Помимо удовлетворения трём нормальным формам, база данных также соответствует требованиям Бойс-Коддовой нормальной формы (*BCNF*) – расширение 3НФ.

Оно требует, чтобы каждый детерминант был кандидатом на ключ. Если неключевой атрибут может определять другой атрибут, то этот детерминант также должен быть уникальным ключом. В базе структура построена так, что для каждой таблицы первичный ключ однозначно определяет каждое поле.

Например, в *Place* первичный ключ *PlaceID* является детерминантом для всех атрибутов. Также и с остальными таблицами.

В результате, все таблицы базы данных нормализованы до третьей нормальной формы и удовлетворяют Бойс-Коддовой нормальной форме, обеспечивая целостность данных и минимизацию избыточности.

Физическая структура базы данных, работающая под управлением системы управления базами данных *Microsoft* *SQL* *Server*, играет ключевую роль в обеспечении производительности и надежности системы. *SQL* *Server* имеет большой функционал, позволяя решать большинство задач: разделение данных на файлы, организация индексов, размещение файлов на дисках. Эти действия направлены на достижение высокой скорости выполнения запросов и минимизации времени отклика при записи и чтении [4].

В *Microsoft* *SQL* *Server* любая база данных, в том числе и система управления событиями, состоит из следующих типов файлов:

Первичный файл (.*mdf*) – является основным и обязательным файлом базы данных, где хранится вся система, метаданные, информация о структуре базы данных, включая таблицы, индексы и схемы. В проекте файл .*mdf* хранит все основные данные, включая записи о местах проведения мероприятий, организаторах, клиентах, событиях и заказах билетов. Располагается в защищенном разделе сервера, где данные имеют постоянный доступ для обработки запросов. Объём файла динамически увеличивается по мере добавления данных в базу.

Вторичные файлы данных (.*ndf*) – дополнительные файлы. Используются при необходимости распределить базу на несколько файлов для оптимизации производительности. Структура базы не требует деления данных по физическим дискам.

Файл журналов транзакций (.*ldf*) – файл, содержащий информацию о всех изменениях, происходящих в базе данных. Он необходим для целостности данных при изменениях и восстановления в случае сбоев. Он фиксирует каждую операцию, совершаемую с базой данных, включая изменения и запросы на изменение данных. Журнал транзакций содержит все действия пользователей, например, добавление нового события, редактирование заказа билетов и позволяет откатывать изменения при сбоях. Рекомендуется размещать его на отдельном диске для снижения нагрузки на основной диск.

Файлы базы данных (.*mdf*, .*ldf*) в проекте размещены в каталоге, доступном для удалённого сервера *SQL* *Server*. Это важно, так как файл .*mdf* содержит критически важную информацию, необходимую для работы приложения.

Каждая таблица в базе данных занимает определенный объём в основном файле данных (.*mdf*), зависящий от структуры полей и количества записей. Таблицы, типы данных и свойства полей уже описаны выше, и они содержат поля различных типов, такие как *int*, *nvarchar*, *real*, date. Каждое из них занимает разное количество байт.

– *int* занимает 4 байта на каждую запись;

– *nvarchar* занимает от 50 до 100 символов, где каждый символ занимает 2 байта;

– *date* в таблице *Events* занимает 3 байта, что экономично и позволяет хранить дату без времени;

– *real* занимает 4 байта и применяется в таблице *Events* для хранения стоимости билета.

**2 АППАРАТНОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ**

**ИНФОРМАЦИОННОЙ СИСТЕМЫ ПРИЛОЖЕНИЯ**

**МЕРОПРИЯТИЙ ГОРОДА**

* 1. **Основные характеристики компьютеров для работы с системой**

Для корректной работы информационной системы необходимо обеспечить подходящее для неё аппаратное и программное обеспечение. Оно должно идеально подходить как для серверной, так и для клиентской части. От грамотного выбора и правильной настройки оборудования и программного обеспечения зависит стабильность, надёжность и производительность всей системы. При тщательном отборе появляется гарантия, что система будет работать с минимальными задержками, а нагрузка не будет вызывать непредвиденных результатов при большом количестве пользователей, работающих одновременно.

Серверное оборудование должно соответствовать требованиям системы, поскольку именно серверы отвечают за обработку, хранение данных, а также доступ ко всем возможностям приложения. Чем мощнее сервер, тем выше ключевые показатели системы: скорость получения и обработки данных, количество одновременно поддерживаемых пользователей, устойчивость к объёмным процессам.

Со стороны клиента аппаратные требования хоть и значительно ниже, но тоже требуют устройств, способных обрабатывать данные без серьезных замедлений. В целом, пользователи могут использовать различные устройства для доступа к функционалу, не опасаясь недостатка вычислительных ресурсов. Это крайне положительный момент, так как это делает систему доступной для широкой аудитории.

Для обеспечения стабильной и надежной работы системы необходимо использовать сервер, имеющий следующие характеристики:

Процессор:

– многоядерный (минимум 4 ядра);

– с тактовой частотой от 2,4 ГГц и выше.

Это связано с тем, что сервер будет обрабатывать многочисленные запросы от пользователей и выполнять операции с базой данных. Оптимально подойдет сервер с процессором от *Intel* или *AMD*, например *Intel* *XEON* или *AMD* *EPYC*. Такие процессоры позволяет увеличить количество пользователей и объём обрабатываемых данных, с которыми система способна работать.

Оперативная память:

– минимум 8 ГБ для запуска;

– минимум 16 ГБ для стабильной работы.

Оперативная память необходима для эффективного выполнения *SQL*-запросов, работы *web*-сервера и поддержки многопоточности при одновременном обращении многих пользователей к системе. Для запуска системы, но с возможными замедлениями при обработке большого количества запросов, подойдет оперативная память такая, как *Kingston* *DDR4* *8GB* *2666MHz* или *Crucial* *DDR4* *8GB* *24000MHz*. Для стабильной работы системы при большой нагрузке отлично подойдёт *Corsair* *Vengeance* *LPX* *16GB* *3200MHz* или *G.SKILL* *Ripjaws* *V* *16GB* *3000MHz*.

Жёсткий диск (*HDD*/*SSD*):

– объём от 500 ГБ;

– скорость от 7200 об/мин.

*SSD* обеспечивает высокую скорость чтения и записи данных. Это позволяет значительно снизить время отклика системы при обработке данных. Для запуска и стабильной работы подойдёт такой диск, как *Western* *Digital* *Blue* *1TB* *HDD*.

Сетевое оборудование:

– канал со скоростью не менее 100 Мбит/с для внешних соединений;

– канал со скоростью до 1 Гбит/с для внутренних соединений.

Высокоскоростное сетевое оборудование делает обработку запросов оперативной, а передачу между сервером и клиентом значительно быстрее. Задержки снижаются при качественном сетевом оборудовании, как и повышается производительность. Подойдет маршрутизатор *TP*-*Link* *Archer* *C7*, имеющий поддержку стандартов *Wi*-*Fi* *802*.*11ac*.

Хоть на стороне клиента требования к программному обеспечению минимальны, важно, чтобы оборудование соответствовало следующим минимальным требованиям:

Процессор:

– двухъядерный;

– с тактовой частотой от 2 ГГц и выше.

Оперативная память:

– минимум 4 ГБ для запуска.

Жёсткий диск (*HDD*/*SSD*):

– 10 ГБ свободного пространства.

Сетевое оборудование:

– подключение к интернету со скоростью не менее 10 Мбит/с.

Надёжное сетевое оборудование позволяет системе адаптироваться к увеличению числа пользователей и объёма данных, обеспечивая стабильную работу.

* 1. **Требования к программному обеспечению и настройка   
      приложения**

Программное обеспечение является основой для всех процессов системы. Архитектура, на которой строится информационная система, состоит из операционной системы сервера, системы управления базами данных, *web*-сервера и фреймворков. Правильный выбор программного обеспечения является ключевым для повышения производительности и безопасности всей системы, улучшает гибкость в расширении функционала.

Серверное программное обеспечение определяет возможности работы приложения. Рекомендуется использовать следующее обеспечение:

– Операционная система *Windows* *Server* 2019 или позднее. Для *Unix*-серверов подойдет *Ubuntu* *20*.*04* *LTS*. Данные операционные системы поддерживают многозадачность, обработку сетевых запросов и взаимодействие с базой данных;

– Система управления базами данных (СУБД) *Microsoft* *SQL* *Server* 2019 или позднее. База данных приложения поддерживает именно этот вид СУБД и позволяет хранить и обрабатывать данные в реляционной базе данных с высокими степенями надёжности и производительности;

– *web*-сервер *Internet* *Information* *Services* (*IIS*) для *Windows* *Server* или на *Unix*-серверах сервер *Nginx* в связке с *Kestrel*. Оба предоставляют полный набор функций для управления *ASP*.*NET* *MVC* приложением, позволяют настраивать права доступа и контролировать загрузку;

– фреймворк .*NET* *3.1* или выше, имеющий кроссплатформенность. Она позволяет успешно запускать приложение как на *Windows*, так и на *Linux*.

Клиентское программное обеспечение представляет собой браузер. Рекомендуется использовать последние версии браузеров, таких как *Google* *Chrome*, *Mozilla* *Firefox*, *Microsoft* *Edge*, *Safari*, имеющие лучшую производительность и поддержку *HTML5* и *JavaScript*.

Сам процесс развертывания информационной системы включает несколько ключевых шагов, которые должен выполнить системный администратор на сервере. Процесс настройки имеет следующий порядок:

– подготовка сервера включает в себя настройку операционной системы, установку и настройку *SQL* *Server*, создание пользователей и подключения к базе данных;

– настройка *web*-сервера, путём указания каталога с файлами приложения, задания порта для работы и установления необходимых сертификатов безопасности;

– создание каталога на сервере, куда будут скопированы все файлы и настройка прав доступа;

– задание строки подключения в *appsettings*.*json* на *SQL* *Server*, с учётом названия базы данных;

– запуск и полное тестирование всех ключевых функций, включая загрузку страниц, обработку заказов;

– настройка доступа к серверу через защищенные соединения, резервного копирования.

**3 СТРУКТУРА ПРИЛОЖЕНИЯ БАЗ ДАННЫХ МЕРОПРИЯТИЙ**

**ГОРОДА**

**3.1 Архитектура программного комплекса**

Структура приложения представляет собой детальное описание всех компонентов и слоев программы, формирующих архитектуру системы. Приложение реализовано на основе архитектуры *ASP.NET MVC* (*Model-View-Contoller*). Такая архитектура обеспечивает высокую гибкость и масштабируемость системы за счёт разделение на следующие три основные части:

– *Model* (модель) включает данные и логику бизнес-операций, взаимодействующие с базой данных;

– *View* (представление) отображает информацию пользователя. Не имеет доступа к бизнес-логике;

– *Contoller* (контроллер) управляет взаимодействием между моделью и представлением. Контроллер получает запросы от пользователя, обрабатывает их с помощью классов моделей и передаёт результат в представление [5].

Это разделение функционала на строго определённые слои даёт ряд следующих преимуществ:

– масштабируемость и простота сопровождения. Каждый слой в архитектуре выполняет строго определённые функции. Это значительно упрощает внесение изменений и добавление новых функций, так как нет необходимости разбираться какой слой что делает. Например, можно обновить внешний вид приложения без обращения к моделям;

– гибкость и удобство. Широкие возможности *ASP.NET MVC* позволяют строить интерактивные интерфейсы для разных пользователей. Тем самым в программе реализован разный функционал для пользователя и для администратора;

– повышенная тестируемость. Каждый из слоев за счёт разделения логики может тестироваться отдельно, независимо от другого слоя.

Помимо основных компонентов, проект включает дополнительные элементы:

– сервисы (*Services*) предоставляет дополнительные функции и вспомогательные операции;

– промежуточное программное обеспечение (*Middleware*) используется для инициализации базы данных и управления запросами;

– модели представлений (*ViewModels*) подготавливают данные к отображению в представлениях.

Структура *MVC* обеспечивает удобный и понятный интерфейс, через который можно быстро находить и просматривать интересующие события, бронировать билеты и управлять заказами. Имеет высокую производительность и стабильность системы, позволяя работать с большими объёмами данных и высоким количеством пользователей. На рисунке 3.1 представлена общая схема архитектуры приложения.

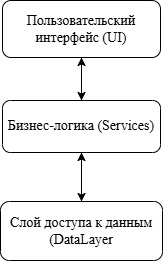


Рисунок 3.1 – Общая схема архитектуры приложения

Ядро приложения состоит из основных классов, моделирующих предметную область. Это классы моделей, отображающие все сущности:

– *Place* (место) содержит идентификатор места *PlaceID*, название места *PlaceName*, геолокацию *Geolocation*, коллекцию *Events*, связанных с местом;

– *Customer* (заказчик) включает идентификатор *CustomerID*, ФИО клиента *FullName*, паспортные данные клиента *PassportData*, а также коллекцию заказов билетов *TicketOrders*, сделанных заказчиком;

– *Organizer* (организатора) включает идентификатор *OrganizerID*, ФИО организатора *FullName*, должность организатора *Post*, а также коллекцию мероприятий *Events*, которые были реализованы данным организатором;

– *ApplicationUser*, содержащий дату регистрации пользователя *RegistrationDate.*

– *Events* (мероприятия) описывается уникальным идентификатором *EventID*, названием мероприятия *EventName*, внешним ключом *PlaceID* для связи с местом, датой проведения мероприятия *EventDate*, ценой билета *TicketPrice*, количеством доступных билетов *TicketAmount*, внешним ключом *OrganizerID* для связи с организаторами, *Place* и *Organizer* как ссылки на связанные объекты, коллекцию заказов билетов *TicketOrders*;

– *TicketOrder* (заказы билетов) содержит уникальный идентификатор заказа билетов *OrderID*, внешний ключ *EventID* для связи с событием, внешний ключ *CustomerID* для связи с заказчиком, дату заказа *OrderDate*, количество билетов *TicketCount*, *Event* и *Customer* как ссылки на связанные объекты.

Помимо классов моделей, ядро приложения содержит классы сервисов и интерфейсы. В приложении реализован один сервис и один соответствующих интерфейс. Сервис *OperationService* необходим для предоставления информации для главной страницы приложения, упрощая взаимодействие с данными с помощью структурированного вида. Интерфейс *IOperationService* содержит один метод *GetHomeViewModel*, который реализован для извлечения данных и формирования моделей представления.

Сервис *OperationService* реализует метод интерфейса и необходимую бизнес-логику: фильтрация данных по дате, ограничение количества выводимых записей, моделирование взаимосвязей, форматирование данных для представления. Реализация интерфейса позволяет инкапсулировать логику работы с базой данных, повышая читаемость и поддерживаемость кода [6]. Чертёж структуры *web*-приложения приведён в приложении А.

**3.2 Инфраструктура проекта**

Инфраструктура приложения является фундаментальным слоем, обеспечивающим взаимодействие с базой данных и управление, хранение и редактирование данных. Для приложения *ASP*.*NET* на *Entity* *Framework* *Core*, инфраструктурный слой включает классы для работы с базой данных, инструменты миграции, репозитории для абстракции доступа к данным. Такой подход упрощает управление данными, гарантирует целостность и масштабируемость приложения, а также отделяет бизнес-логику от механизмов доступа к данным.

Классы *Entity* *Framework* в проекте размещены в папке *Data*. Они являются основным инструментом работы с данными. *EventContext* – основной класс, отвечающий за взаимодействие с базой данных. Является производным от класса *DbContext* из *Entity* *Framework* и представляет объектно-реляционную модель данных. Класс выполняет следующие функции в проекте:

– для каждой сущности объявляет наборы данных (*DbSet*), позволяющие выполнять *CRUD*-операции в таблицах;

– позволяет задать строку подключения с помощью метода *OnConfiguring*, указывающую, как именно приложение будет связываться с базой данных;

Данный класс позволяет работать напрямую с объектами, а не писать сложные *SQL*-запросы, централизовать настройку доступа к данным, инкапсулировать логику взаимодействия с базой данных.

Специализированный класс контекста *ApplicationDbContext* наследуется от *IdentityDbContext* и используется для управления пользователями и их ролями при аутентификации в приложении. Реализует следующие задачи:

– хранение и управление данными о пользователях в базе данных;

– поддержка ролей;

– интеграция с *ASP*.*NET* *Identity*.

Данный класс расширяет функционал приложения, добавляя возможность интегрировать пользователей в систему, а также новые поля для пользователей, если они будут необходимы.

Статистический класс *DbInititalizer* отвечает за инициализацию базы данных. Если база после проверки в *EventContext* оказалась пустой, то вызовется именно этот класс. Он выполняет настройку начальных данных, включающую:

– создание ролей и пользователей, добавляя автоматически администратора и пользователя. Помимо этого, создаёт тестового администратора для возможности входа в систему даже в случае первого заполнения базы данных;

– заполнение таблиц базы данных, генерируя записи для всех сущностей (мест, организаторов, клиентов, событий и заказов билетов) в случае, если база данных при запуске оказалась пустой;

– проверка состояния базы данных, чтобы не вызывать заполнение, если одна из таблиц уже заполнена.

Класс *DbInitializer* является ключевым в критических ситуациях и устраняет необходимость ручного ввода данных, если база данных ещё не заполнена.

*Middleware* является промежуточным программным обеспечением и интегрируется в конвейер обработки *HTTP*-запросов. Класс *DbInitializerMiddleware* позволяет выполнить инициализацию базы данных при запуске с помощью *DbInitializer*. Алгоритм действий *Middleware* следующий:

– проверка содержатся ли данные в базе;

– вызов метода *DbInitializer*;

– передача управления следующему компоненту в конвейере запросов.

Также *Entity* *Framework* предоставляет возможность автоматически изменять структуру базы данных в соответствии с изменениями моделей данных. Для это применены миграции, содержащиеся в папке *Migrations*. Файл миграции обновляет базу данных и хранит историю изменений, позволяя откатываться к предыдущим версиям при необходимости. Миграции обеспечивают единый механизм управления изменениями структуры данных, уменьшают вероятность ошибок при ручном изменении данных базы и позволяют работать с объектной моделью, не заботясь о написании *SQL*-запросов на изменение структуры базы данных.

Для подключения к базе данных используется конфигурационный файл *appsettings*.*json*. Он хранит настройки, которые могут быть динамически загружены и использованы в приложении. Конфигурационный файл хранит строку подключения к удалённой базе данных, а также раздел *Logging*, определяющий уровни логирования для приложения, позволяя контролировать какие ошибки будут записаны в логи в случае возникновения ошибок.

**3.3 *Web*-интерфейс и модульные тесты**

*Web*-интерфейс разработан с использованием архитектуры *ASP*.*NET* *MVC* и включает основные компоненты: контроллеры, представления, модели представлений. Также в структуру входят вспомогательные классы, обеспечивающие функциональность интерфейса.

Контроллеры отвечают за обработку запросов от пользователей, взаимодействие с моделями и передачу данных в представления. Все контроллеры имеют схожую структуру, включающие стандартные действия для выполнения *CRUD*-операций (*Create*, *Read*, *Update*, *Delete*), а также обработку пользовательских фильтров и сортировки. Каждый контроллер имеет методы как *GET*, так и *POST* для получения и записи данных.

В проекты выделены следующие контроллеры:

– *CustomersContoller*;

– *EventsContoller*;

– *HomeContoller*;

– *OrganizersContoller*;

– *PlacesContoller*;

– *TicketOrdersContoller*;

– *UserContoller*.

Данные контроллеры обеспечивают выполнение *CRUD*-операций и содержат следующие методы:

*CustomersContoller*:

– *Index* обеспечивает отражение списка заказчиков с учётом фильтров, сортировки и постраничной навигации. Данные передаются в представление через объект *ViewBag* и типизированные модели представлений;

– *Details* обеспечивает вывод основной информации по заказчику;

– *Create* реализует метод создания нового заказчика;

– *Edit* необходим для реализации редактирования информации о заказчике;

– *Delete* реализует удаление заказчика с базы данных;

– Метод *ClearSessions* предназначен для очистки сессии при нажатии на кнопку, так как данные фильтрации заказчиков в программе сохраняются в сессии.

Контроллеры *PlacesContoller*, *OrganizersContoller* реализуют такие же методы для всех *CRUD*-операций. Единственное отличие – вместо метода *ClearSessions* они имеют метод *ClearFilters* для очистки куки в программе.

*EventsContoller*:

– *Index* обеспечивает отражение списка мероприятий с учётом фильтров, сортировки и постраничной навигации. Данные передаются в представление через объект *ViewBag* и типизированные модели представлений;

– *Details* обеспечивает вывод основной информации мероприятия;

– *Create* реализует метод создания нового мероприятия;

– *Edit* необходим для реализации редактирования информации о мероприятии;

– *Delete* реализует удаление мероприятия с базы данных;

– Метод *ClearFilters* предназначен для очистки сохранённой фильтрации в куки;

– *SearchEventNames*, *SearchPlaceNames* и *SearchOrganizerNames* предназначены для получения данных с других таблиц базы данных и заполнения выпадающих списков.

*TicketOrdersContoller* крайне схож с *EventsContoller* и выполняет аналогичные действия для всех *CRUD*-операций, очистку куки при необходимости, считывание данных с других таблиц для заполнения выпадающих списков.

Два других контроллера, которые не связаны напрямую с базой данных – *HomeContoller* и *UserContoller*. *HomeContoller* использует *Services* для заполнения главной страницы программы данными. *UserContoller* имеет разделение по правам доступа *Roles* = «*admin*», методы для стандартных *CRUD*-операций, а также методы *Login* и *Register* для входа и регистрации соответственно.

Представления реализованы с использованием *Razor*-синтаксиса и обеспечивают визуализацию данных, переданных контроллерами. Каждое представление соответствует определенному действию контроллера и позволяет:

– отобразить списки данных с использованием *HTML*-таблиц;

– предоставить формы для ввода данных;

– реализовать элементы пользовательского интерфейса, такие как фильтры, кнопки навигации, постраничный переход.

Каждое представление разделено в проекте на папки, такие как *Customers*, *Events*, *Home*, *Organizers*, *Places*, *TicketOrders*, *User*. Каждая папка содержит методы *Create*.*cshtml*, *Delete*.*cshtml*, *Details*.*cshtml*, *Edit*.*cshtml*, *Index*.*cshtml*. Папка *Home* содержит только *Index*.*cshtml* для вывода.

Помимо вышеописанных папок, реализована *Shared*. Главный её метод – \_*Layout*.*cshtml*, позволяющий реализовать навигации по страницам.

Для передачи данных между контроллерами и представлениями используются *ViewModels*. Они позволяют:

– отделить данные пользовательского интерфейса от моделей базы данных;

– применять атрибуты валидации для проверки пользовательского ввода.

В отличие от стандартных методов *Models*, *ViewModels* позволяют использовать данные разных таблиц, включающие не только идентификаторы. *ViewModels* содержат следующие поля:

– *TicketOrderViewModel*: *OrderID*, *EventID*, *CustomerID*, *CustomerName*, *CustomerPassportData*, *EventName*, *OrderDate*, *TicketCount*;

*– RegisterViewModel: Email, Password.* *ConfirmPassword*;

*– PlaceViewModel: PlaceID, PlaceName, Geolocation;*

*– OrganizerViewModel: OrganizerID, FullName, Post;*

*– LoginViewModel: Email, Password, RememberMe;*

*– HomeViewModel: Events, TicketOrders, Customers, Users, Places, Organizers*;

*– EventViewModel: EventID, EventName, PlaceID, OrganizerID, PlaceName, OrganizerName, EventDate, TicketPrice, TicketAmount, AvailableTickets* для хранения количества доступных для продажи билетов, не храня переменную в базе данных;

*– CustomerViewModel: CustomerID, PassportData;*

– *UserViewModel* имеет различные классы *ViewModel* для просмотра, редактирования, создания. Листинг программы представлен в приложении Б.

Модульное тестирование – процесс тестирования отдельных модулей или отдельных компонентов приложения для проверки их корректной работы в изолированных условиях. Основная цель – убедиться, что каждый метод контроллера работает согласно ожиданиям.

Для модульного тестирования .*NET* используются *xUnit*, *Moq* и *In*-*Memory* *Database*. Они обеспечивают асинхронное тестирование и совместимость с другими инструментами, *moq* позволяет имитировать поведение реальных компонентов, а *In*-*Memory* *Database* позволяет тестировать компоненты, работающие с базой данных без подключения к реальной системе управления базами данных.

Тестирование происходит с использованием *TestDataHelper*. Он содержит списки, используемые в качестве данных для тестирования. *GetFake* используются в каждом виде тестов и позволяют не подключаться к существующей базе данных для тестирования методов контроллеров. Листинг модульных тестов представлен в приложении В.

Всего в проекте реализовано 72 теста, охватывающих все методы контроллеров. На рисунке 3.1 показан результат выполнения тестов:

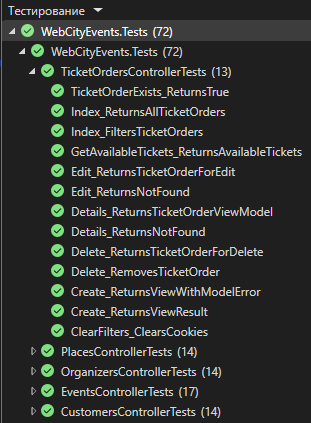


Рисунок 3.1 – Результат выполнения тестов

Модульные тесты проверяют, что *Index* действительно способен возвращать данные, *Create*, *Delete*, *Edit* выполняют необходимые операции.

Помимо тестирования *CRUD*-операций, тестируются также вспомогательные функции. Это функции проверки существования заказа по идентификатору, очистки куки и сессий, проверки получения доступного количества билетов.

**4 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ**

**4.1 Вход в приложение**

Приложение предназначено для управления городскими мероприятиями. Пользователи могут просматривать информацию о мероприятиях, заказывать билеты и управлять своими заказами. Интерфейс интуитивно понятен и ориентирован на удобство работы как для администратора, так и для пользователя. При открытии приложения пользователь должен войти в систему, либо зарегистрироваться. На рисунке 4.1 представлено основное окно входа в программу.

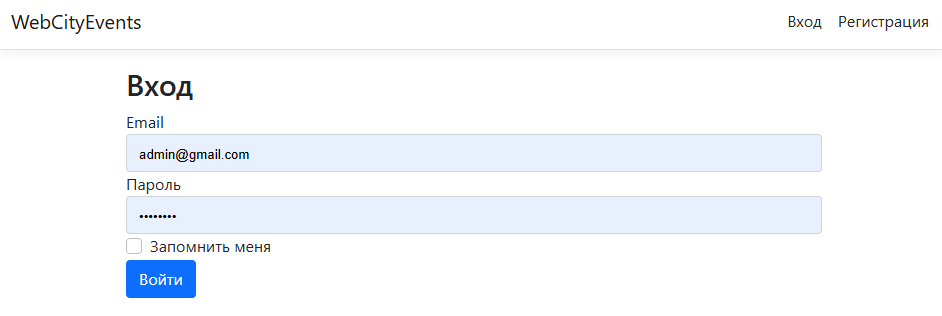


Рисунок 4.1 – Основное окно входа в программу

Данные для входа обязательно должны соответствовать имеющимся в базе данных. Если введён неверный *email* или пароль, то появится соответствующая ошибка. Когда пользователь хочет сначала зарегистрироваться, ему необходимо перейти на вкладку «Регистрация». Окно регистрации представлено на рисунке 4.2.

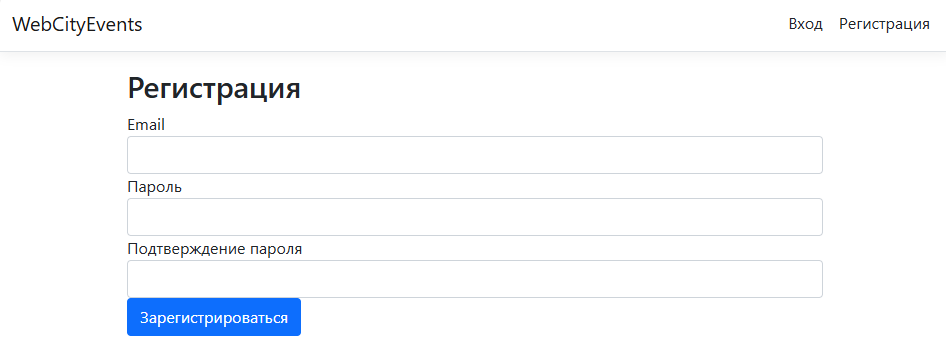


Рисунок 4.2 – Окно регистрации программы

Когда пользователь введёт достаточно надежный пароль, актуальную почту, содержащую *@* и подтвердит пароль, он получит доступ к системе в качестве стандартного пользователя.

Функция авторизации крайне полезна, удобна и позволяет разделить права доступа между пользователями. Основная панель действий программы изменяется в зависимости от роли пользователя в системе, добавляя возможность просмотра, редактирования, удаления пользователей для администратора. На рисунке 4.3 представлено сравнение основной панели для пользователей и для администраторов.



****

Рисунок 4.3 – Дополнительная вкладка для администратора

**4.2 Перечень и описание всех выполняемых функций обработки   
данных**

Приложение позволяет удобно просматривать все таблицы базы данных и управлять ими, а также фильтровать, сортировать информацию. На рисунке 4.4 представлено окно мероприятий.

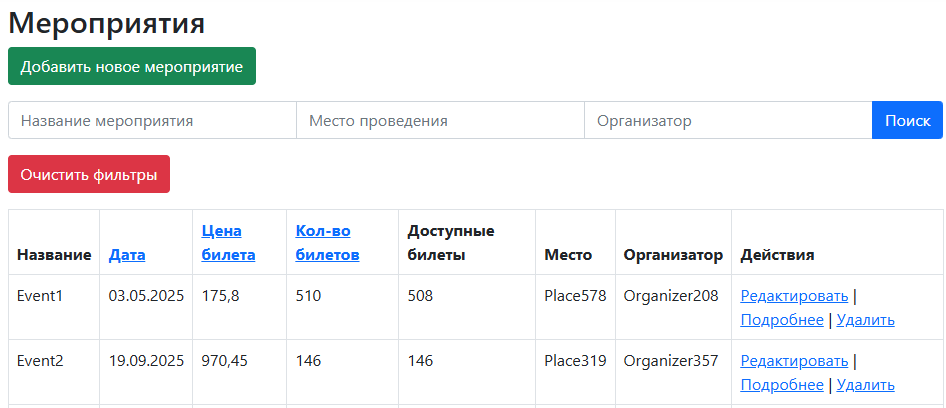


Рисунок 4.4 – Окно мероприятий

Для того, чтобы просмотр мероприятий работал в таком виде необходимо войти в систему с правильным *email* и паролем. В качестве подготовительных действий нужно иметь доступ к интернету для подключения к базе данных. Основные действия вкладки включают в себя:

– фильтрация по названию, месту проведения или организатору;

– навигация по страницам таблицы данных;

– сортировка по дате, цене билета и количестве билетов.

Дополнительные кнопки, такие как «Очистить фильтры» позволяют очистить результат сохранения фильтрации в куки. Окно добавления нового мероприятия, путём нажатия на кнопку «Добавить новое мероприятие», а также окно редактирования при нажатии «Редактировать» у соответствующего мероприятия представлено на рисунке 4.5.

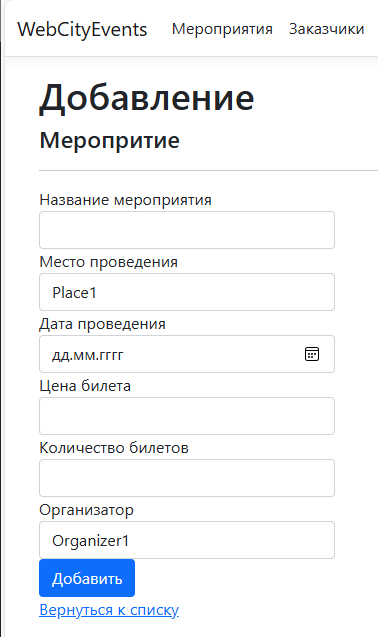
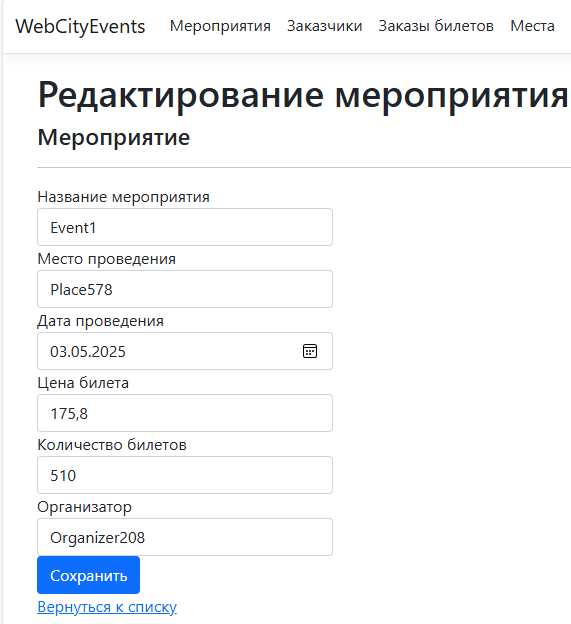
 

Рисунок 4.5 – Окна добавления и редактирования мероприятий

Добавление и редактирование имеют хоть и схожий, но интуитивно понятный интерфейс. Такие данные, как: название, цена билета и количество билетов вводятся с клавиатуры. Место проведения и организатор подгружаются с других таблиц и являются выпадающими списками. Для добавления и редактирования реализованы исключения для всех возможных ошибок, таких как:

– пустое название мероприятия или пустая дата проведения;

– отрицательная цена билета;

– отрицательное или нулевое количество билетов.

Для навигации по страницам используется одинаковая для всех таблиц структура. На рисунке 4.6 представлена навигация по страницам.



Рисунок 4.6 – Окна добавления и редактирования мероприятий

Также приложение позволяет просматривать информацию о мероприятии при нажатии «Подробнее» и удалить, нажав соответствующую «Удалить». На рисунке 4.7 представлены окна подробной информации и удаления мероприятия.

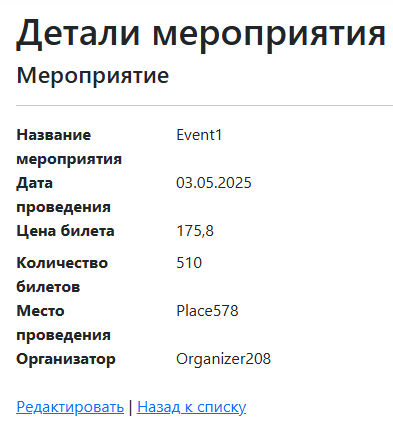
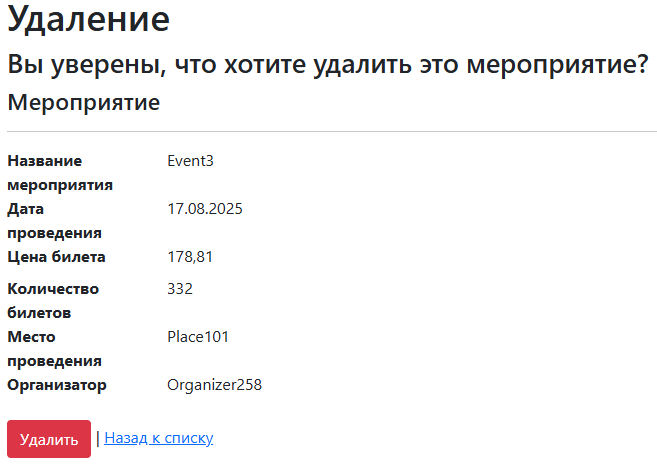
 

Рисунок 4.7 – Окна подробнее и удаления мероприятий

Таблицы заказчики, места и организаторы находятся на стороне отношений «один» и имеют одинаковый интерфейс, а также одинаковую реализацию методов изменений данных. На рисунках 4.8-4.10 представлены окна заказчиков, мест и организаторов.

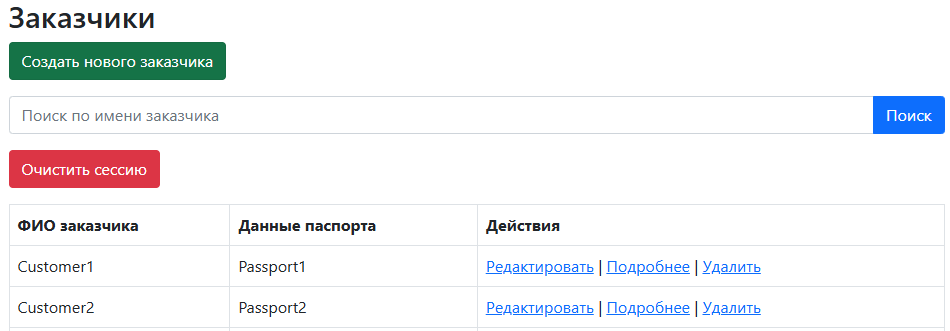


Рисунок 4.8 – Окно заказчиков со всеми функциями

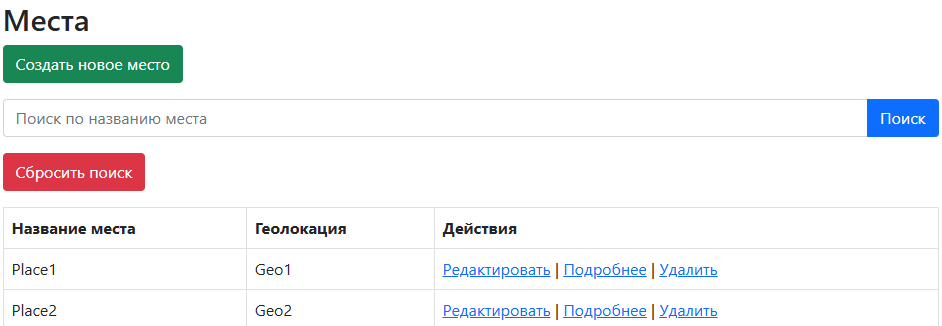


Рисунок 4.9 – Окна мест со всеми функциями

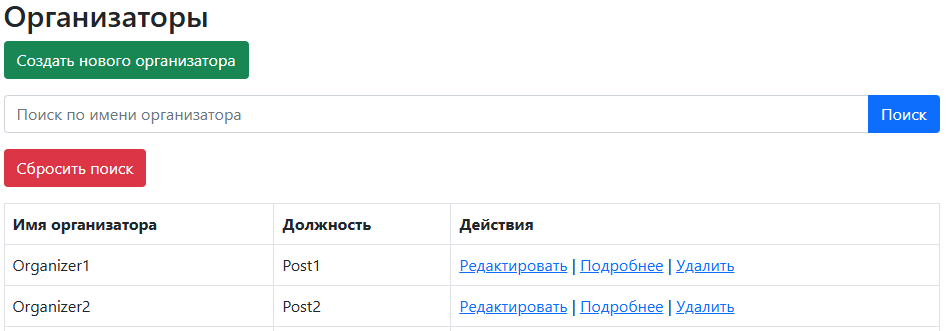


Рисунок 4.10 – Окно организаторов со всеми функциями

Для каждого из окон таблиц на стороне отношений «один» реализованы возможности добавления, редактирования, просмотра и удаления информации при помощи нажатия на специальные кнопки. Добавление и редактирование требует заполнения всех полей, иначе программа выдаст ошибку.

Вкладка «Заказы билетов» является ключевой для всей программы. Она включает вывод всех заказов с информацией с разных таблиц, фильтрации по ФИО заказчика и названию мероприятия, сортировку по количеству билетов и дата заказа. «Заказы билетов» представлены на рисунке 4.11.

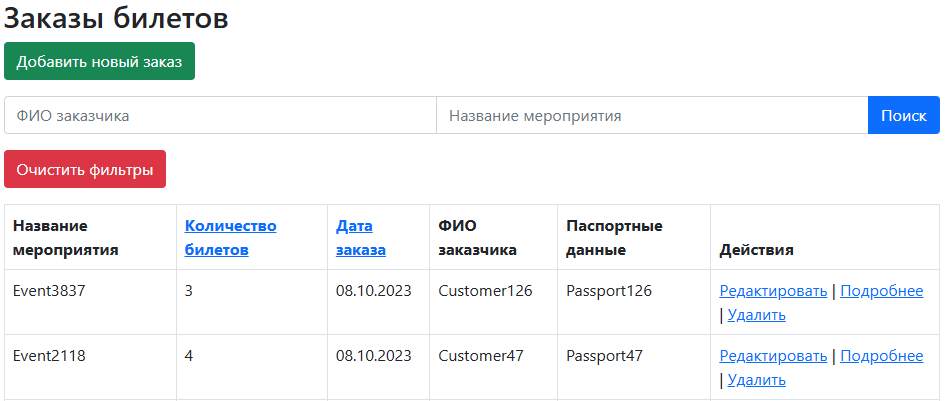


Рисунок 4.11 – Окно заказов билетов

В добавлении и редактировании мероприятия заполняются только тогда, когда начинается ввод названия. Дата и количество доступных билетов являются просто информативными. Окно добавления нового заказа, а также окно редактирования заказа представлено на рисунке 4.12.

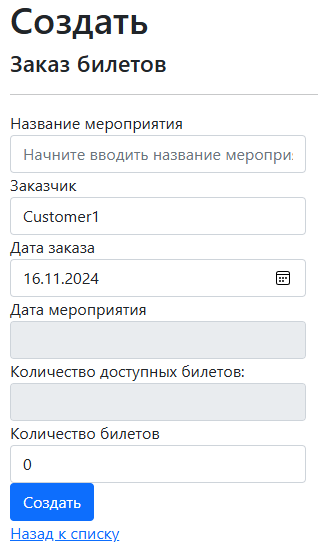
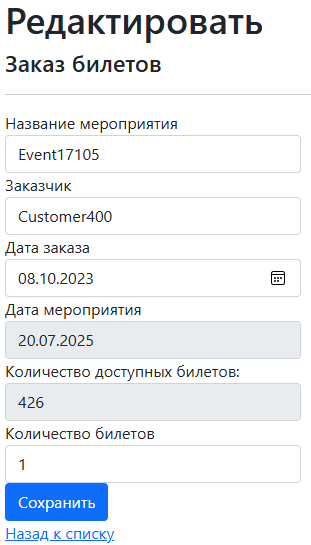
 

Рисунок 4.12 – Окна добавления и редактирования мероприятий

**4.3 Рекомендации по освоению и эксплуатации приложения**

Для успешного освоения и эффективного использования приложения требуется следовать определённым рекомендациям.

Учёт специфики работы с системы и выполнение основных шагов по её настройке, запуску и эксплуатации является ключевым для всей системы. Рекомендации по освоению приложения для пользователей и администраторов включают следующие пункты:

– изучить доступный интерфейс: вкладки мероприятий, заказчиков, заказов билетов, мест, организаторов, а также регистрации и авторизации;

– создать нового пользователя для изучения процесса регистрации;

– ознакомиться с процессом фильтрации и сортировки во всех таблицах базы данных, а также навигации по страницам;

– создать новую запись в одной из таблиц базы данных (лучшего всего сразу в самой сложной – заказы билетов). Просмотреть созданную информацию с помощью кнопки «Подробнее»;

Например, можно добивать новый заказ билетов. Для этого нужно перейти во вкладку «Заказы билетов», затем нажать «Добавить новый заказ». Ввести название мероприятия, например «*Event10*» и выбрать его с выпадающего списка, выбрать любого заказчика, убедиться, что выбранная дата заказа не позже даты мероприятия, выбрать количество билетов, которое не будет превышать количество доступных билетов и затем нажать «Создать».

– отредактировать и удалить созданную запись.

В качестве правил запуска нужно учитывать предварительные требования и этапы запуска. Они включают:

– проверить интернет-соединение;

– открыть приложение в браузере, которое обращается к серверу на операционной системе либо *Windows* *Server*, либо *Linux*;

– убедиться, что подключение к базе данных успешно выполнено;

– авторизоваться или зарегистрироваться для полного доступа к системе;

– вводить данные в точности, как указано в примере, чтобы избежать появления различных ошибок.

Перед началом работы рекомендуется ознакомиться с инструкцией, всегда проверять корректность вводимых данных, в случае возникновения проблем перепроверять вводимую информацию.

Требования к уровню подготовки пользователя не являются особыми и являются типичными для любого приложения баз данных и включают:

– минимальный уровень владения ПК, умение пользоваться браузером и базовыми функциями *web*-сайтов;

– понимание основ управления мероприятиями, пользователями и заказами билетов через интерфейс;

– навыки работы с базовыми настройками приложения.

**5 РУКОВОДСТВО ПРОГРАММИСТА**

Приложение предназначено для управления мероприятиями города, включая их организацию, заказ и учёт билетов, а также регистрацию и авторизацию пользователей, управление местами проведения и информацией о заказчиках и организаторах. Предназначено для организаций, занимающихся проведением городских мероприятий или для личного пользования. Условия применения включают в себя:

– работа в среде с доступом к интернету;

– использование серверов с поддержкой *ASP.NET Core* и *SQL* *Server* для запуска серверной части;

– доступ к *web*-браузеру для работы на клиентской стороне.

Требования к серверу и клиенту описаны в главе 2.

Основные характеристики для работы программы включают следующие режимы работы:

– пользовательский режим: взаимодействие с событиями (просмотр, фильтрация, заказ, редактирование);

– административный режим: аналогичные действия с пользовательским режимом, но с возможностью редактирования пользователей.

Правильность выполнения действий в программе реализована с помощью валидации входных данных на сервере и клиенте, а также обработка всех исключений для предотвращения ошибок работы приложения.

Обращение к программе происходит с помощью запуска приложения через *web*-сервер. Клиентская часть доступна через *URL*, предоставляемый сервером. После запуска серверной части приложение обрабатывает *HTTP*-запросы, перенаправляя их к соответствующим контроллерам.

Пример обращения имеет вид:

– пользователь открывает страницу браузера и вводит *URL* приложения;

– программа принимает запрос, перенаправляет его к соответствующему методу контроллера;

– контроллер обрабатывает полученные данные;

– ответ от сервера возвращается в виде *web*-страницы.

В качестве входных данных в программе используются формат данных и информация, вводимая пользователем через *web*-интерфейс:

– личные данные;

– параметры фильтрации;

– данные для выполнения операций.

В качестве выходных данных происходит кодирование текстовых данных в формат *UTF-8*, преобразование числовых и временных значений в стандартный формат и возвращение следующих данных при обработке:

– *HTML*-страницы с результатами поиска;

– подтверждение операций;

– информация об ошибках.

Возможных ошибок, возникающих при работе с программой большое количество и все они выдают поясняющее сообщение, упрощая понимание пользователя что он сделал не так. Программа генерирует текстовые сообщения в зависимости от контекста, например:

– «Пользователь с таким логином уже существует»;

– «Количество билетов должно быть больше 0»;

– «Дата заказа не может быть позже даты проведения мероприятия».

В качестве сообщений, выводимых не в случае ошибок, можно привести пример «Добро пожаловать, <имя пользователя>!».

В проектировании программы не предусматриваются специализированные требования или ограничения, что обеспечивает её гибкость и доступность для разнообразных пользовательских сред. Встроенная система обратной связи информирует пользователя об ошибках ввода, способствуя коррекции и предотвращению потенциальных проблем в процессе взаимодействия с программой.

Исходные данные для работы приложения поступают непосредственно из базы данных, а также вносятся пользователями через интерфейс программы, когда это необходимо для выполнения определенной функции или задачи. Результатом работы программы могут быть не только успешно выполненные операции, но и уведомления о допущенных ошибках ввода, что значительно упрощает процесс выявления и устранения недочетов.

В процессе регистрации новые пользователи должны предоставить личную информацию, включая имя, логин и пароль. Приложение включает в себя систему проверки подлинности данных, обеспечивая высокий уровень безопасности и предотвращая доступ неавторизованных лиц.

Эти инструкции позволяют программисту разобраться в особенностях работы приложения, запустить и сопровождать его, а также обеспечить обработку возможных ошибок.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсового проекта разработана информационная система, предназначенная для автоматизации процессов, связанных с управлением мероприятиями. Она представляет собой современное *web*-решение, которое обеспечивает удобное управление городской событийной программой, автоматизируя процессы бронирования билетов, регистрации пользователей и администрирования мероприятий.

Полноценная архитектура на базе *ASP*.*NET* *MVC* обеспечивает разделение логики приложения на серверную, клиентскую и модельную части, упрощая дальнейшее сопровождение и модернизацию. Реляционная база данных на основе *SQL* *Server* с использованием *Entity* *Framework* гарантирует целостность данных и их надёжное хранение. Интуитивно понятный интерфейс делает приложение удобным для различных категорий пользователей. Фильтрации и сортировки позволяют эффективно взаимодействовать с большим объёмом данных, а также обеспечивается высокая надёжность системы.

Приложение может быть успешно использовано в различных областях. Организации, занимающиеся проведением массовых мероприятий, туристические агентства, учебные заведения могут эффективно использовать приложение для своих нужд.

Система обладает высокой учебной ценностью, так как она демонстрирует ключевые подходы к проектированию и разработке современных *web*-приложений, позволяя изучить основы проектирования баз данных, принципы построения многоуровневой архитектуры, применение технологий в реальных проектах, практическую реализацию пользовательского интерфейса с удобным взаимодействием с системой.

Практическая значимость заключается в том, что система решает актуальные задачи автоматизации процессов управления мероприятиями. Это позволяет оптимизировать временные и финансовые затраты, повысить уровень обслуживание пользователей и упростить работу с данными.

Разработанная система является не только актуальной, но и значимой для практического применения в реальных условиях с большим количеством событий. Она отвечает современным требованиям к надёжности, функциональности и удобству использования.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Карпенко, В. Проектирование информационных систем / Волков С. – Издательство «Питер», 2020. – 304 с.
2. Microsoft – Документация по ASP.NET Core MVC – Электрон. данные. – Режим доступа: https://learn.microsoft.com/ru-ru/aspnet/core. – Дата доступа: 12.11.2024.
3. Гриффитс, И. Программируем на C# 8.0 / И. Гриффитс – Издательство «Питер», 2021. – 944 с.
4. Обзор языка C# – Руководство по C#: Microsoft Docs. – Электрон. данные. – Режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/>. – Дата доступа: 12.11.2024.
5. Microsoft – Code First and Entity Framework Core Documentation. Entity Framework Core – Электрон. данные. – Режим доступа: https://learn.microsoft.com/en-us/ef/core/. – Дата доступа: 13.11.2024.
6. Норман, Д. Дизайн привычных вещей. Как создавать интерфейсы, которые не раздражают / Иванов, Фербер – Москва, 2018. – 368 с.
7. Дипломное проектирование: учеб.-метод. пособие для студентов специальностей 1-40 04 01 «Информатика и технологии программирования» и 1-40 05 01 «Информационные системы и технологии (по направлениям)» днев. формы обучения / сост.: Т.А. Трохова, И.А. Мурашко, К.С. Курочка. – Гомель: ГГТУ им. П.О. Сухого, 2019. – 55 с. – Режим доступа: <https://elib.gstu.by>. – Дата доступа: 16.11.2024.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**ЧЕРТЁЖ СТРУКТУРЫ *WEB*-ПРИЛОЖЕНИЯ**

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**ЛИСТИНГ ПРОГРАММЫ**

**Код класса *ApplicationUser.cs***

using Microsoft.AspNetCore.Identity;

namespace WebCityEvents.Models

{

public class ApplicationUser : IdentityUser

{

public DateTime RegistrationDate { get; set; }

}

}

**Код класса *Customer.cs***

namespace WebCityEvents.Models

{

public class Customer

{

public int CustomerID { get; set; }

public string FullName { get; set; }

public string PassportData { get; set; }

public ICollection<TicketOrder> TicketOrders { get; set; }

public Customer()

{

TicketOrders = new List<TicketOrder>();

}

}

}

**Код класса *Event.cs***

namespace WebCityEvents.Models

{

public class Event

{

public int EventID { get; set; }

public string EventName { get; set; }

public int PlaceID { get; set; }

public DateTime EventDate { get; set; }

public float TicketPrice { get; set; }

public int TicketAmount { get; set; }

public int OrganizerID { get; set; }

public Place Place { get; set; }

public Organizer Organizer { get; set; }

public ICollection<TicketOrder> TicketOrders { get; set; }

public Event()

{

TicketOrders = new List<TicketOrder>();

}

}

}

**Код класса *Organizer.cs***

namespace WebCityEvents.Models

{

public class Organizer

{

public int OrganizerID { get; set; }

public string FullName { get; set; }

public string Post { get; set; }

public ICollection<Event> Events { get; set; }

public Organizer()

{

Events = new List<Event>();

}

}

}

**Код класса *Place.cs***

namespace WebCityEvents.Models

{

public class Place

{

public int PlaceID { get; set; }

public string PlaceName { get; set; }

public string Geolocation { get; set; }

public ICollection<Event> Events { get; set; }

public Place()

{

Events = new List<Event>();

}

}

}

**Код класса *TicketOrder.cs***

namespace WebCityEvents.Models

{

public class TicketOrder

{

public int OrderID { get; set; }

public int EventID { get; set; }

public int CustomerID { get; set; }

public DateTime OrderDate { get; set; }

public int TicketCount { get; set; }

public Event Event { get; set; }

public Customer Customer { get; set; }

}

}

**Код класса *HomeViewModel.cs***

using WebCityEvents.Models;

namespace WebCityEvents.ViewModels

{

public class HomeViewModel

{

public List<EventViewModel> Events { get; set; }

public List<TicketOrderViewModel> TicketOrders { get; set; }

public List<CustomerViewModel> Customers { get; set; }

public List<ApplicationUser> Users { get; set; }

public List<PlaceViewModel> Places { get; set; }

public List<OrganizerViewModel> Organizers { get; set; }

}

}

**Код класса *CustomerViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels

{

public class CustomerViewModel

{

public int CustomerID { get; set; }

[Required(ErrorMessage = "Поле ФИО обязательно для заполнения")]

[RegularExpression(@"^[А-Яа-яЁёA-Za-z\s]+$", ErrorMessage = "ФИО должно содержать только буквы")]

public string FullName { get; set; }

[Required(ErrorMessage = "Поле паспортные данные обязательно для заполнения")]

public string PassportData { get; set; }

}

}

**Код класса *EventViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels

{

public class EventViewModel

{

public int EventID { get; set; }

[Required(ErrorMessage = "Название мероприятия обязательно")]

public string EventName { get; set; }

public int PlaceID { get; set; }

public int OrganizerID { get; set; }

public string PlaceName { get; set; }

public string OrganizerName { get; set; }

public DateTime EventDate { get; set; }

[Range(0, double.MaxValue, ErrorMessage = "Цена билета не может быть отрицательной")]

public float TicketPrice { get; set; }

[Range(1, int.MaxValue, ErrorMessage = "Количество билетов должно быть больше нуля")]

public int TicketAmount { get; set; }

public int AvailableTickets { get; set; }

}

}

**Код класса *OrganizerViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels

{

public class OrganizerViewModel

{

public int OrganizerID { get; set; }

[Required(ErrorMessage = "Поле ФИО обязательно для заполнения")]

public string FullName { get; set; }

[Required(ErrorMessage = "Поле должность обязательно для заполнения")]

public string Post { get; set; }

}

}

**Код класса *PlaceViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels

{

public class PlaceViewModel

{

public int PlaceID { get; set; }

[Required(ErrorMessage = "Поле название места обязательно для заполнения")]

public string PlaceName { get; set; }

[Required(ErrorMessage = "Поле геолокация обязательно для заполнения")]

public string Geolocation { get; set; }

}

}

**Код класса *TicketOrderViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels

{

public class TicketOrderViewModel

{

public int OrderID { get; set; }

[Required(ErrorMessage = "Необходимо выбрать мероприятие")]

[Range(1, int.MaxValue, ErrorMessage = "Мероприятие должно быть выбрано из списка")]

public int EventID { get; set; }

public int CustomerID { get; set; }

public string CustomerName { get; set; }

public string CustomerPassportData { get; set; }

public string EventName { get; set; }

public DateTime OrderDate { get; set; }

[Required(ErrorMessage = "Количество билетов обязательно")]

[Range(1, int.MaxValue, ErrorMessage = "Количество билетов должно быть больше 0")]

public int TicketCount { get; set; }

}

}

**Код класса *UserViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels.Users

{

public class UserViewModel

{

public string Id { get; set; }

[Display(Name = "Имя")]

public string UserName { get; set; }

[EmailAddress(ErrorMessage = "Некорректный адрес")]

public string Email { get; set; }

[Display(Name = "Дата регистрации")]

[DataType(DataType.Date)]

public DateTime RegistrationDate { get; set; }

[Display(Name = "Роль")]

public string RoleName { get; set; }

}

}

**Код класса *EditUserViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels.Users

{

public class EditUserViewModel

{

public string Id { get; set; }

[Required(ErrorMessage = "Имя пользователя обязательно")]

[Display(Name = "Имя")]

public string UserName { get; set; }

[EmailAddress(ErrorMessage = "Некорректный адрес")]

public string Email { get; set; }

[Display(Name = "Роль")]

public string UserRole { get; set; }

public EditUserViewModel()

{

UserRole = "user";

}

}

}

**Код класса *CreateUserViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels.Users

{

public class CreateUserViewModel

{

[Required(ErrorMessage = "Имя пользователя обязательно")]

[Display(Name = "Имя")]

public string UserName { get; set; }

[EmailAddress(ErrorMessage = "Некорректный адрес")]

public string Email { get; set; }

[Required(ErrorMessage = "Пароль обязателен")]

[Display(Name = "Пароль")]

public string Password { get; set; }

[Display(Name = "Роль")]

public string UserRole { get; set; }

[Display(Name = "Дата регистрации")]

public DateTime RegistrationDate { get; set; }

public CreateUserViewModel()

{

UserRole = "user";

RegistrationDate = DateTime.Now;

}

}

}

**Код класса *LoginViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels

{

public class LoginViewModel

{

[Required]

[EmailAddress]

[Display(Name = "Email")]

public string Email { get; set; }

[Required]

[DataType(DataType.Password)]

[Display(Name = "Пароль")]

public string Password { get; set; }

[Display(Name = "Запомнить меня")]

public bool RememberMe { get; set; }

}

}

**Код класса *RegisterViewModel.cs***

using System.ComponentModel.DataAnnotations;

namespace WebCityEvents.ViewModels

{

public class RegisterViewModel

{

[Required]

[EmailAddress]

[Display(Name = "Email")]

public string Email { get; set; }

[Required]

[DataType(DataType.Password)]

[Display(Name = "Пароль")]

public string Password { get; set; }

[DataType(DataType.Password)]

[Display(Name = "Подтверждение пароля")]

[Compare("Password", ErrorMessage = "Пароли не совпадают.")]

public string ConfirmPassword { get; set; }

}

}

**Код класса *ApplicationDbContext.cs***

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Models;

namespace WebCityEvents.Data

{

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options) { }

}

}

**Код класса *DbInitializer.cs***

using Microsoft.AspNetCore.Identity;

using WebCityEvents.Models;

namespace WebCityEvents.Data

{

public static class DbInitializer

{

public static async Task InitializeAsync(UserManager<ApplicationUser> userManager, RoleManager<IdentityRole> roleManager, EventContext context)

{

string[] roleNames = { "admin", "user" };

foreach (var roleName in roleNames)

{

if (!await roleManager.RoleExistsAsync(roleName))

{

await roleManager.CreateAsync(new IdentityRole(roleName));

}

}

var adminUsers = await userManager.GetUsersInRoleAsync("admin");

if (!adminUsers.Any())

{

var adminUser = new ApplicationUser

{

UserName = "TestAdmin",

Email = "admin@gmail.com",

EmailConfirmed = true

};

await userManager.CreateAsync(adminUser, "0809Vlad");

await userManager.AddToRoleAsync(adminUser, "admin");

}

if (context.Places.Any() && context.Organizers.Any() && context.Customers.Any() && context.Events.Any() && context.TicketOrders.Any())

{

return;

}

for (int i = 1; i <= 600; i++)

{

context.Places.Add(new Place

{

PlaceName = $"Place{i}",

Geolocation = $"Geo{i}"

});

}

for (int i = 1; i <= 500; i++)

{

context.Organizers.Add(new Organizer

{

FullName = $"Organizer{i}",

Post = $"Post{i}"

});

}

for (int i = 1; i <= 500; i++)

{

context.Customers.Add(new Customer

{

FullName = $"Customer{i}",

PassportData = $"Passport{i}"

});

}

var random = new Random();

for (int i = 1; i <= 20000; i++)

{

context.Events.Add(new Event

{

EventName = $"Event{i}",

EventDate = DateTime.Now.AddDays(random.Next(0, 365)),

TicketPrice = 100 + (float)(random.NextDouble() \* 1000),

TicketAmount = 100 + random.Next(500),

PlaceID = random.Next(1, 600),

OrganizerID = random.Next(1, 500)

});

}

for (int i = 1; i <= 25000; i++)

{

context.TicketOrders.Add(new TicketOrder

{

EventID = random.Next(1, 20000),

CustomerID = random.Next(1, 500),

OrderDate = DateTime.Now.AddDays(-random.Next(0, 365)),

TicketCount = random.Next(1, 5)

});

}

await context.SaveChangesAsync();

}

}

}

**Код класса *EventContext.cs***

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Models;

namespace WebCityEvents.Data

{

public class EventContext : DbContext

{

public EventContext(DbContextOptions<EventContext> options) : base(options) { }

public EventContext() : base() { }

public DbSet<Place> Places { get; set; }

public DbSet<Organizer> Organizers { get; set; }

public DbSet<Customer> Customers { get; set; }

public DbSet<Event> Events { get; set; }

public DbSet<TicketOrder> TicketOrders { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

ConfigurationBuilder builder = new();

builder.SetBasePath(Directory.GetCurrentDirectory());

builder.AddJsonFile("appsettings.json");

IConfigurationRoot configuration = builder.Build();

string connectionString = configuration.GetConnectionString("LocalSQLConnection");

optionsBuilder

.UseSqlServer(connectionString)

.LogTo(message => System.Diagnostics.Debug.WriteLine(message));

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<TicketOrder>()

.HasKey(t => t.OrderID);

}

public async Task<bool> IsDatabaseEmptyAsync()

{

return !await Customers.AnyAsync();

}

}

}

**Код класса *DbInitializationMiddleware.cs***

using Microsoft.AspNetCore.Identity;

using WebCityEvents.Data;

using WebCityEvents.Models;

public class DbInitializationMiddleware

{

private readonly RequestDelegate \_next;

public DbInitializationMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context, UserManager<ApplicationUser> userManager, RoleManager<IdentityRole> roleManager, EventContext dbContext)

{

if (await dbContext.IsDatabaseEmptyAsync())

{

await DbInitializer.InitializeAsync(userManager, roleManager, dbContext);

}

await \_next(context);

}

}

**Код интерфейса *IOperationService.cs***

using WebCityEvents.ViewModels;

namespace WebCityEvents.Services

{

public interface IOperationService

{

HomeViewModel GetHomeViewModel(int numberRows = 10);

}

}

**Код класса *OperationService.cs***

using WebCityEvents.Data;

using WebCityEvents.ViewModels;

using Microsoft.EntityFrameworkCore;

namespace WebCityEvents.Services

{

public class OperationService : IOperationService

{

private readonly EventContext \_context;

public OperationService(EventContext context)

{

\_context = context;

}

public HomeViewModel GetHomeViewModel(int numberRows = 10)

{

var events = \_context.Events

.Include(e => e.Organizer)

.OrderByDescending(e => e.EventDate)

.Select(e => new EventViewModel

{

EventID = e.EventID,

EventName = e.EventName,

OrganizerName = e.Organizer.FullName,

EventDate = e.EventDate,

TicketPrice = e.TicketPrice,

TicketAmount = e.TicketAmount

})

.Take(numberRows)

.ToList();

var customers = \_context.Customers

.Select(c => new CustomerViewModel

{

CustomerID = c.CustomerID,

FullName = c.FullName,

PassportData = c.PassportData

})

.Take(numberRows)

.ToList();

var ticketOrders = \_context.TicketOrders

.Include(o => o.Customer)

.Include(o => o.Event)

.OrderByDescending(o => o.OrderDate)

.Select(o => new TicketOrderViewModel

{

OrderID = o.OrderID,

CustomerName = o.Customer.FullName,

EventName = o.Event.EventName,

OrderDate = o.OrderDate,

TicketCount = o.TicketCount

})

.Take(numberRows)

.ToList();

var places = \_context.Places

.Select(p => new PlaceViewModel

{

PlaceID = p.PlaceID,

PlaceName = p.PlaceName,

Geolocation = p.Geolocation

})

.Take(numberRows)

.ToList();

var organizers = \_context.Organizers

.Select(or => new OrganizerViewModel

{

OrganizerID = or.OrganizerID,

FullName = or.FullName,

Post = or.Post,

})

.Take(numberRows)

.ToList();

HomeViewModel homeViewModel = new HomeViewModel

{

Events = events,

Customers = customers,

TicketOrders = ticketOrders,

Places = places,

Organizers = organizers

};

return homeViewModel;

}

}

}

**Код файла конфигурации *appsetings.json***

{

"ConnectionStrings": {

"RemoteSQLConnection": "Data Source= Server=db8077.databaseasp.net; Database=db8077; User Id=db8077; Password=2Em?p3%B8+Fi; Encrypt=False; MultipleActiveResultSets=True;"

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*"

}

**Код класса *Program.cs***

using WebCityEvents.Data;

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Services;

using Microsoft.AspNetCore.Identity;

using WebCityEvents.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<EventContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RemoteSQLConnection")));

builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RemoteSQLConnection")));

builder.Services.AddIdentity<ApplicationUser, IdentityRole>(options =>

{

options.Password.RequireDigit = true;

options.Password.RequiredLength = 6;

options.Password.RequireNonAlphanumeric = false;

})

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

builder.Services.AddScoped<IOperationService, OperationService>();

builder.Services.AddControllersWithViews();

builder.Services.AddSession();

var app = builder.Build();

app.UseMiddleware<DbInitializationMiddleware>();

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseSession();

app.MapControllerRoute(

name: "default",

pattern: "{controller=User}/{action=Login}/{id?}");

app.Run();

**Код класса *UserContoller.cs***

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using WebCityEvents.Models;

using WebCityEvents.ViewModels;

using WebCityEvents.ViewModels.Users;

using Microsoft.EntityFrameworkCore;

namespace WebCityEvents.Controllers

{

public class UserController : Controller

{

private readonly UserManager<ApplicationUser> \_userManager;

private readonly RoleManager<IdentityRole> \_roleManager;

private readonly SignInManager<ApplicationUser> \_signInManager;

public UserController(UserManager<ApplicationUser> userManager, RoleManager<IdentityRole> roleManager, SignInManager<ApplicationUser> signInManager)

{

\_userManager = userManager;

\_roleManager = roleManager;

\_signInManager = signInManager;

}

// GET: User/Register

public IActionResult Register()

{

return View();

}

// POST: User/Register

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Register(RegisterViewModel model)

{

if (ModelState.IsValid)

{

var user = new ApplicationUser

{

UserName = model.Email,

Email = model.Email,

RegistrationDate = DateTime.Now

};

var result = await \_userManager.CreateAsync(user, model.Password);

if (result.Succeeded)

{

await \_userManager.AddToRoleAsync(user, "user");

await \_signInManager.SignInAsync(user, isPersistent: false);

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

}

foreach (var error in result.Errors)

{

ModelState.AddModelError(string.Empty, error.Description);

}

}

return View(model);

}

// GET: User/Login

public IActionResult Login()

{

return View();

}

// POST: User/Login

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Login(LoginViewModel model)

{

if (ModelState.IsValid)

{

var user = await \_userManager.FindByEmailAsync(model.Email);

if (user != null)

{

var result = await \_signInManager.PasswordSignInAsync(user, model.Password, model.RememberMe, lockoutOnFailure: false);

if (result.Succeeded)

{

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

}

else

{

ModelState.AddModelError(string.Empty, "Неправильный логин или пароль");

}

}

else

{

ModelState.AddModelError(string.Empty, "Пользователь с таким email не найден");

}

}

return View(model);

}

// POST: User/Logout

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Logout()

{

await \_signInManager.SignOutAsync();

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

}

[Authorize(Roles = "admin")]

public async Task<IActionResult> Index()

{

var users = \_userManager.Users.OrderBy(user => user.UserName);

var userViewModels = new List<UserViewModel>();

foreach (var user in users)

{

var userRoles = await \_userManager.GetRolesAsync(user);

var roleName = userRoles.FirstOrDefault() ?? string.Empty;

userViewModels.Add(new UserViewModel

{

Id = user.Id,

UserName = user.UserName,

Email = user.Email,

RegistrationDate = user.RegistrationDate,

RoleName = roleName

});

}

return View(userViewModels);

}

[Authorize(Roles = "admin")]

public IActionResult Create()

{

ViewBag.UserRole = new SelectList(\_roleManager.Roles, "Name", "Name");

return View(new CreateUserViewModel());

}

[HttpPost]

[Authorize(Roles = "admin")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(CreateUserViewModel model)

{

if (ModelState.IsValid)

{

var existingUser = await \_userManager.FindByEmailAsync(model.Email);

if (existingUser != null)

{

ModelState.AddModelError("Email", "Пользователь с таким email уже существует");

ViewBag.UserRole = new SelectList(\_roleManager.Roles, "Name", "Name");

return View(model);

}

var user = new ApplicationUser

{

UserName = model.UserName,

Email = model.Email,

RegistrationDate = DateTime.Now

};

var result = await \_userManager.CreateAsync(user, model.Password);

if (result.Succeeded)

{

if (!string.IsNullOrEmpty(model.UserRole))

{

await \_userManager.AddToRoleAsync(user, model.UserRole);

}

return RedirectToAction("Index");

}

foreach (var error in result.Errors)

{

ModelState.AddModelError(string.Empty, error.Description);

}

}

ViewBag.UserRole = new SelectList(await \_roleManager.Roles.ToListAsync(), "Name", "Name");

return View(model);

}

[Authorize(Roles = "admin")]

public async Task<IActionResult> Edit(string id)

{

var user = await \_userManager.FindByIdAsync(id);

if (user == null)

{

return NotFound();

}

var userRoles = await \_userManager.GetRolesAsync(user);

string userRole = userRoles.FirstOrDefault() ?? string.Empty;

var model = new EditUserViewModel

{

Id = user.Id,

UserName = user.UserName,

Email = user.Email,

UserRole = userRole

};

var roles = await \_roleManager.Roles.ToListAsync();

ViewData["UserRole"] = new SelectList(roles, "Name", "Name", userRole);

return View(model);

}

[HttpPost]

[Authorize(Roles = "admin")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(EditUserViewModel model)

{

if (ModelState.IsValid)

{

var user = await \_userManager.FindByIdAsync(model.Id);

if (user != null)

{

var existingUser = await \_userManager.FindByEmailAsync(model.Email);

if (existingUser != null && existingUser.Id != user.Id)

{

ModelState.AddModelError("Email", "Пользователь с таким email уже существует");

ViewData["UserRole"] = new SelectList(await \_roleManager.Roles.ToListAsync(), "Name", "Name", model.UserRole);

return View(model);

}

user.Email = model.Email;

user.UserName = model.UserName;

var oldRoles = await \_userManager.GetRolesAsync(user);

await \_userManager.RemoveFromRolesAsync(user, oldRoles);

if (!string.IsNullOrEmpty(model.UserRole))

{

await \_userManager.AddToRoleAsync(user, model.UserRole);

}

var result = await \_userManager.UpdateAsync(user);

if (result.Succeeded)

{

return RedirectToAction("Index");

}

foreach (var error in result.Errors)

{

ModelState.AddModelError(string.Empty, error.Description);

}

}

}

ViewData["UserRole"] = new SelectList(await \_roleManager.Roles.ToListAsync(), "Name", "Name", model.UserRole);

return View(model);

}

[HttpPost]

[Authorize(Roles = "admin")]

public async Task<IActionResult> Delete(string id)

{

var user = await \_userManager.FindByIdAsync(id);

if (user != null)

{

var roles = await \_userManager.GetRolesAsync(user);

if (roles.Contains("user"))

{

await \_userManager.DeleteAsync(user);

}

}

return RedirectToAction("Index");

}

}

}

**Код класса *TicketOrdersContoller.cs***

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Data;

using WebCityEvents.Models;

using WebCityEvents.ViewModels;

namespace WebCityEvents.Controllers

{

public class TicketOrdersController : Controller

{

private readonly EventContext \_context;

public TicketOrdersController(EventContext context)

{

\_context = context;

}

// GET: TicketOrders

public async Task<IActionResult> Index(string customerName = "", string eventName = "", int page = 1, string sortOrder = "OrderDate", string sortDirection = "asc")

{

const int pageSize = 20;

if (string.IsNullOrEmpty(customerName))

{

customerName = Request.Cookies["CustomerName"] ?? string.Empty;

}

if (string.IsNullOrEmpty(eventName))

{

eventName = Request.Cookies["EventName"] ?? string.Empty;

}

Response.Cookies.Append("CustomerName", customerName);

Response.Cookies.Append("EventName", eventName);

var query = \_context.TicketOrders

.Include(t => t.Customer)

.Include(t => t.Event)

.AsQueryable();

if (!string.IsNullOrEmpty(customerName))

{

query = query.Where(t => t.Customer.FullName.Contains(customerName));

}

if (!string.IsNullOrEmpty(eventName))

{

query = query.Where(t => t.Event.EventName.Contains(eventName));

}

query = sortOrder switch

{

"TicketCount" => sortDirection == "asc" ? query.OrderBy(t => t.TicketCount) : query.OrderByDescending(t => t.TicketCount),

\_ => sortDirection == "asc" ? query.OrderBy(t => t.OrderDate) : query.OrderByDescending(t => t.OrderDate),

};

var totalCount = await query.CountAsync();

var orders = await query.Skip((page - 1) \* pageSize)

.Take(pageSize)

.Select(t => new TicketOrderViewModel

{

OrderID = t.OrderID,

CustomerName = t.Customer.FullName,

CustomerPassportData = t.Customer.PassportData,

EventName = t.Event.EventName,

OrderDate = t.OrderDate.Date,

TicketCount = t.TicketCount

})

.ToListAsync();

if (!orders.Any())

{

ViewBag.Message = "По заданным фильтрам ничего не найдено";

}

ViewBag.CurrentPage = page;

ViewBag.TotalPages = (int)Math.Ceiling(totalCount / (double)pageSize);

ViewBag.CustomerName = customerName;

ViewBag.EventName = eventName;

ViewBag.SortOrder = sortOrder;

ViewBag.SortDirection = sortDirection;

return View(orders);

}

// GET: TicketOrders/Details/5

public async Task<IActionResult> Details(int? id)

{

if (id == null)

{

return NotFound();

}

var ticketOrder = await \_context.TicketOrders

.Include(t => t.Customer)

.Include(t => t.Event)

.FirstOrDefaultAsync(m => m.OrderID == id);

if (ticketOrder == null)

{

return NotFound();

}

var model = new TicketOrderViewModel

{

OrderID = ticketOrder.OrderID,

CustomerName = ticketOrder.Customer.FullName,

EventName = ticketOrder.Event.EventName,

OrderDate = ticketOrder.OrderDate.Date,

TicketCount = ticketOrder.TicketCount

};

return View(model);

}

// GET: TicketOrders/Create

public IActionResult Create()

{

var model = new TicketOrderViewModel

{

OrderDate = DateTime.Now

};

ViewBag.CustomerID = new SelectList(\_context.Customers, "CustomerID", "FullName");

return View(model);

}

// POST: TicketOrders/Create

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(TicketOrderViewModel model)

{

var eventItem = await \_context.Events.FindAsync(model.EventID);

if (eventItem == null)

{

ModelState.AddModelError("", "Мероприятие не найдено.");

return View(model);

}

if (model.OrderDate > eventItem.EventDate)

{

ModelState.AddModelError("", "Дата заказа не может быть позже даты проведения мероприятия.");

ViewBag.CustomerID = new SelectList(\_context.Customers, "CustomerID", "FullName");

return View(model);

}

var soldTickets = await \_context.TicketOrders

.Where(o => o.EventID == model.EventID)

.SumAsync(o => o.TicketCount);

var availableTickets = eventItem.TicketAmount - soldTickets;

if (model.TicketCount > availableTickets)

{

ModelState.AddModelError("", "Недостаточно доступных билетов на данное мероприятие.");

ViewBag.CustomerID = new SelectList(\_context.Customers, "CustomerID", "FullName");

return View(model);

}

var newOrder = new TicketOrder

{

EventID = model.EventID,

CustomerID = model.CustomerID,

TicketCount = model.TicketCount,

OrderDate = DateTime.Now

};

\_context.TicketOrders.Add(newOrder);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

// GET: TicketOrders/Edit/5

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

return NotFound();

}

var ticketOrder = await \_context.TicketOrders

.Include(t => t.Event)

.Include(t => t.Customer)

.FirstOrDefaultAsync(m => m.OrderID == id);

if (ticketOrder == null)

{

return NotFound();

}

var model = new TicketOrderViewModel

{

OrderID = ticketOrder.OrderID,

EventID = ticketOrder.EventID,

CustomerID = ticketOrder.CustomerID,

OrderDate = ticketOrder.OrderDate,

TicketCount = ticketOrder.TicketCount,

EventName = ticketOrder.Event.EventName

};

ViewData["CustomerID"] = new SelectList(\_context.Customers, "CustomerID", "FullName", ticketOrder.CustomerID);

ViewData["EventID"] = new SelectList(\_context.Events, "EventID", "EventName", ticketOrder.EventID);

return View(model);

}

// POST: TicketOrders/Edit/5

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int id, [Bind("OrderID,EventID,CustomerID,OrderDate,TicketCount")] TicketOrderViewModel ticketOrderViewModel)

{

if (id != ticketOrderViewModel.OrderID)

{

return NotFound();

}

if (ModelState.IsValid)

{

try

{

var eventItem = await \_context.Events.FindAsync(ticketOrderViewModel.EventID);

if (eventItem == null)

{

return NotFound();

}

if (ticketOrderViewModel.OrderDate > eventItem.EventDate)

{

ModelState.AddModelError("", "Дата заказа не может быть позже даты проведения мероприятия.");

ViewData["CustomerID"] = new SelectList(\_context.Customers, "CustomerID", "FullName", ticketOrderViewModel.CustomerID);

ViewData["EventID"] = new SelectList(\_context.Events, "EventID", "EventName", ticketOrderViewModel.EventID);

return View(ticketOrderViewModel);

}

var totalSoldTickets = await \_context.TicketOrders

.Where(o => o.EventID == ticketOrderViewModel.EventID && o.OrderID != id)

.SumAsync(o => o.TicketCount);

var availableTickets = eventItem.TicketAmount - totalSoldTickets;

if (ticketOrderViewModel.TicketCount > availableTickets)

{

ModelState.AddModelError("", "Недостаточно доступных билетов на данное мероприятие.");

ViewData["CustomerID"] = new SelectList(\_context.Customers, "CustomerID", "FullName", ticketOrderViewModel.CustomerID);

ViewData["EventID"] = new SelectList(\_context.Events, "EventID", "EventName", ticketOrderViewModel.EventID);

return View(ticketOrderViewModel);

}

var ticketOrder = await \_context.TicketOrders.FindAsync(id);

if (ticketOrder == null)

{

return NotFound();

}

ticketOrder.EventID = ticketOrderViewModel.EventID;

ticketOrder.CustomerID = ticketOrderViewModel.CustomerID;

ticketOrder.OrderDate = ticketOrderViewModel.OrderDate;

ticketOrder.TicketCount = ticketOrderViewModel.TicketCount;

\_context.Update(ticketOrder);

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!TicketOrderExists(ticketOrderViewModel.OrderID))

{

return NotFound();

}

else

{

throw;

}

}

return RedirectToAction(nameof(Index));

}

ViewData["CustomerID"] = new SelectList(\_context.Customers, "CustomerID", "FullName", ticketOrderViewModel.CustomerID);

ViewData["EventID"] = new SelectList(\_context.Events, "EventID", "EventName", ticketOrderViewModel.EventID);

return View(ticketOrderViewModel);

}

// GET: TicketOrders/Delete/5

public async Task<IActionResult> Delete(int? id)

{

if (id == null)

{

return NotFound();

}

var ticketOrder = await \_context.TicketOrders

.Include(t => t.Customer)

.Include(t => t.Event)

.FirstOrDefaultAsync(m => m.OrderID == id);

if (ticketOrder == null)

{

return NotFound();

}

var model = new TicketOrderViewModel

{

OrderID = ticketOrder.OrderID,

CustomerName = ticketOrder.Customer.FullName,

EventName = ticketOrder.Event.EventName,

OrderDate = ticketOrder.OrderDate.Date,

TicketCount = ticketOrder.TicketCount,

};

return View(model);

}

// POST: TicketOrders/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

var ticketOrder = await \_context.TicketOrders.FindAsync(id);

if (ticketOrder != null)

{

\_context.TicketOrders.Remove(ticketOrder);

}

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

public bool TicketOrderExists(int id)

{

return \_context.TicketOrders.Any(e => e.OrderID == id);

}

public IActionResult ClearFilters()

{

Response.Cookies.Delete("CustomerName");

Response.Cookies.Delete("EventName");

return RedirectToAction(nameof(Index));

}

[HttpGet]

public async Task<JsonResult> GetAvailableTickets(int eventId)

{

var eventItem = await \_context.Events.FindAsync(eventId);

var soldTickets = await \_context.TicketOrders

.Where(o => o.EventID == eventId)

.SumAsync(o => o.TicketCount);

var availableTickets = eventItem.TicketAmount - soldTickets;

return Json(availableTickets);

}

[HttpGet]

public async Task<IActionResult> SearchEvents(string term)

{

if (string.IsNullOrEmpty(term))

{

return Json(new List<string>());

}

var events = await \_context.Events

.Where(e => e.EventName.StartsWith(term))

.Select(e => new { e.EventID, e.EventName })

.ToListAsync();

return Json(events);

}

[HttpGet]

public async Task<JsonResult> GetEventDetails(int eventId)

{

var eventItem = await \_context.Events

.Where(e => e.EventID == eventId)

.Select(e => new { e.EventDate, e.TicketAmount })

.FirstOrDefaultAsync();

if (eventItem == null)

{

return Json(new { availableTickets = 0, eventDate = (DateTime?)null });

}

var soldTickets = await \_context.TicketOrders

.Where(o => o.EventID == eventId)

.SumAsync(o => o.TicketCount);

var availableTickets = eventItem.TicketAmount - soldTickets;

return Json(new { availableTickets, eventDate = eventItem.EventDate });

}

}

}

**Код класса *PlacesContoller.cs***

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Data;

using WebCityEvents.Models;

using WebCityEvents.ViewModels;

namespace WebCityEvents.Controllers

{

public class PlacesController : Controller

{

private readonly EventContext \_context;

private const int PageSize = 20;

public PlacesController(EventContext context)

{

\_context = context;

}

// GET: Places

public async Task<IActionResult> Index(string searchPlace, int page = 1)

{

if (string.IsNullOrEmpty(searchPlace))

{

searchPlace = Request.Cookies["SearchPlace"] ?? "";

}

if (page <= 0)

{

page = int.TryParse(Request.Cookies["Page"], out int savedPage) ? savedPage : 1;

}

Response.Cookies.Append("SearchPlace", searchPlace, new CookieOptions { Expires = DateTimeOffset.UtcNow.AddDays(1) });

Response.Cookies.Append("Page", page.ToString(), new CookieOptions { Expires = DateTimeOffset.UtcNow.AddDays(1) });

var query = \_context.Places.AsQueryable();

if (!string.IsNullOrEmpty(searchPlace))

{

query = query.Where(p => p.PlaceName.Contains(searchPlace));

}

var totalPlaces = await query.CountAsync();

var totalPages = (int)Math.Ceiling(totalPlaces / (double)PageSize);

var places = await query

.Skip((page - 1) \* PageSize)

.Take(PageSize)

.Select(p => new PlaceViewModel

{

PlaceID = p.PlaceID,

PlaceName = p.PlaceName,

Geolocation = p.Geolocation

})

.ToListAsync();

ViewBag.CurrentPage = page;

ViewBag.TotalPages = totalPages;

ViewBag.SearchPlace = searchPlace;

return View(places);

}

// GET: Places/Details/5

public async Task<IActionResult> Details(int id)

{

var place = await \_context.Places.FindAsync(id);

if (place == null)

{

return NotFound();

}

var placeViewModel = new PlaceViewModel

{

PlaceID = place.PlaceID,

PlaceName = place.PlaceName,

Geolocation = place.Geolocation

};

return View(placeViewModel);

}

// GET: Places/Create

public IActionResult Create()

{

return View();

}

// POST: Places/Create

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(PlaceViewModel model)

{

if (ModelState.IsValid)

{

var place = new Place

{

PlaceName = model.PlaceName,

Geolocation = model.Geolocation

};

\_context.Places.Add(place);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

return View(model);

}

// GET: Places/Edit/5

public async Task<IActionResult> Edit(int id)

{

var place = await \_context.Places.FindAsync(id);

if (place == null)

{

return NotFound();

}

var placeViewModel = new PlaceViewModel

{

PlaceID = place.PlaceID,

PlaceName = place.PlaceName,

Geolocation = place.Geolocation

};

return View(placeViewModel);

}

// POST: Places/Edit/5

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(PlaceViewModel placeViewModel)

{

if (ModelState.IsValid)

{

var place = await \_context.Places.FindAsync(placeViewModel.PlaceID);

if (place == null)

{

return NotFound();

}

place.PlaceName = placeViewModel.PlaceName;

place.Geolocation = placeViewModel.Geolocation;

\_context.Update(place);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

return View(placeViewModel);

}

// GET: Places/Delete/5

public async Task<IActionResult> Delete(int id)

{

var place = await \_context.Places.FindAsync(id);

if (place == null)

{

return NotFound();

}

var placeViewModel = new PlaceViewModel

{

PlaceID = place.PlaceID,

PlaceName = place.PlaceName,

Geolocation = place.Geolocation

};

return View(placeViewModel);

}

// POST: Places/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

var place = await \_context.Places.FindAsync(id);

if (place != null)

{

\_context.Places.Remove(place);

await \_context.SaveChangesAsync();

}

return RedirectToAction(nameof(Index));

}

public bool PlaceExists(int id)

{

return \_context.Places.Any(e => e.PlaceID == id);

}

public IActionResult ClearFilters()

{

Response.Cookies.Delete("SearchPlace");

Response.Cookies.Delete("Page");

return RedirectToAction(nameof(Index));

}

}

}

**Код класса *OrganizersContoller.cs***

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Data;

using WebCityEvents.Models;

using WebCityEvents.ViewModels;

namespace WebCityEvents.Controllers

{

public class OrganizersController : Controller

{

private readonly EventContext \_context;

private const int PageSize = 20;

public OrganizersController(EventContext context)

{

\_context = context;

}

// GET: Organizers

public async Task<IActionResult> Index(string searchName, int page = 1)

{

if (string.IsNullOrEmpty(searchName))

{

searchName = Request.Cookies["SearchName"] ?? "";

}

if (page <= 0)

{

page = int.TryParse(Request.Cookies["Page"], out int savedPage) ? savedPage : 1;

}

Response.Cookies.Append("SearchName", searchName, new CookieOptions { Expires = DateTimeOffset.UtcNow.AddDays(1) });

Response.Cookies.Append("Page", page.ToString(), new CookieOptions { Expires = DateTimeOffset.UtcNow.AddDays(1) });

var query = \_context.Organizers.AsQueryable();

if (!string.IsNullOrEmpty(searchName))

{

query = query.Where(o => o.FullName.Contains(searchName));

}

var totalOrganizers = await query.CountAsync();

var totalPages = (int)Math.Ceiling(totalOrganizers / (double)PageSize);

var organizers = await query

.Skip((page - 1) \* PageSize)

.Take(PageSize)

.Select(o => new OrganizerViewModel

{

OrganizerID = o.OrganizerID,

FullName = o.FullName,

Post = o.Post

})

.ToListAsync();

ViewBag.CurrentPage = page;

ViewBag.TotalPages = totalPages;

ViewBag.SearchName = searchName;

return View(organizers);

}

// GET: Organizers/Details/5

public async Task<IActionResult> Details(int id)

{

var organizer = await \_context.Organizers.FindAsync(id);

if (organizer == null)

{

return NotFound();

}

var organizerViewModel = new OrganizerViewModel

{

OrganizerID = organizer.OrganizerID,

FullName = organizer.FullName,

Post = organizer.Post

};

return View(organizerViewModel);

}

// GET: Organizers/Create

public IActionResult Create()

{

return View();

}

// POST: Organizers/Create

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(OrganizerViewModel model)

{

if (ModelState.IsValid)

{

var organizer = new Organizer

{

FullName = model.FullName,

Post = model.Post

};

\_context.Organizers.Add(organizer);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

return View(model);

}

// GET: Organizers/Edit/5

public async Task<IActionResult> Edit(int id)

{

var organizer = await \_context.Organizers.FindAsync(id);

if (organizer == null)

{

return NotFound();

}

var organizerViewModel = new OrganizerViewModel

{

OrganizerID = organizer.OrganizerID,

FullName = organizer.FullName,

Post = organizer.Post

};

return View(organizerViewModel);

}

// POST: Organizers/Edit/5

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(OrganizerViewModel model)

{

if (ModelState.IsValid)

{

var organizer = await \_context.Organizers.FindAsync(model.OrganizerID);

if (organizer == null)

{

return NotFound();

}

organizer.FullName = model.FullName;

organizer.Post = model.Post;

\_context.Update(organizer);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

return View(model);

}

// GET: Organizers/Delete/5

public async Task<IActionResult> Delete(int id)

{

var organizer = await \_context.Organizers.FindAsync(id);

if (organizer == null)

{

return NotFound();

}

var model = new OrganizerViewModel

{

OrganizerID = organizer.OrganizerID,

FullName = organizer.FullName,

Post = organizer.Post

};

return View(model);

}

// POST: Organizers/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

var organizer = await \_context.Organizers.FindAsync(id);

if (organizer != null)

{

\_context.Organizers.Remove(organizer);

await \_context.SaveChangesAsync();

}

return RedirectToAction(nameof(Index));

}

public bool OrganizerExists(int id)

{

return \_context.Organizers.Any(e => e.OrganizerID == id);

}

public IActionResult ClearFilters()

{

Response.Cookies.Delete("SearchName");

Response.Cookies.Delete("Page");

return RedirectToAction(nameof(Index));

}

}

}

**Код класса *HomeContoller.cs***

using Microsoft.AspNetCore.Mvc;

using WebCityEvents.Services;

namespace WebCityEvents.Controllers

{

public class HomeController : Controller

{

private readonly IOperationService \_operationService;

private const int PageSize = 20;

public HomeController(IOperationService operationService)

{

\_operationService = operationService;

}

[HttpGet]

public IActionResult Index(int numberRows = 10)

{

var model = \_operationService.GetHomeViewModel(numberRows);

return View(model);

}

}

}

**Код класса *EventsContoller.cs***

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Data;

using WebCityEvents.Models;

using WebCityEvents.ViewModels;

namespace WebCityEvents.Controllers

{

public class EventsController : Controller

{

private readonly EventContext \_context;

private const int PageSize = 20;

public EventsController(EventContext context)

{

\_context = context;

}

// GET: Events

public async Task<IActionResult> Index(int page = 1, string eventName = "", string placeName = "", string organizerName = "", DateTime? startDate = null, DateTime? endDate = null, string sortOrder = "", string sortDirection = "asc")

{

eventName = string.IsNullOrEmpty(eventName) ? Request.Cookies["EventName"] : eventName;

placeName = string.IsNullOrEmpty(placeName) ? Request.Cookies["PlaceName"] : placeName;

organizerName = string.IsNullOrEmpty(organizerName) ? Request.Cookies["OrganizerName"] : organizerName;

startDate ??= DateTime.TryParse(Request.Cookies["StartDate"], out DateTime start) ? start : (DateTime?)null;

endDate ??= DateTime.TryParse(Request.Cookies["EndDate"], out DateTime end) ? end : (DateTime?)null;

Response.Cookies.Append("EventName", eventName ?? "");

Response.Cookies.Append("PlaceName", placeName ?? "");

Response.Cookies.Append("OrganizerName", organizerName ?? "");

Response.Cookies.Append("StartDate", startDate?.ToString("yyyy-MM-dd") ?? "");

Response.Cookies.Append("EndDate", endDate?.ToString("yyyy-MM-dd") ?? "");

var eventsQuery = \_context.Events

.Include(e => e.Organizer)

.Include(e => e.Place)

.AsQueryable();

if (!string.IsNullOrEmpty(eventName))

eventsQuery = eventsQuery.Where(e => e.EventName.ToLower() == eventName.ToLower());

if (!string.IsNullOrEmpty(placeName))

eventsQuery = eventsQuery.Where(e => e.Place.PlaceName.ToLower() == placeName.ToLower());

if (!string.IsNullOrEmpty(organizerName))

eventsQuery = eventsQuery.Where(e => e.Organizer.FullName.ToLower() == organizerName.ToLower());

if (startDate.HasValue)

eventsQuery = eventsQuery.Where(e => e.EventDate >= startDate.Value);

if (endDate.HasValue)

eventsQuery = eventsQuery.Where(e => e.EventDate <= endDate.Value);

eventsQuery = sortOrder switch

{

"Tickets" => sortDirection == "asc" ? eventsQuery.OrderBy(e => e.TicketAmount) : eventsQuery.OrderByDescending(e => e.TicketAmount),

"Price" => sortDirection == "asc" ? eventsQuery.OrderBy(e => e.TicketPrice) : eventsQuery.OrderByDescending(e => e.TicketPrice),

"Date" => sortDirection == "asc" ? eventsQuery.OrderBy(e => e.EventDate) : eventsQuery.OrderByDescending(e => e.EventDate),

\_ => eventsQuery.OrderBy(e => e.EventID)

};

var totalEvents = await eventsQuery.CountAsync();

if (totalEvents == 0)

{

ViewBag.Message = "По вашему запросу ничего не найдено";

}

var totalPages = (int)Math.Ceiling(totalEvents / (double)PageSize);

var events = await eventsQuery

.Skip((page - 1) \* PageSize)

.Take(PageSize)

.Select(e => new EventViewModel

{

EventID = e.EventID,

EventName = e.EventName,

PlaceName = e.Place.PlaceName,

OrganizerName = e.Organizer.FullName,

EventDate = e.EventDate,

TicketPrice = e.TicketPrice,

TicketAmount = e.TicketAmount,

AvailableTickets = e.TicketAmount - \_context.TicketOrders

.Where(o => o.EventID == e.EventID)

.Sum(o => o.TicketCount)

})

.ToListAsync();

ViewBag.CurrentPage = page;

ViewBag.TotalPages = totalPages;

ViewBag.EventName = eventName;

ViewBag.PlaceName = placeName;

ViewBag.OrganizerName = organizerName;

ViewBag.StartDate = startDate?.ToString("yyyy-MM-dd");

ViewBag.EndDate = endDate?.ToString("yyyy-MM-dd");

ViewBag.SortOrder = sortOrder;

ViewBag.SortDirection = sortDirection;

return View(events);

}

// GET: Events/Details/5

public async Task<IActionResult> Details(int? id)

{

if (id == null)

{

return NotFound();

}

var eventDetails = await \_context.Events

.Include(e => e.Organizer)

.Include(e => e.Place)

.Where(e => e.EventID == id)

.Select(e => new EventViewModel

{

EventID = e.EventID,

EventName = e.EventName,

PlaceName = e.Place.PlaceName,

OrganizerName = e.Organizer.FullName,

EventDate = e.EventDate,

TicketPrice = e.TicketPrice,

TicketAmount = e.TicketAmount

})

.FirstOrDefaultAsync();

if (eventDetails == null)

{

return NotFound();

}

return View(eventDetails);

}

// GET: Events/Create

public IActionResult Create()

{

ViewData["OrganizerName"] = new SelectList(\_context.Organizers, "OrganizerID", "FullName");

ViewData["PlaceName"] = new SelectList(\_context.Places, "PlaceID", "PlaceName");

return View();

}

// POST: Events/Create

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(EventViewModel model)

{

if (ModelState.IsValid)

{

var newEvent = new Event

{

EventName = model.EventName,

PlaceID = model.PlaceID,

EventDate = model.EventDate,

TicketPrice = model.TicketPrice,

TicketAmount = model.TicketAmount,

OrganizerID = model.OrganizerID

};

\_context.Add(newEvent);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

ViewData["OrganizerName"] = new SelectList(\_context.Organizers, "OrganizerID", "FullName", model.OrganizerID);

ViewData["PlaceName"] = new SelectList(\_context.Places, "PlaceID", "PlaceName", model.PlaceID);

return View(model);

}

// GET: Events/Edit/5

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

return NotFound();

}

var eventDetails = await \_context.Events

.Include(e => e.Organizer)

.Include(e => e.Place)

.Where(e => e.EventID == id)

.Select(e => new EventViewModel

{

EventID = e.EventID,

EventName = e.EventName,

PlaceID = e.PlaceID,

PlaceName = e.Place.PlaceName,

OrganizerID = e.OrganizerID,

OrganizerName = e.Organizer.FullName,

EventDate = e.EventDate,

TicketPrice = e.TicketPrice,

TicketAmount = e.TicketAmount

})

.FirstOrDefaultAsync();

if (eventDetails == null)

{

return NotFound();

}

ViewData["OrganizerID"] = new SelectList(\_context.Organizers, "OrganizerID", "FullName", eventDetails.OrganizerID);

ViewData["PlaceID"] = new SelectList(\_context.Places, "PlaceID", "PlaceName", eventDetails.PlaceID);

return View(eventDetails);

}

// POST: Events/Edit/5

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int id, EventViewModel model)

{

if (id != model.EventID)

{

return NotFound();

}

if (ModelState.IsValid)

{

try

{

var eventToUpdate = await \_context.Events.FindAsync(id);

if (eventToUpdate == null)

{

return NotFound();

}

eventToUpdate.EventName = model.EventName;

eventToUpdate.PlaceID = model.PlaceID;

eventToUpdate.EventDate = model.EventDate;

eventToUpdate.TicketPrice = model.TicketPrice;

eventToUpdate.TicketAmount = model.TicketAmount;

eventToUpdate.OrganizerID = model.OrganizerID;

\_context.Update(eventToUpdate);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

catch (DbUpdateConcurrencyException)

{

}

}

ViewData["OrganizerID"] = new SelectList(\_context.Organizers, "OrganizerID", "FullName", model.OrganizerID);

ViewData["PlaceID"] = new SelectList(\_context.Places, "PlaceID", "PlaceName", model.PlaceID);

return View(model);

}

// GET: Events/Delete/5

public async Task<IActionResult> Delete(int? id)

{

if (id == null)

{

return NotFound();

}

var eventToDelete = await \_context.Events

.Include(e => e.Organizer)

.Include(e => e.Place)

.FirstOrDefaultAsync(m => m.EventID == id);

if (eventToDelete == null)

{

return NotFound();

}

var eventViewModel = new EventViewModel

{

EventID = eventToDelete.EventID,

EventName = eventToDelete.EventName,

PlaceName = eventToDelete.Place.PlaceName,

OrganizerName = eventToDelete.Organizer.FullName,

EventDate = eventToDelete.EventDate,

TicketPrice = eventToDelete.TicketPrice,

TicketAmount = eventToDelete.TicketAmount

};

return View(eventViewModel);

}

// POST: Events/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

var @event = await \_context.Events.FindAsync(id);

if (@event != null)

{

\_context.Events.Remove(@event);

}

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

public bool EventExists(int id)

{

return \_context.Events.Any(e => e.EventID == id);

}

public IActionResult ClearFilters()

{

Response.Cookies.Delete("EventName");

Response.Cookies.Delete("PlaceName");

Response.Cookies.Delete("OrganizerName");

Response.Cookies.Delete("StartDate");

Response.Cookies.Delete("EndDate");

return RedirectToAction(nameof(Index));

}

public async Task<IActionResult> SearchEventNames(string term)

{

var events = await \_context.Events

.Where(e => e.EventName.Contains(term))

.Select(e => e.EventName)

.Distinct()

.Take(5)

.ToListAsync();

return Json(events);

}

public async Task<IActionResult> SearchPlaceNames(string term)

{

var places = await \_context.Events

.Where(e => e.Place.PlaceName.Contains(term))

.Select(e => e.Place.PlaceName)

.Distinct()

.Take(5)

.ToListAsync();

return Json(places);

}

public async Task<IActionResult> SearchOrganizerNames(string term)

{

var organizers = await \_context.Events

.Where(e => e.Organizer.FullName.Contains(term))

.Select(e => e.Organizer.FullName)

.Distinct()

.Take(5)

.ToListAsync();

return Json(organizers);

}

}

}

**Код класса *CustomersContoller.cs***

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using WebCityEvents.Data;

using WebCityEvents.Models;

using WebCityEvents.ViewModels;

namespace WebCityEvents.Controllers

{

public class CustomersController : Controller

{

private readonly EventContext \_context;

private const int PageSize = 20;

public CustomersController(EventContext context)

{

\_context = context;

}

// GET: Customers

public async Task<IActionResult> Index(string searchName, int page = 1)

{

searchName ??= HttpContext.Session.GetString("SearchName") ?? "";

page = page <= 0 ? HttpContext.Session.GetInt32("Page") ?? 1 : page;

HttpContext.Session.SetString("SearchName", searchName);

HttpContext.Session.SetInt32("Page", page);

var query = \_context.Customers.AsQueryable();

if (!string.IsNullOrEmpty(searchName))

{

query = query.Where(c => c.FullName.Contains(searchName));

}

var totalCustomers = await query.CountAsync();

var totalPages = (int)Math.Ceiling(totalCustomers / (double)PageSize);

var customers = await query

.Skip((page - 1) \* PageSize)

.Take(PageSize)

.Select(c => new CustomerViewModel

{

CustomerID = c.CustomerID,

FullName = c.FullName,

PassportData = c.PassportData

})

.ToListAsync();

ViewBag.CurrentPage = page;

ViewBag.TotalPages = totalPages;

ViewBag.SearchName = searchName;

return View(customers);

}

// GET: Customers/Details/5

public async Task<IActionResult> Details(int id)

{

var customer = await \_context.Customers.FindAsync(id);

if (customer == null)

{

return NotFound();

}

var customerViewModel = new CustomerViewModel

{

CustomerID = customer.CustomerID,

FullName = customer.FullName,

PassportData = customer.PassportData

};

return View(customerViewModel);

}

// GET: Customers/Create

public IActionResult Create()

{

return View();

}

// POST: Customers/Create

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Create(CustomerViewModel model)

{

if (ModelState.IsValid)

{

var customer = new Customer

{

FullName = model.FullName,

PassportData = model.PassportData

};

\_context.Customers.Add(customer);

\_context.SaveChanges();

return RedirectToAction(nameof(Index));

}

return View(model);

}

// GET: Customers/Edit/5

public async Task<IActionResult> Edit(int id)

{

var customer = await \_context.Customers.FindAsync(id);

if (customer == null)

{

return NotFound();

}

var customerViewModel = new CustomerViewModel

{

CustomerID = customer.CustomerID,

FullName = customer.FullName,

PassportData = customer.PassportData

};

return View(customerViewModel);

}

// POST: Customers/Edit/5

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(CustomerViewModel customerViewModel)

{

if (ModelState.IsValid)

{

var customer = await \_context.Customers.FindAsync(customerViewModel.CustomerID);

if (customer == null)

{

return NotFound();

}

customer.FullName = customerViewModel.FullName;

customer.PassportData = customerViewModel.PassportData;

\_context.Update(customer);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

return View(customerViewModel);

}

// GET: Customers/Delete/5

public IActionResult Delete(int id)

{

var customer = \_context.Customers.Find(id);

if (customer == null)

{

return NotFound();

}

var model = new CustomerViewModel

{

CustomerID = customer.CustomerID,

FullName = customer.FullName,

PassportData = customer.PassportData

};

return View(model);

}

// POST: Customers/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public IActionResult DeleteConfirmed(int id)

{

var customer = \_context.Customers.Find(id);

if (customer != null)

{

\_context.Customers.Remove(customer);

\_context.SaveChanges();

}

return RedirectToAction(nameof(Index));

}

public bool CustomerExists(int id)

{

return \_context.Customers.Any(e => e.CustomerID == id);

}

public IActionResult ClearSession()

{

HttpContext.Session.Clear();

return RedirectToAction(nameof(Index));

}

}

}

**Код схемы *Customers/Create.cshtml***

@model WebCityEvents.ViewModels.CustomerViewModel

@{

ViewData["Title"] = "Создать";

}

<h1>Создать</h1>

<h4>Заказчик</h4>

<hr />

<div class="row">

<div class="col-md-4">

<form asp-action="Create">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<div class="form-group">

<label asp-for="FullName" class="control-label">ФИО</label>

<input asp-for="FullName" class="form-control" />

<span asp-validation-for="FullName" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="PassportData" class="control-label">Данные паспорта</label>

<input asp-for="PassportData" class="form-control" />

<span asp-validation-for="PassportData" class="text-danger"></span>

</div>

<div class="form-group">

<input type="submit" value="Создать" class="btn btn-primary" />

</div>

</form>

</div>

</div>

<div>

<a asp-action="Index">Назад к списку</a>

</div>

@section Scripts {

@{

await Html.RenderPartialAsync("\_ValidationScriptsPartial");

}

}

**Код схемы *Events/Details.cshtml***

@model WebCityEvents.ViewModels.EventViewModel

@{

ViewData["Title"] = "Детали мероприятия";

}

<h1>Детали мероприятия</h1>

<div>

<h4>Мероприятие</h4>

<hr />

<dl class="row">

<dt class="col-sm-2">

Название мероприятия

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.EventName)

</dd>

<dt class="col-sm-2">

Дата проведения

</dt>

<dd class="col-sm-10">

@Model.EventDate.ToString("dd.MM.yyyy")

</dd>

<dt class="col-sm-2">

Цена билета

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.TicketPrice)

</dd>

<dt class="col-sm-2">

Количество билетов

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.TicketAmount)

</dd>

<dt class="col-sm-2">

Место проведения

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.PlaceName)

</dd>

<dt class="col-sm-2">

Организатор

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.OrganizerName)

</dd>

</dl>

</div>

<div>

<a asp-action="Edit" asp-route-id="@Model.EventID">Редактировать</a> |

<a asp-action="Index">Назад к списку</a>

</div>

**Код схемы *Organizers/Edit.cshtml***

@model WebCityEvents.ViewModels.OrganizerViewModel

@{

ViewData["Title"] = "Редактирование";

}

<h1>Редактирование организатора</h1>

<h4>Организатор</h4>

<hr />

<div class="row">

<div class="col-md-4">

<form asp-action="Edit">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<input type="hidden" asp-for="OrganizerID" />

<div class="form-group">

<label asp-for="FullName" class="control-label">ФИО</label>

<input asp-for="FullName" class="form-control" />

<span asp-validation-for="FullName" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="Post" class="control-label">Должность</label>

<input asp-for="Post" class="form-control" />

<span asp-validation-for="Post" class="text-danger"></span>

</div>

<div class="form-group">

<input type="submit" value="Сохранить" class="btn btn-primary" />

</div>

</form>

</div>

</div>

<div>

<a asp-action="Index">Назад к списку</a>

</div>

@section Scripts {

@{

await Html.RenderPartialAsync("\_ValidationScriptsPartial");

}

}

**Код схемы *Places/Index.cshtml***

@model IEnumerable<WebCityEvents.ViewModels.PlaceViewModel>

@{

ViewData["Title"] = "Места";

}

<h2>Места</h2>

<a asp-action="Create" class="btn btn-success mb-3">Создать новое место</a>

<form asp-action="Index" method="get" class="mb-3">

<div class="input-group">

<input type="text" name="searchPlace" value="@ViewBag.SearchPlace" class="form-control" placeholder="Поиск по названию места">

<button type="submit" class="btn btn-primary">Поиск</button>

</div>

</form>

<form asp-action="ClearFilters" method="post" class="mb-3">

<button type="submit" class="btn btn-danger">Сбросить поиск</button>

</form>

@if (!Model.Any())

{

<p class="text-danger">Места не найдены</p>

}

else

{

<table class="table table-bordered">

<thead>

<tr>

<th>Название места</th>

<th>Геолокация</th>

<th>Действия</th>

</tr>

</thead>

<tbody>

@foreach (var place in Model)

{

<tr>

<td>@Html.DisplayFor(modelItem => place.PlaceName)</td>

<td>@Html.DisplayFor(modelItem => place.Geolocation)</td>

<td>

<a asp-action="Edit" asp-route-id="@place.PlaceID">Редактировать</a> |

<a asp-action="Details" asp-route-id="@place.PlaceID">Подробнее</a> |

<a asp-action="Delete" asp-route-id="@place.PlaceID">Удалить</a>

</td>

</tr>

}

</tbody>

</table>

<div class="d-flex justify-content-between align-items-center">

<span>Страница @ViewBag.CurrentPage из @ViewBag.TotalPages</span>

<div>

<a asp-action="Index" asp-route-page="@(ViewBag.CurrentPage - 1)"

asp-route-searchPlace="@ViewBag.SearchPlace"

class="btn btn-primary me-2"

disabled="@(ViewBag.CurrentPage == 1 ? "disabled" : null)">

Предыдущая

</a>

<form asp-action="Index" method="get" class="d-inline">

<input type="hidden" name="searchPlace" value="@ViewBag.SearchPlace" />

<input type="number" name="page" min="1" max="@ViewBag.TotalPages" value="@ViewBag.CurrentPage" class="form-control d-inline w-25 me-2" />

<button type="submit" class="btn btn-primary">Перейти</button>

</form>

<a asp-action="Index" asp-route-page="@(ViewBag.CurrentPage + 1)"

asp-route-searchPlace="@ViewBag.SearchPlace"

class="btn btn-primary"

disabled="@(ViewBag.CurrentPage == ViewBag.TotalPages ? "disabled" : null)">

Следующая

</a>

</div>

</div>

}

**Код схемы *TicketOrders/Delete.cshtml***

@model WebCityEvents.ViewModels.TicketOrderViewModel

@{

ViewData["Title"] = "Удалить";

}

<h1>Удалить</h1>

<h3>Вы уверены, что хотите удалить это?</h3>

<div>

<h4>Заказ билетов</h4>

<hr />

<dl class="row">

<dt class="col-sm-2">Дата заказа</dt>

<dd class="col-sm-10">

@Model.OrderDate.ToString("dd.MM.yyyy")

</dd>

<dt class="col-sm-2">Количество билетов</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.TicketCount)

</dd>

<dt class="col-sm-2">Название мероприятия</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.EventName)

</dd>

<dt class="col-sm-2">ФИО заказчика</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.CustomerName)

</dd>

</dl>

<form asp-action="Delete">

<input type="hidden" asp-for="OrderID" />

<input type="submit" value="Удалить" class="btn btn-danger" /> |

<a asp-action="Index">Назад к списку</a>

</form>

</div>

**Код схемы *User/Login.cshtml***

@model WebCityEvents.ViewModels.LoginViewModel

@{

ViewData["Title"] = "Вход";

}

<h2>Вход</h2>

<form asp-action="Login" method="post">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<div class="form-group">

<label asp-for="Email"></label>

<input asp-for="Email" class="form-control" />

<span asp-validation-for="Email" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="Password"></label>

<input asp-for="Password" class="form-control" />

<span asp-validation-for="Password" class="text-danger"></span>

</div>

<div class="form-group form-check">

<input asp-for="RememberMe" class="form-check-input" />

<label asp-for="RememberMe" class="form-check-label"></label>

</div>

<button type="submit" class="btn btn-primary">Войти</button>

</form>

**ПРИЛОЖЕНИЕ В**

(обязательное)

**ЛИСТИНГ МОДУЛЬНЫХ ТЕСТОВ**

**Код класса *TestDataHelper***

using WebCityEvents.Models;

namespace WebCityEvents.Tests

{

internal static class TestDataHelper

{

public static List<Customer> GetFakeCustomersList()

{

return new List<Customer>

{

new Customer { CustomerID = 1, FullName = "John Doe", PassportData = "AB123456" },

new Customer { CustomerID = 2, FullName = "Jane Doe", PassportData = "CD789012" },

new Customer { CustomerID = 3, FullName = "Sam Smith", PassportData = "EF345678" }

};

}

public static List<Event> GetFakeEventsList()

{

return new List<Event>

{

new Event

{

EventID = 1,

EventName = "Concert",

PlaceID = 1,

EventDate = new DateTime(2024, 11, 15),

TicketPrice = 100,

TicketAmount = 200,

OrganizerID = 1,

Place = new Place

{

PlaceID = 1,

PlaceName = "Main Hall"

},

Organizer = new Organizer

{

OrganizerID = 1,

FullName = "John Smith"

}

},

new Event

{

EventID = 2,

EventName = "Art Exhibition",

PlaceID = 2,

EventDate = new DateTime(2024, 12, 1),

TicketPrice = 50,

TicketAmount = 150,

OrganizerID = 2,

Place = new Place

{

PlaceID = 2,

PlaceName = "Gallery"

},

Organizer = new Organizer

{

OrganizerID = 2,

FullName = "Art Inc."

}

}

};

}

public static List<TicketOrder> GetFakeTicketOrdersList()

{

return new List<TicketOrder>

{

new TicketOrder

{

OrderID = 1,

EventID = 1,

CustomerID = 1,

TicketCount = 2,

OrderDate = DateTime.Now.AddDays(-10),

Event = GetFakeEventsList().First(e => e.EventID == 1),

Customer = GetFakeCustomersList().First(c => c.CustomerID == 1)

},

new TicketOrder

{

OrderID = 2,

EventID = 2,

CustomerID = 2,

TicketCount = 3,

OrderDate = DateTime.Now.AddDays(-5),

Event = GetFakeEventsList().First(e => e.EventID == 2),

Customer = GetFakeCustomersList().First(c => c.CustomerID == 2)

}

};

}

public static List<Place> GetFakePlacesList()

{

return new List<Place>

{

new Place { PlaceID = 1, PlaceName = "Main Hall", Geolocation = "50.123, 30.567" },

new Place { PlaceID = 2, PlaceName = "Gallery", Geolocation = "50.456, 30.789" },

new Place { PlaceID = 3, PlaceName = "Open Air Stage", Geolocation = "50.789, 30.101" }

};

}

public static List<Organizer> GetFakeOrganizersList()

{

return new List<Organizer>

{

new Organizer

{

OrganizerID = 1,

FullName = "John Smith",

Post = "Manager"

},

new Organizer

{

OrganizerID = 2,

FullName = "Art Inc.",

Post = "Director"

},

new Organizer

{

OrganizerID = 3,

FullName = "Jane Doe",

Post = "Coordinator"

}

};

}

}

}

**Код класса *EventsControllerTests***

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace WebCityEvents.Tests

{

public class EventsControllerTests

{

private readonly DbContextOptions<EventContext> \_options;

public EventsControllerTests()

{

\_options = new DbContextOptionsBuilder<EventContext>()

.UseInMemoryDatabase(databaseName: "TestEventDatabase")

.Options;

}

private EventContext CreateContext()

{

return new EventContext(\_options);

}

private EventsController CreateController(EventContext context)

{

return new EventsController(context);

}

private void SeedDatabase(EventContext context)

{

context.Database.EnsureDeleted();

context.Database.EnsureCreated();

context.Events.AddRange(TestDataHelper.GetFakeEventsList());

context.SaveChanges();

}

private EventsController CreateControllerWithSession(EventContext context)

{

var controller = new EventsController(context);

var httpContext = new DefaultHttpContext

{

Session = new MockHttpSession()

};

controller.ControllerContext = new ControllerContext

{

HttpContext = httpContext

};

return controller;

}

[Fact]

public async Task Index\_ReturnsAllEvents()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

var result = await controller.Index() as ViewResult;

var model = Assert.IsAssignableFrom<List<EventViewModel>>(result.Model);

Assert.Equal(2, model.Count);

}

[Fact]

public async Task Index\_FiltersEvents()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

var result = await controller.Index(eventName: "Art Exhibition") as ViewResult;

var model = Assert.IsAssignableFrom<List<EventViewModel>>(result.Model);

Assert.Single(model);

Assert.Equal("Art Exhibition", model.First().EventName);

}

[Fact]

public async Task Details\_ReturnsEventViewModel()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

var result = await controller.Details(1) as ViewResult;

Assert.NotNull(result);

var model = Assert.IsType<EventViewModel>(result.Model);

Assert.Equal(1, model.EventID);

Assert.Equal("Concert", model.EventName);

}

[Fact]

public async Task Details\_ReturnsNotFound()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

var result = await controller.Details(999);

Assert.IsType<NotFoundResult>(result);

}

[Fact]

public void Create\_ReturnsViewResult()

{

using var context = CreateContext();

var controller = new EventsController(context);

var result = controller.Create();

Assert.IsType<ViewResult>(result);

}

[Fact]

public async Task Create\_AddsNewEvent()

{

using var context = CreateContext();

var controller = new EventsController(context);

var newEvent = new EventViewModel

{

EventName = "New Event",

PlaceID = 1,

EventDate = DateTime.UtcNow,

TicketPrice = 150,

TicketAmount = 100,

OrganizerID = 1

};

var result = await controller.Create(newEvent);

var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);

Assert.Equal(nameof(controller.Index), redirectToActionResult.ActionName);

var createdEvent = context.Events.SingleOrDefault(e => e.EventName == "New Event");

Assert.NotNull(createdEvent);

Assert.Equal(150, createdEvent.TicketPrice);

}

[Fact]

public async Task Create\_ReturnsViewWithModelError()

{

using var context = CreateContext();

var controller = new EventsController(context);

var newEvent = new EventViewModel

{

EventName = "",

PlaceID = 1,

EventDate = DateTime.UtcNow,

TicketPrice = 150,

TicketAmount = 100,

OrganizerID = 1

};

controller.ModelState.AddModelError("EventName", "Event name is required");

var result = await controller.Create(newEvent);

var viewResult = Assert.IsType<ViewResult>(result);

var model = Assert.IsType<EventViewModel>(viewResult.Model);

Assert.Equal(newEvent, model);

}

[Fact]

public async Task Edit\_ReturnsEventViewModel()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

var result = await controller.Edit(1) as ViewResult;

var model = Assert.IsType<EventViewModel>(result.Model);

Assert.Equal("Concert", model.EventName);

}

[Fact]

public async Task Edit\_UpdatesEvent()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

var updatedEvent = new EventViewModel

{

EventID = 1,

EventName = "Updated Concert",

TicketPrice = 100

};

var result = await controller.Edit(1, updatedEvent);

var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);

Assert.Equal(nameof(controller.Index), redirectToActionResult.ActionName);

var eventToUpdate = await context.Events.FindAsync(1);

Assert.Equal("Updated Concert", eventToUpdate.EventName);

}

[Fact]

public async Task Edit\_ReturnsNotFound()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

var updatedEvent = new EventViewModel

{

EventID = 999,

EventName = "Non-existent Event"

};

var result = await controller.Edit(999, updatedEvent);

Assert.IsType<NotFoundResult>(result);

}

[Fact]

public async Task Delete\_ReturnsEventForDelete()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

var result = await controller.Delete(1) as ViewResult;

var model = Assert.IsType<EventViewModel>(result.Model);

Assert.Equal("Concert", model.EventName);

}

[Fact]

public async Task Delete\_RemovesEvent()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

var result = await controller.DeleteConfirmed(1);

var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);

Assert.Equal(nameof(controller.Index), redirectToActionResult.ActionName);

var deletedEvent = context.Events.Find(1);

Assert.Null(deletedEvent);

}

[Fact]

public void EventExists\_ReturnsTrue()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new EventsController(context);

bool exists = controller.EventExists(1);

Assert.True(exists);

}

[Fact]

public void ClearFilters\_ClearsCookies()

{

using var context = CreateContext();

var controller = new EventsController(context);

controller.ControllerContext = new ControllerContext

{

HttpContext = new DefaultHttpContext()

};

controller.HttpContext.Response.Cookies.Append("EventName", "TestEvent");

var result = controller.ClearFilters();

Assert.Null(controller.HttpContext.Request.Cookies["EventName"]);

var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);

Assert.Equal(nameof(controller.Index), redirectToActionResult.ActionName);

}

[Fact]

public async Task SearchEventNames\_ReturnsMatching()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateController(context);

var result = await controller.SearchEventNames("Concert") as JsonResult;

Assert.NotNull(result);

var events = Assert.IsType<List<string>>(result.Value);

Assert.Single(events);

Assert.Contains("Concert", events);

}

[Fact]

public async Task SearchPlaceNames\_ReturnsMatching()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateController(context);

var result = await controller.SearchPlaceNames("Gallery") as JsonResult;

Assert.NotNull(result);

var places = Assert.IsType<List<string>>(result.Value);

Assert.Single(places);

Assert.Contains("Gallery", places);

}

[Fact]

public async Task SearchOrganizerNames\_ReturnsMatching()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateController(context);

var result = await controller.SearchOrganizerNames("John Smith") as JsonResult;

Assert.NotNull(result);

var organizers = Assert.IsType<List<string>>(result.Value);

Assert.Single(organizers);

Assert.Contains("John Smith", organizers);

}

}

}

**Код класса *TicketOrdersControllerTests***

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using Microsoft.EntityFrameworkCore;

namespace WebCityEvents.Tests

{

public class TicketOrdersControllerTests

{

private readonly DbContextOptions<EventContext> \_options;

public TicketOrdersControllerTests()

{

\_options = new DbContextOptionsBuilder<EventContext>()

.UseInMemoryDatabase(databaseName: "TestTicketOrderDatabase")

.Options;

}

private EventContext CreateContext()

{

return new EventContext(\_options);

}

private void SeedDatabase(EventContext context)

{

context.Database.EnsureDeleted();

context.Database.EnsureCreated();

context.TicketOrders.AddRange(TestDataHelper.GetFakeTicketOrdersList());

context.SaveChanges();

}

private TicketOrdersController CreateControllerWithSession(EventContext context)

{

var controller = new TicketOrdersController(context);

var httpContext = new DefaultHttpContext

{

Session = new MockHttpSession()

};

controller.ControllerContext = new ControllerContext

{

HttpContext = httpContext

};

return controller;

}

[Fact]

public async Task Index\_ReturnsAllTicketOrders()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

var result = await controller.Index() as ViewResult;

var model = Assert.IsAssignableFrom<List<TicketOrderViewModel>>(result.Model);

Assert.Equal(2, model.Count);

}

[Fact]

public async Task Index\_FiltersTicketOrders()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

var result = await controller.Index(customerName: "John Doe") as ViewResult;

var model = Assert.IsAssignableFrom<List<TicketOrderViewModel>>(result.Model);

Assert.Single(model);

Assert.Equal("John Doe", model.First().CustomerName);

}

[Fact]

public async Task Details\_ReturnsTicketOrderViewModel()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new TicketOrdersController(context);

var result = await controller.Details(1) as ViewResult;

Assert.NotNull(result);

var model = Assert.IsType<TicketOrderViewModel>(result.Model);

Assert.Equal(1, model.OrderID);

Assert.Equal("Concert", model.EventName);

}

[Fact]

public async Task Details\_ReturnsNotFound()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new TicketOrdersController(context);

var result = await controller.Details(999);

Assert.IsType<NotFoundResult>(result);

}

[Fact]

public void Create\_ReturnsViewResult()

{

using var context = CreateContext();

var controller = new TicketOrdersController(context);

var result = controller.Create();

Assert.IsType<ViewResult>(result);

}

[Fact]

public async Task Create\_ReturnsViewWithModelError()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

var model = new TicketOrderViewModel

{

EventID = 999,

CustomerID = 1,

TicketCount = 2,

OrderDate = DateTime.Now

};

var result = await controller.Create(model);

var viewResult = Assert.IsType<ViewResult>(result);

Assert.True(controller.ModelState.ContainsKey(""));

Assert.Equal("Мероприятие не найдено.", controller.ModelState[""].Errors.First().ErrorMessage);

}

[Fact]

public async Task Edit\_ReturnsTicketOrderForEdit()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

int existingOrderId = 1;

var result = await controller.Edit(existingOrderId);

var viewResult = Assert.IsType<ViewResult>(result);

var model = Assert.IsType<TicketOrderViewModel>(viewResult.Model);

Assert.Equal(existingOrderId, model.OrderID);

Assert.Equal(context.TicketOrders.First(o => o.OrderID == existingOrderId).EventID, model.EventID);

Assert.Equal(context.TicketOrders.First(o => o.OrderID == existingOrderId).CustomerID, model.CustomerID);

Assert.Equal(context.TicketOrders.First(o => o.OrderID == existingOrderId).TicketCount, model.TicketCount);

Assert.Equal(context.TicketOrders.First(o => o.OrderID == existingOrderId).Event.EventName, model.EventName);

var customerSelectList = Assert.IsType<SelectList>(viewResult.ViewData["CustomerID"]);

var eventSelectList = Assert.IsType<SelectList>(viewResult.ViewData["EventID"]);

Assert.Equal(model.CustomerID, customerSelectList.SelectedValue);

Assert.Equal(model.EventID, eventSelectList.SelectedValue);

}

[Fact]

public async Task Edit\_ReturnsNotFound()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new TicketOrdersController(context);

var updatedOrder = new TicketOrderViewModel

{

OrderID = 999,

CustomerName = "Non-existent Customer"

};

var result = await controller.Edit(999, updatedOrder);

Assert.IsType<NotFoundResult>(result);

}

[Fact]

public async Task Delete\_ReturnsTicketOrderForDelete()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new TicketOrdersController(context);

var result = await controller.Delete(1) as ViewResult;

var model = Assert.IsType<TicketOrderViewModel>(result.Model);

Assert.Equal("Concert", model.EventName);

}

[Fact]

public async Task Delete\_RemovesTicketOrder()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = new TicketOrdersController(context);

var result = await controller.DeleteConfirmed(1);

var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);

Assert.Equal(nameof(controller.Index), redirectToActionResult.ActionName);

var deletedOrder = context.TicketOrders.Find(1);

Assert.Null(deletedOrder);

}

[Fact]

public void TicketOrderExists\_ReturnsTrue()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

int existingOrderId = 1;

bool exists = controller.TicketOrderExists(existingOrderId);

Assert.True(exists);

}

[Fact]

public void ClearFilters\_ClearsCookies()

{

using var context = CreateContext();

var controller = new TicketOrdersController(context);

controller.ControllerContext = new ControllerContext

{

HttpContext = new DefaultHttpContext()

};

controller.HttpContext.Response.Cookies.Append("CustomerName", "TestCustomer");

var result = controller.ClearFilters();

Assert.Null(controller.HttpContext.Request.Cookies["CustomerName"]);

var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);

Assert.Equal(nameof(controller.Index), redirectToActionResult.ActionName);

}

[Fact]

public async Task GetAvailableTickets\_ReturnsAvailableTickets()

{

using var context = CreateContext();

SeedDatabase(context);

var controller = CreateControllerWithSession(context);

int eventId = 1;

var result = await controller.GetAvailableTickets(eventId);

var jsonResult = Assert.IsType<JsonResult>(result);

var availableTickets = Assert.IsType<int>(jsonResult.Value);

Assert.True(availableTickets >= 0);

}

}

}