Лабораторная работа №1
Разработка каркаса приложения

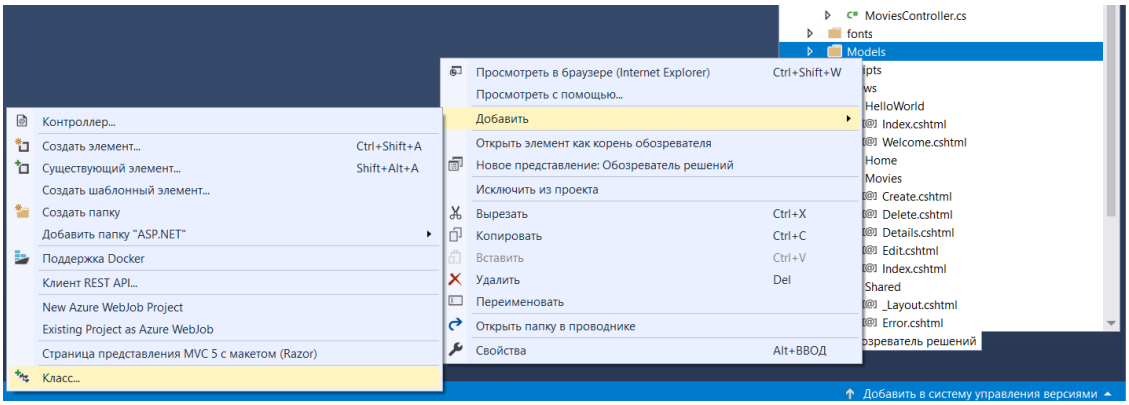
Методические указания

Лабораторная работа посвящена разработке основных контроллеров для обработки запросов по созданию, редактированию и удалению справочных данных, то есть данных, которые используются для отражения хозяйственных операций и действий предметной области.

В ASP.NET MVC используется технология .NET Framework доступа к данным, известная как Entity Framework, позволяющая создавать и работать с классами сущностями. Поддерживает Entity Framework (часто обозначается как EF), возможность использования парадигмы разработки Code First. Во-первых, код позволяет создавать объекты модели путем написания простых классов. Затем можно создавать базу данных в режиме реального времени из классов моделей, которые позволяют очень просто и быстро организовать рабочий процесс.

Добавление классов модели

В обозревателе решений, щелкните правой кнопкой мыши по папке моделей, выберите добавить, а затем выберите класс.



Введите имя класса "Movie".

Добавьте следующие свойства в класс Movie:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public int Year { get; set; }

    public decimal Price { get; set; }
}
```

Мы будем использовать класс Movie для представления фильмов в базе данных. Каждый экземпляр объекта Movie будет соответствовать записи в таблице базы данных, а каждое свойство класса Movie сопоставляется со столбцом в таблице.

Чтобы использовать System.Data.Entity и связанный класс, необходимо установить пакет NuGet Entity Framework.

Для доступа к контакту базы данных добавьте следующий класс MovieDBContext:

```
    public class MovieDBContext : DbContext
    {
        public MovieDBContext(DbContextOptions<MovieDBContext> options) :
base(options)
        {

        }
        public DbSet<Movie> Movies { get; set; }
    }
```

Класс MovieDBContext представляет контекст базы данных movie Entity Framework, который выполняет получение, хранение и обновление экземпляров (записей) Movie в базе данных. MovieDBContext является производным от базового класса DbContext, предоставляемого платформой Entity Framework.

Чтобы иметь возможность ссылаться на DbContext и DbSet, необходимо добавить следующие инструкция using в верхней части файла.

```
using Microsoft.EntityFrameworkCore;
```

Миграция баз данных. Аннотации данных

Модель данных в процессе разработки может измениться и перестанет соответствовать базе данных. Всегда можно удалить базу данных, и Entity Framework (EF) создаст для вас новую версию, в точности соответствующую модели, но такая процедура приводит к потере текущих данных. Функция миграции в EF позволяет последовательно применять изменения схемы к базе данных, чтобы синхронизировать ее с моделью данных в приложении без потери существующих данных.

Например, мы решили внести в нее новые свойства. Но при этом у нас уже имеется существующая база данных, в которой есть какие-то данные. И чтобы без потерь обновить базу данных ASP.NET MVC предлагает нам такой механизм как миграции.

И допустим, у нас есть вся инфраструктура для работы с этой моделью - представления, контроллеры, и у нас есть уже в базе данных несколько объектов данной модели. Но в какой-то момент мы решили изменить модельную базу приложения. Например, мы добавили еще одно поле в модель Movie и новую модель данных:

```
    public class Genre
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
```

И новое поле в модели Movie. Добавлено два поля класса для одного свойства модели. Это сделано по требованию Entity Framework. Согласно этой технологии внешний ключ создается следующим образом: создается int поле, которое именуется следующим образом: <ИмяМодели>Id, и создается «навигационное» поле типа модели внешней сущности. При этом в базе данных физически создастся поле <ИмяМодели>Id, а «навигационное» поле, будет заполнятся при выборке данных для удобства работы с данными.

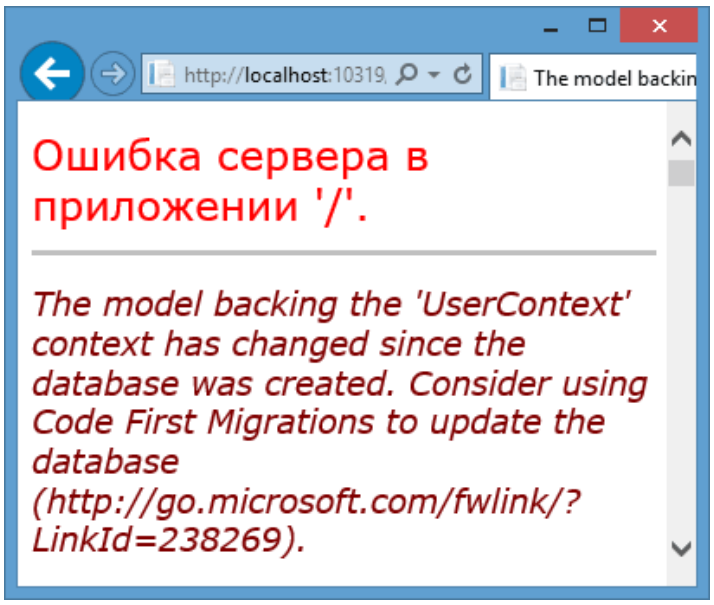
```
        public int GenreId { get; set; }
        public Genre? Genre { get; set; }
```

Таким образом, контекст данных у нас уже меняется следующим образом:

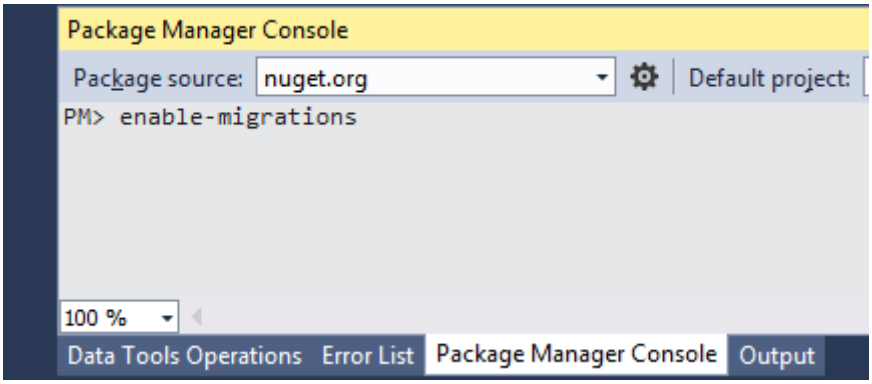
```
    public class MovieDBContext : DbContext
    {
        public MovieDBContext(DbContextOptions<MovieDBContext> options) :
base(options)
        {

        }
        public DbSet<Movie> Movies { get; set; }
        public DbSet<Genre> Genres { get; set; }
    }
```

После такого, после запуска системы будет выдано сообщение об ошибке:



Контекст данных изменился, и теперь нам надо провести миграцию от старой схемы базы данных к новой. И первым делом найдем внизу Visual Studio окно Package Manager Console, введем в нем команду: enable-migrations и нажмем Enter:



После выполнения этой команды Visual Studio в проекте будет создана папка Migrations, в которой можно найти файл Configuration.cs. Этот файл содержит объявление одноименного класса Configuration, который устанавливает настройки конфигурации. Теперь нам надо создать саму миграцию. Там же в консоли Package Manager Console введем команду:

```
PM> Add-Migration "MigrateDB"
```

После этого Visual Studio автоматически сгенерирует класс миграции. В котором есть два метода Up и Down. Метод Up содержит код позволяющий добавить в базу данных новые поля, изменить существующие поля, добавить новые таблицы и т.п.

Метод Down содержит методы для отката изменений, создаваемых миграцией.

И в завершении чтобы выполнить миграцию, применим этот класс, набрав в той же консоли команду:

```
PM> Update-Database
```

После этого, если мы посмотрим на состав базы данных, то увидим, что к ней были применены изменения в соответствии с выполненной миграцией.

Итак, миграция выполнена, и мы можем уже использовать обновленные модели и контекст данных.

Аннотации данных

Атрибут Key

Entity Framework считает, что у каждой сущности есть первичный ключ, и этот ключ используется для отслеживания сущностей. Атрибут Key указывает свойство/столбец, являющийся частью первичного ключа объекта, и применяется только к скалярным свойствам.

```
    public class DepartmentMaster
    {
        [Key]
        public int DepartmentId { get; set; }
}
```

Атрибут ForeignKey

Этот атрибут указывает внешний ключ для навигационного свойства.

```
public class Employee
{
    [Column("ID", Order = 1)]
    public int EmployeeId { get; set; }
    [Column("Name", Order = 2, TypeName = "Varchar(100)")]
    public string EmployeeName { get; set; }
    [ForeignKey("Department ")]
    public int DepartmentId { get; set; }
    [ForeignKey("DepartmentId")]
    public DepartmentMaster Department { get; set; }
}
```

Атрибут NotMapped

В подходе модели Code First каждое свойство модели представлено в виде столбца таблицы в базе данных. Это не всегда так, нам может потребоваться какое-то свойство в модели или объекте, которого нет в таблице базы данных. Например, объект «Отдел» имеет свойство с именем «DepartmentCodeName», это свойство возвращает комбинацию кода и имени, разделенных двоеточием (:). Это свойство может быть создано динамически, и нет необходимости хранить его в базе данных. Его можно пометить аннотацией NotMapped.

```
public class DepartmentMaster
{
        [NotMapped]
        public string DepartmentCodeName
        {
            get
            {
                return Code + ":" + Name;
            }
        }
…
….
}

[NotMapped]
public class InternalClass
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Атрибут Required

Атрибут Required сообщает Entity Framework, что это свойство должно иметь значение, и этот атрибут заставит Entity Framework убедиться, что в нем есть данные. Этот атрибут также будет участвовать в создании базы данных (помечая этот столбец как «not null»).

```
[Required]
public string Code { get; set; }
```

Атрибут MinLength

Этот атрибут используется для проверки свойства, имеет ли свойство минимальную длину строки.

```
[MinLength(5)]
public string Name { get; set; }
```

Атрибут MaxLength

Атрибут MaxLength позволяет указать дополнительные проверки свойств, чтобы установить максимальную длину строки. Этот атрибут также будет участвовать в создании базы данных (путем установки длины свойства).

```
[MinLength(5)]
[MaxLength(100)]
public string Name { get; set; }
```

Атрибут StringLength

StringLength используется для указания максимальной длины строки. Этот атрибут применяется только к свойствам строкового типа. Мы также можем указать минимальную длину символов, разрешенных в поле данных. Этот атрибут также будет участвовать в создании базы данных (путем установки длины свойства).

```
[StringLength(100, MinimumLength = 5)]
public string Name { get; set; }
```

Атрибут Display

Свойство Name атрибута Display содержит строку, которая будет отображаться вместо имени свойства.

```
public class Book
    {
        public int Id { get; set; }
        [Display(Name = "Название")]
        public string Name { get; set; }
        [Display(Name = "Автор")]
        public string Author { get; set; }
        [Display(Name = "Год")]
        public int Year { get; set; }
    }
```

Атрибут DataType

Атрибут DataType позволяет предоставлять среде выполнения информацию об использовании свойства. Например, допустим, у нас есть свойство Password:

```
[DataType(DataType.Password)]
public string Password { get; set; }
```

Перечисление DataType может принимать несколько различных значений:

Значение	Описание
Currency	Отображает текст в виде валюты
DateTime	Отображает дату и время
Date	Отображает только дату, без времени
Time	Отображает только время
Text	Отображает однострочный текст
MultilineText	Отображает многострочный текст (элемент textarea)
Password	Отображает символы с использованием маски
Url	Отображает строку URL
EmailAddress	Отображает электронный адрес

Атрибут Range

Позволяет указать диапазон значений. Свойство Message позволяет указать текст сообщения об ошибке.

```
public class Student
    {
        [Key]
        public int StdntID { get; set; }

        [DisplayName("Фамилия студента: ")]
        [StringLength(50, MinimumLength = 3)]
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        [Range(18, 60, ErrorMessage = "Возраст должен быть от 18 до 60 лет.")]
        public int Age { get; set; }

        [NotMapped]
        public int FatherName { get; set; }
    }
```

Общий пример

```
    public class StudentModel
    {
        [ScaffoldColumn(false)]
        public int Id { get; set; }
        [Required(ErrorMessage = "Введите имя студента")]
        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }
        [Required(ErrorMessage = "Введите значение Email")]
        [DataType(DataType.EmailAddress)]
        [MaxLength(50)]
        [RegularExpression(@"[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}", ErrorMessage =
"Введен неверный адрес")]
        public string Email { get; set; }
        [Required(ErrorMessage = "Введите значение Email")]
        [DataType(DataType.EmailAddress)]
        [System.ComponentModel.DataAnnotations.Compare("Email", ErrorMessage = "Email
не совпадает")]
        public string ConfirmEmail { get; set; }
        [Required(ErrorMessage = "Введите возраст")]
        [Range(18, 60, ErrorMessage = "Возраст должен быть от 18 до 60 лет.")]
        public int Age { get; set; }
    }
```

Создание строки подключения и работа с SQL Server LocalDB

Созданный класс MovieDBContext обрабатывает подключения к базе данных и сопоставляет объекты Movie для записи базы данных.

Нет необходимости указывать какую базу данных использовать, Entity Framework по умолчанию будет использовать LocalDB.

LocalDB — это облегченная версия SQL Server Express Database Engine, запускаемая по запросу и работает в пользовательском режиме. LocalDB выполняется в специальном режимм выполнения SQL Server Express, который позволяет работать с базами данных mdf файлов. Как правило, хранятся файлы базы данных LocalDB в папке App_Data проекта.

В Visual Studio LocalDB устанавливается по умолчанию с помощью Visual Studio.

Откройте в корневом каталоге приложения файл appsettings.json, показанный ниже. Добавьте следующую строку подключения для элемента ConnectionStrings в файл.

```
  "ConnectionStrings": {
    "MovieDB": "Server=(localdb)\\MSSQLLocalDB;Database=MovieDB;Integrated
Security=True;Trusted_Connection=True;MultipleActiveResultSets=True;"
  }
```

Должно получится следующим образом:

```
{
```

```
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MovieDB": "Server=(localdb)\\MSSQLLocalDB;Database=MovieDB;Integrated
Security=True;Trusted_Connection=True;MultipleActiveResultSets=True;"
  }
}
```

После этого необходимо «зарегистрировать» контекст данных в нашем приложении. Для этого в файле program.cs нужно добавить сервис для работы с контекстом данных:

```
builder.Services.AddDbContext<MovieDBContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("MovieDB")));
```

В итоге процесс построения системы должен выглядеть следующим образом:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<MovieDBContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("MovieDB")));

var app = builder.Build();
```

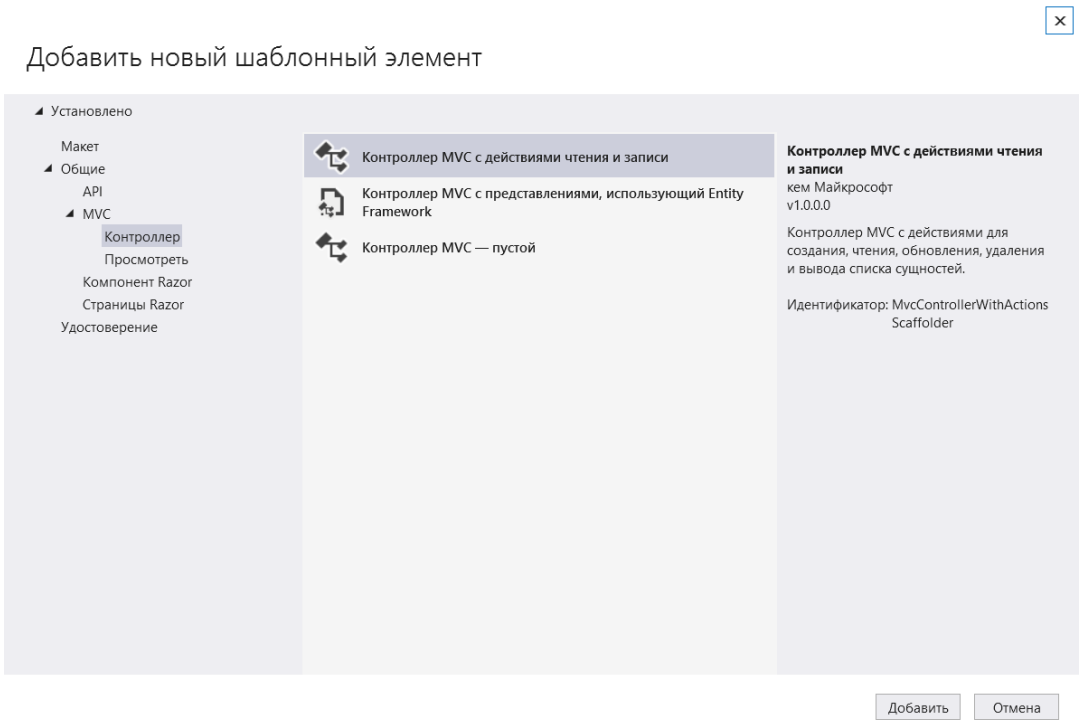
Доступ к данным модели из контроллера

Создадим новый класс MoviesController и напишем код, который извлекает данные фильма и отображает его в браузере с помощью шаблона представления.

Соберите приложение перед переходом к следующему шагу. Если вы не соберете приложение, вы получите ошибку при добавлении контроллера.

В обозревателе решений щелкните правой кнопкой мыши по папке контроллеров и нажмите кнопку Добавить, затем «Контроллер».

В диалоговом окне Добавление шаблона щелкните контроллер MVC 5 с представлениями, использующий Entity Framework, а затем нажмите кнопку Добавить.

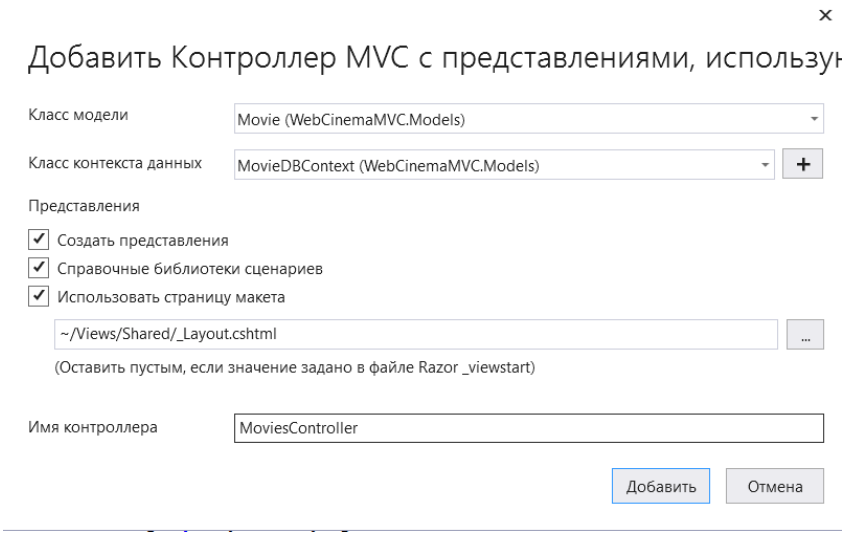


Выберите Movie (MvcMovie.Models) класс модели.

Выберите MovieDBContext (MvcMovie.Models) для класса контекста данных.

Имя контроллера введите MoviesController.

На следующем рисунке показано заполненное диалоговое окно.



Нажмите кнопку Добавить. Если отобразится сообщение об ошибке, возможно, не было построено приложение перед началом добавления контроллера. Visual Studio создает следующие файлы и папки:

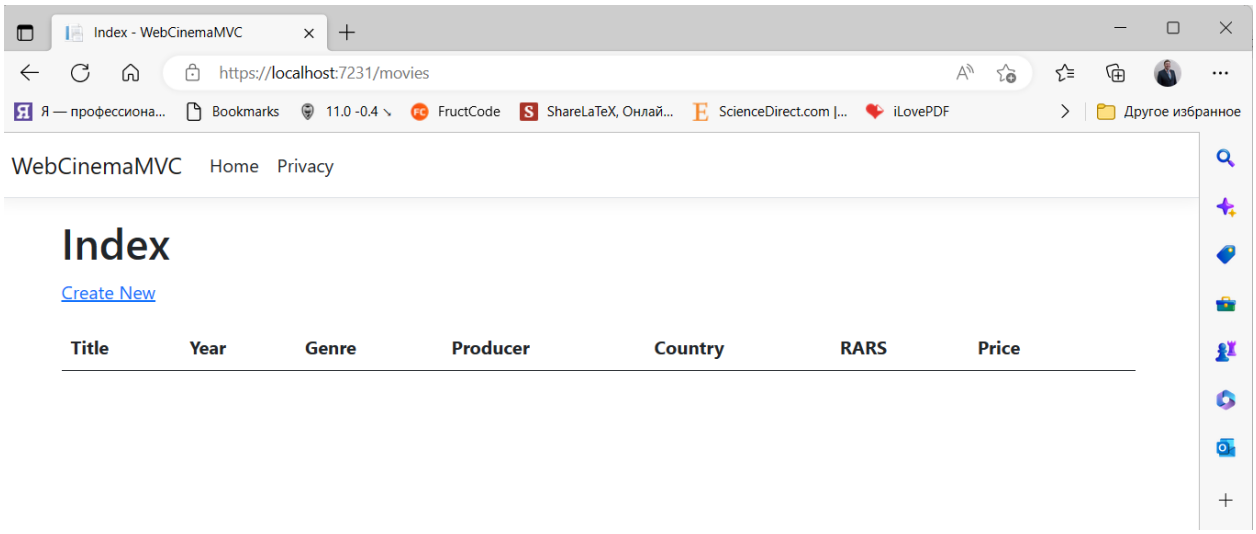
Файл MoviesController.cs в папке контроллеров.

Папку Views\Movies.

Файлы представлений Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml, и Index.cshtml в новой папке Views\Movies.

Visual Studio автоматически создается CRUD (Создание, чтение, обновление и удаление) методы действий и представления (автоматическое создание действий CRUD-методов и представления называется формированием шаблонов). Теперь у вас есть полнофункциональное веб-приложение, которое служит для создания, перечисления, редактирования и удаления записей фильмов.

Запустите приложение и перейдите к контроллеру Movies, добавив /Movies в URL-адресе в адресной строке браузера. Так как приложение полагается на маршрутизацию по умолчанию (определенный в файле App_Start\RouteConfig.cs), запрос браузера http://localhost:xxxxx/Movies направляется по умолчанию Index метод действия Movies контроллера. Другими словами, запрос браузера http://localhost:xxxxx/Movies так же, как запрос браузера http://localhost:xxxxx/Movies/Index. Результатом является пустой список фильмов, так как вы их еще не добавили.



Создание фильма

Щелкните ссылку Create New (Создать). Введите некоторые сведения о фильм, а затем нажмите кнопку Создать.

При нажатии на кнопку «Создать» выполняется отправка формы на сервер, где сведения о фильме сохраняются в базе данных. Затем вы перейдете к URL-адресу /Movies, где вы увидите информацию о только что созданном фильме.

Изучение созданного кода

Откройте файл Controllers\MoviesController.cs.

В начале класса создано поле класса для работы с контекстом данных.

```
private readonly MovieDBContext _context;

public MoviesController(MovieDBContext context)
{
    _context = context;
}
```

Ссылка на экземпляр контекста приходит в конструктор через механизм внедрения зависимостей.

Часть контроллера Movie с методом Index() приведен ниже.

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movies.ToListAsync());
}
```

Запрос на контроллер Movies получает все записи в таблице Movies и затем передает результаты в представление Index. В первой строке класса MoviesController создается экземпляр контекста базы данных фильмов. Контекст базы данных Movie позволяет запрашивать, изменять и удалять элементы.

Строго типизированные модели и @model ключевое слово

Контроллер может передать данные или объекты в шаблон представления с помощью объекта ViewBag. ViewBag является динамическим объектом, который предоставляет удобный способ для передачи информации в представление с поздним связыванием.

MVC также предоставляет возможность создавать строго типизированные представления. Строго типизированные представления позволяют использовать возможности IntelliSense в редакторе Visual Studio. Этот подход используется в механизме шаблонного формирования объектов (контроллеров или представлений) в Visual Studio с класса MoviesController и представление шаблонов при создании методов и представлений.

В файле Controllers\MoviesController.cs изучите созданный метод Details. Метод Details приведен ниже.

```
        // GET: Movies/Details/5
        public async Task<IActionResult> Details(int? id)
        {
            if (id == null || _context.Movies == null)
            {
                return NotFound();
            }

            var movie = await _context.Movies
                .FirstOrDefaultAsync(m => m.ID == id);
            if (movie == null)
            {
                return NotFound();
            }

            return View(movie);
        }
```

Параметр id обычно передается в качестве данных маршрута, например http://localhost:1234/movies/details/1 задаст запрос к контроллеру фильмов, действие – details и id – 1. Можно также передавать в идентификаторе со строкой запроса следующим образом:

http://localhost:1234/movies/details?id=1

Изучим содержимое файла Views\Movies\Details.cshtml:

```
@model WebCinemaMVC.Models.Movie

@{
    ViewData["Title"] = "Details";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Year)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Year)
```

```
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Genre.Id)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Producer)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Producer.Id)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Country)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Country.Id)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.RARS)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.RARS.Id)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model?.ID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Включив инструкцию @model в верхней части файла шаблона представления, можно указать тип объекта, который будет ожидаться представлением. При создании контроллера movie Visual Studio автоматически включает следующий оператор @model в начало файла Details.cshtml:

```
@model MvcMovie.Models.Movie
```

Эта директива @model обеспечивает доступ к фильму, который контроллер передал в представление с использованием строго типизированного объекта Model.

Изучив шаблон представления Index.cshtml и Index метод в MoviesController.cs файла, стоит обратить внимание на то, как в коде создается

объект List при вызове вспомогательного метода View. Затем код передает этот список Movies в представление:

```
// GET: Movies
public ActionResult Index()
{
    return View(db.Movies.ToList());
}
```

При создании контроллера movie Visual Studio автоматически включает следующую инструкцию @model в верхней части файла Index.cshtml:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

Работа с SQL Server LocalDB

Code First Entity Framework видит, что указывается строка подключения базы данных, в который была предоставлена база данных Movies, которая не существует, поэтому Code First создает базу данных автоматически. Убедитесь, что он создан в папке App_Data. Если вы не видите Movies.mdf щелкните кнопку «Показать все файлы» в обозревателе решений панели инструментов и нажмите кнопку обновить.

Дважды щелкните по Movies.mdf в Обозревателе серверов, затем разверните таблицы для просмотра. Обратите внимание на значок ключа рядом с идентификатором. По умолчанию EF сделает свойство с именем идентификатор первичного ключа.

Задание на лабораторную работу

Продолжить разработку Web-приложения по вашей тематике:

1. Разработать модели данных
2. Разработать представления для выполнения CRUD действий.
3. Снабдить все модели необходимыми аннотациями для корректного отображения и валидации.
4. Сделать все представления в дружественном стиле.