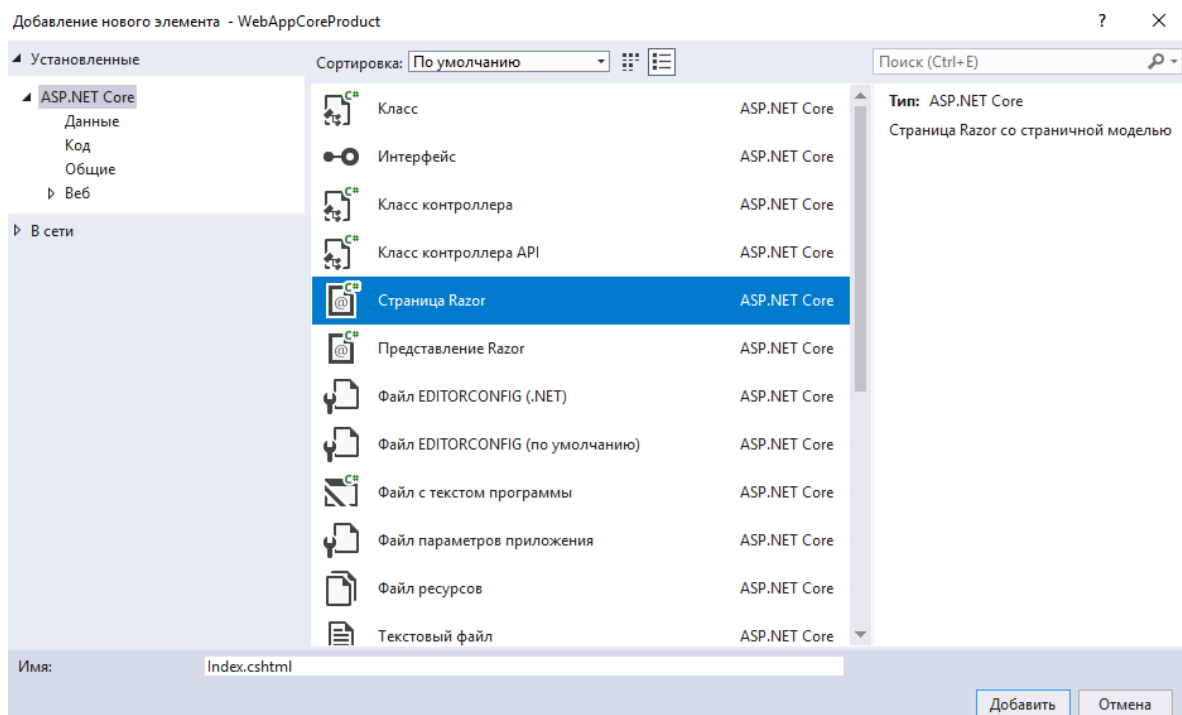


Lab 1. Введение в создание приложений ASP.NET Core на основе Razor Pages

Упражнение 1. Создание проекта на основе RazorPages

В этом упражнении вы создадите пустой проект ASP NET Core, добавите в него базовую функциональность и исследуете логику работе GET-запроса.

- Создайте новый проект по шаблону **Empty** (Пустой) по имени **WebAppCoreProduct**.
- В новый проект добавьте папку **Pages** – там в дальнейшем будем хранить страницы Razor Pages).
- В папку Pages добавьте новую страницу Razor – для этого в контекстном меню папки выберите пункт **Add** (Добавить)→ **New Item** (Создать элемент).
- Выберите шаблон **Razor Page** (Страница Razor) и оставьте имя файла по умолчанию – **Index.cshtml**:



- Проверьте, что после создания добавленной страницы в папке **Pages** будут добавлены два файла – сама страница **Index.cshtml** и связанный с ней файл кода **Index.cshtml.cs**.
- Откройте файл **Index.cshtml** и проверьте указание директивы `@model` (может быть не указано пространство имен):

```
@page
@model WebAppCoreProduct.Pages.IndexModel
@{
}
```

Директива `@page` указывает, что это страница Razor.

Директива `@model` - в данном случае это класс привязанного к странице кода `IndexModel`. Согласно условностям, класс модели называется по имени страницы плюс суффикс "Model".

- Откройте файл `Index.cshtml.cs` и изучите определение модели `IndexModel`, обратите внимание на наличие метода `onGet()`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace EmptyRazorPagesApp.Pages
{
    public class IndexModel : PageModel
    {
        public void OnGet()
        {
        }
    }
}
```

- Постройте и запустите приложение. Должна отобразиться страница с приветственной надписью "Hello World!" (причем наличие добавленной страницы не оказало влияния на результат запуска):
 - Откройте файл `Srartup.cs` и изучите класс `Startup` – он предназначен для настроек служб и конвейера запросов приложения.
 - Найдите в методе `Configure()` реализацию ответа по умолчанию: `context.Response.WriteAsync("Hello World!");`
- Для загрузки именно razor-страниц необходимо добавить соответствующие сервисы – перейдите по ссылке, указанной в комментариях к методу, и изучите документацию по запуску приложения в ASP.NET Core.
 - Обратите внимание на версию ASP.NET Core. Содержание методов класса `Startup` существенно различается, например для версий 2.1 и 3.0 и выше.
- Для версии ASP.NET Core 2.1:
 - В метод `ConfigureServices()` добавьте:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

- В методе `configure()` удалите `app.Run()` с вызовом "Hello World!" и добавьте несколько компонентов ПО промежуточного слоя в конвейер запросов:

```
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseCookiePolicy();
app.UseMvc();
```

- Для версии ASP.NET Core 3.0 и выше класс Startup может быть таким:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages(); // сервисы Razor Pages
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseDeveloperExceptionPage();

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages(); // маршрутизация для RazorPages
        });
    }
}
```

Теперь добавим в проект бизнес-логику. Сначала она будет добавлена прямо в страницу Index, затем вы выделите ее в отдельную модель. Предположим, что страница Index Razor Page принимает через запрос некоторые данные и выводит их на страницу.

Требуется, чтобы страница принимала название продукта и его цену, а также была реализована проверка вводимой цены.

- Добавьте требуемые свойства и реализуете передачу данных (с проверкой) в методе OnGet():

```
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace WebAppCoreProduct.Pages
{
    public class IndexModel : PageModel
    {
        public string Name { get; set; }
        public decimal? Price { get; set; }
        public bool IsCorrect { get; set; } = true;

        public void OnGet(string name, decimal? price)
        {
            if (price == null || price < 0 ||
string.IsNullOrEmpty(name))
            {
                IsCorrect = false;
                return;
            }
            Price = price;
            Name = name;
        }
    }
}
```

Данный класс определяет три свойства: Name представляет полученное название продукта, Price – его цену, а IsCorrect указывает, переданы ли корректные данные.

В методе OnGet() через параметры name и price реализуется получение переданных через строку запроса данных.

- Добавьте код страницы Index.cshtml:

```
@page
@model WebAppCoreProduct.Pages.IndexModel
@{
    @if (Model.IsCorrect)
    {
        <p>Name: <b>@Model.Name</b></p>
        <p>Price: <b>@Model.Price</b></p>
    }
    else
    {
        <p>Переданы некорректные данные</p>
    }
}
```

С помощью директивы @model устанавливается модель страницы. Поэтому через свойство Model можно обращаться к модели страницы, в том числе к ее свойствам.

- Запустите приложение. Если странице не переданы никакие значения (или переданы некорректные значения), то страница выведет соответствующее сообщение: *Переданы некорректные данные*.
- Передадите значения через параметры строки запроса (параметр из строки запроса должен совпадать по названию с параметром метода OnGet):

`https://localhost:ваш_порт/Index?name=Tomato&price=31`

Приложение получит данные и выведет на страницу результат согласно указанной разметки страницы Index.cshtml.

Добавление в проект файла _ViewImports.cshtml

Представления и страницы могут использовать директивы-Razor для импорта пространств имен и использования внедрения зависимостей. Директивы, используемые несколькими представлениями, можно указать в общем файле _ViewImports.cshtml, который по умолчанию содержится в папке **Pages** проекта ASP.NET Core по типу Web Application. Файл _ViewImports.cshtml можно поместить в любую папку, но в этом случае он будет применяться только к страницам или представлениям в этой папке и вложенных в нее папках.

- Добавьте в папку **Pages** новый файл типа **Razor View** (Представление Razor) и укажите ему имя _ViewImports.cshtml.
- Добавьте в новый файл следующий код:

```
@using WebAppCoreProduct
@namespace WebAppCoreProduct.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

В этом файле используются только директивы синтаксиса Razor. Первые две строки добавляют функциональность пространства имен текущего проекта, третья добавляет функциональность встроенных tag-хелперов из пространства имен Microsoft.AspNetCore.Mvc.TagHelpers. Эти возможности потребуются в дальнейшем.

- Проверьте добавленную функциональность общего импорта – в разметке страницы Index.cshtml удалите теперь уже лишнее указание пространства имен (WebAppCoreProduct.Pages):

```
@page
@model IndexModel
```

- Постройте и запустите приложение. Функциональность не должна измениться.

Упражнение 2. Применение модели для представления данных

В этом упражнении вы выделите свойства, характеризующие сущность товара в отдельный класс – модель, а в классе IndexModel оставите только свойства, характерные для логики отображения.

- Добавьте в проект новую папку **Models**.
- В новую папку добавьте новый класс – Product.
- Перенесите свойства Name и Price из класса IndexModel в класс Product. Проверку вводимых данных оставим модели IndexModel.
- В классе IndexModel оставьте свойство для реализации проверки, добавьте ссылку на свойство класса продукта и измените метод OnGet() для заполнения полей класса продукта:

```
public bool IsCorrect { get; set; } = true;
public Product Product { get; set; }

public void OnGet(string name, decimal? price)
{
    Product = new Product();
    if (price == null || price < 0 || string.IsNullOrEmpty(name))
    {
        IsCorrect = false;
        return;
    }
    Product.Price = price;
    Product.Name = name;
}
```

- В коде страницы Index.cshtml измените доступ к полям:

```
<p>Name: <b>@Model.Product.Name</b></p>
<p>Price: <b>@Model.Product.Price</b></p>
```

- Запустите приложение. Функциональность не должна измениться.

Упражнение 3. Создание формы для заполнения данных

В этом упражнении вы добавите форму для отправки данных. Удобным способом получения данных является использование параметров метода OnGet() или OnPost().

- Добавьте в код разметки Index.cshtml форму для отправки данных (форма по умолчанию будет использовать метод OnGet()):

```
@model IndexModel
<form>
```

```

<label for="name">Название товара</label><br />
<input type="text" name="name" /><br />
<label for="price">Цена</label><br />
<input type="number" name="price" /><br /><br />
<input type="submit" value="Получить скидку" /><br />
</form>

```

- В конце страницы добавьте код для отображения сообщения о результате:

```
<h3>@Model.MessageResult</h3>
```

- В классе IndexModel в метод OnGet () добавьте код для расчета скидки и получения результирующего сообщения:

```

var result = price * (decimal?)0.18;
MessageResult = $"Для товара {name} с ценой {price} скидка получится {result}";

```

- Запустите приложение. Протестируйте работу формы. Обратите внимание на выполнение GET-запроса.
- Добавьте в класс IndexModel новый метод OnPost () с такими же параметрами и логикой, как у метода onGet (), только реакция на неправильный ввод другой – не требуется свойство IsCorrect:

```

public void OnPost(string name, decimal? price)
{
    Product = new Product();
    if (price == null || price < 0 || string.IsNullOrEmpty(name))
    {
        MessageResult = "Переданы некорректные данные. Повторите
ввод";
        return;
    }

    var result = price * (decimal?)0.18;
    MessageResult = $"Для товара {name} с ценой {price} скидка
получится {result}";
    Product.Price = price;
    Product.Name = name;
}

```

- Предыдущий метод OnGet () удалите, а вместо него добавьте новый без параметров и с простой логикой:

```

public void OnGet ()
{
    MessageResult = "Для товара можно определить скидку";
}

```

- Для отправки запроса методом onPost () в разметке Index.cshtml добавьте в тег form соответствующий атрибут method="post" и так как теперь при загрузке формы нет необходимости проверять данные, то удалите конструкцию if-else:

```

@page
@model IndexModel
<form method="post">
    <label for="name">Название товара</label><br />
    <input type="text" name="name" /><br />
    <label for="price">Цена</label><br />
    <input type="number" name="price" /><br /><br />
    <input type="submit" value="Получить скидку" /><br />

```

</form>

<h3>@Model.MessageResult</h3>

- Постройте и запустите приложение. При получении get-запроса приложение будет отдавать страницу с формой ввода. При получении post-запроса класс IndexModel получит отправленные данные и подсчитает результат, который потом выводится на страницу.

Упражнение 4. Применение макетных страниц

В этом упражнении вы разделите страницы интерфейса и примените мастер-страницу. Как правило, веб-приложения содержат основную (стартовую страницу) и несколько страниц, связанных с различными сущностями. В текущем проекте роль основной страницы играет Index.cshtml и поэтому для определения сущности *Продукт* требуется отдельная страница.

- Добавьте в папку **Pages** новую Razor-страницу Product.cshtml (появится также связанный с ней файл кода Product.cshtml.cs).
- Из файла Index.cshtml.cs перенесите всю логику – свойства (кроме свойства IsCorrect, его можно удалить, оно больше не нужно), методы onGet() и onPost() в новый файл Product.cshtml.cs. Подключите пространство имен WebAppCoreProduct.Models. В файле оставьте Index.cshtml.cs только пустой метод onGet().
- Из файла разметки Index.cshtml перенесите весь код разметки (кроме директив @page и @model IndexModel) в файл разметки Product.cshtml.

Теперь в приложении есть основная (стартовая) страница (пока пустая) и страница для работы с сущностью *Продукт*.

- Постройте и запустите приложение. После загрузки пустой страницы в адресную строку браузера введите Product – должна отобразиться форма на новой странице. Протестируйте работу формы – функциональность не должна измениться.

По умолчанию общая мастер-страница помещается в проекте в папку **Shared** (но в принципе это может быть любая папка).

- Следуя общепринятому расположению мастер-страниц, добавьте в проект в папку **Pages** новую папку **Shared**.
- В папку **Shared** добавьте новое Razor-представление _Layout.cshtml.
- В новый файл разметки добавьте следующий код:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Razor Pages</title>
</head>
<body>
  <div>
    <h1>Система учета</h1>
```

```

        <div>
            <ul>
                <li><a asp-page="/Index">Главная</a></li>
                <li><a asp-page="/Product">Продукт</a></li>
            </ul>
        </div>
        @RenderBody()
    </div>
    <div>© 2021 - Осипов Н.А.</div>
</body>
</html>

```

Мастер-страница содержит базовый каркас html-страницы, а с помощью вызова `@RenderBody()` будет вставляться содержимое других страниц. Обратите внимание на список навигационных ссылок.

- Добавьте в код разметки стартовой страницы `Index.cshtml` указание на ее связь с мастер-страницей и какой-нибудь еще произвольный код:

```

@{
    Layout = "_Layout";
}

<h3>Стартовая страница</h3>

```

- Запустите приложение. Проверьте, что стартовая страница отобразилась в соответствии с содержимым мастер-страницы. Воспользуйтесь ссылкой и перейдите на страницу продукта.
- Для придания общего вида странице `Product`, добавьте в файл разметки `Product.cshtml` связь с мастер-страницей:

```

@{
    Layout = "_Layout";
}

```

- Запустите приложение. Проверьте, что теперь и стартовая страница и страница продукта имеет вид в соответствии с мастер страницей.

Установка мастер-страницы для каждой Razor-страницы по отдельности не представляется удобным вариантом – можно установить мастер-страницу глобально для всех Razor-Pages (глобальная установка).

- Для реализации глобальной установки добавьте в папку **Pages** новое представление `_ViewStart.cshtml`.
- Укажите в новой странице связь с мастер-страницей:

```

@{
    Layout = "_Layout";
}

```

- Из страниц `Index.cshtml` и `Product.cshtml` удалите код привязки с мастер-страницей.
- Запустите приложение. Проверьте, что также и стартовая страница и страница продукта имеет вид в соответствии с мастер страницей.

Упражнение 5. Применение специальных обработчиков страницы Razor

В этом упражнении вы реализуете специальный (пользовательский) обработчик запроса для определенного вида (сценария). Возможна реализация как GET, как и POST специальных запросов.

- Добавьте в класс `ProductModel` новый метод – обработчик запроса POST определенного сценария (например, учитывается особая личная скидка), он должен в имени иметь приставку `onPost`:

```
public void OnPostDiscont(string name, decimal? price, double discount)
{
    Product = new Product();
    var result = price * (decimal?)discount/100;
    MessageRezult = $"Для товара {name} с ценой {price} и скидкой {discount}
получится {result}";

    Product.Price = price;
    Product.Name = name;
}
```

- В коде разметки `Product.cshtml` добавьте в код формы новые требуемые по сценарию поля и задайте явным образом нужные обработчики с помощью параметра **asp-page-handler** (в том числе и для обработчика `onPost`):

```
<form method="post">
    <label for="name">Название товара</label><br />
    <input type="text" name="name" /><br />
    <label for="price">Цена</label><br />
    <input type="number" name="price" /><br />
    <input type="submit" asp-page-handler="" value="Получить скидку" /><br />
    <label for="discount">Введите свою скидку(%)</label><br />
    <input type="number" name="discount" /><br />
    <input type="submit" asp-page-handler="Discont" value="С учетом своей скидки"/><br />
</form>
```

- Постройте и запустите приложение. Теперь на странице продукта определена форма, но в зависимости от того, на какую кнопку нажмет пользователь, будет передаваться запрос тому или иному обработчику.
- Обратите внимание на значение адреса в строке браузера при выборе второго обработчика – `https://localhost:порт/Product?handler=Discont`.

Параметры, которые идут после вопросительного знака в url представляют параметры строки запроса (это изучалось ранее), а данные, которые разделены в url слешами, представляют параметры маршрута, таким образом один url может содержать как параметры маршрута, так и параметры запроса.

- Чтобы определить параметры маршрута, необходимо после на странице после директивы `@page` определить шаблон маршрута – укажите для страницы `Product.cshtml`:

```
@page "{handler?}"
```

Вопросительный знак после названия параметра `"{handler?}"` указывает, что данный параметр является необязательным.

- Запустите приложение. Проверьте, что теперь при выборе специального обработчика в строке адреса отображается маршрут: `https://localhost:порт/Product/Discont`.
- Добавьте еще один обработчик запроса POST на ваше усмотрение. Реализуйте его функционирование.

Руководство составлено по материалам официальной документации

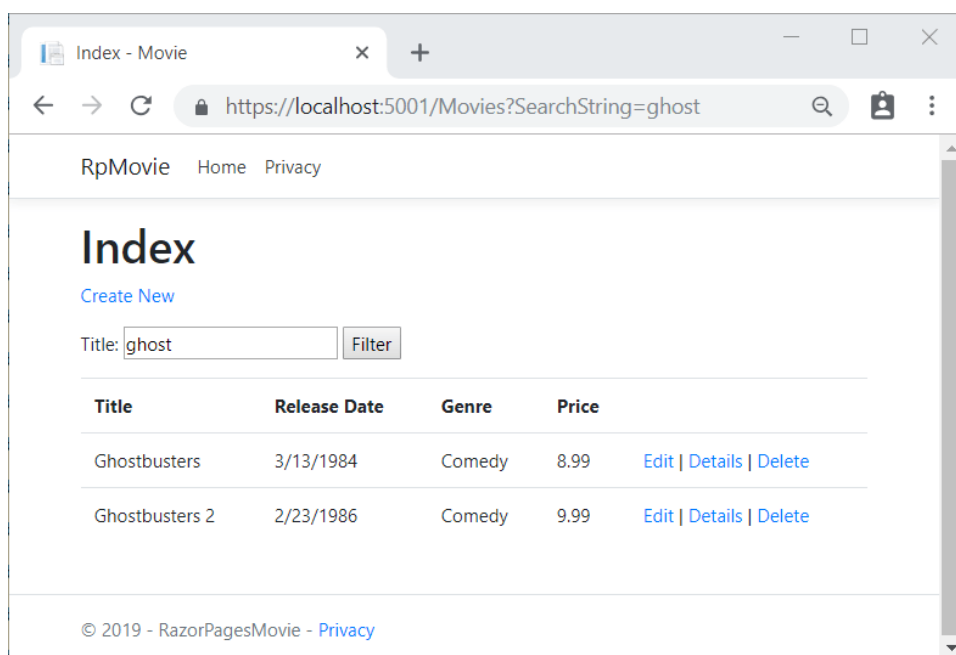
<https://docs.microsoft.com/ru-ru/aspnet/core/tutorials/razor-pages/?view=aspnetcore-3.1>

Lab 2. Создание веб-приложения Razor Pages с помощью ASP.NET Core

В этом руководстве описывается процесс создания веб-приложения Razor Pages:

1. Создание веб-приложения Razor Pages
2. Добавление модели в приложение Razor Pages
3. Формирование шаблона (создание) страницы Razor Pages
4. Работа с базой данных
5. Обновление Razor Pages
6. Добавление поиска
7. Добавление нового поля
8. Добавление проверки

В итоге вы получите приложение, которое позволяет работать с данными.



Код итогового приложения можно [Просмотреть или скачать пример кода](#).

1. Создание проекта с Razor Pages в ASP.NET Core

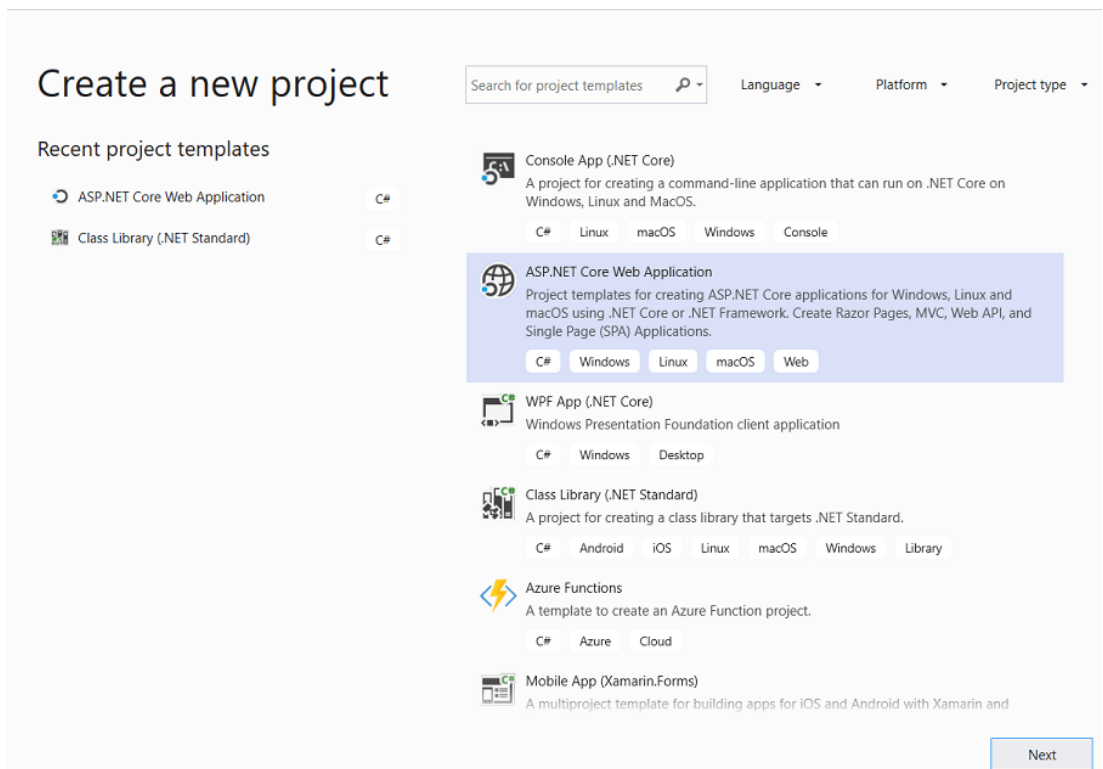
В этом разделе рассмотрены следующие задачи.

- Создание веб-приложения Razor Pages.
- Запуск приложения.
- Анализ файлов проекта.

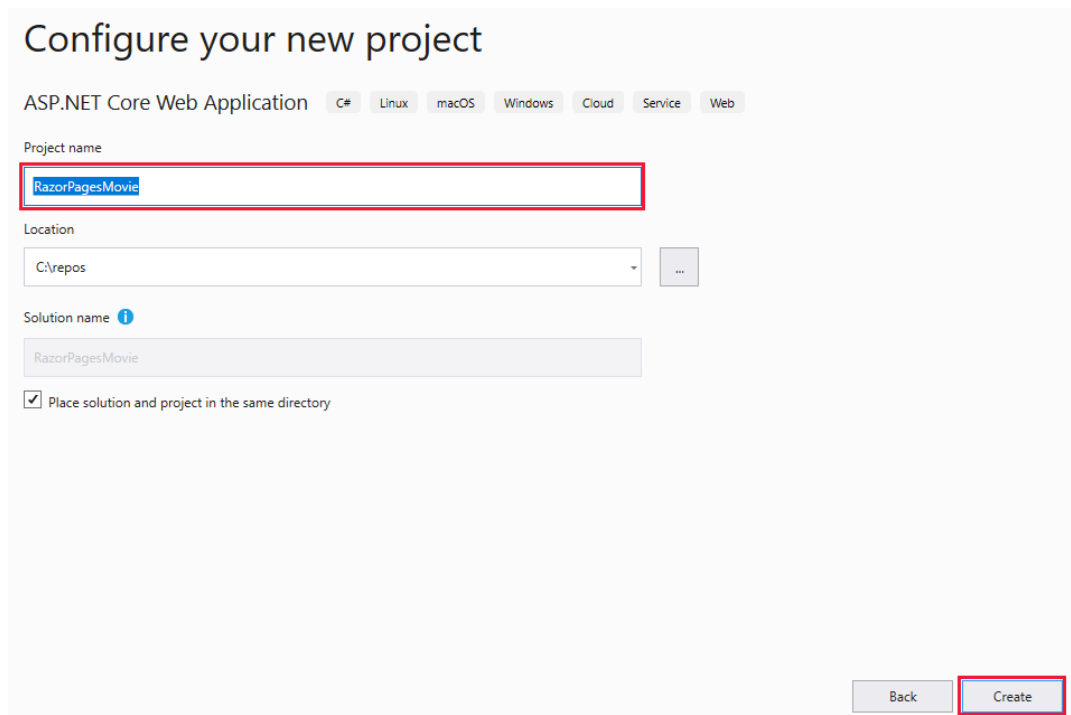
Выполнив действия этого раздела, вы получите рабочее веб-приложение Razor Pages, работа с которым описана в последующих разделах руководства.

Создание веб-приложения *Razor Pages*

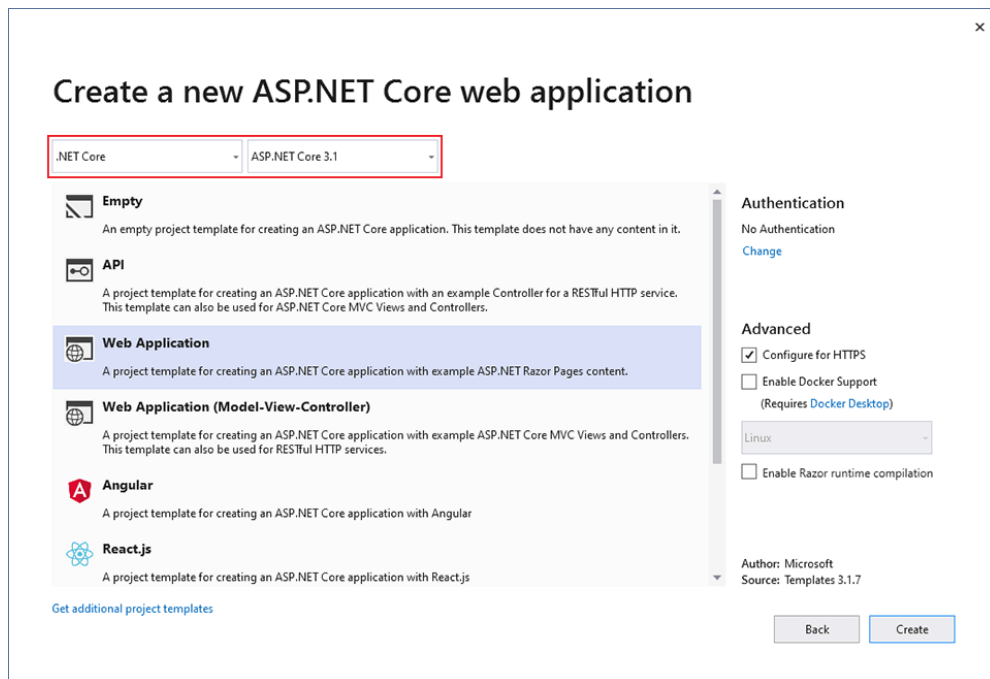
- В Visual Studio в меню **Файл** щелкните **Создать** > **Проект**.
- Создайте веб-приложение ASP.NET Core и нажмите кнопку **Далее**.



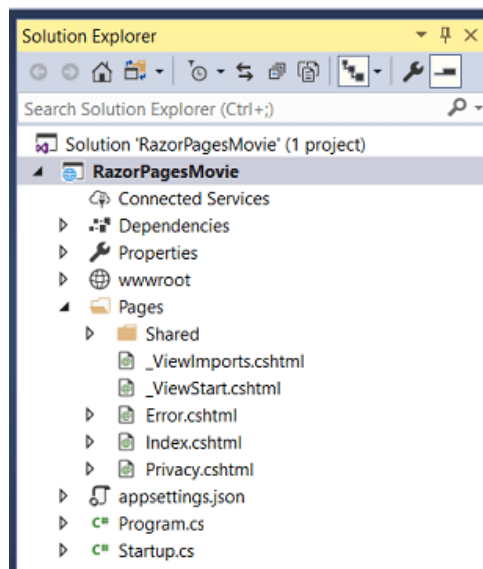
- Присвойте проекту имя **RazorPagesMovie**. Очень важно, чтобы проект имел имя *RazorPagesMovie*, так как пространства имен при копировании и вставке кода должны совпасть.



- Выберите в раскрывающемся списке пункт **ASP.NET Core 3.1**, затем — **Веб-приложение** и нажмите кнопку **Создать**.



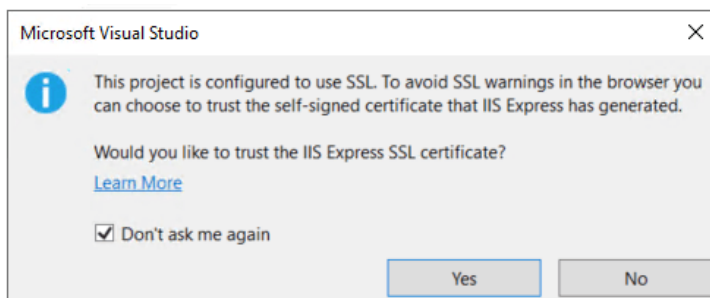
Создается следующий начальный проект:



Запуск приложения

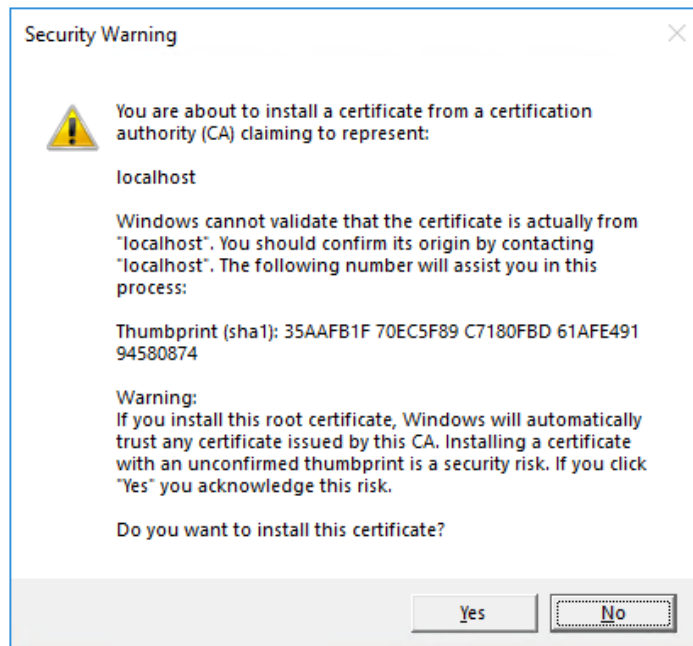
- Нажмите клавиши CTRL+F5, чтобы выполнить запуск без отладчика.

Visual Studio отображает следующее диалоговое окно.



- Выберите **Да**, чтобы сделать SSL-сертификат IIS Express доверенным.

Отобразится следующее диалоговое окно.



- Выберите **Да**, если согласны доверять сертификату разработки.

Visual Studio запускает [IIS Express](#), а затем приложение. В адресной строке указывается `localhost:port#`, это связано с тем, что `localhost` — стандартное имя узла для локального компьютера. Когда Visual Studio создает веб-проект, для веб-сервера используется случайный порт.

Анализ файлов проекта

- Ознакомьтесь с основными папками и файлами проекта, с которыми вы будете работать в последующих разделах.

Папка Pages

Содержит страницы Razor и вспомогательные файлы.

Каждая страница Razor — это пара файлов:

- Файл `.cshtml` с разметкой HTML и кодом C# использует синтаксис Razor.
- Файл `.cshtml.cs` с кодом C#, который обрабатывает события страницы.

Имена вспомогательных файлов начинаются с символа подчеркивания. Например, файл `_Layout.cshtml` настраивает элементы пользовательского интерфейса, общие для всех страниц. Этот файл настраивает меню навигации в верхней части страницы и уведомление об авторских правах в нижней части страницы. Для получения дополнительной информации см. [Макет в ASP.NET Core](#).

Папка `wwwroot`

Содержит статические файлы, такие как HTML-файлы, файлы JavaScript и CSS-файлы. Для получения дополнительной информации см. [Статические файлы в ASP.NET Core](#).

`appSettings.json`

Содержит данные конфигурации, например строки подключения. Для получения дополнительной информации см. [Конфигурация в .NET Core](#).

`Program.cs`

Содержит точку входа для программы. Для получения дополнительной информации см. [Универсальный узел .NET в ASP.NET Core](#).

`Startup.cs`

Содержит код, задающий поведение приложения. Для получения дополнительной информации см. [Запуск приложения в ASP.NET Core](#).

2. Добавление модели в приложение Razor Pages в ASP.NET Core

В этом разделе добавляются классы для управления фильмами.

Классы модели приложения используются в [Entity Framework Core](#) (EF Core) для работы с базой данных. EF Core – это объектно-реляционный сопоставитель (ORM), упрощающий получение доступа к данным. Классы моделей называются классами РОСО (plain old CLR objects — "старые добрые объекты CLR"), так как они не зависят от EF Core. Эти классы определяют свойства данных, которые хранятся в базе данных.

Добавление модели данных

- Щелкните проект **RazorPagesMovie** правой кнопкой мыши и выберите пункт **Добавить > Новая папка**. Присвойте папке имя *Models*.
- Щелкните правой кнопкой мыши папку *Models*. Выберите **Добавить > Класс**. Присвойте классу имя **Movie**.
- Добавьте в класс `Movie` следующие свойства:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

```
}  
}
```

Класс `Movie` содержит:

- Поле `ID` — является обязательным для первичного ключа базы данных.
- `[DataType(DataType.Date)]`: Атрибут [DataType](#) указывает тип данных (`Date`). С этим атрибутом:
 - пользователю не требуется вводить сведения о времени в поле даты.
 - Отображается только дата, а не время.

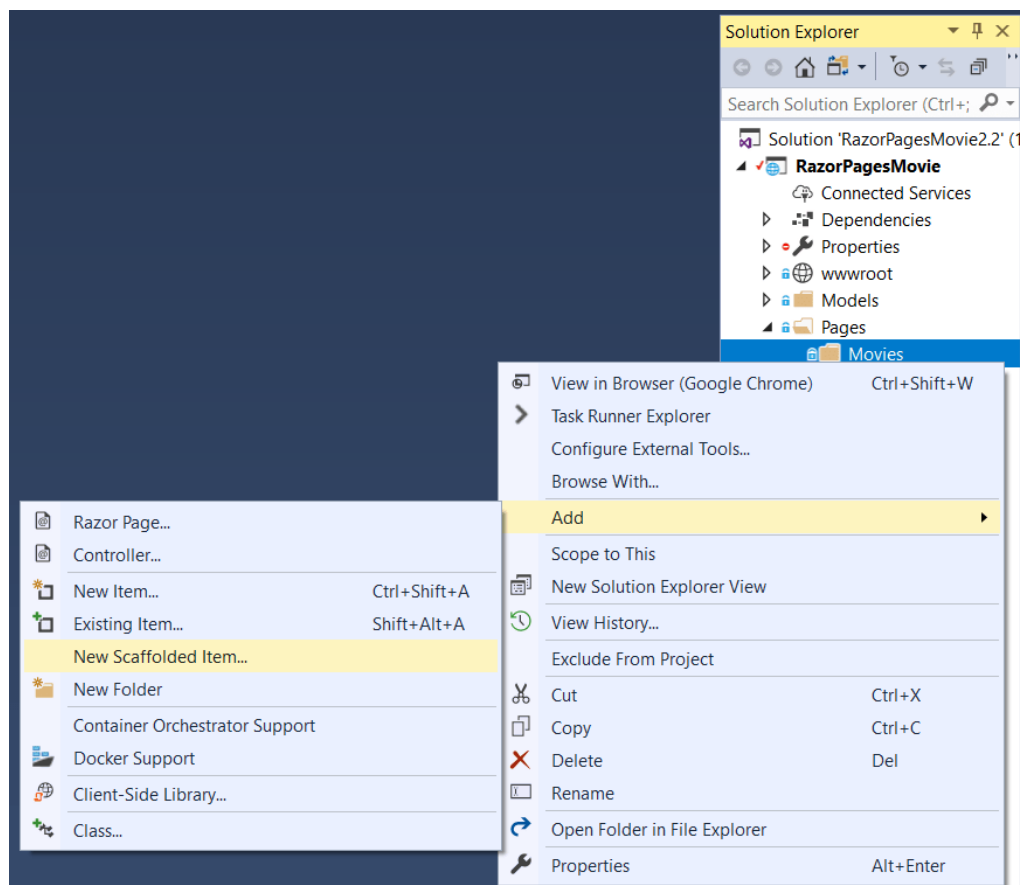
[DataAnnotations](#) рассматриваются в следующих разделах руководства.

- Выполните сборку проекта, чтобы убедиться в отсутствии ошибок компиляции.

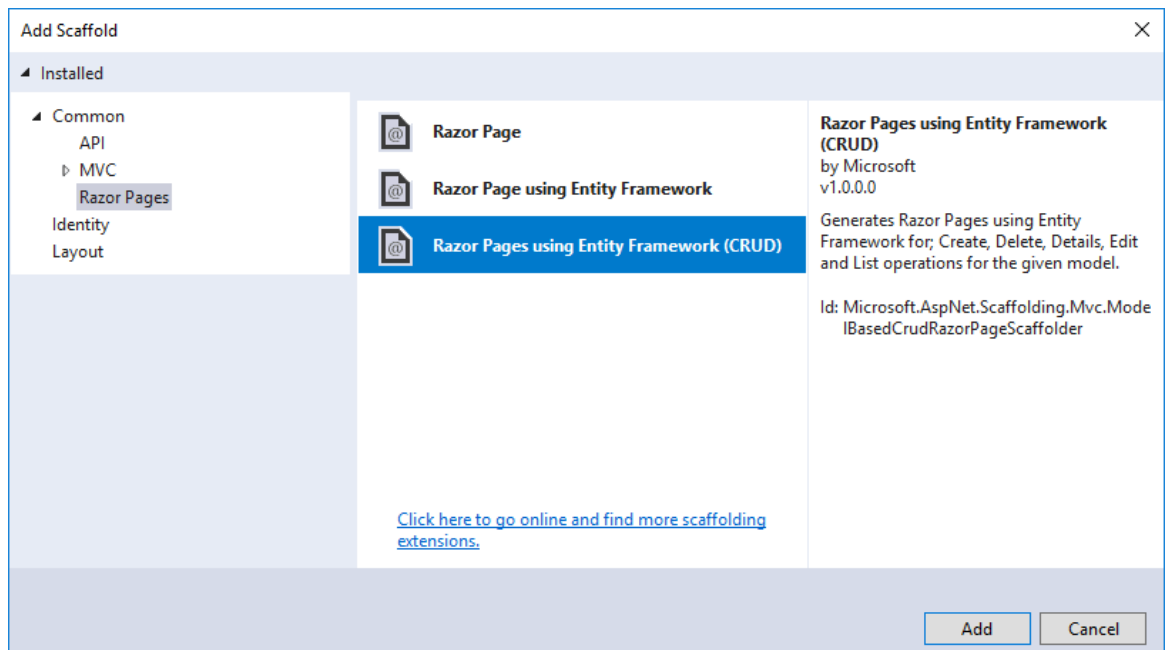
Создание шаблонов модели фильма

В этом разделе создаются шаблоны работы с данными на основе модели фильма. С помощью средства формирования шаблонов будут созданы страницы для операций создания, чтения, обновления и удаления для модели фильма.

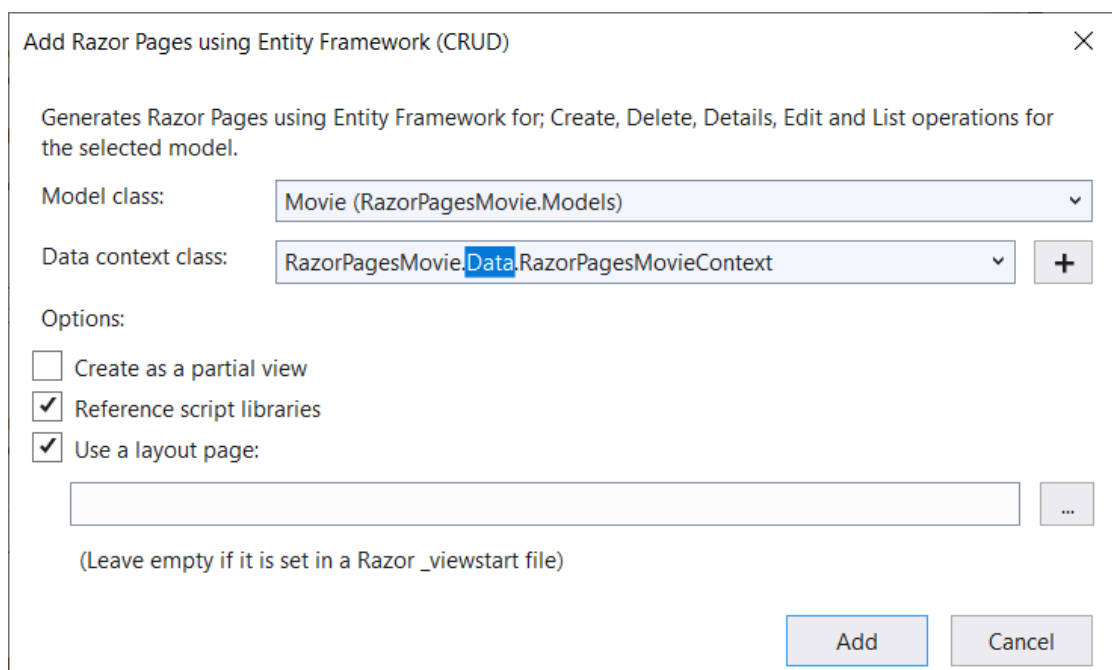
- В папке *Pages* создайте новую папку *Movies*.
- В контекстном меню папки *Movies* выберите **Add(Добавить) > New Scaffolded Item** (Создать шаблонный элемент):



- В диалоговом окне **Добавление шаблона** выберите **Razor Pages на основе Entity Framework (CRUD) → Добавить**.



- Заполните поля в диалоговом окне **Добавление Razor Pages на основе Entity Framework (CRUD)**:
 - В раскрывающемся списке **Класс модели** выберите **Фильм (RazorPagesMovie.Models)**.
 - В строке **Класс контекста данных** щелкните значок плюса + и измените сгенерированное имя **RazorPagesMovie.Models.RazorPagesMovieContext** на **RazorPagesMovie.Data.RazorPagesMovieContext** – будет создан класс контекста базы данных с правильным пространством имен.
 - Нажмите **Добавить**.



Файл *appsettings.json* обновляется с указанием строки подключения, используемой для подключения к локальной базе данных.

Создаваемые файлы

- Проверьте, что в итоге процесса формирования шаблонов были созданы следующие файлы:
 - *Pages/Movies: Create, Delete, Details, Edit и Index.*
 - *Data/RazorPagesMovieContext.cs*

Обновленные возможности

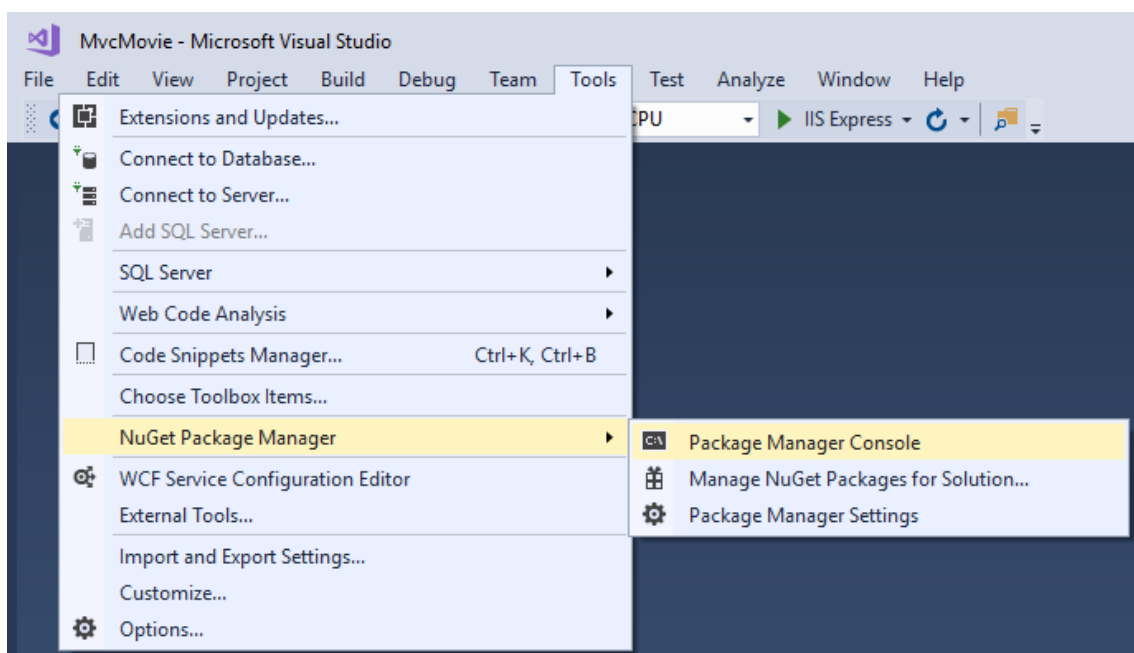
- Проверьте, что в итоге процесса формирования шаблонов обновился файл *Startup.cs*.

В следующем разделе приводится описание созданных и обновленных файлов.

Первоначальная миграция

В этом разделе консоль диспетчера пакетов (PMC) используется для добавления первоначальной миграции и обновления базы данных с ее помощью.

- В меню **Tools (Средства)** последовательно выберите пункты **Диспетчер пакетов NuGet > Консоль диспетчера пакетов**.



- В PMC последовательно введите следующие команды:

```
Add-Migration InitialCreate  
Update-Database
```

В результате выполнения указанных команд выводится следующее предупреждение: "Для десятичного столбца Price в типе сущности Movie не указан тип. Это приведет к тому, что значения будут усекаться без вмешательства пользователя, если они не помещаются в значения точности и масштаба по умолчанию. С помощью метода 'HasColumnType()' явно укажите тип столбца SQL Server, который может вместить все значения".

- Пропустите это предупреждение, так как оно будет рассмотрено на более позднем этапе.

Команда миграции формирует код для создания схемы исходной базы данных. Схема создается на основе модели, указанной в `DbContext`. Аргумент `InitialCreate` используется для присвоения имен миграциям. Можно использовать любое имя, однако по соглашению выбирается имя, которое описывает миграцию.

Команда `update` выполняет метод `Up` в миграциях, которые не были применены. В этом случае `update` запускает метод `Up` в файле `Migrations/<timestamp>_InitialCreate.cs`, который создает базу данных.

Проверка контекста, зарегистрированного с помощью внедрения зависимостей

ASP.NET Core поддерживает [внедрение зависимостей](#). С помощью внедрения зависимостей службы контекст базы данных EF Core, регистрируются во время запуска приложения. Затем компоненты, которые используют эти службы, например Razor Pages, обращаются к ним через параметры конструктора. Код конструктора, который получает экземпляр контекста базы данных, приведен далее в этом руководстве.

Средство формирования шаблонов автоматически создает контекст базы данных и регистрирует его с использованием контейнера внедрения зависимостей.

- Проверьте метод `Startup.ConfigureServices`. Средством формирования шаблонов была добавлена выделенная строка:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

`RazorPagesMovieContext` координирует функции EF Core, такие как `Create` (создание), `Read` (чтение), `Update` (обновление) и `Delete` (удаление) для модели `Movie`.

Контекст данных (`RazorPagesMovieContext`) получен из класса [Microsoft.EntityFrameworkCore.DbContext](#). Контекст данных указывает сущности, которые включаются в модель данных (папка `Data`):

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Data
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (
            DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
    }
}
```

```
}  
}
```

В классе создано свойство [DbSet<Movie>](#) для набора сущностей. В терминологии Entity Framework набор сущностей обычно соответствует таблице базы данных, а конкретная сущность соответствует строке в таблице.

Имя строки подключения передается в контекст путем вызова метода для объекта [DbContextOptions](#). При локальной разработке [система конфигурации](#) считывает строку подключения из файла `appsettings.json`.

Тестирование приложения

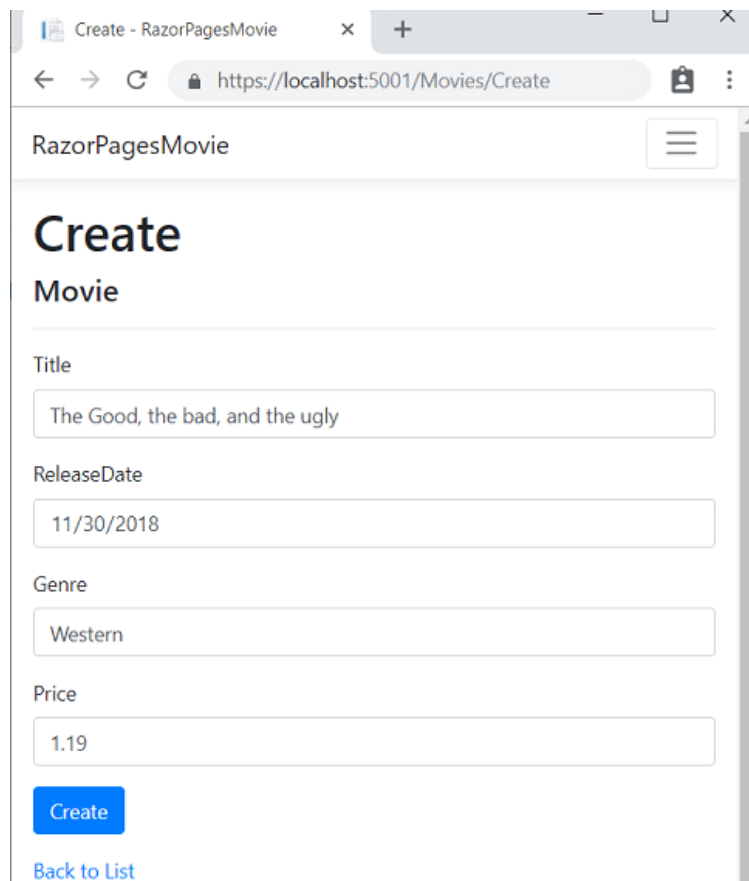
- Запустите приложение и добавьте `/Movies` к URL-адресу в браузере (`http://localhost:port/movies`).

Если возникает ошибка:

SqlException: Cannot open database "RazorPagesMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.

то вы пропустили шаг миграции.

- Протестируйте ссылку **Создать**:



The screenshot shows a web browser window with the address `https://localhost:5001/Movies/Create`. The page title is 'RazorPagesMovie'. The main heading is 'Create Movie'. Below it are four input fields: 'Title' with the text 'The Good, the bad, and the ugly', 'ReleaseDate' with the date '11/30/2018', 'Genre' with the text 'Western', and 'Price' with the value '1.19'. At the bottom left of the form is a blue 'Create' button, and below it is a blue link 'Back to List'.

Примечание. В поле `Price` нельзя вводить десятичные запятые. Чтобы обеспечить поддержку [проверки jQuery](#) для других языков, кроме английского, используйте вместо десятичной точки запятую (,), а для отображения данных в форматах для других языков, кроме английского, выполните глобализацию приложения. Инструкции по глобализации см. на [сайте GitHub](#).

- Протестируйте ссылки **Изменить**, **Сведения** и **Удалить**.

3. Формирование шаблона страницы Razor Pages

В этом разделе описываются страницы Razor Pages, созданные путем формирования шаблонов в предыдущем разделе.

Страницы Create, Delete, Details и Edit

- Откройте и изучите модель страницы *Pages/Movies/Index.cshtml.cs*:

```
// Unused usings removed.
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    #region snippet1
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext
            _context;

        public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext
            context)
        {
            _context = context;
        }
        #endregion
        public IList<Movie> Movie { get; set; }

        public async Task OnGetAsync()
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Страницы Razor Pages являются производными от класса `PageModel`. Как правило, класс, производный от `PageModel`, называется `<PageName>Model`.

Используя внедрение зависимостей, конструктор добавляет на страницу `RazorPagesMovieContext`:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }
}
```

Когда к странице направляется запрос, метод `OnGetAsync` возвращает на страницу Razor список фильмов (дополнительные сведения об асинхронном программировании с использованием Entity Framework см. [Асинхронный код](#)). В

Razor Page для инициализации состояния страницы вызывается `OnGetAsync` или `OnGet`. В этом случае `OnGetAsync` возвращает список фильмов для отображения.

Когда `OnGet` возвращает `void` или `OnGetAsync` возвращает `Task`, оператор `return` не используется. Например, страница "Конфиденциальность":

```
public class PrivacyModel : PageModel
{
    private readonly ILogger<PrivacyModel> _logger;

    public PrivacyModel(ILogger<PrivacyModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
    }
}
```

Если возвращаемый тип — `ActionResult` или `Task<ActionResult>`, необходимо предоставить оператор `return`. Например, метод `OnPostAsync` в *Pages/Movies/Create.cshtml.cs*:

```
public async Task<ActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

- Изучите страницу *Pages/Movies/Index.cshtml*:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
        </tr>
    </thead>
</table>
```

```

        <th>
            @Html.DisplayNameFor(model => model.Movie[0].Price)
        </th>
    </th></th>
</tr>
</thead>
<tbody>
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

```

Razor может выполнять переход с HTML на C# или на разметку Razor: если за символом @ следует зарезервированное ключевое слово Razor, он переходит на разметку Razor (например, @Html), а если нет, то на C# (например, @foreach). Дополнительные сведения см. в [Синтаксис Razor](#).

Директива @page

Директива Razor @page преобразует файл в действие MVC, а значит, он может обрабатывать запросы. @page должна быть первой директивой Razor на странице. @page и @model являются примерами перехода на разметку, относящуюся к Razor.

Директива @model

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

```

Директива @model определяет тип модели, передаваемой на страницу Razor. В приведенном выше примере строка @model делает класс, производный от PageModel, доступным для страницы Razor. Модель используется на странице в [вспомогательных методах HTML](#) @Html.DisplayNameFor и @Html.DisplayFor.

- Изучите лямбда-выражение, которое используется в следующем вспомогательном методе HTML:

```

@Html.DisplayNameFor(model => model.Movie[0].Title)

```

Вспомогательный метод HTML DisplayNameExtensions.DisplayNameFor проверяет свойство Title, указанное в лямбда-выражении, и определяет

отображаемое имя. Лямбда-выражение проверяется, а не вычисляется. Это означает, что в случае, если `model`, `model.Movie` или `model.Movie[0]` имеют значение `null` или пусты, права доступа не нарушаются. При вычислении лямбда-выражения, например с помощью `@Html.DisplayFor(modelItem => item.Title)`, вычисляются значения для свойств модели.

Страница макета

Обратите внимание, что меню на каждой странице имеют одинаковый макет. Макет меню реализован в файле *Pages/Shared/_Layout.cshtml*.

- Откройте и изучите файл *Pages/Shared/_Layout.cshtml*.

Шаблоны макета позволяют сделать для макета контейнера HTML следующее:

- указать его в одном расположении;
- применить его на нескольких страницах сайта.
- Найдите строку `@RenderBody()`. `RenderBody` — это заполнитель для отображения всех представлений определенных страниц, упакованных в страницу макета. Например, если щелкнуть на ссылку **Конфиденциальность**, то представление *Pages/Privacy.cshtml* отобразится в методе `RenderBody`.

ViewData и макет

- Изучите разметку из файла *Pages/Movies/Index.cshtml*:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

Выделенная выше разметка представляет собой пример перехода Razor на C#. Символы `{` и `}` ограничивают блок кода C#.

Базовый класс `PageModel` содержит свойство словаря `ViewData`. Оно позволяет передать данные в представление. Объекты добавляются в словарь `ViewData` с помощью шаблона ***key value** `_`. В приведенном выше примере в словарь `ViewData` добавляется свойство `Title`.

Свойство `Title` используется в файле *_Pages/Shared/_Layout.cshtml*. Ниже показаны первые несколько строк файла *_Layout.cshtml*.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>

    @*Markup removed for brevity.*@
```

Строка `@*Markup removed for brevity.*@` представляет собой комментарий Razor. В отличие от комментариев HTML `<!-- -->` комментарии Razor не отправляются клиенту.

Обновление макета

- В файле *Pages/Shared/_Layout.cshtml* измените элемент `<title>` так, чтобы вместо **Movie** отображалось **RazorPagesMovie**.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ ViewData["Title"] - Movie</title>
```

- Найдите в файле *Pages/Shared/_Layout.cshtml* следующий элемент привязки.

```
<a class="navbar-brand" asp-area="" asp-page="/Index">RazorPagesMovie</a>
```

- Измените указанный выше элемент на следующую разметку.

```
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

Указанный выше элемент привязки является [вспомогательной функцией тега](#). В данном случае он является [вспомогательной функцией тега привязки](#). Атрибут вспомогательной функции тега `asp-page="/Movies/Index"` и его значение создают ссылку на страницу Razor `/Movies/Index`. Атрибут `asp-area` имеет пустое значение, поэтому эта область не используется в ссылке.

- Сохраните изменения и протестируйте приложение, выбрав ссылку **RpMovie**. Если у вас возникли проблемы, ознакомьтесь с файлом [_Layout.cshtml](#) в [GitHub](#).
- Проверьте ссылки **Home**, **RpMovie**, **Create**, **Edit** и **Delete**. Каждая страница задает заголовок, который отображается на вкладке браузера. При добавлении страницы в избранное заголовок используется в закладках.

Примечание. В поле `Price` нельзя вводить десятичные запятые. Чтобы обеспечить поддержку [проверки jQuery](#) для других языков, кроме английского, используйте вместо десятичной точки запятую (","), а для отображения данных в форматах для других языков, кроме английского, выполните действия, необходимые для глобализации приложения. Инструкции по добавлению десятичной запятой см. в [вопросе № 4076 на сайте GitHub](#).

- Проверьте, что свойство `Layout` определяется в файле *Pages/_ViewStart.cshtml*:

```
@{
    Layout = "_Layout";
}
```

Данный код задает файл разметки *Pages/Shared/_Layout.cshtml* для всех файлов Razor в папке *Pages*.

Модель Create

- Изучите модель *Pages/Movies/Create.cshtml.cs*:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
```

```

{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext
_context;

        public CreateModel(RazorPagesMovie.Data.RazorPagesMovieContext
context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}

```

Метод `OnGet` инициализирует все состояния, необходимые для страницы.

Страница `Create` не содержит никаких состояний для инициализации, поэтому возвращается `Page`. Далее будет показан пример инициализации состояния `OnGet`. Метод `Page` создает объект `PageResult`, который формирует страницу *Create.cshtml*.

Для указания согласия на привязку модели в свойстве `Movie` используется атрибут [\[BindProperty\]](#). Когда форма `Create` публикует свои значения, среда выполнения ASP.NET Core связывает переданные значения с моделью `Movie`.

Метод `OnPostAsync` выполняется, когда страница публикует данные формы:

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

```

Если в модели есть ошибки, форма отображается снова вместе со всеми опубликованными данными этой формы. Большинство ошибок в модели может быть

перехвачено на стороне клиента до публикации формы, например, ввод значения для поля даты, которое нельзя конвертировать в дату.

Проверка на стороне клиента и проверка модели обсуждаются подробнее далее в этом руководстве.

Если нет ошибок модели, то данные будут сохранены и браузер перенаправляется на страницу Index.

- Изучите файл страницы *Razor Pages/Movies/Create.cshtml*:

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-
label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>
```


[Вспомогательная функция тега Label](#) (<label asp-for="Movie.Title" class="control-label"></label>) создает подпись к метке и атрибут [for] для свойства Title.

[Вспомогательная функция тега Input](#) (<input asp-for="Movie.Title" class="form-control">) использует атрибуты [DataAnnotations](#) и создает HTML-атрибуты, необходимые для проверки jQuery на стороне клиента.

Дополнительные сведения о вспомогательных функциях тегов, таких как <form method="post">, см. [Вспомогательные функции тегов в ASP.NET Core](#).

4. Работа с базой данных

Объект `RazorPagesMovieContext` обрабатывает задачу подключения к базе данных и сопоставления объектов `Movie` с записями базы данных. Контекст базы данных регистрируется с помощью контейнера [внедрения зависимостей](#) в методе `ConfigureServices` в файле *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<RazorPagesMovieContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

Система конфигурации ASP.NET Core считывает ключ `ConnectionString`. Для разработки на локальном уровне конфигурация получает строку подключения из файла *appsettings.json*.

Созданная строка подключения будет выглядеть следующим образом (JSON):

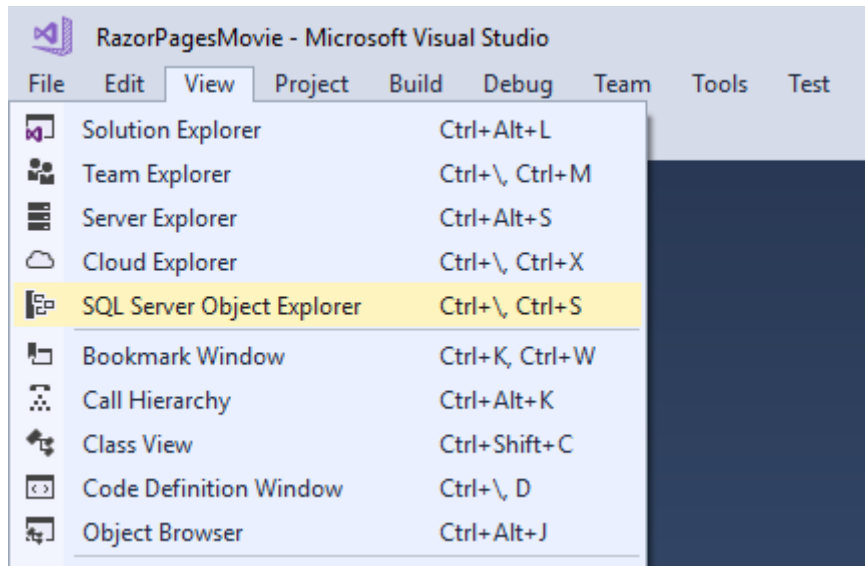
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext":
"Server=(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
bc;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

Если приложение разворачивается на тестовом или рабочем сервере, можно задать строку подключения к тестовому или рабочему серверу базы данных с помощью переменной среды.

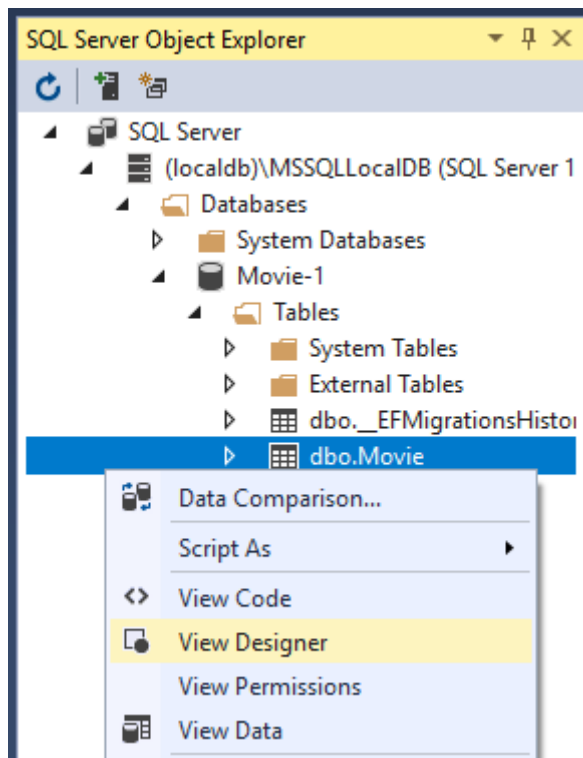
SQL Server Express LocalDB

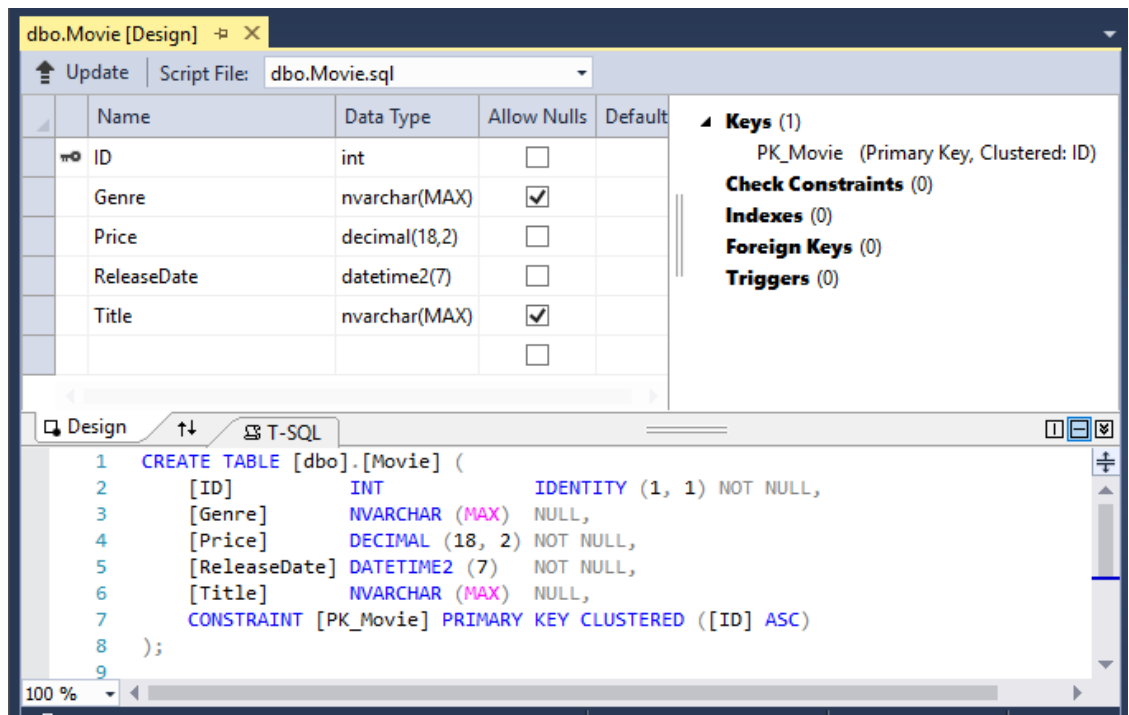
LocalDB представляет собой упрощенную версию ядра СУБД SQL Server Express, предназначенную для разработки программ. LocalDB запускается по запросу в пользовательском режиме. По умолчанию база данных LocalDB создает файлы *.mdf в каталоге C:\Users\<user>\.

- В меню **View (Вид)** откройте **обозреватель объектов SQL Server (SSOX)**.



- Щелкните правой кнопкой мыши таблицу `Movie` и выберите пункт **Конструктор представлений**:





Обратите внимание на значок с изображением ключа рядом с ID. По умолчанию EF создает свойство с именем ID для первичного ключа.

- Щелкните правой кнопкой мыши таблицу Movie и выберите пункт **Просмотреть данные:**

| ID | Genre | Price | ReleaseDate | Title |
|------|---------|-------|--------------------|-----------------------------|
| 3 | Action | 1.99 | 1/1/0001 12:00:... | Conan |
| 2003 | Comedy | 1.99 | 8/4/2017 6:27:3... | Back to the Future |
| 2004 | Western | 1.19 | 8/4/2017 7:08:3... | The Good, the bad, and t... |
| NULL | NULL | NULL | NULL | NULL |

Заполнение базы данных

- Создайте класс `SeedData` в папке *Models* со следующим кодом:

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{

```

```

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new RazorPagesMovieContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<RazorPagesMovieContext>>()))
        {
            // Look for any movies.
            if (context.Movie.Any())
            {
                return;    // DB has been seeded
            }

            context.Movie.AddRange(
                new Movie
                {
                    Title = "When Harry Met Sally",
                    ReleaseDate = DateTime.Parse("1989-2-12"),
                    Genre = "Romantic Comedy",
                    Price = 7.99M
                },

                new Movie
                {
                    Title = "Ghostbusters ",
                    ReleaseDate = DateTime.Parse("1984-3-13"),
                    Genre = "Comedy",
                    Price = 8.99M
                },

                new Movie
                {
                    Title = "Ghostbusters 2",
                    ReleaseDate = DateTime.Parse("1986-2-23"),
                    Genre = "Comedy",
                    Price = 9.99M
                },

                new Movie
                {
                    Title = "Rio Bravo",
                    ReleaseDate = DateTime.Parse("1959-4-15"),
                    Genre = "Western",
                    Price = 3.99M
                }
            );
            context.SaveChanges();
        }
    }
}

```

Если в базе данных есть фильмы, возвращается инициализатор заполнения и фильмы не добавляются:

```

if (context.Movie.Any())
{
    return;
}

```

Добавление инициализатора заполнения

- Замените содержимое файла *Program.cs* следующим кодом:


```

using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger =
services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                })
            ;
    }
}

```

В приведенном выше коде метод `Main` изменен, чтобы сделать следующее:

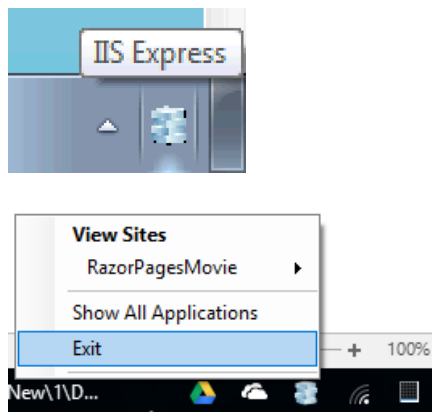
- Получение экземпляра контекста базы данных из контейнера внедрения зависимостей.
- Вызов метода `seedData.Initialize`, передав ему экземпляр контекста базы данных.
- Высвобождение контекста после завершения работы метода заполнения. Оператор `using` гарантирует удаление контекста.

Если `Update-Database` не выполнялось, возникает следующее исключение:

`SqlException: Cannot open database "RazorPagesMovieContext-" requested by the login. The login failed. Login failed for user 'user name'.`

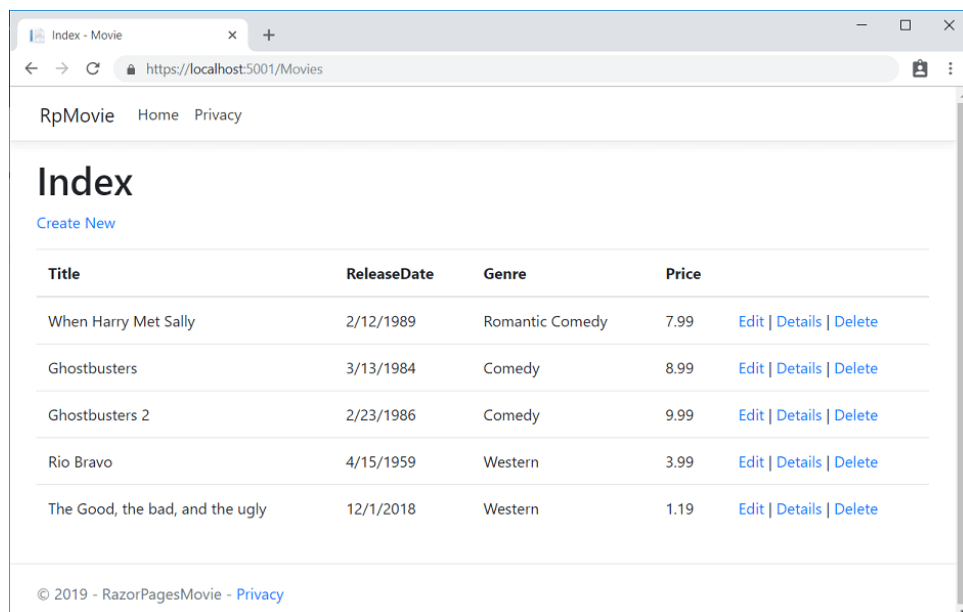
Тестирование приложения

- Удалите все записи в базе данных. Используйте ссылки удаления в браузере или в SSOX.
- Необходимо выполнить инициализацию (вызывать методы в классе `Startup`), чтобы запустить метод заполнения. Для этого следует остановить и перезапустить IIS Express. Закройте и перезапустите службы IIS с помощью любого из перечисленных ниже подходов.
 - Щелкните правой кнопкой мыши значок IIS Express в области уведомлений и выберите **Выйти** или **Остановить сайт**.



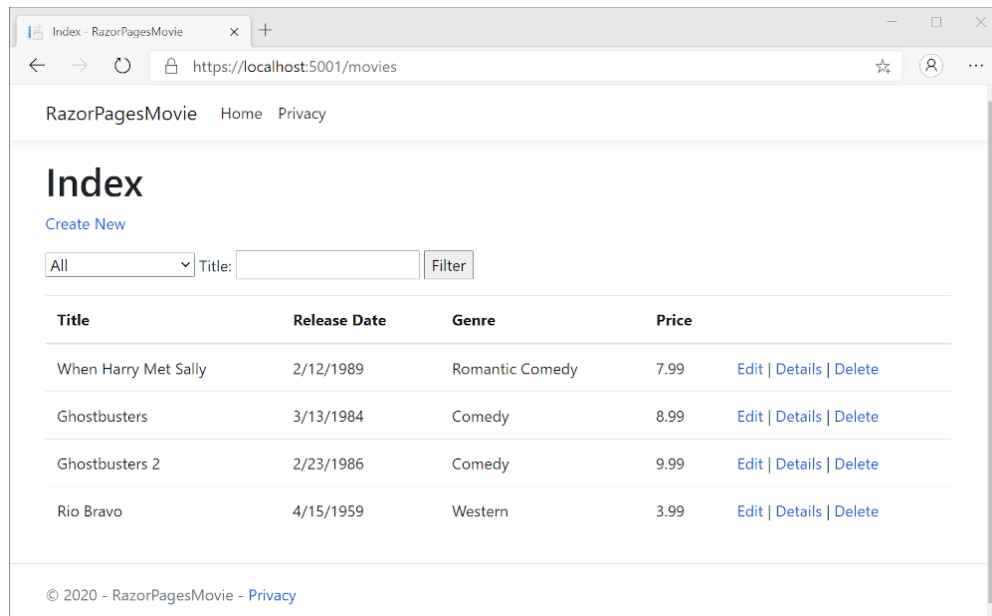
- Если приложение работает в режиме без отладки, нажмите клавишу `F5` для запуска в режиме отладки.
- Если приложение находится в режиме отладки, остановите отладчик и нажмите клавишу `F5`.

В приложении должны отображаться заполненные данные.



5. Изменение созданных страниц в приложении ASP.NET Core

Приложение для работы с фильмами подготовлено, но представление данных для практического использования неудобно, например, вместо **ReleaseDate** должно стоять **Release Date** (или можно на русском языке так и указать **Дата выпуска**), то есть два слова:



Обновление созданного кода

- Откройте файл *Models/Movie.cs* и добавьте указанные ниже выделенные строки кода:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

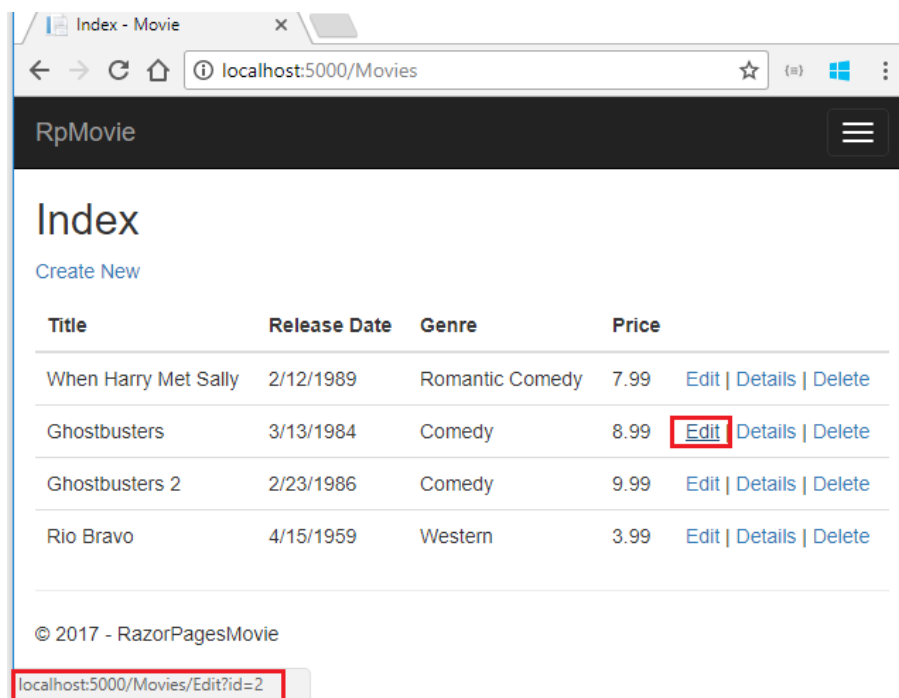
В добавленном коде:

- Атрибут к данным `[Column(TypeName = "decimal(18, 2)")]` позволяет Entity Framework Core корректно сопоставить Price с валютой в базе данных. Дополнительные сведения см. в [Типы данных](#).

- Атрибут [\[Display\]](#) указывает на отображаемое имя поля. В приведенном выше коде ReleaseDate заменен на Release Date ("Дата выпуска").
- Атрибут [\[DataType\]](#) указывает тип данных (Date). Сведения о времени, хранящиеся в поле, не отображаются.

Пространство имен **DataAnnotations** будет рассмотрено далее.

- Запустите приложение, перейдите к *Pages/Movies* и наведите указатель мыши на ссылку **Edit** (Изменение), чтобы просмотреть целевой URL-адрес.



Ссылки **Edit**, **Details** и **Delete** создаются вспомогательной функцией тегов привязки в файле *Pages/Movies/Index.cshtml*.

```
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>
```

Вспомогательные функции тегов позволяют серверному коду участвовать в создании и отображении HTML-элементов в файлах Razor.

В приведенном выше коде вспомогательная функция привязки тегов динамически создает значение атрибута HTML `href` на основе Razor Page (маршрут является относительным), атрибут `asp-page` и идентификатор маршрута (`asp-route-id`). Дополнительные сведения см. в разделе [Формирование URL-адресов для страниц](#).

- Для проверки созданной разметки используйте в браузере параметр **Просмотреть исходный код**. Ниже показана часть созданного кода HTML:

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

В динамически созданных ссылках идентификаторы фильмов передаются с помощью строки запроса.

Например, `?id=1` в `https://localhost:5001/Movies/Details?id=1`.

Добавление шаблона маршрута

- Обновите страницы `Edit`, `Details` и `Delete` так, чтобы использовался шаблон маршрута `{id:int}` — для этого измените директиву страницы для каждой из этих страниц с `@page` на `@page "{id:int}"`.
- Запустите приложение и просмотрите исходный код.
- Проверьте, что созданный код HTML добавляет идентификатор в путь URL-адреса:

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

Запрос к странице с шаблоном маршрута `{id:int}`, который **не включает** в себя целое число, приводит к ошибке HTTP 404 (не найдено). Например, `https://localhost:5001/Movies/Details` приведет к ошибке 404. Чтобы сделать идентификатор необязательным, добавьте `?` к ограничению маршрута:

`@page "{id:int?}"`

- Чтобы проверить поведение `@page "{id:int?}"`:
 - Задайте директиву страницы `Pages/Movies/Details.cshtml` как `@page "{id:int?}"`.
 - Установите точку останова на `public async Task<IActionResult> OnGetAsync(int? id)` в `Pages/Movies/Details.cshtml.cs`.
 - Перейдите к `https://localhost:5001/Movies/Details/`.

Из-за директивы `@page "{id:int}"` точка останова не достигается. Механизм маршрутизации возвращает ошибку HTTP 404. При использовании `@page "{id:int?}"` метод `OnGetAsync` возвращает `NotFound` (HTTP 404):

```

public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Movie = await _context.Movie.FirstOrDefaultAsync(m => m.ID == id);

    if (Movie == null)
    {
        return NotFound();
    }
    return Page();
}

```

Проверка обработки исключений блокировки

- Проверьте метод `OnPostAsync` в файле *Pages/Movies/Edit.cshtml.cs*.

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.ID == id);
}

```

Этот код обнаруживает исключения параллелизма, когда один клиент удаляет фильм, а другой вносит изменения в фильм.

- Чтобы протестировать блок `catch`, выполните указанные ниже действия:
1. Задайте точку останова в `catch (DbUpdateConcurrencyException)`.
 2. Выберите команду **Изменить** у фильма, внесите изменения, но не вводите **Сохранить**.

3. В другом окне браузера щелкните ссылку **Delete** для этого же фильма, а затем удалите его.
4. В первом окне браузера опубликуйте изменения для фильма.

Коду в рабочей среде может потребоваться обнаружение конфликтов параллелизма. Дополнительные сведения см. в статье [Обработка конфликтов параллелизма](#).

Проверка публикации и привязки

- Изучите файл *Pages/Movies/Edit.cshtml.cs*:

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Movie = await _context.Movie.FirstOrDefaultAsync(m => m.ID == id);

        if (Movie == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(Movie.ID))
            {
                return NotFound();
            }
            else
            {
                // ...
            }
        }
    }
}
```

```

        throw;
    }
}

return RedirectToPage("../Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.ID == id);
}

```

При выполнении HTTP-запроса GET к странице *Movies/Edit*, например <https://localhost:5001/Movies/Edit/3>, происходит следующее:

- Метод `OnGetAsync` извлекает запись фильма из базы данных и возвращает метод `Page`.
- Метод `Page` отображает страницу *Razor Pages/Movies/Edit.cshtml*. Файл *Pages/Movies/Edit.cshtml* содержит директиву модели `@model RazorPagesMovie.Pages.Movies.EditModel`, которая делает модель фильма доступной на странице.
- Отображается форма *Edit* со значениями из записи фильма.

При публикации страницы *Movies/Edit* происходит следующее:

- Значения формы на странице привязываются к свойству `Movie`. Атрибут `[BindProperty]` обеспечивает привязку модели.

```

[BindProperty]
public Movie Movie { get; set; }

```

- При наличии ошибок в состоянии модели, например `ReleaseDate` невозможно преобразовать в дату, форма отображается повторно с предоставленными значениями.
- Если ошибки модели отсутствуют, данные фильма сохраняются.

Методы HTTP GET на страницах *Razor Index*, *Create* и *Delete* работают аналогично.

Метод HTTP POST `OnPostAsync` на странице *Razor Create* работает аналогично методу `OnPostAsync` на странице *Razor Edit*.

6. Добавление поиска

В этой части руководства добавляется поиск фильмов по *жанру* или *имени*.

Поиск по имени

- Откройте файл *Pages/Movies/Index.cshtml.cs* и добавьте следующие выделенные инструкцию **using** и свойства:

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using System.Collections.Generic;

```



```

using System.Linq;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext
_context;

        public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext
context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get; set; }
        [BindProperty(SupportsGet = true)]
        public string SearchString { get; set; }
        public SelectList Genres { get; set; }
        [BindProperty(SupportsGet = true)]
        public string MovieGenre { get; set; }
    }
}

```

Добавлены свойства:

- SearchString: будет содержать текст, который пользователи вводят в поле поиска. SearchString также имеет атрибут [\[BindProperty\]](#). [BindProperty] связывает значения из формы и строки запроса с тем же именем, что и у свойства. [BindProperty(SupportsGet = true)] требуется для привязки в запросах HTTP GET.
- Genres: содержит список жанров. Genres дает пользователю возможность выбрать жанр в списке. Для SelectList требуется using Microsoft.AspNetCore.Mvc.Rendering;.
- MovieGenre: содержит конкретный жанр, выбранный пользователем. Например, "Боевик".
- Genres и MovieGenre рассматриваются позднее в этом руководстве.

Замечание. В целях безопасности следует задать привязку данных запроса GET к свойствам страничной модели. Проверьте введенные данные пользователя, прежде чем сопоставлять их со свойствами. Привязка GET полезна при обращении к сценариям, использующим строку запроса или значения маршрутов. Чтобы привязать свойство к запросам GET, для свойства SupportsGet атрибута [BindProperty] было задано значение true:

```
[BindProperty(SupportsGet = true)]
```

- Обновите метод OnGetAsync страницы Index, используя следующий код:

```

public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                  select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }
}

```

```

        Movie = await movies.ToListAsync();
    }

```

Замечания по добавленному коду:

- В первой строке метода `OnGetAsync` создается запрос LINQ для выбора фильмов:

```

// using System.Linq;
var movies = from m in _context.Movie
              select m;

```

Этот запрос только определяется в этой точке и не выполняется для базы данных.

- Если свойство `SearchString` не равно `NULL` и не пусто, запрос фильмов изменяется для фильтрации по строке поиска:

```

if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}

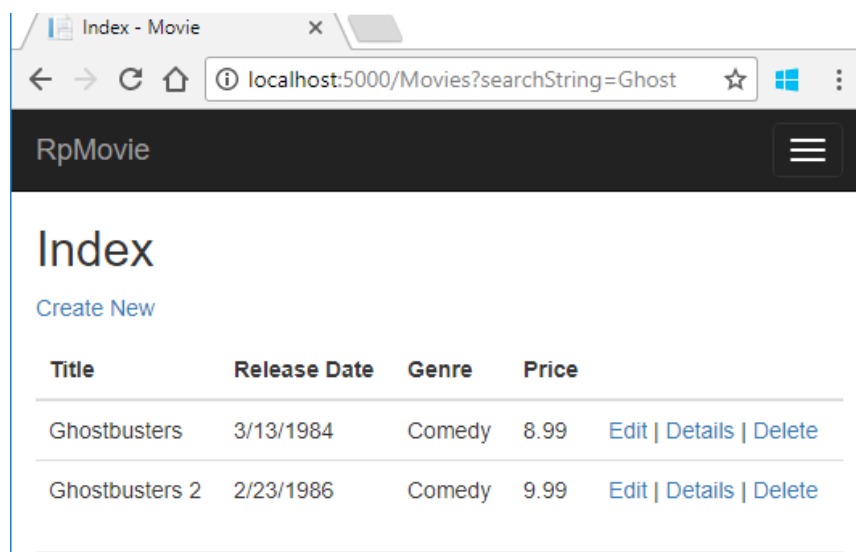
```

Код `s => s.Title.Contains()` представляет собой лямбда-выражение. Лямбда-выражения используются в запросах LINQ на основе методов в качестве аргументов стандартных методов операторов запроса, таких как метод `Where` или `Contains`. Запросы LINQ не выполняются, если они определяются или изменяются путем вызова метода, например `Where`, `Contains` или `OrderBy`. Вместо этого выполнение запроса откладывается. Вычисление выражения откладывается до тех пор, пока не будет выполнена итерация его реализованного значения или не будет вызван метод `ToListAsync()`.

Метод `Contains()` выполняется в базе данных, а не в коде C#. Регистр символов запроса учитывается в зависимости от параметров базы данных и сортировки. В SQL Server метод `Contains()` сопоставляется с SQL LIKE, в котором регистр символов не учитывается.

- Перейдите на страницу `Movies` и добавьте строку запроса, например, `?searchString=Ghost`, к URL-адресу.

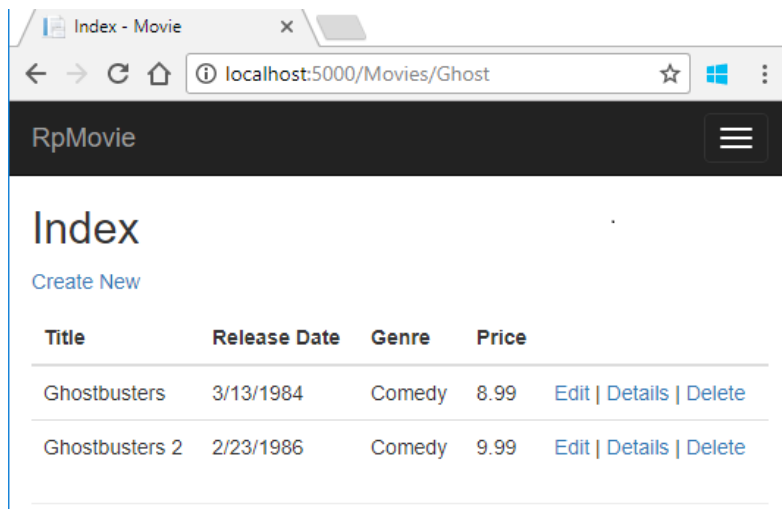
Должны отобразиться отфильтрованные фильмы.



- Добавьте на страницу `Index` следующий шаблон маршрута,

```
@page "{searchString?}"
```

- Теперь передайте в строку адреса запрос поиска в виде сегмента URL-адреса. Например, `https://localhost:5001/Movies/Ghost`.



Предыдущее ограничение маршрута разрешало поиск названия в виде данных маршрута (сегмент URL-адреса) вместо значения строки запроса.

Символ `?` в `"{searchString?}"` означает, что этот параметр является необязательным.

Замечание. Среда выполнения ASP.NET Core использует привязку модели, чтобы присвоить значение свойства `SearchString` по строке запроса (`?searchString=Ghost`) или данным маршрута (`https://localhost:5001/Movies/Ghost`) — однако пользователи не могут изменять URL-адрес для поиска фильма.

На этом шаге добавляется пользовательский интерфейс для поиска фильмов.

- Откройте файл `_Pages/Movies/Index.cshtml` и удалите добавленное недавно ограничение маршрута `"{searchString?}"`
- Добавьте разметку, выделенную в следующем коде:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

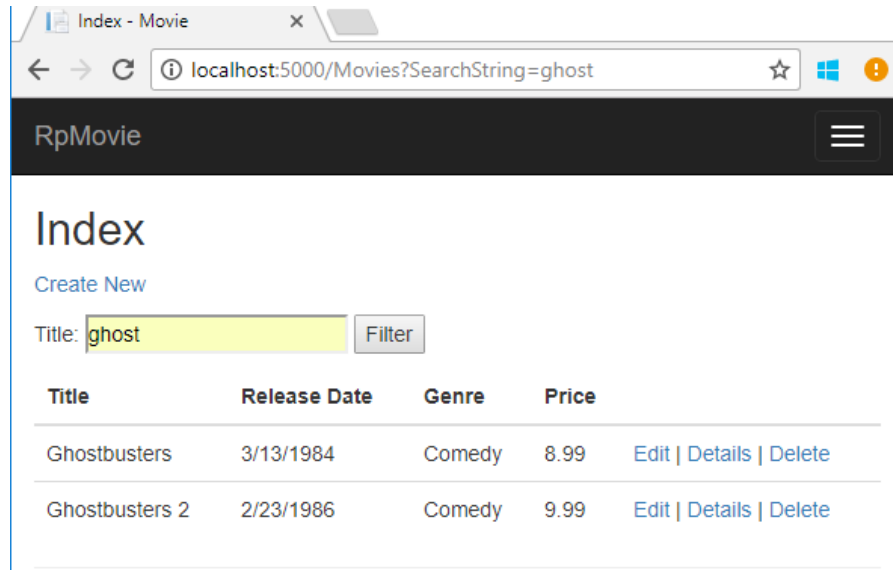
<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

Тег HTML `<form>` использует следующие вспомогательные функции тегов:

- вспомогательная функция тега форм. При отправке формы строка фильтра отправляется на страницу *Pages/Movies/Index* через строку запроса.
- вспомогательная функция тега Input
- Сохраните изменения и проверьте работу фильтра.



Поиск по жанру

- Добавьте в метод `OnGetAsync` страницы `Index` код, реализующий запрос по жанру:

```
public async Task OnGetAsync()
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                  select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

Пояснение к коду. Определяется запрос LINQ, который извлекает все жанры из базы данных:

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

Список жанров `SelectList` создается путем проецирования отдельных жанров:

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Добавление поиска по жанру на страницу Razor

- В файле *Index.cshtml* обновите элемент `<form>`, как показано в следующей разметке:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

- Проверьте работу приложения, выполнив поиск по жанру, по названию фильма и по обоим этим параметрам.

7. Добавление нового поля

В этом разделе руководства будет использоваться механизм миграции **Entity Framework Code First Migrations** для добавления нового поля в модель и переноса изменений в схему в базу данных.

Если для автоматического создания базы данных используется EF Code First, то он добавляет в базу данных таблицу [__EFMigrationsHistory](#), которая позволяет отслеживать синхронизацию схемы базы данных с классами модели, на основе которой она была создана. Если классы модели не синхронизированы с базой данных, EF выдает исключение.

Автоматическая проверка синхронизации схемы и модели упрощает поиск несогласованных проблем в коде базы данных.

Добавление свойства *Rating* в модель *Movie*

- Откройте файл *Models/Movie.cs* и добавьте свойство `Rating`:

```
public class Movie
{
    public int ID { get; set; }
```

```

public string Title { get; set; }

[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }
public string Genre { get; set; }

[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
public string Rating { get; set; }
}

```

- Постройте приложение.
- В файл *Pages/Movies/Index.cshtml* добавьте новое поле Rating:

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
        {
            <tr>

```

```

        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>
        <td>
            <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a>
            |
            <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
            <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

```

- Обновите следующие cshtml-страницы, добавив в них разметку для поля `Rating` (аналогично как это сделано для других полей модели):
 - Delete и Details.
 - Create и Edit.
- Запустите приложение и перейдите на страницу `Movies`. При запуске приложения без обновления базы данных возникает исключение `SqlException`:

`SqlException: Invalid column name 'Rating'.`

Исключение `SqlException` связано с тем, что обновленный класс модели `Movie` отличается от схемы таблицы `Movie` в базе данных. В таблице базы данных нет столбца `Rating`. Для работы приложения необходимо обновить базу данных, включив в нее новое поле.

Устранить эту ошибку можно несколькими способами:

- Можно с помощью Entity Framework автоматически удалить и повторно создать базу данных на основе новой схемы класса модели. Этот подход удобен на ранних стадиях цикла разработки. В этом случае развитие модели и схемы базы данных осуществляется одновременно. Недостатком такого подхода является потеря существующих данных в базе. В рабочей базе данных применять этот подход не следует. При разработке приложения часто выполняется удаление базы данных при изменении схемы, для чего используется инициализатор для автоматического заполнения базы тестовыми данными.
- Можно явно изменить схему существующей базы данных в соответствии с новыми классами модели. Преимущество подхода в том, что он сохраняет

данные. Внести это изменение можно вручную либо путем создания скрипта изменения для базы данных.

- Можно обновить схему базы данных с помощью Code First Migrations.

В этом руководстве используется Code First Migrations.

*Добавление миграции для поля **Rating***

- Откройте окно **Консоль диспетчера пакетов**.
- В консоль последовательно введите следующие команды:

```
Add-Migration Rating
```

```
Update-Database
```

Команда `Add-Migration` задает следующие инструкции для платформы:

- Сравнить модель `Movie` со схемой базы данных `Movie`.
- Создать код для переноса схемы базы данных в новую модель.

В качестве имени файла переноса используется произвольное имя "Rating". Рекомендуется присваивать этому файлу понятное имя.

Команда `Update-Database` указывает платформе, что к базе данных нужно применить изменения схемы, а также сохранить существующие данные.

- Запустите приложение. Перейдите на страницу `Movies` и проверьте функциональность – возможность создания, редактирования и отображения фильмов с использованием поля `Rating`.
- Обратите внимание, что поле `Rating` изначально пустое.
- Обновите класс `SeedData` так, чтобы он предоставлял значение нового столбца для каждого фильма. Ниже приведен пример изменения, которое необходимо реализовать для каждого блока `new Movie`:

```
context.Movie.AddRange(  
    new Movie  
    {  
        Title = "When Harry Met Sally",  
        ReleaseDate = DateTime.Parse("1989-2-12"),  
        Genre = "Romantic Comedy",  
        Price = 7.99M,  
        Rating = "R"  
    },  
    );
```

Можете уточнить содержимое файла [готовый файл SeedData.cs](#).

- Постройте решение.

Для того, чтобы метод `SeedData.Initialize()` заполнял базу начальными значениями необходимо и в нее будет включено поле `Rating` необходимо удалить все записи из базы данных. Это можно сделать с помощью ссылок удаления в браузере или из обозревателя объектов SQL Server (SSOX).

Другой вариант — удалить базу данных и использовать миграции для повторного создания базы данных.

- Удалите базу данных с применением SSOX:

- Выберите базу данных в SSOX.
- Щелкните базу данных правой кнопкой мыши и выберите **Удалить**.
- В появившемся окне выберите **Заккрыть существующие соединения**.
- Нажмите кнопку **ОК**.
- Обновите базу данных в консоли диспетчера пакетов:
Update-Database
- Запустите приложение и проверьте возможность создания, редактирования и отображения фильмов с использованием поля `Rating`.

8. Проверка данных

В этом разделе к модели `Movie` добавляется логика проверки. Правила проверки применяются каждый раз, когда пользователь создает или редактирует фильм.

Проверка

Ключевой принцип разработки программного обеспечения называется DRY (от английского "**D**on't **R**epeat **Y**ourself" — не повторяйся). При разработке Razor Pages рекомендуется задавать любые функциональные возможности лишь один раз и затем при необходимости отражать их в рамках всего приложения. Принцип "Не повторяйся" может помочь сократить объем кода в приложении и снизить вероятность возникновения ошибки в коде, а также упростить его тестирование и поддержку.

Примером применения принципа "Не повторяйся" является поддержка проверки, реализуемая в Razor Pages и на платформе Entity Framework:

- ✓ Правила проверки декларативно определяются в одном месте — в классе модели и применяются по всему приложению.

Добавление правил проверки к модели фильма

Пространство имен `DataAnnotations` предоставляет:

- Набор встроенных атрибутов проверки, которые декларативно применяются к классу или свойству.
- Атрибуты форматирования (такие как `[DataType]`), которые обеспечивают форматирование и не предназначены для проверки.
- Добавьте в класс `Movie` указанные ниже атрибуты, чтобы использовать преимущества встроенных атрибутов проверки `[Required]`, `[StringLength]`, `[RegularExpression]` и `[Range]`:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
```

```

{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9"''\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}

```

Атрибуты проверки определяют поведение для свойств модели, к которым они применяются:

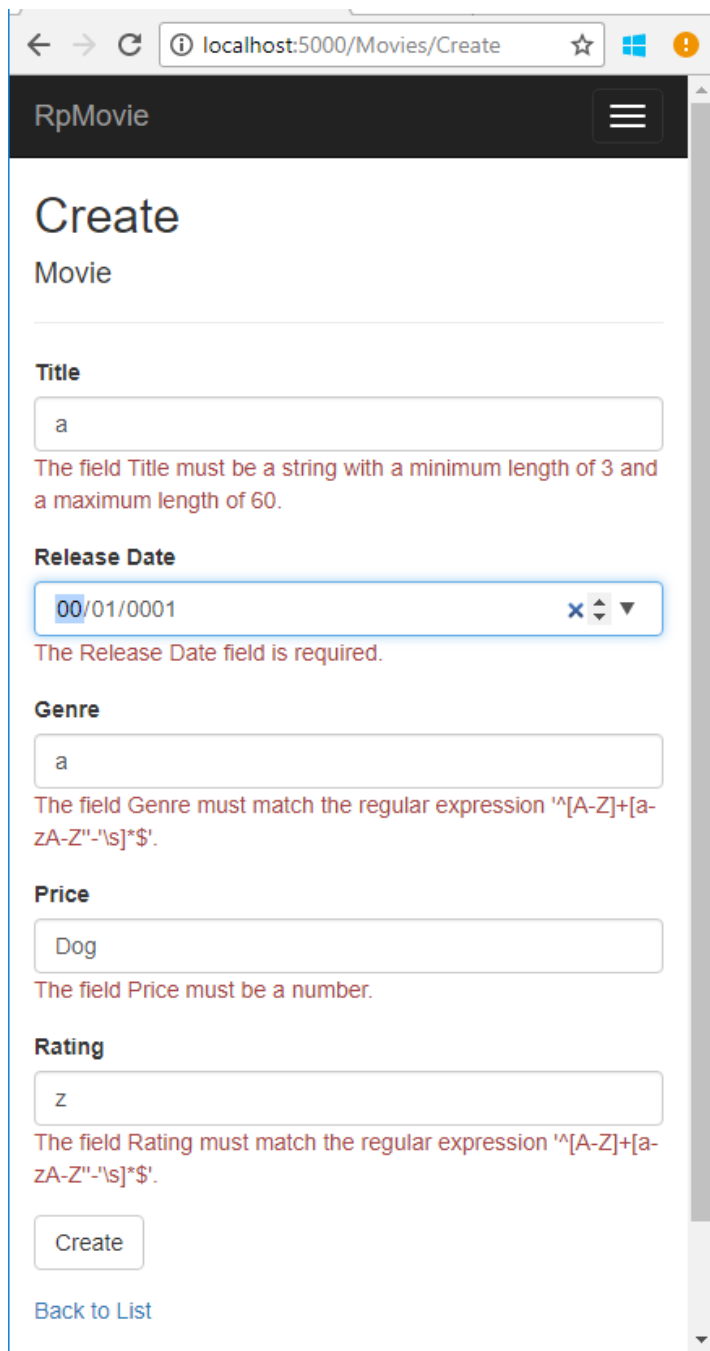
- Атрибуты [Required] и [MinimumLength] означают, что свойство должно иметь значение. Пользователь может свободно вводить пробелы для выполнения этой проверки.
- Атрибут [RegularExpression] ограничивает набор допустимых для ввода символов. В приведенном выше коде в Genre:
 - должны использоваться только буквы;
 - первая буква должна быть прописной; пробелы, цифры и специальные символы не допускаются.
- RegularExpression Rating:
 - первый символ должен быть прописной буквой;
 - допускаются специальные символы и цифры, а также последующие пробелы. значение "PG-13" допустимо для рейтинга, но недопустимо для Genre;
- атрибут [Range] ограничивает значения указанным диапазоном.
- Атрибут [StringLength] может задавать максимальную и, при необходимости, минимальную длину строкового свойства.
- Типы значений (например, decimal, int, float, DateTime) по своей природе являются обязательными и не требуют атрибута [Required].

Указанные выше правила проверки используются для демонстрации, они не являются оптимальными для рабочей системы. Например, предыдущее исключение не позволяет ввести модель Movie с двумя символами и не допускает специальные знаки в Genre.

Наличие правил проверки, которые автоматически применяются ASP.NET Core, помогает повысить степень надежности приложения и сократить возможность сохранения недопустимых данных в базе данных.

Тестирование пользовательского интерфейса проверки ошибок в Razor Pages

- Запустите приложение и перейдите в раздел "Pages/Movies".
- Щелкните ссылку **Create New**. Введите в форму какие-либо недопустимые значения. Если функция проверки jQuery на стороне клиента обнаруживает ошибку, сведения о ней отображаются в соответствующем сообщении.



The screenshot shows a web browser window with the address bar displaying 'localhost:5000/Movies/Create'. The page title is 'RpMovie'. The main heading is 'Create Movie'. Below the heading, there are five input fields, each with a validation error message in red text:

- Title:** The field Title must be a string with a minimum length of 3 and a maximum length of 60. (The input is 'a').
- Release Date:** The Release Date field is required. (The input is '00/01/0001').
- Genre:** The field Genre must match the regular expression `^[A-Z]+[a-zA-Z'-\s]*$`. (The input is 'a').
- Price:** The field Price must be a number. (The input is 'Dog').
- Rating:** The field Rating must match the regular expression `^[A-Z]+[a-zA-Z'-\s]*$`. (The input is 'z').

At the bottom of the form, there is a 'Create' button and a 'Back to List' link.

Примечание. Возможно, вы не сможете вводить десятичные запятые в полях для десятичных чисел. Чтобы обеспечить поддержку [проверки jQuery](#) для других языков, кроме английского, используйте вместо десятичной точки запятую (","), а

для отображения данных в форматах для других языков, кроме английского, выполните действия, необходимые для глобализации вашего приложения. Инструкции по добавлению десятичной запятой см. в [вопросе № 4076 на сайте GitHub](#).

Обратите внимание, что для каждого поля, содержащего недопустимое значение, в форме автоматически отображается сообщение об ошибке проверки. Эти ошибки применяются как на стороне клиента (с помощью JavaScript и jQuery), так и на стороне сервера (если пользователь отключает JavaScript).

Основным преимуществом является то, что на страницах создания или редактирования **не требуется** изменять код. Пользовательский интерфейс проверки активируется после применения к модели атрибутов проверки достоверности.

В Razor Pages, создаваемом в рамках этого руководства, правила проверки применяются автоматически (для этого к свойствам класса модели `Movie` применяются атрибуты проверки). При проверке страницы редактирования применяются те же правила.

Данные формы передаются на сервер только после того, как будут устранены все ошибки проверки на стороне клиента.

- Чтобы убедиться, что данные формы не отправляются поместите точку останова в метод `OnPostAsync` (файл `Create.cshtml`). При тестировании неправильными данными проверьте, что точка останова не достигается ни при каких обстоятельствах.

Проверка на стороне сервера

Если в браузере отключен JavaScript, форма с ошибками отправляется на сервер.

Реализация проверки на стороне сервера:

- Отключите JavaScript в браузере. JavaScript можно отключить с помощью средств разработчика в браузере. Если JavaScript невозможно отключить в этом браузере, попробуйте использовать другой браузер.

Можно отключить проверку на стороне клиента закомментировав ссылку на `_ValidationScriptsPartial` в файле страницы CSHTML:

```
@section Scripts {  
    @* @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}*  
}
```

- Поместите точку останова в метод `OnPostAsync` страниц создания или редактирования.
- Отправьте форму с недопустимыми данными.

Проверка недопустимого состояния модели будет остановлена следующим кодом:

```
if (!ModelState.IsValid)  
{  
    return Page();  
}
```

```
}
```

В следующем коде демонстрируется часть страницы *Create.cshhtml*, созданной ранее в рамках этого руководства. Она используется на страницах создания и редактирования для выполнения следующих действий:

- Отображения начальной формы.
- Повторного отображения формы в случае ошибки.

```
<form method="post">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
  </div>
```

Вспомогательная функция тега `Input` использует атрибуты `DataAnnotations` и создает HTML-атрибуты, необходимые для проверки jQuery на стороне клиента. Вспомогательная функция тега `Validation` отображает ошибки проверки.

На страницах создания и редактирования не определены правила проверки. Правила проверки и строки ошибок указываются только в классе `Movie`. Они автоматически применяются к Razor Pages, которые редактируют модель `Movie`.

Любые необходимые изменения логики проверки осуществляются исключительно в модели. Проверка применяется согласованно на уровне всего приложения, для чего логика проверки определяется в одном месте. Такой подход позволяет максимально оптимизировать код и упростить его поддержку и обновление.

Использование атрибутов `DataType`

- Откройте класс `Movie`.

В пространстве имен `System.ComponentModel.DataAnnotations` в дополнение к набору встроенных атрибутов проверки предоставляются атрибуты форматирования.

Атрибут `[DataType]` применяется к свойствам `ReleaseDate` и `Price`.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

Атрибуты `[DataType]` предоставляют:

- Указания для обработчика представлений для форматирования данных.
- Атрибуты, например `<a>` для URL-адресов и `` для электронной почты.

Используйте атрибут `[RegularExpression]` для проверки формата данных.

Атрибут `[DataType]` позволяет указать тип данных с более точным определением по сравнению со встроенным типом базы данных. Атрибуты `[DataType]` не предназначены для проверки. В том же приложении отображается только дата (без времени).

Перечисление `DataType` предоставляет множество типов данных, таких как `Date`, `Time`, `PhoneNumber`, `Currency`, `EmailAddress` и т. д.

Атрибуты `[DataType]`:

- Обеспечивают автоматическое предоставление функций для определенных типов в приложении. Например, можно создавать ссылку `mailto:` для `DataType.EmailAddress`.
- Могут предоставить селектор даты `DataType.Date` в браузерах, поддерживающих HTML5.
- Создают атрибуты HTML 5 `data-`, которые используются браузерами с поддержкой HTML 5.
- **Не предназначены** для проверки.

`DataType.Date` не задает формат отображаемой даты. По умолчанию поле данных отображается с использованием форматов, установленных в параметрах `CultureInfo` сервера.

Требуются заметки к данным `[Column(TypeName = "decimal(18, 2)"]`, чтобы Entity Framework Core корректно сопоставила `Price` с валютой в базе данных.

С помощью атрибута `[DisplayFormat]` можно явно указать формат даты:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

Параметр `ApplyFormatInEditMode` указывает, что при отображении значения для редактирования будет применяться форматирование. Это поведение может быть нежелательным для некоторых полей. Например, в полях валюты в пользовательском интерфейсе редактирования использовать символ денежной единицы, как правило, не требуется.

Атрибут `[DisplayFormat]` может использоваться отдельно, однако чаще всего его рекомендуется применять вместе с атрибутом `[DataType]`. Атрибут `[DataType]` передает семантику данных (в отличие от способа их вывода на экран). Атрибут `[DataType]` дает следующие преимущества, недоступные в `[DisplayFormat]`:

- Поддержка функций HTML5 в браузере, например отображение элемента управления календарем, соответствующего языковому стандарту символа валюты, ссылок электронной почты и т. д.
- По умолчанию формат отображения данных в браузере определяется в соответствии с установленным языковым стандартом.
- Атрибут `[DataType]` обеспечивает поддержку платформы ASP.NET Core для выбора соответствующего шаблона поля, применяемого при отображении

данных. При отдельном использовании атрибут `DisplayFormat` базируется на строковом шаблоне.

Примечание. Проверка jQuery не работает с атрибутом `[Range]` и `DateTime`. Например, следующий код всегда приводит к возникновению ошибки проверки на стороне клиента, даже если дата попадает в указанный диапазон:

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

Лучшей методикой будет не компилировать модели с фиксированными датами, поэтому использовать атрибуты `[Range]` и `DateTime` следует крайне осторожно. Используйте конфигурацию для диапазонов дат и других значений, подверженных частым изменениям, а не указывайте их в коде.

В следующем коде демонстрируется объединение атрибутов в одной строке:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TableName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

Применение миграции

`DataAnnotation`, примененные к классу, изменяют схему. Например, `DataAnnotation`, примененные к полю `Title`, выполняют следующее:

```
[StringLength(60, MinimumLength = 3)]
[Required]
public string Title { get; set; }
```

- ограничивают число символов до 60;
- не допускают значение `null`.

Сейчас таблица `Movie` имеет следующую схему:

```
CREATE TABLE [dbo].[Movie] (
    [ID] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (MAX) NULL,
    [ReleaseDate] DATETIME2 (7) NOT NULL,
    [Genre] NVARCHAR (MAX) NULL,
    [Price] DECIMAL (18, 2) NOT NULL,
    [Rating] NVARCHAR (MAX) NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

Предыдущие изменения схемы не приводят к созданию исключения EF. Но следует создать миграцию, чтобы схема соответствовала модели.

- Перейдите в окно консоли диспетчера пакетов и введите следующие команды:

```
Add-Migration New_DataAnnotations
```

```
Update-Database
```

Update-Database **выполняет методы Up класса** New_DataAnnotations.

- Проверьте метод Up.

```
public partial class New_DataAnnotations : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "Title",
            table: "Movie",
            maxLength: 60,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);

        migrationBuilder.AlterColumn<string>(
            name: "Rating",
            table: "Movie",
            maxLength: 5,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);

        migrationBuilder.AlterColumn<string>(
            name: "Genre",
            table: "Movie",
            maxLength: 30,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);
    }
}
```

Обновленная таблица Movie имеет следующую схему:

```
CREATE TABLE [dbo].[Movie] (
    [ID] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (60) NOT NULL,
    [ReleaseDate] DATETIME2 (7) NOT NULL,
    [Genre] NVARCHAR (30) NOT NULL,
    [Price] DECIMAL (18, 2) NOT NULL,
    [Rating] NVARCHAR (5) NOT NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```