

**ASP.NET Core**

Книга практических занятий

Тагиров Радмир Дамирович

Оглавление

[Блок 1 2](#_Toc119754466)

[УПРАЖНЕНИЕ №1: Установка среды разработки 2](#_Toc119754467)

[УПРАЖНЕНИЕ №2: Первая программа 3](#_Toc119754468)

[САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ 6](#_Toc119754469)

[Блок 2 7](#_Toc119754470)

[УПРАЖНЕНИЕ №1: Отображение страницы приветствий 7](#_Toc119754471)

[САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ 7](#_Toc119754472)

[Блок 3 9](#_Toc119754473)

[УПРАЖНЕНИЕ №1: Маршрутизация конечных точек 9](#_Toc119754474)

[САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ 10](#_Toc119754475)

[Блок 4 11](#_Toc119754476)

[УПРАЖНЕНИЕ №1: Маршрутизация конечных точек 11](#_Toc119754477)

# Блок 1

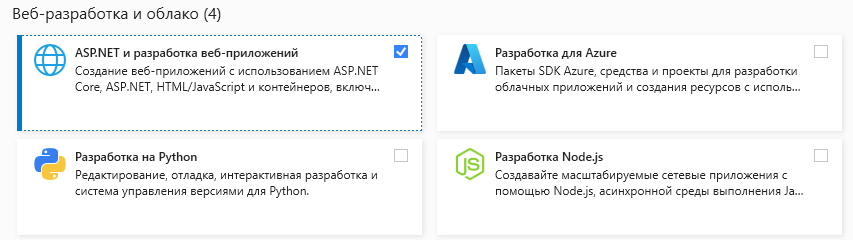
## УПРАЖНЕНИЕ №1: Рисунок 10Установка среды разработки

Задание для участников

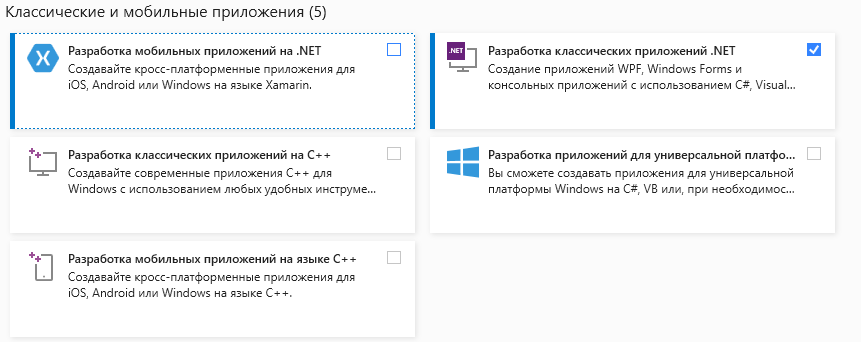
У Microsoft есть два редактора для работы с языком С# и .NET Core: Visual Studio и Visual Studio Code (сокращенно VS Code). Есть несколько версий Visual Studio, в том числе и бесплатная версия для сообщества, но в ней все равно есть ряд лицензионных ограничений, в частности запрещена коммерческая разработка. В отличии от Visual Studio, VS Code абсолютно бесплатна даже для коммерческой разработки. Для целей обучения можно использовать бесплатную версию Visual Studio, поэтому мы воспользуемся ей. Она немного проще в использовании особенно для новичков. Скачать Visual Studio можно по этому адресу:  [www.visualstudio.com/vs/community](http://www.visualstudio.com/vs/community).

Запустите скаченный файл и на вкладке «Рабочие нагрузки» (Workloads), выберете:

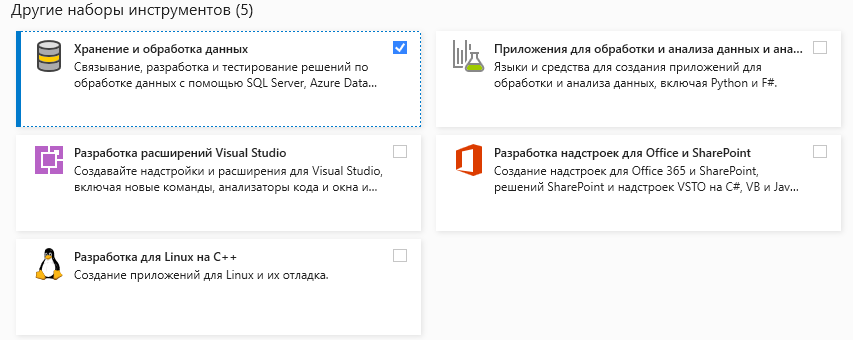
* в разделе «Веб-разработка и облако» (Web & Cloud) пункт «ASP.NET и разработка веб-приложений» (ASP.NET and web-development).



* в разделе «Классические и мобильные приложения» (Desktop & Mobile) пункт «Разработка классических приложений .NET» (.NET desktop development).



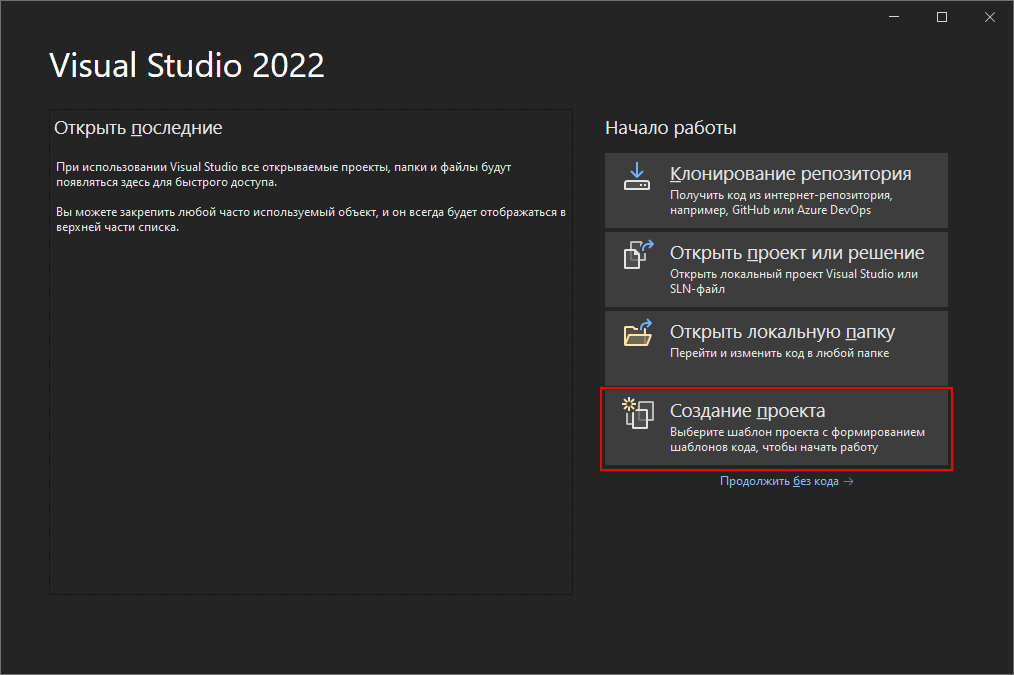
* в разделе «Другие наборы инструментов» (Other Toolsets) пункт «Хранение и обработка данных» (Data storage and processing).



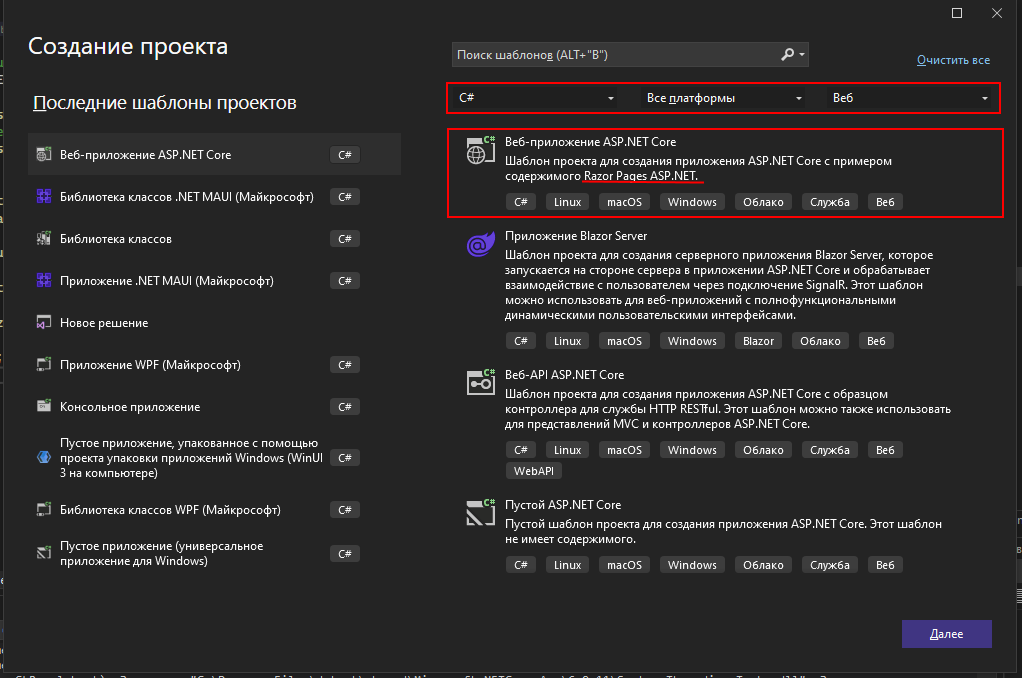
## УПРАЖНЕНИЕ №2: Рисунок 10Первая программа

Задание для участников

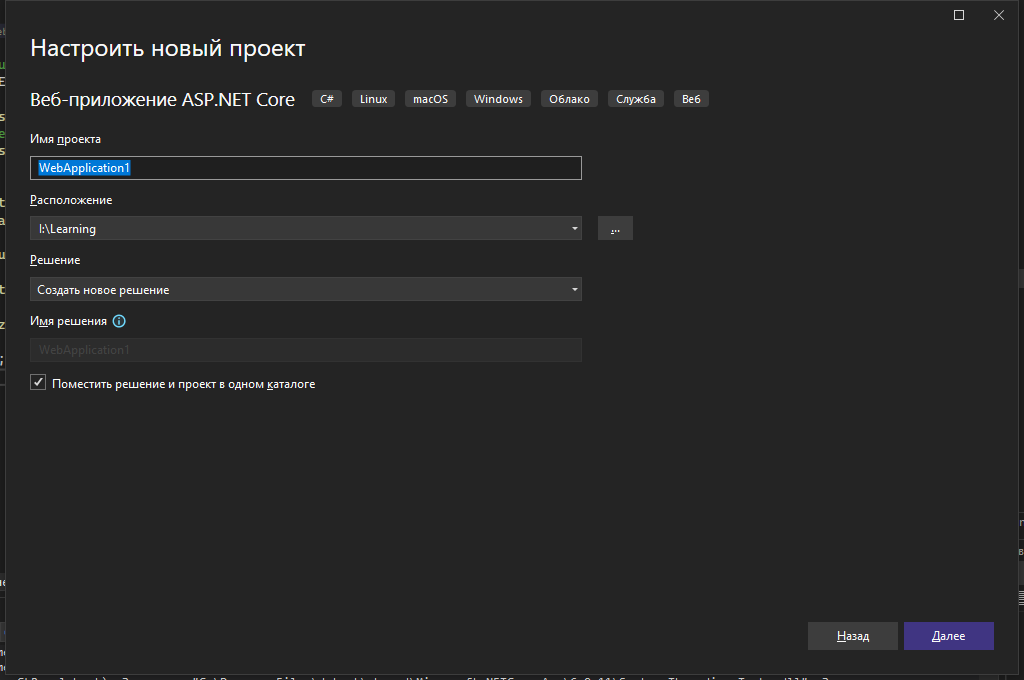
Запустите Visual Studio и в появившемся окне выберите пункт «Создание проекта».



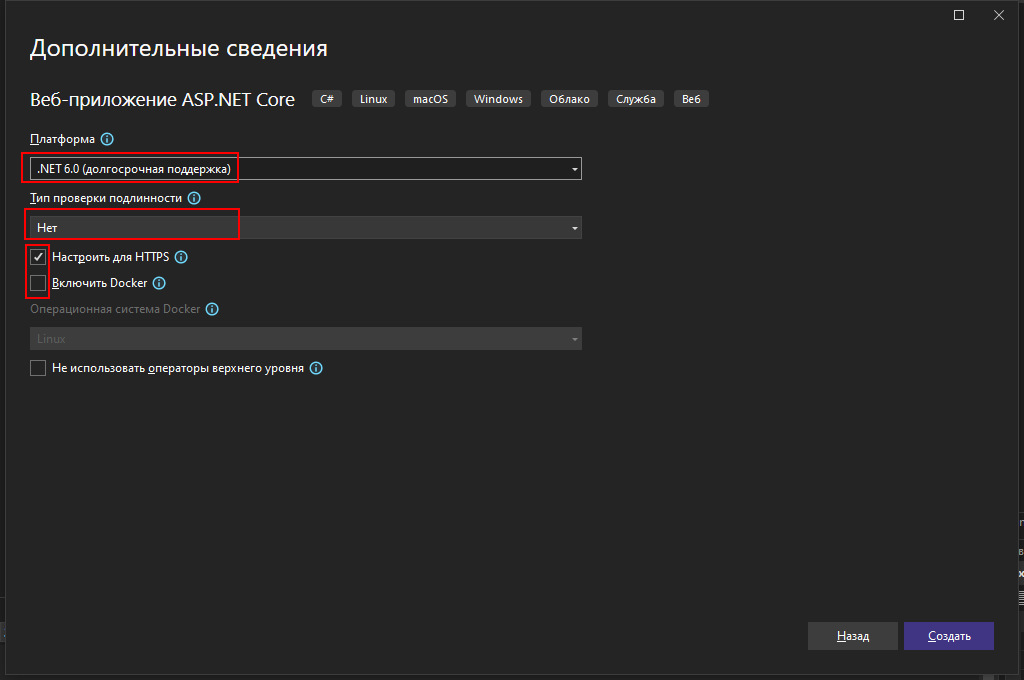
Затем в выпадающих списках выберите язык C#, Все платформы и тип проекта Вэб. Выберите проект Веб-приложение ASP.NET Core и нажмите кнопку «Далее».



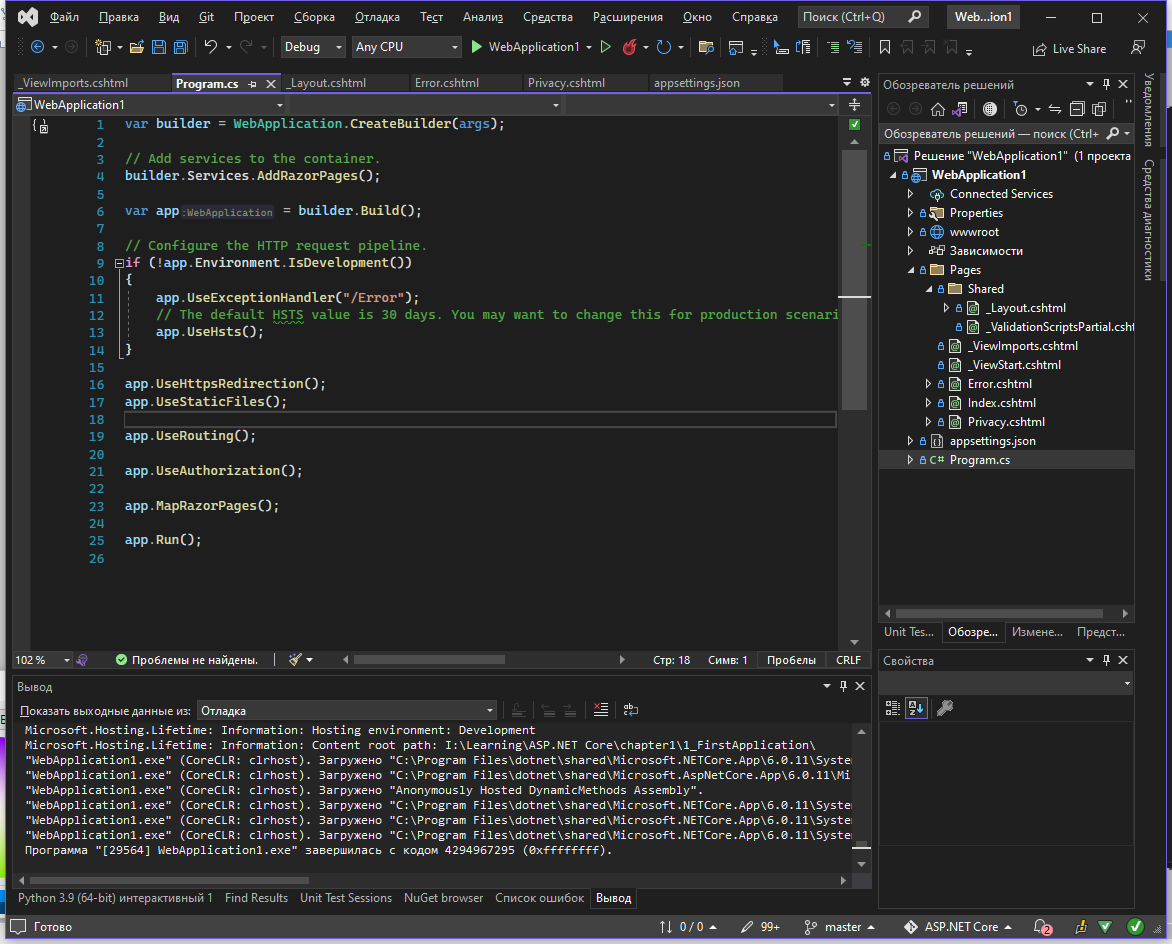
Вводим имя проекта WebApplication и нажимаем кнопку «Далее».



Задайте дополнительные сведения как указано на рисунке и нажмите кнопку «Создать».



Откроется окно, содержащее созданную программу.



## САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ

1. Запустите приложение нажав кнопку (F5 или ). В открывшемся браузере попробуйте попереключать страницы. Завершите выполнения программы в среде разработки (Shift + F5 или )
2. Измените заголовок страницы Index.cshtml c “Home Page”на “ASP.NET Core”
3. В файле Program.cs Отключите компонент работы со статическими файлами и посмотрите как изменится страница в браузере.

# Блок 2

## УПРАЖНЕНИЕ №1: Рисунок 10Отображение страницы приветствий

Задание для участников

В этом упражнении вы узнаете, как создать один из простейших конвейеров промежуточного программного обеспечения, состоящий только из компонента WelcomePageMiddleware. WelcomePageMiddleware предназначен для быстрого предоставления образца страницы при первой разработке приложения. Вы вряд ли стали бы использовать его в промышленном окружении, так как здесь нельзя настроить вывод, но это единственный автономный компонент, который можно использовать, чтобы гарантировать правильную работу своего приложения.

Запрос передается на веб-сервер ASP.NET Core, который создает представление запроса и передает его в конвейер. Поскольку это первый (и единственный!) компонент в конвейере WelcomePageMiddleware получает запрос и должен решить, как его обработать. Промежуточное ПО генерирует ответ в виде HTML-кода, независимо от того, какой запрос получает. Этот ответ возвращается на веб-сервер ASP.NET Core, который пересылает его пользователю для отображения в его браузере.

Добавляя компонент к объекту IApplicationBuilder. Чтобы создать свой первый конвейер, состоящий из одного компонента, вам понадобится всего лишь один вызов метода.

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseWelcomePage();

app.Run();

Как видите у приложения нет конфигурации и сервисов, поэтому код крайне простой.

Единственное что мы делаем, вызываем метод UseWelcomePage.

Вы создаете конвейер промежуточного программного обеспечения в ASP.NET Core, вызывая методы класса IApplicationBuilder, но этот интерфейс не определяет такие методы, как UseWelcomePage. Это методы *расширения*.

Использование методов расширения позволяет эффективно добавлять функциональные возможности к классу IApplicationBuilder, сохраняя при этом их реализации изолированными от него.  Под капотом методы обычно вызывают еще один метод *расширения*, чтобы добавить промежуточное программного обеспечения в конвейер. Например, за кулисами метод UseWelcomePage добавляет WelcomePageMiddleware в конвейер, используя

UseMiddleware<WelcomePageMiddleware>();

Это соглашение о создании метода расширения для каждого компонента промежуточного программного обеспечения. Имя метода начинается со слова Use. Это делается для того, чтобы при добавлении промежуточного программного обеспечения в ваше приложение, его проще было обнаружить.

Вызов метода UseWelcomePage добавляет WelcomePageMiddleware в качестве следующего компонента в конвейер. Хотя здесь используется только один компонент, важно помнить, что порядок, в котором вы вызываете IApplicationBuilder определяет порядок, в котором компонент будет запускаться в конвейере.

## САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ

1. Добавьте в проект обработчик статических файлов, используя метод расширения UseStaticFiles или компонент StaticFileMiddleware. Добавьте поддержку сервисов Razor Pages вызвав метод AddRazorPages() в настройке конфигурации сервисов, добавьте промежуточное ПО маршрутизации и подключите поддержку страниц Razor Pages в компоненте конечных точек MapRazorPages().Модифицируйте конвейер так, чтобы при наличии отображалась страница Razor в противном случае отображалась страница приветствия.
2. Добавьте в проект собственную страницу для отображения ошибки 404 и модифицируйте код проекта так, чтобы при возникновении этой ошибки отображалась эта страница.
3. Добавьте в проект страницу, Products и внесите изменения в модель страницы Index так, чтобы при вызове метода OnGet происходило перенаправление на страницу Products

# Блок 3

## УПРАЖНЕНИЕ №1: Рисунок 10Маршрутизация конечных точек

Задание для участников

В этом упражнении вы узнаете, как добавить маршрутизацию в страницы Razor.

Откройте решение ECommerce.sln

Наиболее частой причиной настройки шаблона маршрута страницы Razor является добавление *параметра маршрута*. Например, у вас есть одна страница Razor для отображения товаров на сайте для онлайн торговли по пути Pages/Products/Index.cshtml.

Если вы запустите приложение, то увидите, что на странице отображаются список товаров, с ссылками, но эти ссылки ведут в никуда. Мы же хотим, чтобы эти ссылки вели на страницы с детальной информацией о товаре.

Для начала добавим шаблон станицы с детальной информацией, для этого в папку Pages/Products добавим новую страницу ProductDetail.cshtml и отредактируем шаблон страницы добавив к директиве @page параметр маршрута id

@page "{id}"

@model ECommerce.Pages.Products.ProductDetailModel

@{

}

Теперь, если запустить приложение, то вы увидите, что ссылки ведут в страницы с детальной информацией о продукте, но пока эти страницы пустые.

Добавим в модель привязку к параметру запроса, для этого добавим свойство Id и задекорируем его атрибутом [BindProperty(SupportsGet = true)]

public class ProductDetailModel : PageModel

{

[BindProperty(SupportsGet = true)]

public string Id { get; set; }

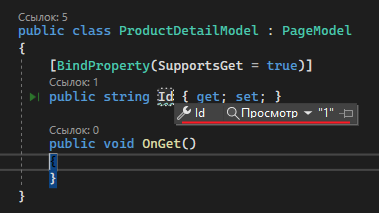
public void OnGet()

{

}

}

Если вы сейчас поставите точку останова в методе OnGet, и попытаетесь перейти по ссылке то увидите, что свойство Id содержит значение параметра {id} запроса:



Смотреть на пустые страницы довольно скучно, поэтому добавим отображение свойства Id на странице.

@page "{id}"

@model ECommerce.Pages.Products.ProductDetailModel

@{

}

@Model.Id

Запустите приложение и убедитесь, что параметр отображается.

## САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ

1. Сейчас страница с детальной информацией отображается по маршруту Products/ProductDetail/{id} модифицируйте настройки маршрутизации так, чтобы маршрут соответствовал RESTful т.е. имел вид Products/{id}
2. Добавьте к маршруту ограничения для параметра id – он должен быть целочисленным и больше нуля.
3. Добавьте к маршруту страницы Products/Index.cshtml необязательный универсальный параметр {query} и отобразите его значение на странице.
4. Добавьте страницу Products/AddProduct.cshtml которая будет принимать параметры name и price и выполните привязку этих данных к классу:

class InputModel

{

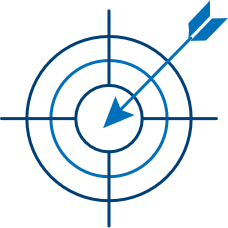
public string Name { get; set; }

public string Price { get; set; }

}

# Блок 4

## УПРАЖНЕНИЕ №1: Валидация модели на стороне сервера

 Задание для участников

В этом упражнении вы узнаете, как добавить валидацию в страницы Razor.

Откройте решение ECommerce.sln

И запустите его.

В браузере перейдите на страницу добавления нового продукта:

<https://localhost:7163/Products/add>

и попытайтесь ввести некорректные данные:

-пустое имя

- отрицательную цену

И нажмите кнопку Submit, страница никак не реагирует на некорректные данные.

Теперь, остановим выполнение проекта и откроем модель страницы AddProduct:

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ECommerce.Pages.Products

{

public class AddProductModel : PageModel

{

[BindProperty(SupportsGet = true)]

public InputModel Input { get; set; }

public void OnGet()

{

}

}

public class InputModel

{

public string Name { get; set; }

public string Price { get; set; }

}

}

Задекорируем свойства класса InputModel атрибутами DataAnnotation

public class InputModel

{

[Required]

[MinLength(4)]

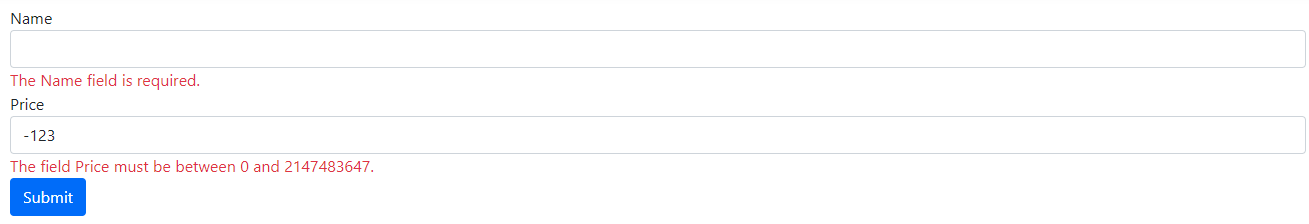
public string Name { get; set; }

[Range(0, int.MaxValue)]

public string Price { get; set; }

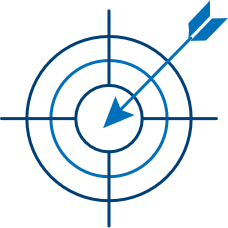
}

Снова запустите приложение и снова перейдите на страницу добавления нового продукта и снова попробуем ввести некорректные данные.



Теперь мы видим, что валидация модели на стороне сервера выполняется.

## УПРАЖНЕНИЕ №2: Динамическая генерация страницы

 Задание для участников

Вы, наверное, уже заметили, что на странице Products/Index.cshtml список товаров задан «жестко». Это очень неудобно, т.к. при любом изменении номенклатуры товаров придется редактировать шаблон страницы. Давайте добавить в наш проект динамическую генерацию содержимого страницы на основе данных, предоставляемых бизнес-моделью. Все необходимые сервисы для взаимодействия с бизнес логикой уже имеются в проекте и зарегистрированы в контейнерах DI. Поэтому нам достаточно «запросить» сервис получения списка товаров в конструкторе модели страницы Products/Index.

Открываем модели страницы и создаем конструктор класса IndexModel с параметром IProductsService productsService:

public IndexModel(IProductsService productsService)

{

}

Теперь при создании модели ASP.NET Core будет автоматически подставлять в конструктор экземпляр класса реализующий интерфейс IProductsService

Теперь добавим приватное поле productsService для того, чтобы иметь доступ к сервису из методов обработки запросов.

public class IndexModel : PageModel

{

private readonly IProductsService productsService;

public IndexModel(IProductsService productsService)

{

this.productsService = productsService;

}

}

Теперь в методе OnGet мы можем заполнить модель данными предоставляемых сервисом:

public class IndexModel : PageModel

{

private readonly IProductsService productsService;

public IndexModel(IProductsService productsService)

{

this.productsService = productsService;

}

public IEnumerable<Product> Products { get; set; }

public void OnGet()

{

Products = this.productsService.GetProducts();

}

}

И в шаблон страницы добавим код генерации страницы на основе этих данных.

@page

@model ECommerce.Pages.Products.IndexModel

@{

}

<ul>

@foreach (var product in Model.Products)

{

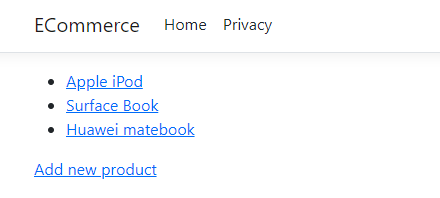
<li><**a** **asp-area**="" **asp-page**="/Products/ProductDetail" **asp-route-id**="@product.id">@product.Name</**a**></li>

}

</ul>

<**a** **asp-page**="/Products/AddProduct">Add new product</**a**>

Запустим приложение и убедимся, что данные страницы создаются динамически:



## САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ

1. Внесите изменения в модель и шаблон страницы Products/ProductDetail, так чтобы страницы также генерировалась динамически и на ней отображалось наименование и стоимость выбранного продукта. Для получения продукта по идентификатору воспользуйтесь методом IProductServices.GetProducts(int productId)
2. Добавьте страницу редактирования данных продукта Products/EditProduct, создайте частичное преставление ProductEditor и добавьте его использование в странице Products/EditProduct и Products/AddProduct

# Блок 5

Для выполнения заданий откройте проект ECommerce.sln из папки chapter5

## САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ

1. Для страницы Products/AddProduct добавьте код добавления нового товара (для этого используйте метод IProductServices.AddProduct) Перед операцией добавления не забудьте выполнить валидацию модели. При успешном завершении пользователь должен быть перенаправлен на страницу Products/Index
2. Активируйте валидацию на стороне клиента для страниц Products/AddProduct Products/EditProduct
3. На странице Products/ProductDetail с помощью тег-хелперов добавьте ссылку на страницу редактирования товара.

# Блок 6

## САМОСТОЯТЕЛЬНЫЕ ЗАДАНИЯ

1. Добавьте в решение новый проект ECommerceApi по шаблону ASP.NET Core WebAPI
2. Добавьте контроллер OrdersContoller и метод AddOrder
3. Настройте маршрутизацию запроса с глаголом POST для метода действия AddOrder по адресу api/v1/orders
4. Добавьте в метод AddOrder чтения данных из тела запроса соответствующих следующему контракту:

class Order

{

public int CustomerId { get; set; }

public int[] Prorducts { get; set; }

public string DeliveryAddress { get; set; }

}

1. Добавьте интерфейс:

interface IOrdersService

{

void PlaseOrder(Order order);

}

и реализуйте сервис «заглушку» для этого интерфейса

1. Добавьте зависимость IOrderService в OrdersContoller
2. Зарегистрируйте scoped зависимость для этого интерфейса и «заглушки»

# Блок 7

1. Добавьте к проекту ECommerce поддержку EF Core
2. Добавьте контекст доступа к данным (AppDataContext) и добавьте в него DataSet предоставляющий данный о товарах (Product), в качестве БД используйте SQLite.
3. Создайте начальную миграцию для создания схемы данных.
4. Изменить сервис ProductsService для работы с БД
5. Реализовать логику добавления нового товара на странице AddProduct

# Блок 8

1. Добавить фильтр валидации модели для страниц AddProduct и EditProduct
2. Добавить фильтр проверки наличия товара для страницы EditProduct

# Блок 9

1. К проекту добавить аутентификацию с иcпользованием Identity
2. Создать миграцию добавляющую модель данных Identity.
3. Создайте собственную страницу регистрации пользователя.
4. Разрешить добавлять и редактировать товары только зарегистрированным пользователям
5. Модифицировать страницы Products/Index и Products/ProductDetail так чтобы ссылки на для добавления и редактирования товаров были доступны только зарегистрированным пользователям