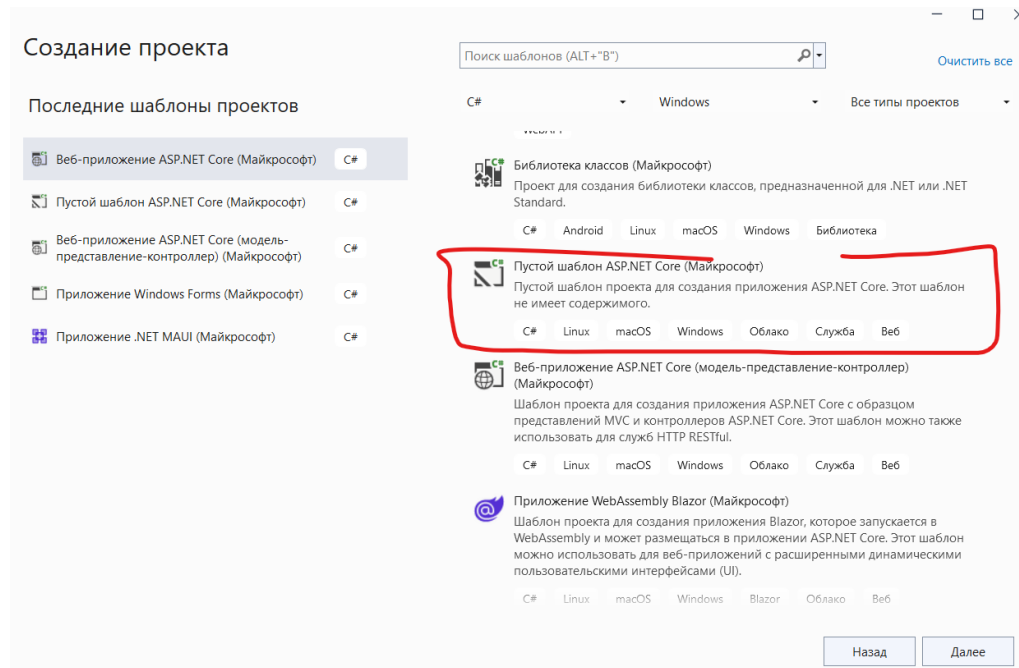


## Lab 1. Введение в создание приложений ASP.NET Core на основе Razor Pages

### 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**.



Шаблон ASP.NET Core создал приложение готовое к запуску.

- Откройте файл `Program.cs` и изучите его содержимое:

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

app.MapGet("/", () => "Hello World!");

app.Run();
```

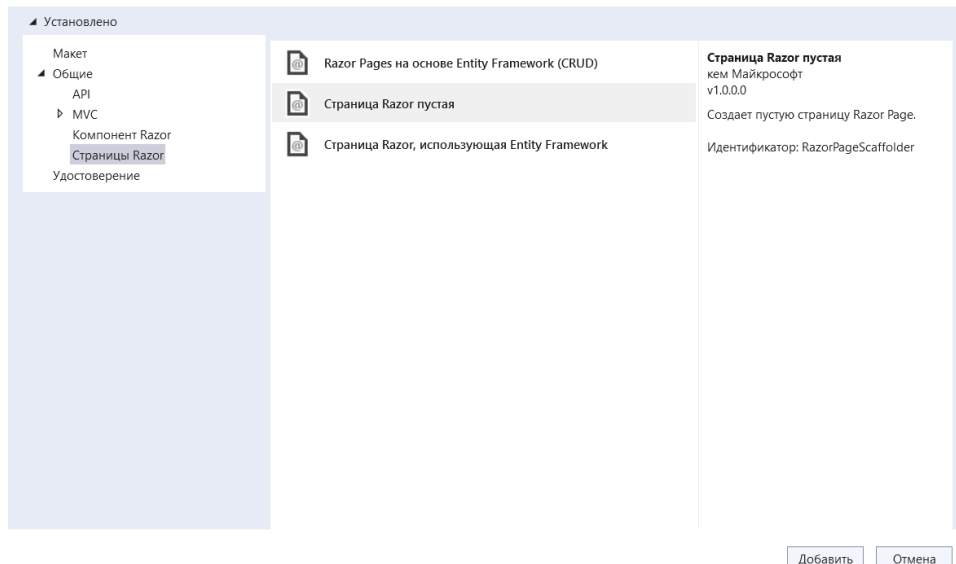
Данный код включает одну конечную точку с использованием метода [MapGet](#): при отправке HTTP-запроса GET на корневой URL-адрес / выполняется делегат запроса и в ответ HTTP записывается Hello World!.

Если метод запроса не является GET или если корневой URL-адрес не /, сопоставление маршрута не выполняется и возвращается сообщение об ошибке HTTP 404.

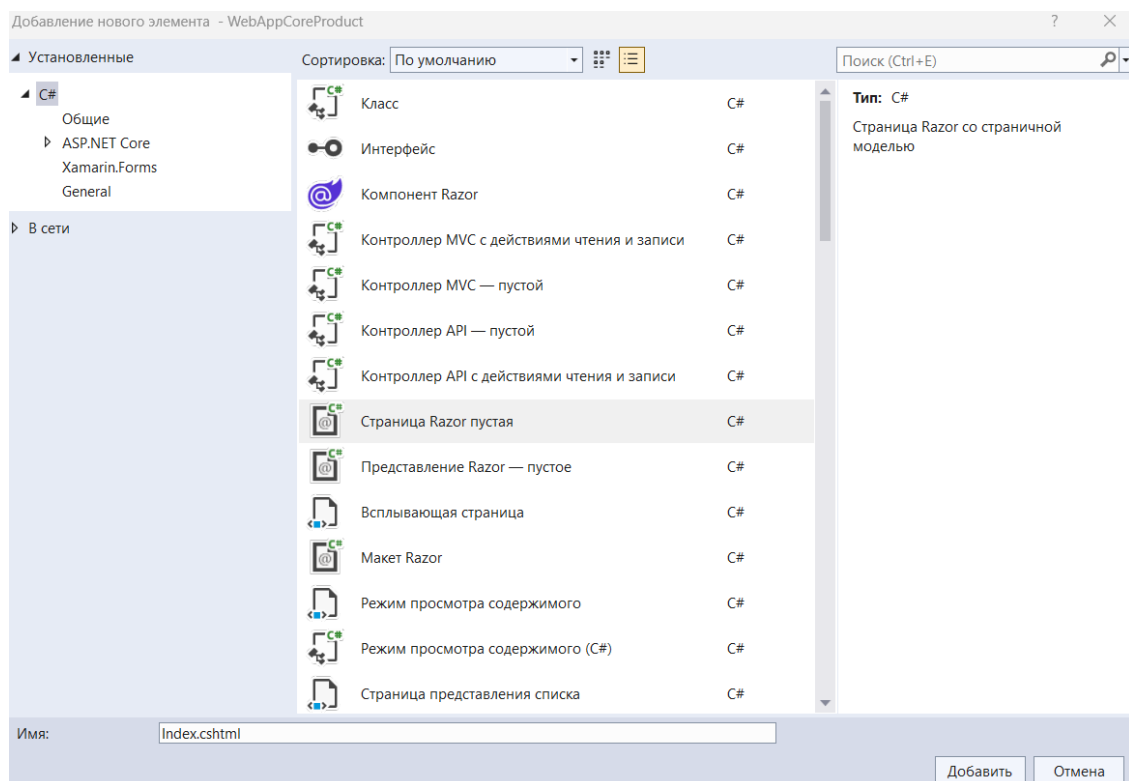
- Постройте и запустите приложение. Должна отобразиться страница с приветственной надписью "Hello World!"

- 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 Page Razor** (Добавить страницу Razor).
- Выберите шаблон для пустой **Razor Page** (Страница Razor пустая):

Добавить новый шаблонный элемент



- 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:

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

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

- Откройте файл Program.cs и замените его содержимое на следующий код:

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

// Add services to the container.
builder.Services.AddRazorPages();
var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change
    this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

В этом коде:

- `AddRazorPages()` добавляет в приложение службы для Razor Pages.
- `MapRazorPages()` добавляет в `IEndpointRouteBuilder` конечные точки для Razor Pages.
- Также добавлены несколько компонентов ПО промежуточного слоя в конвейер запросов.

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` 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>
}

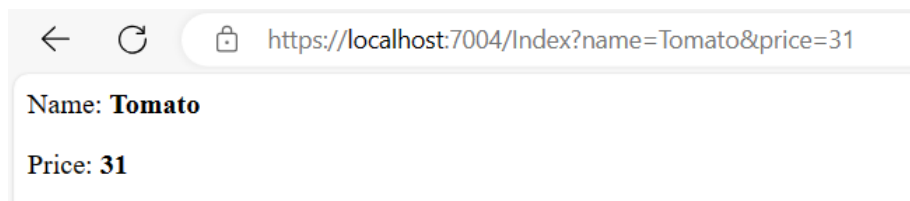
```

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

- Запустите приложение. Сейчас странице не переданы значения (т.е. они являются некорректными значениями) и поэтому 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`

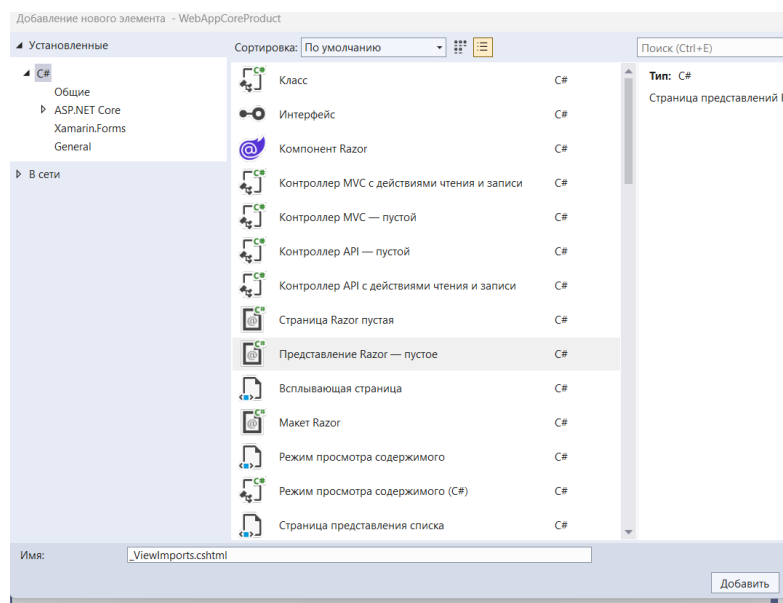
Приложение получит данные и выведет на страницу результат согласно указанной разметки страницы `Index.cshtml`. например:



### 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 пустое) 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.MessageRezult</h3>
```

- В классе IndexModel объявите новое поле для результирующего сообщения:

```
public string? MessageRezult { get; private set; }
```

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

```
var result = price * (decimal?)0.18;
MessageRezult = $"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))
    {
        MessageRezult = "Переданы некорректные данные. Повторите ввод";
        return;
    }

    var result = price * (decimal?)0.18;
    MessageRezult = $"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 ()
{
    MessageRezult = "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 />
    <input type="submit" value="Получить скидку" /><br />
</form>

<h3>@Model.MessageRezult</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

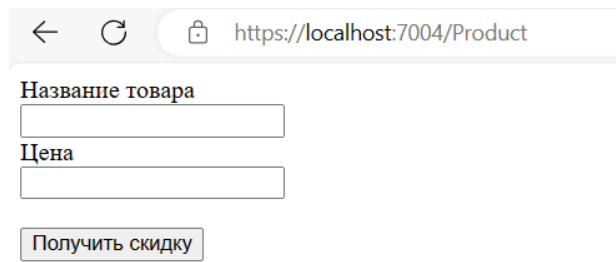
В этом упражнении вы разделите страницы интерфейса и примените мастер-страницу. Как правило, веб-приложения содержат основную (стартовую страницу) и несколько страниц, связанных с различными сущностями. В текущем проекте роль основной страницы играет `Index.cshtml` и поэтому для определения сущности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





### Для товара можно определить скидку

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>© @DateTime.Now.Year.ToString() - 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).

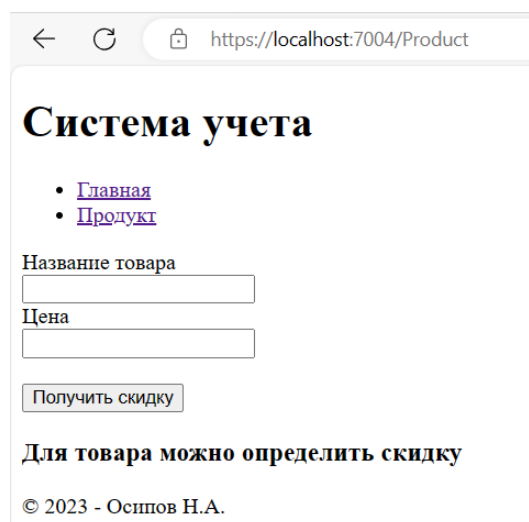
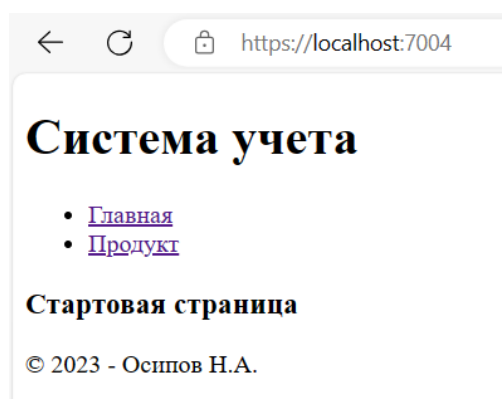
- Для реализации глобальной установки добавьте в папку **Pages** новое представление `_ViewStart.cshtml`.
- Укажите в новой странице связь с мастер-страницей:

```
@{
```

```
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;
```

```

        MessageResult = $"Для товара {name} с ценой {price} и скидкой {discont}
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 /><br />
    <input type="submit" asp-page-handler="" value="Получить скидку" /><br />
    <label for="discont">Введите свою скидку(%)</label><br />
    <input type="number" name="discont" /><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.