Представления

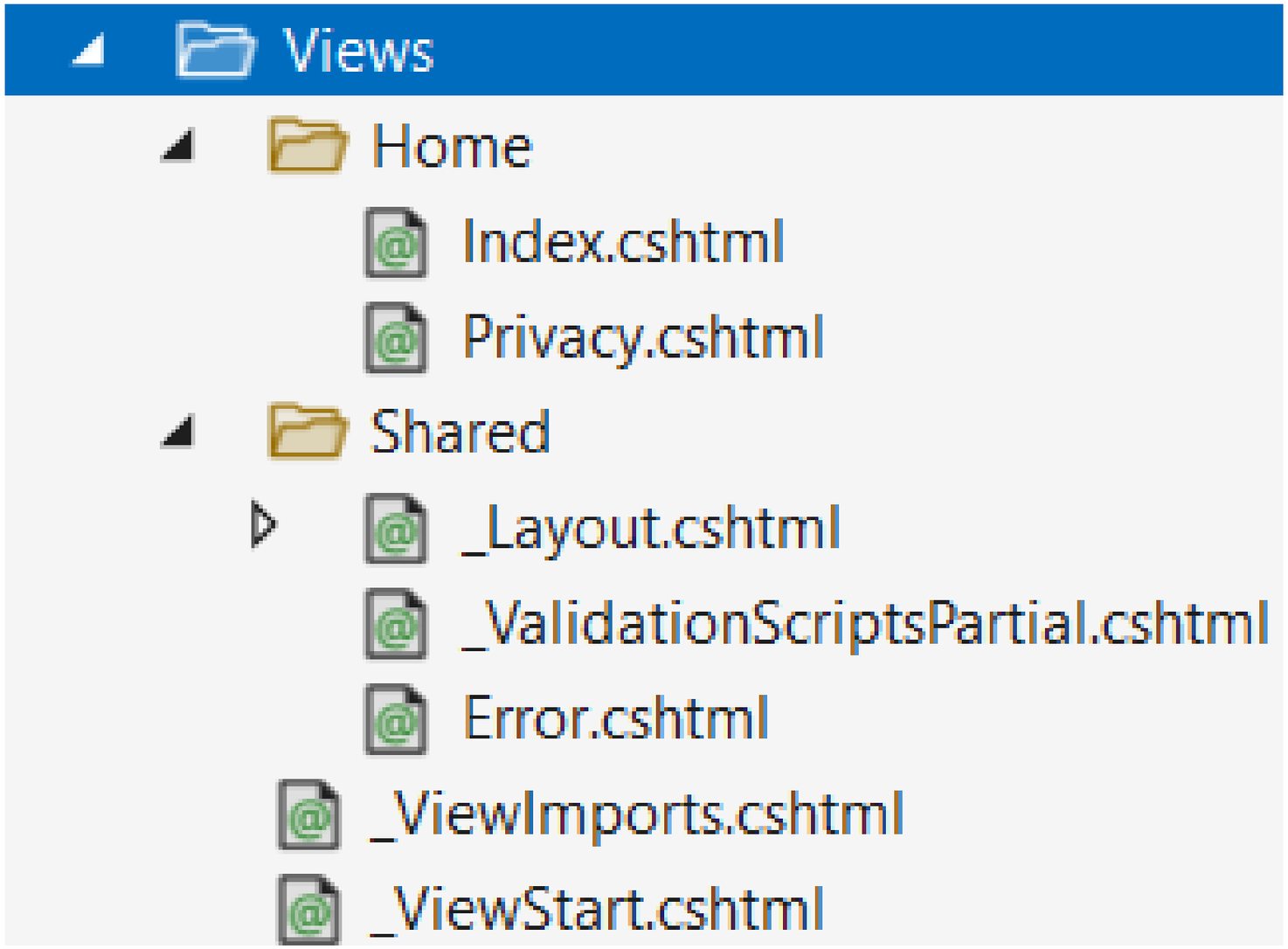
Введение в представления

В ASP.NET MVC Core **представления** - это файлы с расширением cshtml, которые содержат код пользовательского интерфейса в основном на языке html, а также конструкции Razor - специального движка представлений, который позволяет переходить от кода html к коду на языке C#.

Введение в представления

Для хранения представлений в проекте ASP.NET Core предназначена папка **Views**. Например, если мы возьмем проект по типу **ASP.NET Core Web App (Model-View-Controller)**, то мы увидим, что он содержит ряд представлений.

Введение в представления



Введение в представления

Подобный проект для хранения представлений в папке Views определяет некоторую структуру:

- Home
- Shared
- _ViewImports.cshtml и _ViewStart.cshtml

Home

Как правило, для каждого контроллера в проекте создается подкаталог в папке Views, который называется по имени контроллера и который хранит представления, используемые методами данного контроллера. Так, по умолчанию имеется контроллер HomeController и для него в папке Views есть подкаталог **Home** с представлениями для методов контроллера HomeController - в данном случае это файлы **Index.cshtml** и **Privacy.cshtml.**

Shared

Папка **Shared** хранит общие представления для всех контроллеров. По умолчанию это файлы **_Layout.cshtml** (используется в качестве мастер-страницы), **Error.cshtml** (используются для отображения ошибок) и **_ValidationScripsPartial.cshtml** (частичное представление, которое подключает скрипты валидации формы).

Определение контроллера и вызов представления. ViewResult

Для начала определим в проекте папку **Controllers**, в которую добавим новый контроллер - **HomeController**

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Определение контроллера и вызов представления. ViewResult

За работу с представлениями отвечает объект **ViewResult**. Он производит рендеринг представления в веб-страницу и возвращает ее в виде ответа клиенту. Чтобы возвратить объект **ViewResult**, в методе контроллера вызывается метод **View.**

Определение контроллера и вызов представления. ViewResult

Вызов метода View возвращает объект ViewResult. Затем уже ViewResult производит рендеринг определенного представления в ответ. По умолчанию контроллер производит поиск представления в проекте по следующим путям:

/Views/Имя_контроллера/Имя_представления.cshtml

/Views/Shared/Имя_представления.cshtml

Определение контроллера и вызов представления. ViewResult

Согласно настройкам по умолчанию, если название представления не указано явным образом, то в качестве представления будет использоваться то, имя которого совпадает с именем действия контроллера. Например, вышеопределенное действие Index по умолчанию будет производить поиск представления **Index.cshtml** в папке **/Views/Home/.**

Создание представлений

Сначала требуется создать в проекте новую папку **Views**, а в ней определим новый каталог **Home** - для представлений контроллера HomeController.

Создание представлений

Далее добавим в каталог **Views/Home** новый элемент по типу **Razor View - Empty,** который назовем **index.cshtml**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello</title>
    <meta charset="utf-8" />
</head>
<body>
    <h2>Hello, world!</h2>
</body>
</html>
```

Создание представлений

Данное представление напоминает обычную страницу html. Здесь могут быть определены все стандартные элементы разметки html, здесь могут подключаться стили, скрипты. Но полноценной html-страницей представление все равно не является, потому что во время выполнения эти представления компилируются в сборки и уже затем используются для генерации html-страниц, которые видит пользователь в своем браузере.

Подключение функционала представлений

Для подключения в приложение функционала контроллеров с представлениями применяется вызов **AddControllersWithViews()** (также можно применять методы **AddMvcCore()** и **AddMvc()**)

Подключение функционала представлений

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

// добавляем поддержку контроллеров с представлениями
builder.Services.AddControllersWithViews();
var app = builder.Build();

// устанавливаем сопоставление маршрутов с контроллерами
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Подключение функционала представлений

И после запуска приложения и обращении к методу Index контроллера Home браузер отобразит нам веб-страницу, которая будет сгенерирована на основе представления Index.cshtml.

ViewResult и пути к представлениям

Чтобы возвратить объект ViewResult, который производит рендеринг представления в веб-страницу и возвращает ее в виде ответа клиенту, используется метод **View().** Например, в пример выше применялась версия этого метода, которая не применяла параметров:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

ViewResult и пути к представлениям

Но вообще метод View() имеет четыре перегруженных версии:

- View(): для генерации ответа используется представление, которое по имени совпадает с вызывающим методом
- View(string? viewName): в метод передается имя представления, что позволяет переопределить используемое по умолчанию представление
- View(object? model): передает в представление данные в виде объекта model
- View(string? viewName, object? model): переопределяет имя представления и передает в него данные в виде объекта model

ViewResult и пути к представлениям

Вторая версия метода позволяет переопределить используемое представление. Если представление находится в той же папке, которая предназначена для данного контроллера, то в метод View() достаточно передать название представления без расширения:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View("About");
    }
}
```

ViewResult и пути к представлениям

В этом случае метод **Index** будет использовать представление по пути *Views/Home/About.cshtml*. Если же представление находится в другой папке, то нам надо передать полный путь к представлению:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View("~/Views/Some/About.cshtml");
    }
}
```

В данном случае предполагается, что представление располагается по пути *Views/Some/About.cshtml*

Движок представлений Razor

Движок представлений Razor

Представления в ASP.NET Core MVC может содержать не только стандартный код html, но и также вставки кода на языке C#. Для обработки кода, который содержит как элементы html, так и конструкции языка C#, применяется движок представлений.

Движок представлений Razor

По умолчанию в ASP.NET Core MVC применяется один движок представлений - **Razor**. Хотя при желании мы можем также использовать какие-то другие сторонние движки или создать свой движок представлений самостоятельно. Цель движка представлений Razor - определить переход от разметки html к коду C#.

Движок представлений Razor

Синтаксис Razor довольно прост - все его конструкции предваряются символом @, после которого происходит переход к коду C#. Например, определим следующее представление:

```
<!DOCTYPE html>
<html>
<head>
    <title>Date</title>
    <meta charset="utf-8" />
</head>
<body>
    <h2>Time: @DateTime.Now.ToShortTimeString()</h2>
</body>
</html>
```

Типы конструкций Razor

Все конструкции Razor можно условно разделить на два вида: однострочные выражения и блоки кода.

Пример применения однострочных выражений:

```
<p>Date: @DateTime.Now.ToLongDateString()</p>
```

В данном случае используется объект DateTime и его метод ToLongDateString()

Типы конструкций Razor

Или еще один пример:

```
<p>@(20 + 30)</p>
```

Так как перед скобками стоит знак @, то выражение в скобках будет интерпретироваться как выражение на языке C#. Поэтому браузер выведет число 50, а не "20 + 30".

Типы конструкций Razor

Блоки кода могут иметь несколько выражений. Блок кода заключается в фигурные скобки, а каждое выражение завершается точкой с запятой аналогично блокам кода и выражениям на C#.

Типы конструкций Razor

```
@{
   string head = "Hello, world!";              // определяем переменную head
   string text = "ASP.NET Core Application";// определяем переменную text
}
<!DOCTYPE html>
<html>
<head>
    <title>Hello</title>
    <meta charset="utf-8" />
</head>
<body>
    <h2>@head</h2> <!-- используем переменную head -->
    <div>@text</div> <!-- используем переменную text -->
</body>
</html>
```

Типы конструкций Razor

Если необходимо вывести значение переменной без каких-либо html-элементов, то мы можем использовать специальный снипет <text>:

```
@{
    int i = 8;
    <text>@i</text>
}
<text>@(i+1)</text>
```

Типы конструкций Razor

В Razor могут использоваться комментарии. Они располагаются между символами @**@:

```
@* текст комментария *@
```

Условные конструкции

Также мы можем использовать условные конструкции:

```
@{
    string morning = "Good Morning";
    string evening = "Good Evening";
    string hello = "Hello";
    int hour = DateTime.Now.Hour;
}
@if (hour < 12)
{
    <h2>@morning</h2>
}
else if (hour > 17)
{
    <h2>@evening</h2>
}
else
{
    <h2>@hello</h2>
}
```

Условные конструкции

Конструкция switch:

```
@{
    string language = "german";
}
@switch(language)
{
    case "russian":
        <h3>Привет мир!</h3>
        break;
    case "german":
        <h3>Hallo Welt!</h3>
        break;
    default:
        <h3>Hello World!</h3>
        break;
}
```

Циклы

Кроме того, мы можем использовать все возможные циклы. Цикл for:

```
@{
    string[] people = { "Tom", "Sam", "Bob" };
}
<ul>
    @for (var i = 0; i < people.Length; i++)
    {
        <li>@people[i]</li>
    }
</ul>
```

Циклы

Цикл foreach:

```
@{
    string[] people = { "Tom", "Sam", "Bob" };
}
<ul>
    @foreach (var person in people)
    {
        <li>@person</li>
    }
</ul>
```

Циклы

Цикл while:

```
@{
    string[] people = { "Tom", "Sam", "Bob" };
}
<ul>
    @while ( i < people.Length)
    {
        <li>@people[i++]</li>
    }
</ul>
```

Циклы

Цикл do..while:

```
@{
    var i = 0;
}
<ul>
    @do
    {
        <li>@(i * i)</li>
    }
    while ( i++ < 5);
</ul>
```

try...catch

Конструкция try...catch...finally, как и в C#, позволяет обработать исключение, которое может возникнуть при выполнение кода:

```
@try
{
    throw new InvalidOperationException("Something wrong");
}
catch (Exception ex)
{
    <p>Exception: @ex.Message</p>
}
finally
{
    <p>finally</p>
}
```

Вывод текста в блоке кода

Обычный текст в блоке кода мы не сможем вывести:

```
@{
    bool isEnabled = true;
}
@if (isEnabled)
{
    Hello World
}
```

Вывод текста в блоке кода

В этом случае Razor будет рассматривать строку "Hello" как набор операторов языка C#, которых, естественно в C# нет, поэтому мы получим ошибку.

Вывод текста в блоке кода

И чтобы вывести текст как есть в блоке кода, нам надо использовать выражение «@:»:

```
@{
    bool isEnabled = true;
}
@if (isEnabled)
{
    @: Hello
}
```

Функции

Директива @functions позволяет определить функции, которые могут применяться в представлении. Например:

```
@functions
{
    public int Sum(int a, int b)
    {
        return a + b;
    }
    public int Square(int n) => n * n;
}
<p>Sum of 5 and 4: <b> @Sum(5, 4)</b></p>
<p>Square of 4: <b>@Square(4)</b></p>
```

Передача данных в представление

Передача данных в представление

Существуют различные способы передачи данных из контроллера в представление:

- **ViewData**
- **ViewBag**
- **Модель представления**

ViewData

ViewData представляет словарь из пар ключ-значение:

```
public IActionResult Index()
{
    ViewData["Message"] = "Hello, world!";
    return View();
}
```

Здесь динамически определяется во ViewData объект с ключом "Message" и значением "Hello, world!". При этом в качестве значения может выступать любой объект.

ViewData

И после этому мы можем его использовать в представлении:

```
@{
    ViewData["Title"] = "ASP.NET Core MVC";
}

<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>
```

ViewData

Причем не обязательно устанавливать все объекты во ViewData в контроллере. Так, в данном случае объект с ключом "Title" устанавливается непосредственно в представлении. В итоге в браузере мы увидим значения обоих элементов из ViewData.

ViewBag

ViewBag во многом подобен ViewData. Он позволяет определить различные свойства и присвоить им любое значение. Так, мы могли бы переписать предыдущий пример следующим образом:

```
public IActionResult Index()
{
    ViewBag.Message = "Hello, world!";

    return View();
}
```

ViewBag

И таким же образом нам надо было бы изменить представление:

```
@{
    ViewBag.Title = "ASP.NET Core MVC";
}
<h2>@ViewBag.Title.</h2>
<h3>@ViewBag.Message</h3>
```

ViewBag

При этом свойства ViewBag могут содержать не только простые объекты типа string или int, но и сложные данные. Например, передадим список:

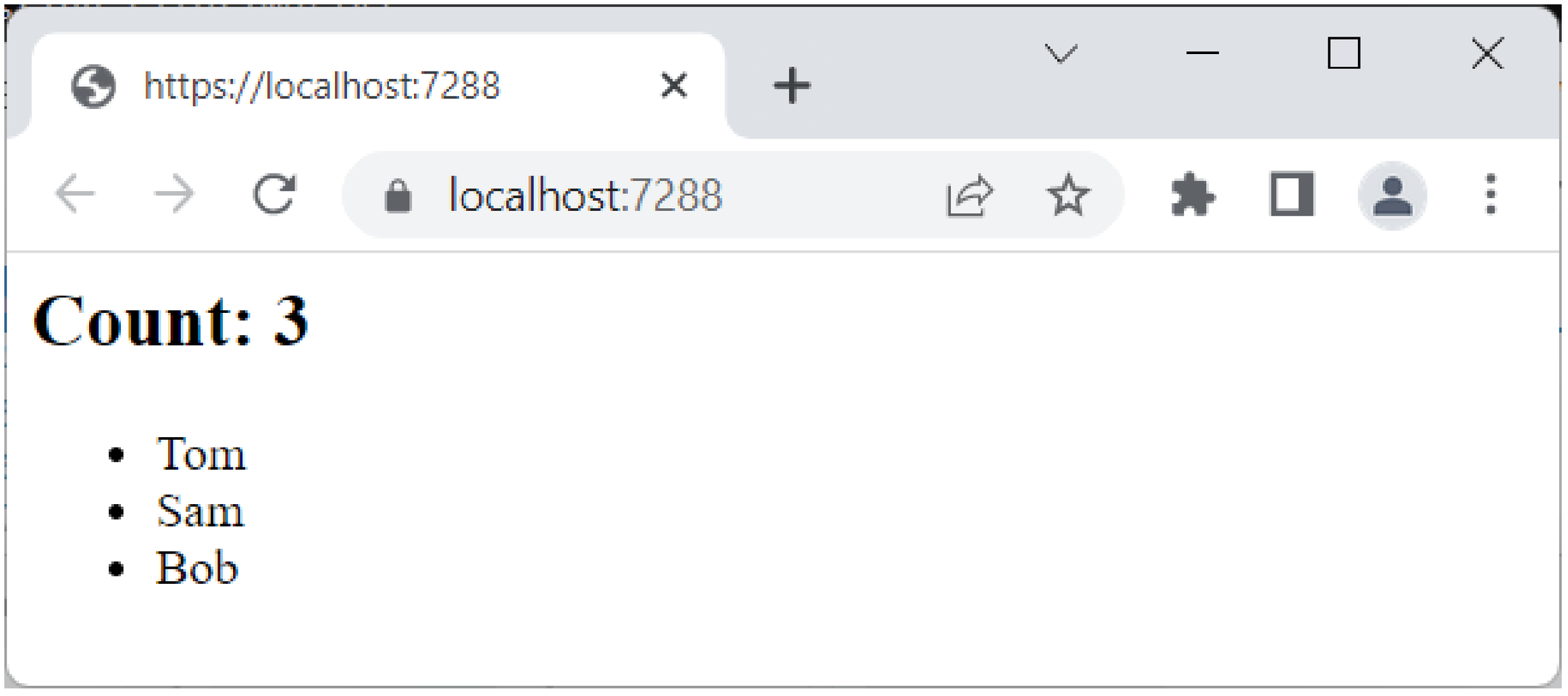
```
public IActionResult Index()
{
    ViewBag.People = new List<string> { "Tom", "Sam", "Bob" };
    return View();
}
```

ViewBag

И также в представлении мы можем получить этот список:

```
<h2>Count: @ViewBag.People.Count</h2>
<ul>
@foreach(string person in ViewBag.People)
{
    <li>@person</li>
}
</ul>
```

ViewBag



Модель представления

Модель представления является во многих случаях более предпочтительным способом для передачи данных в представление. Для передачи данных в представление используется одна из версий метода View:

```
public IActionResult Index()
{
    var people = new List<string> { "Tom", "Sam", "Bob" };
    return View(people);
}
```

Модель представления

В метод View передается список, поэтому моделью представления Index.cshtml будет тип **List<string>** (либо **IEnumerable<string>**). И теперь в представлении мы можем написать так:

```
@model List<string>

<h2>Count: @Model.Count</h2>
<ul>
@foreach(string person in Model)
{
    <li>@person</li>
}
</ul>
```

Модель представления

В самом начале представления с помощью директивы @model устанавливается модель представления. Тип модели должен совпадать с типом объекта, который передается в метод View() в контроллере.

```
@model List<string>

<h2>Count: @Model.Count</h2>
<ul>
@foreach(string person in Model)
{
    <li>@person</li>
}
</ul>
```

Модель представления

Установка модели указывает, что объект **Model** теперь будет представлять объект List<string> или список. И мы сможем использовать Model в качестве списка.

```
@model List<string>

<h2>Count: @Model.Count</h2>
<ul>
@foreach(string person in Model)
{
    <li>@person</li>
}
</ul>
```

Мастер-страницы layout

Мастер-страницы layout

Мастер-страницы или layout позволяют задать единый шаблон для представлений и применяются для создания единообразного, унифицированного вида сайта. По сути мастер-страницы - это те же самые представления, которе могут включать в себя другие представления.

Мастер-страницы layout

Для добавления мастер-страниц в Visual Studio можно использовать шаблон файла Razor Layout.

Мастер-страницы layout

Пример файла _Layout.cshtml:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <h2>@ViewBag.Title</h2>
    <div><a href="/Home/Index">Home</a> | <a href="/Home/About">About</a></div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Мастер-страницы layout

Главное же отличие от обычных представлений состоит в использовании метода **@RenderBody()**, который является плейсхолдером и на место которого потом будут подставляться другие представления, использующие данную мастер-страницу. В итоге мы сможем легко установить для всех представлений веб-приложения единообразный стиль оформления.

Мастер-страницы layout

Пример файла Index.cshtml:

```
@{
    ViewBag.Title = "Index";
    Layout = "/Views/Shared/_Layout.cshtml";
}

<h3>Index Content</h3>
```

ViewStart

Хотя выше приведенный код вполне успешно работает, у нас есть одна проблема - мы сталкиваемся с необходимостью в каждом представлении явным образом прописывать, какую мастер-страницу layout будет применять представление. Чтобы упростить данное действие, можно применять файлы **_ViewStart.cshtml**

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

Файл _ViewImports.cshtml

Файл _ViewImports.cshtml

Файл _ViewImports.cshtml позволяет по умолчанию подключить в представления некоторый функционал.

Файл _ViewImports.cshtml

Например, у нас в проекте есть некоторый класс Person:

```
namespace Person
{
    public class Person {
        public string Name {get; set; }
        public int Age { get; set; }
    }
}
```

Файл _ViewImports.cshtml

Допустим, мы хотим использовать тип Person в представлении Index.cshtml:

```
@using Person @* Подключаем пространство имен класса Person *@
@{
    Layout = null;
    Person tom = new Person("Tom", 37);
}
<h2>Person Data</h2>
<h3>Name: @tom.Name</h3>
<h3>Age: @tom.Age</h3>
```

Файл _ViewImports.cshtml

Чтобы использовать тип Person в представлении, мы вынуждены импортировать с помощью директивы using пространство имен, где этот тип определен. Это может создавать некоторые неудобства. Файл **_ViewImports.cshtml** поможет избежать этих неудобств.

```
@using Person @* Подключаем пространство имен класса Person *@
```

Файл _ViewImports.cshtml

Для каждой группы представлений в одной папке мы можем определить свой файл *_ViewImports.cshtml*. Например, если мы хотим отдельно подключать ряд пространств имен в только в представления контроллера *HomeController*, тогда нам надо добавить файл *_ViewImports.cshtml* в каталог *Views/Home*.

Маршрутизация

Маршрутизация

Для добавления маршрутов в MVC мы можем применять следующие методы IEndpointRouteBuilder:

- MapControllerRoute()
- MapDefaultControllerRoute()
- MapAreaControllerRoute()
- MapControllers()
- MapFallbackToController()

MapControllerRoute

MapControllerRoute() определяет произвольный маршрут и принимает следующие параметры:

- name: название машрута
- pattern: шаблон машрута
- defaults: значения параметров маршрутов по умолчанию
- constraints: ограничения маршрута
- dataTokens: определения токенов маршрута

MapControllerRoute

```
MapControllerRoute(string name, string pattern, [object defaults = null],
[object constraints = null], [object dataTokens = null])
```

MapDefaultControllerRoute

MapDefaultControllerRoute() определяет стандартный маршрут, фактически эквивалентен вызову

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

MapAreaControllerRoute

MapAreaControllerRoute() определяет маршрут, который также учитывает область приложения. Имеет следующие параметры:

```
MapAreaControllerRoute(string name, string areaName, string pattern, [object defaults =
   null], [object constraints = null], [object dataTokens = null])
```

MapControllers

MapControllers() сопоставляет действия контроллера с запросами, используя маршрутизацию на основе атрибутов. Про атрибуты маршрутизации будет сказано в последующих статьях.

MapFallbackToController

MapFallbackToController() определяет действие контроллера, которое будет обрабатывать запрос, если все остальные определенные маршруты не соответствуют запросу. Принимает имя контроллера и его метода:

```
MapFallbackToController(string action, string controller)
```

Получение параметров маршрутов в контроллере

Мы можем получить в контроллере все параметры маршрута, используя объект **RouteData**:

```
public class HomeController : Controller
{
    public string Index()
    {
        var controller = RouteData.Values["controller"];
        var action = RouteData.Values["action"];
        return $"controller: {controller} | action: {action}";
    }
}
```

КОНЕЦ