Представления

Проектирование и разработка веб-сервисов

Вводные понятия

- Структура веб-приложения
- Шаблоны приложений ASP.NET Core
- Структура проекта ASP.NET Core
- Концепция паттерна MVC

Представления (View)

В большинстве случаев при обращении к веб-приложению пользователь ожидает получить веб-страницу с какими-нибудь данными. В MVC для этого, как правило, используются представления, которые и формируют внешний вид приложения.

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#.

Простейшее представление

```
@{
    Layout = null;
}
<!doctype html>
<html>
<head>
    <title>Hello ASP.NET</title>
    <meta charset="utf-8" />
</head>
<body>
    <h2>Привет ASP.NET Core!</h2>
</body>
</html>
```

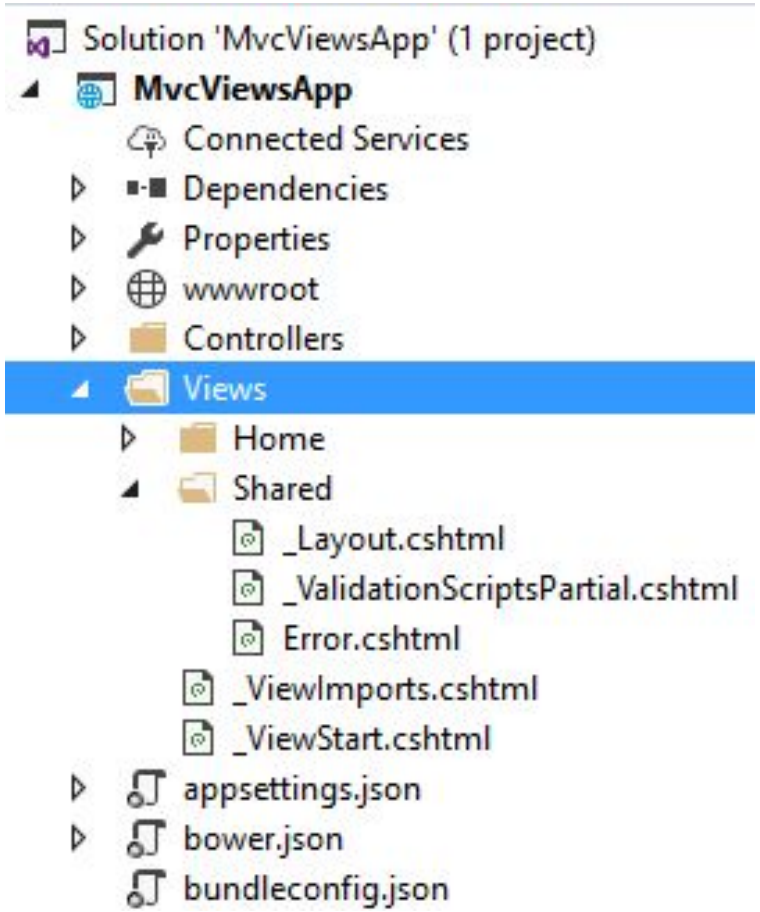
Представления

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.

Представления

Для хранения представлений в проекте ASP.NET MVC предназначена папка Views:



Представления

В этой папке уже есть некоторая подструктура. Во-первых, 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.

Также здесь есть папка Shared, которая хранит общие представления для всех контроллеров. По умолчанию это файлы:

- *_Layout.cshtml* (используется в качестве мастер-страницы),
- *Error.cshtml* (используются для отображения ошибок)
- *_ValidationScripsPartial.cshtml* (частичное представление, которое подключает скрипты валидации формы).

Представления

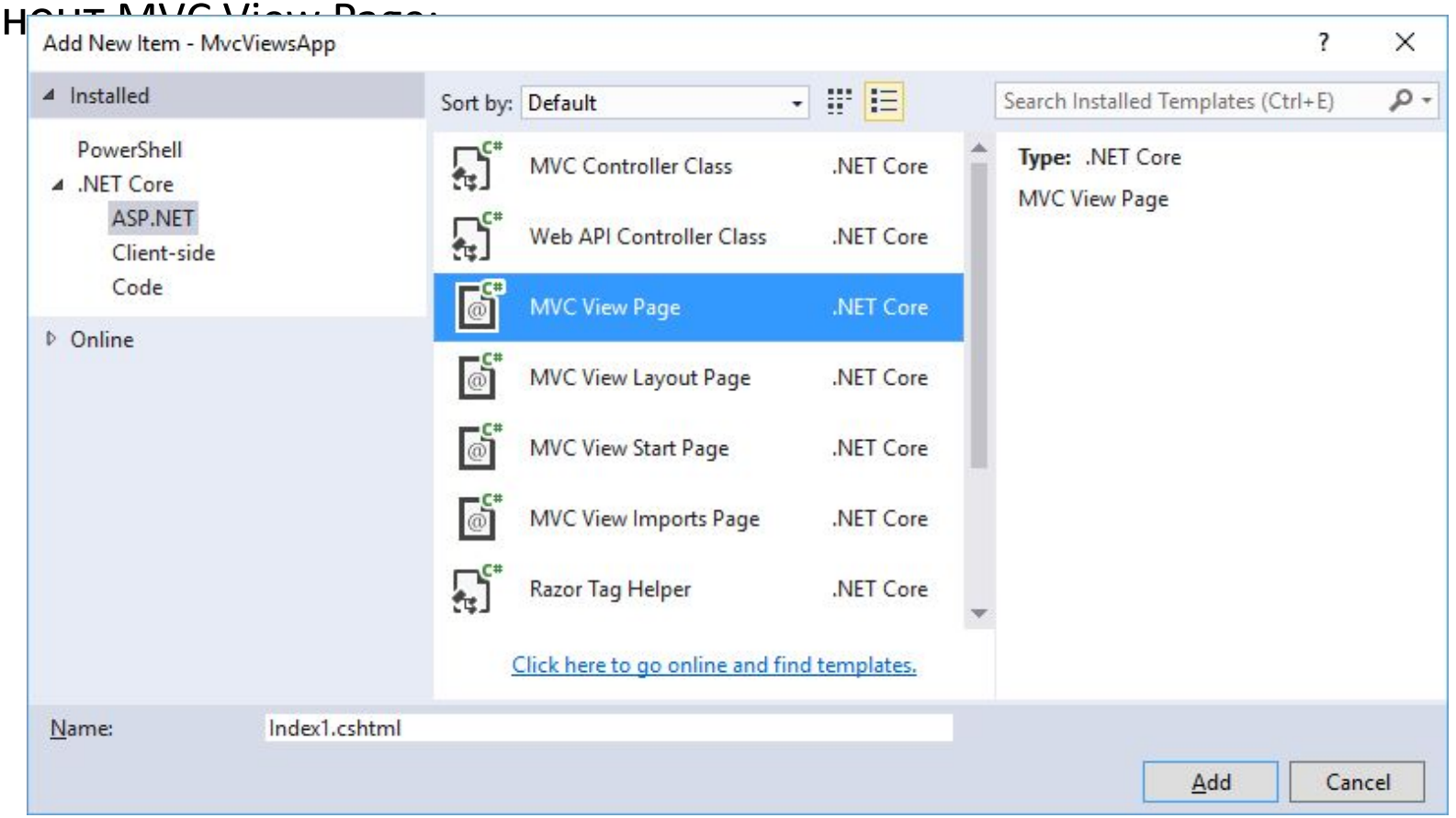
И в корне каталога Views также можно найти два файла:

1. _ViewImports.cshtml
2. _ViewStart.cshtml.

Эти файлы содержат код, который автоматически добавляется ко всем представлениям. _ViewImports.cshtml устанавливает некоторые общие для всех представлений пространства имен, а _ViewStart.cshtml устанавливает общую мастер-страницу.

Представления

При необходимости мы можем добавлять в каталог Views какие-то свои представления, каталоги для представлений. И они необязательно должны быть связаны с контроллерами и их методами. Для добавления представления нужно правой кнопкой мыши на подкаталог в папке Views (или на саму папку Views) и в контекстном меню выбрать Add -> New Item. Затем в появившемся окне добавления нового элемента выбрать компонент MVC View Page:



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 используется метод View:

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

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

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/.

ViewResult

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

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

В этом случае метод Index будет использовать представление 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/Index.cshtml");
    }
}
```

Razor

В действительности при вызове метода View контроллер не производит рендеринг представления и не генерирует разметку html. Контроллер только готовит данные и выбирает, какое представление надо возвратить в качестве объекта ViewResult. Затем уже объект ViewResult обращается к движку представления для рендеринга представления в выходной ответ.

По умолчанию в ASP.NET MVC Core используется 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#.

Синтаксис Razor довольно прост - все его конструкции предваряются символом @, после которого происходит перед к коду C#.

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

Razor

Пример применения однострочных выражений:

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

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

Или еще один пример:

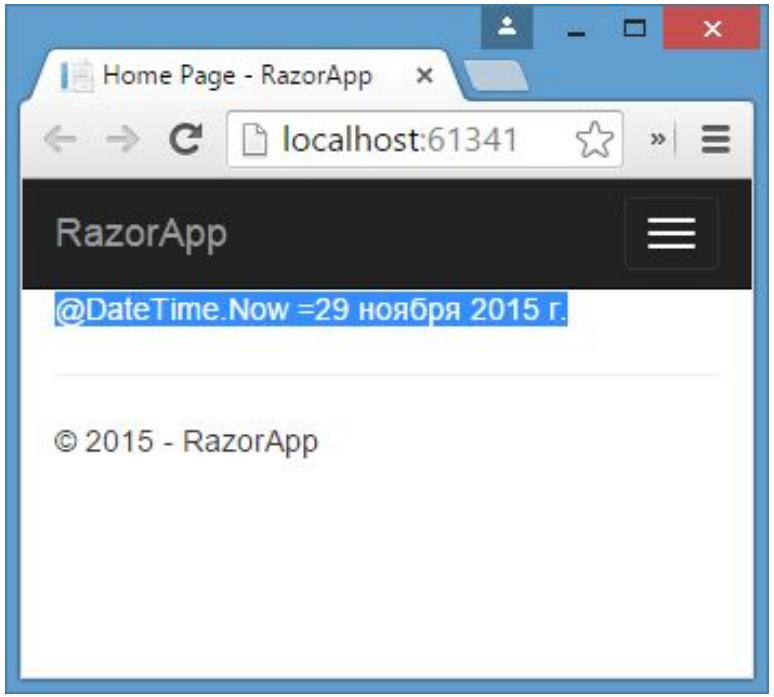
```
<p>@(20 + 30)</p>
```

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".

Razor

Но если вдруг мы создаем код html, в котором присутствует символ @ не как часть синтаксиса Razor, а сам по себе, то, чтобы его отобразить, нам надо его дублировать:

* @@DateTime.Now =@DateTime.Now.ToLongDateString() *



Razor

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

```
@{
    string head = "Привет мир!!!";
    head = head + " Добро пожаловать на сайт!";
}
<h3>@head</h3>
```

В блоках кода мы можем определить обычные переменные и потом их использовать в представлении.

Razor

Весь код в пределах блока расценивается как код c#. Однако с помощью конструкции @: мы можем в блоке кода выводить на веб-страницу текст:

```
@{
    string head = "Hello world";
    @: <b>Привет мир!</b>
    head = head + "!!";
}
<p>@head</p>
```

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>
```

Razor

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

@ текст комментария *@*

Razor

Управляющие конструкции языка C# в Razor:

- If, else
- Switch
- For
- Foreach
- While
- Do…while
- Using
- Try, catch, finally

Razor – *if, else*

```
@{
    string head = "Привет мир";
    bool isEnabled = false;
}
@if (isEnabled)
{
    <p>Добро пожаловать</p>
}
else
{
    <p>@head</p>
}
```

Razor – *switch*

```
@{
    int x = 6;
}

@switch(x)
{
    case 5:
        <p>@(x* x)</p>
        break;
    case 6:
        <p>@(x+ x)</p>
        break;
}
```

Razor – *for*

```
@for (var i = 1; i < 6; i++)
{
    <p>Строка: @i</p>
}
```

Razor – *while*

```
@{
    int x = 1;
}

@while(x<6)
{
    <p>Строка: @x</p>
    x++;
}
```

Razor – do…*while*

```
@{
    int x = 1;
}

@do
{
    <p>Строка: @x</p>
    x++;
}
while (x < 6);
<p>Конец</p>
```

Razor – *foreach*

```
@{
    string[] phones = { "Lumia 950", "iPhone 6S", "Galaxy S 6",
"LG G4" };
}
<ul>
    @foreach (var phone in phones)
    {
        <li>@phone</li>
    }
</ul>
```

Razor – *try…catch…finally*

```
@try
{
    throw new InvalidOperationException("Что-то пошло не так");
}
catch (Exception ex)
{
    <p>Возникло исключение: @ex.Message</p>
}
finally
{
    <p>Блок finally</p>
}
```

Передача данных в представление

Существуют различные способы передачи данных из контроллера в представление:

- ViewData
- ViewBag
- TempData
- Модель представления

ViewData

ViewData представляет словарь из пар ключ-значение:

```
public IActionResult About()
{
    ViewData["Message"] = "Hello ASP.NET Core";

    return View();
}
```

ViewData

Здесь динамически определяется во ViewData объект с ключом "Message" и значением "Hello ASP.NET Core". При этом в качестве значения может выступать любой объект. И после этому мы можем его использовать в представлении:

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Причем не обязательно устанавливать все объекты во ViewData в контроллере. Так, в данном случае объект с ключом "Title" устанавливается непосредственно в представлении.

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 About()
{
    ViewBag.Message = "Hello ASP.NET Core";

    return View();
}
```

И не важно, что изначально объект ViewBag не содержит никакого свойства Message, оно определяется динамически.

ViewBag

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

```
public IActionResult About()
{
    ViewBag.Countries = new List<string> { "Бразилия", "Аргентина", "Уругвай", "Чили" };
    return View();
}
```

И также в представлении мы можем получить этот список:

```
@foreach(string country in ViewBag.Countries)
{
    <p>@country</p>
}
```

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 About()
{
    List<string> countries = new List<string> { "Бразилия", "Аргентина", "Уругвай", "Чили" };
    return View(countries);
}
```

Модель представления

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

```
@model List<string>
@{
    ViewBag.Title = "About";
}

<h3>В списке @Model.Count элемента</h3>
@foreach(string country in Model)
{
    <p>@country</p>
}
```

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.

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

Представления, для которых определена модель, еще называют **строго типизированными.**

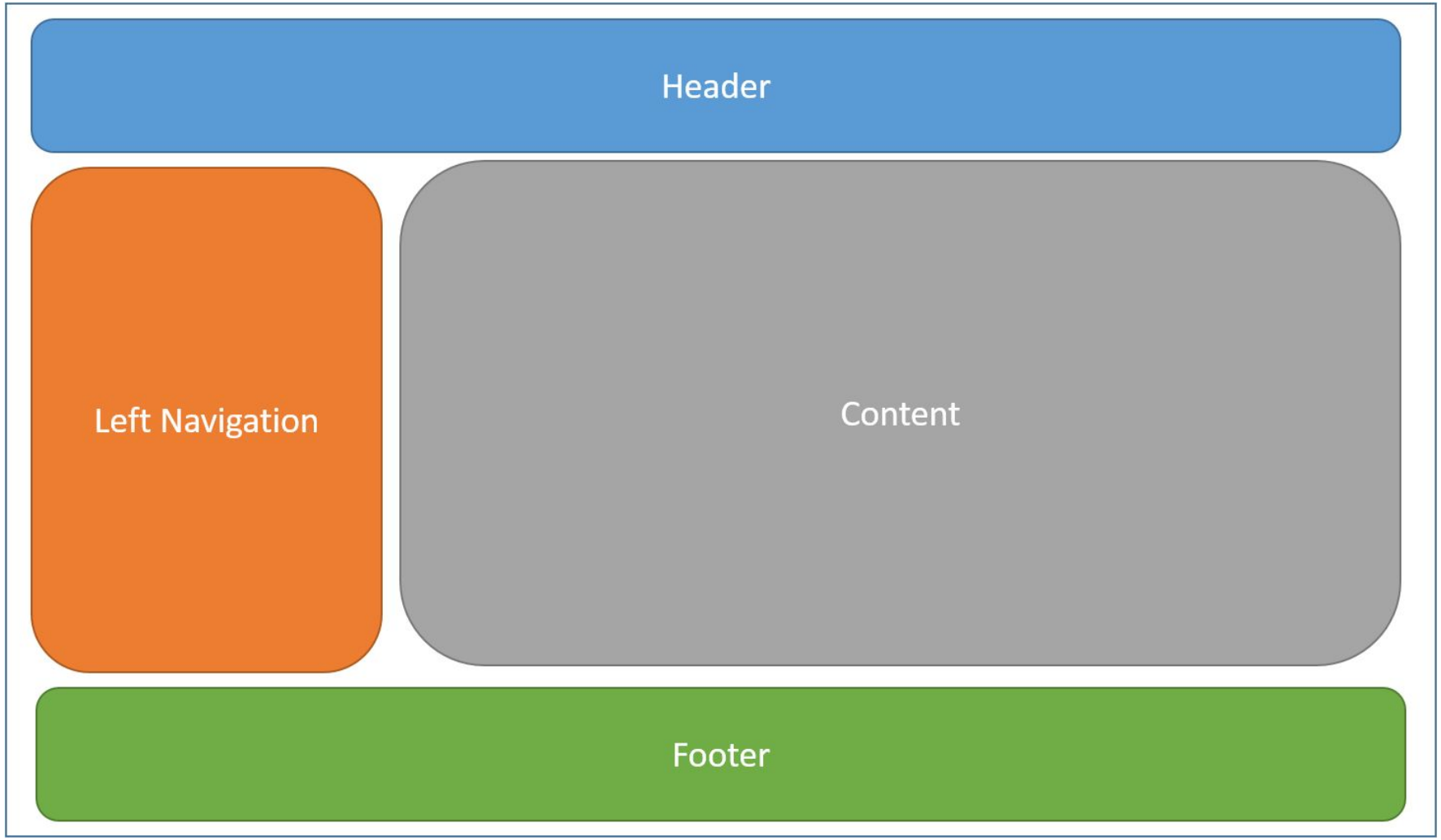
Мастер-страницы

Когда у нас в проекте много представлений, и все они содержат какие-то общие элементы, то вместо того, чтобы пописывать все эти элементы в каждом представлении, гораздо удобнее задать один общий шаблон. В этом случае при изменении каких-то общих элементов будет достаточно изменить один раз в общем шаблоне, не изменяя всех остальных представлений. В ASP.NET MVC таким шаблоном являются мастер-страницы.

Мастер-страницы применяются для создания единообразного, унифицированного вида сайта. По сути мастер-страницы - это те же самые представления, которе могут включать в себя другие представления. Например, можно определить на мастер-странице общие для всех остальных представлений меню, а также подключить общие стили и скрипты.

Специальные теги позволяют вставлять в определенное место на мастер-страницах другие представления.

Мастер-страницы



Мастер-страницы

По умолчанию при создании нового проекта ASP.NET MVC Core в проект уже добавляется мастер-страница под названием _Layout.chtml, которую можно найти в каталоге Views/Shared.

Код мастер-страницы напоминает полноценную веб-страницу: здесь присутсвуют основные теги <html>, <head>, <body> и так далее. И также здесь могут использоваться конструкции Razor. Фактически это то же самое представление. 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.

ViewStart

По умолчанию представления уже подключают мастер-страницу за счет файла _ViewStart.cshtml. Этот файл можно найти в проекте в папке Views. Код этого файла добавляется в самое начало кода преставлений при их запуске.

По умолчанию файл _ViewStart.cshtml содержит следующий код:

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

В каждом представлении через синтаксис Razor доступно свойство Layout, которое хранит ссылку на мастер-страницу. Здесь в качестве мастер страницы устанавливается файл _Layout.cshtml. При этом расширение можно не использовать.

Когда будет происходить рендеринг представления, то система будет искать мастер страницу _Layout по следующим путям:

- /Views/[Название_контроллера]/_Layout.cshtml
- /Views/Shared/_Layout.cshtml

ViewStart

Код из _ViewStart.cshtml выполняется до любого кода в представлении. И чтобы переопределить мастер-страницу, в представлении достаточно установить свойство Layout. Мы можем вообще не использовать мастер-страницу, тогда нам надо присвоить значение null:

```
@{
    Layout = null;
}
```

Секции

Кроме метода RenderBody(), который вставляет освновное содержимое представлений, мастер-страница может также использовать специальный метод **RenderSection()** для вставки секций. Мастер-страница может иметь несколько секций, куда представления могут поместить свое содержимое. Например, добавим к мастер-странице _Master.cshtml секцию footer:

Секции

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
    <footer> @RenderSection("Footer") </footer>
</body>
</html>
```

Секции

Теперь при запуске предыдущего представления Index мы получим ошибку, так как секция Footer не определена. По умолчанию представление должно передавать содержание для каждой секции мастер-страницы. Поэтому добавим вниз представления Index секцию footer. Это мы можем сделать с помощью выражения @section:

```
@{
    ViewData["Title"] = "Home Page";
    Layout = "~/Views/_Master.cshtml";
}
<h2>Представление Index.cshtml</h2>

@section Footer {
    Все права защищены. Site Corp. 2016.
}
```

Секции

Но при таком подходе, если у нас есть куча представлений, и мы вдруг захотели определить новую секцию на мастер-странице, нам придется изменить все имеющиеся представления, что не очень удобно. В этом случае мы можем воспользоваться одним из вариантов гибкой настройки секций.

@RenderSection("Scripts", required: false)

_ViewImports.cshtml

Содержание файла по умолчанию в проекте MVC:

```
@using Microsoft.AspNetCore.Identity
@using WebTest
@using WebTest.Models
@using WebTest.Models.AccountViewModels
@using WebTest.Models.ManageViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

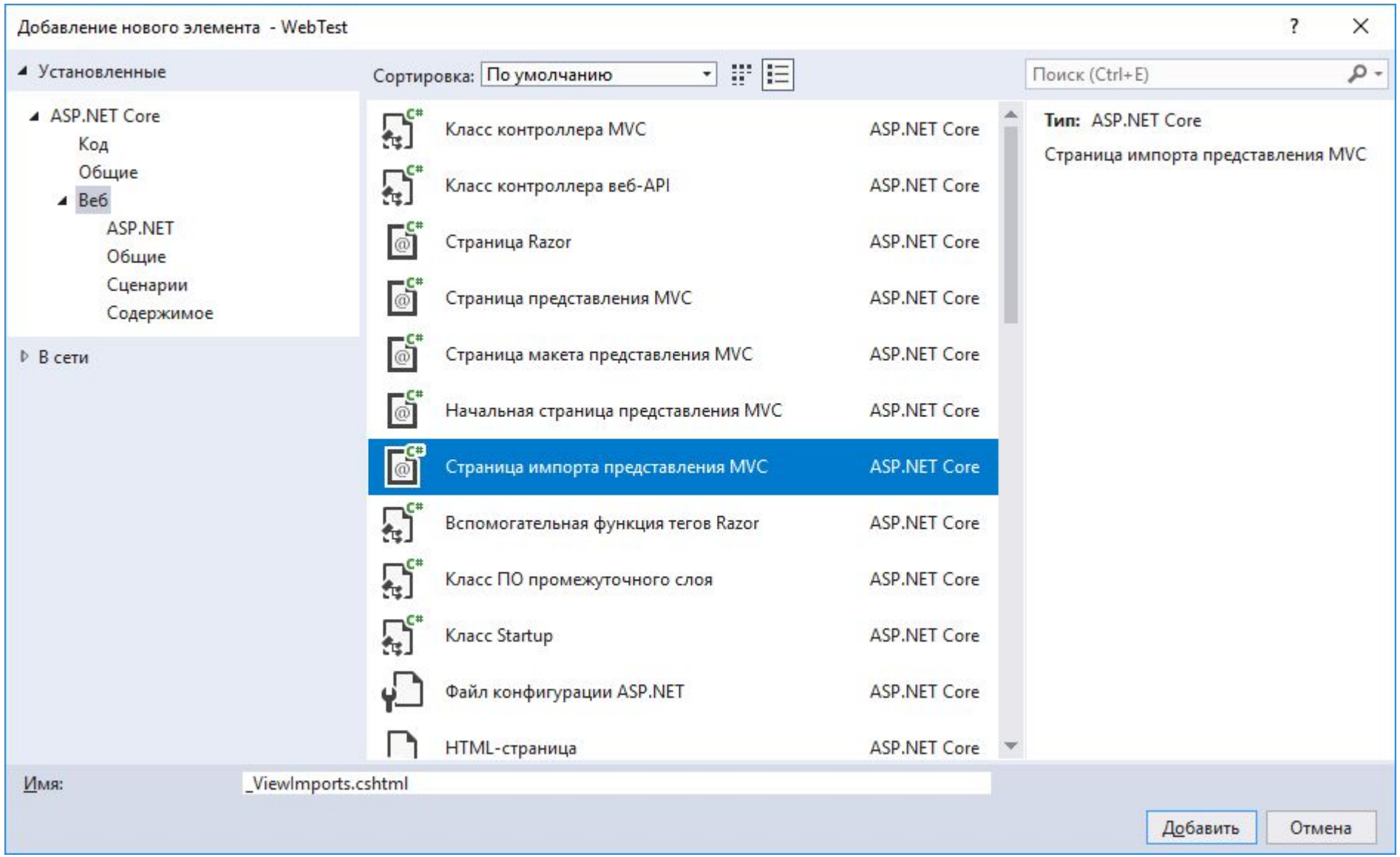
_ViewImports.cshtml

Для каждой группы представлений в одной папке мы можем определить свой файл _ViewImports.cshtml. Так пускай, в проекте будет добавлен новый контроллер OtherController, в папке Views для его представлений будет добавлена подпапка Other.

В эту папку Views/Other добавим новый файл _ViewImports.cshtml. Для его добавления можно выбрать спецальный шаблон «MVC View Imports Page».

И теперь все директивы и выражения из файла *Views/Other/_ViewImports.cshtml* будут применяться к представлениям только из папки Views/Other. Кроме того, ко всем представлениям во всех папках продолжит применяться глобальный файл *Views/_ViewImports.cshtml.*

_ViewImports.cshtml



Частичные представления

В приложениях на ASP.NET MVC кроме обычных представлений и мастер-страниц можно также использовать частичные представления или partial views. Их отличительной особенностью является то, что их можно встраивать в другие обычные представления. Частичные представления могут использоваться также как и обычные, однако наиболее удобной областью их использования является рендеринг результатов AJAX-запроса. По своему действию частичные представления похожи на секции, которые использовались в прошлой теме, только их код выносится в отдельные файлы.

Частичные представления полезны для создания различных панелей веб-страницы, например, панели меню, блока входа на сайт, каких-то других блоков.

Частичные представления

За рендеринг частичных представлений отвечает объект **PartialViewResult**, который возвращается методом PartialView. Итак, определим в контроллере HomeController новое действие GetMessage:

```
public class HomeController : Controller
{
    public ActionResult GetMessage()
    {
        return PartialView("_GetMessage");
    }
    //.....................
}
```

Частичные представления

По своему действию частичное представление похоже на обычное, только для него по умолчанию не определяется мастер-страница.

Для встраивания частичного представления в обычные изменим представление Index.cshtml:

```
@{
    ViewData["Title"] = "Home Page";
}
<h2>Представление Index.cshtml</h2>

@Html.Partial("_GetMessage")
```

Обращения к методу GetMessage() в контроллере при этом не происходит.

Частичные представления

Одна из перегруженных версий методов Html.Partial позволяет передать модель в частичное представление. В итоге у нас получится стандартное строго типизированное представление. Например, в качестве второго параметра список строк:

```
@Html.Partial("_GetMessage", new List<string> { "Lumia 950",
"iPhone 6S", "Samsung Galaxy s 6", "LG G 4" })
```

Работа с формами

Формы представляют одну из форм передачи наборов данных на сервер. Как правило, для создания форм и их элементов в MVC применяются либо **html-хелперы**, либо **tag-хелперы**, которые рассматриваются далее. Однако в данном случае мы рассмотрим взаимодействие на примере стандартных тегов html, которые создают элементы формы.

Работа с формами

Например, у нас есть действие Login:

```
public class HomeController : Controller
{
    [HttpGet]
    public IActionResult Login()
    {
        return View();
    }
    [HttpPost]
    public IActionResult Login(string login, string password)
    {
        string authData = $"Login: {login}   Password: {password}";
        return Content(authData);
    }
}
```

Работа с формами

Одно действие расщеплено на два метода: GET-версию, которая отдает представление с формой ввода, и POST-версию, которая принимает введенные в эту форму данные.

Теперь создадим само представление. Добавим в папку Views/Home новый файл Login.cshtml:

```
@{
    ViewData["Title"] = "Login";
}
<form method="post">
    <label>Логин:</label><br />
    <input type="text" name="login" /><br /><br />
    <label>Пароль:</label><br />
    <input type="text" name="password" /><br /><br />
    <input type="submit" value="Отправить" />
</form>
```

Работа с формами

Чтобы инфраструктура MVC могла автоматически связать данные из формы с параметрами метода, значения атрибутов name у полей формы совпадают с именами параметров.

Таким образом, когда форма отправится на сервер, при обработке запроса фреймворк MVC автоматически свяжет значения полей формы с параметрами.

Работа с формами

В HTML мы можем использовать ряд встроенных элементов:

1. Списки (select)
2. Поля для ввода многострочного текста (textarea)
3. Элементы для ввода однострочного текста разных типов:
   - Текст (text)
   - Пароли (password)
   - Числа (number)
   - Логические значения (checkbox)
   - Радиокнопки (radio)