GeekBrains

ASP.NET MVC и углубление в изучение C#

MVC. Начало работы

На этом уроке

1. Углубимся в Razor
2. Узнаем какие есть технологии отображения страниц HTML
3. Создадим первое приложение MVC
4. Разберем отличия между Web API
5. Познакомимся с ViewModel

Оглавление

Детальнее об Razor

Часть приложений ASP строится на генерации верстки на стороне сервера. Генерация верстки связана с шаблонизацией некоторых текстовых файлов, в которых есть управляющие конструкции. Проще всего представить шаблонизацию как замену подстроки с помощью функции String.Replace(). Для упрощения работы с шаблонами используется более продвинутая техника, чем замена подстроки. Был разработан специальных синтаксис, который позволяет встраивать .NET код внутрь шаблонов. Такой механизм называется Razor (в переводе “бритва”). Файлы содержащие razor-разметку обычно имеют cshtml-расширение. Razor очень похож на движки шаблонизации, применяемые в клиентской разработке при построении SPA-приложений. Например, React имеет JSX-синтаксис встраивания JS-кода внутрь HTML.

Шаблоны состоят преимущественно из HTML-верстки, в которую вставляют небольшие участки кода C# с управляющими методами, которые начинаются с символа @. Символ собаки является зарезервированным символом в шаблонах. Например:

```
<p>@DateTime.Now</p>

<p>@DateTime.IsLeapYear(2021)</p>
```

Он позволяет вывести дату внутри тега параграфа и булевое значение внутри другого параграфа. Так любое значение, выводимое внутри шаблона будет приведено к строке посредством неявного вызова метода ToString у возвращенного в конструкции с символом @.

```
<p>@GenericMethid<int>()</p>
```

При этом выражения с дженериками не поддерживаются, так как символ < и > интерпретируются как символы разметки. Такой синтаксис не валидный. Сборка приложения приведет к ошибке. Выражения внутри шаблонов не обязательно писать однострочными, поддерживаются выражения. Для этого нужно оборачивать выражения после символа @ круглыми скобками (и), определяя границы выражения:

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>

<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>

<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>

@{

      var joe = new Person("Joe", 33);

}


<p>Age@(joe.Age)</p>
```

Скобки используются в строке 1 и позволяют вывести разницу дат в виде корректного значения. В отличие от этого, в строке 2 выражение не оборачивается в скобки и происходит вывод некорректного результата (строка 5). Первая часть выражения вычислялась, а вторая - нет, так как знак @ управляет только выводом первой конструкции. Вторая конструкция интерпретируется как отдельная строка. Выражение может состоять из несколько операторов. Более того, переменные, определенные в одном выражении, могут быть использованы далее в шаблоне строки 7-11 так, как если бы это был обычный C#-код. Для вывода похожей на тэги (верстку) информации сделан специальный помощник @Html.Raw("Hello World"). Html.Raw позволяет выводить потенциально опасный контент, в котором могут содержаться открывающие и закрывающие теги.

Помимо выражений могут быть определены целые блоки кода. Для этого используются фигурные скобки { и }. Оборачивая в такие скобки, можно встраивать любой C#-код внутрь блока. В том числе динамически определять собственные методы в шаблоне, которые можно будет использовать для вызовов строки 13-21 и результата строки 23-24.

```
@{
      var quote = "The future depends on what you do today. - Mahatma
Gandhi";
}

<p>@quote</p>

@{
      quote = "Hate cannot drive out hate, only love can do that. -
Martin Luther King, Jr.";
}

<p>@quote</p>

@{
      void RenderName(string name)
      {
      <p>Name: <strong>@name</strong></p>
      }

      RenderName("Mahatma Gandhi");
      RenderName("Martin Luther King, Jr.");
}
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

Вся эта магия работает благодаря движку Razor, который переводит шаблон из cstml-файла в некоторый сгенерированный код на C#, собственно который уже и занимается формированием конечной строки с HTML:

```
@{
      var quote = "Getting old ain't for wimps! - Anonymous";
}

<div>Quote of the Day: @quote</div>
```

Этот код трансформируется в итоге в классический код на C#, который и будет вызываться для ответа пользователю:

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
      public override async Task ExecuteAsync()
      {
      var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
      Write(output);
        WriteLiteral("</div>");
      }
}
```

Шаблоны могут иметь модель данных, которая используется для генерации шаблона. Например, можно вывести статическую информацию в шаблоне об адресе компании. Либо можно вывести список адресов компании, которые берутся из базы данных. В первом случае можно сделать обычный шаблон, а во втором - определить модель, через которую можно передать данные из источника в шаблон для дальнейшей генерации. Для этого в самом шаблоне нужно описать тип модели через @model IndexModel, где IndexModel - название типа модели, который будет использоваться в процессе генерации шаблона. Можно указать любой тип @model List<string>, тогда в самом шаблоне будет доступна переменная @Model (с большой буквы), в которой будут содержаться данные. Эти данный наполняются из обработчика (механизм будет рассмотрен позже). Определяя тип модели, мы, по сути, ограничиваем шаблон от использования произвольных данных. Так, например, по умолчанию

можно передать объект любого типа в шаблон. Модель шаблона при этом будут считаться как dynamic-тип. Это несет ряд проблем с поддержкой кода, например, при переименовании поля объекта, переданного как dynamic, потеряются статические проверки и ошибка может попасть в публичным доступ. Также механизм с dynamic-типами сложнее с точки зрения вычислений и может замедлять приложения.

Разница между технологиями

Есть несколько подходов к построению веб-приложения. ASP дает сделать практически любое приложение с генерацией верстки на стороне сервера – Razor Pages и ASP MVC.

Приложения с серверной генерацией содержат всю логику и исходный код на сервере. Так устройства, работающие с таким сайтом, могут обладать меньшими вычислительными способностями. Серверная генерация позволяет быстрее отрисовать страницу в браузере, а также позволяет запускать сайты без наличия javascript на клиенте. Серверные приложения позволяют использовать всю мощь сервера для выполнения прикладных задач. Пример приложений: персональные сайты, предоставляющие страницы с данными или формами; сайты с информацией в виде статическими списков; блоги; системы управления контентом.

Приложения с клиентской генерацией позволяют создавать страницу на лету, используя браузера клиента. Преимущества таких приложений: практически мгновенное взаимодействие и ответ на действия пользователя без необходимости запросов на сервере и ожидания ответа; позволяет делать частичные обновления без необходимости перезагрузки всей страницы; может работать без наличия сервера и синхронизировать изменения с появлением связи; снижает расходы на сервер за счет снижения необходимости генерировать верстку.

Таким образом, подходы можно разделить на серверный и клиентский. Серверный - это ASP MVC и ASP Pages. Клиентский - это классический Web API. Серверный подход занимается генерацией верстки на стороне сервера. Здесь есть файлы с версткой и сервер возвращает страницы со сгенеренной версткой. Клиентский же подход, это в браузере на стороне, соответственно, клиента. При этом в клиентском подходе данные передаются посредством API в виде JSON формата. ASP Pages и MVC делятся структурно. Pages позволяет описать логику работы с данными в модели. MVC вводит новый слой и разделяет данные от логики.

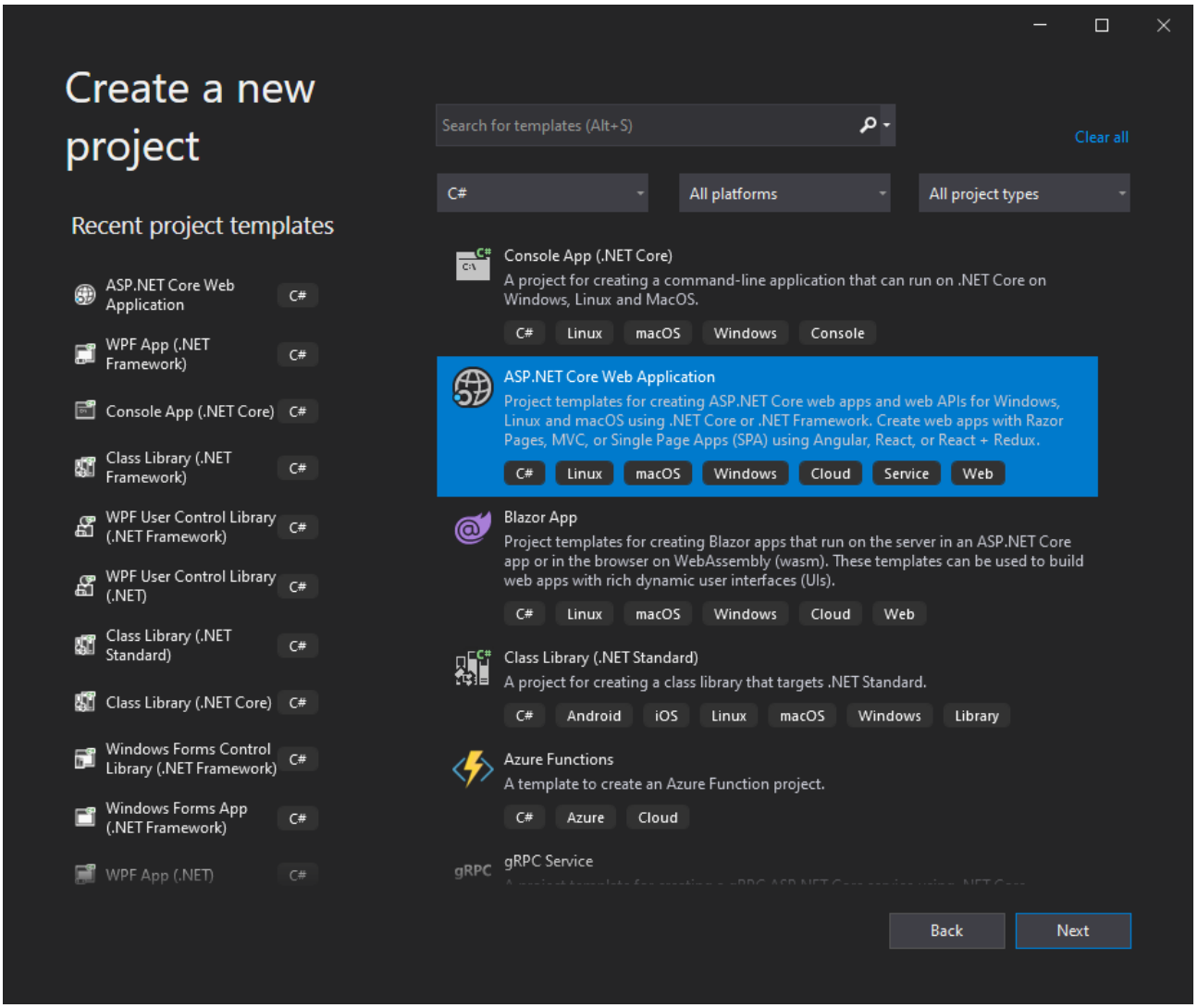
ASP MVC (Model View Controller)

Сам паттерн MVC позволяет разделить ответственность работы с представлением и логикой работы на отдельные компоненты. Так в отличие, к примеру от Razor Pages, View и Model остаются и

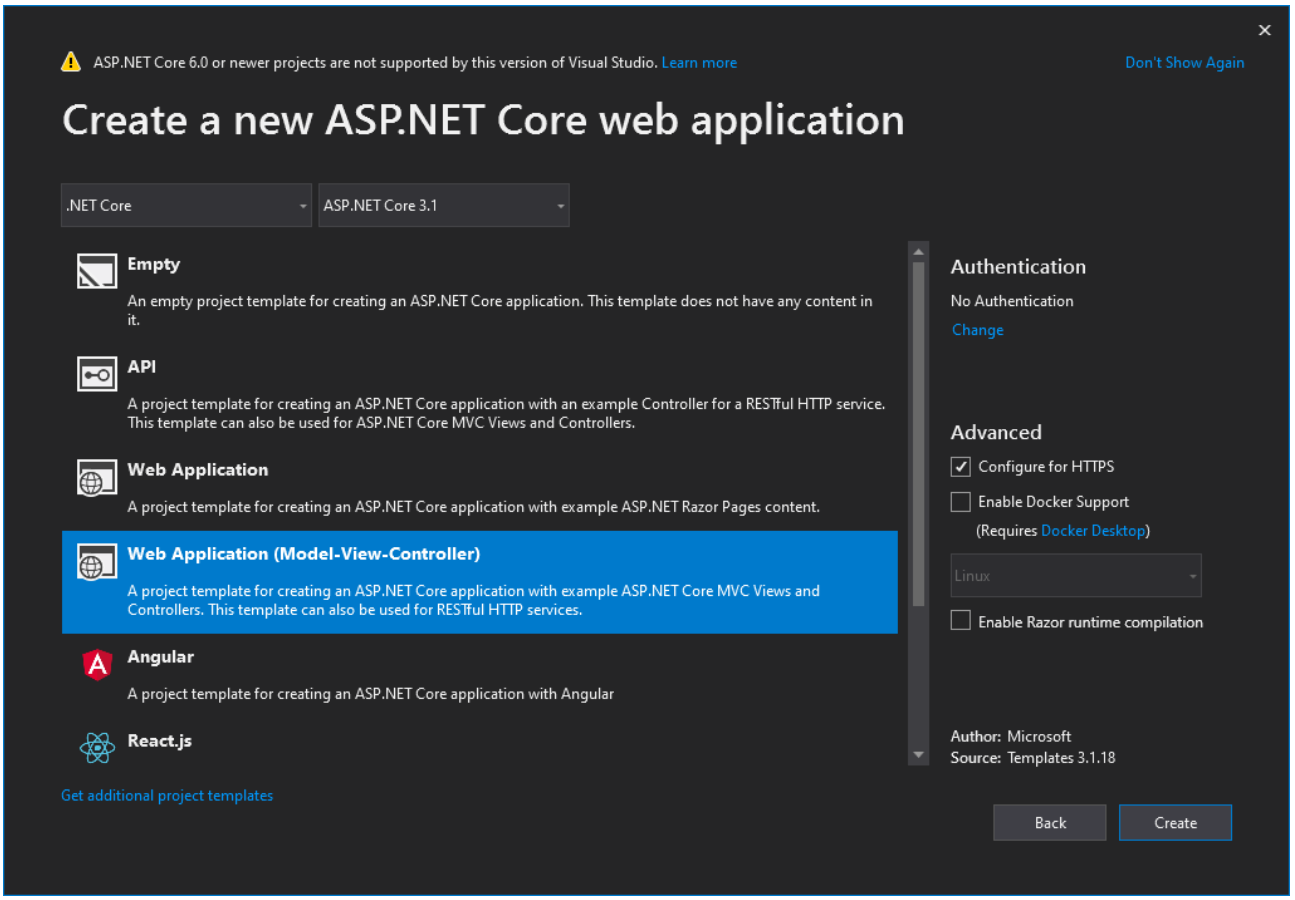
меняется только модель, которая перестает содержать логику обработки запросов. Логика переходит на Controller. При этом возрастает гибкость в отношении возможностей кастомизации самого приложения. Вместо ответа страницами на клиента можно подменят ответ на другие статус-коды, или подменять шаблоны на лету, или расширять приложение с размера “десятки страниц” на “сотни страниц” без потери скорости и качества разработки.

Конфигурация

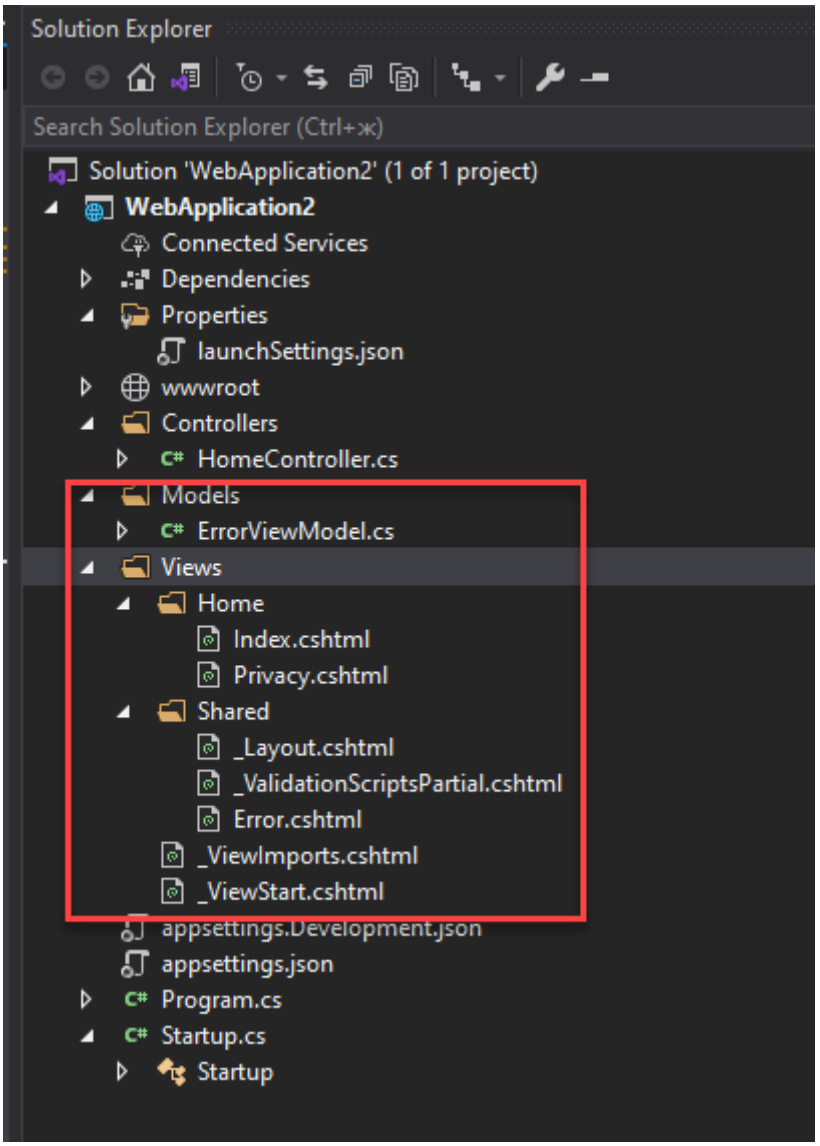
Сперва стоит создать само приложение в Visual Studio.



После, когда вы уже укажите, где сохранять его, выберите именно пункт с Model-View-Controller. Он и создаст всю основу для дальнейшей работы.



Вся эта манипуляция создаст базовый проект, который, уже будет можно изучить и начать править. Если посмотреть на структуру проекта – она не сильно отличается от того проекта, который базируется на обычном Web API сервисы. Схожее расположение файлов, схожее название директорий.



Самое главное отличие, это, конечно же, директории Views и Models. Но сначала стоит взглянуть на файл Startup.cs, а точнее как он изменился по сравнению с привычным нам Web API.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApplication2
{
	public class Startup
	{
	public Startup(IConfiguration configuration)
	{
		Configuration = configuration;
	}
```

```
      public IConfiguration Configuration { get; }

      // This method gets called by the runtime. Use this method to add
services to the container.
      public void ConfigureServices(IServiceCollection services)
      {
            services.AddControllersWithViews();
      }

      // This method gets called by the runtime. Use this method to
configure the HTTP request pipeline.
      public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
      {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
            }
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

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

Обратите внимание на метод ConfigureServices. Именно в нем идет добавление поддержки отображения страниц HTML, посредством вызова AddControllerWithViews()

Роутинг

В MVC настройка роутинга является обязательной. Обратите также внимание, как указан паттерн с некоторыми предопределенными константами - controller и action. Данная конвенция говорит, что путь в запросе делится на три сегмента. Первый - контроллер со своим значением по умолчанию, второй - методом со значением по умолчанию и третий - опциональный параметр. Данный паттерн будет применен к поступающим запросам, например http://localhost/Products/Details/42, будет разобран на части, где контроллер будет содержать значение Products акшен-значение Details, а идентификатор -

значение 42 { controller = Products, action = Details, id = 5 }. Данный парсинг позволяет определить, какой контроллер приложения будет использоваться для обработки запроса, а также какой метод контроллера будет использован и какой параметр будет передан в метод. В случае отсутствия метода, например, на запрос http://localhost/Products будет подставлен метод по умолчанию Index, а в случае отсутствия контроллера в адресе запроса http://localhost/ - по умолчанию будет подставлен контроллер Home. Знак вопроса “?” говорит, что параметр является опциональным. Символ “/” обозначает сам себя и соответствует разделителю. В итоге запрос приведет к инсценированию соответствующего контроллера и вызову его метода Index у контроллера HomeController. Запустите приложение с установленным брейкпоинтом в этом методе.

Отображение View

Нашей отправной точкой для изучения отображение страниц является контроллер HomeController. Если его открыть, то можно заметить значительные отличия от обычного контроллера с Web API.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using WebApplication2.Models;

namespace WebApplication2.Controllers
{
      public class HomeController : Controller
      {
      private readonly ILogger<HomeController> _logger;

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

      public IActionResult Index()
      {
            return View();
      }

      public IActionResult Privacy()
      {
            return View();
      }

      [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
      public IActionResult Error()
      {
```

```
            return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }
      }
}
```

Методы возвращают некий результат выполнения некого метода View(), который вернет объект типа ViewResult, в которой будет хранится как и шаблон, так и данные, то есть модель. Обратите внимание на метод отображения ошибок. В аргументах этого метода передается ViewModel, для использования данных при генерации страницы (как в WPF).

Сами страницы находятся в директории Views/{Имя контроллера}/{Имя метода}.cshtml

То есть для метода Privacy контроллера HomeController страница будет располагаться в View/Home/Privacy.cstml

Исключением является только метод Error у HomeController. Его View находится в директории с общими шаблонами – Shared.

Практическое задание

1. Создайте своё первое MVC-приложение. Модифицируйте главную страницу так, чтобы она отображала какую-либо ViewModel, к примеру - информацию о работе вымышленного офиса. Следуйте аналогии Error-страницы. Данные могут быть константными либо браться из базы данных.

Дополнительные материалы

1. Статья «Razor».

Используемые источники

1. Статья «Razor Engine».
2. Статья Razor Engine