ASP.NET Core

Что такое ASP.NET Core?

ASP.NET Core представляет технологию для создания веб-приложений на платформе .NET, развиваемую компанией Microsoft. В качестве языков программирования для разработки приложений на ASP.NET Core используются C#.

Модели разработки

- Базовый ASP.NET Core
- ASP.NET Core MVC
- Razor Pages
- ASP.NET Core Web API
- Blazor

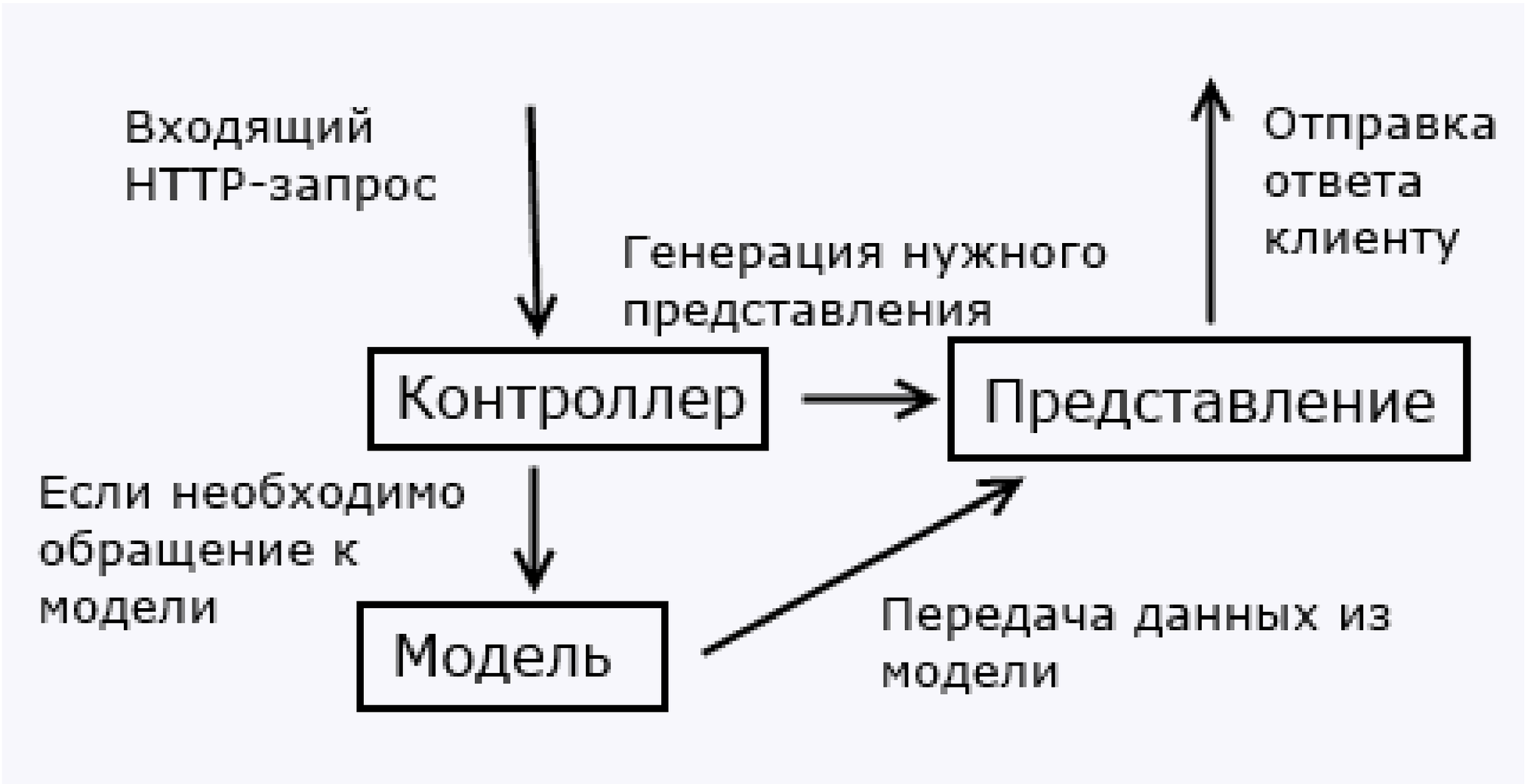
Базовый ASP.NET Core

Базовый ASP.NET Core поддерживает все основные моменты, необходимые для работы соввременного веб-приложения: маршрутизация, конфигурация, логгирования, возможность работы с различными системами баз данных и т.д. Все остальные модели разработки работаю поверх базового функционала ASP.NET Core

ASP.NET Core MVC

ASP.NET Core MVC представляет в общем виде построения приложения вокруг трех основных компонентов - Model (модели), View (представления) и Controller (контроллеры), где модели отвечают за работу с данными, контроллеры представляют логику обработки запросов, а представления определяют визуальную составляющую.

ASP.NET Core MVC



Razor Pages

Razor Pages представляет модель, при котором за обаботку запроса отвечают специальные сущности - страницы Razor Pages. Каждую отдельную такую сущность можно ассоциировать с отдельной веб-страницей.

ASP.NET Core Web API

ASP.NET Core Web API представляет реализацию паттерна REST, при котором для каждого типа http-запроса (GET, POST, PUT, DELETE) предназначен отдельный ресурс. Подобные ресурсы определяются в виде методов контроллера Web API. Данная модель особенно подходит для одностраничных приложений, но не только.

Blazor

Blazor представляет фреймворк, который позволяет создавать интерактивные приложения как на стороне сервера, так и на стороне клиента и позволяет задействовать на уровне браузера низкоуровневый код WebAssembly.

Особенности платформы

- ASP.NET Core работает поверх платформы .NET и, таким образом, позволяет задействовать весь ее функционал.
- С помощью ASP.NET Core мы можем как создавать кросс-платформенные приложения на Windows, на Linux и Mac OS, так и запускать на этих ОС.
- Благодаря модульности фреймворка все необходимые компоненты веб-приложения могут загружаться как отдельные модули через пакетный менеджер Nuget.
- Поддержка работы с большинством распространенных систем баз данных: MS SQL Server, MySQL, Postgres, MongoDB.

ASP.NET Core MVC

Фреймворк ASP.NET Core MVC

Фреймворк ASP.NET Core MVC является частью платформы ASP.NET Core, его отличительная особенность - применение паттерна MVC. Преимуществом использования фрейморка ASP.NET Core MVC по сравнению с "чистым" ASP.NET Core является то, что он упрощает в ряде ситуаций и сценариев организацию и создание приложений, особенно это относится к большим приложениям.

Концепция паттерна MVC

- Модель (model)
- Представление (view)
- Контроллер (controller)

Модель

Модель (model): описывает используемые в приложении данные, а также логику, которая связана непосредственно с данными, например, логику валидации данных. Как правило, объекты моделей хранятся в базе данных.

В MVC модели представлены двумя основными типами: модели представлений, которые используются представлениями для отображения и передачи данных, и модели домена, которые описывают логику управления данными.

Модель

Модель может содержать данные, хранить логику управления этими данными. В то же время модель не должна содержать логику взаимодействия с пользователем и не должна определять механизм обработки запроса. Кроме того, модель не должна содержать логику отображения данных в представлении.

Представление

Представление (view): отвечают за визуальную часть или пользовательский интерфейс, нередко html-страница, через который пользователь взаимодействует с приложением. Также представление может содержать логику, связанную с отображением данных. В то же время представление не должно содержать логику обработки запроса пользователя или управления данными.

Контроллер

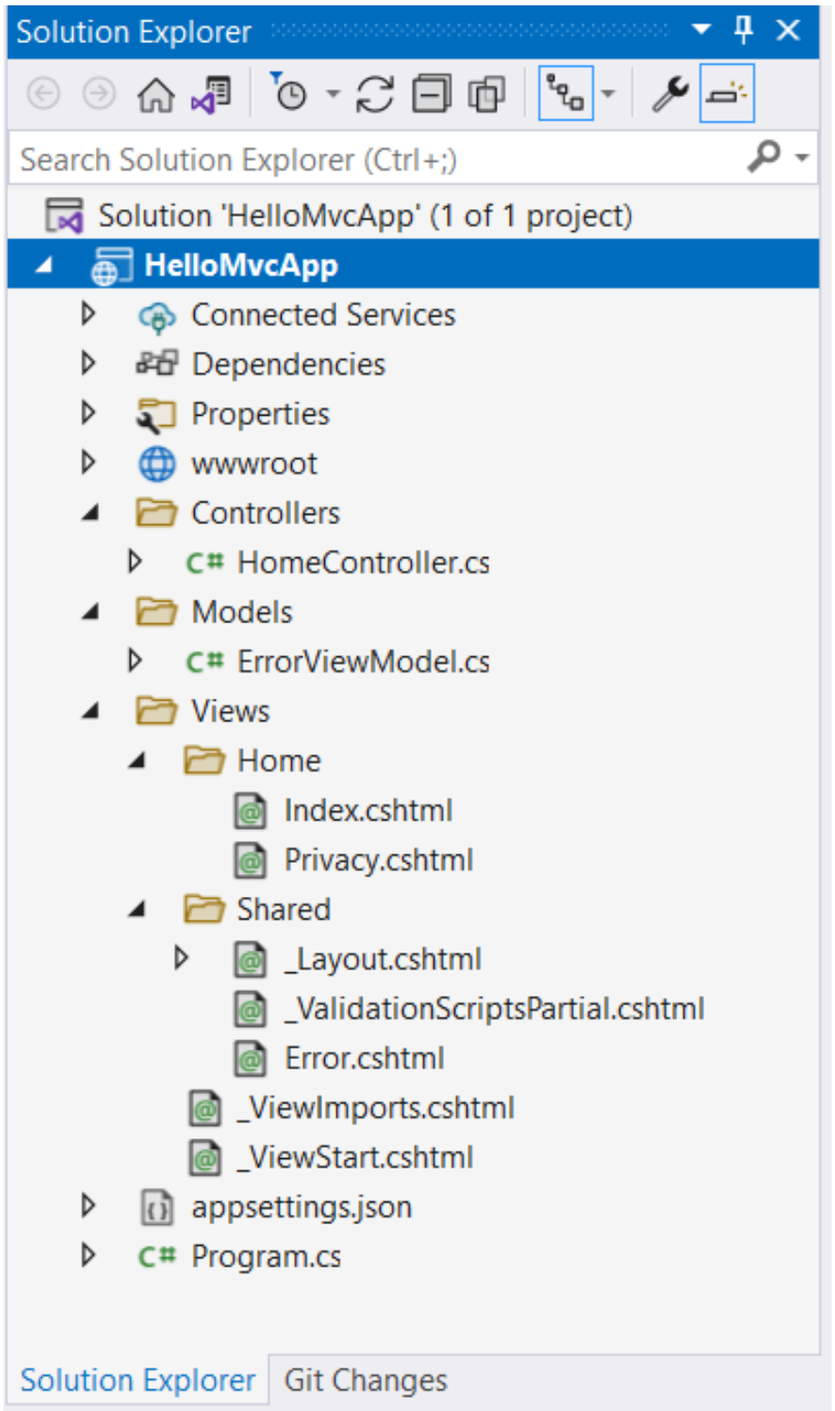
Контроллер (controller): представляет центральный компонент MVC, который обеспечивает связь между пользователем и приложением, представлением и хранилищем данных. Он содержит логику обработки запроса пользователя. Контроллер получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления, наполненного данными моделей.

Структура проекта

Структура проекта на ASP.NET Core MVC

Для создания проекта на ASP.NET Core MVC мы можем выбрать любой тип проекта на ASP.NET Core и в нем уже добавлять необходимые компоненты. Однако для упрощения .NET предоставляет для этого шаблон **ASP.NET Core MVC (Model-View-Controller).**

Структура проекта на ASP.NET Core MVC



Структура проекта на ASP.NET Core MVC

- **Dependencies:** все добавленные в проект пакеты и библиотеки.
- **wwwroot:** этот узел (на жестком диске ему соответствует одноименная папка) предназначен для хранения статических файлов - изображений, скриптов javascript, файлов css и т.д., которые используются приложением.
- **Controllers:** папка для хранения контроллеров, используемых приложением. По умолчанию здесь уже есть один контроллер – Homecontroller.
- **Models:** каталог для хранения моделей. По умолчанию здесь создается модель ErrorviewModel.

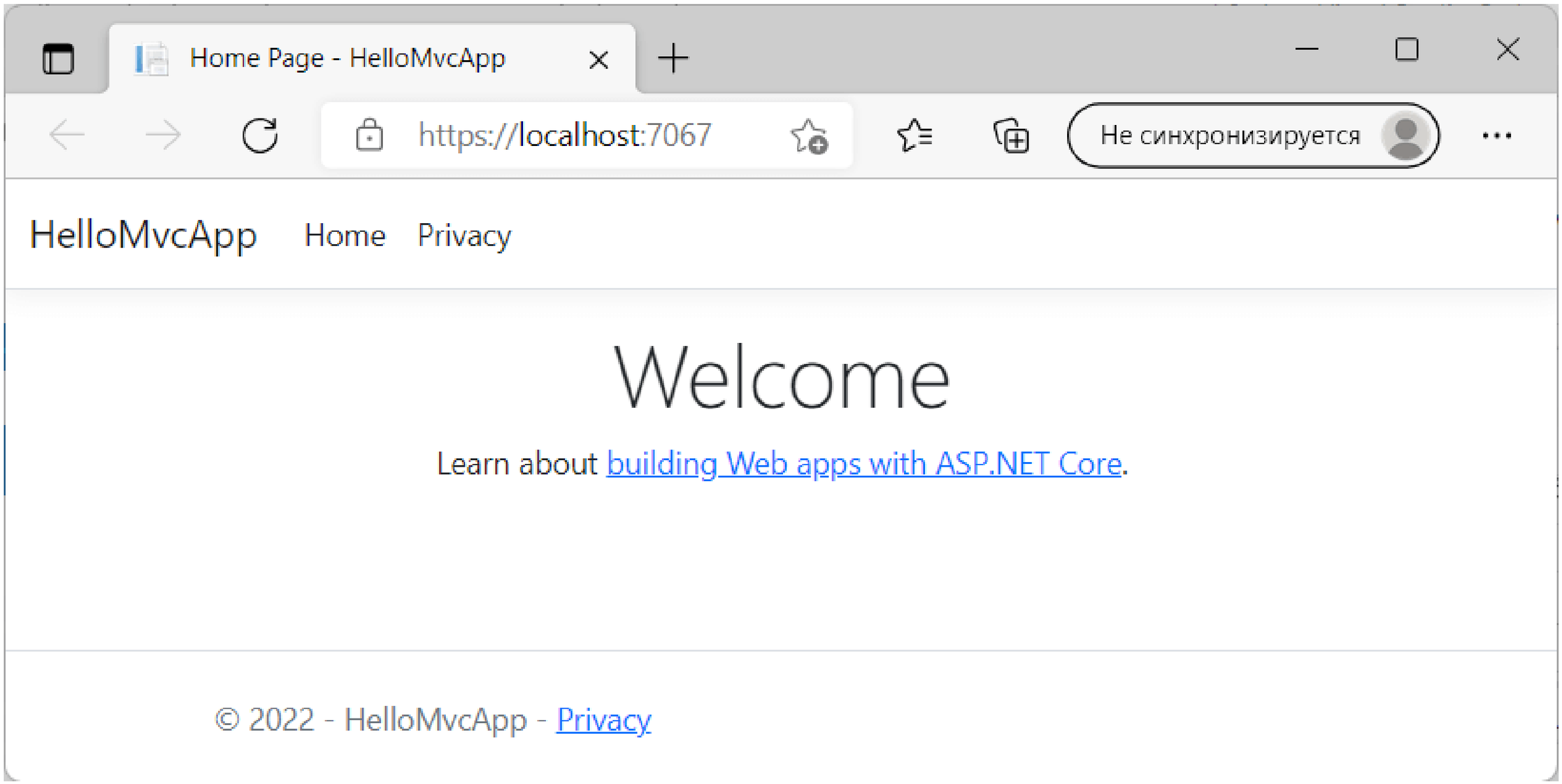
Структура проекта на ASP.NET Core MVC

- **Views:** каталог для хранения представлений. Здесь также по умолчанию добавляются ряд файлов – представлений.
- **appsettings.json:** хранит конфигурацию приложения
- **Program.cs:** файл, который определяет входную точку в приложение ASP.NET Core

Структура проекта на ASP.NET Core MVC

Фактически эта та же структура, что и у проекта по типу Empty за тем исключением, что здесь также добавлены по умолчанию папки для ключевых компонентов фреймворка MVC: контроллеров и представлений. А также есть дополнительные узлы и файлы для управления зависимостями клиентской части приложения.

Home Page


Home Page - HelloMvcApp
https://localhost:7067
Не синхронизируется
HelloMvcApp
Home
Privacy
Welcome
Learn about building Web apps with ASP.NET Core.
© 2022 - HelloMvcApp - Privacy

WebApplication

WebApplication

В центре приложения ASP.NET находится класс **WebApplication**. Например, если мы возьмем проект ASP.NET по типу ASP.NET Core Empty, то в файле **Program.cs** мы встретим следующий код:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

WebApplication

Переменная app в данном коде как раз представляет объект **WebApplication**. Однако для создания этого объекта необходим другой объект - **WebApplicationBuilder**, который в данном коде представлен переменной **builder**.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

WebApplicationBuilder

Создание приложения по умолчанию фактически начинается с класса **WebApplicationBuilder**.
Для его создания объекта этого класса вызывается статический метод **WebApplication.CreateBuilder()**:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
```

WebApplicationBuilder

Создание приложения по умолчанию фактически начинается с класса **WebApplicationBuilder**.

Для его создания объекта этого класса вызывается статический метод **WebApplication.CreateBuilder()**:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
```

WebApplicationBuilder

Для инициализации объекта WebApplicationBuilder в этот метод могут передаваться аргументы командной строки, указанные при запуске приложения (доступны через неявно определенный параметр **args**):

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

Либо можно передавать объект WebApplicationOption:

```
WebApplicationOptions options = new() { Args = args };
WebApplicationBuilder builder = WebApplication.CreateBuilder(options);
```

WebApplicationBuilder

Кроме создания объекта WebApplication класс **WebApplication Builder** выполняет еще ряд задач, среди которых можно выделить следующие:

- Установка конфигурации приложения
- Добавление сервисов
- Настройка логгирования в приложении
- Установка окружения приложения
- Конфигурация объектов IHostBuilder и IWebHostBuilder, которые применяются для создания хоста приложения

WebApplicationBuilder

Для реализации этих задач в классе WebApplicationBuilder определены следующие:

- **Configuration**: представляет объект ConfigurationManager, который применяется для добавления конфигурации к приложению.
- **Environment**: предоставляет информацию об окружении, в котором запущено приложение.
- **Host**: объект **IHostBuilder**, который применяется для настройки хоста.
- **Logging**: позволяет определить настройки логгирования в приложении.
- **Services**: представляет коллекцию сервисов и позволяет добавлять сервисы в приложение.
- **WebHost**: объект **IWebHostBuilder**, который позволяет настроить отдельные настройки сервера.

WebApplication

Метод **build()** класса WebApplicationBuilder создает объект **WebApplication**. Класс WebApplication применяется для управления обработкой запроса, установки маршрутов, получения сервисов и т.д.

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
WebApplication app = builder.Build();
```

WebApplication

Класс WebApplication применяет три интерфейса:

- **IHost:** применяется для запуска и остановки хоста, который прослушивает входящие запросы
- **IApplicationBuilder:** применяется для установки компонентов, которые участвуют в обработке запроса
- **IEndpointRouteBuilder:** применяется для установки маршрутов, которые сопоставляются с запросами

WebApplication

IEndpointRouteBuilder: применяется для установки маршрутов, которые сопоставляются с запросами:

- **Configuration:** представляет конфигурацию приложения в виде объекта **IConfiguration**
- **Environment:** представляет окружение приложения в виде **IWebHostEnvironment**
- **Lifetime:** позволяет получать уведомления о событиях жизненного цикла приложения
- **Logger**: представляет логгер приложения по умолчанию
- **Services**: представляет сервисы приложения
- **Urls**: представляет набор адресов, которые использует сервер

WebApplication

Для управления хостом класс WebApplication определяет следующие методы:

- **Run():** запускает приложение
- **RunAsync():** асинхронно запускает приложение
- **Start():** запускает приложение
- **StartAsync():** запускает приложение
- **StopAsync():** останавливает приложение

WebApplication

Таким образом, после вызова метод Run/Start/RunAsync/StartAsync приложение будет заущено, и мы сможем к нему обращаться:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
WebApplication app = builder.Build();
app.Run();
```

WebApplication

При необходимости с помощью метода **StopAsync()** можно программным способом завершить выполнение приложения:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();

WebApplication app = builder.Build();

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

await app.StartAsync();
await Task.Delay(10000);
await app.StopAsync();  // через 10 секунд завершаем выполнение приложения
```

Конвейер обработки запроса

Конвейер обработки запроса

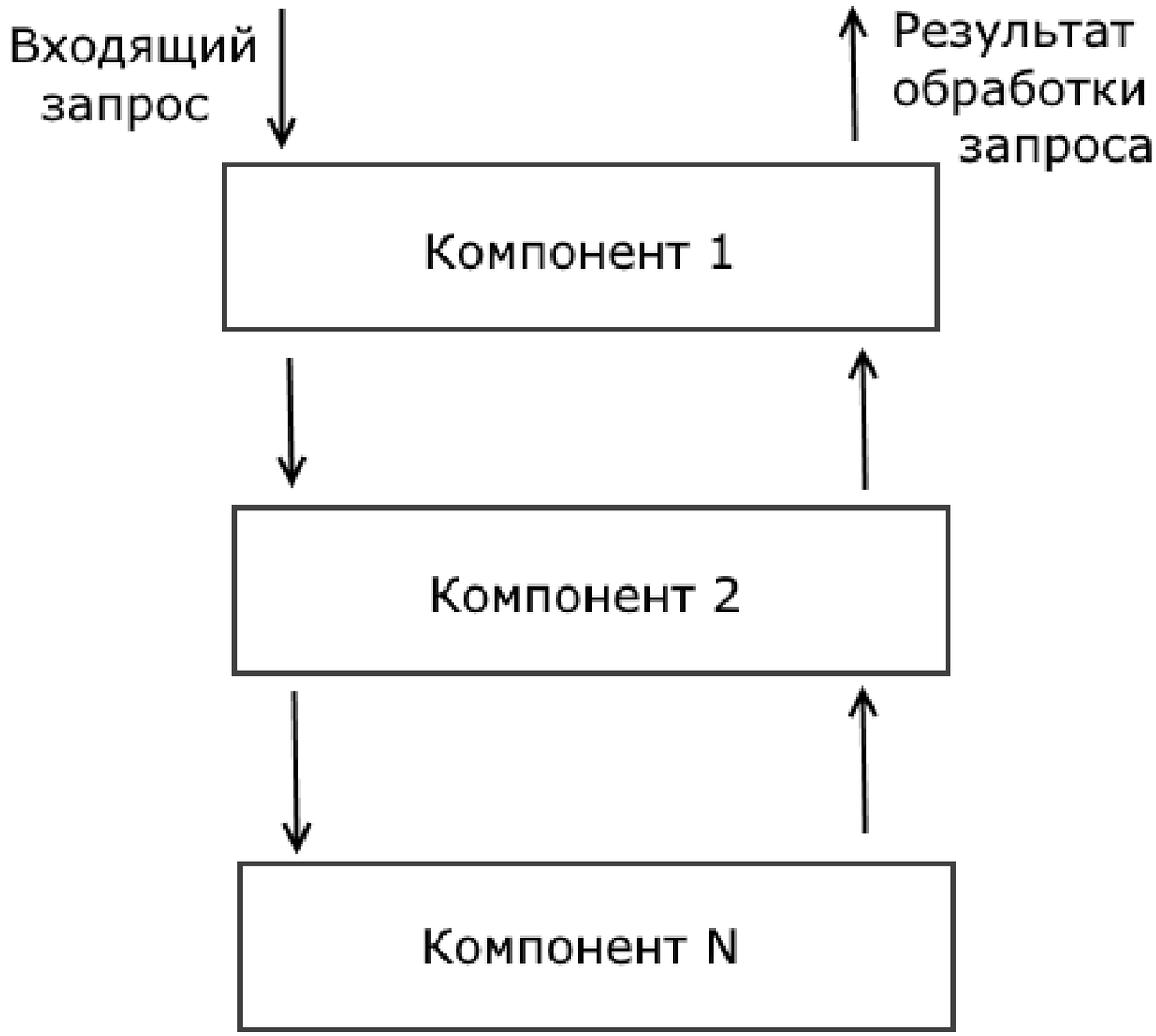
Одна из основных задач приложения - это обработка входящих запросов. Обработка запроса в ASP.NET Core устроена по принципу конвейера, который состоит из компонентов. Подобные компоненты еще называются **middleware**.

Конвейер обработки запроса

При получении запроса сначала данные запроса получает первый компонент в конвейере. После обработки запроса компонент middleware он может закончить обработку запроса - такой компонент еще называется терминальным компонентом (terminal middleware). Либо он может передать данные запроса для обработки далее по конвейеру - следующему в конвейере компоненту и так далее. После обработки запроса последним компонентом, данные запроса возвращаются к предыдущему компоненту.

Конвейер обработки запроса

Схематически это можно отобразить так:



Встроенные компоненты middleware

Стоит отметить, что ASP.NET Core уже по умолчанию предоставляет ряд встроенных компонентов middleware для часто встречающихся задач:

- **Authentication:** предоставляет поддержку аутентификации
- **Authorization:** предоставляет поддержку авторизации
- **Cookie Policy:** отслеживает согласие пользователя на хранение связанной с ним информации в куках
- **HTTP Logging:** логгирует информацию о входящих запросах и генерируемых ответах
- **HTTPS Redirection:** перенаправляет все запросы HTTP на HTTPS
- **Endpoint Routing:** предоставляет механизм маршрутизации
- И т.д.

Встроенные компоненты middleware

Для встраивания этих компонентов в конвейер обработки запроса для интерфейса IApplicationBuilder определены методы расширения типа **UseXXX**.

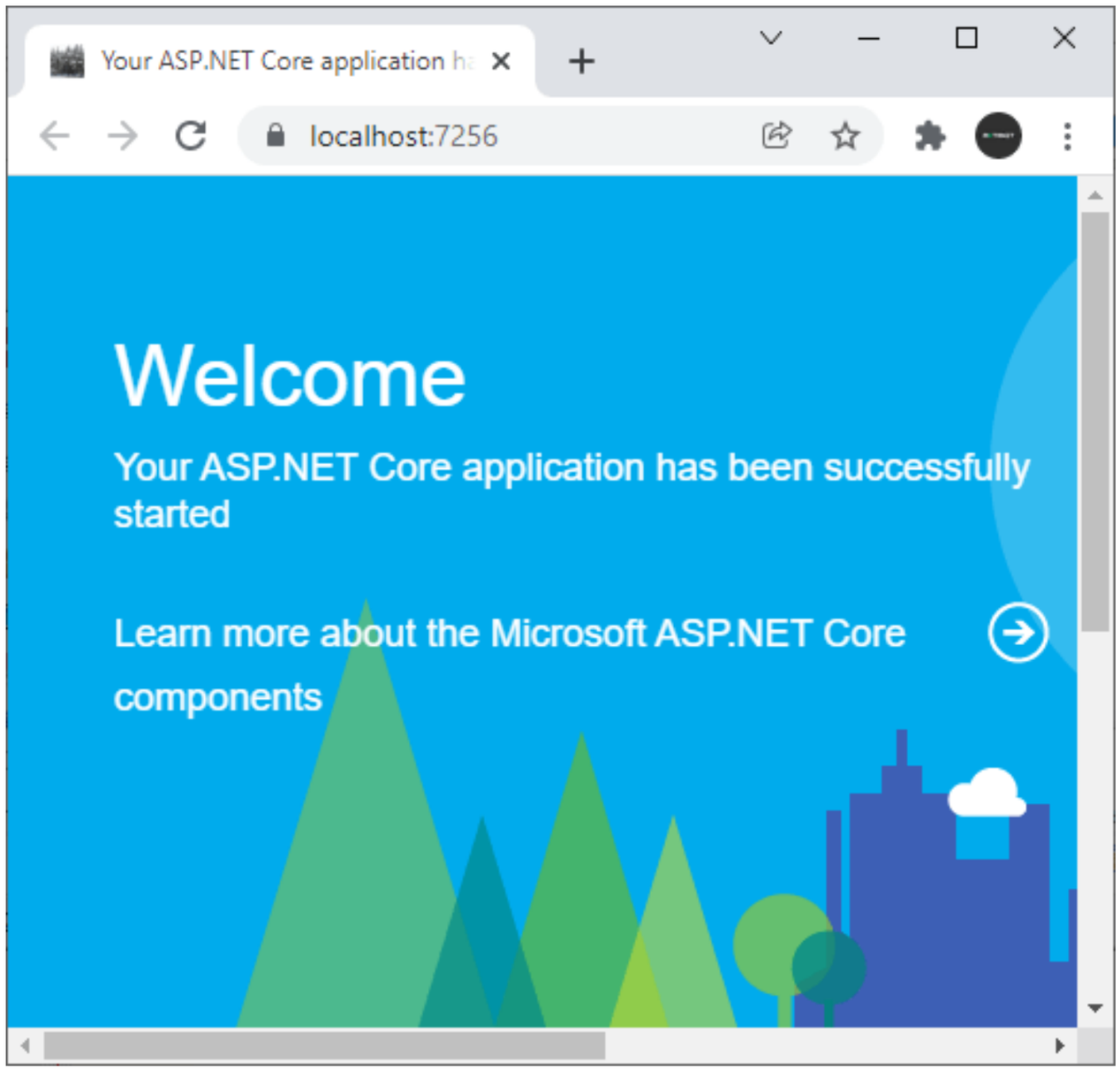
Встроенные компоненты middleware

Например, фреймворк ASP.NET Core по умолчанию предоставляет такой middleware как **WelcomePageMiddleware**, который отправляет клиенту некоторую стандартную веб-страницу. Для подключения этого компонента в конвейер запроса применяется метод расширения **UseWelcomePage**():

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.UseWelcomePage();   // подключение WelcomePageMiddleware
app.Run();
```

Встроенные компоненты middleware

И при выполнении этого приложения браузер представит нашему взору следующую красочную страницу:



Контроллеры

Контроллеры

Основным элементом в архитектуре ASP.NET Core MVC является **контроллер**. При получении запроса система маршрутизации выбирает для обработки запроса нужный контроллер и передает ему данные запроса. Контроллер обрабатывает эти данные и посылает обратно результат обработки.

Контроллеры

При использовании контроллеров существуют некоторые условности. Прежде всего обычно в проекте контроллеры помещаются в каталог Controllers.

Контроллеры

В ASP.NET Core MVC контроллер представляет обычный класс на языке C#, который обычно наследуется от абстрактного базового класса Microsoft.AspNetCore.Mvc.Controller и который, как и любой класс на языке C#, может иметь поля, свойства, методы. Согласно соглашениям об именовании названия контроллеров обычно оканчиваются на суффикс "Controller", остальная же часть до этого суффикса считается именем контроллера, например, HomeController. Но в принципе эти условности также необязательны.

Контроллеры

Но есть также и обязательные условности, которые предъявляются к контроллерам. В частности, класс контроллера должен удовлетворять как минимум одному из следующих условий.

Контроллеры

Класс контроллера имеет суффикс "Controller"

```
public class HomeController
{
    //.............
}
```

Контроллеры

Класс контроллера наследуется от класса, который имеет суффикс "Controller"

```
public class Home : Controller
{
    //.............
}
```

Контроллеры

К классу контроллера применяется атрибут [Controller]

```
[Controller]
public class Home
{
    //..................
}
```

Действия контроллера

Ключевым элементом контроллера являются его **действия**. Действия контроллера - это публичные методы, которые могут сопоставляться с запросами. Например, возьмем контроллер HomeController и определим в нем метод Index:

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Hello, world!";
        }
    }
}
```

Обращение к действиям контроллера

Сопоставление запроса с контроллером и его действием происходит благодаря системе маршрутизации. И для настройки сопоставления запросов с контроллерами перейдем к файлу **Program.cs** и изменим его следующим образом:

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

builder.Services.AddControllers();  // добавляем поддержку контроллеров

var app = builder.Build();

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

app.Run();
```

Обращение к действиям контроллера

Чтобы обратиться контроллеру из веб-браузера, нам надо в адресной строке набрать адрес_сайта/Имя_контроллера/Действие_контроллера. Так, по запросу адрес_сайта/Home/Index система маршрутизации по умолчанию вызовет метод Index контроллера HomeController для обработки входящего запроса.

Атрибут NonController

Возможно, сопоставление по умолчанию бывает не всегда удобно. Например, у нас есть класс в папке Controllers, но мы не хотим, чтобы он мог обрабатывать запрос и использоваться как контроллер. Чтобы указать, что этот класс не является контроллером, нам надо использовать над ним атрибут [NonController]:

```
[NonController]
public class Home
{
    //....................
}
```

Атрибут NonAction

Аналогично, если мы хотим, чтобы какой-либо публичный метод контроллера не рассматривался как действие, то мы можем использовать над ним атрибут **NonAction**:

```
[NonAction]
public string Hello()
{
    //..................
}
```

Атрибут ActionName

Атрибут **[ActionName]** позволяет для метода задать другое имя действия. Например:

```
[ActionName("Welcome")]
public string Hello()
{
    return "Hello ASP.NET";
}
```

В этом случае чтобы обратиться к этому методу, надо отправить запрос *localhost:xxxx/Home/Welcome*.

А запрос l*ocalhost:xxxx/Home/Hello* работать не будет.

Типы запросов

Кроме того, методы могут обслуживать разные типы запросов. Для указания типа запроса HTTP нам надо применить к методу один из атрибутов: **[HttpGet]**, **[HttpPost]**, **[HttpPut]**, **[HttpPatch]**, **[HttpDelete]** и **[HttpHead].** Если атрибут явным образом не указан, то метод может обрабатывать все типы запросов: **GET**, **POST**, **PUT**, **DELETE.**

Типы запросов

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        [HttpGet]
        public string Index() => "Hello ASP.NET";
    }
}
```

Получение данных через строку запроса

Вместе с запросом приложению могут приходить различные данные. И чтобы получить эти данные, мы можем использовать разные способы. Самым распространенным способом считается применение параметров.

Получение данных через строку запроса

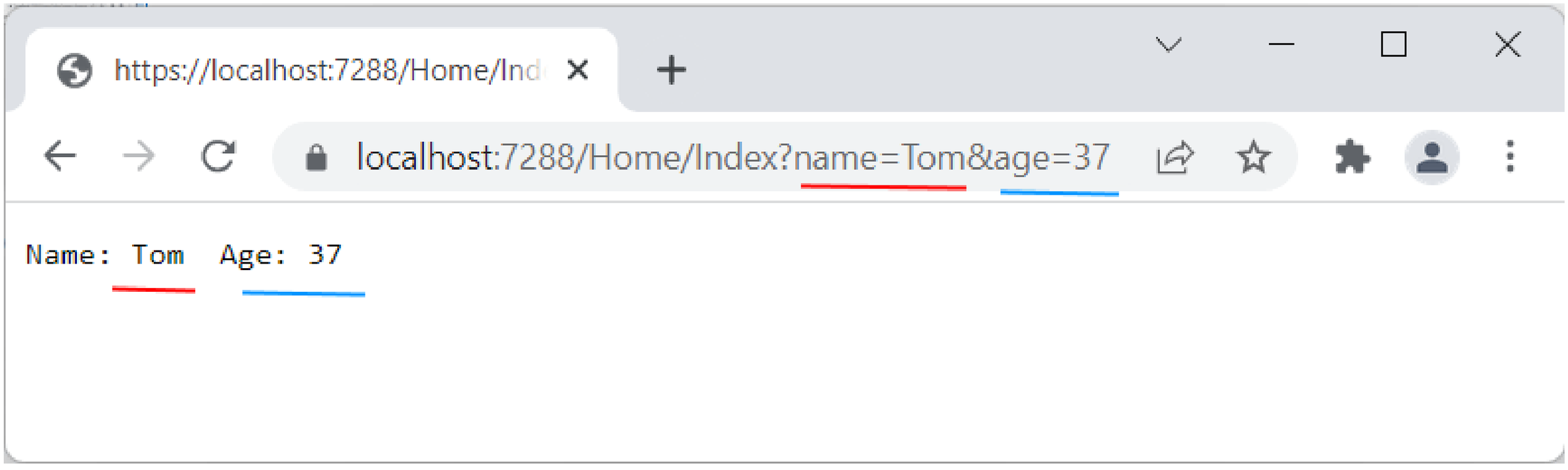
```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index(string name, int age)
        {
            return $ "Name: {name}  Age: {age}";
        }
    }
}
```

Получение данных через строку запроса

В этом случае мы можем обратиться к действию, набрав в адресной строке

https://localhost:7288/Home/Index?name=Tom&age=37.

Получение данных через строку запроса



Передача сложных объектов

Хотя строка запроса преимущественно используется для передачи данных примитивных типов, но мы также можем принимать более сложные объекты.

Передача сложных объектов

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index(Person person)
        {
            return $"Person Name: {person.Name}  Person Age: {person.Age}";
        }
    }
}
```

https://localhost:7288/Home/Index?name=Tom&age=37.

Передача массивов

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index(string[] people)
        {
            string result = "";
            foreach (var person in people)
                result = $"{result}{person}; ";
            return result;
        }
    }
}
```

https://localhost:7288/Home/Index?people=Tom&people=Bob&people=Sam

Передача массивов

```
using Microsoft.AspNetCore.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public string Index(string[] people)
        {
            string result = "";
            foreach (var person in people)
                result = $"{result}{person}; ";
            return result;
        }
    }
}
```

https://localhost:7288/Home/Index?people=Tom&people=Bob&people=Sam

Вызов представления.

```
using Microsoft.AspNetCore.Mvc;

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

КОНЕЦ