Современные веб-технологии

ASP.NET Core

- ASP.NET Core opensource-фреймворк.
- Все исходные файлы фреймворка доступны на GitHub.
- ASP.NET Core может работать поверх кросс-платформенной среды .NET Core, которая может быть развернута на основных популярных операционных системах: Windows, Mac OS X, Linux.
- Для развертывания веб-приложения можно использовать традиционный IIS, либо кросс-платформенный веб-сервер Kestrel.

ASP.NET Core

- Благодаря модульности фреймворка все необходимые компоненты веб-приложения могут загружаться как отдельные модули через пакетный менеджер Nuget.

- Кроме того, в отличие от предыдущих версий платформы нет необходимости использовать библиотеку System.Web.dll.

ASP.NET Core

- ASP.NET Core включает в себя фреймворк MVC, который объединяет функциональность MVC, Web API и Web Pages.
- Они объединены в одну программную модель ASP.NET Core MVC.

Кроме объединения вышеупомянутых технологий в одну модель в MVC был добавлен ряд дополнительных функций.

- Tag helper, которые позволяют более органично соединять синтаксис html с кодом C#.
- Упрощено управление зависимостями и конфигурирование проекта.

ASP.NET Core

- В качестве инструментария разработки используется Visual Studio. Кроме того, можно создавать приложения в среде Visual Studio Code, которая является кросс-платформенной и может работать как на Windows, так и на Mac OS X и Linux.

- Для обработки запросов теперь используется новый конвейер HTTP, который основан на компонентах Katana и спецификации OWIN. А его модульность позволяет легко добавить свои собственные компоненты.

Ключевые ASP.NET Core

- Новый легковесный и модульный конвейер HTTP-запросов.
- Возможность развертывать приложение как на IIS, так и в рамках своего собственного процесса.
- Использование платформы .NET Core и ее функциональности.
- Распространение пакетов платформы через NuGet.
- Единый стек веб-разработки, сочетающий Web UI и Web API.

Ключевые ASP.NET Core

- Конфигурация для упрощенного использования в облаке.
- Встроенная поддержка для внедрения зависимостей.
- Расширяемость.
- Кроссплатформенность: возможность разработки и развертывания приложений ASP.NET на Windows, Mac и Linux.
- Развитие как open source, открытость к изменениям.

ASP.NET Core

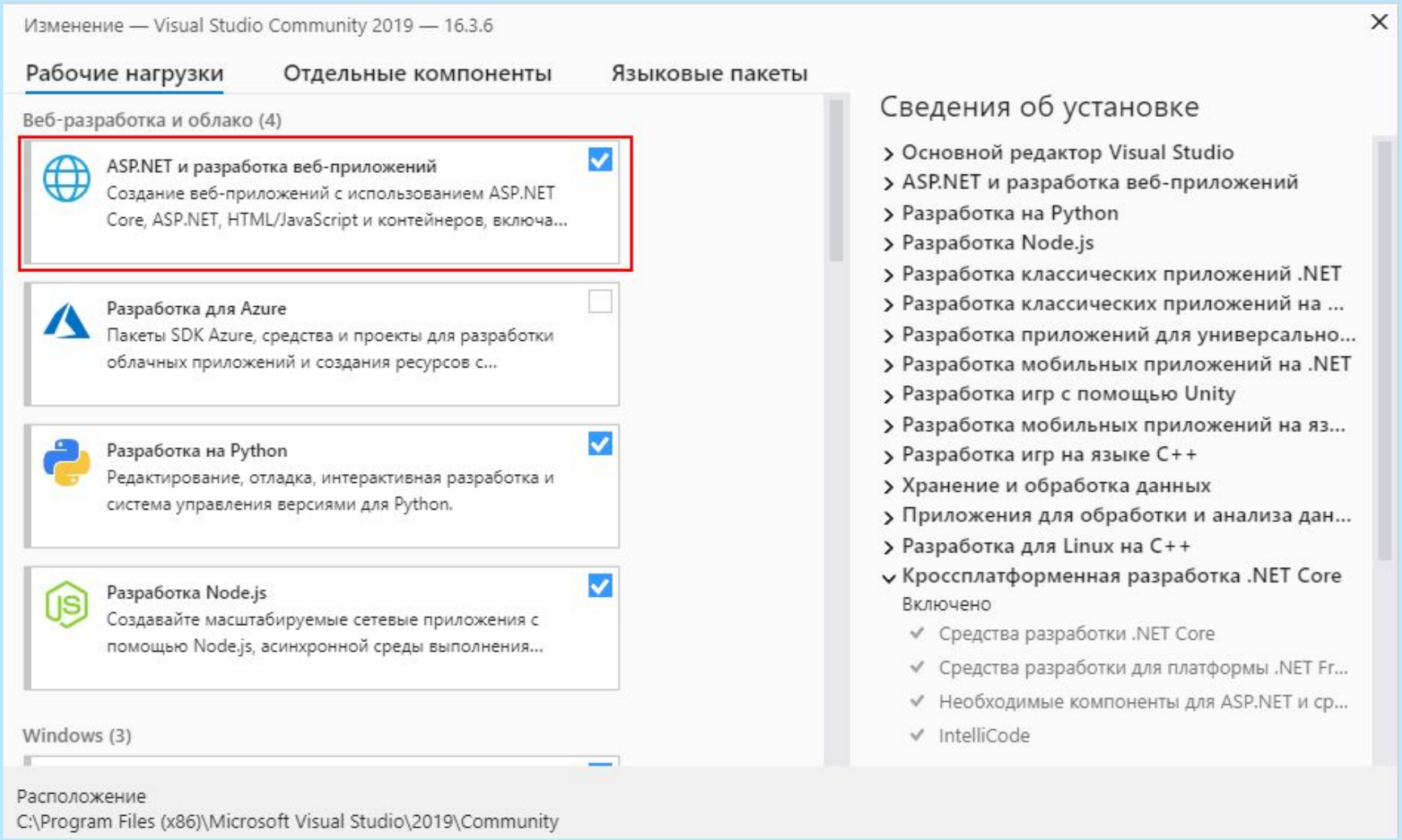
Для разработки под ASP.NET Core можно использовать различный инструментарий

Visual Studio 2019 Community

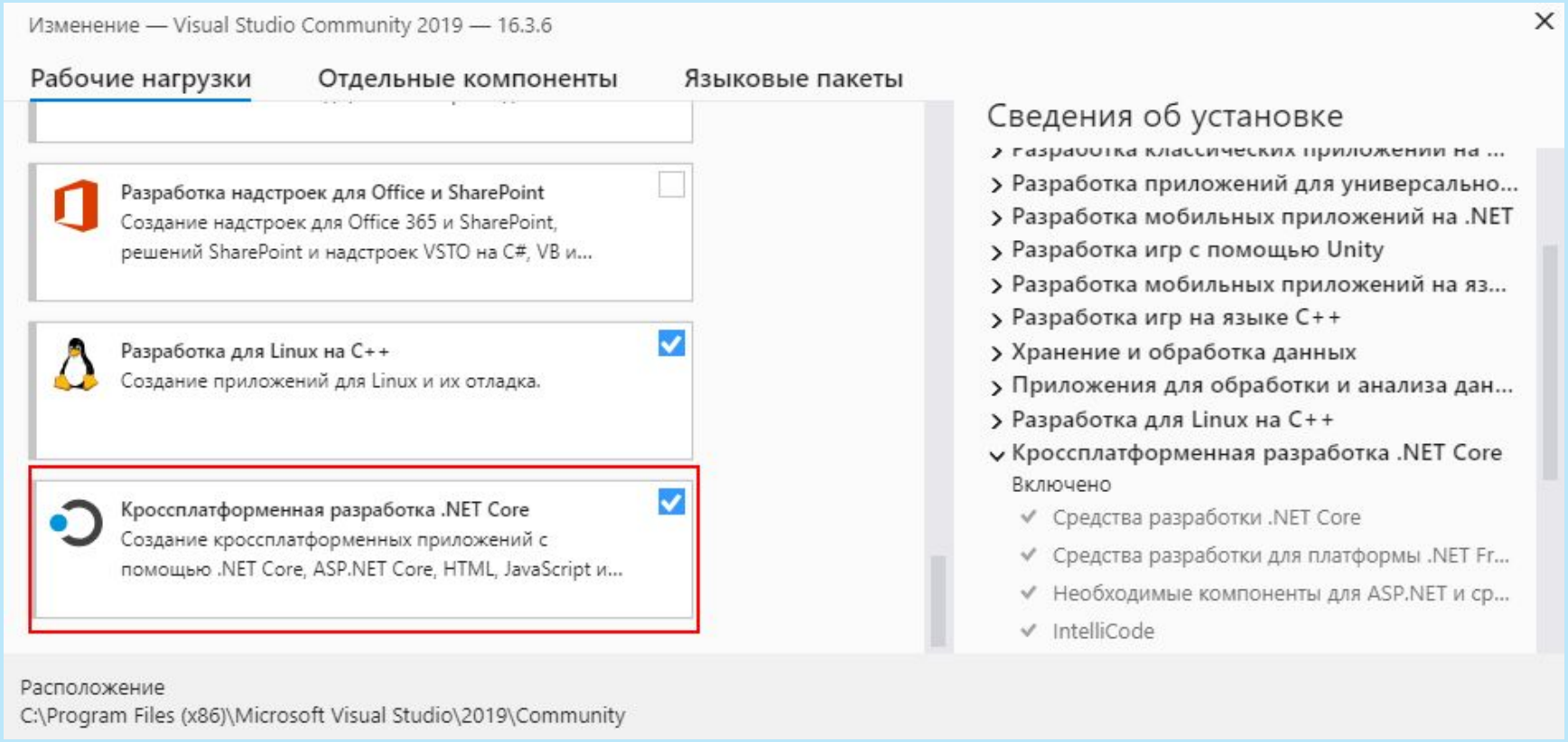
Visual Studio Code

https://visualstudio.microsoft.com/ru/downloads/

ASP.NET Core

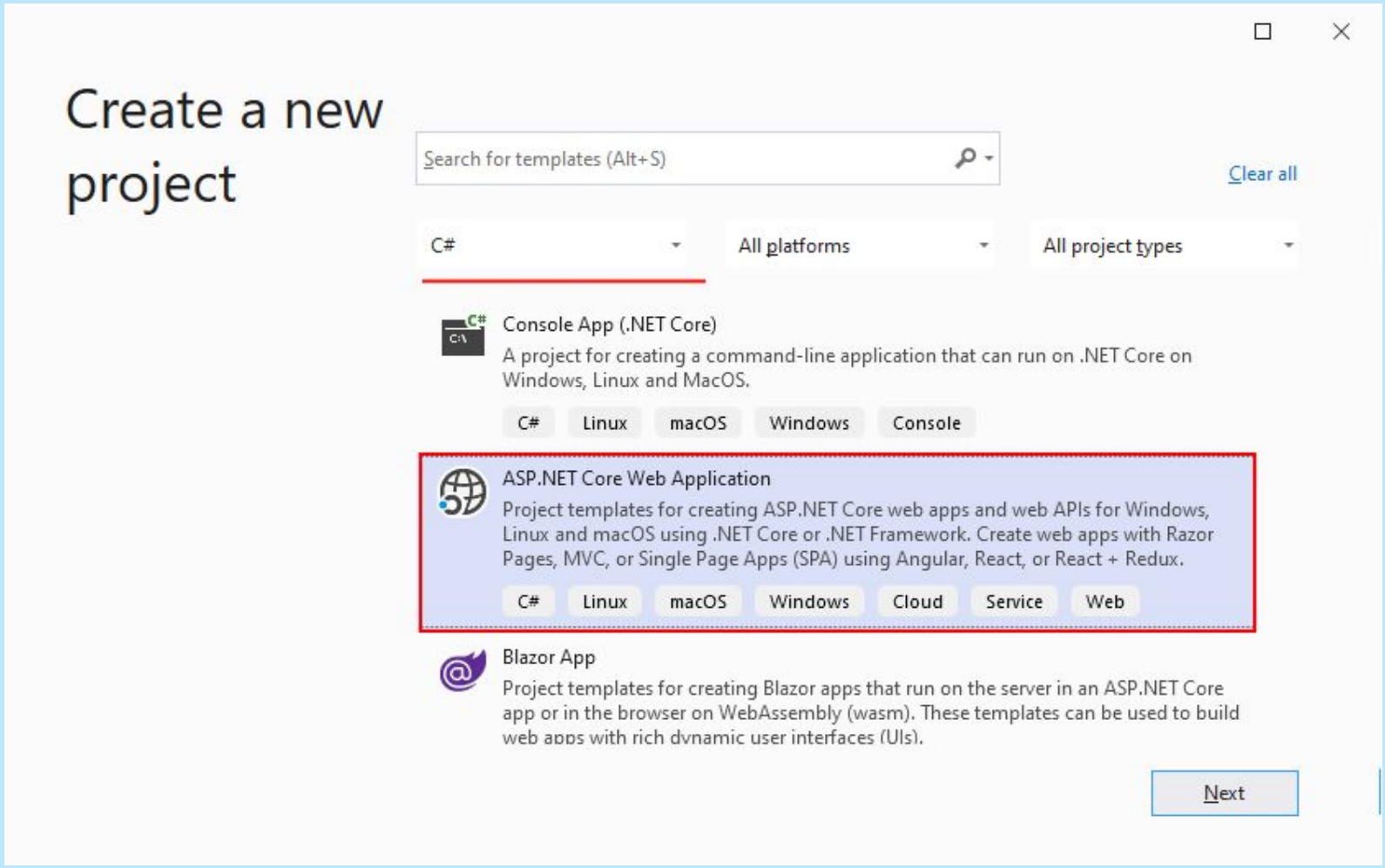

Изменение — Visual Studio Community 2019 — 16.3.6
Рабочие нагрузки
Отдельные компоненты
Языковые пакеты
Веб-разработка и облако (4)
ASP.NET и разработка веб-приложений
Создание веб-приложений с использованием ASP.NET Core, ASP.NET, HTML/JavaScript и контейнеров, включа...
Разработка для Azure
Пакеты SDK Azure, средства и проекты для разработки облачных приложений и создания ресурсов с...
Разработка на Python
Редактирование, отладка, интерактивная разработка и система управления версиями для Python.
Разработка Node.js
Создавайте масштабируемые сетевые приложения с помощью Node.js, асинхронной среды выполнения...
Windows (3)
Сведения об установке
Основной редактор Visual Studio
ASP.NET и разработка веб-приложений
Разработка на Python
Разработка Node.js
Разработка классических приложений .NET
Разработка классических приложений на ...
Разработка приложений для универсально...
Разработка мобильных приложений на .NET
Разработка игр с помощью Unity
Разработка мобильных приложений на яз...
Разработка игр на языке C++
Хранение и обработка данных
Приложения для обработки и анализа дан...
Разработка для Linux на C++
Кроссплатформенная разработка .NET Core
Включено
Средства разработки .NET Core
Средства разработки для платформы .NET Fr...
Необходимые компоненты для ASP.NET и ср...
IntelliCode
Расположение
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community

ASP.NET Core


Изменение — Visual Studio Community 2019 — 16.3.6
Рабочие нагрузки
Отдельные компоненты
Языковые пакеты
Разработка надстроек для Office и SharePoint
Создание надстроек для Office 365 и SharePoint, решений SharePoint и надстроек VSTO на C#, VB и...
Разработка для Linux на C++
Создание приложений для Linux и их отладка.
Кроссплатформенная разработка .NET Core
Создание кроссплатформенных приложений с помощью .NET Core, ASP.NET Core, HTML, JavaScript и...
Сведения об установке
Разработка приложений для универсально...
Разработка мобильных приложений на .NET
Разработка игр с помощью Unity
Разработка мобильных приложений на яз...
Разработка игр на языке C++
Хранение и обработка данных
Приложения для обработки и анализа дан...
Разработка для Linux на C++
Кроссплатформенная разработка .NET Core
Включено
Средства разработки .NET Core
Средства разработки для платформы .NET Fr...
Необходимые компоненты для ASP.NET и ср...
IntelliCode
Расположение
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community

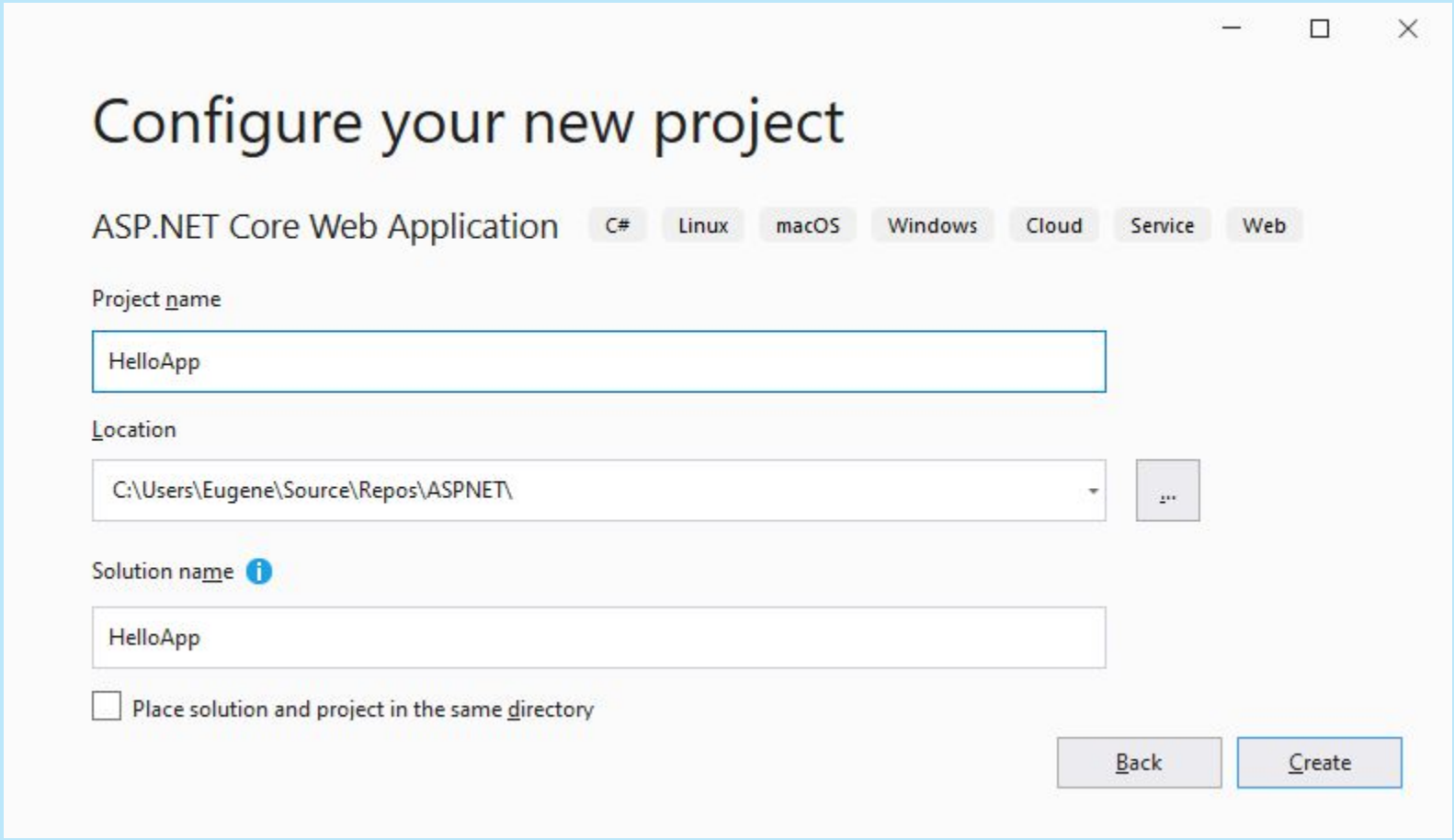
ASP.NET Core

После установки при создании проекта выберем пункт ASP.NET Core Web Application



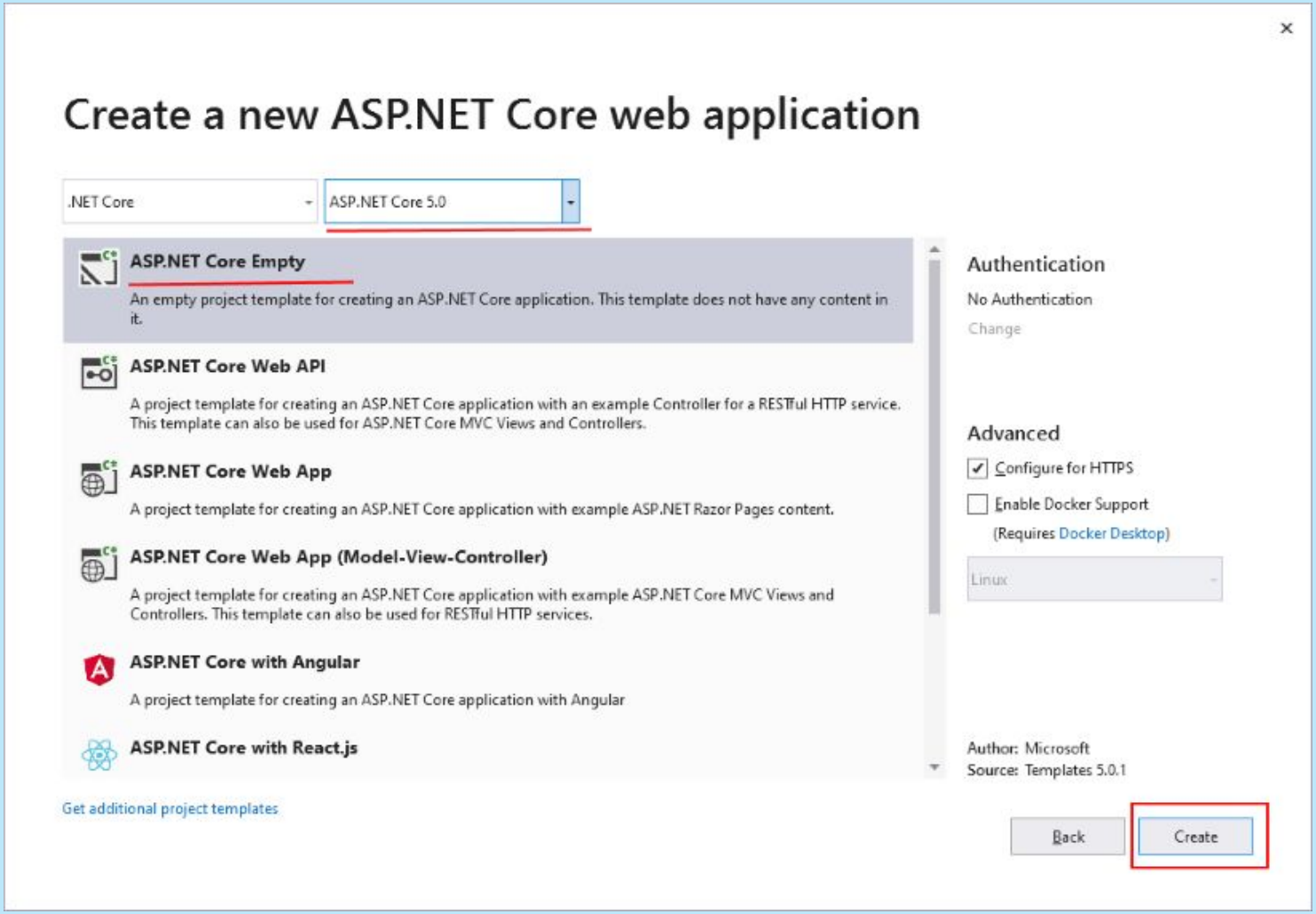
ASP.NET Core

На следующем шаге зададим имя проекта и определим для него местоположение на жестком диске:



ASP.NET Core

После этого отобразится окно выбора шаблона нового приложения:



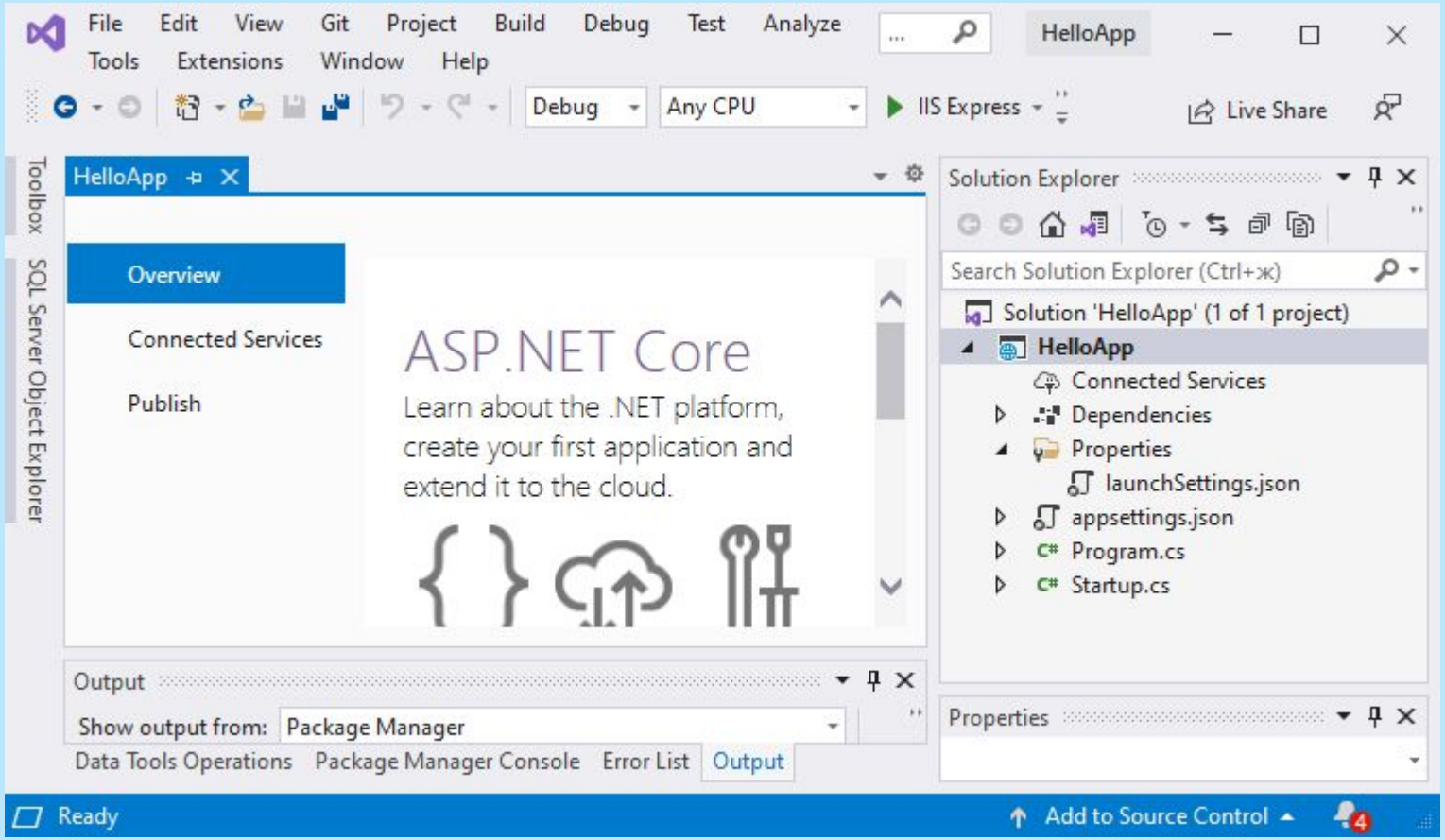
ASP.NET Core типы проектов

- ASP.NET Core Empty: пустой шаблон с самой минимальной функциональностью для создания приложений с нуля.
- ASP.NET Core Web API: проект веб-приложения, который использует архитектуру REST для создания веб-сервиса.
- ASP.NET Core Web App: проект, который для обработки запросов по умолчанию использует Razor Pages.

ASP.NET Core типы проектов

- ASP.NET Core Web App(Model-View-Controller): проект, который использует архитектуру MVC.
- ASP.NET Core with Angular: проект, предназначенный специально для работы с Angular 2+.
- ASP.NET Core with React.js: проект, который использует React.JS.
- ASP.NET Core with React.js and Redux: проект, который использует React.JS и Redux.

ASP.NET Core Empty



Структура проекта ASP.NET Core

- Connected Services: подключенные сервисы из Azure.
- Dependencies: все добавленные в проект пакеты и библиотеки, иначе говоря зависимости.
- Properties: узел, который содержит некоторые настройки проекта. В частности, в файле launchSettings.json описаны настройки запуска проекта, например, адреса, по которым будет запускаться приложение.

Структура проекта ASP.NET Core

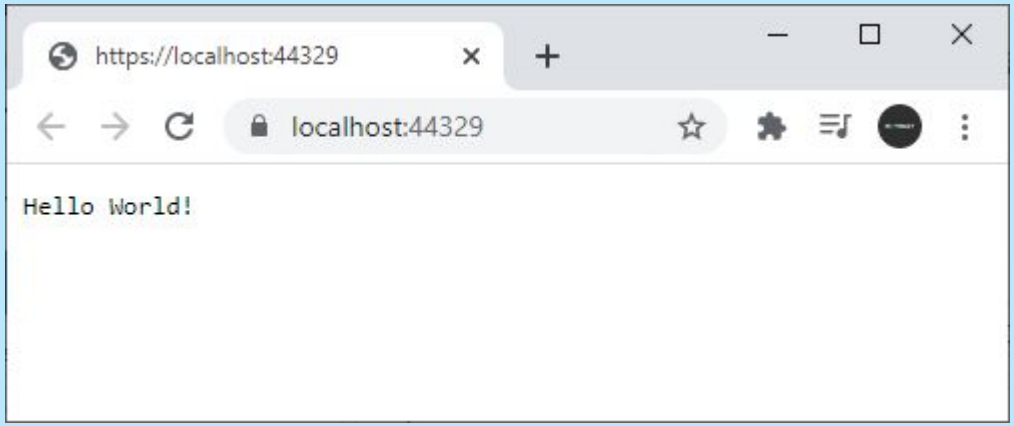
- appsettings.json: файл конфигурации проекта в формате json.

- Program.cs: главный файл приложения, с которого и начинается его выполнение. Код этого файла настривает и запускает веб-хост, в рамках которого разворачивается приложение.

- Startup.cs: файл, который определяет класс Startup и который содержит логику обработки входящих запросов.

ASP.NET Core

Данная структура, конечно, не представляет проект полнофункционального приложения. И если запустить проект, то в браузере увидим только строку "Hello World!", которая отправляется в ответ клиенту с помощью класса Startup:



Program.cs

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

namespace HelloApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args)
=>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

ASP.NET Core

Чтобы запустить приложение ASP.NET Core, необходим объект **IWebHost**, в рамках которого развертывается веб-приложение.

Для создания IWebHost применяется объект **IWebHostBuilder**.

В программе по умолчанию в статическом методе CreateWebHostBuilder как раз создается и настраивается IWebHostBuilder. Непосредственно создание IWebHostBuilder производится с помощью метода **WebHost.CreateDefaultBuilder(args)**.

Host.CreateDefaultBuilder(args)

- Устанавливает корневой каталог (для этого используется свойство Directory.GetCurrentDirectory). Корневой каталог представляет папку, где будет производиться поиск различного содержимого, например, представлений.

- Устанавливает конфигурацию хоста. Для этого загружаются переменные среды с префиксом "DOTNET_" и аргументы командной строки.

Host.CreateDefaultBuilder(args)

- Устанавливает конфигурацию приложения. Для этого загружается содержимое из файлов appsettings.json и appsettings.{Environment}.json, а также переменные среды и аргументы командной строки. Если приложение в статусе разработки, то также используются данные Secret Manager (менеджера секретов), который позволяет сохранить конфиденциальные данные, используемые при разработке.
- Добавляет провайдеры логирования.
- Если проект в статусе разработки, то также обеспечивает валидацию сервисов.

ConfigureWebHostDefaults()

- Загружает конфигурацию из переменных среды с префиксом "ASPNETCORE_".
- Запускает и настраивает веб-сервер Kestrel, в рамках которого будет разворачиваться приложение.
- Добавляет компонент Host Filtering, который позволяет настраивать адреса для веб-сервера Kestrel.
- Если переменная окружения ASPNETCORE_FORWARDEDHEADERS_ENABLED равна true, добавляет компонент Forwarded Headers, который позволяет считывать из запроса заголовки "X-Forwarded-".
- Если для работы приложения требуется IIS, то данный метод также обеспечивает интеграцию с IIS.

ASP.NET Core

В методе Main вызывается метод у созданного объекта IWebHostBuilder вызывается метод Build(), который создает хост IWebHost. А затем для непосредственного запуска у IWebHost вызывается метод Run:

```
CreateWebHostBuilder(args).Build().Run();
```

После этого приложение запущено, и веб-сервер начинает прослушивать все входящие HTTP-запросы.

ASP.NET Core

Класс **Startup** является входной точкой в приложение ASP.NET Core. Этот класс производит конфигурацию приложения, настраивает сервисы, которые приложение будет использовать, устанавливает компоненты для обработки запроса или middleware.

Метод webBuilder.UseStartup<Startup>() устанавливает класс Startup в качестве стартового. И при запуске приложения среда ASP.NET будет искать в сборке приложения класс с именем Startup и загружать его.

ASP.NET Core

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args)
=>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

ASP.NET Core

Класс Startup должен определять метод **Configure()**, и также опционально в Startup можно определить конструктор класса и метод **ConfigureServices()**.

Необязательный метод ConfigureServices() регистрирует сервисы, которые используются приложением. В качестве параметра он принимает объект IServiceCollection, который и представляет коллекцию сервисов в приложении. С помощью методов расширений этого объекта производится конфигурация приложения для использования сервисов. Все методы имеют форму Add[название_сервиса].

ASP.NET Core

В проекте по типу Empty данный метод не выполняет каких-либо действий:

```
public void ConfigureServices(IServiceCollection services)
{
}
```

ASP.NET Core

Метод Configure устанавливает, как приложение будет обрабатывать запрос.

Этот метод является обязательным. Для установки компонентов, которые обрабатывают запрос, используются методы объекта **IApplicationBuilder**.

Объект IApplicationBuilder является обязательным параметром для метода Configure.

ASP.NET Core

Кроме того, метод нередко принимает еще один необязательный параметр - объект **IWebHostEnvironment**, который позволяет получить информацию о среде, в которой запускается приложение, и взаимодействовать с ней.

Но в принципе в метод Configure в качестве параметра может передаваться любой сервис, который зарегистрирован в методе ConfigureServices или который регистрируется для приложения по умолчанию (например, IWebHostEnvironment).

Метод Configure() в проекте по типу Empty

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // если приложение в процессе разработки
    if (env.IsDevelopment())
    {
        // то выводим информацию об ошибке, при наличии ошибки
        app.UseDeveloperExceptionPage();
    }
    // добавляем возможности маршрутизации
    app.UseRouting();

    // устанавливаем адреса, которые будут обрабатываться
    app.UseEndpoints(endpoints =>
    {
        // обработка запроса - получаем констекст запроса в виде объекта
context
        endpoints.MapGet("/", async context =>
        {
            // отправка ответа в виде строки "Hello World!"
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

Конструктор Startup

Конструктор является необязательной частью класса Startup. В конструкторе, как правило, производится начальная конфигурация приложения.

Если мы создаем проект ASP.NET Core по типу Empty, то класс Startup в таком проекте по умолчанию не содержит конструктор. Но при необходимости мы можем его определить.

Конвейер обработки запроса и middleware

Обработка запроса в ASP.NET Core устроена по принципу конвейера.

Сначала данные запроса получает первый компонент в конвейере.

После обработки он передает данные HTTP-запроса второму компоненту и так далее.

Эти компоненты конвейера, которые отвечают за обработку запроса, называются middleware.

В ASP.NET Core для подключения компонентов middleware используется метод Configure из класса Startup.

Конвейер обработки запроса и middleware

Компонент middleware может либо передать запрос далее следующему в конвейере компоненту, либо выполнить обработку и закончить работу конвейера.

Также компонент middleware в конвейере может выполнять обработку запроса как до, так и после следующего в конвейере компонента.

Конвейер обработки запроса и middleware

Компоненты middleware конфигурируются с помощью методов расширений **Run**, **Map** и **Use** объекта IApplicationBuilder, который передается в метод Configure() класса Startup. Каждый компонент может быть определен как анонимный метод (встроенный inline компонент), либо может быть вынесен в отдельный класс.

Метод Configure из класса Startup

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
    // если проект в процессе разработки
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

ASP.NET Core

Все вызовы типа app.UseXXX как раз и представляют собой добавление компонентов middleware для обработки запроса. То есть у нас получается примерно следующий конвейер обработки:

- Компонент обработки ошибок - Diagnostics. Добавляется через app.UseDeveloperExceptionPage()
- Компонент маршрутизации - EndpointRoutingMiddleware. Добавляется через app.UseRouting()
- Компонент EndpointMiddleware, который отправляет ответ, если запрос пришел по маршруту "/" (то есть пользователь обратился к корню веб-приложения). Добавляется через метод app.UseEndpoints()

Встроенные компоненты middleware

- Authentication: предоставляет поддержку аутентификации.
- Cookie Policy: отслеживает согласие пользователя на хранение связанной с ним информации в куках.
- CORS: обеспечивает поддержку кроссдоменных запросов.
- Diagnostics: предоставляет страницы статусных кодов, функционал обработки исключений, страницу исключений разработчика.

Встроенные компоненты middleware

- Forwarded Headers: перенаправляет заголовки запроса.
- Health Check: проверяет работоспособность приложения asp.net core.
- HTTP Method Override: позволяет входящему POST-запросу переопределить метод.
- HTTPS Redirection: перенаправляет все запросы HTTP на HTTPS.

Встроенные компоненты middleware

- HTTP Strict Transport Security (HSTS): для улучшения безопасности приложения добавляет специальный заголовок ответа.
- MVC: обеспечивает функционал фреймворка MVC.
- Request Localization: обеспечивает поддержку локализации.
- Response Caching: позволяет кэшировать результаты запросов.
- Response Compression: обеспечивает сжатие ответа клиенту.

Встроенные компоненты middleware

- URL Rewrite: предоставляет функциональность URL Rewriting.
- Endpoint Routing: предоставляет механизм маршрутизации.
- Session: предоставляет поддержку сессий.
- Static Files: предоставляет поддержку обработки статических файлов.
- WebSockets: добавляет поддержку протокола WebSockets.

Жизненный цикл middleware

Метод Configure выполняется один раз при создании объекта класса Startup, и компоненты middleware создаются один раз и живут в течение всего жизненного цикла приложения.

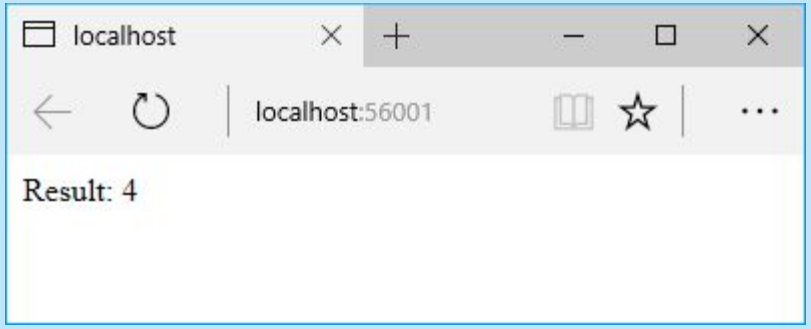
То есть для последующей обработки запросов используются одни и те же компоненты.

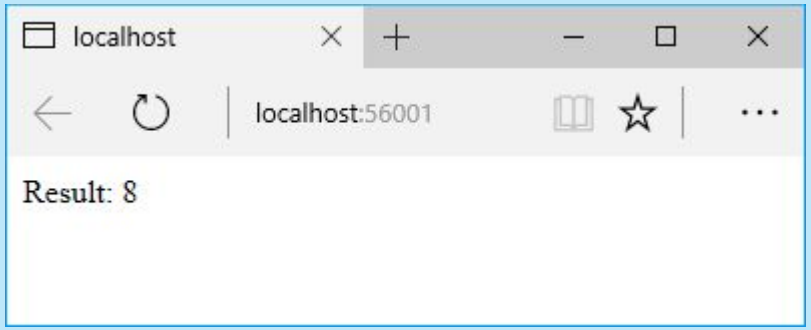
ASP.NET Core

```
public class Startup
{
    public void ConfigureServices(IServiceCollection
services)
    {
    }

    public void Configure(IApplicationBuilder app)
    {
        int x = 2;
        app.Run(async (context) =>
        {
            x = x * 2;  //  2 * 2 = 4
            await context.Response.WriteAsync($"Result:
{x}");
        });
    }
}
```

ASP.NET Core


localhost
localhost:56001
Result: 4


localhost
localhost:56001
Result: 8

Класс Startup

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello
World!");
        });
    }
}
```

ASP.NET Core

Метод **Run** представляет собой простейший способ для добавления компонентов middleware в конвейер. Однако компоненты, определенные через метод **Run**, не вызывают никакие другие компоненты и дальше обработку запроса не передают.

В качестве параметра метод **Run** принимает делегат RequestDelegate.

ASP.NET Core

Этот делегат имеет следующее определение:

```
public delegate Task RequestDelegate(HttpContext context);
```

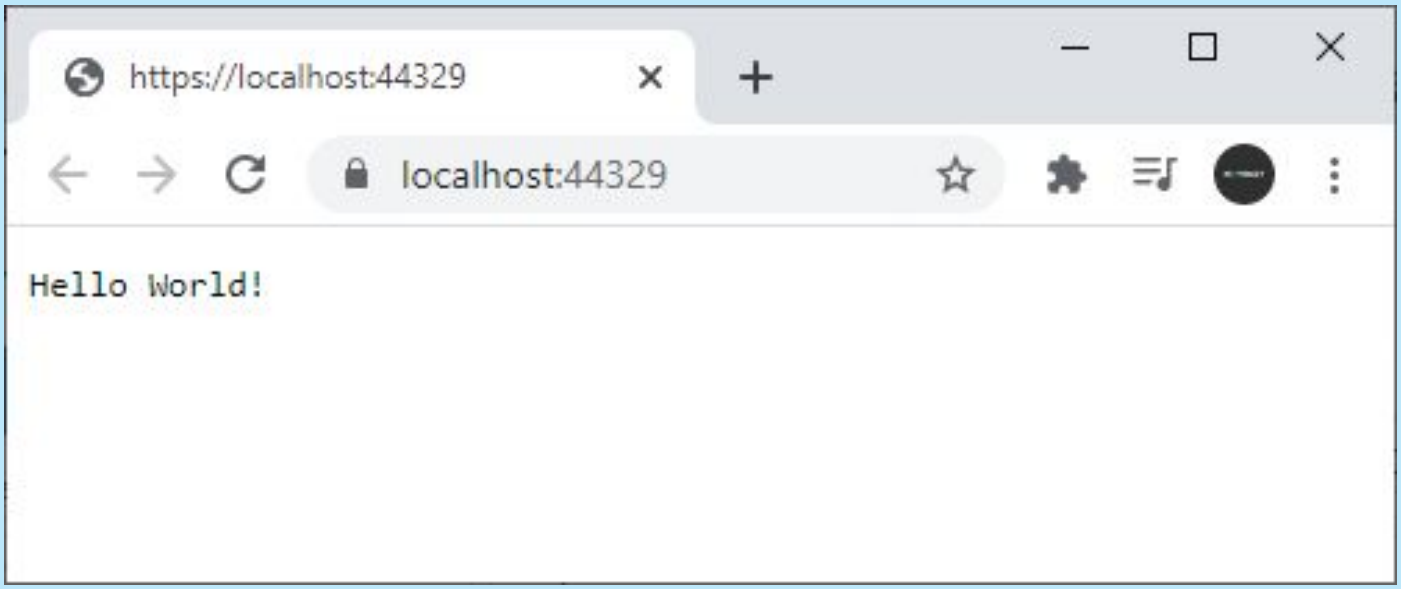
Он принимает в качестве параметра контекст запроса HttpContext и возвращает объект Task. Поэтому в методе **Run** делегат в качестве параметра context принимает контекст запроса - объект HttpContext.

ASP.NET Core

Данный метод определяет один единственный делегат запроса, который обрабатывает все запросы к приложению. Суть этого делегата заключается в отправке в ответ на запросы сообщения "Hello World!". Причем так как данный метод не передает обработку запроса далее по конвейеру, то его следует помещать в самом конце. До него же могут быть помещены другие методы.

ASP.NET Core

В итоге при запуске проекта в браузере увидим приветствие:



Метод **Use** также добавляет компоненты middleware, которые также обрабатывают запрос, но в нем может быть вызван следующий в конвейере запроса компонент middleware. Например, изменим метод Configure() следующим образом:

```
public void Configure(IApplicationBuilder app)
{
    int x = 5;
    int y = 8;
    int z = 0;
    app.Use(async (context, next) =>
    {
        z = x * y;
        await next.Invoke();
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync($"x * y = {z}");
    });
}
```

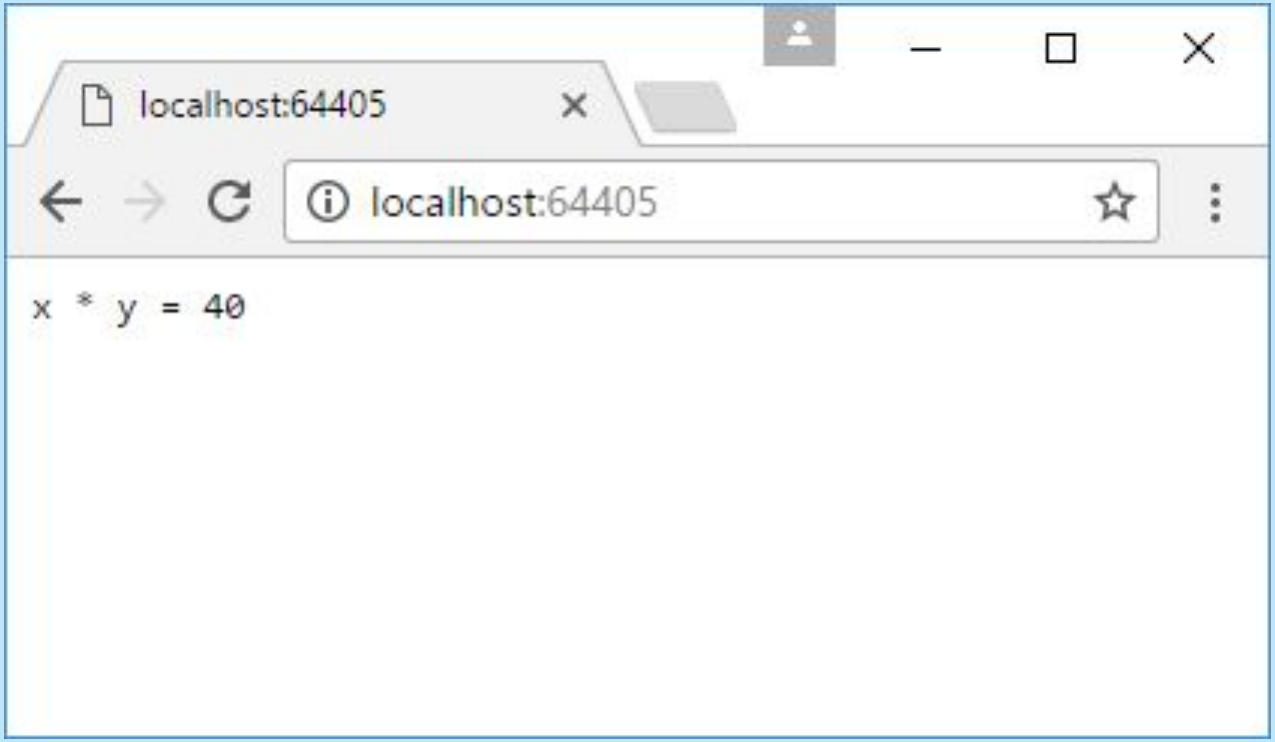
ASP.NET Core

В данном случае используем перегрузку метода **Use**, которая в качестве параметров принимает контекст запроса - объект HttpContext и делегат Func<Task>, который представляет собой ссылку на следующий в конвейере компонент middleware.

Метод app.Use реализует простейшую задачу - умножение двух чисел и затем передает обработку запроса следующим компонентам middleware в конвейере.

То есть при вызове await next.Invoke() обработка запроса перейдет к тому компоненту, который установлен в методе app.Run().

ASP.NET Core



Выполнение app.Use

```
public void Configure(IApplicationBuilder app)
{
    int x = 2;
    app.Use(async (context, next) =>
    {
        x = x * 2;       // 2 * 2 = 4
        await next.Invoke();    // вызов app.Run
        x = x * 2;       // 8 * 2 = 16
        await context.Response.WriteAsync($"Result:
{x}");
    });

    app.Run(async (context) =>
    {
        x = x * 2;  //  4 * 2 = 8
        await Task.FromResult(0);
    });
}
```

Выполнение app.Use

Если компоненты middleware в app.Use использует вызов next.Invoke() для передачи обработки дальше по конвейеру, то выполнение такого компонента фактически делится на две части: до next.Invoke() и после next.Invoke().

Здесь определена переменная x, которая равна 2.

Последующие вызовы компонентов middleware увеличивают ее значение в два раза.

Выполнение app.Use

1. Вызов компонента app.Use.

2. Увеличение переменной x в два раза: x = x * 2;. Теперь x равно 4.

3. Вызов await next.Invoke(). Управление переходит следующему компоненту в конвейере - к app.Run.

4. Увеличение переменной x в два раза: x = x * 2;. Теперь x равно 8.

5. Метод app.Run закончил свою работу, и управление обработкой возвращается к app.Use. Начинает выполняться та часть кода, которая идет после await next.Invoke().

6. Увеличение переменной x в два раза: x = x * 2;. Теперь x равно 16.

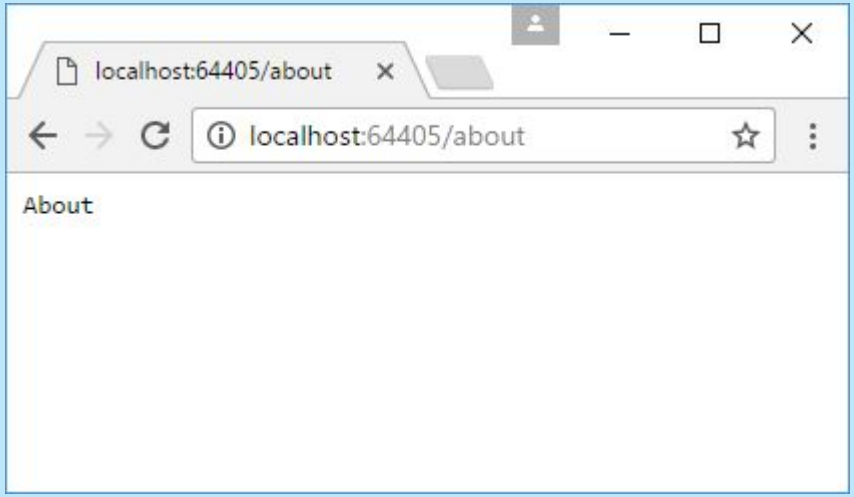
7. Отправка ответа клиенту с помощью вызова

await context.Response.WriteAsync($"Result: {x}").

Метод Map

Метод **Map** (и методы расширения MapXXX()) применяется для сопоставления пути запроса с определенным делегатом, который будет обрабатывать запрос по этому пути.

Теперь обращения к приложению типа http://localhost:xxxx/about будут обрабатываться с помощью метода About, а запросы типа http://localhost:xxxx/index - методом Index. А все остальные запросы будут обрабатываться делегатом из app.Run().



Метод Map

```
public void Configure(IApplicationBuilder app)
{
    app.Map("/index", Index);
    app.Map("/about", About);

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Page Not
Found");
    });
}

private static void Index(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Index");
    });
}
private static void About(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("About");
    });
}
```

Вложенные методы Map

```
public void Configure(IApplicationBuilder app)
{
    app.Map("/home", home =>
    {
        home.Map("/index", Index);
        home.Map("/about", About);
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Page Not
Found");
    });
}

private static void Index(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Index");
    });
}
private static void About(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("About");
```

Вложенные методы Map

Метод Map может иметь вложенные методы Map, которые обрабатывают подмаршруты.

Теперь метод About будет обрабатывать запрос не *http://localhost:xxxx/about*, а *http://localhost:xxxx/home/about*

Конвейер обработки запроса

- Как правило, для обработки запроса применяется не один, а несколько компонентов middleware. И в этом случае большую роль может играть порядок их помещения в конвейер обработки запроса, а также то, как они взаимодействуют с другими компонентами.

- Кроме того, каждый компонент middleware может обрабатывать запрос до и после последующих в конвейере компонентов. Данное обстоятельство позволяет предыдущим компонентам корректировать результат обработки последующих компонентов.

Конвейер обработки запроса

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace HelloApp
{
    public class RoutingMiddleware
    {
        private readonly RequestDelegate _next;
        public RoutingMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            string path = context.Request.Path.Value.ToLower();
            if (path == "/index")
            {
                await context.Response.WriteAsync("Home Page");
            }
            else if (path == "/about")
            {
                await context.Response.WriteAsync("About");
            }
            else
            {
                context.Response.StatusCode = 404;
            }
            //await _next.Invoke(context);
        }
    }
}
```

Конвейер обработки запроса

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace HelloApp
{
    public class AuthenticationMiddleware
    {
        private RequestDelegate _next;
        public AuthenticationMiddleware(RequestDelegate
next)
        {
            _next = next;
        }
        public async Task InvokeAsync(HttpContext context)
        {
            var token = context.Request.Query["token"];
            if (string.IsNullOrWhiteSpace(token))
            {
                context.Response.StatusCode = 403;
            }
            else
            {
                await _next.Invoke(context);
            }
        }
    }
}
```

ASP.NET Core

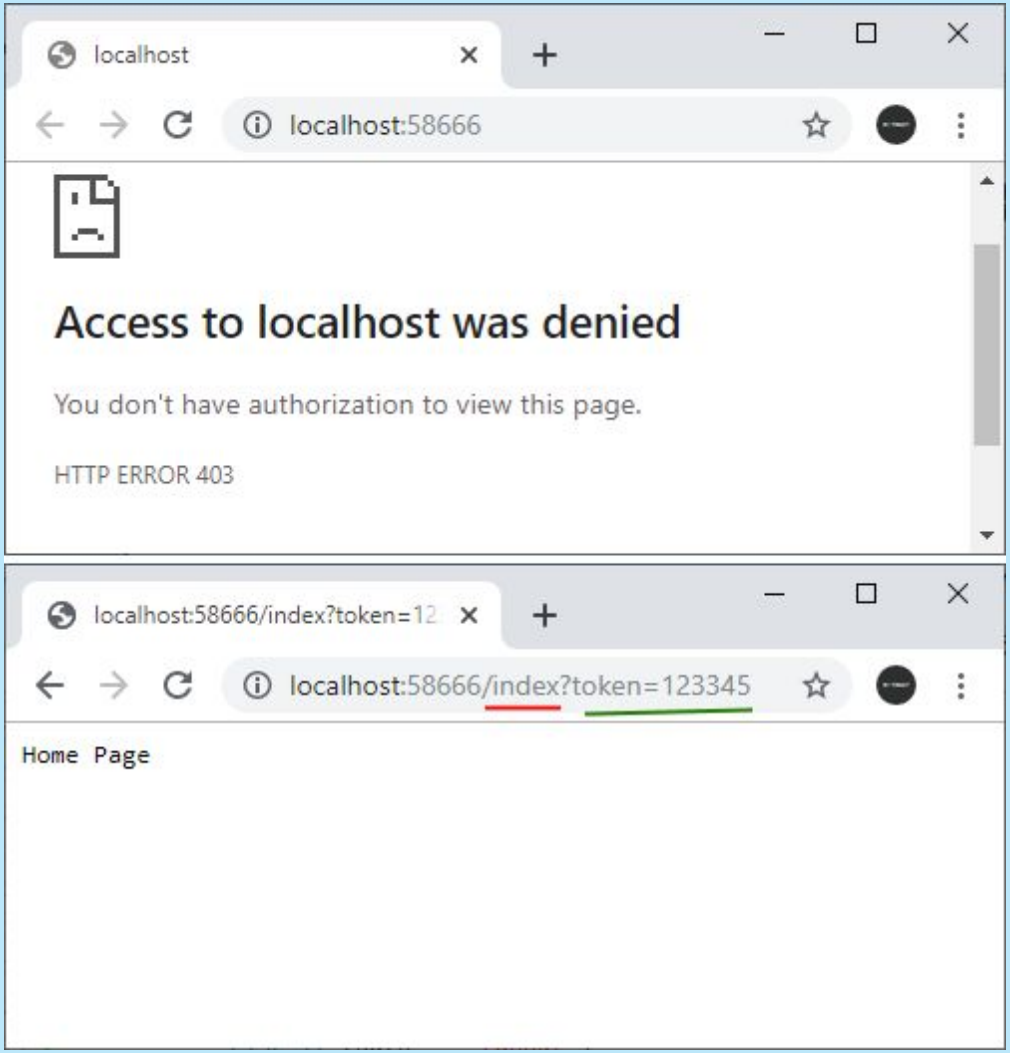
Условно будем считать, что если в строке запроса есть параметр token и он имеет какое-нибудь значение, то пользователь аутентифицирован. А если он не аутентифицирован, то надо необходимо ограничить доступ пользователям к приложению. Если пользователь не аутентифицирован, то устанавливаем статусный код 403, иначе передаем выполнение запроса следующему в конвейере делегату.

Поскольку компоненту RoutingMiddleware нет смысла обрабатывать запрос, если пользователь не аутентифицирован, то в конвейере компонент AuthenticationMiddleware должен быть помещен перед компонентом RoutingMiddleware:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseMiddleware<AuthenticationMiddleware>();
        app.UseMiddleware<RoutingMiddleware>();
    }
}
```

Таким образом, если мы сейчас запустим проект и обратимся по пути /index или /about и не передадим параметр token, то мы получим ошибку. Если же обратимся по пути /index или /about и передадим значение параметра token, то увидим искомый текст.

ASP.NET Core


localhost
localhost:58666
Access to localhost was denied
You don't have authorization to view this page.
HTTP ERROR 403
localhost:58666/index?token=12
localhost:58666/index?token=123345
Home Page

ASP.NET Core

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace HelloApp
{
    public class ErrorHandlingMiddleware
    {
        private RequestDelegate _next;
        public ErrorHandlingMiddleware(RequestDelegate next)
        {
            _next = next;
        }
        public async Task InvokeAsync(HttpContext context)
        {
            await _next.Invoke(context);
            if (context.Response.StatusCode == 403)
            {
                await context.Response.WriteAsync("Access
Denied");
            }
            else if (context.Response.StatusCode == 404)
            {
                await context.Response.WriteAsync("Not Found");
            }
        }
    }
}
```

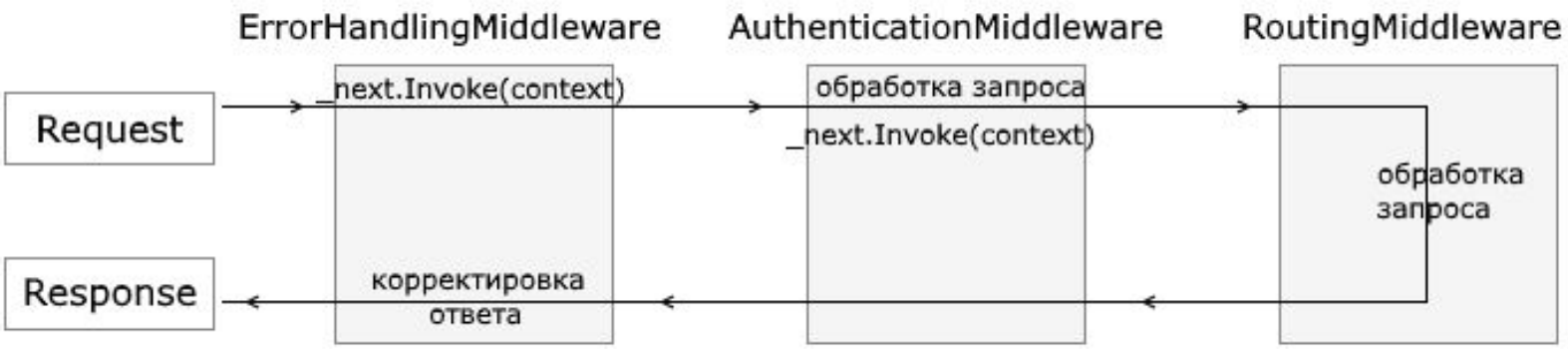
ASP.NET Core

В отличие от предыдущих двух компонентов ErrorHandlingMiddleware сначала передает запрос на выполнение последующим делегатам, а потом уже сам обрабатывает. Это возможно, поскольку каждый компонент обрабатывает запрос два раза: вначале вызывается та часть кода, которая идет до await _next.Invoke(context);, а после завершения обработки последующих компонентов вызывается та часть кода, которая идет после await _next.Invoke(context);.

И в данном случае для ErrorHandlingMiddleware важен результат обработки запроса последующими компонентами. В частности, он устанавливает сообщения об ошибках в зависимости от того, как статусный код установили другие компоненты. Поэтому ErrorHandlingMiddleware должен быть помещен первым из всех трех компонентов:

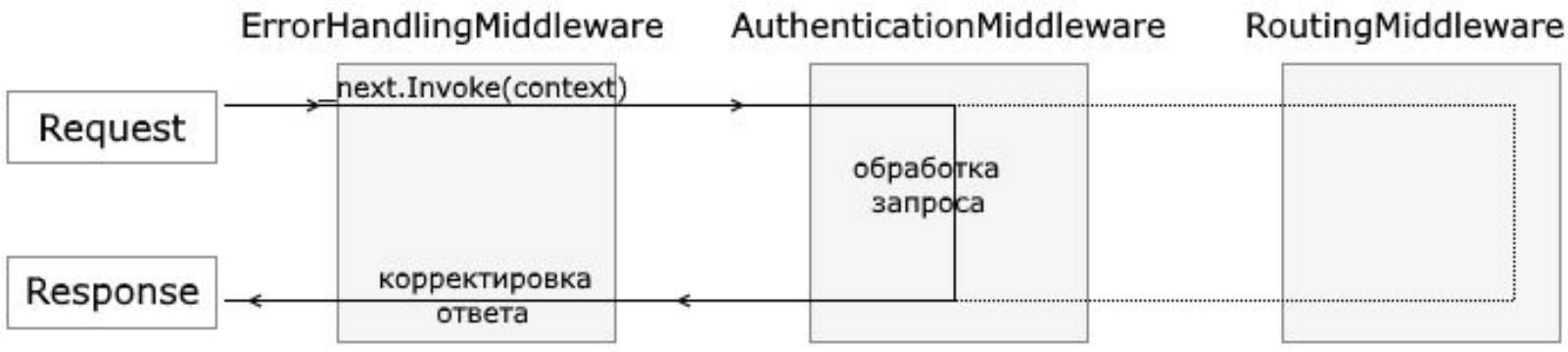
```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseMiddleware<ErrorHandlingMiddleware>();
        app.UseMiddleware<AuthenticationMiddleware>();
        app.UseMiddleware<RoutingMiddleware>();
    }
}
```

Конвейер обработки запроса

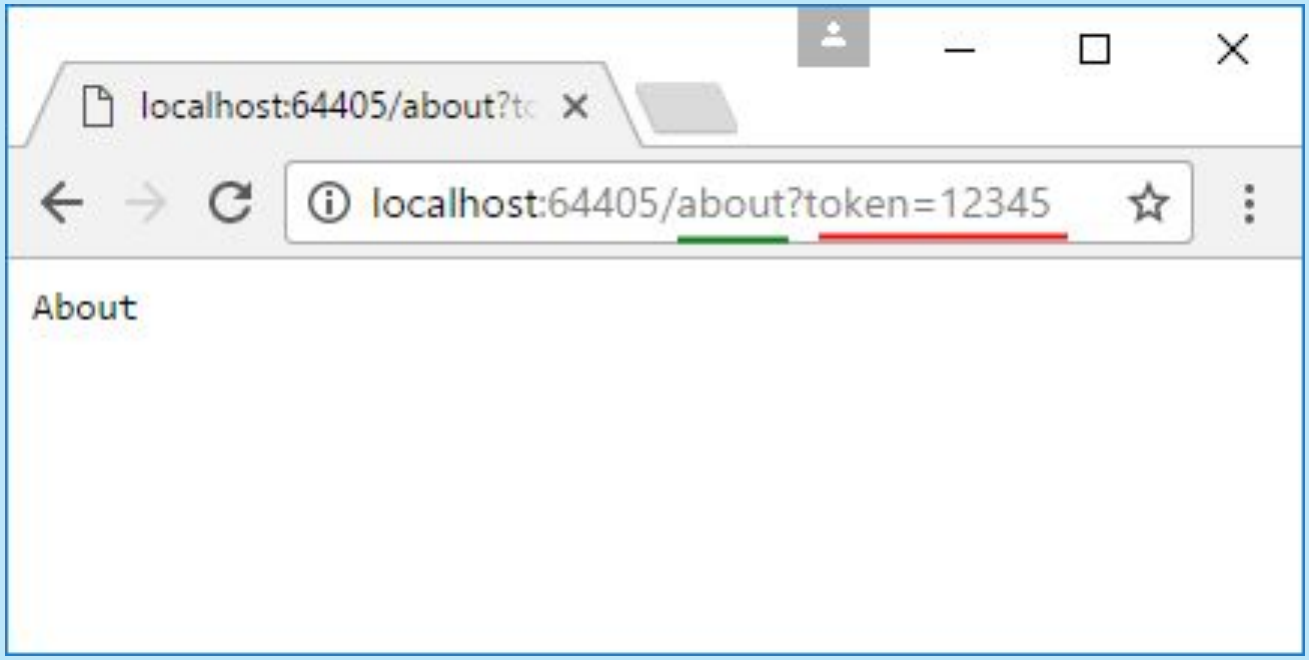


ASP.NET Core

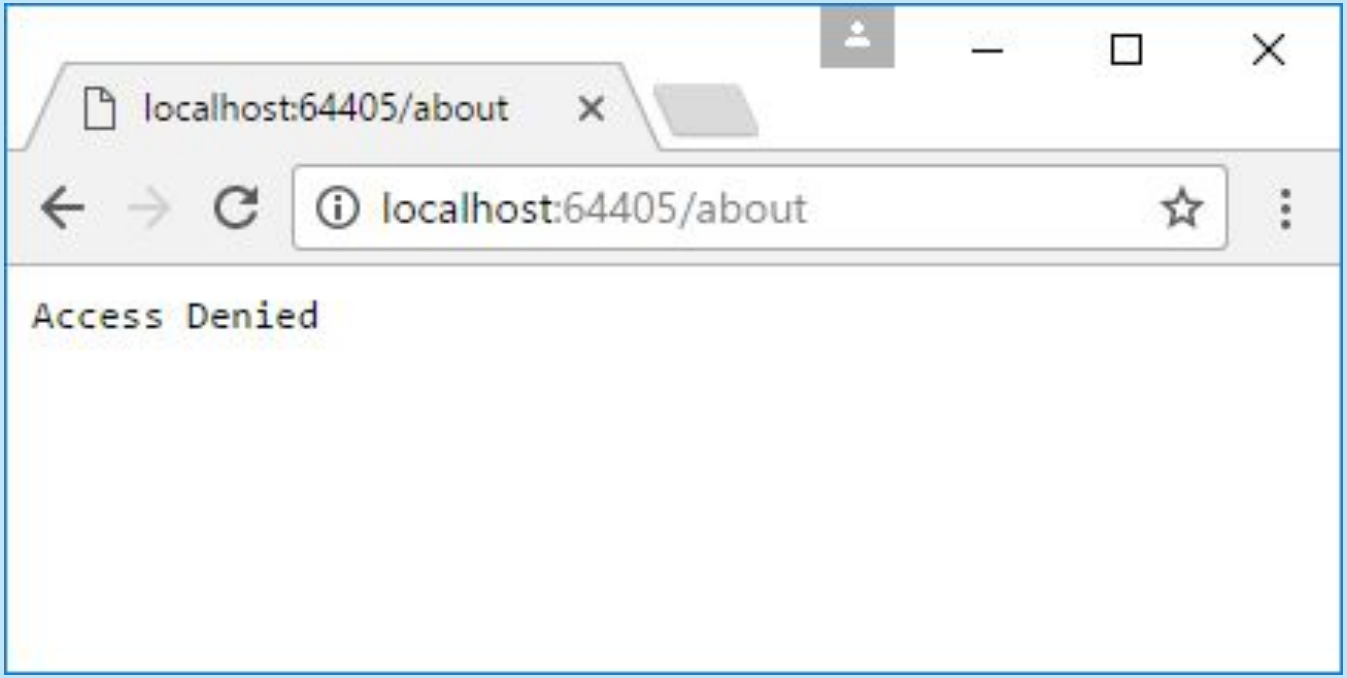
В то же время, если к приложению обратится пользователь, не указав в строке запроса параметр token, то AuthenticationMiddleware не будет передавать дальше запрос на обработку, а конвейер обработки будет выглядеть так:



В первом случае, если указан параметр token, то запрос будет обработан RoutingMiddleware:



Иначе пользователь получит ошибку 403:



ASP.NET Core

Для взаимодействия со средой, в которой запущено приложение, применяются объекты, реализующие интерфейс **IHostingEnvironment**. Этот интерфейс предлагает ряд свойств, с помощью которых мы можем получить информацию об окружении:

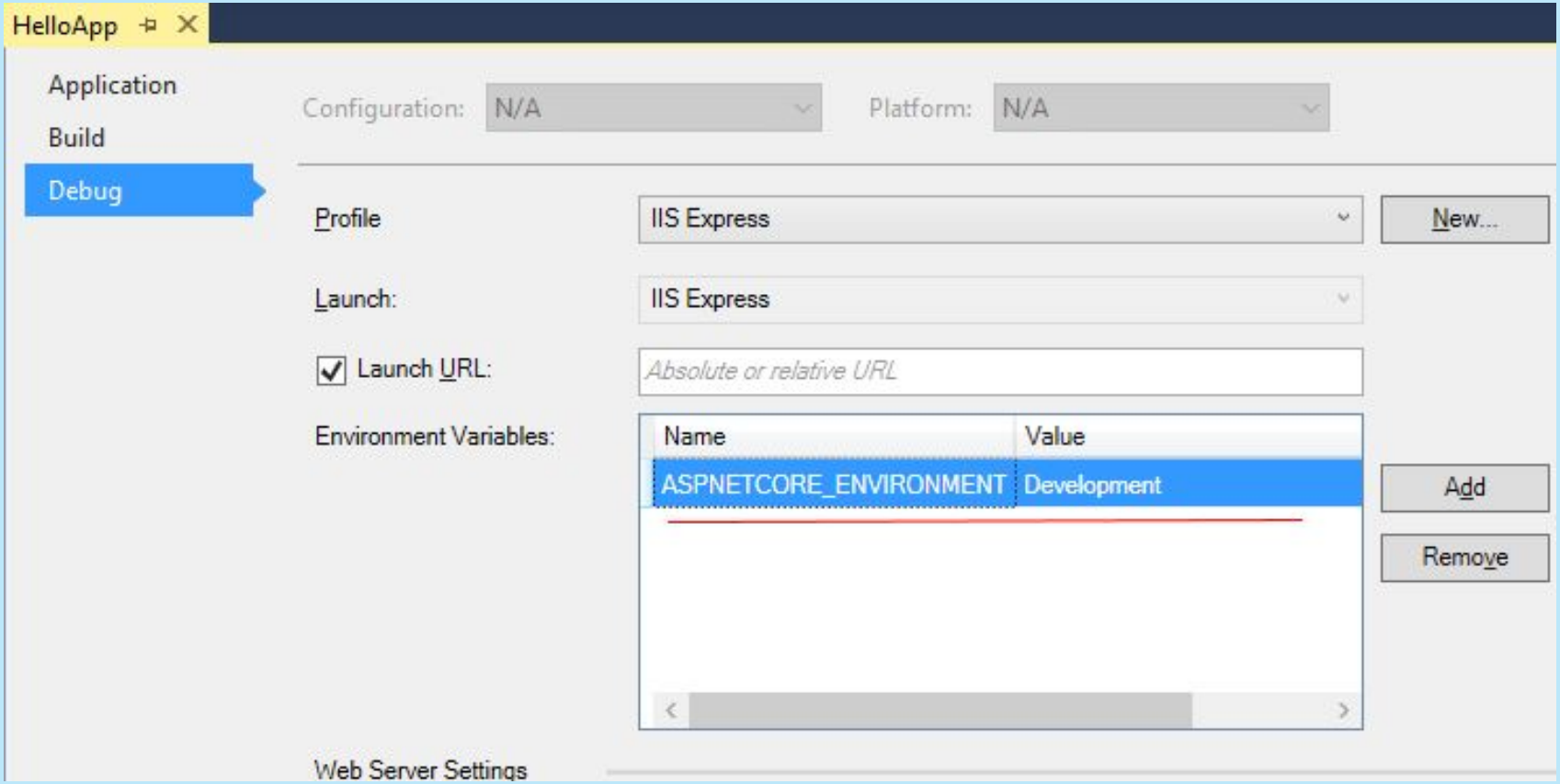
- **ApplicationName**: возвращает имя приложения
- **EnvironmentName**: возвращает описание среды, в которой хостируется приложение
- **ContentRootPath**: возвращает путь к корневой папке приложения

ASP.NET Core

- **WebRootPath:** возвращает путь к папке, в которой хранится статический контент приложения, как правило, это папка wwwroot
- **ContentRootFileProvider:** возвращает реализацию интерфейса Microsoft.AspNetCore.FileProviders.IFileProvider, которая может использоваться для чтения файлов из папки ContentRootPath
- **WebRootFileProvider:** возвращает реализацию интерфейса Microsoft.AspNetCore.FileProviders.IFileProvider, которая может использоваться для чтения файлов из папки WebRootPath

ASP.NET Core

При разработке можно использовать эти свойства. Но наиболее часто при разработке придется сталкиваться со свойством **EnvironmentName**. По умолчанию имеются три варианта значений для этого свойства: Development, Staging и Production.



ASP.NET Core

В проекте это свойство задается через установку переменной среды ASPNETCORE_ENVIRONMENT. Ее текущее значение можно посмотреть в свойствах проекта на вкладке Debug.

Здесь же также можно изменить значение этой переменной.

Кроме того, в проекте в папке Properties есть файл launchSettings.json, который также содержит описания переменных сред.

Для определения значения этой переменной для интерфейса IWebHostEnvironment определены специальные методы расширения:

- **IsEnvironment(string envName):** возвращает true, если имя среды равно значению параметра envName.
- **IsDevelopment()**: возвращает true, если имя среды - Development
- **IsStaging()**: возвращает true, если имя среды – Staging
- **IsProduction()**: возвращает true, если имя среды - Production

ASP.NET Core

Например, при создании нового проекта в методе Configure() класса Startup можно найти следующие строки:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
```

ASP.NET Core

Таким образом, если имя среды имеет значение "Development", то есть приложение находится в состоянии разработки, то при ошибке разработчик увидит детальное описание ошибки. Если же приложение развернуто на хостинге и соответственно имеет другое имя хостирующей среды, то простой пользователь при ошибке ничего не увидит. Таким образом, в зависимости от стадии, на которой находится проект, можно скрывать или задействовать часть функционала приложения.

Определение своих состояний среды

Хотя по умолчанию среда может принимать три состояния: Development, Staging, Production, но мы можем при желании вводить новые значения. Например, нам надо отслеживать какие-то дополнительные состояния. К примеру, изменим в файле launchSettings.json значение "ASPNETCORE_ENVIRONMENT" на "Test" (значение может быть произвольное).

Определение своих состояний среды

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": " http://localhost:58666",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "Login": "Tom",
        "ASPNETCORE_ENVIRONMENT": "Test"
      }
    },
    "HelloApp": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Test"
      },
      "applicationUrl": " http://localhost:5000"
    }
  }
}
```

Определение своих состояний среды

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Hosting;

namespace HelloApp
{
    public class Startup
    {
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.Run(async (context) =>
            {
                context.Response.Headers["Content-Type"] = "text/html; charset=utf-8";

                if (env.IsEnvironment("Test")) // Если проект в состоянии "Test"
                {
                    await context.Response.WriteAsync("В состоянии тестирования");
                }
                else
                {
                    await context.Response.WriteAsync("В процессе разработки или в
продакшене");
                }
            });
        }
    }
}
```

Определение своих состояний среды