Распределенные информационно-аналитические системы

Практическое занятие № 6. «Основы в ASP.NET Core. Часть 6»

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

Учебные вопросы:

1. **Метод Map.**
2. **Классы middleware.**
3. **Построение конвейера обработки запроса.**
4. **IWebHostEnvironment и окружение.**

1. Метод Map

Метод **Map()** применяется для создания ветки конвейера, которая будет обрабатывать запрос по определенному пути. Этот метод реализован как метод расширения для типа **IApplicationBuilder** и имеет ряд перегруженных версий. Например:

```
public static IApplicationBuilder Map (this IApplicationBuilder app, string pathMatch, Action<IApplicationBuilder> configuration);
```

В качестве параметра **pathMatch** метод принимает путь запроса, с которым будет сопоставляться ветка. А параметр **configuration** представляет делегат, в который передается объект **IApplicationBuilder** и в котором будет создаваться ветка конвейера.

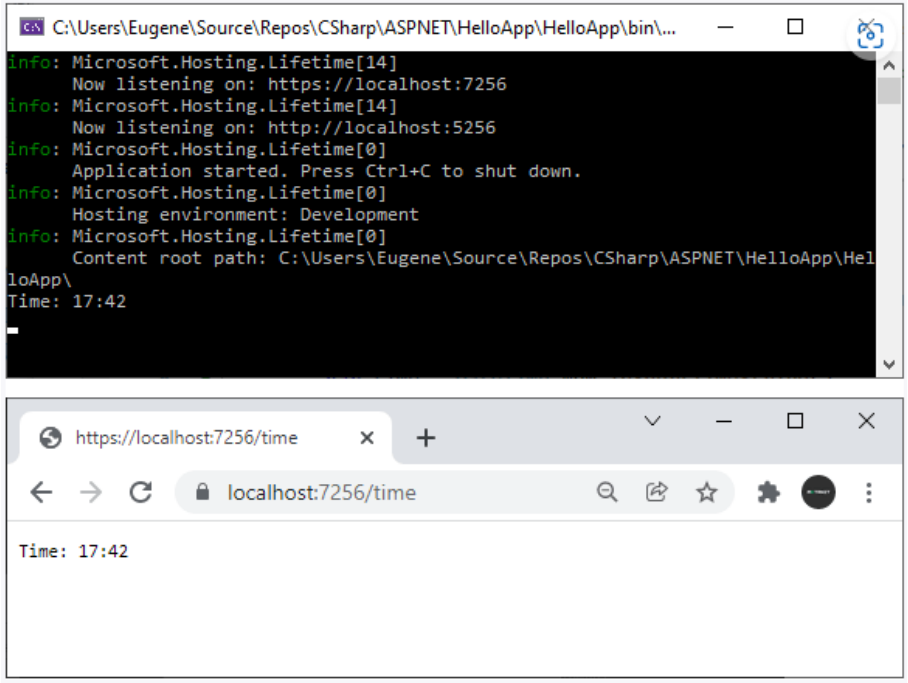
Рассмотрим простой пример:

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

app.Map("/time", appBuilder =>
{
    var time = DateTime.Now.ToShortTimeString();

    // логгируем данные - выводим на консоль приложения
    appBuilder.Use(async(context, next) =>
    {
        Console.WriteLine($"Time: {time}");
        await next();   // вызываем следующий middleware
    });

    appBuilder.Run(async context => await context.Response.WriteAsync($"Time: {time}"));
});

app.Run(async (context) => await context.Response.WriteAsync("Hello METANIT.COM"));

app.Run();
```

В данном случае метод **app.Map()** создает ответвление конвейера, которое будет обрабатывать запросы по пути "/time":

```
appBuilder =>
{
    var time = DateTime.Now.ToShortTimeString();
    // логгируем данные - выводим на консоль приложения
    appBuilder.Use(async (context, next) =>
    {
        Console.WriteLine($"Time: {time}");
        await next();   // вызываем следующий middleware
    });

    appBuilder.Run(async context => await context.Response.WriteAsync($"Time: {time}"));
}
```

Созданная ветка конвейера содержит два **middleware**, встраиваемые с помощью методов **Use()** и **Run()**. Вначале получаем текущее время и в первом **middleware** логгируем это время на консоль. Во втором – терминальном компоненте **middleware** отправляем информацию о времени в ответ клиенту.

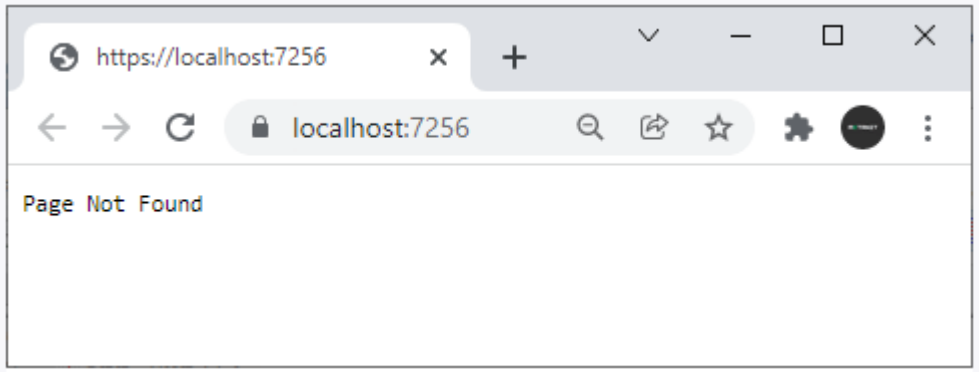


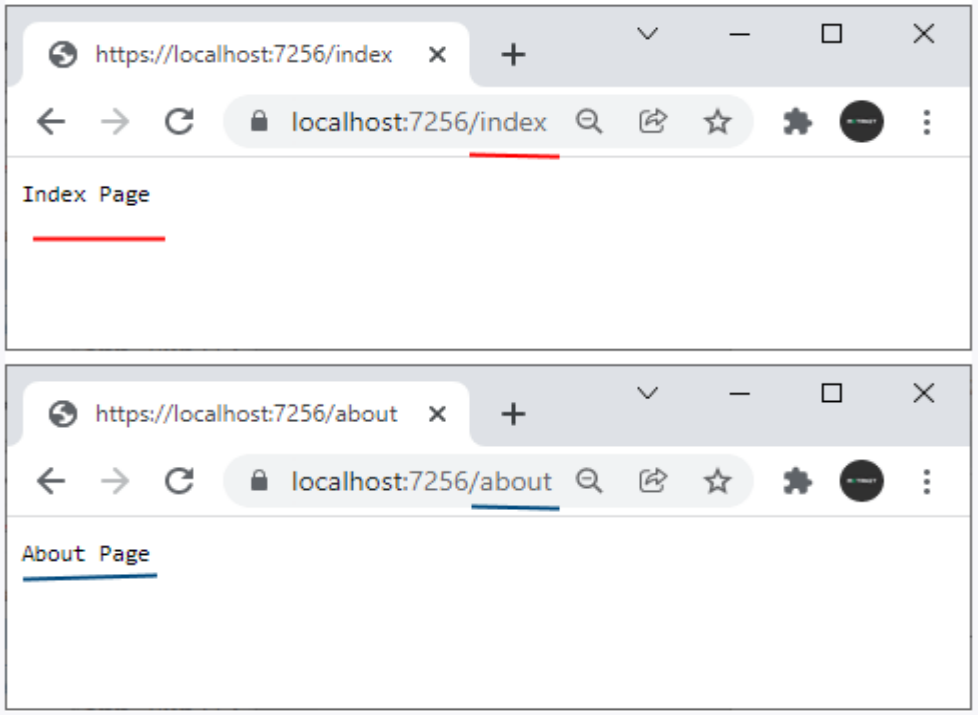
При других путях запросах, отличных от "/time", запрос будет обрабатываться основным потоком конвейера, который состоит в данном случае из одного компонента:

```
app.Run(async (context) => await context.Response.WriteAsync("Hello METANIT.COM"));
```

Подобным образом можно создавать ветки для разных путей:

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

app.Map("/index", appBuilder =>
{
    appBuilder.Run(async context => await context.Response.WriteAsync("Index Page"));
});
app.Map("/about", appBuilder =>
{
    appBuilder.Run(async context => await context.Response.WriteAsync("About Page"));
});

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

app.Run();
```





При необходимости создание веток конвейера можно вынести в отдельные методы:

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

app.Map("/index", Index);
app.Map("/about", About);

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

app.Run();

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

Вложенные методы Map

Ветка конвейера, которая создается в методе **Map()**, может иметь вложенные ветки, которые обрабатывают подзапросы. Например:

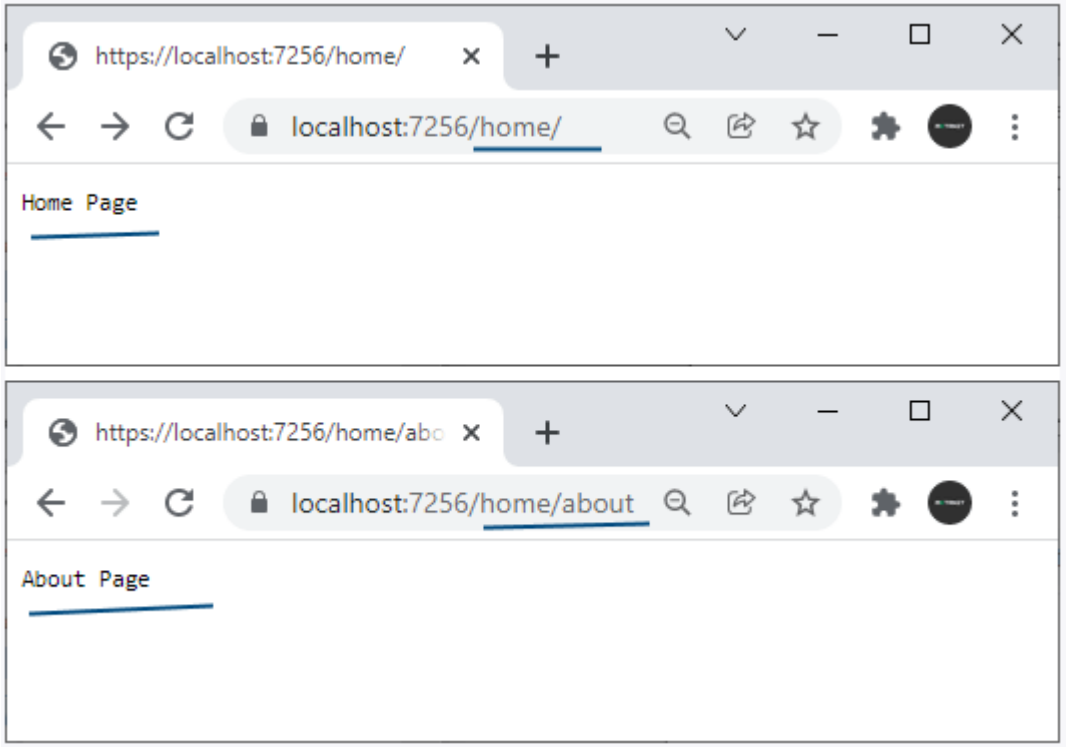
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/home", appBuilder =>
{
    appBuilder.Map("/index", Index); // middleware для "/home/index"
    appBuilder.Map("/about", About); // middleware для "/home/about"
    // middleware для "/home"
    appBuilder.Run(async (context) => await context.Response.WriteAsync("Home Page"));
});

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

app.Run();

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

Здесь ветка создается с помощью вызова

```
app.Map("/home", appBuilder =>
```

Эта ветка будет обрабатывать запросы по пути "/home".

Внутри этой ветки создаются две вложенные ветки, которые будут обрабатывать запросы по путям относительно пути основной ветки. То есть теперь метод **About** будет обрабатывать запрос по пути "/home/about", а не "/about".



2. Классы middleware

В прошлых учебных вопросах используемые компоненты **middleware** фактически представляли методы, то есть так называемые **inline middleware**. Однако также ASP.NET Core позволяет определять **middleware** в виде отдельных классов.

Итак, добавим в проект новый класс, который назовем **TokenMiddleware** и который будет иметь следующий код:

```
public class TokenMiddleware
{
    private readonly RequestDelegate next;

    public TokenMiddleware(RequestDelegate next)
    {
        this.next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var token = context.Request.Query["token"];
        if (token!="12345678")
        {
            context.Response.StatusCode = 403;
            await context.Response.WriteAsync("Token is invalid");
        }
        else
        {
            await next.Invoke(context);
        }
    }
}
```

Класс **middleware** должен иметь конструктор, который принимает параметр типа **RequestDelegate**. Через этот параметр можно получить ссылку на тот делегат запроса, который стоит следующим в конвейере обработки запроса.

Также в классе должен быть определен метод, который должен называться либо **Invoke**, либо **InvokeAsync**. Причем этот метод должен возвращать объект **Task** и принимать в качестве параметра контекст запроса – объект **HttpContext**. Данный метод? Собственно? и будет обрабатывать запрос.

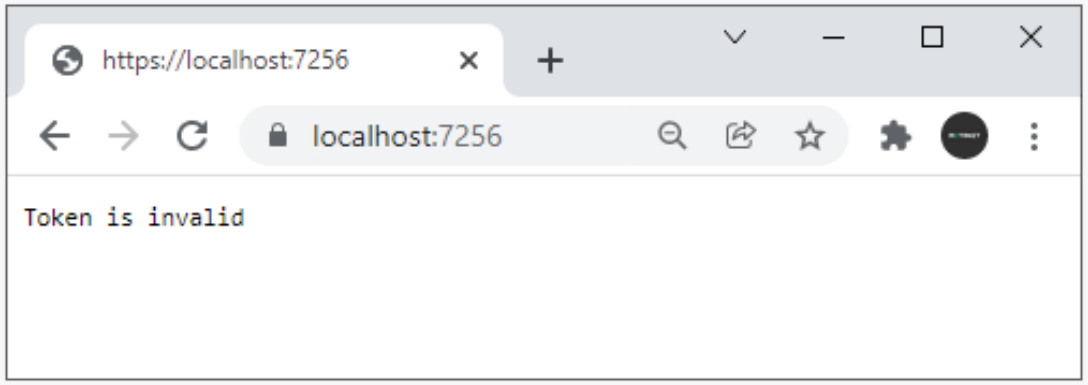
Суть действия класса заключается в том, что мы получаем из запроса параметр **"token"**. Если полученный токен равен строке "12345678", то передаем запрос дальше следующему компоненту, вызвав метод **_next.Invoke()**. Иначе возвращаем пользователю сообщение об ошибке.

Для добавления компонента **middleware**, который представляет класс, в конвейер обработки запроса применяется метод **UseMiddleware()**. Так, изменим файл Program.cs следующим образом:

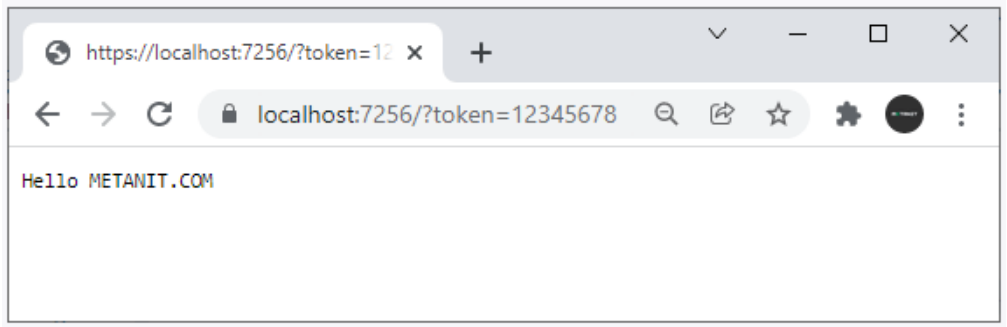
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.UseMiddleware<TokenMiddleware>();

app.Run(async(context) => await context.Response.WriteAsync("Hello METANIT.COM"));

app.Run();
```

С помощью метода **UseMiddleware<T>** в конструктор объекта **TokenMiddleware** будет внедряться объект для параметра **RequestDelegate next**. Поэтому явным образом передавать значение для этого параметра нам не нужно.

Запустим проект. И если мы не передадим через строку запроса параметр **token** или передадим для него значение, отличное от "12345678", то браузер отобразит ошибку:



Если же будет передан корректный токен, то вызов **app.UseMiddleware<TokenMiddleware>()** передаст обработку запроса в компонент **middleware из app.Run()**:



Метод расширения для встраивания middleware

Также нередко для встраивания подобных компонентов **middleware** определяются специальные методы расширения. Так, добавим в проект новый класс, который назовем **TokenExtensions**:

```
public static class TokenExtensions
{
    public static IApplicationBuilder UseToken(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<TokenMiddleware>();
    }
}
```

Здесь создается метод расширения для типа **IApplicationBuilder**. И этот метод встраивает компонент **TokenMiddleware** в конвейер обработки запроса. Как правило, подобные методы возвращают объект **IApplicationBuilder**.

Теперь применим этот метод в коде программы в Program.cs:

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

app.UseToken();

app.Run(async(context) => await context.Response.WriteAsync("Hello METANIT.COM"));

app.Run();
```

Передача параметров

Изменим класс **TokenMiddleware**, чтобы он извне получал образец токена для сравнения:

```
public class TokenMiddleware
{
    private readonly RequestDelegate next;
    string pattern;
    public TokenMiddleware(RequestDelegate next, string pattern)
    {
        this.next = next;
        this.pattern = pattern;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var token = context.Request.Query["token"];
        if (token != pattern)
        {
            context.Response.StatusCode = 403;
            await context.Response.WriteAsync("Token is invalid");
        }
        else
        {
            await next.Invoke(context);
        }
    }
}
```

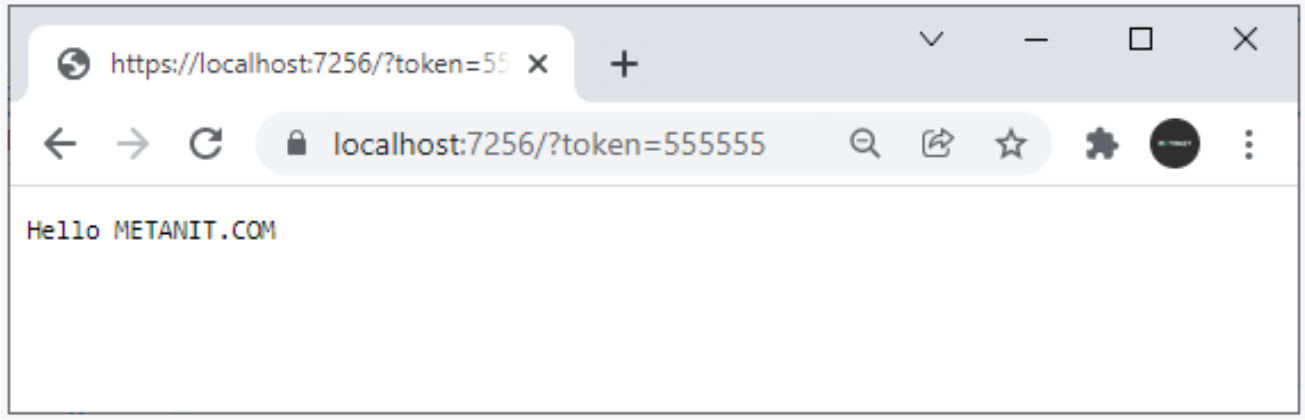
Образец токена, с которым идет сравнение, устанавливается через конструктор. Чтобы передать его в конструктор, изменим класс **TokenExtensions**:

```
public static class TokenExtensions
{
    public static IApplicationBuilder UseToken(this IApplicationBuilder builder, string pattern)
    {
        return builder.UseMiddleware<TokenMiddleware>(pattern);
    }
}
```

В метод **builder.UseMiddleware** можно передать набор значений, которые передаются в конструктор компонента **middleware**.

И при вызове метода расширения **UseToken** в него можно передать конкретное значение:

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

app.UseToken("555555");

app.Run(async(context) => await context.Response.WriteAsync("Hello METANIT.COM"));

app.Run();
```



3. Построение конвейера обработки запроса

Обычно 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.

Рассмотрим простейший пример. Определим следующий класс **RoutingMiddleware** для обработки запроса:

```
public class RoutingMiddleware
{
    readonly RequestDelegate next;
    public RoutingMiddleware(RequestDelegate next)
    {
        this.next = next;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        string path = context.Request.Path;
        if (path == "/index")
        {
            await context.Response.WriteAsync("Home Page");
        }
        else if (path == "/about")
        {
            await context.Response.WriteAsync("About Page");
        }
        else
        {
            context.Response.StatusCode = 404;
        }
    }
}
```

Этот компонент в зависимости от строки запроса возвращает либо определенную строку, либо устанавливает код ошибки.

Допустим, мы хотим, чтобы пользователь был аутентифицирован при обращении к нашему приложению. Для этого добавим новый класс **AuthenticationMiddleware**:

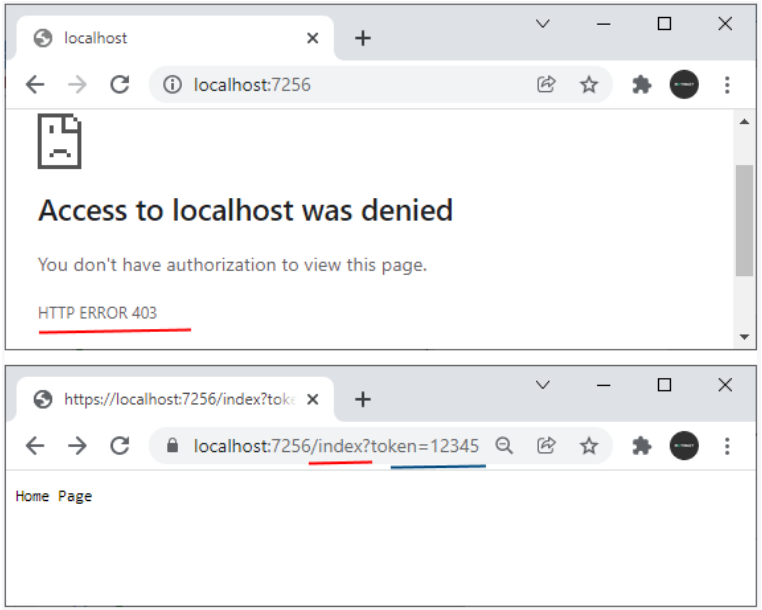
```
public class AuthenticationMiddleware
{
    readonly RequestDelegate next;
    public AuthenticationMiddleware(RequestDelegate next)
    {
        this.next = next;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        var token = context.Request.Query["token"];
        if (string.IsNullOrWhiteSpace(token))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

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**:

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

app.UseMiddleware<AuthenticationMiddleware>();
app.UseMiddleware<RoutingMiddleware>();

app.Run();
```

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:



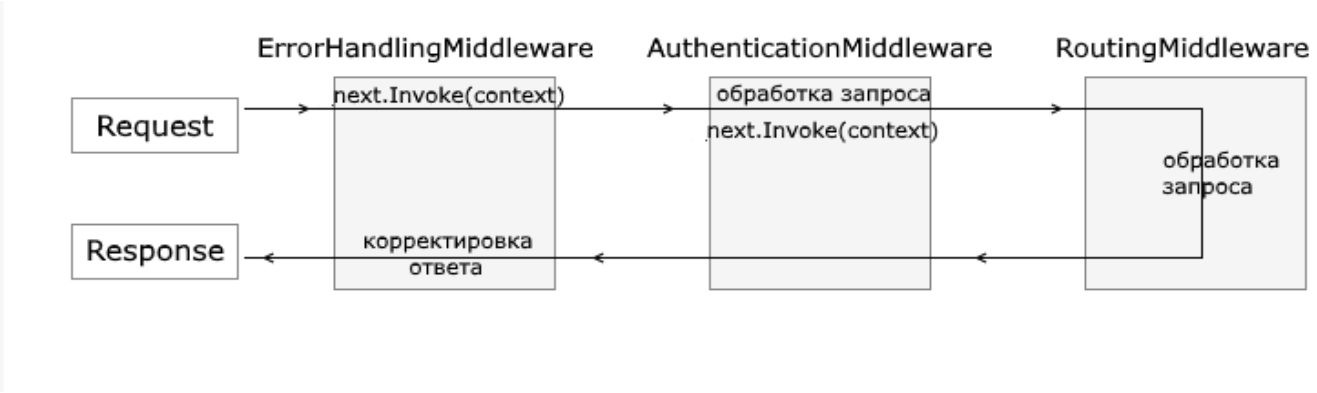
Добавим еще один компонент **middleware**, который назовем **ErrorHandlingMiddleware**:

```
public class ErrorHandlingMiddleware
{
    readonly RequestDelegate next;
    public ErrorHandlingMiddleware(RequestDelegate next)
    {
        this.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");
        }
    }
}
```

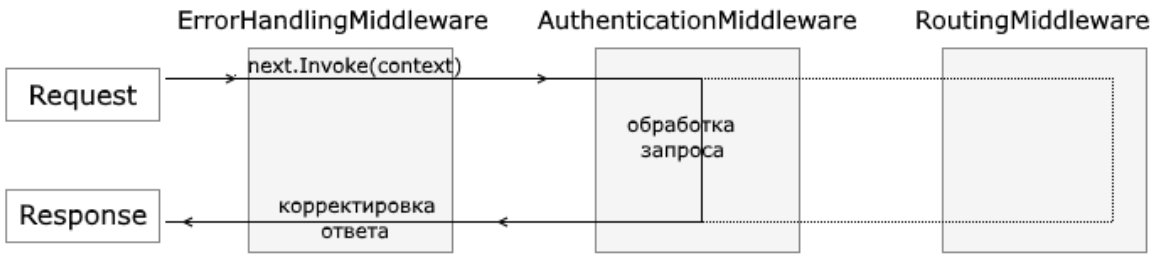
В отличие от предыдущих двух компонентов **ErrorHandlingMiddleware** сначала передает запрос на выполнение последующим делегатам, а потом уже сам обрабатывает. Это возможно, поскольку каждый компонент обрабатывает запрос два раза: вначале вызывается та часть кода, которая идет до **await next.Invoke(context);**,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

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

app.UseMiddleware<ErrorHandlingMiddleware>();
app.UseMiddleware<AuthenticationMiddleware>();
app.UseMiddleware<RoutingMiddleware>();

app.Run();
```

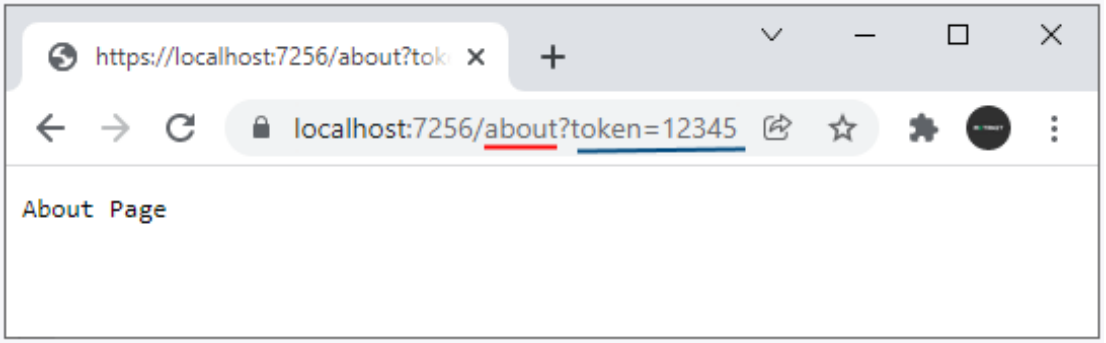
Схематично конвейер обработки запроса будет выглядеть следующим образом:



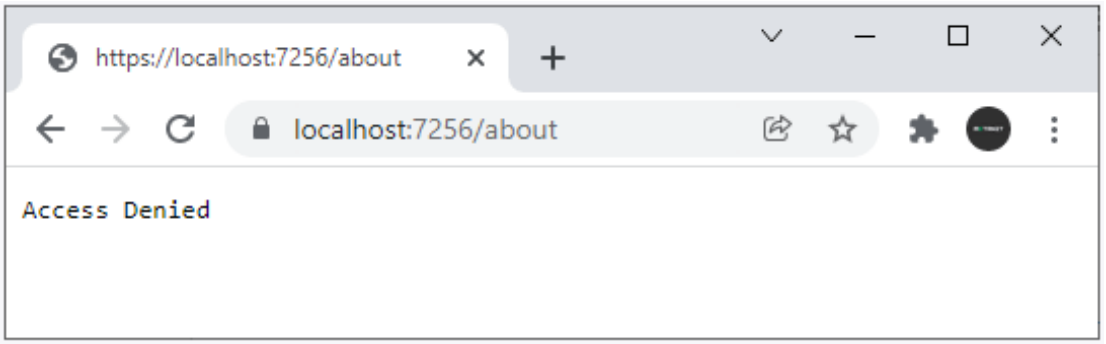
В то же время, если к приложению обратится пользователь, не указав в строке запроса параметр **token**, то **AuthenticationMiddleware** не будет передавать дальше запрос на обработку, а конвейер обработки будет выглядеть так:



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



Иначе пользователь получит ошибку 403:



4. IWebHostEnvironment и окружение

Для взаимодействия с окружением, в котором запущено приложение, фреймфорк ASP.NET Core предоставляет интерфейс **IWebHostEnvironment**. Этот интерфейс предлагает ряд свойств, с помощью которых мы можем получить информацию об окружении:

ApplicationName: хранит имя приложения.

EnvironmentName: хранит название среды, в которой хостируется приложение.

ContentRootPath: хранит путь к корневой папке приложения.

WebRootPath: хранит путь к папке, в которой хранится статический контент приложения. По умолчанию это папка wwwroot

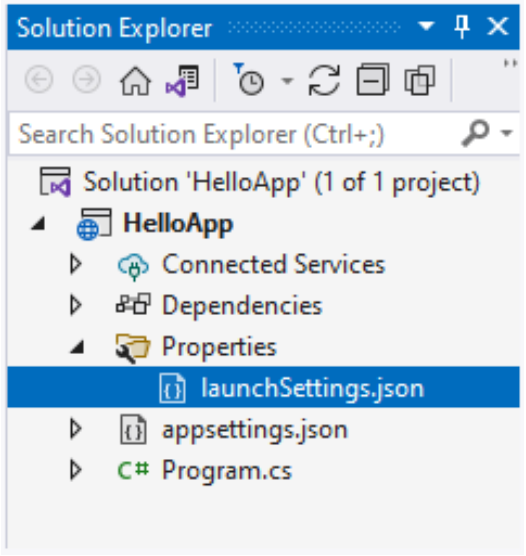
ContentRootFileProvider: возвращает реализацию интерфейса **Microsoft.AspNetCore.FileProviders.IFileProvider**, которая может использоваться для чтения файлов из папки ContentRootPath.

WebRootFileProvider: возвращает реализацию интерфейса **Microsoft.AspNetCore.FileProviders.IFileProvider**, которая может использоваться для чтения файлов из папки WebRootPath.

При разработке мы можем 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**.

В проекте это свойство задается через установку переменной среды ASPNETCORE_ENVIRONMENT. Текущее значение данного параметра задается в файле launchSettings.json, который располагается в проекте в папке Properties.



Откроем данный файл:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:56234",
      "sslPort": 44384
    }
  },
  "profiles": {
    "HelloApp": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7256;http://localhost:5256",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Здесь можно увидеть, что переменная **"ASPNETCORE_ENVIRONMENT"** встречается два раза – для запуска через **IISExpress** и для запуска через **Kestrel.** В обоих случаях она имеет значение **Development**. Но мы можем поменять значение этой переменной.

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**.

Данная функциональность позволяет нам выполнять определенный код в зависимости от того, на какой стадии находится приложение. Если приложение в процессе разработки, то мы можем выполнять один код, а при развертывании для полноценного использования другой код:

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

if (app.Environment.IsDevelopment())
{
    app.Run(async (context) => await context.Response.WriteAsync("In Development Stage"));
}
else
{
    app.Run(async (context) => await context.Response.WriteAsync("In Production Stage"));
}
Console.WriteLine($"{app.Environment.EnvironmentName}");

app.Run();
```

Так, если мы посмотрим на код класса **WebApplicationBuilder**, который применяется для создания приложения, то мы там можем увидеть там следующие строки:

```
if (context.HostingEnvironment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
```

Здесь 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, мы можем скрывать или задействовать часть функционала приложения.

Если мы хотим поменять значение среды, необязательно изменять файл launchSettings.json. Это можно сделать также программно:

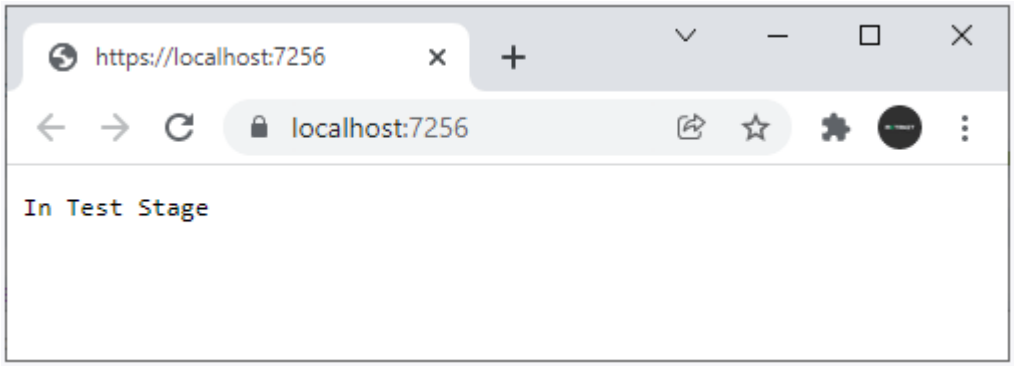
```
var builder = WebApplication.CreateBuilder();
WebApplication app = builder.Build();

app.Environment.EnvironmentName = "Production";
```

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. Это можно сделать через изменение файла launchSettings.json либо программно.

Например, изменим название среды на "Test" (значение может быть произвольное):

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

app.Environment.EnvironmentName = "Test";   // изменяем название среды на Test

if (app.Environment.IsEnvironment("Test")) // Если проект в состоянии "Test"
{
    app.Run(async (context) => await context.Response.WriteAsync("In Test Stage"));
}
else
{
    app.Run(async (context) => await context.Response.WriteAsync("In Development or Production Stage"));
}

app.Run();
```



Также можно изменить значение "ASPNETCORE_ENVIRONMENT" на "Test" или любое другой в файле launchSettings.json для используемого профиля:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:56234",
      "sslPort": 44384
    }
  },
  "profiles": {
    "HelloApp": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7256;http://localhost:5256",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Test"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```