Распределенные информационно-аналитические системы

Практическое занятие № 10. «Маршрутизация. Часть 2»

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

Учебные вопросы:

1. **Создание ограничений маршрутов.**
2. **Передача зависимостей в конечные точки.**
3. **Сопоставление запроса с конечной точкой.**
4. **Сочетание конечных точек с другими middleware.**

1. Создание ограничений маршрутов

Хотя фреймворк **ASP.NET Core** по умолчанию предоставляет большой набор встроенных ограничений, их может быть недостаточно. И в этом случае мы можем определить свои кастомные ограничения маршрутов.

Для создания собственного ограничения маршрута нужно реализовать интерфейс **IRouteConstraint** с одним единственным методом **Match**, который имеет следующее определение:

```
public interface IRouteConstraint
{
    bool Match(HttpContext? httpContext,
            IRouter? route,
            string routeKey,
            RouteValueDictionary values,
            RouteDirection routeDirection);
}
```

Параметры метода:

httpContext: объект **HttpContext**, который инкапсулирует информацию о HTTP-запросе.

route: объект **IRouter**, который представляет маршрут, в рамках которого применяется ограничение.

routeKey: объект **String** – название параметра маршрута, к которому применяется ограничение.

values: объект **RouteValueDictionary**, который представляет набор параметров маршрута в виде словаря, где ключи - названия параметров, а значения - значения параметров маршрута.

routeDirection: объект перечисления **RouteDirection**, которое указывает, применяется ограничение при обработке запроса, либо при генерации ссылки.

В качестве результата метод возвращает значение типа **bool**: **true**, если запрос удовлетворяет данному ограничению маршрута, и **false**, если не удовлетворяет.

Ограничение маршрута применяет этот интерфейс **IRouteConstraint**. Это вынуждает движок маршрутизации вызвать для ограничения маршрута метод **IRouteConstraint.Match**, чтобы определить, применяется ли данное ограничение к данному запросу или нет. И только если данный метод возвращает **true**, запрос может быть сопоставлен с маршрутом (конечно, если запрос также удовлетворяет и другим ограничениям, которые могут применяться).

Ограничение для параметра маршрута

Допустим, клиент в запросе через параметр должен передавать код, который равен некоторой строке. Для этого определим следующий класс ограничения маршрута:

```
public class SecretCodeConstraint : IRouteConstraint
{
    string secretCode;    // допустимый код
    public SecretCodeConstraint(string secretCode)
    {
        this.secretCode = secretCode;
    }

    public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDire
    {
        return values[routeKey]?.ToString() == secretCode;
    }
}
```

9 public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)

Класс **SecretCodeConstraint** через конструктор принимает некий условно секретный код, которому должен соответствовать параметр маршрута. В методе **Match()** с помощью выражения **values[routeKey]** получаем из словаря **values** значение параметра маршрута, имя которого передается через **routeKey**. И затем это значение сравниванием с секретным кодом:

```
return values[routeKey]?.ToString() == secretCode;
```

Если оба значения равны, то возвращаем **true**, что указывает, что маршрут соответствует данному ограничению.

Применим данное ограничение:

```
var builder = WebApplication.CreateBuilder();
// проецируем класс SecretCodeConstraint на inline-ограничение secretcode
builder.Services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("secretcode", typeof(SecretCodeConstraint)));

// альтернативное добавление класса ограничения
// builder.Services.AddRouting(options => options.ConstraintMap.Add("secretcode", typeof(SecretConstraint)));

var app = builder.Build();

app.Map(
    "/users/{name}/{token:secretcode(123466)}/",
    (string name, int token) => $"Name: {name} \nToken: {token}"
);
app.Map("/", () => "Index Page");

app.Run();

public class SecretCodeConstraint : IRouteConstraint
{
    string secretCode;    // допустимый код
    public SecretCodeConstraint(string secretCode)
    {
        this.secretCode = secretCode;
    }

    public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDire
    {
        return values[routeKey]?.ToString() == secretCode;
    }
}
```

27 public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)

Здесь надо отметить следующий момент. Если мы хотим использовать класс ограничения как inline-ограничение внутри шаблона маршрута, то нам необходимо изменить настройки для сервиса **RouteOptions**:

```
builder.Services.Configure<RouteOptions>(options =>
    options.ConstraintMap.Add("secretcode", typeof(SecretCodeConstraint)));
```

В данном случае параметр **options** представляет объект **RouteOptions**. Его свойство **ConstraintMap** представляет коллекцию применяемых ограничений. Метод **Add()** добавляет в эту коллекцию ограничение. Причем первый параметр этого метода представляет ключ ограничения в коллекции ограничений, а второй параметр – собственно класс ограничения. Далее ключ ограничения затем можно будет применять как inline-ограничение, на которое проецируется класс ограничения.

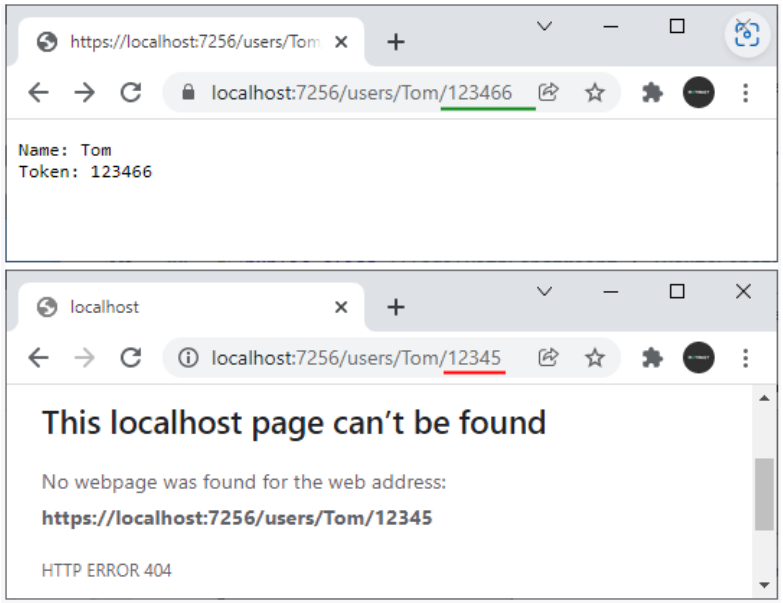
В качестве альтернативы вместо этого вызова мы могли бы обратиться к сервисам маршрутизации и также настроить объект **RouteOptions**:

```
builder.Services.AddRouting(options =>
                options.ConstraintMap.Add("secretcode", typeof(SecretConstraint)));
```

После этого мы сможем использовать inline-ограничение в шаблоне маршрута:

```
"/users/{name}/{token:secretcode(123466)}/"
```

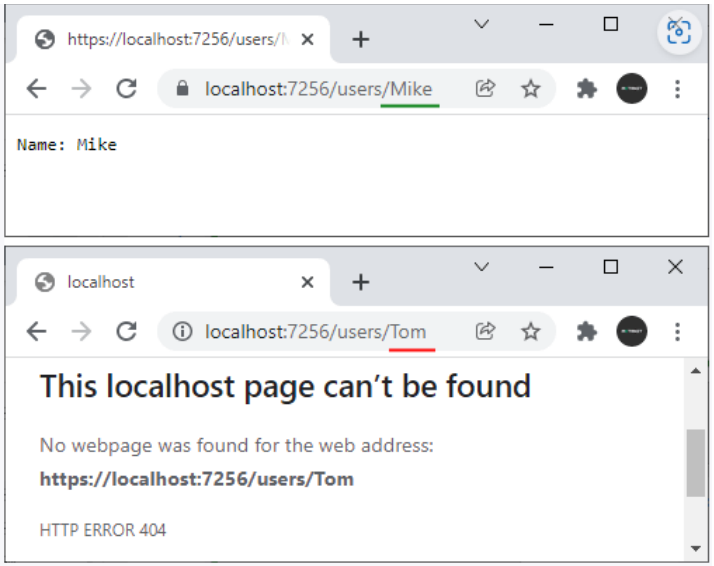
То есть здесь к параметру маршрута **token** будет применяться ограничение **SecretCodeConstraint**. А строка 123466 – это тот секретный код, который будет передаваться в конструктор объекта **SecretCodeConstraint**.



Другой пример. Допустим, мы хотим, чтобы параметр не мог иметь одно из набора значений:

```
var builder = WebApplication.CreateBuilder();
builder.Services.AddRouting(options =>
                options.ConstraintMap.Add("invalidnames", typeof(InvalidNamesConstraint)));
var app = builder.Build();

app.Map("/users/{name:invalidnames}", (string name) => $"Name: {name}");
app.Map("/", () => "Index Page");

app.Run();

public class InvalidNamesConstraint : IRouteConstraint
{
    string[] names = new[] { "Tom", "Sam", "Bob" };
    public bool Match(HttpContext? httpContext, IRouter? route, string routeKey,
        RouteValueDictionary values, RouteDirection routeDirection)
    {
        return !names.Contains(values[routeKey]?.ToString());
    }
}
```

В данном случае параметр маршрута **name** НЕ должен представлять имена "Tom", "Sam" и "Bob":



2. Передача зависимостей в конечные точки

Фреймворк **ASP.NET Core** предоставляет простой и удобный способ для передачи зависимостей в конечные точки. Все добавляемые в коллекцию сервисов приложения зависимости можно получить через параметры делегата, который отвечает за обработку запроса.

Например, определим следующее приложение:

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

builder.Services.AddTransient<TimeService>();   // Добавляем сервис

var app = builder.Build();

app.Map("/time", (TimeService timeService) => $"Time: {timeService.Time}");
app.Map("/", () => "Hello METANIT.COM");

app.Run();

// сервис
public class TimeService
{
    public string Time  =>  DateTime.Now.ToLongTimeString();
}
```

Здесь класс **TimeService** выступает в качестве сервиса, свойство **Time** которого возвращает текущее время в формате "hh:mm:ss".

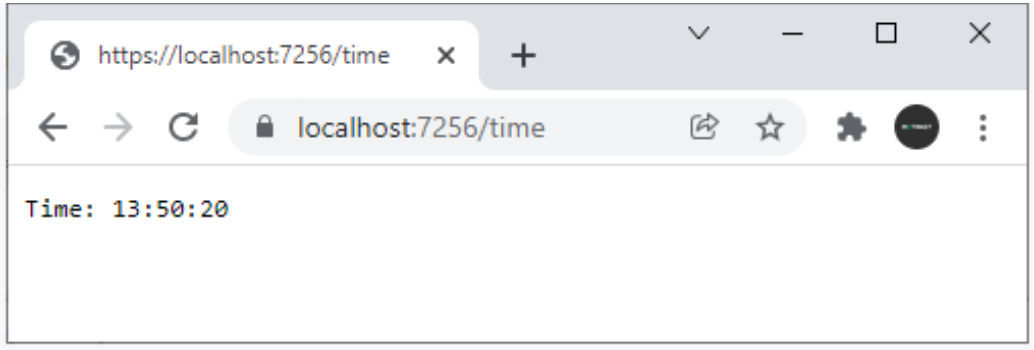
Этот сервис добавляется в коллекцию сервисов приложения:

```
builder.Services.AddTransient<TimeService>();
```

Далее через параметр делегата, который передается в качестве второго параметра в метод **Map()** мы можем получить эту зависимость:

```
app.Map("/time", (TimeService timeService) => $"Time: {timeService.Time}");
```

Таким образом, при обращении по адресу "/time" приложение возвратит клиенту текущее время:



Подобным образом можно получить зависимости, если обработчик маршрута конечной точки вынесен в отдельный метод:

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

builder.Services.AddTransient<TimeService>();   // Добавляем сервис

var app = builder.Build();

app.Map("/time", SendTime);
app.Map("/", () => "Hello METANIT.COM");

app.Run();

string SendTime(TimeService timeService)
{
    return $"Time: {timeService.Time}";
}
public class TimeService
{
    public string Time  =>  DateTime.Now.ToLongTimeString();
}
```

3. Сопоставление запроса с конечной точкой

Сопоставление адреса **URL** или **URL matching** представляет процесс сопоставления запроса с конечной точкой. Данный процесс основывается на пути запроса и полученных в запросе заголовках. Данный процесс проходит **ряд этапов**:

1. Сначала выбираются все конечные точки, шаблон маршрута которых совпадает с путем запроса.

2. Далее из полученного на предыдущем этапе набора конечных точек удаляются те, которые не соответствуют ограничениям маршрута.

3. Затем из полученного на предыдущем этапе набора конечных точек удаляются те, которые не удовлетворяют политике объекта **MatcherPolicy** (вкратце: класс **MatcherPolicy** позволяет определить порядок сравнения конечных точек и адреса **URL**).

4. И в самом конце применяется объект **EndpointSelector** для выбора из полученного на предыдущем этапе списка конечной точки, которая в конечном счете и будет обрабатывать запрос.

Приоритет конечных точек зависит от **двух факторов**:

1. Порядок следования в наборе конечных точек.

2. Приоритетность шаблона маршрута.

Приоритетность шаблонов маршрута зависит от специфичности шаблона. **Специфичность шаблона определяется на основе следующих критериев**:

1. Шаблон маршрута с большим количеством сегментов более специфичен, чем шаблон меньшим количеством сегментов.

2. Сегмент с текстовым литералом (статический сегмент) более специфичен, чем сегмент с параметром маршрута.

3. Сегмент с параметром, к которому применяется ограничение маршрута, более специфичен, чем сегмент с параметром без ограничения.

4. Комплексный сегмент более специфичен, чем сегмент с параметром с ограничением.

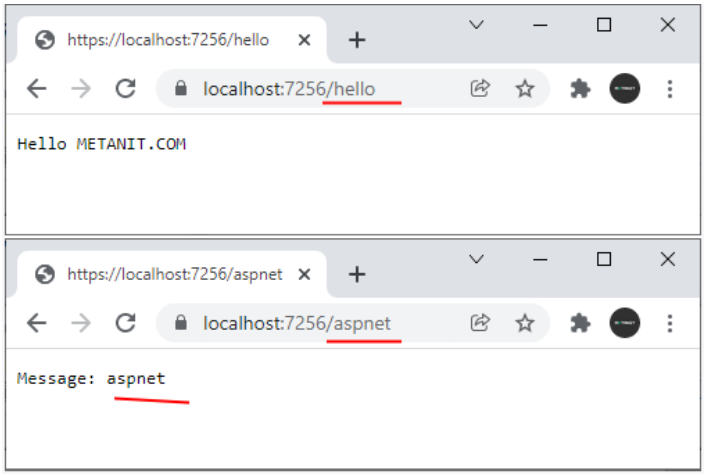
5. Параметр **catch-all** (параметр, который соответствует неопределенному количеству сегментов) наименее специфичен.

Если в конечном счете осталось две и более конечных точек, которые соответствуют запрошенному адресу, и соответственно система маршрутизации не может выбрать, какая из этих конечных точек должна обрабатывать маршрут, то генерируется исключение.

Например, пусть у нас есть два шаблона маршрута: "/hello" и "/{message}".

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

var app = builder.Build();

app.Map("/hello", () => "Hello METANIT.COM");
app.Map("/{message}", (string message) => $"Message: {message}");

app.Map("/", () => "Index Page");

app.Run();
```

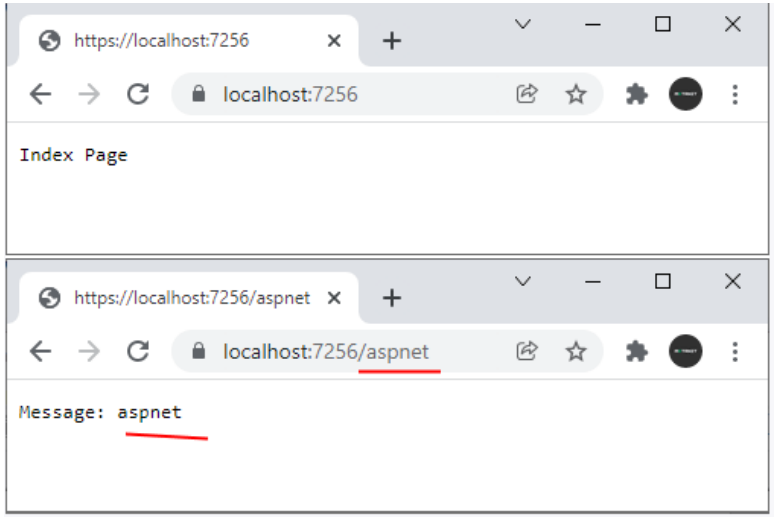
Оба этих маршрута соответствуют пути запроса "/hello". Однако шаблон маршрута "/{message}" более общий – вместо параметра **message** может идти что угодно. Тогда как шаблон "/hello" состоит из статического сегмента и соответственно более конкретный, более специфичный, поэтому конечная точка этого шаблона маршрута и будет выбрана для обработки запроса по пути "/hello".



Другой пример:

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

app.Map("/{message?}", (string? message) => $"Message: {message}");
app.Map("/", () => "Index Page");

app.Run();
```

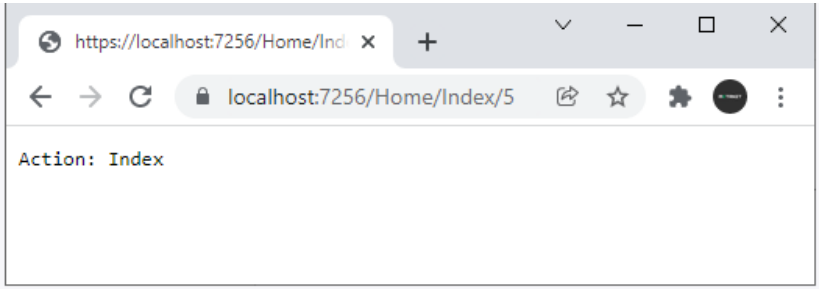
Здесь первый шаблон маршрута применяет необязательный параметр **message**. Второй шаблон маршрута соответствует корню веб-приложения. И в принципе оба этих шаблона соответствуют пути запроса "/". Однако второй шаблон представляет статический сегмент, поэтому его конечная точка будет выбрана в конечном счете для обработки запроса:



Более сложный пример:

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

app.Map("/{controller}/Index/5", (string controller) => $"Controller: {controller}");
app.Map("/Home/{action}/{id}", (string action) => $"Action: {action}");

app.Run();
```

Здесь опять же поскольку во втором маршруте первый сегмент представляет статический сегмент, то именно вторая конечная точка будет выбираться для обработки маршрута:



4. Сочетание конечных точек с другими middleware

Кроме конечных точек запрос в конвейере обработки могут обрабатывать и другие компоненты **middleware**. При этом надо учитывать общий процесс обработки запроса и вызова конечных точек.

Так, если приложение содержит конечные точки, то система маршрутизации на основе процесса **URL matching** или сопоставления адреса **URL** с шаблонами маршрута выбирает для обработки определенную конечную точку.

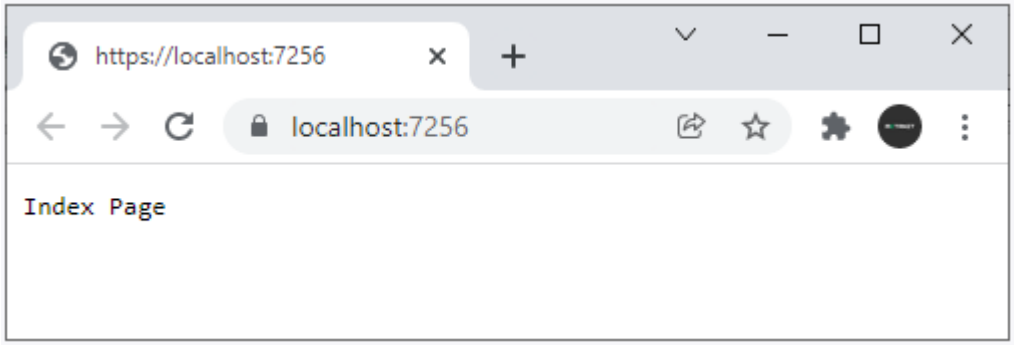
Если в приложении есть такая конечная точка, которая соответствует запросу, то компонент **middleware Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware** устанавливает у объекта **HttpContext** конечную точку для будущей обработки запроса, которую можно получить с помощью метода **HttpContext.GetEndpoint()**. Кроме того, устанавливаются значения маршрута, которые можно получить через коллекцию **HttpRequest.RouteValues**.

Однако конечная точка начинает обрабатывать запрос только после того, как все **middleware** в конвейере начнут обработку запроса. Например, возьмем следующий код приложения:

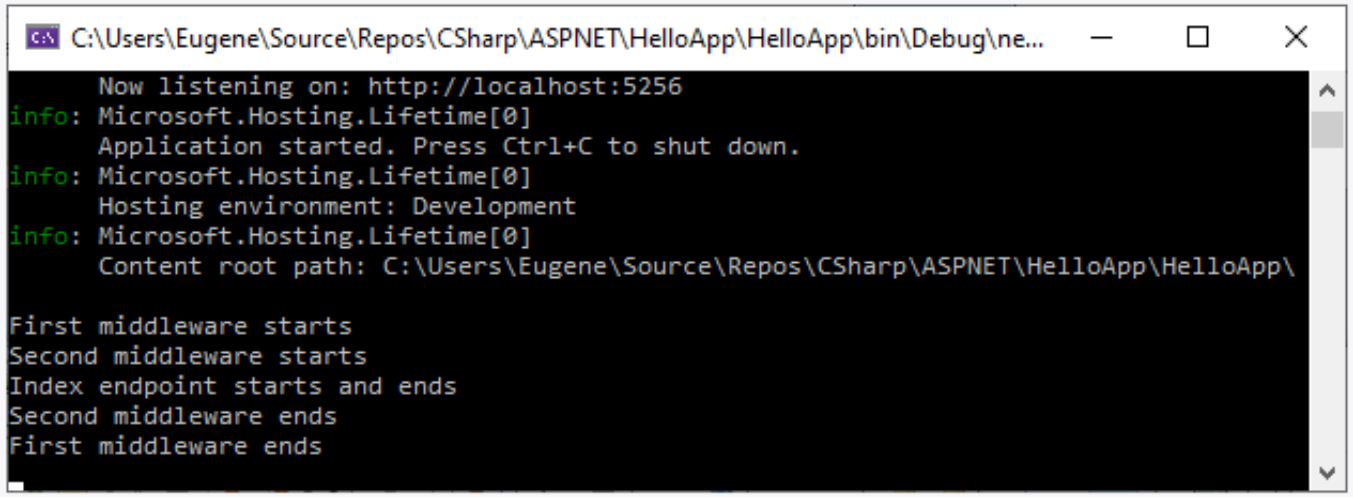
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Use(async (context, next) =>
{
    Console.WriteLine("First middleware starts");
    await next.Invoke();
    Console.WriteLine("First middleware ends");
});
app.Map("/", () =>
{
    Console.WriteLine("Index endpoint starts and ends");
    return "Index Page";
});
app.Use(async (context, next) =>
{
    Console.WriteLine("Second middleware starts");
    await next.Invoke();
    Console.WriteLine("Second middleware ends");
});
app.Map("/about", () =>
{
    Console.WriteLine("About endpoint starts and ends");
    return "About Page";
});
app.Run();
```

Здесь до и после первой конечной точки с помощью метода **app.Use()** в конвейер встроены два **middleware**. Для получения общей картины выполнения приложения процесс выполнения логгируется на консоль приложения.

Если мы запустим приложение, то при запросе по адресу "/" ожидаемо для обработки запроса будет выбрана первая конечная точка



Но теперь взглянем на консоль приложения:



Мы видим, что конечная точка выполняется после того, как начнет выполняться компонент **middleware**, который в коде идет после добавления этой конечной точки.

При этом в компонентах **middleware** также можно обрабатывать запросы по определенным адресам:

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

app.Use(async (context, next) =>
{
    if (context.Request.Path == "/date")
        await context.Response.WriteAsync($"Date: {DateTime.Now.ToShortDateString()}");
    else
        await next.Invoke();
});

app.Map("/", () => "Index Page");
app.Map("/about", () => "About Page");

app.Run();
```

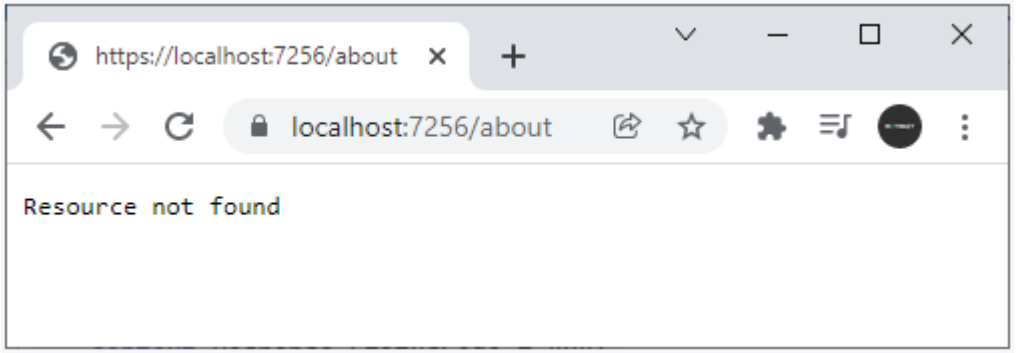
Кроме того, **middleware** могут быть полезны для некоторого постдействия – выполнения некоторых действий, когда конечная точка уже выбрана. Или, наоборот, если ни одна из конечных точек не обработала запрос, и в **middleware** мы можем обработать эту ситуацию:

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

app.Use(async(context, next) =>
{
    await next.Invoke();

    if (context.Response.StatusCode == 404)
        await context.Response.WriteAsync("Resource Not Found");
});
app.Map("/", () => "Index Page");
app.Map("/about", () => "About Page");
app.Run();
```

Однако если в конце конвейера располагается терминальный компонент, то он будет выполняться даже если конечная точка соответствует запрошенному пути:

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

app.Map("/", () => "Index Page");
app.Map("/about", () => "About Page");

app.Run(async context =>
{
    context.Response.StatusCode = 404;
    await context.Response.WriteAsync("Resource not found");
});
app.Run();
```

В данном случае по результату программы мы видим, что даже при запросах по адресу "/" и "/about" будет выполняться **middleware** из метода **app.Run**:



Почему такое происходит? Опять же, потому что, сначала выполняются все **middleware** в конвейере, и только потом выполняется конечная точка.