Распределенные информационно-аналитические системы

Практическое занятие № 9. «Маршрутизация. Часть 1»

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

Учебные вопросы:

1. **Конечные точки. Метод Map.**
2. **Параметры маршрута.**
3. **Ограничения маршрутов.**

1. Конечные точки. Метод Map

Система маршрутизации отвечает за сопоставление входящих запросов с маршрутами и на основании результатов сопоставления выбирает для обработки запроса определенную конечную точку приложения. Конечная точка или **endpoint** представляет некоторый код, который обрабатывает запрос. По сути, конечная точка объединяет шаблон маршрута, которому должен соответствовать запрос, и обработчик запроса по этому маршруту.

ASP.NET Core по умолчанию предоставляет простой и удобный функционал для создания конечных точек. Ключевым типом в этом функционале является интерфейс **Microsoft.AspNetCore.Routing.IEndpointRouteBuilder**. Он определяет ряд методов для добавления конечных точек в приложение. И поскольку класс **WebApplication** также реализует данный интерфейс, то соответственно все методы интерфейса мы можем вызывать и у объекта **WebApplication**.

Для использования системы маршрутизации в конвейер обработки запроса добавляются два встроенных компонента **middleware**:

Microsoft.AspNetCore.Routing.EndpointMiddleware добавляет в конвейер обработки запроса конечные точки. Добавляется в конвейер с помощью метода **UseEndpoints()**.

Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware добавляет в конвейер обработки запроса функциональность сопоставления запросов и маршрутов. Данный **middleware** выбирает конечную точку, которая соответствует запросу, и которая затем обрабатывает запрос. Добавляется в конвейер с помощью метода **UseRouting()**.

Причем обычно не требуется явным образом подключать эти два компонента **middleware**. Объект **WebApplicationBuilder** автоматически сконфигурирует конвейер таким образом, что эти два **middleware** добавляются при использовании конечных точек.

Метод Map

Самым простым способом определения конечной точки в приложении является метод **Map**, который реализован как метод расширения для типа **IEndpointRouteBuilder**. Он добавляет конечные точки для обработки запросов типа **GET**. Данный метод имеет три версии:

```
public static RouteHandlerBuilder Map (this IEndpointRouteBuilder endpoints, RoutePattern pattern, Delegate handler);
public static IEndpointConventionBuilder Map (this IEndpointRouteBuilder endpoints, string pattern, RequestDelegate requestDelegate);
public static RouteHandlerBuilder Map (this IEndpointRouteBuilder endpoints, string pattern, Delegate handler);
```

Во всех трех реализациях этот метод в качестве параметра **pattern** принимает шаблон маршрута, которому должен соответствовать запрос. Данный параметр может представлять тип **RoutePattern** или **string**.

Последний параметр представляет действие, которое будет обрабатывать запрос. Это может быть делегат типа **RequestDelegate**, либо делегат **Delegate**.

Стоит отметить, что не стоит путать этот метод с одноименным методом **Map()**, который реализован как метод расширения для типа **IApplicationBuilder**.

Например, определим следующее приложение:

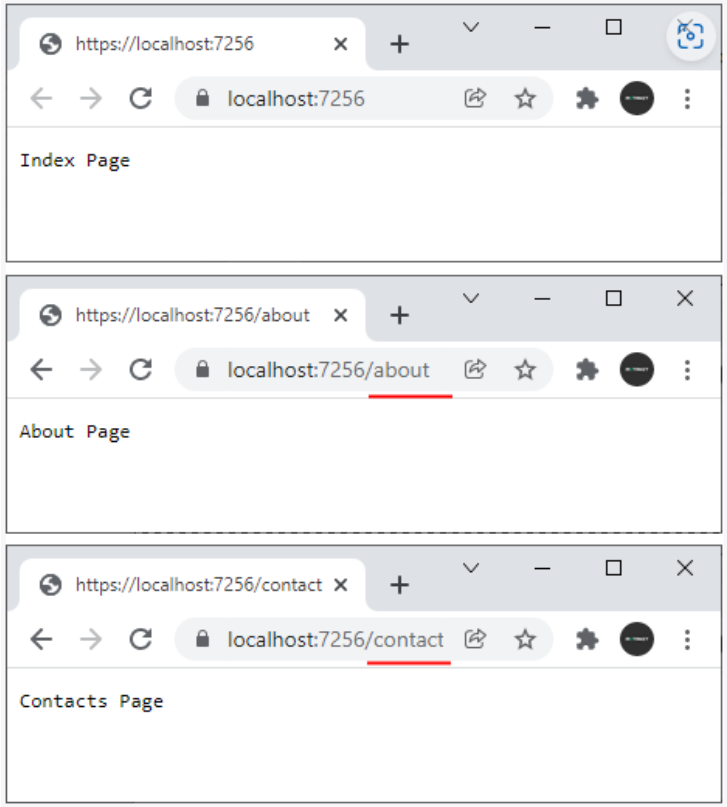
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

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

app.Run();
```

Здесь определено три конечных точки с помощью трех методов **app.Map()**. Первый вызов добавляет конечную точку, которая будет обрабатывать запрос по пути **"/"**. В качестве обработчика выступает действие

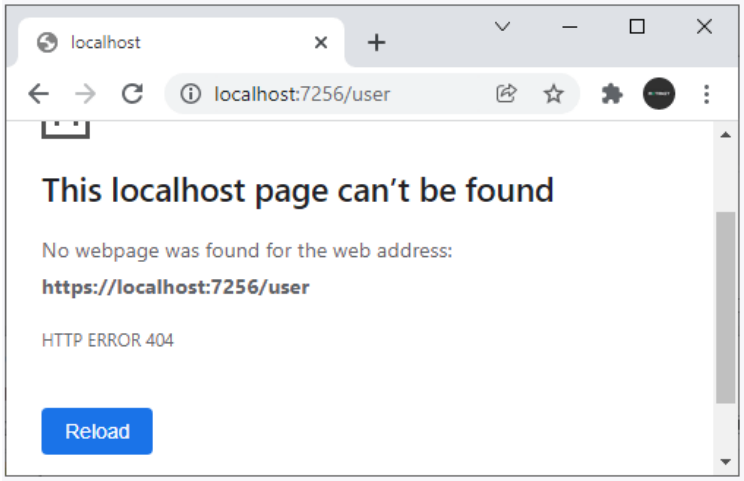
```
() => "Index Page"
```

Результат этого действия – строка "Index Page" – это то, что будет отправляться в ответ клиенту.

Аналогично второй и третий вызовы метода **Map** добавляют конечные точки для обработки запросов по путям "/about" и "/contact":



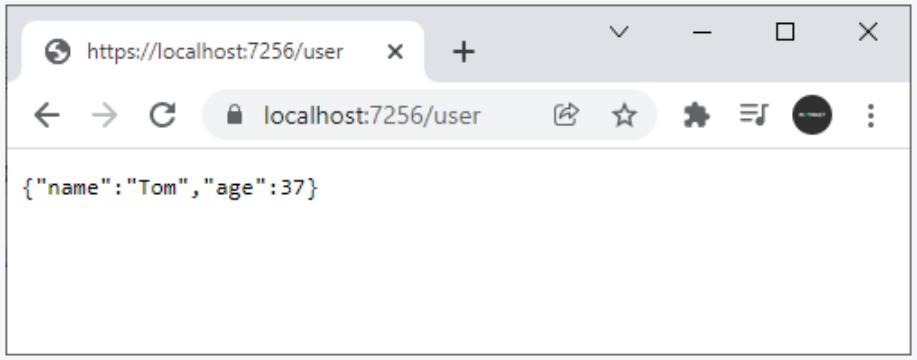
Если путь запроса не соответствует ни одной из конечных точек, то приложение отправит ошибку 404:



Выше в примере обработчики маршрутов возвращали строки, но в принципе это может быть любое значение, например:

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

app.Map("/", () => "Index Page");
app.Map("/user", () => new Person("Tom", 37));


app.Run();

record class Person(string Name, int Age);
```

Здесь обработчик запроса второй конечной точки возвращает в ответ объект **Person**. По умолчанию подобные объекты при отправке сериализируются в **JSON**:



В принципе можно ничего из обработчика не возвращать, просто выполнять некоторые действия:

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

app.Map("/", () => "Index Page");
app.Map("/user", ()=>Console.WriteLine("Request Path: /user"));

app.Run();
```

В данном случае в обработчике второй конечной точки просто логгируем на консоль некоторую информацию.

При необходимости обработчик маршрута можно вынести в полноценный метод:

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

app.Map("/", IndexHandler);
app.Map("/user", UserHandler);



app.Run();

string IndexHandler()
{
    return "Index Page";
}
Person UserHandler()
{
    return new Person("Tom", 37);
}
record class Person(string Name, int Age);
```

В примерах выше обработчик запроса представлял делегат **Delegate**. Если же необходимо получить полный доступ к контексту **HttpContext**, то можно использовать другую версию метода, которая принимает делегат **RequestDelegate**:

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

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

Получение всех маршрутов приложения

ASP.NET Core позволяет легко получить все имеющиеся в приложении конечные точки. Так, определим следующий код:

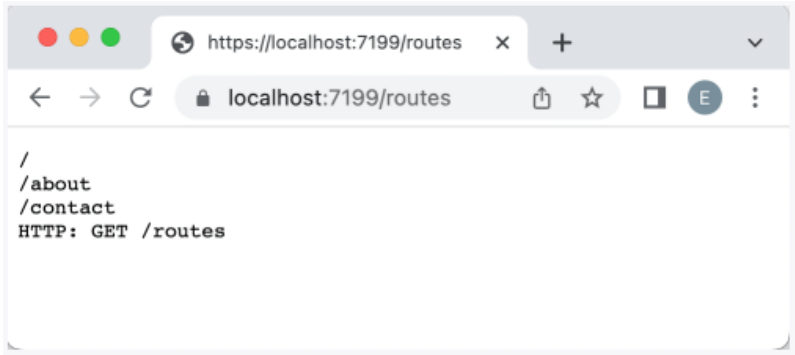
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

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

app.MapGet("/routes", (IEnumerable<EndpointDataSource> endpointSources) =>
        string.Join("\n", endpointSources.SelectMany(source => source.Endpoints)));

app.Run();
```

Здесь определены четыре конечных точки. Три первых конечных точки стандартные. Поэтому рассмотрим четвертую конечную точку, которая обрабатывает запросы по маршруту "/routes" и которая и будет выводить список всех конечных точек.

Через механизм внедрения зависимостей в обработчик маршрута четвертой конечной точки передается объект **IEnumerable<EndpointDataSource>** – некоторый набор данных о конечных точках. Каждый отдельный элемент этого набора – объект **EndpointDataSource**, который хранит набор конечных точек в свойстве-списке **Endpoints**. Каждая конечная точка в этом списке представляет класс **Endpoint**.

С помощью метода **endpointSources.SelectMany()** выбираем из коллекции **Endpoints** все конечные точки. С помощью метода **Join()** они склеиваются в одну строку и разделяются переводом строки \n.

В итоге мы увидим в браузере список из четырех конечных точек:



При необходимости можно получить более детальную и подробную информацию по каждой конечной точке:

```
using System.Text;

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

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

app.MapGet("/routes", (IEnumerable<EndpointDataSource> endpointSources) =>
{
    var sb = new StringBuilder();
    var endpoints = endpointSources.SelectMany(es => es.Endpoints);
    foreach (var endpoint in endpoints)
    {
        sb.AppendLine(endpoint.DisplayName);

        // получим конечную точку как RouteEndpoint
        if (endpoint is RouteEndpoint routeEndpoint)
        {
            sb.AppendLine(routeEndpoint.RoutePattern.RawText);
        }

        // получение метаданных
        // данные маршрутизации
        // var routeNameMetadata = endpoint.Metadata.OfType<Microsoft.AspNetCore.Routing.RouteNameMetadata>().FirstOrDefault();
        // var routeName = routeNameMetadata?.RouteName;
        // данные http - поддерживаемые типы запросов
        //var httpMethodsMetadata = endpoint.Metadata.OfType<HttpMethodMetadata>().FirstOrDefault();
        //var httpMethods = httpMethodsMetadata?.HttpMethods; // [GET, POST, ...]
    }
    return sb.ToString();
});

app.Run();
```

2. Параметры маршрута

Шаблон маршрута, который сопоставляется с конечной точкой, может иметь параметры. Параметры имеют имя и определяются в шаблоне маршрута внутри фигурных скобок: {название_параметра}.

Например, определим конечную точку, которая принимает параметр:

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

app.Map("/users/{id}", (string id) => $"User Id: {id}");
app.Map("/users", () => "Users Page");
app.Map("/", () => "Index Page");

app.Run();
```

Здесь определено три конечных точки, которые используются шаблоны маршрутов "/users/{id}", "/users" и "/". Причем шаблон маршрута для первой конечной точки принимает параметр с именем **id**:

```
"/users/{id}
```

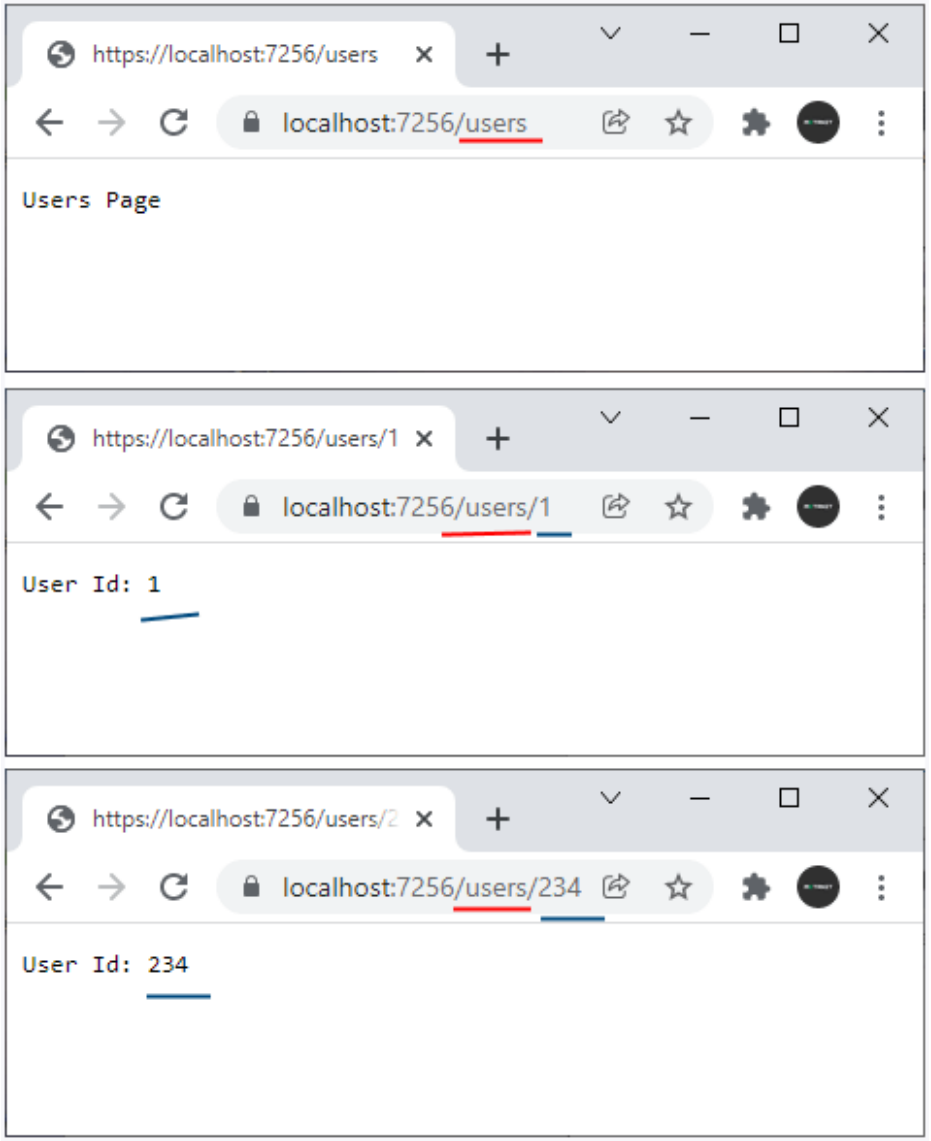
Параметр **id** будет представлять содержимое последнего сегмента. Например, в пути запроса "/users/134" последний сегмент "134" как раз будет представлять параметр **id**. Если же параметр **id** не передается, а путь запроса – "/users", то такой запрос обрабатывается второй конечной точкой.

Если маршрут принимает параметр, то в делегате-обработчике маршрута мы можем получить значение этого параметра. Для этого мы можем просто определить действие, которое принимает соответствующий параметр:

```
(string id) => $"User Id: {id}"
```

То есть здесь через параметр **id** мы получаем одноименный параметр маршрута и далее можем его использовать при обработке запроса. Стоит отметить, что название параметра маршрута и название параметра лямбда-выражения совпадают.

Также стоит отметить, что здесь для параметра **id** указан тип – тип **string** (то есть здесь мы ожидаем получить через параметр маршрута некоторую строку). Поскольку метод **Map** может принимать в качестве второго параметра делегат **RequestDelegate**, который тоже принимает один параметр – **HttpContext**, то, если не указать тип, то делегат-обработчик будет рассматриваться именно как **RequestDelegate**.

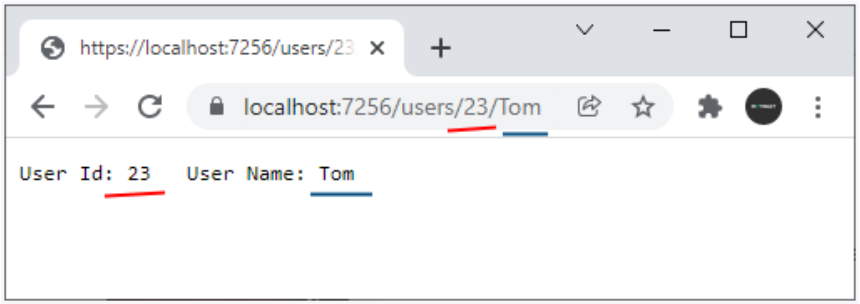


Определение нескольких параметров

Подобным образом можно определять и большее количество параметров маршрута:

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

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

app.Run();
```

В данном случае маршрут первой конечной точки принимает два параметра – **id** и **name**:



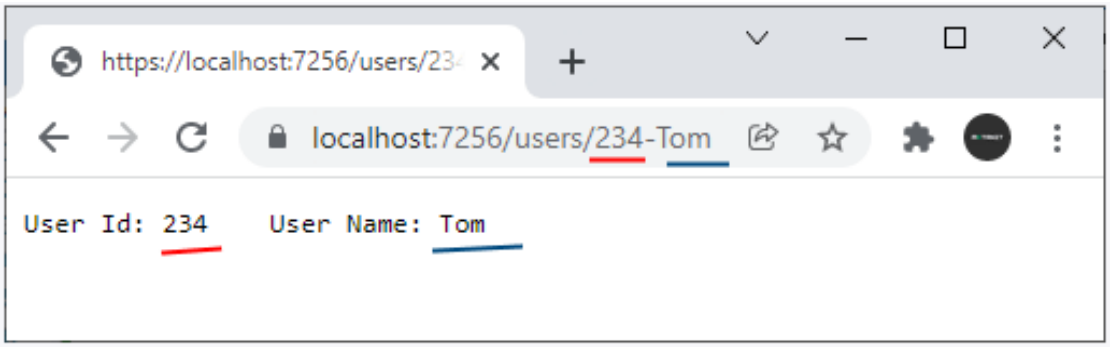
Следует учитывать, что если параметры маршрута определяются подряд, то между ними должен располагаться разделитель. Например, следующий шаблон маршрута работать не будет:

```
"/users/{id}{name}"
```

так как между параметрами **id** и **name** нет разделителя.

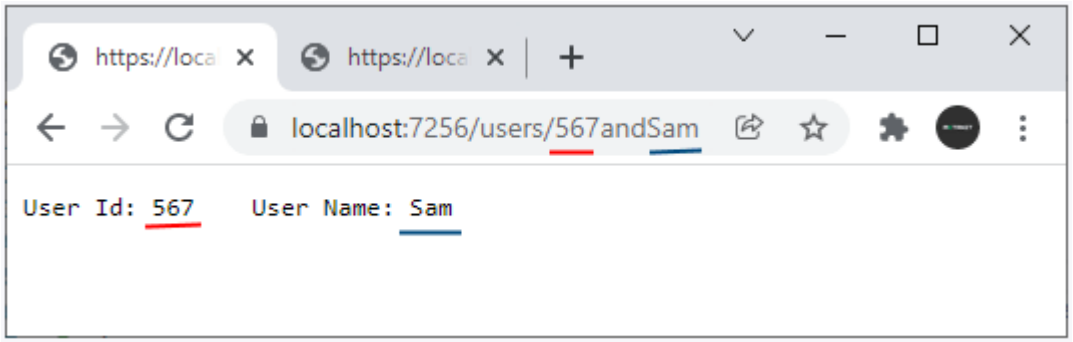
Обычно в качестве разделителя выступает слеш, который оформляет отдельный сегмент в пути запроса. Но это необязательно. В качестве разделителя могут применяться и другие символы. Например, установим в качестве разделителя дефис:

```
app.Map(
    "/users/{id}-{name}",
    (string id, string name) => $"User Id: {id}    User Name: {name}"
);
```



Более того это может быть набор символов:

```
app.Map(
    "/users/{id}and{name}",
    (string id, string name) => $"User Id: {id}    User Name: {name}"
);
```



Вынесение обработчика маршрута в отдельный метод

Если обработчик маршрута выносится в отдельный метод, то он также может определять параметры, который сопоставляются с параметрами маршрута по имени:

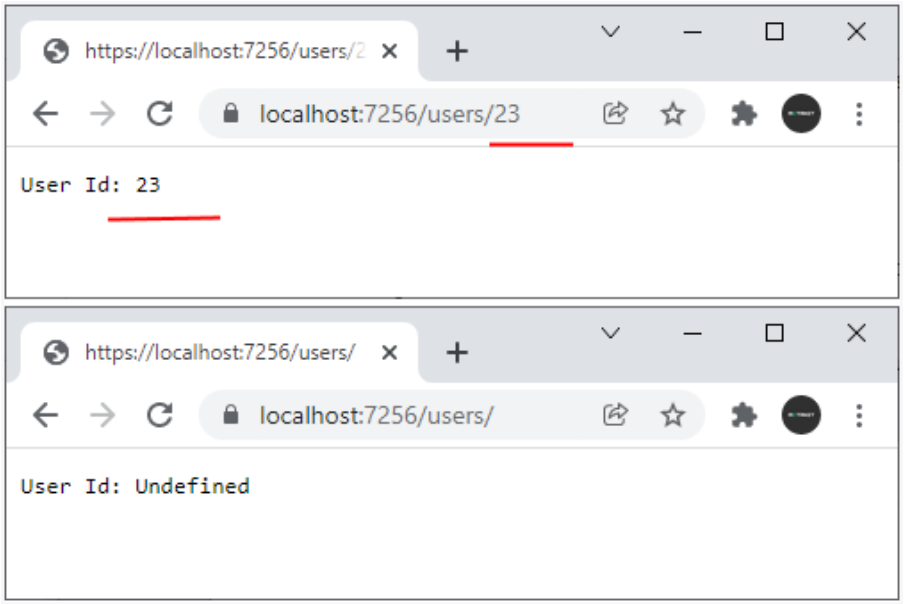
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/users/{id}/{name}", HandleRequest);
app.Map("/users", () => "Users Page");
app.Map("/", () => "Index Page");

app.Run();

string HandleRequest(string id, string name)
{
    return $"User Id: {id}   User Name: {name}";
}
```

Параметры маршрута могут быть необязательными. Чтобы определить параметр как необязательный, после его названия указывается знак вопроса. Например, определим следующее приложение:

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

app.Map("/users/{id?}", (string? id) => $"User Id: {id??"Undefined"}");
app.Map("/", () => "Index Page");

app.Run();
```

Шаблон маршрута первой конечной точки состоит из двух сегментов, причем второй сегмент представляет параметр **id**, который помечен как необязательный. А это значит, что мы можем в запросе игнорировать значение для этого сегмента. Например, данный шаблон будет соответствовать двум следующим url:

```
/users/
/users/23
```



Необязательные параметры следует помещать в конце шаблона маршрута, как в случае с параметром **id** в примере выше. То есть, к примеру, шаблон "/users/{id?}/{name}" не будет работать корректно, если мы для параметра **id** не передадим значение. А вот шаблон "/users/{name}/{id?}" будет работать нормально.

Значения параметров по умолчанию

Параметрам маршрута также можно назначить значения по умолчанию на случай, если им не переданы значения:

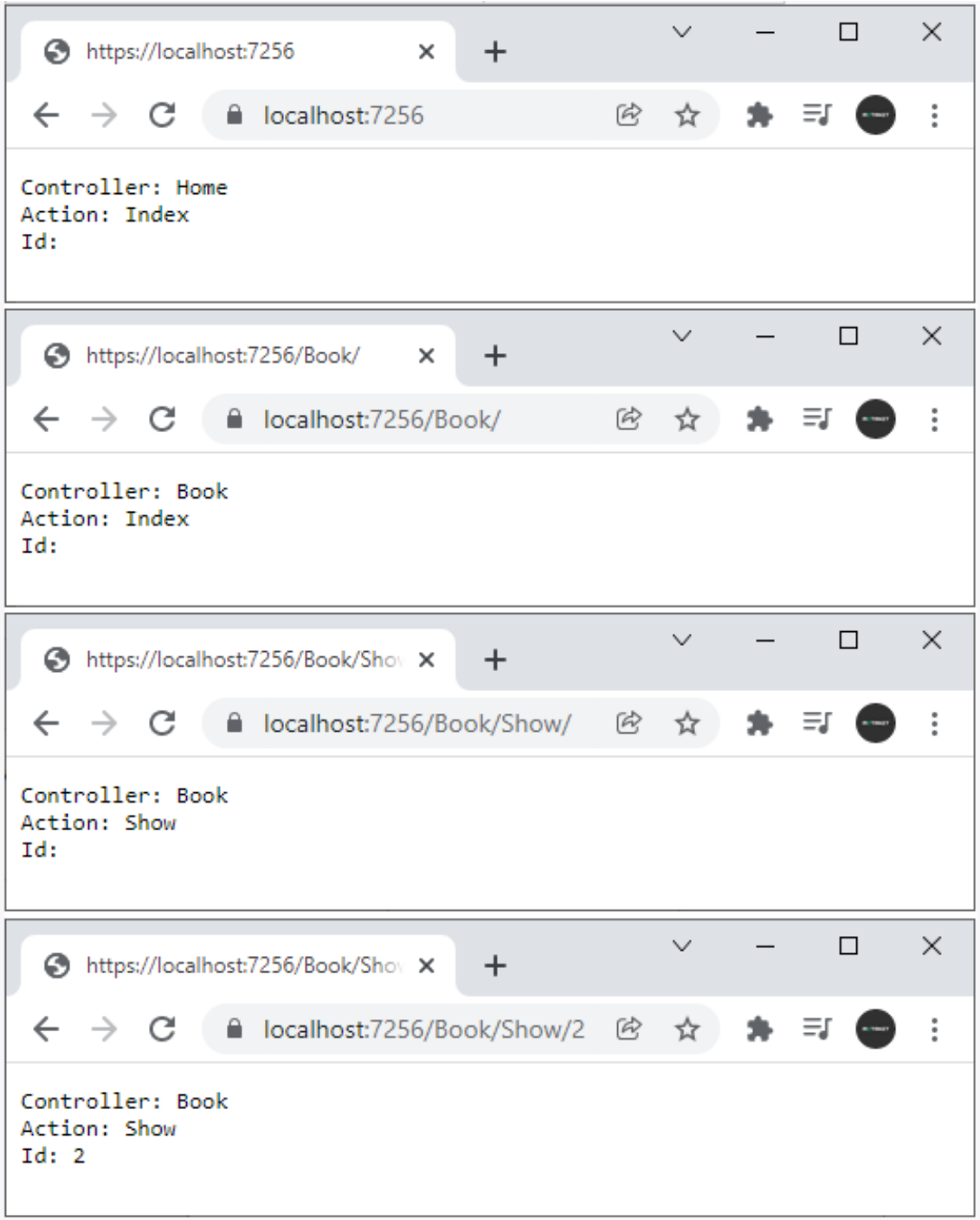
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map(
    "{controller=Home}/{action=Index}/{id?}",
    (string controller, string action, string? id) =>
        $"Controller: {controller} \nAction: {action} \nId: {id}"
);

app.Run();
```

Здесь определена одна конечная точка, которая использует следующий шаблон маршрута:

```
"{controller=Home}/{action=Index}/{id?}"
```

То есть шаблон состоит из трех параметров. Параметр **"controller"** имеет значение по умолчанию **"Home"**. Параметр **"action"** имеет значение по умолчанию **"Index"**. Параметр **"id"** определен как необязательный. В итоге при различных запросах у нас получатся следующие значения:

Запрос	Параметры запроса
https://localhost:7256/	controller=Home action=Index
https://localhost:7256/Book	controller=Book action=Index
https://localhost:7256/Book/Show	controller=Book action=Show
https://localhost:7256/Book/Show/2	controller=Book action=Show id=2

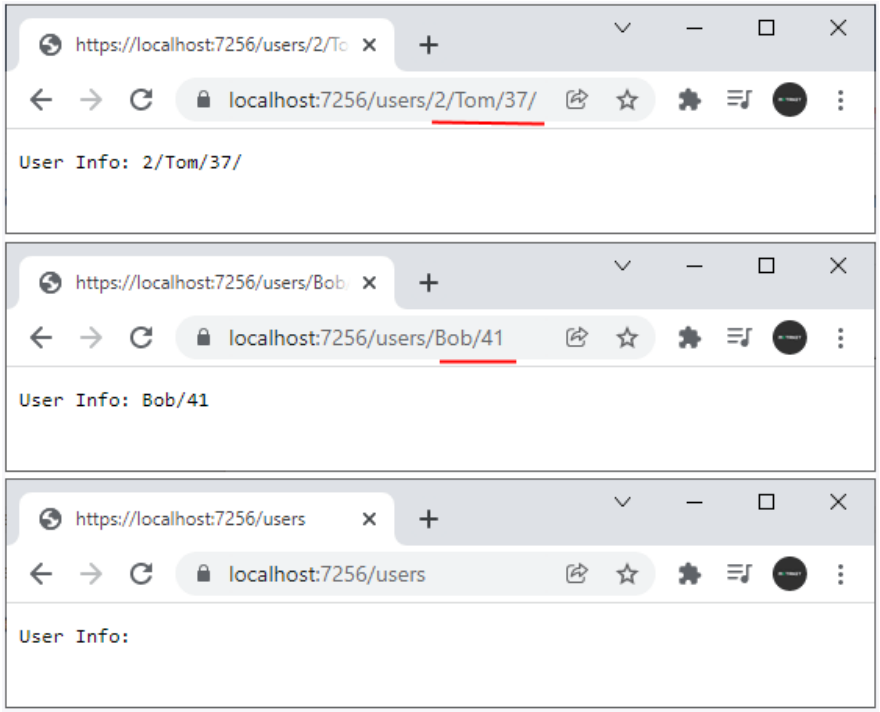

https://localhost:7256
localhost:7256
Controller: Home
Action: Index
Id:
https://localhost:7256/Book/
localhost:7256/Book/
Controller: Book
Action: Index
Id:
https://localhost:7256/Book/Sho
localhost:7256/Book/Show/
Controller: Book
Action: Show
Id:
https://localhost:7256/Book/Sho
localhost:7256/Book/Show/2
Controller: Book
Action: Show
Id: 2

Передача произвольного количества параметров в запросе

Мы можем обозначить любое количество сегментов в запросе, чтобы не быть жестко привязанным к числу сегментов с помощью параметра со знаком * ("звездочка") или ** (две звездочки) (это так называемый **catchall**-параметр):

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

app.Map("users/{**info}", (string info) =>$"User Info: {info}");

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

app.Run();
```

Для первой конечной точки определяется шаблон маршрута, в котором параметр **info** будет представлять все, что идет после "/users". Это может быть в том числе пустая строка.



3. Ограничения маршрутов

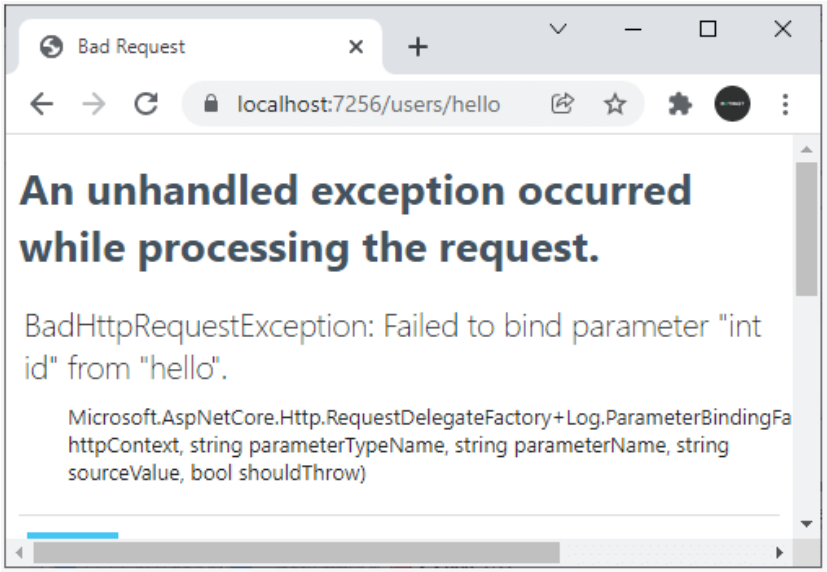
При обработке запроса система маршрутизации на основании шаблон маршрута автоматически извлекает из строки запроса значения для параметров маршрута вне зависимости от содержимого этих значений. Однако это не всегда бывает удобно. Например, мы хотим, чтобы какой-то параметр представлял только числа, а другой параметр начинался строго с определенного символа. И для этого необходимо задать ограничения маршрута (route constraints).

Ограничения маршрутов выполняются при парсинге пути запроса, сопоставлении его с шаблоном маршрута и выделении из него значений для параметров маршрута. Ограничения маршрутов применяются для разных задач. Прежде всего они решают, допустимо ли для параметра маршрута значение, которое выделено из пути запроса. Также ограничения маршрута могут решать, можно ли вообще сопоставить путь запроса с определенным маршрутом. Кроме того, ограничения маршрута могут применяться при генерации ссылок.

Вначале рассмотрим следующую ситуацию. Пусть у нас есть следующее приложение:

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

app.Map("/users/{id}", (int id) => $"User Id: {id}");
app.Map("/", () => "Index Page");

app.Run();
```

Первая конечная точка использует маршрут, который подразумевает наличие параметра **id**. А в обработчике маршрута мы хотим получить значение этого параметра в виде числа. То есть мы ожидаем, что в пути запроса будет передаваться число, например, "/users/123". Тем не менее мы можем передать вместо числа и строку:



На скриншоте выше обращение идет по адресу "/users/hello". И этот путь запроса в принципе соответствует шаблону маршрута "/users/{id}". В этом случае параметр маршрута **id** получит значение "hello", а для обработки запроса будет запущено действие

```
(int id) => $"User Id: {id}";
```

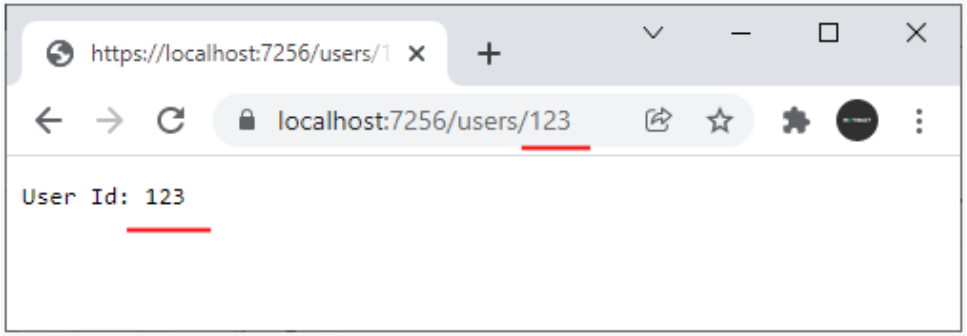
Однако параметр **id** обработчика представляет тип **int**, а строка "hello" никак не может быть конвертирована в число. Соответственно мы получим ошибку при преобразовании, о чем, собственно, и говорит скриншот.

Теперь применим ограничения – укажем, что параметр **id** в маршруте должен представлять тип **int**:

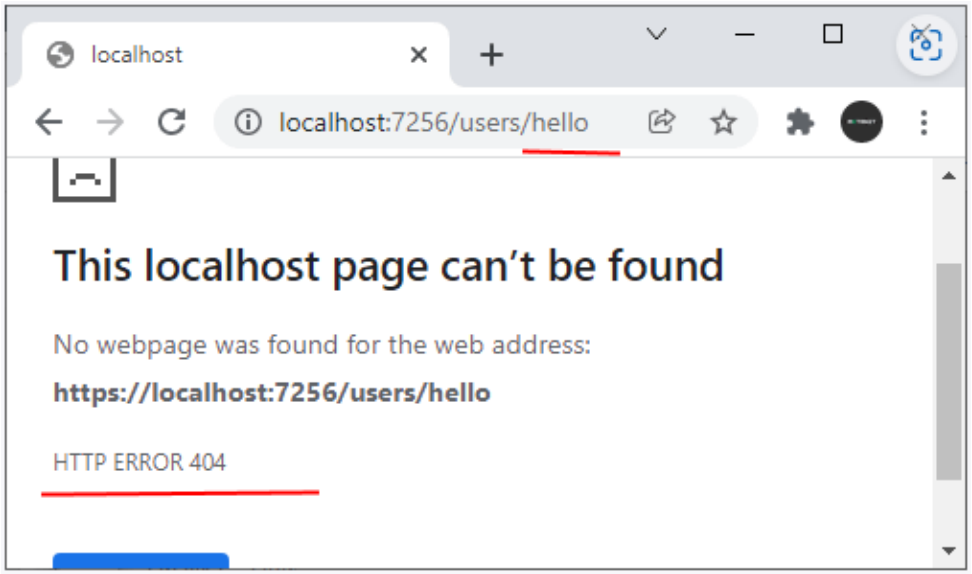
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/users/{id:int}", (int id) => $"User Id: {id}");
app.Map("/", () => "Index Page");

app.Run();
```

Для установки ограничения после названия параметра через двоеточие указывается название ограничения. Ограничение **int** указывает, что параметр должен представлять тип **int**.

В этом случае нам надо передать этому параметру число:



Да, мы по-прежнему можем передавать в пути запроса строку вместо числа:



Но в этом случае инфраструктура **ASP.NET Core** просто не сможет сопоставить данный путь запроса с шаблоном маршрута. И клиент получит ошибку 404, то есть ресурс не найден.

ASP.NET Core предоставляет следующий **ряд ограничений маршрута**:
int. Соответствие целому числу. Представляет класс **IntRouteConstraint**

```
{id:int}
```

bool. Представляет класс **BoolRouteConstraint**. Соответствие значению **true** или **false**

```
{active:bool}
```

datetime. Соответствие дате и времени. Представляет класс **DateTimeRouteConstraint**

```
{date:datetime}
```

decimal. Соответствие значению **decimal**. Представляет класс **DecimalRouteConstraint**

```
{price:decimal}
```

double. Соответствие значению типа **double**. Представляет класс **DoubleRouteConstraint**

```
{weight:double}
```

float. Соответствие значению типа **float**. Представляет класс **FloatRouteConstraint**

```
{height:float}
```

guid. Соответствие значению типа **Guid**. Представляет класс **GuidRouteConstraint**

```
{id:guid}
```

long. Соответствие значению типа **long**. Представляет класс **LongRouteConstraint**

```
{id:long}
```

minlength(value). Строка должна иметь символов не меньше **value**. Представляет класс **MinLengthRouteConstraint**

```
{name:minlength(3)}
```

maxlength(value). Строка должна иметь символов не больше **value**. Представляет класс **MaxLengthRouteConstraint**

```
{name:maxlength(20)}
```

length(value). Строка должна иметь ровно столько символов, сколько определено в параметре **value**. Представляет класс **LengthRouteConstraint**

```
{name:length(10)}
```

length(min, max). Строка должна иметь символов не меньше **min** и не больше **max**. Представляет класс **LengthRouteConstraint**

```
{name:length(3, 20)}
```

min(value). Число должно быть не меньше **value**. Представляет класс **MinRouteConstraint**

```
{age:min(3)}
```

max(value). Число должно быть не больше **value**. Представляет класс **MaxRouteConstraint**

```
{age:max(20)}
```

range(min, max). Число должно быть не меньше **min** и не больше **max**. Представляет класс **RangeRouteConstraint**

```
{age:range(18, 99)}
```

alpha. Строка должна состоять из одного и более алфавитных символов. Представляет класс **AlphaRouteConstraint**

```
{name:alpha}
```

regex(expression). Строка должна соответствовать регулярному выражению **expression**. Представляет класс **RegexRouteConstraint**

```
{phone:regex(^\d{{3}}-\d{{3}}-\d{{4}}$)}
```

required. Параметр является обязательным, и его значение должно быть определено. Представляет класс **RequiredRouteConstraint**

```
{name:required}
```

Каждое подобное ограничение представляет определенный класс. Все классы ограничений находятся в пространстве имен **Microsoft.AspNetCore.Routing.Constraints**.

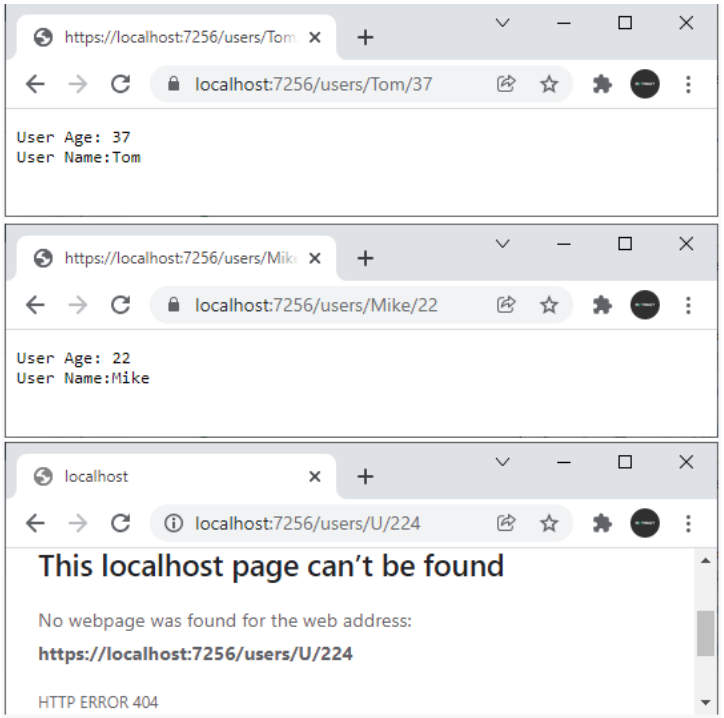
Ограничения можно комбинировать. При применении нескольких ограничений одновременно, они отделяются друг о друга двоеточием Например:

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

app.Map(
    "/users/{name:alpha:minlength(2)}/{age:int:range(1, 110)}",
    (string name, int age) => $"User Age: {age} \nUser Name:{name}"
);
app.Map(
    "/phonebook/{phone:regex(^7-\\d{{3}}-\\d{{3}}-\\d{{4}}$)}/",
    (string phone) => $"Phone: {phone}"
);
app.Map("/", () => "Index Page");


app.Run();
```

Первая конечная точка использует шаблон маршрута

```
"/users/{name:alpha:minlength(2)}/{age:int:range(1, 110)}"
```

Здесь предполагается, что параметр **name** принимает только алфавитные символы, а его минимальная длина должна представлять два символа. Второй же параметр маршрута – **age** должен представлять целое число и должен находиться в диапазоне между 1 и 110:



Вторая конечная точка использует шаблон маршрута

```
"/phonebook/{phone:regex(^7-\\d{{3}}-\\d{{3}}-\\d{{4}}$)}"
```

Параметр **phone** принимает номер телефона в формате 7-xxx-xxx-xxxx, любые другие форматы будут некорректными: