Распределенные информационно-аналитические системы

Практическое занятие № 12. «Состояние приложения.

Куки. Сессии. Обработка ошибок»

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

Учебные вопросы:

1. **Состояние приложения. Куки. Сессии.**
    - 1.1. HttpContext.Items.
    - 1.2. Куки.
    - 1.3. Сессии.
2. **Обработка ошибок.**
    - 2.1. Обработка исключений.
    - 2.2. Обработка ошибок HTTP.

1. Состояние приложения. Куки. Сессии

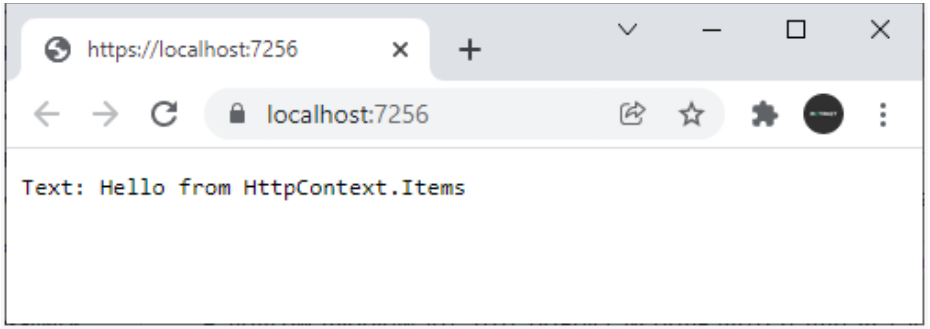
1.1. HttpContext.Items

Приложение **ASP.NET Core** может хранить некоторое состояние. Это могут быть как какие-то глобальные данные, так и данные, которые непосредственно относятся к запросу и пользователю. И в зависимости от вида данных, существуют различные способы для их хранения.

Один из простейших способов хранения данных представляет коллекция **HttpContext.Items** – объект типа **IDictionary<object, object>**. Эта коллекция предназначена для таких данных, которые непосредственно связаны с текущим запросом. После завершения запроса все данные из **HttpContext.Items** удаляются. Каждый объект в этой коллекции имеет ключ и значение. И с помощью ключей можно управлять объектами коллекции.

В каком случае мы можем применить данную коллекцию? Например, если у нас обработка запроса вовлекает множество компонентов **middleware**, и мы хотим, чтобы для этих компонентов были доступны общие данные, то как раз можем применить эту коллекцию. Например, пусть в приложении определен следующий код:

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

app.Use(async (context, next) =>
{
    context.Items["text"] = "Hello from HttpContext.Items";
    await next.Invoke();
});

app.Run(async (context) => await context.Response.WriteAsync($"Text: {context.Items["text"]}"));

app.Run();
```



Здесь в одном middleware определяется ключ **"text"** со значением "Hello from HttpContext.Items":

```
context.Items["text"] = "Hello from HttpContext.Items";
```

В другом **middleware** этот объект используется для установки отправляемого ответа.

HttpContext.Items предоставляет **ряд методов для управления элементами**:

void Add(object key, object value): добавляет объект value с ключом key.

void Clear(): удаляет все объекты.

bool ContainsKey(object key): возвращает **true**, если словарь содержит объект с ключом **key**.

bool Remove(object key): удаляет объект с ключом **key**, в случае удачного удаления возвращает **true**.

bool TryGetValue(object key, out object value): возвращает **true**, если значение объекта с ключом **key** успешно получено в объект **value**.

Применение некоторых методов:

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

app.Use(async (context, next) =>
{
    context.Items.Add("message", "Hello METANIT.COM");
    await next.Invoke();
});

app.Run(async (context) =>
{
    if (context.Items.ContainsKey("message"))
        await context.Response.WriteAsync($"Message: {context.Items["message"]}");
    else
        await context.Response.WriteAsync("Random Text");
});

app.Run();
```

1.2. Куки

Куки представляют самый простой способ сохранить данные пользователя. Куки хранятся на компьютере пользователя и могут устанавливаться как на сервере, так и на клиенте. Так как куки посылаются с каждым запросом на сервер, то их максимальный размер ограничен 4096 байтами.

Для работы с куками также можно использовать контекст запроса **HttpContext**, который передается в качестве параметра в компоненты **middleware**, а также доступен в контроллерах и **RazorPages**.

Чтобы получить куки, которые приходят вместе с запросом к приложению, нам надо использовать коллекцию **Request.Cookies** объекта **HttpContext**. Эта коллекция представляет объект **IRequestCookieCollection**, в котором каждый элемент – это объект **KeyValuePair<string, string>**, то есть некоторую пару ключ-значение.

Для этой коллекции определено несколько **методов**:

bool ContainsKey(string key): возвращает **true**, если в коллекции кук есть куки с ключом **key**.

bool TryGetValue(string key, out string value): возвращает **true**, если удалось получить значение куки с ключом **key** в переменную **value**.

Стоит отметить, что куки – это строковые значения. Неважно, что вы пытаетесь сохранить в куки – все это необходимо приводить к строке и соответственно получаете из кук вы тоже строки.

Например, получим куку **"name"**:

```
1  if (context.Request.Cookies.ContainsKey("name"))
2      string name = context.Request.Cookies["name"];
```

Повторюсь, что коллекция **context.Request.Cookies** служит только для получения значений кук.

Для установки кук, которые отправляются в ответ клиенту, применяется объект **context.Response.Cookies**, который представляет интерфейс **IResponseCookies**. Этот интерфейс определяет **два метода**:

Append(string key, string value): добавляет для куки с ключом **key** значение **value**.

Delete(string key): удаляет куку по ключу.

Например, объединим в приложении установку и получение кук:

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

app.Run(async (context) =>
{
    if (context.Request.Cookies.ContainsKey("name"))
    {
        string? name = context.Request.Cookies["name"];
        await context.Response.WriteAsync($"Hello {name}!");
    }
    else
    {
        context.Response.Cookies.Append("name", "Tom");
        await context.Response.WriteAsync("Hello World!");
    }
});

app.Run();
```

При получении запроса мы смотрим, установлена ли кука **"name"**:

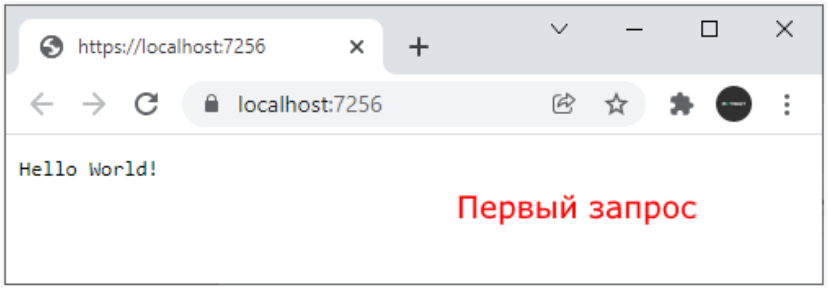
```
if (context.Request.Cookies.ContainsKey("name"))
```

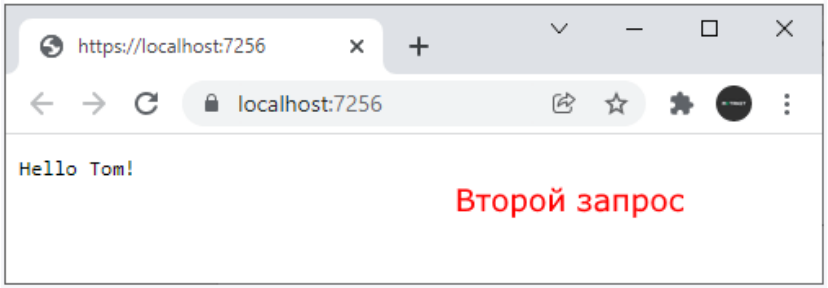
Если кука не установлена (например, при первом обращении к приложению), то устанавливаем ее и отправляем пользователю в ответ строку "Hello World!".

```
context.Response.Cookies.Append("name", "Tom");
await context.Response.WriteAsync("Hello World!");
```

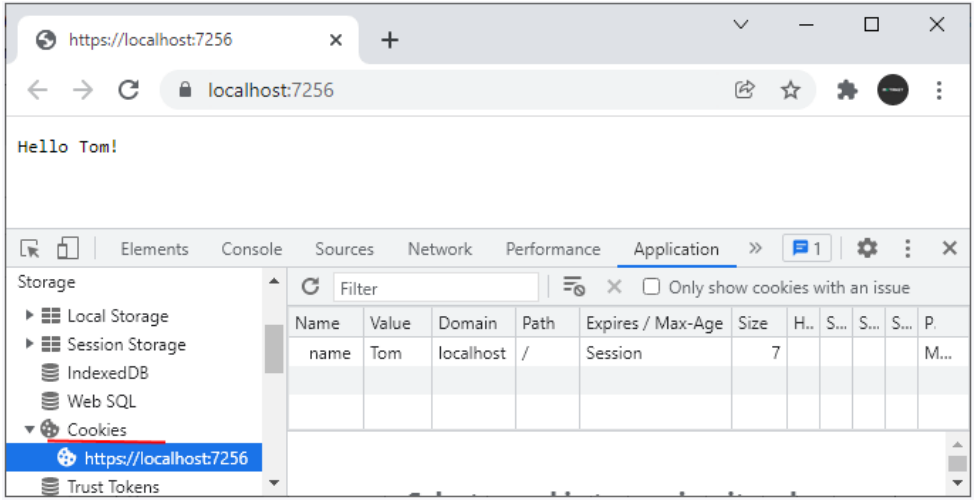
Если кука установлена, то получаем ее значение и отправляем его пользователю

```
string? name = context.Request.Cookies["name"];
await context.Response.WriteAsync($"Hello {name}!");
```





После установки кук в результате первого запроса в браузере будет установлена кука **name**. И через средства разработчика в веб-браузере мы сможем увидеть ее значение (если браузер поддерживает подобное). Например, в Google Chrome посмотреть куки можно на вкладке **Application -> Cookies**:



В примере выше мы также могли бы применить метод **TryGetValue()** для получения кук:

```
app.Run(async (context) =>
{
    if (context.Request.Cookies.TryGetValue("name", out var login))
    {
        await context.Response.WriteAsync($"Hello {login}!");
    }
    else
    {
        context.Response.Cookies.Append("name", "Tom");
        await context.Response.WriteAsync("Hello World!");
    }
});
```

Стоит отметить, что пользователь через подобные инструменты (если веб-браузер позволяет) может вручную поменять значение кук, либо вовсе удалить их.

1.3. Сессии

Сессия представляет собой ряд последовательных запросов, совершенных в одном браузере в течение некоторого времени. Сессия может использоваться для сохранения каких-то временных данных, которые должны быть доступны, пока пользователь работает с приложением, и не требуют постоянного хранения.

Для хранения состояния сессии на сервере создается словарь или хеш-таблица, которая хранится в кэше и которая существует для всех запросов из одного браузера в течение некоторого времени. На клиенте хранится идентификатор сессии в куках. Этот идентификатор посылается на сервер с каждым запросом. Сервер использует этот идентификатор для извлечения нужных данных из сессии. Эти куки удаляются только при завершении сессии. Но если сервер получает куки, которые установлены уже для истекшей сессии, то для этих кук создается новая сессия.

Сервер хранит данные сессии в течение ограниченного промежутка времени после последнего запроса. По умолчанию этот промежуток равен 20 минутам, хотя его также можно изменить.

Следует учитывать, что данные сессии специфичны для одного браузера и не разделяются между браузерами. То есть для каждого браузера на одном компьютере будет создаваться свой набор данных.

Все сессии работают поверх объекта **IDistributedCache**, и **ASP.NET Core** предоставляет встроенную реализацию **IDistributedCache**, которую мы можем использовать. Для этого определим в приложении следующий код:

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

builder.Services.AddDistributedMemoryCache();// добавляем IDistributedMemoryCache
builder.Services.AddSession();  // добавляем сервисы сессии

var app = builder.Build();

app.UseSession();   // добавляем middleware для работы с сессиями

app.Run(async (context) =>
{
    if (context.Session.Keys.Contains("name"))
        await context.Response.WriteAsync($"Hello {context.Session.GetString("name")}!");
    else
    {
        context.Session.SetString("name", "Tom");
        await context.Response.WriteAsync("Hello World!");
    }
});

app.Run();
```

Сначала добавляем необходимые сервисы:

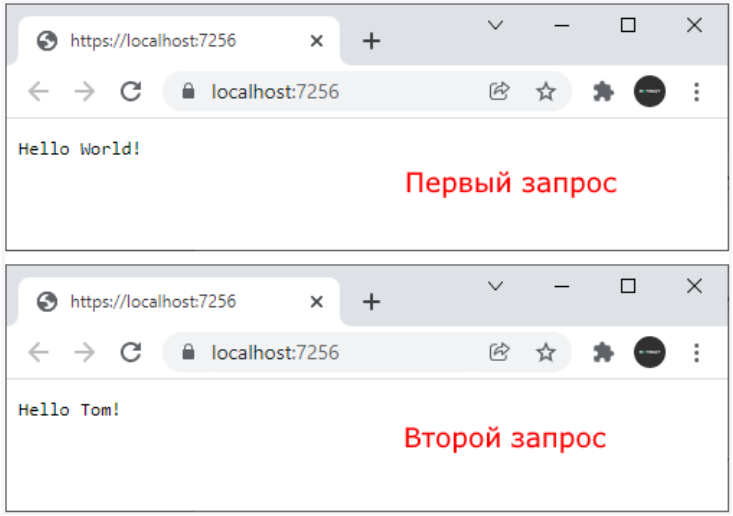
```
builder.Services.AddDistributedMemoryCache();// добавляем IDistributedMemoryCache
builder.Services.AddSession();  // добавляем сервисы сессии
```

Затем с помощью метода **UseSession()** встраиваем в конвейер обработки запроса **middleware** для работы с сессиями:

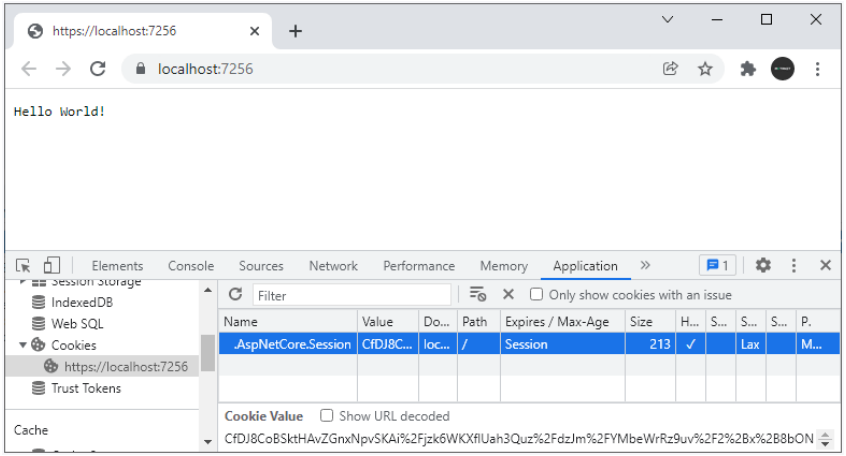
```
app.UseSession();   // добавляем middleware для работы с сессиями

app.Run(async (context) =>
{
    if (context.Session.Keys.Contains("name"))
        await context.Response.WriteAsync($"Hello {context.Session.GetString("name")}!");
    else
    {
        context.Session.SetString("name", "Tom");
        await context.Response.WriteAsync("Hello World!");
    }
});
```

Если мы вдруг не используем **app.UseSession()** или попробуем обратиться к сессии до применения этого метода, то получим исключение **InvalidOperationException**.

После вызова **app.UseSession()** мы сможем управлять сессиями через свойство **HttpContext.Session**, которое представляет объект интерфейса **ISession**. В данном случае мы проверяем, определен ли в сессии ключ **"name"**. Если ключ определен, то передаем ответ значение по этому ключу. Если ключ не определен, устанавливаем его.



После первого запроса мы сможем через инструменты браузера для разработчиков найти куку **.AspNetCore.Session**, которая будет представлять идентификатор сессии:



Управление сессией

Объект **ISession** определяет ряд свойств и методов, которые мы можем использовать:

Keys: свойство, представляющее список строк, который хранит все доступные ключи.

Clear(): очищает сессию.

Get(string key): получает по ключу **key** значение, которое представляет массив байтов.

GetInt32(string key): получает по ключу **key** значение, которое представляет целочисленное значение.

GetString(string key): получает по ключу **key** значение, которое представляет строку.

Set(string key, byte[] value): устанавливает по ключу **key** значение, которое представляет массив байтов.

SetInt32(string key, int value): устанавливает по ключу **key** значение, которое представляет целочисленное значение **value**.

SetString(string key, string value): устанавливает по ключу **key** значение, которое представляет строку **value**.

Remove(string key): удаляет значение по ключу.

Настройка сессии

Для разграничения сессий для них устанавливается идентификатор. Каждая сессия имеет свой идентификатор, который сохраняется в куках. По умолчанию эти куки имеют название **".AspNet.Session"**. И также по умолчанию куки имеют настройку **CookieHttpOnly=true**, поэтому они не доступны для клиентских скриптов из браузера. Но мы можем переопределить ряд настроек сессии с помощью свойств объекта **SessionOptions**:

Cookie.Name: имя куки.

Cookie.Domain: домен, для которого устанавливаются куки.

Cookie.HttpOnly: доступны ли куки только при передаче через HTTP-запрос.

Cookie.Path: путь, который используется куками.

Cookie.Expiration: время действия куки в виде объекта **System.TimeSpan**.

Cookie.IsEssential: при значении **true** указывает, что куки критичны и необходимы для работы этого приложения.

IdleTimeout: время действия сессии в виде объекта **System.TimeSpan** при неактивности пользователя. При каждом новом запросе таймаут сбрасывается. Этот параметр не зависит от **Cookie.Expiration**.

Для использования этих свойств изменим установку сервисов:

```
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(options =>
{
    options.Cookie.Name = ".MyApp.Session";
    options.IdleTimeout = TimeSpan.FromSeconds(3600);
    options.Cookie.IsEssential = true;
});
```

Хранение сложных объектов

В случае выше в сессиях хранились простые строки. Если же надо сохранить какой-то сложный объект, то его надо сериализовать в строку, а при получении из сессии – обратно десериализовать. Как правило, для этого определяются методы расширения для объекта **ISession**. В частности, добавим следующий класс:

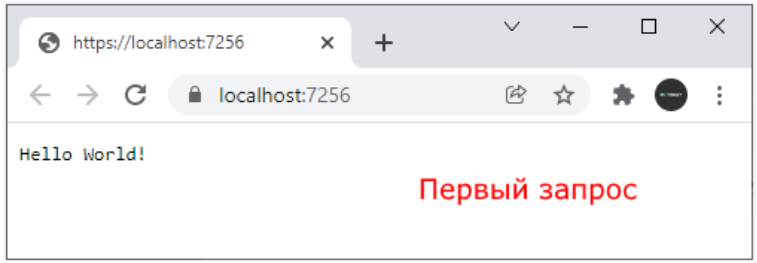
```
using System.Text.Json;

public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonSerializer.Serialize<T>(value));
    }

    public static T? Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return value == null ? default(T) : JsonSerializer.Deserialize<T>(value);
    }
}
```

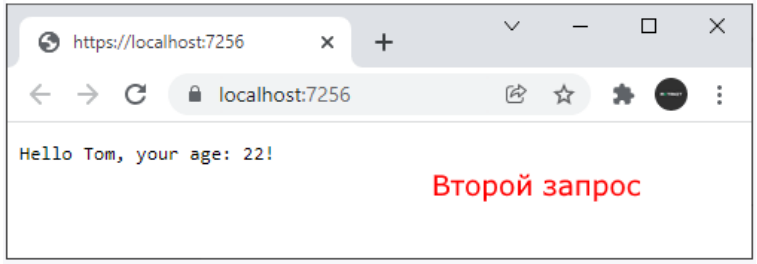
Метод **Set** сохраняет в сессию данные, а метод **Get** извлекает их из сессии.
Допустим, у нас есть класс **Person**:

```
class Person
{
    public string Name { get; set; } = "";
    public int Age { get; set; }
}
```

В приложении выполним сериализацию и десериализацию объекта в сессию:

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

builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();

var app = builder.Build();

app.UseSession();

app.Run(async (context) =>
{
    if (context.Session.Keys.Contains("person"))
    {
        Person? person = context.Session.Get<Person>("person");
        await context.Response.WriteAsync($"Hello {person?.Name}, your age: {person?.Age}!");
    }
    else
    {
        Person person = new Person { Name = "Tom", Age = 22 };
        context.Session.Set<Person>("person", person);
        await context.Response.WriteAsync("Hello World!");
    }
});

app.Run();
```





2. Обработка ошибок

2.1. Обработка исключений

Ошибки в приложении можно условно разделить на два типа: исключения, которые возникают в процессе выполнения кода (например, деление на 0), и стандартные ошибки протокола HTTP (например, ошибка 404).

Обычные исключения могут быть полезны для разработчика в процессе создания приложения, но простые пользователи не должны будут их видеть.

UseDeveloperExceptionPage

Для обработки исключений в приложении, которое находится в процессе разработки, предназначен специальный **middleware** – **DeveloperExceptionPageMiddleware**. Однако вручную нам его не надо добавлять, поскольку оно добавляется автоматически. Так, если мы посмотрим на код класса **WebApplicationBuilder**, который применяется для создания приложения, то мы там можем увидеть там следующие строки:

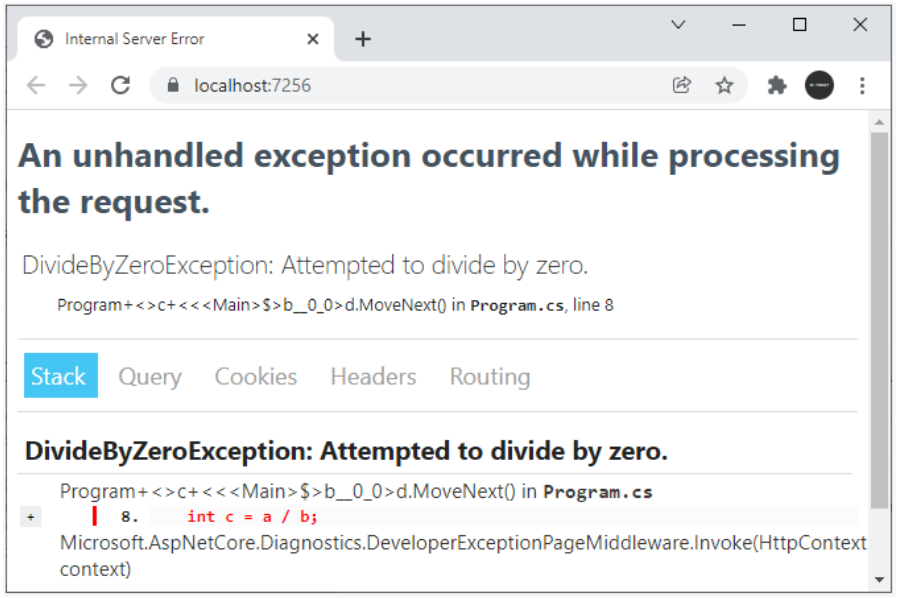
```
if (context.HostingEnvironment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
```

Если приложение находится в состоянии разработки, то с помощью **middleware app.UseDeveloperExceptionPage()** приложение перехватывает исключения и выводит информацию о них разработчику.

Например, изменим код приложения следующим образом:

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

app.UseDeveloperExceptionPage();

app.Run(async (context) =>
{
    int a = 5;
    int b = 0;
    int c = a / b;
    await context.Response.WriteAsync($"c = {c}");
});

app.Run();
```

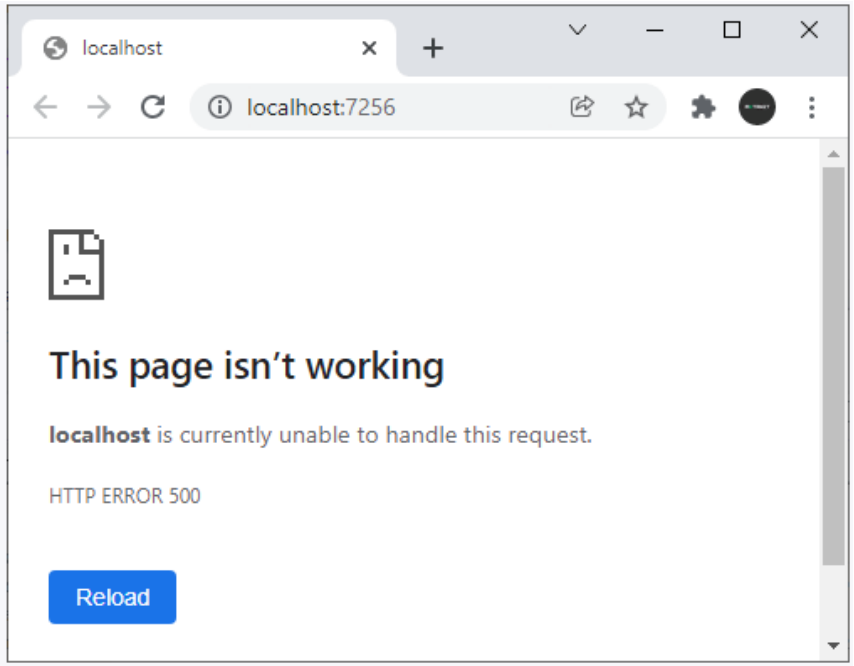
В **middleware** в методе **app.Run()** симулируется генерация исключения путем деления на ноль. И если мы запустим проект, то в браузере мы увидим информацию об исключении:



Этой информации достаточно, чтобы определить, где именно в коде произошло исключение.
Теперь посмотрим, как все это будет выглядеть для простого пользователя. Для этого изменим код приложения:

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

app.Environment.EnvironmentName = "Production"; // меняем имя окружения

app.Run(async (context) =>
{
    int a = 5;
    int b = 0;
    int c = a / b;
    await context.Response.WriteAsync($"c = {c}");
});

app.Run();
```

Выражение **app.Environment.EnvironmentName = "Production"** меняет имя окружения с **"Development"** (которое установлено по умолчанию) на **"Production"**. В этом случае среда **ASP.NET Core** больше не будет расценивать приложение как находящееся в стадии разработки. Соответственно **middleware** из метода **app.UseDeveloperExceptionPage()** не будет перехватывать исключение. А приложение будет возвращать пользователю ошибку 500, которая указывает, что на сервере возникла ошибка при обработке запроса. Однако реально природа подобной ошибки может заключать в чем угодно.



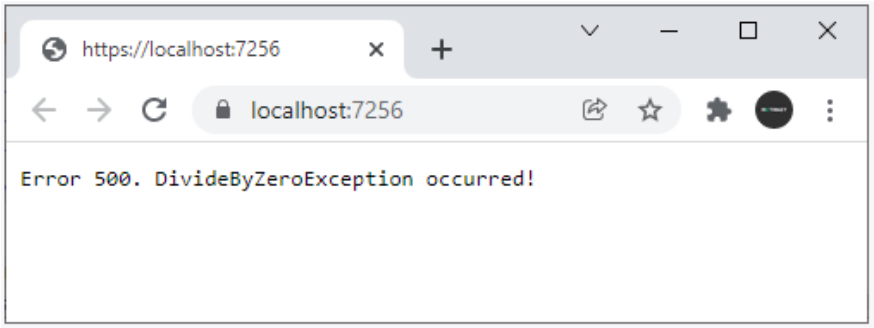
UseExceptionHandler

Это не самая лучшая ситуация, и нередко все-таки возникает необходимость дать пользователям некоторую информацию о том, что же все-таки произошло. Либо потребуется как-то обработать данную ситуацию. Для этих целей можно использовать еще один встроенный **middleware** – **ExceptionHandlerMiddleware**, который подключается с помощью метода **UseExceptionHandler()**. Он перенаправляет при возникновении исключения на некоторый адрес и позволяет обработать исключение. Например, изменим код приложения следующим образом:

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

app.Environment.EnvironmentName = "Production"; // меняем имя окружения

// если приложение не находится в процессе разработки
// перенаправляем по адресу "/error"
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
}

// middleware, которое обрабатывает исключение
app.Map("/error", app => app.Run(async context =>
{
    context.Response.StatusCode = 500;
    await context.Response.WriteAsync("Error 500. DivideByZeroException occurred!");
}));

// middleware, где генерируется исключение
app.Run(async (context) =>
{
    int a = 5;
    int b = 0;
    int c = a / b;
    await context.Response.WriteAsync($"c = {c}");
});

app.Run();
```

Метод **app.UseExceptionHandler("/error")**; перенаправляет при возникновении ошибки на адрес "/error".

Для обработки пути по определенному адресу здесь использовался метод **app.Map()**. В итоге при возникновении исключения будет срабатывать делегат из метода **app.Map()**, в котором пользователю будет отправляться некоторое сообщение со статусным кодом 500.



Однако данный способ имеет небольшой недостаток – мы можем напрямую обратиться по адресу **"/error"** и получить тот же ответ от сервера. Но в этом случае мы можем использовать другую версию метода **UseExceptionHandler()**, которая принимает делегат, параметр которого представляет объект **IApplicationBuilder**:

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

app.Environment.EnvironmentName = "Production";

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler(app => app.Run(async context =>
    {
        context.Response.StatusCode = 500;
        await context.Response.WriteAsync("Error 500. DivideByZeroException occurred!");
    }));
}

app.Run(async (context) =>
{
    int a = 5;
    int b = 0;
    int c = a / b;
    await context.Response.WriteAsync($"c = {c}");
});

app.Run();
```

Следует учитывать, что **app.UseExceptionHandler()** следует помещать ближе к началу конвейера **middleware**.

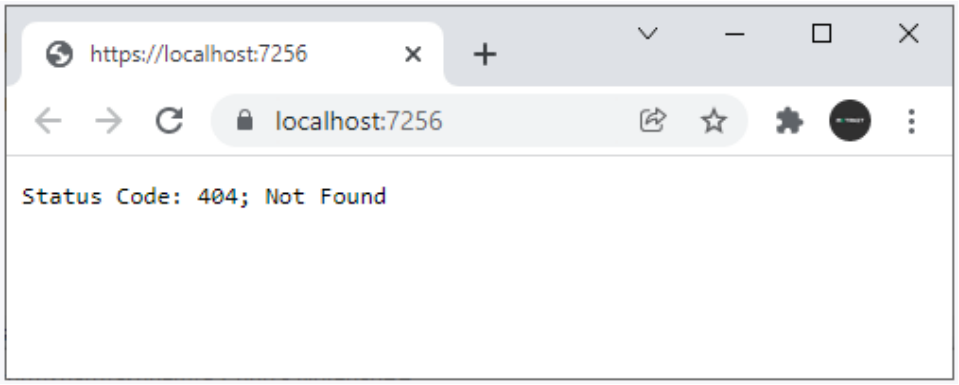
2.2. Обработка ошибок HTTP

В отличие от исключений стандартный функционал проекта **ASP.NET Core** почти никак не обрабатывает ошибки HTTP, например, в случае если ресурс не найден. При обращении к несуществующему ресурсу мы увидим в браузере пустую страницу, и только через консоль веб-браузера мы сможем увидеть статусный код. Либо браузер может отобразить какую-то стандартную страницу.

Но с помощью компонента **StatusCodePagesMiddleware** можно добавить в проект отправку информации о статусном коде. Для этого применяется метод **app.UseStatusCodePages()**:

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

// обработка ошибок HTTP
app.UseStatusCodePages();

app.Map("/hello", () => "Hello ASP.NET Core");

app.Run();
```

Здесь мы можем обращаться только по адресу "/hello". При обращении ко всем остальным адресам браузер отобразит базовую информацию об ошибке:



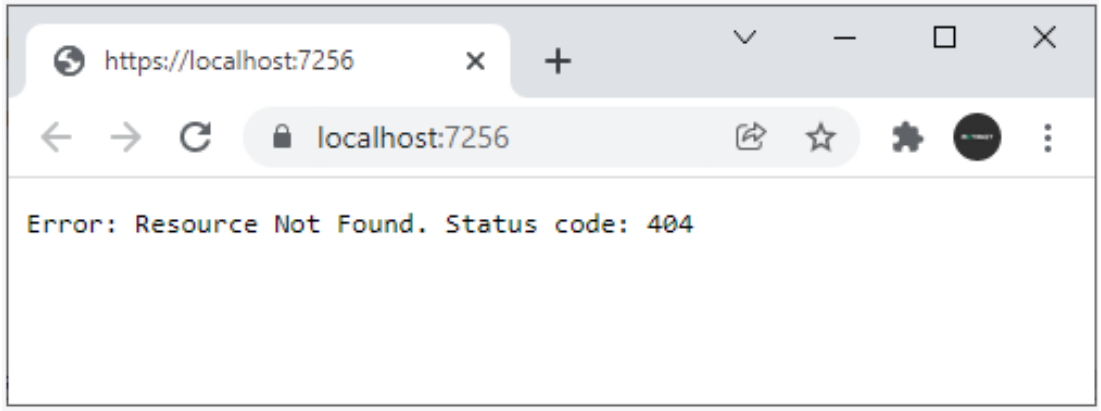
Метод **UseStatusCodePages()** следует вызывать ближе к началу конвейера обработки запроса, в частности, до добавления middleware для работы со статическими файлами и до добавления конечных точек.

Настройка сообщения

Сообщение, отправляемое методом **UseStatusCodePages()** по умолчанию, не очень информативное. Однако одна из версий метода позволяет настроить отправляемое пользователю сообщение. В частности, мы можем изменить вызов метода так:

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

// обработка ошибок HTTP
app.UseStatusCodePages("text/plain", "Error: Resource Not Found. Status code: {0}");

app.Map("/hello", () => "Hello ASP.NET Core");

app.Run();
```

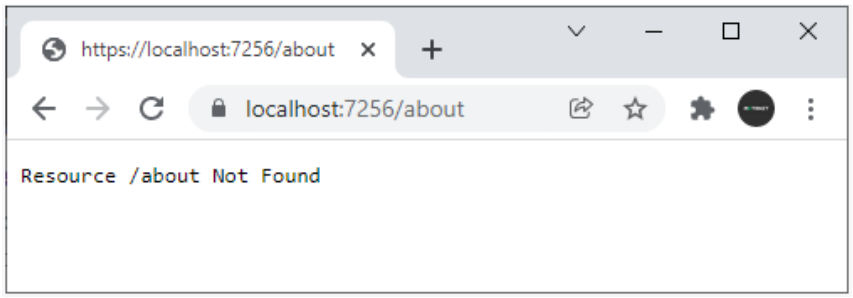
В качестве первого параметра указывается MIME-тип ответа – в данном случае простой текст (""text/plain""). В качестве второго параметра передается, собственно, то сообщение, которое увидит пользователь. В сообщение мы можем передать код ошибки через плейсхолдер "{0}".



Установка обработчика ошибок

Еще одна версия метода **UseStatusCodePages()** позволяет более детально задать обработчик ошибок. В частности, она принимает делегат, параметр которого – объект **StatusCodeContext**. В свою очередь, объект **StatusCodeContext** имеет свойство **HttpContext**, из которого мы можем получить всю информацию о запросе и настроить отправку ответа. Например:

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

// обработка ошибок HTTP
app.UseStatusCodePages(async statusCodeContext =>
{
    var response = statusCodeContext.HttpContext.Response;
    var path = statusCodeContext.HttpContext.Request.Path;

    response.ContentType = "text/plain; charset=UTF-8";
    if (response.StatusCode == 403)
    {
        await response.WriteAsync($"Path: {path}. Access Denied ");
    }
    else if (response.StatusCode == 404)
    {
        await response.WriteAsync($"Resource {path} Not Found");
    }
});

app.Map("/hello", () => "Hello ASP.NET Core");

app.Run();
```



Методы UseStatusCodePagesWithRedirects и UseStatusCodePagesWithReExecute

Вместо метода **UseStatusCodePages()** мы также можем использовать еще пару других, которые также обрабатывают ошибки HTTP.

С помощью метода **app.UseStatusCodePagesWithRedirects()** можно выполнить переадресацию на определенный метод, который непосредственно обработает статусный код:

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

app.UseStatusCodePagesWithRedirects("/error/{0}");

app.Map("/hello", () => "Hello ASP.NET Core");
app.Map("/error/{statusCode}", (int statusCode) => $"Error. Status Code: {statusCode}");

app.Run();
```

Здесь будет идти перенаправление по адресу "/error/{0}". В качестве параметра через плейсхолдер "{0}" будет передаваться статусный код ошибки.

Но теперь при обращении к несуществующему ресурсу клиент получит статусный код 302 / Found. То есть формально несуществующий ресурс будет существовать, просто статусный код 302 будет указывать, что ресурс перемещен на другое место – по пути "/error/404".

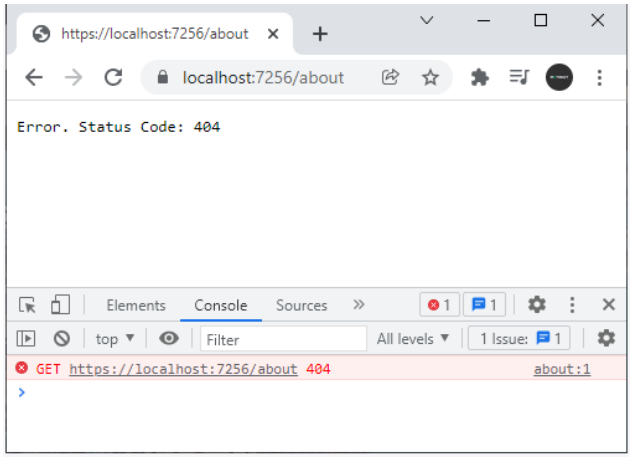
Подобное поведение может быть неудобно, особенно с точки зрения поисковой индексации, и в этом случае мы можем применить другой метод **app.UseStatusCodePagesWithReExecute()**:

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

app.UseStatusCodePagesWithReExecute("/error/{0}");

app.Map("/hello", () => "Hello ASP.NET Core");
app.Map("/error/{statusCode}", (int statusCode) => $"Error. Status Code: {statusCode}");

app.Run();
```

В качестве параметра метод **UseStatusCodePagesWithReExecute()** приминает путь к ресурсу, который будет обрабатывать ошибку. И также с помощью плейсхолдера {0} можно передать статусный код ошибки. То есть в данном случае при возникновении ошибки будет вызываться конечная точка

```
app.Map("/error/{statusCode}", (int statusCode) => $"Error. Status Code: {statusCode}");
```

Формально мы получим тот же ответ, так как так же будет идти перенаправление на путь "/error/404". Но теперь браузер получит оригинальный статусный код 404.



Также можно задействовать другую версию метода, которая в качестве второго параметра принимает параметры строки запроса

```
app.UseStatusCodePagesWithReExecute("/error", "?code={0}");
```

Первый параметр метода указывает на путь перенаправления, а второй задает параметры строки запроса, которые будут передаваться при перенаправлении. Вместо плейсхолдера {0} опять же будет передаваться статусный код ошибки. Пример использования:

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

app.UseStatusCodePagesWithReExecute("/error", "?code={0}");

app.Map("/hello", () => "Hello ASP.NET Core");
app.Map("/error",  (string code) => $"Error Code: {code}");


app.Run();
```