**Обработка ошибок**

**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](https://github.com/dotnet/aspnetcore/blob/main/src/DefaultBuilder/src/WebApplicationBuilder.cs), который применяется для создания приложения, то мы там можем увидеть там следующие строки:

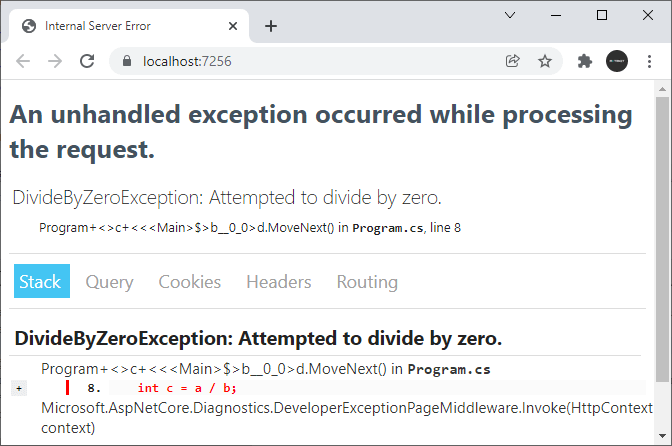
|  |  |
| --- | --- |
| 1  2  3  4 | if (context.HostingEnvironment.IsDevelopment())  {      app.UseDeveloperExceptionPage();  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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:

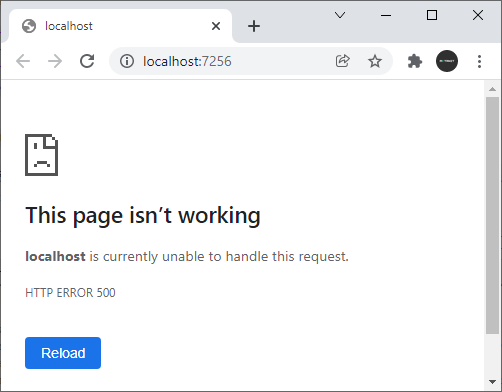


Этой информации достаточно, чтобы определить где именно в коде произошло исключение.

Теперь посмотрим, как все это будет выглядеть для простого пользователя. Для этого изменим код приложения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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.



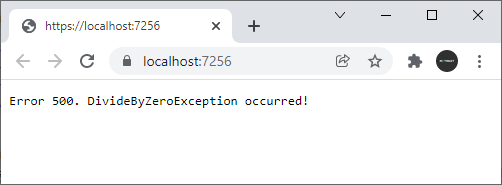
### UseExceptionHandler

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | 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 некотоое 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | 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.

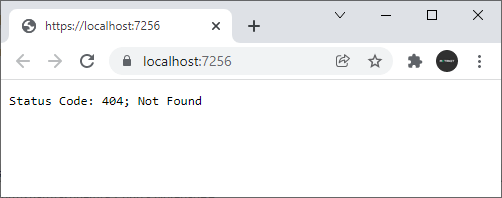
## Обработка ошибок HTTP

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()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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:



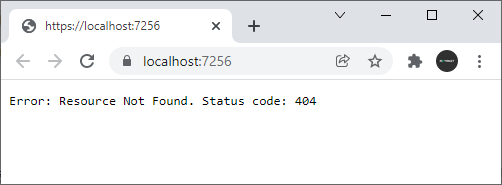
Метод **UseStatusCodePages()** следует вызывать ближе к началу конвейера обработки запроса, в частности, до добавления middleware для работы со статическими файлами и до добавления конечных точек.

### 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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(); |

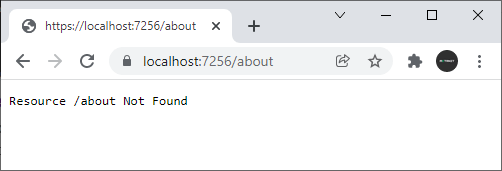
В качестве первого параметра указывается MIME-тип ответа - в данном случае простой текст (""text/plain""). В качестве второго параметра передается собственно то сообщение, которое увидит пользователь. В сообщение мы можем передать код ошибки через плейсхолдер "{0}".



### Установка обработчика ошибок

Еще одна версия метода **UseStatusCodePages()** позволяет более детально задать обаботчик 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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".

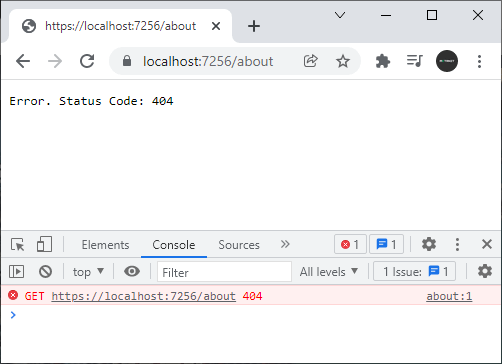
Подобное поведение может быть неудобно, особенно с точки зрения поисковой индексации, и в этом случае мы можем применить другой метод **app.UseStatusCodePagesWithReExecute()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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

|  |  |
| --- | --- |
| 1 | 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

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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(); |