# Основы Razor Pages

## Определение страниц Razor

Основой функциональности платформы Razor Pages являются страницы Razor, которые обрабатывают приходящие к приложению запросы. Рассмотрим, как определять страницы Razor. Для этого создадим новый проект по типу ASP NET Core Empty. По умолчанию проект этого типа не подключает функциональность Razor Pages, поэтому после создания проекта первым делом подключим в него функциональность Razor Pages. Для этого откроем файл Program.cs и изменим его следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var builder = WebApplication.CreateBuilder(args);    // добавляем в приложение сервисы Razor Pages  builder.Services.AddRazorPages();    var app = builder.Build();    // добавляем поддержку маршрутизации для Razor Pages  app.MapRazorPages();    app.Run(); |

## Добавление страницы Razor Page

По умолчанию согласно условностям страницы Razor размещаются в проекте в папке Pages. А строка запроса URL будет сопоставляться с определенной страницей Razor на основании ее расположения в проекте в папке Pages. Примеры строк URL и сопоставленных с ними страниц:

Путь к странице URL

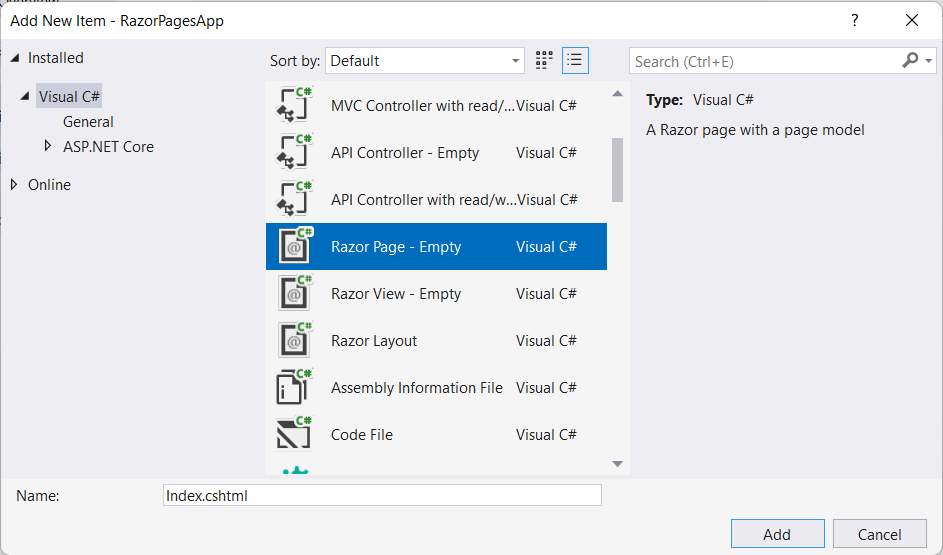
*/Pages/Index.cshtml /* или */Index*

*/Pages/Contact.cshtml /Contact*

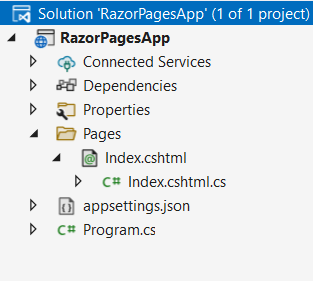
*/Pages/Store/Contact.cshtml /Store/Contact*

*/Pages/Store/Index.cshtml /Store* или */Store/Index*

Поэтому добавим в проект папку Pages. Затем в папку Pages добавим новую страницу Razor. Для этого нажмем на данную папку правой кнопкой мыши и выберем в контекстном меню пункт Add -> New Item. Далее среди шаблонов выберем шаблон Razor Page - Empty и назовем новый файл Index.cshtml:



После создания этой страницы в проект в папку Pages будут добавлены два файла - сама страница Index.cshtml и связаный с ней файл кода Index.cshtml.cs.



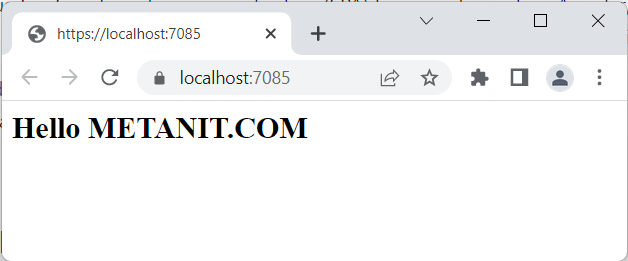
По умолчанию файл Index.cshtml выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4 | @page  @model RazorPagesApp.Pages.IndexModel  @{  } |

Любая страница Razor должна начинаться с директивы @page - именно эта директива указывает, что данный файл представляет страницу Razor. После этой директивы может помещаться код html, какие-то выражения синтаксиса Razor. Так, в данном случае с помощью директивы @model определяет модель представления - в данном случае класс IndexModel - класс, который определен в файле Index.cshtml.cs. Но в принципе опреление модели необязательно. Например, изменим код страницы следующим образом:

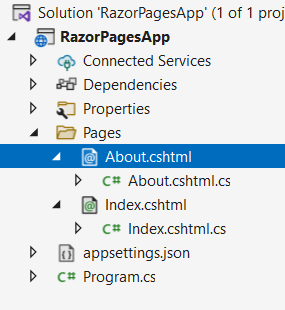
|  |  |
| --- | --- |
| 1  2  3 | @page    <h2>Hello METANIT.COM</h2> |

Это простейшая страница Razor, которая определяет заголовок



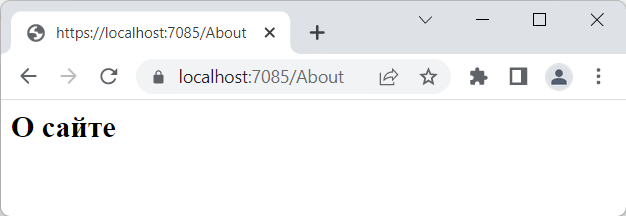
При этом использование модели на странице в принципе необязательно, как видно из примера выше.

Подобным образом мы можем использовать и другие страницы. Например, добавим в папку Pages новую страницу Razor, которую назовем About.cshtml



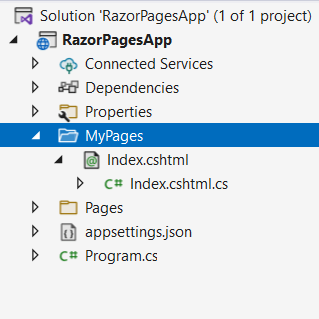
Определим на ней какое-нибудь простейшее содержимое:

|  |  |
| --- | --- |
| 1  2  3 | @page    <h2>О сайте</h2> |



### Переопределение каталога страниц

Хотя обычно страницы Razor помещаются в проекте в каталог Pages, но это необязательно. Например, определим в проекте новую папку MyPages. И далее в эту папку добавим новую страницу Razor Index.cshtml:



Определим на этой странице какой-нибудь простейший контент, например:

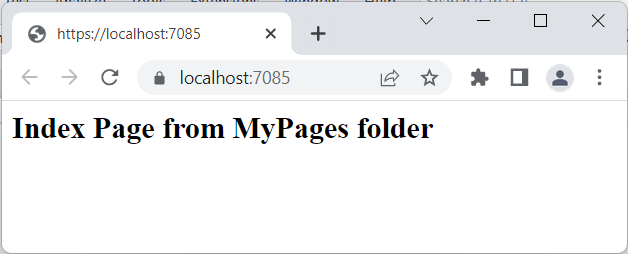
|  |  |
| --- | --- |
| 1  2  3 | @page    <h2>Index Page from MyPages folder</h2> |

Для изменения каталога для Razor Pages необходимо настроить соответствующие сервисы. Для этого перейдем к файлу Program.cs и изменим его следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var builder = WebApplication.CreateBuilder(args);    // добавляем в приложение сервисы Razor Pages  builder.Services.AddRazorPages(options => options.RootDirectory = "/MyPages");    var app = builder.Build();    // добавляем поддержку маршрутизации для Razor Pages  app.MapRazorPages();    app.Run(); |

Метод AddRazorPages() в качестве параметра принимает делегат Action, который, в свою очередь, имеет параметр RazorPagesOptions. Через этот параметр можно задать некоторые базовые настройки для Razor Pages. В частности, свойство options.RootDirectory позволяет установить корневой каталог для страниц Razor.

Соответственно теперь запросы к приложению будут обрабатываться страницами из папки "MyPages":

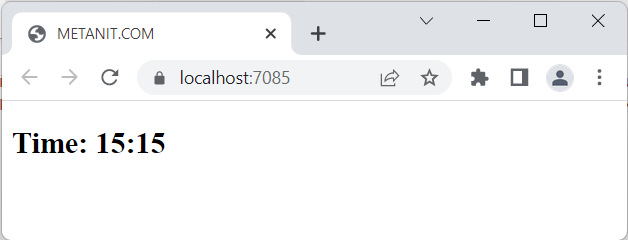


## Синтаксис Razor

Ключевым моментом в определении интерфейса на страницах Razor Page является использование конструкций движка Razor. Благодаря Razor мы можем применять на странице выражения языка C#. Синтаксис Razor довольно прост - все его конструкции предваряются символом @, после которого происходит переход от кода HTML к коду C#. При генерации ответа клиенту Razor обрабатывает выражения языка C# и на их основе генерирует код HTML. Например, определим следующее представление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @page    <!DOCTYPE html>  <html>  <head>      <title>METANIT.COM</title>      <meta charset="utf-8" />  </head>  <body>      <h2>Time: @DateTime.Now.ToShortTimeString()</h2>  </body>  </html> |

Здесь вместо выражения @DateTime.Now.ToShortTimeString() при рендеринге представления будет вставляться текущее время:



Стоит отметить, что по умолчанию Razor подключает на страницы следующие пространства имен

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.Rendering;  using Microsoft.AspNetCore.Mvc.ViewFeatures; |

Соответственно мы можем использовать функционал этих пространств имен на страницах Razor, как в примере выше структуру DateTime из пространства System.

## Типы конструкций Razor

Все конструкции Razor можно условно разделить на два вида: однострочные выражения и блоки кода.

Пример применения однострочных выражений:

|  |  |
| --- | --- |
| 1 | <p>Date: @DateTime.Now.ToLongDateString()</p> |

В данном случае используется объект DateTime и его метод ToLongDateString()

Или еще один пример:

|  |  |
| --- | --- |
| 1 | <p>@(20 + 30)</p> |

Так как перед скобками стоит знак @, то выражение в скобках будет интерпретироваться как выражение на языке C#. Поэтому браузер выведет число 50, а не "20 + 30".

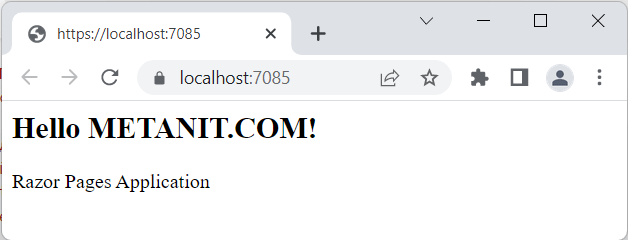
Но если вдруг мы создаем код html, в котором присутствует символ @ не как часть синтаксиса Razor, а сам по себе, то, чтобы его отобразить, нам надо его дублировать:

|  |  |
| --- | --- |
| 1 | <p>@@DateTime.Now =@DateTime.Now.ToLongDateString()</p> |

Блоки кода могут иметь несколько выражений. Блок кода заключается в фигурные скобки, а каждое выражение завершается точкой с запятой аналогично блокам кода и выражениям на C#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @page    @{      string head = "Hello METANIT.COM!"; // определяем переменную head      string text = "ASP.NET Core Application";   // определяем переменную text  }    <h2>@head</h2> <!-- используем переменную head -->  <div>@text</div> <!-- используем переменную text --> |

В блоках кода мы можем определить обычные переменные и потом их использовать в представлении.



Если необходимо вывести значение переменной без каких-либо html-элементов, то мы можем использовать специальный снипет <text>:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @page    @{      int i = 8;      <text>@i</text>  }  <text>@(i+1)</text> |

В Razor могут использоваться комментарии. Они располагаются между символами @\*\*@:

|  |  |
| --- | --- |
| 1 | @\* текст комментария \*@ |

## Условные конструкции

Также мы можем использовать условные конструкции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @page    @{      string morning = "Good Morning";      string evening = "Good Evening";      string hello = "Hello";      int hour = DateTime.Now.Hour;  }  @if (hour < 12)  {      <h2>@morning</h2>  }  else if (hour > 17)  {      <h2>@evening</h2>  }  else  {      <h2>@hello</h2>  } |

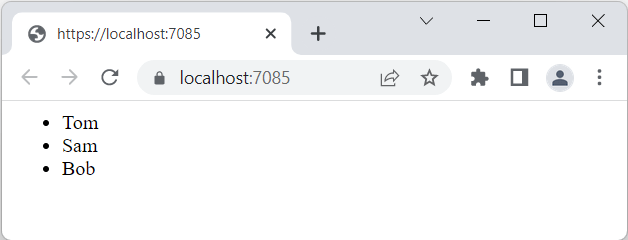
Конструкция switch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @page    @{      string language = "german";  }  @switch(language)  {      case "russian":          <h3>Привет мир!</h3>          break;      case "german":          <h3>Hallo Welt!</h3>          break;      default:          <h3>Hello World!</h3>          break;  } |

## Циклы

Кроме того, мы можем использовать все возможные циклы. Цикл **for**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @page    @{      string[] people = { "Tom", "Sam", "Bob" };  }  <ul>      @for (var i = 0; i < people.Length; i++)      {          <li>@people[i]</li>      }  </ul> |



Цикл **foreach**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @page    @{      string[] people = { "Tom", "Sam", "Bob" };  }  <ul>      @foreach (var person in people)      {          <li>@person</li>      }  </ul> |

Цикл **while**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @page    @{      string[] people = { "Tom", "Sam", "Bob" };      var i = 0;  }  <ul>      @while ( i < people.Length)      {          <li>@people[i++]</li>      }  </ul> |

Цикл **do..while**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @page    @{      var i = 1;  }  <ul>      @do      {          <li>@(i \* i)</li>      }      while ( i++ < 5);  </ul> |

### try...catch

Конструкция try...catch...finally, как и в C#, позволяет обработать исключение, которое может возникнуть при выполнение кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @page    @try  {      throw new InvalidOperationException("Something wrong");  }  catch (Exception ex)  {      <p>Exception: @ex.Message</p>  }  finally  {      <p>finally</p>  } |

Если в блоке try выбрасывается исключение, то выполняется блок catch. И в любом случае в конце блока try и catch выполняется блок finaly.

### Вывод текста в блоке кода

Обычный текст в блоке кода мы не сможем вывести:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @page    @{      bool isEnabled = true;  }  @if (isEnabled)  {      Hello World  } |

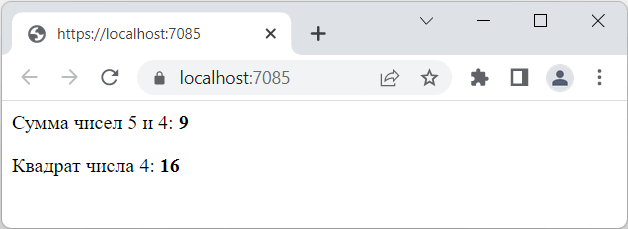
В этом случае Razor будет рассматривать строку "Hello" как набор операторов языка C#, которых, естественно в C# нет, поэтому мы получим ошибку. И чтобы вывести текст как есть в блоке кода, нам надо использовать выражение @::

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @page    @{      bool isEnabled = true;  }  @if (isEnabled)  {      @: Hello  } |

### Функции

Директива **@functions** позволяет определить функции, которые могут применяться в представлении. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @page    @functions  {      public int Sum(int a, int b)      {          return a + b;      }      public int Square(int n) => n \* n;  }  <p>Sum of 5 and 4: <b> @Sum(5, 4)</b></p>  <p>Square of 4: <b>@Square(4)</b></p> |

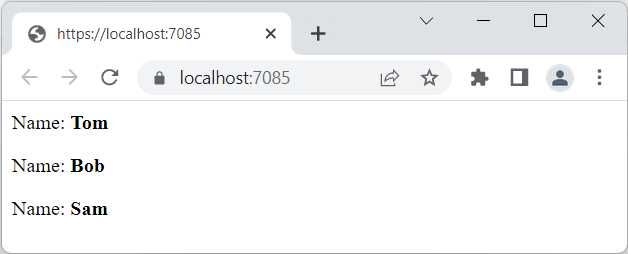


### Локальные функции

В блоках кода также можно определять локальные функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @page    @{      void RenderName(string name)      {          <p>Name: <b>@name</b></p>      }        RenderName("Tom");      RenderName("Bob");  }    <div>@{RenderName("Sam");}</div> |

В данном случае функция RenderName выводит некоторую разметку html, в которую передается значение параметра name:

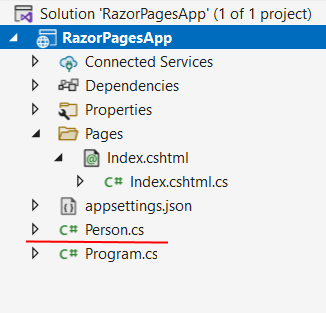


### Инструкция using

С помощью директивы **using** можно подключать на страницу Razor различные пространства. Например, определим в проекте новый класс **Person**:

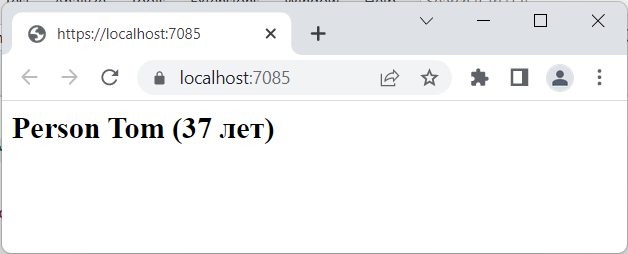
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | namespace RazorPagesApp  {      public class Person      {          public string Name { get; }          public int Age { get; }          public Person(string name, int age)          {              Name = name;              Age = age;          }          public override string ToString() => $"Person {Name} ({Age} лет)";      }  } |

В данном случае класс Person расположен в пространстве имен RazorPagesApp:



Чтобы использовать данный класс на странице Razor, его пространство имен необходимо подключить с помощью директивы **@using**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @page    @using RazorPagesApp @\* подключение пространства имен RazorPagesApp \*@    @{      Person tom = new Person("Tom", 37);  }    <h2>@tom</h2> |



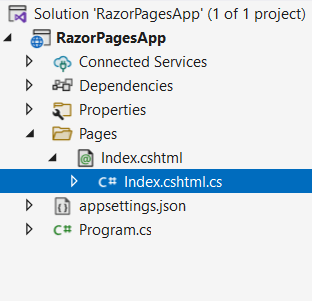
В качестве альтрнативы, как и в общем в C#, можно было бы указать полное имя класса с учетом пространства имен:

|  |  |
| --- | --- |
| 1 | RazorPagesApp.Person tom = new RazorPagesApp.Person("Tom", 37); |

Но директива using позволяет сократить код.

## Модель страницы Razor

Вместе со страницей Razor в проект добавляется файл связанного кода на языке C#. Этот код определяется в виде класса, который по умолчанию называется по имени страницы плюс суффикс "Model" (например, для страницы Index - класс IndexModel) и который помещается в файл с именем **[файл\_страницы\_razor].cs**.



Например, при добавлении страницы **Index.cshtml** вместе с ней будет добавляться файл **Index.cshtml.cs**, который будет содержать определение класса **IndexModel**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public void OnGet()          {          }      }  } |

Класс модели страницы Razor должен обязательно наследоваться от абстрактного класса **PageModel**. По умолчанию данный класс содержит пустой метод **OnGet()**, который призван обрабатывать get-запросы. Здесь мы можем определить какие-то данные, какую-то логику, которая будет применяться на странице razor.

К пример, изменим этот класс следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; }          public IndexModel()          {              Message = "Hello METANIT.COM";          }          public string PrintTime() => DateTime.Now.ToShortTimeString();      }  } |

В данном случае класс IndexModel определяет свойство Message и метод PrintTime, который возвращает текущее время.

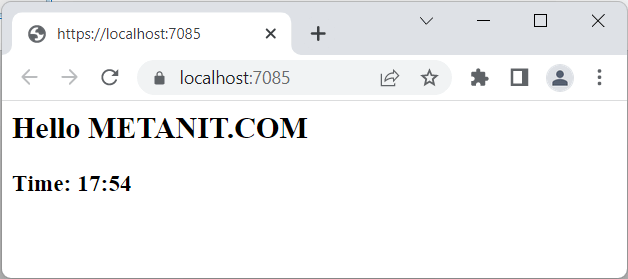
Для подключения класса-модели на страницу Razor применяется директива **@model**, после которой идет имя класса. Например, изменим определение страницы **Index.cshtml** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @page    @model RazorPagesApp.Pages.IndexModel  <h2>@Model.Message</h2>    <h3>Time: @Model.PrintTime()</h3> |

Обратите внимание, что после директивы @model указывается полное имя класса с учетом пространства имен.

После подключения модели мы можем обращаться к ее функционалу через свойство **Model**. Например, с помощью выражения @Model.Message мы можем обратиться к свойству Message в модели.

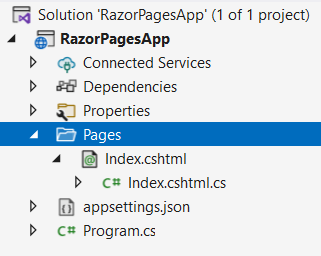
В итоге при обращении к странице Index и генерации ответа будет использоваться функционал модели IndexModel:



## Обработка запросов. Контекст страницы Razor

Когда мы обращаемся к приложению, то посылаем ему запрос определенного типа - get, post, put и т.д. Для обработки запросов в Razor Pages можно определять соответствующие методы, которые называются по шаблону *On[Тип\_запроса]*, либо можно использовать их асинхронные двойники *On[Тип\_запроса]Async*. Например, запрос GET обрабатывается методами **OnGet()** и **OnGetAsync()**, а запросы Post - методами **OnPost()** и **OnPostAsync()**. Подобные методы можно определять как на модели страницы, так и на самой странице Razor.

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:



В файле **Index.cshtml.cs** определим следующий класс IndexModel:

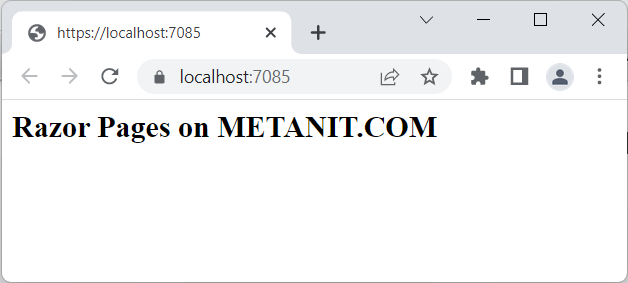
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet()          {              Message = "Razor Pages on METANIT.COM";          }      }  } |

Эта самая примитивная модель, цель которой - просто продемонстрировать обработку запросов. Когда к странице придет GET-запрос, то будет вызван метод OnGet(), котором будет установлено свойство Message.

Далее изменим код самой страницы **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4 | @page    @model RazorPagesApp.Pages.IndexModel  <h2>@Model.Message</h2> |

В данном случае просто выводим значение свойства Message модели в заголовке



Стоит отметить, что для обработки запросов необязательно в принципе использовать модель, все действия можно сделать на самой странице. Например, изменим страницу **Index.cshtml** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @page    <h2>@Message</h2>    @functions {      public string Message { get; set; } = "";      public void OnGet()      {          Message = "Razor Pages on METANIT.COM";      }  } |

Теперь класс IndexModel никак не используется, вся логика определена непосредственно на странице, поэтому работать она будет также как и ранее. Однако не все такой подход может показаться привлекательным из-за того, что визуальное представление смешивается с логикой и представлением данных, особенно если логика более комплексная, а не просто установка сообщения для вывода. Поэтому, как правило, подход с классом, производным от PageModel, является более предпочтительным.

### Контекст страницы

При обращении к странице Razor Pages среда ASP.NET создает контекст страницы, который содержит различные связанные с запросом данные. Контекст страницы можно получить на самой странице и в ее модели через свойство **PageContext**, которое представляет одноименный класс PageContext. Этот класс определяет ряд важный свойств:

* **HttpContext**: содержит информацию о контексте запроса
* **ActionDescriptor**: возвращает дескриптор страницы - объект CompiledPageActionDescriptor, который содержит информацию о странице Razor, которая обрабатывает запрос
* **ModelState**: возвращает словарь ModelStateDictionary, который используется для валидации данных, отправленных пользователем
* **RouteData**: возвращает данные маршрута

Для получения информации о запросе нас прежде всего будет интересовать свойство HttpContext, которое представляет объект Microsoft.AspNetCore.Http.HttpContext. В принципе это тот же самый объект, который нам доступен в любом компоненте middleware в ASP.NET Core. Этот объект также доступен через свойство HttpContext страницы Razor и ее модели. То есть следующие вызовы будут обращаться к одному и тому же объекту:

|  |  |
| --- | --- |
| 1  2 | var ctx1 = PageContext.HttpContext;  var ctx2 = HttpContext; |

Объект HttpContext инкапсулирует всю информацию о запросе. В частности, он определяет следующие свойства:

* **Request**: содержит собственно информацию о текущем запросе.
* **Response**: управляет ответом
* **User**: представляет текущего пользователя, который обращается к приложению
* **Session**: объект для работы с сессиями

### Request

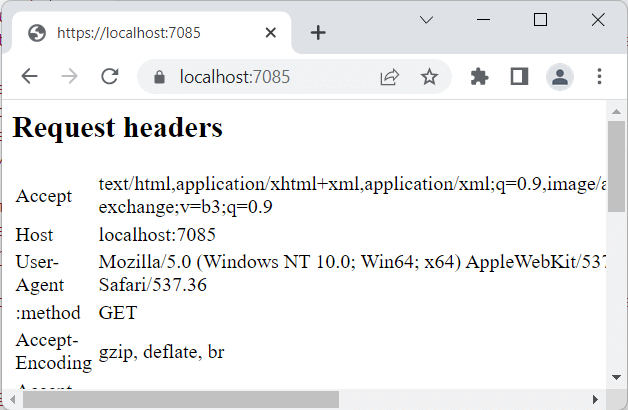
Свойство **HttpContext.Request** представляет объект **HttpRequest** и предоставляет разнообразную информацию о запросе. Этот же объект доступен через свойство **Request** класса контроллера. Среди свойств объекта Request можно выделить следующие:

* **Body**: объект Stream, который используется для чтения данных запроса
* **Cookies**: куки, полученные в запросе
* **Form**: коллекция значений отправленных форм
* **Headers**: коллекция заголовков запроса
* **Path**: возвращает запрошенный путь - строка запроса без домена и порта
* **Query**: возвращает коллекцию переданных через строку запроса параметров
* **QueryString**: возвращает ту часть запроса, которая содержит параметры. Например, в запросе *http://localhost:52682/Home/Index?alt=4* это будет ?alt=4

//Request.Cookies.TryGetValue("Navbar", out string? s);

Вся основная информация нам доступна из заголовков. Например, получим из запроса все заголовки и выведем их в браузере. Для этого изменим страницу **Index.cshtml** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @page    <h2>Request headers</h2><table>  @foreach (var header in Request.Headers)  {      <tr><td>@header.Key</td><td>@header.Value</td></tr>  }  </table> |



## Передача данных на страницу Razor в GET-запросе

В get-запросе также можно передать на страницу данные. Распространенным способом передачи данных в GET-запросе представляет **строка запроса**. Строка запроса представляет ту часть адреса, которая идет после знака вопроса ? и представляет набор параметров, где каждый параметр отделен от другого с помощью амперсанда:

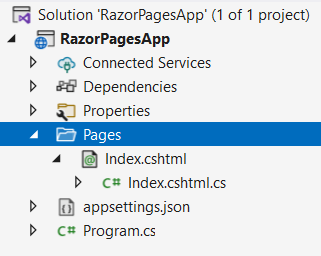
|  |  |
| --- | --- |
| 1 | название\_ресурса?параметр1=значение1&параметр2=значение2 |

Например, в адресе:

|  |  |
| --- | --- |
| 1 | <https://localhost:7085/Index?name=Tom&age=37> |

часть *?name=Tom&age=37* как раз представляет строку запроса, которая содержит два параметра: name и age. Значение параметра name - "Tom", а значение параметра age - 37.

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:

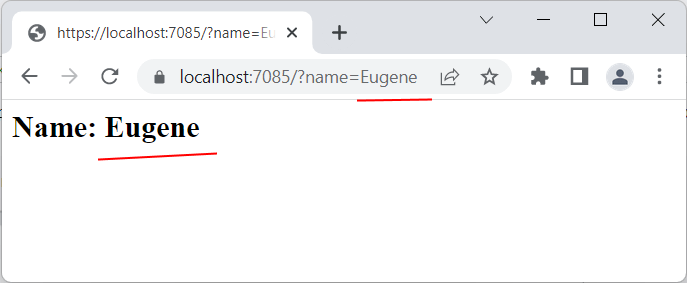


Передадим в страницу **Index.cshtml** через строку запроса данные для параметра name. Для этого определим следующую модель **IndexModel**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet(string name)          {              Message = $"Name: {name}";          }      }  } |

А на странице **Index.cshtml** выведем значение свойства Message модели:

|  |  |
| --- | --- |
| 1  2  3  4 | @page    @model RazorPagesApp.Pages.IndexModel  <h2>@Model.Message</h2> |



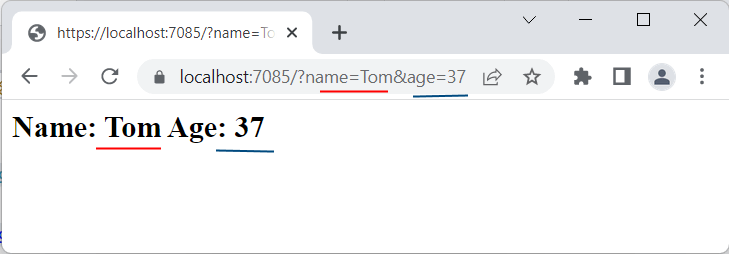
То есть в данном случае при обращении к методу Index с запросом https://localhost:7085/Index?name=Eugene параметру name будет передаваться значение "Eugene".

Система привязки по умолчанию сопоставляет параметры запроса и параметры метода по имени. То есть, если в строке запроса идет параметр name, то его значение будет передаваться именно параметру метода, который также называется name. При этом должно быть также соответствие по типу, то есть если параметр метода принимает числовое значение, то и через строку запроса надо передавать для этого параметра число, а не строку.

Подобным образом можно передать значения для нескольких параметров. Например, изменим модель **IndexModel** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet(string name, int age)          {              Message = $"Name: {name}  Age: {age}";          }      }  } |

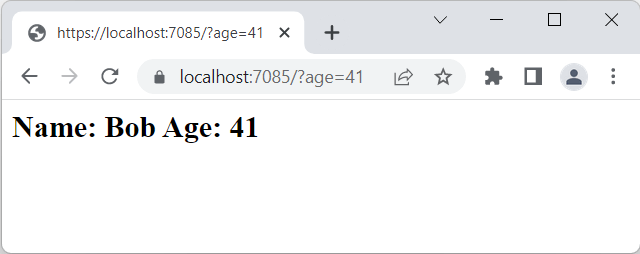
В этом случае мы можем обратиться к действию, набрав в адресной строке *https://localhost:7085?name=Tom&age=37*.



Если же мы не используем параметры в строке запроса, то мы можем использовать параметры по умолчанию, которые будут работать, если через строку запроса не передается никаких параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet(string name = "Bob", int age = 33)          {              Message = $"Name: {name}  Age: {age}";          }      }  } |

Например, при отправке запроса *https://localhost:7085/Index?age=41* параметр name будет иметь значение по умолчанию.



Подобным образом можно получать данные из строки запроса непосредственно на странице **Index.cshtml**

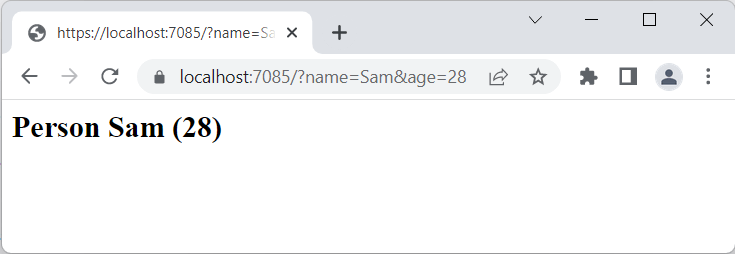
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @page    <h2>@Message</h2>    @functions {      public string Message { get; set; } = "";      public void OnGet(string name, int age)      {          Message = $"Name: {name}  Age: {age}";      }  } |

### Передача сложных объектов

Хотя строка запроса преимущественно используется для передачи данных примитивных типов, но мы также можем принимать более сложные объекты. Например, определим рядом с моделью класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet(Person person)          {              Message = $"Person  {person.Name} ({person.Age})";          }      }      public record class Person(string Name, int Age);  } |

Класс Person определяет два свойства: Name и Age. И в модели IndexModel метод OnGet принимает параметр типа Person. В этом случае значения через строку запроса передаются как и в предыдущем случае. При этом параметры строки запроса должны соответствовать по имени свойствам объекта. Регистр названий параметров при этом не учитывается:



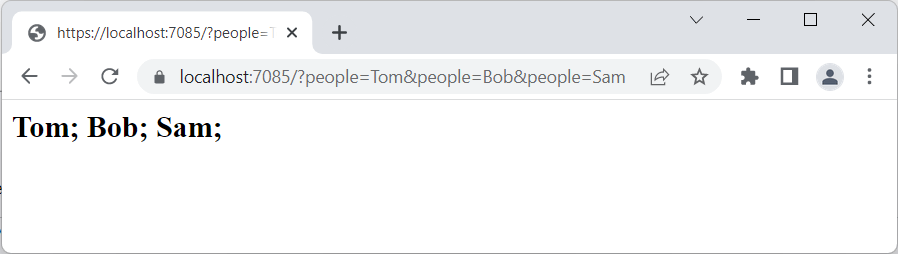
### Передача массивов

Допустим, метод OnGet принимает массив строк-имен:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet(string[] people)          {              string result = "";              foreach (var person in people)                  result = $"{result}{person}; ";              Message = result;          }      }  } |

Для передачи методу данных через строку запроса применяется название параметра:

https://localhost:7085/?people=Tom&people=Bob&people=Sam



Можно явным образом указать индексы элементов:

https://localhost:7085/?people[0]=Tom&people[2]=Bob&people[1]=Sam

Можно просто ограничиться индексами:

https://localhost:7085/?[0]=Tom&[2]=Bob&[1]=Sam

### **Передача массивов сложных объектов**

Подобным образом можно принимать массивы сложных объектов. Например, определим следующую модель IndexModel, которая в методе OnGet принимает массив объектов некоторого класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet(Person[] people)          {              string result = "";              foreach (Person person in people)              {                  result = $"{result} {person.Name}; ";              }              Message = result;          }      }      public record class Person(string Name, int Age);  } |

И чтобы передать в этот метод данные, нам надо использовать запрос типа

https://localhost:7085/?people[0].name=Tom&people[0].age=37&people[1].name=Bob&people[1].age=41

В этом случае в массиве people будут два объекта Person.

Также можно опустить название параметра и оставить только индексы:

https://localhost:7085/?[0].name=Tom&[0].age=37&[1].name=Bob&[1].age=41

### **Передача словарей Dictionary**

Передача в метод словарей аналогично передаче массивов, только у передаваемых элементов необходимо установить ключ. Например, пусть метод OnGet получает в качестве параметра объект Dictionary:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public void OnGet(Dictionary<string, string> items)  {      string result = "";      foreach (var item in items)      {          result = $"{result} {item.Key} - {item.Value}; ";      }      Message = result;  } |

Для отправки данных мы можем использовать строку запроса:

https://localhost:7085/?items[germany]=berlin&items[france]=paris&items[spain]=madrid

Каждый отдельный элемент этой строки типа items[germany]=berlin будет представлять элемент словаря, где items - название словаря, "germany" - ключ, а "berlin" - значение.

### **Объект Request.Query**

Параметры представляют самый простой способ получения данных, но в действительности нам необязательно их использовать. На странице и в ее модели доступен объект **Request**, у которого можно получить как данные строки запроса через свойство **Request.Query**. Это свойство представляет объект IQueryCollection, где по ключу - названию параметра можно получить его значение. Например:

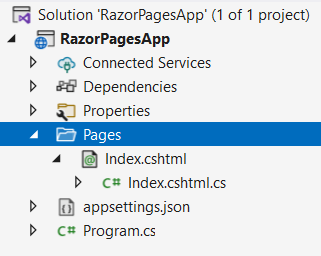
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public void OnGet()  {      string name = Request.Query["name"];      string age = Request.Query["age"];      Message = $"Name: {name}  Age: {age}";  } |

В данном случае мы можем передать странице данные через запрос типа *https://localhost:7085/Index?name=Tom&age=37*.

## POST-запросы и отправка форм

Кроме GET-запросов для отправки данных приложению также широко применяются POST-запросы. Как правило, такие запросы отправляются с помощью форм на html-странице. Но основные принципы передачи данных будут теми же, что и в GET-запросах. Рассмотрим различные сценарии при передаче данных в POST-запросе странице Razor.

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:



Например, пусть страница **Index.cshtml** будет отправлять форму и получать из формы некоторое значение. Для этого определим следующую модель **IndexModel**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {        [IgnoreAntiforgeryToken]      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet()          {              Message = "Введите свое имя";          }          public void OnPost(string username)          {              Message = $"Ваше имя: {username}";          }      }  } |

Здесь определено два метода. В методе OnGet обрабатываются get-запросы, что сводится к установки свойства Message.

В методе **OnPost()** обрабатываем POST-запросы. Данный метод будет получать извне из отправленной формы некоторую строку через параметр username и с его помощью переустанавливает значение свойства Message.

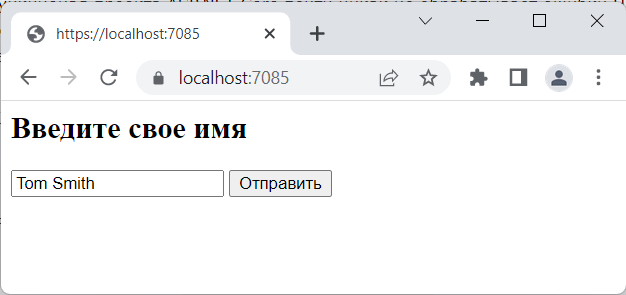
Самый важный момент при обработки отправляемых форм - класс модели Razor для валидации полученных форм использует специальный токен - **AntiforgeryToken**. Далее мы рассмотрим, как его устанавливать. Однако мы можем отключить валидацию формы с помощью этого токена с помощью атрибута **[IgnoreAntiforgeryToken]**, который применяется к классу модели (также можно применять глобально в приложении).

Для отправки формы на странице **Index.cshtml** определим следующий код:

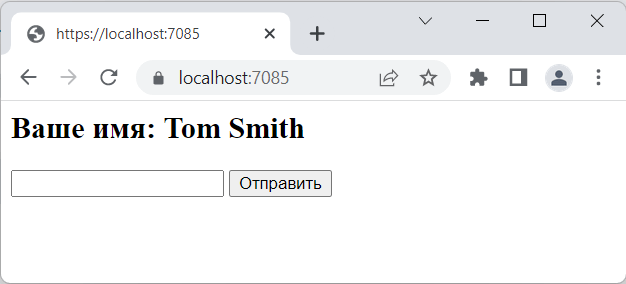
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>@Model.Message</h2>  <form method="post" >      <input type="text" name="username" />      <input type="submit" value="Отправить" />  </form> |

Здесь определена форма для ввода условного имени. При получении запроса get приложение просто будет отдавать данную страницу с формой ввода. При получении запроса post класс IndexModel получит отправленное значение и изменит значение свойства Message, которое потом выводится на страницу. При этом, чтобы система могла связать параметры запроса со значениями формы поля форм (их атрибут name) должны называться также, как и параметры методов OnPost или OnGet. То есть параметр в методе OnPost называется **username**, и поле формы (значение его атрибута name) тоже называется **username**.

При запуске страница отобразит приглашение к вводу:



А при отправке введенного значения отобразится результат:



Для обработки запроса можно использовать класс, производный от PageModel. Однако страница Razor уже сама по себе представляет модель. И мы можем всю логику обработки и свойства модели определить напрямую в странице. Например, изменим страницу **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @page    <h2>@Model.Message</h2>  <form method="post" >      <input type="text" name="username" />      <input type="submit" value="Отправить" />  </form>    @functions {        public string Message { get; private set; } = "";      public void OnGet() => Message = "Введите свое имя";        public void OnPost(string username) => Message = $"Ваше имя: {username}";  } |

Но в этом случае мы можем опять же столкнуться с необходимостью верификации токена AntiForgery-токена. Чтобы решить проблему с этим токеном, мы можем отключить его глобально. Для этого изменим файл **Program.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using Microsoft.AspNetCore.Mvc;    var builder = WebApplication.CreateBuilder(args);    // добавляем в приложение сервисы Razor Pages  builder.Services.AddRazorPages(options =>  {      // отключаем глобально Antiforgery-токен      options.Conventions.ConfigureFilter(new IgnoreAntiforgeryTokenAttribute());  });    var app = builder.Build();    // добавляем поддержку маршрутизации для Razor Pages  app.MapRazorPages();    app.Run(); |

### Получение сложных объектов

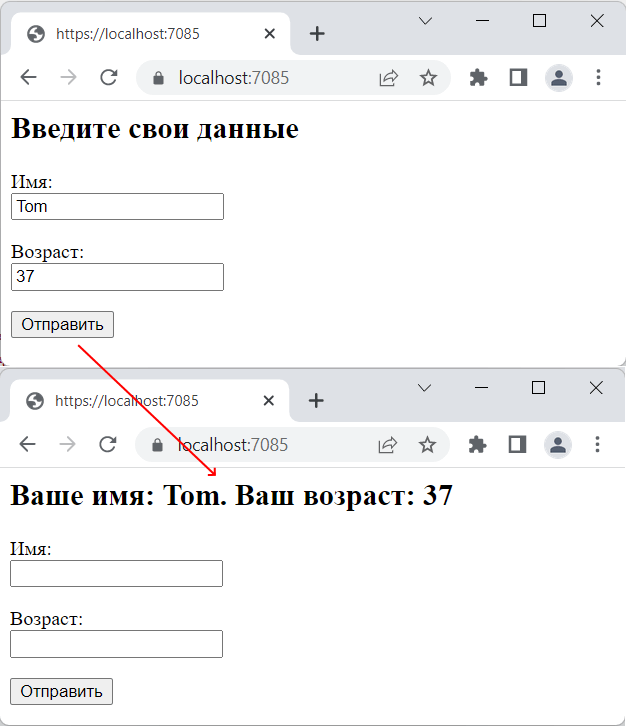
Подобным образом мы можем получать и большее количество данных. Например, определим на странице **Index.cshtml** следующую форму:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>@Model.Message</h2>  <form method="post" >      <p>          <label>Имя:</label><br />          <input type="text" name="name" />      </p>      <p>          <label>Возраст:</label><br />          <input type="number" name="age" />      </p>      <input type="submit" value="Отправить" />  </form> |

В данном случае пользователь должен ввести имя и возраст.

А в классе модели **IndexModel** определим следующую логику:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {        [IgnoreAntiforgeryToken]      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet()          {              Message = "Введите свои данные";          }          public void OnPost(string name, int age)          {              Message = $"Ваше имя: {name}. Ваш возраст: {age}";          }      }  } |



Хотя мы можем получать весь набор отправляемых данных по отдельности, однако также можно все эти значения объединить в более сложные объекты. Например, определим рядом с моделью класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet(Person person)          {              Message = $"Person  {person.Name} ({person.Age})";          }      }      public record class Person(string Name, int Age);  } |

Класс Person определяет два свойства: Name и Age. И в модели IndexModel метод OnPost принимает параметр типа Person. При чем поля формы name и age соответствуют по названию свойствам класса Person, поэтому вместо одиночных разрозненных значений мы можем получить отправленную форму в виде объекта Person.

### Получение массивов

Для передачи массивов с помощью формы надо создать набор одноименных полей, которые называются по имени массива. Например, определим следующую модель **IndexModel**:

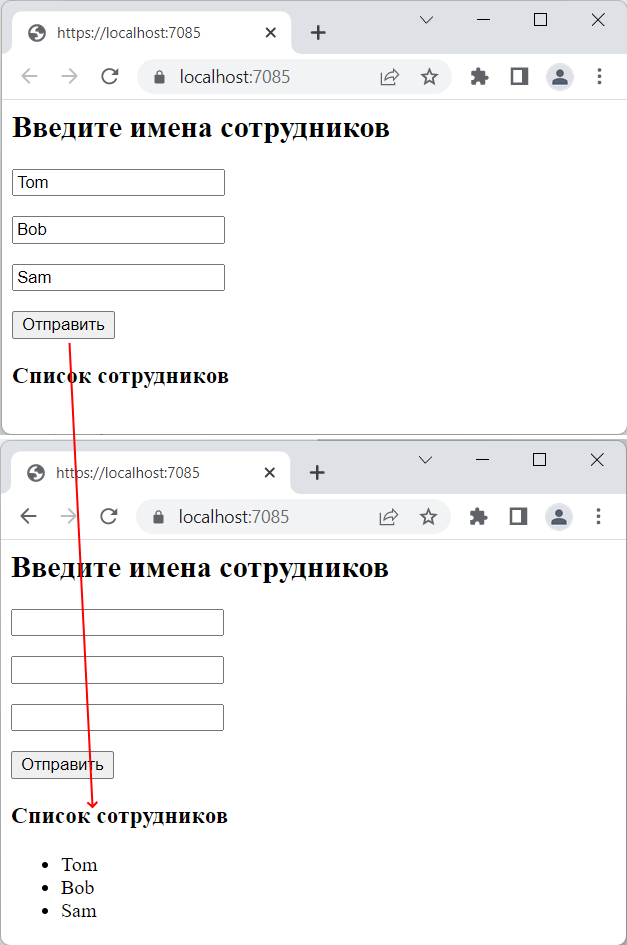
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {        [IgnoreAntiforgeryToken]      public class IndexModel : PageModel      {          public string[] People { get; private set; } = Array.Empty<string>();            public void OnPost(string[] people)          {              People = people;          }      }  } |

В данном случае модель в методе OnPost получает массив строк и передает его свойству People.

Также определим на странице **Index.cshtml** отправки данных и вывода элементов из People следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>Ввeдите имена сотрудников</h2>  <form method="post">      <p><input name="people" /></p>      <p><input name="people" /></p>      <p><input name="people" /></p>      <input type="submit" value="Отправить" />  </form>  <h3>Список сотрудников</h3>  <ul>      @foreach(var person in Model.People)      {          <li>@person</li>      }  </ul> |

В данном случае на форме, отправляемой пользователю, расположены три поля с именем "people". В итоге при отправке формы будет сформирован массив people из трех элементов, который можно получить в модели методе OnPost:



И также у элементов формы можно было бы явным образом указать индексы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <form method="post">      <p><input name="people[0]" /></p>      <p><input name="people[2]" /></p>      <p><input name="people[1]" /></p>      <input type="submit" value="Send" />  </form> |

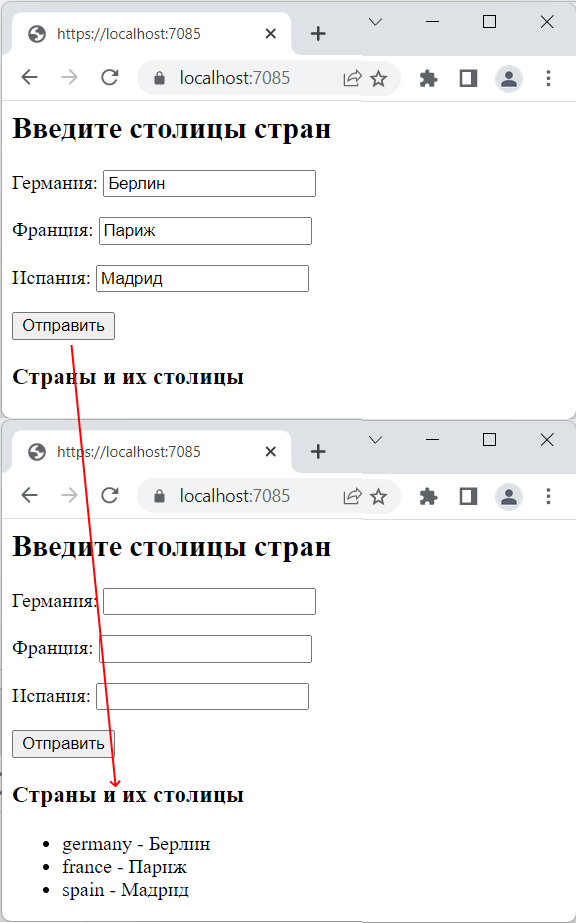
Либо можно было бы вовсе ограничиться одними индексами

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <form method="post">      <p><input name="[0]" /></p>      <p><input name="[2]" /></p>      <p><input name="[1]" /></p>      <input type="submit" value="Send" />  </form> |

### Передача словарей Dictionary

Передача словарей в метод контроллера аналогична передаче элементов массивов за тем исключением, что для каждого элемента устанавливается ключ. Так, определим следующую страницу **Index.cshtml**:

|  |  |
| --- | --- |
| 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  30 | @page    <h2>Ввeдите столицы стран</h2>  <form method="post">      <p>          Германия: <input type="text" name="items[germany]" />      </p>      <p>          Франция: <input type="text" name="items[france]" />      </p>      <p>          Испания: <input type="text" name="items[spain]" />      </p>      <input type="submit" value="Отправить" />  </form>  <h3>Страны и их столицы</h3>  <ul>      @foreach(var country in Countries)      {          <li>@country.Key - @country.Value</li>      }  </ul>    @functions{      Dictionary<string, string> Countries { get; set; } = new();      public void OnPost(Dictionary<string, string> items)      {          Countries = items;      }  } |

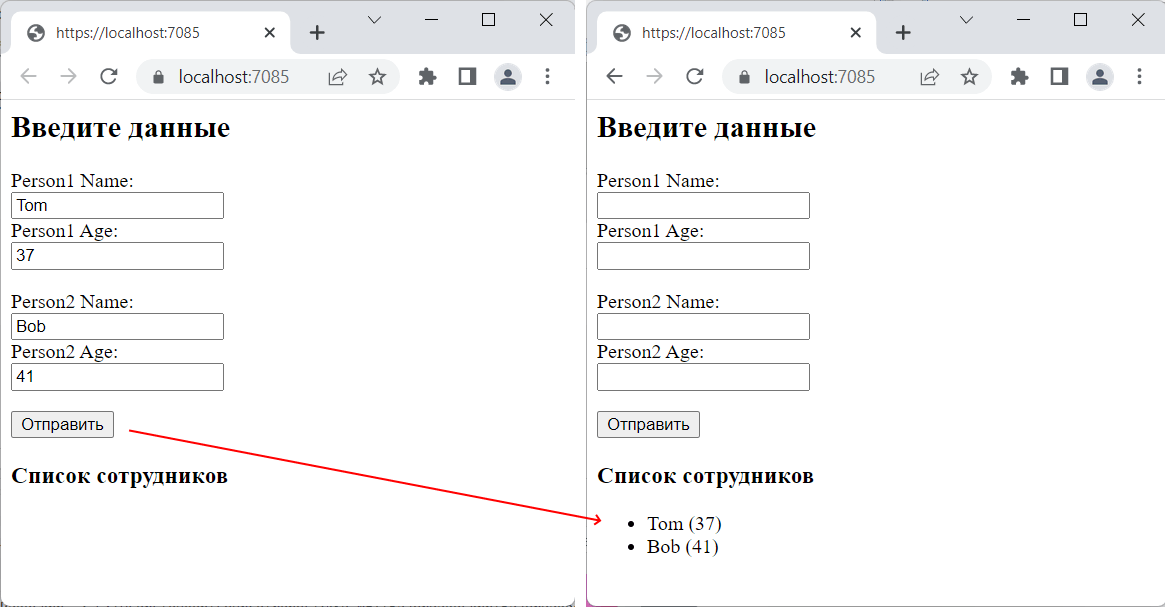


### Отправка массивов сложных объектов

При отправке массивов сложных объектов на форме также определяется набор полей, где каждое поле привязано к определенному свойству объекта. Например, изменим страницу **Index.cshtml** 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  30  31  32  33  34  35 | @page    <h2>Ввeдите данные</h2>  <form method="post">      <p>          Person1 Name:<br/>          <input name="people[0].name" /><br/>          Person1 Age:<br/>          <input name="people[0].age" />      </p>      <p>          Person2 Name:<br/>          <input name="people[1].name" /><br/>          Person2 Age:<br/>          <input name="people[1].age" />      </p>      <input type="submit" value="Отправить" />  </form>  <h3>Список сотрудников</h3>  <ul>      @foreach(var person in People)      {          <li>@person.Name (@person.Age)</li>      }  </ul>    @functions{      Person[] People { get; set; } = { };      public void OnPost(Person[] people)      {          People = people;      }        public record class Person(string Name, int Age);  } |

В данном случае на форму вводятся значения для свойств Name и Age двух объектов Person. После отправке эти объекты уйдут в виде массива people второму методу Index.



### Получение данных из контекста запроса

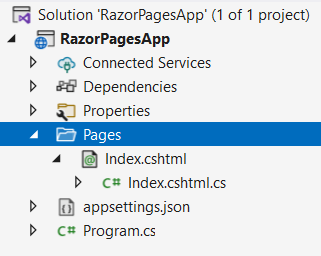
Для получения данных отправленных форм через параметры метода OnPost также на страницах Razor и в их моделях можно использовать свойство **Request.Form**. Это свойство представляет коллекцию IFormsCollection, где каждый элемент имеет ключ и значение. В качестве ключа элемента выступает название поля формы, а в качестве значения - введенные в это поле данные. Например, используем **Request.Form** на странице **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @page    <h2>@Message</h2>  <form method="post" >      <p>          <label>Имя:</label><br />          <input type="text" name="name" />      </p>      <p>          <label>Возраст:</label><br />          <input type="number" name="age" />      </p>      <input type="submit" value="Отправить" />  </form>    @functions{      string Message { get; set; } = "";      public void OnPost()      {          string name = Request.Form["name"];          string age = Request.Form["age"];          Message = $"Ваше имя: {name}. Ваш возраст: {age}";      }  } |

## Привязка свойств страниц и моделей Razor к параметрам запроса

Для получения отправленных данных мы можем использовать параметры в методах OnGet/OnPost/OnPut/OnDelete и затем передавать их значения свойствам или как-то иначе обрабатывать. Однако Razor Pages позволяет напрямую установить привязку свойств страницы Razor Pages и ее модели и параметров запроса с помощью атрибута **BindProperty**, что в ряде случаев может упростить обработку.

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:



Определим в файле **Index.cshtml.cs** следующий код:

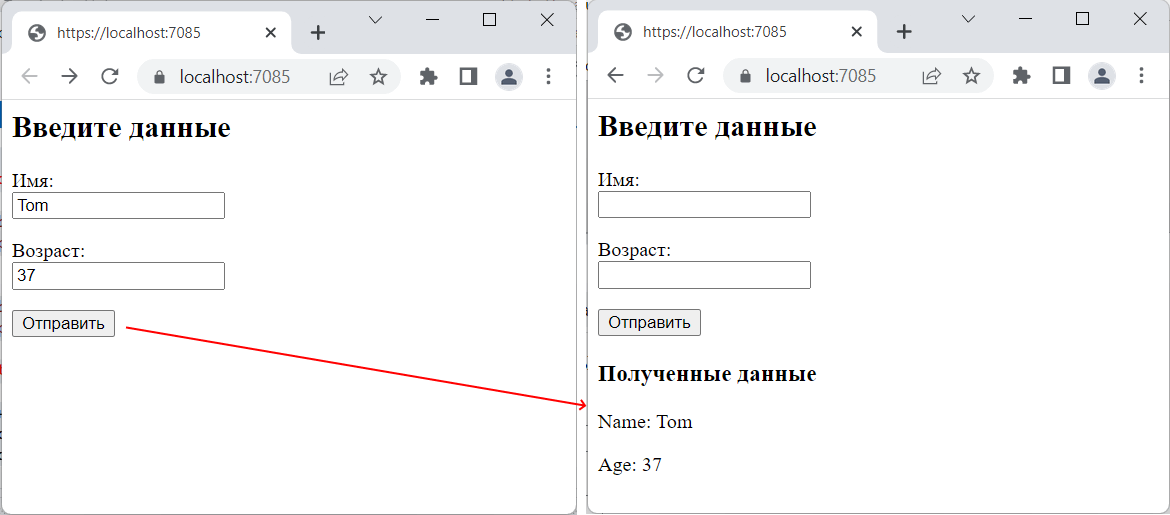
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {        [IgnoreAntiforgeryToken]      public class IndexModel : PageModel      {          [BindProperty]          public string Name { get; set; } = "";            [BindProperty]          public int Age { get; set; }        }  } |

В модели IndexModel определены два свойства, к которым применяется атрибут **BindProperty**. Причем в модели нет никаких методов OnGet или OnPost. Тем не менее если в запросе будут данные с ключами name и age (регистр названий ключей не имеет значения), то эти данные будут автоматически передаваться одноименным свойствам.

Далее на странице **Index.cshtml** определим форму для ввода данных и элементы для их вывода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>Введите данные</h2>  <form method="post" >      <p>          <label>Имя:</label><br />          <input type="text" name="name" />      </p>      <p>          <label>Возраст:</label><br />          <input type="number" name="age" />      </p>      <input type="submit" value="Отправить" />  </form>  @if(Request.Method == "POST")  {      <h3>Полученные данные</h3>      <p>Name: @Model.Name</p>      <p>Age: @Model.Age</p>  } |

В данном случае названия полей формы соответствуют именам свойств модели IndexModel, за счет чего с помощью атрибута BindProperty будет происходить автоматическая привязка



При этом можно также обрабатывать запросы в методах OnGet/OnPost, например:

|  |  |
| --- | --- |
| 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 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {        [IgnoreAntiforgeryToken]      public class IndexModel : PageModel      {          [BindProperty]          public string Name { get; set; } = "";            [BindProperty]          public int Age { get; set; }            public string Message { get; private set; } = "";          public void OnGet()          {              Message = "Введите данные";          }          public void OnPost()          {              Message = $"Имя: {Name}  Возраст: {Age}";          }        }  } |

### Привязка к сложным объектам

Выше каждое из отправляемых из формы значений связывалось с определенным свойством. Однако можно также определить общую модель, которая будет объединять все эти значения. Например, изменим модель **IndexModel** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {        [IgnoreAntiforgeryToken]      public class IndexModel : PageModel      {          [BindProperty]          public Person? Person { get; set; }      }      public record class Person (string Name, int Age);  } |

Теперь данные будут связываться с объектом класса Person.

На странице **Index.cshtml** изменим код вывода полученных значений:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>Введите данные</h2>  <form method="post" >      <p>          <label>Имя:</label><br />          <input type="text" name="name" />      </p>      <p>          <label>Возраст:</label><br />          <input type="number" name="age" />      </p>      <input type="submit" value="Отправить" />  </form>  @if(Request.Method == "POST")  {      <h3>Полученные данные</h3>      <p>Name: @Model.Person?.Name</p>      <p>Age: @Model.Person?.Age</p>  } |

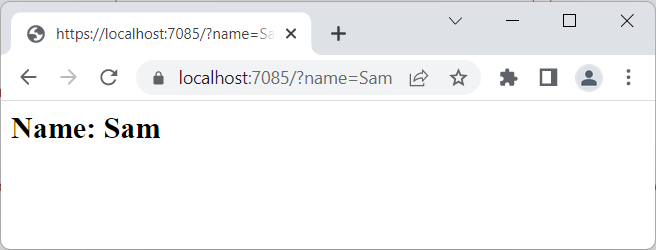
### Получение данных из GET-запросов

По умолчанию атрибут **BindProperty** не поддерживает привязку свойств к значениям из GET-запрос. Для добавления этого типа запросов у атрибута для свойства **SupportGet** необходимо установить значение **true**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {        [IgnoreAntiforgeryToken]      public class IndexModel : PageModel      {          [BindProperty(SupportsGet = true)]          public string? Name { get; set; }      }  } |

На странице **Index.cshtml** выведем полученное значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>Name: @Model.Name</h2> |

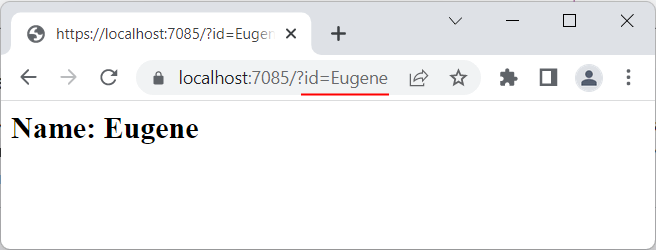


### Переопределение названий параметров

Выше свойства модели и параметры запросы связывались по имени. Однако мы можем переопределить имя параметра запроса с помощью свойства **Name** атрибута **BindProperty**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          [BindProperty(SupportsGet = true, Name = "id")]          public string? Name { get; set; }      }  } |

В данном случае свойству Name будет передаваться значение параметра "id":



## Параметры маршрутов в Razor Pages

Кроме получения данных из параметры запроса и из отправленных форм в Razor Pages также можно получать данные из параметров маршрута. И в данном случае важно понимать отличие параметров строки запроса от параметров маршрута. Например:

|  |  |
| --- | --- |
| 1  2 | localhost:5000/Index?id=21  localhost:5000/Index/21 |

Параметры, которые идут после вопросительного знака в адресе url, представляют **параметры строки запроса**. А данные, которые разделены в url слешами, представляют **параметры маршрута**. Один url может содержать как параметры маршрута, так и параметры запроса.

Чтобы определить параметры маршрута, необходимо после на странице после директивы **@page** определить шаблон маршрута. Например, мы хотим получать из маршрута параметр id:

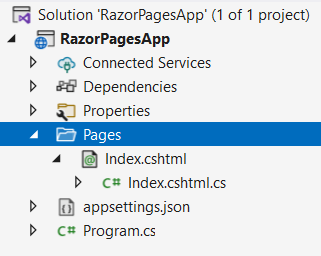
|  |  |
| --- | --- |
| 1 | @page "{id}" |

Для получения параметров маршрута на странице Razor и в ее модели применяется словарь **RouteData.Values**. С помощью ключа - названия параметра можно получить его значение:

|  |  |
| --- | --- |
| 1 | object? id = RouteData.Values["id"]; |

Стоит при этом учитывать, что возвращаемое значение по умолчанию представляет тип **object?**

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:

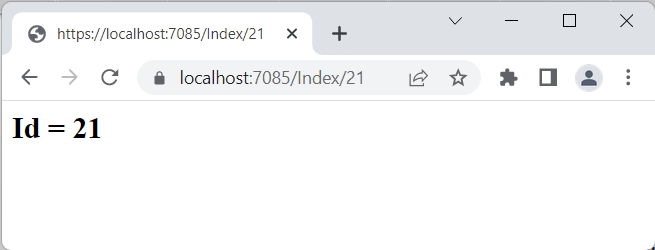


Например, определим в файле **Index.cshtml.cs** следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public object? Id { get; private set; }            public void OnGet()          {              Id = RouteData.Values["id"];          }      }  } |

Используем на странице **Index.cshtml** параметр id:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @page "{id}"    @model RazorPagesApp.Pages.IndexModel    <h2>Id = @Model.Id</h2> |



Также можно получить значение параметра непосредственно на странице **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @page "{id}"  @{      var userid = RouteData.Values["id"];  }  <h2>Id = @userid</h2> |

### Получение параметров маршрута через параметры метода OnGet

Также мы можем получить параметры маршрута через одноименные параметры метода **OnGet()**. Так, изменим модель **IndexModel**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public int Id { get; private set; }            public void OnGet(int id)          {              Id = id;          }      }  } |

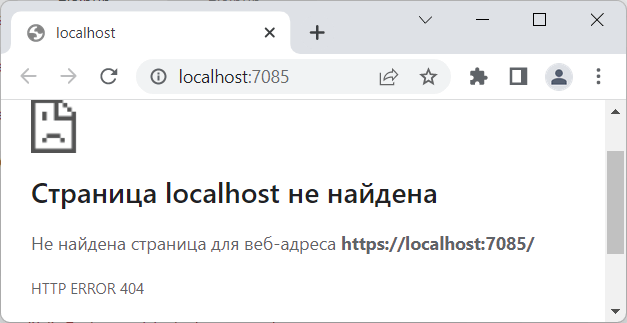
Код страницы **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4 | @page "{id}"  @model RazorPagesApp.Pages.IndexModel    <h2>Id = @Model.Id</h2> |

Преимуществом такого подхода является то, что инфраструктура Razor Pages может автоматически преобразовать значение к некоторому типу. Так, в данном случае мы получаем параметр маршрута как значение типа int, а не object?.

### Необязательные параметры

Если мы возьмем пример выше и обратимся к странице без передачи параметра, то браузер отобразит нам ошибку 404:



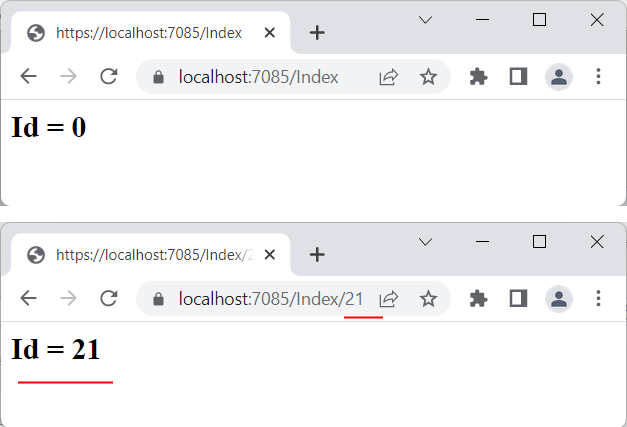
В одних сценариях это может приемлимо и даже необходимо, в других, возможно, нежелательно. И на этот случай мы можем установить параметр как необязательный, указав после названия параметра вопросительный знак **?**:

|  |  |
| --- | --- |
| 1  2  3  4 | @page "{id?}"  @model RazorPagesApp.Pages.IndexModel    <h2>Id = @Model.Id</h2> |

Тогда в методе OnGet мы можем определить параметр как значение nullable-типа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public int Id { get; private set; }            public void OnGet(int? id)          {              Id = id ?? 0;          }      }  } |

В данном случае если параметр id передан, то свойство Id получает его значение. Если параметр не передан, то свойство Id получает значение 0.



В качестве альтернативы для параметра можно предусмотреть значение по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4 | public void OnGet(int id = 1)  {      Id = id;  } |

### Привязка к свойствам

И также, как и в случае с параметрами строки запроса, мы можем получить значение параметра маршрута, используя атрибут **BindProperty**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          [BindProperty(SupportsGet = true)]          public int Id { get; set; }      }  } |

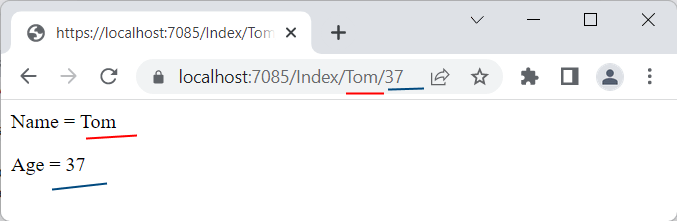
### Передача нескольких параметров

При необходимости мы можем передавать несколько параметров маршрута. Каждый параметр заключается в фигурные скобки и отделяется от других параметров слешами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Name { get; set; } = "";          public int Age { get; set; }            public void OnGet(string name, int age)          {              Name = name;              Age = age;          }      }  } |

В данном случае получаем два параметра - name и age. На странице **Index.cshtml** определим данные параметры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @page "{name}/{age}"  @model @model RazorPagesApp.Pages.IndexModel    <p>Id = @Model.Id</p>  <p>Name = @Model.Name</p>  <p>Age = @Model.Age</p> |



Также оба параметра можно было бы привязать к свойствам:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          [BindProperty(SupportsGet = true)]          public string Name { get; set; } = "";          [BindProperty(SupportsGet = true)]          public int Age { get; set; }      }  } |

### Ограничения параметров

Как и в стандартной модели MVC для параметров маршрутов в Razor Pages можно применять ограничения:

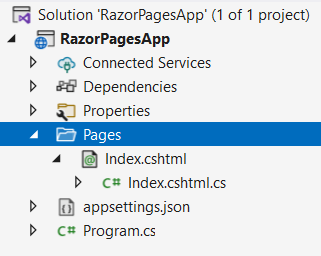
|  |  |
| --- | --- |
| 1  2  3  4  5 | @page "{name:alpha:minlength(3)}/{age:int}"  @model PersonModel    <p>Name = @Model.Name</p>  <p>Age = @Model.Age</p> |

В данном случае параметр age должен обязательно представлять целое число, а параметр name может содержать только алфавитные символы, причем не менее трех символов. В данном случае действуют все те же ограничения, что применяются к параметрам маршрута в целом в ASP.NET Core и про которые можно посмотреть в статье [Ограничения маршрутов](https://metanit.com/sharp/aspnet6/3.3.php).

## Обработчики страницы в Razor Pages

Для обработки запроса каждая страница может использовать методы OnGet/OnPost/OnPut и так далее, то есть такие методы, которые предназначены для определенного вида запросов. Однако что если нам необходимо определить на странице для разных ситуаций несколько методов для обработки запроса одного и того же типа запросов (GET или POST)? В этом случае мы можем добавить в код страницы дополнительные обработчики запросов.

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:



Например, пусть в проекте определен класс Person, который представляет данные:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Person  {      public string Name { get; set; }      public int Age { get; set; }  } |

В файле коде **Index.cshtml.cs** определим следующий код:

|  |  |
| --- | --- |
| 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  30  31  32  33 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          // начальные данные - список людей          List<Person> people = new ()          {              new Person ("Tom Smith", 23),              new Person ("Sam Anderson", 23),              new Person ("Bob Johnson", 25),              new Person ("Tom Anderson", 25)          };          // отображаемые данные          public List<Person> DisplayedPeople { get; private set; } = new();            public void OnGet()          {              DisplayedPeople = people;          }            public void OnGetByName(string name)          {              DisplayedPeople = people.Where(p => p.Name.Contains(name)).ToList();          }          public void OnGetByAge(int age)          {              DisplayedPeople = people.Where(p => p.Age == age).ToList();          }      }      public record class Person(string Name, int Age);  } |

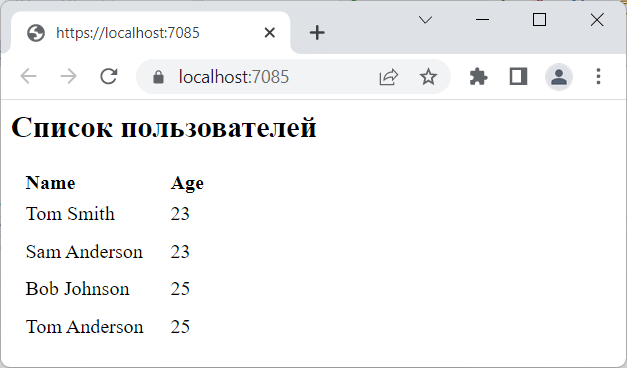
Класс содержит некоторые начальные данные в переменной people - список объектов Person. Свойство DisplayedPeople же будет содержать отображаемые на странице данные, которые могут отличаться от списка people. При стандартном get-запросе в свойство DisplayedPeople передается весь этот список.

И также определены два метода - обработчики для фильтрации списка по имени и возрасту объекта Person. Так как оба этих обработчики будут также обрабатывать запросы get, то их название начинается с префикса **OnGet**.

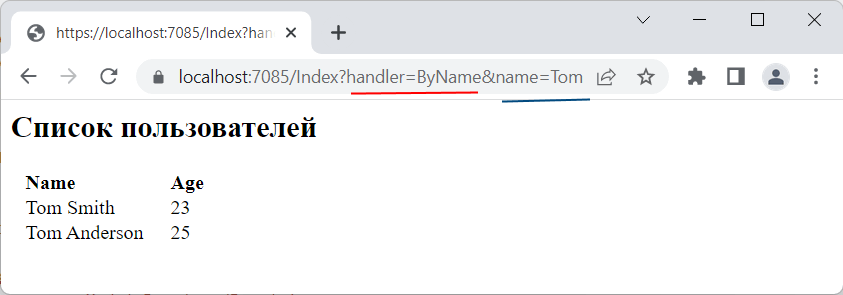
На странице **Index.cshtml** определим вывод списка DisplayedPeople из IndexModel:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @page    @using RazorPagesApp.Pages  @model IndexModel    <h2>Список пользователей</h2>  <table>      <tr><th>Name</th><th>Age</th></tr>      @foreach(Person person in Model.DisplayedPeople)      {          <tr>              <td>@person.Name</td>              <td>@person.Age</td>          </tr>      }  </table> |

И при обычном запросе к странице сработает метод OnGet, поэтому на страницу будет выведен полный список:



Теперь обратимся к конкретному обработчику. Для этого в строку запроса передается параметр "handler" с указанием имени обработчика. Причем именем обработчика считается та часть названия метода, которая идет после префикса *OnGet*:



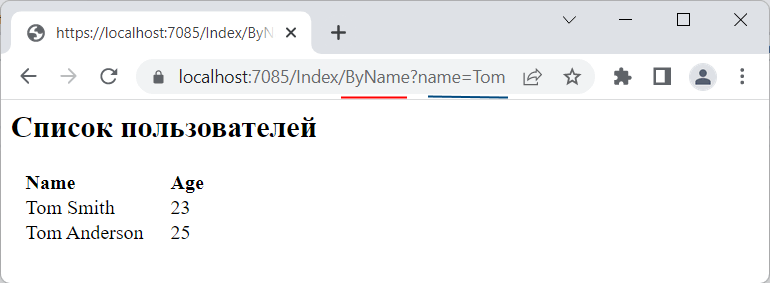
В данном случае обращение идет к обработчику OnGetByName. Поскольку он принимает параметр name, то через строку запроса для этого параметра передается значение: *https://localhost:7085/Index?handler=ByName&name=Tom*.

### Передача обработчика через параметр маршрута

Не всегда может устроить добавление названия обработчика через строку запроса, наподобие **?handler=ByName**. В этом случае можно передать название обработчика через параметр маршрута. Для этого изменим код страницы Person.cshtml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @page "{handler?}"    @using RazorPagesApp.Pages  @model IndexModel    <h2>Список пользователей</h2>  <table>      <tr><th>Name</th><th>Age</th></tr>      @foreach(Person person in Model.DisplayedPeople)      {          <tr>              <td>@person.Name</td>              <td>@person.Age</td>          </tr>      }  </table> |

Вопросительный знак после названия параметра "{handler?}" указывает, что данный параметр является необязательным. И в этом случае можно просто указать в url последним сегментом название обработчика:



## Возвращение результата в Razor Pages

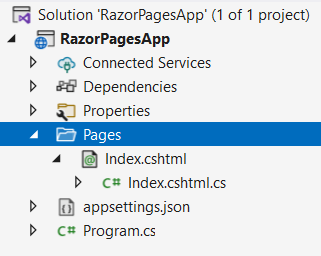
Все рассмотренные в прошлых статьях методы OnGet и OnPost, предназначенные для обработки GET и POST-запросов, ничего не возвращали, то есть имели в качестве возвращаемого типа тип **void** и просто выполняли некоторое действие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public void OnGet()  {      // обработка запроса  }  public void OnPost()  {      // обработка запроса  } |

В реальности данные методы по умолчанию возвращали связанную с классом страницу Razor. Однако иногда возникает необходимость возвратить некоторый результат, например, статусный код ошибки или сделать редирект. Для подобных ситуаций данные методы также могут возвращать объекты **IActionResult**. И для этого в классе **PageModel** определено ряд методов:

* **Content()** возвращает объект ContentResult, то есть фактически некоторое текстовое содержимое
* **File()** возвращает с помощью различных перегруженных версий объекты FileContentResult/FileStreamResult/VirtualFileResult, то есть отправляет клиенту файл
* **Forbid()** возвращает статусный код 403
* **LocalRedirect()/LocalRedirectPermanent()** выполняет переадресацию по определенному локальному адресу
* **NotFound()** возвращает статусный код 404
* **PhysicalFile()** возвращает файл по физическому пути
* **Page()** возвращает объект PageResult или фактически текущую страницу Razor
* **Redirect()/RedirectPermanent()** выполняет переадресацию по определенному адресу
* **RedirectToAction()/RedirectToActionPermanent()** выполняет переадресацию на определенное действие контроллера
* **RedirectToPage()/RedirectToPagePermanent()** выполняет переадресацию на определенную страницу Razor
* **RedirectToRoute()/RedirectToRoutePermanent()** выполняет переадресацию по определенному маршруту
* **StatusCode()** возвращает объект StatusCodeResult, то есть посылает статусный код
* **Unauthorized()** возвращает объект UnauthorizedResult, то есть статусный код ошибки 401
* Для отправки json нет определенного метода, но определен специальный тип **JsonResult**, в конструктор которого можно передавать отправляемый код json.

Пусть у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:



Например, на странице **Index.cshtml** определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>Hello Razor Pages!</h2> |

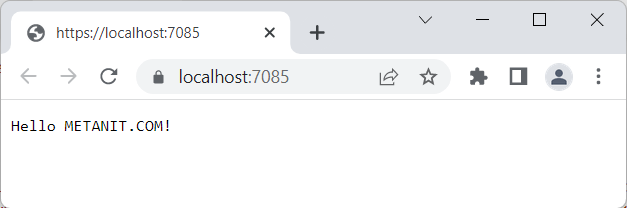
Здесь выводится заголовок "Hello Razor Pages!".

А в файле **Index.cshtml.cs** определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public IActionResult OnGet()          {              return Content("Hello METANIT.COM!");          }      }  } |

Теперь метод **OnGet()** возвращает объект IActionResult. В частности, он вызывает другой метод - **Content()**. В этот метод передается отправляемое клиенту текстовое содержимое. Результатом метода Content() является объект ContentResult - реализация интерфейса IActionResult.

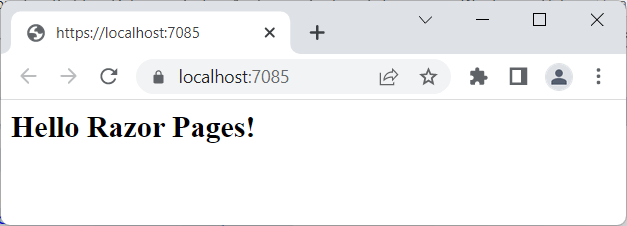
В итоге при запуске приложения мы увидим в браузере не то содержимое, которое определено на странице Index.cshtml, а тот текст, который отправляется из метода **Content()**



То есть в данном случае страница Razor никак не задействуется. Где это может пригодиться? Это может пригодиться в ситуациях, когда нам надо возвратить клиенту результат, отличный от страницы razor, например, статусный код, какой-то контент в формате json, отправить файл, сделать переадресацию или что-то другое.

В то же время мы можем также явным образом возвратить код страницы Razor с помощью метода **Page()**:

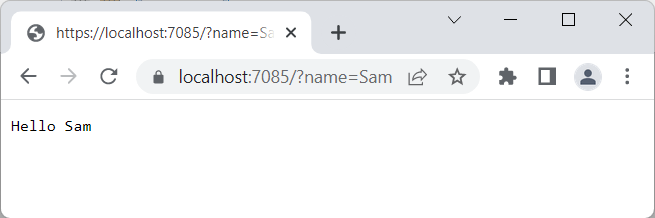
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public IActionResult OnGet()          {              return Page();          }      }  } |



Использование типа IActionResult в качестве возвращаемого значения может пригодиться также, когда необходимо возвратить разные результаты в зависимости от условий. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public IActionResult OnGet(string? name)          {              if(name != null) return Content($"Hello {name}");              return Page();          }      }  } |

В данном случае если при обращении к странице через строку запроса передан параметр name, то возвращается результат метода **Content()**. Если параметр name не установлен, то возвращается связанная страница Razor.



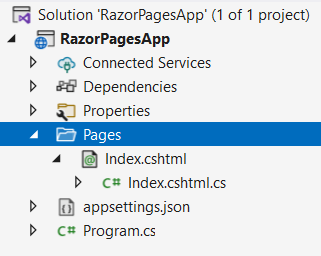
## Отправка статусных кодов в Razor Pages

Нередко возникает необходимость отправить в ответ на запрос какой-либо статусный код. Например, если пользователь пытается получить доступ к ресурсу, который недоступен, или для которого у пользователя нету прав. Либо если просто нужно уведомить браузер пользователя с помощью статусного кода об успешном выполнении операции, как иногда применяется в ajax-запросах. Для этого на страницах Razor и их моделях можно применять различные методы и классы результатов.

### StatusCodeResult

**StatusCodeResult** позволяет отправить любой статусный код клиенту. Для создания этого результата используется метод **StatusCode()**, в который передается отправляемый код статуса.

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:



В файле **Index.cshtml.cs** определим следующую модель:

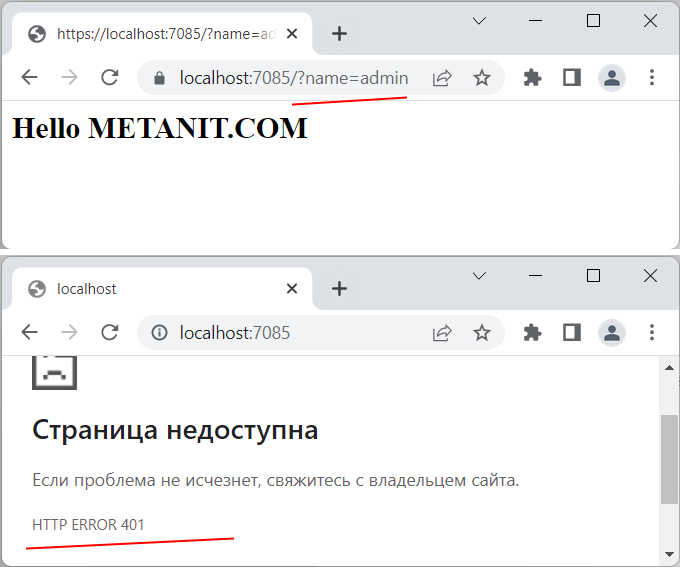
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public IActionResult OnGet(string? name)          {              if(name != "admin") return StatusCode(401);              return Page();          }      }  } |

В данном случае, если значение параметра name не равно строке "admin", то посылаем в ответ пользователю статусный код 401, который говорит о том, что доступ запрещен. Иначе возвращаем страницу Razor.

На самой странице **Index.cshtml** определим какое-нибудь простейшее содержимое для тестирования:

|  |  |
| --- | --- |
| 1  2  3  4 | @page    @model RazorPagesApp.Pages.IndexModel  <h2>Hello METANIT.COM</h2> |

То есть если странице не будет передан параметр name со значением "admin", то пользователь увидит сообщение об ошибке 401. Иначе же ему отобразиться заголовок из страницы Razor.



Подобным образом мы можем послать браузеру любой другой статусный код. Но для отдельных кодов статуса предназначены свои отдельные классы.

### UnauthorizedResult и UnauthorizedObjectResult

**UnauthorizedResult** посылает код 401, уведомляя пользователя, что он не автризован для доступа к ресурсу:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public IActionResult OnGet(string? name)  {      if(name != "admin") return Unauthorized();      return Page();  } |

Для создания ответа используется метод **Unauthorized()**.

**UnauthorizedObjectResult** также посылает код 401, только позволяет добавить в ответ некоторый объект с информацией об ошибке. Конструктор этого класса принимает произвольный объект, который отправляется клиенту в формате json:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public IActionResult OnGet(string? name)  {      if(name != "admin")          return new UnauthorizedObjectResult(new {Message="Unauthorized access"});      return Page();  } |

### HttpNotFoundResult и HttpNotFoundObjectResult

**NotFoundResult** и **NotFoundObjectResult** посылает код 404, уведомляя браузер о том, что ресурс не найден. Второй класс в дополнении к статусному коду позволяет отправить доплнительную информацию, которая потом отобразится в браузере.

Объекты обоих классов создаются методом **NotFound**. Для первого класса - это метод без параметров, для второго класса - метод, который в качестве параметра принимает отправляемую информацию. Например, используем NotFoundObjectResult:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public IActionResult OnGet(string? name)  {      // если строка name пуста или равна null      if(string.IsNullOrEmpty(name)) return NotFound("Resource not found");      return Page();  } |

### BadResult и BadObjectResult

**BadResult** и **BadObjectResult** посылают код 400, что говорит о том, что запрос некорректный. Второй класс в дополнении к статусному коду позволяет отправить доплнительную информацию, которая потом отобразится в браузере.

Эти классы можно применять, например, если в запросе нет каких-то параметров или данные представляют совсем не те типы, которые мы ожидаем получить, и т.д.

Объекты обоих классов создаются методом **BadRequest**. Для первого класса - это метод без параметров, для второго класса - метод, который в качестве параметра принимает отправляемую информацию:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public IActionResult OnGet(string? name)  {      if (string.IsNullOrEmpty(name)) return BadRequest("Name undefined");      return Page();  } |

### OkResult и OkObjectResult

**OkResult** и **OkObjectResult** посылают код 200, уведомляя об успешном выполнении запроса. Первый класс просто отправляет статусный код 200:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public IActionResult OnGet(string? name)  {      // если параметр name передан и это не пустая строка, то посылаем статусный код 200      if(!string.IsNullOrEmpty(name))  return new OkResult();      return Page();  } |

Второй класс в дополнении к статусному коду позволяет отправить доплнительную информацию, которая передается через конструктор и которая отправляется в формате json:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public IActionResult OnGet(string? name)  {      // если параметр name передан и это не пустая строка, то посылаем объект со свойством username      if(!string.IsNullOrEmpty(name))          return new OkObjectResult(new {username = name});      return Page();  } |

## Переадресация в Razor Pages

### Переадресация на страницу Razor

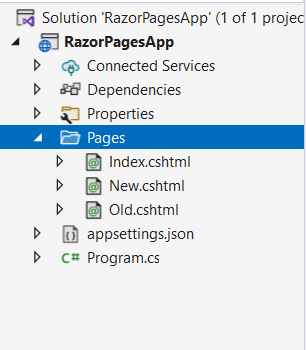
Для переадресации на страницу Razor применяются методы **RedirectToPage()** и **RedirectToPagePermanent()**. Метод **RedirectToPage()** выполняет временную переадресацию, отправляя статусный код 302. А метод **RedirectToPagePermanent()** выполняет постоянную переадресацию и отправляет статусный код 301.

Рассмотрим варианты метода RedirectToPage():

* RedirectToPage(): выполняет переадресацию на текущую страницу, отправляя статусный код 302
* RedirectToPage(object routeValues): выполняет переадресацию на текущую страницу, отправляя статусный код 302 и объект с параметрами для маршрутов
* RedirectToPage(string pageName): выполняет переадресацию на определенную страницу
* RedirectToPage(string pageName, object routeValues): выполняет переадресацию на определенную страницу, отправляя объект с параметрами маршрута
* RedirectToPage(string pageName, string pageHandler): выполняет переадресацию на определенную страницу pageName и обращается к ее обработчику pageHandler
* RedirectToPage(string pageName, string pageHandler, string fragment): выполняет переадресацию на определенную страницу pageName и обращается к ее обработчику pageHandler, добавляя к адресу переадресации фрагмент fragment
* RedirectToPage(string pageName, string pageHandler, object routeValues, string fragment): выполняет переадресацию на определенную страницу pageName и обращается к ее обработчику pageHandler, добавляя к адресу переадресации фрагмент fragment и отправляя объект с параметрами маршрута

Метод RedirectToPagePermanent() имеет аналогичные версии.

Обычно применяется версия метода, которая получает страницу Razor, на которую надо выполнить переадресацию. Например, пусть у нас в проекте есть две страницы **New.cshtml** и **Old.cshtml**.



На странице **New.cshtml** определим какое-нибудь простейшее содержимое:

|  |  |
| --- | --- |
| 1  2  3 | @page    <h2>New Page</h2> |

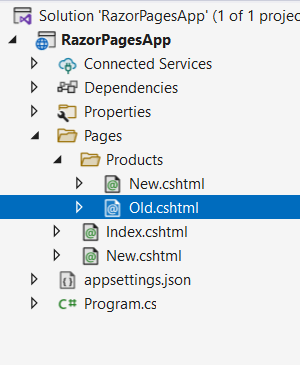
А на странице **Old.cshtml** определим редирект на **New.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @page    <h2>Old Page</h2>    @functions{      public IActionResult OnGet()      {          return RedirectToPage("New");      }  } |

В данном случае для краткости метод OnGet, который обрабатывает GET-запросы и выполняет редирект на страницу Old, определен непосредственно на странице Razor, но также можно было бы сделать редирект и в модели страницы.

И в этом случае при обращении по адресу "/Old" произойдет редирект на адрес "/New".

Стоит отметить, что передаваемый путь объединяется с путем к текущей странице, из которой вызывается данный метод. Рассмотрим некоторые возможные сопоставления передаваемых путей и страниц Razor. Например, страница **Old.cshtml** располагается в папке **Pages/Products**



Допустим, из этой страницы мы выполняем редирект на страницу New.cshtml:

|  |  |
| --- | --- |
| Вызов метода | На какую страницу ведет |
| RedirectToPage("/New") | *Pages/New* |
| RedirectToPage("./New") | *Pages/Products/New* |
| RedirectToPage("../New") | *Pages/New* |
| RedirectToPage("New") | *Pages/Products/New* |

#### Передача параметров маршрута

Рассмотрим передачу параметров маршрута. Допустим, страница **New.cshtml** принимает два параметра маршрута:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @page "{name}/{age}"    <h2>Person</h2>  <p>Name: @RouteData.Values["name"]</p>  <p>Age: @RouteData.Values["age"]</p> |

Для передачи значений для параметров name и age мы могли бы на странице **Old.cshtml** выполнить следующую переадресацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @page    @functions{      public IActionResult OnGet()      {          return RedirectToPage("/New", new{name="Sam", age=28});      }  } |

### Переадресация на другой адрес

Вместо переадресации на конкретную страницу можно выполнить переадресацию на любой адрес. Для временной переадресации применяется метод **Redirect**, в который передается адрес для перехода:

|  |  |
| --- | --- |
| 1  2  3  4 | public IActionResult OnGet()  {      return Redirect("New/Sam/23");  } |

В данном случае идет переадресация на локальный адрес "New/Sam/23".

Ну и кроме того, также можно обращаться к внешнему ресурсу:

|  |  |
| --- | --- |
| 1  2  3  4 | public IActionResult OnGet()  {      return Redirect("[https://metanit.com](https://metanit.com/)");  } |

Для постоянной переадресации подобным образом используется метод **RedirectPermanent**. Принцип его применения тот же самый:

|  |  |
| --- | --- |
| 1  2  3  4 | public IActionResult OnGet()  {      return RedirectPermanent("New/Sam/23");  } |

Для обращения к локальным адресам в нашей системе мы можем использовать класс **LocalRedirectResult**. Для создания временной переадресации применяется метод **LocalRedirect()**, а для создания постоянной переадресации - метод **LocalRedirectPermanent**

Эти методы также принимают адрес ресурса:

|  |  |
| --- | --- |
| 1  2  3  4 | public IActionResult OnGet()  {      return LocalRedirect("~/Home/About");  } |

## Передача зависимостей на страницу

Как и любой класс, Страница Razor и ее модель могут получать сервисы приложения через встроенный в ASP.NET Core механизм dependency injection. Это можно делать следующими способами:

* Через конструктор
* Через параметр метода, к которому применяется атрибут FromServices
* Через свойство HttpContext.RequestServices
* С помощью директивы **@inject**

Например, пусть у нас есть следующий файл **Program.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var builder = WebApplication.CreateBuilder(args);    builder.Services.AddRazorPages();  // добавляем сервис ITimeService  builder.Services.AddTransient<ITimeService, SimpleTimeService>();    var app = builder.Build();    app.MapRazorPages();    app.Run();    public interface ITimeService  {      string Time { get; }  }  public class SimpleTimeService : ITimeService  {      public string Time => DateTime.Now.ToString("HH:mm:ss");  } |

В данном случае определен интерфейс ITimeService и его реализация SimpleTimeService. И в приложении происходит регистрация сервиса ITimeService.

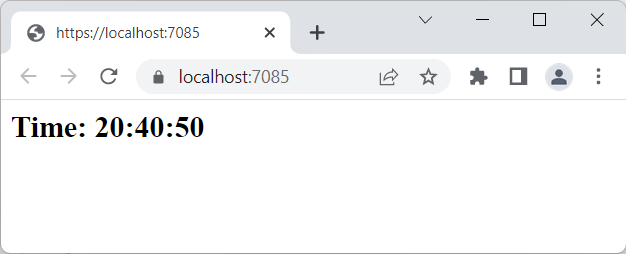
### Передача через конструктор

Получим зависимость в конструкторе модели IndexModel:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; }          public IndexModel(ITimeService timeService)          {              Message = $"Time: {timeService.Time}";          }      }  } |

Выведем значение свойства Message модели на странице **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4 | @page    @model RazorPagesApp.Pages.IndexModel  <h2>@Model.Message</h2> |



### Передача зависимостей в методы с помощью атрибута FromServices

Иногда зависимость используется только в одном методе. И в этом случае нет необходимости передавать ее в конструктор, поскольку она напрямую может быть внедрена в сам метод, который ее использует. Для передачи зависимости в метод применяется атрибут **[FromServices]**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet([FromServices] ITimeService timeService)          {              Message = $"Time: {timeService.Time}";          }      }  } |

Атрибут FromServices определен в пространстве имен Microsoft.AspNetCore.Mvc.

### HttpContext.RequestServices

На странице и в методах модели можно обращаться к объекту контекста запроса через свойство HttpContext, а через свойство **HttpContext.RequestServices** можно получить все зарегистрированные в приложении сервисы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public string Message { get; private set; } = "";          public void OnGet()          {              ITimeService? timeService = HttpContext.RequestServices.GetService<ITimeService>();              Message = $"Time: {timeService?.Time}";          }      }  } |

### Директива @inject

На самой странице Razor также можно получать сервисы с помощью директивы **@inject**. Например, изменим код страницы **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4 | @page    @inject ITimeService timeService  <h2>Current Time: @timeService.Time</h2> |

Директива **@inject** принимает два параметра: первый параметр представляет тип сервиса (в данном случае ITimeService), а второй - название переменной этого типа (в данном случае timeService).

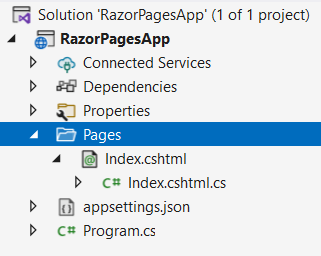
## ViewBag и ViewData

Для передачи данных из кода c# на страницу Razor можно использовать ряд способов. Самый распространенный и предпочтительный - использование свойств модели страницы. Тем не менее есть и другие способы, в частности, примение объектов **ViewBag** и **ViewData**.

### ViewData

**ViewData** представляет словарь из пар ключ-значение.

Например, у нас есть страница Razor **Index.cshtml** и код связанной модели IndexModel в файле **Index.cshtml.cs**:



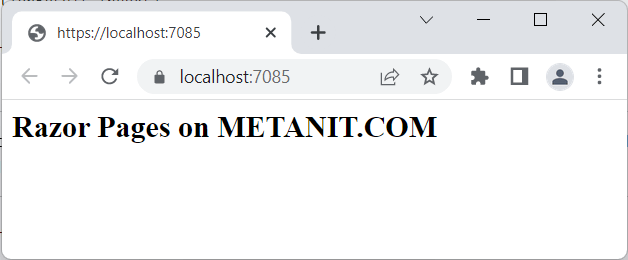
В коде модели IndexModel в файле **Index.cshtml.cs** определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public void OnGet()          {              ViewData["Message"] = "Razor Pages on METANIT.COM";          }      }  } |

В данном случае в словаре ViewData определяется элемент с ключом "Message", значением которого является строка "Razor Pages on METANIT.COM". На странице **Index.cshtml** получим это значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>@ViewData["Message"]</h2> |

Здесь получаем из словаря ViewData элемент с ключом "Message" и выводим его в заголовке.

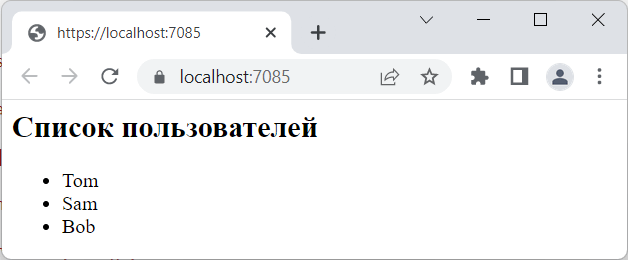


Подобным образом через ViewData можно передавать и более комплексные данные, только стоит учитывать, что в этом случае может потрбеоваться приведение типов. Например, определим в коде модели передачу списка строк:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public void OnGet()          {              ViewData["Message"] = "Список пользователей";              ViewData["People"] = new List<string> { "Tom", "Sam", "Bob" };          }      }  } |

На странице Index.cshtml выведем данные из списка:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>@ViewData["Message"]</h2>  <ul>      @if(ViewData["People"] is List<string> people)      {          foreach(var person in people)          {              <li>@person</li>          }      }  </ul> |



Как видно, здесь потребовалось преобразование данных. С этой точки зрения использование свойств модели вместо ViewData было бы более предпочтительным.

При этом данные ViewData можно определять непосредственно на самой странице Razor:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @page    @{      ViewData["Message"] = "Razor Pages on METANIT.COM";  }    <h2>@ViewData["Message"]</h2> |

В конкретно данном случае применение ViewData не имеет большого смысла, посколько можно было бы просто определить переменную, тем не менее в ряде сценариев может потребоваться передать данные из страницы Razor в другие представления или страницы.

### ViewBag

**ViewBag** во многом подобен ViewData. Он позволяет динамически определить различные свойства и присвоить им любое значение. Так, мы могли бы переписать предыдущий пример следующим образом::

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @page    @{      ViewBag.Message = "Razor Pages on METANIT.COM";  }    <h2>@ViewBag.Message</h2> |

Стоит отметить, что свойство ViewBag можно использовать только на странице Razor, а в классе модели оно не доступно.

На странице Razor также можно получать все данные, переданные через ViewData, через **ViewBag**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @page    @{      ViewData["Message"] = "Razor Pages on METANIT.COM";  }    <h2>@ViewBag.Message</h2> |

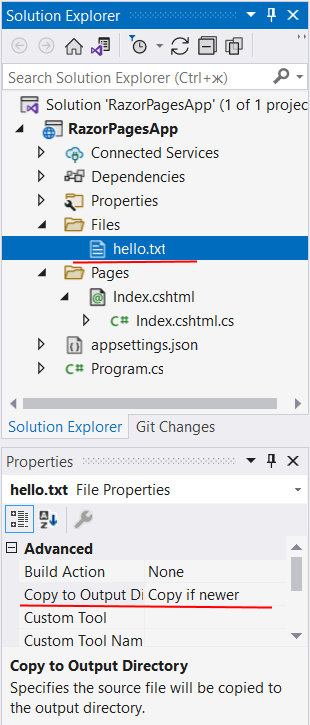
## Отправка файлов в Razor Pages

Для отправки клиенту файлов предназначен абстрактный класс FileResult, функционал которого реализуется в классах-наследниках:

* **FileContentResult**: отправляет клиенту массив байтов, считанный из файла
* **VirtualFileResult**: представляет простую отправку файла напрямую с сервера по виртуальному пути
* **FileStreamResult**: создает поток - объект System.IO.Stream, с помощью которого считывает и отправляет файл клиенту
* **PhysicalFileResult**: также отправляет файл с сервера, но для отправки используется реальный физический путь

Во первых трех случаях для отправки файлов применяется метод **File()**, а для создания объекта PhysicalFileResult используется метод **PhysicalFile()**. Только в зависимости от выбранного способа используется соответствующая перегруженная версия этого метода.

Для примера добавим в корень проекта папку **Files**, в которой находится файл **hello.txt**:



Для файла **hello.txt** установим копирование в выходной каталог.

### Загрузка файла по пути. PhysicalFileResult

Воспользуемся PhysicalFileResult для отправки файла hello.txt клиенту:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          public IActionResult OnGet()          {              // Путь к файлу              string file\_path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Files/hello.txt");              // Тип файла - content-type              string file\_type = "text/plain";              // Имя файла - необязательно              string file\_name = "hello.txt";              return PhysicalFile(file\_path, file\_type, file\_name);          }      }  } |

Чтобы получить полный физический путь каталога относительно проекта, воспользуемся свойством **AppDomain.CurrentDomain.BaseDirectory**, которое возвращает путь к папке, где находится текущее приложение. И при обращении к странице будет загружаться файл hello.txt.

### Загрузка массива байтов

Похожим образом работает и класс **FileContentResult**, только используется метод **File()**, а вместо имени файла передается массив байтов, в который был считан файл:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Отправка массива байтов  public IActionResult OnGet()  {      string path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Files/hello.txt");      byte[] mas = System.IO.File.ReadAllBytes(path);      string file\_type = "text/plain";      string file\_name = "hello2.txt";      return File(mas, file\_type, file\_name);  } |

### Отправка потока. FileStreamResult

Если мы хотим возвратить объект FileStreamResult, то в качестве первого аргумента в методе File идет объект Stream для отправляемого файла:

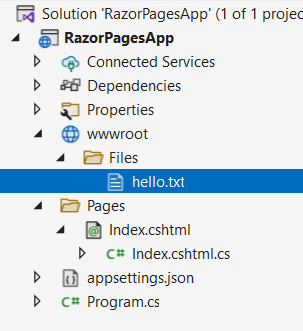
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Отправка потока  public IActionResult OnGet()  {      string path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Files/hello.txt");      FileStream fs = new FileStream(path, FileMode.Open);      string file\_type = "text/plain";      string file\_name = "hello3.txt";      return File(fs, file\_type, file\_name);  } |

### VirtualFileResult

VirtualFileResult работает похожим образом, только возвращает файл по виртуальному пути. Здесь надо учитывать, что по умолчанию все пути к файлам в данном случае будут сопоставляться с папкой wwwroot. То есть нам надо помещать папки с файлами или отдельные файлы в каталог wwwroot:

|  |  |
| --- | --- |
| 1  2  3  4 | public IActionResult OnGet()  {      return File("Files/hello.txt", "text/plain", "hello4.txt");  } |

В данном случае предполагается, что файл "hello.txt" располагается в папке "wwwroot/Files/".



Во всех выше перечисленных случаях использование имени файла в качестве третьего параметра метода File/PhysicalFile необязательно. А вот тип файла обязательно надо передавать. Но подобное поведение может быть не всегда удобным: мы можем точно не знать тип отправляемых файлов, или файлы представляют самые разные типы. И в этом случае мы можем использовать универсальный тип **application/octet-stream**.