# Exercise: Web, Forms and Routing

Problems for exercises and homework for the "Web Dev Basics" course from the official "Applied Programmer" curriculum. In these exercises we will create a very **simple HTTP Server**. We will extend it every time and design it to mimic Microsoft’s IIS.

## Implement Routing

The goal of this task is to **implement routing**, which means that we will be able to **map request URLs** to certain **responses** like this:

Text

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated Graphical user interface, text, application, chat or text message

Description automatically generated

### Step 1: Add Specific HTTP Responses

Our first task in this exercise is to create **different types of responses**, which will be later returned by our server.

Start by creating a **folder** called "Responses" in our "BasicWebServer.Server" **project** and there we will put our **response classes**:

Graphical user interface, text, application, chat or text message

Description automatically generated

We will create **two types of responses** here: **action responses** and **content responses**. Our action responses will include responses for **different status codes**, except for the "200 OK", as when the request is successful a content response is returned. We will define **action classes** for "302 Found", "400 Bad Request", "401 Unauthorized" and "404 Not Found" **responses** and **content responses** for returning a **plain text** or **HTML**.

Classes should have the **relations**, shown in the diagram below:

Diagram

Description automatically generated

Start by **adding status codes** to the StatusCode **enum class** for the **action responses** like this:

Text, letter

Description automatically generated

As you know, different **HTTP responses** are defined by their **headers**. Each **header** should be **added to the response**, which means that we should be able to **iterate through the headers collection**. For this reason, our HeaderCollection **class** should **implement** the IEnumerable **interface** and have the **two** GetEnumerator() **methods**. **Modify** the HeaderCollection **class** like this:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now let’s create a **base class for all action responses**. They should all **inherit** **the** Response **class**. Let’s create the "Bad Request", "Not Found" and "Unauthorized" **response classes**, as they are all with the **same structure**. Each of them should only **use the** StatusCode **class** and set the corresponding **status code** of the response.

Write the BadRequestResponse, the NotFoundResponse and the UnauthorizedResponse **classes** like this:

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Our last response is the "Redirect" **response**. It also sets a **status code** but also an **additional header** for the **location** (the URL of the page, which the server should redirect to).

Before we write the RedirectResponse **class**, let’s add **all header names** to the Header **class** as **constants**, so that they are not hardcoded anymore. Add the following lines to the Header **class**:

Text

Description automatically generated

We can also now improve the Response **class** by using the **constants** from above:

Graphical user interface, text, application

Description automatically generated

Now let’s go back and write the RedirectResponse **class**, which should **set the status code** and **add a location header**:

Graphical user interface, text, application

Description automatically generated

We already have classes for our **action responses**.

A screenshot of a computer

Description automatically generated with medium confidence

Let’s create **content responses** now – one to **return the response** **as HTML** and one to **return it as a plain text**. It is a good idea to create a class with **constants for those content types**. Create the ContentType **class** in the "HTTP" **folder** and **add the constants** like this:

Text

Description automatically generated with low confidence

We want to **create a base class** called "ContentResponse", which should be a **base class for all content responses** and should **inherit the** Response **class**. The ContentResponse **class** should also **accept content** and **content type**. The **content type** should be added as **header values** to the HTTP response and its **body** should have the **content**. If you remember, we also need to **add the** "Content-Length" **header** to the **response** but we will see how to do this later.

Write the **class** as shown below:

Graphical user interface, text, application

Description automatically generated

Create the **HTML** and **plain text response classes**. They should both **accept the content** and **set a content type** from the ContentType **constants class**. Write them as shown below:

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated with medium confidence

We will see how to use these classes to **return specific HTTP responses** to the browser later in the exercise.

### Step 2: Implement Routing Table

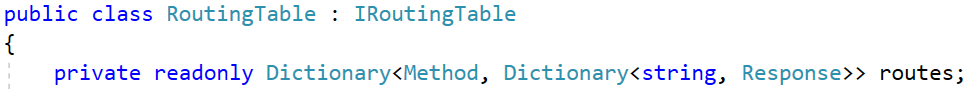
Our task now is to **implement a routing table** to **map an URL** to **an HTTP response**. First, **create a folder** called "Routing" in the "BasicWebServer.Server" **project**, in which we will **create our routing classes**.

Next, we will create the IRoutingTable **interface**, which will have a **method**, accepting an **URL**, an **HTTP method** and a **response**, and **another two methods for "GET" and "POST" requests**, accepting only an **URL** and a **method**. The **interface** looks like this:

Graphical user interface, text

Description automatically generated

Create the RoutingTable **class**, which should implement the IRoutingTable **interface**. It should also have a **nested dictionary field**, in which the **HTTP method is assigned an URL and an HTTP response** – these are the **routes**. Write the class and its field:



Next, create a **constructor**, which **initializes the main dictionary**. Then, depending on the **method**, it should also **initialize the nested dictionary**:

Graphical user interface, text

Description automatically generated

As you can see, the code structure is very unpleasant. It can be improved with the new **operator**, which you can use to **initialize a variable of a known type**. Modify the **constructor**:

Text, letter

Description automatically generated

Let’s implement the Map(string url, Method method, Response response) **method**, which **invokes another method**, depending on the **HTTP method,** or **throws an error**. For now, we will support only the "GET" and "**POST**" **HTTP** **method**. Write the mapping method:

Graphical user interface, text, application

Description automatically generated

The MapGet(string url, Response response) **method** should **map the given URL to the HTTP response** (fill in the nested dictionary). The method should return the **current** IRoutingTable **instance** when done, so that we can **chain mappings**:

Graphical user interface, text, application

Description automatically generated

The **mapping method** for the "POST" **request** is almost the same but it **differs in the method** – it should be "POST":

Graphical user interface, text, application

Description automatically generated

Finally, we should also **add a method to get the HTTP response of an existing HTTP request**, as we are going to need it. In this method, **get the HTTP method** and the **URL** of the **request** and **return the response** as shown below. If the request is not present in the dictionary, return a "Not Found" **response**:

Graphical user interface, text, application, email

Description automatically generated

Our **routing table** now supports "GET" **requests**. Let’s see how to use it to map requests to responses and see the response in the browser.

### Step 3: Use the Response Classes and Routing

Our first task is to **integrate the routing table in our server**. Go to the HttpServer **class** and **add a field for the** **routing table**. The **constructor** will accept a **function** Action<IRoutingTable>, not just IRoutingTable, as we want to enhance the way we will define routes later. Add routing to the server like this:

Graphical user interface, text, application, email

Description automatically generated

Let's create **two more constructor methods**, which use our current one. The reason to do this is to have a **default IP** **address** and **port**, so that they should not be given every time. Create the additional constructors:

Graphical user interface, text, application

Description automatically generated

Until now, the Start() **method** of the HttpServer **class** simply **printed the HTTP request** from the browser on the console. Now we want to readthe **request**, parseit to an **HTTP request**, get the **response** of that request from the **routing table** and **write the response to the network stream**. Modify the Start() method. We already have the **request text** – use the Parse(string request) **method** of the Request **class** to **parse the request** to an HTTP request:

Graphical user interface, text, application, email

Description automatically generated

Then, use the MatchRequest(Request request) **method** of the RoutingTable **class** to **get the response**:



Finally, **write the whole response to the network stream**. The WriteResponse(NetworkStream networkStream, string message) should be modified to **accept the** **response**, which we will do on the next step. Now **pass the response** to the method and close the Start() method:

A picture containing text

Description automatically generated

Let's modify the WriteResponse(NetworkStream networkStream) **method**, as well. If you remember we **hardcoded** **a response** like this:

Graphical user interface, text, application, email

Description automatically generated

As we have the response, **improve the method to accept it** and **write it to the network stream**. Note that the HTTP response should be converted to a **byte array**. To make that possible, it should first be **converted to a string in a valid** **HTTP response message format**. That's why we will override the ToString() **method** of the **HTTP response** in the Response **class**:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

In the ToString() **method** we should **initialize a** StringBuilder **variable** to **keep our request message lines**. Add the **response status line**, each of the **headers** and the **body** on **separate lines**. Note that the **body may be empty** and should not be appended. Return the StringBuilder as a **string**. The whole method is the following:

Graphical user interface, application

Description automatically generated

Note that we use the ToString() **method** of the Header **class** to **convert the headers to string**. However, we need to **override** that method, too, so that our headers are in the format we want. Go to the Header **class** and **override** the ToString() **method** like this:

A picture containing table

Description automatically generated

A picture containing table

Description automatically generated

Also, we need to modify the ContentResponse **class**. It uses the ToString() **method** of its **base** Response **class** but it should **add** the "Content-Length" **header** when the **body has content**. We should not add the header in the class **constructor**, as the **body may change** after the initialization of the response class, and the **content length value** won't be correct – this will result in a **wrong display of the content**.

Go to the Content **class** and **override** the ToString() **method**. If the **body is not empty**, add the "Content-Length" **header**, which should **get the body content length** in **bytes**. Finally, **invoke the base method**:

Graphical user interface, text, application

Description automatically generated

Now go back to the HttpServer **class** and **modify its** WriteResponse(NetworkStream networkStream, string message) **method** to **accept a response**, turn it to a **byte array** and **write it to the network stream**:

Graphical user interface, text

Description automatically generated

Now our server can **read HTTP request** from the browser and **return a response** because of the **routing** we added. To use the routing, we should **create a routing table** and **map an URL to a response**.

Do this in the Startup **class** of the "BasicWebServer.Demo" **project**. Use the MapGet(string url, Response response) **method** of the RoutingTable **class**. Try **mapping URLs to different responses**. We will try to **return a text**, an **HTML** and a **redirect response**:

Text

Description automatically generated

Note that we are able to **chain the mapping commands** because we used the Action<IRoutingTable> **function** in the HttpServer **class**. Try the **routing in the browser**. Run the "BasicWebServer.Demo" project and go to the localhost **address** with the **correct port**. The **correct text response** should be displayed on the browser:

Text

Description automatically generated 🡪 Graphical user interface, text, application, chat or text message

Description automatically generated

**Examine the request method** and **status code** and the **returned response headers** in the browser DevTools:

Graphical user interface, text, application

Description automatically generated

Try **accessing** the "/HTML" **route** – it should **return the response text** in an **HTML format**:

Graphical user interface, text, application

Description automatically generated

Try the **redirect response** on "/Redirect", as well. You should be **redirected** to the <https://softuni.org/> **site**. Examine the **request method** and **status code**. Also, note that the **response headers contain** the one for the **location**, which we added especially for the **redirect response**:

Graphical user interface, text, application, chat or text message

Description automatically generated

Now we have **routing in our HTTP server** and can **return different types of responses**.

## Add Form Functionality

In this task we will **create a simple HTML form** and see **how to obtain and use the submitted form data**. We want to have a **form for submitting name and age**. When we **enter data** and **click on the** [Submit] **button**, a "POST" **request** should be sent to the server. Then our server will **display the form data**, which was submitted, as a **text**.

Graphical user interface, text, application

Description automatically generated 🡪Graphical user interface, text, application

Description automatically generated 🡪Graphical user interface, text, application, chat or text message

Description automatically generated

### Step 1: Create the Form and Map Requests

As you can see, we should visit the "/HTML" **route** and a **form should appear**. Create the form as a **string constant** and use it as a part of an **HTML response** in the Startup **class** of the "BasicWebServer.Demo" **project**. The form should have **action** and **method** **attributes** as shown below:

Text

Description automatically generated

You can copy the **form field** from here:

|  |
| --- |
| private const string HtmlForm = @"<form action='/HTML' method='POST'>  Name: <input type='text' name='Name'/>  Age: <input type='number' name ='Age'/>  <input type='submit' value ='Save' />  </form>"; |

Now we should **set the action** of our form to "/HTML" and the **method** to "POST". This means that when the form is submitted, it will send a "POST" **request** to the "/HTML" **path**:

Graphical user interface, text

Description automatically generated

If you **open the browser** now and visit "http://localhost:8080/HTML", you should see the form we created. However, if you press the [Submit] **button**, you will not see the form data on the page, as we returned an **empty** TextResponse:

Graphical user interface, text, application, chat or text message

Description automatically generated

However, the **form data** we sent is part of **our request body** and you can see it in the "**Payload**" tab:

Graphical user interface, application

Description automatically generated

Our task now is to **read that data separately from the body** and **use it**.

### Step 2: Implement Form Parsing in the Request

Let's see how to make our **server read form data**. Before that, do some **additions to classes**, which we will need later. Go to the ContentType **class** of the "BasicWebServer.Server" **project** and **add a constant for the form content** **type**, as we will need it to parse the form:

Text

Description automatically generated with medium confidence

Also, in the HeaderCollection **class** we will need a Contains(string name) and an **indexer**. Indexers allow instances of a class to be indexed just like arrays and the indexed value can be set or retrieved without explicitly specifying a type or instance member. Our Contains(string name) **method** will return whether there is a **header with the given name**.

**Add the indexer** and the **method** to the HeaderCollection **class** like this:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now we can see **how to parse and use our form data**. In addition, it is a good idea to **modify** the Add(string name, string value) to **use the indexer** – this way if the header already exists it **won't be added** to the collection, which **prevents you from duplicating headers**. Change the method like this:



The **form data** is part of our **requests**, so modify the Request **class**. Start by **adding a field for the form data**, which should be a **dictionary holding key-value pairs** for the **name** and **value** of each **form field**:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Then, the **parsing of the form** should be part of our Parse(string request) **method**. The form parsing will be in a **separate method**, which we will implement later. Now **get the form** and **make it a part of the request** in the method like this:

A picture containing text

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Now **implement** the ParseForm(HeaderCollection headers, string body) **method**. It should **return a collection of form data pairs**. Create the **method** and **initialize the collection** like this:

Graphical user interface, text

Description automatically generated with medium confidence

Check whether the **headers** contain the "Content-Type" **header** for **forms**:

Text

Description automatically generated

If the **if-check returns** true then we have a **form** and we need to **parse** it. For parsing we will create a **separate** **method** in the Request **class** – ParseFormData(string bodyLines). The method will **accept the request body** as a string, **decode** it and **split it into parts** to get the **key** and **value** of **each pair of form data**. It looks like this:

Text

Description automatically generated

|  |  |
| --- | --- |
| Icon  Description automatically generated | You need to **decode** the **URL-encoded form data**, using HttpUtility.UrlDecode(string). Otherwise, special characters like "**/"** or " " in the form fields will remain encoded as "%2F" or "+". |

**Use this method** in the if-statement from above and **parse the** **form to a dictionary**. Then, **add each key-value pair to** **the form collection**. At the end, you should just **return the collection** to the method. The whole ParseForm(HeaderCollection headers, string body) **method** is the following:

Graphical user interface, text, application

Description automatically generated

### Step 3: Modify Response Result

Now we have the functionality to **obtain the form data from the HTTP request**. Now we should **return a response**, visualizing the data in the way we want.

First, go to the Startup **class** and **modify** the "POST" **request**, which returns a response when the **HTML form** is **submitted**:

Text

Description automatically generated

As you can see, we have a **second parameter** in the TextResponse **class** **constructor**: this is the **name** of a **pre-render** **action**, which will **use the request** and **modify the response** before it is returned to the browser. Let's see how we will **modify the response** to be as we want it.

First, we will allow the **action to be executed in the response**. Go to the Response **class** and **add** the PreRenderAction **property**, which accepts a Request and Response and **does not return a result**:

Graphical user interface, text, application, email

Description automatically generated

Then, **modify** the ContentResponse **child class** to **accept the action** through the **constructor** as an **optional parameter** and set it like this:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

At the end, go to the TextResponse **class** and **change its constructor**, as well:

Graphical user interface, text, application

Description automatically generated

We have the **pre-response action property** in the Response. It should be **executed after the request** is parsed from the browser, as its **form data** is used, and **before the response is returned** to the browser. That's why we should add the following lines to the while-**loop** of the Start() **method** in the HttpServer **class**:

Text

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

### Step 4: Create a Pre-Render Action and Try It Out

Now we can add an **action** to be executed before the response is returned. Go back to the Startup **class** of the "BasicWebServer.Demo" **project** and **create** the AddFormDataAction(Request request, Response response) **method**, in which we will **add the form data to the response body**:

Graphical user interface, text, application

Description automatically generated

In this method, you should only **clear the response body content**, so that only current data is displayed, and **add each key-value pair of form** **data** to it like this:

Graphical user interface, text

Description automatically generated

Now try the form parsing. **Connect to our server** in the browser, **go** to "/HTML", **fill in the form** and see if the **returned form data is correct**. Examine the **requests** and **responses** in the Network **tab** in the DevTools:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated