# Exercises: Web Server and HTTP Protocol

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.

## Create the Web Server

The goal of this task is to **implement a very simple HTTP server**, which can respond to a standard "GET" **request**:

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

Description automatically generated

To begin with, create a new **console app** in Visual Studio named "BasicWebServer":

Картина, която съдържа текст

Описанието е генерирано автоматично

To **create a Web server**, we need make the **browser connect to our console application** and **receive a response in a valid HTTP format**. For this reason, we should create and open a **network connection**, which our app will use to connect to the browser. As the server and the app are **on the same machine**, they should **connect through the** localhost **IP address** (127.0.0.1).

### Step 1: Implement a Server Functionality

Let’s start by writing **the whole code** in the Main() **method** of the Program **class** so that it is simpler. First, we will use the IPAddress **class** from the System.Net **namespace** to **create an IP address instance with the locahost address**:

Картина, която съдържа текст

Описанието е генерирано автоматично

Next, we should **choose a port**, on which our console app will work. Choose **any free port** except for the default ones (80 and 443) as they may be taken, for example 8080:



With this **IP adress** and **port**, our app will be on <http://localhost:8080>. Now we need to **make the browser connect to this address and port and to our app**. To do this, we will use the TCPListener **class**, which allows us to **accept requests from the browser**:



**Start the server listener** like this:



Now we **have the connection to the browser**, but if we run the app nothing will happen:

Картина, която съдържа текст

Описанието е генерирано автоматично

The reason for this is that we need to **make our server wait for the browser to connect to it**. Get the connection from the browser like this:



Let’s **run** our console app again. For better understanding, let’s first **add some comments** **on the console** to know that the server is started. Also, **put a break point** after the connection is accepted to **debug**:

Text

Description automatically generated

**Run the app** in **debug mode**:

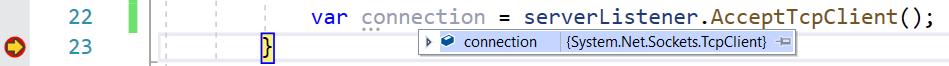
Text

Description automatically generated

Our server is **listening for requests**. Open a **browser** and go [to http://localhost:8080](http://localhost:8080). A **connection** will be made between the **browser** and our **server**:

Graphical user interface, application

Description automatically generated 🡪



As we already **have a connection**, now we need to **return a response from our server to the browser to visualize**. To do this, first we need to **create a stream**, through which **data is received or sent** to the browser as a **byte array**:



We will use this **network stream to send our response** to the browser. Createa **message**, which will be sent, and get its **length in bytes** (bytes length is often different from the string length):



Then, let’s **construct our response**. It should be in **HTML format** and have the "Content-Type" and "Content-Length" **headers**. The content type will be "text/plain; charset=UTF-8" as we want to **send the content as a plain text** with the **UTF-8 encoding** to accept more symbols (for example from the Bulgarian alphabet). Note that you should not have **excessive spaces** in the response, as they break the **HTML format**. **Write the response** and **convert it to a byte array**:

Картина, която съдържа текст

Описанието е генерирано автоматично

Use the **network stream** to **send the response bytes to the browser**:



At the end, it is important that we **close the connection** to the browser or it may remain open and other connections to the server will fail:

A picture containing graphical user interface

Description automatically generated

Now **run the server** and **open the browser**. The **response content should be visualized on the browser page**. Look at the request and response in the browser DevTools:

Картина, която съдържа текст

Описанието е генерирано автоматично

As you can see, our server accepts **only a single request** and stops. To make it **accept many requests**, add a while(true) **loop** to create and close a connection every time:

A picture containing graphical user interface

Description automatically generated

A picture containing graphical user interface

Description automatically generated

### Step 2: Create a Separate HTTP Server Class

Now let’s create a **new project** of type "Class Library", in which we will separate our **Web server to methods**. Name it "BasicWebServer.Server". The solution should contain **two projects** now. Delete the "Class1" class, as we don’t need it:

Картина, която съдържа текст

Описанието е генерирано автоматично

Let’s create the HttpServer **class** first. This will be our **server class**, which will have **fields** for the **IP address**, the **port** and the **TCP listener**. It should also have a **constructor**, which accepts an **address** and a **port**. Get the **constructor** **content** from the Program **class** in the "BasicWebServer" **project**:

Картина, която съдържа текст

Описанието е генерирано автоматично

Add a Start() **method** to **start the server** like this:

Картина, която съдържа текст

Описанието е генерирано автоматично

Graphical user interface, text, application, email

Description automatically generated

As you can see, we have a **special method for writing the response in the network stream**. The WriteResponse(NetworkStream networkStream, string message) **method** looks like this:

Graphical user interface, text, application, email

Description automatically generated

Now we have the HttpServer **class** with all the code, which was previously in the Program **class** of the "BasicWebServer" **project**. We don’t have **content** in that class anymore so we can just **remove the whole project**, as this is its only class.

**Create a new console app** named "BasicWebServer.Demo", which will **use the server**. Note that it should **reference** the "BasicWebServer.Server" **project** to access the server classes. You can also rename its Program **class** to Startup. The **solution structure** now looks like this:

Картина, която съдържа текст

Описанието е генерирано автоматично

The Startup **class** will **start the** **server** with an **IP address** and **port** in the Main() **method** like this:

Graphical user interface, text, application

Description automatically generated

## Create Additional Classes

Our task now is to **create separate classes** for the **HTTP request** and the **HTTP response**, which we will use later as they are an important part of the **communication between the browser and our server**. Create a **folder** called "HTTP" for the **HTTP-connected classes** and **create all additional classes in it**:

Картина, която съдържа текст

Описанието е генерирано автоматично

### HTTP Response Class

Let’s start by **creating a simple class** for the **HTTP response**. As you already know, the **response** consists of a **start lin**e, **headers** and **content** (body). The **status line** consists of an **HTTP version** and a **status code**. We need to **create a separate enumerator class** for the **status code**, as it changes and is important for our response.

Create the StatusCode **enum** **class**, which will only have the "200 OK" **status** for now:

A picture containing text

Description automatically generated

Then, we also need to **create a header collection class** for the **response headers**. Before that, however, let’s create a **class** for the **HTTP header**. The **header** has a **name** and a **value**, accepted through the **constructor**. We will leave the header value **accessible for modification** – you will need to modify it later in the exercises. The Header **class** looks like this:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now we can **create the** HeaderCollection **class**, which has a **collection of type** Header, a **constructor**, a **property** for the **headers count** and a **method for adding a header**. We will keep **headers as a dictionary** with the **header name as a key** and **the header itself as a value**, so that we can our searching in the headers is easier.

The HeaderCollection **class** with its components looks like this: Graphical user interface, text

Description automatically generated with medium confidence

As we already have classes for the **status code** and the **headers** of the **HTTP response**, let’s **create the** Response **class**. Its **constructor** accepts a **status code** and **adds some default headers**:

Graphical user interface, text

Description automatically generated

### HTTP Request Class

Now we will **create a class for the HTTP request**. The **HTTP request** consists of an **HTTP request line** (with an **HTTP** **method**, a **URL** and **HTTP version**), **headers** and a **body**.

Start by creating an **enum** **class** for the **HTTP method**, as methods are predefined. Write the Method **class** like this:

Text

Description automatically generated

We already have **all helper classes** to create the Request **class**. The Request **class** should have **properties** for the **method**, the **URL**, the **headers** and the **body** of the request. It should also have a **method**, which **accepts a request as a string and parses it to a** Request. The **class** **properties** look like this:

Graphical user interface, text, application

Description automatically generated

To **parse the request string** to an **HTTP request** we need to first **separate each line** and **get the first one**, which contains our **method** and **URL**, split by a **space**:

Text

Description automatically generated with medium confidence

The **start line** is an **array** with the **method** and the **URL** as **strings**:



However, we need to **parse the given method string to an HTTP method**. That’s why we have used a **separate method** called ParseMethod(string method), which **tries to** **parse the method** and **return it** like this:

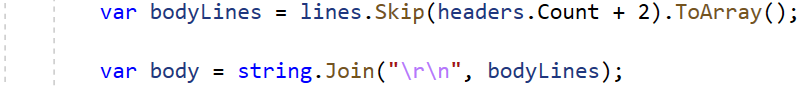
Graphical user interface, text, application, email

Description automatically generated

Take the **headers**, starting from the second request line. They also need **parsing**, which will happen in the ParseHeaders(string headers) **method**, which we will implement later:



Then, we need to get the lines of the body part of the request. **Skip** the start line and the header lines, **get the body lines** and **join them to form the body part**:

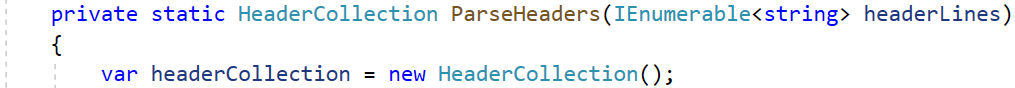


Finally, return the **parsed HTTP request** with all its components to finish the Parse(string request) **method**:

Text

Description automatically generated

At last, let’s write the **method** for **parsing the request header lines**. First, initialize a **header collection**, which the method will return:



Then, **for each of the header lines** we need to **separate the header name and value** by a colon ("**:**"), instantiate the Header **class** and **add the header to the header collection**. We have also added some **validations** against errors:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Now you should just **return the collection** like this:

A picture containing text

Description automatically generated

## Reading Requests

As we now have a **class for the HTTP request**, let’s see how to **read the request from the browser**. We will do this, as we will need to **use the request** in the following exercises. Go back to the HttpServer **class** and let’s write the ReadRequest(NetworkStream networkStream) **method**, which will use the **network stream** to **get the HTTP** **request** from the browser as **bytes** and **return it as a string**. Also, we will **create a buffer** to read the request **in parts**, as our server may crash if the request is **too large**.

First, our method should **accept a network stream** to read from. Our buffer for reading will have a **length** of **1024** **bytes** and will be a **byte array**. It should look like this:

A picture containing text

Description automatically generated Text

Description automatically generated with medium confidence

Now let’s create a StringBuilder, to which we will **append the request strings** and which will be **returned as a string** to the method:



Next, create a do-while **loop**, which will **read bytes from the network stream**, **parse** them into a **string** and **append** the **string** to the StringBuilder. The **loop** should be active until **there is no more data from the stream**:

Text

Description automatically generated

Our last step is to **return the** StringBuilder **as a string**:

Text

Description automatically generated

|  |  |
| --- | --- |
| Icon  Description automatically generated | Warning: the above code **may not run correctly** when the request is sent over the Internet, because packets can be delivered with a delay over the network. Delayed packets will cause NetworkStream.DataAvailable to return **false** before the end of the HTTP request.  Fixing this problem is **complicated**: you need to read and follow the Content-Length header. We shall use the above simplified approach, but keep in mind, that it can fail in real-world scenarios.  Learn more at: [https://stackoverflow.com/questions/4261365](https://stackoverflow.com/questions/4261365/) |

We now have the ReadRequest(NetworkStream networkStream) **method**, which provides us with the **HTTP request** from the browser as a string. For now we will just **write the request to the console**. In the following exercises we will use the request with its content to add routing to our server. For now, you just need to **write the request text to the console**. Do this in the Start() **method** of the HttpServer **class**:

A picture containing text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

**Run the server** again – it should **write the HTTP request on the console**. Start the "BasicWebServer.Demo" **console app**, **open the browser** and go to "http://localhost:8080/". Look at the console – it should **show the** GET **request** from the browser:

Text

Description automatically generated

If you **refresh the page**, a **new request will be read and written on the console**.

## Prevent Too Long Request

In our HttpServer **class** we take care to **read the browser HTTP request in parts**, so that the server doesn’t crash from too much data at once. However, it is a good idea to **check the length of the request** and **stop the reading** if it is **too large**. To do this, we will **check the total bytes received in the request** and **throw an error** if they exceed a certain count.

Go to the ReadRequest(NetworkStream networkStream) **method** in the HttpServer **class** and **add the following lines**:

A picture containing text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

## Add a Null-Checking Class

When creating a software, which relies on user input, it is a good idea to **add some checks of input data**. That’s why we will **check every time when we create a class** if there are any properties with a NULL **value**, as they may be problematic. To do this, let’s create a **static class** Guard, which should **check for null values**. The Guard **class** will be in a **folder** called "Common" in the "BasicWebServer.Server" **project**:

Картина, която съдържа текст

Описанието е генерирано автоматично

The Guard **class** will have **a single method** called AgainstNull(object value, string name = null), which will **accept a value and optionally a name**. If the **value is null**, an **exception** will be thrown. The class looks like this:

Картина, която съдържа текст

Описанието е генерирано автоматично

Then, **use the class** in the Header **class** to **check the header name and value** in the **constructor** like this:

Text

Description automatically generated with medium confidence

Next times we will continue **extending our server** with more and more functionalities.