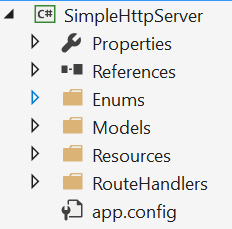
# Lab: Handmade HTTP Server

Problems for exercises and homework for the [“C# Web Basics” course @ SoftUni](https://softuni.bg/courses/csharp-web-development-basics).

Following to the end this document will help you to create your own HTTP Server (similar to Apache). Later in the course we will use it as starting point when we are building our custom MVC framework.

## Create New Project

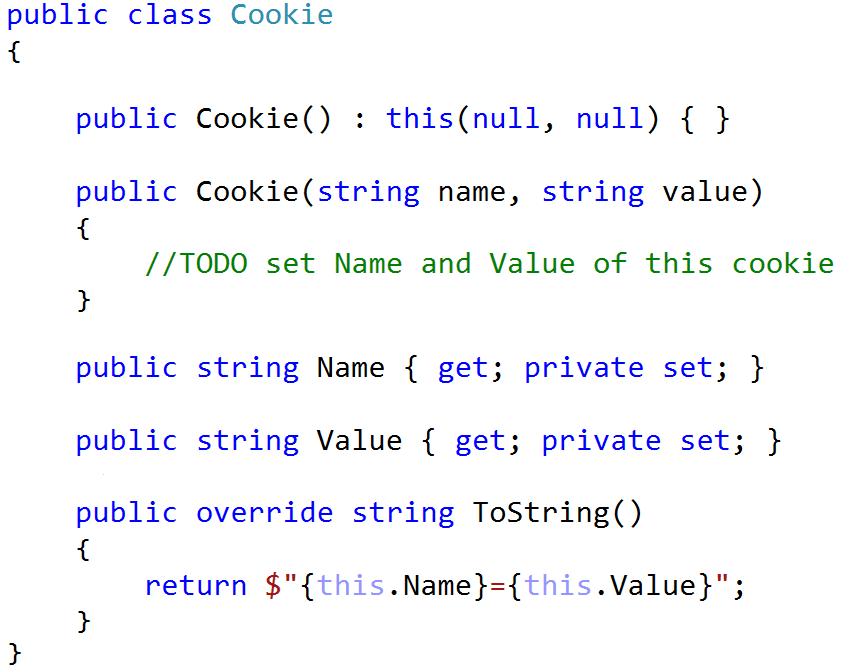
Create new **Class Library project** with name of your choice and folder structure like the picture below.



## Cookie and CookieCollection

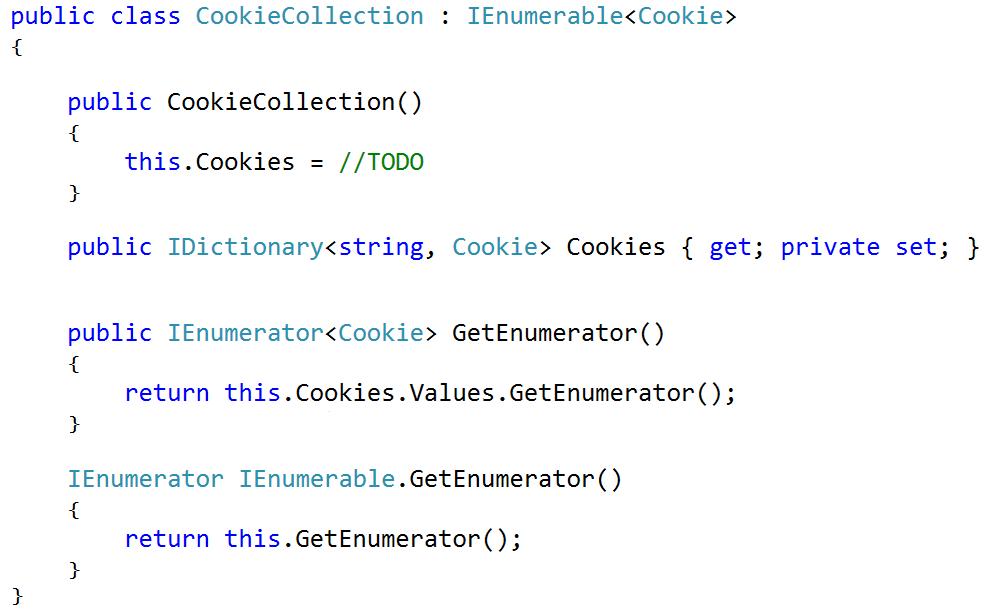
The first models we would create are the Cookie and the CookieCollection. Cookie is a simple key value pair of 2 strings. CookieCollection is a data structure that would keep set of Cookies but would have methods for easily and fast adding and searching for a cookie.

In the Models folder add 2 classes Cookie and CookieCollection.



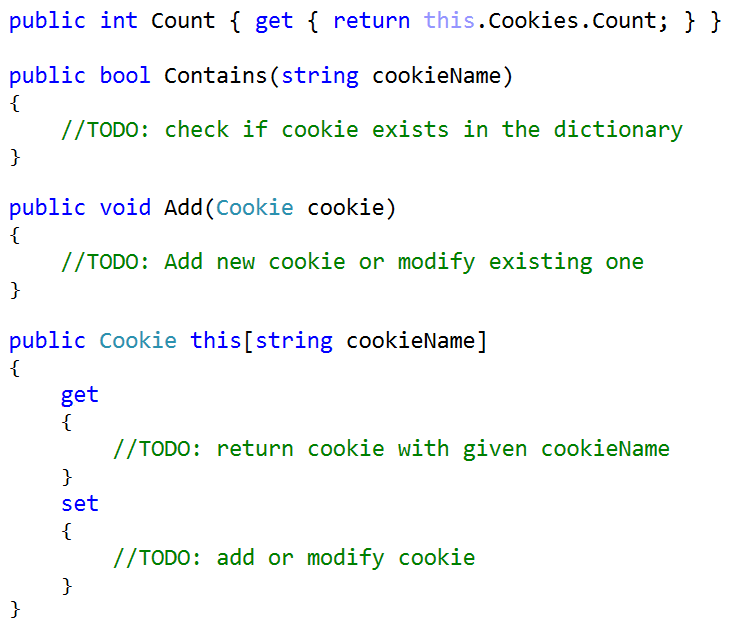
We’ve overridden ToString() method to print the cookie in its native way that is transferred by the requests.

The class CookieCollection class would implement the IEnumerable<Cookie> interface so the collection can be easily iterated and we would use a Dictionary<string, Cookie> to keep all cookies. The **key** of the dictionary would be the **name of the cookie** and the **value** would be **the cookie itself**.

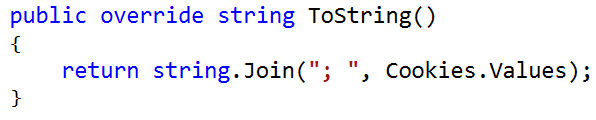


We need to add methods for:

* **Adding new cookie** to the collection
* Check if **cookie exists** by given name
* **Count** of all cookies
* **Modifying value** of a cookie
* **Retrieving cookie** by given name

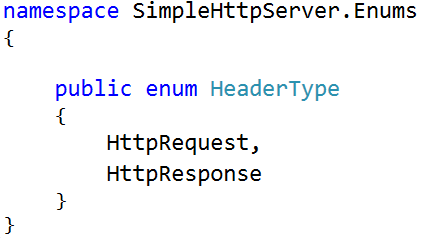


You can also override the ToString() method of the collection so it will return all the cookies separated with “; “ (semi-column and a single space)



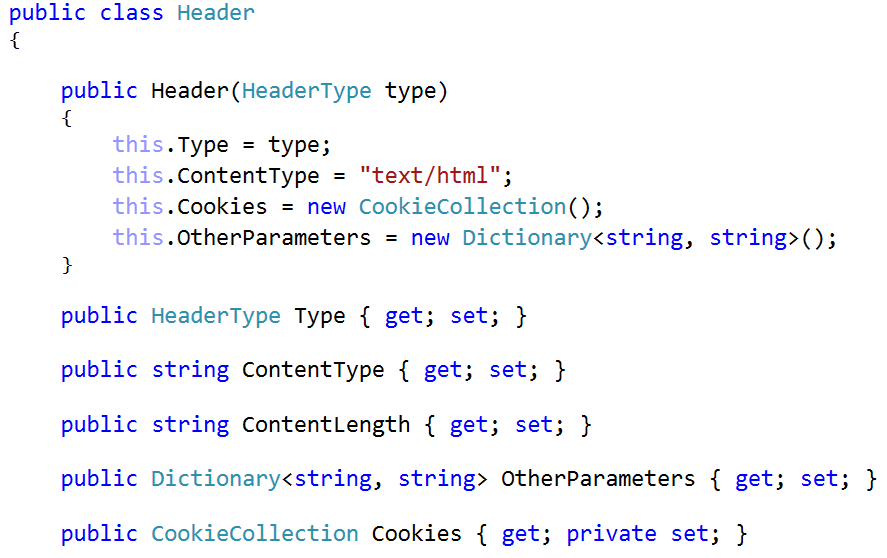
## Header

There are 2 types of headers – **HttpRequest** and **HttpResponse**. Create new enumeration HeaderType with those 2 values.

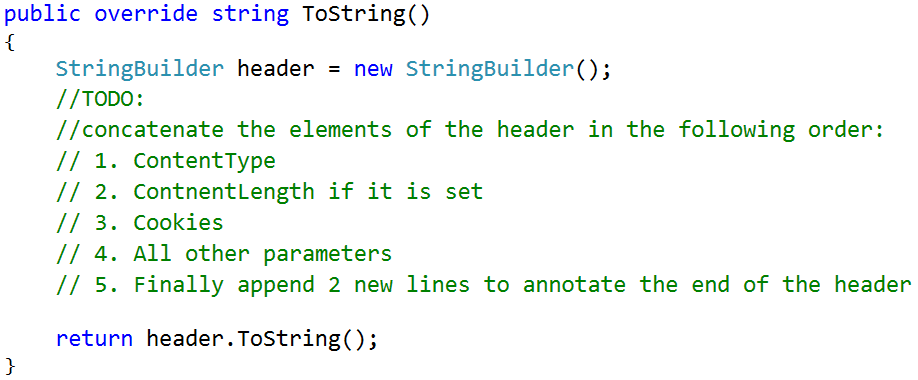


In the Models folder create Header class that would be responsible for keeping all the information for each HTTP request or response. It should have properties for **Content Type,** **Header Type, Content Type, Content Length Collection of Cookies** and dictionary that would keep all **other parameters of the header**.

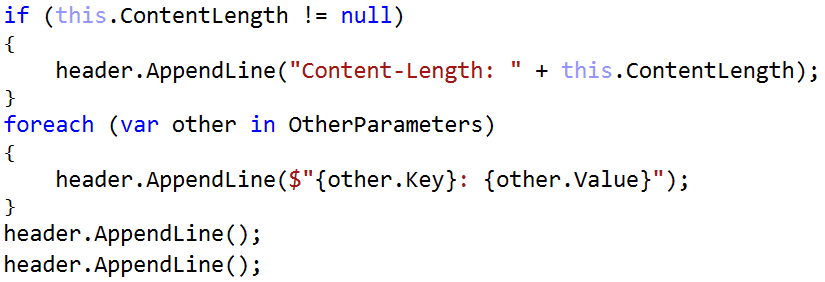
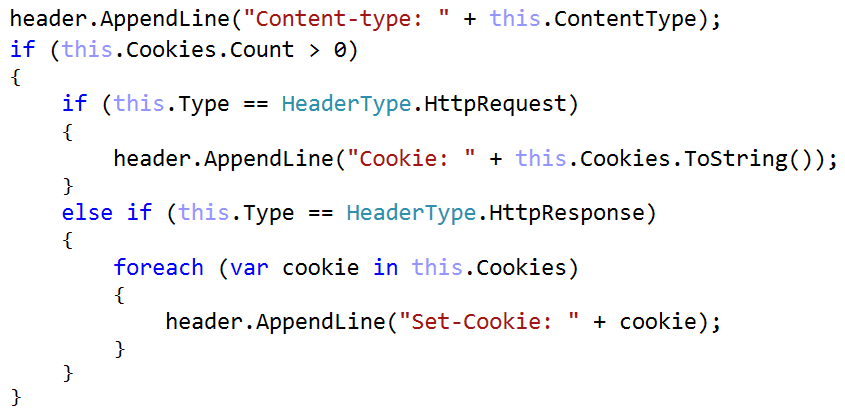
When the header is created, its type must be set. The **default Content Type** of a Header is “text/html”.



Then we need to override the ToString() method so it prints the header accordingly to the **HTTP standard**. (field-value pairs). Check in the internet to see how cookies are sent or retrieved depending on the header (whether is http request header or http response header). Take that in mind when overring the header ToString() method.



If you find it difficult [search](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview) in internet to see how http request and response headers should look like or see the solution below.



Notice how we iterate over the collection of cookies and print each in format “Set-Cookie: {cookie}”. We cannot retrieve them at once separated with “; “. That is because actually the cookie can keep more information than just name and value. For example, the cookie can also keep when it will expire so in that case when setting a cookie for language preference and session might look like this:

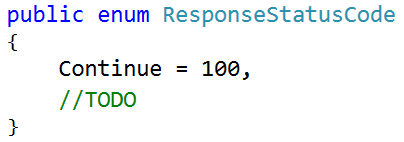
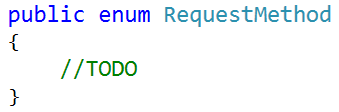
Set-Cookie: lang=en

Set-Cookie: SID=2sfx343; Expires=Wed, 21 Oct 2017 07:28:00 GMT;

But for sake of simplicity we will keep our cookies to keep only name and value.

## Request Method and Status Code

Create 2 more enumerations – **Request Method** and **Response Status Code**. For the sake of simplicity our HTTP server will only provide functionalities for **POST** and **GET** request methods. The status code is actually the code that is returned along with the HTTP response. Here is a [**list of all Response codes**](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)**.**



## HTTP Request

Create HttpRequest class in the models folder. The request has

* **Method** – request method
* **Url** – the path of the resource to fetch
* **Header** – header of type HTTP Request
* **Content** – if the request method is post here it will be kept the data send from the POST request

Override the ToString() method so it can print the Request accordingly to the HTTP standard.

{Method} {URL} HTTP/1.0

{Header}

{Content if present}

### Examples

|  |  |
| --- | --- |
| **GET Request Example** | **POST Request Example** |
| GET /users/register HTTP/1.0  Host: localhost:8081  Accept-Language: en | POST /users/add HTTP/1.0  Host: localhost:8081  Accept-Languagage: fr  username=potter&password=123 |

## HTTP Response

Create HttpResponse class in the models folder. The response has properties:

* **Status code** - indicating if the request has been successful, or not, and why.
* **Status message** – Short description of the status code. Get the name of the status code of the response.
* **Header** – header of type HTTP Response
* **Content** – byte array that would contain the content of the response
* **Content as UTF8** – setter that would transform given string to byte array and assign it to the Content

Override the ToString() method so it can print the Response header accordingly to the HTTP standard

HTTP/1.0 {StatusCode} {Status Message}

{Header}

### Examples

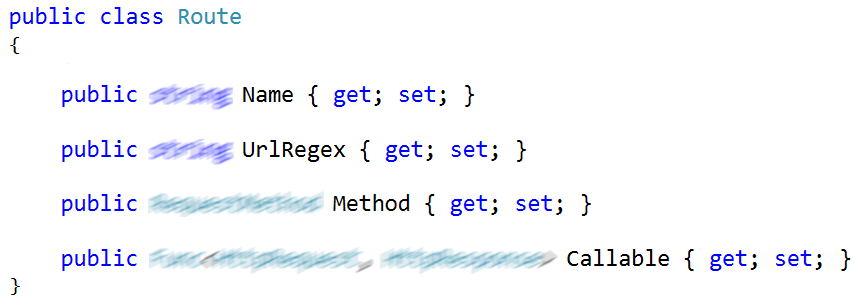
|  |  |
| --- | --- |
| **Response Example 1** | **Response Example 2** |
| HTTP/1.0 200 Ok  Content-type: text/html | HTTP/1.0 404 Page Not Found  Content-type: text/html |

Notice that we do not format the content of the response. We will implement later that functionality.

## Route

Create Route class in the models folder. That class should have only properties –

* **Name** – descriptive name for no other purpose than easier debugging
* **Url Regex** – regex pattern that would be used when trying to match URL address from request
* **Request Method**
* **Callable** - Function that would accept HttpRequest and return HttpResponse

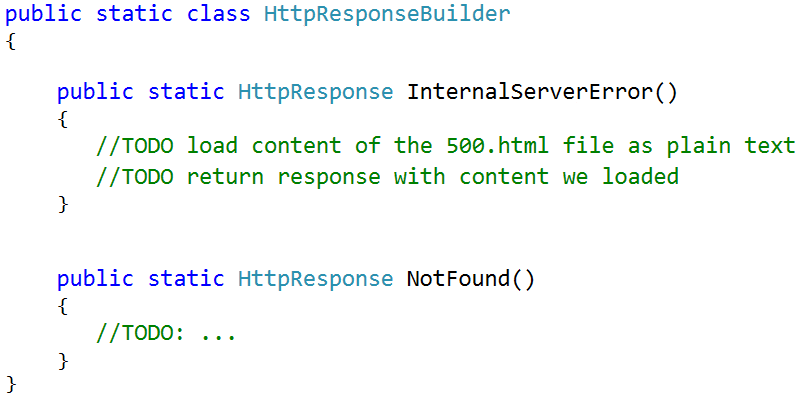


## Response Builder

Final step before assembling all the models we have made so far is to create a static class that would help to generate custom responses for 2 of the most common response statuses

* **404** – Resource Not Found
* **500** – Internal Sever Error

In the **Resources folder** create new folder called **Pages**. Inside it make 2 new files **404.html** and **500.html**. Design them as you find appropriate for you. Those pages will be displayed to the user in case anything goes wrong.



## Stream Utilities

We would write 1 helper class to help us when reading ad wring to a stream (such as NetworkStream).

Create class called StreamUtils with 2 static methods inside:

* ReadLine(Stream steam**) –** using the provided stream while should use the following algorithm:
  + read byte
  + convert it to char
  + if the char is ‘\n’ => break the loop
  + if the char is ‘\r\ => continue with the next iteration of the loop
  + if the char is ‘-1’ => put the thread to sleep for 1 millisecond and then continue with the next iteration of the loop
  + add the char to a string that keep all the read characters before that

after exiting the loop of reading bytes return the string that kept all the read characters

* WriteResponse(Stream stream, HttpResponse response) **–** that method would write on the stream first the response header then the response content

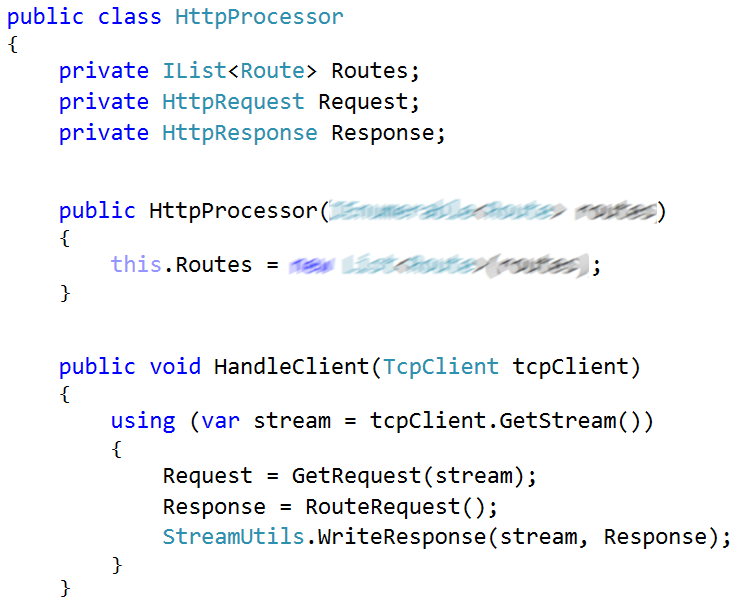


## HTTP Processor

Time to create the hearth of our server. That would be The HttpProcessor class. That class would be responsible for transforming HTTP request to HTTP response. It will have Request, Response and List of Routes as properties. When it is initialized it must be initialized with list of routes. Also, it should have 3 important methods

#### Handle Client

That method would receive as parameter TcpClient, then retrieves it’s NetworkStream uses it to convert the incoming bytes to string then to HttpRequest then generate Response to that request and finally write back the response on the stream.

****

#### Get Request

that method has to do several steps to create HTTP request

* **Read single request line –** the first line of the incoming request and split by ‘ ‘ (single space) to obtain the request method, URL, and the protocol version.

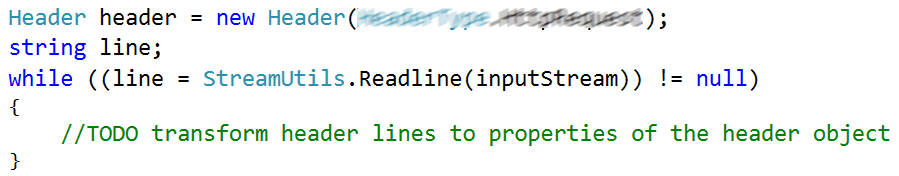
First line is in format: “{Method} {URL} {ProtocolVersion}”

For example, “GET /users/all HTTP/1.0”

If some of those elements is missing throw exception.

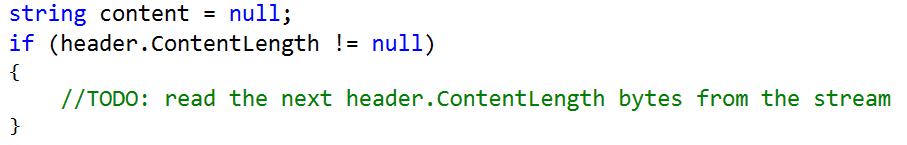
* **Read Headers** – we need to obtain information contained in the following lines until we reach empty line

Those lines will be in format “{Name}: {Value}”

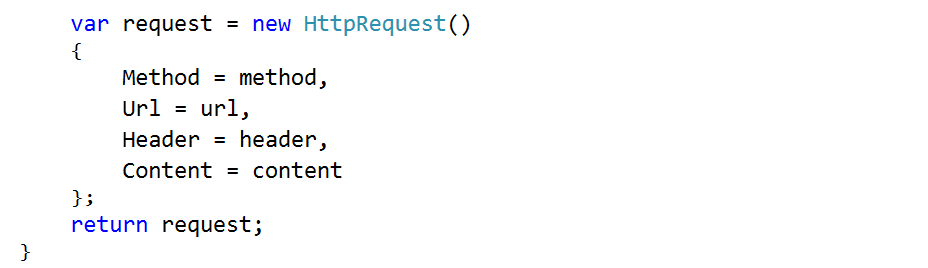
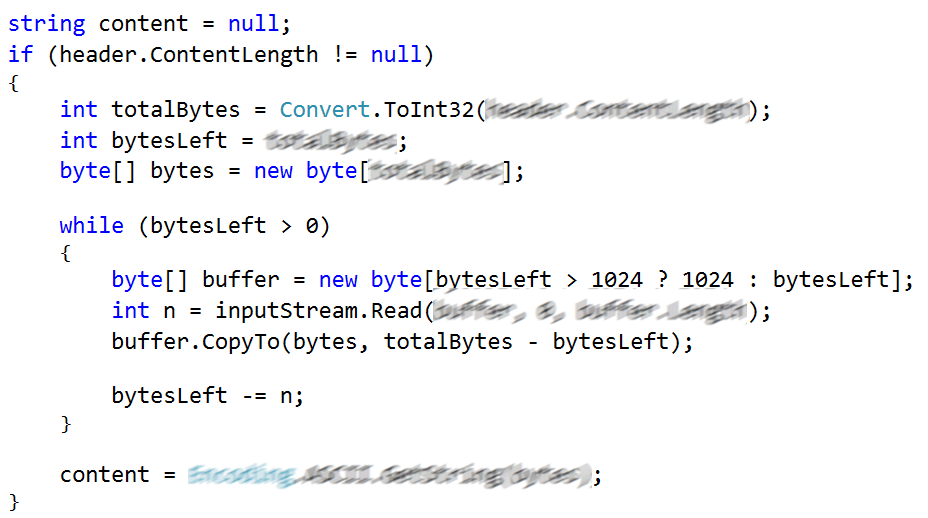
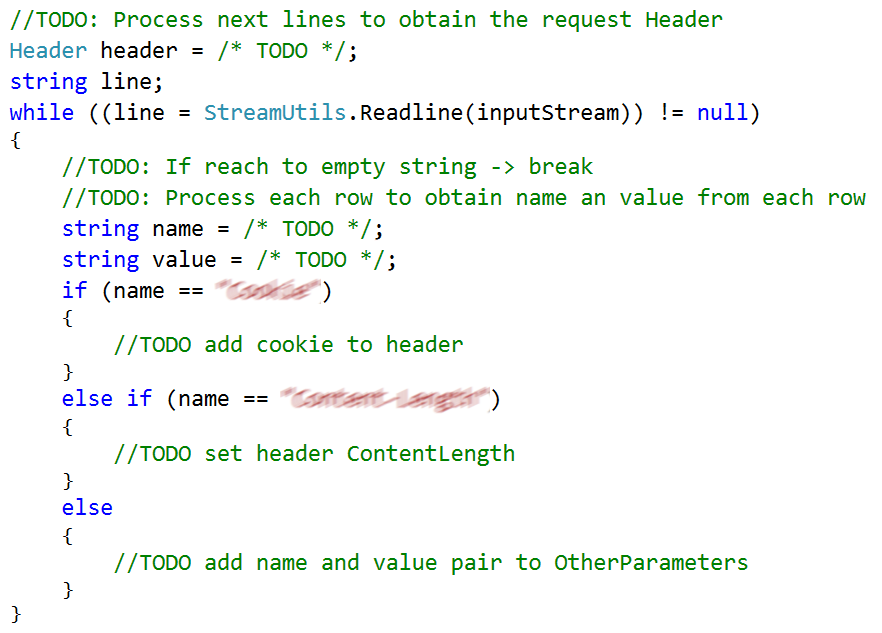
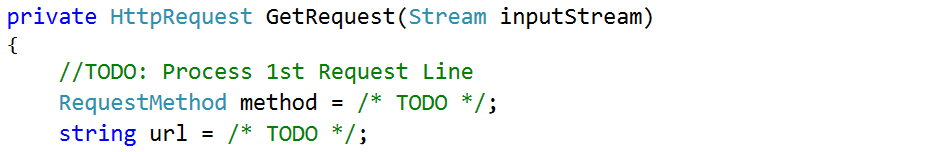


Split the row by “:” to obtain Name and Value. Then if the name is **Cookie** – from the Value create Cookie object(s) and **add** it(them) to the **cookie collection** in the header. If the name is **Content-Length** set the header’s ContentLength property. In all other cases add the name and value to the header’s OthersParameters dictionary.

* **Read Content** (if present) - When all header name-value pairs are parsed successfully we should check if the request contains Content-Length if it is contained that means the request has also content that must be read from the stream. (For example, when we receive request with POST method).



If you find it challenging this might help you:



#### Route Request

Here we should check whether the **URL from the request would match any of the predefined routes’ regex patterns**. There might be only **one or several routes** that would match the requested URL.

If there are **no routes matching** the URL from the request, we should **return Not Found Response**.

Of those routes who match the requested URL we should check whether some route has the **same request method** as the one in the request. If no route’s method matches the request method **return Method Not Allowed Response**

At this point **we should have the route that can handle the request** via its Callable property. So just execute it by passing a request to it.

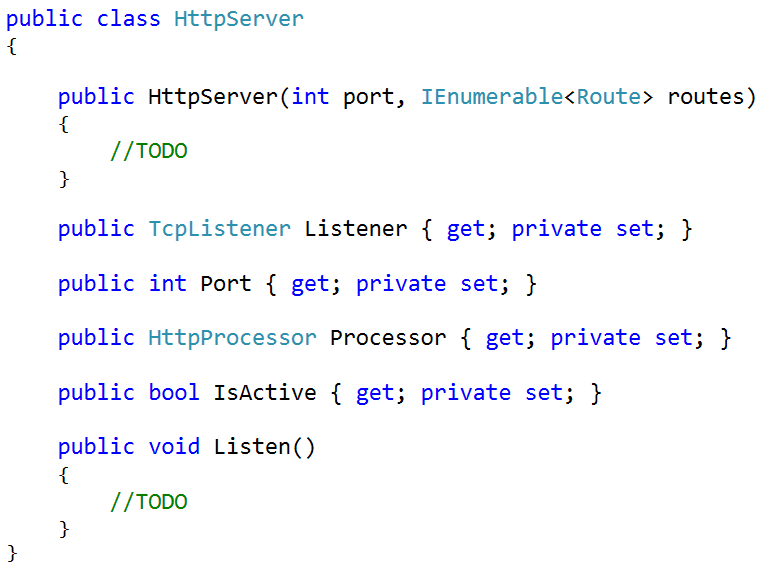
Surround the line where execute the Callable function of the request with try…catch block to catch **all Exceptions**. If exception is thrown anywhere after that **log the error** on the console or in a file and **return Internal Server Error Response.**

## HTTP Server

Time to create the core of our HTTP server – HttpServer class. The server has as properties

* **TCP Listener** – it will listen at given port for incoming requests
* **Port** – port at which the listener will listen (positive number in range [0 – 65535])
* **Is Active** – Boolean variable indicating whether the server is running
* **HTTP Processor** – hearth of our server. It will transform HTTP request to HTTP response.

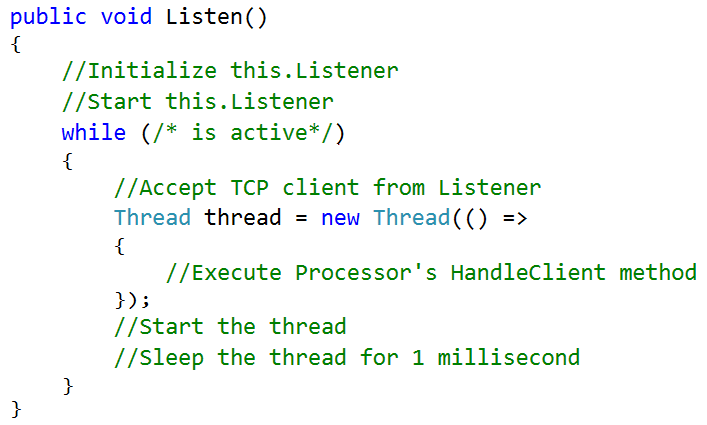
The HttpServer when it is initialized port number and collection of Routes must be passed to the constructor as parameters. In the constructor, also to IsActive property must be set to true. This class should have only 1 method Listen().



Check out the [**TcpListener.AcceptTcpClient()**](https://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener.accepttcpclient(v=vs.110).aspx) method. In the Listen() method we should create new TcpListener for any IP address with the port given when initializing the server. Then we should start that listener.

While the server is active we should

1. Accept the TCP Client from the listener.
2. Create new thread that would execute the HttpProcessor’s method HandleClient(client).
3. Start the thread with that method
4. Put the thread to sleep for 1 millisecond.



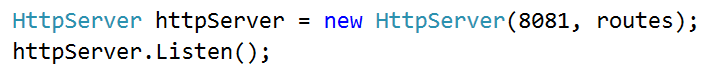
## First Run

If you made it so far it is time finally to run it. Create new Console Application project with name of your choice and in the main method create list of routes.

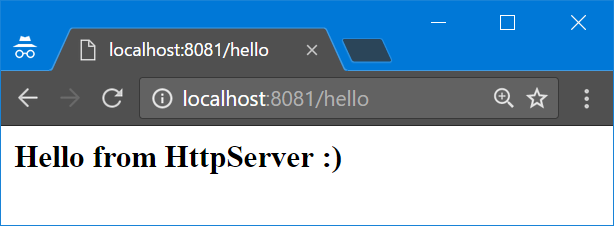
For example



Then use the following code to run your server



If we run the application now and in any browser, try to reach <http://localhost:8081/hello> we would get.



We could add as many routes we want to generate different pages.