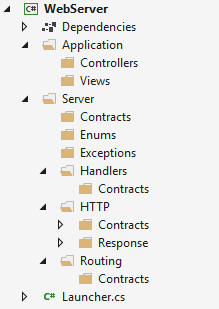
# Lab: Handmade HTTP Server

Problems for exercises and homework for the [“C# Web Development 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. Later in the course we will extend it further by adding sessions, cookies and a MVC framework.

## Create New Project

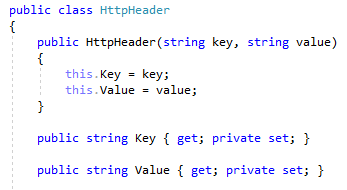
Create a new .NET Core console application and call it whatever you want. In it, create the following folders:



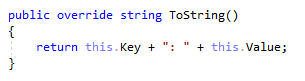
## Header and HeaderCollection

HTTP headers allow the client and the server to pass additional information with the request or the response. A request header consists of its case-insensitive name followed by a colon ':', then by its value (without line breaks).

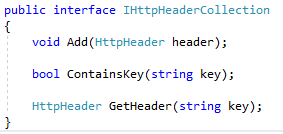
In the Server/HTTP folder create a class called HttpHeader that will be responsible for keeping all the information for each HTTP request or response. It should have properties for a key and a value (just like a dictionary entry).



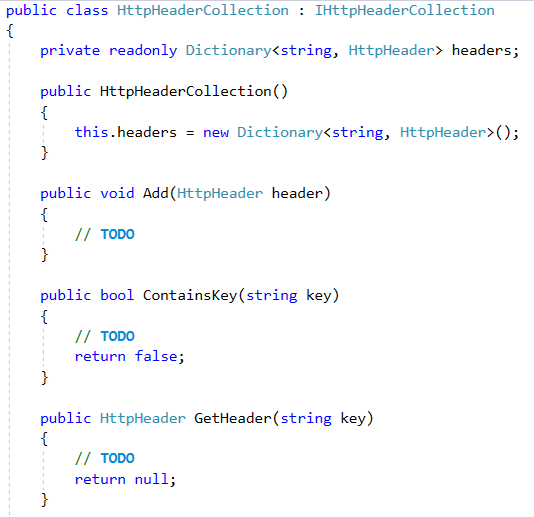
We also need to override the ToString() method of the class.



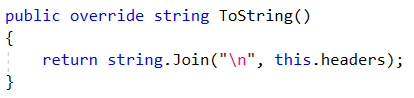
We also need a custom data structure to help us collect all the headers in one place. Create a new interface (contract) in the HTTP/Contracts folder and call it IHttpHeaderCollection.



Now create a HttpHeaderCollection class in the HTTP folder. Implement the interface you just wrote and the functionality of the methods.



We need to override the ToString() method here as well, but it won’t be anything too complicated – we need to print out all headers, each on a new line:

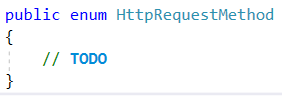


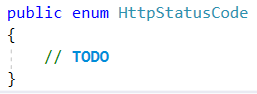
## Request Methods and Status Codes

Create 2 enumerations – **Request Method** and **Response Status Code**, both being in the Enums folder (obviously). For the sake of simplicity our HTTP server will only provide functionalities for **POST** and **GET** request methods. The status code is returned along with the HTTP response. Again, for the sake of simplicity, we will use a few codes:

* **200 – OK**
* **301 – Moved Permanently**
* **302 – Found**
* **303 – Moved Temporarily**
* **401 – Not Authorized**
* **404 – Not Found**
* **500 – Internal Server Error**

If you are curious, here is a [**list of all Response codes**](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)**.**

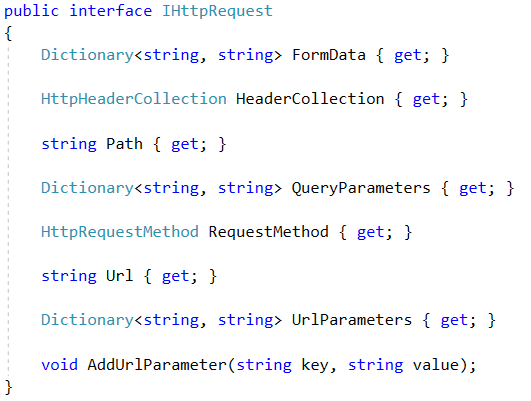




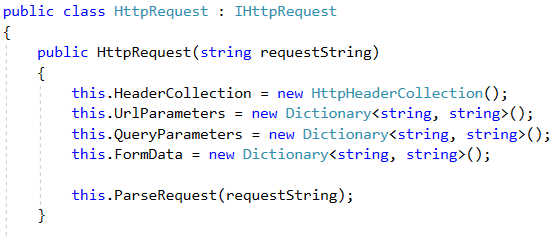
## HTTP Request

Create an interface called IHttpRequest in the HTTP/Contracts folder. An HTTP Request has:

* **Form Data –** data passed in by a POST-request form
* **Header Collection –** for collecting all headers
* **URL –** the URL the user enters
* **URL Parameters –** host, path, etc.
* **Path –** the part of the URL after the “/” sign
* **Query Parameters –** everything in the path past the “&” sign
* **Request Method –** GET, POST, etc.



Now create the HttpRequest class in the HTTP folder and implement the interface you just wrote. Using common sense, you should be able to code the AddUrlParemeter method by yourself. The constructor of this class will only accept a request string as a parameter. The parsing of the request string will be done internally:



The first part will be to put our request string in an array. A HTTP request has the following format:

{Method} {URL} HTTP/1.1

{Headers}

{Empty line}

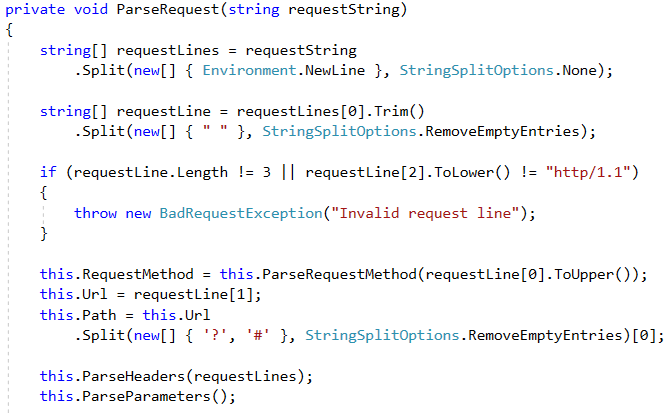
{Form Data}

*Note: Each header comes on a new line.*

### Examples

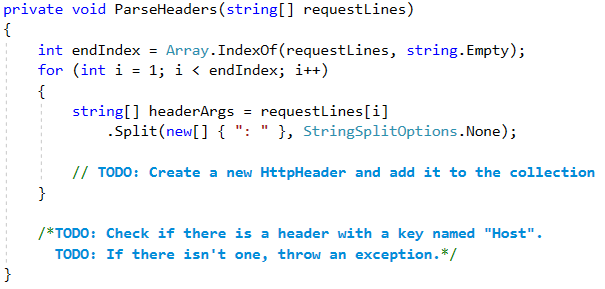
|  |  |
| --- | --- |
| **GET Request Example** | **POST Request Example** |
| GET /users/register HTTP/1.0  Host: localhost:8081  Accept-Language: en | POST /users/add HTTP/1.0  Host: localhost:8081  Accept-Languagage: fr  username=potter&password=123 |

We need to split the request line from the rest, since it contains the most important things – the request method, the URL and the protocol version. If the request contains an outdated/invalid protocol version, it is considered **bad**. We will use a custom exception for bad requests - BadRequestException. Create it in the Exceptions folder. After that, we need to parse the headers and any parameters that get passed.

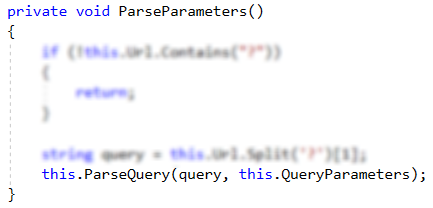


**Hint:** For the ParseRequestMethod method, find a way to parse a string to an enumeration. If any errors occur while parsing, this means that the input request method is invalid and the request is considered bad.

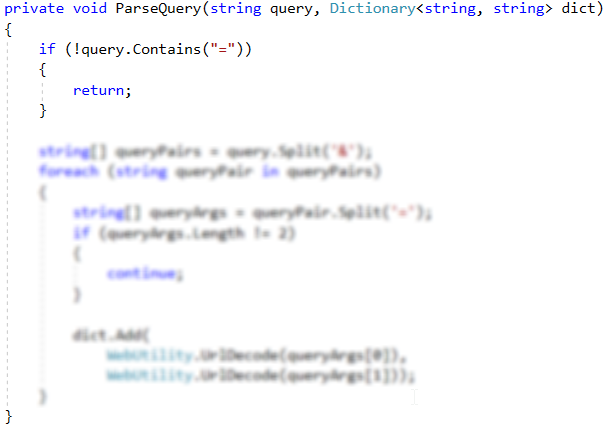
All headers end with a trailing empty string – we can get the index of that empty string and create a loop up to that index to go through all headers and parse them properly. If there isnt a header named “Host” the request is once again considered bad.



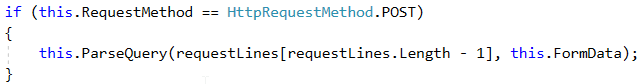
When it comes to the parameters, there won’t be anything too complicated – check if there are any parameters, and if there are, split the URL and parse the query itself:



The ParseQuery method will accept a dictionary as a parameter, in which we will add all the query parameters.

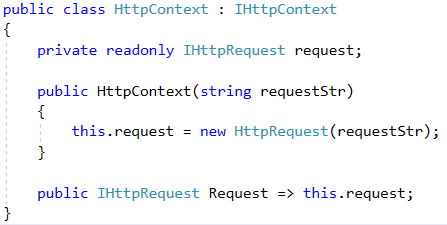


Finally, back in the ParseRequest method – we need to check if the request method is POST. If it is, we need to fetch the form data which comes with it. The form data is located at the last index of the request headers, right after the empty line.



## HTTP Context

We need to create a context for the requests to be sent through. In the HTTP/Contracts folder create an interface called IHttpContext. It will only contain a HttpRequest. Once you are done, create the respective class in the HTTP folder.



## HTTP Response

We have the request all done, but we also need a response from our server. In the HTTP/Contracts create an interface called IHttpResponse. A response will store the response itself in the form of a string. It will also have a method which adds a header to its header collection. Responses will also have a **view** – a HTML page displayed to the user. Create an interface called IView in the Server/Contracts folder. In it, have a single method called View() which returns a string.

Now, in the HTTP/Response folder create an abstract class called HttpResponse. We will have 2 types of responses – a redirect (e.g. when the user completes a register form and we want to send him to the login page) and a view (every page the user sees). Therefore, we will need 2 different constructors. In both cases, however, we need to instantiate a HeaderCollection.



*Note: You need to implement the AddHeader() method yourself.*

Finally, we need to construct the response itself. The standard for responses is:

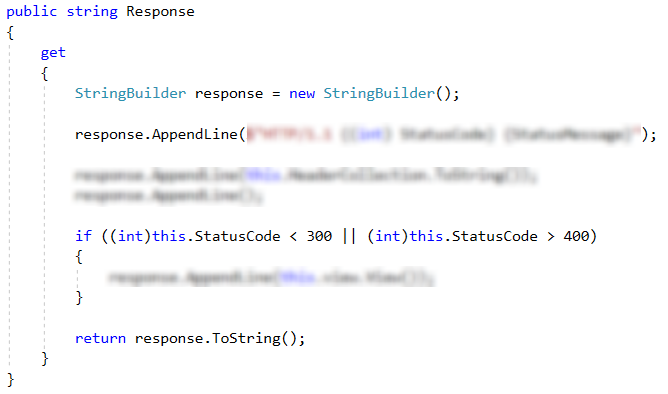
HTTP/1.1 {Status Code} {Status Message}

{Headers}

### Examples

|  |  |
| --- | --- |
| **Response Example 1** | **Response Example 2** |
| HTTP/1.1 200 Ok  Content-type: text/html | HTTP/1.1 404 Page Not Found  Content-type: text/html |

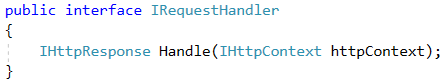
Our server will also hold the content displayed to the user in the response. Codes in the 300s (e.g. 302 – Found) are considered redirection status codes and, therefore, have no content, which means that there is no need to send a view.



Once we are done with this, we can create our two response types – the redirect and the view. Create 2 classes named accordingly – RedirectResponse and ViewResponse in the HTTP/Response folder. Inherit the abstract response and create a constructor in each of them, calling one of the base constructors (you should figure out which constructor you should use).

## RequestHandler

We need a way to handle the requests sent to our server. In the Handlers/Contracts folder create an interface called IRequestHandler. It will have a single method – Handle() which will accept a HTTP request and return a HTTP response.



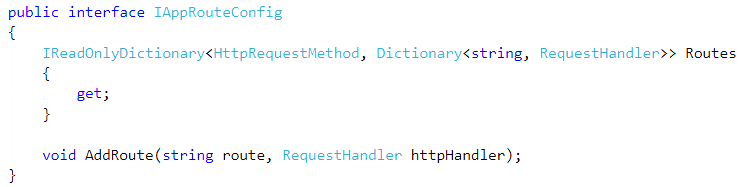
Now, in the Server/Hanlers create the abstract class – RequestHandler. Implement the interface you just wrote. It will have a single field – a **function** which will accept a request and return a response. In our Handle() method we need to invoke the function, add a header and return the generated response.



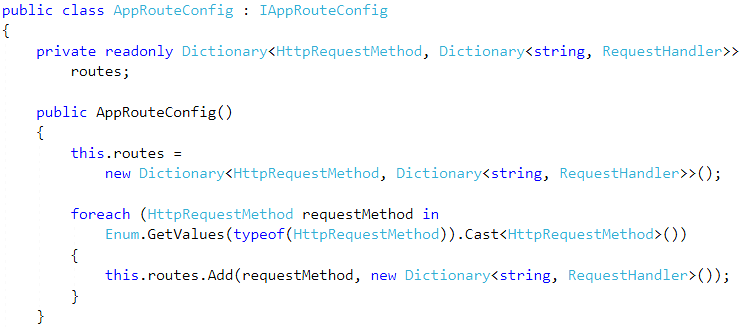
Now, in the same folder, create handlers for the two request methods our server has – GET and POST – in the Handlers folder. Inherit the abstract handler. Both of them will accept a function in the constructor and both of them will call the base constructor.

## Routing and Configuring Routes

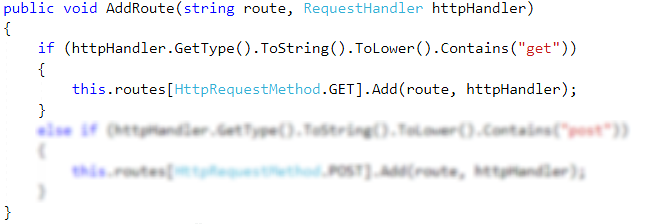
We have our requests and responses, but we need routes to and from which we can get said requests and repsonses. First we need to create a route configuration for the application itself. After that, we will create the configuration for the server itself, which will use the application configuration. Create a new interface called IAppRouteConfig in the Routing/Contracts folder. It will hold a dictionary for our routes and a way to add a route to it.



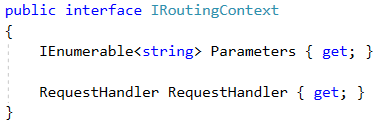
Now, in the routing folder add a class and call it AppRouteConfig. Implement the interface you wrote above. The constructor won’t accept any parameters, but it will instantiate the dictionary and we will add a new key-value pair for each of the request methods, which, for now, are just two.



As for the AddRoute() method, it will be very simple:

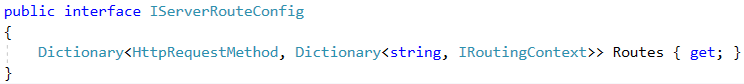


Next up, we need to have a context in which we will store the routes. In the Routing/Contracts create an interface called IRoutingContext. It should have a list of strings and a request handler in it.

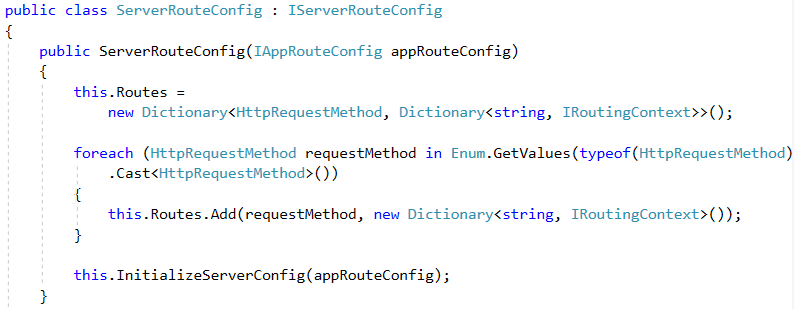


In the Routing folder create a class called RoutingContext. Implement the interface you wrote and encapsulate the list.

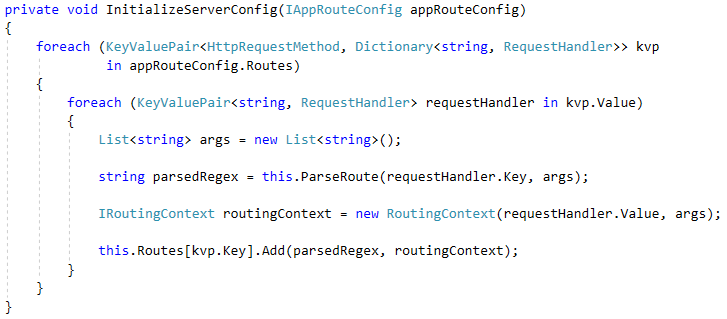
Finally, we need to create the configuration the server will use. In the Routing/Contracts folder create one last interface – IServerRouteConfig. The server configuration will have a dictionary, containing all the routes, having a request method as a key to another dictionary:



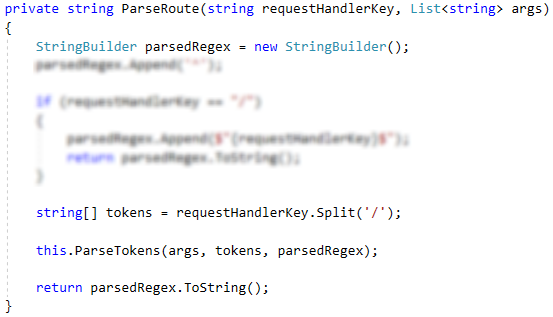
Now create your ServerRouteConfig class and implement the interface. The constructor is almost exactly the same as the application configuration with the only difference being that it will accept an actual AppConfig, which we will use to initialize this configuration internally:



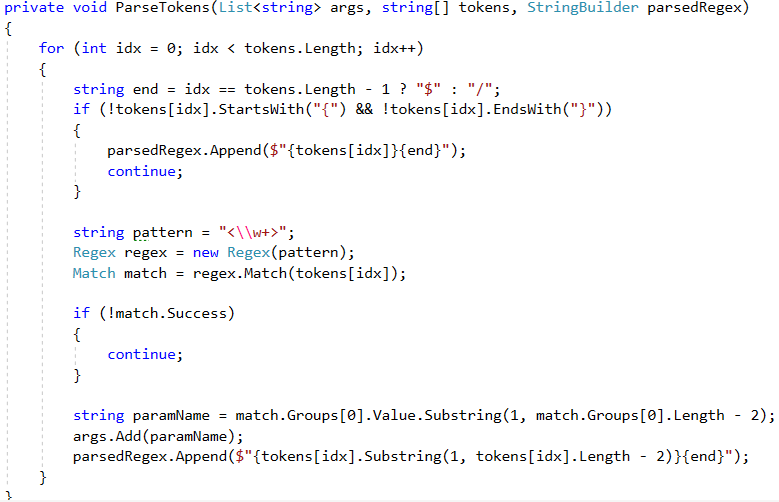
When it comes to initializing the configuration, we need to get all the routes from the app route configuration and parse them to a valid regular expression. If the route has any parameters in it, we need to add them to a list.



If the route is “/” we don’t need to do anything too special – just return the path, only that it is parsed as a regex (surrounded by a ^ and a $). If the route is longer, however, we need to split the individual tokens and parse them. In order to separate the parameters, when we set a route for the specific request, we will surround it with curly brackets, which we will make a condition for.

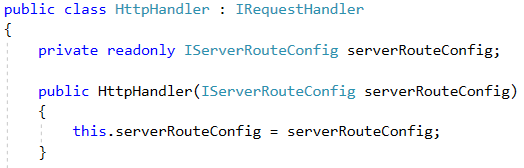


We splitted the individual tokens, but we need to bind them back together. If the current token is the last, we need to add a “$” to the end of it. Otherwise, add a “/”. If the token is not a parameter (not surrounded with curly brackets), we will just add it to the route regex, otherwise, we need to take the parameter name and the name only. For that, we will use another regex.

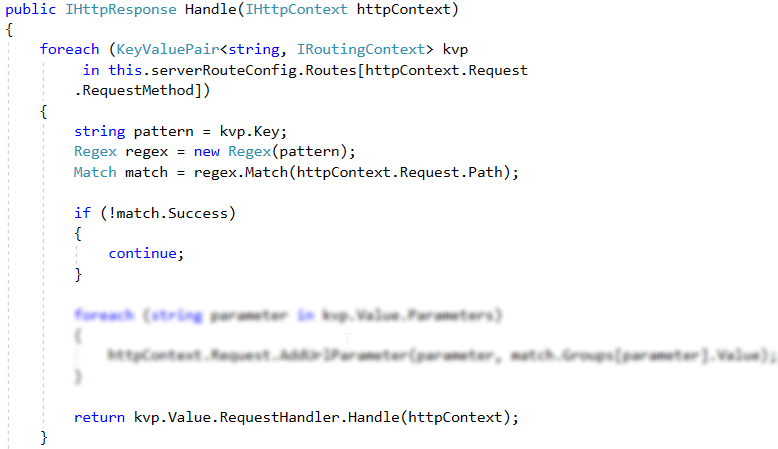


## HttpHandler

Going back to the Handlers folder, create a class called HttpHandler. Implement the IRequestHandler interface. The main difference between the other handlers and this handler is that this one will use and accept as a constructor parameter a server configuration.

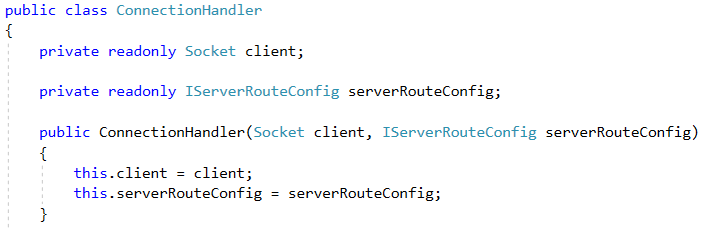


When it comes to handling, we need to do the following things: Look through all the pre-configured routes which have the same request method as the request itself and if there is one, we must add any parameters that come with specific routing context before returning a response to the user:

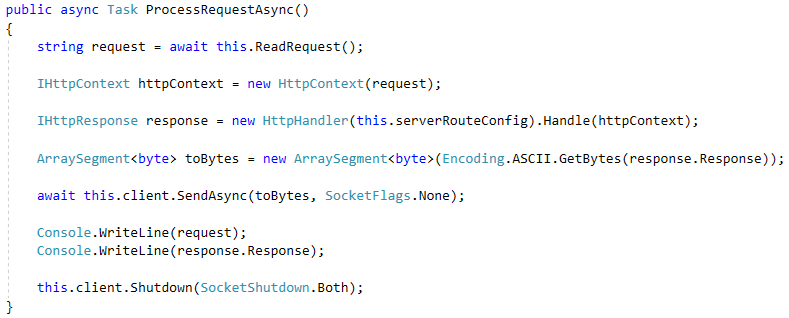


## ConnectionHandler

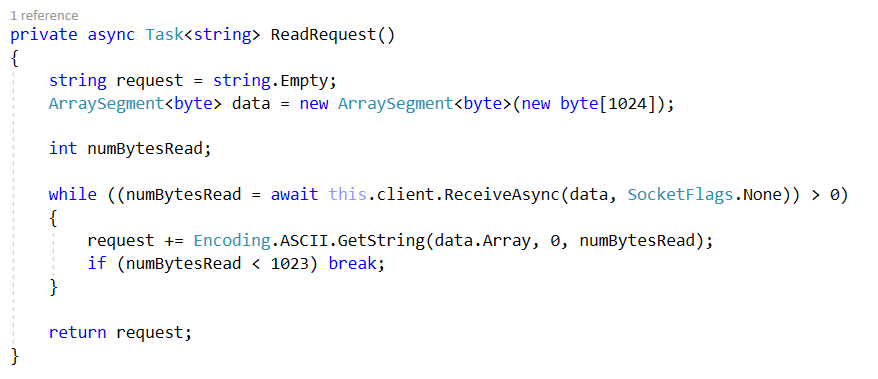
The ConnectionHandler class will be the reader and processor for our requests. Create it in the Server folder. It will use a [socket](https://msdn.microsoft.com/en-us/library/system.net.sockets.socket(v=vs.110).aspx) and the server route configuration.



Since our main task is to make our server asynchronous and have multiple concurrent users, we are going to use [tasks](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task(v=vs.110).aspx). The process will have to read the request itself, then create a context with that request and handle it. Since sockets work with bytes, we need to convert the returned response to bytes, which we will keep in an [array segment](https://msdn.microsoft.com/en-us/library/1hsbd92d(v=vs.110).aspx). Once it is coverted, we can send it via our socket. In the end, always shut down the socket. This ensures that all data is sent and received on the connected socket before it is closed.



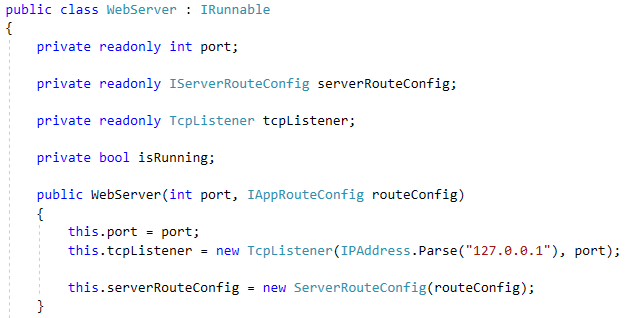
When it comes to reading the request – we will receive it in the form of bytes, too. We will use an array segment once again. This structure is especially useful when performing concurrent operations. Here we will use the array segment as a buffer to read all bytes of our request:



After each time we fill the buffer we should check weather it is full or not and break the loop if we have read all the bytes. Otherwise our program will wait forever to receive data that will never come.

## WebServer

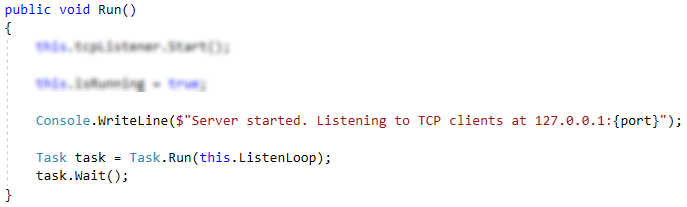
We wrote, requests, responses, etc. It’s finally time to code the server. Create a class called WebServer in the Server folder. It will implement an interface called IRunnable (which has a single void methord – Run(); created in the Server/Contracts folder). Our server will use a [[TcpListener](https://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener(v=vs.110).aspx)](https://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener(v=vs.110).aspx). It will also have fields for the port the server will run on, the server route configuration and a Boolean variablie, which will check whether the server is running. The constructor will accept the port and an application route configuration. A TcpListener accepts an IP address and a port to run on. In our case, the IP address will be localhost (127.0.0.1).



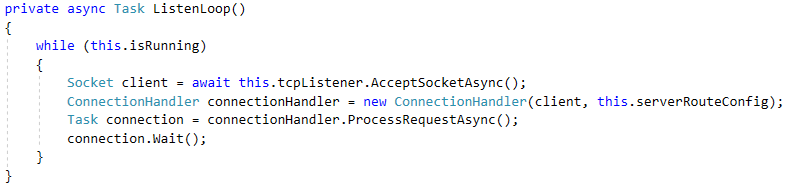
The Run() method will:

* Start the listener;
* Set the server state to “running”;
* Create a loop, which will listen for requests.

*Note: the loop will be asynchronous, since we want multiple users on it at any given time.*

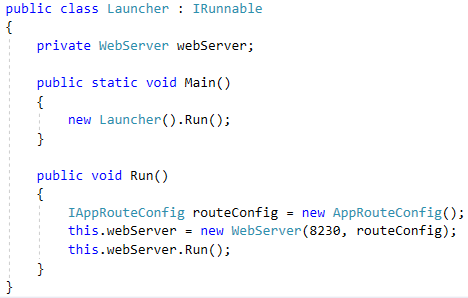


We need to accept a socket from the listener, create a ConnectionHandler using that socket and process the request, using said handler.



## Testing Our Server

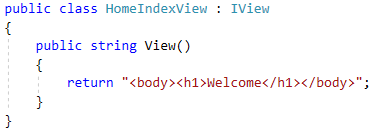
We can finally proceed to testing the server itself. You should have a class called Launcher outside of any folders. This will be our starting point for the entire server. Implement the IRunnable interface for it. The launcher will have a single field – the server itself. The main method will create a new instance of the launcher and run it.



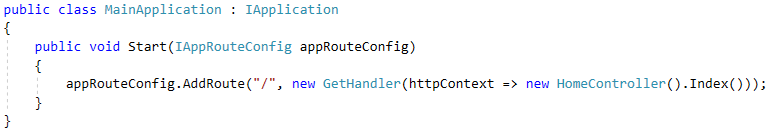
Here, we see something is off: we never actually set any routes for the server. That’s where the Application folder comes to play. In it, we will use something which we will implement more in-depth later on – a MVC (Model-View-Controller) framework. In the Application/Controllers folder create a class and call it HomeController. In it, have a single method – Index(), which will return a HTTP view response.

The view will be placed in the Application/Views folder. We will call it HomeIndexView(). The view should implement the IView interface and in the implemented interface we will hold the HTML that will be returned in the form of a string.

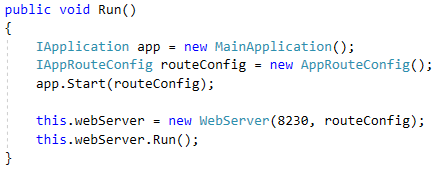




Now, we need to reference a route to that controller. In the application folder, create a class called MainApplication. It will implement an interface called IApplication (create it in the Server/Contracts) folder, which has a single method – Start(), which accepts an application route configuration, in which we will place all the routes for the server.

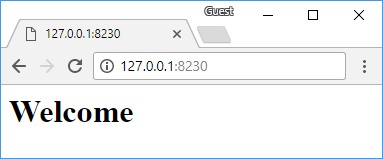


We need to start the application inside the launcher:

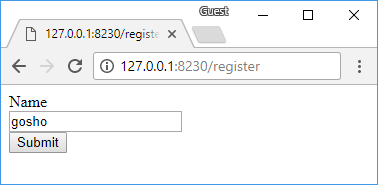


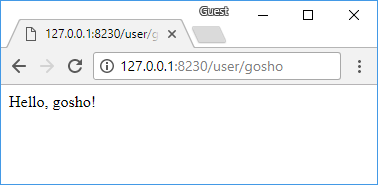
### Hello World!

Now, if we run the project, enter our browser and type localhost:8230 or 127.0.0.1:8230 in the location bar we should be greeted by:

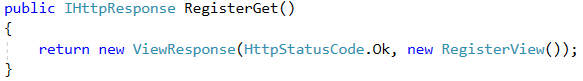


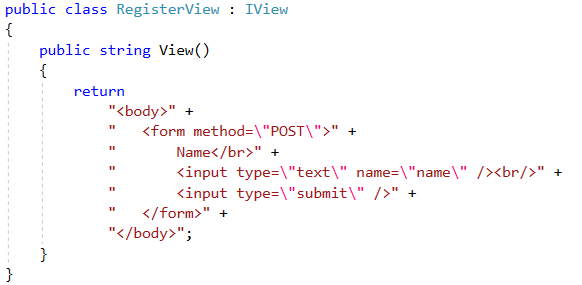
### Hello, {name}!



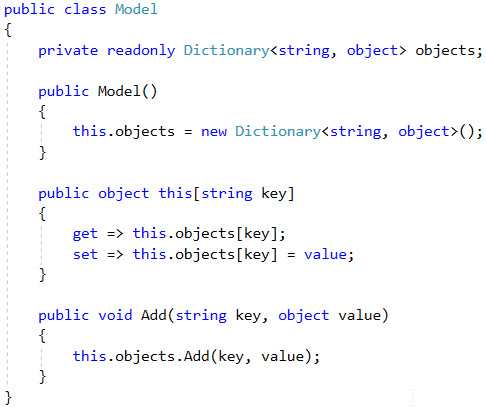


We will create very simple register and user details pages. Create a class called UserController in the Application/Controllers folder. We first need to create the logic behind the register. Create a method called RegisterGet() – it will return a view response and the view itself will contain the form for the user to fill.

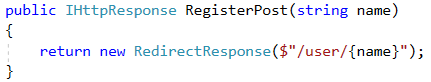




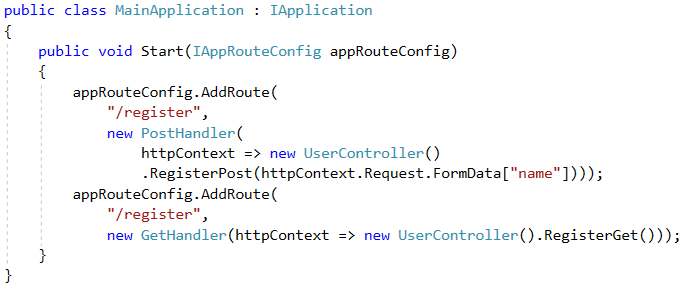
Once the form is submitted, it will send a POST request, holding some data. We need to redirect the user to the details page, while holding the sent form data. We need a **model** to hold it in. Create a class called Model in the Server folder. It will contain a dictionary, holding the names of the parameters along with their values.



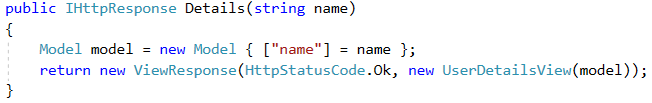
Now, back in the user controller, create a method called RegisterPost(). It will have a **redirect** response to the details page.

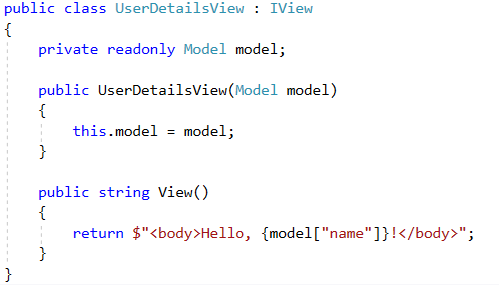


Now we need to set the routes for these controllers:

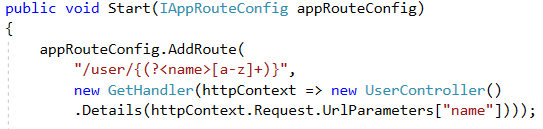


Finally, we need to make the details page itself. Once again, back to the controller, add a method and call it Details() – it will accept the name as a string, after which it will create a model, containing the name and return a view, using that model.





At last, we need to set a route for that page. Since the name is a parameter, we need to surround it with curly brackets:



Now, if you run your server, go to “127.0.0.1:8230/register”, fill the form and press “Submit”, you should be greeted by name!

Congratulations! You created your own HTTP server. It is still very basic, but we will continue on expanding it later on.