# SIS – SoftUni Information Services

SIS is a combination of a Web Server and a MVC Framework. Ultimately it is designed to mimic Microsoft’s IIS and ASP.NET Core. Following several Lab documents you will build all components of the SIS.

# SIS: Handmade HTTP Server

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

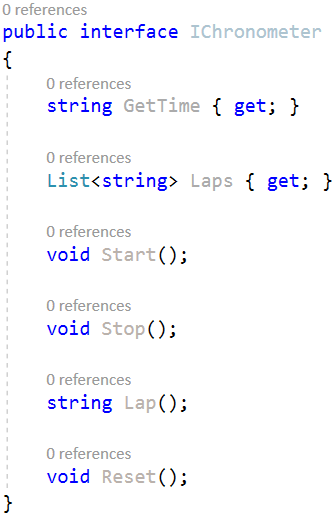
Following to the end this document will help you to create your own very simple HTTP Server. Later in the course we will extend it by adding sessions, cookies etc. We will eventually build a MVC Framework, with which we can build MVC Web Application which will be hosted on the Handmade HTTP Server.

But before that, let’s have a little practice on asynchronous tasks.

## Chronometer

The Chronometer is one of the easiest examples of an asynchronous processes. Let’s implement a simple Chronometer.

Create an interface IChronometer like this:



... and implement a class **Chronometer**, that implements it.

Implement a program which provides a Chronometer functionality, that responds to several commands from the user input:

start – starts counting time in milliseconds, seconds and minutes.

stop – stops the process of counting time, but the counted time remains.

lap – creates a lap at the current time.

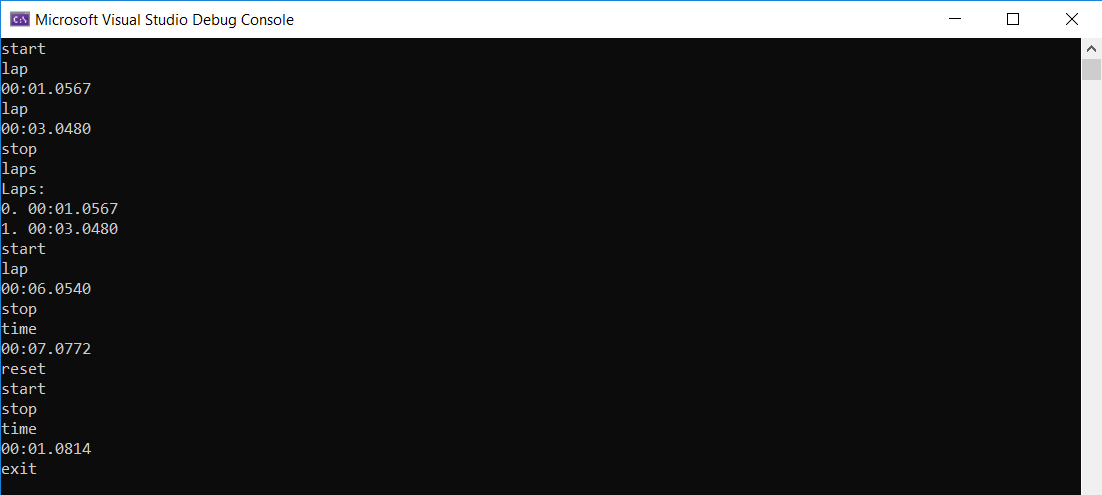
laps – returns all of the currently recorded laps.

time – returns the currently recorded time.

reset – stops the Chronometer, resets the currently recorded time and deletes all of the currently recoded laps.

exit – stops and exits the program.

Here is an example screenshot of the functionality:



The time is outputted in the following format: "{minutes}:{seconds}:{milliseconds}", each of them should be **padded** with **zeros**.

Upon **making** a **lap** you should print the **time** at which it was made.

Requesting **all laps** should print them in the following format:

Laps:  
0. {lap1}  
1. {lap2}  
...

In case there are no laps, you should print "Laps: no laps".

## Parallel MergeSort

MergeSort is one of the fastest and most efficient sorting algorithms. It disolves a collection into 2 halves and then disolves each half into 2 halves, and so on it repeats the process, until the halves are made up of 1 element. Then it merges them back in a sorted way, eventually merging the whole collection back. It is a Divide and Conquer recursive algorithm. Due to its recursiveness, which essentially represents running several instances of the same process, it can be distributed into several threads – making it a parallel merge sort. A very fast algorithm in the hands of a casual processor.

* Implement a simple merge sort, you can find the algorithm’s description and pseudocode anywhere in the net, but here’s a [link](https://softwareengineering.stackexchange.com/questions/324593/recursion-in-merge-sort-algorithm-how-is-it-obvious-to-use-this-type-of-recursi/324596#324596).
* Here’s a [link](https://www.random.org/integer-sets/), for generating large sets of random numbers, in a formatted form, so that you can have test data. Test your algorithm with at least **10000** elements.

Implement a Parallel MergeSort algorithm, but be careful, put a limit to the threads distribution. You don’t want to put int.MaxValue threads into work, or your computer will hate you.

## Asynchronous SIS WebServer

### SIS.WebServer Project Architecture

The **WebServer Project** holds the main classes that **establish** the **connection** over **TCP Link**. These classes are used the ones from the **HTTP Project**. The Project expose several classes, which should be used from the outside, in order to **implement** an **application**.

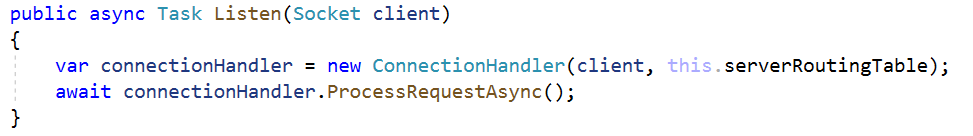
**Server Class**

The Run() method should be used to **start** the **listening process**. Тhe listening process should be **asynchronous** to ensure **concurrent client functionality**.



We also have a little message notifying us that nothing has exploded brutally in the process.

The Listen() method is the main processing of the **client connection**:

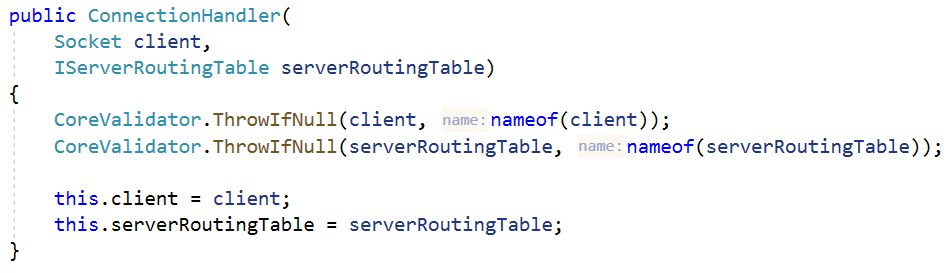


**ConnectionHandler Class**

The ConnectionHandler class is the **client** **connection processor**. It receives the **connection**, **extracts** the **request** **string data** from it, **processes** it **using** the **routing table**, and then **sends back** the **Response** in a byte format, throughout the **TCP link**.



The constructor should just **initialize** the **socket** (the **wrapper object** for a **client connection**) and the **routing table**.



The ProcessRequestAsync() method is an **asynchronous** method which contains the main functionality of the class. It uses the other methods to **read** the **request**, **handle** it, **generate** a **response**, **send** it to the **client**, and finally, **close** the **connection**.



The ReadRequest() method is an **asynchronous** method which reads the **byte data** from the **client connection**, **extracts** the **request string data** from it, and then **maps** it to a HttpRequest object.



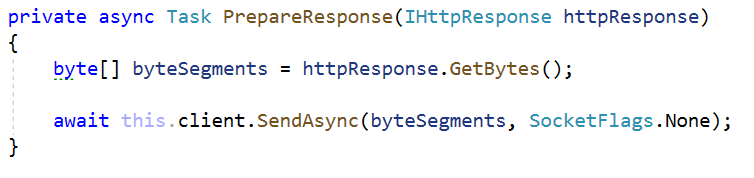
As you can see the **Requests** are quite limited to **1024 bytes**. This is intentional.

The HandleRequest() method **checks** if the **routing table** has a **handler** for the **given Request**, using the **Request’s** **Method** and **Path**.

* If there is **no such handler** a “Not Found” **Response** is returned.
* If there is a **handler**, its **function** is **invoked**, and its resulting **Response** – returned.



The PrepareResponse() method **extracts** the **byte data** from the **Response**, and **sends** it to the **client**.

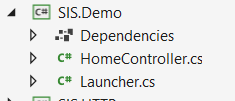


And with that we are finished with the ConnectionHandler and the **WebServer Project** as a whole. Now, before we embark on a journey to implement applications with our **SIS**. Let’s first check a very simple **Hello World! Demo app**.

## Hello, World!

Implement a third project called SIS.Demo. Reference both the SIS.HTTP and SIS.WebServer projects to it.

Create the following classes:



### HomeController

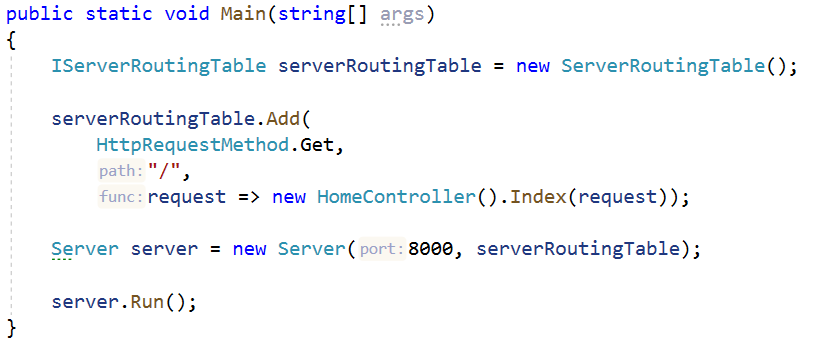
The HomeController class should hold a single method – Index() which looks like this:



### Launcher

The Launcher class should hold the Main method, which instantiates a Server and **configures** it to **handle** **requests** using the ServerRoutingTable.

**Configure** only the “/” route with a **lambda function** which **invokes** the HomeController.Index method.



Now run the SIS.Demo project, and you should see this, if everything up until now was done correctly:



Open your browser, then go to localhost:8000. And you should see this.



**Congratulations**! You have completed your async **Hello World app** with the SIS!