# Exercises: Asynchronous Programming

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.

## 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:

Graphical user interface, application

Description automatically generated

Shape, rectangle

Description automatically generated... 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".

### Hints

Let's start implementing our **asynchronous chronometer**. First, we need to create the Chronometer **class**, which implements the IChronometer **interface**:



Use the Stopwatch **C#** **class**, which provides a **set of methods and properties** that you can use to accurately measure elapsed **time**. Create a **field** for the stopwatch. Also, create a **collection for the laps**. Initialize the **fields** in the **constructor** like this:

Graphical user interface, text, application

Description automatically generated

We have the GetTime **property**, which should return the **currently recorded time** since the start of the chronometer counter. Use the Elapsed **property** of the Stopwatch **class** to **get the total elapsed time**. This property returns the time as a TimeSpan, so you should convert it to string in the **correct format**. Do it like this:



The other property we have is the Laps **property**. It should just **return the current laps collection**:



Next, we should **implement** the Start() and Stop() **methods** of the Chronometer **class**. The Stopwatch **class** has its own **methods for starting and stopping** – use them as shown below:

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

Description automatically generated

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

Description automatically generated

The Lap() **method** returns the **current elapsed time** as a string and **adds it to a collection of laps**. It uses the GetTime **property**:

Text

Description automatically generated

Finally, we have the Reset() **method**, which should **invoke the** Reset() **method** of the Stopwatch **class** and **clear the laps collection**. Do it like this:

Text

Description automatically generated

As we already have the Chronometer **class** let's use it and invoke its methods depending on **commands** from the console. **Instantiate the chronometer** in the Main() **method** of a **class** called Startup:

Graphical user interface, text

Description automatically generated

Then, we will **read a command** from the console, until the "exit" **command**.

Logo

Description automatically generated with low confidence

In the while **loop**, work with the chronometer depending on the **read command**. Don't forget that the Start() **command** of the Chronometer **class** should be **run as a task to be asynchronous**. After the while **loop** you should **stop the chronometer**. Complete the Startup **class** like this:

Text

Description automatically generated with medium confidenceGraphical user interface, text, application

Description automatically generated

## Make the Web Server Run Asynchronously

Now we go back to our **server** in the "BasicWebServer" **project** to **make it run asynchronously**, as this improves overall application **performance** and **responsiveness**. To do this, we should **replace the synchronous methods with** **asynchronous**, where possible, and **await** them.

### Step 1: Modify the HttpServer Class

Go to the HttpServer **class** and **make it run concurrently**. Start with the ReadRequest() **method**. In it, we read bytes from the network stream, which should be made **asynchronously** so that bigger requests **don't freeze the program**. Use the ReadAsync(byte[] buffer, int offset, int count) **method** and the await **keyword**:

A picture containing logo

Description automatically generatedGraphical user interface, text

Description automatically generated with medium confidence

When we add the keyword await to await the task, we also need to make the method async and **make it return a** Task. Change the ReadRequest(NetworkStream networkStream):

A picture containing logo

Description automatically generated

Then, do the same with the WriteResponse(NetworkStream networkStream, Response response) method, which should use an **asynchronous method** for writing to the network stream. Change the method like this:

A picture containing logo

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Next, go to the Start() **method** of the HttpServer **class**. Here, our aim is to **accept and process multiple requests** from the browser in **parallel**, otherwise each request should wait for the others to be completed. To do this, use the AcceptTcpClientAsync() **method** of the TcpListener **class** to **connect to the network**:

A picture containing logo

Description automatically generatedGraphical user interface, text

Description automatically generated

Then, we need to **await the methods**, which we made **asynchronous** earlier:

Graphical user interface, text, application, email

Description automatically generated

Finally, make the whole Start() **method** **asynchronous** like this:

A picture containing logo

Description automatically generated

In addition, we can separate the **network connection** initialization and using in the Startup() method to a separate **task** to **run asynchronously**. In order not to **await the task**, assign it a **variable** like this:

A picture containing logo

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

### Step 2: Modify the Startup Class

Now let's modify the Main() **method** of the Startup **class** in the "BasicWebServer.Demo" **project** to work **asynchronously** with the **HTTP server**. Do it as shown below:

Graphical user interface, text

Description automatically generated

## Download Web Content as a File

Our task now is to **asynchronously get a part of the HTML content** from **given** **sites** and **add it to a** .txt **file**. In our "BasicWebServer.Demo" **app** we will **click on a button** and **download** **the created file**. The workflow will be as shown below:

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

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

Description automatically generatedGraphical user interface, text, application

Description automatically generated

### Step 1: Create File Response

To start with, we will need to **create an HTTP response**, which **returns a file**, and the **file is downloaded**. To do this, we will need to **add the** "Content-Disposition" **header** to the **response headers**, which indicates whether the browser should display or download a given file.

Let's first add a **constant** to the Header **class** for this **header's name**:

Text

Description automatically generated

Now we should **create a class for the file response**. The TextFileResponse will return **file content as a plain text** and will **inherit** the Response **class:**



First, we should also have a **property** for the **file name**, which should be accepted through the **constructor**. The constructor should also **initialize** the FileName **property** and **add** the "Content-Type" **header**. Do it like this:

Graphical user interface, text, application

Description automatically generated

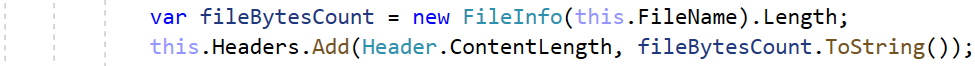
Then, the ToString() method of the class should **set the correct headers** and the **body**,if a **file with the given** **name** exists. Check this, using the File **class** and **set the file content as a response body** asynchronously:

Text

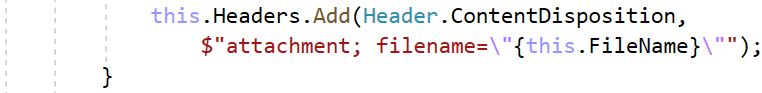
Description automatically generated

We will need to add the "Content-Length" and "Content-Disposition" **headers** to the **response**.

Then, use the FileInfo **class** to **get the length** of the **file** with a given name in **bytes** and **add it as a content length header value**:



Add the last **header** for the **content disposition**, as well. It should be of type "attachment" and **contain the file name**:



At the end, return the **base** ToString() **method**:

A picture containing text

Description automatically generated

### Step 2: Modify the Startup Class

As we have created the TextFileResponse **class** let's use it. First, go to the Startup **class** of the "BasicWebServer.Demo" **project** and **create a simple form**, which should only have a [Download] **button** to **trigger the action on** **the** "/Content" **page** with **method** "POST":

Text

Description automatically generated

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

|  |
| --- |
| private const string DownloadForm = @"<form action='/Content' method='POST'>  <input type='submit' value ='Download Sites Content' />  </form>"; |

This **form should be displayed** on the "/Content" **page** through a "GET" **request**. Then, when the **button is clicked** it should **send a** "POST" **request** to "/Content" and a TextFileResponse should be returned (and the file should be **downloaded**). Modify the Main() **method** as shown below to **create the needed mappings**:

Text

Description automatically generated

As you can see, we **set the file name** from a **class constant**. It is the following:

Text

Description automatically generated

As we have created the "POST" request mapping to return a TextFileResponse with a **file name**, we should **fill in the** **file with data** before the **response is returned**. To do this, we will create the DownloadSitesAsTextFile(…) method in the Startup **class** and **invoke it** before the **mappings** are created. With this method, we will **get the** **HTML content** of **several sites** and **write it to a text file** with a given name.

Before we write the above method, let's write another one – the DownloadWebSiteContent(string url) **method**, which should get the **first 2000 symbols** of the **HTML content** of a **site** on a **given URL**.

A picture containing text

Description automatically generated

To **get the content**, we should **send a** "GET" **request** to the site and **read its content**. For this reason, we will use the HttpClient **class**, which sends **HTTP requests** and receives **HTTP responses** from a resource, identified by a **URL**. The class provides us with the GetAsync(string requestUri) and the ReadAsStringAsync() **methods**.

Use the HttpClient **class** with its **methods** as shown below. At the end, **return only part of the HTML content**, so that the result **file is not too big**:

Graphical user interface, text, application

Description automatically generated

Note that **all** async **methods should be awaited** and that's why the DownloadWebSiteContent(string url) should return Task<string>.

The next method we will write is the DownloadSitesAsTextFile() **method**, which **downloads the content** of given sites **asynchronously**. Then, the whole **HTML content will be joined** and **written** to a single **file**.

First, the method should **accept a file name** and a **string array with URLs** of sites to be downloaded:

A picture containing text

Description automatically generated

Then, create a **collection of type** Task<string>, which holds the **tasks** for getting the **HTML content** from the sites, and **fill in the collection**:

Graphical user interface, text, application

Description automatically generated

**Wait** for all tasks to be **executed together** (in parallel) and **get the result** like this:



Now **join all the content** from the responses in a way you want and **get the result**. You can do it as shown below:

Text

Description automatically generated

Finally, use the File **class** to **write the HTML content** of the sites to a **file** with a given name **asynchronously**:



We already have the methods for **downloading** and **saving the HTML content** of **sites** to a **file**. Use them in the Main() method **before the mappings** like this:

Text

Description automatically generated

As you can see, we have changed the method **structure** a little bit, so that we can **invoke the method** we want. Also, we have **added** the <https://judge.softuni.org/> and <https://softuni.org/> **sites** as **URLs** and their **HTML** **content** will be added to the "content.txt" **file**.

### Step 3: Try File Downloading

**Run** the application. Note that the Main() **method** of the Startup **class is the first executed method of a program**, so the **file** is created **before the server starts**. To look at it, open File Explorer  "BasicWebServer.Demo"  "bin"  "Debug"  "net5.0" and you should **see the HTML content file**:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Then, **open the browser** and go to "/Content". When you **click** on the [Download] **button**, the same **file** should be **automatically downloaded**:

Graphical user interface, application

Description automatically generated