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.

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.

1. Solution Architecture

First, we will create the initial Solution Architecture. Create a **Solution** called "**SIS**" and create **2 Class Library projects** in it:

- **SIS.HTTP**
- **SIS.WebServer**

2. SIS.HTTP Project Architecture

The HTTP Project will hold all of the models (and their interfaces) which will be used to implement the HTTP Communication over the TCP Link between the Client and our Server. Naturally, we can work with plain strings and byte arrays, but it will be much more comfortable if everything has its own class and its own place in the code.

Create the following namespace architecture in the project:

As you can see the folder architecture is quite segregated. Now let's start filling the folders with our classes.

Common Names

The **Common** namespace will hold classes which are commonly used in the whole project. We will have two classes - **GlobalConstants** and **CoreValidator**.

GlobalConstants

Create a static class, called **GlobalConstants**, which we will use for our shared constants:

```
public static class GlobalConstants
{
    public const string HttpOneProtocolFragment = "HTTP/1.1";

    public const string HostHeaderKey = "Host";

    public const string HttpNewLine = "\r\n";
}
```

These are the only constants we will need for now.

CoreValidator

Create class **`CoreValidator`**, which will hold two methods for checking for null or empty values:

```
public class CoreValidator
{
    public static void ThrowIfNull(object obj, string name)
    {
        if (obj == null)
        {
            throw new ArgumentNullException(name);
        }
    }

    public static void ThrowIfNullOrEmpty(string text, string name)
    {
        if (string.IsNullOrEmpty(text))
        {
            throw new ArgumentException(message: $"{name} cannot be null or empty.", name);
        }
    }
}
```

Enums Namespace

The **Enums** namespace will hold our **enumerations**. There are **2 enumerations** we will need for the current implementation of the server – **`HttpRequestMethod`** and **`HttpResponseStatusCode`**.

HttpRequestMethod

Create an enumeration, called **`HttpRequestMethod`**, which will be used to **define** the **method** of the **request** our Server is receiving.

```
public enum HttpRequestMethod
{
    Get,
    Post,
    Put,
    Delete
}
```

Our Server will support only **GET**, **POST**, **PUT** and **DELETE** requests. There is no need for it to process more complex requests for now.

HttpResponseStatusCode

Create an enumeration, called **`HttpResponseStatusCode`**, which will be used to **define** the **status** of the **response** our Server will be sending. This **enumeration** should hold **values** which are the **statuses** and **integer values** which will be the **codes.**

```
public enum HttpResponseStatusCode
{
    Ok = 200,
    Created = 201,
    Found = 302,
    SeeOther = 303,
    BadRequest = 400,
    Unauthorized = 401,
    Forbidden = 403,
    NotFound = 404,
    InternalServerError = 500
}
```

For now, our simple handmade web server **does NOT need** to support other **HTTP status codes**. These are quite enough for a normal communication between a client a server.

Exceptions Namespace

The **`Exceptions`** namespace will hold classes which will be used for error handling on the Server. There will be 2 such exception classes for now – **`BadRequestException`** and **`InternalServerErrorException`**.
Those exceptions will be used as a promise, that the Server will **always return** a **Response**, even in the **rare event** of a **Runtime Error**.

The Server will catch errors of **`BadRequestException`** type, first. If it catches an error of this type, a **`400 Bad Request Response`** will be returned, with the Exception's **`message`** as content.

Any other errors will be caught as **`InternalServerErrorException`** or the base **`Exception`**. In that case, a **`500 Internal Server Error`** will be returned, with the InternalServerErrorException's **`message`** as content.

BadRequestException

Create a class, called **`BadRequestException`**. This exception will be thrown when there is an error with the **parsing** of the **HttpRequest**, e.g. **Unsupported HTTP Protocol**, **Unsupported HTTP Method**, **Malformed Request** etc.

The class should **derive** from the **`Exception`** class, and should have a **default message**:

"**`The Request was malformed or contains unsupported elements.`**"

InternalServerErrorException

Create a class, called **`InternalServerErrorException`**. This exception will be thrown whenever there is an error that the Server was not suppoused to encounter.

The class should **derive** from the **`Exception`** class, and should have a **default message**:

"**`The Server has encountered an error.`**"

Extensions Namespace

The **`Extensions`** namespace will hold classes with helpful extension methods – helper methods… Good Ol' C#.
There will be **1 class** which you'll have to implement – **`StringExtensions`**.

StringExtensions

Create a class, called **`StringExtensions`**. In the class, implement a **`string`** extension method called **`Capitalize()`**, which literally just **capitalizes** the **String** (makes the **first letter** – **capital** and **all other** – **lowercase**).

Headers Namespace

The **Headers** namespace will hold the **classes** and **interfaces** which will be used to **store** the **data** of the **HTTP Headers** of the **Requests** and **Responses**.

HttpHeader

Create a class, called **HttpHeader**, which will be used to store data about a **HTTP Request / Response Header**.

```
public HttpHeader(string key, string value)
{
    CoreValidator.ThrowIfNullOrEmpty(text: key, name: nameof(key));
    CoreValidator.ThrowIfNullOrEmpty(text: value, name: nameof(value));

    this.Key = key;
    this.Value = value;
}

public string Key { get; }

public string Value { get; }

public override string ToString()
{
    return $"{this.Key}: {this.Value}";
}
```

The **key** will be the **header's name**, and the **value** – its **value**. There is also a useful **ToString()** method, which brings a well-formatted **web ready** (it can be used in web communication **without further formatting**) string representation of the header.

IHttpHeaderCollection

Create an interface, called **IHttpHeaderCollection**, which will describe the behaviour of a **Repository-like object** for the **HttpHeaders**.

```
public interface IHttpHeaderCollection
{
    void AddHeader(HttpHeader header);

    bool ContainsHeader(string key);

    HttpHeader GetHeader(string key);
}
```

HttpHeaderCollection

Create a class, called **HttpHeaderCollection**, which implements the **IHttpHeaderCollection** interface. The class is a Repository-like class. It should hold a **Dictionary collection** of **Headers** and should implement the interface's methods:

```
public class HttpHeaderCollection : IHttpHeaderCollection
{
    private readonly Dictionary<string, HttpHeader> headers;

    public HttpHeaderCollection()
    {
        this.headers = new Dictionary<string, HttpHeader>();
    }

    public void AddHeader(HttpHeader header)...

    public bool ContainsHeader(string key)...

    public HttpHeader GetHeader(string key)...

    public override string ToString()...
}
```

Implement each of these methods with the following functionalities:

- **`AddHeader()`** – **adds** the header to the **`Dictionary`** collection with **`key`** – the key of the **`Header`**, and **`value`** – the **`Header`**.
- **`ContainsHeader()`** – the main reason for the use of a **`Dictionary`**. **Fast search** using the **`Dictionary's hashtable`**. Returns a **`boolean`** result **depending** on weather the collection **contains** a **`Header`** with the **given key**.
- **`GetHeader()`** – **retrieves** from the collection and **returns** the **`Header`** with the **given key**, **if present**. If there is **NO such `Header`**, the method should return **`null`**.
- **`ToString()`** – **returns all** of the **`Headers`' string representations**, **separated** by **new line** ("**`/r/n`**"). or **`Environment.NewLine.`**

Requests Namespace

The time has come for us to aggregate everything into the main functionality classes.

The **`Requests`** namespace will hold classes and interfaces for storing and manipulating data about HTTP Requests.

IHttpRequest

Create an interface, called **`IHttpRequest`**, which will describe the behaviour of a **Request object**.

```
public interface IHttpRequest
{
    string Path { get; }

    string Url { get; }

    Dictionary<string, object> FormData { get; }

    Dictionary<string, object> QueryData { get; }

    IHttpHeaderCollection Headers { get; }

    HttpRequestMethod RequestMethod { get; }
}
```

HttpRequest

Create a class, called **HttpRequest**, which implements the **IHttpRequest** interface. The class should implement the Interface's methods.

```
public HttpRequest(string requestString)
{
    CoreValidator.ThrowIfNullOrEmpty(text: requestString, name: nameof(requestString));

    this.FormData = new Dictionary<string, object>();
    this.QueryData = new Dictionary<string, object>();
    this.Headers = new HttpHeaderCollection();

    //TODO: Parse request data...
}

public string Path { get; private set; }

public string Url { get; private set; }

public Dictionary<string, object> FormData { get; }

public Dictionary<string, object> QueryData { get; }

public IHttpHeaderCollection Headers { get; }

public HttpRequestMethod RequestMethod { get; private set; }
```

As you can see the **HttpRequest** holds its **Path**, **Url**, **RequestMethod**, **Headers**, **Data** etc. Those things come from the requestString, which is passed to its constructor. That's how a HttpRequest should be instantiated.

The requestString will be something in the following format:

```
{method} {url} {protocol}
{header1key}: {header1value}
{header2key}: {header2value}
...
<CRLF>
{bodyparameter1key}={bodyparameter1value}&{bodyparameter2key}={bodyparameter2value}...
```

NOTE: As you should already know, the **body parameters** are optional.

Now let's **destructure** a **normal request** and see how we should **map** each of its **components** to our **class**'s **properties**.

GET Request

Request Line	`GET /home/index?search=nissan&category=SUV#hashtag HTTP/1.1`

HTTP Request Headers	`Host: localhost:8000`
	`Accept: text/plain`
	`Authorization: Bearer POwJDsBzl5nrxDF4jah64RtAM022XBFypl8h6lcgi`
	`Cache-Control: no-cache`
	`User-Agent: Chrome/64.5`

Empty line (/r/n)	`<CRLF>`

Request Line: The request line is simple, it holds:

- **The Request Method** – The method however, is completely uppercase, which means you must somehow format it in order for you to be able to parse it into the **HttpRequestMethod** Enumeration. (**NO switch / cases** and **if / else**'s).

- **The Request URL** – The whole URL, holding the **Path**, the **Query String** and the **Fragment**.
  - Extract the **Path** by **splitting** and **formatting** the **URL**, and map it to the **Path** property.

  - Extract the **Query string** and **map** its **parameters** to the **Query Data Dictionary**. Parameters should be mapped as follows: **parameterName** = **key**, **parameterValue** = **value**.

  - **Fragments** are mostly used on the client side, so there is no need to store them in our class, thus there is no property for them.

- **The Request Protocol** – It **MUST** be equal to "**HTTP/1.1**".

Request Headers – They can easily be **parsed** in the following format "**{key}: {value}**" you just have to **split them**, and **create** an **instance** of **HttpHeader**, then add it to the **Headers** of the **Request**.

Empty Line – Denotes the end of the **Request Headers**.

POST Request

```
POST /home/index HTTP/1.1
Host: localhost:8000
Accept: text/plain
Authorization: Bearer POwJDsBzl5nrxDF4jah6 RtAM022XBFypl8h6lcgi
Cache-Control: no-cache
User-Agent: Chrome/64.5
<CRLF>
username=pesho&password=12345
```

Request Body

A **POST Request** is almost the same, except that it has a **body**. The **Request Body** holds parameters, which should be mapped to the **Form Data Dictionary** in the same way that query parameters are **mapped** to the **Query Data Dictionary**.

That's a lot of things to do, which in itself means a lot of methods if you want to write High-Quality Code. Here is some hints. Implement the following methods:

```
private bool IsValidRequestLine(string[] requestLine)...

private bool IsValidRequestQueryString(string queryString, string[] queryParameters)...

private void ParseRequestMethod(string[] requestLine)...

private void ParseRequestUrl(string[] requestLine)...

private void ParseRequestPath()...

private void ParseHeaders(string[] requestContent)...

private void ParseCookies()...

private void ParseQueryParameters()...

private void ParseFormDataParameters(string formData)...

private void ParseRequestParameters(string formData)...

private void ParseRequest(string requestString)...
```

The **`ParseRequest()`** method is the root call:

```
public HttpRequest(string requestString)
{
    CoreValidator.ThrowIfNullOrEmpty(text: requestString, name: nameof(requestString));

    this.FormData = new Dictionary<string, object>();
    this.QueryData = new Dictionary<string, object>();
    this.Headers = new HttpHeaderCollection();

    this.ParseRequest(requestString);
}
```

Now let's see what it looks like:

```
private void ParseRequest(string requestString)
{
    string[] splitRequestContent = requestString
        .Split(separator: new[] {GlobalConstants.HttpNewLine}, StringSplitOptions.None);

    string[] requestLine = splitRequestContent[0].Trim().
        Split(separator: new [] {' '}, StringSplitOptions.RemoveEmptyEntries);

    if (!this.IsValidRequestLine(requestLine))
    {
        throw new BadRequestException();
    }

    this.ParseRequestMethod(requestLine);
    this.ParseRequestUrl(requestLine);
    this.ParseRequestPath();

    this.ParseHeaders(splitRequestContent.Skip(1).ToArray());
    this.ParseCookies();

    this.ParseRequestParameters(splitRequestContent[splitRequestContent.Length - 1]);
}
```

As you can see it splits the requestString, creating an array of lines. Then it takes the 1st line (The **Request Line**) and splits, creating the **split `requestLine`**. The method than proceeds with a sequence of **method calls** to **validate** and **map** the **string data** to the **properties**.

These methods will be **yours to implement**. You will, however, be explained, what each methods does, to ease your implementation, so let's start!

IsValidRequestLine() Method

This method checks if the **split `requestLine` holds exactly 3 elements**, and if the **3rd element** is **equal** to "**`HTTP/1.1`**". Returns a **boolean** result.

IsValidRequestQueryString() Method

This method is used in the **`ParseQueryParameters()`** method. It checks if the **`Query string`** is **NOT NULL** or **empty** and if there is **atleast 1** or more **`queryParameters`**.

ParseRequestMethod() Method

Sets the **`Request`**'s **`Method`**, by **parsing** the **1st element** from the **split `requestLine`**.

ParseRequestUrl() Method

Sets the **`Request`**'s **`Url`** to the **2nd element** from the **split `requestLine`**.

ParseRequestPath() Method

Sets the **`Request`**'s **`Path`**, by **splitting** the **`Request`**'s **`Url`** and taking **only** the **path** from it.

ParseHeaders() Method

Skipping the **first line** (the **request line**), **traverses** the **request lines** until it reaches an **empty line** (the **<CRLF> line**). Each line represents a header, which must be split and parsed. Then the **string data** is **mapped** to an **`HttpHeader` object**, and the **object** itself is **added** to the **`Headers`** property of the **Request**.

Throws a **`BadRequestException`** if there is no "**`Host`**" **Header** present **after** the **parsing**.

ParseQueryParameters() Method

Extracts the **`Query string`**, by **splitting** the **`Request`**'s **`Url`** and taking **only** the **query** from it. Then **splits** the **`Query string`** into **different parameters**, and **maps** each of them into the **`Query Data Dictionary`**.

Validates the **`Query string`** and parameters by calling the **`IsValidrequestQueryString()`** method.

Does **nothing** if the **`Request`**'s **`Url`** contains NO **`Query string`**.

Throws a **`BadRequestException`** if the **`Query string`** is **invalid**.

ParseFormDataParameters() Method

Splits the **`Request`**'s **Body** into **different parameters**, and **maps** each of them into the **`Form Data Dictionary`**.

Does **nothing** if the **`Request`** contains NO **Body**.

ParseRequestParameters() Method

Invokes the **`ParseQueryParameters()`** and the **`ParseFormDataParameters()`** methods. Just a wrapping method.

If you implement all methods **correctly**, you should be able to parse even complex requests with no problems.

Responses Namespace

The **`Responses`** namespace will hold **classes** and **interfaces** for storing and manipulating data about **HTTP Responses**.

IHttpResponse

Create an interface, called **IHttpResponse**, which will describe the behaviour of a **Response object**.

```
public interface IHttpResponse
{
    HttpResponseStatusCode StatusCode { get; set; }

    IHttpHeaderCollection Headers { get; }

    byte[] Content { get; set; }

    void AddHeader(HttpHeader header);

    byte[] GetBytes();
}
```

HttpResponse

Create a class, called **HttpResponse**, which implements the **IHttpResponse** interface. The class should implement the Interface's methods,

```
public class HttpResponse : IHttpResponse
{
    public HttpResponse()
    {
        this.Headers = new HttpHeaderCollection();
        this.Content = new byte[0];
    }

    public HttpResponse(HttpResponseStatusCode statusCode)
        : this()
    {
        CoreValidator.ThrowIfNull(statusCode, name: nameof(statusCode));
        this.StatusCode = statusCode;
    }

    public HttpResponseStatusCode StatusCode { get; set;  }

    public IHttpHeaderCollection Headers { get; }

    public byte[] Content { get; set; }

    public void AddHeader(HttpHeader header)...

    public byte[] GetBytes()...

    public override string ToString()...
}
```

As you can see the **HttpResponse** holds its **StatusCode**, **Headers**, **Content** etc. These are the only things we will need for now. Unlike the Request, the Response is gradually being built, depending on the processing of the request. An **HttpResponse** is instantiated with an object with **NULL** or **default values**.

The **Server** receives **Requests** in **text format** and **should return Responses** in the **same format**.
The string representations of the **HTTP Responses** are in the following format:

```
{protocol} {statusCode} {status}
{header1key}: {header1value}
{header2key}: {header2value}
...
```

<CRLF>
{content}

NOTE: As you should already know, the **content** (**Response body**) is **optional**.

Now, while building our **HttpResponse** object, we can set its **StatusCode** or we can do that later in time. Normally, we would just set it upon **initialization** through the **constructor**.

AddHeader() Method

We can add **Headers** to it, gradually with the processing of the **Request**, using the **AddHeader()** method.

```
public void AddHeader(HttpHeader header)
{
    CoreValidator.ThrowIfNull(header, name: nameof(header));
    this.Headers.Add(header);
}
```

The other properties, **StatusCode** and **Content** can be set from outside using their **public setters**.

Now let's see what the **ToString()** and **GetBytes()** methods do.

ToString() Method

The **ToString()** method forms the **Response line** – the line holding the **protocol**, the **status code** and the **status**, and the **Response Headers** along with the **<CRLF> line**. These properties are **concatenated** in a **string** and returned.

```
public override string ToString()
{
    StringBuilder result = new StringBuilder();

    result
        .Append($"{GlobalConstants.HttpOneProtocolFragment} {(int)this.StatusCode} {this.StatusCode.ToString()}")
        .Append(GlobalConstants.HttpNewLine)
        .Append(this.Headers)
        .Append(GlobalConstants.HttpNewLine);

    result.Append(GlobalConstants.HttpNewLine);

    return result.ToString();
}
```

And now you might wonder, why is the **Content** of the **Response** a **byte[]** value, and why does the string representation of the **Response** not holding the **Content**. Well, that is because the **Content** can also be a direct file data, like **images** and **audio**. That is why we need a **byte[]** array to store that data.

And that's where the **GetBytes()** method comes.

GetBytes() Method

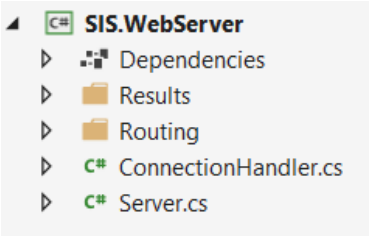
The **GetBytes()** method **converts** the **result** from the **ToString()** method to a **byte[]** array, and **concatenates** to it the **Content bytes**, thus forming the full **Response** in **byte format**. Exactly what we need to send to the Client.

And with that we are finished with the **HTTP work** for now. We can proceed to the main functionality of the Server.

3. SIS.WebServer Project Architecture

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

Create the following namespace and class architecture in the project:



Results Namespace

The **`Results`** namespace will hold **several classes** which **derive** from the **`HttpResponse`** class. These classes will be used to implement basic web applications using the **SIS**. There are **`3 classes`** which you must put here – **`TextResult`**, **`HtmlResult`** and **`RedirectResult`**.

TextResult

Designed to hold text contents, this is a simple plain **text response**. It should have a **`Content-Type`** header – **text/plain**.

```
public class TextResult : HttpResponse
{
    public TextResult(string content, HttpResponseStatusCode responseStatusCode,
        string contentType = "text/plain; charset=utf-8")
        : base(responseStatusCode)
    {
        this.Headers.AddHeader(new HttpHeader(HttpHeader.ContentType, contentType));
        this.Content = Encoding.UTF8.GetBytes(content);
    }

    public TextResult(byte[] content, HttpResponseStatusCode responseStatusCode,
        string contentType = "text/plain; charset=utf-8")
        : base(responseStatusCode)
    {
        this.Content = content;
        this.Headers.AddHeader(new HttpHeader(HttpHeader.ContentType, contentType));
    }
}
```

HtmlResult

Designed to hold HTML contents, this is a simple **HTML response**, with which we can return **HTML pages** or just **simple messages**. It should have a **`Content-Type`** header – **text/html**.

```
public class HtmlResult : HttpResponse
{
    public HtmlResult(string content, HttpResponseStatusCode responseStatusCode)
        : base(responseStatusCode)
    {
        this.Headers.AddHeader(new HttpHeader(
            HttpHeader.ContentType, "text/html; charset=utf-8"));
        this.Content = Encoding.UTF8.GetBytes(content);
    }
}
```

RedirectResult

Designed to hold **NO CONTENT**, and its only purpouse is to **redirect** the **client**. This **Response** has a **location** though. Its **status** is **predefined** also. It has status – **`SeeOther`**.

```
public class RedirectResult : HttpResponse
{
    public RedirectResult(string location)
        : base(HttpResponseStatusCode.SeeOther)
    {
        this.Headers.AddHeader(new HttpHeader(HttpHeader.Location, location));
    }
}
```

These are all the **`Results`** we need for basic application development.

Routing Namespace

The **`Routing`** namespace will hold the **routing logic** and the **configuration** of the Server. It will hold one interface and one class – I**`ServerRoutingTable`** and **`ServerRoutingTable`**.

```
public interface IServerRoutingTable
{
    void Add(HttpRequestMethod method, string path, Func<IHttpRequest, IHttpResponse> func);

    bool Contains(HttpRequestMethod requestMethod, string path);

    Func<IHttpRequest, IHttpResponse> Get(HttpRequestMethod requestMethod, string path);
}
```

This class holds a collosal collection of **nested dictionaries**, which will be used for **routing**:

```
public class ServerRoutingTable : IServerRoutingTable
{
    private readonly Dictionary<HttpRequestMethod, Dictionary<string, Func<IHttpRequest, IHttpResponse>>> routes;

    public ServerRoutingTable()
    {
        this.routes = new Dictionary<HttpRequestMethod, Dictionary<string, Func<IHttpRequest, IHttpResponse>>>
        {
            [HttpRequestMethod.Get] = new Dictionary<string, Func<IHttpRequest, IHttpResponse>>(),
            [HttpRequestMethod.Post] = new Dictionary<string, Func<IHttpRequest, IHttpResponse>>(),
            [HttpRequestMethod.Put] = new Dictionary<string, Func<IHttpRequest, IHttpResponse>>(),
            [HttpRequestMethod.Delete] = new Dictionary<string, Func<IHttpRequest, IHttpResponse>>()
        };
    }

    public void Add(HttpRequestMethod method, string path, Func<IHttpRequest, IHttpResponse> func)...

    public bool Contains(HttpRequestMethod requestMethod, string path)...

    public Func<IHttpRequest, IHttpResponse> Get(HttpRequestMethod requestMethod, string path)...
}
```

This is basically the main algorithm for **Request Handling**. A **Request Handler** is configured by **setting** the **Request Method** and the **Path** of the **Request**. Then the **Handler** itself is a **Function** which **accepts** a **Request parameter** and **generates** a **Response parameter**.

<Method, <Path, Func>>

We will see an example further below. Now let's get to the real deal.

Server class

The **Server** class is the main wrapper class for the **TCP connection**. It uses a **TcpListener** to capture Client connections and then passes them to the **ConnectionHandler**, which processes them.

```
public class Server
{
    private const string LocalhostIpAddress = "127.0.0.1";

    private readonly int port;

    private readonly TcpListener listener;

    private readonly IServerRoutingTable serverRoutingTable;

    private bool isRunning;

    public Server(int port, IServerRoutingTable serverRoutingTable)...

    public void Run()...

    public async Task Listen(Socket client)...
}
```

The **constructor** should be used to initialize the **Listener** and the **RoutingTable**.

```
public Server(int port, IServerRoutingTable serverRoutingTable)
{
    this.port = port;
    this.listener = new TcpListener(IPAddress.Parse(LocalhostIpAddress), port);

    this.serverRoutingTable = serverRoutingTable;
}
```

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

```
public void Run()
{
    this.listener.Start();
    this.isRunning = true;

    Console.WriteLine(value:$"Server started at http://{LocalhostIpAddress}:{this.port}");

    while (this.isRunning)
    {
        Console.WriteLine(value:"Waiting for client...");

        var client = this.listener.AcceptSocket();

        this.Listen(client);
    }
}
```

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**:

```
public void Listen(Socket client)
{
    var connectionHandler = new ConnectionHandler(client, this.serverRoutingTable);
    connectionHandler.ProcessRequest();
}
```

As you can see we **instantiate** a new **ConnectionHandler** for each client connection, and then we pass the client to the **ConnectionHandler**, along with the **routing table**, so that the **Request** can be **processed**.

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**.

```
public class ConnectionHandler
{
    private readonly Socket client;

    private readonly IServerRoutingTable serverRoutingTable;

    public ConnectionHandler(
        Socket client,
        IServerRoutingTable serverRoutingTable)...

    public void ProcessRequest()...

    private IHttpRequest ReadRequest()...

    private IHttpResponse HandleRequest(IHttpRequest httpRequest)...

    private void PrepareResponse(IHttpResponse httpResponse)...
}
```

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

```
public ConnectionHandler(
    Socket client,
    IServerRoutingTable serverRoutingTable)
{
    CoreValidator.ThrowIfNull(client, name: nameof(client));
    CoreValidator.ThrowIfNull(serverRoutingTable, name: nameof(serverRoutingTable));

    this.client = client;
    this.serverRoutingTable = serverRoutingTable;
}
```

The **`ProcessRequest ()`** method 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**.

```
public void ProcessRequest()
{
    try
    {
        var httpRequest = this.ReadRequest();

        if (httpRequest != null)
        {
            Console.WriteLine(value:$"Processing: {httpRequest.RequestMethod} {httpRequest.Path}...");

            var httpResponse = this.HandleRequest(httpRequest);

            this.PrepareResponse(httpResponse);
        }
    }
    catch (BadRequestException e)
    {
        this.PrepareResponse(new TextResult(e.ToString(), HttpResponseStatusCode.BadRequest));
    }
    catch (Exception e)
    {
        this.PrepareResponse(new TextResult(e.ToString(), HttpResponseStatusCode.InternalServerError));
    }

    this.client.Shutdown(how:SocketShutdown.Both);
}
```

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

```
private IHttpRequest ReadRequest()
{
    var result = new StringBuilder();
    var data = new ArraySegment<byte>(array:new byte[1024]);

    while (true)
    {
        int numberOfBytesRead = this.client.Receive(data.Array, SocketFlags.None);

        if (numberOfBytesRead == 0)
        {
            break;
        }

        var bytesAsString = Encoding.UTF8.GetString(data.Array, index:0, count:numberOfBytesRead);
        result.Append(bytesAsString);

        if (numberOfBytesRead < 1023)
        {
            break;
        }
    }

    if (result.Length == 0)
    {
        return null;
    }

    return new HttpRequest(result.ToString());
}
```

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.

```
private IHttpResponse HandleRequest(IHttpRequest httpRequest)
{
    if (!this.serverRoutingTable.Contains(httpRequest.RequestMethod, httpRequest.Path))
    {
        return new TextResult( content: $"Route with method {httpRequest.RequestMethod} and path
          \"{httpRequest.Path}\" not found.", HttpResponseStatusCode.NotFound);
    }

    return this.serverRoutingTable.Get(httpRequest.RequestMethod, httpRequest.Path).Invoke
      (httpRequest);
}
```

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

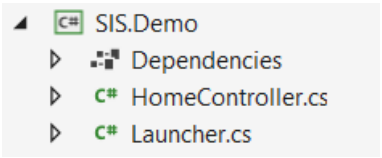
```
private void PrepareResponse(IHttpResponse httpResponse)
{
    byte[] byteSegments = httpResponse.GetBytes();

    this.client.Send(byteSegments, SocketFlags.None);
}
```

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**.

4. 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:

```
public class HomeController
{
    public IHttpResponse Index(IHttpRequest request)
    {
        string content = "<h1>Hello, World!</h1>";

        return new HtmlResult(content, HttpResponseStatusCode.Ok);
    }
}
```

Launcher

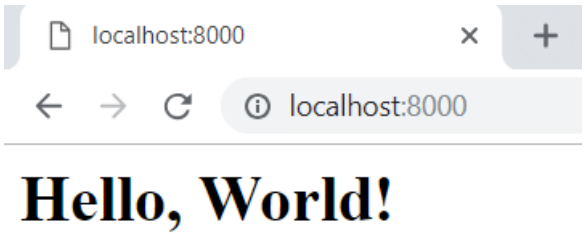
The **Launcher** class should hold the **`Main`** method, which instantiates a **`Server`** and **configures** it to **handle requests** using the **`ServerRoutingTable`**.

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

```
public static void Main(string[] args)
{
    IServerRoutingTable serverRoutingTable = new ServerRoutingTable();

    serverRoutingTable.Add(
        HttpRequestMethod.Get,
        path: "/",
        func: request => new HomeController().Index(request));

    Server server = new Server(port: 8000, serverRoutingTable);

    server.Run();
}
```

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 first **Hello World app** with the **SIS**!