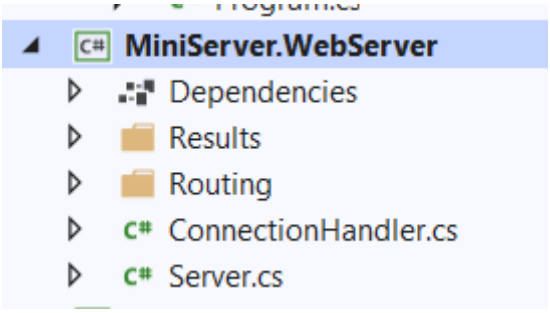
1. Mini HTTP Server – Упражнение

2. SIS.WebServer Архитектура

"**WebServer**" проекта ще съдържа информация за класовете, които изграждат връзка с "**TCP**". Тези класове ще комуникират с класовете от "**HTTP Project**". Проекта ще изнася няколко класа, които ще служат за "**външният**" свят, за да създаваме приложния.

Създайте следните папки и класове:



Results папка

"Results" папка ще съдържа няколко класа, които са наследени от "**HttpResponse**" класа. Тези класове, ще използват за имплементираме прости приложения с "**MiniServer**". Трябва да създадем три класа вътре - "**TextResult**", "**HtmlResult**" и "**RedirectResult**".

TextResult

Създаден е така, че да връща текст, като отговор. Трябва да има "**Content-Type**" и 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

Създаваме този клас, да връща HTML в себе си. Така чрез този клас, ние можем да върнем HTML или просто съобщение. Трябва да има "**Content-Type**" и 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

Този клас, не трябва да има Content. Единствената задача е да бъде пренасочен. Този "**Response**" има локация. Статуса трябва да бъде – "**SeeOther**".

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

Routing папка

В папка, ще съдържа логиката за рутиране и конфигурация на сървъра. Ще съдържа един интерфейс и един клас - "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);
}
```

Този клас съдържа големи колекции от насложени асоциативни масиви, които ще се използват за рутиране.

```
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)...
}
```

Това е главният алгоритъм за "**Request Handling**". "**Request Handler**" се конфигурира, като настройва "**Request Method**" и "**Path**" на заявката. "**Handler**" сам по себе си е функция, която приема "**Request**" параметър и генерира "**Response**" параметър.

<Method, <Path, Func>>

Ще видим по-надолу в примерите как работи.

Server клас

`Server` класа е обвивка за **TCP connection**. Използва **`TcpListener`** , за да запише връзката с клиента и да я подаде на **`ConnectionHandler`**, която го изпълнява.

```
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`**.

Конструкторът се използва, за да бъде инициализиран **`Listener`** и **`RoutingTable`**

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

    this.serverRoutingTable = serverRoutingTable;
}
```

Този метод се използва да се процеса на слушане. Процесът трябва да бъде асинхронен, за да подсигури функционалността, когато двама клиента изпратят заявка.

```
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.AcceptSocketAsync().GetAwaiter().GetResult();

        Task.Run(() => this.Listen(client));
    }
}
```

`Listen()` метода е главният процес при свързване с клиента.

```
public async Task Listen(Socket client)
{
    var connectionHandler = new ConnectionHandler(client, this.serverRoutingTable);
    await connectionHandler.ProcessRequestAsync();
}
```

Както виждате, ние създаваме нов **`ConnectionHandler`**, за всяка нова връзка и я подаваме на **`ConnectionHandler`**, заедно с **routing table**, така че заявката да бъде изпълнена.

ConnectionHandler клас

`ConnectionHandler` е клас, който произвежда връзката с клиента. Приема връзката, изважда заявката, като низ и минава процес през **routing table**, като я изпраща обратно на "Response" в байт формат, чрез **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)...
}
```

Конструктора се използва, за да се инициализира **socket** и **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;
}
```

`ProcessRequestAsync()` метода е асинхронен метод, който съдържа главната функционалност на класа. Използва и други методи да чете **заявки**, да ги **обработва** и да създава **Response**, Който да бъде върнат на клиента и най-накрая да затвори връзката.

```
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);
}
```

`ReadRequest()` метода е асинхронен метод, който чете байт данни, от връзката с клиента, изважда низа от заявката и след това го обръща в **`HttpRequest`** обект.

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

    while (true)
    {
        int numberOfBytesRead = await this.client.ReceiveAsync(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());
}
```

`HandleRequest()` метода проверява ако **routing table** има **handler** за дадената заявка, като използва **Request's Method** и **Path**

- Ако няма такъв **handler** "**`Not Found`**" отговор е върнат.
- Ако има такъв **handler**, функцията е извикана и резултата е върнат.

```
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);
}
```

PrepareResponse() метода изважда байт данни от отговора и ги изпраща на клиента.

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

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

И ето тука приключваме с **ConnectionHandler и WebServer** проекта, като цяло.

3. Hello, World!

Създайте трети проект, който да се казва **MiniServer.Demo**. Той трябва да реферира и двата проека **MiniServer.HTTP** и **MiniServer.WebServer**.

Създайте следните класове:

HomeController клас

HomeController класа трябва да има един метод – **Index()**, който да изглежда по този начин:

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

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

Launcher клас

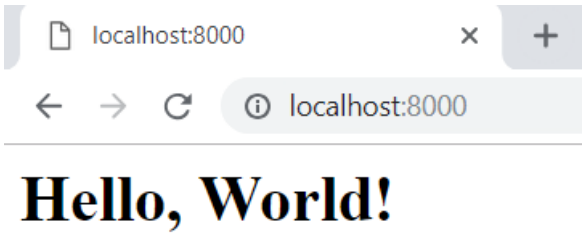
Launcher класа трябва да съдържа **Main** метода, който инстанцира **Server** и го конфигурира да се справя със заявките, като използва **ServerRoutingTable.**

Конфигурирайте само пътя “**/**”, като използва ламбда функция, която извиква **HomeController.Index** метода**.**

```
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();
}
```

Сега стартирайте **MiniServer.Demo** проекта и трябва да видите това на конзолата, ако всичко е наред:

Отворете браузъра и отидете на **`localhost:8000`** и трябва да видите това:



Поздравления! Завършихте първото си приложение с **MiniServer**.

Министерство на образованието и науката (МОН)

- Настоящият курс (презентации, примери, задачи, упражнения и др.) е разработен за нуждите на Национална програма "**Обучение за ИТ кариера**" на МОН за подготовка по професия "Приложен програмист".

- Курсът е базиран на учебно съдържание и методика, предоставени от **фондация "Софтуерен университет"** и се разпространява под **свободен лиценз CC-BY-NC-SA** (Creative Commons Attribution-Non-Commercial-Share-Alike 4.0 International).