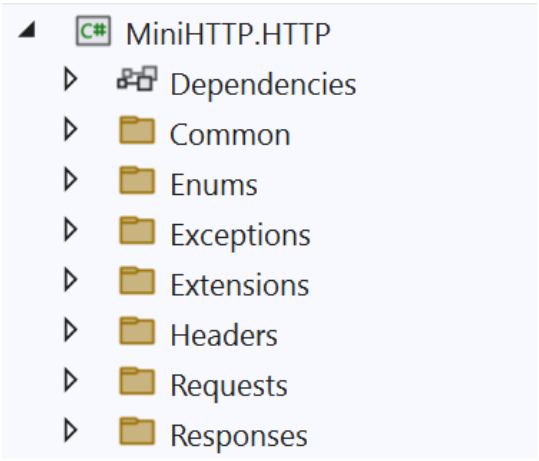
Mini HTTP Server

С помощта на този документ, вие ще създадете малък HTTP сървър, който изпраща и приема заявки. Първо нека да създадем архитектурата на нашият проект. Създайте нов Solution и го кръстете MiniHTTP. Добавете към него два проекта от тип Library: ***MiniHTTP.HTTP*** *и* ***MiniHTTP.WebServer****.*

MiniServer.HTTP

MiniServer.HTTP проекта ще съдържа всички класове (и техните интерфейси), които ще бъда използвани да имплементираме HTTP комуникацията с TCP Link между клиента и нашият сървър. Можем да работим само с низове и байт масиви, но ще следваме добрите практики и ще го направим кода да бъде лесно четим и преизползваем.

Създайте следните папки в проекта:



Както виждате архитектурата на папките е много добре разделена.
Нека сега да започнем със създаването на класовете.

Common папка

"**Common**" папката, ще съдържа класове, които се използват в целият проект. Ще имаме два класа - "**`GlobalConstants`**" и "**`CoreValidator`**".

GlobalConstants

Създайте статичен клас "**`GlobalConstants`**", който ще бъде използван за споделните константи:

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

    public const string HostHeaderKey = "Host";

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

Това са единствените константи, от които имаме нужда засега.

CoreValidator

Създайте клас **"CoreValidator"**, който ще има два метода, за проверка за "null" стойности или празни стрингове:

```
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 папка

Enums папката ще съдържа "**enumerations**". Има два енъма, от които сървърът ще се нуждае - "**HttpRequestMethod**" и "**HttpResponseStatusCode**"

HttpRequestMethod

Създайте **Enum**, с името "**HttpRequestMethod**". Той ще дефинира, метода ,които сървъра получава.

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

Нашият сървър, ще поддържа само "**GET**", "**POST**", "**PUT**" и "**DELETE**" и заявки. Нямаме нужда от по-сложни заявки засега.

HttpResponseStatusCode

Създайте **Enum**, с името "**HttpResponseStatusCode**". Той ще дефинира статус кода от отговора на нашият сървър. Този Enum, ще съдържа стойности, които са стутусите и цели числа, които ще представляват статус кода.

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

За сега нашият малък сървър, няма нужда да съдържа всички други статус кодове. Тези достатъчно сървъра и клиента да си комуникират.

Exceptions папка

"**Exceptions**" папката ще съдържа класове, които отговорят за правилното менажиране на грешките в сървъра. За начало ще имаме класа, които ще отговарят за грешките - "**BadRequestException**" и "**InternalServerErrorException**". Тези грешки, ще помагат, така, че сървъра винаги да връща отговор, дори в случай на Runtime Error.

Сървърът първо ще хваща грешки, които са от тип "**BadRequestException**". Ако хване грешка от този тип, сървъра трябва да върне "**400 Bad Request Response**" и съобщение са грешката.

Всички други грешки ще бъдат от тип "**InternalServerErrorException**" или от базовия клас "**Exception**". Ако прихванем една от тези грешки, сървъра трябва да върне а "**500 Internal Server Error**" и съобщение за грешката.

BadRequestException

Създайте клас, който се казва "**BadRequestException**". Тази грешка ще бъде хвърлена, когато сървъра не успее да парсне "HttpRequest", като Unsupported HTTP Protocol, Unsupported HTTP Method, Malformed Request и т.н.

"**BadRequestException**" трябва да наследява, "**Exception**" класа и трябва да има съобщение по подразбиране: "**The Request was malformed or contains unsupported elements.**"

InternalServerErrorException

Създайте клас, който се казва "**InternalServerErrorException** ". Тази грешка ще бъде хвърлена, когато не се е предполагало сървъра да се справи с нея.

"**InternalServerErrorException** " трябва да наследява, "**Exception**" класа и трябва да има съобщение по подразбиране: "**The Server has encountered an error.**"

Extensions папка

"**Extensions**" папката, ще съдържа **extension** методи, които ще ни помагат в разработката на нашият сървър.

Ще има един клас - "**StringExtensions**"

StringExtensions

В този клас, имплементирайте нисък **extension** метод, който се казва **Capitalize()**. Той трябва да направи първата буква **главна** и всички други малки.

Headers папка

"**Headers**" папката, ще съдържа класове и интерфейси, които ще съхраняват данни за **HTTP Headers** на **заявката** и **отговора**.

HttpHeader

Създайте клас, който се казва "**HttpHeader**". Той ще съхранява данните за **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}";
}
```

Пропъртито "**Key**", ще се използва за името на **Header-a** и пропъртито "**Value**", ще съдържа стойността. Имаме и в помощ "**ToString()**" метода, който ще връща добре форматиран и готов за използване **Header.**

IHttpHeaderCollection

Създайте интерфейс, който се казва "**IHttpHeaderCollection**", който ще опише действията на "**Repository-like object**" за **HttpHeaders**.

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

    bool ContainsHeader(string key);

    HttpHeader GetHeader(string key);
}
```

HttpHeaderCollection

Създайте клас, който се казва "**HttpHeaderCollection**", който имплементира "**IHttpHeaderCollection**" интерфейса. Този клас е като "Repository". Трябва да има **Dictionary** колекция на всички **Headers** и трябва да имплементирате всички методи на интерфейса.

```
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()...
}
```

Имплементирайте всеки един от тези методи със следните функционалности:

- `AddHeader()` – Добавя `Header-a`. в речника с ключ – ключа на `Header-a` и стойност самият `Header`.
- `ContainsHeader()` – Главна причина да използва `Dictionary`. Позволява ни бързо търсене. Трябва върнем `boolean`, в зависимост от това дали колекцията съдържа даденият ключ.
- `GetHeader()` – Връща `Header-a` от колекцията с дадения ключ. Ако не съществува такъв `Header`, метода трябва да върне `null`.
- `ToString()` – Връща всички `Headers`, като низ, разделени с нов ред - (`"/r/n"`) или `Environment.NewLine`

Requests папка

Сега е време да съберем всичко написано до момента в главните функциониращи класове.

`"Requests"` папката ще съдържа класове и интерфейси за съхранение и манипулиране данни за HTTP заявките.

IHttpRequest

Създайте интерфейс, който се казва "**`IHttpRequest`**", който ще описва поведението на **Request** обекта.

```
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

Създайте клас, който се казва "**HttpRequest**", който имплементира **IHttpRequest** интерфейса. Класът трябва да имплементира и методите на интерфейса.

```
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; }
```

Както виждате "**HttpRequest**", съдържа **Path**, **Url**, **RequestMethod**, **Headers**, **Data**. Тези данни идвам променлива "**requestString**", която се подава в конструктора. Това е начина, по които "**HttpRequest**" ще се инициализира.

"**requestString**" ще изглежда по този начин:

{method} {url} {protocol}
{header1key}: {header1value}

{header2key}: {header2value}
...
<CRLF>
{bodyparameter1key}={bodyparameter1value}&{bodyparameter2key}={bodyparameter2value}...

ВНИМАНИЕ: Както вече знаете, че **body parameters** не са задължителни.

Нека да разбием една нормална заявка и да видим как тя трябва да се мапне към нашите пропъртита.

GET заявка

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 Method` – Името на метода е винаги с главни букви, което означа, че някак си трябва да бъде форматиран, когато се парсва към Enum-а "`HttpRequestMethod`". (Не използвайте switch/case или if/else конструкции за преобразуването към Enum).
- `The Request URL` – Целият URL, съдържа "Path", "Query String" и "Fragment".
  - Вземете "Path" частта от URL-а, като го разделите, форматирате и запишете стойността в "Path" пропъртито.
  - `Вземете "Query String" частта и добавете стойностите към "Query Data" речника.`
    `Параметрите трябва да бъдат преобразувани по следният начин": 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.
  - `Fragments` предимно се използва в "client-side" частта, затова няма пропърти за тях.
- `The Request Protocol` – Трябва да бъде: "`HTTP/1.1`".

Те лесно могат да се преобразуват в следният формат: "**{key}: {value}**". Трябва да ги разделите и да създадете нова инстанция на "HttpHeader", и след това да се добави към "Headers" на "Request".

`Empty Line` –краят на "**Request Headers**"

POST заявка

```
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

"**`POST Request`**" е почти същият, освен неговото "**body**". "**Request Body**" съдържа параметри, които трябва да бъдат прехвърлени към "**Form Data**" речника, по същият начин, както "**Query Parameters**" бяха прехвърлени към "**Query Data**" речника.

Сега е време да имплементираме повече логика, което означава много методи, ако искаме да спазваме принципите за "**High-Quality Code**". Имплементирайте следните методи.

```
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)...
```

`ParseRequest()` е метода откъдето започва всичко:

```
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);
}
```

Нека да видим как изглежда той:

```
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]);
}
```

Както виждате "**requestString**" е разделен на нови редове в масив. Взимаме първият ред (The **Request Line**) и го разделяме. След това следват серия от проверки и присвояване не стойности към пропъртита.

Ще се наложи вие да имплементирате тези методи. Разбира се, ще ви бъдат дадени насоки, как да се справите с тях.

IsValidRequestLine() метод

Този метод проверя дали, разделеният "**requestLine**" съдържа точно 3 елемента и също така дали последният елемент е равен на "**HTTP/1.1**". Метода връща булев резултат.

IsValidRequestQueryString() метод

Този метод се използва в "**ParseQueryParameters()**" метода. Проверява дали "**Query**" низа е **NOT NULL** или **празен** и също така дали има поне **един** или много **queryParameters**.

ParseRequestMethod() метод

`RequestMethod` присвоя стойността, като преобразуваме първият елемент от разделеният "**`requestLine`**".

ParseRequestUrl() метод

`Url` присвоява стойността от вторият елемент на разделеният "**`requestLine`**".

ParseRequestPath() метод

`Path` присвоява стойността, като разделим **`Url`** и вземем само пътя от него.

ParseHeaders() метод

Пропускаме първият ред от "**`requestLine`**" и обхождаме всички останали редове, докато не стигнем празен ред. Всеки ред представлява "**`header`**", който трябва да бъде разделен и преобразуван към правилният тип. След това информацията от низа е прехвърлена към "**`HttpHeader`**" обекта и е добавен към "**`Headers`**" пропертито на "**`Request`**".

Хвърлете "**`BadRequestException`**", ако "**`Host"`** липсва след преобразуването.

ParseQueryParameters() метод

Извадете "**`Query`**" низа, като разделите "**`Request's Url`**" и вземете само "**`query`**" от него. След това разделете "**`Query`**" низа в различни параметри и го прехвърлете към "**`Query Data Dictionary`**".

Валидирайте "**`Query`**" низа, като извикате "**`IsValidrequestQueryString()`**"метода.

Ако в " **`Request's Url`**" липсва " **`Query`**" низа, не предприемайте действия.

Хвърлете "**`BadRequestException`**", ако "**`Query`**" не е валиден.

ParseFormDataParameters() метод

Разделете "**`Request`**'s **`Body`** " в различни параметри и го добавате към "**`Form Data Dictionary`**".

Не предприемайте действия, ако "**`Request`**" не съдържа тяло.

ParseRequestParameters() метод

Този метод извиква "**`ParseQueryParameters()`**" и " **`ParseFormDataParameters()`**" методите. Това е просто "**wrapping**" метод.

Ако сте имплементирали всички правилно, би трябвало да преобразувате дори и много сложни заявки без проблем.

Responses папка

"**Responses**" папката ще съдържа класове и интерфейси, които съдържат и манипулират информация за "**HTTP Responses**".

IHttpResponse

Създайте интерфейс, който се казва "**IHttpResponse**" и ще се съдържа следните пропъртита и методи:

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

    IHttpHeaderCollection Headers { get; }

    byte[] Content { get; set; }

    void AddHeader(HttpHeader header);

    byte[] GetBytes();
}
```

HttpResponse

Създайте клас, който се казва "**HttpResponse**" и имплементира "**IHttpResponse**" интерфейса.

```
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()...
}
```

Както виждате " **HttpResponse** " съдържа "**StatusCode**", "**Headers**", "**Content**" и т.н. Това са единствените неща, от които ние се нуждаем за сега. "**HttpResponse**" се инициализира с обект с Null ли по подразбиране стойности.

Сървърът получава "**Requests**" в текстов формат и трябва върне "**Responses**" в същият формат.

Репрезентацията на низа от "**HTTP Responses**" са в следният формат:

```
{protocol}                    {statusCode}                    {status}
{header1key}:                                                 {header1value}
{header2key}:                                                 {header2value}
...
<CRLF>
{content}
```

ЗАБЕЛЕЖКА: Както вече знаете, съдържанието (**Response body**) не е задължително.

Сега, докато изграждаме нашият "**HttpResponse**" обект, може да присвоим стойност за нашият "**StatusCode**" или може да го оставим за напред. Най-често ще присвояваме стойностите чрез конструктора.

AddHeader() метод

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

Можем добавяме "**Headers**", като използваме "**AddHeader()**" метода.

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

Другите пропъртита, "**StatusCode**" и "**Content**" могат да бъдат присвоени стойности от "външният свят", като използват публичните им сетъри.

Сега нека да видим "**ToString()**" и "**GetBytes()**" какво правят.

ToString() метод

"**ToString()**" метода формира "**Response**" реда – този ред съдържа протокола, статус кода, статус и "**Response Headers**", като завършва с празен ред. Тези пропъртита са съединени в един низ и върнати в края.

```
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();
}
```

И точно сега се нуждаем от "**GetBytes()**" метода.

GetBytes() метод

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

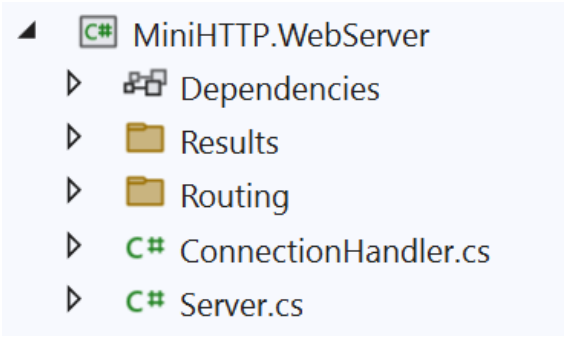
"**GetBytes()**" метода конвертира резултата от "**ToString()**" метода до "**byte[]**" масив, и долепя към него "**Content bytes**", затова формираме целият "**Response**" до байт формат. Точна това, което трябва да изпратим до сървъра.

И вече приключихме с работата по нашият HTTP сървър за сега.

MiniServer.WebServer

MiniServer.WebServer проекта е от тип **Console** и ще съдържа информация за класовете, които изграждат връзка с **TCP**. Тези класове ще комуникират с класовете от **MiniServer.HTTP**. Проекта ще изнася няколко класа, които ще служат за "външния" свят, за да създаваме приложния.

Създайте следните папки и класове:



Results папка

"Results" папката ще съдържа няколко класа, които са наследени от **`HttpResponse`** класа. Тези класове, ще използват за имплементираме прости приложения. 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 responceStatusCode,
        string contentType = "text/plain; charset=utf-8") : base(responceStatusCode)
    {
        this.Headers.AddHeader(new HttpHeader("Content-Type", contentType));
        this.Content = Encoding.UTF8.GetBytes(content);
    }

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

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

Routing папка

В папка, ще съдържа логиката за рутиране и конфигурация на сървъра. Ще съдържа един интерфейс и един клас **IServerRoutingTable** and **ServerRoutingTable**.

IServerRoutingTable

```
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);
}
```

ServerRoutingTable

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)...
}
```

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 за процеса на слушане. Процесът трябва да бъде асинхронен, за да подсигури функционалността, когато двама клиенти 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 table;

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

    public async Task ProcessRequestAsync()...

    private async Task<IHttpRequest> ReadRequest()...

    private IHttpResponse HandleRequest(IHttpRequest request)...

    private async void PrepareResponse(IHttpResponse response)...

}
```

Конструктора се използва, за да се инициализира **socket** и **routing table**.

```
public ConnectionHandler(Socket client, IServerRoutingTable table)
{
    CoreValidator.ThrowIfNull(client, nameof(client));
    CoreValidator.ThrowIfNull(table, nameof(table));

    this.client = client;
    this.table = table;
}
```

`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 async Task ProcessRequestAsync()
{
    try
    {
        var request = await ReadRequest();
        if (request != null)
        {
            Console.WriteLine($"Processing: {request.RequestMethod} {request.Path}");
            IHttpResponse response = HandleRequest(request);
            PrepareResponse(response);
        }
    }
    catch (BadRequestException e)
    {
        PrepareResponse(new TextResult(e.ToString(), HttpResponseStatusCode.BadRequest));
    }
    catch (Exception e)
    {
        PrepareResponse(new TextResult(e.ToString(), HttpResponseStatusCode.InternalServerError));
    }
    client.Shutdown(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, то „`Not Found`" отговор е върнат.**
- **Ако има такъв handler, функцията е извикана и резултата е върнат.**

```
private IHttpResponse HandleRequest(IHttpRequest request)
{
    if (!table.Contains(request.RequestMethod, request.Path))
    {
        return new TextResult($"Route with method {request.RequestMethod} and path {request.Path} not found.",
                HttpResponseStatusCode.NotFound);
    }
    return table.Get(request.RequestMethod, request.Path).Invoke(request);
}
```

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

```
private async void PrepareResponse(IHttpResponse response)
{
    byte[] bytes = response.GetBytes();
    await client.SendAsync(bytes, SocketFlags.None);
}
```

Това е финалният вид на нашия **`ConnectionHandler и`** проект.

MiniHTTP.Demo

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

Създайте следните класове:

HomeController клас

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

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

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

Launcher клас

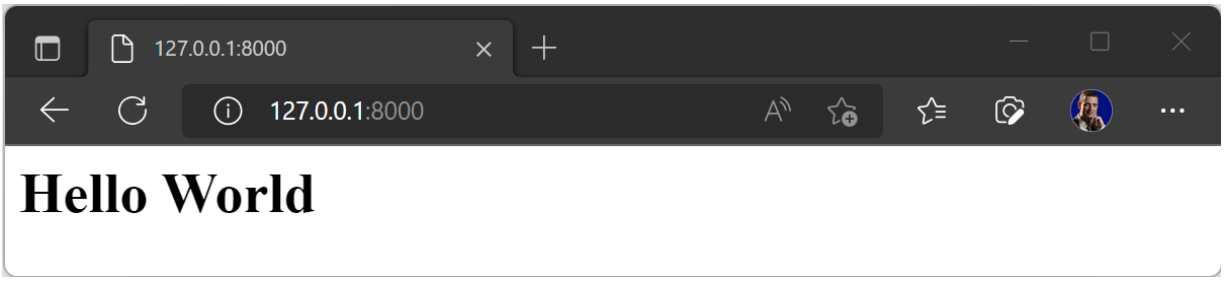
Launcher класа трябва да съдържа **Main** метода, който инстанцииpа **Server** и го конфигурира да се справя със заявките, като използва **ServerRoutingTable.** Конфигурирайте само пътя "**/**", като използва ламбда функция, която извиква **HomeController.Index** метода**.**

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        IServerRoutingTable serverRoutingTable = new ServerRoutingTable();
        serverRoutingTable.Add
        (
            HTTP.Enums.HttpRequestMethod.Get,
            "/",
            request => new HomeController().Index(request)
        );

        Server server = new Server(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!