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:

A picture containing graphical user interface

Description automatically generated

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 изпозлват в целият проект. Ще имаме два класа - "GlobalConstants**"** и **"CoreValidator".**

### GlobalConstants

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

Graphical user interface, text

Description automatically generated

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:

Graphical user interface, text, application, email

Description automatically generated

## 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.

Text

Description automatically generated

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.

Text

Description automatically generated

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 a "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 низ **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**.

Graphical user interface, application

Description automatically generated

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**.

Graphical user interface, text, application

Description automatically generated

### 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.

Graphical user interface, text, application

Description automatically generated

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.

Text

Description automatically generated

### HttpRequest

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

Graphical user interface, text

Description automatically generated

Както виждате "HttpRequest", съдържа Path, Url, RequestMethod, Headers, Data. Тези данни идвам променлива "**requestString**", която се подава в констуктора. 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 за задължителни.

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

### GET заявка

Text, letter

Description automatically generated

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-a "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-a, 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

Text

Description automatically generated

"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.

Graphical user interface, application

Description automatically generated

ParseRequest() е метода откъдето започва всичко:

Text

Description automatically generated

Нека да видим как изглежда той:

Graphical user interface, text, application, email

Description automatically generated

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:

Graphical user interface, text, application

Description automatically generated

### HttpResponse

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

Graphical user interface, text, application

Description automatically generated

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.

Graphical user interface, text

Description automatically generated with medium confidence

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.

Graphical user interface, text, application, email

Description automatically generated

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:

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

Description automatically generated

## Results папка

"Results" папката ще съдържа няколко класа, които са наследени от HttpResponseкласа. 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"

A screen shot of a computer code

Description automatically generated

### 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"

Graphical user interface, text, application

Description automatically generated

### RedirectResult

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

A close-up of a computer code

Description automatically generated

## Routing папка

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

### IServerRoutingTable

Graphical user interface, text, application

Description automatically generated

### ServerRoutingTable

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

Graphical user interface, text

Description automatically generated

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.

Graphical user interface, text, application

Description automatically generated

Конструкторът се използва, за да бъде инициализиран Listener и RoutingTable

Graphical user interface, text, application

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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

Text

Description automatically generated

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.**

A screenshot of a computer program

Description automatically generated

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

A screen shot of a computer code

Description automatically generated

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.

A screenshot of a computer program

Description automatically generated

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.

Graphical user interface, text, application, email

Description automatically generated

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.

A screenshot of a computer program

Description automatically generated

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

A close-up of a computer code

Description automatically generated

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:

Text

Description automatically generated

## HomeController клас

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

Graphical user interface, text, application

Description automatically generated

## Launcher клас

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

Text

Description automatically generated with medium confidence

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

Text

Description automatically generated

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

Graphical user interface, application

Description automatically generated

Поздравления!