Mini HTTP Server

С помощта на този документ, вие ще създаде малък HTTP сървър, който изпраща и приема заявки. В последствие ще създадем малък MVC Framework, който ще работи с нашият HTTP сървър.

1. Архитектура

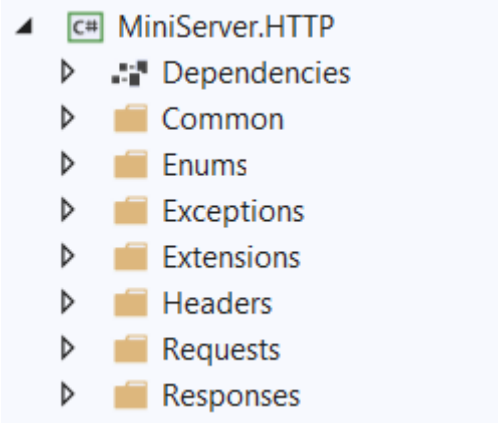
Първо нека да създадем архитектурата на нашият проект. Създайте нов Solution и го кръстете MiniServer. Добавете два проекта към него:

- **`MiniHTTP.HTTP`**
- **`MiniHTTP.WebServer`**

2. MiniServer.HTTP Архитектура

HTTP проекта ще съдържа всички класове (и техните интерфейси), които ще бъда използвани да имплементираме HTTP комуникацията с TCP сокет връзка 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 MiniServer.HTTP



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

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 споделените 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 статусите 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 хвана 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 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**.

```
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-а** 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`**

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 конскструктора.

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.

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).