NodeC++

Release 0.0.1

Sep 27, 2019

Setup

CHAPTER 1

Building NodeC++ with cmake

1.1 Requirements

- cmake (version 3.14 or above)
- a MacOS/Linux computer (this dosen't compile on windows because it doesn't use WinSock)

1.2 Downloading

You will need to download NodeC++ from Github. We recommend that you use the master branch (this is the most stable). If you want to use the development branch you can but there **will** be issues with it.

Now open a terminal and navigate to where you downloaded NodeC++

1.3 Running cmake

From the location of where you downloaded NodeC++ type

```
cmake -H. -B../build/
```

This will setup the build system in ../build

Once this completes navigate to ../build/ and run

```
make
```

This will fully build the project. This will produce the binaries in the output/ folder

Congratulations, you have build cmake

CHAPTER 2

Starting a HTTP Server

2.1 Initalize the class

Note:

This guide expects that you have already included the httpServer header.
I included my header like this:

```
#include "../headers/server.hpp"
```

Lets start by creating the httpServer class object. The initializer takes only one paramater, that is the port number.

```
int main() {
    http_server httpServer(3000);
    return 0;
}
```

I've started my server on port 3000, but you can start your server on any port (including port 80)

Warning: This HTTP Server does not https. If you choose to run the server on port *443*, you may find that the web browsers (such as Chrome) won't open the page.

2.2 Start the server

If you complile and run the program, it will exit without the actually starting the http_server. This is now what we need to do.

Add this to your code

```
httpServer.start();
```

Yay! The server has started. If you go to http://localhost:3000 you will discover a welcome message.

Warning: Anything that you type after the start function will **not** be run until the server closes! The server takes over the thread.

CHAPTER 3

Creating Pages & Routes

To create pages in NodeC++ you need to add routes that route your request to the right page. All of the routes require a callback function (this function will get called every single time a request is made to the path). I'm going to create a very simple index page.

3.1 Creating the callback function

I'm going to name my callback function *indexCallback*.

```
1 void indexCallback(Request req, responce res) {
2     res.write("Index Page");
3     res.send("200");
4     res.end();
5 }
```

While most of what this function does will be covered in the responceClass, i'll just go over the very important bits. The first line (where the function is decleared), requires two paramaters:

- a Request Struct
- a responce class

Warning: Without the correct paramaters the program will (most likley) not compile correctly

3.2 Registering the callback function

Currently, we have a callback function, but it does nothing. It is never called.

In order for us to be able to view the page, we need to registar it with the http_server object.

So back where the http_server object was defined, we need to add the following:

```
httpServer.path_callback("/", &indexCallback);
```

Note: You need to add the callback **before** you start the server

CHAPTER 4

Creating a public directory

4.1 What is the point

The point of public directories is to hold your static infomation (such as .css, .js & .png). Everything in the defined folder is publicly avaliable for anyone to access, so make sure that you are careful with what you put in the public folder

4.2 How to registar the public folder

There is a requiremnet before you start this process, you have to create a folder and you have to know the path to it (absoulute or relative should work)

Place this line of code **before** you start the server

```
httpServer.middleware_public("public/", "/public");
```

What does this mean?

- The first argument (where I type *public/*) is the location of the folder (this path happens to be relative).
- The second argument (where i type */public*) is the path that prefixes the files in the folder.

CHAPTER 5

HTTP Server Functions

start (int port)

```
Starts the HTTP Server

Paramaters:
    - port - The port number

returns void
```

path_callback (std::string path, pointer callback)

```
Registars a path along with its callback function

Paramaters:
    - path - The path (on the web). Example is /index
    - callback - the function that will get called when the path is called

returns void
```

middleware_public (std::string folder, std::string path)

```
Registars the public folder

Everything inside this folder is avaliable to anyone on the web

Paramaters:
    - folder - The folder on the computer
    - path - the path that will enable people to access it


returns void
```

CHAPTER 6

Plain Text Responce

Note: This launches off from Creating Pages. All of the code should be run from inside a callback function.

Lets talk about the responce object. It's inside **every** callback function. It's job is very simple (in theroy), it handles the HTTP Responce.

This object is will be refered to as *res*

There are five parts of a responce:

1. HTTP Headers (content-type etc)
2. HTTP Responce Code (hopefully you always send 200)
3. Body Content (in this case its just plain text)
4. Sending the Request (Actually send the responce to the client)
5. Closing the Socket (Close the currently open socket after the responce has finished)

So with that out of the way, lets start coding!

The first line that i'm going to write is the actual message. But what about the headers? If you don't include the *Content-Type* header in your callback, it will automaticlly add the *plain/text* header.

```
res.write("Hello World");
```

This line of code has added *Hello World* to the write list. It hasn't atually been written to the client, but it is preped and ready to be sent.

Now we actually send the message. We need to include the HTTP Responce code. Our responce code is 200

```
res.write("Hello World");
res.send("200");
```

Yay the request is sent to the client!

But were not done yet, the socket is still open, potentially hindering the browser from actually displaying the message. So lets close it.

```
res.write("Hello World");
res.send("200");
res.end();
```

The request is now completely finished. The sockets have been closed.

Start the server and go to the path that your specifided when you registared the callback function. It should say:

```
Hello World!
```

CHAPTER 7

Responce Class

write (std::string append_message)

```
Add a message to the write stack

This is for text based messages (not binary etc)

Paramaters:
    - append_message - The message to write to the client

returns int
```

writeBinary (char *binaryData, int length)

```
Registars a path along with its callback function

Paramaters:
    - binaryData - The binary data. This needs to be a char *
    - length - the length of the binaryData

returns int
```

html (std::string location, HTMLContent content[], int arrayLength)

```
Serves a HTML page

Paramaters:
    - location - location inside the views/ folder
    - content[] - the array for subsitution
    - arrayLength - the length of the array

TODO:
    - The arrayLength paramater should be dropped

returns void
```

header(std::string type, std::string value)

```
Adds headers to the responce

Paramaters:
    - type - The header name
    - value - The value of the header

Example:
    - The function res.header("Content-Type", "plain/text");
      would add "Content-Type: plain/text" to the HTTP responce

returns int
```

contentType(std::string content_type)

```
Quick way of defining the ContentType header

Paramaters:
    - content_type - the value to assign to the ContentType Header

returns int
```

send(std::string http_code)

```
Sends the HTTP Responce.

After this function is called

Paramaters:
    - http_code - the HTTP code that will be sent with the responce

TODO:
    - change the http_code from a std::string to int

returns int
```

Note: This project will not compile on Windows. This is because it uses the unix socket system (there in no WinSock compatibility)
