

---

# **NodeC++**

***Release 0.0.1***

**Sep 27, 2019**



<b>1</b>	<b>Building NodeC++ with cmake</b>	<b>1</b>
1.1	Requirements . . . . .	1
1.2	Downloading . . . . .	1
1.3	Running cmake . . . . .	1
1.4	Footnotes . . . . .	2
<b>2</b>	<b>Starting a HTTP Server</b>	<b>3</b>
2.1	Initialize the class . . . . .	3
2.2	Start the server . . . . .	3
<b>3</b>	<b>Creating Pages &amp; Routes</b>	<b>5</b>
3.1	Creating the callback function . . . . .	5
3.2	Registering the callback function . . . . .	5
<b>4</b>	<b>Creating a public directory</b>	<b>7</b>
4.1	What is the point . . . . .	7
4.2	How to registrar the public folder . . . . .	7
<b>5</b>	<b>HTTP Server Functions</b>	<b>9</b>
<b>6</b>	<b>Plain Text Responce</b>	<b>11</b>
<b>7</b>	<b>Responce Class</b>	<b>13</b>



---

## Building NodeC++ with cmake

---

### 1.1 Requirements

- cmake (version 3.14 or above)
- a MacOS/Linux computer (NodeC++ doesn't compile on Windows)<sup>1</sup>

### 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 built NodeC++ using cmake!

---

<sup>1</sup> You may be able to build NodeC++ on Windows 10 using WSL (Windows Subsystem for Linux), but this is not tested.

## 1.4 Footnotes

---

## Starting a HTTP Server

---

### 2.1 Initialize 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 compile 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.



---

## 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, response 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 responseClass, i'll just go over the very important bits. The first line (where the function is declared), requires two parameters:

- a Request Struct
- a response class

**Warning:** Without the correct parameters the program will (most likely) 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 register 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

---

---

### Creating a public directory

---

#### 4.1 What is the point

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

#### 4.2 How to register the public folder

There is a requirement before you start this process, you have to create a folder and you have to know the path to it (absolute 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.



---

## HTTP Server Functions

---

### **start (int port)**

Starts the HTTP Server

**Parameters:**

- port - The port number

returns **void**

### **path\_callback (std::string path, pointer callback)**

Registers a path along with its callback function

**Parameters:**

- 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)**

Registers the public folder

Everything inside this folder is available to anyone on the web

**Parameters:**

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

returns **void**



## CHAPTER 6

---

### Plain Text Response

---

---

**Note:** This launches off from [Creating Pages](#). All of the code should be run from inside a callback function.

---

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

This object is will be referred to as *res*

There are five parts of a response:

1. HTTP Headers (content-type etc)
2. HTTP Response Code (hopefully you always send 200)
3. Body Content (in this case its just plain text)
4. Sending the Request (Actually send the response to the client)
5. Closing the Socket (Close the currently open socket after the response 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 automatically add the *plain/text* header.

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

This line of code has added *Hello World* to the write list. It hasn't actually 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 Response code. Our response 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!
```



---

## Response 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)**

Registers 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** substitution
- 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 response

**Parameters:**

- `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 response

returns `int`

### contentType(std::string content\_type)

Quick way of defining the ContentType header

**Parameters:**

- `content_type` - the value to assign to the ContentType Header

returns `int`

### send(std::string http\_code)

Sends the HTTP Response.

After this function is called

**Parameters:**

- `http_code` - the HTTP code that will be sent with the response

**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 is no WinSock compatibility)

---