C++ Micro Web Framework

Created by Tirthraj Mahajan

Introduction

This project is a micro web framework created in C++, utilizing the Windows Socket API (Winsock2) for network communication. Winsock2 provides the necessary functionality to enable networking capabilities in Windows applications, allowing the framework to handle incoming connections and HTTP requests.

About Sockets

Sockets are communication endpoints that allow different processes to communicate over a network.
In this project, I have used Winsock2 is used to create and manage sockets.

When a server socket is created, it listens for incoming connections on a specified port. When a client connects to this port, the server socket accepts the connection and creates a new socket dedicated to that client. This new socket is then used to send and receive data between the server and the client.

Windows Socket API (Winsock2)

The Windows Socket API (Winsock2) is a library that provides a set of functions for networking applications on Windows operating systems. It allows developers to create network-aware applications that can communicate over TCP/IP networks, such as the Internet.

Winsock2 abstracts the complexities of network communication and provides a standardized interface for creating and managing sockets, establishing connections, sending and receiving data, and handling network events.

How Winsock2 Communicates with the OS

Underneath the Winsock2 library, the operating system's network stack handles the actual transmission and reception of data. Winsock2 interacts with the operating system through system calls and low-level network APIs to perform various networking operations.

When an application using Winsock2 makes a network-related function call, such as creating a socket or sending data, Winsock2 translates these requests into system calls that the operating system understands. The operating system then performs the necessary operations, such as allocating resources, establishing connections, and transmitting data over the network.

Winsock2 also handles asynchronous events, such as incoming connections and data arrival, by using mechanisms like callback functions or polling. This allows applications to efficiently manage multiple network connections and handle network events in a non-blocking manner.

Overall, Winsock2 serves as a bridge between the application and the underlying operating system, enabling network communication in Windows applications while abstracting the complexities of network programming.

Requirements

To use this framework, you need the following:

- C++ 14 or later
- C++ Compiler (e.g., g++)
- Git (for cloning the repository)
- Windows Operating System
- sqlite3.dll file

Installation

Step 1: Create Project Directory

```
mkdir Project
cd Project
```

Step 2: Clone the Repository

```
git clone https://github.com/tirthraj07/CPP-Web-Server.git .
```

Step 3: Open the Project Directory in Terminal and Compile the Code

```
g++ -o demo demo.cpp Webserver/*.cpp sqlite3.dll -lws2_32 -I./Webserver
```

Step 4: Run the Code

```
./demo.exe
```

Features

- **Easily create a web server** by specifying IP address and Port
- **Simple file structure** (Similar to Flask)
- **Supports GET, POST, PUT, PATCH, and DELETE** requests.
- **Supports Query Parameters and Request Body Parameters**
- **Supports easy rendering of HTML pages** and linking them to CSS and JS
- **Supports serving static files** (Images, pdfs, etc.) easily
- You can **create routes** by linking them to functions (similar to Flask and Express)
- **Supports chaining multiple middleware functions** for request processing
- **Supports HTTP redirection** to different URLs
- **Supports SQLite Database Connectivity**

Documentation

1. Getting Started

To get started, your project needs to contain the following file structure:

```
Project
    demo.cpp (Server)
    sqlite3.dll
    WebServer
    templates
        index.html
        (other html files to be rendered)
```

```
    static
        css
            style.css
            (other css files to be linked)
        js
            script.js
            (other js files to be linked)
    public
        (images, pdfs, etc. to be rendered)
    database
        database.db
        (other database files)
```

2. Run the Server

To start the server, include the `server.h` file from `WebServer` folder. Specify the IP address and Port and then create an instance of the Webserver. Then run the server.

```
#include "WebServer/server.h"
int main(){
    // Declaring the PORT and IP Address
    const char* PORT = "5000";
    const char* IPAddr = "127.0.0.1";

    // Instantiate the Server
    WebServer server = WebServer(PORT, IPAddr);

   // Run the server
   server.run();
}
```

Output:

```
Server listening on http://127.0.0.1:5000
```

3. Render HTML CSS and JS to a particular route

Create an `index.html` page in the `templates` folder (add html content) and link to JavaScript file and CSS file which reside in `static/js` and `static/css` files respectively.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title></title>
<link rel="stylesheet" href="../static/css/style.css">
</head>
<body>
    <!-- Add HTML Content -->

    <script src="../static/js/script.js"></script>
</body>
</html>
```

Note: if you have an image in html, then the image must be in the `/public/` directory

```
<img src="../public/profile.jpeg" alt="Profile Picture">
```

Create a function with return type `Response` with a parameter of type `Request`. Create a response object and using the `render_template` function, render the `index.html`. Return the response object.

```
Response HomePage(Request& req){
    Response res;
    res.render_template("index.html");
    // Path to index.html relative to the 'templates' folder
    return res;
}
```

Create a route by linking it with the function using the server's `get` method.

```
server.get("/", &HomePage);
```

Run the server and visit `localhost:5000` to see the rendered `index.html`.

4. Redirect to a Page

To redirect to a page, you can use the `redirect()` method in the response object. The redirected URLs can be relative or absolute URL.

```
// Function that redirects to 'https://www.google.com'
Response redirectToGoogle(Request &req){
    Response res;
    res.redirect("https://www.google.com");
    return res;
}

server.get("/google",&redirectToGoogle);
```

5. Serve a Document

To serve a document (like images, pdfs, etc.), you can use the `serveFile()` function of the response object. It takes the document name as the first parameter and directory as the second parameter.

```
// Function that serves an image
// (Assume cppImage.png is in public directory)
Response serveImage(Request &req){
    Response res;
    res.serveFile("cppImage.png", "/public/");
    return res;
}

server.get("/cpp", &serveImage);
```

6. `setContent()`, `setContentType()`, `setStatusCode()` and `getRequestQuery()`

You can get the request query parameters using the `getRequestQuery()` method of the request object. It returns an `std::unordered_map` of type `<std::string, std::string>` which can be used to get the parameters in constant time.

You can also set the content, its type, and status code using the `setContent()`, `setContentType()`, `setStatusCode()` function of the response object.

Here's a simple example of a GET Request. This function searches for the `search` parameter in the URL. If not found, then all the links are returned in `application/json` format. Else it returns the specified content.

```
Response GETRequestAPI(Request& req){
    std::unordered_map<std::string,std::string> queryParams =
    req.getRequestQuery();

    Response res;
    res.setContentType("application/json");
    std::string jsonContent;

    auto it = queryParams.find("search");

    if(it == queryParams.end()){

    jsonContent = R"({
"linkedin":"https://www.linkedin.com/in/tirthraj-mahajan/",
"github":"https://github.com/tirthraj07",
"instagram":"https://www.instagram.com/tirthraj07/"
    })";

        res.setContent(jsonContent);
        res.setStatusCode(200);
    }
    else{
        // Check for specific query parameter values
        // Set appropriate content and status code

        if(it->second == "linkedin"){
            jsonContent = R"({
"linkedin":"https://www.linkedin.com/in/tirthraj-mahajan/
             "})";

            res.setContent(jsonContent);
            res.setStatusCode(200);
        }
        // .. add more
        else{
            jsonContent = R"({"error":"Not Found"})";
            res.setContent(jsonContent);
            res.setStatusCode(404);
        }
    }

    return res;
}

server.get("/api/social-media", &GETRequestAPI);
```

7. HTTP Requests

All HTTP Requests like GET, POST, PUT, PATCH, and DELETE can be handled using the `get()`, `post()`, `put()`, `patch()`, and `del()` function of the `WebServer` instance.

```
server.get("/home", &HomePage);
server.post("/form", &HandleForm);
// etc.
```

8. `getRequestBody()`

For POST, PUT, PATCH, and DELETE requests, you can use the `getRequestBody()` function of the Request object. It returns an `std::unordered_map` of type `<std::string, std::string>` to get the request body parameters in constant time.

Here is a simple example:

```
// Function that handles the '/api/form' route
Response POSTRequestAPI(Request& req){
    std::unordered_map<std::string, std::string> requestBody =
 req.getRequestBody();

    // Process request body parameters
    // Set appropriate content and status code

    std::string name = requestBody["name"];
    std::string email = requestBody["email"];

    Response res;
    res.setContentType("application/json");

    if(name == "" || email == ""){

    res.setContent(R"({"status":
                  "error: incomplete credentials"})");

        res.setStatusCode(400);
        return res;
    }

    std::cout<<"Entered name: "<<name<<std::endl;
    std::cout<<"Entered email: "<<email<<std::endl;
    res.setContent(R"({"status":"success"})");
```

```
    res.setStatusCode(201);

    return res;
}

server.post("/api/form", &POSTRequestAPI);
```

9. Add Middleware Function

The Middleware class manages a list of middleware functions that process HTTP requests. This class is designed to allow chaining multiple middleware functions that each take a Request object as a parameter and return a Response object. If a middleware function returns a Response object different from the predefined `next()` object, the execution stops and that Response object is returned. Otherwise, it proceeds to the next middleware function.

Create a Response function which will be used for middleware for a particular route

Then create a list of middleware functions using the `Middleware` class and chain the function using the `push()` method of the Middleware instance

Link the route to the Response Function and Middleware list using the overloaded `get` (or equivalent http request type) function

Important Note: to proceed to the next middleware, the function must return `Middleware::next()`

For example:

```
// First Middleware for '/treasure' route
Response middlewareFunctionForTreasurePage(Request& req){
    // Process request parameters
    // Return appropriate response or proceed to the next middleware

    std::unordered_map<std::string, std::string> queryParams =
 req.getRequestQuery();

    std::string treasureKey = queryParams["key"];
    if(treasureKey == "123"){
        return Middleware::next();  // Proceed to the next middleware
    }
```

```
    Response res;
    res.setContentType("application/json");
    res.setContent(R"({"error":"invalid key", "hint":"key=123"})");
    return res;
// Stop execution of the route processing and return this response
}

// Second Middleware for '/treasure' route
Response anotherMiddlewareFunctionForTreasurePage(Request& req){
    // Process request parameters
    // Return appropriate response or proceed to the next middleware
    std::cout<<"Someone is accessing treasure 0_0"<<std::endl;
    return Middleware::next();
}

// Function to handle the '/treasure' route
Response loadTreasurePage(Request& req){
    // Process request parameters
    // Return appropriate response
    Response res;
    res.render_template("treasure.html");
    return res;
}

// Create a middleware list for the '/treasure' route
Middleware treasureRouteMiddleware;
treasureRouteMiddleware.push(middlewareFunctionForTreasurePage);
treasureRouteMiddleware.push(anotherMiddlewareFunctionForTreasurePage);

// Link the route with function and middleware list
server.get("/treasure", &loadTreasurePage, treasureRouteMiddleware);
```

10. Add SQLite database

The `Webserver` libaray supports **SQLite** database by using `sqlite3.h`. Include the `database.h` file from `WebServer` and create an instance of the `SQLiteDatabase` class by specifying the database file name inside `database` folder

```
#include <WebServer/database.h>

SqliteDatabase database("database.db");
```

The `SqliteDatabase` class provides four main methods: `executeQuery()` , `executeParameterizedQuery()` , `executeSelectQuery` and `databaseError()`

CREATE and DELETE Operations

These operations can be performed using `executeQuery()` function. The `executeQuery()` function returns a type `bool` determining the `success` of the operation The parameter is `query` which is type `std::string`

RETURN VALUE -> `True` is query is successful | `False` if the query is unsuccessful

Example of **CREATE** operation:

```
// Initializes the database
#include <WebServer/database.h>

SqliteDatabase database("database.db");

bool InitDatabase(){
    std::string query = "CREATE TABLE IF NOT EXISTS users (" \
                        "NAME TEXT NOT NULL," \
                        "EMAIL TEXT NOT NULL PRIMARY KEY" \
                        ");";

    bool success = database.executeQuery(query);
    if(success == false){
        std::cerr << "Database Initialization Failed" << std::endl;
        return false;
    }

    std::cerr << "Database Initialization Success" << std::endl;
    return true;

}

// Initialize the database and run the server
if(InitDatabase()){
    // Run the server
    server.run();
};
```

`databaseError()` Method of the `SqliteDatabase` Class

The `databaseError()` function returns the most recent error which occured associated to the database connection

For example: We can modify the existing initDatabase() function to log the error message if the database failed to initialize.

```
bool InitDatabase(){
    std::string query = "CREATE TABLE IF NOT EXISTS users (" \
                        "NAME TEXT NOT NULL," \
                        "EMAIL TEXT NOT NULL PRIMARY KEY" \
                        ");";

    bool success = database.executeQuery(query);
    if(success == false){
        std::cerr << "Database Initialization Failed" <<
database.databaseError() << std::endl;
        return false;
    }

    std::cerr << "Database Initialization Success" << std::endl;
    return true;

}
```

INSERT Operation

The **INSERT** operation can be performed using the `executeParameterizedQuery()`. The parameters: `std::string query` `std::vector < SqliteDatabase::SqlParam> params`

RETURN VALUE : The return value if type `bool` indicating the success of the operation. `True` if insertion is successful | `False` if insertion is unsuccessful

For Example:

```
Response POSTRequestAPI(Request& req){

    std::unordered_map<std::string, std::string> requestBody =
 req.getRequestBody();

    std::string name = requestBody["name"];
    std::string email = requestBody["email"];
```

```
    // Do validation of the query parameters

    Response res;
    res.setContentType("application/json");

    std::vector<SqliteDatabase::SqlParam> params;
    params.emplace_back(name);
    params.emplace_back(email);

    bool success = database.executeParameterizedQuery
("INSERT INTO users (NAME, EMAIL) VALUES (?, ?)", params);

    if(success){
        res.setContent(R"({"status":"success"})");
        res.setStatusCode(201);
    }
    else {
        std::string jsonErrorMessage = R"({"status":")" +
 database.databaseError() + R"("})";

        res.setContent(jsonErrorMessage);
        res.setStatusCode(400);
    }
    return res;
}
```

SELECT Operation

The **SELECT** operation can be perfromed using the `executeSelectQuery()` operation. The input parameter is `query` of type `std::string`

RETURN VALUE : The return value is of type `std::vector<std::vector<std::string>>`.

For example:

```
std::vector<std::vector<std::string>> result =
 database.executeSelectQuery("SELECT * FROM users;");

for(int i=0; i<result.size(); i++){
    for(int j=0; j<result[0].size(); j++){
        std::cout<< result[i][j] << " ";
    }
    std::cout<<std::endl;
```

```
}
```

Library Documentation

SERVER.H

This header file contains the `WebServer` class, which is the core of the framework.

WebServer Class

The WebServer class provides functionality to create, configure, and run a basic HTTP web server. It listens for incoming connections, accepts client requests, and serves static files and dynamic content based on the requested routes.

The server supports GET requests for serving static files such as CSS, JavaScript, and other resources stored in predefined directories. Additionally, it allows users to register custom response functions for specific routes, enabling dynamic content generation.

```
class WebServer {
private:
    WORD wVersionRequested; ///< Winsock version requested
    WSADATA wsaData;        ///< Winsock data structure
    int iResult;            ///< Winsock operation result

    SOCKET serverSocket;

    struct addrinfo* result = NULL; ///< Address information
    struct addrinfo hints;
    ///< Address hints for socket configuration

    const char* IPAddr;         ///< IP address for the server
    const char* PORT;           ///< Port number for the server
```

```
    SOCKET clientSocket;  ///< Socket for communicating with client

    AVLTree GetRouteTree; ///< AVL tree to store GET routes

    AVLTree PostRouteTree; ///< AVL tree to store POST routes

    AVLTree PutRouteTree; ///< AVL tree to store PUT routes

    AVLTree PatchRouteTree; ///< AVL tree to store PATCH routes

    AVLTree DeleteRouteTree; ///< AVL tree to store DELETE routes

    std::string cssDirectory = "/static/css/";
///< Directory for serving CSS files

    std::string jsDirectory = "/static/js/";
///< Directory for serving JavaScript files

    std::string publicDirectory = "/public/";
///< Directory for serving other public files

    int initializeWinsock();
    int createUnboundedSocket();
    int setupAddressInfo();
    int bindSocketToAddress();
    int listenForConnections();
    int acceptConnectionRequest();
    int handleClientRequest();
    std::string searchGETTree(Request& requestObject);
    std::string searchPOSTTree(Request& requestObject);
    std::string searchPUTTree(Request& requestObject);
    std::string searchPATCHTree(Request& requestObject);
    std::string searchDELETETree(Request& requestObject);

    bool startsWith(const std::string& str,
const std::string& prefix);

std::string getRemainingPath(const std::string& str,
const std::string& prefix);

    std::string serveCSSFile(std::string cssFilePath);
    std::string serveJSFile(std::string jsFilePath);
    std::string servePublicFile(std::string publicFilePath);

public:
    WebServer(const char* PORT, const char* IPAddr);
    ~WebServer();

    int run();
```

```
    void get(std::string route,
Response (*responseFunction)(Request&));

    void get(std::string route,
Response (*responseFunction)(Request &), Middleware &middleware);

    void post(std::string route,
Response (*responseFunction)(Request&));

    void post(std::string route,
Response (*responseFunction)(Request &), Middleware &middleware);

    void put(std::string route,
Response (*responseFunction)(Request&));

    void put(std::string route,
Response (*responseFunction)(Request &), Middleware &middleware);

    void patch(std::string route,
Response (*responseFunction)(Request&));

    void patch(std::string route,
Response (*responseFunction)(Request &), Middleware &middleware);

    void del(std::string route,
Response (*responseFunction)(Request&));

    void del(std::string route,
Response (*responseFunction)(Request &), Middleware &middleware);

};
```

Class Members

Private Members

- Winsock Configuration:
  - `WORD wVersionRequested;` - Winsock version requested.
  - `WSADATA wsaData;` - Winsock data structure.
  - `int iResult;` - Winsock operation result.
- Sockets:

- `SOCKET serverSocket;` - Server socket for listening to incoming connections.
- `SOCKET clientSocket;` - Socket for communicating with client.

- Address Information:
  - `struct addrinfo* result = NULL;` - Address information.
  - `struct addrinfo hints;` - Address hints for socket configuration.
  - `const char* IPAddr;` - IP address for the server.
  - `const char* PORT;` - Port number for the server.

- Routing Trees:
  - `AVLTree GetRouteTree;` - AVL tree to store GET routes and associated response functions.
  - `AVLTree PostRouteTree;` - AVL tree to store POST routes and associated response functions.
  - `AVLTree PutRouteTree;` - AVL tree to store PUT routes and associated response functions.
  - `AVLTree PatchRouteTree;` - AVL tree to store PATCH routes and associated response functions.
  - `AVLTree DeleteRouteTree;` - AVL tree to store DELETE routes and associated response functions.

- Directories:
  - `std::string cssDirectory = "/static/css/";` - Directory for serving CSS files.
  - `std::string jsDirectory = "/static/js/";` - Directory for serving JavaScript files.
  - `std::string publicDirectory = "/public/";` - Directory for serving other public files.

Private Methods

- Winsock Initialization:
  - `int initializeWinsock();`
  - `int createUnboundedSocket();`
  - `int setupAddressInfo();`
  - `int bindSocketToAddress();`
  - `int listenForConnections();`
  - `int acceptConnectionRequest();`

- Request Handling:
  - `int handleClientRequest();`
  - `std::string searchGETTree(Request& requestObject);`
  - `std::string searchPOSTTree(Request& requestObject);`
  - `std::string searchPUTTree(Request& requestObject);`
  - `std::string searchPATCHTree(Request& requestObject);`
  - `std::string searchDELETETree(Request& requestObject);`

- Helper Functions:
  - `bool startsWith(const std::string& str, const std::string& prefix);`

- `std::string getRemainingPath(const std::string& str, const std::string& prefix);`
- `std::string serveCSSFile(std::string cssFilePath);`
- `std::string serveJSFile(std::string jsFilePath);`
- `std::string servePublicFile(std::string publicFilePath);`

Public Methods

- Constructor and Destructor:
  - `WebServer(const char* PORT, const char* IPAddr);`
  - `~WebServer();`
- Server Operations:
  - `int run();`
- Route Handling:
  - `void get(std::string route, Response (*responseFunction)(Request&));`
  - `void get(std::string route, Response (*responseFunction)(Request &), Middleware &middleware);`
  - `void post(std::string route, Response (*responseFunction)(Request&));`
  - `void post(std::string route, Response (*responseFunction)(Request &), Middleware &middleware);`
  - `void put(std::string route, Response (*responseFunction)(Request&));`
  - `void put(std::string route, Response (*responseFunction)(Request &), Middleware &middleware);`
  - `void patch(std::string route, Response (*responseFunction)(Request&));`
  - `void patch(std::string route, Response (*responseFunction)(Request &), Middleware &middleware);`
  - `void del(std::string route, Response (*responseFunction)(Request&));`
  - `void del(std::string route, Response (*responseFunction)(Request &), Middleware &middleware);`

Expanded Section: Sockets

Following is the general initialization steps to setup sockets

1. Creating a Socket:
   - A socket is created using the `socket()` function, which specifies the communication domain, type, and protocol.

2. **Binding a Socket:**
    - The socket is bound to an IP address and port using the `bind()` function. This makes the socket a "server socket" that listens for incoming connections.

3. **Listening for Connections:**
    - The `listen()` function puts the socket in a state where it can listen for incoming connection requests from clients.

4. **Accepting Connections:**
    - When a client tries to connect, the server accepts the connection using the `accept()` function, creating a new socket for communication with the client.

5. **Data Transmission:**
    - Data can be sent and received using the `send()` and `recv()` functions, respectively.

6. **Closing the Socket:**
    - Once communication is done, the socket is closed using the `closesocket()` function.