Part Two the Web Server

In this exercise we will be building a simple web server to handle serving both static, and dynamic web pages.

To start let's talk about what a dynamic web page is in contrast to a static web page. You may have heard the analogy that javascript is like the muscles of a web page, this analogy is made because javascript is used to make web pages change over time, and based on user interaction, and it may be tempting to call a website with a lot of javascript like https://sprite-creator.com/web dynamic, and in many ways it is a very dynamic site, but it is not dynamically generated. What I mean by this is when you go to the site, I go to the site, and our friend goes to the site we all start off seeing the same thing. It's not like going to Gmail to check your email where we each would see a different list of emails. This happens because the web server we connect to in order to get our email from gmail is dynamically created the web page based on what email we have logged into their site with. The Gmail web server knows that when I access my email I should only get the emails their systems have received for my email, and not the ones your email received.

We will be using a library made by Yuji Hirose called httplib which will help us to both serve the static web pages requested by the user as well as to help use create our own dynamically generated pages. This library has been chosen for it's simplicity, as well as it's syntactical similarities to popular web development frameworks like express.js which is a very powerful, and fast way to build out web servers.

Now to begin we will need to clone the following repl:
https://replit.com/@AndrewRubinstei/CSC211FinalProject-Start-1#main.cpp

This will start us out with the http lib already downloaded, a make file to help us build our project, a web directory where we will store all of our static assets like HTML, CSS, and Javascript. In other words the things that we will not be using C++ to generate dynamically. It also has a couple of helper functions we can use, namely

readFile, which is a function that takes a string that is a file path, and reads the file at that path into a string it returns for us.

So if we wrote the code

```
  cout << readFile("main.cpp") << "\n";
```

We would be reading our main.cpp file, and then giving it to cout to print to the console.

The next is named now, and when we call it the function will give us the current time in microseconds since 1970, we can use this later to seed rand if we need a random number.

The next thing you will notice is some code has already been populated in the main:

```
httplib::Server svr;
auto ret = svr.set_mount_point("/", "./web");
cout << readFile("main.cpp") << "\n";
cout << "start server..." << endl;
svr.listen("0.0.0.0", 8080);
```

The first line is creating a web server object using the httplib written by Yuji Hirose.

The second is setting where the static files can be found for the server to serve. It has two parameters, the first sets the path these static files will be available on the site from. Currently it is "/" which indicates that these files should be available from the root of the site, or continuing with the example above that they should be accessible from http://sprite-creator.com/<filename> if our domain was sprite-creator.com

The second parameter defines the folder on the web server where these files exists. If you leave it as "./" then it will make all the files in the directory where the executable is, and all it's subdirectories, but we don't want to expose our web server code to the client, so we should set it to only access a specific subdirectory which we have named "web", so in the code we should specify that like so:

```
auto ret = svr.set_mount_point("/", "./web");
```

The next two lines are responsible for letting the user who starts our web server that it has successfully started, and to tell the http server library to listen on a specific port for incoming connections

```
cout << "start server..." << endl;
svr.listen("0.0.0.0", 8080);
```

To quickly cover the two parameters here the first is the ip address we will accept incoming connections from "0.0.0.0" is all addresses, and the second is the port number we will listen to incoming connections on.

The IP address identifies a computer on a network, but one computer can have many simultaneous connections, so we need to specify port that we are listening for incoming connections on.

You can think of the IP identifying the computer, and the port identifying the specific application, in our case we will be listening on port 8080 for new incoming connections, this means when we send information to the client from the server it will be to their port 8080, and only the application that made the connection will have access to that port on the client, so we can send information back using the http lib and know the appropriate client will receive it, in the appropriate client application.

All of this has been done simply to setup a static web server that will serve any files stored in the web directory, so now if we look in our web directory, in our index.html we should see an HTML file which has the following form which we analyzed in the last part:

```
<form action="/option" method="GET">
  <label for="Company">Company:</label><br>
  <input type="text" id="Company" name="Company" value="">
  <input type="submit" value="Submit">
</form>
```

If you recall this form will send a get request to the route “action” on our web server, and wait for the web server to send back a new dynamically generated page, but we don’t have an option path setup on our web server, so let’s do that.

To begin we need to learn bait about how this library works.

```
  svr.Get("/option", [](const auto &req, auto &res) {
  });
```

The above line of code defines a route for option when a get request is made. What this means is when a get request is made to the path option like our above form will do upon submission this function will be called.

there are three important portions here:
First we call the Get method on our http server. This is to say I want to create a route for a get request
Second we have the first parameter the string “/option” this defines what path must be requested to execute this function, or in the form what action.

The last part actually can be broken down further and that is a callback function, or a function that will be called when a certain event has occurred. It takes two parameters, a request which is the data sent from the client, and a response, which we will use to send back a dynamically created page to the client.

Let’s start simply
To begin let’s just use the readFile function to read in a file, and send it to the client requesting option. To do this first we must read a file into a string variable, let’s use the main again

```
    string responseHTML = readFile("main.cpp");
    res.set_content(responseHTML, “text/html");
```

The above code first loads main, and saves it to a string, then we use the response object’s set_content method to set the string of data we want to return in this case the code we read in for our main.cpp, and also the format of the string or mime type, this could also be text/json for instance if we wanted to send off json data

Currently your main should look like this, and when you press submit on the initial web page you should see something like this:

Congratulations you have used C++ to create a web page dynamically, but this is a little oversimplified, and doesn't exemplify how we can use C++ to generate a different page each time.

So instead why don't we try to generate a random number each time we request the option path, format that in some HTML, and respond with that instead of our main.cpp

```
      string responseHTML = "<h2>" + to_string(rand()) + "</
h2>";
```

We can create our response as above, and send that to the client like before.

At this point your main should look like this:

```
int main()
{
  srand(now());
  httplib::Server svr;
  auto ret = svr.set_mount_point("/", "./web");
    svr.Get("/option", [](const auto &req, auto &res) {
    string responseHTML = "<h2>" + to_string(rand()) + "</h2>";
    res.set_content(responseHTML, "text/html");
  });
  cout << "start server..." << endl;
  svr.listen("0.0.0.0", 8080);
}
```

But what about the form data?

There is form data being sent to the web server from the form in the index file we created earlier, however we are not using it for anything, in order to create truly useful dynamic sites using the input the user provides becomes very important.
Think about the Gmail example, if it didn't use the username, and password you input how would it know what emails to show you?

Now we will cover how to actually access the data that has been sent to us from the user, and use it to change what we send back, initially let's just add the data the user sends to us to the HTML response we send back to the client.

To access this data first we must ensure it is present which we must query the request object for like so:

```
if (req.has_param("Company")) {
      string companyName = req.get_param_value("Company");
      cout<<companyName<<"\n";
    }
```

The above example code checks to see if the request has the parameter named "Company", and if it does it simply prints out the company name sent to the console like any C++ program you've worked with before. The only big difference is instead of reading this value in from the console we got it from the internet.

Now let's use that company name to generate some HTML to send back, and add it to what we have already been sending

```
  svr.Get("/option", [](const auto &req, auto &res) {
    string responseHTML = "<h2>" + to_string(rand()) + "</h2>";
    if (req.has_param("Company")) {
      string companyName = req.get_param_value("Company");
      responseHTML += "<h3>" + companyName + "</h3>";
      res.set_content(responseHTML, "text/html");
    }
    else
        res.set_content(responseHTML, "text/html");
  });
```

Let's change our code to look like the code above, and analyze what's going on here.

The first line should be familiar we are just building some HTML for our random number, and saving it to a string.

Next we check to see if the request has the Company name parameter present, and if it does it read that into the string companyName on this line:

```
    string companyName = req.get_param_value(“Company");
```

Then on the next line we concatenate the HTML with the random number we have already created with a new HTML string we are creating to store the company name, and finally sending that all back, but if there is no Company param we simply send back the unaltered randomly generated number.

Congratulations you have created your first dynamic web page that takes input from the user. This can be extended in all sorts of ways from using the input to read from a database of emails for a given account, and generating a list of emails to be sent back to connecting two players online to play a chess match against each other, or even more complex games like League, or Dota.