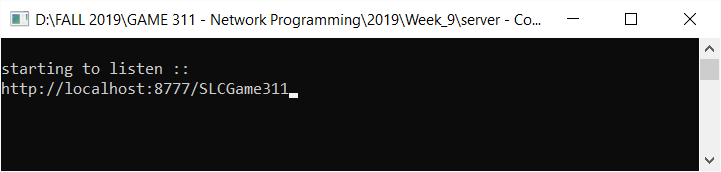
Walkthrough 6: C++ Web Server

**Part 1: Server Setup / Testing**

1. Download the base project from blackboard
2. Open up server.sln.
3. Open up the main.cpp file and click the Run button.

You should see something like the following output:



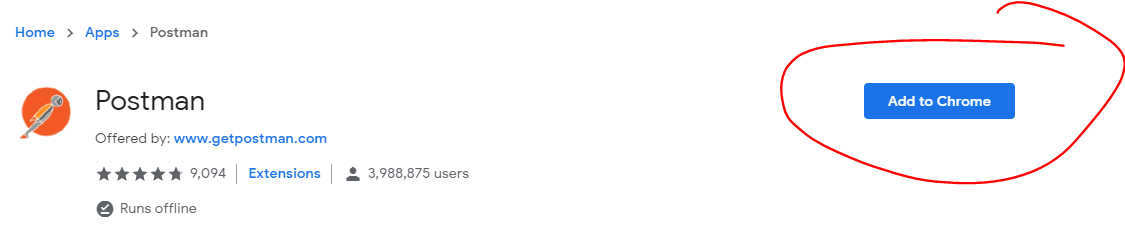
This is your server waiting to receive a request from a client.

1. To simulate a client connecting open up your browser and type in localhost:8777/SLCGame311.
2. Did anything happen on your server?

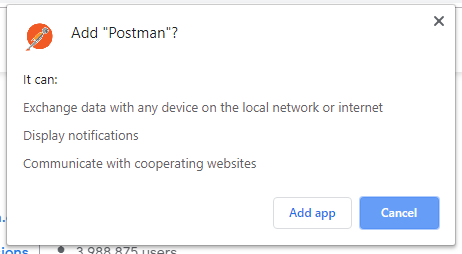
We will now install a chrome app called postman which can be used to emulate a clients request and format header and body data.

1. Open up google chrome and search for “postman chrome app”.

Click the first link and click the blue add to Chrome button on the top right.



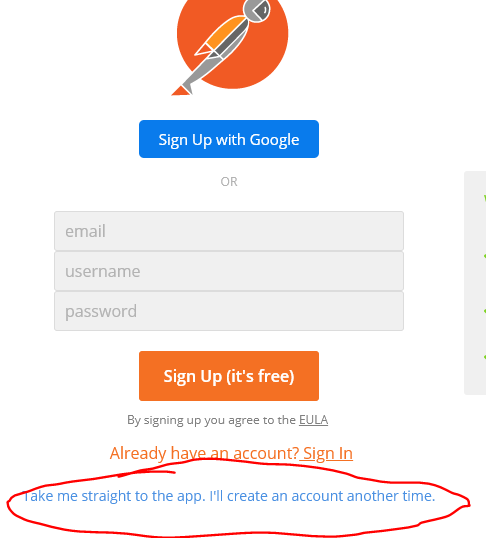
Click Add App.



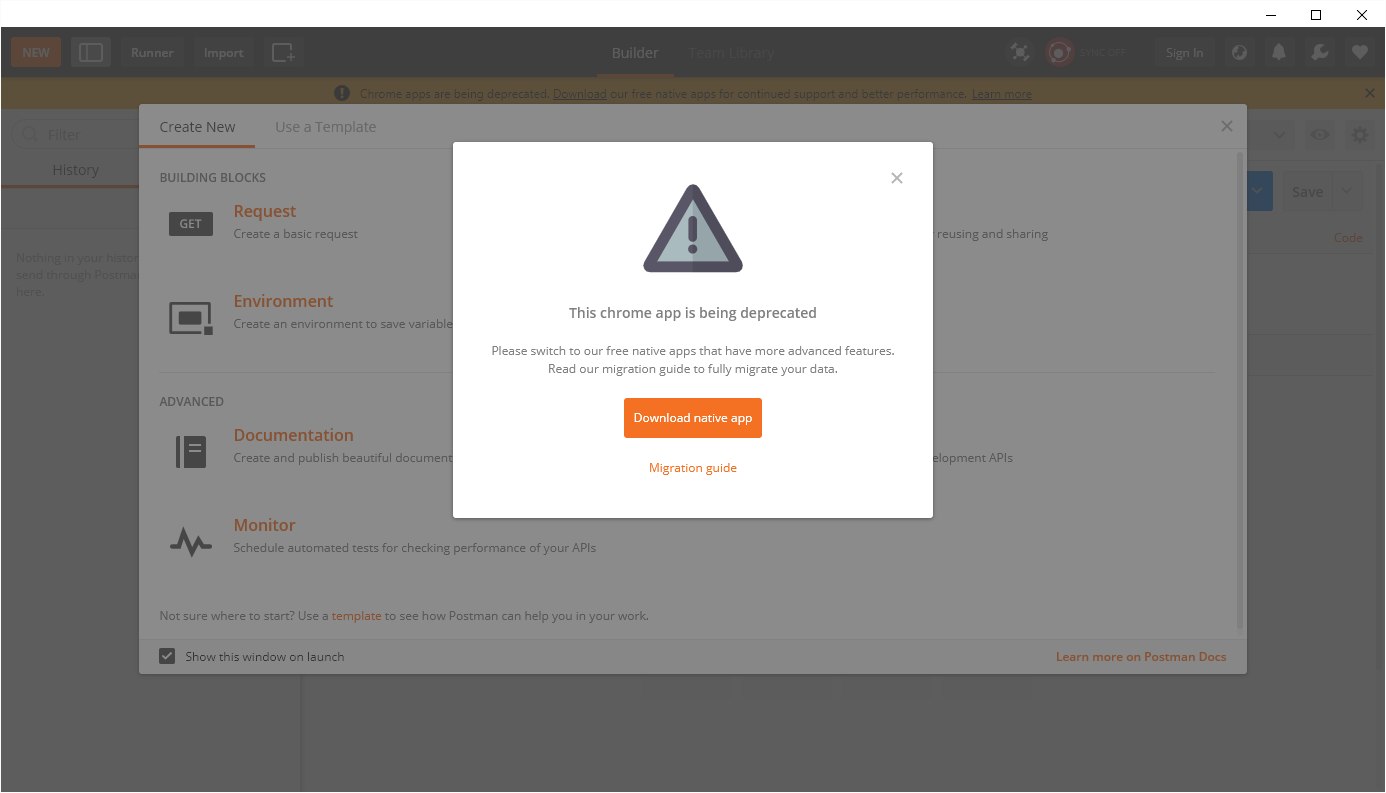
In the list of icons returned, click on the Postman Logo:



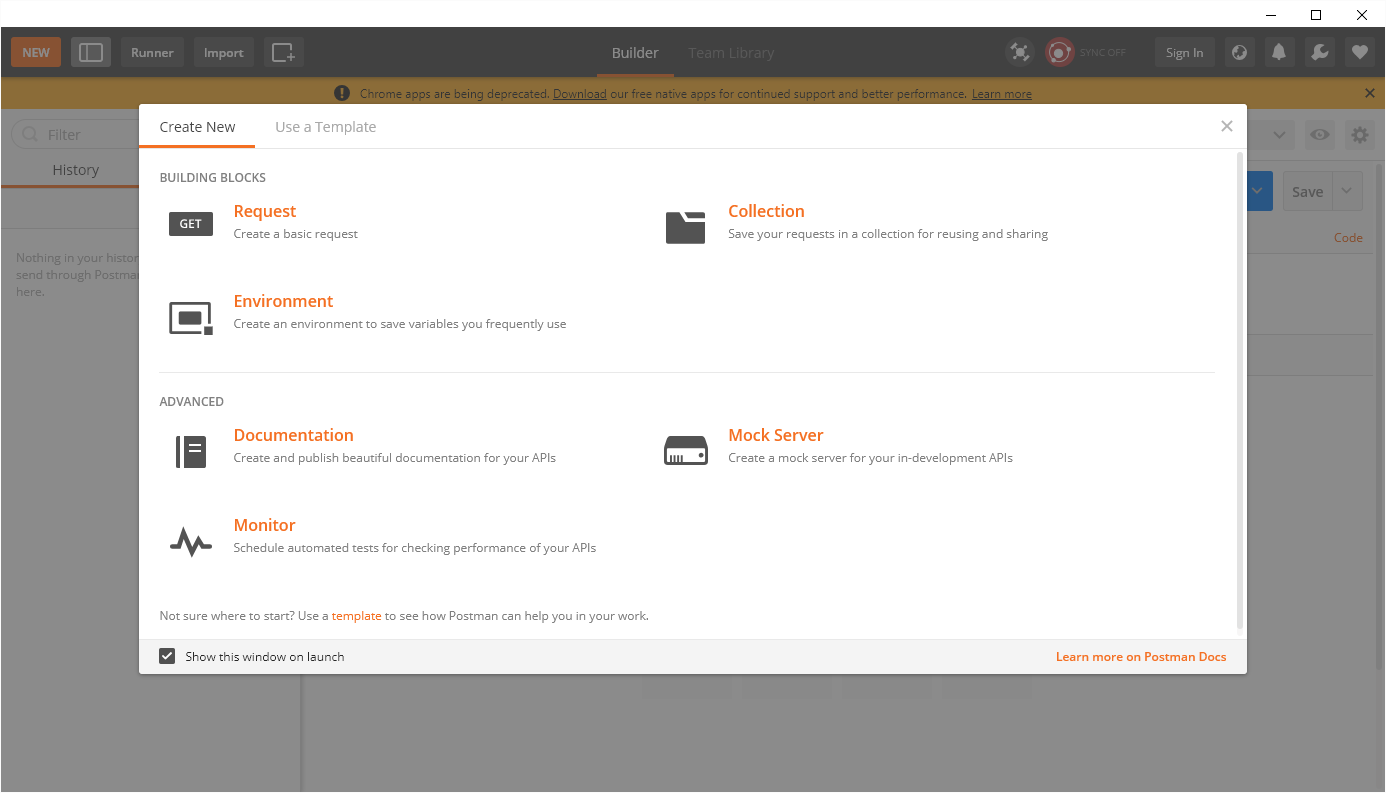
You can bypass the login by clicking the blue link at the bottom:



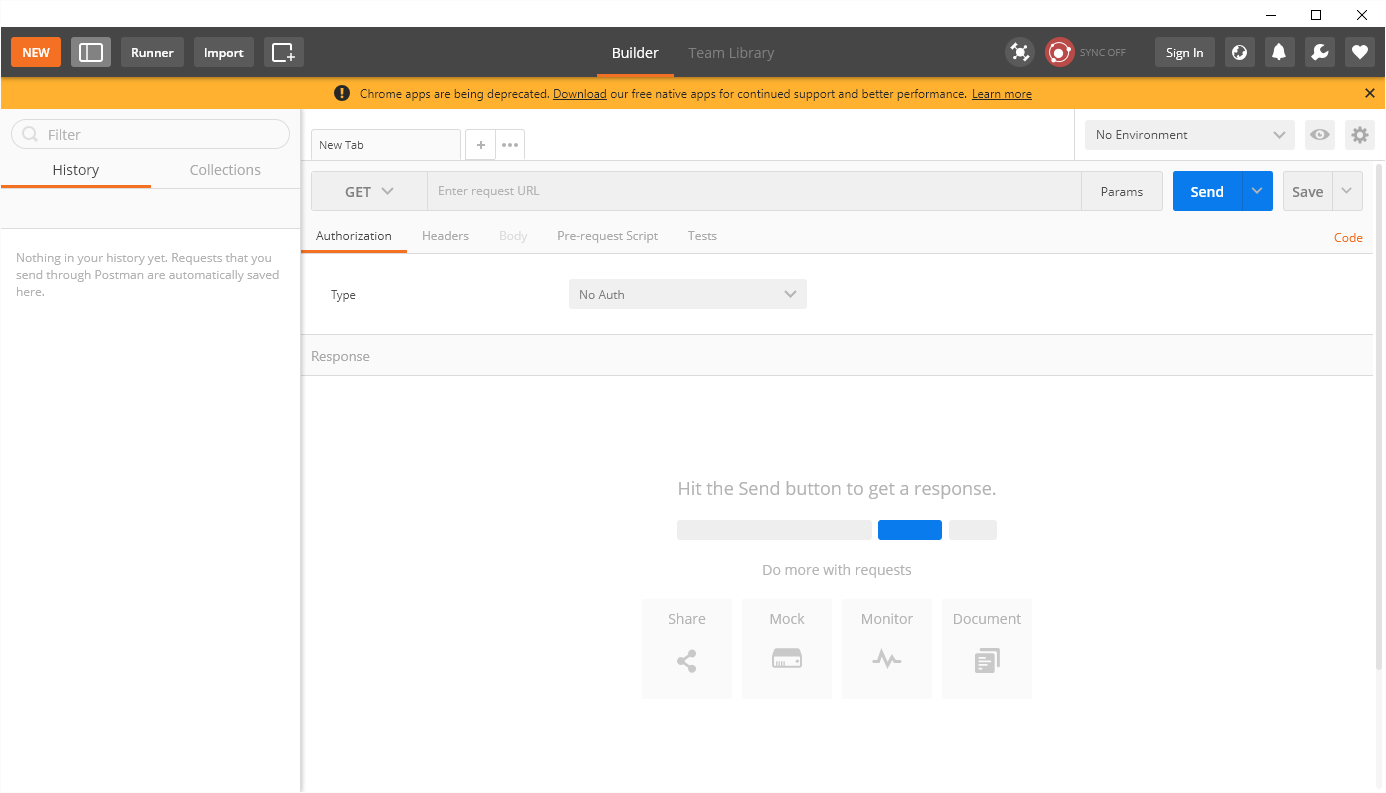
Ignore the following warning about chrome apps being deprecated by clicking the top right X:



Click X on the following popup as well.

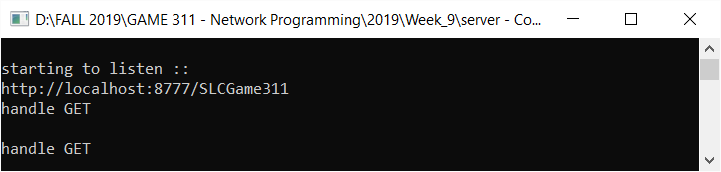


You should now be presented with the main POSTMAN interface:



1. Let’s do a quick test from Postman and type localhost:8777/SLCGame311 beside the GET button and hit send.

**You should see output on the server that says: “handle GET”,**

you will not get a response back in postman so go ahead and cancel 

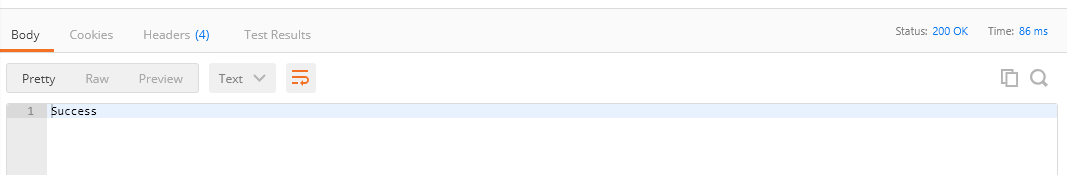
**Part 2: Basic Get Request**

1. Go back to your main.cpp of your solution, stop the server from running and navigate to the handle\_get function.
2. add the following to the bottom of the function:



1. Re-run the server and hit send again in postman:

**You should see the following:**

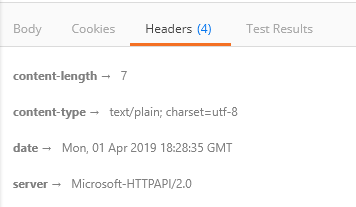


The body tab is selected and shows out success message we replied with.

The top right shows 200 ok which is our status code we replied with.

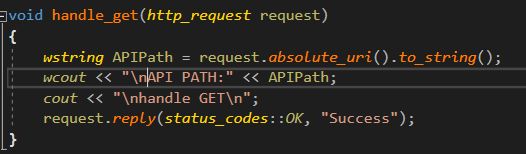
You’ll also notice the header tab shows 4 headers.

Click the tab to review:



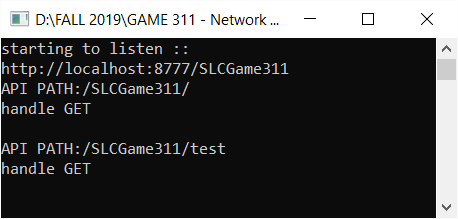
When we reply from a request using the Casablanca SDK, it automatically adds these headers for us.

1. back in our code stop our debugger, after the handle GET is printed, we need to determine what the client has requested for an API endpoint. To do this we will first print out the request objects URI:

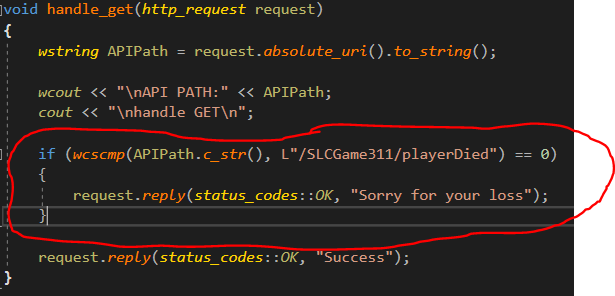


1. Run again, and hit send once more in postman. Note the results of the command line for the server.
   * Try changing postman to add an API endpoint like <http://localhost:8777/restdemo/playerDied> and click send.

Your output should like something like:



1. Kill the server. We will now parse our request path for specific API requests to handle them individually. To compare two wstrings we can use the **wcscmp** function which returns 0 if the wstrings match.
   * **The L”” indicates that this is a wide chararacter string**

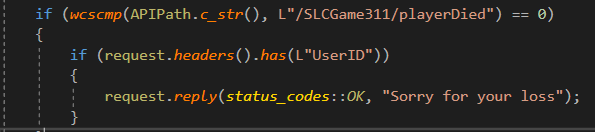


If you run in postman again adding playerDied to your path, you should get your sorry message instead of the success.

**Part 3: Headers and JSON Response**

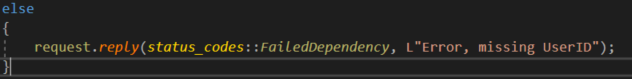
The playerDied request from our lab/walkthrough 4 attached a header to our request, so we will need to check for that on the server.

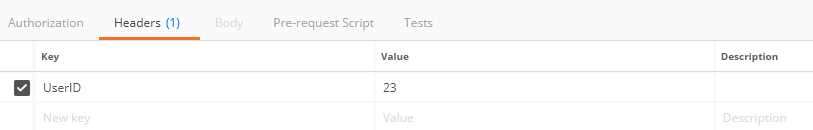
1. To check for a header, use the request.headers().has function to wrap our reply:



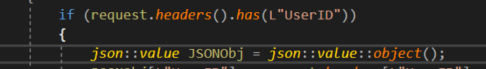
If we run our code and go back to postman now and hit send our call will reply success.

This is because we get into the API endpoint, however we don’t get into the if statement, so we never reply…

1. We need to add an else case that informs the client of why the request is not valid: 
2. If we go back to postman we can attempt the request with the UserID header added as well:



Lastly, our game client is expecting JSON data returned to be deserialized. Right now we are just returning a string. Remove our line of code to reply with “Sorry for your Loss” and replace it with the following within our existing header if statements body to create a JSON object:



1. We can retrieve the UserID to send back to the client from the header. To begin with we will need to get the http\_headers:



This object contains all headers attached to the request.

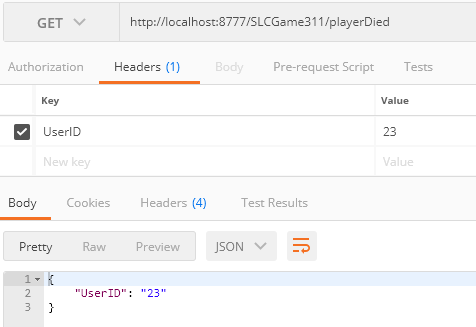
1. We can then assign to the JSON object using a key as index and the header value as our return value. We can use json::value::string to convert our header value(wstring) into a json value.



1. Last step is to return our json object in the response instead of the string:



1. Run the server and test out the request, your results should look like this:



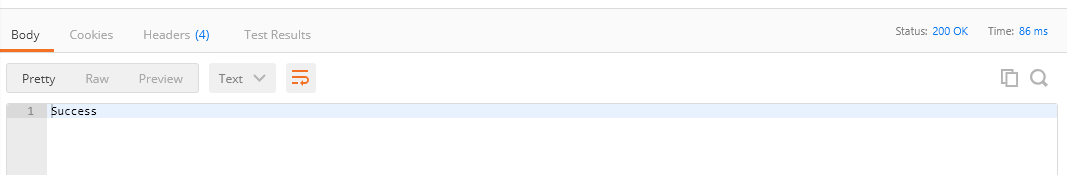
**Part 4: Basic POST Request**

1. Go back to your main.cpp of your solution, stop the server from running and navigate to the handle\_post function.
2. Add the following to the bottom of the function:



1. Re-run the server and change your request from GET to POST in postman and hit send.

**You should see the following:**

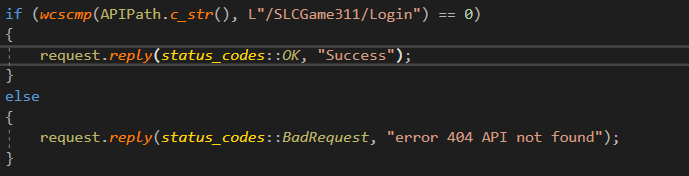


You’ll notice the post request is handled identically to our GET request from the previous portion of the walkthrough.

1. Back in our code stop our debugger, after the handle POST is printed, we need to determine what the client has requested for an API endpoint. To do this we will first print out the request objects URI**:** 
   * **( you can copy paste this from the GET tutorial as a base they are identical)**



1. We will now parse our request path for specific API requests to handle them individually. To compare two wstrings we can use the **wcscmp** function which returns 0 if the wstrings match.
   * **Again, you can copy paste the comparison code from the GET and change the path compared to Login.**
   * **Change the response if not our Login to be a BadRequest status code and an error string**



If you run in postman again, you should get your **error message** instead of the success from postman.

If you change the uri to add /Login you should see **Success.**

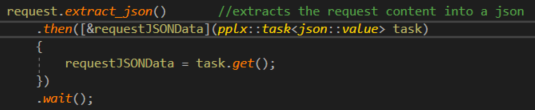
**Part 5: JSON Request / JSON Response**

The Login request from our lab/walkthrough 4 attached a JSON body to our request, so we will need to parse that data on the server.

1. To begin with we will need to add a JSON object to store the data from the request:

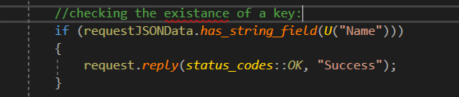


1. To parse our request JSON we can use a synchronous lambda function like so:
   * *Add this code inside the if**statement* ***before your reply***



|  |
| --- |
| BreakDown: |
| * The extract\_json function returns a concurrency process. This means it can be run asynchronously. In our case we need the data so we ensure we add a .wait() to the process to pause this threads application until the data has been processed. * Once the extract\_json function does its thing it returns a pplx::task which returns the processed json data.   + We can access this processed JSON data using task.get() inside of the then() which is automatically called after extract\_json() completes.   + We pass in a lambda function to process the results of the task completion and pass in the requestJSONData as a capture variable reference so we can use the data outside of the lambda function afterwards. |

1. At this point we now have the JSON data and can parse it to determine if it has the proper data we expect in it. To check the JSON data we can use the following:



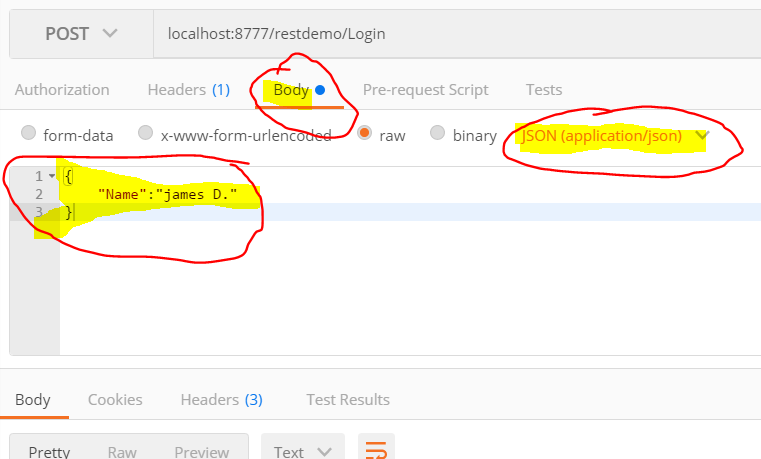
1. To obtain the value of an existing JSON key we can use the following:
   * **Place this before the reply**



The U here before (“Name”) is saying to treat the string as a UTF-32:

<https://docs.microsoft.com/en-us/cpp/cpp/string-and-character-literals-cpp?view=vs-2019>

1. **Run the Server Code.** To test our request with JSON in postman we need to modify the body of the request**:** 
   * **Set the radio button for data type to ‘Raw’**
   * **when we do this postman automatically adds a header for us for Content-Type. Change the last dropdown to JSON to populate this header**



1. We will need to have some way to keep track of multiple users logging into the server. To Manage this we will use a map. At the top of your file, include map:



Then also in global space above, create a map of string/int to track our players logged in.

We will also create a global int to track session ID’s:



1. Back in our POST handler, we can use this map to store each user logging in.

To begin with create an assignment to a new initials variable:



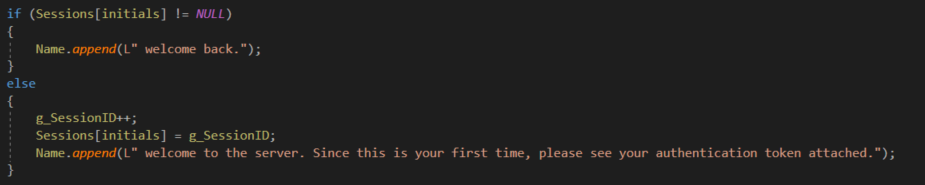
Next, increment our g\_SessionID counter, then assign it to the users initials in the map:



1. We can then use the players name to send a response back to the user:



1. We don’t always want to generate a new SessionID every time they log into the server. To handle this, we can perform a simple comparison of the Name passed in through JSON to the map.
   * **Wrap your code in an if /else like so:**



1. To send our response back we can just replace the reply success with our name variable:



**SIGN OFF WITH INSTRUCTOR FOR WALKTHROUGH 6**

**Complete the Lab 4 available on blackboard.**