CS 261 Mini-Lab: C++ REST SDK

NOTE: Labs are designed to provide an opportunity to ask for help!

Links: C++ Rest SDK, Documentation, Tutorial, Samples

Setup

- Ensure that you're using the class VM in VirtualBox 6+ (or equivalent configuration). See the CS 261 Moodle for more details on the class configuration.
- You should not need to update your OS for this lab, but it's good practice to start each session with:
    1. sudo apt-get update
    2. sudo apt-get upgrade
    3. sudo reboot
2. We will be using Visual Studio 2019 for this lab, which is installed on the lab machines. You could use 2017 instead, without much (if any) change.

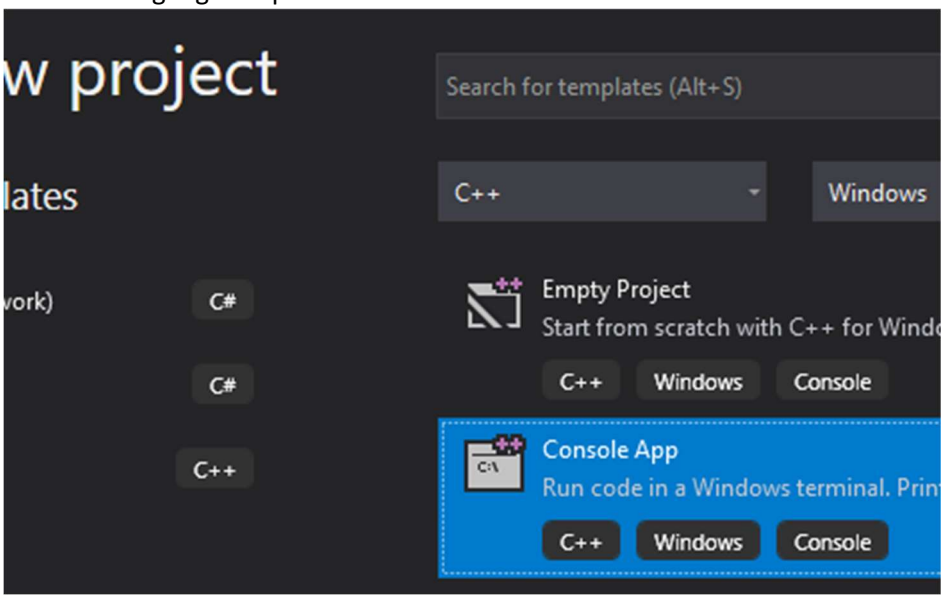
Part 1: Server Setup

1. In your VM, create the directory **/srv/node/labJsonServer**
2. Extract the CS261_LabJsonServer.zip archive that accompanied this document, and copy the files into the directory you just created.
    - If you downloaded it using the browser in the VM, just cp it.
    - If you downloaded it using a Windows browser, drag-and-drop it into the SVN or use the shared-folder feature.
3. In Terminal, go to the directory you created (if you're not already there).
4. Run **npm install**, which will install the packages required by this server app.
5. Run **node labJsonServer.js**
6. Within the VM, open a browser at http://localhost:4000/lab, and note that you get a "Cannot GET /lab" response. This is expected behavior.
7. Within the VM, open Postman, and import CS261_LabJsonServer_Postman.json (which accompanied this document).
8. Run the imported Postman collection, and verify that you get 3 passed tests.
9. In VirtualBox, set up port forwarding from host port 4000 to guest port 4000
    - If you're unsure how to do this, complete the Port Forwarding mini-lab.
10. Test your port forwarding by opening a browser in the host OS (Windows) and enter http://localhost:4000/lab. You should get a "Cannot GET /lab", as expected, instead of a timeout.

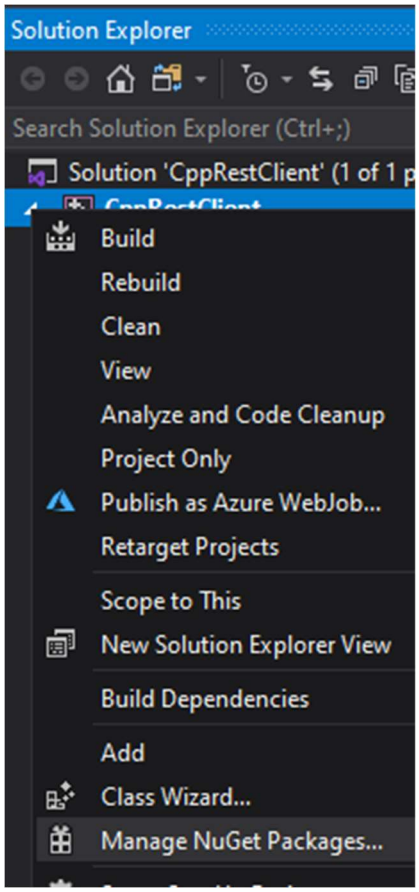
Part 2: Client Project Setup

11. Launch Visual Studio 2019
12. Choose Create New Project, and select the C++ Console App, and click Next.

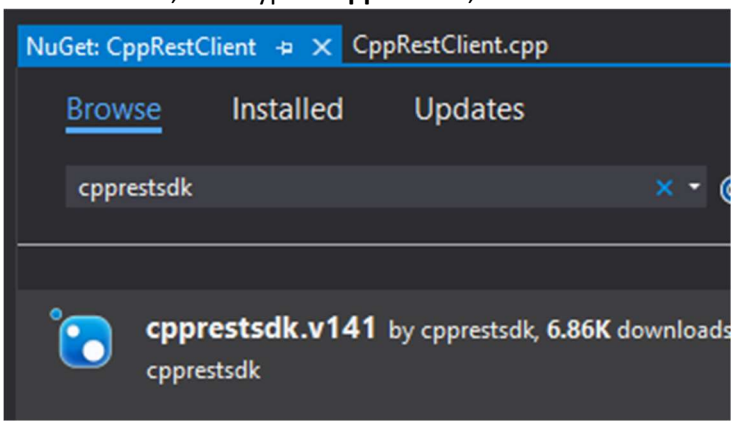
- Note the language drop-down below:



13. Name the project CppRestClient. Select a location that's appropriate to your PC.
14. Click "Create".
15. In Solution Explorer, right click on the CppRestClient **project** (not the solution), and select Manage NuGet Packages.



16. Select Browse, then type in **cpprestsdk**, and select the v141 option (with the most downloads).



17. On the right side of the NuGet dialog, select **Install**.
    - If a "Preview changes" dialog pops up, click OK.
18. Close the NuGet dialog.
19. Build the project (without any code changes), to make sure it's working so far.
    - Note that you did **not** modify the include, lib, etc. directories for the project! NuGet takes care of that for you at build and link time.

Part 3: Client Development

For the remainder of the lab, you will be given a series of goals and suggestions of where to look, but it's up to you to write the code.

Your overall goal is to send a POST request with a JSON body to http://localhost:4000/lab, and then output the response to the console. The JSON body should take the form of:

{

input: "something"

}

The expected output should be:

{

echo: "something", // or whatever was provided as input

id: "some ID string"

}

In general:

- **Make sure you build after each step is complete**, to make sure you're in good shape.
- The SDK is naturally async, using Microsoft's standard pplx::task library. If you're unfamiliar with it, you can read the C++ Rest SDK's documentation page for Programming with Tasks. You can also infer the behavior by reviewing the entire tutorial, including the syntax for modern C++ anonymous functions.

  - Note: one quick trick to get the synchronous, blocking behavior (generally not good, but we're just experimenting here!) is to add **.get()** to the end of a function, like **request(…)**, that returns a task.
- The provided server has some console output, which you will see in your VM's terminal, that may help you. If you want more help – modify the server and add more logging.

In order to achieve this goal, you should achieve several smaller goals along the way.

1) Determine a good starting set of #include and namespace references that will allow us to experiment.
   a. Review the tutorial (link above), and consider that you will need a **HTTP client** and **JSON** functionality. That implies two headers.
   b. Copy over the tutorial's common "using" block, and add the "unused" using for web::json.
2) Build a http client object.
   a. In the tutorial, there's a line to create a http_client. We can copy that line, and replace their URL with ours (i.e., http://localhost:4000). *Make sure you keep the U()!*
3) Build the URI for the request. There's a single line in the tutorial to create a **uir_builder**; that should be all you need. Replace their URI (/search) with yours (/lab). *Make sure you keep the U()!*
4) Send the request – we'll worry about the JSON body in a minute. There's one line to call client.request; that's what you need. Note that the method should be POST. You can return the task returned by request() if you're following the asynchronous syntax, or you could add .wait().
   a. If you're trying to follow the asynchronous pattern, check the whole tutorial code file – near the bottom of the page – to see how the composite task is called.
5) Output the content of the response to cout. Note that the http_response object you got after completing the request task has an extract_utf8string() function that seems useful for this purpose, but note that this function also returns a task!
   a. If your output is Cannot GET /lab, you forgot to update the method to methods::POST.
   b. If your output is Bad Request, you're in good shape – our server is responding correctly, since we didn't include a json body.
6) Build the JSON object that we will include in the request body. You'll find a relevant page, JSON, linked on the right side of the documentation page.
   a. The documentation page shows several possible approaches – experiment!
   b. Note that we are building an object with an internal property – "input".
   c. Note that you can create a json::value::object() and store it in a json::value().
   d. Note that you can access myJsonValue[U("someProperty")] regardless of whether it already exists as a member (just like JavaScript).
   e. Make sure you're consistently using U() around string values!
   f. Note that there is another overload of http_client::request that takes *three* arguments, where the third argument is a json::value that will be used as the body. Handy!

If your program works, you should now see the expected output sent to the console.
Congratulations!

Suggested Extensions

Once you complete these steps, there are several additional steps you could take:

- Parse the response body as a JSON value, and extract the values from the JSON object. Output those values themselves to the console output, instead of the whole HTTP response.
    - Be careful with your text encoding!
- Use a command-line argument to your client application as the "input" value.
    - Again, be careful with your text encoding!
- Run your own Assignment 1 server on your VM, remembering to update/add a port forwarding rule for the port it's using. Then, try using different APIs, methods, etc. with your client program.
- Examine the status code in the response, and experiment with how you might handle different error status codes (400, 401, etc.).
- Experiment with using different encodings (removing U(), etc.) and see what happens.

Notes

- A major source of difficulty in working with modern text-reliant code, like the C++ Rest SDK, is being deliberate about character width and encoding. Note the use of U() and L() throughout the documentation and samples.
- The NuGet package manager works well for a DigiPen lab environment, but there are other solutions:
    - On the Git page for the SDK, Microsoft recommends vcpkg. Currently, this lab doesn't use that because it requires other software to be installed as admin on Windows.
    - On Linux, you would install the SDK using the instructions provided on the Git page.
- Once the package is "installed" in this way, you can reference it with system includes (<>, instead of ""), on both platforms.
- You could use the task API with anonymous function closures, or you could use them by themselves. You could wait for the API calls directly, or you could build a compound task with **.then** continuations and call **wait()** just once.