Lab 8: Native Addons for Node using C++

In this exercise, you will:

- Setup a native plugin using C++.
- Compile C++ into machine level code.
- Access the C++ function using Node.
- Build a simple sum calculator using addons.
- See the changes reflected in the Electron app.

1. Check out the `Lab-Native-Addons` branch from the remote repository.

```
git checkout origin/Lab-Native-Addons
```

Note: The HEAD of this branch provides the completed code for the entire exercise.

---

2. Run `npm install` to ensure all the dependencies have been installed.

3. Create a new local branch for your work based on the "starting point..." commit.

```
git log --oneline
git checkout <hash for starting point commit>
git checkout -b Lab-Native-Addons-mine
```

4. Make sure that the following npm packages are saved in your dependencies.

```
npm install ---save node-gyp
npm install --save robotjs
```

5. Navigate to the `assets/js/native` folder of the app, and let's set up our `binding.gyp` file to describe the build configuration.

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc"
      ],
      "include_dirs": [
        "<!(node -e \"require('nan')\")"
      ]
    }
  ]
}
```

6. Let's write some native code, we're going to write a simple summation function that iterates over a few thousand times. We'll explore the relevance of the iteration towards the end of this lab.

Note: This C file makes use of the node headers as well as the Chrome v8 engine, docs linked below.

```
#include <node.h>

void Sum(const v8::FunctionCallbackInfo<v8::Value>& args) {
v8::Isolate* isolate = args.GetIsolate();

int i;
double a = 3.1415926, b = 2.718;
for (i = 0; i < 100000; i++) {
a += b;
}

auto total = v8::Number::New(isolate, a);
args.GetReturnValue().Set(total);
}

void Initialize(v8::Local<v8::Object> exports) {
NODE_SET_METHOD(exports, "sum", Sum);
}

NODE_MODULE(addon, Initialize)
```

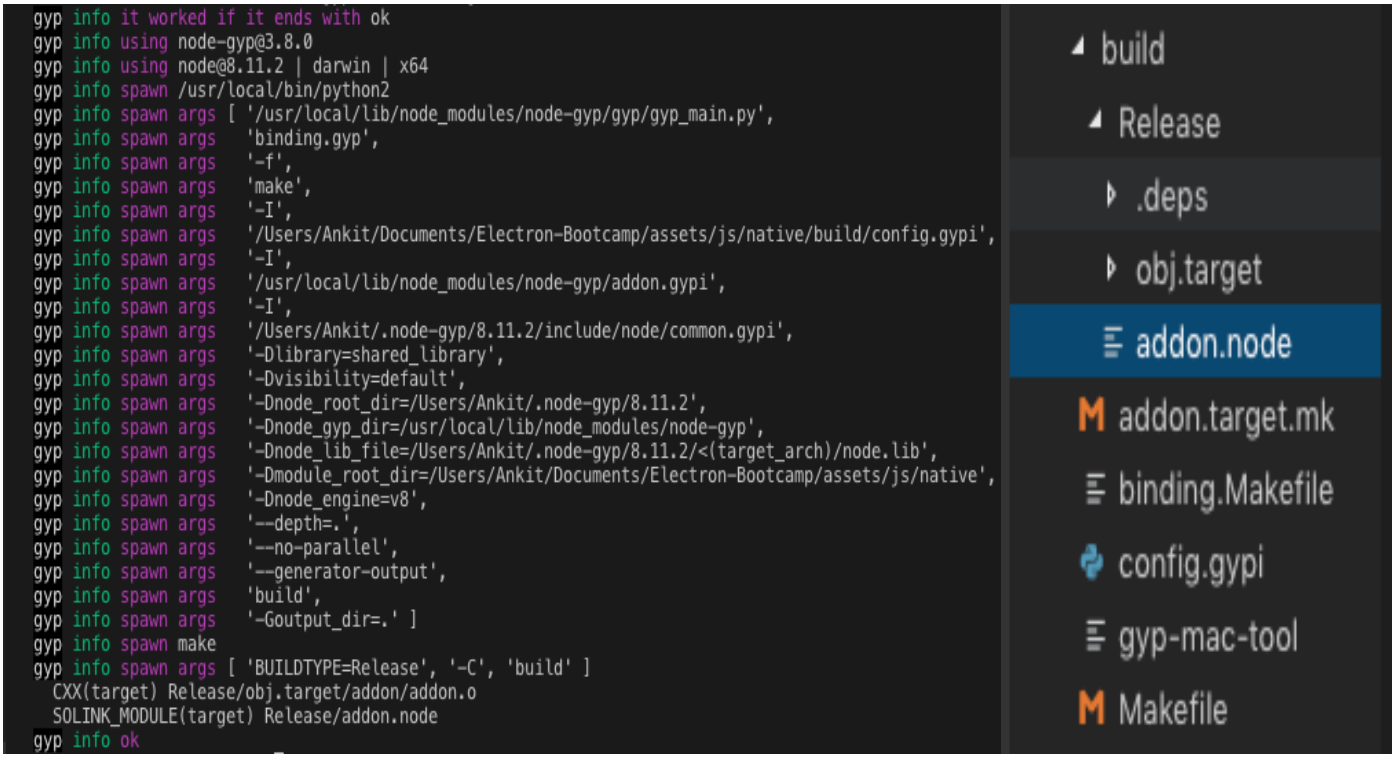
7. Once we have our target and source file in there, we can now proceed to build our native addon which we will then use in our node file. Make sure you're in the `assets/js/native` folder before you run the following command in the terminal.

```
node-gyp clean configure build
```

8. If everything went smoothly, you should see an "*ok*" in the terminal window and a new "`build`" folder in the same directory that you're in. This new build folder contains the node file that we can incorporate into our JS work on the app, and we're just a few more steps away from finishing up.

Reminder: Make sure to build the file each time that you make changes to the `addon.cc` file!

Note: The `addon.node` file won't be visible in your IDE or text editor that you're using, but you can open it up in notepad to view the machine level code in binary.



9. Let's create our JS file in the same directory, we need to import our `addon.node` file, so that we can utilize the "sum" function that we have created from there. If you try to open the ".node" extension, you'll see that it's low level machine code that's been translated from C (read note above).

```
const addon = require('./build/Release/addon');
```

10. We're now going to replicate the code from our C file into our JS file, so that we have another "sum" function to compare our native function to.

```
const {performance} = require('perf_hooks')
const addon = require('./build/Release/addon')

function sum() {
  let a = 3.1415926, b = 2.718
  for (let i = 0; i < 100000; i++) {
    a += b
  }

  let total = a
  return total
}

let t1 = performance.now()
addon.sum()
let t2 = performance.now()
console.log("c++: " + (t2-t1) + " ms")

let t3 = performance.now()
sum()
let t4 = performance.now()
console.log("js: " + (t4-t3) + " ms")
```

11. Once that's done and you're able to see the results in the console, all that's left for us to do is to retrieve this information and display it on the front end in our `native-addon.html` file

```
let timecpp = t2-t1
let timejs = t4-t3

window.onload = function() {
  let cppTimeDiv = document.querySelector("#cpp-time")
  let jsTimeDiv = document.querySelector("#js-time")

  cppTimeDiv.innerHTML = timecpp
  jsTimeDiv.innerHTML = timejs
}
```

12. Start the app.

```
npm run start
```

Error Handling: Manually building the module seems to resolve the issue with NODE_MODULE_VERSION, navigate to the `assets/js/native` directory and type the following code if you're getting "unhandled" issues (be sure to replace the target with your current version of Electron)

```
HOME=~/.electron-gyp node-gyp rebuild --target=X --arch=x64 --dist-
url=https://atom.io/download/electron
```

13. The main thing you'll notice here is that the C function runs very quickly, whereas JS might take a little longer. The difference might be subtle for a simple iteration, but imagine if we increase to maybe a few million times or did some more complex operations. That's going to add up a lot and the toll it will take on the processor too, so we can see that C/C++ does things a bit more efficiently and effectively. Now combine that with Node, and you have a very powerful tool which you can do amazing things with!

14. Last thing to keep in mind is that this is only one of the ways to create a native module, there's a few other options which require a little more configuration. For the purposes of this bootcamp, this is the simplest method we chose to allow us to get up and running right away.

Bonus: You can try to incorporate RobotJS into your code, this will allow you to have control over your mouse, keyboard and screen. Try it out if you're interested, the link is provided below and they have some good examples.

Helpful Links

https://v8docs.nodesource.com/
https://nodeaddons.com/
http://robotjs.io/