Basic Node Hands-On

Dr. Patrick Young, Stanford University

CS193C

Node.js (commonly referred to simply as Node is a standalone JavaScript runtime environment built on top of V8, the same JavaScript engine that powers Google's Chrome web browser. Node allows us to run JavaScript outside of the web browser environment. Running Node on your computer will give it full access to your computer including access to the file system, just as a language written in a traditional language such as C or C++ can.

Node is often used to act as a web server or more generally as an application server.[1] As a server, Node presents a single-threaded event-handling model that mirrors the web browser's single-threaded with event queue approach.

Installing Node

Go to https://nodejs.org/en/ to download a copy of Node. For our purposes, either the LTS (Long Term Support) or the Current download will work fine. If you're doing actual development work with Node, the LTS version is probably a better choice, as it provides a stable platform to develop on. Whereas if you want to experiment with the latest Node or JavaScript features the Current download is the best choice, however, updates of Current may break code written for earlier Current builds.

Starting Up Node

You can run Node from the command line by using the Command Prompt on Windows or using the Terminal Application on Macintosh. Bring up the Command Prompt or Terminal Application (Google this if you need more detailed instructions).

Type "Node" at the command line to execute Node.

Interacting with Node

Node provides in interactive JavaScript environment. Try executing some basic JavaScript like:

```
2 + 3
```

or something more complicated like:

[1] A web server specifically serves up content using the HTTP protocol, whereas an application server (often abbreviated as simply 'app server') is a server that may serve up content using other protocols, including application specific protocols.

```
var price = 5;
var amount = 10;
price * amount;
```

You can also define functions and then execute them. Try entering:

```
function square(x) {
   return x * x;
}

square(5);
```

To exit Node, you can type a ctrl+D.

Executing Programs

While using Node interactively works great when you're experimenting with JavaScript, if you're using Node as a server, you'll want to write your program in a separate JavaScript file and execute it. Create a file named "example.js" with the following contents:

```
console.log("This is a Node program");
```

You can now startup Node and get it to execute the code in your JavaScript file by typing:

```
node example
```

at the command line prompt. Make sure that you execute this command while in the same directory that contains your "example.js" file.

Using a Node Module

Node has a number of standard modules. You can see a full list here:

https://nodejs.org/dist/latest-v8.x/docs/api/

Node does not use the standard ECMAScript 6 module mechanisms, instead it has its own mechanisms for importing modules which predate ECMAScript 6.

To import a module, we use "require" for example, we load the fs module (fs stands for File System) like this:[2]

```
var fs = require("fs");
```

We use the object returned by require to access the different elements provided by the module. You can see the various elements provided by "fs" here:

https://nodejs.org/dist/latest-v8.x/docs/api/fs.html#fs_fs_existssync_path

In this simple program, we check if a file named "example.js" exists in the current directory:

[2] You can also use JavaScript destructuring on the value returned by "require" to give more direct access to functions, classes, or variables exported by the module.

```
var fs = require("fs");

if(fs.existsSync("example.js")) {
    console.log("example.js exists");
} else {
    console.log("example.js does not exist");
}
```

Synchronous and Asynchronous Calls

Many Node methods provide synchronous and asynchronous calls. For example if we want to get information about a file, we can either call fs.stat or fs.statSynch. As its name implies the later is a Synchronous call. This means that if we call it, Node will stop executing code until the operating system returns with information about the file. We can use the Synchronous version like this:

```
var fs = require("fs");

var stat = fs.statSync("example.js");

if(stat.isFile()) {
    console.log("example.js is a file");
} else {
    console.log("example.js is not a file");
}

console.log("Okay, we've finished calling statSynch");
```

If you execute this code, notice that the “example.js is a file” is printed before “Okay, we’ve finished calling statSynch”.

Calling the Asynchronous version allows our JavaScript program to continue running until the operating system returns with the result. Instead, we provide a callback function which will execute once the Operating System has returned with our information. Here we call stat instead of statSync. Notice that the Synchronous version is specially marked with Sync, whereas the Asynchronous version is just called stat, as it’s considered the “standard” version of the method. In general, with Node you should call the Asynchronous versions, as this will allow Node to run much more efficiently.

```
var fs = require("fs");

function handleStat(err,stat) {
   if(err) {
      console.log("handleStat returned error");
      return;
   }
   if(stat.isFile()) {
       console.log("example.js is a file");
   } else {
       console.log("example.js is not a file");
   }
}

var stat = fs.stat("example.js",handleStat);


console.log("Okay, we've finished calling stat");
```

If we run this code, the “Okay, we’ve finished calling stat” prints out before “example.js is a file”. This is because our handleStat method won’t run until our original code has completed execution. Remember there’s only one thread of execution.