Introduction to Node.js - Labs

Completed source code for all labs (for checking your work) can be found at:

`https://github.com/watzthisco/intro-to-node`

Version 2.0, March 2019
by Chris Minnick

www.watzthis.com

Disclaimers and Copyright Statement

Disclaimer

WatzThis? takes care to ensure the accuracy and quality of this courseware and related courseware files. We cannot guarantee the accuracy of these materials. The courseware and related files are provided without any warranty whatsoever, including but not limited to implied warranties of merchantability or fitness for a particular purpose. Use of screenshots, product names and icons in the courseware are for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the courseware, nor any affiliation of such entity with WatzThis?.

Third-Party Information

This courseware contains links to third-party web sites that are not under our control and we are not responsible for the content of any linked sites. If you access a third-party web site mentioned in this courseware, then you do so at your own risk. We provide these links only as a convenience, and the inclusion of the link does not imply that we endorse or accept responsibility for the content on those third-party web sites. Information in this courseware may change without notice and does not represent a commitment on the part of the authors and publishers.

Copyright

Help us improve our courseware

Please send your comments and suggestions via email to info@watzthis.com

Credits

About the Author

Chris Minnick is a prolific published author, blogger, trainer, web developer and co-founder of WatzThis?. Minnick has overseen the development of hundreds of web and mobile projects for customers from small businesses to some of the world's largest companies, including Microsoft, United Business Media, Penton Publishing, and Stanford University.

Since 2001, Minnick has trained thousands of Web and mobile developers. In addition to his in-person courses, Chris has written and produced online courses for Ed2Go.com, O'Reilly Media, and Pluralsight.

Minnick has authored and co-authored books and articles on a wide range of Internet-related topics including JavaScript, HTML, CSS, mobile apps, e-commerce, Web design, SEO, and security. His published books include JavaScript for Kids, Writing Computer Code, Coding with JavaScript For Dummies, Beginning HTML5 and CSS3 For Dummies, Webkit For Dummies, CIW eCommerce Certification Bible, and XHTML.

For 16 consecutive years, Chris was among the elite group of 20 software professionals and industry veterans chosen by Dr. Dobb's Journal to be a judge for the Jolt Product Excellence Awards.

In addition to his role with WatzThis?, Chris is a novelist, cheese-, beer-, and winemaker, swimmer, and musician.

Table of Contents

Setup Instructions

Course Requirements

To complete the labs in this course, you will need:

- A computer with MacOS, Windows, or Linux.
- Access to the Internet.
- A modern web browser.
- Ability to install software globally (or certain packages pre-installed as specified below).

Classroom Setup

These steps must be completed in advance if the students will not have administrative access to the computers in the classroom. Otherwise, these steps can be completed during the course as needed.

1. Install node.js on each student's computer. Go to nodejs.org and click the link to download the latest version from the LTS branch.
2. Install a code editor. We use Visual Studio Code in the course
3. Make sure Google Chrome is installed.
4. Install git on each student's computer. Git can be downloaded from *http://git-scm.com*. Select all the default options during installation.

Testing the Setup

1. Open a command prompt.
    a. Use Terminal on MacOS (/Applications/Utilities/Terminal).
    b. Use gitbash on Windows (installed with git).
2. Enter `cd` to navigate to the user's home directory (or change to a directory where student files should be created).
3. Enter the following:

```
git clone https://github.com/watzthisco/intro-to-node
```

The lab solution files for the course will download into a new directory called intro-to-node.

4. Enter `cd intro-to-node` to switch to the new directory.
5. Enter `npm install`

This step may take some time. If it fails, the likely problem is that your firewall is blocking ssh access to github.com and/or registry.npmjs.org.

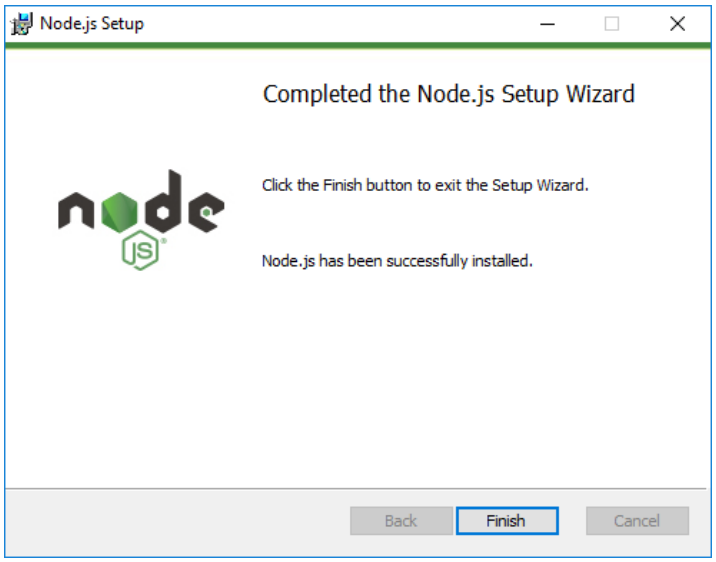
6. When everything is done, enter `npm run build`
7. If you get an error, delete the node_modules folder (by entering `rm -r node_modules`) and run `npm install` again, followed by `npm run build`.
8. A series of things will happen and then a message will appear and tell you that the test passed.

Lab 1 - Getting Started with Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It can be used to create server-side programs with JavaScript as well as for automating development tasks. In this lab, you'll install Node.js, learn to use the interactive shell (aka REPL), write a node application, and learn about using npm for package management.

Part 1: Installing Node.js

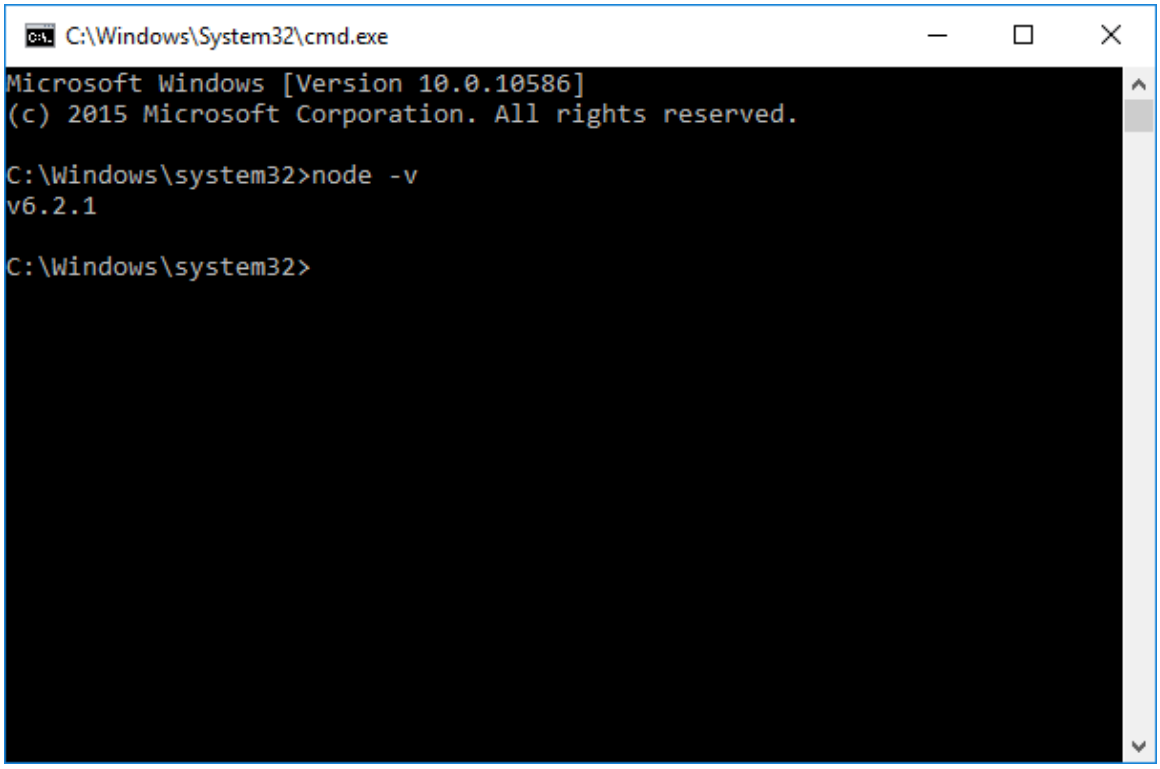
- ☐ 1. Go to *https://nodejs.org* and download the latest version of Node from the LTS (Long Term Support) branch.
- ☐ 2. When it finishes downloading, launch the installer to install Node.js
- ☐ 3. Select all the default options.



Part 2: Getting to Know Node.js

In this part, you will learn the basics of using Node.js.

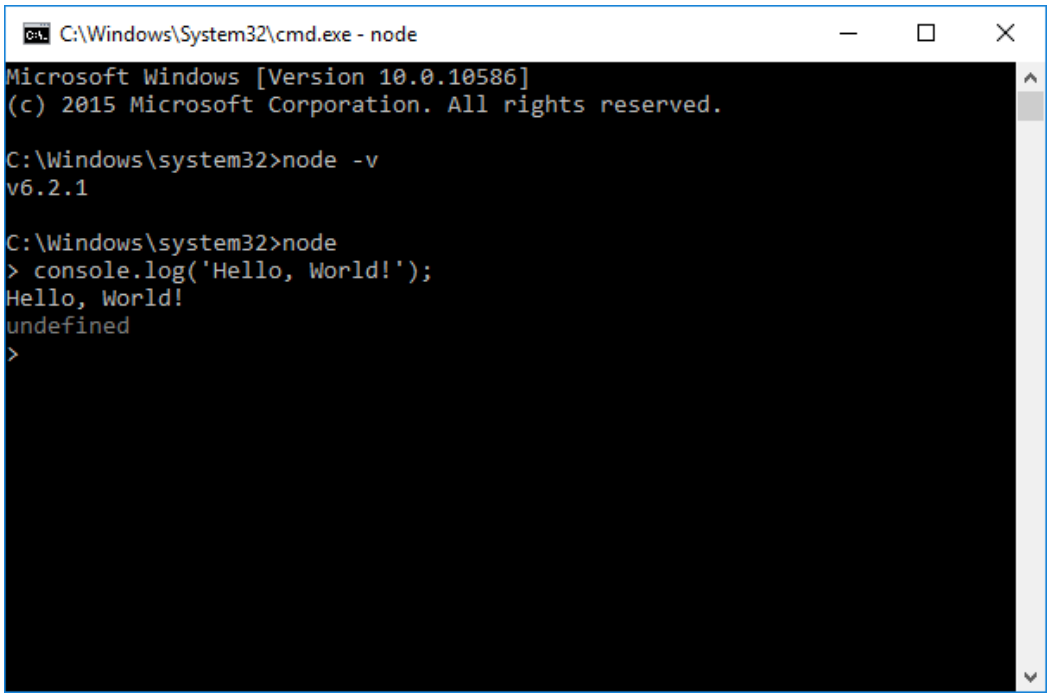
- ☐ 1. Open a command line application.
  - a. MacOS: Navigate to Applications / Utilities and double click on **Terminal**.
  - b. Windows 7, 8, or 10: Open a search box and enter **cmd** to locate the Command Prompt. Open it.
- ☐ 2. To check whether Node.js is property installed, enter `node -v`
  You should see something like the following:



☐ 3. Enter `node` to open the interactive shell.

Note: You can enter any JavaScript statement into the interactive shell and you have access to all the Node.js modules.

☐ 4. Enter `console.log('Hello, World!');` into the shell.



Note: Every JavaScript statement has a return value. The default return value is `undefined`. So, if you execute a command that doesn't have any other return value, as in this case, node outputs `undefined` after the results of running the statement.

The other way to execute code with node is to write your JavaScript into a file and execute that file.

- ☐ 5. Open the code editor of your choice and create a file named **javascript.js** inside the **intro-to-node/labs/lab01** folder.
- ☐ 6. Enter the following code into **javascript.js**:

```
console.log('Hello, World!');
```

- ☐ 7. Save javascript.js
- ☐ 8. Exit node's interactive shell by pressing **CTRL-C** twice.
- ☐ 9. In Terminal (MacOS) or the Command Prompt (Windows), navigate to the **intro-to-node/labs/lab01** folder.

Note: You can use the cd command (MacOS and Windows) to change directories. To go up a directory use `cd ../`
To go into a directory, enter `cd` followed by the name of the directory. You can list the contents of a directory by using `ls` (on MacOS) or `dir` (on Windows).

- ☐ 10. Once you've located javascript.js, enter `node javascript.js` to run it.

Lab 02: First Look at Asynchronous Code

Perhaps one of the most difficult things for beginners to understand about Node is its event-driven nature. In this lab, you'll see an example of an application that was written in a synchronous way and one that's written in an asynchronous way.

- ☐ 1. Open the code editor of your choice and create a file named **sync.js** inside the **labs/lab02** folder.
- ☐ 2. Inside of **sync.js**, write the following code to make Node read a file in a synchronous way.

```
const fs = require('fs');
let content = fs.readFileSync('file.md', 'utf-8');
console.log(content);
console.log("Done!");
```

- ☐ 3. Create a file in the same directory named `file.md` and put some text content into it.
- ☐ 4. In your terminal or command line, navigate to the **labs/lab02** folder and run the program by typing:

```
node sync.js
```

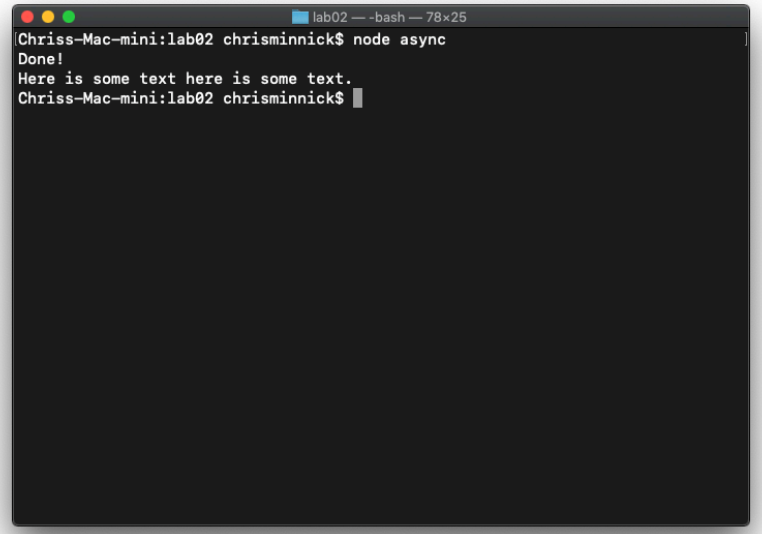
If everything works correctly, the program should output the contents of `file.md`, followed by 'Done!'.

- ☐ 5. In your code editor, create another file inside the **labs/lab02** folder and name the file **async.js**.
- ☐ 6. Inside of **async.js**, write the following code, which uses the asynchronous readFile method:

```
const fs = require('fs');
let content = fs.readFile('file.md', 'utf-8',
function(err,data){
  console.log(data);
});

console.log("Done!");
```

- ☐ 7. In your terminal or command line, navigate to the **labs/lab02** folder and run the program by typing:

```
node async.js
```



Notice that the asynchronous program outputs the "Done!" message before outputting the contents of the file.

Can you explain what's happening here?

Challenge: modify the asynchronous program to produce the same output as the synchronous program?

Challenge: modify the asynchronous program to display an error message when an error occurs (such as if you rename or delete file.md).

Lab 03: npm

The node package manager (npm) is the tool for installing and managing node modules created by the node community. In this part, you will learn about the basic npm commands.

☐ 1. In your command line, enter `npm -v` to find out what version of npm is installed on your computer.

☐ 2. Enter `npm install npm -g`

This command will install the latest version of npm.

Note: If the installation of npm fails on MacOSX, you may need to preface it with sudo to install as the super user.

☐ 3. Enter `npm -v` to see what version of npm is now installed.

☐ 4. Enter `npm ls -g`

This command will list all the packages that are installed on your computer currently. Use it without the `-g` to see only packages installed into your current project.

☐ 5. Enter `npm help ls`

The help command will show you documentation for a package. On Windows, it may open in a browser. On MacOS, the help will display in the Terminal.

☐ 6. If the help file displayed in the console window, type `q` to exit the help system.

☐ 7. Enter `npm update` or `npm update -g`

`npm update` will search the npm registry for newer versions of installed packages and install them along with their dependencies.

These are all the basic commands you need to know to get started with npm. In future labs, we will be using npm extensively to install and manage packages used by our projects.

Part 2: Initializing npm

In this part, you will initialize npm for your project and learn about the package.json file.

☐ 1. Open a command line (Git Bash or cmd.exe on Windows or Terminal on Mac) and navigate to the intro-to-node directory (type **`cd intro-to-node`**) and then navigate to **intro-to-node/labs/lab01**).

☐ 2. In your console, enter **`npm init`**.

☐ 3. You will be asked some questions to configure npm for your project. The default values are shown in parentheses after the questions. Press **Enter** or **Return** to accept each of these default values. After you have gone through all the questions, a new file, package.json, is created in this folder.

Note: When using the Git Bash shell on Windows, the configuration script may hang after the last question. When this happens, press **Ctrl+C**. Everything has run and the package.json file was created, but it just doesn't exit correctly.

☐ 4. Open **package.json** in your code editor.

The package.json file configures npm. When you want to install your project in a new directory, you will enter `npm install` and it will follow instructions in this file to do the job.

☐ 5. Type **`npm install`** in the console. There's nothing for npm to do at this point because you don't have any modules installed or instructions inside package.json

Going forward, when you install new project dependencies, they'll be added to your package.json file. Having your dependencies tracked in package.json makes it easy to upgrade them and to install new instances of your development environment on different computers.

Challenge

Install **learnyounode** (https://github.com/workshopper/learnyounode) by typing the following into your terminal:

```
npm install -g learnyounode
```

Note: you may need to preface the above with `sudo` on Mac and Linux.

Run **learnyounode** by typing the following into your terminal:

```
learnyounode
```

Complete exercises 1-4 in learnyounode.

Lab 04: Making a Web Server

In this lab, you'll use Node.js and the http module to create a simple web server that listens for connections and responds with a simple Hello World message.

☐ 1. Open the code editor of your choice and create a file named **server.js** inside the **labs/lab04** folder.

☐ 2. Enter the following code inside of server.js

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer(function(req, res) {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World\n');
});

server.listen(port, hostname, function() {
    console.log(`Server running at
http://${hostname}:${port}/`);
});
```

Note: The console.log statement in this program uses a new (ES6) JavaScript feature called template literals to combine dynamic values with static text. The characters surrounding the string starting with "`Server running at`" are backticks (in the upper left of the keyboard), not single quotes.

☐ 3. Save the **server.js** file.

☐ 4. In your terminal or command line, navigate to the **labs/lab04** folder and run the program by typing:

```
node server.js
```

If everything works correctly, you should see a message that the server is running.

☐ 5. In your web browser, go to the address shown in your console window, which should be **http://127.0.0.1:3000**

The server will return a message that will display in your browser.

☐ 6. Modify the script to return a different message or a full html page.

Lab 05: Writing a Node.js Module

In this lab, you'll write your first node module and then use that module in a program.

☐ 1. Open the code editor of your choice and create a file named **app.js** inside the **labs/lab05** folder. This file will be your main program file, which will use your custom modules to produce output.

☐ 2. Create a second file, named **sumModule.js** inside the same **lab05** folder. This file will contain your module. The module you will create will take two numbers as arguments and return the sum of the two numbers.

☐ 3. Make sumModule.js export a function. The returned function should take three arguments: `number1`, `number2`, and a `callback` function. Type the following in your sumModule.js document:

```
module.exports = function(number1,number2,callback){
};
```

☐ 4. Inside the module before the closing `};`, add the numbers together, like this:

```
var sum = number1 + number2;
```

☐ 5. Next, add some code that will check whether the result of adding the numbers together is a number and call the `callback` function with just a single argument (the `error` argument) if it's not a number.

```
module.exports = function(number1,number2,callback){
    var sum = number1 + number2;
    if (isNaN(sum)) {
        callback("sum is not a number");
    }


};
```

☐ 6. Call the `callback` with `null` as the first argument and the `sum` as the 2nd argument.

```
module.exports = function(number1,number2,callback){
    var sum = number1 + number2;
    if (isNaN(sum)) {
        callback("sum is not a number");
    }
    callback(null,sum);
};
```

☐ 7. Return to your **app.js** file, and require the module:

```
var sumModule = require('./sumModule.js');
```

☐ 8. Call the `sumModule()` function, passing in two numbers and a `callback` function. The `callback` function, per Node's conventions, should have two parameters: `err` and `data`.

```
sumModule(1,5,function(err,data){

});
```

☐ 9. Check whether err has a value and throw an error if so.

```
sumModule(1,5,function(err,data){
if(err) throw err;

});
```

☐ 10. If `err` is `null`, the program will go to the next line. Let's output the sum here:

```
if(err) throw err;
console.log(data);
```

☐ 11. Save the **app.js** file, which should look like this:

```
var sumModule = require('./sumModule.js');

sumModule(1,5,function(err,data){
    if(err) throw err;

    console.log(data);

});
```

☐ 12. Save the **sumModule.js** file, which should look like this:

```
module.exports = function(number1,number2,callback){
    var sum = number1 + number2;
    if (isNaN(sum)) {
        callback("sum is not a number");
    }
    callback(null,sum);
};
```

☐ 13. In your terminal or command line, navigate to the **labs/lab05** folder and run the program by typing:

```
node app.js
```

If everything works correctly, the program should output the sum of the two numbers you passed into the module.

☐ 14. In the app.js file, update the arguments you pass into the module so that at least one of them isn't a number.

```
sumModule(1,"egg",function(err,data){
```

☐ 15. Save the **app.js** file.

☐ 16. Return to your command line and run the program again.

```
node app.js
```

The program will throw an error.

```
Mac-mini-2:node-getting-started chrisjminnick$ node app.js

/Users/chrisjminnick/WebstormProjects/jslabs/solutions/node-getting-started/app.js:4
    if(err) throw err;
            ^
sum is not a number
Mac-mini-2:node-getting-started chrisjminnick$
```

☐ 17. Go back to your **app.js** file and experiment a bit.

☐ 18. Challenge: Change the numbers in the following line, save the file, run the program, and marvel at the results:

```
sumModule(1,"5",function(err,data){
```

Can you fix the module so that it will correctly add a number contained inside quotes to a number that's not in quotes?

Challenge

Complete exercises 5 and 6 in **learnyounode**.

Lab 06: Working with Streams

Part 1: Read Streams

This part of the lab is an introduction to Streams and Buffers. You are probably already familiar with Buffers if you have ever listened to Internet radio or watched streaming video. Rather than transferring an entire file from a source to a destination before using it, Buffers allow you to transport usable bite-sized chunks of data. You can start Streaming a video on YouTube as soon as a few Buffers load. In this lab, we are going to be streaming a copy of the text of Herman Melville's classic of American literature, Moby Dick.

☐ 1. Create a file called **readStream.js** in your lab05 directory.

☐ 2. Open **readStream.js** with your editor of choice and start off the code by requiring the FileSystem module, the Node.js core module used for reading and writing data from files.

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

☐ 3. Create a read stream connected to **MobyDick.txt**. Use the `__dirname` global property that stores your current directory to locate the **MobyDick.txt** file.

```
var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt');
```

The `fs.FileSystem` module uses the method `createReadStream()` to create a read stream object. This read stream can be stored in a variable, passed to functions, and so on.

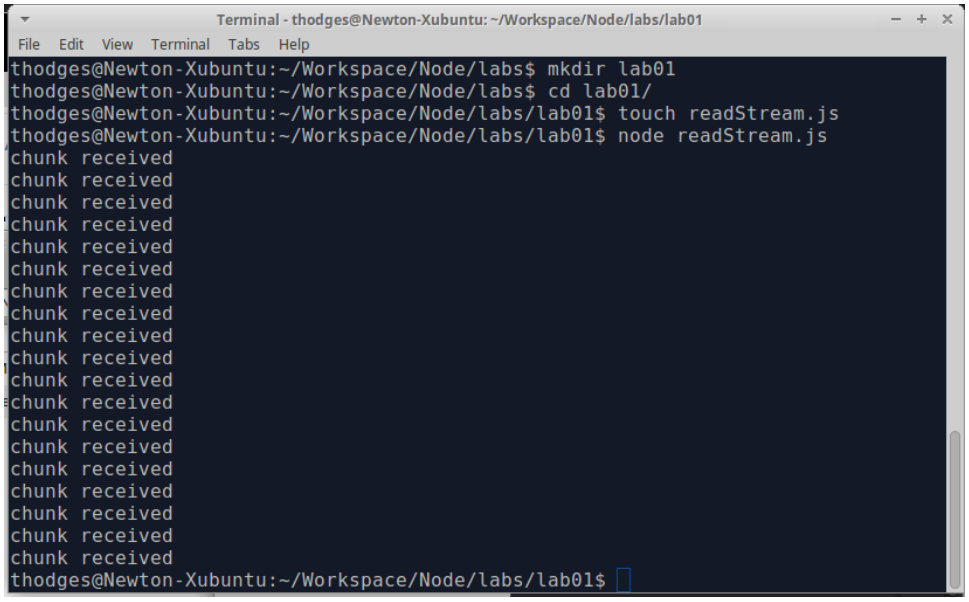
☐ 4. Create an event listener for `myReadStream` that logs to the console whenever a buffer of data is received:

```
myReadStream.on('data', chunk => {
    console.log('chunk received');
});
```

All *stream* objects are instances of EventEmitter. The `EventEmitter.on()` function registers *listeners*. Listeners allow functions to be attached to events emitted by the object. You may have registered an `onclick` event listener in JavaScript code in the past using `object.addEventListener("click", myScript);` One of the events broadcast by `fs.ReadStream` is 'data', which signals when a new chunk of data is ready to be processed.

☐ 5. Open your terminal and run **readStream.js** with node:

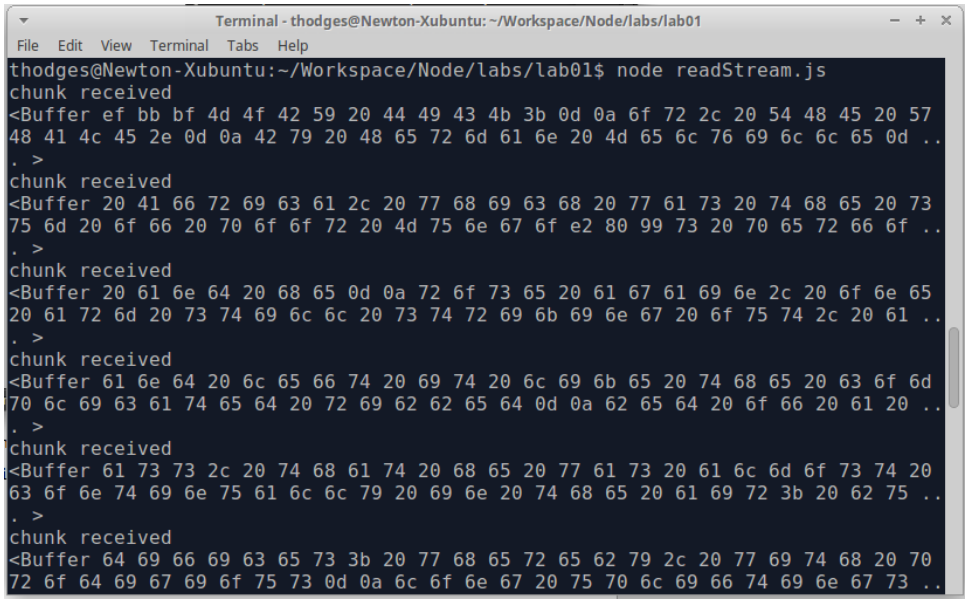
```
node readStream.js
```



Moby Dick is large enough that it is sent to us in nineteen pieces! Each of these data buffers is stored in the variable `chunk` that we have passed to the callback function.

☐ 6. Add a line to log each buffer to the console and then execute your code again:

```
myReadStream.on('data', chunk => {
    console.log('chunk received');
    console.log(chunk);
});
```



The data you get from the server (and that you store in the variable chunk) will be a Node Buffer object. To make it readable, you need to read it to a string, or convert it to a string.

☐ 7. Set the character encoding in the definition of the `myReadStream` to `utf8`:

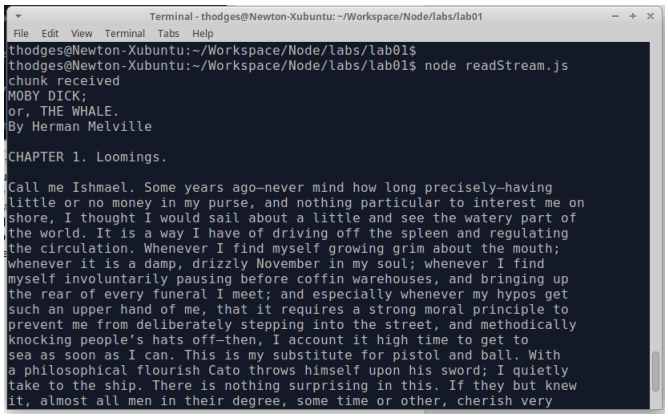
```
var myReadStream =
      fs.createReadStream(__dirname + '/MobyDick.txt',
'utf8');
```

☐ 8. Run **readStream.js** again to see Moby Dick output to your console:

```
node readStream.js
```

Note: Another way to output the chunks as text is to Use the `toString()` method before logging it to the console, like this:

```
myReadStream.on('data', chunk => {
    console.log('chunk received');
    var mobyChunk = chunk.toString();
    console.log(mobyChunk);
});
```



If your terminal program preserves enough of the scrollback you can hunt for the 'chunk received' notifications throughout the text of Moby Dick.

Part 2: Write Streams

In this part of the lab, we are going to pair our read stream object with a write stream object. This will let us copy Moby Dick to a new file.

☐ 1. Create a file named **writeStream.js** in your **lab06** directory and copy over all the code from **readStream.js**.

```
var fs = require('fs');
var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
myReadStream.on('data', chunk => {
```

```
        console.log('chunk received');
            console.log(chunk);
});
```

☐ 2. Define a new stream called `myWriteStream` under your definition of `myReadStream`.

```
var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');
```

☐ 3. Comment out the `console.log` functions from the read stream listener.

```
myReadStream.on('data', chunk => {
    //console.log('chunk received');
    //console.log(chunk);
});
```

☐ 4. Add a line to the read stream listener to use the `fs.write()` method to send our buffer to `myWriteStream`.

```
myReadStream.on('data', chunk => {
    //console.log('chunk received');
    //console.log(chunk);
    myWriteStream.write(chunk);
});
```

Notice that we haven't had to bother with verifying the existence of our file and we won't have to bother with manually closing our file when we are done writing data. There are options available to more precisely control the flow of data to a writeable stream which can be found in the Node.js API.

☐ 5. Run **writeStream.js** from node.

```
node writeStream.js
```

You should see no actual output, but you will discover that your working directory now has a **WriteMe.txt** file which contains a copy of the text of Moby Dick.

Challenge: Pass 'chunk received' messages to the writeable stream so that they now appear in the copied file.

Lab 07: Pipes

Part 1: Basic Pipes

In the previous lab, we created a Read Stream and a Write Stream. We attached a listener to our Read Stream and sent the data buffers that we received to our Write Stream. With pipes we can simplify the process of directing data streams.

☐ 1. In your **lab07** directory, create a file called **pipeStream.js**. Copy over your code from **writeStream.js** in the previous lab.

```
var fs = require('fs');
var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt','utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');

myReadStream.on('data', chunk => {
    myWriteStream.write(chunk);
});
```

☐ 2. Run this program in your **lab07** directory to confirm that it creates the **WriteMe.txt** file.

Read Streams inherit the readable.pipe() method which attaches to a Write Stream. The flow of data is automatically managed so as not to overwhelm a slower Write Stream.

☐ 3. Replace myReadStream.on with a pipe.

```
var fs = require('fs');
var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');
myReadStream.pipe(myWriteStream);
```

☐ 4. Run **pipeStream.js** in your terminal to see the file restored.

Part 2: Duplex and Transform Streams

So far you have used readable and writeable streams. There are two other classes of streams: Duplex streams and Transform streams.
Duplex Streams implement both Readable and Writeable interfaces.
Transform Streams are duplex streams where the output is somehow related to the input.
We are going to write and implement basic transform streams.

☐ 1. In your **lab07** directory, create a new file called **makeBig.js**.

We are going to use this file to make a node module to provide a transform stream that converts text to upper case.

Note: You may find it helpful to review the lab on making modules before proceeding to the next step.

☐ 2. Extend the `stream.Transform` class to make a transform stream.

```
const Transform = require('stream').Transform;
const makeBig = new Transform();
```

☐ 3. Create an instance of `stream.Transform` and pass appropriate methods as constructor objects.

```
const makeBig = new Transform({
      transform(chunk, encoding, callback){}
});
```

For this lab, we're using the simplified construction of a transform stream. The three parameters passed to the constructor are:

☐ 4. chunk: a chunk of buffered data passed to the function
☐ 5. encoding: if chunk is a string encoding is the encoding of the string, otherwise encoding may be ignored
☐ 6. callback: this function is called when processing is completed for the supplied chunk.
☐ 7. Convert all the letters in the chunk to capital letters.

```
const makeBig = new Transform({
      transform(chunk, encoding, callback){
            chunk = chunk.toString().toUpperCase();
      }
});
```

☐ 8. Use the `readable.push()` method to emit a 'data' event. This lets the next receiving stream know that data is available to be processed.

```
const makeBig = new Transform({
      transform(chunk, encoding, callback){
            chunk = chunk.toString().toUpperCase();
            this.push(chunk);
      }
});
```

☐ 9. Finally, executing the callback function that was passed to the module to signal that we are done processing the current buffer.

```
const makeBig = new Transform({
      transform(chunk, encoding, callback){
            chunk = chunk.toString().toUpperCase();
            this.push(chunk);
            callback();
      }
});
```

☐ 10. Export the module and save the **makeBig.js** file.

```
module.exports = makeBig;
```

Your finished makeBig module should look like this:

```
const Transform = require('stream').Transform;
const makeBig = new Transform({
    transform(chunk, encoding, callback){
        chunk = chunk.toString().toUpperCase();
        this.push(chunk);
        callback();
    }
});

module.exports = makeBig;
```

☐ 11. In **pipeStream.js**, import the makeBig module, and pipe the Read Stream through makeBig before sending it to the Write Stream.

```
var fs = require('fs');
var makeBig = require('./makeBig');

var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');

myReadStream
.pipe(makeBig)
.pipe(myWriteStream);
```

☐ 12. In your terminal, run the **pipeStream.js** file with node, and look at **WriteMe.txt** to see Moby Dick in all capital letters.

Part 3: One more transform stream

Just for fun, we are going to create an additional module called `MakePig` that will transform text into Pig Latin and apply it to our stream. *

* "Pig Latin" is a made-up language formed from English by transferring the initial consonant or consonant cluster of each word to the end of word and adding a vocalic syllable to create a suffix. So, "chicken soup" becomes "ickenchay oupsay."

☐ 1. Create a new file called **makePig.js**. Copy over all your code from **makeBig.js**.

```
const Transform = require('stream').Transform;
const makeBig = new Transform({
      transform(chunk, encoding, callback){
            chunk = chunk.toString().toUpperCase();
            this.push(chunk);
            callback();
      }
});

module.exports = makeBig;
```

2. We are going to transform our text into Pig Latin using a crude single line regular expression transformation. If you have the initiative, you are more than welcome to make improvements to this code.

```
chunk = chunk.toString()
            .replace(/\b(\w)(\w+)\b/g, '$2$1ay');
```

☐ 2. Change the `makeBig` constant to `makePig`, both in the definition and the module export.

```
const Transform = require('stream').Transform;
const makePig = new Transform({
      transform(chunk, encoding, callback){
            chunk =
            chunk.toString().replace(/\b(\w(\w+)\b/g,
            '$2$1ay');
            this.push(chunk);
            callback();
      }
});
module.exports = makePig;
```

Note: Because the names of constants and variables are not revealed to the calling functions with module exports, we are not required to make this change. Our constant, with whatever name it takes, is exported by the module and is given a new name when it is imported.

☐ 3. Return to **pipeStream.js** to import the `makePig` module and pipe the read stream through that instead of `makeBig`.

```
var fs = require('fs');
var makeBig = require('./makeBig');
var makePig = require('./makePig');

var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');

myReadStream
```

```
.pipe(makePig)
.pipe(myWriteStream);
```

5. Open your terminal and run **pipeStream.js** from node. Then open **WriteMe.txt** to see Moby Dick badly translated into Pig Latin.

allCay emay shmaelIay. omeSay earsyay goaay—evernay indmay owhay onglay reciselypay—avinghay ittlelay roay onay oneymay niay ymay ursepay, ndaay othingnay articularpay otay nterestiay emay noay horesay, I houghttay I ouldway ailsay boutaay a ittlelay ndaay eesay hetay ateryway artpay foay hetay orldway.

- ☐ 4. Try piping the read stream through both `makeBig` and `makePig` before sending to the write stream to see what happens.

Challenge: Using the code from the Making a Web Server lab, pipe your output to 'res' rather than your Write Stream to display it in a browser

Challenge: Modify your code from the previous challenge to pipe in to the browser the **TheProject.html** file rather than a text file

Lab 8: Process

Process is a global object that provides information about and control over the current Node.js process. This lab only touches on two features of Process, so it's advised to look at the NodeJS API to see what else it is capable of.

Part 1: Process.argv

When calling a `.js` file from node, it's possible to specify additional arguments from the command line. These arguments are stored in the `process.argv` array. The array holds all the entries on the line used to call your node process, so index 0 is the path to node, index 1 is the path to your `.js` file and the remaining indices are any other arguments that you supplied. If you want to test this yourself, create a `.js` file with the single command `console.log(process.argv)` and experiment with running it.
For the first part of this lab, we are going to create a function that will return a custom transformer stream that will perform a find/replace with whatever values we have sent our function. Then, you'll modify `pipeStream.js` (from the previous lab) to accept arguments from the command line and replace character names in Moby Dick with those values.

☐ 1. Create a new file, `processStream.js`. Copy over the last version of your `pipeStream.js` code.

```
var fs = require('fs');
var makeBig = require('./makeBig');
var makePig = require('./makePig');

var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');

myReadStream
.pipe(makePig)
.pipe(makeBig)
.pipe(myWriteStream);
```

☐ 2. We don't need `makePig` or `makeBig` right now so we can comment those out and delete them from the pipe stream. If you'd like to use them after we finish this lab, make sure that you have copies of the module files in the current directory.

```
var fs = require('fs');
// var makeBig = require('./makeBig');
// var makePig = require('./makePig');

var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');
```

```
myReadStream
.pipe(myWriteStream);
```

Next, you'll create a new module with a function that will return a transformer. This module needs to accept two parameters, call them `finder` and `replacer`.

☐ 3. Create a new file named `makeTransformer.js` and inside of that file create a skeleton framework based on these specifications.

```
const Transform = require('stream').Transform;

var makeTransformer = (finder, replacer) => {
      //code to make myTransform will go in here
      return myTransform;
};

module.exports = makeTransformer;
```

☐ 4. To define a transformer inside of the `makeTransformer` function, start by pulling in code from `makePig`.

```
var makeTransformer = (finder, replacer) => {
      const makePig = new Transform({
            transform(chunk, encoding, callback){
                  chunk =
                  chunk.toString().replace(/\b(\w)(\w+)\
                  b/g, '$2$1ay');
                  this.push(chunk);
                  callback();
            }
      });
      return myTransform;
};
```

☐ 5. Change `const makePig` to `const myTransform`.

```
const myTransform = new Transform({
    ...    }
});
```

☐ 6. Modify the `.replace()` method to search for all instances of `finder` and replace them with `replacer`.

```
chunk = chunk.toString().replace(new RegExp(finder,
"gi"), replacer);
```

`New RegExp(finder, "gi")` constructs a regular expression around the contents of finder with the modifiers `g` and `i`. The `g` modifier tells `.replace()` to do a global search and the `i` modifier tells `.replace()` to ignore case. A full discussion of `.replace()` and regular expressions lie outside of the scope of this lab.

Your finished module should look like this:

```
const Transform = require('stream').Transform;

var makeTransformer = (finder, replacer) => {
    const myTransform = new Transform({
        transform(chunk, encoding, callback){
            chunk = chunk.toString().replace(new
RegExp(finder, "gi"), replacer);
            this.push(chunk);
            callback();
        }
    });
    return myTransform;
};


module.exports = makeTransformer;
```

☐ 7. Go back to `processStream.js` and include `makeTransformer`.

```
var makeTransformer = require('./makeTransformer');
```

Next, you'll use this module to return a transformer that will replace the word "whale" with "unicorn".

☐ 8. Call `makeTransformer` with the appropriate parameters and store the result in `richTransformer`. Then, pass `richTransformer` to a pipe between `myReadStream` and `myWriteStream`.

```
var fs = require('fs');
var makeTransformer = require('./makeTransformer');
// var makeBig = require('./makeBig');
// var makePig = require('./makePig');

var myReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');

var whaleTransformer = makeTransformer('whale',
'unicorn');

myReadStream
.pipe(whaleTransformer)
.pipe(myWriteStream);
```

☐ 9. Run `processStream.js` and open `WriteMe.txt` to read your customized version of Moby Dick. Use your text editor's search function to look for uses of the word 'unicorn.'

☐ 10. Now it's time to use `process.argv`. We are going to have three optional arguments on the command line, the first will be the replacement text for 'Moby Dick', the second for 'Ishmael' and the third for 'Ahab'. These will be in indices 2, 3 and 4. Use these to create three custom transformers.

```
var mobyTransformer = makeTransformer(new
RegExp(/Moby\s*Dick/), (process.argv[2] || 'Moby
Dick'));
var ishmaelTransformer = makeTransformer(new
RegExp(/Ishmael/), (process.argv[3] || 'Ishmael'));
var ahabTransformer = makeTransformer(new
RegExp(/Ahab/), (process.argv[4] || 'Ahab'));
```

Note: In making the first transformer, a Regular Expression was used for 'Moby Dick' rather than a string because it is possible to account for an indeterminate number of spaces between Moby and Dick due to typesetting nuances like the name split between two lines or some other irregular spacing. The second and third transformers use regular expressions to maintain consistency. All three of these are written with a pattern such that if no value is provided, the original value is used.

☐ 11. Pipe `myReadStream` through these three transformers.

```
 myReadStream
.pipe(mobyTransformer)
.pipe(ishmaelTransformer)
.pipe(ahabTransformer)
.pipe(myWriteStream);
```

☐ 12. Run `processStream.js` from the command line with three command line parameters.

```
node processStream.js Eggs Bacon 'Charlie Sheen'
```

Here is a sample from WriteMe.txt:

…
"Captain Charlie Sheen," said Tashtego, "that white whale must be the same that some call Eggs."

"Eggs?" shouted Charlie Sheen. "Do ye know the white whale then, Tash?"

Part 2: Process as a Stream

Besides `process.argv`, the process object can also be used as a stream. `process.stdout` and `process.stderr` can be configured either as duplex streams or as writable streams. If you completed the challenges on the pipes lab, you will have already undoubtedly discovered `process.stdout` on your own. `process.stdin` can

be configured either as a duplex stream or a readable stream. For details on these configuration options, it is recommended to refer to the NodeJs API.
Now let's use process.stdout as a writable stream.

☐ 1. In processStream.js, change the last pipe from `myWriteStream` to `process.stdout`.

```
myReadStream
     .pipe(mobyTransformer)
     .pipe(ishmaelTransformer)
     .pipe(ahabTransformer)
     .pipe(process.stdout);
```

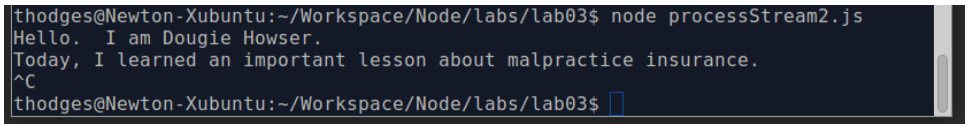
☐ 2. Now run `processStream.js` with or without parameters and the text of Moby Dick should scroll rapidly on your screen.

Let's use `process.stdin` as a readable stream.

☐ 3. Create a new file called `processStream2.js` and fill it with the following code:

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

var myWriteStream = fs.createWriteStream(__dirname +
'/WriteMe.txt');

process.stdin
.pipe(myWriteStream);
```

☐ 4. This program should take text that you type into the terminal and write it to `WriteMe.txt` each time you press `Enter` until you break out with ctrl-c. When you are done, open `WriteMe.txt` to verify that the streams worked as expected.

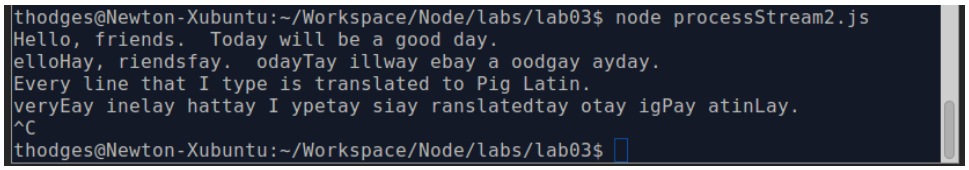


Finally, let's bring in the `makePig` stream to make a command line Pig Latin translator.

☐ 5. Make sure that makePig.js from the pipes lab is in your current directory before proceeding:

```
var fs = require('fs');
var makePig = require('./makePig');

// var myWriteStream = fs.createWriteStream(__dirname
+ '/WriteMe.txt');

process.stdin
.pipe(makePig)
```

```
.pipe(process.stdout);
```



Lab 09: Promises

Part 1: Asynchronous Callbacks

Callback and Promise patterns are both useful in implementing asynchronous functions that return a single value or set of values. To demonstrate some of the advantages of Promises, we will begin by implementing asynchronous functions with Callbacks.
We are going to create an asynchronous function called `makeTimeouts()`. This function will accept two parameters, a number and a callback. In defining asynchronous functions with the callback pattern, the callback function is always passed as the last parameter. In this function, after a delay measured by the number of milliseconds of the first parameter, the function will randomly call the callback function passed in with either an error value or a number between 0 and 5000.
Then we will call this asynchronous function with the first parameter as the number 1000 and the second parameter as a callback function that accepts two parameters, an error and a number. In callback functions used in this asynchronous pattern, the first parameter is always designated to catch any errors. The second parameter is designated to accept any value returned by the asynchronous function. If *error* is defined, then the callback will log that to the console. If the error is undefined, then the callback will focus on the returned value and log that to the console.

☐ 1. Create a file called **makeTimeouts.js** and open it in your preferred editor. Start by defining a function `makeTimeouts()` matching the description above:

```
makeTimeouts = (time, callback) => {
    setTimeout(() => {
        if (Math.random() > 0.8) {
            callback('Fail!');
        } else {
            callback(undefined,
Math.floor(Math.random() * 5000));
        }
    }, time);
};
```

Notice that this function uses `setTimeout()` to set a delay equal to the number of milliseconds as the time parameter. Twenty percent of the time, the callback is called with a single parameter representing an error. Eighty percent of the time, the first parameter is left undefined and for the second parameter is passed an integer between 0 and 5000.

☐ 2. Now we need to call this async function and define a callback to receive the results. This again will match the above description:

```
makeTimeouts(1000, (err, data) => {
    if(err){
        console.error(err);
    } else {
        console.log(data);
    }
```

```
});
```

Notice how the pattern of providing err as the first parameter allows the callback to ignore the nonexistent data parameter when err is defined.

☐ 3. After calling the `makeTimeouts` function, log to the console that your process has completed:

```
console.log('boom!');
```

☐ 4. Run **makeTimeouts.js** several times in the command line, and admire the results. Notice that in all these cases, the function following the asynchronous function (and the callback) is executed first. If this concept confuses you, look back at and review your material on asynchronous code. The `setTimeout()` function simulates a delay that you might encounter not unlike getting data from a server or a disk drive.

```
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeouts.js
boom!
264
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeouts.js
boom!
3911
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeouts.js
boom!
3266
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$
```

☐ 5. Now using callbacks, let's simulate a scenario where there are multiple callbacks, each dependent on the successful completion of prior callbacks. Copy your `makeTimeouts()` function call, and drop the copy into the very same function immediately after logging a successful result:

```
makeTimeouts(1000, (err, data) => {
    if(err){
        console.error(err);
    } else {
        console.log(data);
        makeTimeouts(1000, (err, data) => {
            if(err){
                console.error(err);
            } else {
                console.log(data);
            }
        });
    }
});
```

☐ 6. Change the time value in this second call to the data value returned by the first call:

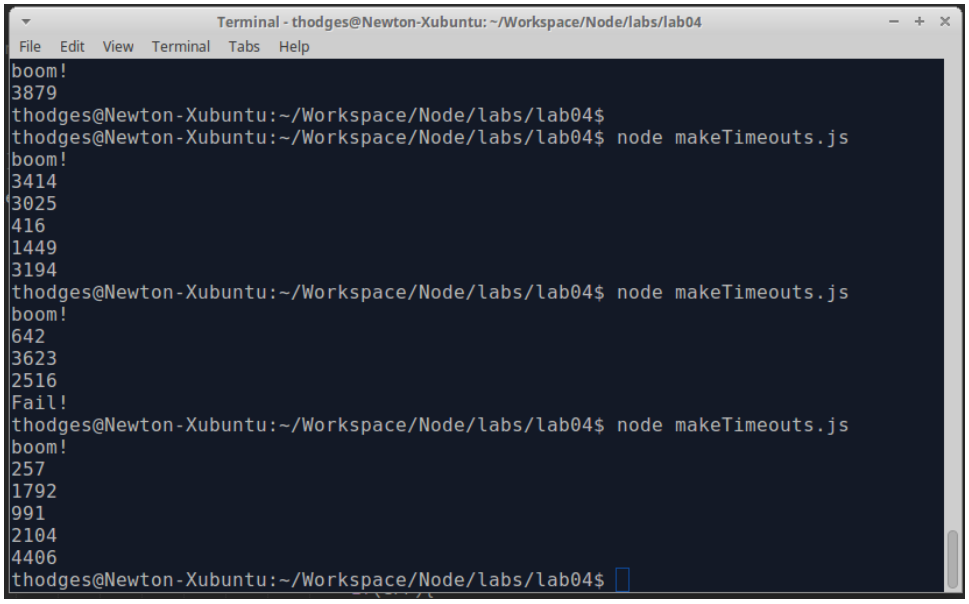
```
makeTimeouts(1000, (err, data) => {
    if(err){
```

```
            console.error(err);
        } else {
            console.log(data);
            makeTimeouts(data, (err, data) => {
                if(err){
                    console.error(err);
                } else {
                    console.log(data);
                }
            });
        }
});
```

☐ 7. Repeat this process three more times:

```
makeTimeouts(1000, (err, data) => {
    if(err){
        console.error(err);
    } else {
        console.log(data);
        makeTimeouts(data, (err, data) => {
            if(err){
                console.error(err);
            } else {
                console.log(data);
                makeTimeouts(data, (err, data) => {
                    if(err){
                        console.error(err);
                    } else {
                        console.log(data);
                        makeTimeouts(data, (err, data) => {
                            if(err){
                                console.error(err);
                            } else {
                                console.log(data);
                                makeTimeouts(data, (err, data) => {
                                    if(err){
                                        console.error(err);
                                    } else {
                                        console.log(data);
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
});
console.log('boom!');
```

☐ 8. Open your command line and execute this a few times, and welcome yourself to Callback Hell.



While the code seems to run fine, there are several issues stemming from the code structure.

- Execution of the code is controlled by the callbacks, not by the calling function. This is inversion of control.
- The code follows this multi-layered nesting pattern of increasing complexity that can be difficult to trace, particularly if there are a variety of asynchronous functions being called.
- The example above contains five separate error handlers buried in the pyramid of code.

Thankfully there is a better way!

Part 2: Promises

A promise serves as a placeholder and container for a final result.
Highlight or underline that sentence and read it slowly and carefully to yourself.
Some of the advantages of promises are as follows:

- There is no inversion of control – promises don't directly control execution via callbacks.
- Chaining is simpler. This will be demonstrated shortly.
- When composing complex asynchronous calls, you have data in the form of Promise objects that you can work with.
- Error handling is simple, whereas with callbacks errors were handled by the callback function, now asynchronous errors are handled by the calling function in the same manner as exceptions.
- The code looks a lot cleaner and easier to maintain.

Let's recast the above functionality with promises.

☐ 1. Create a **makeTimeoutsPromises.js** file in your working directory and open this file with your preferred editor.

To get started, we'll create a function that returns a promise.

☐ 2. Begin with the basic structure, a function that returns an empty promise:

```
makeTimeoutsPromises = () => {
    return new Promise((resolve, reject) =>{});
}
```

A promise will always be in a pending or a settled state. A settled promise will either be resolved or rejected. Once the settled state is reached, a promise cannot be unsettled, it maintains its resolved or rejected state. The resolve and reject parameters in the arrow function inside of `return new Promise()` are callbacks passed in automatically to handle the resolved or rejected state. Your role is to decide when these callbacks should be called and what value they should return.
The simplest possible demonstration follows:

☐ 3. Define a resolve function inside of the returned promise that returns a static value.

```
makeTimeoutsPromises = () => {
    return new Promise((resolve, reject) =>{
        resolve("It is done.");
    });
}
```

☐ 4. Call this function and chain to it a `.then()` function that will display the resolved variable. Run your file to see the result.

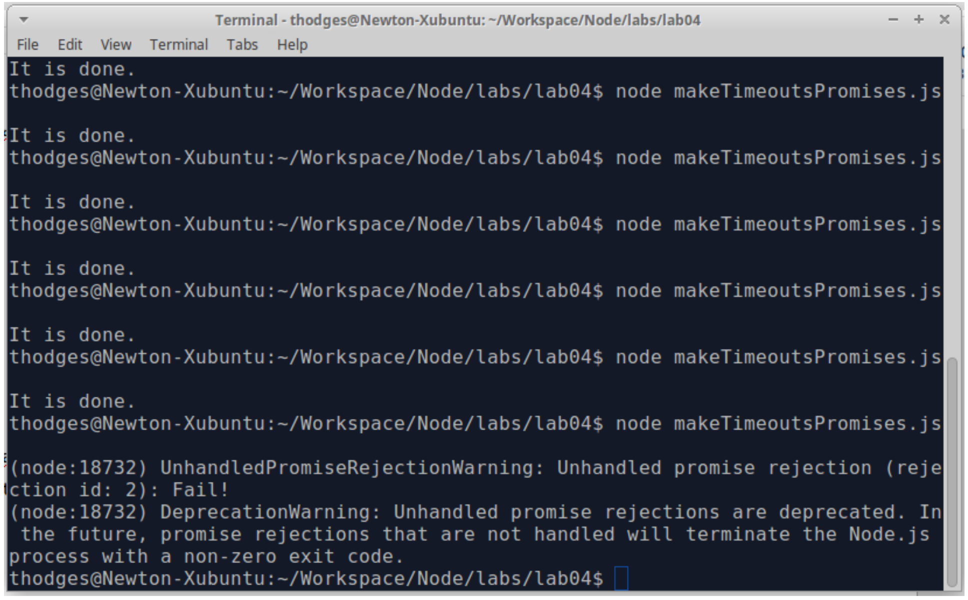
```
makeTimeoutsPromises()
.then(x=>console.log(x));
```

```
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$
```

In this case, because there was no reject potentiality in our returned promise, we could ignore error handling.

☐ 5. To simulate error handling, use random chance, as in the callbacks in part 1, to simulate an error condition. Drop into the terminal and run this a few times to see the result.

```
makeTimeoutsPromises = () => {
    return new Promise((resolve, reject) =>{
        if (Math.random() > 0.8) {
            reject('Fail!');
        } else {
            resolve("It is done.");
        }
    });
}

makeTimeoutsPromises()
.then(x=>console.log(x));
```

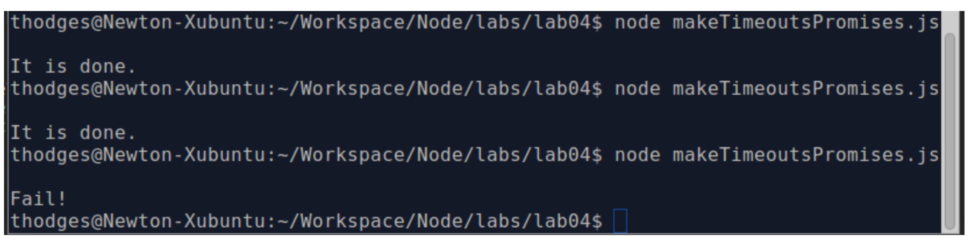


```
Terminal - thodges@Newton-Xubuntu: ~/Workspace/Node/labs/lab04
File  Edit  View  Terminal  Tabs  Help
It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

(node:18732) UnhandledPromiseRejectionWarning: Unhandled promise rejection (reje
ction id: 2): Fail!
(node:18732) DeprecationWarning: Unhandled promise rejections are deprecated. In
 the future, promise rejections that are not handled will terminate the Node.js
process with a non-zero exit code.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$
```

Clearly error handling is need here.

☐ 6. Add a .catch() function to your `makeTimeoutsPromises()` call to handle errors and run your file a few more times to see the result.

```
makeTimeoutsPromises()
.then(x=>console.log(x))
.catch(x=>console.error(x));
```



```
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

It is done.
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$ node makeTimeoutsPromises.js

Fail!
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab04$
```

In the `makeCallbacks()` asynchronous function from part 1, we were able to accept a parameter that we used to make a time delay for returning a value.

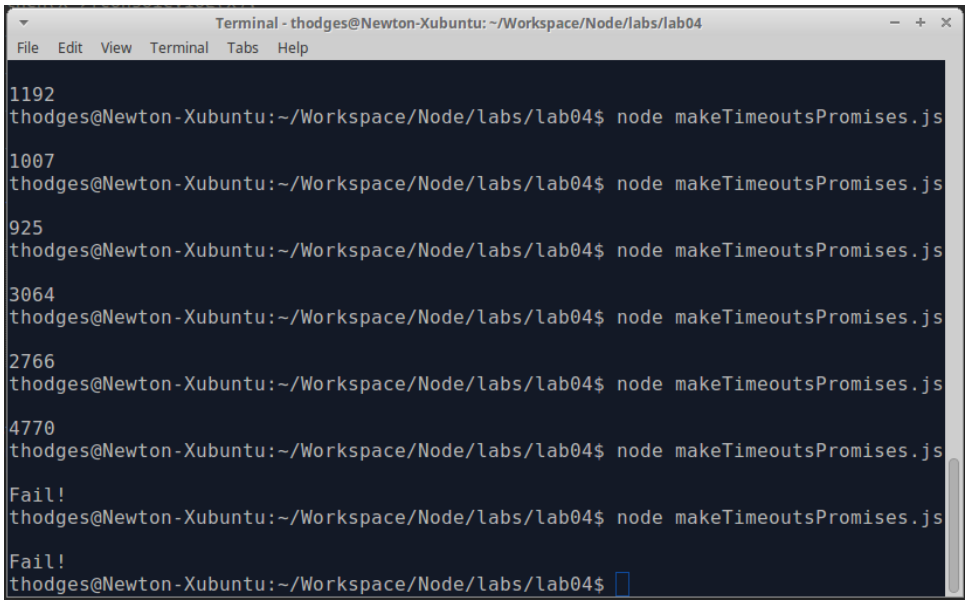
☐ 7. Add a parameter to the `makeTimeoutsPromises` arrow function and wrap all of the code inside of the promise in a `setTimeout()` using that parameter.

```
makeTimeoutsPromises = (time) => {
    return new Promise((resolve, reject) =>{
        setTimeout(() =>{
            if (Math.random() > 0.8) {
                reject('Fail!');
            } else {
                resolve("It is done.");
            }
        }, time);
    });
```

```
}
```

☐ 8. To complete this step, change the resolve value to a random value using the same pattern as in `makeTimeouts.js` and then pass a value to the `makeTimeoutsPromises()` function when you call it. Run this program to see it in action.

```
makeTimeoutsPromises = (time) => {
    return new Promise((resolve, reject) =>{
        setTimeout(() =>{
            if (Math.random() > 0.8) {
                reject('Fail!');
            } else {
                resolve(Math.floor(Math.random() * 5000));
            }
        }, time);
    });
}

makeTimeoutsPromises(1000)
.then(x=>console.log(x))
.catch(x=>console.error(x));
```



Now that we have this structure in place, it's a simple enough task to chain multiple promises together, each of which is dependent on the last.

☐ 9. Copy and paste the `.then()` function so it looks like two are chained together.

```
makeTimeoutsPromises(1000)
.then(x=>console.log(x))
.then(x=>console.log(x))
.catch(x=>console.error(x));
```

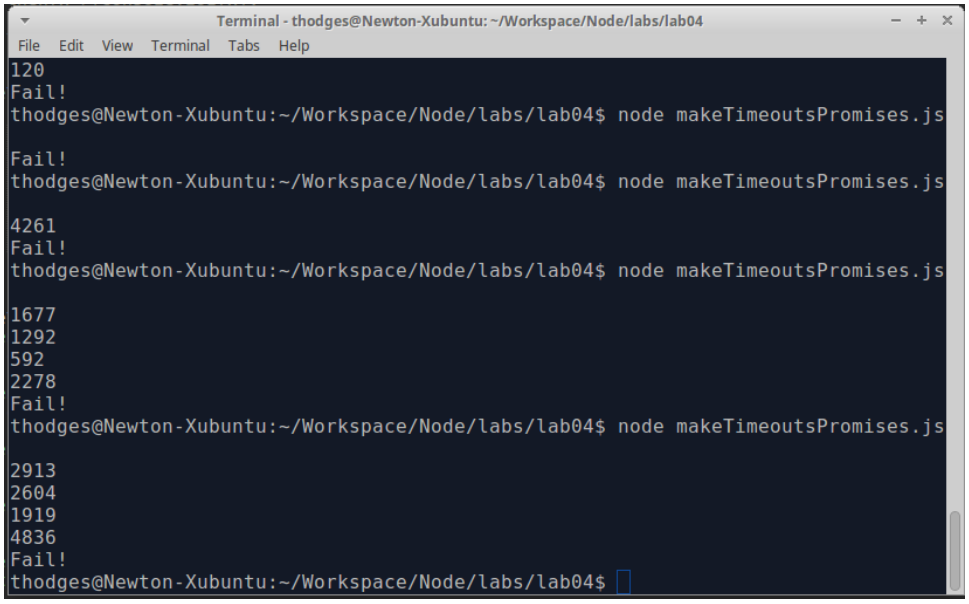
Although resembling chained functions, this code is useless to you as it stands. If you run the file you will see that the first `.then()` (often) returns a value, while the second always returns undefined. Our first `.then()` has handled our settled promise and our second `.then()` has nothing to work with. Think about how you might solve this before going to the next step.

☐ 10. Modify the first `.then()` function to, after logging the value of x to the console, create and return a new promise using the value of x.

```
makeTimeoutsPromises(1000)
.then(x=>{
    console.log(x);
    return makeTimeoutsPromises(x);})
.then(x=>console.log(x))
.catch(x=>console.error(x));
```

☐ 11. Paste three more copies of this first .then() function into the promise chain and test your code.

```
makeTimeoutsPromises = (time) => {
    return new Promise((resolve, reject) =>{
        setTimeout(() =>{
            if (Math.random() > 0.8) {
                reject('Fail!');
            } else {
                resolve(Math.floor(Math.random() * 5000));
            }
        }, time);
    });
}

makeTimeoutsPromises(1000)
.then(x=>{
    console.log(x);
    return makeTimeoutsPromises(x);})
.then(x=>{
    console.log(x);
    return makeTimeoutsPromises(x);})
.then(x=>{
    console.log(x);
    return makeTimeoutsPromises(x);})
.then(x=>{
    console.log(x);
    return makeTimeoutsPromises(x);})
.then(x=>console.log(x))
.catch(x=>console.error(x));
```



You should notice that we have created a cleaner version of the asynchronous callback code from part 1. Notice also that whenever we chance upon a reject, the chain of .then() stop immediately.

For the challenges: everything we have covered so far is part of the normal es6 library and are documented on MDN: **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise**

Additional Promise functions are available by installing and requiring Promise node modules. Some popular modules are q, bluebird and promisejs. The two challenges that follow employ the promisejs module.
To install the module, drop into terminal and issue a command to node package manager:

```
npm install promise -g
```

Then, at the top of makeTimeoutPromises.js, include this module:

```
var Promise = require('promise');
```

If you run makeTimeoutPromises.js again, everything should function normally, as the `.then()` and `.catch()` functions work the same way as they do in the normal es6 libraries.
To complete the challenges, you will need to look at the promisejs api:
https://www.promisejs.org/api/

Challenge: Look as the promisejs api to find out how to use Promise.finally(). Use this to add a message to the end of the above code that will display whether or not one of the promises has settled to a rejected state.

Challenge: Promise.then() and Promise.done() can both handle resolved or rejected promises with onFulfilled and onRejected functions.[1] Replace one `.then()` in your makeTimeoutsPromises.js code with a `.done()`, run the code, and try to work out from the error and the promisejs api why `.done()` doesn't work in this promise chain.

[1] It is generally considered bad form to handle errors inside of Promise.then() in a Promise string because that results in error handlers being scattered throughout your code, just like in Callback Hell. A single Promise.catch() at the end of a string is ideal.

Lab 10: Getting Data with HTTP

In this module, you'll use the http module to do an HTTP `get` request to a server. You'll then use the response stream to log each chunk of data from the server to the console.

☐ 1. Using your code editor, create a new file in the **lab10** folder named **app.js**.

☐ 2. In the new app.js file, require the http module by entering:

```
var http = require('http');
```

☐ 3. On the next line, get the url passed into the program from the command line and store it in a local variable named URL.

```
var urlToGet = process.argv[2];
```

Remember: The first element in the `process.argv` array (`process.argv[0]`) is "node" and the second element (`process.argv[1]`) is the path to the file being run. So, the first argument you pass into the program is the third element in the array, or `process.argv[2]`.

☐ 4. Use the `http.get()` method to make a request to the URL. Press Enter twice after the last code you entered and type:

```
http.get(urlToGet,function(response){
/* do something with the response here */
});
```

☐ 5. Next, enter the following to listen for data events on the response stream.

```
http.get(urlToGet, function(response) {
  response.on("data", function(chunk) {
    /* do something with data here */
  });
});
```

☐ 6. Output each chunk of data to the console as it comes in (replace the line that says `/* do something with the response here */`) with the `console.log` line).

```
http.get(urlToGet, function(response) {
  response.on("data", function(chunk) {
    console.log("CHUNK: " + chunk.toString());
  });
});
```

Remember: The data you get from the server (and that you store in the variable chunk) will be a Node Buffer object. Convert it to a string using the `toString()` method before logging it to the console.

☐ 7. Handle any error events.

```
http.get(urlToGet, function(response) {

    response.on("data", function(chunk) {
        console.log("CHUNK: " + chunk.toString());
    });
}).on('error', function(e) {
    console.log("Got error: " + e.message);
});
```

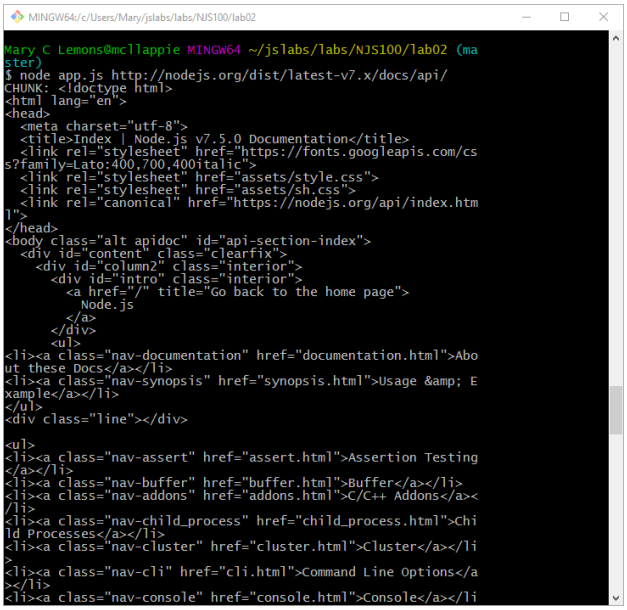
☐ 8. The final program should look like this:

```
var http = require('http');
var urlToGet = process.argv[2];

http.get(urlToGet, function(response) {

    response.on("data", function(chunk) {
        console.log("CHUNK: " + chunk.toString());
    });
}).on('error', function(e) {
    console.log("Got error: " + e.message);
});
```

☐ 9. Make sure you're in the **lab10** folder, and then run the program, passing in a URL from the command line:

node app.js http://nodejs.org/dist/latest-v7.x/docs/api/

The chuck of data from the server will appear in your command line as follows. You have successfully used the HTTP `get` request to pull data from a server.



- ☐ 10. Challenge: Concatenate the chunks together and only output them when all the data has been received.
- ☐ 11. Challenge: Also output the total number of characters in the web page you retrieved with `http.get()`.

Lab 11: Installing and Running a Spark Bot

In this lab, you'll create a bot for Cisco Webex Teams using sample code from CiscoDevNet. You'll use ngrok to allow the bot to communicate with Webex, and you'll configure Webhooks to notify your bot of events that occur on Webex.

☐ 12. Download ngrok from https://ngrok.com/download.

☐ 13. Double-click the ngrok zip file and then open ngrok.exe (on Windows) or extract it and copy it into /usr/local/bin (on Mac) by executing the following command on the command line (make sure you're in the same directory where you expanded the compressed file on Mac only).

```
sudo mv ngrok /usr/local/bin
```

You'll be asked to enter your password. This is the password you use to log into your computer.

☐ **14.** Expose the bot you'll build to the Internet, using http, on port 8080.

```
ngrok http 8080
```

Note: If the above command doesn't work, close and reopen your Terminal emulator (Terminal.app on Mac or cmd.exe on Windows, for example).

☐ 15. Go to https://developer.webex.com/ and sign in if you're not already signed in.

☐ 16. Click your user icon in the upper right corner and select **My Webex Teams Apps**.

☐ 17. Click the Create a New App button.

☐ 18. Click **Create a Bot**.

☐ 19. Fill out the New Bot form. Type in a **Display Name**, such as **My Test Bot**, create a **Bot Username** of your choice (it will automatically show availability of the username you pick). Select an icon or paste a URL for an image you want to use. We used this URL: http://images.clipartpanda.com/robots-clipart-robot5.png.

☐ 20. Write a description for your bot. For example: **"A simple test bot."**

☐ 21. Click Create Bot.

On the next screen, you'll see a generated Bot ID, the information you entered on the previous screen, and a Bot Access Token.

☐ 22. Copy the Bot Access Token and paste it somewhere on your computer. This is the only time you'll see the Access token, so make sure to save it before you move on. Copy and paste the Bot ID somewhere too.

☐ 23. Copy the full bot username (including @webex.bot)

☐ 24. If you don't already have it, download Cisco Webex Teams from https://www.webex.com/downloads.html and follow the instructions to install it.

☐ 25. Open Webex Teams and log in using the same account you created to log into developer.webex.com.

☐ 26. Click the plus sign at the top of the left column to search for a user to add to the room you are creating. Enter your bot's username (including webex.bot) into the search field to create a room with your bot as a participant. Click **Go Chat**.

☐ 27. Open a new terminal window (open Git Bash in Windows), leaving ngrok running in the one where you started it, and download the Cisco Developer Network Webex Teams Sparkbot samples with this command:

```
git clone https://github.com/CiscoDevNet/node-sparkbot-samples
```

☐ 28. Change the newly downloaded directory to the working directory.

```
cd node-sparkbot-samples
```

☐ 29. Install the samples.

```
npm install
```

☐ 30. Make the examples directory the working directory.

```
cd examples
```

☐ 31. Run the helloworld bot by typing this command, replacing *token* with your bot's access token.

```
ACCESS_TOKEN=token DEBUG=sparkbot* node helloworld.js
```

Note: Make sure there are no spaces around your token or it will not work.

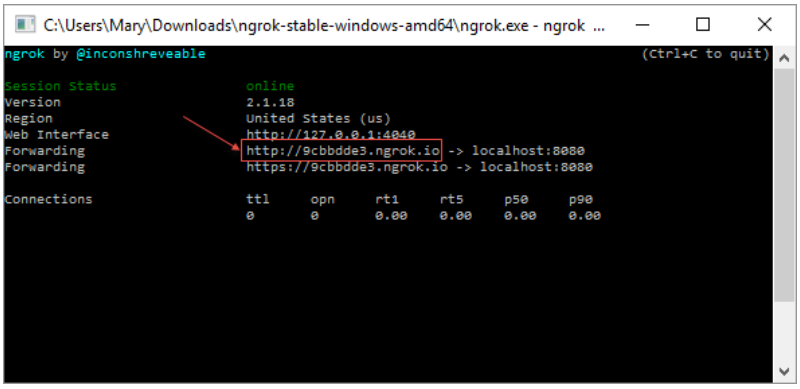
☐ 32. Go to the Cisco Spark Create a Webhook documentation at this address:

https://developer.ciscospark.com/endpoint-webhooks-post.html

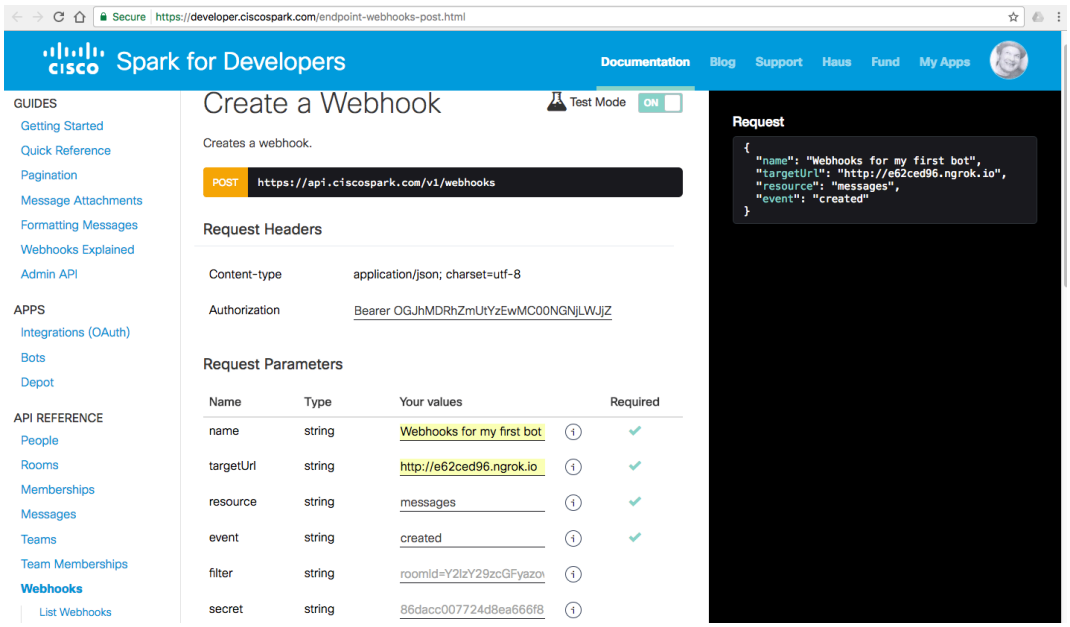
☐ 33. Change the authorization code (leave Bearer in place) to the bot's Access Token.

☐ 34. Enter a name (anything will do).

☐ 35. Make the **target url** be your ngrok Forwarding url that's displayed in the ngrok command-line window (something like https://25bcc582.ngrok.io)

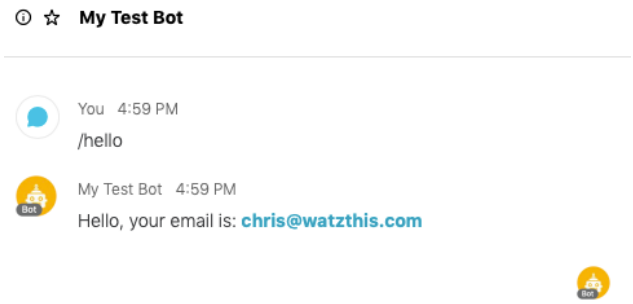


- ☐ 36. Set resources to **messages**.
- ☐ 37. Set event to **created**.



- ☐ 38. Click **Run**. Your first Webhook will be created, which will listen for messages to your bot.
- ☐ 39. Next, modify the resource field in this same form to create a Webhook that will listen for when your bot is added to a room by changing resources to **memberships**.
- ☐ 40. Click **Run** again to create a second webhook.
- ☐ 41. Go into the Webex Teams room you created with your bot and type **/hello**.

If everything works, your bot will respond to the message.



Note: When you finish this lab, you can stop your bot from running by pressing **Ctrl+C**. Leave ngrok running for the next lab. If you have to stop ngrok and restart it, you'll get a new URL and you'll need to update your Webhooks using the Update a Webhook form here: https://developer.ciscospark.com/endpoint-webhooks-webhookId-put.html.

Lab 12: Making a Hello World Bot

In this lab, you'll program your first bot, which will just say hello when it's started up.

Note: Before completing this lab, make sure that ngrok is running and that your Webhooks are properly configured for your bot. (See the previous lab.)

☐ 42. Open your terminal application and type the following to change the working directory to **labs/lab12**.

```
cd labs/lab12
```

☐ 43. Initialize npm in the directory by typing:

```
npm init
```

Accept the defaults, and then press **Ctrl+C** to exit if needed.

☐ 44. Install node-sparkclient, by typing

```
npm install --save-dev node-sparkbot node-sparkclient
```

☐ 45. In your code editor, create a new file named **hellobot.js** inside the **lab12** folder.

☐ 46. Require node-sparkbot and node-sparkclient in **hellobot.js** by typing the following:

```
var SparkBot = require("node-sparkbot");
var SparkAPIWrapper = require("node-sparkclient");
```

☐ 47. Create an instance of the SparkBot

```
var bot = new SparkBot();
```

☐ 48. Write a statement to check that the program was started correctly, with the `ACCESS_TOKEN` variable.

```
if (!process.env.SPARK_TOKEN) {
    console.log("This bot requires a Cisco Webex Teams
API access token.");
    console.log("Please add env variable ACCESS_TOKEN
on the command line");
    console.log("Example: ");
    console.log("> ACCESS_TOKEN=XXXXXXXXXXXX
DEBUG=sparkbot* node hellobot.js");
    process.exit(1);
}
```

☐ 49. Create an instance of the node-sparkclient with your bot's `id`.

```
var spark = new
SparkAPIWrapper(process.env.ACCESS_TOKEN);
```

☐ 50. Make an event handler that will listen for users being added to rooms and ignore anyone who is added to the room who isn't this bot.

```
bot.onEvent("memberships", "created", function
(trigger) {
    var newMembership = trigger.data; // see specs
here: https://developer.ciscospark.com/endpoint-
memberships-get.html
    if (newMembership.personId !=
bot.interpreter.person.id) {
        // ignoring
        console.log("new membership fired, but it is
not us being added to a room. Ignoring...");
        return;
    }
```

☐ 51. Display a message in the console if this bot is added to a room.

```
console.log("bot was just added to room: " +
trigger.data.roomId);
```

☐ 52. Write a `createMessage` method that will post "Hello, World!" to the room and that will display an error message if the `createMessage` method doesn't work.

```
spark.createMessage(trigger.data.roomId, "Hi, I am the
Hello World bot! I just say Hello.", { "markdown":true
}, function(err, message) {
    if (err) {
        console.log("WARNING: could not post Hello
message to room: " + trigger.data.roomId);
        return;
    }
});
});
```

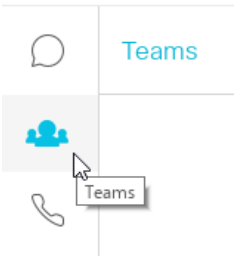
☐ 53. Save your **hellobot.js** file if necessary.

☐ 54. In your console or terminal application, change the working directory to **lab12** if it's not already open and start your bot:

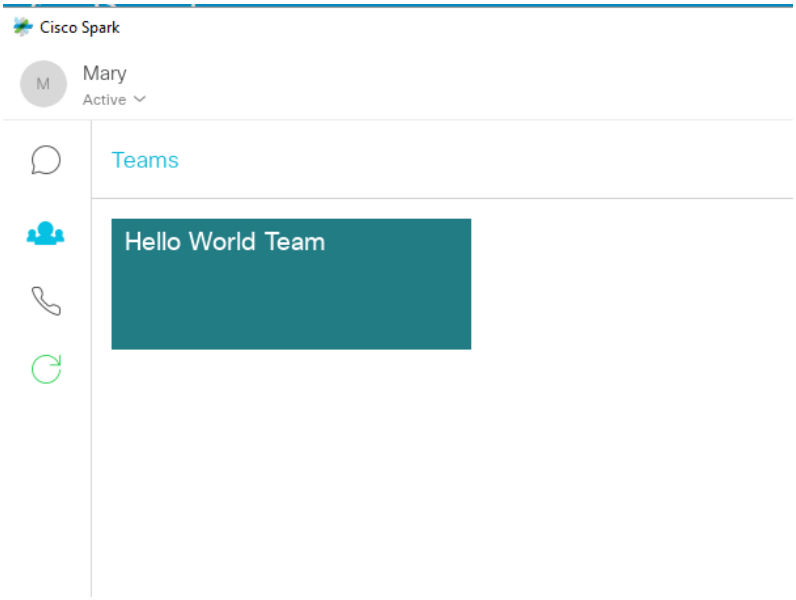
```
ACCESS_TOKEN=XXXXXXXXXXXX DEBUG=sparkbot* node
hellobot.js
```

Note: In the above command, replace the Xs after ACCESS_TOKEN with your bot's Access Token.

☐ 55. Go to your Webex Teams app and click the **Teams** button.

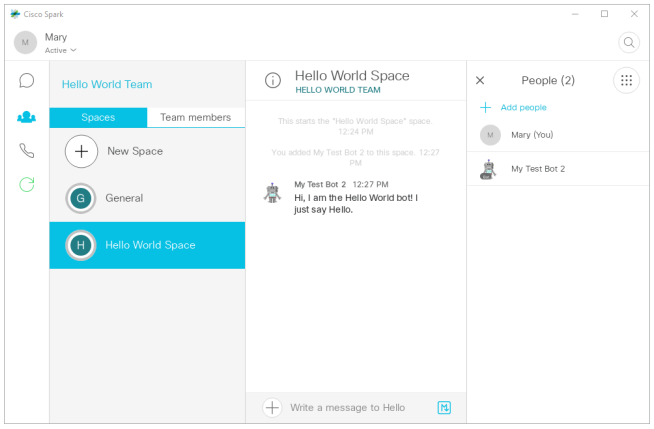


☐ 56. Click New Team.
☐ 57. Name your team. We'll call ours **Hello World Team**. Click **Create** after you've named your team.



☐ 58. Click the **Hello World Team** team (or whatever you named yours).
☐ 59. Click the **+** button to create a new space, and give it a name. We'll call ours Hello World Space. Press **Enter**. The space opens in the right pane.
☐ 60. Click the button with nine dots in it, and then click **People**.

☐ 61. Click **Add people** and then select the test bot you created earlier.
☐ 62. You should get an automatic response from your test bot:



☐ 63. Write a new event listener to make your bot report the time when it's asked.

Hint: Look at the code from the previous lab to see how to find out how to make a Bot respond to a command.

Remember: You'll need to stop and restart your bot for the changes to be reflected in Webex Teams.

Lab 13: Testing Node.js

Node's built-in **assert** module can be used for basic testing of expressions. It checks if the outputs from a function match the expected outputs for a given set of inputs. Assert has many methods, the most useful of which are `assert.equal()`, `assert.deepEqual()`, `assert.ifErr()`, `assert.throws()`. In this lab, you'll use some of the methods of the assert module to test an existing function.

Part 1: assert.equal()

In your **lab13** directory, you should find a file called `sumModule.js`. Here is what is inside:

```
const sumModule = (number1, number2, callback) => {
  var sum = number1 + number2;
  if (isNaN(sum)) {
    callback("sum is not a number");
  } else if ((number1 <= 0) || (number2 <= 0)) {
    callback("all inputs must be positive")
  }
  callback(null,sum);
};

module.exports = sumModule;
```

This file contains a module that takes two inputs and returns either the sum of the two inputs (if they are positive numbers) or an error if either of the inputs is not a number or not positive. Recall that in mathematics, 0 is neither positive nor negative, so positive real numbers are strictly defined as those numbers greater than 0.
This module provides a perfect testing ground for working with asserts.

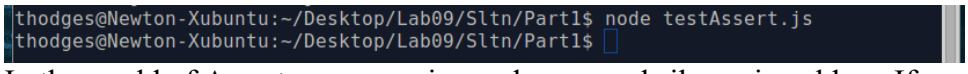
- ☐ 1. Create a new file in your working directory called **testAssert.js** and open it in your preferred editor.
- ☐ 2. Start off by importing `sumModule` and assert:

```
const sumModule = require('./sumModule');
const assert = require('assert');
```

- ☐ 3. Call `sumModule`, passing a callback which tests `assert.equal()`:

```
sumModule(1, 2, (err, data) => {
  assert.equal(data,3);
});
```

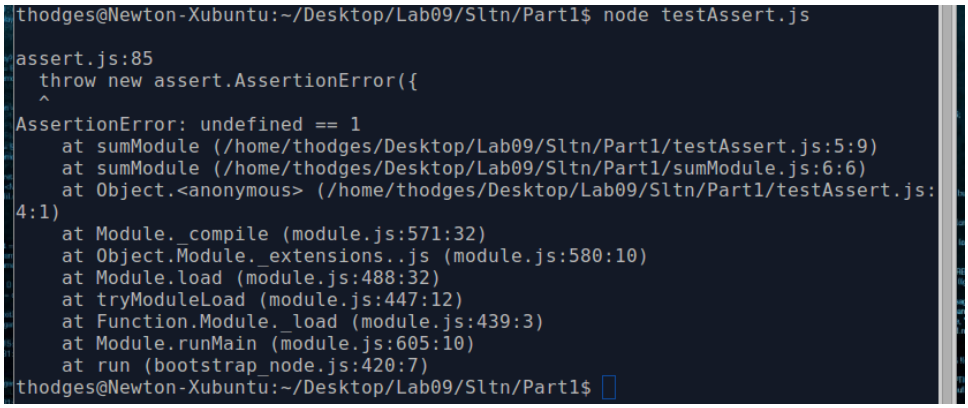
- ☐ 4. Drop into the terminal and run **testAssert.js** in node and you will see.....nothing.



In the world of Asserts, no news is good news and silence is golden. If you see no output, that means that your assertion passed.

☐ 5. Modify the code so that the assert will fail and try again – pass in a negative number. Then run it again:

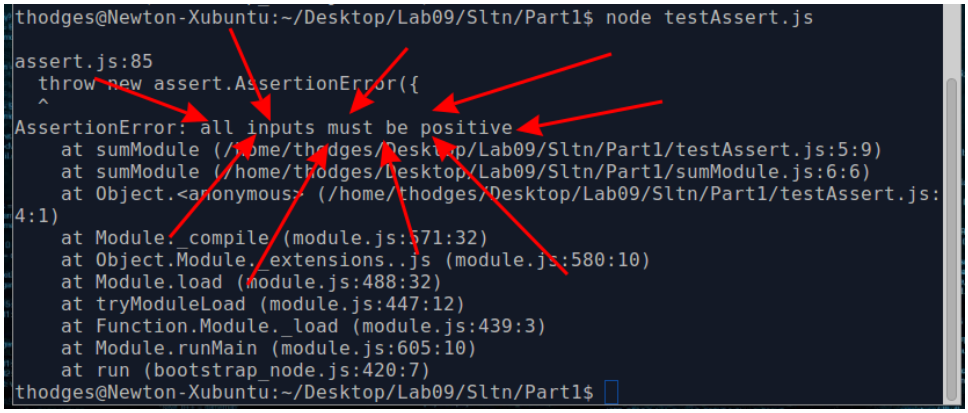
```
sumModule(-1, 2, (err, data) => {
  assert.equal(data,1);
});
```



We see an error, but this is of no use to us unless we can see why the error is thrown! `Assert.equal()` takes three parameters, the two values to be compared, and the third the error message to accompany a failure.

☐ 6. Modify **testAssert.js** to accept an error parameter as being the error message passed back from `sumModule`, and run the code again for a more satisfying result:

```
sumModule(-1, 2, (err, data) => {
  assert.equal(data, 1, err);
});
```



☐ 7. Now try modifying the sumModule call to get a different error:

```
sumModule('one', 2, (err, data) => {
  assert.equal(data, 3, err);
});
```

```
thodges@Newton-Xubuntu:~/Desktop/Lab09/Sltn/Part1$ node testAssert.js

assert.js:85
  throw new assert.AssertionError({
  ^
AssertionError: sum is not a number
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/testAssert.js:5:9)
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/sumModule.js:4:9)
    at Object.<anonymous> (/home/thodges/Desktop/Lab09/Sltn/Part1/testAssert.js:
4:1)
    at Module._compile (module.js:571:32)
    at Object.Module._extensions..js (module.js:580:10)
    at Module.load (module.js:488:32)
    at tryModuleLoad (module.js:447:12)
    at Function.Module._load (module.js:439:3)
    at Module.runMain (module.js:605:10)
    at run (bootstrap_node.js:420:7)
thodges@Newton-Xubuntu:~/Desktop/Lab09/Sltn/Part1$
```

☐ 8. Before we move on, try feeding `assert.equal()` false information about the expected value of a sum and reading the ouput:

```
sumModule(1, 2, (err, data) => {
  assert.equal(data, 1, err);
});
```

```
thodges@Newton-Xubuntu:~/Desktop/Lab09/Sltn/Part1$ node testAssert.js

assert.js:85
  throw new assert.AssertionError({
  ^
AssertionError: 3 == 1
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/testAssert.js:13:9)
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/sumModule.js:8:5)
    at Object.<anonymous> (/home/thodges/Desktop/Lab09/Sltn/Part1/testAssert.js:
12:1)
    at Module._compile (module.js:571:32)
    at Object.Module._extensions..js (module.js:580:10)
    at Module.load (module.js:488:32)
    at tryModuleLoad (module.js:447:12)
    at Function.Module._load (module.js:439:3)
    at Module.runMain (module.js:605:10)
    at run (bootstrap_node.js:420:7)
thodges@Newton-Xubuntu:~/Desktop/Lab09/Sltn/Part1$
```

☐ 9. Compare this error, `3 == 1`, to the error that you got the first time you produced an error but hadn't modified the code to display an error value. Comparing that error message to this one, see if you can work out why the return was `undefined == 1`.

Part 2: assert.ifError()

1. The final assert method that we will try out today is `assert.ifError()`. This only throws true values, and is therefore useful for testing the first (error) parameter of callbacks.

```
sumModule('one', 2, (err,data) => {
  assert.ifError(err);
});
```

```
thodges@Newton-Xubuntu:~/Desktop/Lab09/Sltn/Part1$ node testAssert.js

assert.js:372
assert.ifError = function ifError(err) { if (err) throw err; };
                                                 ^

sum is not a number
thodges@Newton-Xubuntu:~/Desktop/Lab09/Sltn/Part1$
```

In this case, the results are much more concise.

Challenge:
Modify **`sumModule.js`** to accept negative values, and then test your new version with

```
sumModule(-1, 2, (err, data) => {
  assert.equal(data, 1, err);
});
```

Part 3: Mocha and Should.js

Now let's convert these tests to use the Mocha test framework and the should.js assertion library.

☐ 1. Initialize npm.

```
npm init --yes
```

This will create a package.json file with the default settings in your directory.

☐ 1. Install mocha and should.js.

```
npm install --save-dev mocha should
```

☐ 2. Create a new directory named **test**.
☐ 3. Make a file named **sumModule.test.js** inside the **test** directory.
☐ 4. Inside **sumModule.test.js**, require mocha, should, and sumModule.js.

```
const sumModule = require('../sumModule');
const mocha = require('mocha');
const should = require('should');
```

☐ 5. Create a test suite inside sumModule.test.js, using Mocha's `describe` function.

```
describe('sumModule', function() {

});
```

☐ 6. Create a spec (also known as a test) using Mocha's `it` function.

```
describe('sumModule', function() {
    it('should add numbers together', function(done){
```

```
    });
});
```

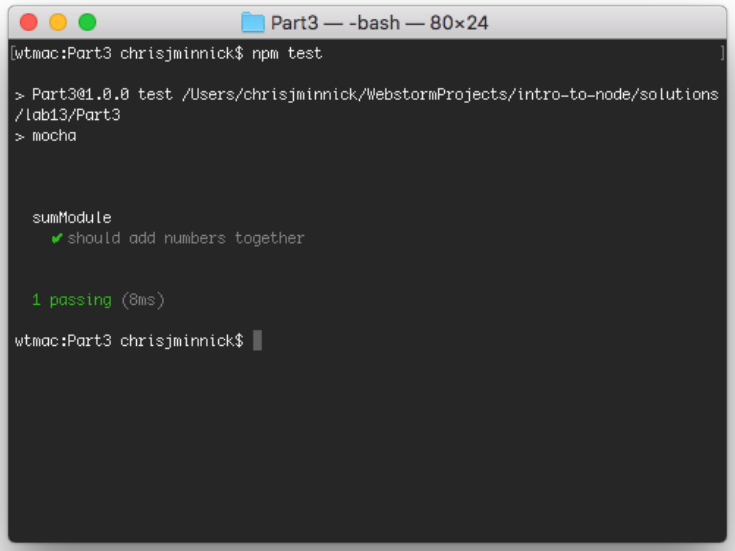
☐ 7. Write an assertion using `should` inside the test suite.

```
describe('sumModule', function () {
    it('should add numbers together', function(done) {
        sumModule(2, 2, function (err, result) {
            result.should.equal(4);
            done();
        });
    });
});
```

☐ 8. Inside `package.json`, modify the `test` script to run mocha.

```
"scripts": {
  "test": "mocha"
},
```

☐ 9. Run your test by entering **npm test** in the terminal.



☐ 10. Write a second test (within the same spec) that tests whether the correct error is returned when one of the arguments passed to `sumModule` is negative.

☐ 11. Make a test suite with at least one test for a function called `productModule` that should multiply two numbers together. After verifying that this test fails, proceed to make the module to pass this test.

Lab 14: Express

Part 1: Basic Setup and Routing

Express is a versatile framework that runs under Node.js. It defines a routing table to respond to HTTP Get, Post, Delete or Put methods, and has a templating engine that can dynamically render pages based on arguments passed in.
We'll begin our journey by constructing a barebones express server.

☐ 1. Use node package manager to install the express module and its dependencies.

```
npm install express
```

☐ 2. In your working directory, create a file called `basicExpress.js` and open it in your favorite editor. Like usual, we have to start by including the express package.

```
var express = require('express');
```

☐ 3. Now we have to set up an express app that will have all the methods that we need to use. Call the express function and assign it to app:

```
var app = express();
```

☐ 4. Then set up your app to listen on an open port, for example, port 8080:

```
app.listen(8080);
```

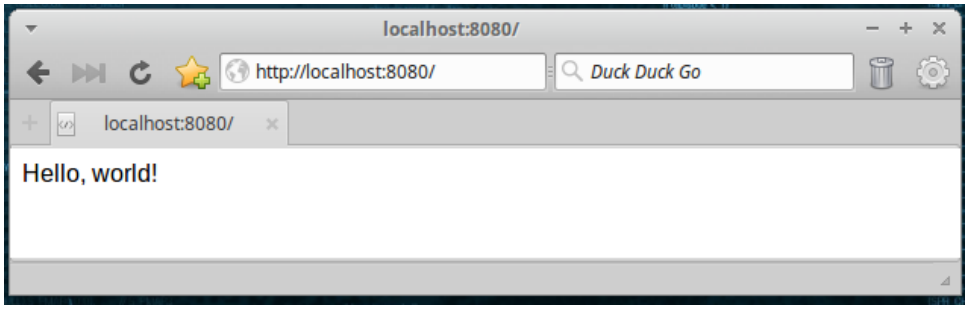
You may already be familiar with the basic HTTP methods GET, POST, DELETE and PUT. Express has handlers for all of these methods, and these handlers are called with two parameters: a route and a function. The function has two parameters, `req` and `res`, which correspond to the request and the response.

☐ 5. Create the most basic implementation of `app.get()`:

```
app.get('/', (req, res) => {
    res.send('Hello, world!');
});
```

This function uses `res.send()` to send a string to the browser. The route is specified as / and the arrow function contains the method to respond to the request.

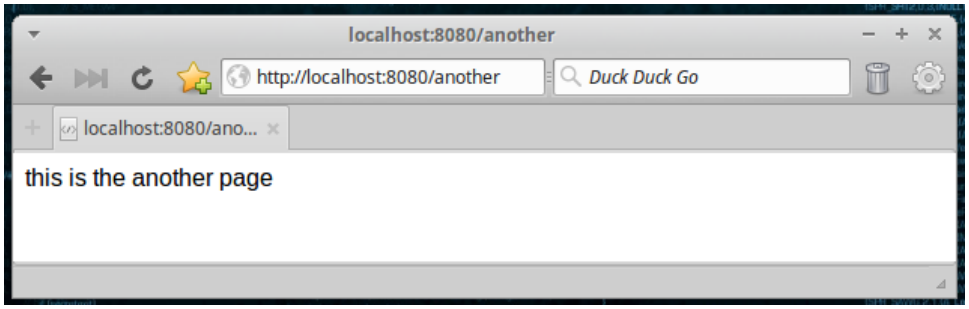
☐ 6. In your web browser, open `localhost:8080` to see the result.



☐ 7. Copy and paste the `app.get()` code above and make a new route with a different message:

```
app.get('/', (req, res) => {
    res.send('Hello, world!');
});

app.get('/another', (req, res) => {
    res.send('this is the another page');
});
```

☐ 8. Interrupt and restart this program and open your new page in a browser to see the result:



Part 2: Handling GET Requests

In your working directory, you will find a file called `get_request.html`. Here is what it contains:

```
<html>
   <body>
      <form action = "http://localhost:8080/getRequest" method = "GET">
         Moby Dick: <input type = "text" name = "moby">  <br>
         Ishmael: <input type = "text" name = "ishmael"> <br>
         Ahab: <input type = "text" name = "ahab"> <br>
         <input type = "submit" value = "Submit">
      </form>
   </body>
</html>
```

We are going to use Express to serve up this file and then process the GET request. Looking at the above code, you likely have guessed the eventual porpoise of this experiment, so you had best stop blubbering or we'll switch to The Humpback of Notre Dame.

☐ 9. Create a new file called `getRequestExpress.js` and paste in all of your code from `basicExpress.js`, omitting the second `app.get()` (for `'/another'`).

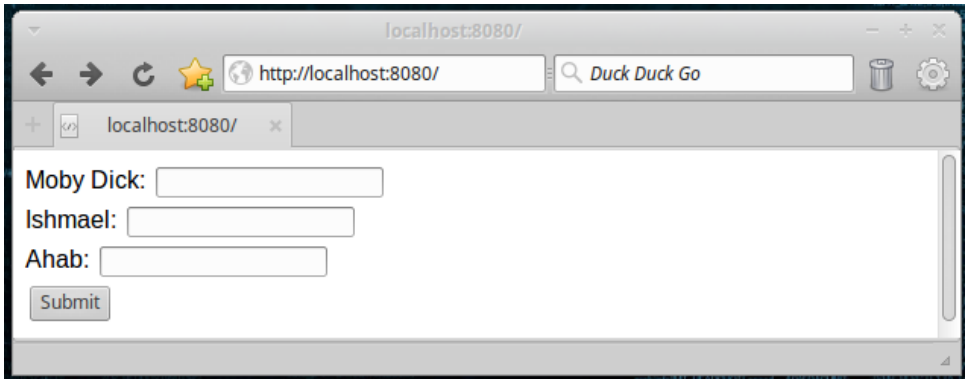
```
var express = require('express');
var app = express();
app.listen(8080);
app.get('/', (req, res) => {
    res.send('Hello, world!');
});
```

We need to modify `app.get` to serve up the `get_request.html` file from the current directory.

☐ 10. In the `app.get()` function, replace `res.send()` with `res.sendFile()`, specifying the location of `get_request.html`:

```
app.get('/', (req, res) => {
    res.sendFile( __dirname + '/get_request.html' );
});
```

☐ 11. Cancel the node process for `basicExpress.js` (if it is still running), start up `getRequestExpress.js`, and load `localhost:8080` in your browser.



If you fill in this form and hit Submit you'll get the following error:
`Cannot GET /getRequest`
Let's make a handler for this route.

☐ 12. Copy and paste the `app.get()` code for `/` and change the route to `/getRequest`:

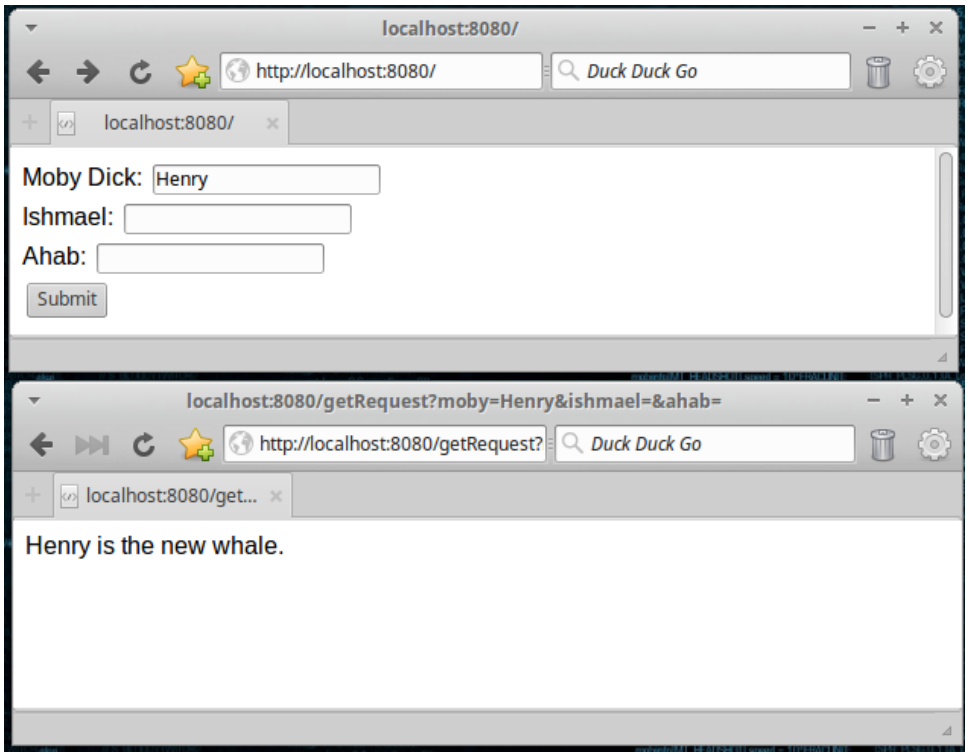
```
app.get('/', (req, res) => {
    res.sendFile( __dirname + "/" + "get_request.html"
);
});

app.get('/getRequest', (req, res) => {
```

```
    res.sendFile( __dirname + "/" + "get_request.html"
);
});
```

☐ 13. Replace `res.sendFile()` in the new route with `res.send()` and the following content:

```
app.get('/getRequest', (req, res) => {
    res.send(req.query.moby + " is the new whale.");
});
```

The input field in our form for Moby Dick has `name="moby"`. The contents of this field is stored in the `req` parameter under `req.query.moby`. With a small amount of thought, you may be able to work out how to access the contents of the Ishmael and Ahab fields as well.
If you stop and restart this node process, reload `localhost:8080`, enter a value for Moby Dick and press submit you can test your code.



Examine the URL to see the Henry value (or whatever name you chose) encoded there.

Part 3: Handling POST Requests

In a POST request, data is passed in the HTTP message body rather than the URL. Handling these in Express is only slightly different from handling GET requests.

☐ 1. Create a new file called `post_request.html`. Paste in the contents of `get_request.html` but change the action url to route to `/postRequest` and the method to `POST`:

```
<html>
   <body>
      <form action = "http://localhost:8080/postRequest" method = "POST">
         Moby Dick: <input type = "text" name = "moby">  <br>
         Ishmael: <input type = "text" name = "ishmael"> <br>
         Ahab: <input type = "text" name = "ahab"> <br>
         <input type = "submit" value = "Submit">
      </form>
   </body>
</html>
```

☐ 2. Create a new file called **postRequestExpress.js**. Paste in the contents of **getRequestExpress.js** and change the route for '**/**' to **post_request.html**:

```
var express = require('express');
var app = express();
app.listen(8080);
app.get('/', (req, res) => {
    res.sendFile( __dirname + "/" + "post_request.html"
);
});
app.get('/getRequest', (req, res) => {
    res.send(req.query.moby + " is the new whale.");
});
```

☐ 3. Now modify the second get request to a post request, replacing `app.get()` with `app.post()`, `/getRequest` with `/postRequest`, and `req.query.moby` with `req.body.moby`.

```
app.post('/postRequest', (req, res) => {
    res.send(req.body.moby + " is the new whale.");
});
```

If you run this file, enter a name for Moby and try to run the file you'll get an error:

```
TypeError: Cannot read property 'moby' of undefined
```

I'll bet that you expected that to work on the first try. It so happens that Node 4 and above require a middleware layer called `bodyParser` to handle POST requests. Previously this was part of the Express package.

☐ 4. Drop into the terminal and use node package manager to grab `bodyParser`:

```
npm install body-parser
```

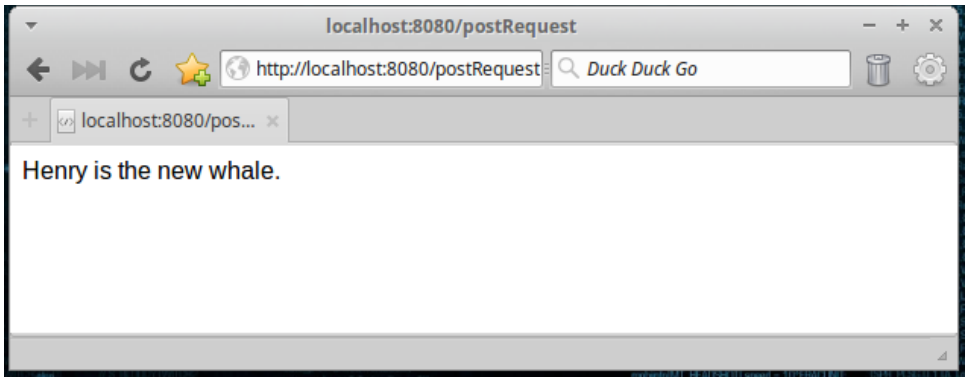
☐ 5. Include body-parser in `postRequestExpress.js`.

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
```

☐ 6. At this point we have to configure express to use `bodyParser` as middleware:

```
app.use(bodyParser.urlencoded({extended: false}));
app.use(bodyParser.json());
app.listen(8080);
```

☐ 7. Now when you run `postRequestExpress.js`, everything should magically work.



Notice that the data from the form is no longer encoded in the URL.

Part 4: Wiring up a Stream.

This is the moment you have been waiting for, wiring up the streams that we created back in the process lab with Express. It is recommended that if you are lost by any of these explanations, that you review the labs on Streams and Pipes.
While we could use either GET or PUT to make our transform streams, I'm going to advocate for GET in this case, because then our narrative will be directly accessible by a unique URL that can be posted to other people's FaceBook walls.

☐ 1. Create a new file called `mobyExpress.js` and copy over everything from `getRequestExpress.js`.

```
var express = require('express');
var app = express();
app.listen(8080);
app.get('/', (req, res) => {
    res.sendFile( __dirname + "/" + "get_request.html"
);
});

app.get('/getRequest', (req, res) => {
    res.send(req.query.moby + " is the new whale.");
});
```

☐ 2. Copy **makeTransformer.js** and **MobyDick.txt** from the directory for the process lab over to your working directory, and include `makeTransformer` and the `fs` module in **mobyExpress.js**:

```
var express = require('express');
var fs = require('fs');
var makeTransformer = require('./makeTransformer');
```

☐ 3. Create a read stream for **MobyDick.txt** with `utf8` encoding. Remember that the read stream will sit waiting for as long as we like it to before we request chunks of data from it.

```
var app = express();
var mobyReadStream = fs.createReadStream(__dirname +
'/MobyDick.txt', 'utf8');
```

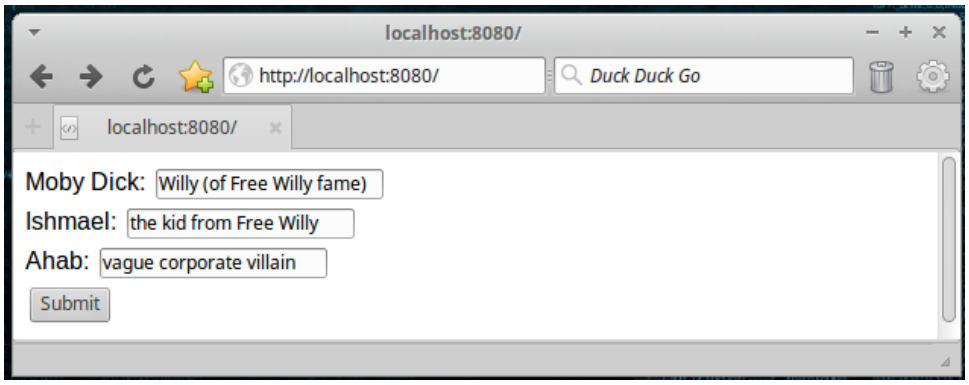
☐ 4. We are going to define our transform streams inside of our `app.get()` function corresponding to '`/getRequest`.' I'm not going to bother to change the route name because I would like to use the file already created for **get_request.html**, and that has '`/getRequest`' hard coded.

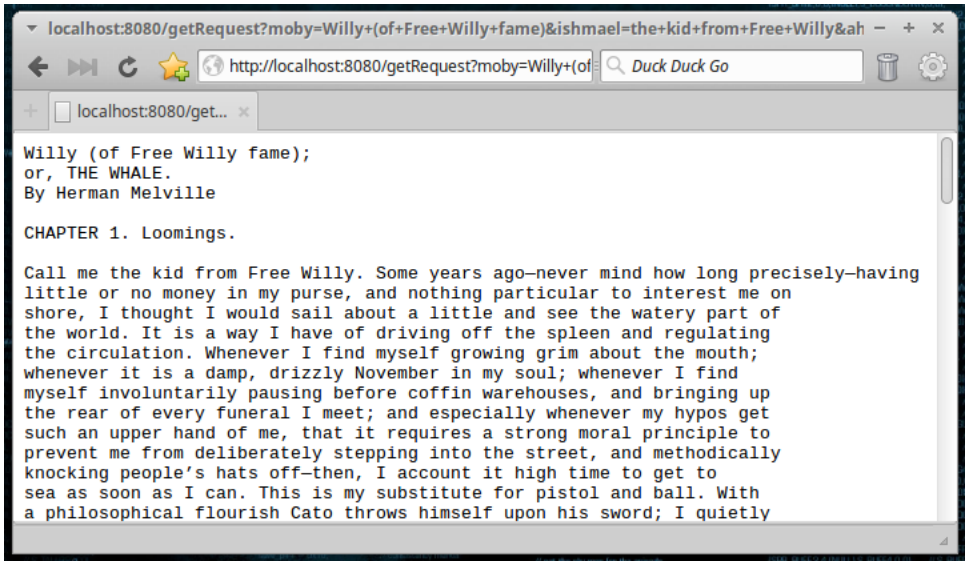
```
app.get('/getRequest', (req, res) => {
    var newMoby = req.query.moby;
    var newIshmael = req.query.ishmael;
    var newAhab = req.query.ahab;

    var mobyTransformer = makeTransformer(new
RegExp(/Moby\s*Dick/), (newMoby || 'Moby Dick'));
    var ishmaelTransformer = makeTransformer(new
RegExp(/Ishmael/), (newIshmael || 'Ishmael'));
    var ahabTransformer = makeTransformer(new RegExp(/Ahab/),
(newAhab || 'Ahab'));
});
```

☐ 5. Wire up the pipes using the `mobyReadStream` read stream, our transformers, and `res` as a write stream:

```
mobyReadStream
     .pipe(mobyTransformer)
     .pipe(ishmaelTransformer)
     .pipe(ahabTransformer)
     .pipe(res);
```

☐ 6. Start up `mobyExpress.js`, load up `localhost:8080` in your browser, and try this out!





Challenge: You may notice that if you hit the back button and type in new values for Moby Dick, Ishmael and Ahab and hit submit that you only get a blank page. Try to work out why this is happening and find a fix.

Challenge: Rewrite part 4 using POST instead of GET.

Lab 15: Working with MongoDB Databases

Node can access many kinds of databases. One of the most popular is MongoDB. In this lab, you'll install and configure MongoDB, create a database, and create a simple Web Application using Express and MongoDB.

☐ 1. Check whether Mongo is installed on your computer by typing mongo --version into your command line.
☐ 2. If it's not already installed, download MongoDB Community Server from https://mongodb.com/download and following the instructions linked from the download page.
☐ 3. Create a data directory, if necessary:

- On Windows, the data directory should be C:\data\db
- On Mac, the data directory should be /data/db

☐ 4. In your terminal program, start a mongod process with the following command:

```
mongod --dbpath=/data
```

☐ 5. Type `mongo` to start the command line interface (CLI).
☐ 6. Once you're in the Mongo CLI, type `show dbs` to list all your existing databases.
☐ 7. In your Lab15 directory, initialize npm.

```
npm init
```

☐ 8. Install the MongoDB driver and express

```
npm install --save mongodb express
```

☐ 9. Create a file named **`app.js`**.
☐ 10. Enter the following code into app.js to create a simple server.

```
const express = require('express');
const mongodb = require('mongodb').MongoClient;
const app = express();
app.listen(5000, function(err){
    console.log('running server');
});
```

☐ 11. In app.js, create an array of data that will be inserted into the database, and set the URL for your database.

```
const express = require('express');
const mongodb = require('mongodb').MongoClient;
const app = express();
```

```
const books = [
    {title:'East of Eden'},
    {title:'War and Peace'}
    ];
const url = 'mongodb://localhost:27017/booksApp';
app.listen(5000, function(err){
    console.log('running server');
});
```

☐ 12. Create the addData route.

```
app.get('/addData', function(req,res){
    mongodb.connect(url,
        function(err,database) {

        let myDb = database.db('booksApp');
        let collection =
                myDb.collection('books');

        collection.insertMany(books,
            function(err,results){
                res.send(results);
                myDb.close();
            });
    });
});
```

☐ 13. Run app.js

```
node app.js
```

☐ 14. Open a web browser and go to http://localhost:5000/addData

You should see your data, which is now in the database you created.

☐ 15. Open a new Terminal window and view the contents of the books collection.

```
mongo booksApp
> show collections
> db.books.find()
```

☐ 16. Create the /viewAll Route in app.js.

```
app.get('/viewAll', function(req,res){
    mongodb.connect(url,function(err,database){
        let myDb = database.db('booksApp');
        let collection =
             myDb.collection('books');

        collection.find({}).toArray(
            function(err, results) {
```

```
                    res.send(results);
                }
            );
        });
    });
```

☐ 17. Stop and start app.js.

☐ 18. Go to http://localhost:5000/viewAll in your browser.

You'll see the array that you inserted into the books collection!

☐ 19. Try the following challenge exercises:

- Select just one book from the database.
- Display the book information in an HTML template
- Create a form to insert new records
- Make a route that displays a list of books, with a link next to each to a separate route that displays just one book.

Lab 16: Make Me hapi

Hapi is a framework for building web applications and servers. In this lab, you'll learn about hapi using a self-guided workshop called makemehapi.

To get started, go to https://www.npmjs.com/package/makemehapi and follow the instructions to install makemehappi. When it's installed, run it by typing makemehapi on the command line.

Do as many of the exercises as you have time for. After each one verify it with the makemehapi program.

Note 1: You may need to preface the command to install makemehapi with sudo if you're using MacOS.

Note 2: If launching makemehapi doesn't work on Windows, you will need to locate the makemehapi.cmd file and add it to the system path. Make sure to restart your terminal or cmd.exe program after doing this.