BACK-END JAVASCRIPT LITERACY - LABS

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

`https://github.com/watzthisco/jslabs`

Version 1.0, June 2017
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, 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

WatzThis?
PO Box 161393
Sacramento, CA 95816
info@watzthis.com
www.watzthis.com

Help us improve our courseware

- Please send your comments and suggestions via email to info@watzthis.com or post to the course issues here:
- https://github.com/watzthisco/jslabs/issues

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. His current online courses, Creating Mobile Apps with HTML5 and Achieving Top Search Engine Placements, are consistently among the most popular courses offered by online training provider Ed2Go.com.

As a writer, 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 winemaker, a contributor to several blogs (including chrisminnick.com), and an avid swimmer, cook, and musician.

Introduction and Setup Instructions

Course Requirements

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

- 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. Go to *https://nodejs.org* and download the latest version of Node from the LTS (Long Term Support) branch.
2. Install a code editor. We use Atom in the course (https://atom.io/).
3. Make sure Google Chrome is installed (https://www.google.com/chrome/).
4. Install Git on each student's computer. Git can be downloaded from https://git-scm.com/downloads. Select all default options during installation. Be sure to choose the correct download for the operating system being used.

Testing the Setup

1. Open a command prompt.
   - MacOS: Use Terminal (/Applications/Utilities/Terminal).
   - Windows:
     a. Option 1 (preferred): Install Bash (Windows 10 only) per the instructions here: http://www.windowscentral.com/how-install-bash-shell-command-line-windows-10
     b. Option 2: Use Git Bash on Windows (installed with Git). Click **Start** and type **Git Bash**, and then select it when it appears in the Start menu.
2. Type **`cd`** and press **Enter** to navigate to the user's home directory (or change to a directory where student files should be created).
3. Type the following command and press **Enter**:

```
git clone https://github.com/watzthisco/jslabs.git
```

The lab solution files for the course will download into a new directory called jslabs.

1. Type **`cd jslabs`** and press **Enter** to switch to the new directory.
2. Type **`npm install`**, and press **Enter**.

This step will 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.

1. When everything is done, type **`npm run test`** and press **Enter**.
2. If you get an error, delete the node_modules folder (by entering **`rm -r node_modules`**) and running **`npm install`** again, followed by **`npm run test`**.
3. A series of things will happen and then a message will appear and tell you that the test passed.

Icons and Conventions

Tools and Techniques icon. Indicates that a lab demonstrates how to use important tools or techniques commonly used by professional developers. You may skip these labs if you wish; however, we do recommend you do them if you have the experience and the time.

Code font. Monospaced text indicates code that is run by the computer.

Command font. Bold text outside of headings indicates commands you should type (when used in conjunction with monospaced text) or menu items you should select; for example, from a user menu.

Numbered lists. Numbered lists with squares to the left indicate steps for you to perform to complete the lab. If you like, you can check the box next to each step to keep track of the steps you've completed.

Module: Intro to the Web Platform

Lab 01: Working with the Unix/Linux Command Shell

Many of the tools used by professional web developers are only available through a command line interface. In this lab, you'll learn the most commonly used Unix commands and how to use them to find your way around a command line interface.

Note: You need to install Git and download the files for this course before performing this lab. You can download it here: https://git-scm.com/downloads. Follow the instructions in the Introduction and Setup Instructions section.

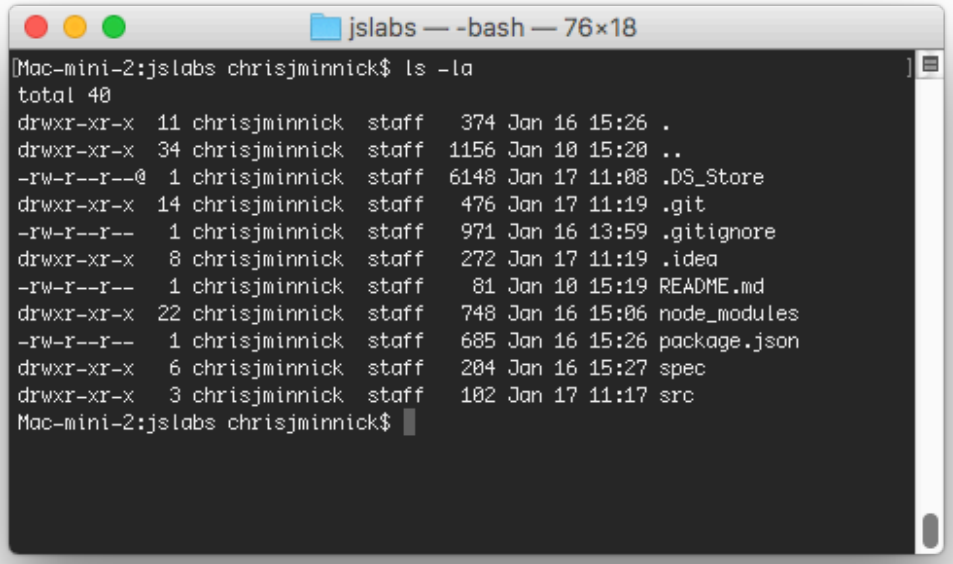
For brevity, when we say to enter or type a command at the command line, you should always press **Enter** after you type the command unless we indicate otherwise. If you are not already in the **jslabs** folder, follow these steps. If the JS folder is already open in your command prompt, skip to step 6:

- ☐ 1. Open Terminal (on Mac) or Git Bash (on Windows).
- ☐ 2. Type the **`pwd`** command to see your current directory.
- ☐ 3. Enter the **`ls`** command to see what files and directories are in the current directory.
- ☐ 4. Type the **`cd`** command to switch to your home directory (which is typically something like /Users/*[your user name]*, where *[your user name]* is actually your name).

Note: If you're inside a directory and want to go "up" a directory in the hierarchy, use **`cd ../`** to move into the current directory's parent directory.

- ☐ 5. Navigate to your project directory, which should be named jslabs (type **`cd jslabs`** from your home directory).

☐ 6. Type **`ls -la`** to see a detailed list of files and folders. You should see something like the following.



☐ 7. Display the contents of the README.md file by entering the following command:

```
cat README.md
```

☐ 8. Type the **`cat`** command, followed by just the letters **`RE`**, and then press the **Tab** key. Notice that the command shell automatically completes the rest of the command since there aren't any other files in the directory that start with RE. Press **Enter** to see the file.

Note: This tab completion feature works anywhere in the command shell and can be a great time-saver.

☐ 9. Enter the following command to create a new blank text file named **help.txt**:

```
touch help.txt
```

☐ 10. Type **`cat help.txt`**. Notice that nothing comes back, because the file is blank. (The `cat` command is used to display the file.)

Now it's time for you to put this knowledge to work. Use the commands in the following table combined with what you just learned, and complete the steps that come after the table.

Command	Description
`cd`	change directory
`ls`	list files
`pwd`	print working directory
`mkdir`	create a new directory
`cp`	copy
`mv`	move, or rename
`rm`	remove files or directories
`help`	get bash commands

Here's a hint. If you need help with a command, type the command followed by **`--help`**. So if you need to know how to move or rename a file, type **`mv --help`**.

- ☐ 1. Make a copy of help.txt and call it **morehelp.txt**.
- ☐ 2. Make a new directory named **help-files**.
- ☐ 3. Move help.txt and morehelp.txt into the help-files directory.
- ☐ 4. Change the working directory to help-files.
- ☐ 5. List the files in this directory to verify that it contains help.txt and morehelp.txt.
- ☐ 6. Delete morehelp.txt
- ☐ 7. Rename help.txt to **help.html**.

As your instructor if you need help with this lab.

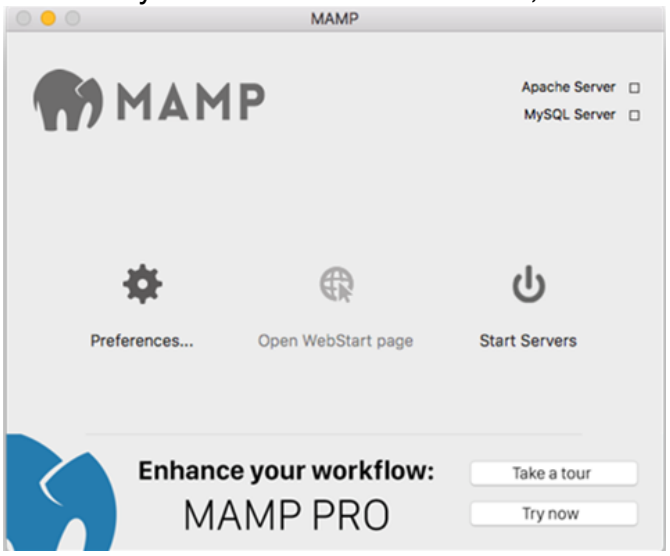
Lab 02: Configuring an HTTP (Web) Server

As mentioned in the presentation, the web works by a client (browser) requesting the data, the server responding with the data, and the client (browser) rendering and displaying the data. In this lab, you'll configure and test a web server on your local development machine, and then view the data in a web browser. This will help you understand this process a little more clearly.

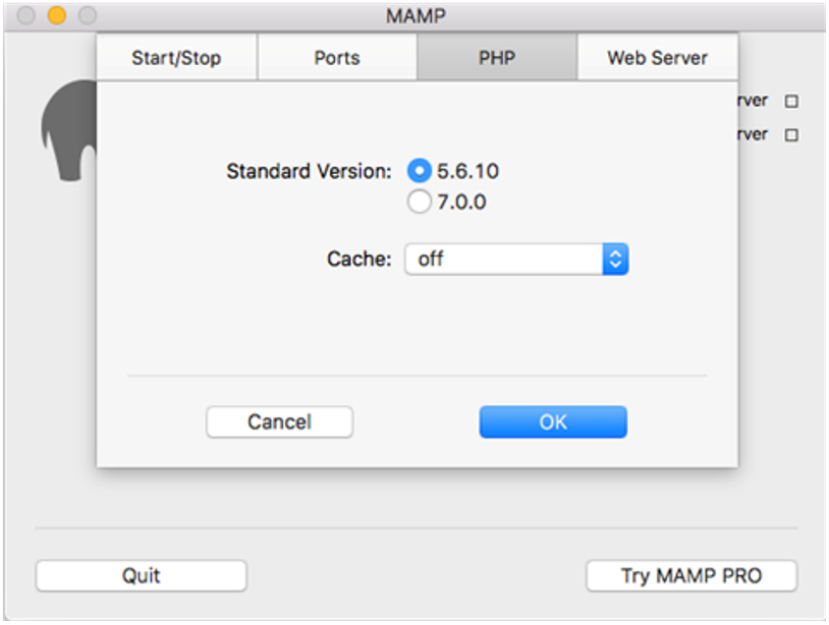
☐ 1. Go to https://www.mamp.info/en/downloads/ and download the appropriate version of MAMP for your computer.
☐ 2. Extract the file and follow the instructions to install MAMP.
☐ 3. Start MAMP by locating the application and launching it. Ignore the information for MAMP Pro. The free version will do fine for our purposes.

Note: If you receive an error stating Apache needs an open Port 80 and another service is already using it, close out of MAMP and then close Skype if you have it open. Skype also uses Port 80. Restart MAMP.

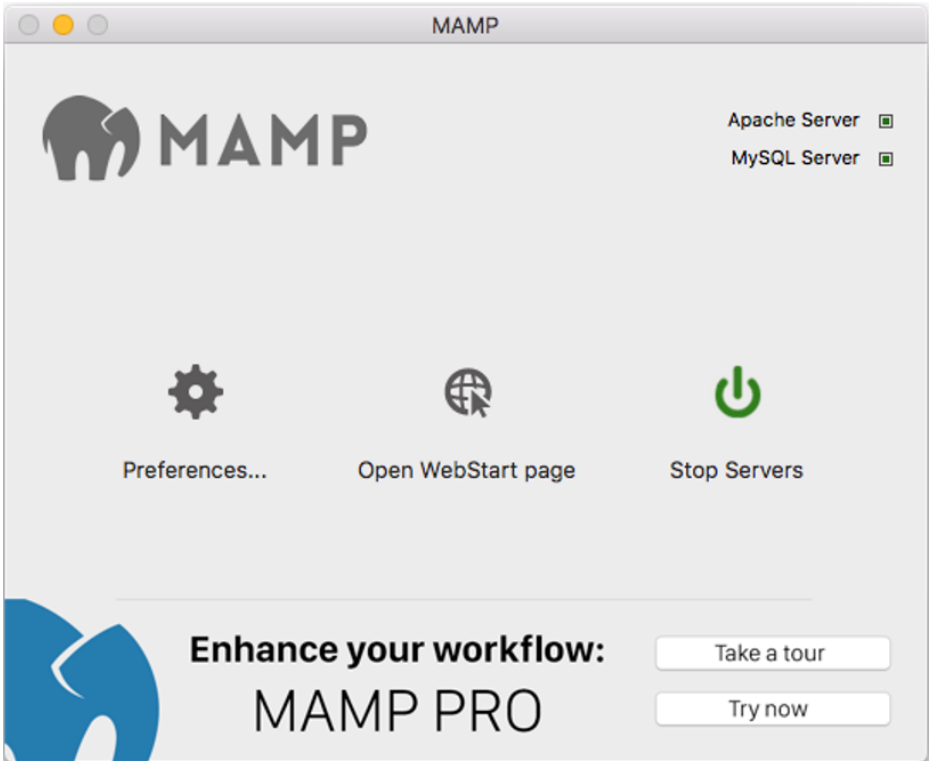
MAMP loads and you'll see the user interface, as shown below.



☐ 4. Click **Preferences**. The following screen appears.



☐ 5. Click the **Web Server** tab.
☐ 6. Make sure the Apache radio button is selected. Click **Select** and find the jslabs folder for the course.
☐ 7. Select the jslabs folder and click **OK**. This folder will be the "root" directory for your web site. In other words, when you go to **http://*www.yourwebsite.com/***, this is the place on your computer where the web server will look.
☐ 8. Click the **Ports** tab.
☐ 9. Make sure that the Apache Port is set to 8888. If it's not, change it, and then click **OK**.
☐ 10. Click **Start Servers** and wait for the square next to Apache Server to become green and change to Stop Servers.



☐ 11. Open a web browser and type **http://localhost:8888** into the address bar. You should see a welcome message.

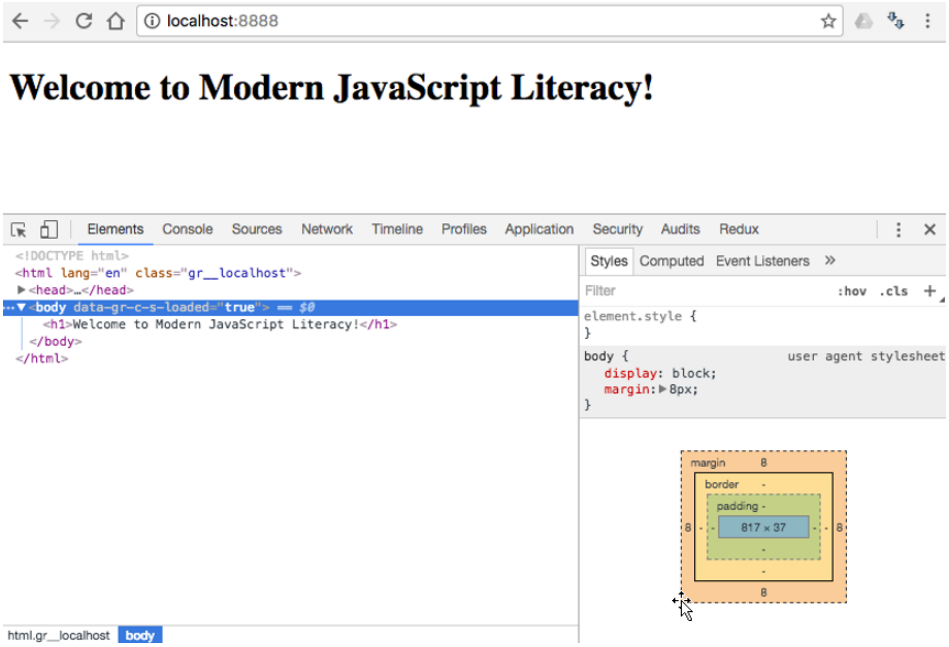
Welcome to Modern JavaScript Literacy!

☐ 12. Leave the browser page open (you'll use it in the next lab), and then go back to the MAMP application and click **Quit**.

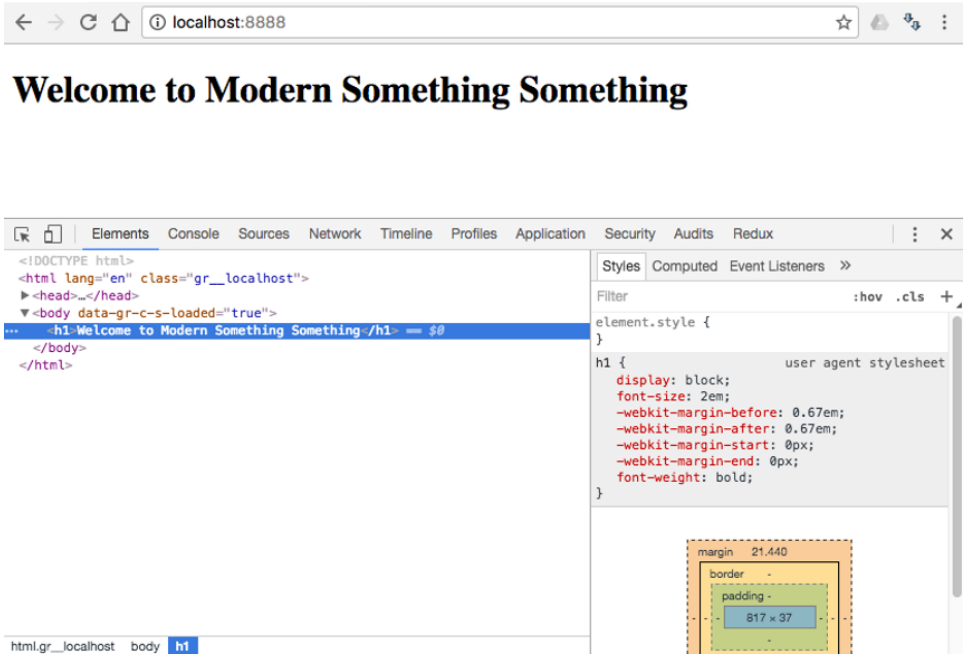
Lab 03: Working with Chrome Developer Tools–Element Tab

The Elements Tab of the Chrome Developer Tool gives you information about the currently rendered HTML page. In this lab, you'll explore the Elements pane and see how changes to the HTML affect the display of the web page in two different circumstances: 1) a page in your control and 2) a page that belongs to someone else.

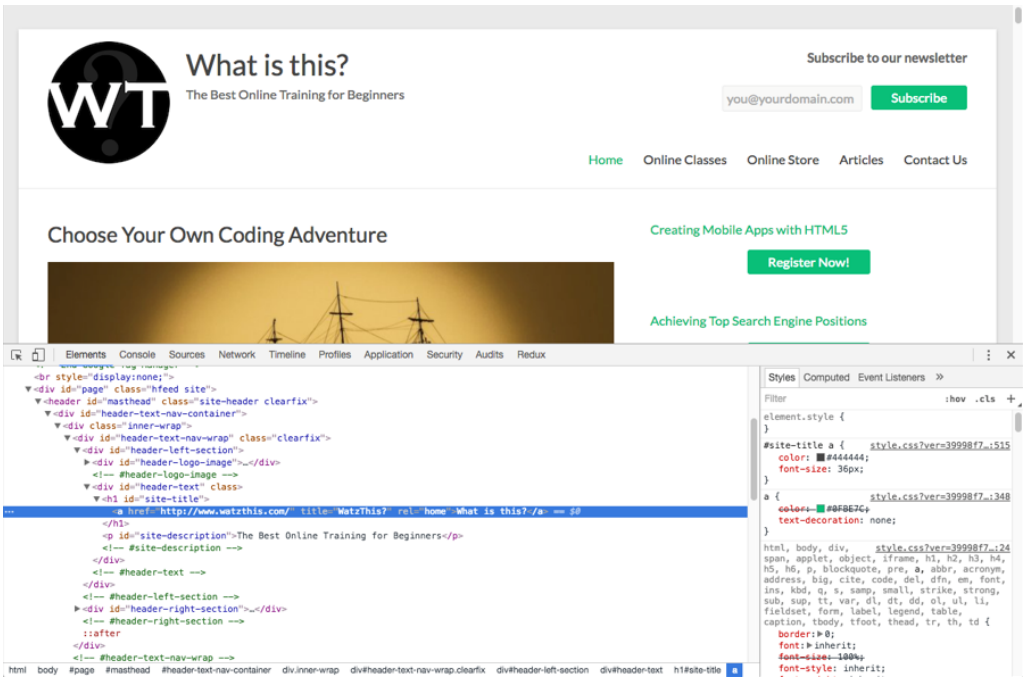
- ☐ 1. Open your Google Chrome Web browser if it's not already open, or install it if you don't already have it.
- ☐ 2. Navigate to http://localhost:8888 if it's not open from the previous lab.
- ☐ 3. Press **Ctrl+Shift+I** (on Windows) or **Command+Option+I** (on Mac). Chrome Developer Tools opens.
- ☐ 4. Select the **Elements** tab if it's not already selected.



- ☐ 5. Right click on the text **Welcome to Modern JavaScript Literacy** in the Elements Pane. Select **Edit as HTML** from the pop-up menu.
- ☐ 6. Change the text between `<h1>` and `</h1>` to anything you like.
- ☐ 7. Click your mouse outside of the editable area in the Elements Pane. The code is no longer be editable, and your change is reflected in the browser window.



☐ 8. Go to any other website; for example, http://www.watzthis.com.
☐ 9. Click the Inspector icon ( )in the upper-right corner of the developer tools. This is the one that looks like a mouse pointer and a square.
☐ 10. While the Inspector is selected, hover over different parts of the web page and notice how the code that creates that part of the web page is highlighted in the Elements Pane.
☐ 11. Click on a word in the browser window with the Inspector Tool selected. That word will remain selected in the Elements Pane.
☐ 12. Right click on the highlighted line in the Elements Pane and select **Edit as HTML**.
☐ 13. Change the text or HTML code, and then click outside of the editable area to apply your change. Your change will be reflected in the web page.



☐ 14. Refresh or reload the page. The change you made in the Elements Pane is overwritten with the version of the page that is downloaded from the web server because this is not your site and the owner's information will always appear on the page.

☐ 15. Close the web page.

Module: Intro to JavaScript

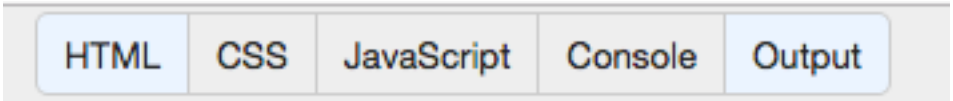
Lab 01: Using JS Bin

JS Bin is an online tool for experimenting with JavaScript and web applications. In this lab, you'll learn how to use it to rapidly test JavaScript features and functions.

Before we get started with the steps, let's explore the interface. In your Chrome web browser, go to http://jsbin.com. This interface consists frames where you can input different types of code, and two panes for outputting the results of running code.

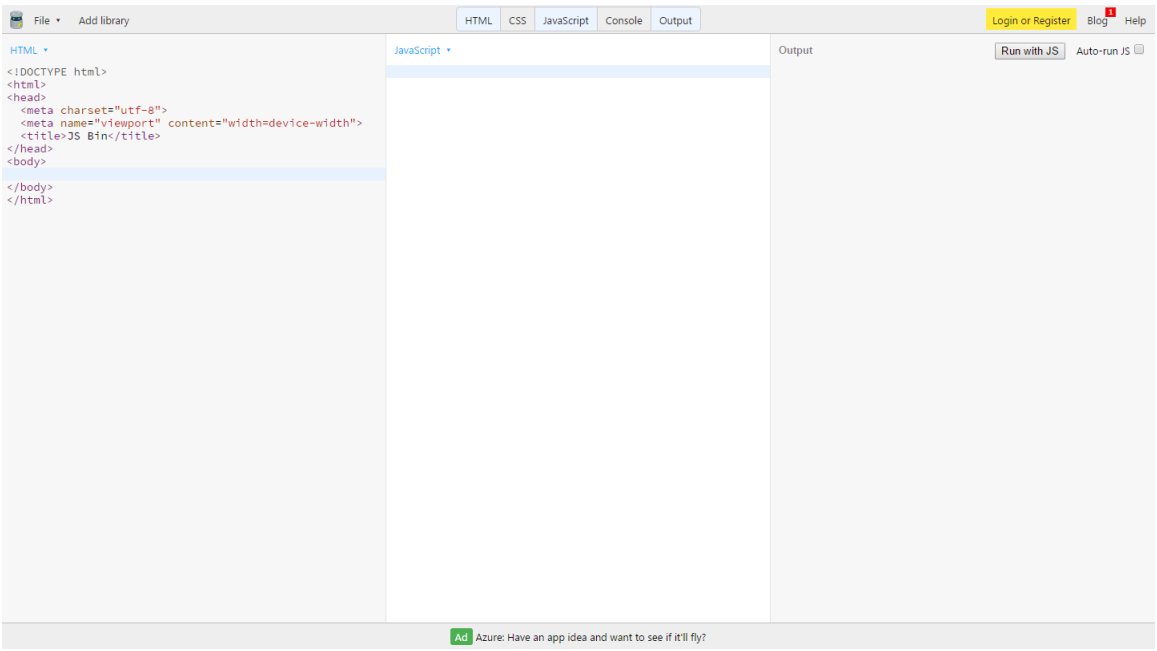
Note: All of the panes in JS Bin are linked together so they function exactly as they would if they were all written in the same web page. For example, CSS written in the CSS pane affects the output of the HTML written in the HTML pane.

At the top and center of JS Bin are the pane toggle buttons.



These buttons alternately enable and disable code and output panes. The highlighted buttons indicate which panes are currently active. Let's get started.

☐ 1. Make sure you're on the http://jsbin.com page, and then click **JavaScript** in the pane toggle buttons. The HTML, JavaScript, and Output panes should now be visible, as shown below.



☐ 2. In the HTML pane, between `<body>` and `</body>` create an HTML `div` element with an id of `"app"` and put some text between its starting and ending tags:

```
<div id="app">Great Books</div>
```

This is where the results of our JavaScript will appear. Right now, notice that this text appears in the Output pane.

☐ 3. In the JavaScript pane, create an array named greatBooks and add some items to it, as follows:

```
var greatBooks = ['East of Eden',
                  'For Whom the Bell Tolls',
                  'JavaScript for Kids'];
```

Caution: JavaScript is case-sensitive, and it can also be very sensitive about punctuation. If your code doesn't work as expected, check that you've typed it exactly right.

☐ 4. In the JavaScript pane, create a variable to store a reference to the `div` you created in the HTML pane.

```
var appDiv = document.getElementById('app');
```

☐ 5. Next, you're going to use the array and the reference to the `appDiv` to put a single item from the array into the HTML page. Type the following statement on the next line in the JavaScript pane.

```
appDiv.innerHTML = greatBooks[0];
```

☐ 6. Click the **Run with JS** button in the Output pane. Notice that writing this statement caused the text that you wrote inside of the `div` element in the HTML pane to be replaced with the contents of the first item in the array.

The `innerHTML` property refers to whatever is between the start and end tags of the selected item. This statement used the assignment operator (=) to give it a new value, which caused the browser to display the new value, even though if you look at the HTML pane, it still has the old text in it.

☐ 7. Next, you'll use a loop to output all of the items in the array. The idea here is very similar to what you did in step 4, but you're just going to do it once for each item in the array. Select the statement you wrote in step 4, and press **Command+X** (Mac) or **Ctrl+X** (Windows) to cut it. Where it was, type the following:

```
for (var i=0; i<greatBooks.length; i++) {}
```

☐ 8. Put your cursor between the curly braces ({}) and press **Enter** to go to a new line.

☐ 9. Paste the statement you copied in step 6 by pressing **Command+V** (Mac) or **Ctrl+V** (Windows).

☐ 10. Replace the `0` in square brackets with the variable **`i`**.

```
appDiv.innerHTML = greatBooks[i];
```

Your completed JavaScript code should look like this:

```
var greatBooks = ['East of Eden',
                  'For Whom the Bell Tolls',
                  'JavaScript for Kids'];
var appDiv = document.getElementById('app');
for (i=0; i<greatBooks.length; i++){
      appDiv.innerHTML = greatBooks[i];
}
```

☐ 11. Click the **Run with JS** button in the Output window. Your Output window should now display just the final item in your array. This is because the `for` statement caused JavaScript to output each item in the array, but each item replaced the one that had previously been there.

☐ 12. Modify the statement inside the loop to be a concatenating assignment statement by putting a + before the =.

```
appDiv.innerHTML += greatBooks[0];
```

☐ 13. Click the **Run with JS** button in the Output window. The original text from the HTML pane will apear in the output pane,

concatenated with each item in the array. Next, we'll make this look a bit better.

☐ 14. In the HTML Pane, move the list title (Great Books) outside of the div element and surround it with an `<h1>` element.

```
<h1>Great Books</h1>

<div id="app"></div>
```

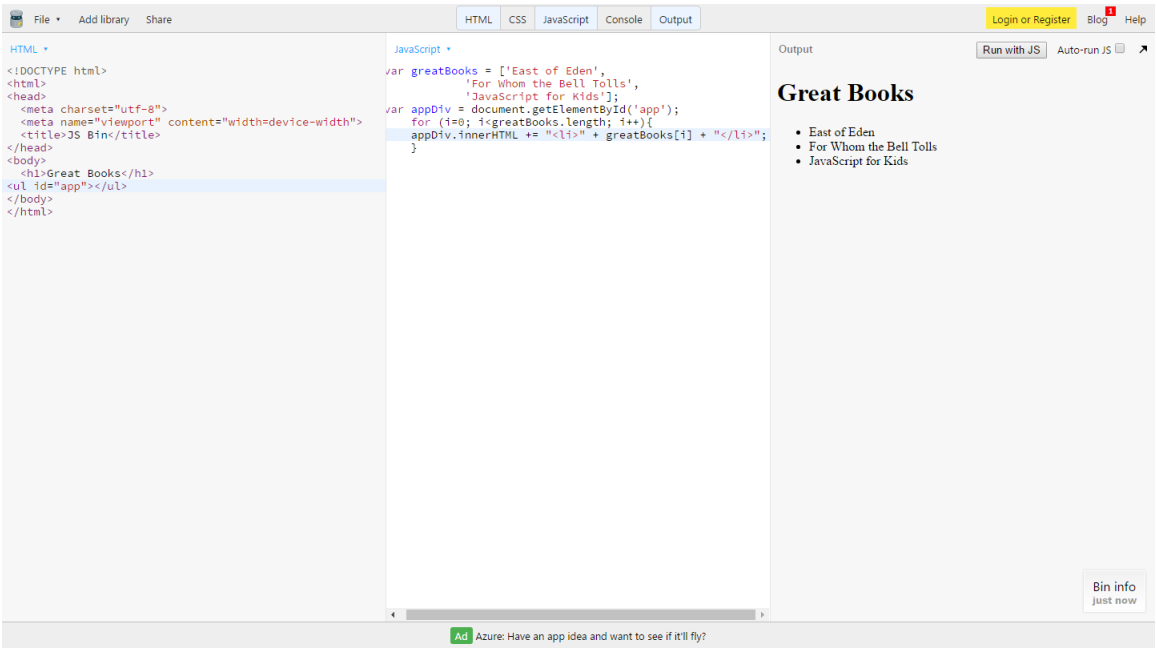
☐ 15. Change the div element to an unordered list element (`ul`).

```
<ul id="app"></ul>
```

☐ 16. In the JavaScript pane, use concatenation to put each array element inside a `li` element:

```
appDiv.innerHTML += "<li>" + greatBooks[i] + "</li>";
```

☐ 17. Click the **Run with JS** button in the Output window. If you entered everything correctly, you should now see a title followed by a bulleted list in the Output pane, as shown below.



Lab 02: Using Chrome Developer Tools: JavaScript Console

The JavaScript console in your browser is where error and debug messages appear during the running of your program. It can also function as a place where you can test JavaScript code within your browser.

Every browser has a JavaScript console. You can access the console through the browser's menu, or by pressing a function key or a combination of keys. The following table shows how to open the JavaScript console in the most popular web browsers.

Browser	**Mac**	**Windows**
Chrome	Command+Option+J	Ctrl+Shift+J
Firefox	Command+Option+K	Ctrl+Shift+K
Safari	Command+Option+C	N/A
Internet Explorer	N/A	F12

In this lab, you'll use Chrome's JavaScript Console to view error messages in a program, and then track down and fix the source of the errors.

☐ 1. Make sure that your MAMP server is running and go to http://localhost:8888/labs/JS100/lab02 in Chrome.

☐ 2. Open the JavaScript Console (see the table).

You'll see some messages and an error message. Notice that the error message gives you several important pieces of information.

- First, it tells you the type of error. In this case, it's an `Uncaught ReferenceError`. This means that you tried to use a variable that doesn't exist.
- The next piece of information is the specific reference error. In this case, it's saying that `somethingiswrong` is not defined.
- The final piece of the puzzle is the exact location of the problem: `"at script.js:3"`

This translates to "The JavaScript engine encountered an error on line 3 of script.js."

☐ 3. Open **labs/JS100/lab02/script.js** in your code editor and look at line 3:

```
console.log(somethingiswrong);
```

☐ 4. Update the code so that this line doesn't produce an error. You have a few options:

- Initialize a variable named `somethingiswrong` prior to line 3.
- Surround `somethingiswrong` with quotes to make it a string, rather than a variable.
- Remove this line of code completely.

☐ 5. When you have the problem fixed, save the file and return to your web browser. Reload the page and look at the JavaScript console to confirm that the error is gone.

JavaScript errors aren't always this easy to solve, and sometimes the console error message can seem frustratingly vague and unhelpful. However, the console should always be the first place you look when something isn't working right.

Lab 03: Using JavaScript Array Methods

JavaScript arrays have some built-in functions that make working with arrays easier and faster. In this lab, you'll use several array methods to manipulate some arrays.

An array is nothing more than a list of items assigned to a single name, or variable. For example, if you want to make a list of songs, it might look something like this:

```
var songs = ['California Girls', 'Area Codes','88 Lines about 44 Women'];
```

If you want to get the value of an item from an array, you can do so by referring to its position (or index) in the array. Array indexes in JavaScript start with `0`. So, to get the first item in this array of song titles, you would use the following:

```
songs[0]
```

As with other types of lists, such as to-do lists and outlines, array lists can have sublists.

It so happens that each of the songs mentioned in the previous array is a 'list' song, which means that the lyrics of the songs have lists within them (sub-lists). You can represent these sub-lists using a multi-dimensional list, which is just a list within a list:

```
var songlists = [['East Coast','Southern','Mid-West','Northern','West Coast'],
                 ['770','404','718','202'],
                 ['Deborah','Carla','Mary','Suzen']];
```

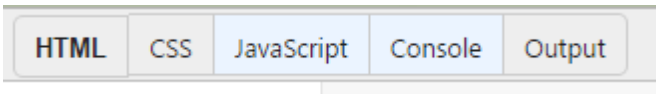
To get the value in a particular spot in a multidimentional array, just use more square brackets. For example, the following will return the words 'West Coast'.

```
songs[0][3]
```

How would you get the value '718' from this array?

Now, suppose that you wanted to make some changes to your list of songs or your list of lists inside of songs. Follow these steps to find out how you can easily add, remove, combine, and otherwise modify arrays in JavaScript.

- ☐ 1. In Chrome, go to http://jsbin.com and select **File > New** to create a new, blank bin.
- ☐ 2. Click on the appropriate toggle buttons to open the **JavaScript** and **Console** panes.



- ☐ 3. In the JavaScript pane, type in the code for both above arrays (`songs` and `songlists`).

```
var songs = ['California Girls', 'Area Codes','88
Lines about 44 Women'];


var songlists =  [['East Coast','Southern','Mid-
West','Northern','West Coast'],
                 ['770','404','718','202'],
                 ['Deborah','Carla','Mary','Suzen']];
```

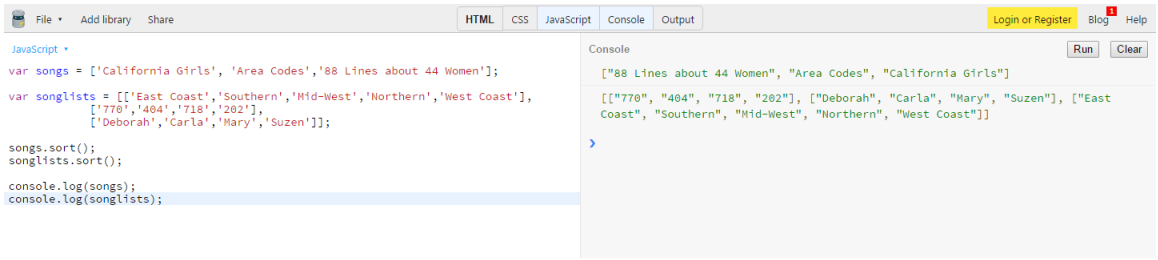
☐ 4. Next, we'll use the `sort()` method to sort the values in each array. By default, the `sort()` sorts values alphabetically and in ascending order. Insert the following below the two arrays:

```
songs.sort();

songlists.sort();
```

☐ 5. Use the `console.log()` method to print the new values (display the data) of each list.

```
console.log(songs);

console.log(songlists);
```

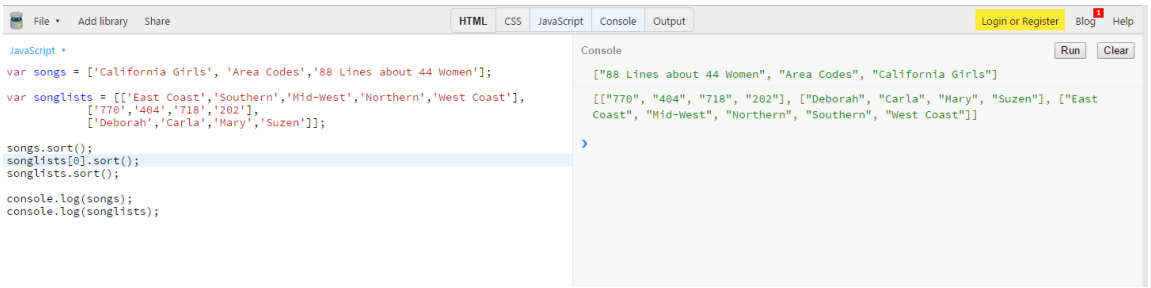
☐ 6. Click the **Run** button to run your program and see the results.



☐ 7. Add another sort between the `songs.sort();` and `songlists.sort();` methods to sort the individual items inside the first sub-array of `songlists`.

```
songlists[0].sort();
```

☐ 8. Click the **Clear** button in the console to clear the window, and then click **Run**.



Notice that the sublist that was first in the `songlists` array (but is third in the sorted array) is now sorted alphabetically (the East Coast… list).

☐ 9. Figure out how to write the statements to sort the other two sub-lists of the `songlists` array.

The next array methods we'll talk about are the `push` and `pop` methods. The `push` method adds new items to the end of an array. The `pop` method removes an item from the end of an array.

☐ 1. Use the following code to add another song to the end of the songs array.

```
songs.push('We Care a Lot');
```

☐ 2. Click the **Clear** button and then the **Run** button to see the result. The `console.log` method you entered earlier produces the results.

☐ 3. Remove the last item from the array using the `pop()` method.

```
songs.pop();
```

☐ 4. Click the **Clear** button and then the **Run** button to see the result. The `console.log` method you entered earlier produces the results.

☐ 5. Can you figure out how to add a new sub-list to the songlists array? Here's some data to add to it:

```
['Raindrops on roses','whiskers on kittens','bright copper kettles','warm woolen mittens']
```

☐ 6. Click the **Clear** button and then the **Run** button to see the result.

Solution: If you need some help, contact your instructor or see the finished code at http://jsbin.com/gikiqa/edit.

Note: If you want to see what other array methods are available, visit https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array.

Lab 04: Using JavaScript Objects

Objects are central to nearly any JavaScript program. In this lab, you'll create an object and use several different techniques to create properties and methods.

JavaScript objects are collections of properties. A property has a name and a value, separated by a colon.

Objects may be created in several different ways, but the most concise way (and so usually the best) is by using what's known as object literal syntax.

To create an object using object literal syntax, you simply assign a pair of curly braces containing comma-separated property-value pairs to a variable.

For example:

```
var painting = {
  museum: "The Met",
  onDisplay: true,
  artist: 'Vincent van Gogh',
  title: 'Wheat Field with Cypresses',
};
```

The value of a property can be any type of data (including other objects or functions). In this object example, we've created a generic painting object for storing data about a painting in The Metropolitan Museum of Art in New York. In this lab, you'll add properties to this painting object and retrieve and modify properties.

☐ 1. Open **labs/JS100/lab05/index.html** in the code editor of your choice.

☐ 2. Make sure that your MAMP server is running. (See the Configuring an HTTP (Web) Server lab).

☐ 3. Open http://localhost:8888/labs/JS100/lab05/ in a browser to see what the page looks like so far.

☐ 4. Return to your code editor and examine the code for index.html. Can you see how it works?

Inside of `$(document).ready`, there are three jQuery statements. These statements each select a different element in the HTML and then replace the contents of those elements with the values of properties in the picture object.

Let's add another property to the painting object and then use jQuery to display that property in an appropriate place in the HTML.

☐ 5. Add a property called `image` to the `painting` object.

☐ 6. Give the `image` property a value of the file name of the jpg in the images folder inside the lab05 folder.

The painting object should now look like this:

```
var painting = {
    museum: "The Met",
    onDisplay: true,
    artist: 'Vincent van Gogh',
    title: 'Wheat Field with Cypresses',
    image: 'DT1567.jpg'
};
```

☐ 7. Add an HTML `img` element inside the `figure` element. Give the image a blank `src` attribute.

```
<img src="" alt="painting" />
```

☐ 8. Under the artist-name selector, write a jQuery selector to select the `img` element. Since the `img` element doesn't have an `id` or `class` attribute, select it by selecting its parent and then using a descendent selector, like this:

```
$('#artist-name').html(painting.artist);
```

```
$('#image>img')
```

☐ 9. Chain a method to this selector to make the image display. For the `img` element, you want to change the value of the `src` attribute, rather than changing the HTML of the element. You can change attribute values using the jQuery `attr()` method, as follows:

Note: Because the image file is inside the images subfolder, we need to add the name of that folder into the attribute value as well. We've done that here by concatenating it with the filename.

```
$('#image>img').attr('src','images/'+painting.image)
```

☐ 10. Add another jQuery statement to change the value of the `img` element's `alt` attribute to the title of the painting.

```
$('#image>img').attr('src','images/'+painting.image);
```

```
$('#image>img').attr('alt',painting.title);
```

Note: The `alt` attribute isn't normally displayed anywhere in the HTML page, but it's used by search engines, screen readers for the blind, and as a placeholder when the image can't be loaded (perhaps because of a network problem).

Save the **index.html** file and then refresh the page in your browser. It should look like this:

Wheat Field with Cypresses

Vincent van Gogh

☐ 11. You should now have two jQuery statements that modify the `img` element. Rather than repeating the selection, a better way to write this is to chain the two modifications to the `img` element together, like this:

```
$('#image>img').attr('src','images/' + painting.image)
               .attr('alt',painting.title);
```

☐ 12. Next, we'll add a checkbox to the page that will control whether the painting is on display at The Met. Above the `figure` element, add a checkbox with a label.

```
<input type="checkbox" id="display"> On Display
```

☐ 13. Save the **index.html** file and then refresh the page in your browser. Check and uncheck the checkbox. Notice that nothing happens.

☐ 14. Now you are going to write some code to detect whether this checkbox is checked and show and fade out the image accordingly. Put a comma after your `image` property in the `painting` object and create a new property named `toggleDisplay`. The value of this property will be a function that will change the value of the `onDisplay` property.

```
image: 'DT1567.jpg',
toggleDisplay: function(){
```

```
        if(painting.onDisplay===true){
              painting.onDisplay = false;
        } else {
              painting.onDisplay = true;
        }
        console.log('onDisplay is ' +
painting.onDisplay);
}
```

Your painting object should now look like this:

```
var painting = {
      museum: "The Met",
      onDisplay: true,
      artist: 'Vincent van Gogh',
      title: 'Wheat Field with Cypresses',
      image: 'DT1567.jpg',
      toggleDisplay: function(){
            if(painting.onDisplay===true){
                  painting.onDisplay = false;
            } else {
                  painting.onDisplay = true;
         }
      console.log('onDisplay is ' +
painting.onDisplay);
      }
};
```

☐ 15. Next, create a jQuery event listener that will run the `toggleDisplay` function when the checkbox (which has an `id` of `'display'`) changes. Enter the following under `.attr('alt',painting.title);`.

```
$('#display').change(painting.toggleDisplay);
```

☐ 16. Save the **index.html file**.

☐ 17. In your Chrome web browser, open the JavaScript console by pressing **Command+Option+J** (on Mac) or **Ctrl+Shift+J** (on Windows).

☐ 18. Refresh the page and click the checkbox. You should see a message display in the console as you change the checkbox from checked to unchecked.

However, you may notice that the checked/unchecked state of the checkbox is out of sync with the message. This is because we didn't detect whether `picture.onDisplay` was true or false and set the checkbox state appropriately when the program started. Let's fix that now.

☐ 19. At the beginning of the `$(document).ready function` (before the other jQuery statements), insert the following:

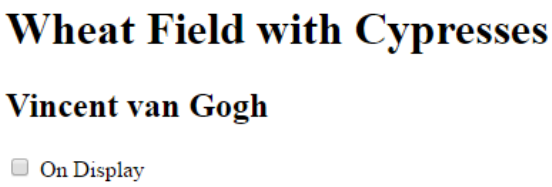
```
if(painting.onDisplay === true) {
  $('#display').prop('checked',true);
}
```

☐ 20. Test it out. If `painting.onDisplay` is set to `true` when the object is created, the checkbox should now be checked.

☐ 21. The last thing, you'll do is fade out the painting when `onDisplay` changes to `false`. Inside the `toggleDisplay` function, fade out the image when `painting.onDisplay` changes to `false`, and fade it in when `painting.onDisplay` changes to `true`.

```
toggleDisplay: function(){
    if(painting.onDisplay===true){
        painting.onDisplay = false;
        $('#image').fadeOut();
    } else {
        painting.onDisplay = true;
        $('#image').fadeIn();
    }
    console.log('onDisplay is ' + painting.onDisplay);
}
```

Now when you click the checkbox, in addition to seeing the results in the console, the image should fadeout when you clear the On Display checkbox.



Lab 05: Performing DOM Manipulation

Now that you're familiar with the JS Bin user interface, it's time to get serious. You're going to use JS Bin to write an exciting game, set in the distant future. You're the pilot of a spaceship armed with a powerful anti-gravity laser system. You've been sent on a mission to destroy approaching asteroids before they strike your home world and cause mass extinctions.

- ☐ 1. Open http://jsbin.com in Chrome. Select **File > New** to create a new, blank bin.
- ☐ 2. Click on the appropriate toggle buttons to open the **HTML**, **CSS**, **JavaScript**, and **Output** panes.
- ☐ 3. Enter the following HTML between the `<body>` and `</body>` tags inside the HTML pane. This code sets a `div` tag with the `id` of `asteroid` and displays an `*`.

```
<div id="asteroid">
 *
</div>
```

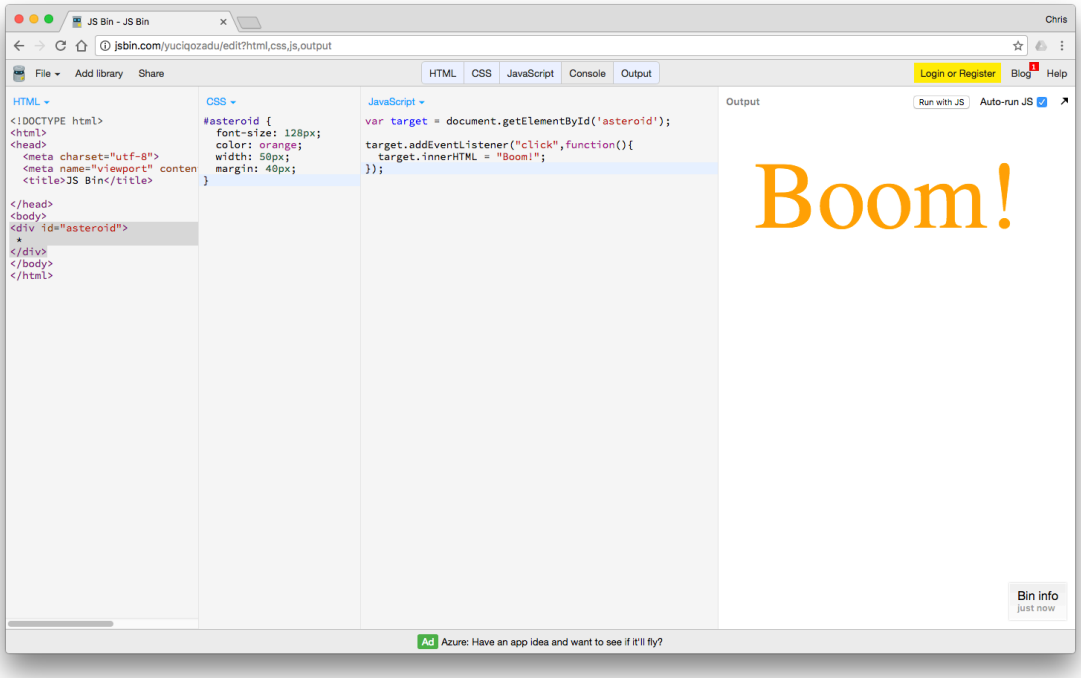
- ☐ 4. Enter the following into the CSS pane. This CSS sets the font size, color, width, and margin.

```
#asteroid {
  font-size: 128px;
  color: orange;
  width: 50px;
  margin: 40px;
}
```

- ☐ 5. Enter the following into the JavaScript pane. This JavaScript sets a variable as a reference to the element with the ID `asteroid`, and then when that target element is clicked, it results in a Boom.

```
var target = document.getElementById('asteroid');


target.addEventListener("click",function(){
  target.innerHTML = "Boom!";
});
```

- ☐ 6. Double-check your work (the slightest miscalculation will cause your mission to fail before it even begins), and then click the **Run with JS** button at the top of the Output pane.
- ☐ 7. When you're ready, click on the asteroid with your mouse pointer and watch the fireworks.


JS Bin - JS Bin
Chris
jsbin.com/yuciqozadu/edit?html,css,js,output
File
Add library
Share
HTML
CSS
JavaScript
Console
Output
Login or Register
Blog
Help
HTML
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<meta name="viewport" conten
<title>JS Bin</title>
</head>
<body>

*

</body>
</html>
CSS
#asteroid {
font-size: 128px;
color: orange;
width: 50px;
margin: 40px;
}
JavaScript
var target = document.getElementById('asteroid');
target.addEventListener("click",function(){
target.innerHTML = "Boom!";
});
Output
Run with JS
Auto-run JS
Boom!
Bin info
just now
Ad
Azure: Have an app idea and want to see if it'll fly?

Lab 06: Using jQuery

jQuery is a popular JavaScript library designed to simplify the writing of client-side JavaScript. In this lab, you'll use jQuery to build a simple program for chatting with yourself or with someone sitting at the same computer as you.

☐ 1. Go to http://jsbin.com in your web browser and click **File > New** to create a new bin.

☐ 2. Open the **HTML**, **CSS**, **JavaScript**, and **Output** panes if they're not already open.

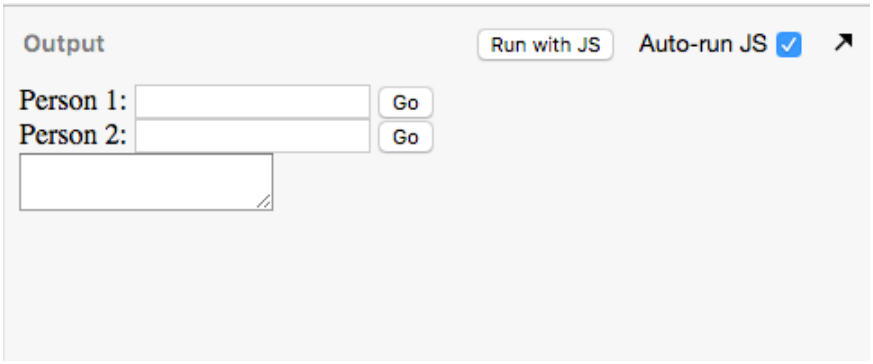
☐ 3. Next, you're going to create some labels with various input types. Enter the following HTML between the `<body>` and `</body>` tags in the HTML pane.

```
  <label>Person 1: <input type="text"
id="person1"></label>

  <button id="person1_submit">Go</button><br>

  <label>Person 2: <input type="text"
id="person2"></label>

  <button id="person2_submit">Go</button><br>

  <textarea id="chat_window"></textarea>
```

☐ 4. Put your cursor right before the `</body>` tag in the HTML pane and create a new blank line.

☐ 5. Click the **Add library** link above the HTML pane and select **JQuery 3.0.0**. A line of code to import the jQuery library will be inserted into your HTML pane.

If you entered everything correctly, your Output window should now look like this:

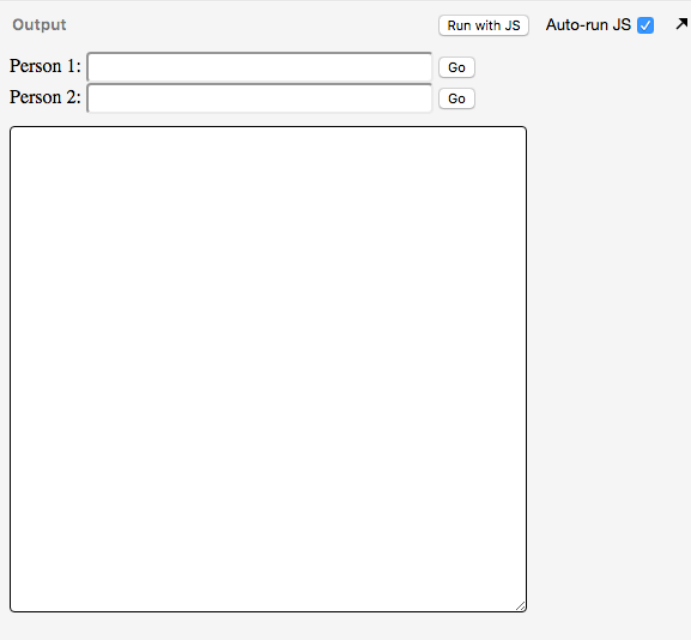


☐ 6. Now, you are going to increase the size of the bottom `textarea`. Enter the following code into the CSS pane.

```
textarea {

  margin-top: 10px;
```

```
  width: 75%;
  border-radius: 4px;
  height: 400px;
}

input {
  width: 50%;
  border-radius: 4px;
  height: 20px;
}
```

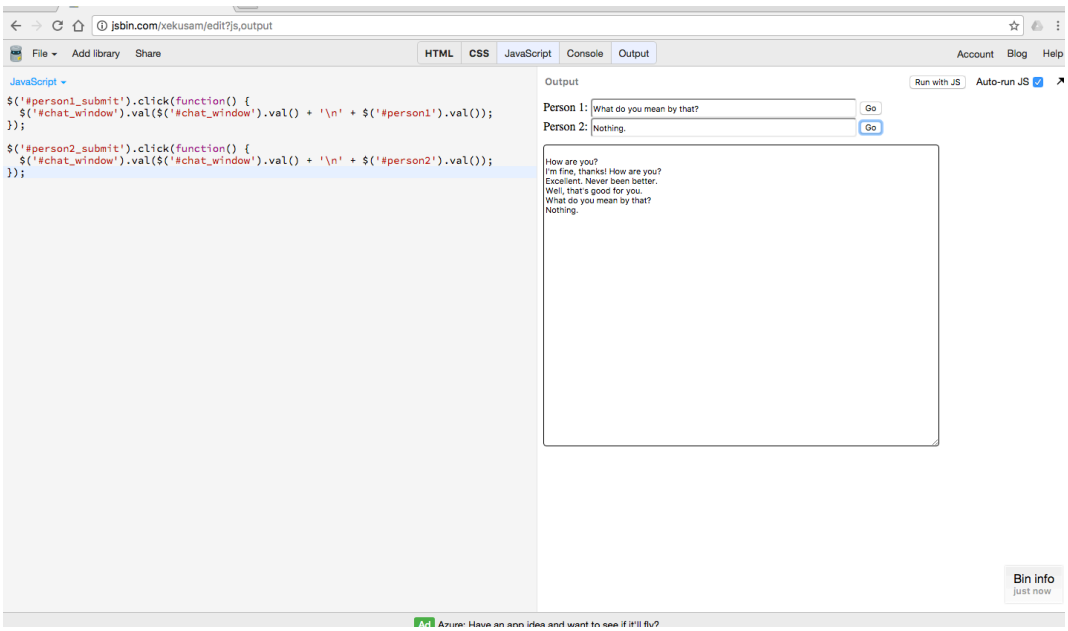
Your Output area should now look like the following:



☐ 7. Enter the following into the JavaScript pane to connect the input boxes to the output box.

```
$('#person1_submit').click(function() {
  $('#chat_window').val($('#chat_window').val() + '\n'
+ $('#person1').val());
});
$('#person2_submit').click(function() {
  $('#chat_window').val($('#chat_window').val() + '\n'
+ $('#person2').val());
});
```

☐ 8. Click **Run with JS** and have some fun chatting with yourself.



☐ 9. Now it's time for you to practice on your own a little. Figure out how to make one, or both, of the following improvements to the chat interface.

- Clear out the input field after the current message is added to the chat window.
- Add the person's name (Person 1 or Person 2) to the chat window before their message.

Solution: If you need some help, contact your instructor or check out the finished code at http://jsbin.com/xekusam.

Lab 07: Using AJAX with jQuery

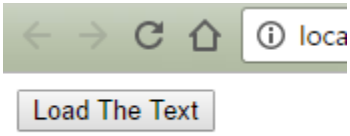
In this lab, you'll use AJAX to dynamically load a file and display its contents inside an HTML page.

☐ 1. In File Explorer (Windows) or Finder (Mac), navigate to your jslabs folder, and then open **JS100\lab07**.

☐ 2. Open **index.html** in the code editor of your choice. This file contains an HTML button and a `div` where we want text to appear when the button is clicked.

☐ 3. Make sure that your MAMP server is running. (See Lab 02: Configuring an HTTP (Web) Server).

☐ 4. Open http://localhost:8888/labs/JS100/lab07/ in a browser to see what the page looks like so far.



☐ 5. Back in your code editor, between `<script>` and `</script>`, type the following jQuery document ready statement:

```
$(document).ready();
```

This statement tells jQuery to wait until the HTML document is fully loaded and rendered before running whatever you put between the parentheses after ready.

☐ 6. Between the parentheses after `ready`, type the following anonymous function.

```
$(document).ready(function(){});
```

An anonymous function is a function that doesn't have a name. Anonymous functions are run by JavaScript immediately when the parser gets to them in the flow of the program. In this example, the contents of this function will run as soon as the document is ready.

When a function is passed into another function as a parameter, as in this case, it's known as a *callback*.

☐ 7. Inside the anonymous function body (between the `{` and `}`) write a jQuery selector to select the button by its ID, as follows:

```
$(document).ready(function(){
  $('#getText')
});
```

Note that the `$` in jQuery code is just an alias for jQuery. Any time you see a `$` in jQuery code, you could substitute it with jQuery and the code will work exactly the same.

Note: Almost every programmer who uses jQuery uses the `$` alias rather than the full jQuery name. Another shortcut method is called method chaining. This technique is used to simplify code when you are calling multiple functions on the same object. So, if we say to chain something, it means you're creating a shortcut for the functions.

☐ 8. Following the selector for the button, chain a `click` event handler by entering the following:

```
$('#getText').click();
```

☐ 9. Next, you enter a callback function that specifies what you want to happen when this `click` event is triggered. You put that between the parentheses after the `click` event handler, as follows:

```
$('#getText').click(function(){


});
```

☐ 10. Inside the callback function call the jQuery ajax function, as follows:

```
$('#getText').click(function(){
  $.ajax();
});
```

☐ 11. The jQuery `ajax()` function takes a JavaScript object as its parameter. Put a placeholder object into the parentheses of the `ajax` function, as follows:

```
$('#getText').click(function(){
  $.ajax({
    property1:value,
    property2:value
  });
});
```

☐ 12. The jQuery `ajax` function can take many different properties, which allow you to customize the different aspects of an ajax request. But, you only need to worry about two of them for this example: `url` and `success`. Change the first placeholder property name to **`url`** and set the value to a string containing the name of the file to load into this document.

```
$('#getText').click(function(){
  $.ajax({
    url:'story.txt',
    property2:value
  });
});
```

☐ 13. The second property you're going to modify is the `success` method. The `success` method tells what the `ajax` function should do if the request is successful. You're going to tell it to display contents of the file inside the HTML element with an `id` of `result`. Replace the `property2:value` with the following:

```
$('#getText').click(function(){
  $.ajax({
    url:'story.txt',
    success:function(data){
      $('#result').html(data);
    }
  });
});
```

When everything is done, your script element should look like this:

```
<script>
    $(document).ready(function(){
        $('#getText').click(function(){
            $.ajax({
                url: 'story.txt',
                success: function (data){
                    $('#result').html(data);
                }
            });
        });
    });
</script>
```

☐ 14. Save **index.html** file and reload the webpage in your browser.

☐ 15. Press the button. If you did everything correctly, the contents of **story.txt** should appear in your browser, below the button.

Load The Text

Metamorphosis

by Franz Kafka

Translator: David Wyllie

I

One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked.

"What's happened to me?" he thought. It wasn't a dream. His room, a proper human room although a little too small, lay peacefully between its four familiar walls. A collection of textile samples lay spread out on the table - Samsa was a travelling salesman - and above it there hung a picture that he had recently cut out of an illustrated magazine and housed in a nice, gilded frame. It showed a lady fitted out with a fur hat and fur boa who sat upright, raising a heavy fur muff that covered the whole of her lower arm towards the viewer.

Gregor then turned to look out the window at the dull weather. Drops of rain could be heard hitting the pane, which made him feel quite sad. "How about if I sleep a little bit longer and forget all this nonsense", he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild, dull pain there that he had never felt before.

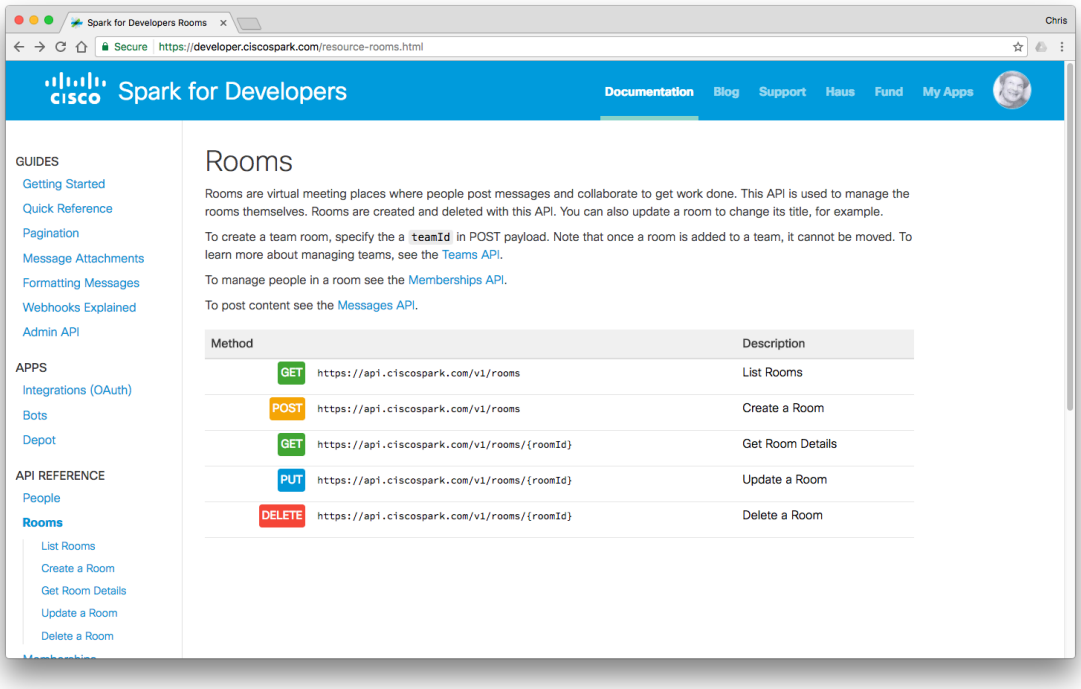
"Oh, God", he thought, "what a strenuous career it is that I've chosen! Travelling day in and day out. Doing business like this takes much more effort than doing your own business at home, and on top of that there's the curse of travelling, worries about making train connections, bad and irregular food, contact with different people all the time so that you can never get to know anyone or become friendly with them. It can all go to Hell!" He felt a slight itch up on his belly; pushed himself slowly up on his back towards the headboard so that he could lift his head better; found where the itch was, and saw that it was covered with lots of little white spots which he didn't know what to make of; and when he tried to feel the place with one of his legs he drew it quickly back because as soon as he touched it he was overcome by a cold shudder.

Continue Reading

Lab 08: Using JSON and REST to Work with Rooms

In this lab, you'll use JSON Data along with REST to list rooms, create rooms, and add people to rooms in Cisco Spark.

- ☐ 1. Create an account if you don't already have one, and log in at developer.ciscospark.com.
- ☐ 2. Click **Get Started**.
- ☐ 3. Click on **Rooms** under API Reference in the left navigation bar.



You'll see a list of the available methods for the Rooms API. Remember that a method is a function that's a property of an object. Using the Room API's method, you can do the following things:

- List Rooms
- Create a Room
- Get Room Details
- Update a Room
- Delete a Room

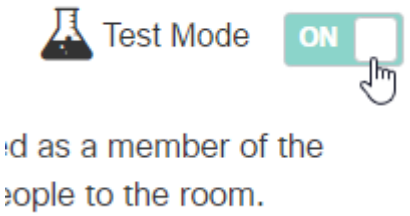
The Rooms API is a RESTful API. REST stands for Representational State Transfer. In short, it's a way for programs to work with web APIs using standard Hypertext Transfer Protocol (HTTP).

The four HTTP request methods are GET, PUT, POST, and DELETE. A RESTful API lets you use these four methods to perform different types of operations. For example, to list (get) rooms, you perform a `GET` request using the URL of the rooms API, which is https://api.ciscospark.com/v1/rooms.

☐ 1. Click the **Create a Room** method (POST). The Create a Room method allows you to create a new room. You're automatically added to any room you create.

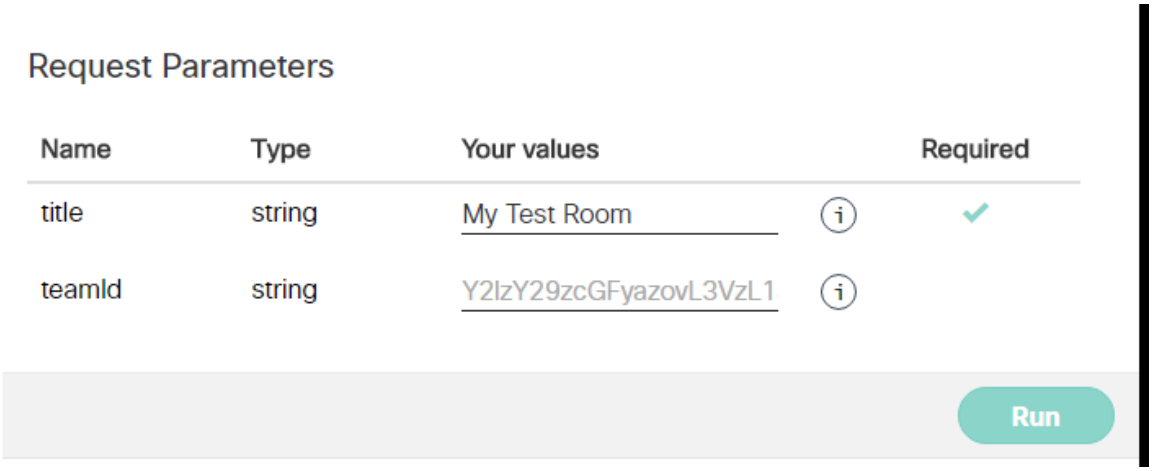
Method		Description
GET	https://api.ciscospark.com/v1/rooms	List Rooms
POST	https://api.ciscospark.com/v1/rooms	Create a Room
GET	https://api.ciscospark.com/v1/rooms/{roomId}	Get Room Details
PUT	https://api.ciscospark.com/v1/rooms/{roomId}	Update a Room
DELETE	https://api.ciscospark.com/v1/rooms/{roomId}	Delete a Room

☐ 2. Click on the **Test Mode** toggle button to turn the test mode on.



The button will change colors and the documentation page will become a form.

☐ 3. Enter **My Test Room** or any other name for a room under the Request Parameters header, in the **title** field.

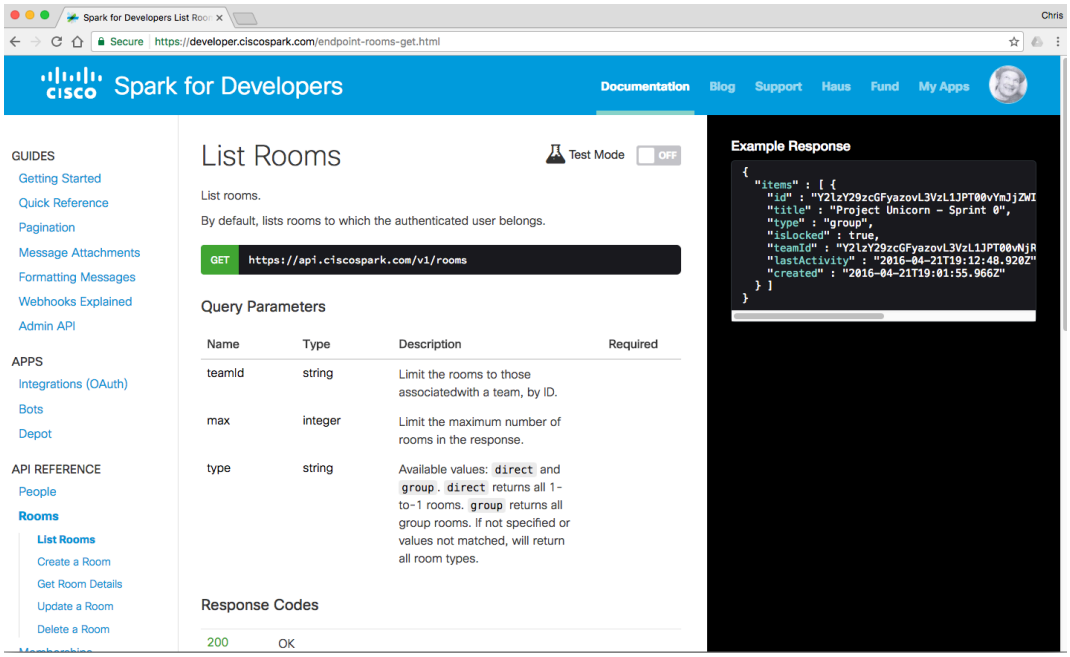


☐ 4. Click **Run**. After a moment, a Response will appear in the black area on the right of the screen.

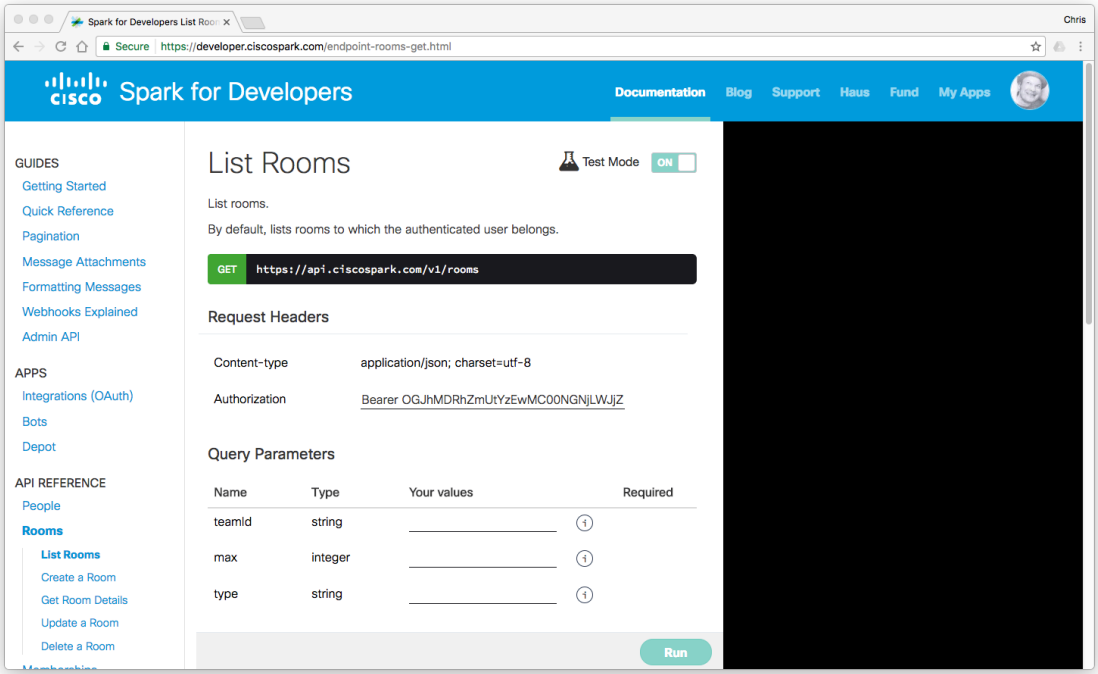
☐ 5. Open your **Cisco Spark** client application.

☐ 6. Click the **Recents** button on the left navigation, and then click **All** to view all your Rooms. Your new room will appear at the top of the list.

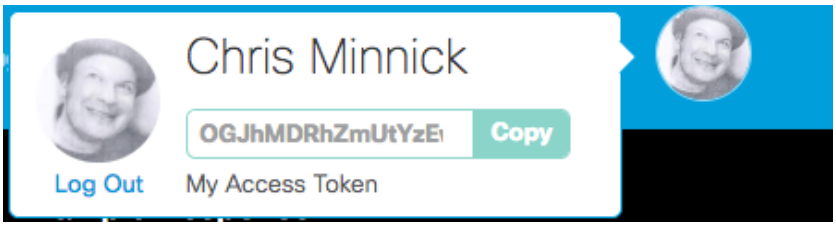
☐ 7. Back on the Spark for Developers site, click on the **List Rooms** under API Reference in the left navigation bar. This opens the detailed documentation for the List Rooms method.



☐ 8. Click the **Test Mode** toogle switch to turn it **ON** (if it's not already on). The List Rooms documentation will turn into a form and the JSON code in the right sidebar will disappear. Notice that that the Content-type and Authorization fields in the form are filled out for you.

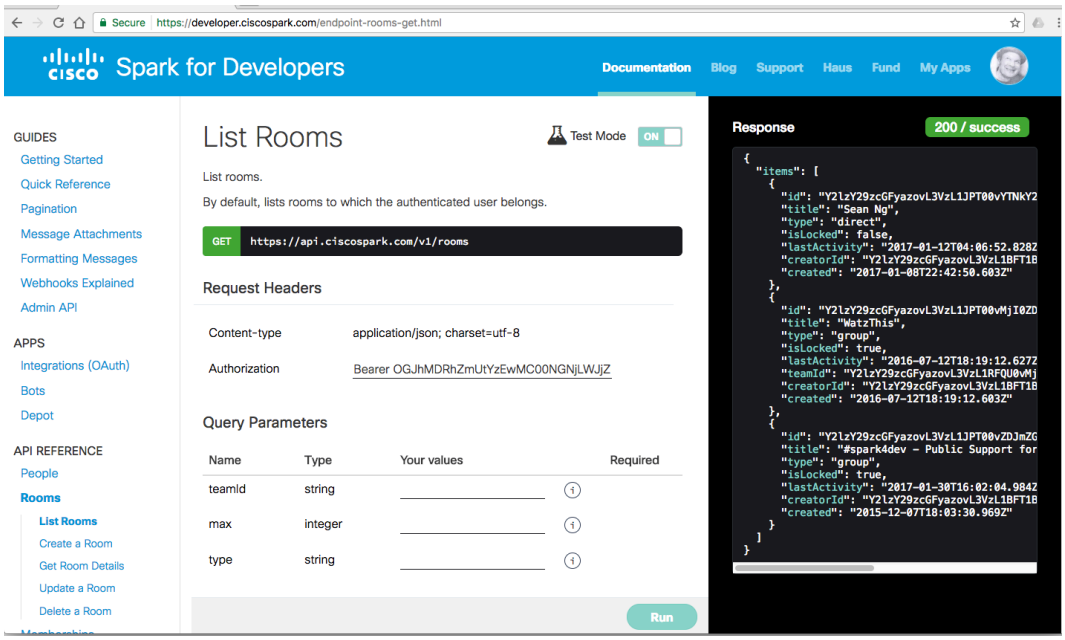


☐ 9. Click your user photo in the upper-right corner of the browser. You'll see a pop-up window containing a field with the text My Access Token. Notice that this token is the same as what's inside the Authorization field in the form.

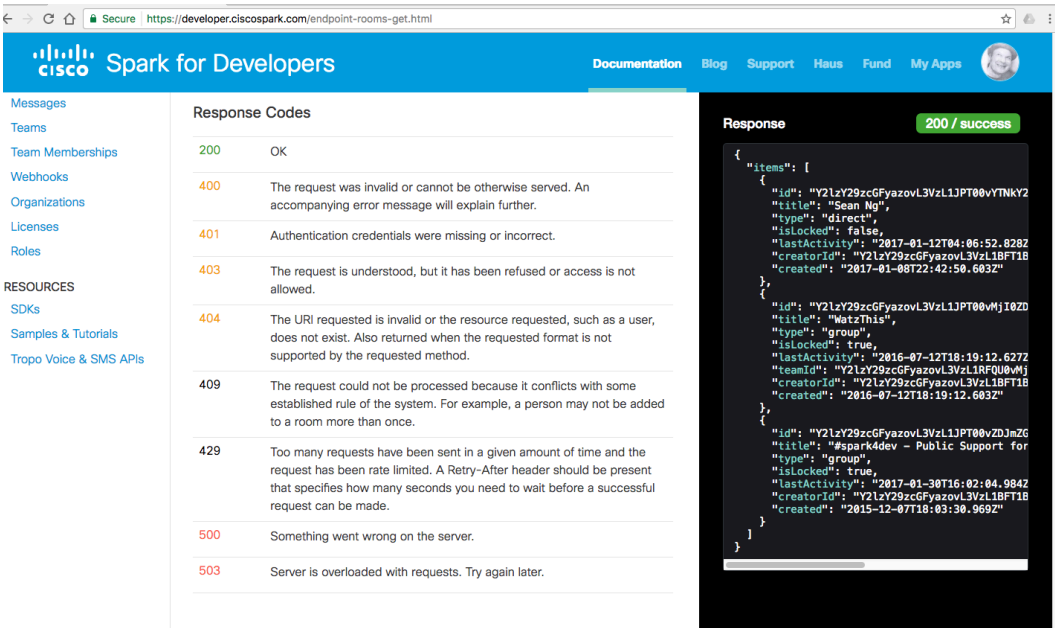


☐ 10. Click outside the pop-up window to return to the List Rooms screen.

☐ 11. In the List Rooms screen, leave the query parameters blank and click the **Run** button. A list of the rooms you belong to is returned, as shown in the following figure.



Notice the 200/success in the upper-right corner of the black response pane. This is the HTTP status code that was returned. Every HTTP response returns a status code. A code of 200 indicates that the request was successful. The other possible response codes are listed below the Test Mode form (scroll in the left pane to view), as shown in the following figure:



☐ 12. Scroll back up to the List Rooms Query Parameters. Enter the number **1** into the teamId field, under Your values, and click **Run**. You'll get a 500 status code and an error in the response pane, because 1 isn't a valid teamId.

☐ 13. Delete the **1** from the teamId field and click the **Run** button. The list of all your rooms will appear in the response pane.
☐ 14. Next, you'll use jQuery to request a list of Rooms from the Spark Rooms API. Find the lab09 folder in your jslabs/labs/JS100 directory and open the file named **listRooms.html** in the code editor of your choice.
☐ 15. Find the variable declaration for the `auth` variable. It looks like this:

```
var auth = "your access token here";
```

☐ 16. Return to **developer.ciscospark.com** and click on your user icon.
☐ 17. Copy your access token from the window the pops up and paste it between the quotes to replace the words "*your access token here*."
☐ 18. Start MAMP to make sure that your web server is running.
☐ 19. In your browser, go to http://localhost:8888/labs/JS100/lab09/listRooms.html. A web page with a button labeled List Rooms will open.
☐ 20. Click the **List Rooms** button.

Note: If the URL doesn't work above. Double-check that the Apache is selected as the web server and the document root is your jslabs folder. Refer to the steps in the Configuring an HTTP (Web) Server lab.

After a moment, JSON data will display in the browser window. This is the same data that you got when you clicked Run in Test Mode earlier.

Lab 09: Using JavaScript Functions

Functions are like programs within programs. Many different functions already exist inside JavaScript (and you saw examples of some of them in the previous labs).

But, what makes functions exciting is that you can write your own functions to accomplish common tasks inside your program.

To write your own function, you use a function definition.

A function definition has three parts:

- The function name
- The function parameters
- The function body

Here's an example of a simple function that adds numbers:

```
function addEmUp(number1,number2) {
  return number1 + number2;
}
```

To use this function, you use the name of the function, followed by parentheses. Inside the parentheses, you can put values to be used by the function. These values are called *arguments*. Inside the function, your arguments become variables with the names from the function definition.

For example:

```
addEmUp(1,2);
```

When you run this code, the function will produce (or *return*) the number 3 because it adds 1 and 2.

Functions are essential to making JavaScript programs that are modular, reusable, and "DRY", which is an acronym for **Don't Repeat Yourself**.

In this lab, you'll write a single function to format data from the Cisco Spark Rooms API in different ways.

☐ 1. You should still have the lab08 **listRoom.html** file open in your code editor. Go to that.

☐ 2. Make sure that your web server is running and visit http://localhost:8888/labs/JS100/lab08/listRooms.html in your browser.

Note: If the URL doesn't work above. Double check that the Apache is selected as the web server and the document root is your jslabs folder. Refer to the steps in the Configuring an HTTP (Web) Server lab.

☐ 3. Click the **List Rooms** button to verify that the script works.

☐ 4. Back in your code editor, place your cursor above the line that starts with `var auth =` in the `script` tags. Enter the following:

```
function formatter(json,style) {
    var output = '';
    if (style === "pretty") {
        for(var i = 0; i < json.items.length; i += 1) {
            output += "Id: " + json.items[i].id + "<br>";
            output += "Title: " + json.items[i].title +
"<br><br>";
        }
    } else {
        output = JSON.stringify(json);
    }
    return output;
}
```

Study the code for this function closely. Can you figure out what it will do?

☐ 5. Find the following block of code (near the bottom of the `<script>` element):

```
.done(function(data) {
    console.log(data);

    var output = JSON.stringify(data);
    $('#result').html(output);

});
```

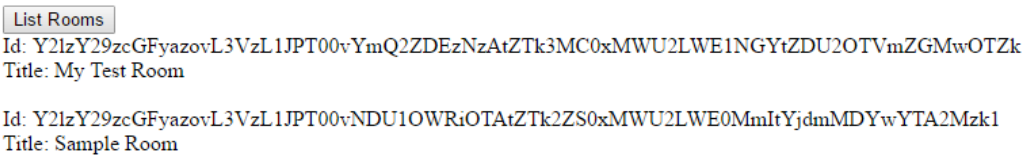
This code outputs the JSON data from the Rooms API into the `<div>` that has an id value of `"result"`. Because we want to format the JSON data using our new formatter function, we're going to modify this to process the JSON data with the function before putting it into the result element.

☐ 6. Update the `done()` method to look like this:

```
}).done(function(data) {
    console.log(data);
    var output = formatter(data,"pretty");
    $('#result').html(output);


});
```

The only difference here is that we're using our custom function, `formatter`, to format the data before we display it.

☐ 7. Refresh the page in your browser and click the **List Rooms** button again. You'll see a formatted version of the data, as shown below.



Now, we'll add a dropdown menu so that you can choose different output styles before pressing the button.

☐ 8. Type the following HTML before the `<input type="button" value="List Rooms" id="go" />` line in your HTML.

```
<select id="output_format">
    <option value="pretty">pretty</option>
    <option value="raw">raw</option>
</select>
```

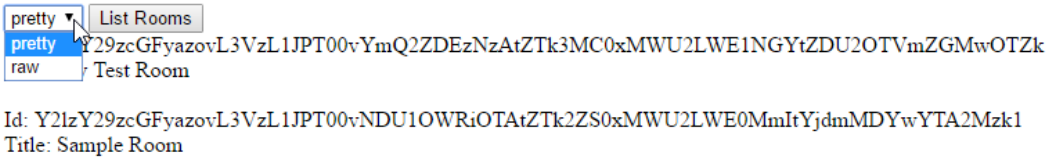
☐ 9. Add the following statement under the `console.log(data);` statement near the bottom of your script.

```
var selectedFormat = $('#output_format').val();
```

☐ 10. Inside the function call on the next line, change the `"pretty"` argument to use the value of `selectedFormat`, like this:

```
var output = formatter(data,selectedFormat);
```

☐ 11. Refresh the page in your browser and try it out!



Module: Tools and Techniques

Lab 01: Controlling Your Versions with Git

Git is a very popular version control system. There are visual tools for working with Git, including ones that are built into code editors. However, many professional developers prefer to work with Git through the command line, and knowing how to do so will make you a better developer. In this lab, you will learn some basic Git commands.

Note: If you haven't installed Git, follow the instructions in the Introduction and Setup Instructions section at the beginning of this document.

In this part, you will learn the most important commands for working with Git.

☐ 1. Open Terminal (on Mac) or Git Bash (on Windows).
☐ 2. Enter **`cd jslabs`** to go to the folder for this course.
☐ 3. Enter **`git status`**.

You should see that that there's nothing to commit and the working directory is clean. You will also see that you're on branch master. The branch master is the default branch of your local repository. In general, you should always aim to keep the branch master clean and working and use branches for any new code. We'll get back to that in a moment.

Note: When you cloned the repository for this course in the setup instructions, you created a local repository on your computer. You can make changes and commit to this local repository, and the changes won't affect the hosted version of the repository that you originally cloned.

The following table shows some of the basic Git commands.

Command	**Action**
`git add –all`	Stages all of your new and modified files.
`git commit -m '`*`first commit`*`'`	Commits your staged files to the repository. Replace the words '*first commit*' with a short description of what you've changed since the last commit.
`git log`	Displays the history of previous commits. Note that each commit has a

	unique identifier.
`git remote add origin [remote url]`	For users with a github.com account. Adds an origin so you can create a remote repository.
`git push -u origin master`	Pushes your changes to an origin once it's been added. The `-u` flag tells Git to remember the parameters. You can simply enter `git push` the next time you push.
`git pull origin master`	Retrieves the latest code from the origin.
`git diff HEAD`	Displays the diff of your most recent commit (called `HEAD`).

Note: If you have a github account, and want to try to create an origin, you can do so now. Make sure you first create the new repository at github.com, but don't initialize it with any files.

Now let's practice using some of these commands.

☐ 1. Type **vim README.md** to open the **README.md** file in vim. If README.md doesn't already exist, it will be created and then opened.

☐ 2. Enter vim's insert mode by pressing **i**. Add some text to README.md. For example:

```
# Modern JavaScript Literacy

## Installation

1. Install Node.js
2. Install git.
```

☐ 3. Save the file by pressing **Esc**, followed by **:wq**.

☐ 4. Enter **git diff HEAD**. You will see a list of differences between the current state of your files and the last commit.

☐ 5. If you're not returned to the normal command prompt after running the `git diff` command, press **q** to quit the editor.

☐ 6. Next, create a new **js** directory in jslabs:

```
mkdir js
```

☐ 7. Create a file inside of it called **app.js**:

```
touch js/app.js
```

☐ 8. Type **git add --all** to stage this new file.

☐ 9. Type **git diff --staged** to see what you have staged.

☐ 10. If you're not returned to the normal command prompt after running the `git diff` command, press **q** to quit the editor.

☐ 11. Type the following to unstage app.js:

```
git reset js/app.js
```

☐ 12. Type **`git diff`** to see that you've unstaged the file. Press **q** and then type **`git status`** to verify the file is unstaged again.

☐ 13. Stage the file again (**`git add --all`**).

☐ 14. You can change files back to how they were at the last commit using `git checkout`. Make some changes to README.md then run this command:

```
git checkout README.md
```

☐ 15. Commit your changes.

```
git commit -m 'updated readme and created js/app.js'
```

☐ 16. Branches are an essential and very frequently used part of Git. Any new feature or bug fix should be done in a branch and then merged back into master. To make a new branch, enter:

```
git branch my_first_branch
```

☐ 17. Once you've created a branch, you can switch to it like this:

```
git checkout my_first_branch
```

☐ 18. In the new branch, type **`vim ap.js`** to open the file in the vim editor, and then press **i** to enter insert mode.

☐ 19. Type the following in the ap.js file:

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

☐ 20. Save the file by pressing **Esc**, followed by **`:wq`**.

☐ 21. Use `git add` and `git commit` your changes. Remember to use a descriptive message.

Note: Refer to the earlier steps if you have questions on how to do this or as your instructor.

☐ 22. Switch back to the master branch using this command:

```
git checkout master
```

☐ 23. Merge your changes from `my_first_branch` back into master:

```
git merge my_first_branch
```

☐ 24. Delete your branch:

```
git branch -d my_first_branch
```

Lab 02: Initializing npm

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

☐ 1. Open a command line (Git Bash on Windows or Terminal on Mac) and navigate to the jslabs directory (type **`cd jslabs`**) and then navigate to **`/labs/FE100/lab02`**).

☐ 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. Notice that the project name and description have been picked up from your README file. Cool!

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. If you haven't exited npm yet, press **Ctrl+C** to do so now.

☐ 6. 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; however, a new folder named node_modules is created in your project.

☐ 7. Add this instruction to the README file on line 3, and then save the readme file:

```
In the console, type: npm install
```

In this lab, you set up your project to track its dependencies using npm. 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.

Lab 03: Using npm

Npm is the tool used to install and manage node modules created by the node community. In this lab, you will learn basic npm commands.

- ☐ 1. Open a command line (Git Bash on Windows or Terminal on Mac) and navigate to your jslabs folder (**`cd jslabs`**).
- ☐ 2. Enter **`npm -v`** to find out which version of npm is installed on your computer.
- ☐ 3. 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` in order to install as the super user.

- ☐ 4. Enter **`npm -v`** to see what version of npm is now installed.
- ☐ 5. Enter **`npm ls -g`**. This command lists all the packages that are currently installed on your computer. To see only packages installed in the jslabs project, simply type **`npm ls`** from the jslabs directory.
- ☐ 6. Enter **`npm help ls`**. This command shows you documentation for a package. On Windows, it may open in a browser or code editor, depending what your default action is for opening help files. On MacOS, the help file is displayed in the Terminal.
- ☐ 7. If the help file is displayed in the console window, type **`q`** to exit the help system.
- ☐ 8. Enter **`npm update`** or **`npm update -g`**. The `npm update` command searches the npm registry for newer versions of installed packages and installs them along with their dependencies.

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

Lab 04: Setting Up a Task Runner

In this lab, you will install and configure a task runner. A task runner can be used to automate many of the tasks involved in front-end development, such as testing, building, and deployment.

There are many different task runners available. We use Gulp in this lab.

☐ 1. Open your command line if you don't have it open already.
☐ 2. Type the following to install gulp.

```
npm install -g gulp-cli
```

Note: Remember that you may need to preface this command with `sudo` (on MacOS or Linux) in order for the global installation to work.

Note: The Gulp Command Line Interface (gulp-cli) is the only package that you will install globally. We do this for convenience so that you can simply enter gulp in order to run it. If you can't install gulp globally, you can install `gulp-cli` locally (without `-g`) and run it by entering **`./node_modules/.bin/gulp`**

☐ 3. Navigate to your local jslabs folder, and then run this command to Install gulp for your project.

```
npm install gulp --save-dev
```

Note: The `--save-dev` flag in this command instructs npm to add the package as a development dependency inside the package.json file.

☐ 4. Type **`gulp -v`** to confirm that you have Gulp installed.

```
$ gulp -v
[14:18:17] CLI version 1.2.2
[14:18:18] Local version 3.9.1
```

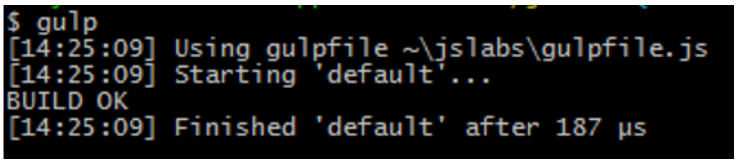
☐ 5. Type **`git add .`** (be sure to include the period) to add all the new files to Git.
☐ 6. Type **`git commit -m "installed gulp"`** to commit your changes.
☐ 7. Type **`touch gulpfile.js`** to create a new file in the root of your project.
☐ 8. Open the gulpfile.js file in your code editor and enter the following code to create your default task in gulpfile.js:

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


//default task

gulp.task('default', function(done) {
```

```
    console.log('BUILD OK');
    done();
});
```

☐ 9. Return to the command line, and type **`gulp`** at the command line to test your new (very simple) automated build.

☐ 10. If everything is working correctly, you should get the following output:



Note: If you get an error, you may have a previous version of gulp installed globally, in which case, you need to uninstall both the global and local versions and then re-install them. To uninstall both the global and local versions, run the following commands individually:

```
gulp rm -g gulp
npm install -g gulpjs/gulp-cli
npm uninstall gulp --save-dev
npm install --save-dev gulp
```

☐ 11. Return to your code editor and add the following to the top of the README.md file under the title information.

```
## Usage
To build:

1. gulp
```

☐ 12. Return to your command prompt, and enter **`git add .`** and **`git commit -m "`*`your comment here`*`"`** to commit everything and insert a comment about the changes you made.

☐ 13. Run **`git status`** to confirm that everything is clean.

Note: If you just type **`git commit`** here, you will be taken to the vim editor to enter the commit comment.

Lab 05: Automating Linting

In this lab, you will install ESLint and integrate it into your automated build.

☐ 1. If your command line isn't already open, open it and go to the jslabs folder.

☐ 2. Type **`npm install eslint --save-dev`** to install ESLint.

☐ 3. Run **`./node_modules/.bin/eslint --init`** to set up the configuration file.

☐ 4. Select **Answer questions about your style** as the answer to the first question.

☐ 5. Answer the questions as follows unless you have a good reason to answer differently. Don't worry if you make a mistake, we'll set all of the options correctly in the config file.

- Are you using ECMAScript 6 features? **Y**
- Are you using ES6 modules? **Y**
- Where will your code run? Select both **Browser** and **Node**
- Do you use CommonJS? **Y**
- Do you use JSX? **Y**
- Do you use React? **Y**
- What style of indentation do you use? (Your choice)
- What quotes do you use for strings? (Your choice)
- What line endings do you use? (Select Windows if you use **Windows**. Otherwise, select **Unix**)
- Do you require semicolons? **Y**
- What format do you want your config file to be in? **JavaScript**

Note: The init script may hang after the last question when using Git Bash shell. Use **Ctrl+C** to exit after the message appears that says "Successfully created .eslintrc.js".

☐ 6. Enter **`npm install --save-dev gulp-eslint`** to install the ESlint plugin for Gulp. You need this plugin so that you can run ESLint from Gulp.

☐ 7. Add `gulp-eslint` to the gulpfile.js file under the `const gulp = require('gulp');` line using this code.

```
const eslint = require('gulp-eslint');
```

☐ 8. Above the default gulp task, create a new task called **`eslint`**, as follows:

```
//eslint
gulp.task('eslint', function () {
    return gulp.src(['**/*.js','!node_modules/**'])
        .pipe(eslint())
```

```
            .pipe(eslint.format())
            .pipe(eslint.failAfterError());
    });
```

- ☐ 9. Run **`gulp eslint`**.
- ☐ 10. Fix the errors reported by ESLint, or adjust the `.eslintrc.js` config file (which was created in the root of the project earlier in these steps) to fit your desired coding style.
- ☐ 11. If you're on Windows, you may need to change the line break style to **`"windows"`**. You should also add a **`"no-console"`** option with a value of **`"warn"`** to override the default value of `"error"`, since we'll be using console.log in upcoming labs.

Here's an example of the .eslintrc.js file.

`.eslintrc.js`

```
module.exports = {
    "env": {
        "browser": true,
        "commonjs": true,
        "es6": true,
        "node": true
    },
    "extends": "eslint:recommended",
    "parserOptions": {
        "ecmaFeatures": {
            "experimentalObjectRestSpread": true,
            "jsx": true
        },
        "sourceType": "module"
    },
    "plugins": [
        "react"
    ],
    "rules": {
        "indent": [
            "warn"
        ],
        "linebreak-style": [
            "warn",
            "windows"
        ],
        "quotes": [
            "warn",
            "single"
        ],
        "semi": [
            "error",
```

```
            "always"
        ],
        "no-console": [
            "warn"
        ]
    }
};
```

Lab 06: Using Chrome Developer Tools: Sources Panel

The Sources Panel in the Chrome Developer Tools provides detailed information and tools for finding and fixing problems in JavaScript.

In this lab, you'll learn how to debug JavaScript using the sources panel.

☐ 1. Make sure that your MAMP server is running and go to http://localhost:8888/labs/FE100/lab06 in Chrome. The JavaScript Rocket Launcher page appears with a button for a countdown timer.

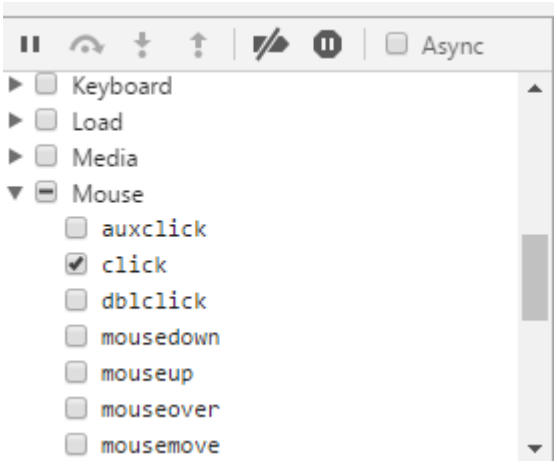
☐ 2. Press the **Start the Countdown!** button to start the countdown! The countdown will begin at T minus 10 seconds. The anticipation builds for the spectacular launch as the time approaches 0, and then...it fizzles out at 1 second left. The rocket never launches, because there's a terrible bug somewhere in the program.

☐ 3. Open the Chrome Developer Tools (**Command+Option+I** on Mac or **Ctrl+Shift+I** on Windows).

☐ 4. Click the **Sources** tab to open the Sources Panel. One of the most important features of the Sources Panel is that it lets you pause the execution of a script so that you can examine the values of your variables at that moment. The tool for pausing a program is called a *breakpoint*.

☐ 5. Click **Event Listener Breakpoints** (it either appears on the right of the Sources Panel or below it) to expand it. You'll see a list of event categories.

☐ 6. Scroll down, and then click the arrow next to the Mouse event category. Click the checkbox next to click.



☐ 7. Click the **Start the Countdown!** button in the browser window again. Execution of the script pauses and the first line of the `init()` function is highlighted.

This `init` function is the first thing that runs when this program starts. Its job is just to create the event listener that you just triggered by clicking on the button.

☐ 8. Click the **Step into next function call** button () to step through the function attached to the event listener. Notice that the Local variables area on the right shows the value of the `countDownTime` variable. Watch how it changes as you step through the function.

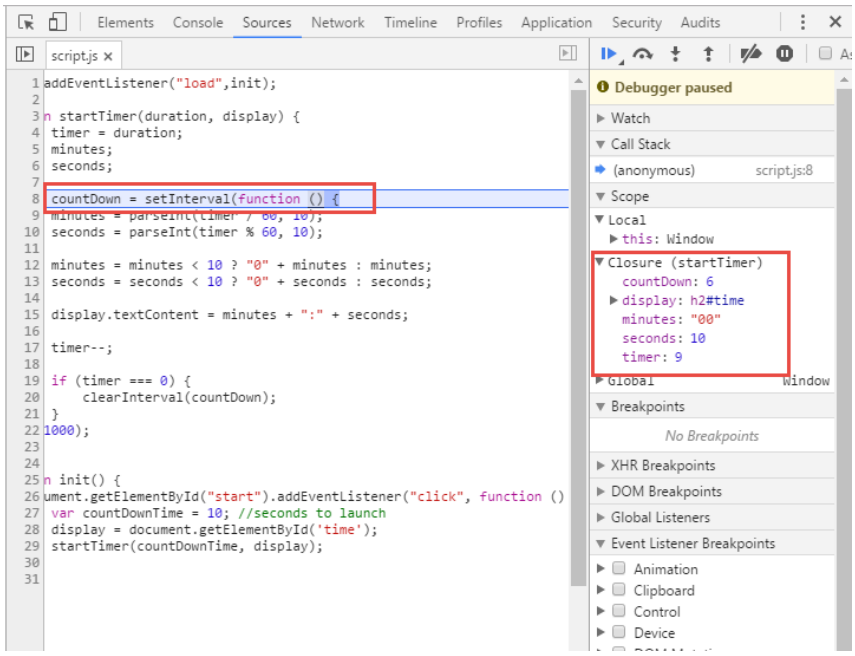
☐ 9. Press the **Step into next function call** button until you reach the `startTimer` function.

Note: If you have any browser extensions enabled, the debugger may make you step through the code for those as well. You'll know this is happening if any other files besides script.js open in the Sources tab as you're stepping through the code. If it happens, open the Extensions window (from the Window menu of Chrome) and disable all of the enabled extensions.

☐ 10. Keep going until you are inside the `countDown` part of the `startTimer` function, you'll notice a new expandable area under the Scope area on the right side called Closure.

A closure is a function inside of another function. It has its own private variables that aren't accessible to the program outside of it.

☐ 11. Expand the **Closure** item. You'll see the variables inside the closure.



☐ 12. Continue stepping through the code. Notice that every time you go past line 17 in the script.js file, the value of the timer variable goes down by one.

Can you guess what the statement on line 17 does? When the highlighted line of code is line 19, how does the value of the timer shown in the browser window compare to the value of the timer variable shown on the right side of the Sources Panel?

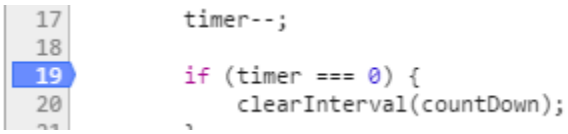
☐ 13. Look at line 19. This line tests the current value of the timer variable. When the timer reaches 0, it should run the `clearInterval` statement, which will cause the timer to stop. That's the idea, anyway.

☐ 14. Click on the **Resume Script Operation** button (). The countdown will continue. By now, you should have a suspicion that the problem with your timer is somewhere around line 17 or 19. We'll use a line-of-code breakpoint to further narrow it down and inspect the script.

☐ 15. Reload the page to reset the timer.

☐ 16. In the Event Listener Breakpoints, navigate to the Mouse section (if necessary), and uncheck the checkbox next to click.

☐ 17. In the Sources Panel, click on the line number for line 19. A blue icon will appear. This icon indicates that there's a breakpoint on this line.



☐ 18. Click on the **Start the Countdown!** button. The script starts running and stops when it gets to line 19.

☐ 19. Click on the **Resume Script Operation** button. The script runs until it gets back to line 19 for the next step in the countdown.

☐ 20. Each time the script pauses at line 19, inspect the value of the timer variable and what shows in the browser.

☐ 21. The problem with the script is that it changes the value of the timer between when it displays the current timer value in the browser and when it checks whether the timer is at 0.

This is why our rockets have been failing to launch! Let's test out a fix for this problem.

☐ 22. Inside the code editor built into the Sources Tab (where your code is being highlighted at breakpoints), delete the **`timer--;`** statement and then re-type it after the closing `}` of the `if` statement, like this:

```
if (timer === 0) {
  clearInterval(countDown);
}
timer--;
```

☐ 23. Press **Command+S** (Mac) or **Ctrl+S** (Windows) to save your changes. The background of the script area changes to red to indicate that you've made a change in the Sources tab.

Warning: When you make changes in the Sources tab, these changes aren't actually being made in the files themselves—just in the in-memory versions of them in the browser. Once you know your fix is good, you need to apply the change to the file using your code editor and save that file.

☐ 24. Click the **Start the Countdown!** button and watch the countdown.
☐ 25. If the fix worked, make the same change to your **script.js** file.

Lab 07: Getting Started with Jasmine

Jasmine is a behavior-driven development framework for JavaScript. In this lab, you will install Jasmine and use it to create your first test suite.

☐ 1. In your terminal or command line, navigate to the **labs/FE100/lab07** folder

☐ 2. Enter the following command to set up the node_modules folder for this directory and accept all the default values for the npm initialization.

```
npm init -y
```

☐ 3. Enter the following command to install jasmine:

```
npm install --save-dev jasmine
```

☐ 4. Initialize jasmine

```
./node_modules/.bin/jasmine init
```

A new folder, named **spec**, will be created. This is where you should put your specs. It also contains a directory named **support**, which contains the jasmine configuration file, jasmine.json.

☐ 5. Open the code editor of your choice and create a new file named **sayHello.js** in the lab07 directory.

☐ 6. create a file named **sayHelloSpec.js** in the **labs/FE100/lab07/spec** folder.

You're going to write a function in sayHello.js that will accept a name as an argument and will return the word "Hello" followed by the name. It's an extremely simple function to write, but we're going to approach it from a TDD perspective and write tests for it first.

☐ 7. Start the following new **suite** in **sayHelloSpec.js**:

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

});
```

☐ 8. Save your spec and let's test it out!

☐ 9. In your command line, enter:

```
./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
```

☐ 10. Jasmine will tell you that you don't have any specs.

```
Mac-mini-2:lab01 chrisjminnick$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started

No specs found
Finished in 0.001 seconds

Mac-mini-2:lab01 chrisjminnick$
```

☐ 11. Inside your first test suite in **sayHelloSpec.js**, create a new spec:

```
describe('Greet', function() {
    it('concats Hello and a name', function() {


    });
)};
```

☐ 12. Run your test again:

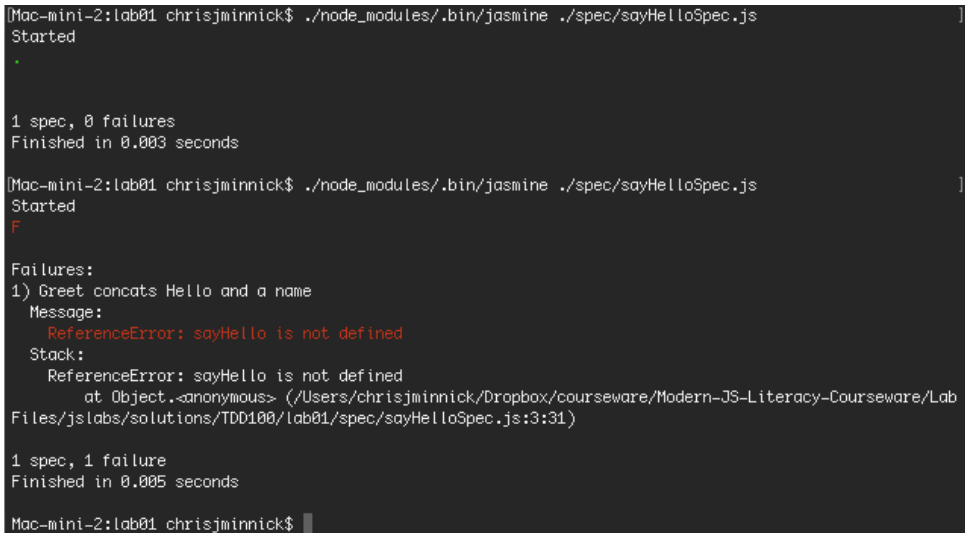
./node_modules/.bin/jasmine ./spec/sayHelloSpec.js

```
Mac-mini-2:lab01 chrisjminnick$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started

No specs found
Finished in 0.001 seconds

Mac-mini-2:lab01 chrisjminnick$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
.


1 spec, 0 failures
Finished in 0.003 seconds

Mac-mini-2:lab01 chrisjminnick$
```

☐ 13. Success! But…we're not testing anything yet. Let's create an **expectation**:

```
it('concats Hello and a name', function() {
    var actual = sayHello.greet('World');
    var expected = 'Hello, World';
    expect(actual).toEqual(expected);
});
```

☐ 14. Run your suite again. Jasmine will complain that it doesn't know what sayHello is.



Excellent. Now we're at what it called a "red bar". Our goal is to get to green. The first thing to solve is that our suite doesn't include the sayHello.js file.

☐ 15. Use CommonJS to require **sayHello.js** as sayHello inside **sayHelloSpec.js**. Enter the following on the first line.

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

☐ 16. Switch to the sayHello.js file or open it if necessary, and then write the bare minimum amount of code to get the test to pass. For example:

```
exports.greet = function greet(name) {

    return 'Hello, ' + name;

};
```

☐ 17. Save the file if necessary, and then run your suite. It should now pass. If it doesn't, figure out why and get it to pass.

```
Mac-mini-2:lab01 chrisjminnick$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
F

Failures:
1) Greet concats Hello and a name
  Message:
    ReferenceError: sayHello is not defined
  Stack:
    ReferenceError: sayHello is not defined
        at Object.<anonymous> (/Users/chrisjminnick/Dropbox/courseware/Modern-JS-Literacy-Courseware/Lab
Files/jslabs/solutions/TDD100/lab01/spec/sayHelloSpec.js:3:31)

1 spec, 1 failure
Finished in 0.005 seconds

Mac-mini-2:lab01 chrisjminnick$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
.


1 spec, 0 failures
Finished in 0.006 seconds

Mac-mini-2:lab01 chrisjminnick$
```

- ☐ 18. Now it's time to refactor. Can you think of any changes you would make to your spec or your `greet()` function that would make it better or more understandable? Make them.
- ☐ 19. Repeat. What else could go wrong? Think of values (or lack of values) that would make your function break or behave in a way you don't want. For example, what happens when no name argument is passed? What should happen?
- ☐ 20. Create a new spec describing what your desired result should be when there's no name argument passed to `greet()`.

```
Mac-mini-2:lab01 chrisjminnick$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
.


1 spec, 0 failures
Finished in 0.006 seconds

Mac-mini-2:lab01 chrisjminnick$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
.F

Failures:
1) Greet says Hello, Friend! when no name is given
  Message:
    Expected 'Hello, undefined' to equal 'Hello, Friend!'.
  Stack:
    Error: Expected 'Hello, undefined' to equal 'Hello, Friend!'.
        at Object.<anonymous> (/Users/chrisjminnick/Dropbox/courseware/Modern-JS-Literacy-Courseware/Lab
Files/jslabs/solutions/TDD100/lab01/spec/sayHelloSpec.js:12:24)

2 specs, 1 failure
Finished in 0.007 seconds

Mac-mini-2:lab01 chrisjminnick$
```

- ☐ 21. Write code to make the test pass.

- ☐ 22. Refactor. Can you make the code you just wrote better? Can you improve this spec? If so, do it.
- ☐ 23. Repeat. Can you think of anything else that might break this function or make it behave in a way you don't want? Write another test to check for this condition and then write code to pass the test.
- ☐ 24. If you're using ESLint, you may get errors due to Jasmine's functions not being defined within your project. Fix this problem by adding jasmine as an environment in the ESLint config file (.eslintrc).

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 1.0, June 2017
by Chris Minnick

www.watzthis.com

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 6.x branch.
2. Install a code editor. We use WebStorm in the course. A 30-day trial version is available from *http://www.jetbrains.com/webstorm.*
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 of 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.

- Enter `cd intro-to-node` to switch to the new directory.
- 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.

- When everything is done, enter `npm run build`
- 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`.

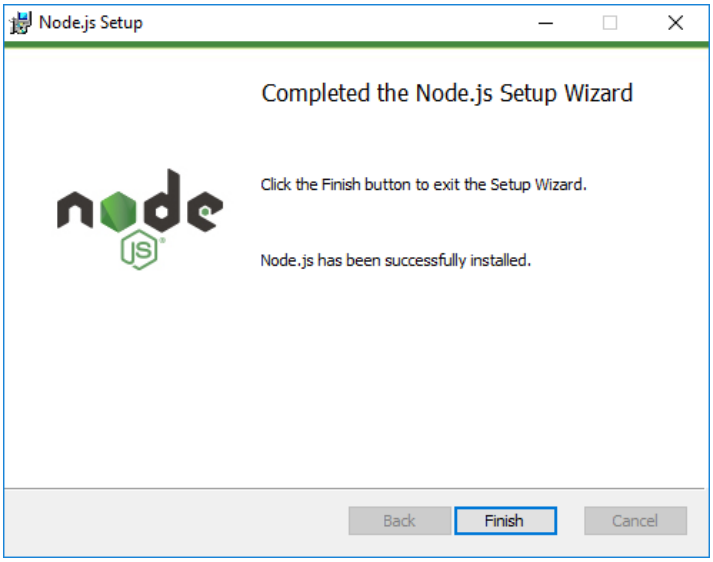
- A series of things will happen and then a message will appear and tell you that the test passed.

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

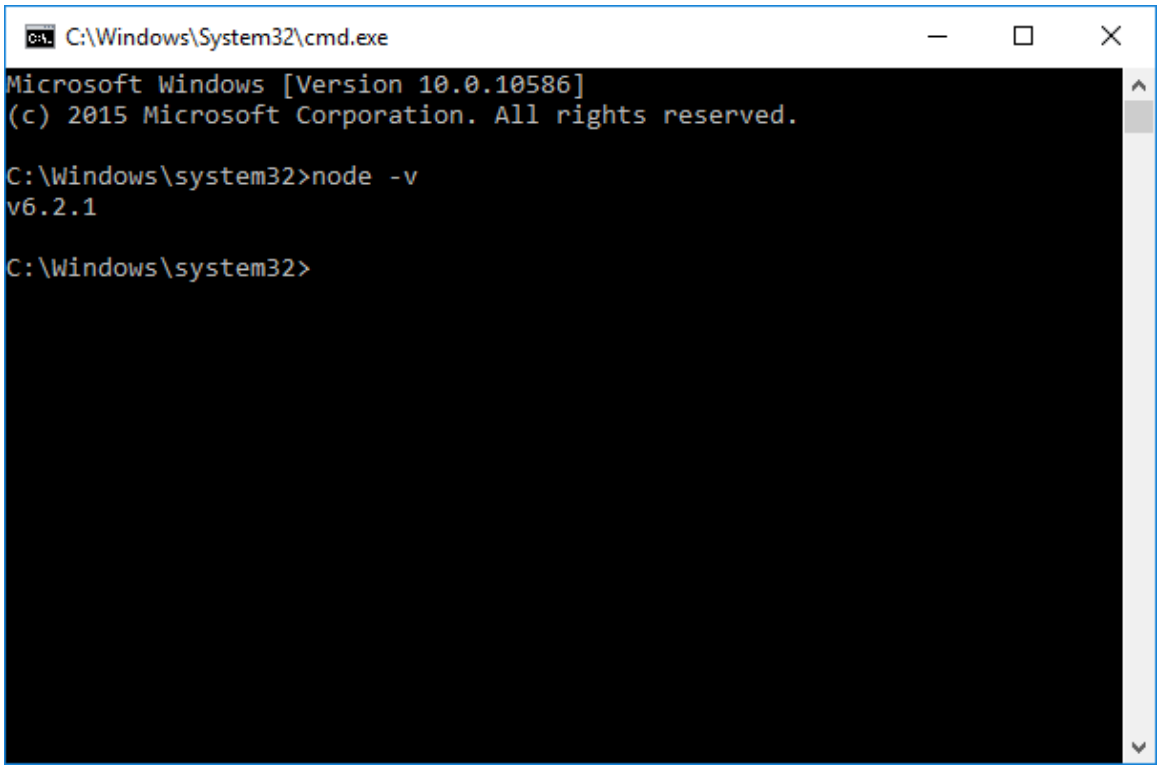
- ☐ 1. Go to *https://nodejs.org* and download the latest version of Node from the LTS (Long Term Support) branch.
- ☐ 2. When it finishes downloading, launch the installer to install Node.js
- ☐ 3. Select all the default options.



Part 2: Getting to Know Node.js

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

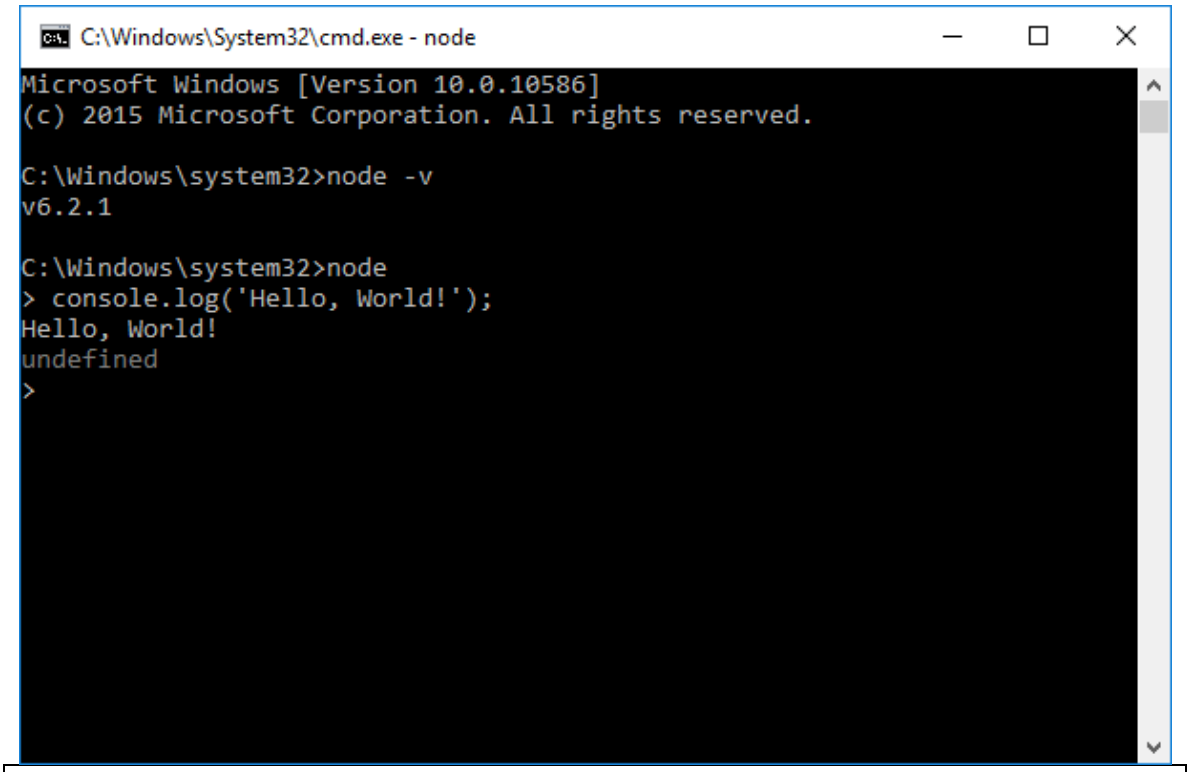
- ☐ 1. Open a command line application.
  a. MacOS: Navigate to Applications / Utilities and double click on **Terminal**.
  b. Windows 7, 8, or 10: Open a search box and enter **cmd** to locate the Command Prompt. Open it.
- ☐ 2. To check whether Node.js is property installed, enter `node -v`
  You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

You will not normally work with node from the interactive shell. 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 **labs/NJS100/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 **labs/NJS100/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.

Part 3: Using 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.

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

☐ 12. 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.

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

☐ 14. 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.

☐ 15. 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.

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

☐ 17. 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.

Lab 02: First Look at Asynchronous Code

Perhaps one of the most difficult things for beginners to understand about Node is its asynchronous 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 count to 10.

```
console.log('starting...');
for(var i=1; i<11; i++){
  console.log(i);
}
console.log('Done!');
```

☐ 3. 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 'starting...' followed by the numbers 1 through 10, followed by 'Done!'.

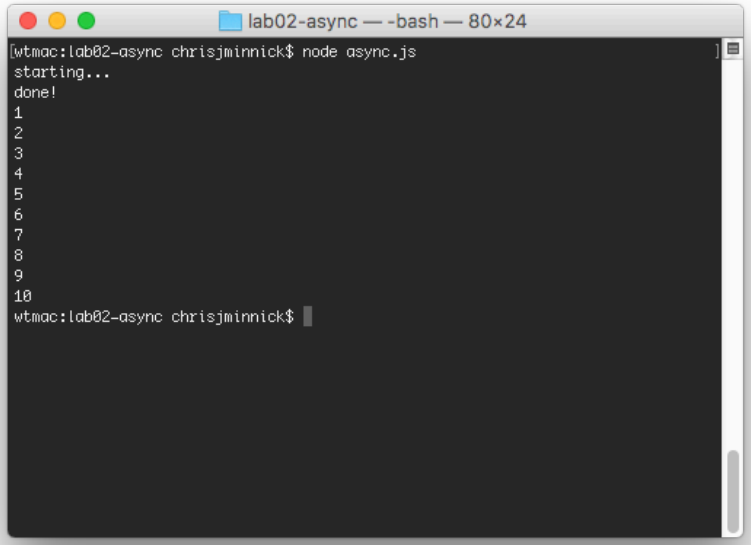
☐ 4. In your code editor, create another file inside the **labs/lab02** folder and name the file **async.js**.

☐ 5. Inside of **async.js**, write the following code:

```
console.log('starting...');

process.nextTick(function(){
    for (var i=1; i<11; i++){
        console.log(i);
    }
});

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

☐ 6. 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 numbers.
Can you explain what's happening here?
Can you modify the asynchronous program to produce the same output as the synchronous program?

Lab 03: 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/lab03** 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/lab03** folder and run the program by typing:

```
node app.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 04: 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/lab04** 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 **lab04** 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/lab04** 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.

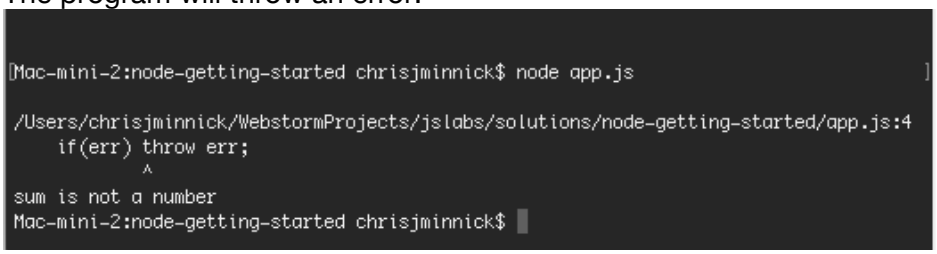
☐ 14. In the app.js file, update the arguments you pass into the module so that at least one of them isn't a number.

```
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.



☐ 17. Go back to your **app.js** file and experiment a bit. Change the numbers in the following line, save the file, run the program, and marvel at your results:

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

Lab 05: 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.

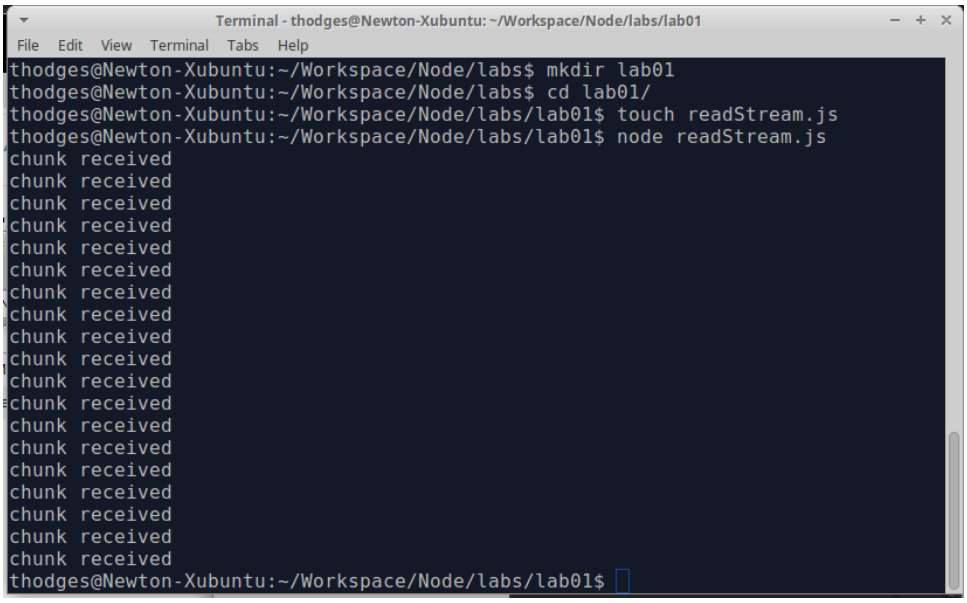
☐ 4. Create an event listener for `myReadStream` that logs to the console whenever a buffer of data is received:

```
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 has buffered is ready to be processed.

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

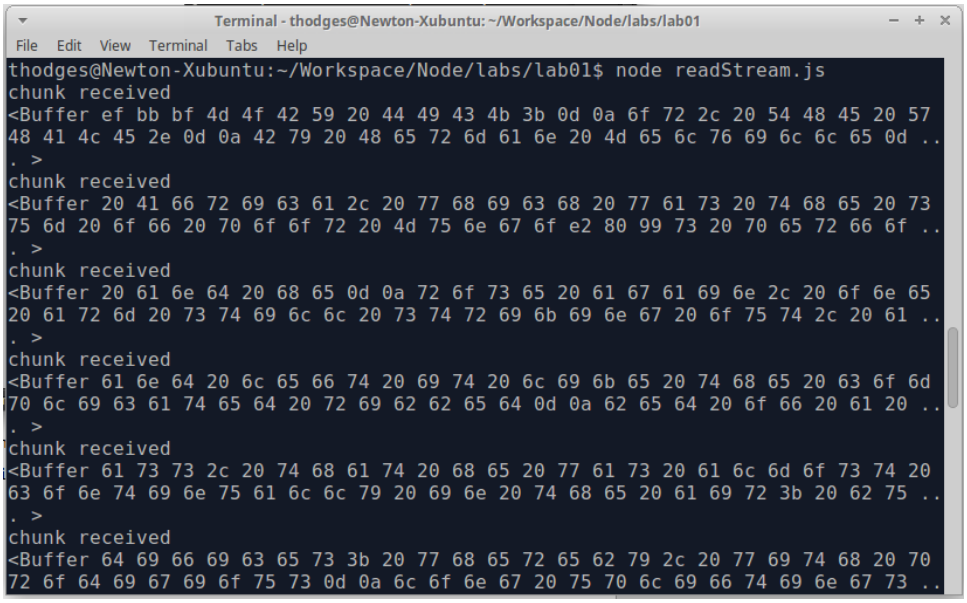
```
node readStream.js
```



Moby Dick is large enough that it is sent to us in nineteen pieces! Each of these data buffers is stored in the variable buffer 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`:

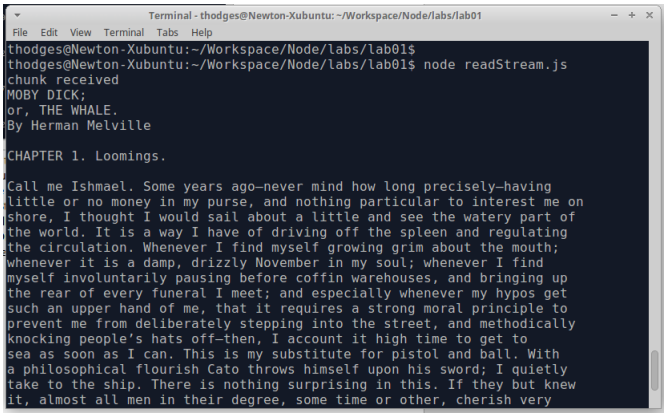
```
var myReadStream =
      fs.createReadStream(__dirname + '/MobyDick.txt',
'utf8');
```

☐ 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 **lab05** 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 06: 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 **lab06** 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 **lab06** directory to confirm that it creates the **WriteMe.txt** file.

Read Streams inherent the readable.pipe() method which attaches 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 **lab06** 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.

☐ 13. 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');
```

☐ 14. 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.

☐ 15. 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 shmaellay. 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.

☐ 16. 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 7: 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);
```

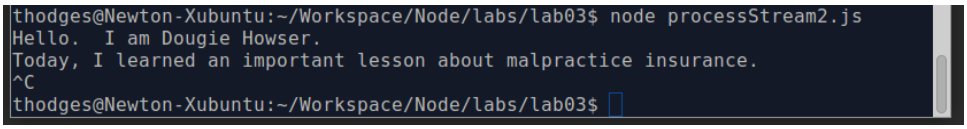
☐ 2. Now run `processStream.js` with or without parameters and the text of Moby Dick should scroll rapidly on your screen.

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);
```

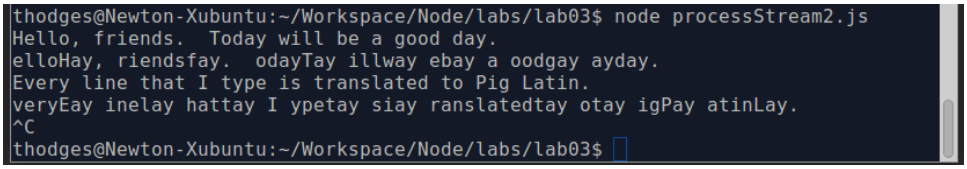
☐ 4. This program should take text that you type into the terminal and write it to `WriteMe.txt` each time you press `Enter` until you break out with ctrl-c. When you are done, open `WriteMe.txt` to verify that the streams worked as expected.



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);
```


thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab03$ node processStream2.js
Hello, friends. Today will be a good day.
elloHay, riendsfay. odayTay illway ebay a oodgay ayday.
Every line that I type is translated to Pig Latin.
veryEay inelay hattay I ypetay siay ranslatedtay otay igPay atinLay.
^C
thodges@Newton-Xubuntu:~/Workspace/Node/labs/lab03$

Lab 08: 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:

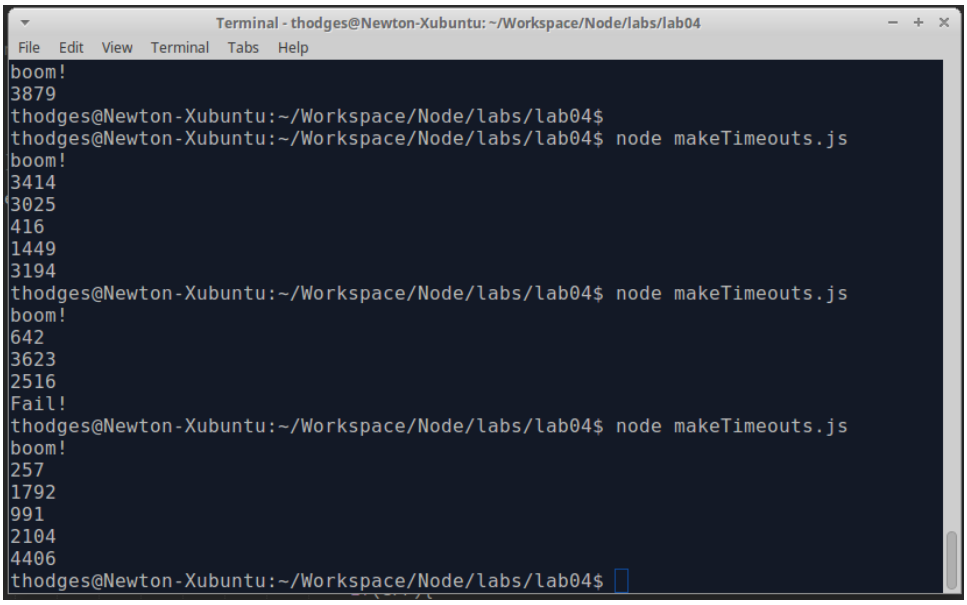
```
makeTimeouts(1000, (err, data) => {
    if(err){
        console.error(err);
    } else {
```

```
            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.

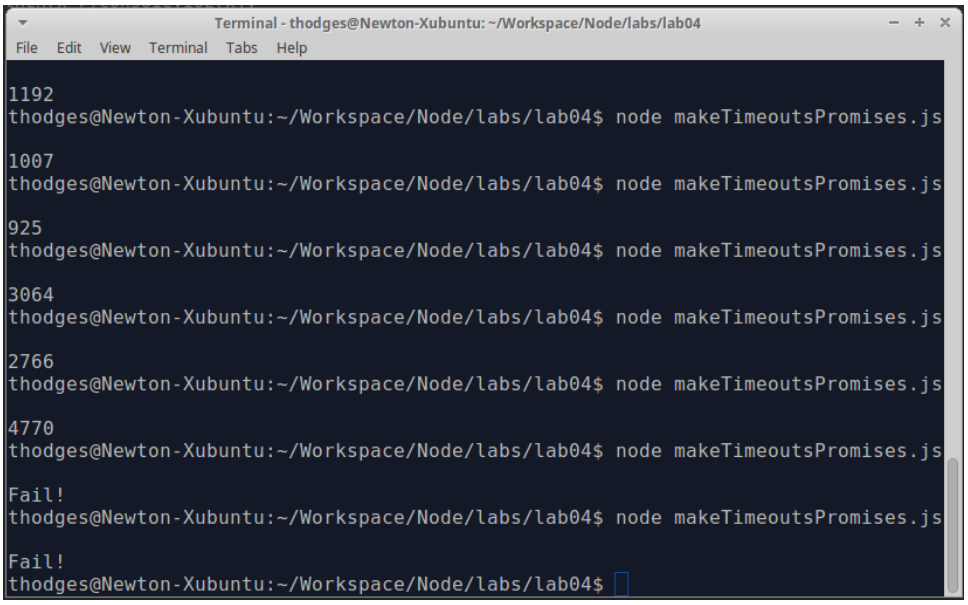
☐ 7. Add a parameter to the `makeTimeoutsPromises` arrow function and wrap all of the code inside of the promise in a `setTimeout()` using that parameter.

```
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));
```

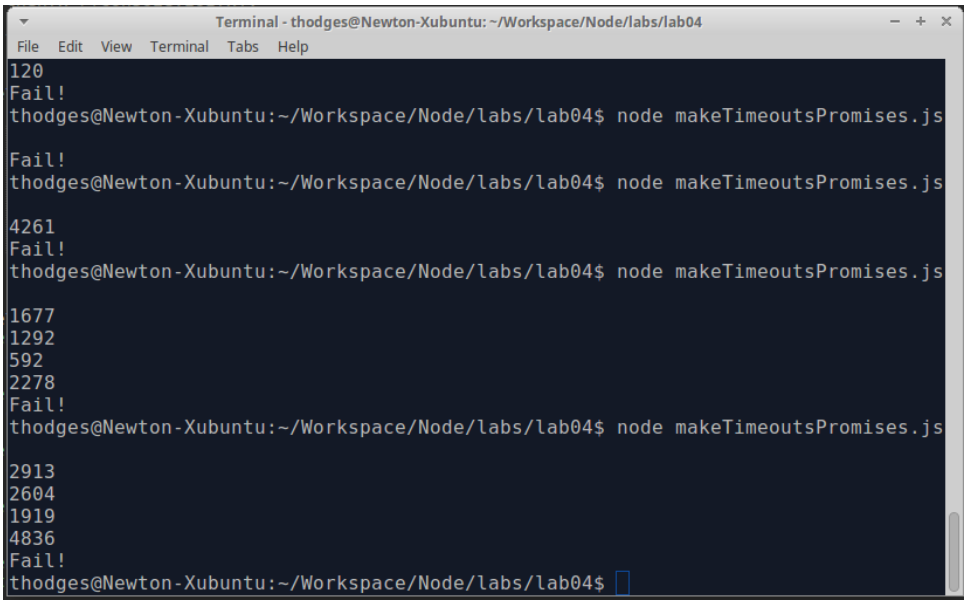
Although resembling chained functions, this code is useless to you as it stands. If you run the file you will see that the first `.then()` (often) returns a value, while the second always returns undefined. Our first `.then()` has handled our settled promise and our second `.then()` has nothing to work 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 09: 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 **lab09** 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);
});
```

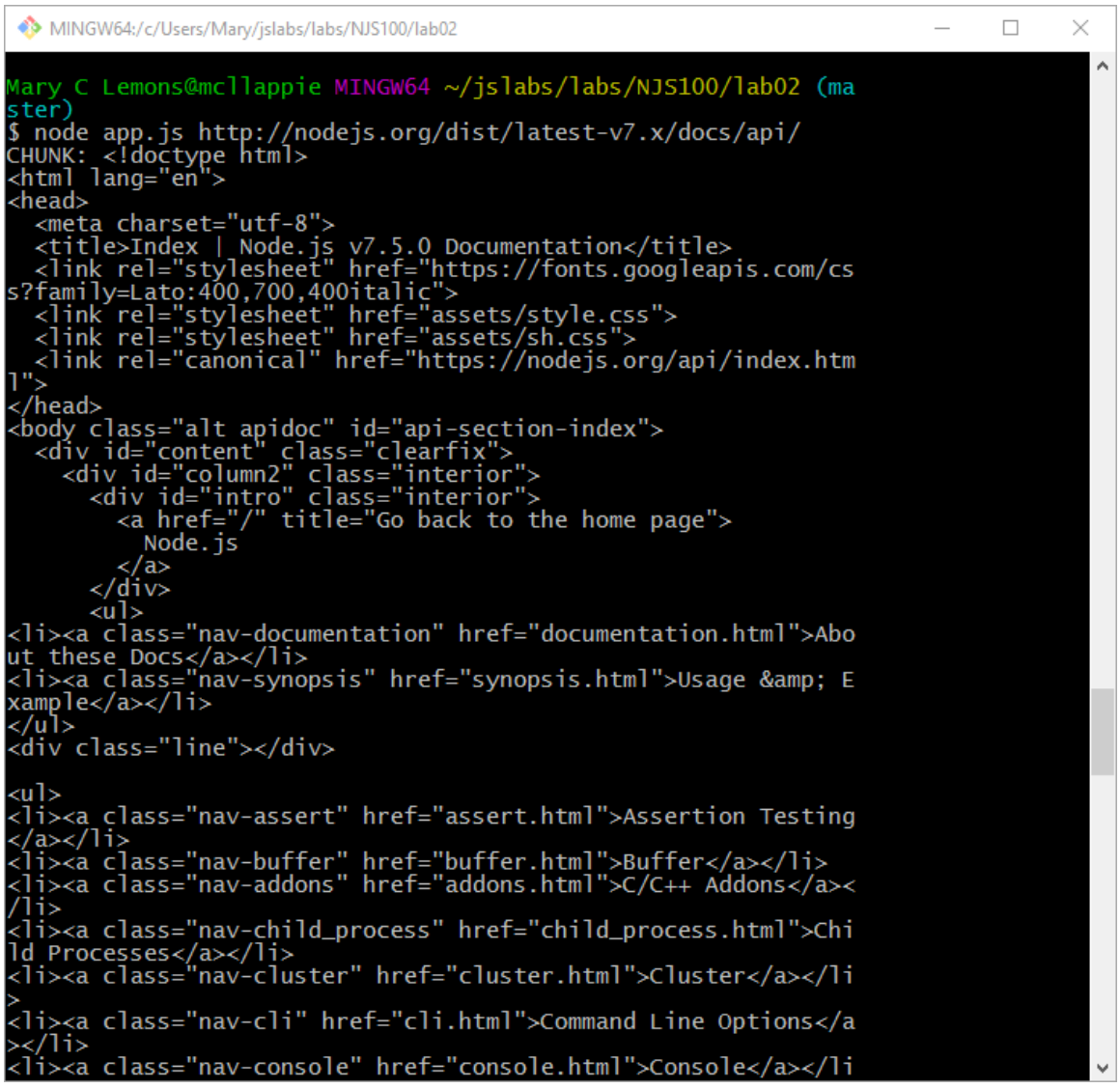
☐ 8. The final program should look like this:

```
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 **lab09** 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 10: Installing and Running a Spark Bot

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

☐ 1. Download ngrok from http://ngrok.com/download.

☐ 2. 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.

☐ 3. Expose the bot you'll build to the Internet, using http, on port 8080. On a Mac, type:

```
./ngrok http 8080
```

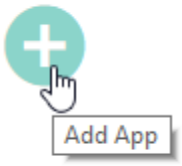
For Windows, type:

```
ngrok http 8080
```

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

☐ 5. Click **My Apps**.

☐ 6. Click the plus sign in the upper-right to create a new app.



☐ 7. Click Create a Bot.

☐ 8. 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). Paste a URL for an image you want to use. We used this URL: http://images.clipartpanda.com/robots-clipart-robot5.png.

☐ 9. Copy the Access token that you get when you click the Add Bot 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.

☐ 10. Open your **Cisco Spark** app. Click the plus sign to search for a user to add to the room you are creating. Enter your bot's username into the

search field to create a room with your bot as a participant. Click **Go Chat**.

☐ 11. 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 Sparkbot samples with this command:

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

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

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

☐ 13. Install the samples.

```
npm install
```

☐ 14. Make the examples directory the working directory.

```
cd examples
```

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

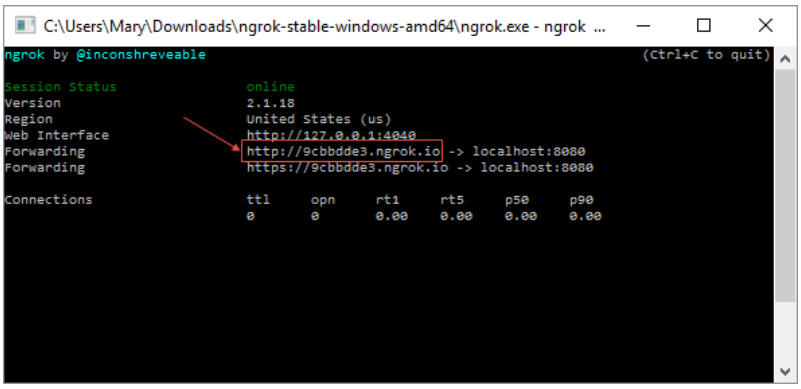
```
SPARK_TOKEN=token DEBUG=sparkbot* node helloworld.js
```

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

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

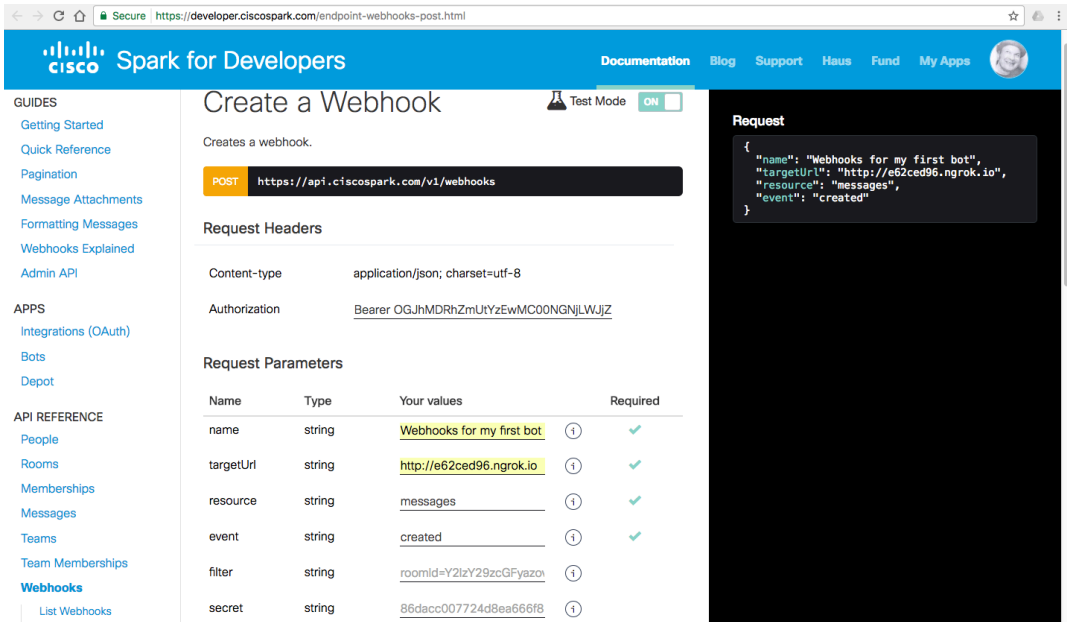
https://developer.ciscospark.com/endpoint-webhooks-post.html

☐ 17. Click **Test Mode** if you're not already in test mode.
☐ 18. Change the authorization code (leave Bearer in place) to the bot's code.
☐ 19. Enter a name (anything will do).
☐ 20. Make the **target url** be your ngrok http url that's displayed in the ngrok command-line window.



☐ 21. Set resources to **messages**.

☐ 22. Set event to **created**.



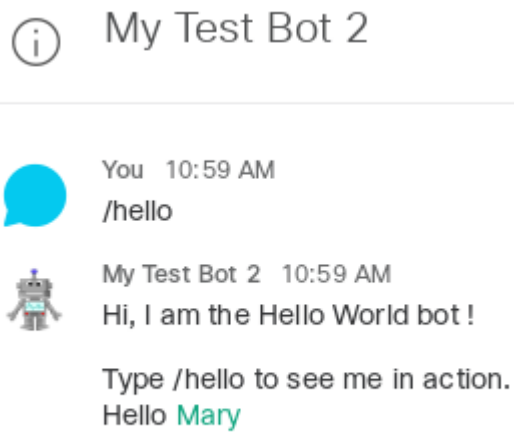
☐ 23. Click **Run**. Your first Webhook will be created, which will listen for messages that to your bot.

☐ 24. 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. Change resources to **memberships**.

☐ 25. Click **Run** again to create a second webhook.

☐ 26. Go into the 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 11: 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 steps 23-26 in the previous lab.)

☐ 1. Open your terminal application and type the following to change the working directory to **labs/lab11**.

```
cd labs/lab11
```

☐ 2. Initialize npm in the directory by typing:

```
npm init
```

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

☐ 3. Install node-sparkclient, by typing

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

☐ 4. In your code editor, create a new file named **hellobot.js** inside the **lab11** folder.

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

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

☐ 6. Create an instance of the SparkBot

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

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

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

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

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

☐ 9. 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;
    }
```

☐ 10. 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);
```

☐ 11. 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;
    }
});
});
```

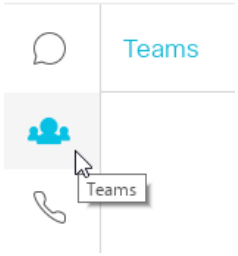
☐ 12. Save your **hellobot.js** file if necessary.

☐ 13. In your console or terminal application, change the working directory to **lab11** if it's not already open and start your bot:

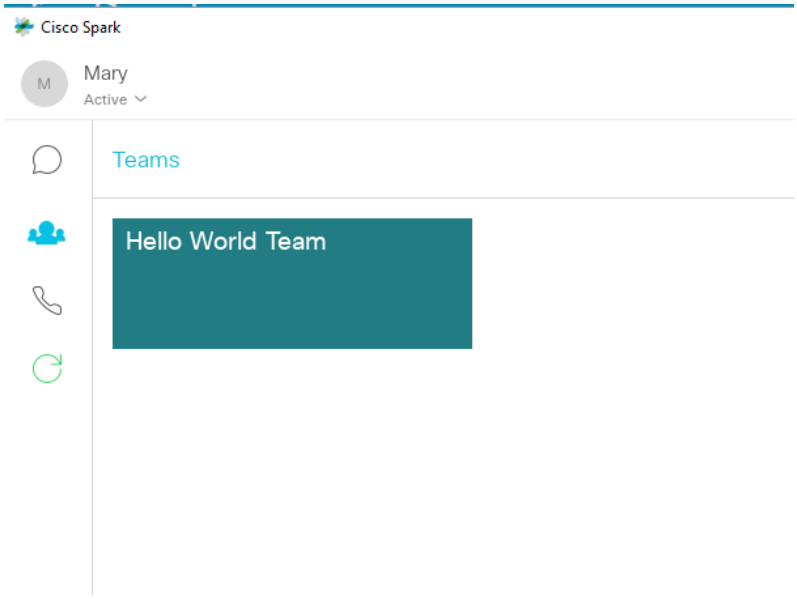
```
SPARK_TOKEN=XXXXXXXXXXX DEBUG=sparkbot* node
hellobot.js
```

Note: In the above command, replace the Xs after SPARK_TOKEN with your bot's Id.

☐ 14. Go to your Cisco Spark app and click the **Teams** button.

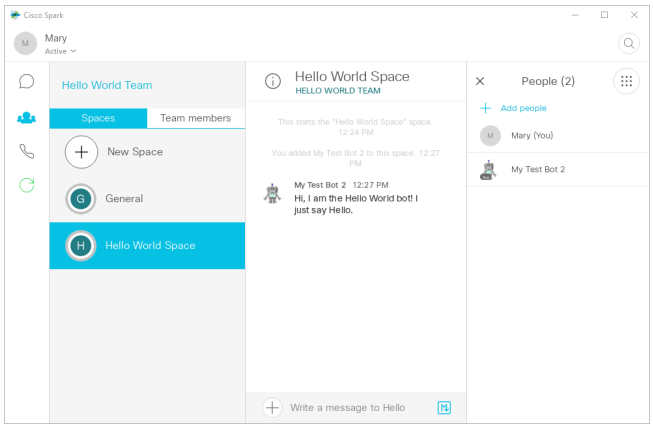


☐ 15. Click New Team.
☐ 16. Name your team. We'll call ours **Hello World Team**. Click **Create** after you've named your team.



☐ 17. Click the **Hello World Team** team (or whatever you named yours).
☐ 18. 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.
☐ 19. Click the button with nine dots in it, and then click **People**.

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


Cisco Spark
Mary
Active
Hello World Team
Spaces
Team members
New Space
General
Hello World Space
Hello World Space
HELLO WORLD TEAM
This starts the "Hello World Space" space. 12:24 PM
You added My Test Bot 2 to this space. 12:27 PM
My Test Bot 2 12:27 PM
Hi, I am the Hello World bot! I just say Hello.
Write a message to Hello
People (2)
Add people
Mary (You)
My Test Bot 2

Lab 12: Assert

Assert is a tool used in node for unit testing. 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()`. We'll work with a few of these in today's lab.

Part 1: assert.equal()

In your **lab12** 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 considered to be 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:

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

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

In the world of Asserts, no news is good news and silence is golden. If you see no output, that means that your assert 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);
});
```

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

assert.js:85
  throw new assert.AssertionError({
  ^
AssertionError: undefined == 1
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/testAssert.js:5:9)
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/sumModule.js:6:6)
    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$
```

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);
});
```

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

assert.js:85
  throw new assert.AssertionError({
  ^
AssertionError: all inputs must be positive
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/testAssert.js:5:9)
    at sumModule (/home/thodges/Desktop/Lab09/Sltn/Part1/sumModule.js:6:6)
    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$
```

☐ 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);
});
```

Lab 13: 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 it's 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);
```

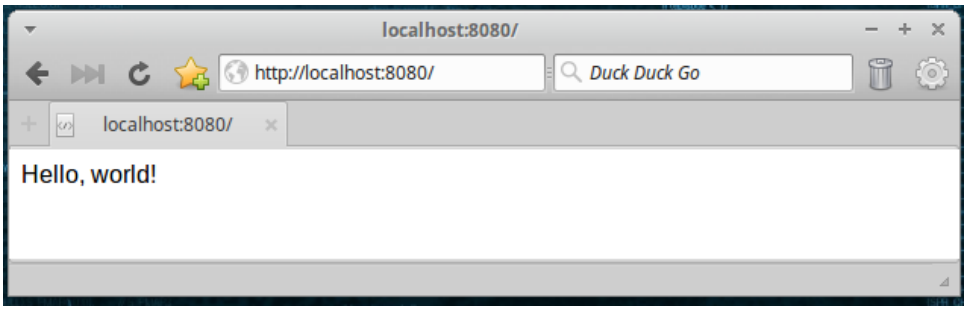
You may already be familiar with the basic HTTP methods GET, POST, DELETE and PUT. Express has handlers for all of these methods, and these handlers are called with two parameters: a route and a function. The function has two parameters, `req` and `res`, which correspond to the request and the response.

☐ 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.

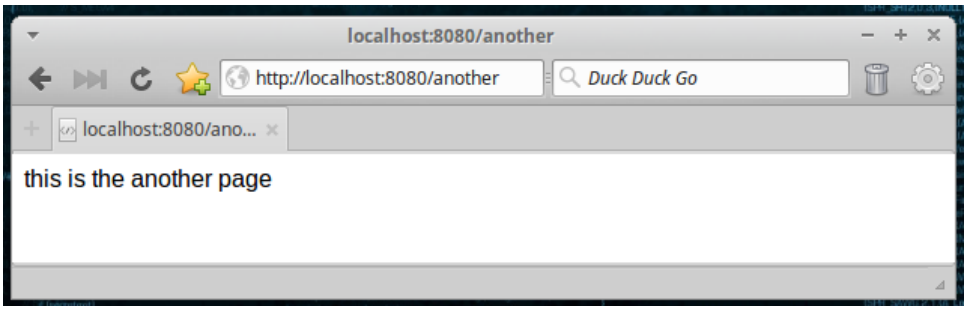
☐ 6. In your web browser, open `localhost:8080` to see the result.



☐ 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'`).

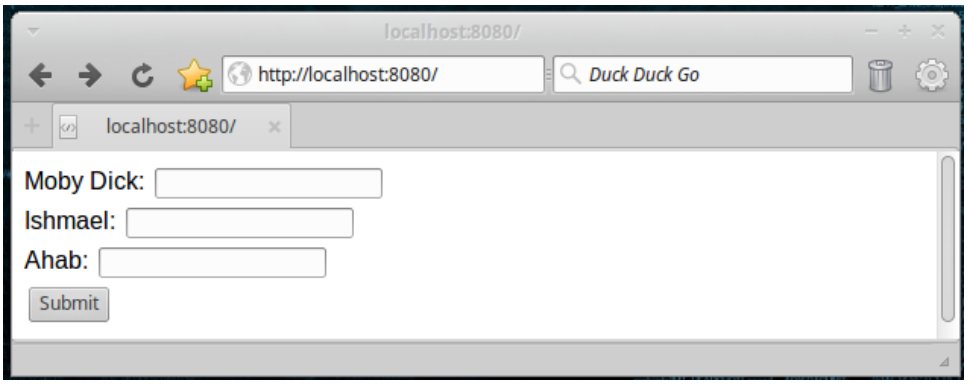
```
var express = require('express');
var app = express();
app.listen(8080);
app.get('/', (req, res) => {
    res.send('Hello, world!');
});
```

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`:

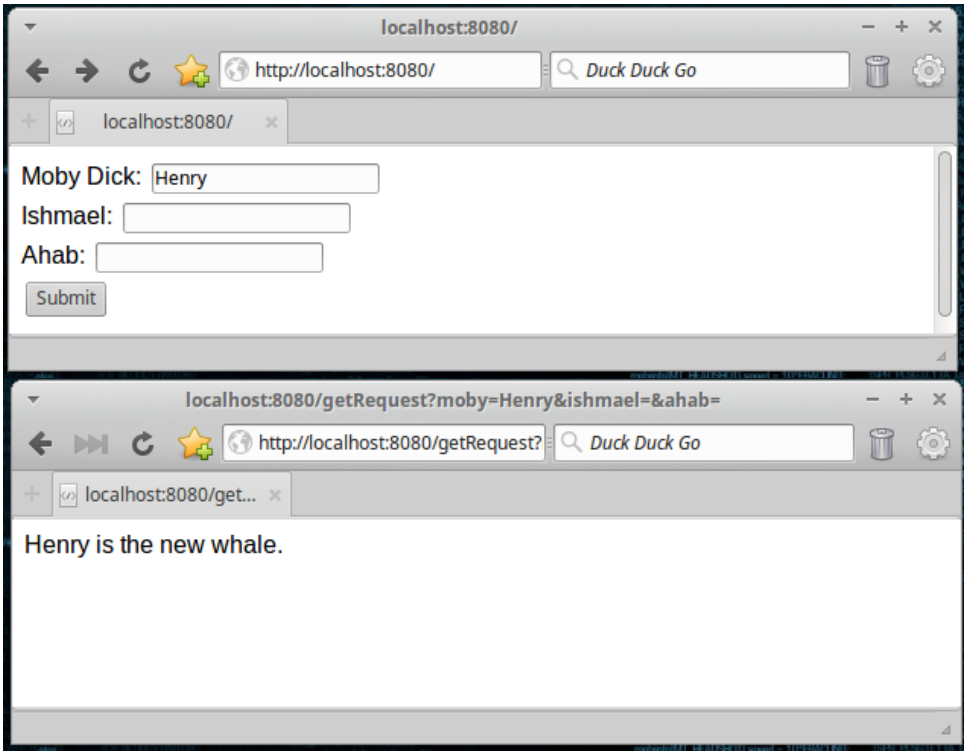
```
app.get('/', (req, res) => {
    res.sendFile( __dirname + "/" + "get_request.html"
);
});

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`.

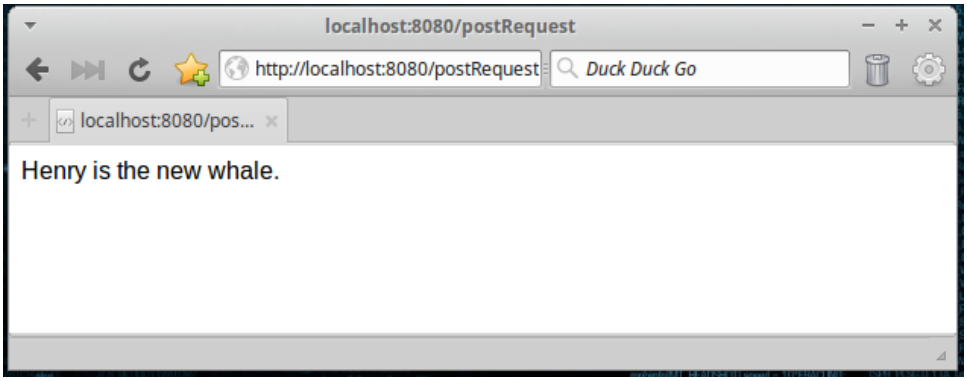
```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
```

☐ 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');
```

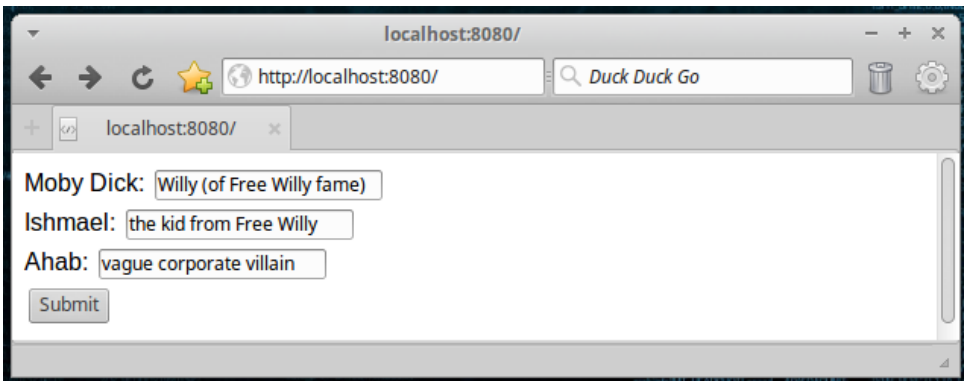
☐ 4. We are going to define our transform streams inside of our `app.get()` function corresponding to '`/getRequest`.' I'm not going to bother to change the route name because I would like to use the file already created for **get_request.html**, and that has '`/getRequest`' hard coded.

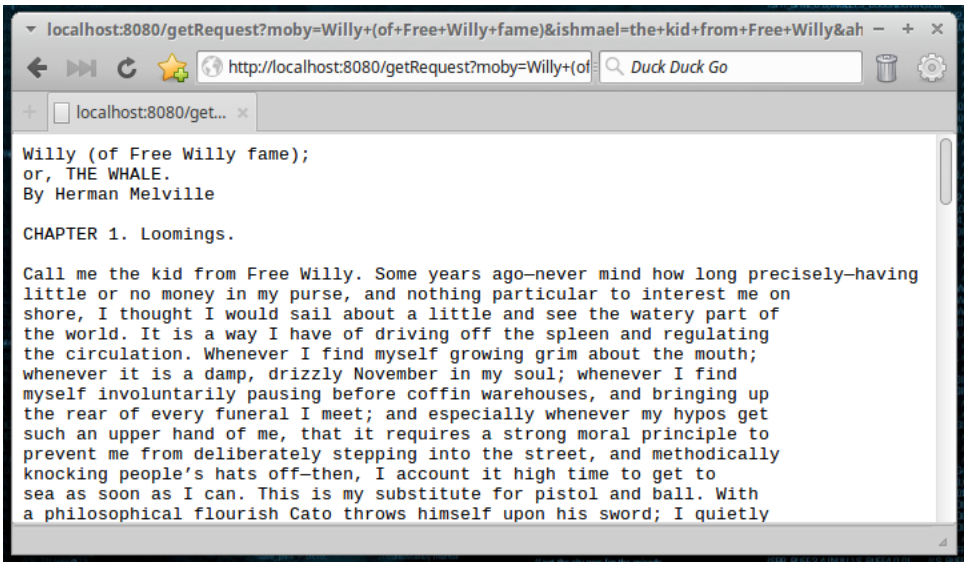
```
app.get('/getRequest', (req, res) => {
    var newMoby = req.query.moby;
    var newIshmael = req.query.ishmael;
    var newAhab = req.query.ahab;

    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.