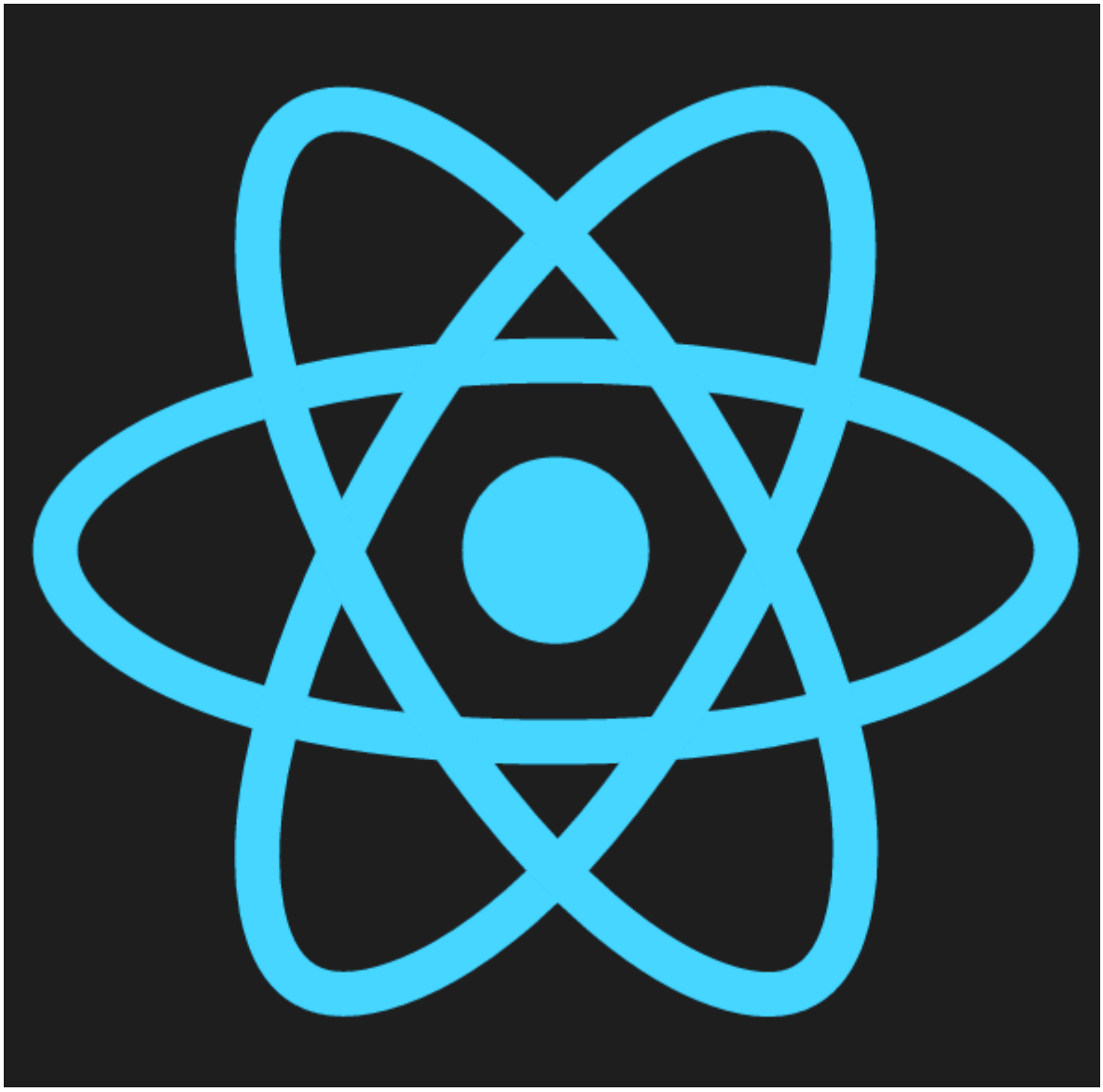
Test-Driven Development Using React.js - Student Labs



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

`https://github.com/watzthisco/tdd-react-labs-v3.x`

Version 3.2, April 2019
by Chris Minnick
Copyright 2019, WatzThis?
www.watzthis.com

Disclaimers and Copyright Statement

Disclaimer

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

Third-Party Information

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

Copyright

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior expressed permission of the owners, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the owners, at info@watzthis.com.

Help us improve our courseware

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

Credits

About the Author

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

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

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

Table of Contents

DISCLAIMERS AND COPYRIGHT STATEMENT **2**

Disclaimer 2
Third-Party Information 2
Copyright 2
Help us improve our courseware 2

CREDITS **3**

About the Author 3

TABLE OF CONTENTS **4**

SETUP INSTRUCTIONS **6**

Course Requirements 6
Classroom Setup 6
Testing the Setup 6

INTRODUCTION AND GIT REPO INFO **8**

LAB 01 - INSTALLING AND CONFIGURING WEBSTORM OR VS CODE **9**

Part 1 - Installing WebStorm 9
Part 2: Creating a New Project 9
Part 3: Configuring WebStorm for ES2015+ and JSX 10
Part 4 - Installing Visual Studio Code 10
Part 5: Creating a New Project 10

LAB 02 - GETTING STARTED WITH NODE.JS **11**

Part 1: Getting to Know Node.js 11
Part 3: Using npm 13

LAB 03 - VERSION CONTROL WITH GIT **15**

Part 1: Installing Git (Windows Only) ..... **Error! Bookmark not defined.**
Part 2: Creating a Repository 15
Part 2: Learn the Ways of Git 18

LAB 04 - INITIALIZE NPM **21**

LAB 05: USING NPM AS A BUILD TOOL **22**

LAB 06 - MANAGING EXTERNAL DEPENDENCIES **25**

Part 1: Create a "version" task 25
Part 2: Adding dependent scripts 26

LAB 07 - AUTOMATE LINTING **28**

LAB 08 - CONFIGURE A WEB SERVER **30**

Part 1: Install http-webserver 30
Part 2: Manual browser testing 30

LAB 09: GETTING STARTED WITH JASMINE **33**

LAB 10: MORE FEATURES! **38**

LAB 11 - IN-BROWSER TESTING WITH KARMA **39**

Part 1: Installing and Configuring Karma 39
Part 2: Automating Karma 41

LAB 12 - DEPLOY WITH WEBPACK **43**
PART 2: MODIFY THE KARMA CONFIG 47
PART 3: INTEGRATE BUNDLING INTO THE BUILD SCRIPT 48
LAB 13 - README UPDATE AND REFACTORING **50**
LAB 14 - BABEL **51**
LAB 15 - CONVERTING TO ES6 ..... **54**
LAB 16 - HELLO, REACT **56**
PART 1: SAY HELLO AND TEST YOUR SETUP 56
PART 2: MAKE A COMPONENT 58
PART 3: CONFIGURE ESLINT AND KARMA FOR REACT 59
LAB 17 - BREAKING UP A UI INTO COMPONENTS **62**
PART 2: BUNDLE TESTS FOR KARMA 65
LAB 18 - STATE AND PROPS **66**
LAB 19 - ADDING STYLE TO REACT COMPONENTS **70**
LAB 20 - CONTROLLING THE FORM **72**
LAB 21 - REFACTORING AND USING JSON DATA **77**
LAB 22 - LIFE CYCLE AND EVENTS **80**
LAB 23 - PROPTYPES **83**
LAB 24 - USING JEST **84**
LAB 25 - MULTIPLE COMPONENTS **87**
LAB 26 - REACT ROUTER V3.X **88**
LAB 27 - REACT ROUTER V4.X **91**
LAB 28 - REDUX THERMOMETER **94**
LAB 29 - REDUX **96**
LAB 30 - SWIMCALC **102**
THE STORY 102
GETTING STARTED 102

Setup Instructions

Course Requirements

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

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

Classroom Setup

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

☐ 1. Install node.js on each student's computer. Go to nodejs.org and click the link to download the latest version from the LTS branch.

☐ 2. Install a code editor. We use WebStorm or Visual Studio Code in the course. Visual Studio Code is available for free at *https://code.visualstudio.com/*

☐ 3. A 30-day trial version of WebStorm is available from *http://www.jetbrains.com/webstorm.*

☐ 4. Make sure Google Chrome is installed.

☐ 5. Install git on each student's computer. Git is installed by default on MacOS. Git for Windows can be downloaded from *http://git-scm.com*. Select all the default options during installation.

Testing the Setup

☐ 1. Open a command prompt.
- Use Terminal on MacOS (/Applications/Utilities/Terminal).
- 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/tdd-react-labs-v3.x
```

The lab solution files for the course will download into a new directory called **tdd-react-labs-v3.x**.

☐ 4. Enter `cd tdd-react-labs-v3.x` to switch to the new directory.

☐ 5. Enter `npm install`

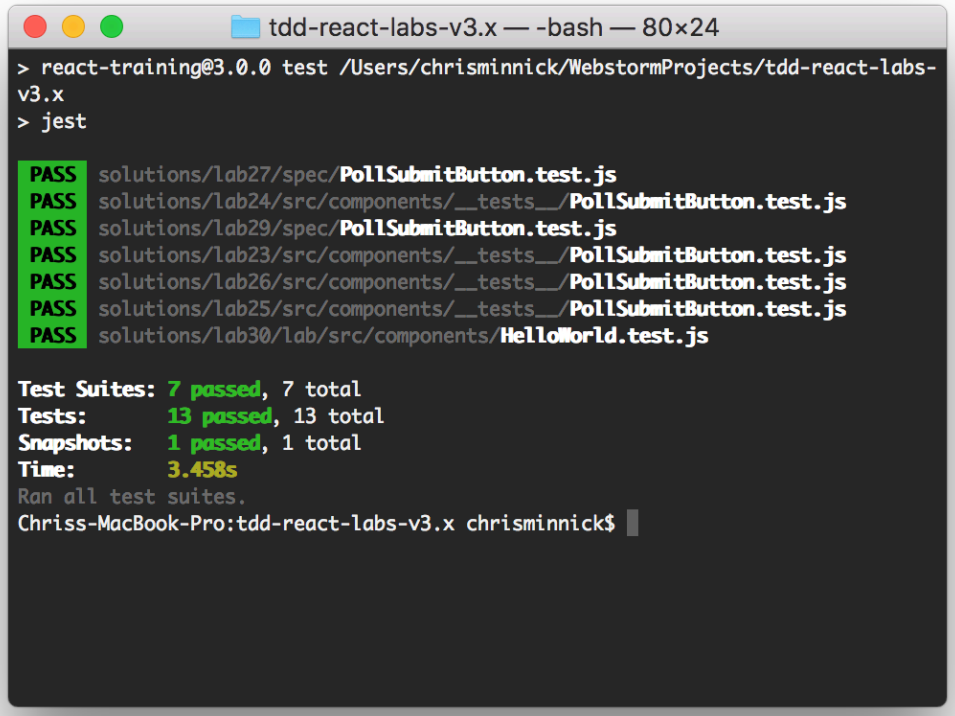
This step will take some time. If it fails, the likely problem is that your firewall is blocking ssh access to github.com and/or registry.npmjs.org.

Note: If you get an error at this point that says npm was not recognized as a command, you'll need to add npm to the system path. This may be helpful: https://stackoverflow.com/questions/27864040/fixing-npm-path-in-windows-8

☐ 6. When everything is done, enter `npm run test`

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 test`.

A series of things will happen and then a message will appear. Some of the tests may fail. As long as they run, everything is correct!



Introduction and Git Repo Info

Most of the labs in this course build on the labs that came before. So, if you don't complete a lab or can't get a certain lab to work, it's possible that you can get stuck and won't be able to move forward until the error is corrected.

To help you check your work and to make it possible to come into the class at any point, the git repository for this course contains finished versions of every lab.

The url for the course repository is:
`https://github.com/watzthisco/tdd-react-labs-v3.x`

The completed labs are in the 'solutions' directory and are numbered using the format labxx. So, if you get stuck and want to check your work on Lab 8, for example, you can look at the completed Lab 8 code inside solutions/lab08.

If you want to work on a lab without first having completed all of the labs that come before it, open the lab that comes immediately before the lab you want to work on.

For example, if you want to start with Lab 20, open the folder named lab19, which will contain the project as it should exist at the beginning of the instructions for Lab 20.

Lab 01 - Installing and Configuring WebStorm or VS Code

WebStorm and Visual Studio Code are Integrated Development Environments for JavaScript. You can use any code editor or IDE you like, but these two have good built-in support for modern frameworks like React and Node.js, as well as built-in integrations with the tools we'll be using in this course.

None of the labs in this course (other than this one) will depend on any particular IDE; so if you prefer another editor, feel free to use it and to adapt instructions in this lab to your own editor.

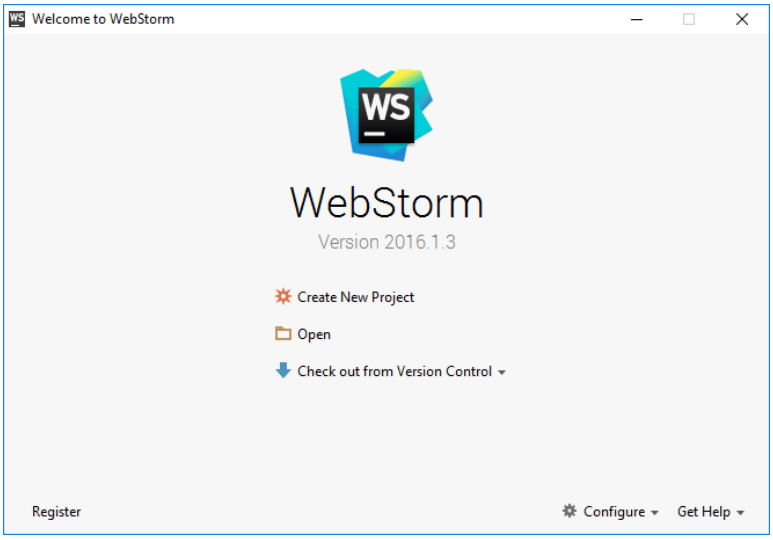
Part 1 - Installing WebStorm

WebStorm includes a 30-day trial license. If you'd like to use Visual Studio Code (which is free), skip to Part 4.

- ☐ 1. Go to http://www.jetbrains.com/webstorm/download and select your operating system.
- ☐ 2. Click the **download** link.
- ☐ 3. When the download completes, launch the installer and follow the prompts to install WebStorm.

Part 2: Creating a New Project

- ☐ 4. The first time you start WebStorm, you'll see the splash screen:



- ☐ 5. Click Create New Project.
- ☐ 6. Highlight Empty Project
- ☐ 7. Select the location to save your project and give your project a name, such as **react-training**.

☐ 8. Click **Create**.

Part 3: Configuring WebStorm for ES2015+ and JSX

☐ 9. Select **WebStorm** > **Preferences** (on MacOS) **or File > Settings** (Windows) from the top menu.
☐ 10. Click Languages & Frameworks and choose JavaScript.
☐ 11. Select React JSX (or JSX Harmony in less current versions) from the JavaScript language version dropdown.
☐ 12. Click **OK**

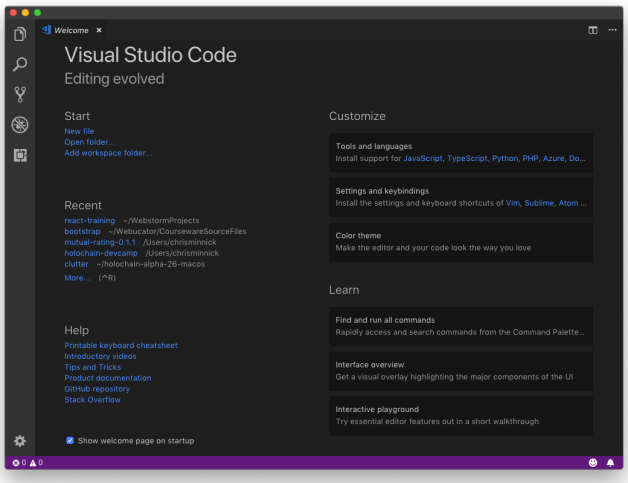
Part 4 - Installing Visual Studio Code

☐ 1. Go to https://code.visualstudio.com/ .
☐ 2. Click the **download** link.

When the download completes, launch the installer and follow the prompts to install Visual Studio Code.

Part 5: Creating a New Project

☐ 1. When you start VS Code, you'll see the splash screen:



☐ 2. Select File > Open or click the Open Folder link under Start on the default start screen.
☐ 3. Select the folder where you want to create your project (such as in My Documents), and click New Folder to create a new folder. Name the new folder **react-training**.
☐ 4. Select the new folder and click **Open**.

Lab 02 - 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 course, we will be using it for the latter purpose.

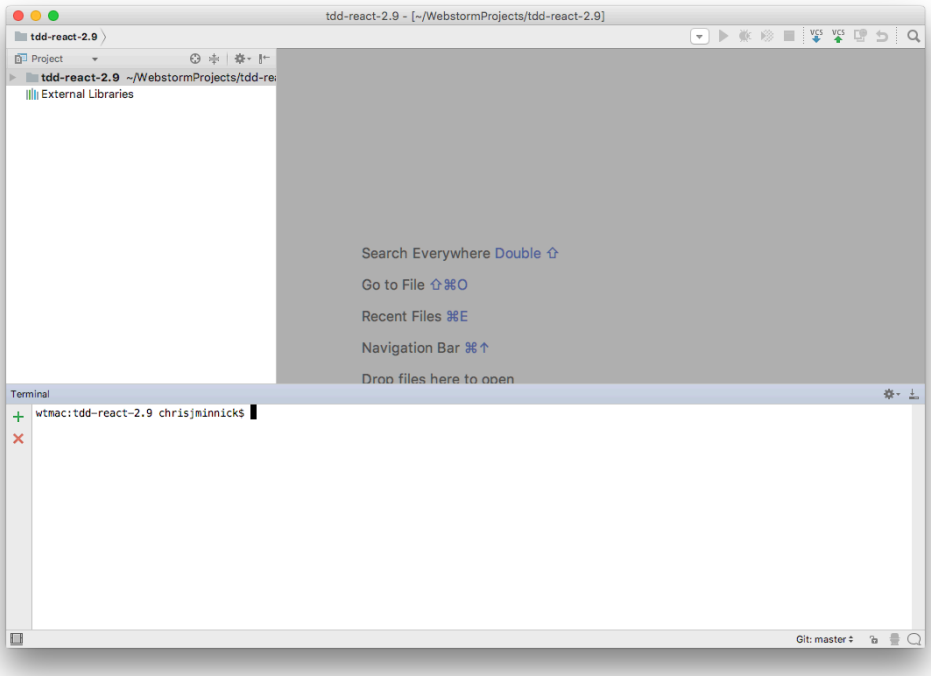
Part 1: Getting to Know Node.js

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

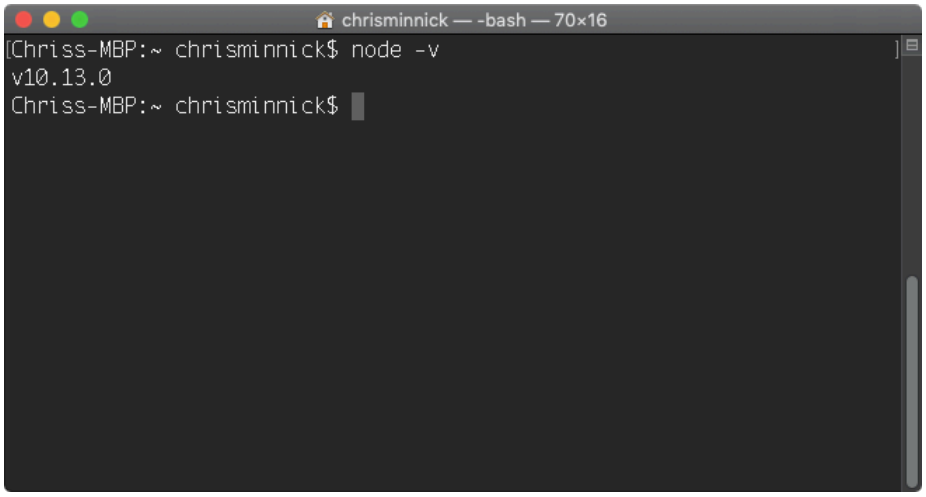
☐ 1. Open a command line application.

- MacOS: Navigate to Applications / Utilities and double click on **Terminal**.
- Windows 7, 8, or 10: Open a search box and enter **cmd** to locate the Command Prompt. Open it.
- You can also use the built-in terminal in your code editor, if it has one. The figures below show how to open the Terminal in Webstorm.





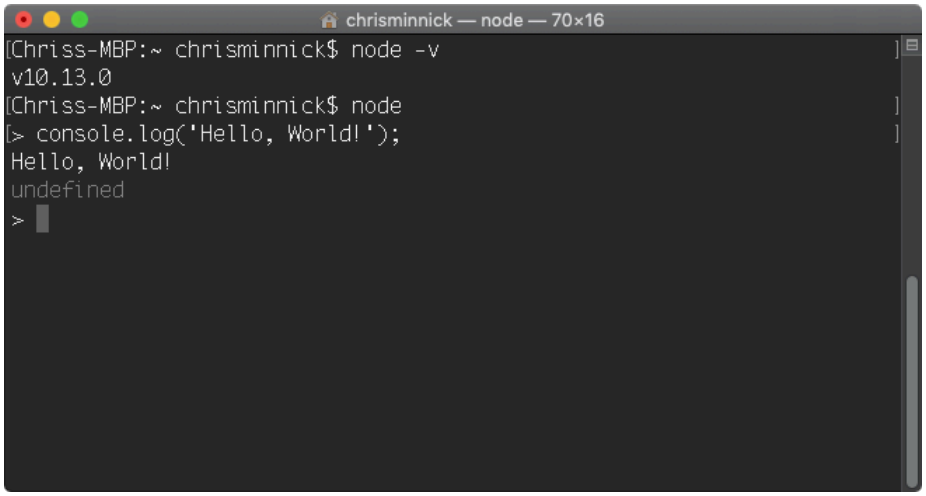
☐ 2. To check whether Node.js is property installed, enter `node -v`
You should see something like the following:



☐ 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. Create a text file using your code editor and enter the following code:

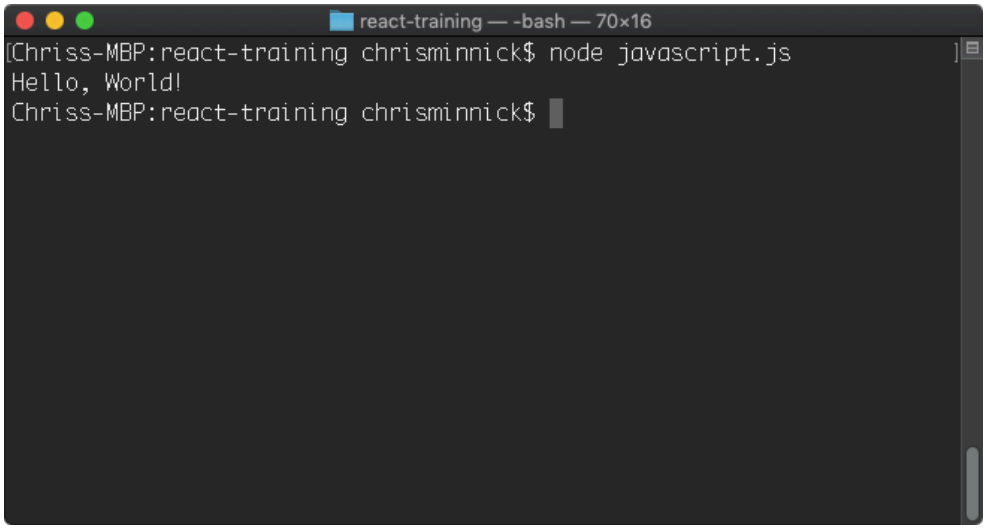
```
console.log('Hello, World!');
```

☐ 6. Save the file as **javascript.js**
☐ 7. Exit node's interactive shell by pressing **CTRL-C** twice.
☐ 8. In your terminal, navigate to the directory where you saved javascript.js.

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).

☐ 9. 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.

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

This command will install the latest version of npm.

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

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

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

☐ 5. Enter `npm help ls`

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

☐ 6. If the help file displayed in the console window, type `q` to exit the help system.
☐ 7. Enter `npm update` or `npm update -g`

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

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

Lab 03 - Version Control with GIT

Git is a very popular version control system. There are visual tools for working with Git, including ones that are built into WebStorm and Visual Studio Code. 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 install git and then learn some basic commands.

Part 1: Creating a Repository

☐ 1. Using the command line (Terminal on MacOS or git bash shell on Windows), navigate to the WebStorm project directory that you created in Lab 1.

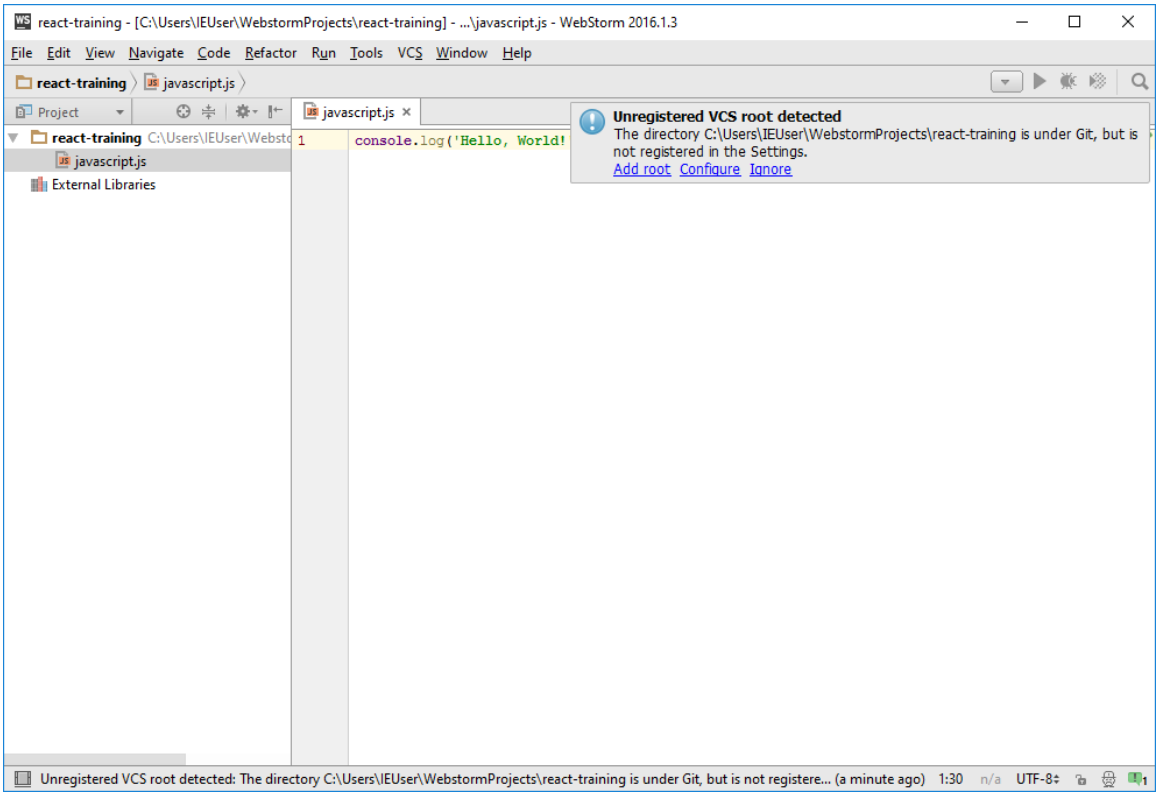
☐ 2. Enter `cd` followed by your project name to access the project directory if you're not already inside it.

☐ 3. Enter `git init`

This will initialize a git repository for the current project.

☐ 4. Enter `ls -la` to view all the files, including hidden files, in the current directory. You will notice that there's a new hidden folder named .git. This folder is where Git will keep all its information about your project.

☐ 5. If you have WebStorm open, it may ask you whether you want to configure git for your project. Click **add root**.



- ☐ 6. In your code editor, create a new file named **README.md** and save it into your project. If WebStorm asks you to associate *.md with a file type, just choose Text.

README.md will hold information about your project that's designed to be read by future users of the project.

- ☐ 7. If WebStorm asks you if you want to add files to Git, say Yes.
- ☐ 8. If WebStorm asks you if you want to install a plugin to support *.md files, choose Install plugin. WebStorm's plugins are generally helpful and it doesn't do any harm to install them, even if you uninstall them later. Note, however, that you may need to restart WebStorm after installing a plugin for the new plugin to work.
- ☐ 9. Inside README.md, enter this basic structure, which you will fill in the details of later:

```
# My Project

This is my project.

## Installation

## Usage
```

```
## Credits

## License
```

- ☐ 10. Save README.md
- ☐ 11. Create a file named **.gitignore**

.gitignore tells Git what files shouldn't be checked into the repository. In general, you never want to check generated binaries or files that are specific to a workstation or developer into Git.

- ☐ 12. Add the following to .gitignore

```
# MacOS X
.DS_Store

# WebStorm
.idea

# npm
node_modules
```

The lines that start with `#` are comments describing the rule that follows them. The first line just ignores a file that MacOS puts in every directory. The second ignores your WebStorm configuration files. The third ignores dependencies that you'll be installing into your project.

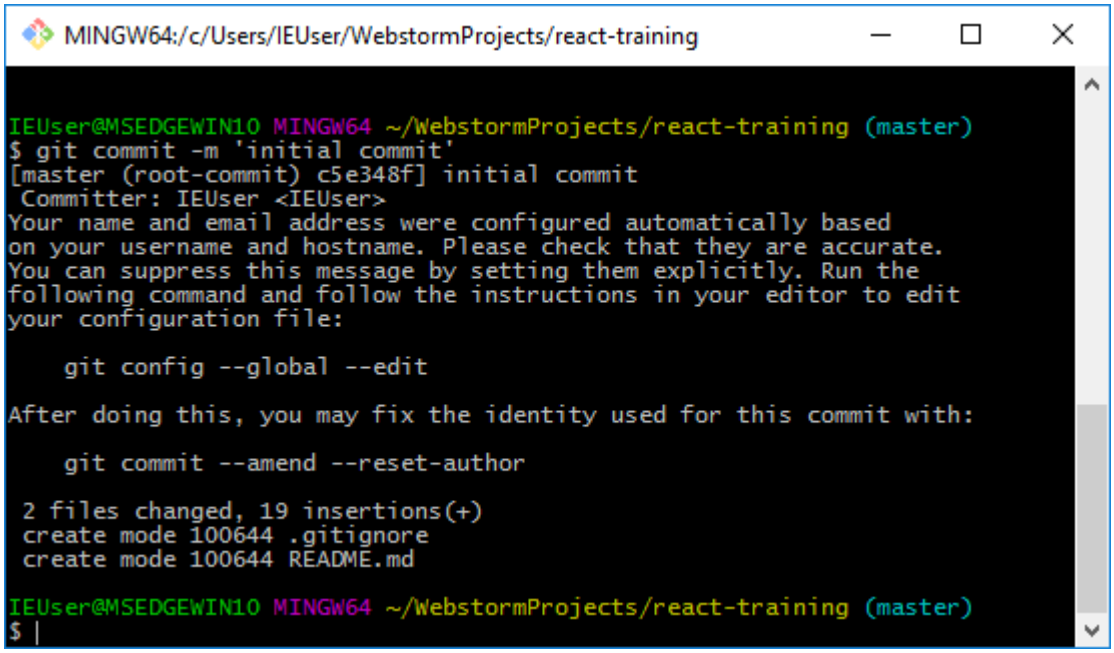
- ☐ 13. In your command line, enter `git add .`

  This will stage any new files so that they're ready to be committed to version control.

- ☐ 14. Enter `git status` to verify that your .gitignore and README.md are staged.
- ☐ 15. Enter `git commit -m 'initial commit'`

  If this is your first time using git, it may ask you to configure your email address and name. Use the commands that it provides to do so. Once you've done that, re-run the commit.

  This will commit your new file and your project into your repository.



Part 2: Learn the Ways of Git

In this part, you will learn the most important commands for working with git.
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 master branch is the default branch of your local repository. In general, you should always aim to keep the master clean and working and use branches for any new code. We'll get back to that in a moment.

☐ 1. Enter `git log`. You will see the history of previous commits. Note that each commit has a unique identifier.

☐ 2. Your most recent commit is called `HEAD`
You can get the diff of your most recent commit like this:

```
git diff HEAD
```

If you run this now, you should see nothing, since you haven't changed anything since your last commit.

☐ 3. Make the following changes to README.md

```
# React Project Template

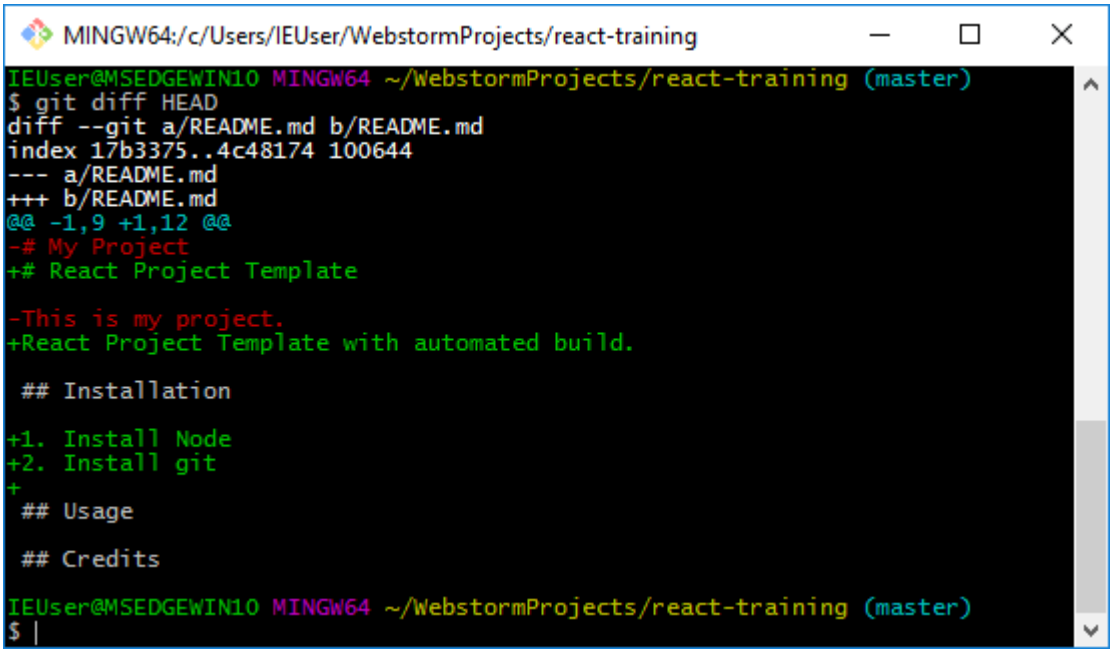
React Project Template with automated build.
```

```
## Installation

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

- ☐ 4. Enter `git diff HEAD`

☐ 5. You will see a list of differences between the current state of your files and the last commit. If you get stuck at the : prompt, press 'q' to exit to the command line.



- ☐ 6. Create a new directory named `src` (using the command line or in WebStorm) and create a file inside of it called **app.js**. Use `git add .` to stage these.
- ☐ 7. Use `git diff --staged` to see what you have staged.
- ☐ 8. Type the following to unstage app.js:

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

- ☐ 9. Use `git diff --staged` and `git status` to see that you've unstaged the file.
- ☐ 10. Stage the file again.
- ☐ 11. 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
```

☐ 12. Commit your changes.

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

☐ 13. 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
```

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

☐ 15. git checkout my_first_branch

☐ 16. In the new branch, add this code to app.js

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

☐ 17. git add and git commit your changes. Remember to use a descriptive message.

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

```
git checkout master
```

☐ 19. Merge your changes from my_first_branch back into master:

```
git merge my_first_branch
```

☐ 20. Delete your branch:

```
git branch -d my_first_branch
```

Lab 04 - Initialize npm

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

☐ 1. In your console, enter:

```
npm init
```

You will be asked some questions to configure npm for your project. The default values will be shown in parentheses after the question. Press Enter or Return to accept each of these default values. Once you have gone through all the questions, you will see that a new file, package.json, has been created in the root of your project.

Note: When using git bash shell on Windows, the configuration script may hang after the last question. When this happens, press Ctrl-C. Everything has run correction and the package.json file has been created, but it just doesn't exit correctly.

☐ 2. Open **package.json** in your code editor. Notice that the project description has 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.

☐ 3. Enter `npm install` in the console.

There's nothing for npm to do at this point, since you don't have any modules installed or instructions inside **package.json**, however a new file named **package-lock.json** will be created in your project. Google "What does package-lock.json do?" to find out more about this file.

☐ 4. Add this instruction to the README file's Installation section:

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

☐ 5. Commit everything to git:

```
git add .
git commit -m "Initialized npm"
```

Lab 05: Using npm as a Build Tool

In this lab, you will learn how to create npm scripts and run them. Npm scripts can be used to automate many of the tasks involved in front-end development, such as testing, building, and deployment.

☐ 1. Open package.json in your code editor.

Npm's default package.json file contains a scripts object. If you didn't specify a test script when you ran npm init, a default test method was created for you inside the scripts object.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit
1"
},
```

☐ 2. Open your command prompt and enter `npm run`.

Npm's run command can be used to run methods inside the scripts object. If you use npm run without any arguments, it will return a list of the available scripts. In this case, you should get the following:

```
Lifecycle scripts included in lab-files:
  test
    echo "Error: no test specified" && exit 1
```

☐ 3. Enter `npm run test`

The test script will run and output the message saying that no test is specified, and then it will exit with an error.

We'll specify a test in a future lab. For now, we're going to create a simple build script, which will run the test script and then exit with a message.

☐ 4. Add another property to the scripts object, named build. For now, the build script will just print out a message.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit
1",
  "build": "echo \"BUILD OK\" && exit 0"
},
```

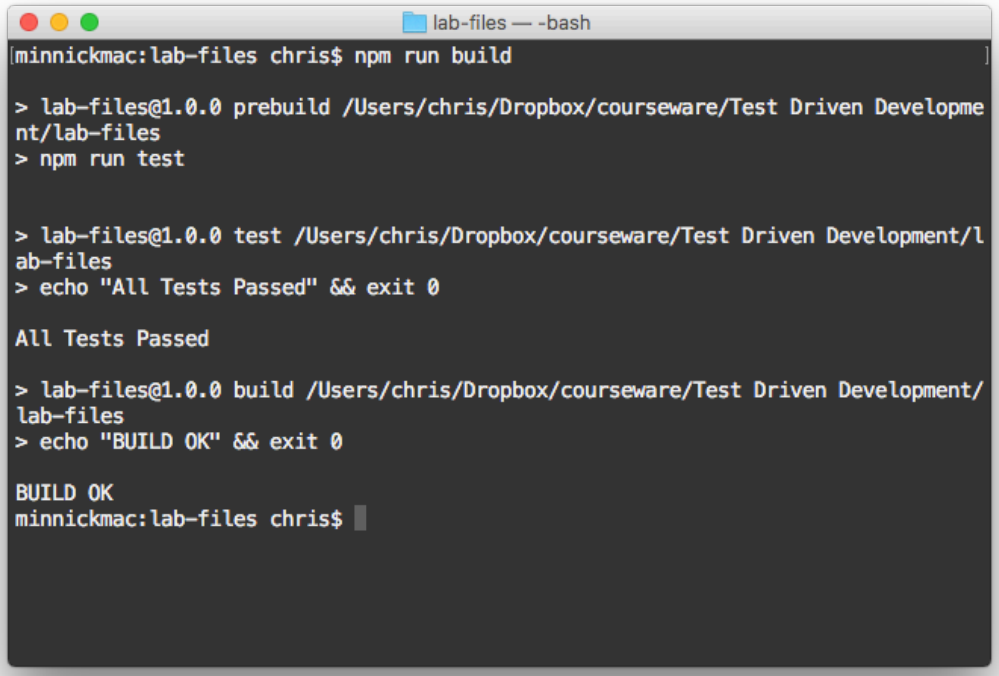
☐ 5. Create a prebuild script. Each script that you create automatically has a pre- and post- script that you can override with your own code. The names of these scripts are the name of your method, with either pre or post prepended. We'll use the prebuild script to run the test script, and we'll change the test script to output a success message for now. **Make sure to change the exit status to 0 to indicate success.**

```
"scripts": {
  "test": "echo \"All tests passed\" && exit 0",
  "build": "echo \"BUILD OK\" && exit 0",

  "prebuild": "npm run test"
},
```

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

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



Next, we'll create a npm config file to set the log level to suppress all the extra lines of output.

☐ 8. In your terminal, enter **vim .npmrc** to create a file named .npmrc and open it for editing.

☐ 9. Press **i** to enter insert mode and enter the following text

```
loglevel=silent
```

☐ 10. Save the file by pressing the **ESC** key, followed by **:wq**

☐ 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. npm run build

☐ 12. In the command line, enter `npm run build` to confirm that it works.

☐ 13. 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.

Note: If you just type **`git commit`** here (without the -m), you will be taken to the vim editor to enter the commit comment. Press ESC, followed by :wq to save and exit.

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

Lab 06 - Managing External Dependencies

In this lab, you will create a script to verify that the correct version of node is installed and to fail with an error if it's not.

Part 1: Create a "version" task

☐ 1. Create a new method in the scripts object in package.json, called **version**.

☐ 2. Inside the version task, tell it to run a node script named version-check.js:

```
"version":"node version-check.js"
```

☐ 3. Add a new property to `package.json` that specifies the node version we want. This course has been tested on Node version 4.4.2 and higher, so we'll set 4.4.2 as the minimum required version. You can also specify a newer version here, of course.

```
...
"description":"Your existing description here",
"engines": {
    "node": "8.11.2"
},
...
```

☐ 4. Your package.json file should look something like this now:

```
{
  "name": "my-react-project",
  "version": "1.0.0",
  "description": "This is my project",
  "main": "javascript.js",
  "engines": {
    "node": "8.11.2"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit
0",
    "build": "echo \"BUILD OK\" && exit 0",
    "prebuild": "npm run test",
    "version": "node version-check.js"
  },
  "author": "",
```

```
  "license": "ISC"
}
```

☐ 5. In order to compare version numbers, which are in the standard `semver` format (v1.2.3), we will need a node package called **semver**. Install it locally. On the command line, enter:

```
npm install --save-dev semver
```

☐ 6. Now you can create your version-check script. In your code editor, create a new file named **version-check.js** in the root directory of your project. It should contain the following script:

```
console.log('Checking node version: ');
const semver          = require('semver');
const packageJson     = require('./package.json');
const expectedVersion = packageJson.engines.node;
const actualVersion   = process.version;

if (semver.gt(expectedVersion,actualVersion)){
    console.log('Incorrect node version. Expected ' +
        expectedVersion + '. Actual: ' +
actualVersion);
    process.exit(1);
} else {
    console.log('Node version ok: ' + actualVersion);
}
```

☐ 7. In the command line, enter **`npm run version`** to test it out.

☐ 8. If there are no errors, commit to git!

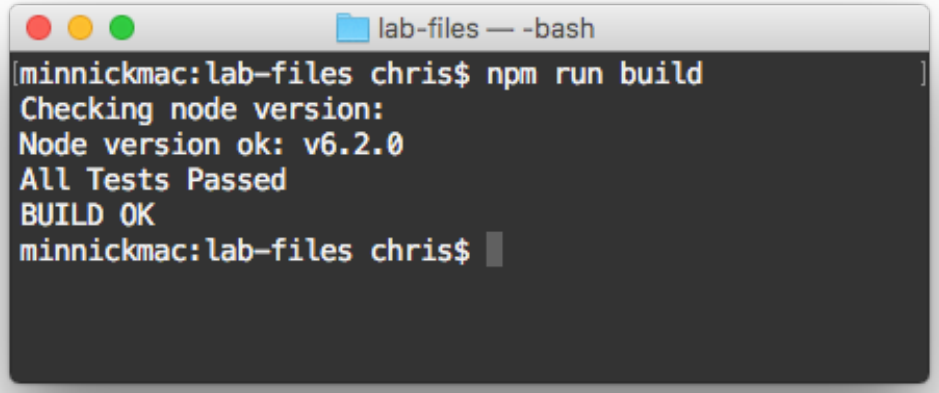
Part 2: Adding dependent scripts

You can specify multiple tasks to run inside of an npm script by using the && operator.

☐ 1. Modify the prebuild task to add version as a dependent task that must run prior to starting the default task.

```
"prebuild": "npm run version && npm run test"
```

☐ 2. Enter `npm run build` into the console to test it out. You should get the following result:



☐ 3. Change the value of the node property in the engines object to a higher version than the one you have installed to verify that it fails.

☐ 4. Change the value of the node property back to your desired minimum node version.

Lab 07 - Automate Linting

Linting is a way to perform static code analysis on your files. Static code analysis will look at the syntax (and the style, in some cases) of your JavaScript and alert you if there are problems. Just as with the version checking task in the last lab, we want our automated build to fail and give us errors if there are problems found.

In this lab, you will install ESLint, use it to check a JavaScript file, and then build it into your automated build.

- ☐ 1. If your terminal program isn't already open, open it and go to your project directory folder.
- ☐ 2. Type **npm install eslint --save-dev** to install ESLint.
- ☐ 3. Run **`./node_modules/.bin/eslint --init`** to set up the configuration file (use .\node_modules\.bin\eslint --init if you're working in cmd.exe).
- ☐ 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.

```
? How would you like to use ESLint? To check syntax, find problems, and enforce code style
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? React
? Where does your code run? Browser, Node
? How would you like to define a style for your project? Answer questions about your style
? What format do you want your config file to be in? JavaScript
? What style of indentation do you use? Spaces
? What quotes do you use for strings? Single
? What line endings do you use? Windows
? Do you require semicolons? Yes
The config that you've selected requires the following dependencies:

eslint-plugin-react@latest
```

- ☐ 6. run `npm install --save-dev eslint-plugin-react@latest` to install the eslint React plugin.
- ☐ 7. Make a file named **.eslintignore** in the root of your project.
- ☐ 8. Add the following text to **.eslintignore**

```
node_modules/*
```

- ☐ 9. Create a new script in package.json called **`lint`**, as follows:

```
"lint": "eslint . --ext .js",
```

I recommend putting it before the "test" script.

- ☐ 10. Run **npm run lint**.
- ☐ 11. 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.
- ☐ 12. If you're on Mac or Linux, you may need to change the line break style to **`"Unix"`**. 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,
        'es6': true,
        'node': true
    },
    'extends': 'eslint:recommended',
    'globals': {
        'Atomics': 'readonly',
        'SharedArrayBuffer': 'readonly'
    },
    'parserOptions': {
        'ecmaFeatures': {
            'jsx': true
        },
        'ecmaVersion': 2018,
        'sourceType': 'module'
    },
    'plugins': [
        'react'
    ],
    'rules': {
        'indent': [
            'error',
            4
        ],
        'linebreak-style': [
            'error',
            'windows'
        ],
        'quotes': [
            'error',
            'single'
        ],
        'semi': [
            'error',
            'always'
        ],
        'no-console': [
            'warn'
        ]
    }
};
```

- ☐ 13. Make the lint script run prior to the test script in the prebuild script.

Lab 08 - Configure a Web Server

In this lab, you will set up a local web server so that you can do manual testing of your application in web browsers. There are numerous web servers you can run on your local machine, and you can even build your own with just a few lines of code in Node.js. We're going to use the `http-server` package.

Part 1: Install http-webserver

☐ 1. Install the http-server package.

```
npm install --save-dev http-server@0.9.0
```

☐ 2. Create a new script in package.json named start:

```
"start":"http-server src",
```

☐ 3. Run **`npm start`** in your command line.

The web server will start.

☐ 4. Create a file named `index.html` and put it in your `src` directory.

☐ 5. Enter the following content into `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Page</title>
</head>
<body>
<h1>Welcome</h1>
<script src="app.js"></script>
</body>
</html>
```

☐ 6. Open a web browser and navigate to *localhost:8080 (or one of the addresses that appeared in the console window when you started http-server)*. You should see the message "Welcome" in the browser window.

☐ 7. Stop the web server by pressing **Control** - **C**.

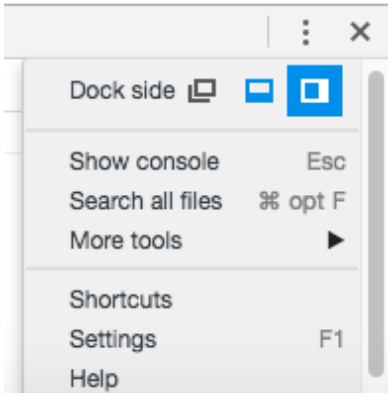
☐ 8. Check in your code.

Part 2: Manual browser testing

Different web browsers have different levels of support for HTML, CSS, and JavaScript features. Because of this, it's essential for front-end developers to test in multiple browsers. Manual browser testing can be tedious and difficult, but each browser has developer tools to make it somewhat easier.

In this part, you will get acquainted with Google Chrome's Web Developer tools for inspecting and debugging your front-end code.

☐ 1. Start your development web server and open your development site in Chrome.
☐ 2. Press **Command**-**Option**-**I** (on MacOS) or **Ctrl**-**Shift**-**I** (Windows) to open the Developer Tools.
☐ 3. Dock the Developer Tools to the right side by clicking the **Customize** button on the right side of the Developer Tools toolbar and selecting **Dock Right**.



☐ 4. Click **Elements**. The current HTML and CSS of your document (as it exists in the DOM) will appear. If you have the livereload option set for the web server, try changing your index.html document to see the change here a moment after you save.
☐ 5. Click the `h1` element. In the styles pane on the right, add `color:blue` to the `element.style` object.
☐ 6. Right-click the `h1` element and select `hide element`.
☐ 7. Click the **Console** tab to open the JavaScript console.

You can also open the JavaScript console at any time by pressing **Ctrl**-**Shift**-**J** (Windows) or **Command**-**Option**-**J** (Mac).

☐ 8. The text **Hello, World!** which was created by `app.js` should be in the console window.
☐ 9. Enter the following into the console, followed by Return (or Enter):

```
document.body.innerHTML = '<h1>Here is some new text!</h1>';
```

The content of the document's `body` element will change to the HTML you just entered.

☐ 10. Click the **Sources** tab.

The JavaScript debugger will open.

☐ 11. Click on **app.js** and click the line number next to the console.log statement to set a breakpoint.
☐ 12. Refresh the page.

Execution of the script will halt prior to the statement running. Clearly, this is a very basic example that doesn't show us much about how the debugger works. But, examine the different options available and hover over the different buttons to find out what they do.

- ☐ 13. Visit *https://developers.google.com/web/tools/chrome-devtools/debug/breakpoints/?hl=en* to learn more about the Sources Panel and working with breakpoints.

Lab 09: Getting Started with Jasmine

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

☐ 1. Enter the following command to install jasmine:

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

☐ 2. Initialize jasmine

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

A new folder, named **spec**, will be created. Inside of spec will be another new directory, named **support**, which contains the jasmine configuration file, jasmine.json.

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

☐ 4. create a file named **sayHelloSpec.js** in the **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.

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

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

});
```

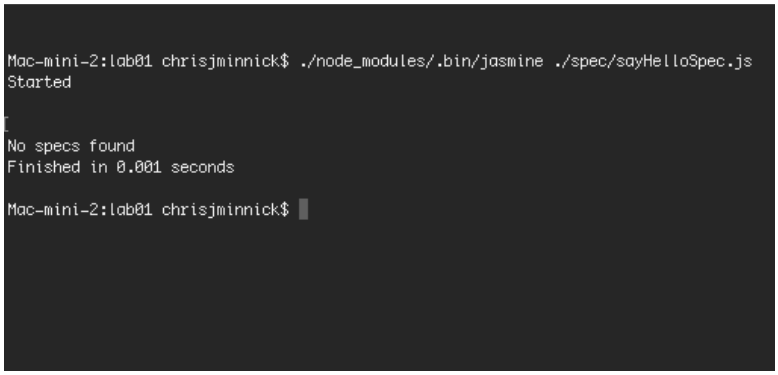
☐ 6. Save your spec and let's test it out!

☐ 7. In your command line, enter:

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

Note: If you get an npm error, enter `npm rebuild` and then try again.

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



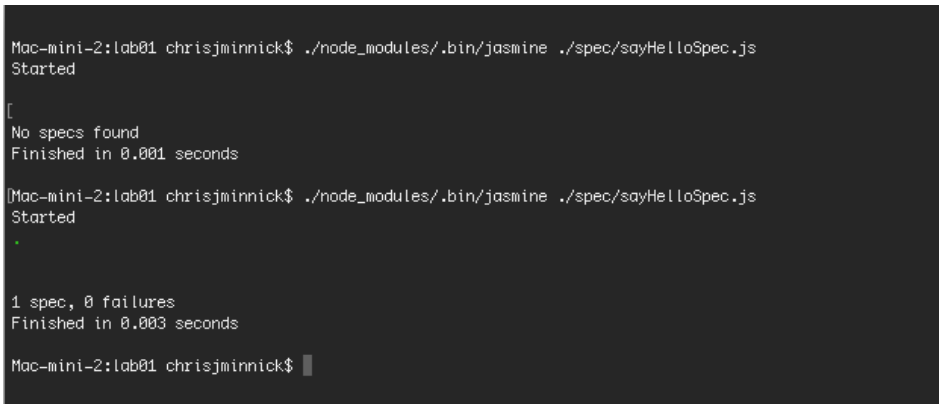
☐ 9. Inside your first test suite in **sayHelloSpec.js**, create a new spec:

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


    });
)};
```

☐ 10. Run your test again:

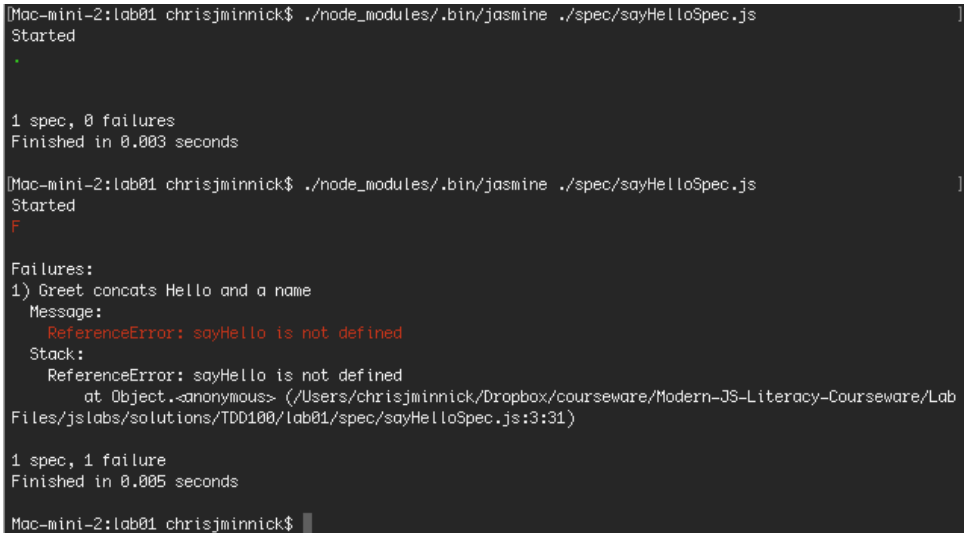
```
./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
```



☐ 11. Success! But…we're not testing anything yet. Let's create an **expectation** (aka assertion):

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

☐ 12. 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.

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

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

☐ 14. 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;

};
```

☐ 15. Save the file 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

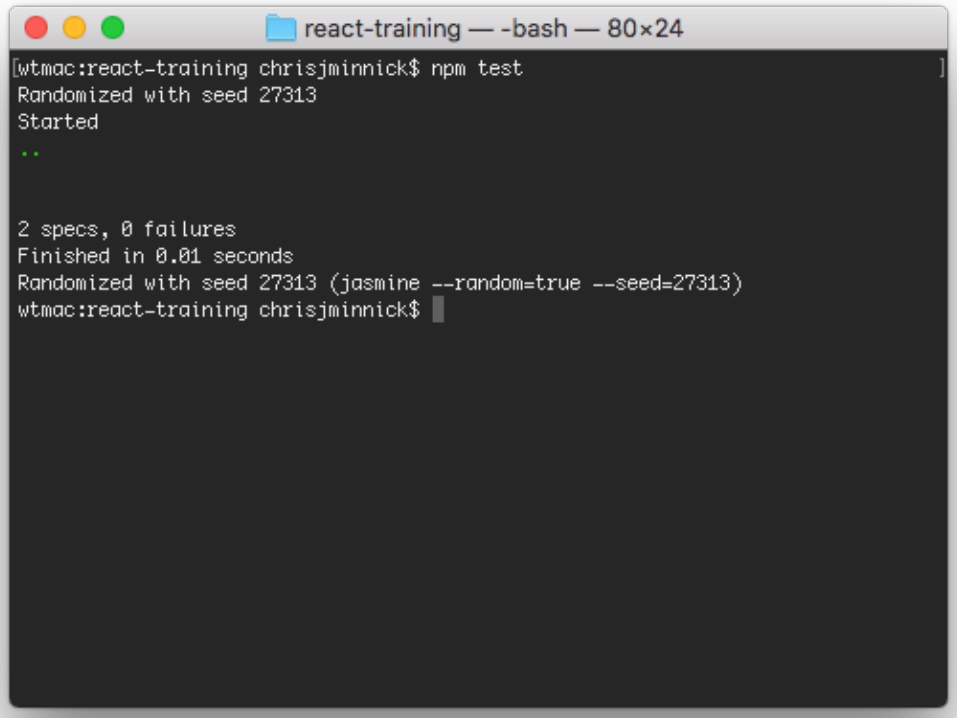
Mac-mini-2:lab01 chrisjminnick$
```

☐ 16. 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.

☐ 17. 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?

☐ 18. Create a new spec describing what your desired result should be when there's no name argument passed to `greet()`.

☐ 19. Write code to make the test pass.



☐ 20. Refactor. Can you make the code you just wrote better? Can you improve this spec? If so, do it.

☐ 21. 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.

☐ 22. Modify your test script in package.json:

```
"test":"jasmine"
```

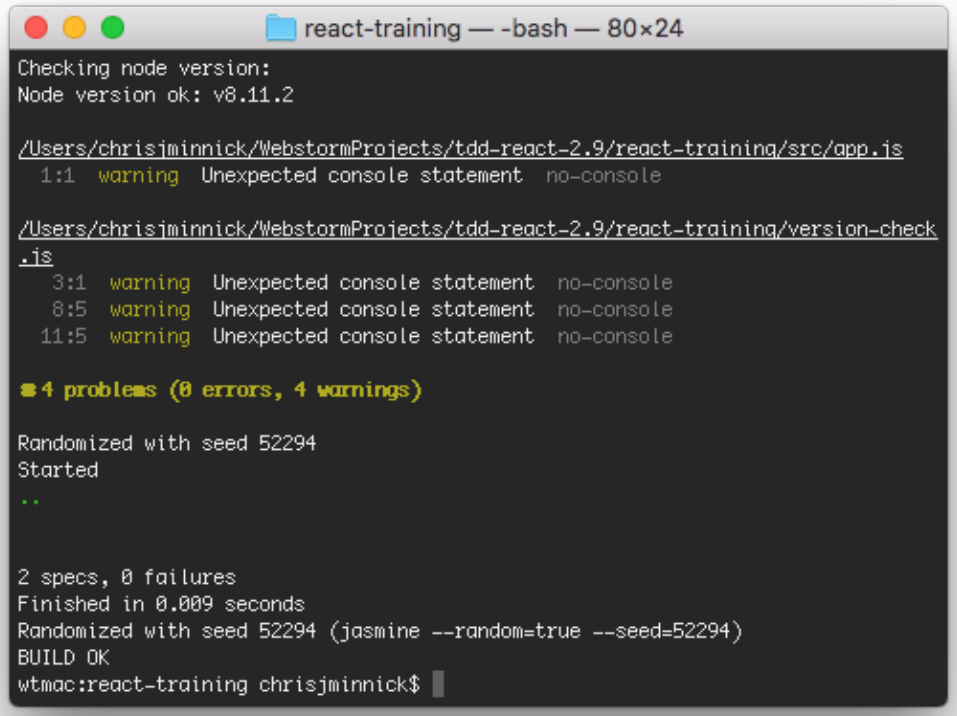
☐ 23. Add jasmine as an environment in the ESLint config file (.eslintrc).

```
"env": {
    "browser": true,
    "commonjs": true,
    "es6": true,
    "node": true,
    "jasmine": true
},
```

☐ 24. Run **`npm run test`**

Note: npm includes shorthand methods for running certain commonly used tasks, including test and start. When you run the npm test task, you can just type `npm test`, rather than `npm run test`.

☐ 25. Run the `build` script to confirm that everything works.



Lab 10: More Features!

In this lab, you'll build on the Hello, World! script that you created in lab 9. Your client has made some vague requests for new features, and it's your job to figure out how to implement them using TDD.

☐ 1. Using TDD, implement the following new features for the Hello World! script.
   - It gives an appropriate hello for the time of day
     - Good morning!
     - Good afternoon!
     - Good evening!
   - It displays a login message if no name is provided
   - It speaks German to Germans
   - It refuses to say hello after the fourth time the function is called

Lab 11 - In-browser Testing with Karma

In this lab, you will install Karma and integrate it with Jasmine to be able to automatically run your tests in multiple browsers.

Part 1: Installing and Configuring Karma

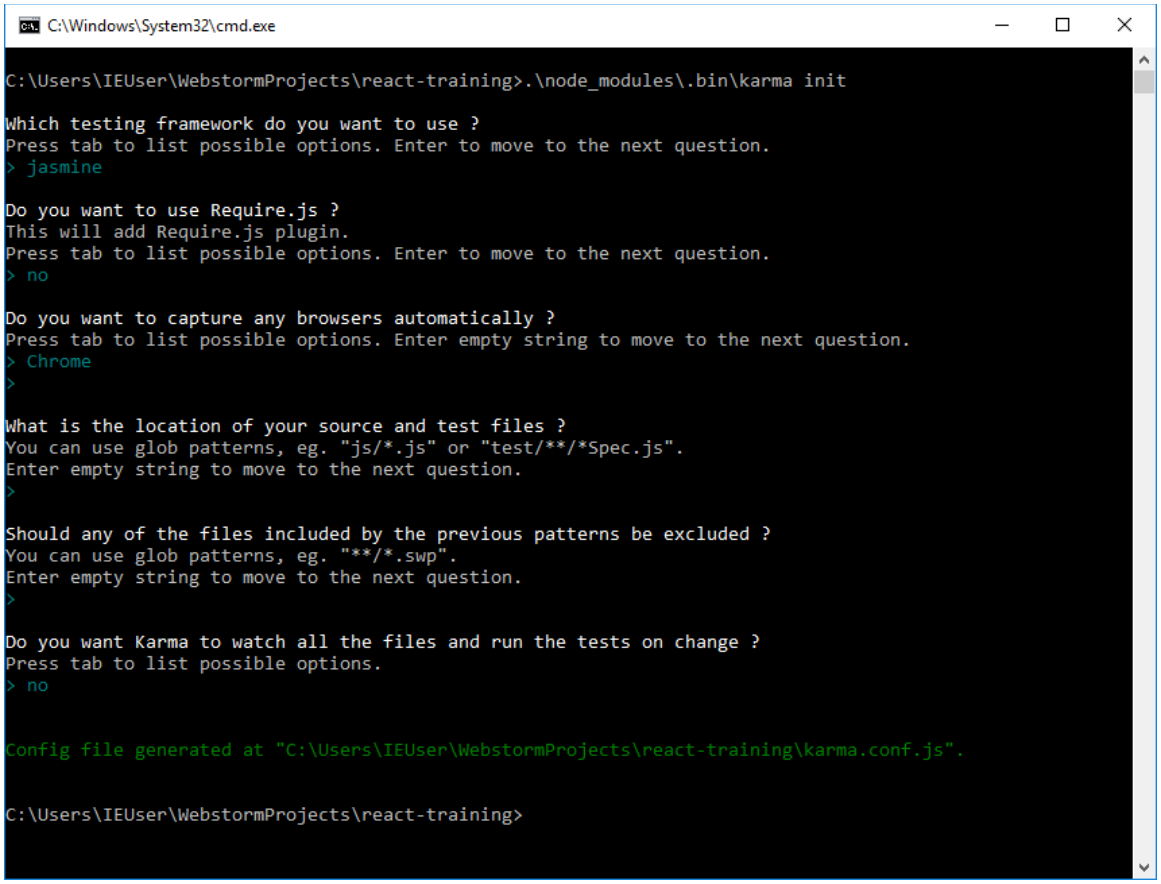
☐ 1. Install karma

```
npm install --save-dev karma
```

☐ 2. Initialize `karma`

```
./node_modules/.bin/karma init
```

You'll be walked through a series of configuration questions. Answer them as follows (switch between choices using the arrow keys):



☐ 3. Run `./node_modules/.bin/karma start`

- ☐ 4. A browser should open and go to localhost:9876 and display that you're connected to Karma.
- ☐ 5. **Leave the karma server running in that Terminal window and open a new terminal window**. Change directories to get to your project's root directory.
- ☐ 6. Run `./node_modules/.bin/karma run` to run tests

You'll get a message that there are no tests.

- ☐ 7. Open **karma.conf.js** and enter the path your tests into the files option.

```
// list of files / patterns to load in the browser
files: [
    'spec/**/*Spec.js'
],
```

- ☐ 8. Install karma-commonjs (`npm install --save-dev karma-commonjs`). This will enable Karma to parse test suites that include commonjs modules.

After installing karma-commonjs, you will need to make sure the module is loaded as a framework and a preprocessor and that both the test directory and the src directory are listed in the files array.

- ☐ 9. Include commonjs in the framework option in **karma.conf.js**

```
frameworks: ['jasmine', 'commonjs'],
```

- ☐ 10. Add your `src` directory to the files option

```
files: [
    'spec/**/*Spec.js',
    'src/**/*.js'
],
```

- ☐ 11. Tell Karma to preprocess the JavaScript files in your `js` and `spec` directories using commonjs before running tests.

```
preprocessors: {
    'src/**/*.js': ['commonjs'],
    'spec/**/*.js': ['commonjs']
},
```

- ☐ 12. Change the reporter to 'dots'

```
reporters: ['dots'],
```

☐ 13. Install karma-jasmine

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

☐ 14. Stop the Karma server (using CTRL-C) and restart it to reload the config file.

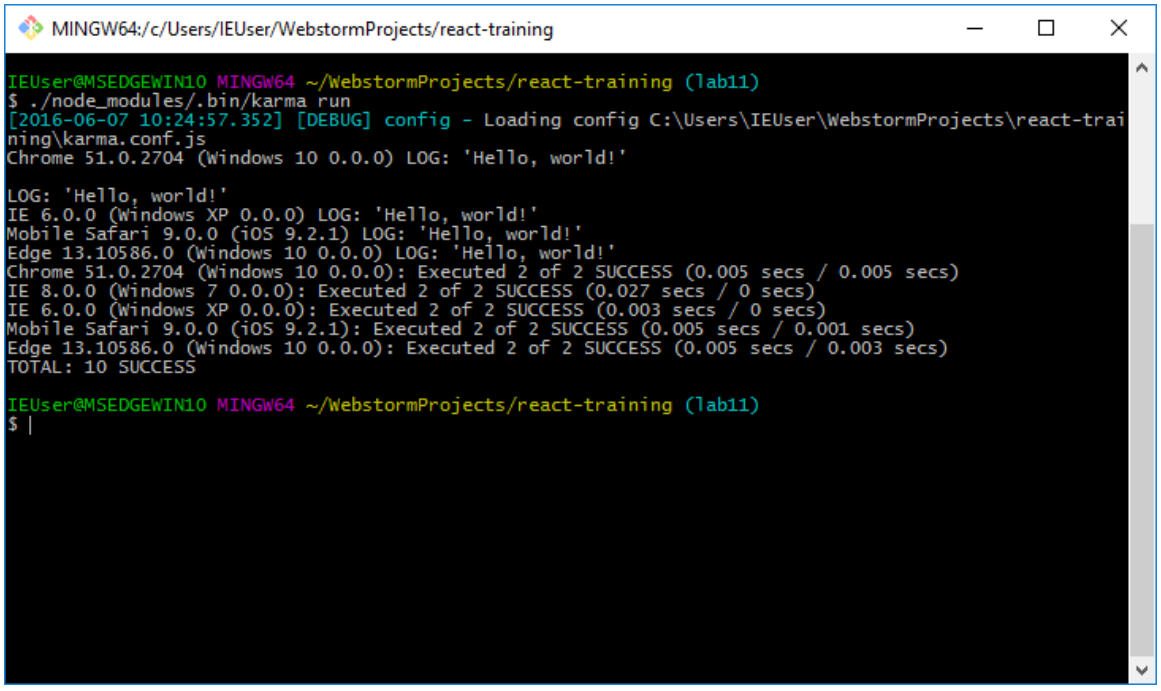
☐ 15. Enter `./node_modules/.bin/karma run` in a different console window.

Your tests should pass.

☐ 16. Open some other browsers (such as Internet Explorer, Firefox, Safari, and anything else you might have on your computer) and navigate to http://localhost:9876 in each.

You should see a Karma connected message in each browser.

☐ 17. Re-run your tests to test your code in each connected browser.



Part 2: Automating Karma

Next, we'll integrate Karma into the build script so that the automated browser tests run when we run our default task.

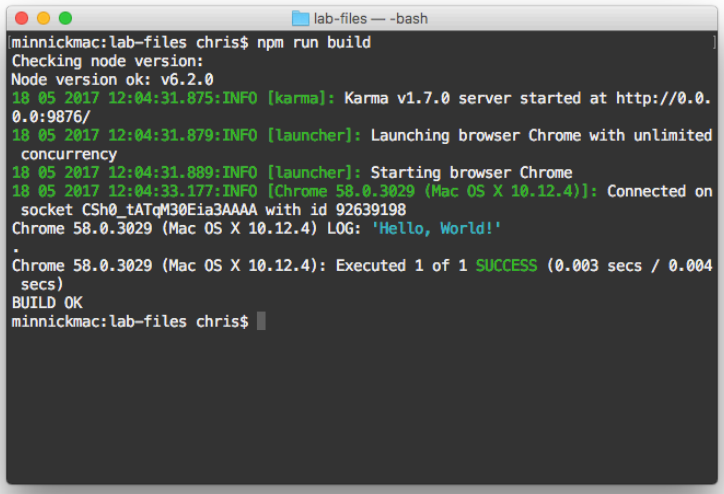
☐ 1. Add karma.conf.js to your **.eslintignore** file.

☐ 2. Change your 'test' script in **package.json** to run the tests with karma.

```
"test": "karma start --singleRun",
```

☐ 3. Stop the karma server if it's running

☐ 4. Run npm run build.



Lab 12 - Deploy with Webpack

Now that we have automated linting and testing, the next step is to automate the building of what will go on the server. You never want to serve your source files directly. You need to process them, minify them, and bundle them first. You can automate this process with webpack.

First, we'll create a homepage for our project and do some refactoring.

- ☐ 1. Create a directory inside **src** called **scripts** and move **app.js** and **sayHello.js** into it.
- ☐ 2. Update the link to **app.js** in **index.html**.
- ☐ 3. Give the `<h1>` element a unique id attribute.

Here's what the **index.html** file should look like now:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>My Page</title>
</head>
<body>
    <h1 id="welcome-message">Welcome</h1>
    <script src="scripts/app.js"></script>
</body>
</html>
```

- ☐ 4. Update the path to **sayHello.js** in **sayHelloSpec.js**
- ☐ 5. Run **`npm run build`** to make sure that this move didn't break anything, and fix anything that it did break.
- ☐ 6. Open **app.js** so we can make it include and use the `sayHello` module.
- ☐ 7. In **app.js**, require **sayHello.js**
- ☐ 8. In **app.js**, write code that uses the greeting function to do something, like this:

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

window.addEventListener('load', function(){
      document.getElementById('welcome-message')
      .innerHTML = sayHello.greet('Your name');
});
```

To be able to use `require` in a web browser, we'll need to preprocess the file using webpack and generate a distribution directory.

☐ 9. Change the webserver `src` in your **package.json** script to **dist**. This will be the directory we'll create using webpack.

```
"start": "http-server dist",
```

☐ 10. Install webpack and the webpack command line interface

```
npm install --save-dev webpack webpack-cli
```

☐ 11. Create a file named **webpack.config.js** in the root of your project.

☐ 12. Require the Node **path** module, so that we can set the default webpack output path.

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

☐ 13. Inside **webpack.config.js**, specify the entry and output and set the mode property to `production` (the other possible value for mode is `development`):

```
module.exports = {
    mode   : 'development',
    entry  : './src/scripts/app.js',
    output : {
        path: path.resolve(__dirname, 'dist'),
        filename : 'scripts/app.js'
    }
};
```

☐ 14. Tell webpack to generate a SourceMap. A sourcemap maps combined or minified files back to their unbuilt state. This makes debugging of the built site possible.

```
module.exports = {
    devtool: 'source-map',
    mode   : 'development',
    entry  : './src/scripts/app.js',
    output : {
        path: path.resolve(__dirname, 'dist'),
```

```
            filename : 'scripts/app.js'
        }
    };
```

☐ 15. Create a new task called `bundle` inside **package.json**.

Here's what it should look like:

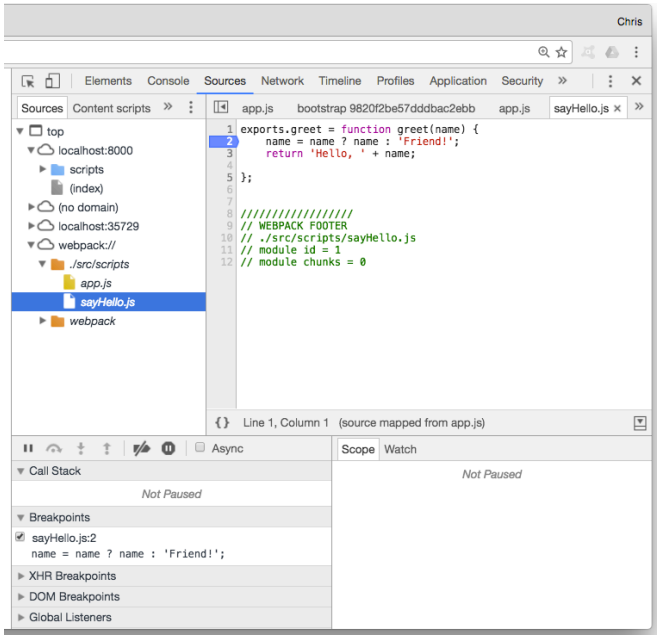
```
"bundle":"webpack"
```

☐ 16. Run **npm run bundle**

The **dist** directory, the **scripts** directory inside it, and the **app.js** file inside the **dist** directory will be created.

☐ 17. Open **dist/scripts/apps.js** and look at the code that's created to make it work.
☐ 18. Make a copy of **index.html** and put it in **dist**.
☐ 19. Enter **`npm start`** and make sure that your page works in your browser.
☐ 20. Open the Chrome Dev Tools and go to the Sources tab. Click on the webpack link on the left pane to view the original source.

Note: You may need to clear your browser cache before you can view the sourcemap.



☐ 21. Since the **dist** directory is generated, we don't want to add it to our repository. So, add it to **.gitignore**

```
# generated files
dist/
```

☐ 22. The generated bundle won't pass our linting rules, so add that to the **.eslintignore** file as well.

☐ 23. It's a good practice to clean up your distribution directory before each build. The goal is to avoid the possibility of any files remaining from previous builds that might cause problems with the app. Everything that's in the **dist** directory should be automatically generated by the build script.

To start getting to that point, we'll install a webpack plugin to clean the **dist** directory before each build.

☐ 24. Install the `clean-webpack-plugin`.

```
npm install clean-webpack-plugin --save-dev
```

☐ 25. Require the clean-webpack-plugin inside webpack-config.js

```
const CleanWebpackPlugin = require('clean-webpack-plugin');
```

☐ 26. Add a new property, named plugins to the webpack config object and inside of it create an instance of the `CleanWebpackPlugin` with the path (from the root of the project) to the **dist** directory as its parameter. Your **webpack.config.js** file should look like this now:

```
module.exports = {
    mode: 'development',
    devtool: 'source-map',
    entry: './src/scripts/app.js',
    output: {
      path: path.resolve(__dirname, 'dist'),
      filename: 'scripts/app.js'
    },
    plugins: [
        new CleanWebpackPlugin()
    ]
};
```

☐ 27. Run the bundle task.

Notice that webpack logs a message telling you that the **dist** directory has been removed prior to creating the bundle.

Next, we need to make the homepage inside the **dist** directory. One way to do this would be to just copy over the **index.html** file from **src** to **dist**. A much cooler way to do it is to have webpack dynamically create the **index.html** file, using a template!

☐ 28. Install **html-webpack-plugin**

```
npm install html-webpack-plugin --save-dev
```

☐ 29. Require html-webpack-plugin inside **webpack.config.js**

```
const HTMLWebpackPlugin = require('html-webpack-
plugin');
```

☐ 30. Configure the plugin in **webpack.config.js**:

```
plugins: [
    new CleanWebpackPlugin(),
    new HTMLWebpackPlugin({
        filename: 'index.html',
        title: 'Welcome to my page!',
        mainDiv: 'welcome-message',
        template: 'src/index.html'
    })
]
```

☐ 31. Update **src/index.html** to be a template, and remove the <script> tag, since that will be inserted by webpack going forward.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title><%= htmlWebpackPlugin.options.title
%></title>
</head>
<body>
<h1 id="<%= htmlWebpackPlugin.options.mainDiv
%>">Welcome</h1>
</body>
</html>
```

☐ 32. Run the **bundle** task and test to make sure everything works.

☐ 33. Take a look at the documentation for **html-webpack-plugin** and see if there's anything else that you want it to do:

```
https://www.npmjs.com/package/html-webpack-plugin
```

Part 2: Modify the Karma Config

The next thing we'll do is to modify our karma configuration so that it will use webpack to bundle the files for testing. Follow these steps:

☐ 1. Install **karma-webpack**.

```
npm install karma-webpack@3.0.0 --save-dev
```

☐ 2. Remove `commonjs` from the frameworks array.

☐ 3. Remove the **src** directory from the **files** array. It should now look like this:

```
// list of files / patterns to load in the browser
files: [
    'spec/**/*Spec.js'
],
```

☐ 4. Change the preprocessors from **commonjs** to **webpack**:

```
preprocessors: {
    'src/**/*.js': ['webpack'],
    'spec/**/*.js': ['webpack']
},
```

☐ 5. Run your **test** script to confirm that your tests pass.

```
npm test
```

Part 3: Integrate Bundling into the Build Script

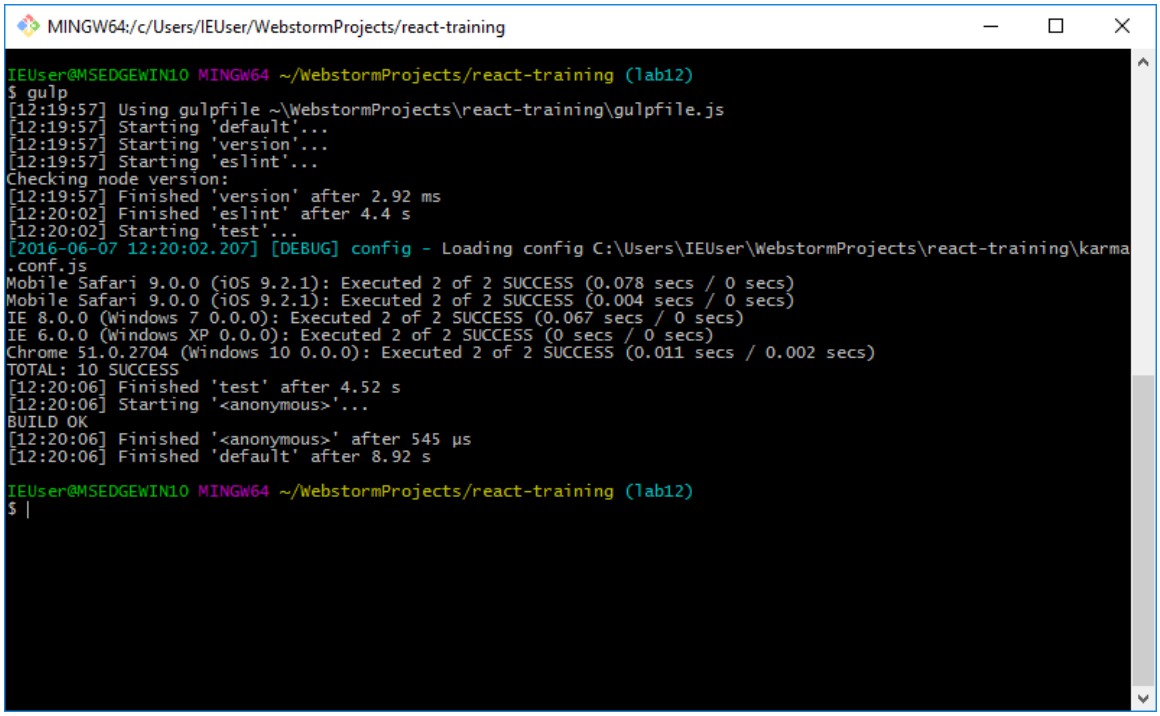
In this part, you'll make the bundling of the assets and the creation of the dist directory a step in your build process.

☐ 1. Add the bundle task to the end of the **prebuild** script in **package.json**.

```
"prebuild": "npm run version && npm run lint && npm
run test && npm run bundle"
```

☐ 2. Run `npm run build`. Everything should work correctly and your tests should all pass.

Note: At this point, you may be getting linting errors and excessive warnings in config files (such as karma.conf.js). If you want to exclude these files from linting, add them to .eslintignore or add `/* eslint-disable */` to the beginning of each file that should be excluded.


MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab12)
$ gulp
[12:19:57] Using gulpfile ~\WebstormProjects\react-training\gulpfile.js
[12:19:57] Starting 'default'...
[12:19:57] Starting 'version'...
[12:19:57] Starting 'eslint'...
Checking node version:
[12:19:57] Finished 'version' after 2.92 ms
[12:20:02] Finished 'eslint' after 4.4 s
[12:20:02] Starting 'test'...
[2016-06-07 12:20:02.207] [DEBUG] config - Loading config C:\Users\IEUser\WebstormProjects\react-training\karma
.conf.js
Mobile Safari 9.0.0 (iOS 9.2.1): Executed 2 of 2 SUCCESS (0.078 secs / 0 secs)
Mobile Safari 9.0.0 (iOS 9.2.1): Executed 2 of 2 SUCCESS (0.004 secs / 0 secs)
IE 8.0.0 (Windows 7 0.0.0): Executed 2 of 2 SUCCESS (0.067 secs / 0 secs)
IE 6.0.0 (Windows XP 0.0.0): Executed 2 of 2 SUCCESS (0 secs / 0 secs)
Chrome 51.0.2704 (Windows 10 0.0.0): Executed 2 of 2 SUCCESS (0.011 secs / 0.002 secs)
TOTAL: 10 SUCCESS
[12:20:06] Finished 'test' after 4.52 s
[12:20:06] Starting '<anonymous>'...
BUILD OK
[12:20:06] Finished '<anonymous>' after 545 µs
[12:20:06] Finished 'default' after 8.92 s
IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab12)
$

Lab 13 - README update and Refactoring

Refactoring and documentation are a very important part of any development process. In this lab, you will take a look at what you've done so far and find ways to clean it up and make it better.

☐ 1. Take some time to update your README file.

Think about what future developers (or your future self) would need to know about how everything works so far. Especially consider what a new developer coming into this project would need to know in order to become productive as quickly as possible.

Visit *https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet* for a guide to formatting with Markdown.

☐ 2. Run `npm dedupe` in your project.

`npm dedupe` searches the local package tree and attempts to simplify the overall structure by moving dependencies further up the tree, where they can be more effectively shared by multiple dependent packages.

Another benefit of `npm dedupe` is that it will eliminate some long paths that can break these tools on Windows (due to the Windows path length limit).

☐ 3. Reorganize your tasks in **package.json**.

Would the tasks be easier to understand if you rearranged them?

Are there any tasks that you want to rename or improve?

Are there any new tasks you want to create?

Lab 14 - Babel

After this lab, we'll start converting our front-end code to make use of ES2015. To be able to run this code, we'll need to install the Babel compiler.

☐ 1. Install babel, babel-loader, babel-preset-env.

```
npm install @babel/core babel-loader @babel/preset-env
--save-dev
```

☐ 2. Create a file named .babelrc in the root of your project. It should have the following contents.

```
{
"presets": [
"@babel/preset-env"
]
}
```

☐ 3. Add the loader and the babel preset into the webpack config:

```
const CleanWebpackPlugin = require('clean-webpack-
plugin');
const HTMLWebpackPlugin = require('html-webpack-
plugin');
const path = require('path');


module.exports = {
    mode: 'development',
    devtool: 'source-map',
    entry: './src/scripts/app.js',
    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'scripts/app.js'
    },
    module : {
      rules: [ {
            test: /.js$/,
            use: {
                  loader: 'babel-loader'
            }
        }]
    },
    plugins: [
```

```
        new CleanWebpackPlugin(),
        new HTMLWebpackPlugin({
            filename: 'index.html',
            title: 'Welcome to my page!',
            mainDiv: 'welcome-message',
            template: 'src/index.html'
        })
    ]
};
```

☐ 4. Add webpack settings inside karma.conf.js

```
preprocessors: {
    'src/**/*.js': ['webpack'],
    'spec/**/*.js': ['webpack']
},

webpack:{
    mode: 'development',
    entry  : './src/scripts/app.js',

    module : {
      rules: [ {
            test: /.js$/,
            use: {
                  loader: 'babel-loader'
            }
        }]
    }
```

☐ 5. Update **src/scripts/app.js** to use ES6:

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

window.addEventListener('load',()=>{
    document.getElementById('welcome-
message').innerHTML = sayHello.greet('Chris');
});
```

☐ 6. Run your tests and build to make sure everything still works.

☐ 7. At this point, you may be noticing that your webpack tasks are taking longer. You can speed things up by excluding the node_modules folder from the loader test, like this (in **webpack.config.js** and **karma.conf.js**):

```
module : {
      rules: [ {
            test: /.js$/,
```

```
            exclude: /node_modules/,
            use: {
            loader: 'babel-loader'
            }
        }]
    }
```

☐ 8. **npm run build**

☐ 9. **npm run start** and confirm that the page still works the same when you open it in a browser.

Lab 15 - Converting to ES6

In this lab, we'll modify sayHello.js and sayHelloSpec.js to make use of some of the features of ES6. Then, we'll build our application and confirm that Babel is compiling the code to ES5 correctly and that it runs in our target web browsers.

- ☐ 1. Open `sayHello.js`
- ☐ 2. Export the module using ES6 syntax:

  replace:

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

  with:

  ```
  export function greet(name) {

  ...
  ```

- ☐ 3. Open **app.js**
- ☐ 4. Import the module using ES6 syntax.

  replace:

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

  with:

  ```
  import * as sayHello from './sayHello.js';
  ```

- ☐ 5. Open `sayHelloSpec.js`
- ☐ 6. Import sayHello using ES6 syntax.

  ```
  import * as sayHello from
  '../src/scripts/sayHello.js';
  ```

- ☐ 7. Run your tests to check that everything works!
- ☐ 8. Convert the `greet()` function into an ES6 arrow function. See if you can do it yourself before turning the page to see my solution.

```
export let greet = (name) => {
    name = name ? name : 'Friend!';
    return 'Hello, ' + name;
};
```

☐ 9. test, build, start, commit!

Lab 16 - Hello, React

Part 1: Say hello and test your setup

In this lab, you will install React and create a simple react component.

☐ 1. Install React and react-dom

```
npm install --save react react-dom
```

Note that we're using --save instead of --save-dev. The reason is that we'll be using `react` and `react-dom` in our production environment, not just development.

☐ 2. Install babel-preset-react

```
npm install --save-dev @babel/preset-react
```

☐ 3. Add the react preset to .babelrc

```
{
  "presets": [
    "@babel/preset-env",
    "@babel/preset-react"
  ]
}
```

☐ 4. Remove the `<h1>` in src/index.html and insert an empty `div` element with an `id` attribute.

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

This will be the hook that we'll use to render the React component.

☐ 5. Open **app.js** and remove everything that's in there currently.
☐ 6. Require react and react-dom in **app.js**:

```
import React from 'react';
import ReactDOM from 'react-dom';
```

☐ 7. Enter the following into **app.js**:

```
ReactDOM.render(
<h1>Hello, world!</h1>,
    document.getElementById('app')
);
```

☐ 8. Run the bundle script

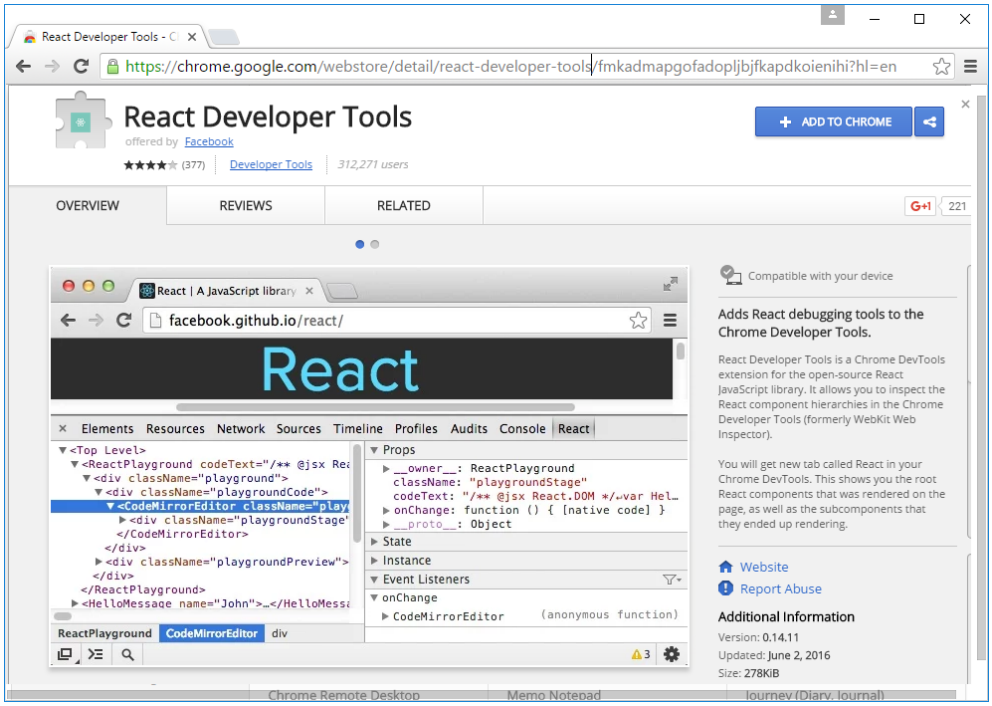
```
npm run bundle
```

☐ 9. If you're using Chrome, open the Developer Tools (Control-Shift-I on Windows or CMD-Option-I on Mac) and click and hold the refresh icon, then select Hard Reload. This will make sure that your browser doesn't try to use a cached version of app.js.

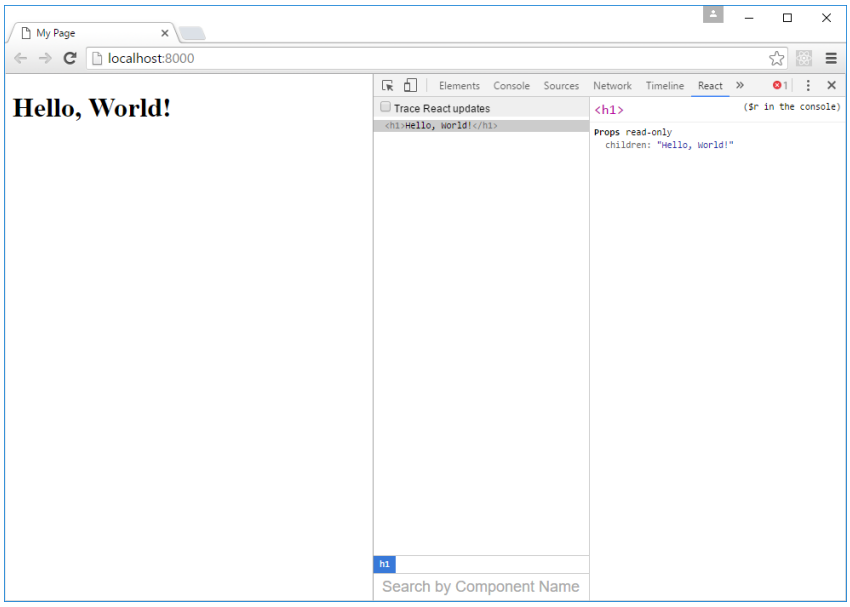
☐ 10. Run the start script

```
npm start
```

☐ 11. Search for and install the React Developer Tools Chrome extension.



☐ 12. Refresh your browser window if necessary, then open the Developer Tools and click the React tab to view the React Developer Tools.



Part 2: Make a component

In this part, you will convert your `greet()` function in the sayHello module into a React.js component.

☐ 1. Create a React class in app.js, before the ReactDOM.render() method you created in part 1.

```
class SayHello extends React.Component {
  render(){
    return (<h1>Hello, {this.props.name}</h1>);
  }
}
```

☐ 2. Modify the ReactDOM render call to use the new component.

```
ReactDOM.render(<SayHello name="World" />,
      document.getElementById("app"));
```

☐ 3. Run **`npm run bundle`** and **`npm start`** to test it out.

Next, we'll move the React component into a separate module.

☐ 4. Rename sayHello.js to SayHello.js

React components start with uppercase letters by convention.

☐ 5. Open **SayHello.js** and delete its contents.
☐ 6. Require react (but not react-dom) in SayHello.js
☐ 7. Move the SayHello class from app.js to SayHello.js
☐ 8. At the bottom of SayHello.js, export SayHello:

```
export default SayHello;
```

☐ 9. Import SayHello in app.js:

```
import SayHello from './SayHello';
```

☐ 10. Do a Hard Reload, then re-bundle (**note:** don't do a build, because your tests won't work until after you complete Part 3) and then npm start to see your app in a browser.

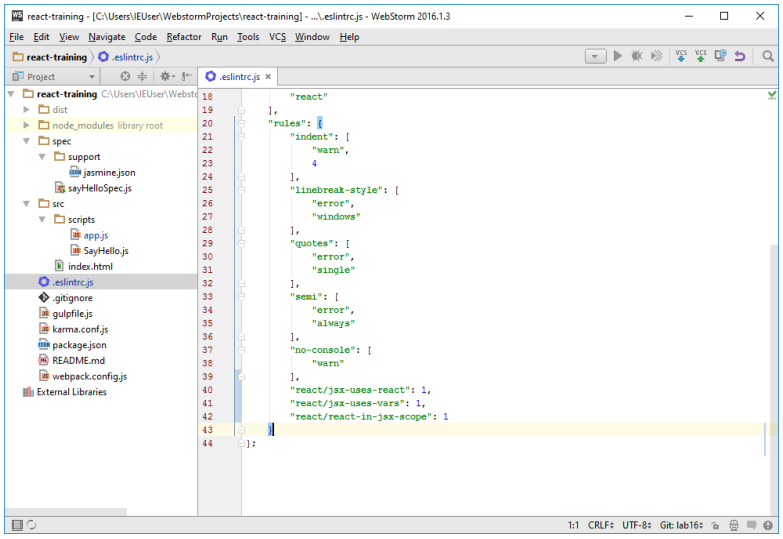
You may need to hard refresh (Ctrl - Shift - R on Windows or Cmd - Shift - R on Mac)

If it's working, Congratulations! Move on to the next part, where we'll get the tests working again.

Part 3: Configure ESLint and Karma for React

☐ 1. Add the following rules to your **.eslintrc.js** file (inside the **rules** object):

```
"react/jsx-uses-react": 1,
"react/jsx-uses-vars": 1,
"react/react-in-jsx-scope": 1
```



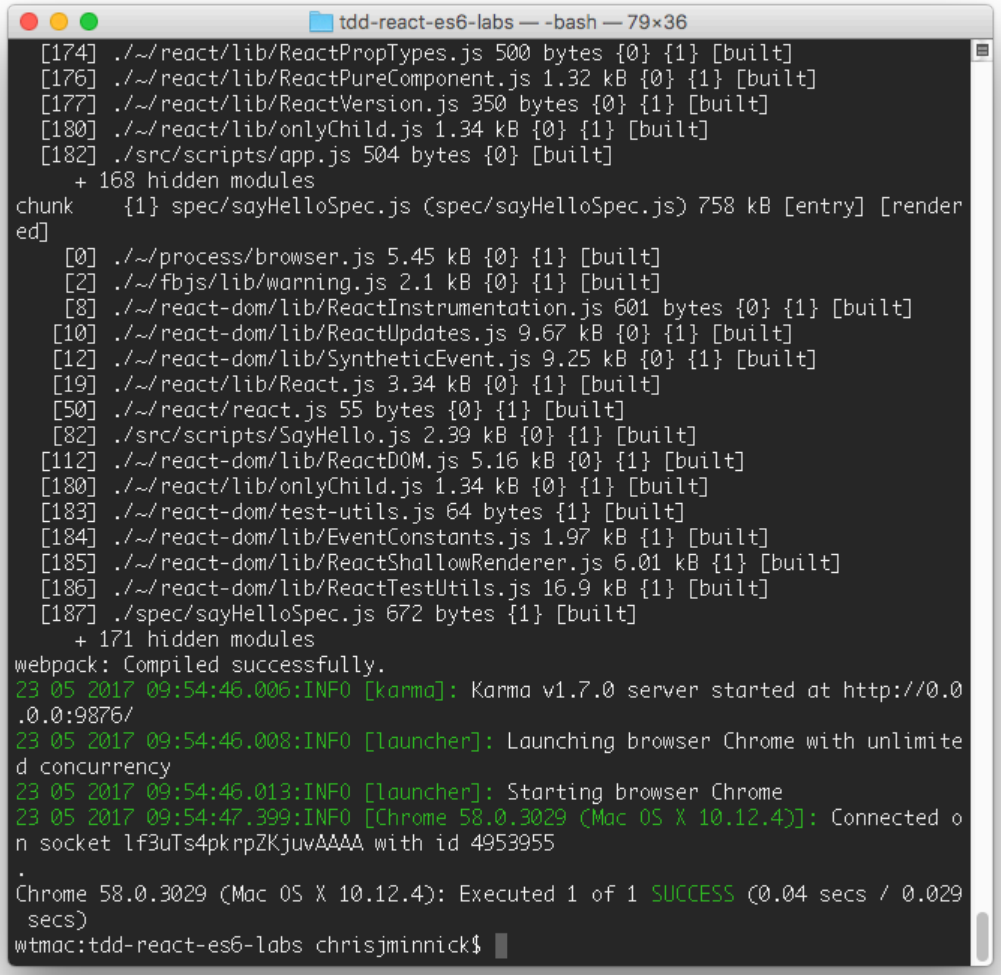
☐ 2. Run the **lint** task and make the necessary changes to your code to get the tests to pass.

☐ 3. Update sayHelloSpec.js to test whether the SayHello component renders.

```
import React from 'react';
import TestUtils from 'react-dom/test-utils';
import SayHello from '../src/scripts/SayHello.js';

describe('Greet', function() {
    it('renders without problems', function () {
        const sayhello =
TestUtils.renderIntoDocument(<SayHello />);
        expect(sayhello).toEqual(jasmine.anything());
    });
});
```

☐ 4. Run the npm test script to run your test.


tdd-react-es6-labs — -bash — 79×36
[174] ./~/react/lib/ReactPropTypes.js 500 bytes {0} {1} [built]
[176] ./~/react/lib/ReactPureComponent.js 1.32 kB {0} {1} [built]
[177] ./~/react/lib/ReactVersion.js 350 bytes {0} {1} [built]
[180] ./~/react/lib/onlyChild.js 1.34 kB {0} {1} [built]
[182] ./src/scripts/app.js 504 bytes {0} [built]
+ 168 hidden modules
chunk {1} spec/sayHelloSpec.js (spec/sayHelloSpec.js) 758 kB [entry] [rendered]
[0] ./~/process/browser.js 5.45 kB {0} {1} [built]
[2] ./~/fbjs/lib/warning.js 2.1 kB {0} {1} [built]
[8] ./~/react-dom/lib/ReactInstrumentation.js 601 bytes {0} {1} [built]
[10] ./~/react-dom/lib/ReactUpdates.js 9.67 kB {0} {1} [built]
[12] ./~/react-dom/lib/SyntheticEvent.js 9.25 kB {0} {1} [built]
[19] ./~/react/lib/React.js 3.34 kB {0} {1} [built]
[50] ./~/react/react.js 55 bytes {0} {1} [built]
[82] ./src/scripts/SayHello.js 2.39 kB {0} {1} [built]
[112] ./~/react-dom/lib/ReactDOM.js 5.16 kB {0} {1} [built]
[180] ./~/react/lib/onlyChild.js 1.34 kB {0} {1} [built]
[183] ./~/react-dom/test-utils.js 64 bytes {1} [built]
[184] ./~/react-dom/lib/EventConstants.js 1.97 kB {1} [built]
[185] ./~/react-dom/lib/ReactShallowRenderer.js 6.01 kB {1} [built]
[186] ./~/react-dom/lib/ReactTestUtils.js 16.9 kB {1} [built]
[187] ./spec/sayHelloSpec.js 672 bytes {1} [built]
+ 171 hidden modules
webpack: Compiled successfully.
23 05 2017 09:54:46.006:INFO [karma]: Karma v1.7.0 server started at http://0.0.0.0:9876/
23 05 2017 09:54:46.008:INFO [launcher]: Launching browser Chrome with unlimited concurrency
23 05 2017 09:54:46.013:INFO [launcher]: Starting browser Chrome
23 05 2017 09:54:47.399:INFO [Chrome 58.0.3029 (Mac OS X 10.12.4)]: Connected on socket lf3uTs4pkrpZKjuvAAAA with id 4953955
.
Chrome 58.0.3029 (Mac OS X 10.12.4): Executed 1 of 1 SUCCESS (0.04 secs / 0.029 secs)
wtmac:tdd-react-es6-labs chrisjminnick$

Lab 17 - Breaking up a UI into Components

In this lab, we'll start with an HTML UI and convert it into static React components.

The application UI we're going to start building is a simple poll application that asks the user a multiple-choice question and displays results.



☐ 1. Think about how you might break this UI into components.

Here's one way you might do it:

```
PollHeader
PollQuestion
PollAnswer
PollAnswer
PollAnswer
PollSubmitButton
```

In addition to these, it's a common pattern to create a component to contain all the other components in the view. So, we'll create another component called `PollContainer`.

☐ 2. Create a directory called components and a directory called containers inside of your **src** directory.

☐ 3. Create a test suite and a first spec for each of the components you'll create and write a simple test based on sayHelloSpec (which we created in the previous lab) that checks whether the component renders.

☐ 4. Run the tests to confirm that they fail.

☐ 5. Make a new file for each of the components in this view.

☐ 6. Require react, insert the basic boilerplate component render method, and export each of the modules. Here's the PollHeader component:

```
import React from 'react';

class PollHeader extends React.Component {
      render(){
            return (
                  <h1>Welcome!</h1>
            );
      }
}

export default PollHeader;
```

☐ 7. Make a new file named **PollContainer.js** inside the containers directory, with the following code:

```
import React from 'react';
import PollHeader from '../components/PollHeader';
import PollQuestion from '../components/PollQuestion';
import PollAnswer from '../components/PollAnswer';
import PollSubmitButton from '../components/PollSubmitButton';

class PollContainer extends React.Component {
    render(){
        return (
            <div className="container">
                <div className="col-sm-4 col-sm-offset-4">
                    <PollHeader />
                    <form>
                      <PollQuestion />
                      <PollAnswer />
                      <PollAnswer />
                      <PollAnswer />
```

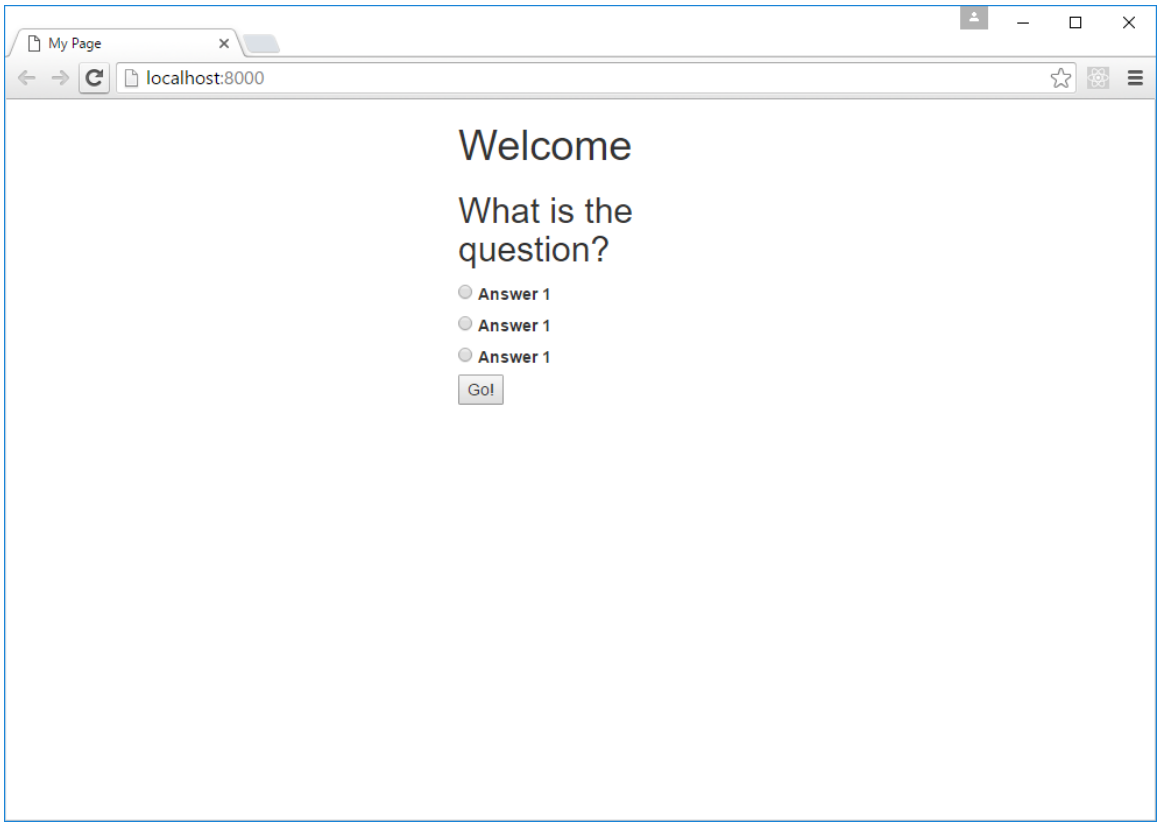
```
                    <PollSubmitButton />
                  </form>
                </div>
              </div>
          );
      }
}

export default PollContainer;
```

- ☐ 8. After you create the components, run your tests to confirm that the associated tests pass.
- ☐ 9. Modify app.js to require and render <PollContainer /> instead of <SayHello />.
- ☐ 10. Put the following CSS include in the `<head>` element of index.html to include the Bootstrap CSS.

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">
```

- ☐ 11. npm run build



☐ 12. Go back through each of your new components and see if you can improve them, then run your tests and repeat.

Part 2: Bundle tests for Karma

Currently, Karma is building each test suite with React separately, which could cause problems as you have more suites. To fix this, we'll wrap all your tests inside a single file.

☐ 1. Create a new file inside /spec named **tests.webpack.js**
☐ 2. Enter the following code into tests.webpack.js:

```
var context = require.context('.', true, /Spec\.js$/);
context.keys().forEach(context);
```

☐ 3. Update the files path in karma.conf.js to point to tests.webpack.js

```
files: [
    'spec/tests.webpack.js'
],
```

☐ 4. Run your tests.

Lab 18 - State and Props

In this lab, you will start adding state to the app.

The first thing we want to do is to allow components to be configured by their 'owner' components. To do this, we'll create state variables in PollContainer and pass them to the 'owned' components of PollContainerSpec.js.

☐ 1. Open **`PollHeaderSpec.js`**

Use beforeEach to render the component before each spec runs:

```
describe('Poll Header', function() {
    var component;
    beforeEach(function(){
        component = TestUtils.renderIntoDocument(
            <PollHeader text="Welcome to the Poll!" />
        );
    });
```

☐ 2. Update the "renders without problems" spec to use the rendered component from the beforeEach.

```
it('renders without problems', function() {
  expect(component).toEqual(jasmine.anything());
});
```

☐ 3. Create a new spec inside the PollHeader test suite:

```
it('prints a message', function() {
    var actual = TestUtils
    .findRenderedDOMComponentWithTag(component,'h1')
    .textContent;
    var expected = 'Welcome to the Poll!';
    expect(actual).toEqual(expected);
});
```

Run your tests to verify that the new spec fails.

☐ 4. Open **`PollHeader.js`**

☐ 5. Replace the text between <h1> and </h1> with a prop:

```
render() {
    return (<h1>{this.props.text}</h1>);
}
```

☐ 6. Run the tests again to verify that it passes.

☐ 7. Follow the same pattern to add specs for PollQuestion and PollAnswer and then make the tests pass.

Note: You may need to surround the PollAnswer text with a unique element, such as in order to be able to select it using findRenderedDOMComponentWithTag().

☐ 8. Create a constructor inside the `PollContainer` component. The constructor will call `super()` and set the initial state for the application.

```
class PollContainer extends React.Component{
  constructor(){
    super();
    this.state = {
    }
  }
```

For this first version, we're going to set the initial state of the application inside the constructor.

☐ 9. Create properties inside the state object for the following:

```
header
question
answer1
answer2
answer3
correctAnswer
```

☐ 10. Set the values of the properties in the state object to any values you like. For example,

```
this.state = {
    header: 'Welcome to the poll!',
    question: 'What is the best?',
    answer1: 'Tacos',
    answer2: 'Pizza',
    answer3: 'Cheese',
```

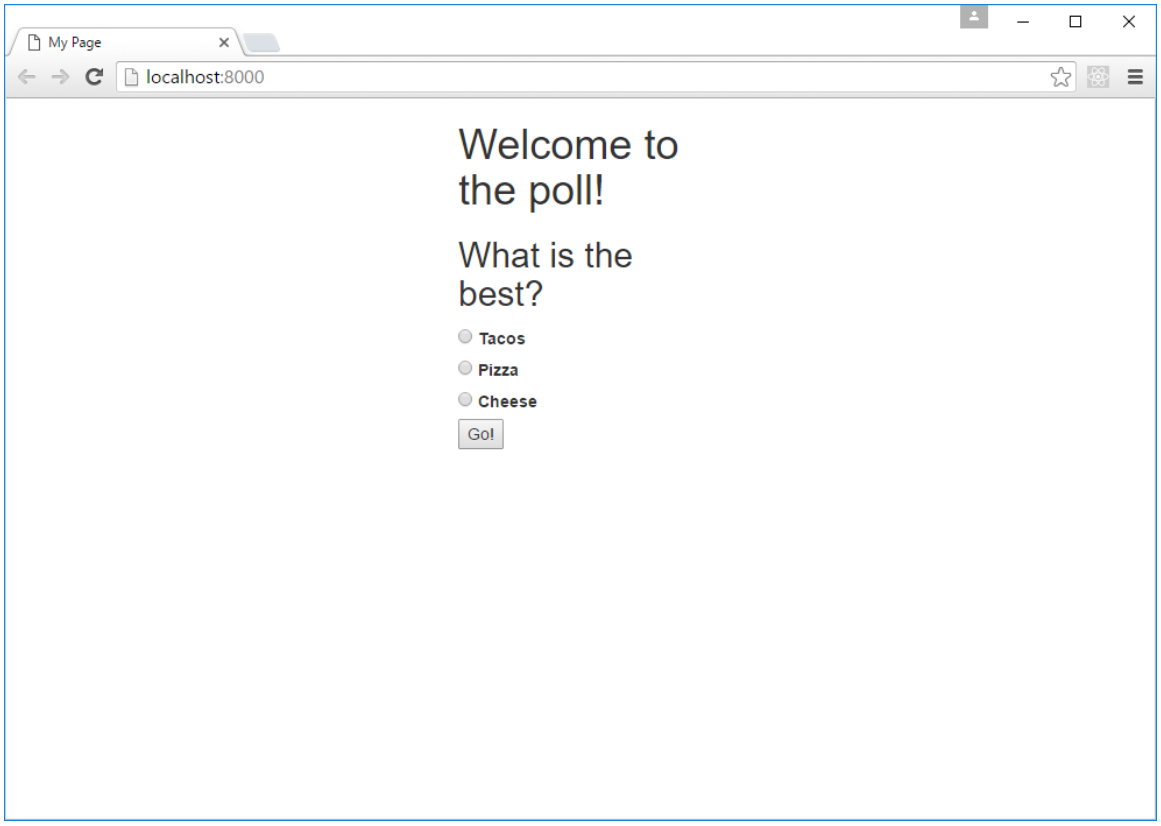
```
    correctAnswer: 'Tacos'
};
```

☐ 11. Modify the child component instances in PollContainer to accept the state properties.

```
<PollHeader text={this.state.header} />
<PollQuestion text={this.state.question}/>
<PollAnswer text={this.state.answer1} />
...
```

☐ 12. Run your tests to confirm that the child components all receive and print out their props correctly.

☐ 13. Build and run your app in a browser.



All this building and running is getting tiresome. Let's set up a task that will watch for changes and automatically re-build the app.

☐ 14. Install webpack-dev-server

```
npm install webpack-dev-server --save-dev
```

☐ 15. Change the npm start task to use webpack-dev-server

```
'start':'webpack-dev-server'
```

☐ 16. View the docs for webpack-dev-server and modify the script with the options you want.

```
https://webpack.js.org/guides/development/#webpack-dev-server
```

Here are some example options you might try out:

```
"start": "webpack-dev-server --progress --inline --open",
```

☐ 17. Start the server with `npm start`

☐ 18. Open **PollContainer.js** and make some changes to the question, header, or answers. Save the file and return to your web browser.

After the bundle is recompiled, you will see the changes reflected in your browser.

Lab 19 - Adding Style to React Components

Next, we'll add some styles to our components to make things look a little better. One common practice for styling React apps is to use a CSS framework (such as Bootstrap) for global styles and layout and use style objects for component-specific styles.

Because "class" is a reserved word in JavaScript (and therefore JSX), if you want to attach a class attribute to the HTML that results from a React component, you need to use the "className" JSX attribute.

There are MANY ways to include Bootstrap in your React app. The most straightforward way is just to put an HTML link tag in your index.html file and link to the Bootstrap CSS on a CDN (which we already did in Lab 17). For the sake of simplicity, we'll stick with this for now.

☐ 1. Add className = "radio" to the root div in PollAnswer.

☐ 2. Add className = "btn btn-success" to the <button> element in PollSubmitButton.

☐ 3. Create a with the className="jumbotron" above the div with className="row" in PollContainer.js and move the PollHeader component into it.

Your PollContainer JSX should now look like this:

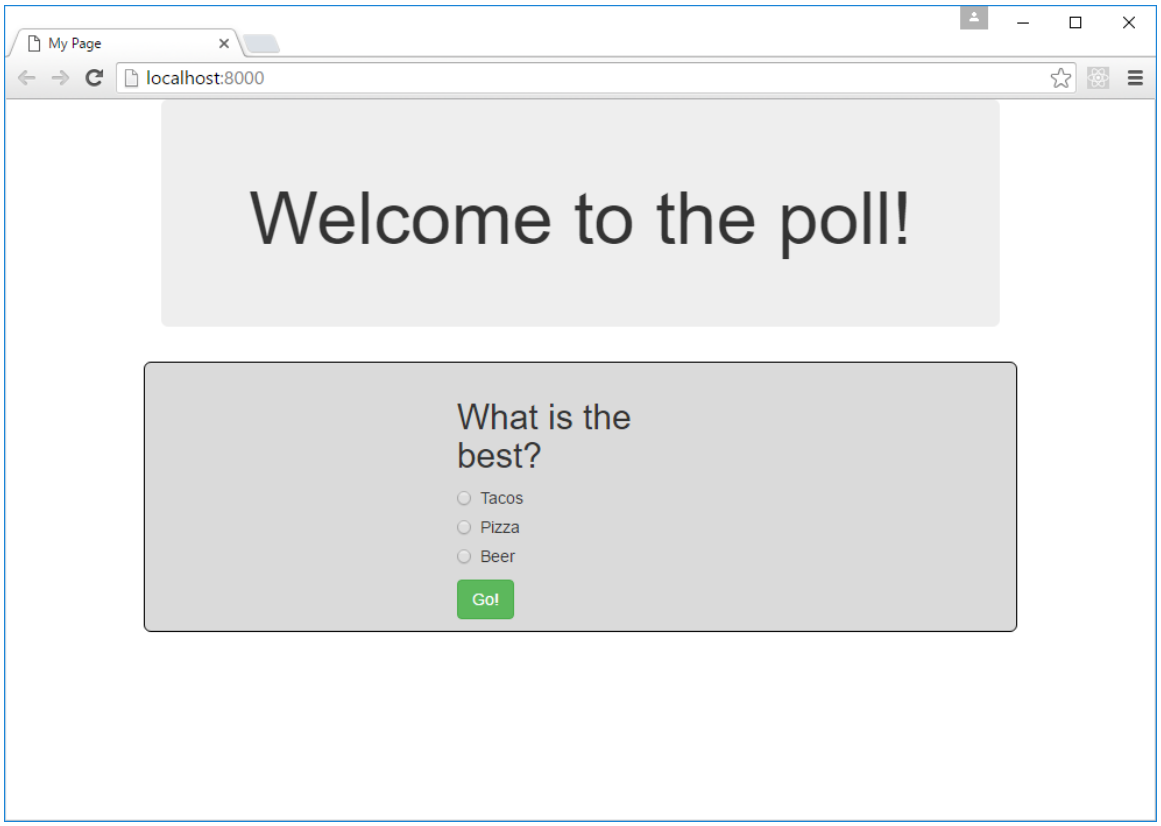
```
<div className="container">
    <div className="jumbotron">
        <PollHeader text={this.state.header} />
    </div>
    <div className="row">
        <div className="col-sm-4 col-sm-offset-4">
            <form>
                <PollQuestion
text={this.state.question}/>
                <PollAnswer
text={this.state.answer1}/>
                <PollAnswer text={this.state.answer2}
/>
                <PollAnswer text={this.state.answer3}
/>
                <PollSubmitButton />
            </form>
        </div>
    </div>
</div>
```

Now that we've added some Bootstrap classes, let's create our first style object.

☐ 4. Create a variable inside the render method of pollContainer called rowStyle, and assign an object to it, and add style=`{rowStyle}` to the `<div className="row">`.

```
var rowStyle = {
  backgroundColor: '#dadada',
  border: '1px solid black',
  borderRadius: '6px',
  padding: '10px'
};
```

- ☐ 5. Add `className="text-center"` to the `<h1>` in PollHeader
- ☐ 6. Make additional CSS changes as time allows and as you wish.
- ☐ 7. Challenge: Figure out how to install Bootstrap with npm and bundle it into your app using Webpack. Hint: it will involve additional loaders.



Lab 20 - Controlling the Form

In this lab, we'll make our inputs be controlled by React and add a method for updating their state.

The first thing we'll do is to create a new component to control the creation of the radio buttons and answer labels so that the question can have as many multiple choice questions as necessary.

- ☐ 1. Change the name of **PollAnswer.js** to **RadioButton.js** and update references and tests accordingly.
- ☐ 2. Create a new module in the components directory containing a component named RadioButtonGroup.
- ☐ 3. Import react and RadioButton into this new module.
- ☐ 4. In the render method for RadioButtonGroup, create a new const called choiceItems and use Array.map to return a RadioButton for each element of the choices array:

```
const choiceItems = this.props.choices.map(choice => {
    const {value, label} = choice;
    const checked = value === this.props.checkedValue;

    return (
        <RadioButton
            key={`radio-button-${value}`}
            label={label}
            name={this.props.name}
            value={value}
            checked={checked}
        />
    );
});
```

- ☐ 5. In the RadioButtonGroup component, return a div containing the value of choiceItems:

```
return (
    <div>
        {choiceItems}
    </div>
);
```

The finished RadioButtonGroup module should look like this:

```
import React from 'react';
import RadioButton from './RadioButton';

class RadioButtonGroup extends React.Component {

    render() {

        const choiceItems = this.props.choices.map(choice => {
            const {value, label} = choice;
            const checked = value === this.props.checkedValue;

            return (
                <RadioButton
                    key={`radio-button-${value}`}
                    label={label}
                    name={this.props.name}
                    value={value}
                    checked={checked}
                />
            );
        });

        return (
            <div>
                {choiceItems}
            </div>
        );
    }

}

export default RadioButtonGroup;
```

Next, we'll make some changes to the RadioButton component so that we can pass values and checked state into the component.

☐ 6. Modify the JSX in RadioButton.js so that it takes additional properties (which we'll create shortly).

```
<div className="radio">
<label>
    <input type="radio"
           name={this.props.name}
           value={this.props.value}
           checked={this.props.checked}
           />
    <span>{this.props.label}</span>
</label>
</div>
```

☐ 7. In PollContainer, import the RadioButtonGroup component.

☐ 8. Replace the 3 instances of `<RadioButton>` with `<RadioButtonGroup>`, like this:

```
<RadioButtonGroup
    name='answer'
    checkedValue={this.state.checkedValue}
    choices={choices} />
```

☐ 9. In the constructor function of PollContainer, delete answer1, answer2, and answer3 from the state object.

☐ 10. In PollContainer, create a new array in the render function for the answer choices.

```
const choices = [
     {value: 'Tacos', label: 'Tacos'},
     {value: 'Pizza', label: 'Pizza'},
     {value: 'Cheese', label: 'Cheese'}
     ];
```

☐ 11. Add a new property to the state, called checkedValue and set its value to an empty string.

```
checkedValue: ''
```

☐ 12. Build and run the app.

Notice that clicking on the radio buttons no longer changes their state.

☐ 13. In the state object in the PollContainer's constructor, change the value of the checkedValue property to **Tacos**.

After the app rebuilds, you should see the radio button next to Tacos checked.

☐ 14. Set the value of **checkedValue** back to ''.

Next we'll wire up an event that will change the state of the controlled radio buttons when the user clicks them.

☐ 15. In PollContainer, create a new method just below the constructor, called `setCheckedValue` that takes a parameter of `value` and uses it to change `checkedValue` in the state. We'll also have it log the current selection so that we can verify that it's working correctly.

```
setCheckedValue(value){
    this.setState({
        checkedValue: value
    });
    console.log("current choice: " + value);
}
```

☐ 16. In the constructor (below the state object) add this line:

```
this.setCheckedValue =
this.setCheckedValue.bind(this);
```

☐ 17. Add `onChange = {this.setCheckedValue}` to the RadioButtonGroup element in the return method of PollContainer.

☐ 18. In RadioButton, add an onChange attribute to the <input>, with a value of onChange={this.handleChange.bind(this)}

☐ 19. Add a new method to RadioButton (above the render function) called **handleChange**. Here's what it should look like:

```
handleChange() {
    this.props.onChange(this.props.value);
}
```

☐ 20. Add an onChange event attribute to the instance of RadioButton in RadioButtonGroup.

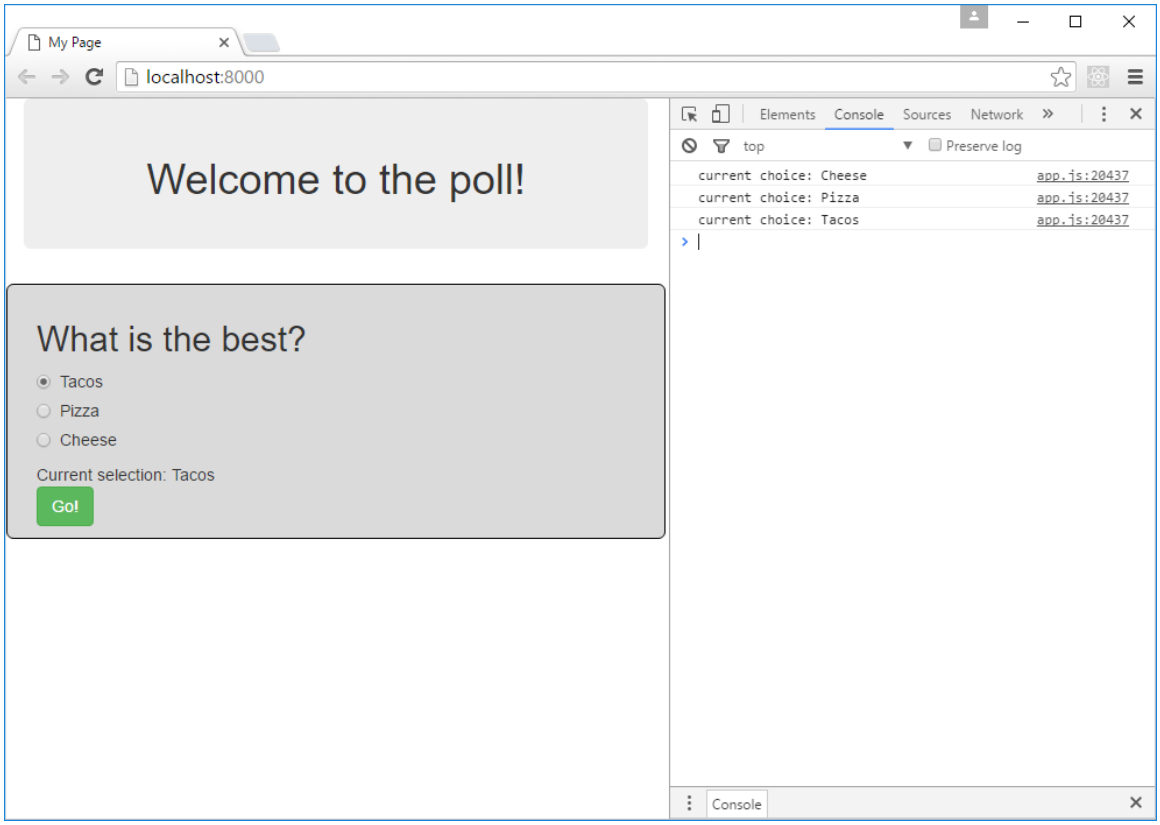
```
<RadioButton
    key={`radio-button-${value}`}
    label={label}
    name={this.props.name}
    value={value}
    checked={checked}
    onChange={this.props.onChange}
/>
```

☐ 21. Build and run to verify that clicking the radio buttons changes which one is selected.

☐ 22. Check the JavaScript console to make sure that the current selection is being logged when you click a radio button.

☐ 23. Create a new component named **CurrentChoice** that will output the value of the currently selected radio button just below the choices in RadioButtonGroup.

If it all works, congratulations! You now have React controlling your form's radio buttons.


My Page
localhost:8000
Welcome to the poll!
What is the best?
Tacos
Pizza
Cheese
Current selection: Tacos
Go!
Elements
Console
Sources
Network
top
Preserve log
current choice: Cheese
app.js:20437
current choice: Pizza
app.js:20437
current choice: Tacos
app.js:20437
Console

Lab 21 - Refactoring and Using JSON Data

In this lab, we'll re-think the structure of our app and make some changes to make it more efficient and simpler.

☐ 1. Rewrite `PollHeader`, `PollQuestion`, `PollSubmitButton`, and `CurrentChoice` as Stateless Functional Components. Here's one to get you started:

```
import React from 'react';

function CurrentChoice(props){
    return(<div>Current selection:
{props.checked}</div>);
}

export default CurrentChoice;
```

After you convert components into functional components, your tests of those components will fail. Functional components can't be used directly with `render` or `renderIntoDocument`. The solution is to wrap them in a wrapper component for testing purposes.

☐ 2. Create a new component named `TestWrapper`:

```
import React from 'react';

class TestWrapper extends React.Component{
      render(){
            return this.props.children;
      }
}

export default TestWrapper;
```

☐ 3. Import TestWrapper into each of your tests of functional components and modify the `renderIntoDocument` as follows:

```
beforeEach(function(){
      component = TestUtils.renderIntoDocument(
            <TestWrapper>
                  <PollHeader text="Welcome to the
Poll!" />
            </TestWrapper>
      );
});
```

Run your tests to make sure they pass.
Next, we'll move the `choices` object, `question`, the `correctAnswer` and the `pollHeader` into a separate file. Later, we can easily replace this module with an AJAX call to a Web API.

- ☐ 4. Create a new directory in the `src` directory called `data`, and a file within it called `data.json`.
- ☐ 5. Inside data.json, write the poll's data using JSON.

Here's one way you could do it:

```
{"poll":
{"header": "Welcome to the Poll!",
  "questions" : [{
    "question": "What is the best?",
    "choices": [
      {"value": "Tacos", "label": "Tacos"},
      {"value": "Pizza", "label": "Pizza"},
      {"value": "Cheese", "label": "Cheese"}
    ],
    "correctAnswer": "Pizza"
  },
    {
      "question": "What's your favorite color?:",
      "choices": [
        {"value": "Orange", "label": "Orange"},
        {"value": "Blue", "label": "Blue"}
      ],
      "correctAnswer": "Blue"
    }
  ]
}}
```

- ☐ 6. Import **data.json** into PollContainer.

```
import data from '../data/data.json';
```

Webpack 4 loads json files by default, so there's no need to install an additional loader for this json file. However, if your .json file isn't saved with UTF-8 encoding, it may fail to load correctly. You can confirm / change the file encoding in WebStorm by selecting File Encoding from the File menu.

☐ 7. Update the references to choices, header, and the question text in PollContainer for the new JSON data. For example, using the code above, the choices array location would now be:

```
data.poll.questions[0].choices
```

☐ 8. Look through each component and find things that can be improved or simplified as time permits.

☐ 9. Check whether your tests still work.

Lab 22 - Life Cycle and Events

In this lab, we'll look at the component life cycle and use the life cycle events to load data using Ajax.

☐ 1. In PollContainer, log a message to the console when each of the life cycle events occurs.

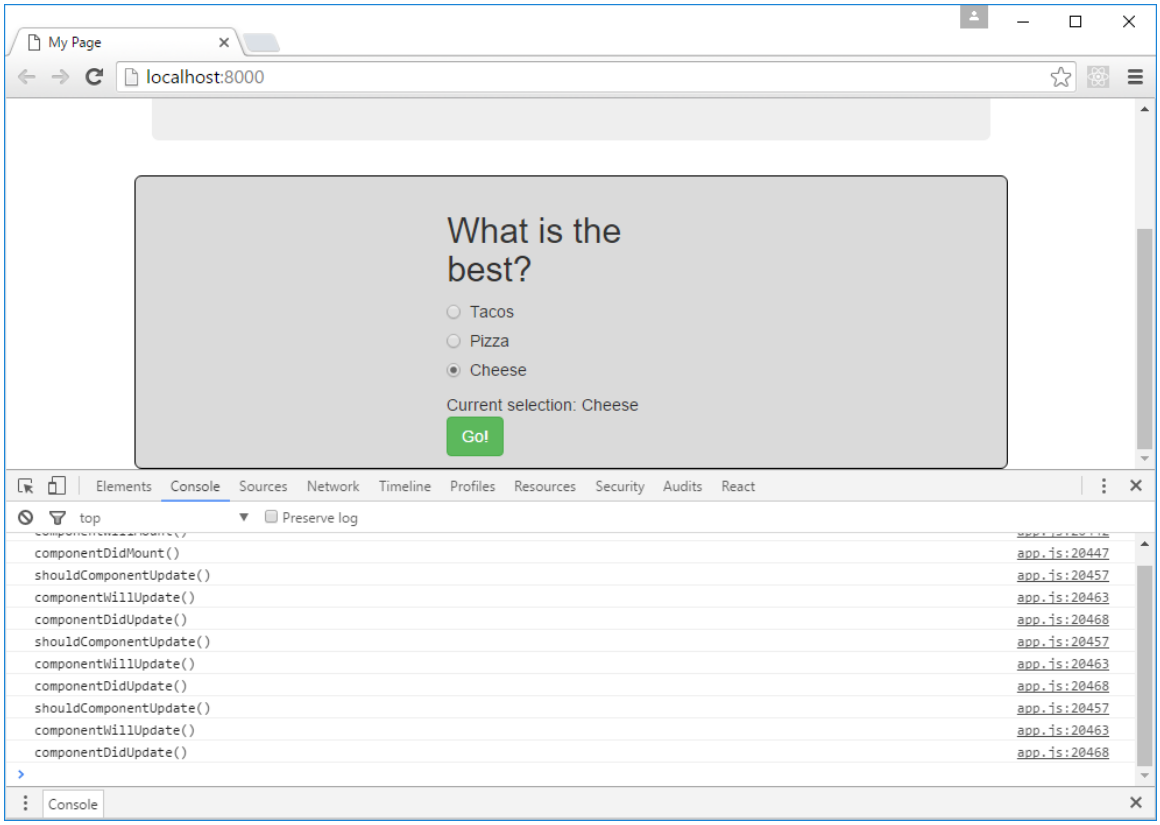
Notice that shouldComponentUpdate() needs to return a Boolean value.

```
UNSAFE_componentWillMount() {
    console.log('componentWillMount()');
}
componentDidMount() {
    console.log('componentDidMount()');
}
UNSAFE_componentWillReceiveProps() {
    console.log('componentWillReceiveProps()');
}
shouldComponentUpdate() {
    console.log('shouldComponentUpdate()');
    return true;
}
UNSAFE_componentWillUpdate() {
    console.log('componentWillUpdate()');
}
componentDidUpdate() {
    console.log('componentDidUpdate()');
}
componentWillUnmount() {
    console.log('componentWillUnmount()');
}
```

NOTE: The lifecycle methods that are prefaced with UNSAFE will be deprecated with version 17, so it's best to stop using them now. React 17 will introduce asynchronous rendering, which will make these synchronous lifecycle methods "unsafe" to use, because they will cause multiple renderings. https://t.co/toQmLDzrTy

☐ 2. Rebuild your app and open the JavaScript console in your browser.

Reload the app and click on the radio buttons and notice which events occur and when they occur.



Next, we'll use the componentDidMount() method to dynamically load data using AJAX.

☐ 3. Install **JQuery**

```
npm install --save jquery
```

☐ 4. Require jquery inside PollContainer as $

```
import $ from 'jquery';
```

☐ 5. Inside the componentDidMount method in PollContainer, use this code to retrieve the json data using AJAX:

```
componentDidMount(){
    console.log('componentDidMount');
    this.serverRequest =
      $.get('http://localhost:8080/data/data.json',
      function (result) {
         var data = result;
         this.setState({
             header: data.poll.header,
             question: data.poll.questions[0].question,
```

```
            choices: data.poll.questions[0].choices,
            correctAnswer: data.poll.questions[0].correctAnswer
        });
    }.bind(this));
}
```

- ☐ 6. Install the copy-webpack-plugin

  ```
  npm install copy-webpack-plugin --save-dev
  ```

- ☐ 7. Include copy-webpack-plugin in webpack.config.js

  ```
  const CopyWebpackPlugin = require('copy-webpack-plugin');
  ```

- ☐ 8. Update the plugins array in webpack-config.js to copy the data directory from src to dist

  ```
  plugins: [
      new CleanWebpackPlugin(),
      new HTMLWebpackPlugin({
          filename: 'index.html',
          title: 'Welcome to my poll!',
          template: 'src/index.html'
      }),
      new CopyWebpackPlugin([
          { from: 'src/data',
            to: 'data/'}
      ])
  ]
  ```

- ☐ 9. Remove the import that imports **data.json** from **PollContainer.**
- ☐ 10. Set initial values for header, question, and choices in the constructor in PollContainer.

  ```
  this.state = {
      header: '',
      question: '',
      correctAnswer: '',
      choices: [],
      checkedValue: ''
  };
  ```

- ☐ 11. Remove the const that sets the value of choices in the render method.
- ☐ 12. Change the <RadioButtonGroup> element in PollContainer to pass this.state.choices to the RadioButtonGroup component.
- ☐ 13. Build, test, debug.
- ☐ 14. Create a new function or component that will display whether the currently selected answer is the correct one.

One way to do this is to detect when the component updates and check the selected input against the correct answer from the data.

Lab 23 - PropTypes

PropTypes allow you to do validation on props passed into components. They're useful for debugging, especially as a program gets larger.

As of React 15.5.0, PropTypes are no longer part of the core React library. Instead, they've been moved into a separate package, called prop-types.

☐ 1. Install prop-types

```
npm install --save prop-types
```

☐ 2. Import PropTypes into your RadioButtonGroup module.

```
import PropTypes from 'prop-types';
```

☐ 3. Under the class definition for RadioButtonGroup, add the following PropTypes:

```
RadioButtonGroup.propTypes = {
    name: PropTypes.array,
    checkedValue: PropTypes.bool,
    choices: PropTypes.number,
    onChange: PropTypes.string
}
```

☐ 4. Build and run the app with the browser console open.
☐ 5. Read the warnings that appear, and then fix them.
☐ 6. Add appropriate PropTypes to the other components that receive props.

Lab 24 - Using Jest

In this lab, you will learn how to get started writing and running tests with Jest.

We'll start with a simple test of one of our stateless components, `PollSubmitButton`.

Jest looks for tests inside any folder named **__tests__** or that are named with **.spec.js** or **.test.js** by default.

- ☐ 1. Create a directory named __tests__ inside src/components.
- ☐ 2. Create a file named **PollSubmitButton.test.js** inside `/components/__tests__`
- ☐ 3. Inside PollsubmitButton.test.js, enter the following:

```
import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-dom/test-utils';
import TestWrapper from '../TestWrapper';

describe('Poll Submit Button', function() {



    it('renders without a problem', function () {
        const PollSubmitButton = require('../PollSubmitButton')
            .default;
        var pollsubmitbutton = TestUtils
            .renderIntoDocument(<TestWrapper><PollSubmitButton
/></TestWrapper>);

        var buttonText =
ReactDOM.findDOMNode(pollsubmitbutton).textContent;

        expect(buttonText).toEqual('Go!');
    });

});
```

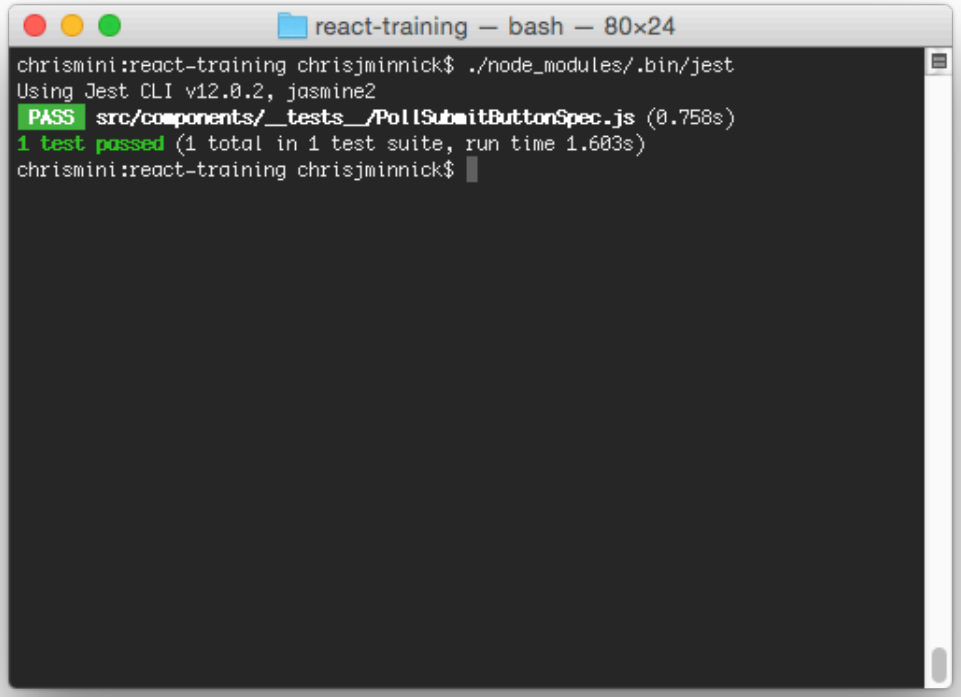
- ☐ 4. Install jest, babel-jest, and react-test-renderer

```
npm install --save-dev jest jest-cli babel-jest react-
test-renderer
```

- ☐ 5. Enter `./node_modules/.bin/jest` in the command line.

Your test will run and should return PASS. If they don't, see if you can make them pass.

Note: If your test doesn't pass, you may need to change PollSubmitButton back to a class from a functional component to make it work with the react-test-utils.



☐ 6. Create a new jest script in package.json

```
"jest":"jest"
```

☐ 7. Test your new script

```
npm run jest
```

☐ 8. Create the following new spec in PollSubmitButton.test.js:

```
it('calls handler function on click', function () {

    const PollSubmitButton = require('../PollSubmitButton').default;

    const handleClick = jest.fn();

    const pollsubmitbutton = TestUtils
        .renderIntoDocument(
            <TestWrapper><PollSubmitButton
                question={0}
                handleClick={handleClick}
            /></TestWrapper>);

    const buttonInstance = ReactDOM.findDOMNode(pollsubmitbutton);
```

```
    TestUtils.Simulate.click(buttonInstance);

    expect(handleClick).toHaveBeenCalled();

    const numberOfCallsMadeIntoMockFunction = handleClick.mock.calls.length;

    expect(numberOfCallsMadeIntoMockFunction).toBe(1);
});
```

☐ 9. Run **`npm run jest`**.

☐ 10. If the test fails, write the code to make the test pass.

Hint: If you don't have it already (from lab 22) add an onClick event handler to the PollSubmitButton component and have it call a function called handleClick.

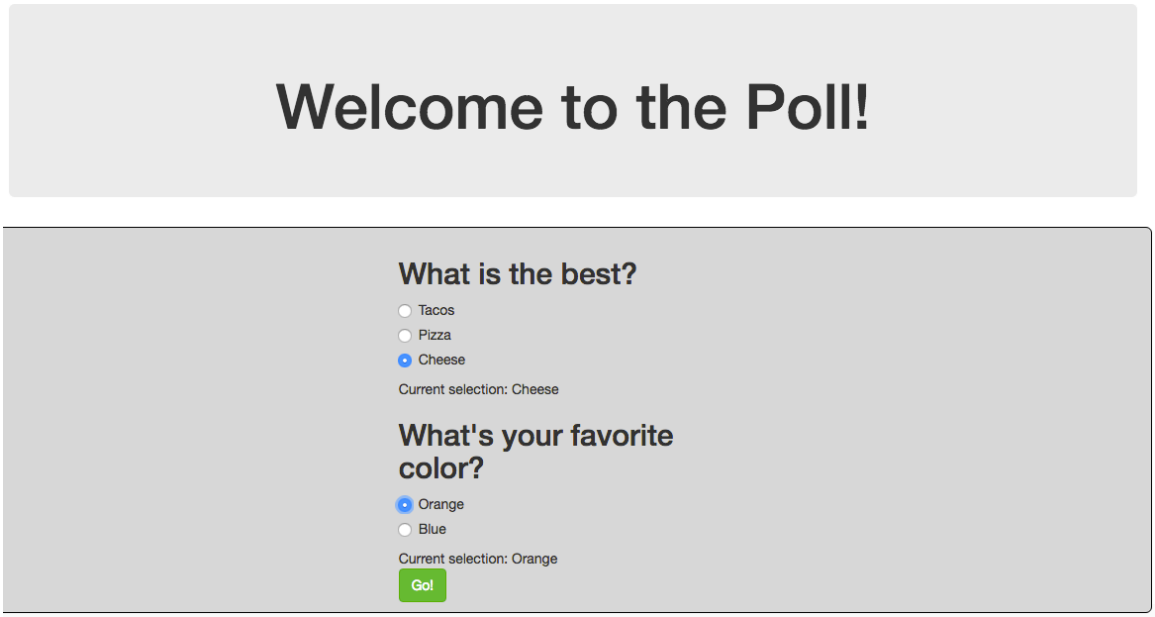
☐ 11. Test to confirm that the test passes.

☐ 12. Rewrite the tests using Enzyme.

Lab 25 - Multiple Components

In this lab, you'll modify the app to display all of questions in the JSON file and track the checked value for each button group.

The result will look like this:



Can you figure out how to modify the script? Hint: look at how the radio button group component is composed.

Try to work it out yourself. But, if you get stuck, check out the answer files at:

https://github.com/watzthisco/tdd-react-labs-v3.x/tree/master/solutions/lab25

Lab 26 - React Router v3.x

In this lab, you'll use React Router to change the UI based on the URL.

NOTE: These instructions apply to version 3.0.0 of React Router. If you're using React Router v4.x, skip to Lab 27. If you have a newer version installed, use `npm uninstall react-router --save` to uninstall it and then install version 3.0.0 using `npm install --save react-router@3.0.0`

☐ 1. Install React Router

```
npm install --save react-router@3.2.0
```

☐ 2. Open scripts/app.js and import Router, Route, and hashHistory from react-router

```
import {Router, Route, hashHistory } from 'react-
router';
```

☐ 3. Render a router instead of PollContainer

```
ReactDOM.render((
      <Router history={hashHistory}>
            <Route path="/" component={App} />
      </Router>),
      document.getElementById('app')
);
```

☐ 4. Create a new component in the containers folder called App, which renders a nav bar and this.props.children.

```
import React from 'react';

class App extends React.Component {
     render(){
         return(
             <div>
                 <ul className = "nav nav-pills">
                     <li role="presentation">Home</li>
```

```
                    <li role="presentation">Poll</li>
                    <li role="presentation">About Us</li>
                </ul>
                <div>
                    {this.props.children}
                </div>
            </div>
        );
    }
}

export default App;
```

- ☐ 5. Import App into scripts/app.js

```
import App from '../containers/App.js';
```

- ☐ 6. Build and test.
- ☐ 7. Import Link into App.js

```
import {Link} from 'react-router';
```

- ☐ 8. Create links to new routes in the nav bar.

```
<li role="presentation"><Link to="/">Home</Link></li>
<li role="presentation"><Link to="poll">Poll</Link></li>
<li role="presentation"><Link to="about">About Us</Link></li>
```

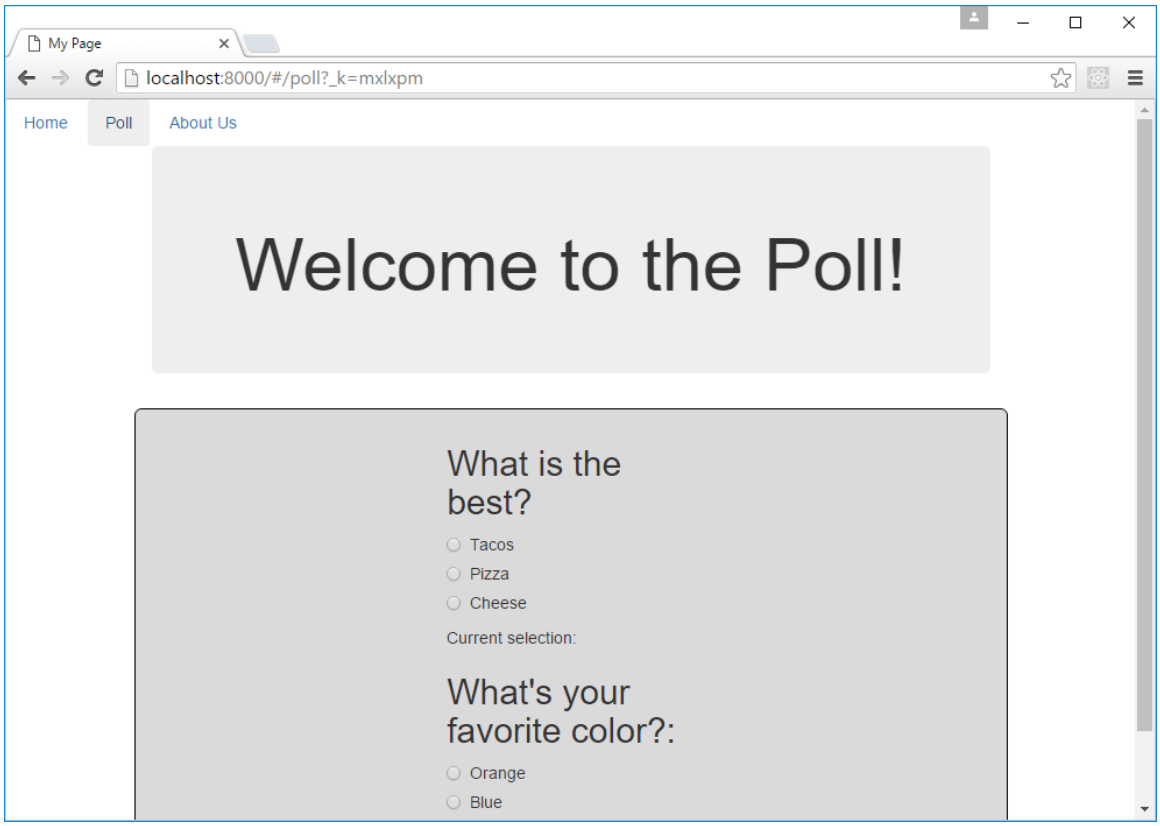
- ☐ 9. Create the new route for PollContainer in scripts/app.js, nested inside the App route

```
<Router history={hashHistory}>
    <Route path="/" component={App}>
        <Route path="/poll" component={PollContainer} />
    </Route>
</Router>
```

- ☐ 10. Create a component called AboutUs inside the components directory and import it into scripts/app.js

```
class AboutUs extends React.Component {
    render() {
        return(<h1>About Us</h1>);
    }
}
export default AboutUs;
```

☐ 11. Test and Build



Lab 27 - React Router v4.x

React Router 4 simplified and changed the way that routes are created. In this lab, you'll use React Router 4.x to do the same thing you did in lab 26.

☐ 1. If you installed version 3.x of React Router in the previous lab, uninstall it.

```
npm uninstall --save react-router
```

☐ 2. Install the latest version of **react-router** and **react-router-dom**

```
npm install --save react-router react-router-dom
```

☐ 3. Open **scripts/app.js** and import BrowserRouter from react-router-dom

```
import {BrowserRouter} from 'react-router-dom';
```

☐ 4. Render a new component (we'll create it in the next step) instead of PollContainer

```
ReactDOM.render((
      <BrowserRouter>
            <App/>
      </BrowserRouter>),
      document.getElementById('app')
);
```

☐ 5. Create a new component in the containers folder called App, which renders a nav bar which uses the Link component to link to paths and contains the routes.

```
import React from 'react';

class App extends React.Component {
     render(){
         return(
             <div>
    <ul className = "nav nav-pills">
```

```
        <li><Link to="/">Home</Link></li>
        <li><Link to="/poll">Poll</Link></li>
        <li><Link to="/about">About Us</Link></li>
    </ul>
    <div>
        <Route path="/poll" component={PollContainer}/>
        <Route path="/about" component={AboutUs}/>
    </div>
</div>
         );
     }
 }

export default App;
```

- ☐ 6. Import App into **scripts/app.js**

```
import App from '../containers/App.js';
```

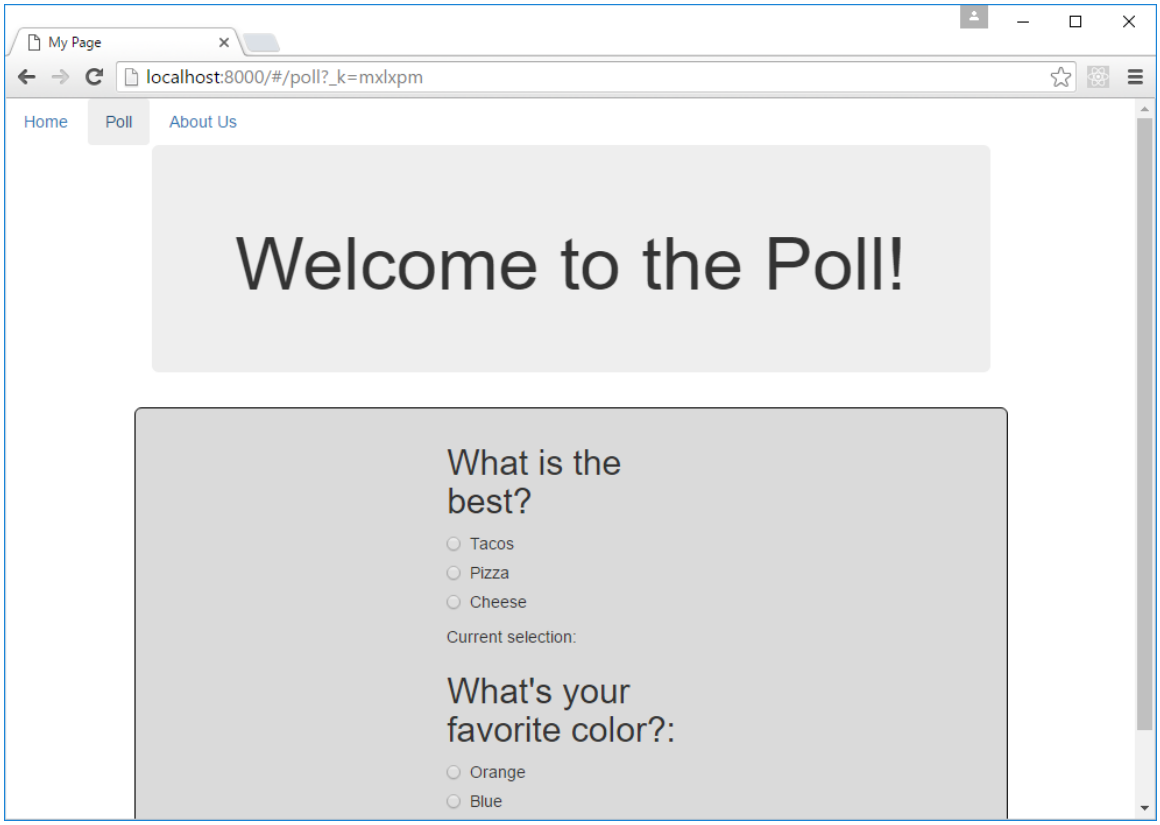
- ☐ 7. Import Link and Route into **App.js**

```
import {Link,Route} from 'react-router-dom';
```

- ☐ 8. Create a component called AboutUs inside the components directory and import it into **App.js**

```
class AboutUs extends React.Component {
      render() {
            return(<h1>About Us</h1>);
      }
}
export default AboutUs;
```

- ☐ 9. Test and Build


My Page
localhost:8000/#/poll?_k=mxlxpm
Home
Poll
About Us
Welcome to the Poll!
What is the best?
Tacos
Pizza
Cheese
Current selection:
What's your favorite color?:
Orange
Blue

Lab 28 - Redux Thermometer

In this lab, you'll get more practice with Redux.

Starting with the Redux counter example app (which is in the 'counter' folder inside the Redux examples that come with Redux, convert it into a thermometer / thermostat app with a graphical output.

☐ 1. Clone Redux into a new directory on your computer.

```
git clone https://github.com/reduxjs/redux.git
```

☐ 2. Navigate to the 'counter' example inside the examples folder.

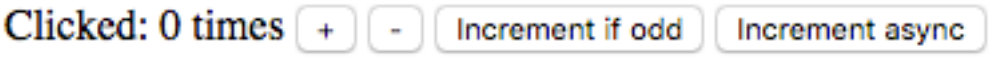
```
cd redux/examples/counter
```

☐ 3. Make a copy of the counter example and name it **thermometer**.

```
cp -R ../counter ../thermometer
```

☐ 4. Install and build the app to see how it works.

```
cd ../thermometer
npm install
npm start
open http://localhost:3000
```



☐ 5. Stop the counter example by pressing Control - C.

☐ 6. Update the tests for a thermometer application. The thermometer will work the same as the counter, but it will display a graphical bar (representing the mercury of a thermometer) that gets larger or smaller depending on whether the + or - button is clicked, as shown below.

☐ 7. Modify the application to make the tests pass and to make the thermometer work.

Current Temp: 97 degrees

+ - Increase if odd Increase async



Lab 29 - Redux

In this lab, you'll convert the Poll application to use Redux. We'll also make several changes to improve and simplify the app.

☐ 1. Install Redux and the bindings for React.

```
npm install --save redux react-redux
```

☐ 2. Create a new file named `store.js` inside of scripts
☐ 3. Import `createStore` from redux inside of `store.js`.

```
import {createStore} from 'redux';
```

☐ 4. Create a new folder named `reducers` and a file inside of it named `index.js`.

This will be the 'root reducer'. We'll write the reducers in a moment.

☐ 5. Import the root reducer into `store.js`.

```
import rootReducer from '../reducers/index';
```

☐ 6. Next, we'll simplify things a bit by moving our question data into a module, rather than fetching it with jQuery. Create a new file inside the data directory named questions.js, copy the questions array from the `data.json` file, and assign it to a new variable named `questions`. Make sure to export `questions` at the end of the file.

```
const questions = [
{
  "question": "What is the best?",
  "choices": [
    {"value": "Tacos", "label": "Tacos"},
    {"value": "Pizza", "label": "Pizza"},
    {"value": "Cheese", "label": "Cheese"}
  ],
  "correctAnswer": "Pizza"
},
{
  "question": "What's your favorite color?:",
  "choices": [
    {"value": "Orange", "label": "Orange"},
    {"value": "Blue", "label": "Blue"}
  ],
```

```
  "correctAnswer": "Blue"
}
];

export default questions;
```

☐ 7. Inside store.js, import the questions.

```
import questions from '../data/questions.js';
```

☐ 8. Inside store.js, create an object for the default data.

```
const defaultState = {
    questions,
    checkedValue: new Array(questions.length)
};
```

☐ 9. Inside store.js, create the store.

```
const store = createStore(rootReducer, defaultState);
```

☐ 10. Export the store.

```
export default store;
```

☐ 11. Create a new folder in `src` named `actions` and a file inside it named `actionCreators.js`

For now, we only have one thing that can happen inside our application, `selectAnswer`.

☐ 12. Create a new action creator inside `actionCreators.js`

```
export function selectAnswer(index,value){
     return {
          type: 'SELECT_ANSWER',
          index,
          value
     };
}
```

- ☐ 13. Inside of the `reducers` folder, we'll need to make a reducer for each piece of state, namely the questions and the `checkedValue`. Create a file inside `/reducers` called `questions.js` and one called `checkedValue.js`.
- ☐ 14. Inside `questions.js`, create a function to take in the action and the current state and return the new state. For now, we'll just log the state and the action.

```
function questions(state = [], action) {
  console.log(state,action);
  return state;
}

export default questions;
```

- ☐ 15. Make another reducer for the `checkedValue` inside `checkedValue.js`

```
function checkedValue(state = [], action) {
  console.log(state,action);
  return state;
}

export default checkedValue;
```

- ☐ 16. Write your root reducer in `reducers/index.js`

```
import { combineReducers } from 'redux';

import questions from './questions';
import checkedValue from './checkedValue';

const rootReducer = combineReducers({questions,checkedValue});

export default rootReducer;
```

- ☐ 17. Import your store into app.js.

```
import store from './store';
```

- ☐ 18. Inside `scripts/app.js`, remove the BrowserRouter (integrating redux and react-router adds another level of complexity that we don't need right now!). Render App in the return statement and pass it a store prop that's equal to `store` (the one we created and just imported).

```
  <App store={store} />
```

Now we can start using the state and action creators from our store inside our application. Whereas in normal React, we would need to pass the state down from the component where it lives using props, in React-Redux we can use Connect to inject the props at the level where we need them.

- ☐ 19. Delete the current contents of `containers/App.js`.
- ☐ 20. Import bindActionCreators from redux into App.js
- ☐ 21. Import connect from react-redux.
- ☐ 22. Import * as actionCreators from '../actions/actionCreators'.
- ☐ 23. Import PollContainer from './PollContainer;

At this point, `App.js` should look like this:

```
import {bindActionCreators} from 'redux';
import {connect} from 'react-redux';
import * as actionCreators from '../actions/actionCreators';
import PollContainer from './PollContainer;
```

- ☐ 24. Create the App component, using `connect()`, and immediately call it against PollContainer.

```
const App = connect(mapStateToProps,
mapDispatchToProps)(PollContainer);
```

- ☐ 25. Above the `const` you just created, create the `mapStateToProps` function, which will allow you to use `this.props.questions` and `this.props.checkedValue` in your components.

```
function mapStateToProps(state) {
  return {
    questions: state.questions,
    checkedValue: state.checkedValue
  }
}
```

- ☐ 26. Write the `mapDispatchToProps` function, which will allow you to use dispatch in your components in response to events.

```
function mapDispatchToProps(dispatch) {
  return bindActionCreators(actionCreators, dispatch);
}
```

☐ 27. Export the App component.

```
export default App;
```

☐ 28. Install the Redux developer tools in your Chrome browser by going to the Chrome Web Store here:

```
https://chrome.google.com/webstore/detail/redux-devtools/
```

☐ 29. Enable the Redux DevTools in your browser by adding a third parameter to your createStore method in **`scripts/store.js`**:

```
const store = createStore(rootReducer, defaultState,
                window.__REDUX_DEVTOOLS_EXTENSION__ &&
                window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Next, we'll update the PollContainer to display the questions.

☐ 30. To clean things up, and because we're no longer getting the data with jQuery, remove the jQuery import statement and the code that was using jQuery for the Ajax.

☐ 31. Inside the render method, set the value of the `questionsArray` to `this.props.questions`

☐ 32. Change the value of the `checkedValue` prop passed into `<RadioButtonGroup>` to `this.props.checkedValue[questionNumber]`

☐ 33. Change the value of the `onChange` prop in `<RadioButtonGroup>` to `this.props.selectAnswer`.

This will cause the `selectAnswer` action creator to fire when a radio button is changed.

Important: When an action is dispatched, every reducer will run. It's up to the reducer to decide whether to act on any particular action. You can see this in action by typing $r.store.dispatch({type: 'SELECT_ANSWER', index:0, value:"Cheese"}) into the console.

☐ 34. Pass a static value into `PollHeader`, rather than worrying about making that dynamic for now.

```
<PollHeader text="Welcome to the Poll!" />
```

The last step is to finish the reducers so that they mutate the state and return their slices of the state, which will be combined by the root reducer.

- ☐ 35. Open reducers/checkedValue.js and write a switch statement to check the action.type value for the 'SELECT_ANSWER' action.

```
switch (action.type) {
  case 'SELECT_ANSWER':
```

- ☐ 36. When it hears the SELECT_ANSWER action, it should return the state with the new value inserted in the appropriate place. Here's how to do that.

```
   switch (action.type) {
     case 'SELECT_ANSWER':
       return state
       .slice(0,action.index)
       .concat([action.value])
       .concat(state.slice(action.index+1));
```

- ☐ 37. Write a default case which will run when the action type isn't `SELECT_ANSWER`. It should just return the state.

```
    default:
      return state;
  }
}
```

- ☐ 38. Add another question to the questions array and confirm that the application still works.

Challenge Steps:

- ☐ 1. Try modifying questions.js so that it changes periodically.
- ☐ 2. Finish the questions reducer.

Lab 30 - SwimCalc

In this lab, you'll build a React application from scratch. You may choose to use Redux or not for this project. Or, start out not using Redux, and then convert it to use Redux.

The Story

Linda is a distance swimmer. Each month, she buys a lap swim pass from the city Department of Parks and Recreation that gets her 20 entries to the pool and is only good for one month.

The current cost of the pass is $50.

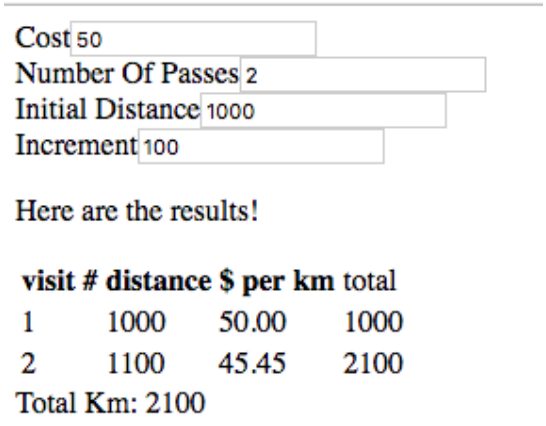
The first time she swims each month, she swims 1 kilometer (1000 meters). She increases her distance by 100 meters each time she swims during the month

Build an app that will tell Linda:

- How far she will have swum if she swims 20 times
- What is her price per kilometer swum
- What do the numbers look like if any of the variables in the equation change: -- Price for the lap swim pass -- Number of times she uses the pass in a month -- Starting distance -- Daily increase in distance

Getting Started

The finished project might look something like this:



Starter Project: https://github.com/watzthisco/tdd-react-labs-v3.x/tree/lab30
Example solutions:
https://github.com/watzthisco/tdd-react-labs-v3.x/tree/lab30solutions