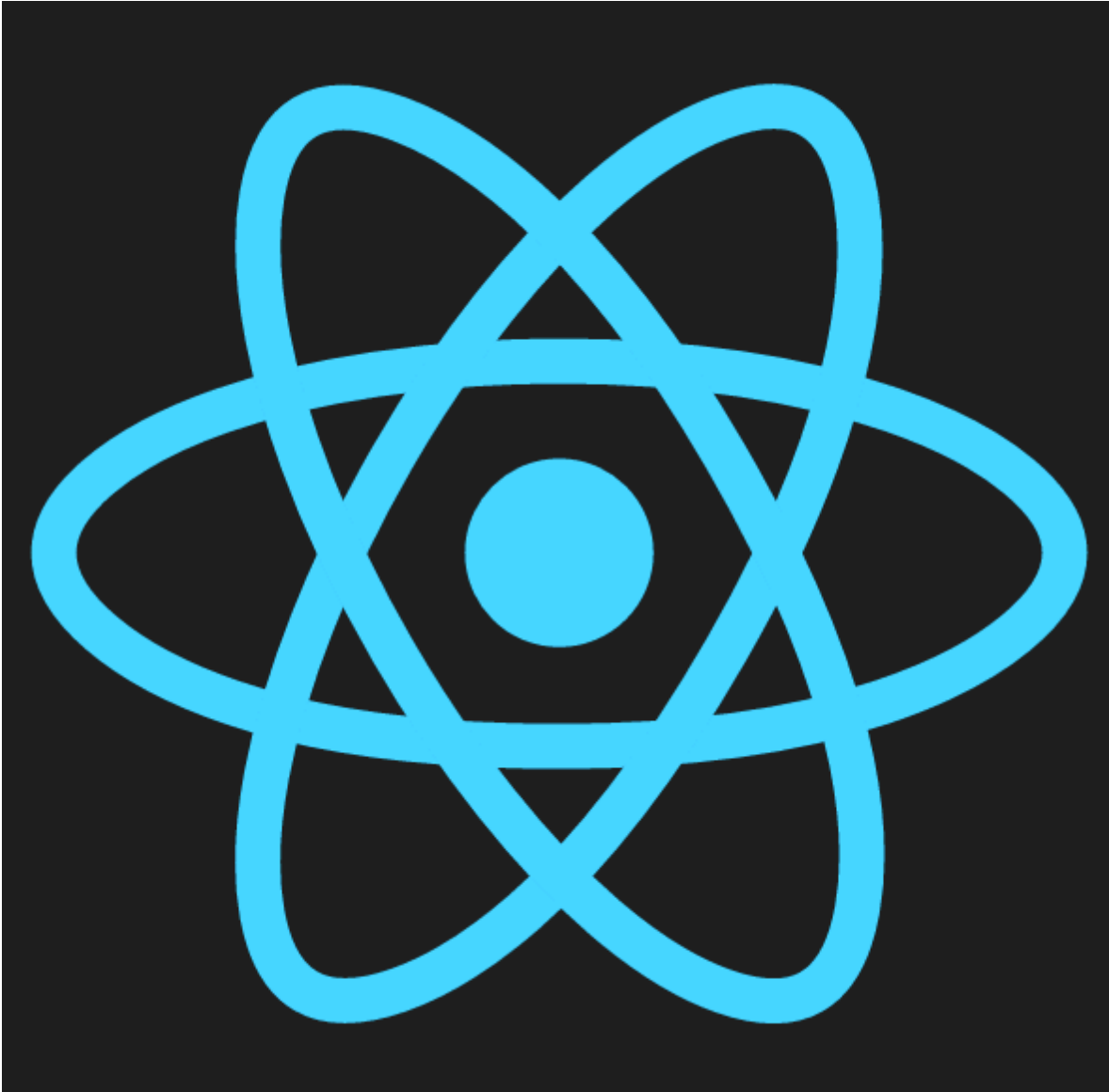


Test-Driven Development Using React.js and ES6 (ES2015) - Student Labs



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

<https://github.com/watzthisco/tdd-react-es6-labs.git>

Version 1.8, April 2017
by Chris Minnick
Copyright 2017, 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 the address below.

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

Help us improve our courseware

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

Credits

About the Author

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

Since 2001, Minnick has trained thousands of Web and mobile developers. His current online courses, Creating Mobile Apps with HTML5 and Achieving Top Search Engine Placements are consistently among the most popular courses offered by online training provider Ed2Go.com.

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

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

In addition to his role with WatzThis?, Chris is a winemaker, a contributor to several blogs (including chrisminnick.com), and an avid swimmer, cook, and musician.

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 1 to 15 - Alternative (skip ahead) Version	9
Lab 1 - Installing and Configuring WebStorm	10
Part 1 - Installing WebStorm.....	10
Part 2: Creating a New Project	10
Part 3: Configuring WebStorm for ES2015 and JSX	11
Lab 2 - Getting Started with Node.js	12
Part 1: Installing Node.js.....	12
Part 2: Getting to Know Node.js.....	12
Part 3: Using npm.....	15
Lab 3 - Version Control with GIT.....	17
Part 1: Installing Git and Creating a Repository	17
Part 2: Learn the Ways of Git	20
Lab 4 - Initialize npm.....	23
Lab 5 - Set Up a Task Runner	24
Lab 6 - Managing External Dependencies	27
Part 1: Create a "version" task.....	27
Part 2: Adding dependent tasks.....	28
Lab 7 - Automate Linting.....	30
Part 1: Installing JSHint and Command Line Linting.....	30
Part 2: Creating a "lint" Task.....	30
Part 3: Setting JSHint Options.....	34
Lab 7 (alternate): ESLint.....	36
Lab 8 - Configure a Web Server	40
Part 1: Install gulp-webserver.....	40
Part 2: Manual browser testing	41
Lab 9 - Getting Started with Jasmine	43

Lab 10 - Integrate Jasmine into Your Build	49
Lab 11 - In-browser Testing with Karma	52
Part 1: Installing and Configuring Karma.....	52
Part 2: Automating Karma	56
Lab 12 - Deploy with Webpack.....	58
Lab 13 - README update and Refactoring.....	65
Lab 14 - Babel	66
Lab 15 - Converting to ES6	68
Lab 16 - Hello, React	70
Part 1: Say hello and test your setup	70
Part 2: Make a component	73
Part 3: Convert to ES6.....	74
Part 4: Configure ESLint and Karma for React	75
Lab 17 - Breaking up a UI into Components.....	77
Part 2: Bundle tests for Karma.....	80
Lab 18 - State and Props	81
Lab 19 - Adding Style to React Components	86
Lab 20 - Controlling the Form.....	88
Lab 21 - Refactoring and Using JSON Data.....	93
Lab 22 - Life Cycle and Events	96
Lab 23 - PropTypes	99
Lab 24 - Using Jest	100
Lab 25 - Multiple Components	103
Lab 26 - React Router.....	104
Lab 27 - React Router Improvements	107
Lab 28 - Redux Thermometer.....	109
Lab 29 - Redux	111
Lab 30 - SwimCalc	120
The Story.....	120
Getting Started	120

Setup Instructions

Course Requirements

In order to complete the labs in this course, you will need:

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

Classroom Setup

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

1. Install node.js on each student's computer. Go to nodejs.org and click the link to download the latest version from the 6.x branch.
2. Install a code editor. We use WebStorm in the course. A 30-day trial version is available from <http://www.jetbrains.com/webstorm>.
3. Make sure Google Chrome is installed.
4. Install git on each student's computer. Git can be downloaded from <http://git-scm.com>. Select all of the default options during installation.
5. Install gulp 4 on each computer by running the following commands in Terminal on MacOS or the Windows Command Prompt.

```
# uninstall previous Gulp installation, if any
$ npm uninstall gulp -g
```

```
# install the latest Gulp 4 CLI tools globally
$ npm install gulpjs/gulp-cli -g
```

Testing the Setup

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

```
git clone https://github.com/watzthisco/tdd-react-es6-labs.git
```

The lab solution files for the course will download into a new directory called `tdd-react-es6-labs`.

- Enter `cd tdd-react-es6-labs` to switch to the new directory.
- Upgrade npm by running this:

```
npm install -g npm
```

- Enter `npm install` (add `--no-optional` on Windows)

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`.

- 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 and tell you that the test passed.

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-es6-labs
```

Each lab in the course has a separate branch containing the finished lab. These 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 using the following commands (to be run in your shell application).

1. Clone the entire repository (note: this only has to be done once).

```
git clone https://github.com/watzthisco/tdd-react-es6-labs
```

2. Change directories into the newly cloned repository.

```
cd tdd-react-es6-labs
```

3. Check out the lab you want to work with.

```
git checkout lab08
```

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

For example, if you want to start with Lab 20, simply check out Lab19, which will contain the project as it should exist at the beginning of the instructions for Lab 20.

Lab 1 to 15 - Alternative (skip ahead) Version

The first 15 labs in this course cover front end web tooling and testing, including:

- git
- gulp
- node.js
- npm
- JSHint
- ESLint
- Chrome Developer Tools
- Jasmine
- Karma
- Webpack

If you're already familiar with these tools or if you want to just use them and learn as you go and come back to tooling at the end of the course, complete this lab to get your development environment set up for TDD with React and skip ahead to Lab 16.

- 1. Open a command shell and enter the following command to clone the repository:

```
git clone https://github.com/watzthisco/tdd-react-es6-labs
```

The completed code for all of the course's labs will download into a directory named `tdd-react-es6-labs`.

- 2. Create a new project in your code editor from this directory.
- 3. In your command shell, change to the `lab15` branch:

```
git checkout lab15
```

- 4. Install the project dependencies.

```
npm install
```

- 5. Open the `readme` file and test out the commands described in it.
- 6. If everything works, use these files as your starting point for the rest of the course. If there are any problems, check that you have the required dependencies installed correctly (gulp 4 and the latest version of node, in particular).

Lab 1 - Installing and Configuring WebStorm

WebStorm is an Integrated Development Environment for JavaScript. You can use any code editor or IDE you like, but we chose WebStorm for this course because of its 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, such as git and gulp with their debugging and unit testing tools.

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

After you install WebStorm, you will set up a very basic file template for ECMAScript 2015 (aka ES6, aka ES2015) since WebStorm doesn't have one built in at the time of this writing. We will be adding to this template and creating additional templates as necessary in future labs.

Part 1 - Installing WebStorm

WebStorm includes a 30-day trial license.

- 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

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



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

Part 3: Configuring WebStorm for ES2015 and JSX

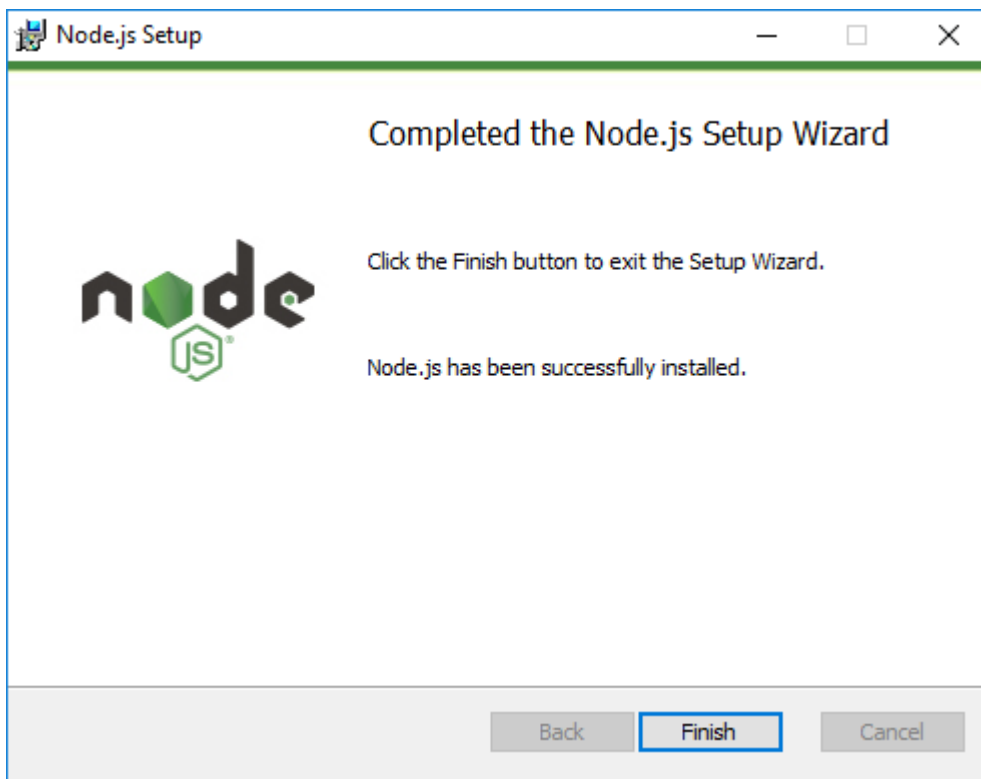
- ☐ 1. Select **WebStorm > Preferences** (on MacOS) or **File > Settings** (Windows) from the top menu.
- ☐ 2. Click **Languages & Frameworks** and choose **JavaScript**.
- ☐ 3. Select React JSX (or JSX Harmony in less current versions) from the JavaScript language version dropdown.
- ☐ 4. Click **OK**

Lab 2 - 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: Installing Node.js

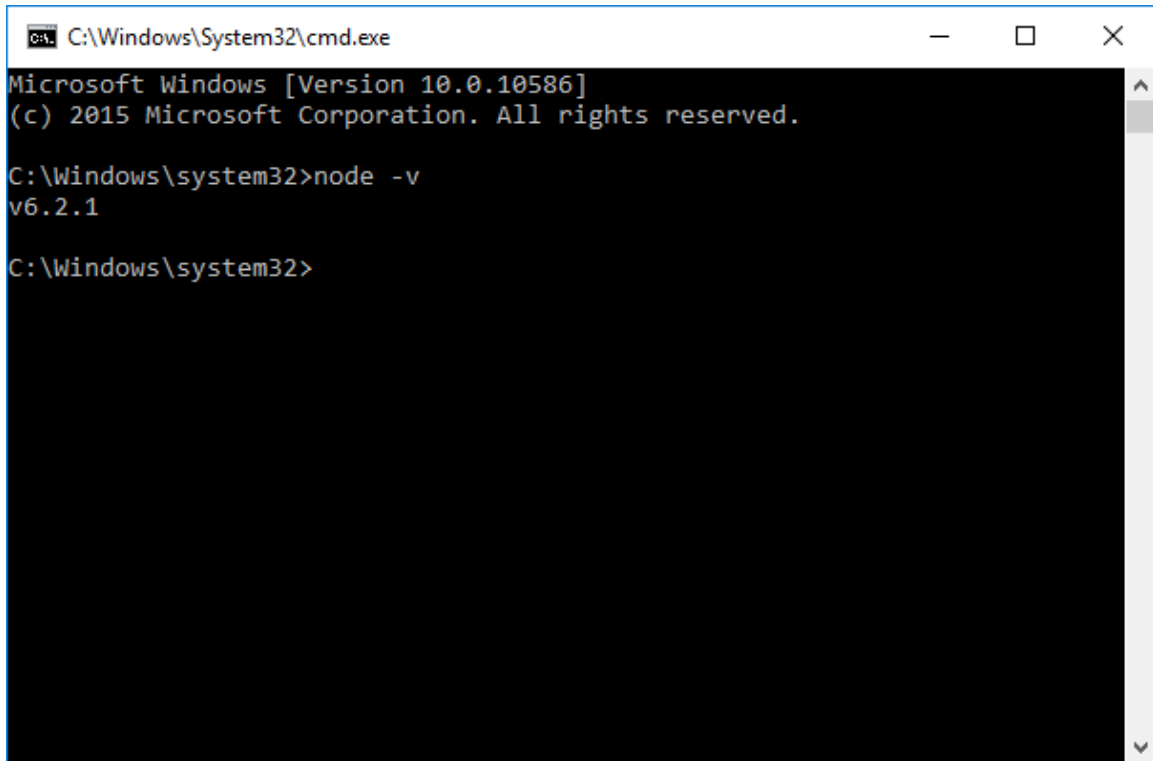
- ☐ 1. Go to <https://nodejs.org> and download the latest version of Node 6.x (6.9.2 as of version 1.5 of this courseware).
- ☐ 2. When it finishes downloading, launch the installer to install Node.js
- ☐ 3. Select all of the default options.



Part 2: Getting to Know Node.js

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

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



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

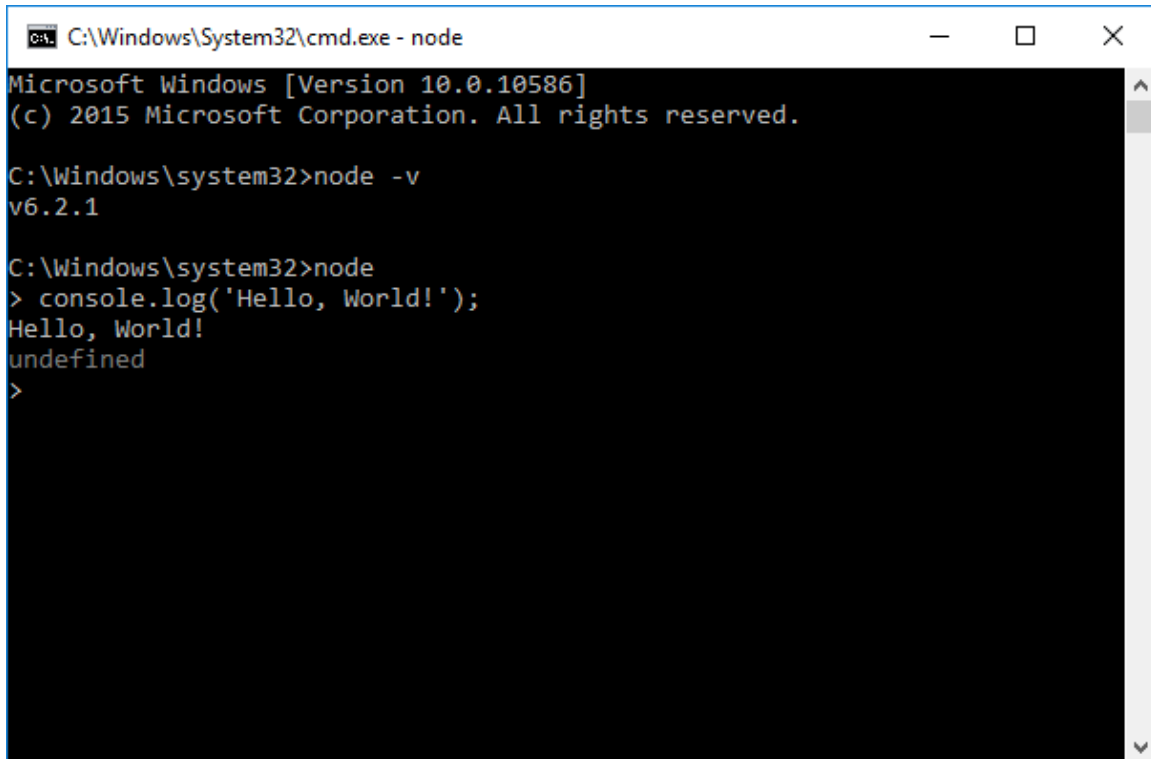
C:\Windows\system32>node -v
v6.2.1

C:\Windows\system32>
```

- 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 of the Node.js modules.

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



```
C:\Windows\System32\cmd.exe - node
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Windows\system32>node -v
v6.2.1

C:\Windows\system32>node
> console.log('Hello, World!');
Hello, World!
undefined
>
```

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.

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

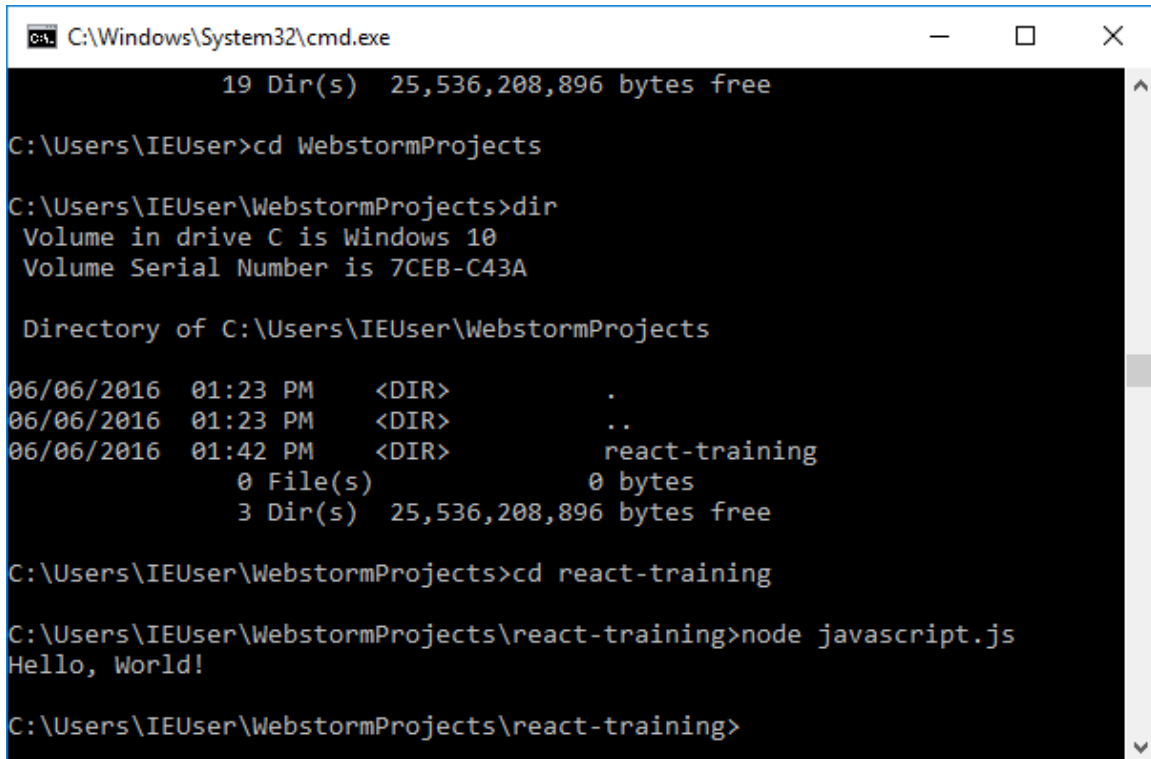
```
console.log('Hello, World!');
```

- 2. Save the file as **javascript.js**
- 3. Exit node's interactive shell by pressing **CTRL-C** twice.
- 4. In Terminal (MacOS) or the Command Prompt (Windows), 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).

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



```
C:\Windows\System32\cmd.exe

19 Dir(s) 25,536,208,896 bytes free

C:\Users\IEUser>cd WebstormProjects

C:\Users\IEUser\WebstormProjects>dir
Volume in drive C is Windows 10
Volume Serial Number is 7CEB-C43A

Directory of C:\Users\IEUser\WebstormProjects

06/06/2016  01:23 PM    <DIR>          .
06/06/2016  01:23 PM    <DIR>          ..
06/06/2016  01:42 PM    <DIR>          react-training
               0 File(s)                0 bytes
               3 Dir(s) 25,536,208,896 bytes free

C:\Users\IEUser\WebstormProjects>cd react-training

C:\Users\IEUser\WebstormProjects\react-training>node javascript.js
Hello, World!

C:\Users\IEUser\WebstormProjects\react-training>
```

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 of 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 3 - 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. 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: Installing Git and Creating a Repository

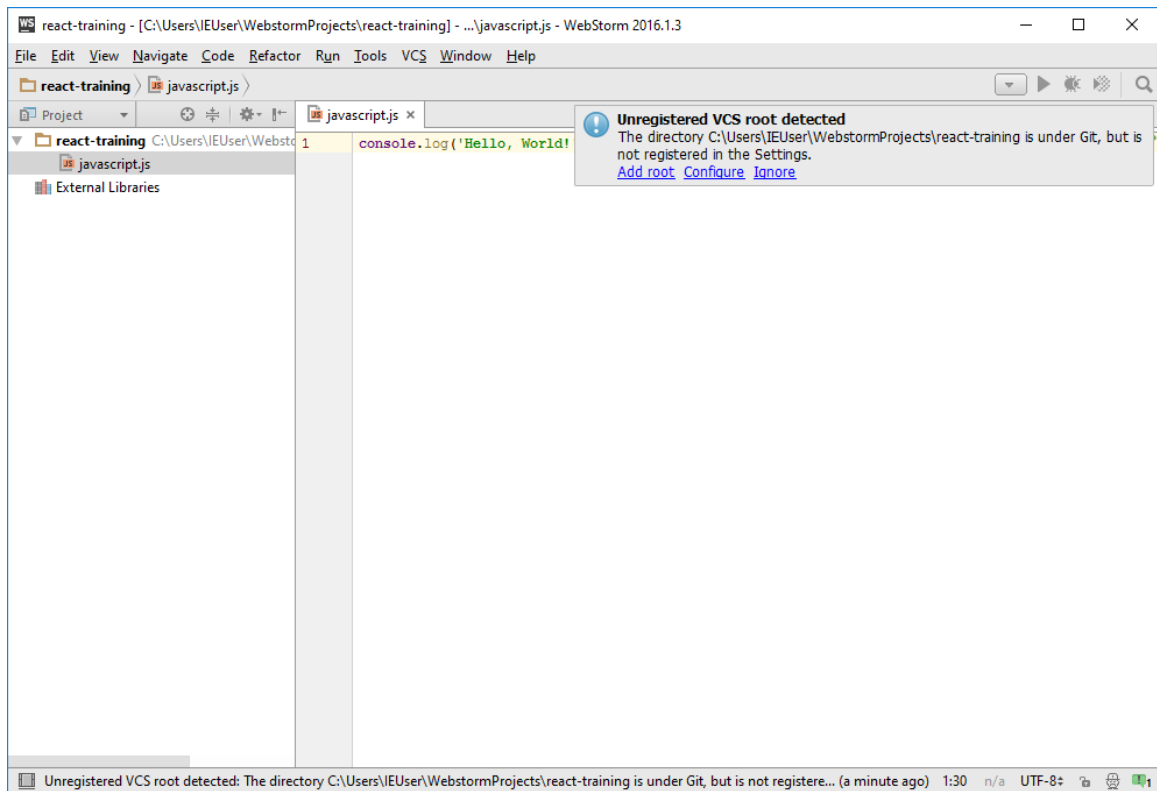
- 1. Go to <http://git-scm.com>
- 2. Download git and start the installation.
- 3. On Windows, select **Use Git from the Windows Command Prompt**. This will give you the option to either use the Windows Command Line or the git bash shell, which emulates a Unix environment.
- 4. Select **Check out Windows-style and commit Unix-style line endings**.
- 5. Select the default options for all other steps in the installation.

After you install Git, Windows users will have a shortcut to the git bash shell on the desktop. This is a Unix-like command prompt for Windows. We will be using this shell going forward in order to keep commands identical between MacOS and Windows computers. MacOS users should continue to use Terminal.

- 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 of 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 of 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 WebStorm, 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 in order for the new plugin to work.
- 9. Inside README.md, enter this basic structure, which you will fill in the details of later on:

```
# 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

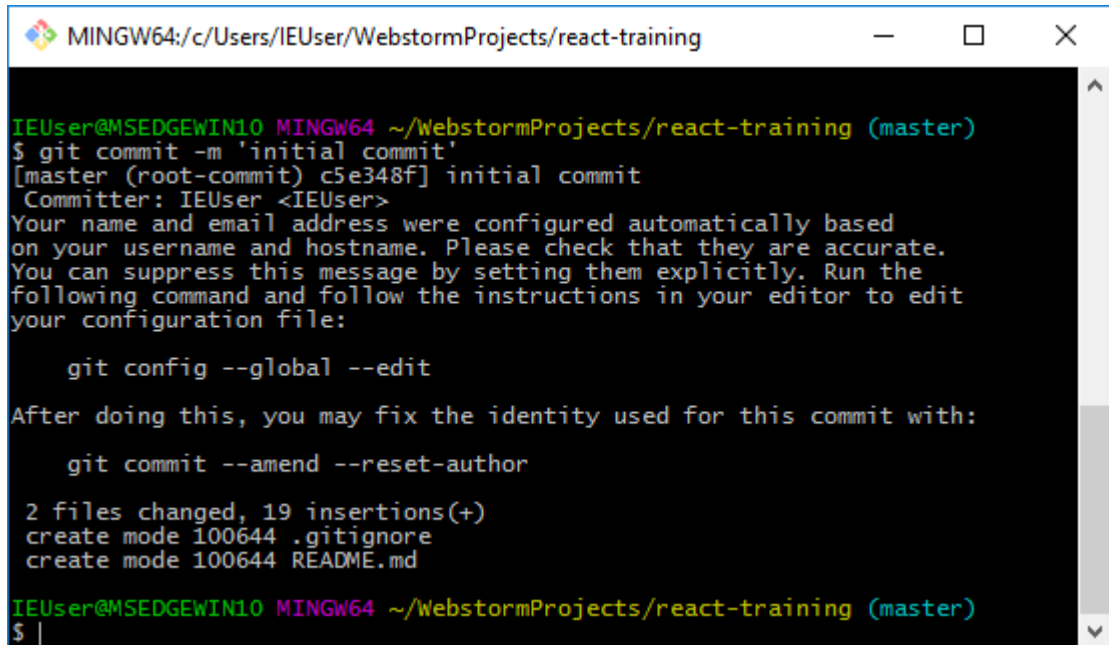
# WebStorm
.idea

# 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.

A screenshot of a terminal window titled 'MINGW64:/c/Users/IEUser/WebstormProjects/react-training'. The terminal shows the execution of 'git commit -m 'initial commit'', which results in a commit with hash 'c5e348f'. It then prompts the user to edit their configuration file using 'git config --global --edit'. After this, it shows 'git commit --amend --reset-author' being run, which updates the commit with two files: '.gitignore' and 'README.md'. The terminal ends with the prompt '\$ |' on the master branch.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$ git commit -m 'initial commit'
[master (root-commit) c5e348f] initial commit
Committer: IEUser <IEUser>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

2 files changed, 19 insertions(+)
create mode 100644 .gitignore
create mode 100644 README.md
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$ |
```

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 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. If you want to create a remote repository, you need to add an origin. The command for this is

```
git remote add origin [remote url].
```

If you have a github account, you can try this now. Make sure you first create the new repository at github.com, but don't initialize it with any files.

- 3. Once you've added an origin, you can push your changes to it with `git push`:

```
git push -u origin master
```

- 4. The `-u` flag tells Git to remember the parameters. You can simply enter `git push` the next time you push.
- 5. Use `git pull` to retrieve the latest code from the origin.

```
git pull origin master
```

- 6. 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.

- 7. Make the following changes to `README.md`

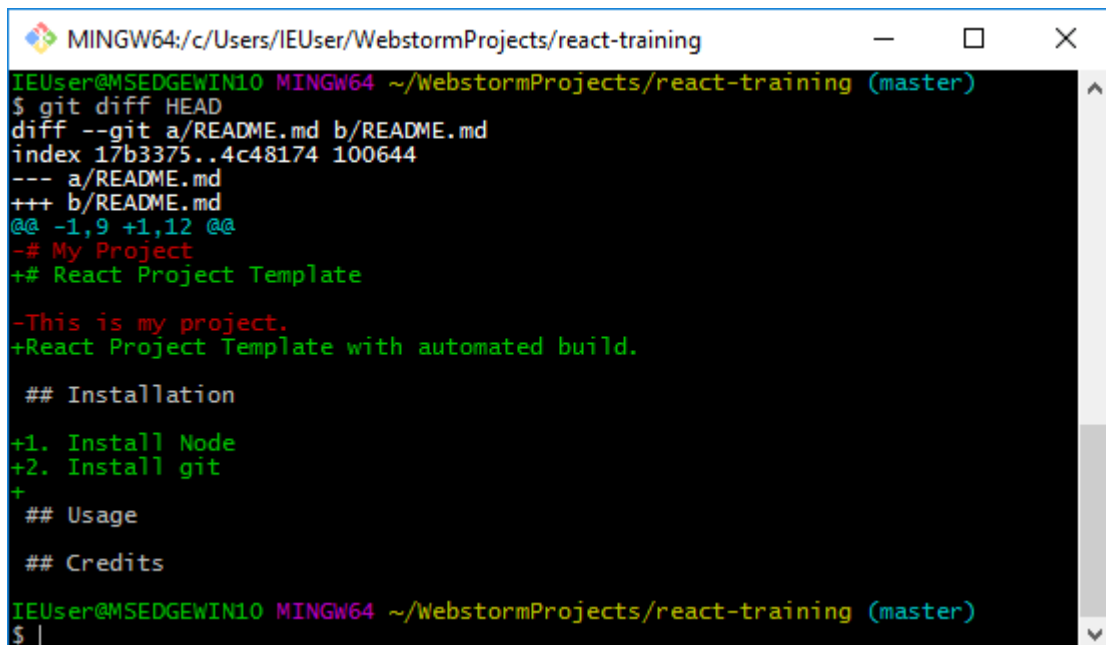
```
# React Project Template
```

```
React Project Template with automated build.
```

```
## Installation
```

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

- 8. Enter `git diff HEAD`
- 9. You will see a list of differences between the current state of your files and the last commit.

A screenshot of a terminal window titled 'MINGW64:/c/Users/IEUser/WebstormProjects/react-training'. The terminal shows the command '\$ git diff HEAD' and its output. The output displays the differences between the current state of the 'README.md' file and the last commit. The changes are highlighted with green text for additions and red text for deletions. The output shows that the file 'README.md' has been updated with new content, including a new title 'React Project Template', a description 'React Project Template with automated build.', and a new installation section. The terminal output is as follows:

```
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$ git diff HEAD
diff --git a/README.md b/README.md
index 17b3375..4c48174 100644
--- a/README.md
+++ b/README.md
@@ -1,9 +1,12 @@
-# My Project
+# React Project Template

-This is my project.
+React Project Template with automated build.

## Installation

+1. Install Node
+2. Install git
+
## Usage

## Credits

IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$ |
```

- 10. 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.
- 11. Use `git diff --staged` to see what you have staged.

- 12. Type the following to unstage app.js:

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

- 13. Use `git diff` and `git status` to see that you've unstaged the file.
- 14. Stage the file again.
- 15. 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
```

- 16. Commit your changes.

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

- 17. 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
```

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

```
git checkout my_first_branch
```

- 19. In the new branch, add this code to app.js

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

- 20. `git add` and `git commit` your changes. Remember to use a descriptive message.
- 21. Switch back to the master branch using this command:

```
git checkout master
```

- 22. Merge your changes from my_first_branch back into master:

```
git merge my_first_branch
```

- 23. Delete your branch:

```
git branch -d my_first_branch
```

Lab 4 - 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 in order 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 folder named `node_modules` will be created in your project.

- 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 5 - Set Up a Task Runner

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

There are many different task runners available. The one we'll be using is Gulp. As of this writing, the standard version of gulp in the npm repository is 3.x, but 4.x is widely used and contains features that we want to use. So, we will be forcing the installation of Gulp 4.0.

- 1. Install gulp globally.

```
npm install gulpjs/gulp-cli -g
```

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

- 2. Install gulp 4.0 in your local **node_modules** folder.

```
npm install --save-dev "gulpjs/gulp.git#4.0"
```

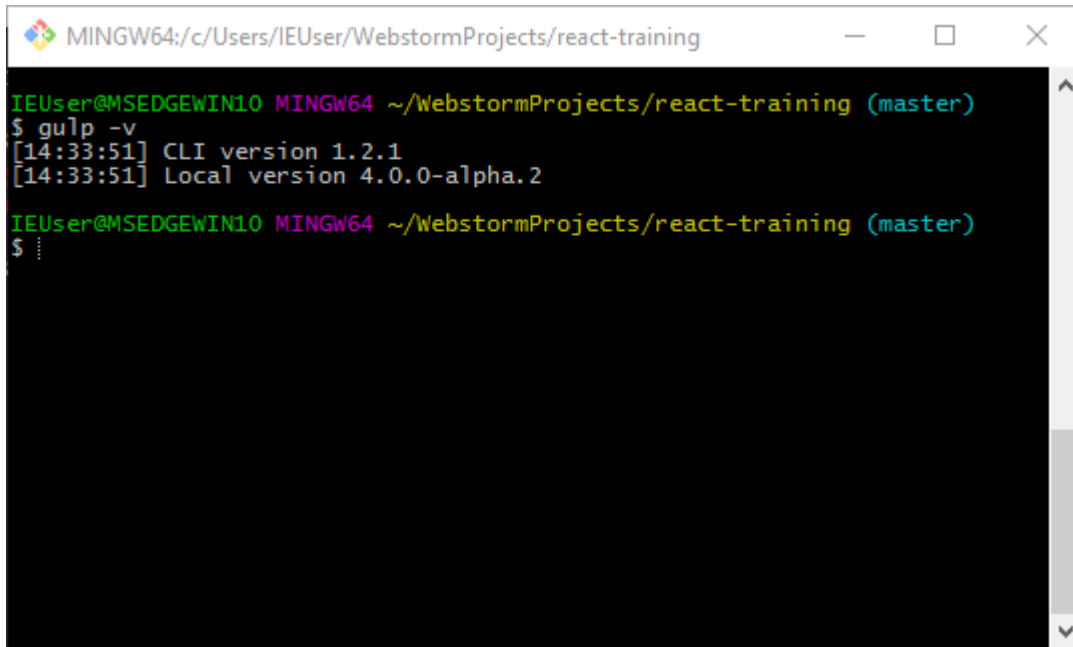
Gulp is the only package that we will be installing globally. We do this for convenience so that you can simply enter `gulp` in order to run it. If you are not able to install gulp globally, you can install the `gulp-cli` it locally (without `-g`) and run it by entering:

```
./node_modules/.bin/gulp
```

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

- 3. Confirm that you have Gulp 4 installed.

```
gulp -v
```


A terminal window titled 'MINGW64:/c/Users/IEUser/WebstormProjects/react-training' with standard window controls. The prompt is 'IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)'. The user enters '\$ gulp -v', and the output shows '[14:33:51] CLI version 1.2.1' and '[14:33:51] Local version 4.0.0-alpha.2'. The user then enters '\$' and the prompt changes to 'IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)'.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$ gulp -v
[14:33:51] CLI version 1.2.1
[14:33:51] Local version 4.0.0-alpha.2
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$
```

- 4. Add all the new files to Git.

```
git add .
```

- 5. Commit your changes.

```
git commit -m "installed gulp"
```

- 6. Create a new file in the root of your project named **gulpfile.js**
- 7. Create your default task in gulpfile.js by entering this code:

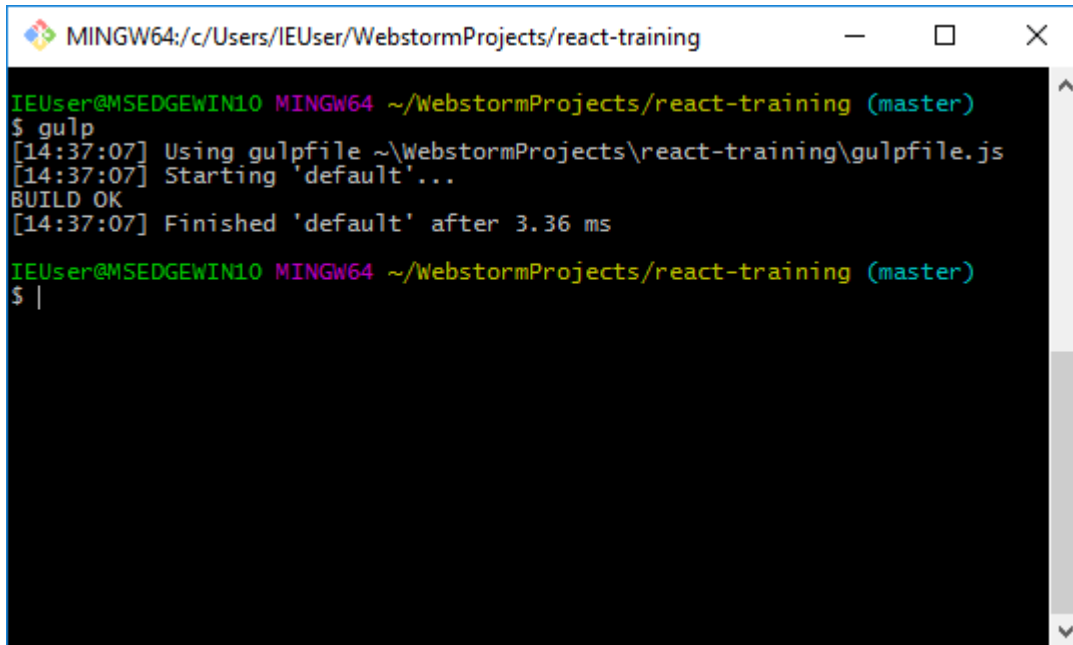
```
const gulp = require('gulp');

//default task
gulp.task('default', function(done) {
  console.log('BUILD OK');
  done();
});
```

Test your new (very simple) automated build by running gulp from the command line:

```
gulp
```

- 8. If everything is working correctly, you should get the following output:

A screenshot of a Windows terminal window titled "MINGW64:/c/Users/IEUser/WebstormProjects/react-training". The terminal shows a user running the command "gulp". The output indicates that gulp is using a local gulpfile, starting the default task, and finishing successfully after 3.36 ms. The prompt then returns to the user.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$ gulp
[14:37:07] Using gulpfile ~\WebstormProjects\react-training\gulpfile.js
[14:37:07] Starting 'default'...
BUILD OK
[14:37:07] Finished 'default' after 3.36 ms
IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (master)
$ |
```

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

```
gulp rm -g gulp
npm install -g gulpjs/gulp-cli
npm uninstall gulp --save-dev
npm install --save-dev "gulpjs/gulp.git#4.0"
```

□ 9. Update README.md

Usage
To build:

1. gulp

□ 10. git add . and git commit

□ 11. git status to confirm that everything is clean.

Lab 6 - Managing External Dependencies

In this lab, you will create a gulp task 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 empty task in your gulpfile, above the default task, called version.

Note: In Gulp 4, the order in which tasks appear in your gulpfile matters. Each of a task's dependencies must appear before the task that's dependent on them in the file.

- 2. Inside the version task, output a message to the console:

```
console.log('Checking node version: ');
```

- 3. Include `package.json` into the task, so that we can get the expected version from it and then create a new constant to hold the expected node version.

```
const packageJson = require('./package.json');
```

- 4. 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": "4.4.2"  
},  
...
```

- 5. Create a new constant in the version task in your gulpfile to hold the expected node version.

```
const expectedVersion = packageJson.engines.node;
```

- 6. Find out the actual version of node that's installed.

```
const actualVersion = process.version;
```

- 7. 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
```

- 8. Require `semver` at the top of `gulpfile.js`, under the `gulp` require.

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

- 9. Now you can use the `semver` package inside your tasks with the `semver` variable. Add the following `if` statement to your version task:

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

- 10. Gulp v4 requires every function or task to return a stream, promise, event emitter, child process, or observable. To fulfill this requirement, add "done" to the function parameters and then add `done()`; to the end of the function. Your finished task should look like this:

```
//check node version
gulp.task('version', function(done) {

  console.log('Checking node version: ');

  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);
  }
  done();
});
```

- 11. In the command line, enter `gulp version` to test it out.
- 12. If there are no errors, commit to git!

Part 2: Adding dependent tasks

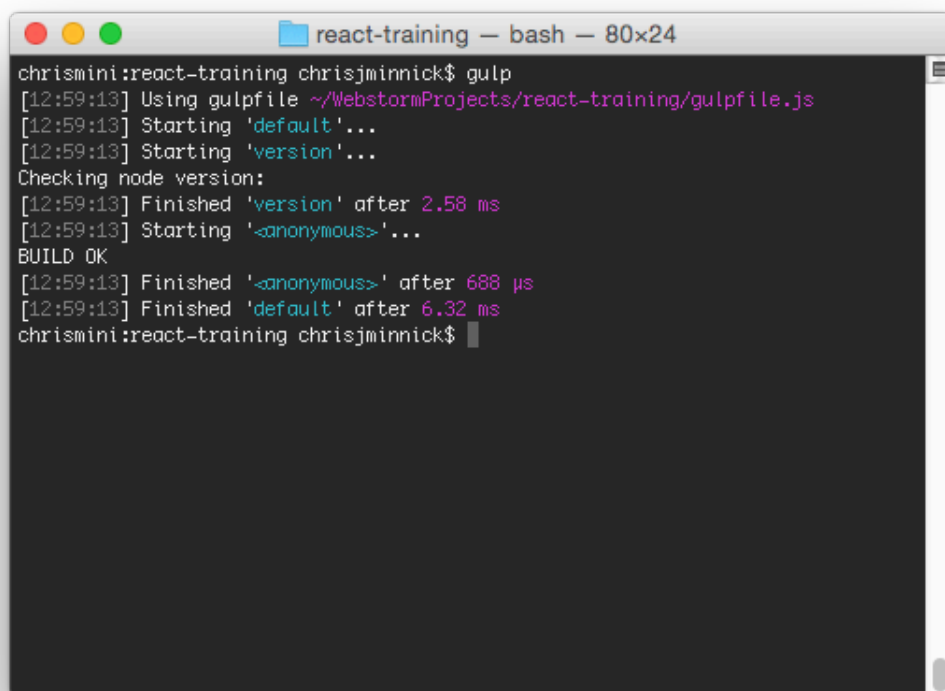
Each gulp task can take other tasks as dependencies. These tasks must run prior to the task that requires them. Dependent tasks are listed in the 2nd parameter of `gulp.task`, and each dependent task can be run either in series or in parallel.

If the order in which dependent tasks runs matters, use `gulp.series` and the next task will wait until the dependent task finishes before starting. If the order doesn't matter, use `gulp.parallel` to run things simultaneously, as fast as possible.

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

```
gulp.task('default', gulp.series('version', function(done){  
  ...  
}));
```

- 2. Enter `gulp` into the console to test it out. You should get the following result:

A terminal window titled "react-training — bash — 80x24" showing the execution of the 'gulp' command. The output shows the gulpfile being used, tasks starting and finishing in sequence, and the final 'BUILD OK' message.

```
chrismini:react-training chrisjminnick$ gulp  
[12:59:13] Using gulpfile ~/WebstormProjects/react-training/gulpfile.js  
[12:59:13] Starting 'default'...  
[12:59:13] Starting 'version'...  
Checking node version:  
[12:59:13] Finished 'version' after 2.58 ms  
[12:59:13] Starting '<anonymous>'...  
BUILD OK  
[12:59:13] Finished '<anonymous>' after 688 μs  
[12:59:13] Finished 'default' after 6.32 ms  
chrismini:react-training chrisjminnick$
```

- 3. `git add` and `git commit`

Lab 7 - 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 JSHint, use it to check a JavaScript file, and then build it into your automated build task.

Part 1: Installing JSHint and Command Line Linting

- 1. Install JSHint:

```
npm install --save-dev jshint
```

- 2. Install the gulp-jshint package:

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

- 3. Test JSHint by passing your gulpfile into it.

```
./node_modules/.bin/jshint gulpfile.js
```

If everything is installed correctly, you will get errors. These are due to the ES6 syntax we used in gulpfile.js. We'll fix that shortly!

Part 2: Creating a "lint" Task

In this part, you will create a new task to perform linting on all of your project's JavaScript files, and you will make that task a dependency of the default gulp task.

- 1. Open gulpfile.js and require gulp-jshint as a new const called jshint.

Note: Now that you're starting to have multiple require statements at the top of your gulpfile, this is a good time to reorder them so the variable names are in alphabetical order, except for gulp, which should always be first.

- 2. Create a new task (above default) called **lint**. Its anonymous function shouldn't take a parameter since we'll be returning the results of the linting from the function.
- 3. Start the task with the following console.log message:

```
console.log('Linting JavaScript: ');
```

- 4. On the next line, pass in the source files and directories that should be checked. You will do this using the `gulp.src` method, which takes an array as its parameter. We'll use wildcards (*) to specify that everything in the `src` folder should be checked as long as it has a `.js` extension.

```
return gulp.src(['gulpfile.js', 'src/**/*.js'])
```

- 5. Next, use `.pipe` to input the results of `gulp.src` into `jshint`.

```
.pipe(jshint({  
}))
```

- 6. The JavaScript object passed to `jshint` is where we can specify options for JSHint. There are several that we need to configure right away, and we'll look at some others in the next part.

```
.pipe(jshint({  
  esversion: 6,  
  node: true,  
  browser: true,  
  globals: []  
}))
```

The first option tells JSHint that we want to conform to ECMAScript 6 (aka ES2015). The last 3 set up the environment for the test. We're saying to include node's objects, web browser objects, and any globals in the array we created (currently empty).

- 7. Pipe the results from the linting into JSHint's reporter methods:

```
.pipe(jshint.reporter('default'))  
.pipe(jshint.reporter('fail'));
```

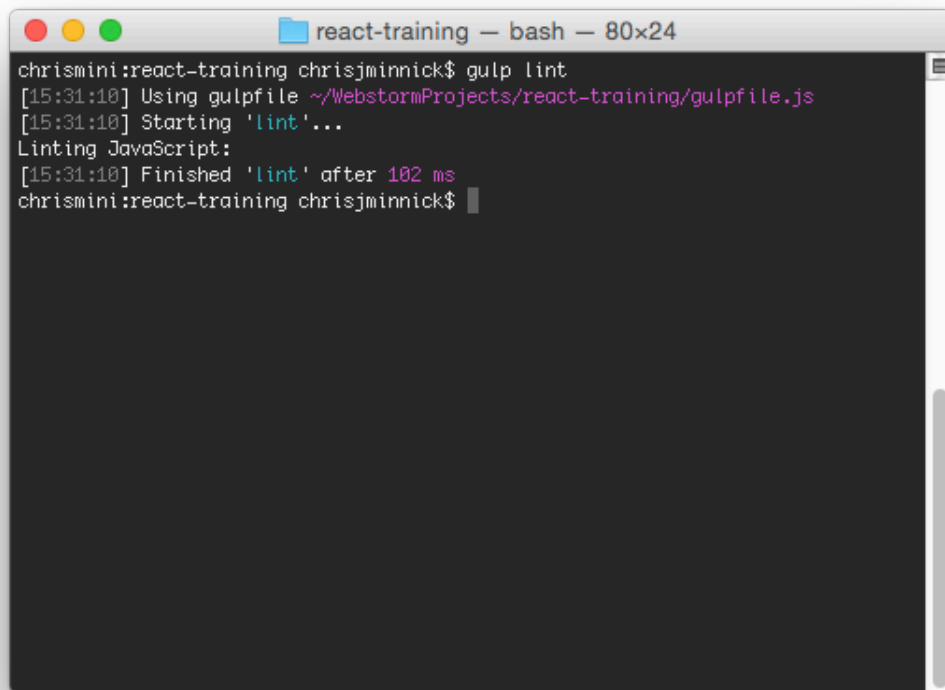
- 8. Finally, close the braces and terminate the statement:

```
});
```

- 9. Run the lint task:

```
gulp lint
```

Your output should look something like this:

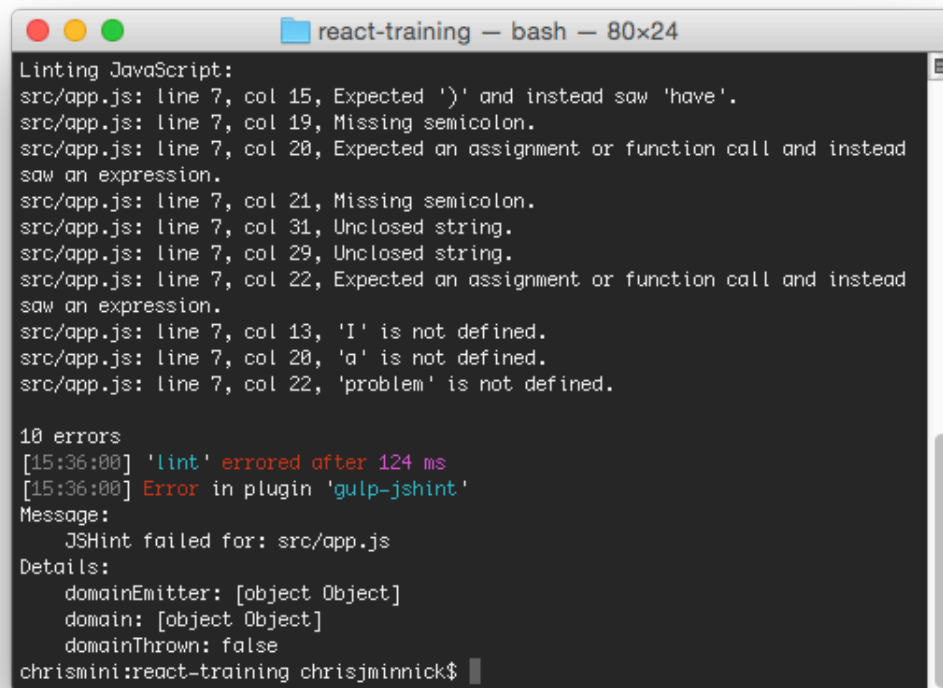
A terminal window titled "react-training — bash — 80x24" with a dark background. The output of the "gulp lint" command is displayed in a monospaced font with syntax highlighting: "chrismini:react-training chrisjminnick\$ gulp lint" (white), "[15:31:10] Using gulpfile ~/WebstormProjects/react-training/gulpfile.js" (pink), "[15:31:10] Starting 'lint'..." (white), "Linting JavaScript:" (white), "[15:31:10] Finished 'lint' after 102 ms" (pink), and "chrismini:react-training chrisjminnick\$" (white).

```
chrismini:react-training chrisjminnick$ gulp lint
[15:31:10] Using gulpfile ~/WebstormProjects/react-training/gulpfile.js
[15:31:10] Starting 'lint'...
Linting JavaScript:
[15:31:10] Finished 'lint' after 102 ms
chrismini:react-training chrisjminnick$
```

- 10. Next, let's put an error in a JavaScript file to verify that it gets caught. Open up **src/app.js** and enter the following code exactly.

```
console.log(I have a problem')
```

- 11. Save the file and run gulp lint. After a moment, you will get your result which should look something like this:

A terminal window titled "react-training — bash — 80x24" showing the output of a JSHint linting task. The output lists 10 errors in src/app.js, including issues with closing parentheses, semicolons, string literals, and undefined variables. The terminal text is as follows:

```
Linting JavaScript:
src/app.js: line 7, col 15, Expected ')' and instead saw 'have'.
src/app.js: line 7, col 19, Missing semicolon.
src/app.js: line 7, col 20, Expected an assignment or function call and instead
saw an expression.
src/app.js: line 7, col 21, Missing semicolon.
src/app.js: line 7, col 31, Unclosed string.
src/app.js: line 7, col 29, Unclosed string.
src/app.js: line 7, col 22, Expected an assignment or function call and instead
saw an expression.
src/app.js: line 7, col 13, 'I' is not defined.
src/app.js: line 7, col 20, 'a' is not defined.
src/app.js: line 7, col 22, 'problem' is not defined.

10 errors
[15:36:00] 'lint' errored after 124 ms
[15:36:00] Error in plugin 'gulp-jshint'
Message:
  JSHint failed for: src/app.js
Details:
  domainEmitter: [object Object]
  domain: [object Object]
  domainThrown: false
chrismini:react-training chrisjminnick$
```

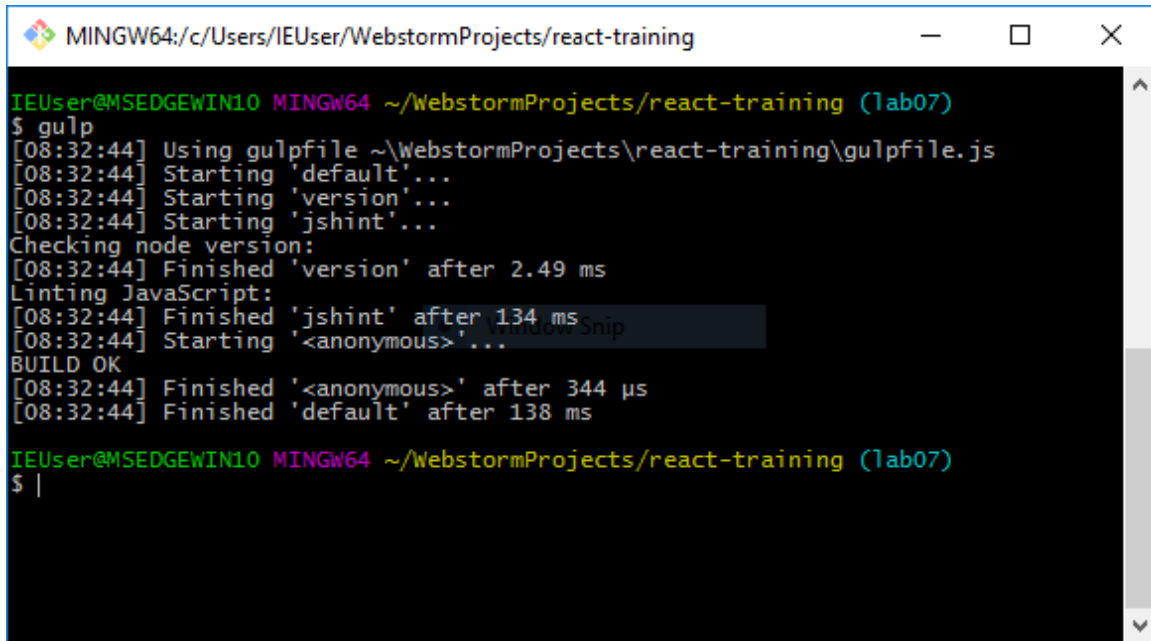
- 12. Fix the syntax errors in app.js and run the gulp lint task again so that it doesn't produce any errors.

Notice how we have each of our tasks configured to produce minimal, if any, output if they run successfully; but they should blow up with error messages if anything goes wrong. This is a common and recommended practice. Don't bother the user with messages they don't need to see.

- 13. Next, add the lint task as a dependency of default. Since it doesn't matter whether lint runs before or after the version task, we can make these two run in parallel, but the combination of these dependencies should always run before the default task finishes. Here's how you do that:

```
gulp.task('default', gulp.series(gulp.parallel('version', 'lint'),
function(done) {
  console.log('BUILD OK');
  done();
}));
```

- 14. Run your default task and make sure it works!



```
MINGW64:/c:/Users/IEUser/WebstormProjects/react-training
IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab07)
$ gulp
[08:32:44] Using gulpfile ~\WebstormProjects\react-training\gulpfile.js
[08:32:44] Starting 'default'...
[08:32:44] Starting 'version'...
[08:32:44] Starting 'jshint'...
Checking node version:
[08:32:44] Finished 'version' after 2.49 ms
Linting JavaScript:
[08:32:44] Finished 'jshint' after 134 ms
[08:32:44] Starting '<anonymous>'...
BUILD OK
[08:32:44] Finished '<anonymous>' after 344 μs
[08:32:44] Finished 'default' after 138 ms
IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab07)
$ |
```

- 15. `git add .` and `git commit`

Part 3: Setting JSHint Options

One of the great things about JSHint is how configurable it is. By default, it checks your code only for real errors. But, if you want to force yourself to write better code, you can specify some options that will cause it to flag issues that aren't technically errors, but are less-than-good practices.

- 1. Open your `gulpfile.js`
- 2. Go to <http://jshint.com/docs/options> in your web browser.
- 3. Read through the possible options and select the ones that you'd like to set in your lint task. Here are our recommended settings:

```
...
.pipe(jshint({
  esversion: 6,
  bitwise: true,
  curly: true,
  eqeqeq: true,
  forin: true,
  freeze: true,
  latedef: 'nofunc',
  nonbsp: true,
  nonew: true,
  node: true,
  browser: true,
  globals: []
})))
...
```

- 4. When you're happy with the options you've chosen, save your file and run your default gulp task.
- 5. If you get errors, either fix your code or adjust your options. If everything checks out, commit your changes to git.

Lab 7 (alternate): ESLint

ESLint is a newer linting tool than JSHint, and it also has better support for linting of React / JSX. In this alternate version of the lab, you will install ESLint and integrate it into your automated build.

- 1. Run `npm install --save-dev eslint`
- 2. Run `./node_modules/.bin/eslint --init`
- 3. Select Answer questions about your style as the answer to the first question.
- 4. Answer the questions as follows unless you have a good reason to answer a different way. Don't worry if you make a mistake, we'll set all of the options correctly in the config file.
 - Are you using ECMAScript 6 features? Y
 - Are you using ES6 modules? Y
 - Where will your code run? Select both Browser and Node.
 - Do you use JSX? Y
 - Do you use React? Y
 - What style of indentation do you use? (your choice)
 - What quotes do you use for strings? (your choice)
 - What line endings do you use? (Select Windows if you use Windows. Otherwise, select Unix)
 - Do you require semicolons? Y
 - What format do you want your config file to be in? JavaScript

Note: The init script may hang after the last question when using git bash shell. Use Ctrl – C to exit after the message appears that says “Successfully created .eslintrc.js”

```
tdd-react-es6 — bash — 86x41

| └─ type-check@0.3.2
|   └─ wordwrap@1.0.0
├─ path-is-inside@1.0.1
├─ pluralize@1.2.1
├─ progress@1.1.8
├─ require-uncached@1.0.2
|   └─ caller-path@0.1.0
|     └─ callsites@0.2.0
├─ resolve-from@1.0.1
├─ shelljs@0.6.0
├─ table@3.7.8
|   └─ bluebird@3.3.5
├─ lodash@4.12.0
├─ slice-ansi@0.0.4
├─ tv4@1.2.7
├─ xregexp@3.1.0
├─ text-table@0.2.0
└─ user-home@2.0.0

npm WARN react-training@1.0.0 No repository field.
chrismini:tdd-react-es6 chrisjminnick$ ./node_modules/.bin/eslint --init
? How would you like to configure ESLint? Answer questions about your style
? Are you using ECMAScript 6 features? Yes
? Are you using ES6 modules? Yes
? Where will your code run? Browser, Node
? Do you use CommonJS? Yes
? Do you use JSX? Yes
? Do you use React? Yes
? What style of indentation do you use? Spaces
? What quotes do you use for strings? Single
? What line endings do you use? Unix
? Do you require semicolons? Yes
? What format do you want your config file to be in? JavaScript
Installing eslint-plugin-react
react-training@1.0.0 /Users/chrisjminnick/WebstormProjects/tdd-react-es6
└─ eslint-plugin-react@5.1.1

npm WARN react-training@1.0.0 No repository field.
Successfully created .eslintrc.js file in /Users/chrisjminnick/WebstormProjects/tdd-react-es6
chrismini:tdd-react-es6 chrisjminnick$
```

- 5. Run `npm install --save-dev gulp-eslint`
- 6. Include `gulp-eslint` in `gulpfile.js`

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

- 7. Create a new task called **eslint**.

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

- 8. Make the new `eslint` task a dependency of `default`.
- 9. Tell `eslint` to ignore the `gulp` config file and any other file you don't need to lint by placing the following at the beginning of the file:

```
/*eslint-disable*/
```

- 10. Run `gulp`
- 11. Fix the errors reported by ESLint, or adjust the `.eslintrc.js` config file to fit your desired coding style.

If you're on Windows, you may need to change the line break style to "windows". You should also add a "no-console" option with a value of "warn" to override the default value of "error", since we'll be using `console.log` in upcoming lessons.

Example `.eslintrc.js`

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

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

Lab 8 - 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. Since we want to automate everything through gulp, we're going to use a package called gulp-webserver at:

<https://www.npmjs.com/package/gulp-webserver>

Part 1: Install gulp-webserver

- 1. Install gulp-webserver using npm.

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

- 2. Check gulp-webserver into your git repository.
- 3. Require gulp-webserver in your gulpfile and assign it to a const called **webserver**.
- 4. Add a new task to your gulpfile called **run**.

```
//run
gulp.task('run', function() {
  gulp.src('src')
    .pipe(webserver({
      livereload: true,
      open: true
    }));
});
```

This new task will be a stand-alone task that won't be a dependent of other tasks at this point.

- 5. Use the package documentation at <https://www.npmjs.com/package/gulp-webserver> to see what options you can pass to the web server and try making customizations to the task, such as changing the port and enabling directory listing.

At the time of this writing, gulp-webserver has a bug where it will ignore the root directory that you pass it if you specify `directoryListing: true`. If you experience this, remove the `directoryListing` option.

- 6. Create a file named `index.html` and put it in your `src` directory.
- 7. 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>
```

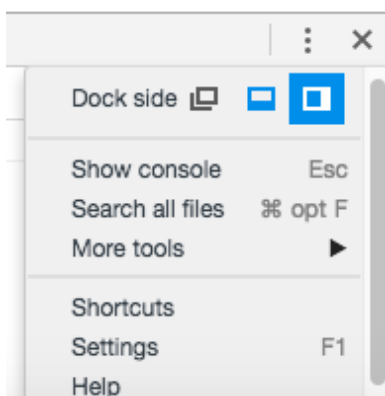
- 8. Start your web server by entering `gulp run` into the command line.
- 9. Open a web browser and navigate to `localhost:8000`. You should see the message "Welcome" in the browser window.
- 10. Stop the web server by pressing **Control - C**.
- 11. Check in your code.

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's 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 9 - Getting Started with Jasmine

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

- 1. In your shell, enter:
`npm install --save-dev jasmine`
- 2. When it completes, commit it to git

```
git add .
git commit -m "installed jasmine"
```

- 3. Initialize jasmine

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

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

- 4. Open `jasmine.json` and look at it. You don't need to change anything or add anything to it at this point.
- 5. Create a new file named **sayHelloSpec.js** in the `spec` directory.
- 6. Create a new file named **sayHello.js** in the `src` directory.

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

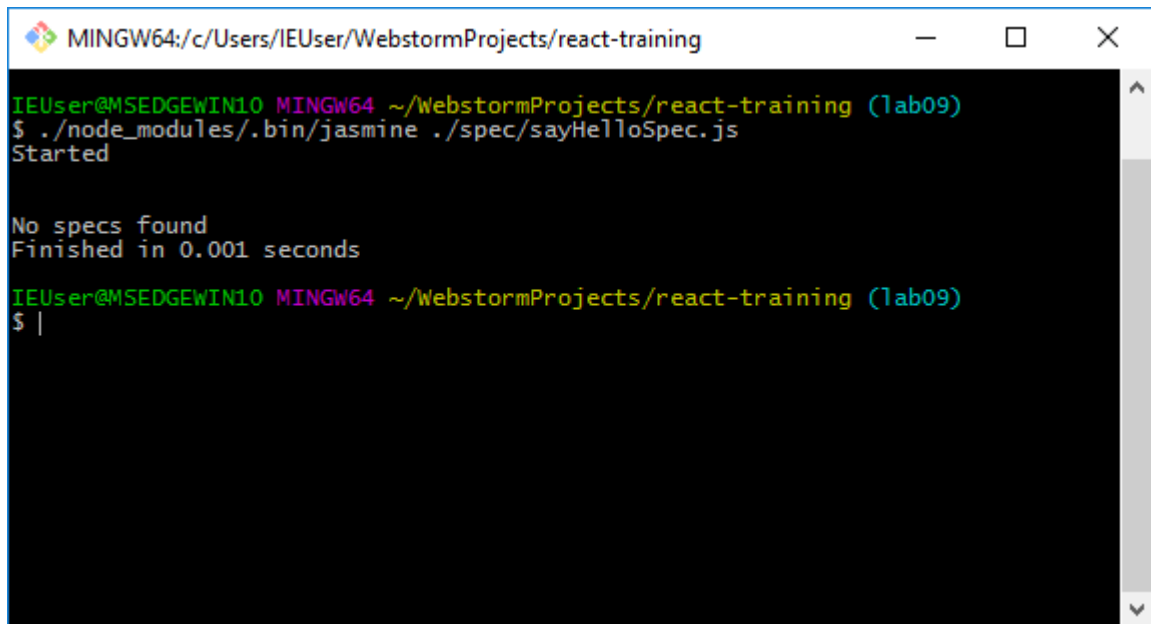
- 7. Start a new **suite** in `sayHelloSpec.js`

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

- 8. Save your spec and let's test it out!
- 9. In your command line, enter:

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

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



```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started

No specs found
Finished in 0.001 seconds

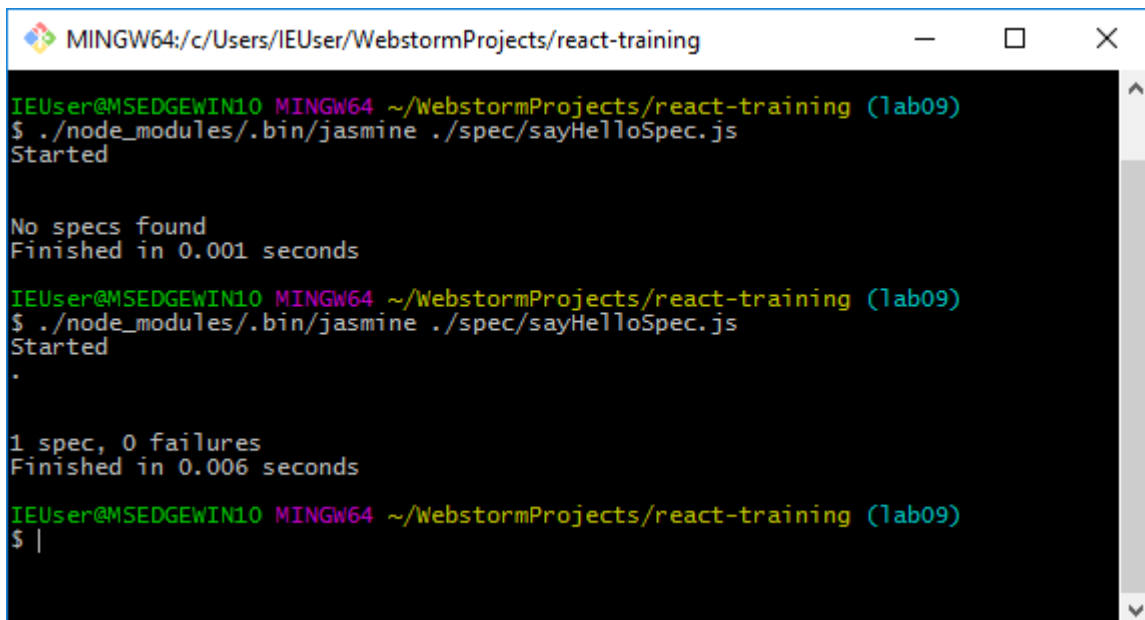
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ |
```

- 11. Inside your first test suite, create a new **spec**.

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

  });
});
```

- 12. Run your test again



```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started

No specs found
Finished in 0.001 seconds

IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
.

1 spec, 0 failures
Finished in 0.006 seconds

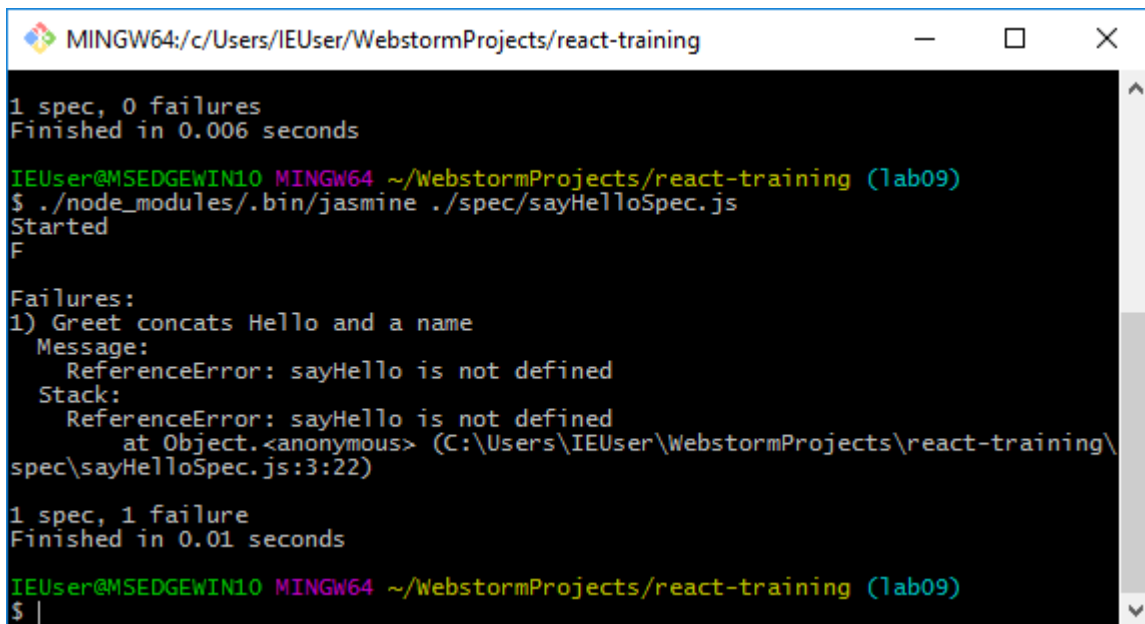
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ |
```

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

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

- 14. Run your suite again.

Jasmine will complain that it doesn't know what sayHello is.

A screenshot of a terminal window titled 'MINGW64:/c/Users/IEUser/WebstormProjects/react-training'. The terminal shows the output of running a Jasmine test suite. It starts with '1 spec, 0 failures' and 'Finished in 0.006 seconds'. Then, the command './node_modules/.bin/jasmine ./spec/sayHelloSpec.js' is entered. The output shows 'Started' followed by a red 'F' indicating a failure. The 'Failures' section shows '1) Greet concatenates Hello and a name' with a 'Message: ReferenceError: sayHello is not defined'. The stack trace points to 'spec/sayHelloSpec.js:3:22'. The terminal ends with '1 spec, 1 failure' and 'Finished in 0.01 seconds'.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training  
1 spec, 0 failures  
Finished in 0.006 seconds  
IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)  
$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js  
Started  
F  
Failures:  
1) Greet concatenates Hello and a name  
  Message:  
    ReferenceError: sayHello is not defined  
  Stack:  
    ReferenceError: sayHello is not defined  
      at Object.<anonymous> (C:\Users\IEUser\WebstormProjects\react-training\spec\sayHelloSpec.js:3:22)  
1 spec, 1 failure  
Finished in 0.01 seconds  
IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)  
$ |
```

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

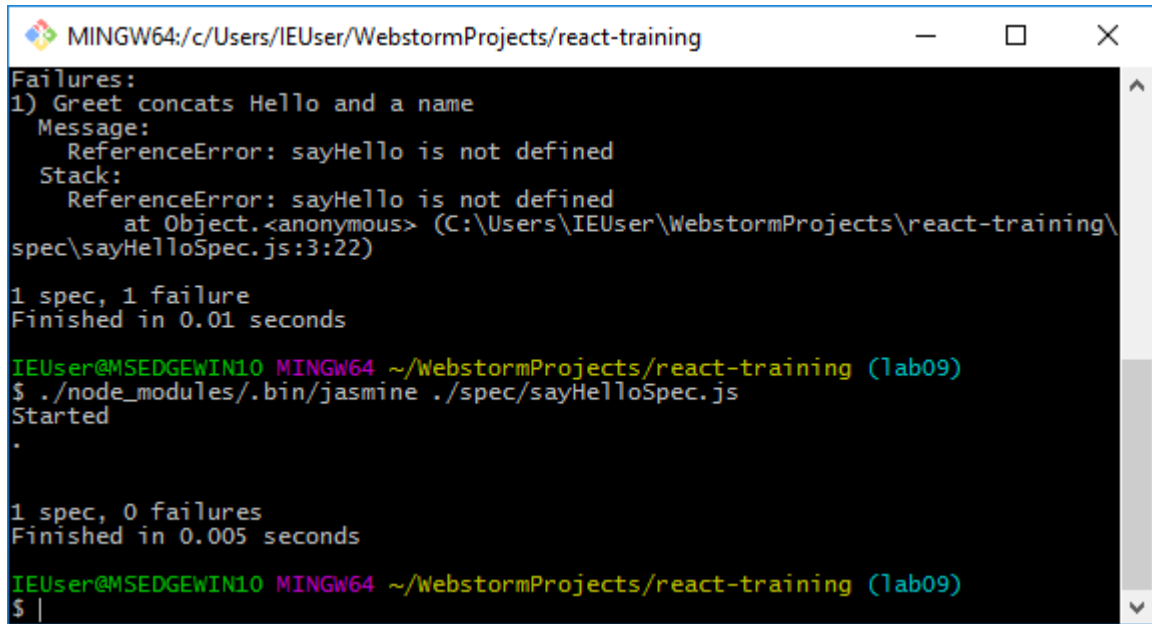
- 15. Use CommonJS to require sayHello.js as sayHello inside sayHelloSpec.js.

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

- 16. Inside of sayHello, write the bare minimum amount of code to get the test to pass. For example:

```
exports.greet = function greet(name) {  
  return 'Hello, ' + name;  
};
```

- 17. Run your suite. It should now pass. If it doesn't, figure out why and get it to pass.



```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
Failures:
1) Greet concats Hello and a name
  Message:
    ReferenceError: sayHello is not defined
  Stack:
    ReferenceError: sayHello is not defined
      at Object.<anonymous> (C:\Users\IEUser\WebstormProjects\react-training\
spec\sayHelloSpec.js:3:22)
1 spec, 1 failure
Finished in 0.01 seconds

IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
.

1 spec, 0 failures
Finished in 0.005 seconds

IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ |
```

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

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training

1 spec, 0 failures
Finished in 0.005 seconds

IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
.F

Failures:
1) Greet says "Hello, Friend!" when no name is given
   Message:
     Expected 'Hello, ' to equal 'Hello, Friend!'.
   Stack:
     Error: Expected 'Hello, ' to equal 'Hello, Friend!'.
           at Object.<anonymous> (C:\Users\IEUser\WebstormProjects\react-training\
spec\sayHelloSpec.js:13:24)

2 specs, 1 failure
Finished in 0.012 seconds

IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ |
```

- 21. Write code to make the test pass.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training

Failures:
1) Greet says "Hello, Friend!" when no name is given
   Message:
     Expected 'Hello, ' to equal 'Hello, Friend!'.
   Stack:
     Error: Expected 'Hello, ' to equal 'Hello, Friend!'.
           at Object.<anonymous> (C:\Users\IEUser\WebstormProjects\react-training\
spec\sayHelloSpec.js:13:24)

2 specs, 1 failure
Finished in 0.012 seconds

IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ ./node_modules/.bin/jasmine ./spec/sayHelloSpec.js
Started
..

2 specs, 0 failures
Finished in 0.008 seconds

IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab09)
$ |
```

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

Lab 10 - Integrate Jasmine into Your Build

In this lab, you will create a task to automatically run Jasmine as part of your default gulp task.

- 1. Install the gulp-jasmine package:

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

- 2. Commit to git
- 3. include gulp-jasmine in your gulpfile.js:

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

- 4. Create a new 'test' task:

```
gulp.task('test', function(done) {  
  gulp.src('spec/*.js')  
    .pipe(jasmine());  
  done();  
});
```

- 5. Run `gulp test`
- 6. Make your test task a dependency of default. It should run after the 'lint' task is done.
- 7. Test it out by running the `gulp` task.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab10)
$ gulp
[09:22:18] Using gulpfile ~/WebstormProjects/react-training/gulpfile.js
[09:22:18] Starting 'default'...
[09:22:18] Starting 'version'...
[09:22:18] Starting 'eslint'...
Checking node version:
[09:22:18] Finished 'version' after 5.03 ms
[09:22:21]
C:\Users\IEUser\WebstormProjects\react-training\gulpfile.js
   9:5  warning  Unexpected console statement  no-console
  46:5  warning  Unexpected console statement  no-console
  48:9  warning  Unexpected console statement  no-console
  66:9  warning  Unexpected console statement  no-console

C:\Users\IEUser\WebstormProjects\react-training\src\app.js
   1:1  warning  Unexpected console statement  no-console

x 5 problems (0 errors, 5 warnings)

[09:22:21] Finished 'eslint' after 2.76 s
[09:22:21] Starting 'test'...
[09:22:21] Finished 'test' after 38 ms
[09:22:21] Starting '<anonymous>'...
BUILD OK
[09:22:21] Finished '<anonymous>' after 286 μs
[09:22:21] Finished 'default' after 2.8 s
..

2 specs, 0 failures
Finished in 0 seconds

IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab10)
$ |
```

Next, we'll make the test report look better.

- 8. Install jasmine-reporters

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

- 9. Include jasmine-reporters in your gulpfile as reporters.

```
const reporters = require("jasmine-reporters");
```

- 10. Modify your test task to pass a reporter object to Jasmine and use the reporter named 'TerminalReporter'.

```
//test
gulp.task('test', function(done) {
  gulp.src('spec/*.js')
    .pipe(jasmine({
      reporter: new reporters.TerminalReporter()
    }));
  done();
});
```

- 11. Run your tests to see the output.
- 12. Change the verbosity level of the reporter by passing the following object to the TerminalReporter():

```
{
  verbosity: 3,
  color: true
}
```

Your finished gulp test task should look like this:

```
//Run unit tests
gulp.task('test', function(done) {
  gulp.src('spec/*.js')
    .pipe(jasmine({
      reporter: new reporters.TerminalReporter({
        verbosity: 3,
        color: true
      })
    }));
  done();
});
```

```
MINGW64: c:/Users/IEUser/WebstormProjects/react-training
$ gulp
[09:26:24] Using gulpfile ~\WebstormProjects\react-training\gulpfile.js
[09:26:24] Starting 'default'...
[09:26:24] Starting 'version'...
[09:26:24] Starting 'eslint'...
Checking node version:
[09:26:24] Finished 'version' after 2.42 ms
[09:26:27]
C:\Users\IEUser\WebstormProjects\react-training\gulpfile.js
 10:5 warning Unexpected console statement no-console
 52:5 warning Unexpected console statement no-console
 54:9 warning Unexpected console statement no-console
 72:9 warning Unexpected console statement no-console

C:\Users\IEUser\WebstormProjects\react-training\src\app.js
  1:1 warning Unexpected console statement no-console

x 5 problems (0 errors, 5 warnings)

[09:26:27] Finished 'eslint' after 2.8 s
[09:26:27] Starting 'test'...
[09:26:27] Finished 'test' after 39 ms
[09:26:27] Starting '<anonymous>'...
BUILD OK
[09:26:27] Finished '<anonymous>' after 2.27 ms
[09:26:27] Finished 'default' after 2.84 s
Greet
  concats Hello and a name ...
  Passed
  says "Hello, Friend!" when no name is given ...
  Passed
2 of 2 passed (0 skipped, 0 disabled).
SUCCESS: 2 specs, 0 failures, 0 skipped, 0 disabled in 0.014s.
IEUser@MSEDEGEWIN10 MINGW64 ~\WebstormProjects\react-training (lab10)
$ |
```

Note: At this point, you're only testing how your code works within Node.js, which isn't very helpful since your programs will eventually be deployed to browsers. In the next lab, you'll learn how to run your Jasmine tests in web browsers.

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
```

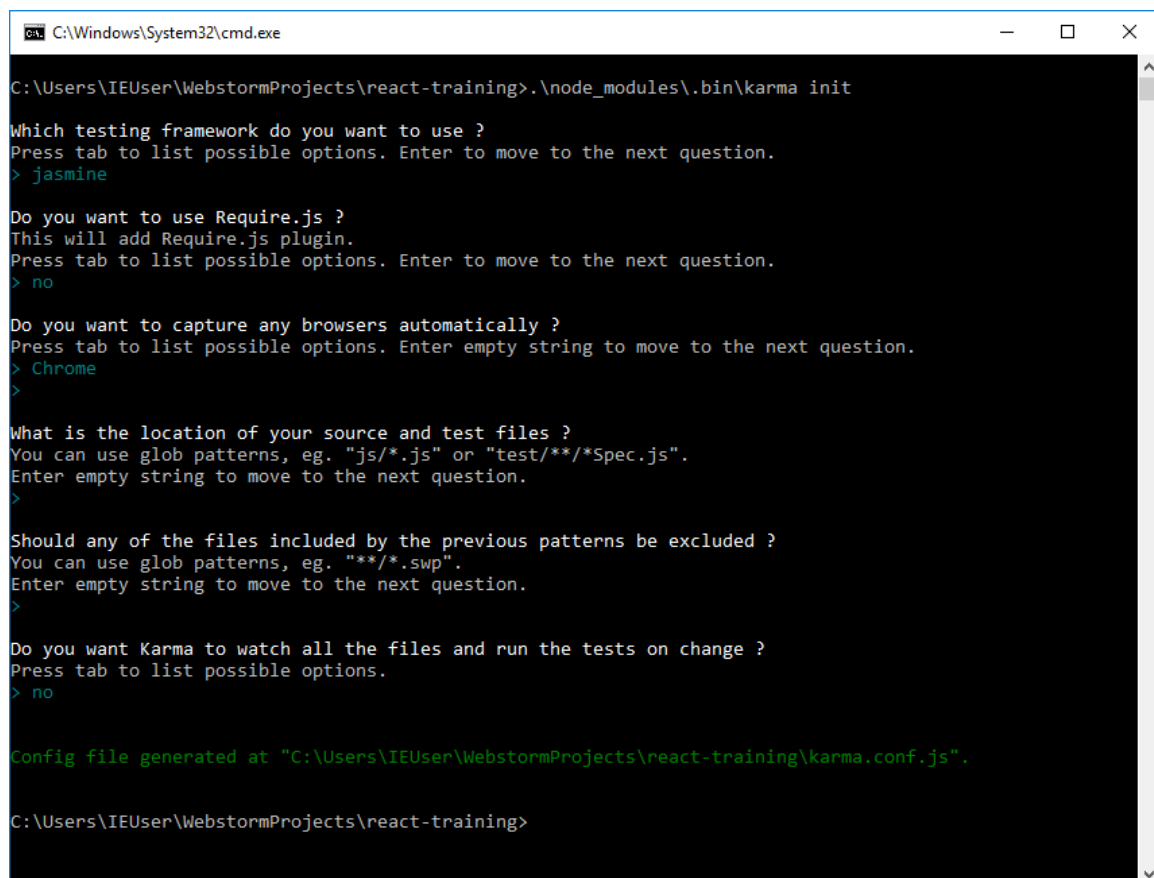
- 2. Run `karma init`

Note: On Windows, the following command will only work in the default Command Prompt (cmd.exe)

MacOS X: `./node_modules/.bin/karma init`

Windows Command Prompt: `.\node_modules\.bin\karma init`

You'll be walked through a series of configuration questions. Answer them as follows:



```
C:\Windows\System32\cmd.exe

C:\Users\IEUser\WebstormProjects\react-training>.\node_modules\.bin\karma init

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*.Spec.js".
Enter empty string to move to the next question.
>

Should any of the files included by the previous patterns be excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>

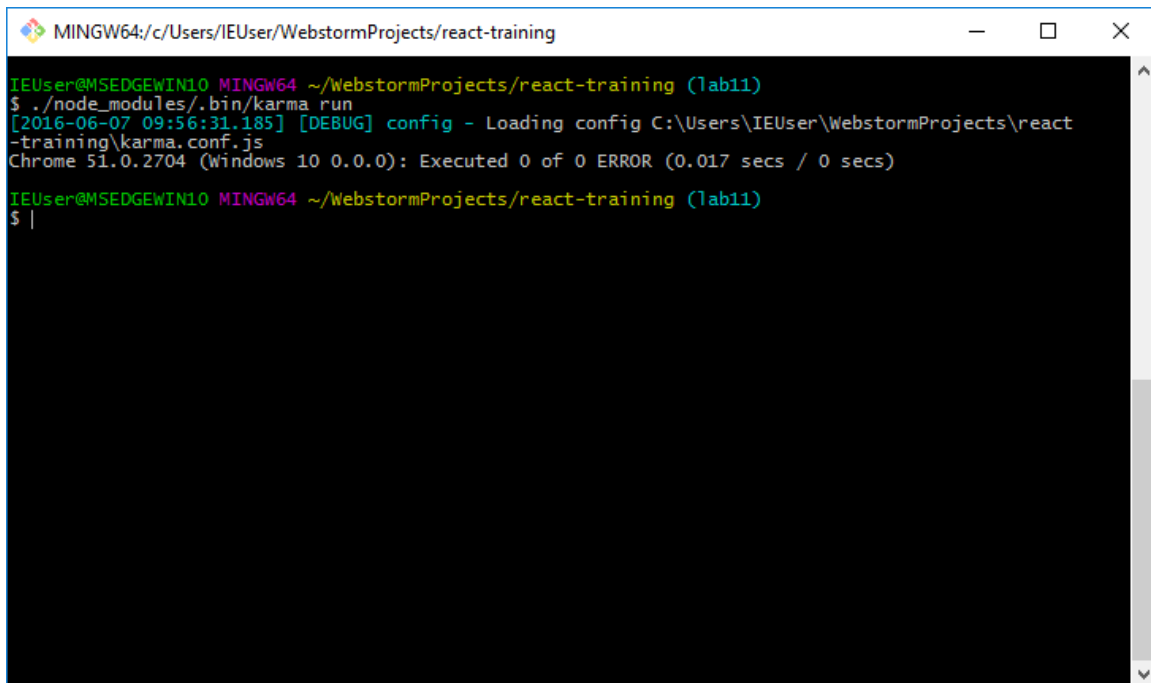
Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> no

Config file generated at "C:\Users\IEUser\WebstormProjects\react-training\karma.conf.js".

C:\Users\IEUser\WebstormProjects\react-training>
```

- 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. Open a new console window.
- 6. Run `./node_modules/.bin/karma run` to run tests

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



```

MINGW64: c:/Users/IEUser/WebstormProjects/react-training
IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (Tab11)
$ ./node_modules/.bin/karma run
[2016-06-07 09:56:31.185] [DEBUG] config - Loading config C:\Users\IEUser\WebstormProjects\react-training\karma.conf.js
Chrome 51.0.2704 (Windows 10 0.0.0): Executed 0 of 0 ERROR (0.017 secs / 0 secs)
IEUser@MSEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (Tab11)
$ |

```

- 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. Stop the Karma server (using CTRL-C) and restart it to reload the config file.
- 9. Run `./node_modules/.bin/karma run`

You'll get a new error message:

```
Uncaught ReferenceError: require is not defined
```

- 10. To fix this, install `karma-commonjs` (`npm install --save-dev karma-commonjs`).

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.

- 11. Include commonjs in the plugins and framework option in `karma.config.js`

```
plugins: ['karma-jasmine', 'karma-chrome-launcher', 'karma-  
commonjs'],  
frameworks: ['jasmine', 'commonjs'],
```

- 12. Add your `src` directory to the files option

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

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

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

- 14. Restart the Karma server.'
- 15. Enter `./node_modules/.bin/karma run` in a different console window.

Your tests should pass.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (tab11)
$ ./node_modules/.bin/karma run
[2016-06-07 10:19:41.762] [DEBUG] config - Loading config C:\Users\IEUser\WebstormProjects\react-trai
ning\karma.conf.js
Chrome 51.0.2704 (Windows 10 0.0.0) LOG: 'Hello, world!'
Chrome 51.0.2704 (Windows 10 0.0.0): Executed 2 of 2 SUCCESS (0.004 secs / 0.004 secs)
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (tab11)
$ |
```

- 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.

```
MINGW64:/c/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (tab11)
$ ./node_modules/.bin/karma run
[2016-06-07 10:24:57.352] [DEBUG] config - Loading config C:\Users\IEUser\WebstormProjects\react-trai
ning\karma.conf.js
Chrome 51.0.2704 (Windows 10 0.0.0) LOG: 'Hello, world!'
LOG: 'Hello, world!'
IE 6.0.0 (Windows XP 0.0.0) LOG: 'Hello, world!'
Mobile Safari 9.0.0 (iOS 9.2.1) LOG: 'Hello, world!'
Edge 13.10586.0 (Windows 10 0.0.0) LOG: 'Hello, world!'
Chrome 51.0.2704 (Windows 10 0.0.0): Executed 2 of 2 SUCCESS (0.005 secs / 0.005 secs)
IE 8.0.0 (Windows 7 0.0.0): Executed 2 of 2 SUCCESS (0.027 secs / 0 secs)
IE 6.0.0 (Windows XP 0.0.0): Executed 2 of 2 SUCCESS (0.003 secs / 0 secs)
Mobile Safari 9.0.0 (iOS 9.2.1): Executed 2 of 2 SUCCESS (0.005 secs / 0.001 secs)
Edge 13.10586.0 (Windows 10 0.0.0): Executed 2 of 2 SUCCESS (0.005 secs / 0.003 secs)
TOTAL: 10 SUCCESS
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (tab11)
$ |
```

Part 2: Automating Karma

- 1. Next, we'll integrate Karma into the build script so that the automated browser tests run when we run our default task.
- 2. Install gulp-karma-runner:

```
npm install --save-dev gulp-karma-runner
```

- 3. Require gulp-karma-runner in gulpfile.js

```
const karma = require("gulp-karma-runner");
```

- 4. Add a new task called karma for starting the karma server:

```
//start karma server
gulp.task('karma', function() {
  return gulp.src([
    'spec/**/*.js',
    'src/**/*.js'
  ],
    {'read': false}).pipe(
    karma.server({
      configFile: __dirname + '/karma.conf.js',
      'singleRun': false
    })
  );
});
```

- 5. Rename your test task to 'jasmine' and create a new task named test to run the tests in Karma:

```
//run tests in Karma
gulp.task('test', function () {
  return gulp.src([
    'spec/**/*.js',
    'src/**/*.js'
  ],
    {'read': false}).pipe(
    karma.runner({
      configFile: __dirname + '/karma.conf.js',
      'singleRun': false
    })
  );
});
```

- 6. Stop the karma server if it's running and restart it by entering **gulp karma**.
- 7. Run **gulp test**
- 8. Make 'test' a dependency of 'default' if it's not already.
- 9. Run **gulp**


```
MINGW64:/c:/Users/IEUser/WebstormProjects/react-training
C:\Users\IEUser\WebstormProjects\react-training\gulpfile.js
11:5 warning Unexpected console statement no-console
53:9 warning Expected indentation of 4 space characters but found 8 indent
60:5 warning Expected indentation of 0 space characters but found 4 indent
66:9 warning Expected indentation of 4 space characters but found 8 indent
79:5 warning Unexpected console statement no-console
81:9 warning Unexpected console statement no-console
99:9 warning Unexpected console statement no-console

C:\Users\IEUser\WebstormProjects\react-training\src\app.js
1:1 warning Unexpected console statement no-console

x 8 problems (0 errors, 8 warnings)

[10:45:04] Finished 'eslint' after 4.07 s
[10:45:04] Starting 'test'...
[2016-06-07 10:45:04.582] [DEBUG] config - Loading config C:\Users\IEUser\WebstormProjects\react-training\karma.conf.js
Chrome 51.0.2704 (Windows 10 0.0.0) LOG: 'Hello, world!'
LOG: 'Hello, world!'
LOG: 'Hello, world!'
IE 8.0.0 (Windows 7 0.0.0) LOG: 'Hello, world!'
Mobile Safari 9.0.0 (iOS 9.2.1) LOG: 'Hello, world!'
Chrome 51.0.2704 (Windows 10 0.0.0): Executed 2 of 2 SUCCESS (0.005 secs / 0.005 secs)
IE 6.0.0 (Windows XP 0.0.0): Executed 2 of 2 SUCCESS (0.004 secs / 0 secs)
Mobile Safari 9.0.0 (iOS 9.2.1): Executed 2 of 2 SUCCESS (0.003 secs / 0 secs)
IE 8.0.0 (Windows 7 0.0.0): Executed 2 of 2 SUCCESS (0.01 secs / 0.001 secs)
Mobile Safari 9.0.0 (iOS 9.2.1): Executed 2 of 2 SUCCESS (0.005 secs / 0 secs)
TOTAL: 10 SUCCESS
[10:45:07] Finished 'test' after 2.65 s
[10:45:07] Starting '<anonymous>'...
BUILD OK
[10:45:07] Finished '<anonymous>' after 381 µs
[10:45:07] Finished 'default' after 6.72 s

IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab11)
$ |
```

Lab 12 - Deploy with Webpack

Now that we have automated linting and testing, the next step is to automate the building of what will actually 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 gulp and webpack.

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

- 1. Create a folder 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.

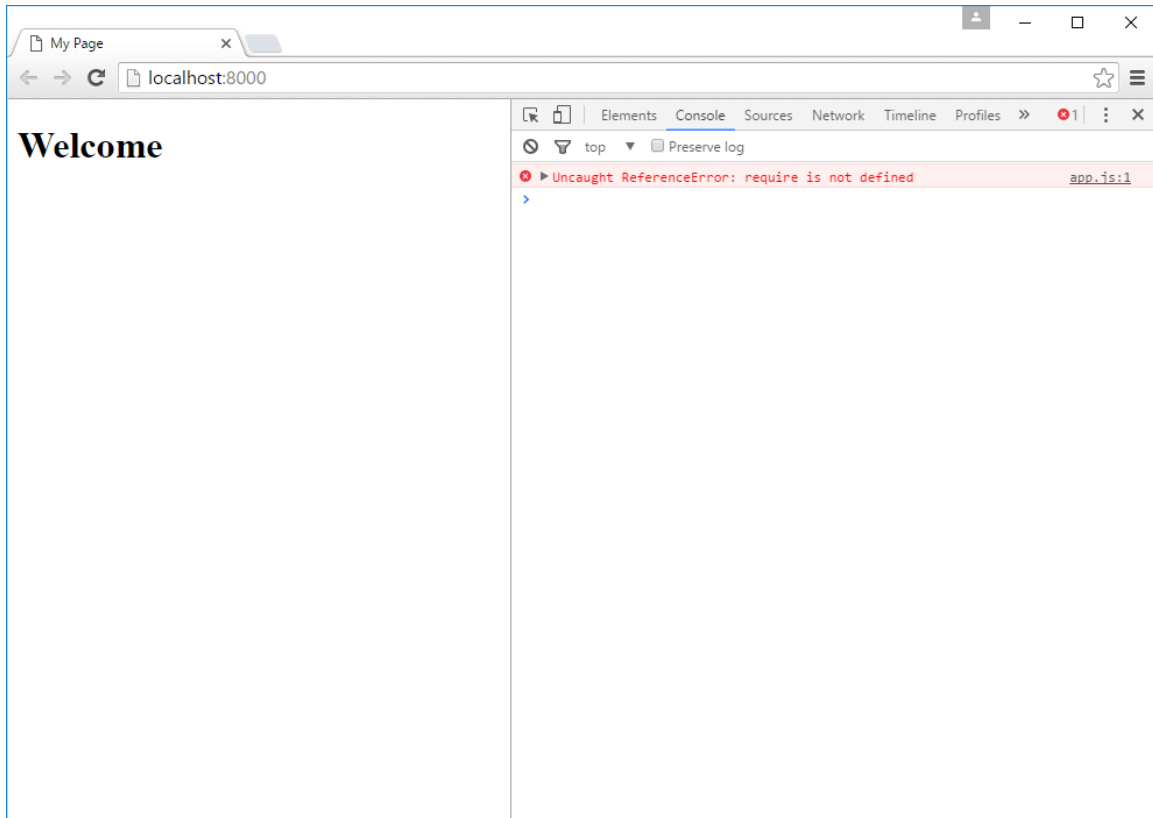
Here's what the index.html file should look like now:

```
<!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. Run gulp to make sure that this move didn't break anything, and fix anything that it did break.
- 5. Check in your code.
- 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');
document.getElementById('welcome-message')
  .innerHTML = sayHello.greet('Your name');
```

- 9. Test the site by using your gulp run task.
- 10. Load the site in your browser at localhost:8000 (or whichever port your local webserver is configured to listen on) and you will see that the page loads, **but the script doesn't run**. The browser doesn't know what require means.



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

- 11. Change the webserver `src` in your gulpfile to **dist**. This will be the directory we'll create using webpack.

```
gulp.task('run', function() {  
  gulp.src('dist')  
  ...  
})
```

- 12. Install webpack and the webpack-stream module

```
npm install --save-dev webpack webpack-stream
```

- 13. Commit to the repo.
- 14. Create a file named **webpack.config.js** in the root of your project.
- 15. Inside `webpack.config.js`, specify the entry and output:

```
module.exports = {  
  entry : './src/scripts/app.js',  
  output : {  
    filename : 'app.js'  
  }  
};
```

- 16. Enabled debug mode and generate a SourceMap.

```
module.exports = {  
  debug: true,  
  devtool: 'source-map',  
  entry : './src/scripts/app.js',  
  output : {  
    filename : 'app.js'  
  }  
};
```

- 17. Require webpack-stream in your gulpfile as **webpack** and create a new task called **webpack**.

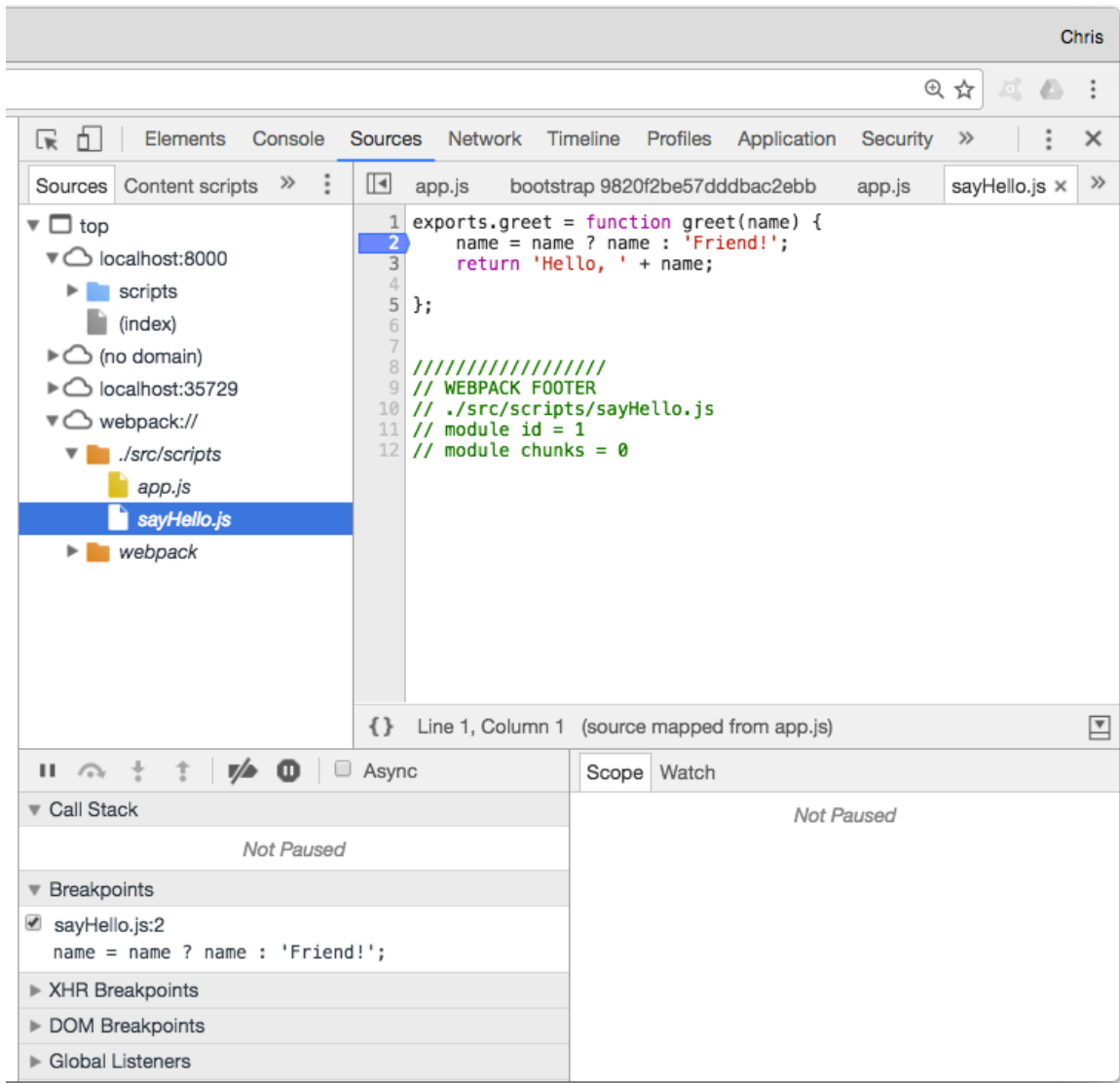
Here's what it should look like:

```
//build the dist directory  
gulp.task('webpack', function() {  
  return gulp.src('src/scripts/app.js')  
    .pipe(webpack(require('./webpack.config.js')))  
    .pipe(gulp.dest('dist/scripts/'));  
});
```

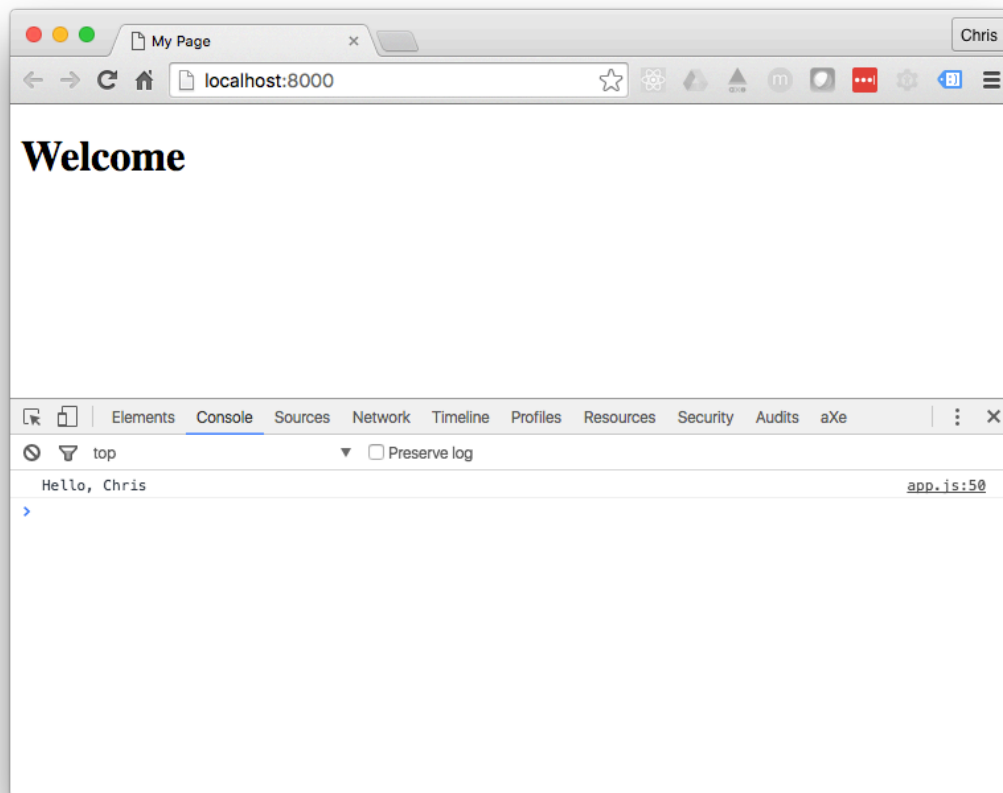
- 1. Run **gulp webpack**

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

- 2. Open **dist/scripts/apps.js** and look at the code that's created to make it work.
- 3. Open the Chrome Dev Tools and go to the Sources tab. Click on the webpack link on the right pane to view the original source.



- ☐ 4. Make a copy of **index.html** and put it in **dist**.
- ☐ 5. Enter gulp run and make sure that your page works in your browser.



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

```
# generated files
dist/
```

- 6. Check in your code.

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 create a new task to delete the **dist** directory before each build.

- 7. Create a task called '**clean**'.

```
gulp.task('clean', function() {
  console.log ('removing dist directory');
  return del([
```

```
        'dist'
    });
});
```

In order to get this working, you will need to install a module, **del**.

- 8. Install **del** and require it in your gulpfile.
- 9. Run **gulp clean**
- 10. Run **gulp webpack**
- 11. Run **gulp clean** again to make sure that it's doing what it should.

Next, we need to copy over index.html and integrate webpack into a full build script that will be a dependency of the run task.

- 12. Create a new task called `copy`.

```
gulp.task('copy', function() {
    return gulp.src('src/*.html').pipe(gulp.dest(DIST));
});
```

This will copy every .html file from src into the destination.

Notice that I've replaced the destination with a constant. Declare this constant at the top of your gulpfile and replace every instance of the dist directory in the gulpfile with this constant.

- 13. Create a new task called **build** that will run **clean** first, followed by **webpack** and **copy** in parallel, followed by **run**.
- 14. Test out your new build script.

```
gulp build
```

- 14. Run **gulp karma** and **gulp** to make sure that they still work.

You will get errors from ESLint. The reason is that ESLint is trying to Lint the dist directory, which contains generated and optimized JavaScript files -- not properly formatted ones. To clear these errors, you'll need to exclude the dist directory from linting.

- 15. Modify your eslint Gulp task to exclude the dist directory and all of the files contained within it.

```
return gulp.src(['**/*.js', '!dist/**', '!node_modules/**'])
```

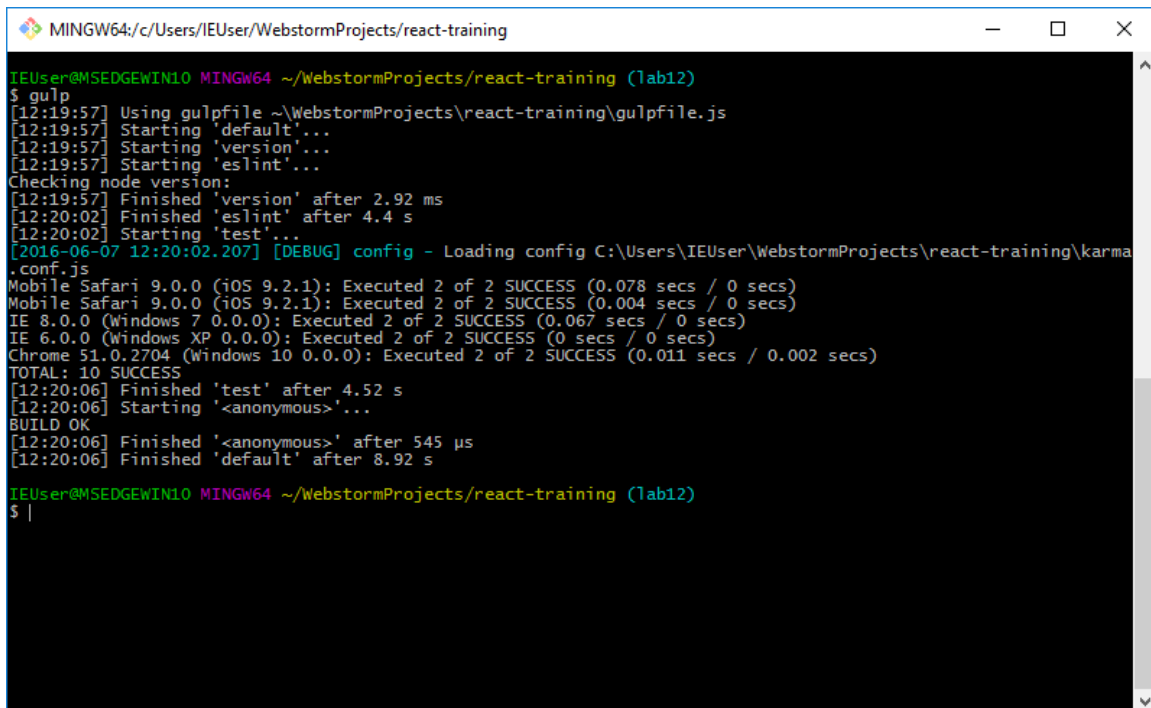
- 16. Test your code again. The built code will produce errors because of the `getElementByld` in app.js. The code is running before the DOM is ready.

- 17. To fix these errors, add an event listener in app.js to wait until the window is loaded before running the DOM manipulation code:

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

- 18. Test again. 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 and gulpfile.js). If you want to exclude these files from linting, 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 gulpfile.

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

Remember that Gulp 4 requires that dependent tasks come before the task that requires them.

Would it be easier to read and manage your gulp tasks if they were in separate files, or is it better to have them all in one file where you can look at them together?

Think about how you might split your gulpfile into multiple files using CommonJS.

Are there any repeated paths in your gulpfile that could be factored out into variables?

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**, **karma-webpack preprocessor**, **babel-preset-es2015**.

```
npm install babel-loader babel-core karma-webpack babel-preset-es2015 --save-dev
```

- 2. Add the loader and the babel preset into the webpack config:

```
module.exports = {
  entry : './src/scripts/app.js',
  output : {
    filename : 'app.js'
  },
  module : {
    loaders: [ {
      test : /\.js$/,
      loader : 'babel-loader',
      query: {
        presets: ['es2015']
      }
    }
  ]
};
```

- 3. Remove commonjs from the plugins and frameworks arrays in **karma.conf.js** and add **karma-webpack**.

```
plugins: ['karma-jasmine', 'karma-webpack'],
frameworks: ['jasmine'],
```

- 4. Remove the **src** directory from the **gulp karma** task:

```
gulp.task('karma', function() {
  return gulp.src([
    'spec/**/*.js'
  ],
  ...
```

- 5. Change the preprocessor in **karma.conf.js** to **webpack** and add **webpack** settings.

```

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

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

  module : {
    loaders: [ {
      test   : /\.js$/,
      loader : 'babel-loader',
      query: {
        presets: ['es2015']
      }
    }
  ]
}
},

```

- 6. Make the following changes to app.js to introduce a little ES6 code into the file.

```

var const sayHello = require('./sayHello.js');
window.addEventListener('load', function() ()=>{
  document.getElementById('welcome-message').innerHTML =
sayHello.greet('Chris');
});

```

- 7. Run your tests and build to make sure everything still works.
- 8. Save and commit.

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. Remove 'use strict;' from the beginning of the file.

Strict mode is implied in ES6 modules, so there's no need to set it explicitly.

- 3. Export the module using ES6 syntax:

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

- 4. Open **app.js**
- 5. Remove 'use strict;' from the beginning of the file (if present).

The use of `import` in a JavaScript file also causes it to be implicitly in strict mode.

- 6. Import the module using ES6 syntax.

```
const sayHello = require('./sayHello.js');  
import * as sayHello from './sayHello.js';
```

- 7. Open **sayHelloSpec.js**
- 8. Remove 'use strict;' from the beginning of the file.
- 9. Import `sayHello` using ES6 syntax.

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

- 10. Start the Karma server and run your tests to check that it's working!
- 11. 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;  
};
```

□ 12. Test and check in your code

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 babel-react preset to webpack.config.js

```
module.exports = {
  entry : './src/scripts/app.js',
  output : {
    filename : 'app.js'
  },
  module : {
    loaders: [ {
      test : /\.js$/,
      loader : 'babel-loader',
      query: {
        presets: ['es2015', 'react']
      }
    }
  ]
}
```

Note: React's development mode is slower than the production mode. To set React to production mode, you will need to set an environment variable.

- 4. Use Webpack's DefinePlugin method to set the production environment variable

```
module.exports = {
  entry : './src/scripts/app.js',
  output : {
    filename : 'app.js'
```

```

    },
    module : {
      loaders: [ {
        test    : /\.js$/,
        loader  : 'babel-loader',
        query: {
          presets: ['es2015','react']
        }
      }
    ]
  },
  plugins: [
    new webpack.DefinePlugin({
      'process.env': {
        'NODE_ENV': '"production"'
      }
    })
  ]
};

```

- 5. Comment out the `process.env` variable for now to make testing / debugging easier

```

/*plugins: [
  new webpack.DefinePlugin({
    'process.env': {
      'NODE_ENV': '"production"'
    }
  })
]*/

```

- 6. Remove the `<h1>` in `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.

- 7. Open `app.js` and remove everything that's in there currently.
- 8. Require `react` and `react-dom` in `app.js`:

```

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

```

- 9. Enter the following into `app.js`:

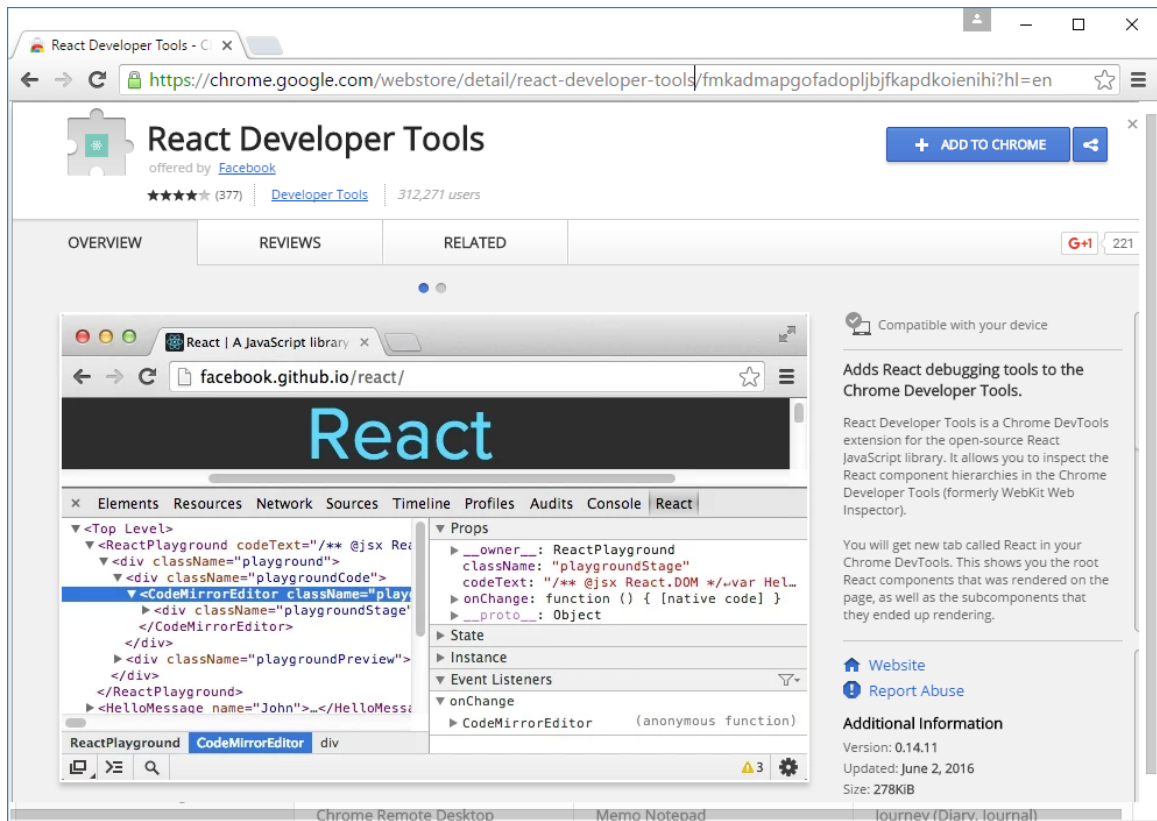
```

ReactDOM.render(
  <h1>Hello, world!</h1>,

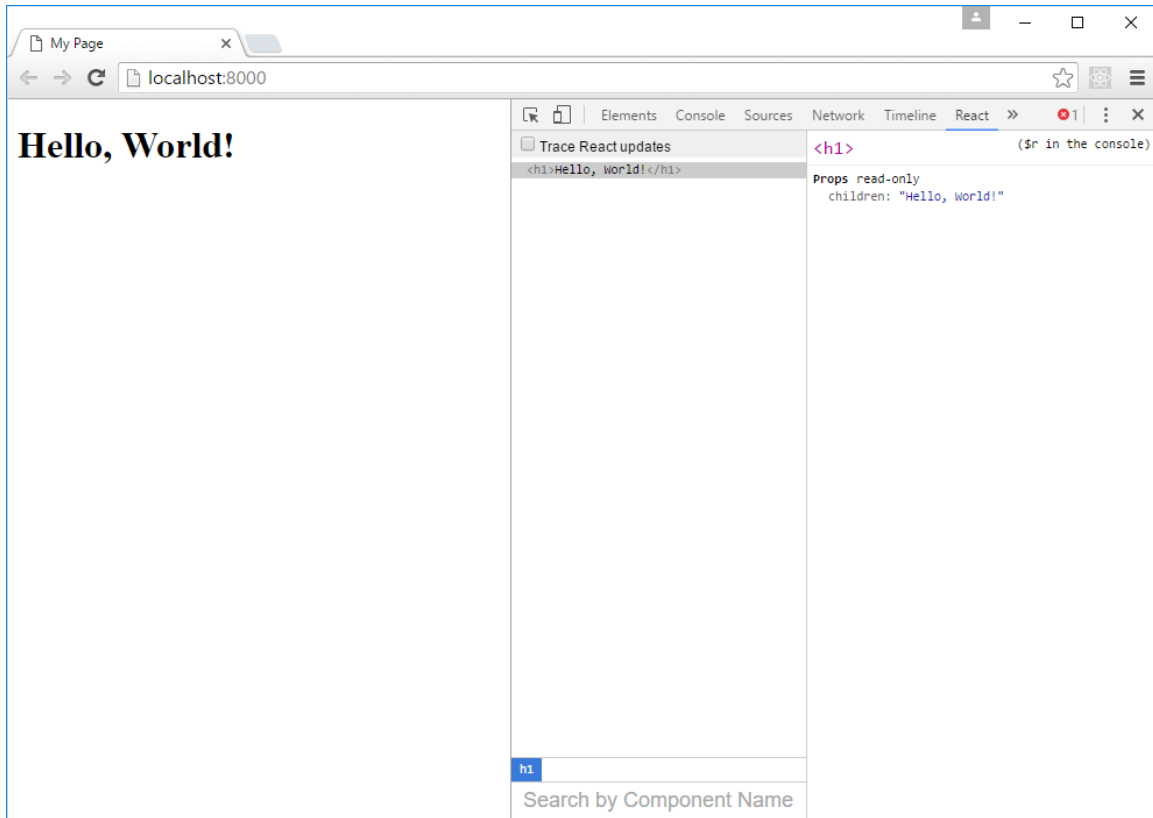
```

```
document.getElementById('app')
);
```

- 10. **gulp build**
- 11. Inspect the Hello, World! app in the browser using the developer tools.
- 12. Search for and install the React Developer Tools Chrome extension.



- 13. 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.

```
var SayHello = React.createClass({
  render: function() {

    return <h1>
      Hello, {this.props.name}!
    </h1>;
  }
});
```

Note: This is the pre-ES6 way of creating React classes. We'll be primarily using ES6 classes going forward, but it's important to show the "old" way of doing it first, because not everything works with the new method just yet and many React classes in the wild are still created using this `createClass()` method.

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

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

- 3. Run **gulp build** 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` using CommonJS syntax:

```
module.exports = SayHello;
```

- 9. Require `SayHello` in `app.js`, using CommonJS syntax.

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

- 10. Re-build and re-run.

Part 3: Convert to ES6

In this part, we'll modify our Hello, World app to use ES6.

- 1. Open `app.js`
- 2. Change the CommonJS requires to ES6 imports:

```
import React from 'react';
import ReactDOM from 'react-dom';
import SayHello from './SayHello';
```

- 3. Change the CommonJS `require` in `SayHello.js` to an ES6 import.
- 4. Change the component to an ES6 class.

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

- 5. Export `SayHello`

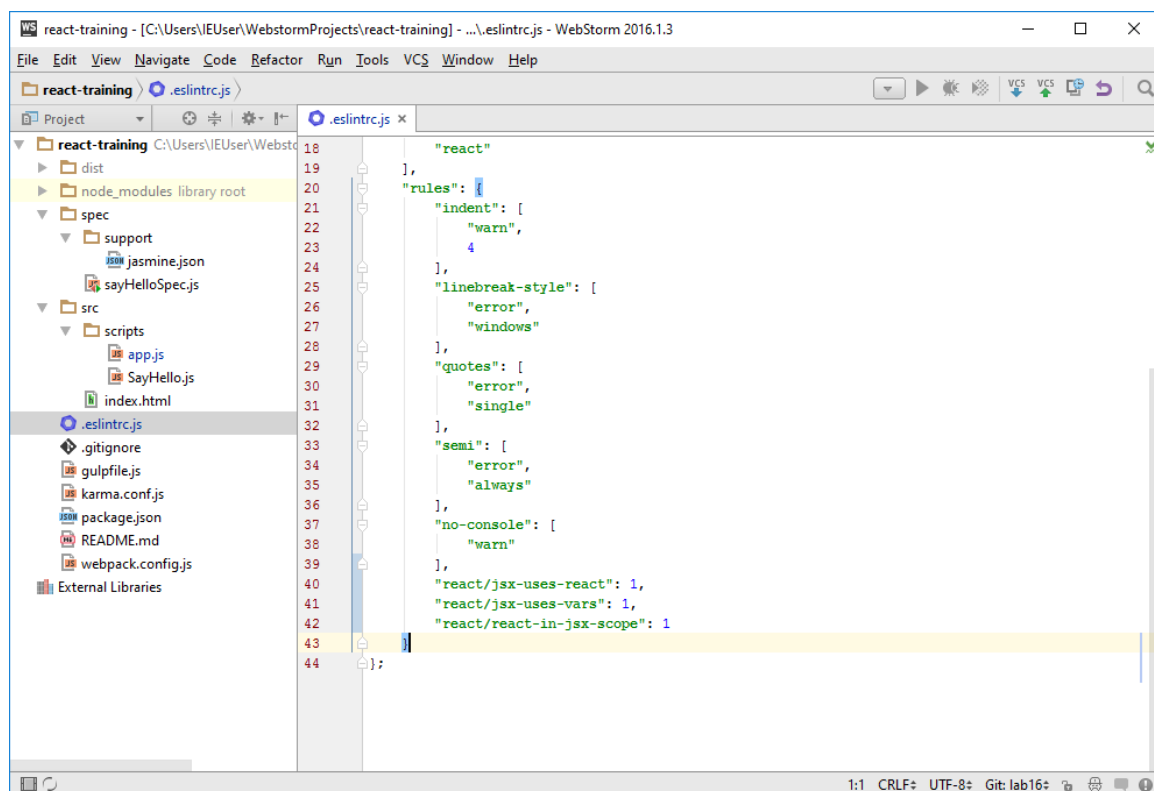
```
export default SayHello;
```

- 6. Build, run, check in

Part 4: Configure ESLint and Karma for React

- 1. Add the following rules to your ESLint config file:

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



- 2. Run the **eslint** Gulp task and make the necessary changes to your code to get the tests to pass.
- 3. Install react-addons-test-utils.

```
npm install --save-dev react-addons-test-utils
```

- 4. Update sayHelloSpec.js to test whether the SayHello component renders.

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

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

- 5. Add the webpack-react preset to karma.conf.js. Your webpack configuration in karma.conf.js should match this:

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

  module : {
    loaders: [ {
      test : /\.js$/,
      loader : 'babel-loader',
      query: {
        presets: ['es2015','react']
      }
    }
  ]
},
```

- 6. Start the Karma server and run the gulp task to run your test.

```
MINGW64:/c:/Users/IEUser/WebstormProjects/react-training
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab16)
$ gulp
[08:58:25] Using gulpfile ~\WebstormProjects\react-training\gulpfile.js
[08:58:25] Starting 'default'...
[08:58:25] Starting 'version'...
[08:58:25] Starting 'eslint'...
Checking node version:
[08:58:25] Finished 'version' after 2.72 ms
[08:58:32] Finished 'eslint' after 6.62 s
[08:58:32] Starting 'test'...
[2016-06-08 08:58:32.184] [DEBUG] config - Loading config C:\Users\IEUser\WebstormProjects\react-training\karma.conf.js
Chrome 51.0.2704 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.008 secs / 0.026 secs)
Mobile Safari 9.0.0 (iOS 9.2.1): Executed 1 of 1 SUCCESS (0.341 secs / 0.238 secs)
Opera 36.0.2130 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.014 secs / 0.01 secs)
Edge 13.10586.0 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.001 secs / 0.018 secs)
IE 8.0.0 (Windows 7 0.0.0): Executed 1 of 1 SUCCESS (0.457 secs / 0.008 secs)
Firefox 45.0.0 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (9.405 secs / 0.008 secs)
TOTAL: 6 SUCCESS
[08:59:28] Finished 'test' after 56 s
[08:59:28] Starting '<anonymous>'...
BUILD OK
[08:59:28] Finished '<anonymous>' after 242 µs
[08:59:28] Finished 'default' after 1.03 min
IEUser@MSEdgeWIN10 MINGW64 ~/WebstormProjects/react-training (lab16)
$ |
```

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.

Welcome!

What is this question?

☐ Answer 1

☐ Answer 2

☐ Answer 3

Go!

- 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 of 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';

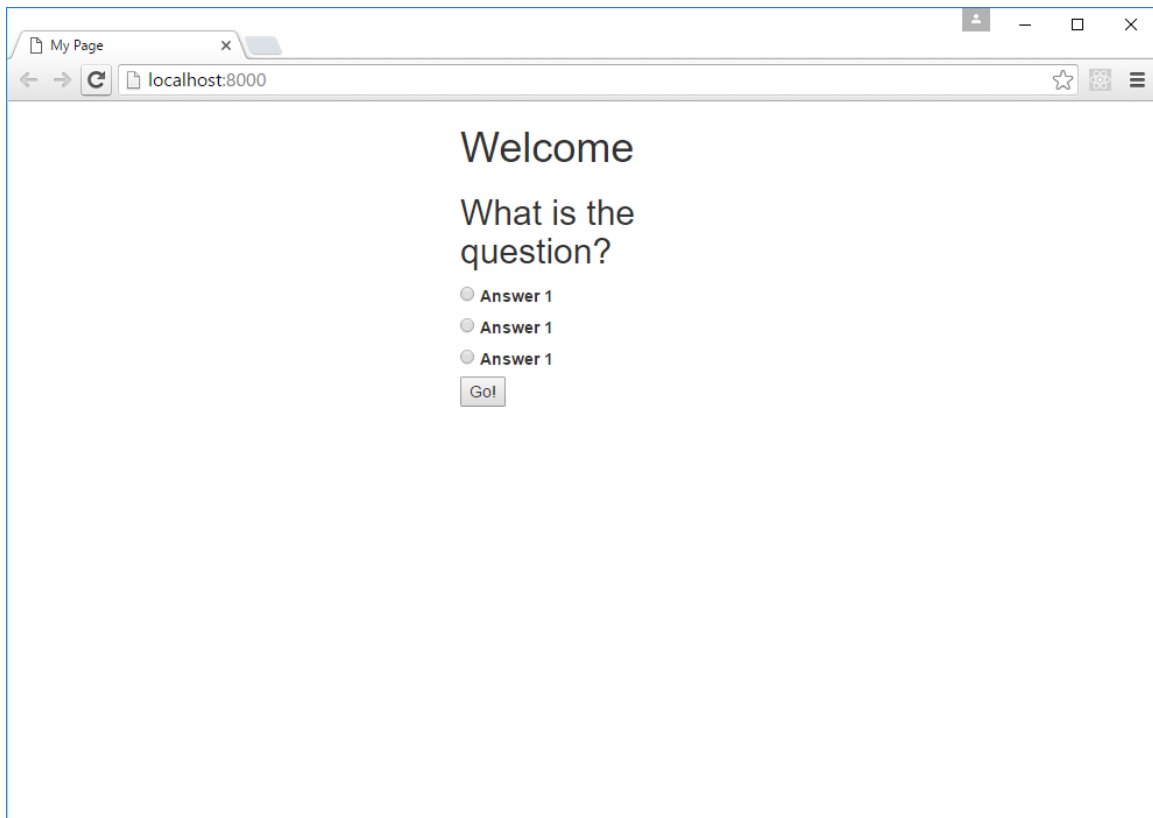
class PollContainer extends React.Component {
  render() {
    return (
      <div className="container">
        <div className="col-sm-4 col-sm-offset-4">
          <PollHeader />
          <form>
            <PollQuestion />
            <PollAnswer />
            <PollAnswer />
            <PollAnswer />
            <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 in order to include the Bootstrap CSS.

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

- 11. `gulp 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 of 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. Inside your gulpfile.js, change the src for the karma task to point to tests.webpack.js

```
//start karma server
gulp.task('karma', function() {
  return gulp.src([
    'spec/tests.webpack.js'
  ],
    {'read': false}).pipe(
    karma.server({
      configFile: __dirname + '/karma.conf.js',
      'singleRun': false
    })
  );
});
```

- 4. Update the files path in karma.conf.js to point to tests.webpack.js

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

- 5. Start (or restart) the Karma server and 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
- 2. 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!" />  
    );  
  });  
});
```

- 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);  
});
```

- 4. Run your tests to verify that the new spec fails.

```
MINGW64:/c:/Users/IEUser/WebstormProjects/react-training
IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab18)
$ gulp
[10:30:37] Using gulpfile ~/WebstormProjects/react-training/gulpfile.js
[10:30:37] Starting 'default'...
[10:30:37] Starting 'version'...
[10:30:37] Starting 'eslint'...
Checking node version:
[10:30:37] Finished 'version' after 2.75 ms
[10:30:43] Finished 'eslint' after 6.09 s
[10:30:43] Starting 'test'...
[2016-06-08 10:30:43.160] [DEBUG] config - Loading config C:\Users\IEUser\WebstormProjects\react-training\karma.conf.js
Firefox 45.0.0 (Windows 10 0.0.0) Poll Header prints a message FAILED
  TypeError: Object doesn't support property or method 'findRenderedComponentWithTag'
    at Anonymous function (C:/Users/IEUser/WebstormProjects/react-training/spec/tests.webpack.js:21261:10)
Chrome 51.0.2704 (Windows 10 0.0.0) Poll Header prints a message FAILED
  TypeError: _reactAddonsTestUtils2.default.findRenderedComponentWithTag is not a function
    at Object.<anonymous> (C:/Users/IEUser/WebstormProjects/react-training/spec/tests.webpack.js:21261:54)
Firefox 45.0.0 (Windows 10 0.0.0): Executed 7 of 7 (1 FAILED) (0.365 secs / 0.252 secs)
Chrome 51.0.2704 (Windows 10 0.0.0): Executed 7 of 7 (1 FAILED) (0.178 secs / 0.122 secs)
TOTAL: 2 FAILED, 12 SUCCESS
[10:30:45] 'test' errored after 2.18 s
[10:30:45] Error in plugin 'gulp-karma-runner.runner'
Message:
  Failed Unit Tests!
Details:
  domainEmitter: [object Object]
  domain: [object Object]
  domainThrown: false
[10:30:45] 'default' errored after 8.28 s
IEUser@MSEEDGEWIN10 MINGW64 ~/WebstormProjects/react-training (lab18)
$ |
```

- 5. Open PollHeader.js
- 6. Replace the text between <h1> and </h1> with a prop:

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

- 7. Run the tests again to verify that it passes.
- 8. 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().

- 9. 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(props) {
    super(props);
    this.state = {
    }
  }
}
```

- 10. For this first version, we're going to set the initial state of the application inside the constructor. Create properties inside the state object for the following:

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

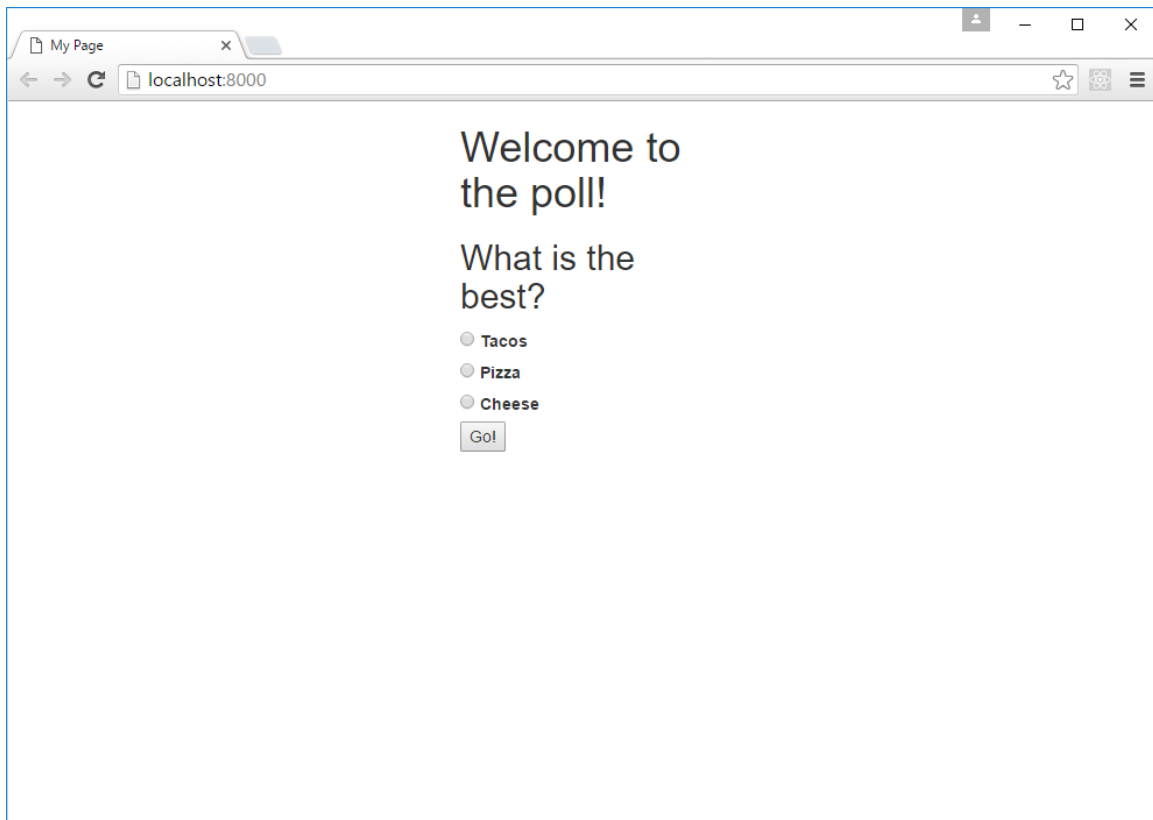
- 11. Set the values of the properties in the state object to any values you like. For example,

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

- 12. 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} />
...
```

- 13. Run your tests to confirm that the child components all receive and print out their props correctly.
- 14. Build and run your app in a browser (using **gulp build**).



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

- 15. Create a new task in your gulpfile called watch. Make sure that it comes after the webpack and copy tasks, since it's dependent on them.

```
//watch
gulp.task('watch', function() {
  gulp.watch(['src/**/*.js',
    'src/*.html'], gulp.series('webpack', 'copy'));
});
```

- 16. Start the server with `gulp run` then open a new console window and run `gulp watch`.
- 17. Open **PollContainer.js** and make some changes to the question, header, or answers. Save the file and return to your web browser.

After the build task completes, you will see the changes reflected in your browser.

- 18. Make the run and watch tasks run in parallel at the end of the build task:

```
gulp.task('build',  
  gulp.series('clean',  
    gulp.parallel('webpack', 'copy'),  
    gulp.parallel('run', 'watch')));
```

Lab 19 - Adding Style to React Components

Next, we'll add some styles to our components to make things look a little better. React recommends using inline styles, specified using objects. For the most part, we'll be using Bootstrap classes, but we'll also implement a few custom styles using this inline method.

- 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 `<div>` with the `className="jumbotron"` above the div with `className="row"` in `PollContainer.js` and move the `PollHeader` component into it.
- 4. Your `PollContainer` JSX should now look like this:

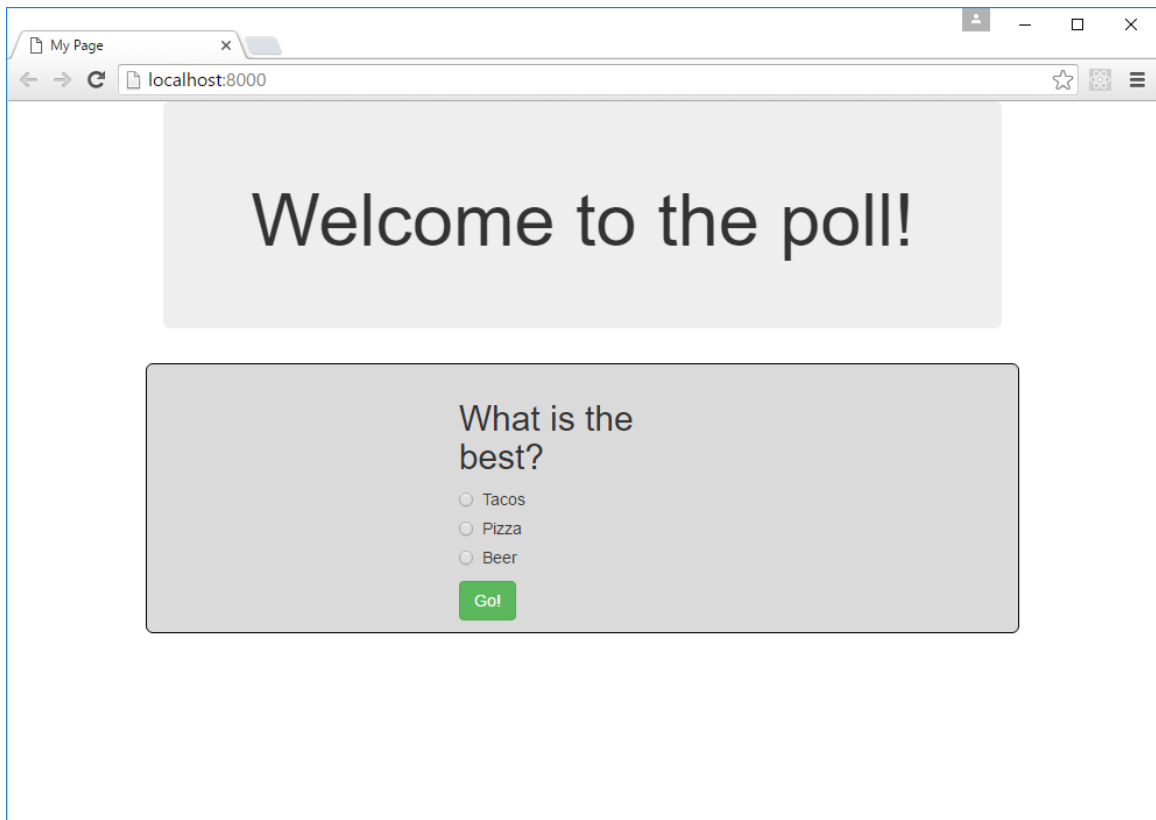
```
<div className="container">
  <div className="jumbotron">
    <PollHeader text={this.state.header} />
  </div>
  <div className="row">
    <div className="col-sm-4 col-sm-offset-4">
      <form>
        <PollQuestion text={this.state.question}/>
        <PollAnswer text={this.state.answer1}/>
  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

- 5. 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'
};
```

- 6. Add `className="text-center"` to the `<h1>` in `PollHeader`

Make additional CSS changes as time allows and as you wish.



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 **.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;

```

- 6. Next, we'll make some changes to the `RadioButton` component so that we can pass values and checked state into the component. 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 `<PollAnswer>` 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

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. At a later date, 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. Install **json-loader**:

```
npm install --save-dev json-loader
```

- 7. Set up the new loader in `webpack.conf` and in the webpack section of `karma.conf.js`.

Note: after this step, you should have two loaders.

```
{ test: /\.json$/, loader: 'json' },
```

- 8. Import **data.json** into `PollContainer`.

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

- 9. 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
```

- 10. Look through each component and find things that can be improved or simplified as time permits.
- 11. Check whether your unit 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.

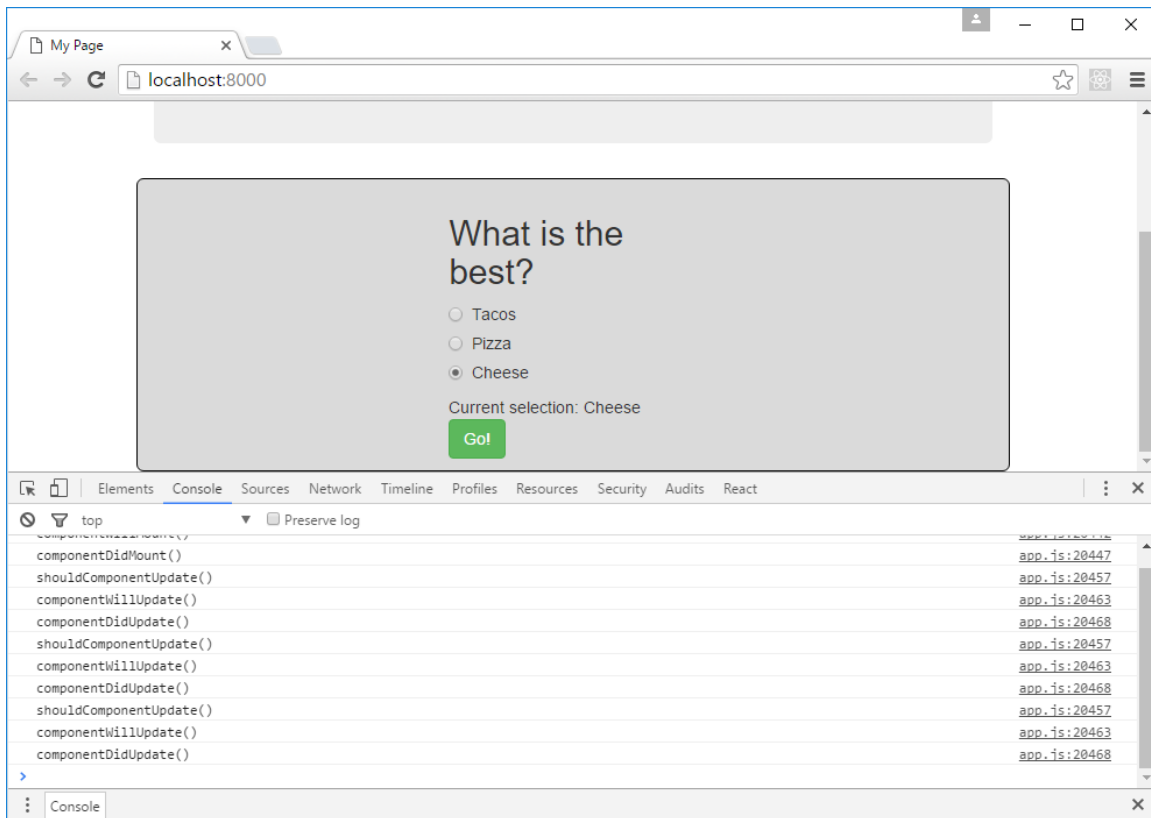
- 1. In PollContainer, log a message to the console when each of the life cycle events occurs.

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

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

- 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:8000/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. Update **gulpfile.js** to copy the **data.json** file to the **dist** directory during the build.

```

//copy html files to dist
gulp.task('copy', function(done) {
  gulp.src('src/*.html').pipe(gulp.dest(DIST));
  gulp.src('src/data/*').pipe(gulp.dest(DIST+"/data"));
  done();
});

```

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

```

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

```

- 9. Remove the const that sets the value of choices in the render method.
- 10. Change the `<RadioButtonGroup>` element in **PollContainer** to pass `this.state.choices` to the **RadioButtonGroup** component.
- 11. Build, test, debug.
- 12. Create a new function and 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__` by default, so we'll use this convention to make things simple.

- 1. Create a subdirectory of `components` named `__tests__`.
- 2. Create a file named **`PollSubmitButtonSpec.js`** inside `__tests__`.
- 3. Inside `PollSubmitButtonSpec.js`, enter the following:

```
import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';

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

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

  it('renders without a problem', function () {

    var pollsubmitbutton = TestUtils
      .renderIntoDocument(<PollSubmitButton />);

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

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

- 4. Install `jest`, `babel-jest`, and `enzyme`

```
npm install --save-dev jest jest-cli babel-jest
```

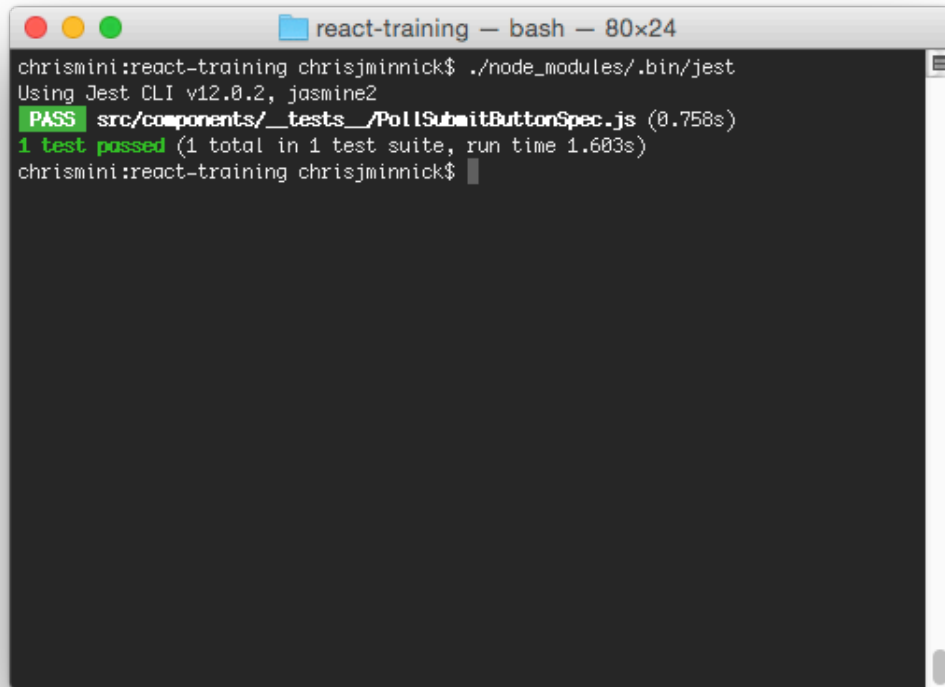
- 5. Create a file named **`.babelrc`** in your `src` directory, containing the following object:

```
{
  "presets": ["es2015", "react"]
}
```

- 6. 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.

A terminal window titled 'react-training — bash — 80x24' showing the execution of Jest. The user runs './node_modules/.bin/jest' in the directory 'chrismini:react-training chrisjminnick\$'. The output shows 'Using Jest CLI v12.0.2, jasmine2', followed by a green 'PASS' for 'src/components/__tests__/PollSubmitButtonSpec.js (0.758s)'. Below this, it says '1 test passed (1 total in 1 test suite, run time 1.603s)'. The prompt returns to 'chrismini:react-training chrisjminnick\$'.

- 7. Require `jest-cli` in `gulpfile.js` as `jest`.
- 8. Create a new task called `'jest'` in `gulpfile.js`

```
//Run jest
gulp.task('jest', function(done) {
  var onComplete = function (result) {
    if (!result) {
      console.log('Jest tests failed!');
    }
    done();
  };
  jest.runCLI({}, __dirname, onComplete);
});
```

- 9. Run `gulp jest`

- 10. Create the following new spec in PollSubmitButtonSpec.js:

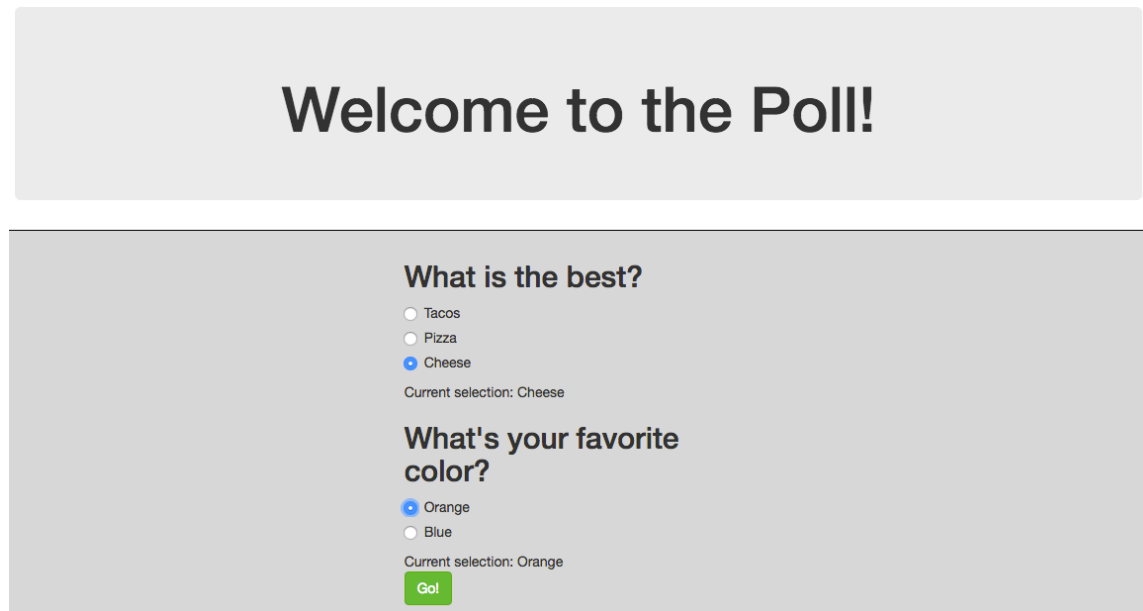
```
it('calls handler function on click', function () {  
  
  var PollSubmitButton =  
    require('../PollSubmitButton').default;  
  
  var handleClick = jest.genMockFunction();  
  
  var pollsubmitbutton = TestUtils  
    .renderIntoDocument(  
      <PollSubmitButton  
        question={0}  
        handleClick={handleClick}  
      />;  
  
  var buttonInstance = ReactDOM.findDOMNode(pollsubmitbutton);  
  
  TestUtils.Simulate.click(buttonInstance);  
  
  expect(handleClick).toBeCalled();  
  
  var numberOfCallsMadeIntoMockFunction =  
    handleClick.mock.calls.length;  
  
  expect(numberOfCallsMadeIntoMockFunction).toBe(1);  
});
```

- 11. Run gulp jest to confirm that the test fails.
- 12. Write the code to make the test pass.
- 13. Test to confirm that the test passes.
- 14. 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 end result will look like this:



The screenshot shows a web application interface with a light gray background. At the top, a large light gray box contains the text "Welcome to the Poll!". Below this, a white box contains two poll questions. The first question is "What is the best?" with three radio button options: "Tacos", "Pizza", and "Cheese". The "Cheese" option is selected, indicated by a blue dot. Below the options, it says "Current selection: Cheese". The second question is "What's your favorite color?" with two radio button options: "Orange" and "Blue". The "Orange" option is selected, indicated by a blue dot. Below the options, it says "Current selection: Orange". At the bottom of the white box is a green button with the text "Go!" in white.

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-es6-labs/tree/master/lab25>

Lab 26 - React Router

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

NOTE: These instructions (and subsequent labs) apply to version 3.0.0 of React Router. 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
```

□ 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>
```

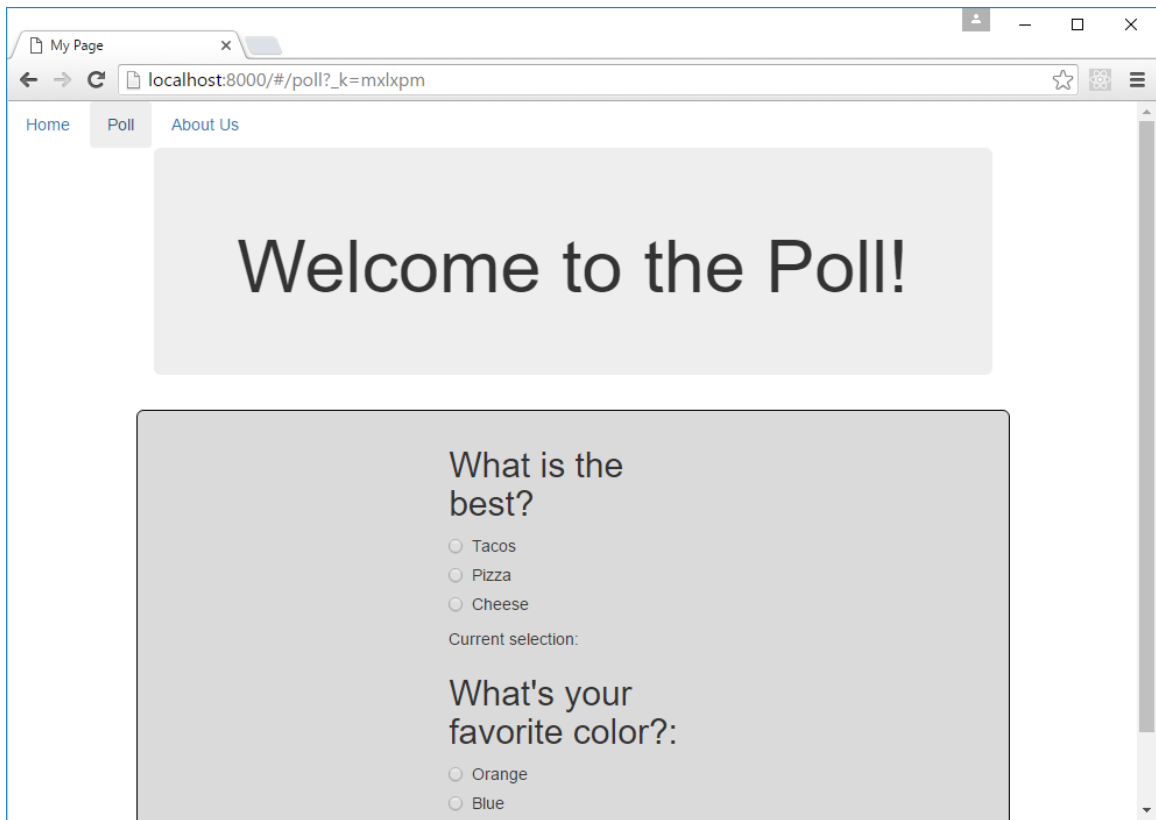
□ 9. Create the new routes in scripts/app.js, nested inside the App route

```
<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 Improvements

In this lab, we'll go deeper into React Router and make some improvements in how the app uses it.

- 1. In scripts/app.js, instead of importing all of react-dom, we can just import render.

```
import { render } from 'react-dom';
```

- 2. Update the render method to reflect the new import. Change it from ReactDOM.render to just render.
- 3. Import browserHistory instead of hashHistory and change the history object in the Router component accordingly.
- 4. Import IndexRoute from react-router. IndexRoute will tell the application which component to render on the homepage. Your react-router import should now look like this:

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

- 5. Create a separate router component, rather than having it directly in the render.

```
const router = (  
  <Router history = {browserHistory}>  
  
  </Router>  
);
```

- 6. Make a Route for the App component and an IndexRoute inside it that will render the PollContainer on the homepage.

```
const router = (  
  <Router history = {browserHistory}>  
    <Route path="/" component={App}>  
      <IndexRoute component={PollContainer} />  
    </Route>  
  </Router>  
);
```

- 7. Create a nested route with a path of /about inside the top-level route.

```
const router = (  
  <Router history = {browserHistory}>  
    <Route path="/" component={App}>  
      <IndexRoute component={PollContainer} />  
      <Route path="/about" component={AboutUs} />  
    </Route>  
  </Router>  
);
```

```
    </Route>
  </Router>
);
```

- 8. Open `containers/App.js` and replace `{this.props.children}` with a call to `React.cloneElement`, so that props can be passed to the children.

```
{React.cloneElement(this.props.children, this.props)}
```

- 9. Inside `scripts/app.js`, pass the router component into the `render` method instead of what's there now.

```
render(router, document.getElementById('app'));
```

- 10. Inside `containers/App.js`, remove the navigation item for `/poll` since we now have the poll displaying on the homepage.

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.

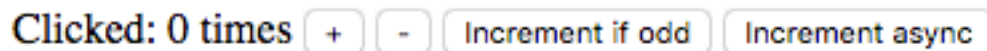
```
git clone https://github.com/reactjs/redux.git
```

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

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

- 3. Install and build the app to see how it works.

```
npm install
npm start
open http://localhost:3000
```



- 4. Stop the counter example by pressing Control - C.
- 5. Make a copy of the counter example and name it **thermometer**.

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

- 6. In the test directory inside your new thermometer directory, 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 a number of changes to improve the app in general.

- 1. Install Redux and the bindings for React and React Router.

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

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

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

- 4. Import `syncHistoryWithStore` from `react-router-redux`.

```
import {syncHistoryWithStore} from 'react-router-redux';
```

- 5. Import `browserHistory` from `react-router`

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

- 6. 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.

- 7. Import the root reducer into `store.js`.

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

- 8. 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;
```

- 9. Inside `store.js`, import the questions.

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

- 10. Inside `store.js`, create an object for the default data.

```

const defaultState = {
  questions,
  checkedValue:[]
}

```

- 11. Create the store.

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

- 12. Sync the history with the store and export it.

```
export const history = syncHistoryWithStore(browserHistory,
store);
```

- 13. Export the store.

```
export default store;
```

- 14. 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`.

- 15. Create a new action creator inside `actionCreators.js`

```

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

```



```
}
```

- 16. 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`.
- 17. 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;
```

- 18. Make another reducer for the `checkedValue` inside `checkedValue.js`

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

```
export default checkedValue;
```

- 19. Write your root reducer in `index.js`

```
import { combineReducers } from 'redux';  
import { routerReducer } from 'react-router-redux';  
  
import questions from '../questions';  
import checkedValue from '../checkedValue';  
  
const rootReducer =  
  combineReducers({questions, checkedValue, routing: routerReducer });  
  
export default rootReducer;
```

- 20. The next step is to link our store to our application. Open `scripts/app.js` and import `Provider` from `react-redux`.

```
import { Provider } from 'react-redux';
```

- 21. Import your store and browser history.

```
import store, { history } from '../store';
```

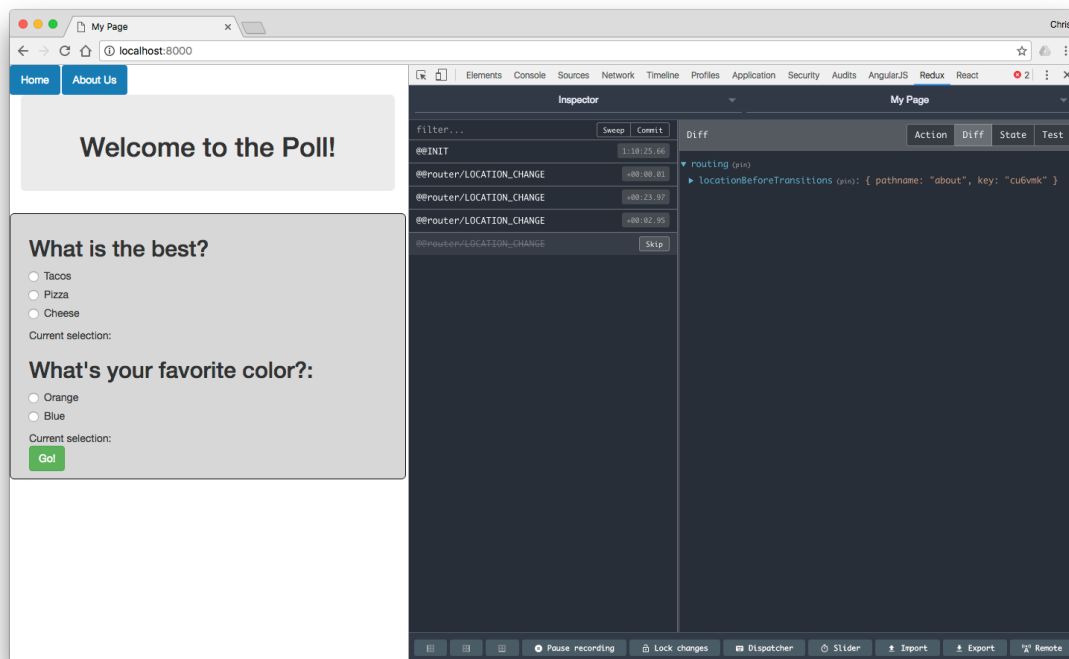
- 22. Install the Redux developer tools in your Chrome browser by going to the Chrome Web Store here:

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

- 23. Enable the Redux DevTools in your browser by adding a third parameter to your createStore method:

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

- 24. Build your app and open the Chrome Developer tools and switch to the Redux tools. Try navigating between the two different routes (Home and About) and watch what happens in the Redux DevTools.
- 25. Click on the Actions in the DevTools to skip and re-apply them and watch how it affects the browser window.



- 26. Go to the Console window and examine the log messages that are returned. You should see the `LOCATION_CHANGE` action, as well as the logged state from the reducers.
- 27. Inside `scripts/app.js`, wrap the `<Router>` in a `<Provider>` element and pass `<Provider>` a store prop that's equal to `store` (the one we created and just imported).

```

const router = (
  <Provider store={store}>
    <Router history = {browserHistory}>
      <Route path="/" component={App}>
        <IndexRoute component={PollContainer} />
        <Route path="/about" component={AboutUs} />
      </Route>
    </Router>
  </Provider>
);

```

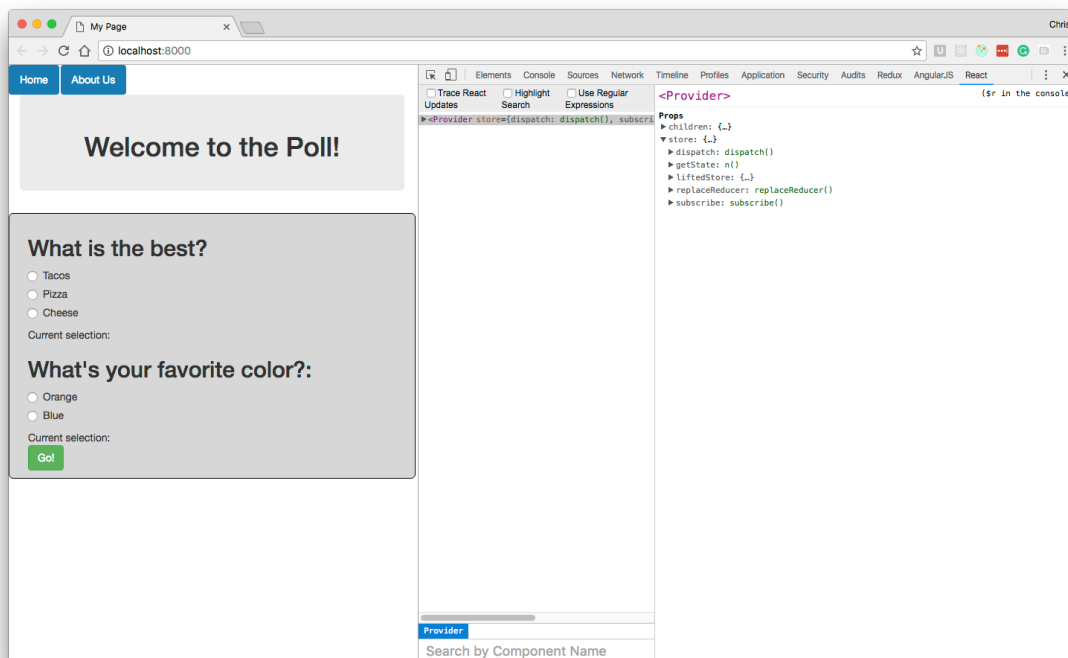
- 28. Change `{browserHistory}` to just `{history}` to use the location history we just imported.

```

<Router history = {history}>

```

- 29. Go to your browser and open the React Dev Tools. You should now see the Provider component with the store inside of it.



- 30. Open the console and type the following to view the state of the Provider.

```

$r.store.getState();

```

```

> $r
< ▶ Provider {props: Object, context: Object, refs: Object, updater: Object, store: Object...}
> $r.store
< ▶ Object {liftedStore: Object}
> $r.store.getState()
< ▼ Object 1
  ▶ checkedValue: Array[0]
  ▶ questions: Array[2]
  ▶ routing: Object
  ▶ __proto__: Object
>

```

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.

In order to use Connect, we need to create a new container component for it. We'll replace our current App component with this new component and move the presentational component we're currently calling App into a new component named Main.

- 31. Create a new file inside `containers/` named `Main.js`.
- 32. Copy the contents of `App.js` into `Main.js`, and rename the component to `Main`. Make sure to change the name at the bottom (in the export statement) too.
- 33. Delete the contents of `containers/App.js`.
- 34. Import `bindActionCreators` from `redux` into `App.js`
- 35. Import `connect` from `react-redux`.
- 36. Import `*` as `actionCreators` from `../actions/`.
- 37. Import `Main` from `'./Main'`;

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 Main from './Main';

```

- 38. Create the App component, using `connect()`, and immediately call it against `Main`.

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

- 39. 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
  }
}
```

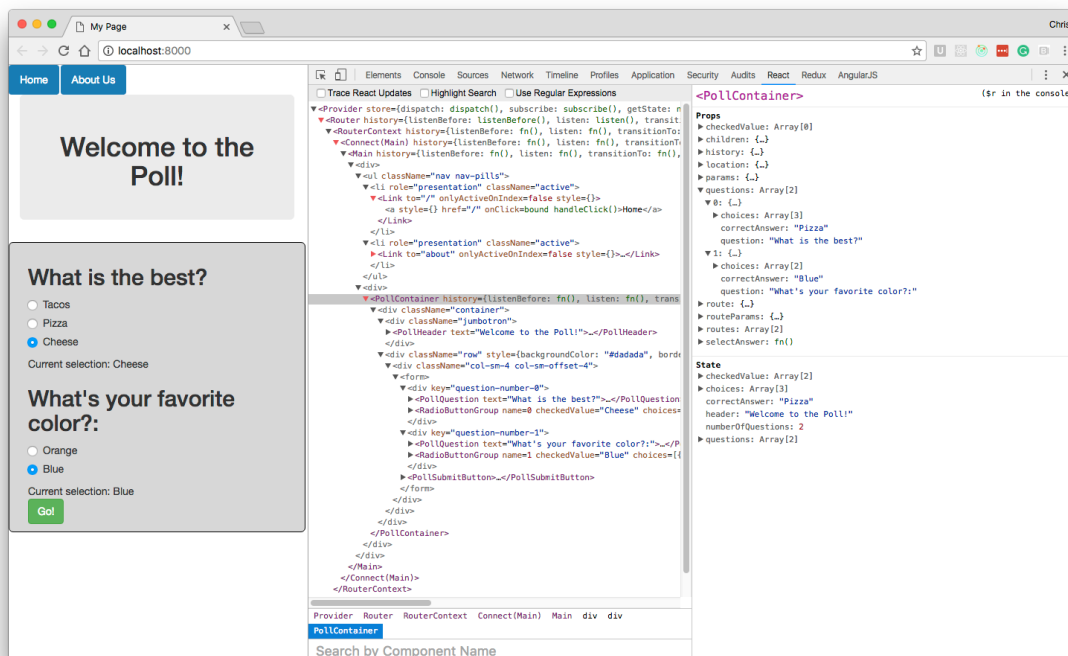
- 40. Write the `mapDispatchToProps` function, which will allow you to use `dispatch` in your components in response to events.

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

- 41. Export the App component.

```
export default App;
```

Now, if you run the application and inspect it in the React DevTools, you'll see that the `questions` and `checkedValue`, as well as the `selectAnswer` action creator are all available to `Main`. The `cloneElement` inside `Main` passes the props down to its child, so they're also now available to the `PollContainer`.



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

- 42. 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.
- 43. Inside the render method, set the value of the `questionsArray` to `this.props.questions`
- 44. Change the value of the `checkedValue` prop passed into `<RadioButtonGroup>` to `this.props.checkedValue[questionNumber]`
- 45. 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.

- 46. 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.

- 47. 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':
```

- 48. 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));
```

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

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

- 50. Add another question to the questions array and confirm that the application still works.
- 51. Try answering the third question first and notice that it produces an unexpected result. Can you figure out why and how to fix it?

Challenge Steps:

- 52. Try modifying questions.js so that it changes periodically.
- 53. 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:

Cost	50
Number Of Passes	2
Initial Distance	1000
Increment	100

Here are the results!

visit #	distance	\$ per km	total
1	1000	50.00	1000
2	1100	45.45	2100

Total Km: 2100

Starter Project: <https://github.com/watzthisco/tdd-react-es6-labs/tree/lab30>

Example solutions:

<https://github.com/watzthisco/tdd-react-es6-labs/tree/lab30solutions>

