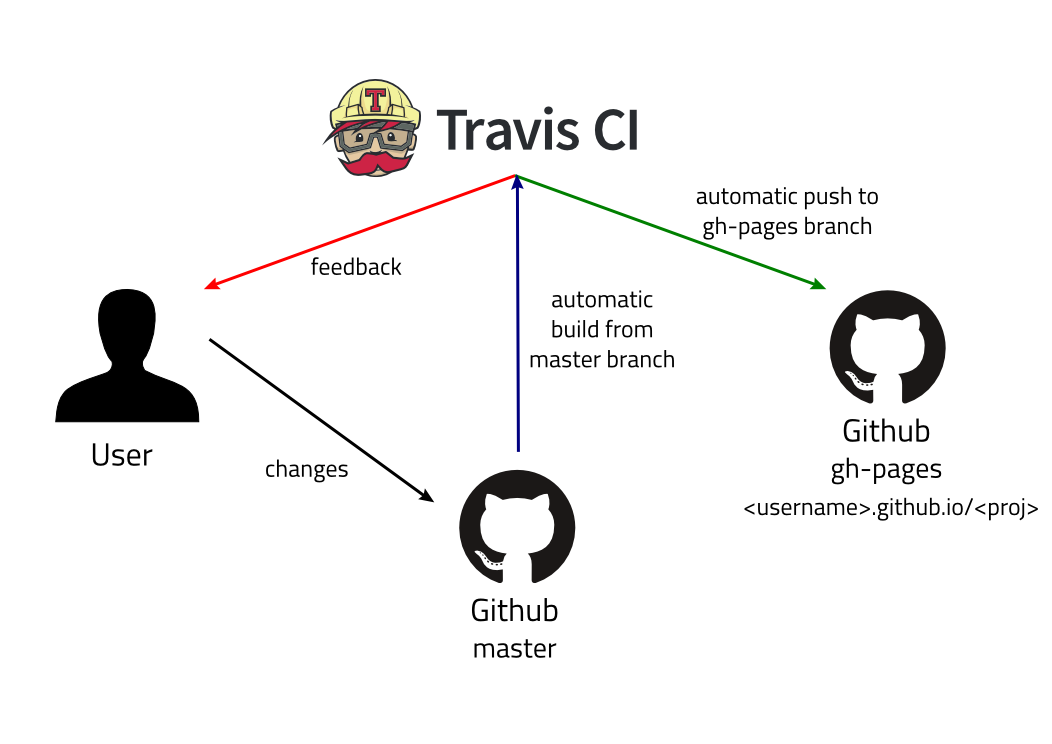
# Lab 05 - Hosting Single Page Apps with GitHub Pages

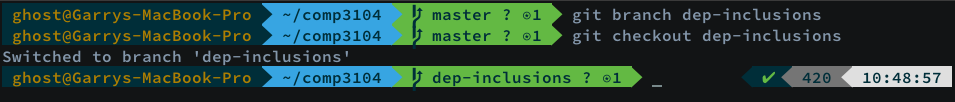


At this point, your project directory should have a package.json with a “scripts” key in it containing an object with a “test” key who's value simply echoes out the fact that we currently do not have any tests defined yet. We’ll get to that but first, let’s add some project dependencies for us to get our SPA up and running.

This is where we’re going to start making some changes that could potentially place our application in a broken state. That being said, it’s considered best practice to create a separate branch for us to work off to mitigate any potential damage to our application. Let’s create a branch called **dep-inclusions**.

We know to create a branch, it’s just a matter of `**git branch NAME\_HERE**`

Let’s run this command and then switch to that branch. Please refer to the git commands from our second lab or follow along with the images I’ve included below.



You’ll see after checking out the “**dep-inclusions**” branch, my terminal has been updated to reflect the change. My shell might look different than yours and should curiosity have you wondering why, it’s due to the shell flavor I’m using. I’m running [oh-my-zsh](https://ohmyz.sh/) as it really improves the shell experience. Instead of bash, it’s running zsh. Apple announced with their next macOS update that they’ll be moving from bash to zsh, so it’s worth me mentioning but not important for you to use. If you’d like to learn more about it, please check out the supplementary-material folder as I’ve left some material in there for you to look at if you’re interested in switching shells.

Now that we’re on a new branch, we can freely make changes without impacting our master/main branch which should always reflect our Production source code. Let’s start experimenting with some changes to the package.json. We’re going to start by adding a “**dependencies**” key which contains some third-party libraries we’ll be using for this course. Feel free to ***carefully*** copy and paste what’s below, I’m also providing an image to help ensure there is no syntactical errors.

`

"dependencies": {

"gh-pages": "^2.0.1",

"react": "^16.8.3",

"react-dom": "^16.8.3",

"react-scripts": "2.1.5",

"travis": "^0.1.1",

"yarn": "^1.13.0"

}

`



You’ll notice above that I chose to place the “dependencies” key as the last key in my package.json which falls right under the “homepage” key. It might look a little bit different for you depending if NPM generated the package.json in another order. What you’ll want to be cognizant of, is to ensure that the preceeding key includes a `,` before the “dependencies” and that the newly added JSON key is within the main JSON object.

With these dependencies, comes abstracted a few new “**scripts**” which we’ll be able to run. Before being having the ability to run these script tasks, we’ll have to include them in the JSON object. Again, feel free to carefully copy and paste but ensure that it looks like the image I’ve also provided.

`

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"predeploy": "yarn run build",

"deploy": "gh-pages -d build",

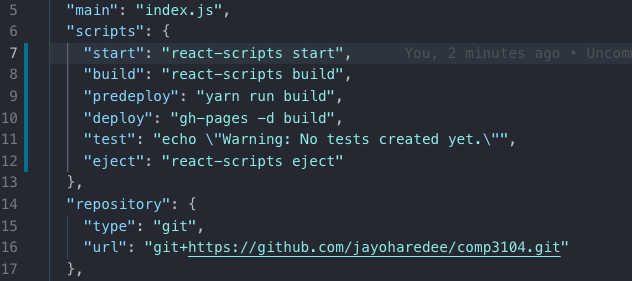
"test": "echo \"Warning: No tests created yet.\"",

"eject": "react-scripts eject"

},

`

Please see the next page for the image of my package.json.

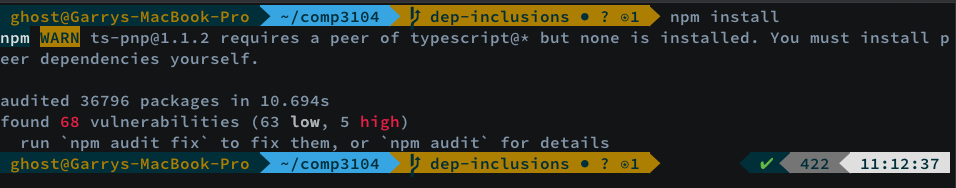


Looking at line 6, you can see the “scripts” key containing an object with some additional keys other than our initial “test” key. The values assigned are some script commands that we get with our newly added dependencies. One of them being our build script assign to our “build” key in the “scripts” object. Before being able to add anything, we need to install our dependencies. Using NPM, we’ll run a command which allow us to do just that.

`**npm install**`

After running the above command at the command line, we should see an output like what’s depicted below after the dependencies are successfully installed.

There might be some output regarding vulnerabilities. We won’t be addressing them at this time but we will be covering solutions in a future lab. For right now, please disregard.



When **running an `npm install` the resulting command execution will always create a new folder in the project directory called “node\_modules”** this is the directory where all our projects dependencies live. We should also have a **package-lock.json which is npm’s way of managing project dependencies**. It’s important as a DevOps Engineer to be aware of this folder as we’ll have to ensure that it’s optimally cached and handled in our pipeline process.

Now that our dependencies are installed, **it’s time to grab two folders from our labs supplementary materials.** These folders are both the “**public**” and “**src**” folder which contain the application source provided to us for use in this course. Let’s copy them into our project directory.

After our “**public**” and “**src**” directories are moved into our **week05\_comp3104\_lab\_exec** project, we need to make one last change to our package.json. This is to include the “eslintConfig” key along with our “browserslist” key. The “eslintConfig” key will be used at a later time when we’re working with sequences in our pipeline and the “browserslist” instructs the build process which browsers we’ll be supporting. I’m placing the following JSON keys under my “dependencies” key.

`

"eslintConfig": {

"extends": "react-app"

},

"browserslist": [

">0.2%",

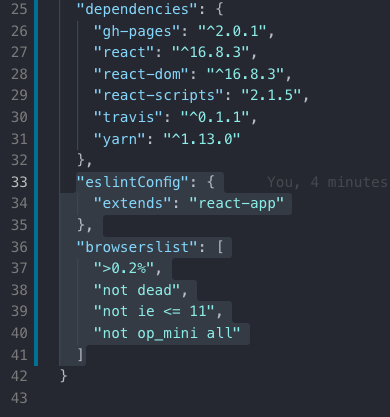
"not dead",

"not ie <= 11",

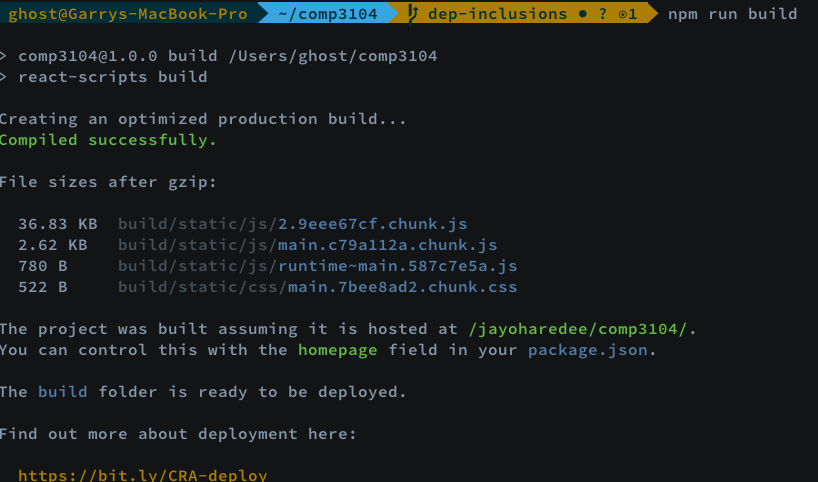
"not op\_mini all"

]

`

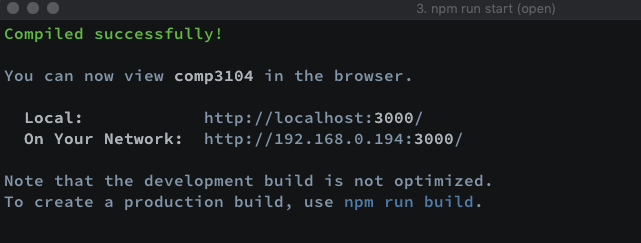


With the last of our package.json changes completed for the time being, let’s run our first application build from the CLI. When we run `**npm run build**` the resulting output should resemble the following:

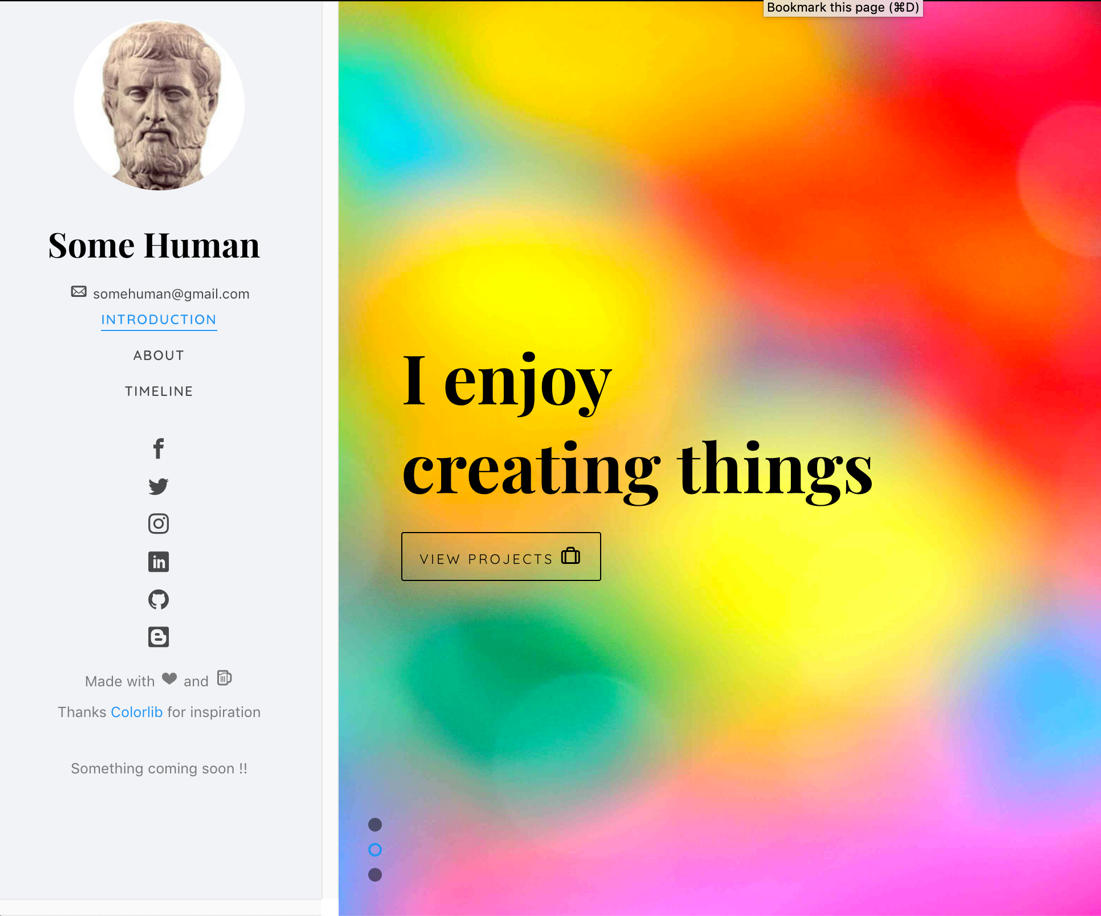


Underneath where it shows “Compiled successfully”, take a look at what the process managed for us; it took care of some compression and bundling for an optimized build which will improve the performance of our application due to the way in which the static assets are managed. We should now have the listed project dependencies in our build folder which we previously created.

After executing `**npm run start**` from the command line, we should see the following:



The project dependencies we added has gifted us with so many awesome capabilities. We were able to take care of bundling, minification, compression and now have a development server launched at the above addresses! A browser window should have opened for you but had it not, head over to <http://localhost:3000/> or whatever address is listed in your terminal. Accessing the page should show our web app on a local web server.



With our web application now launched, we’ll be ensuring that we have our changes checked into version control. Typically, it’s best practice to add changes for tracking as you make them. So applying this theory, I should have added our very first changes which were on the package.json after completing each change. Let’s follow that format below.

`**git add package.json**`

`**git commit -m ‘added project dependencies to our package.json’**`

After running our `npm install`, we know that npm creates a folder called node\_modules along with a package-lock.json to help manage our projects dependencies. Our .gitignore is set to ignore the node\_modules folder as we never want this reaching our remote repository. This is done for two reasons – the first reason is due to the size of the directory but most importantly, the fact that all we need is our package.json which includes our dependencies. When we run `npm install`, the directory is created for us so there is no need to carry the additional weight on a remote server.

Once `npm install` is completed, we can also commit our package-lock file.

`**git add package-lock.json**`

`**git commit -m ‘adding our dependency lock file’**`

We also have a newly added “src” and “public” folder which contains our applications source code. Let’s check those in as well. From the root of the **week05\_comp3104\_lab\_exec** directory, run the following commands:

`**git add src public**`

`**git commit -m ‘including application source’**`

The resulting git work should produce an output like the following:



If you didn’t want to do things step by step, we know by referncing our Git command list from lab 02, we could have also used the –a switch to add everything at once. Doing so would look something like this:

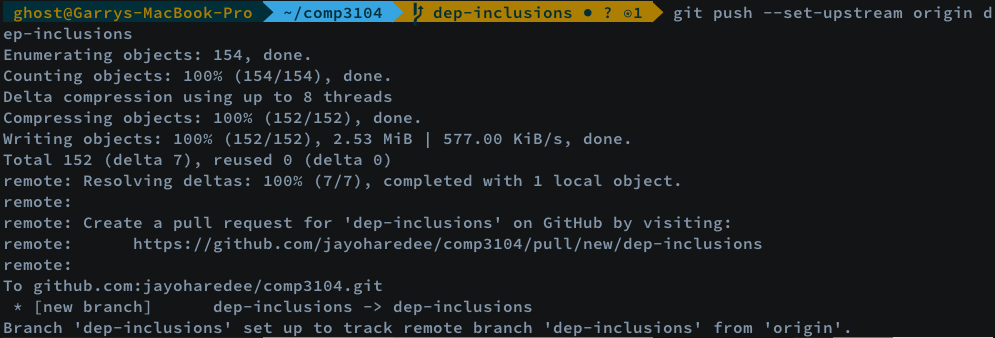
`**git add -a**`

`**git commit -m ‘adding all changes at once which includes package.json work and app src’**`

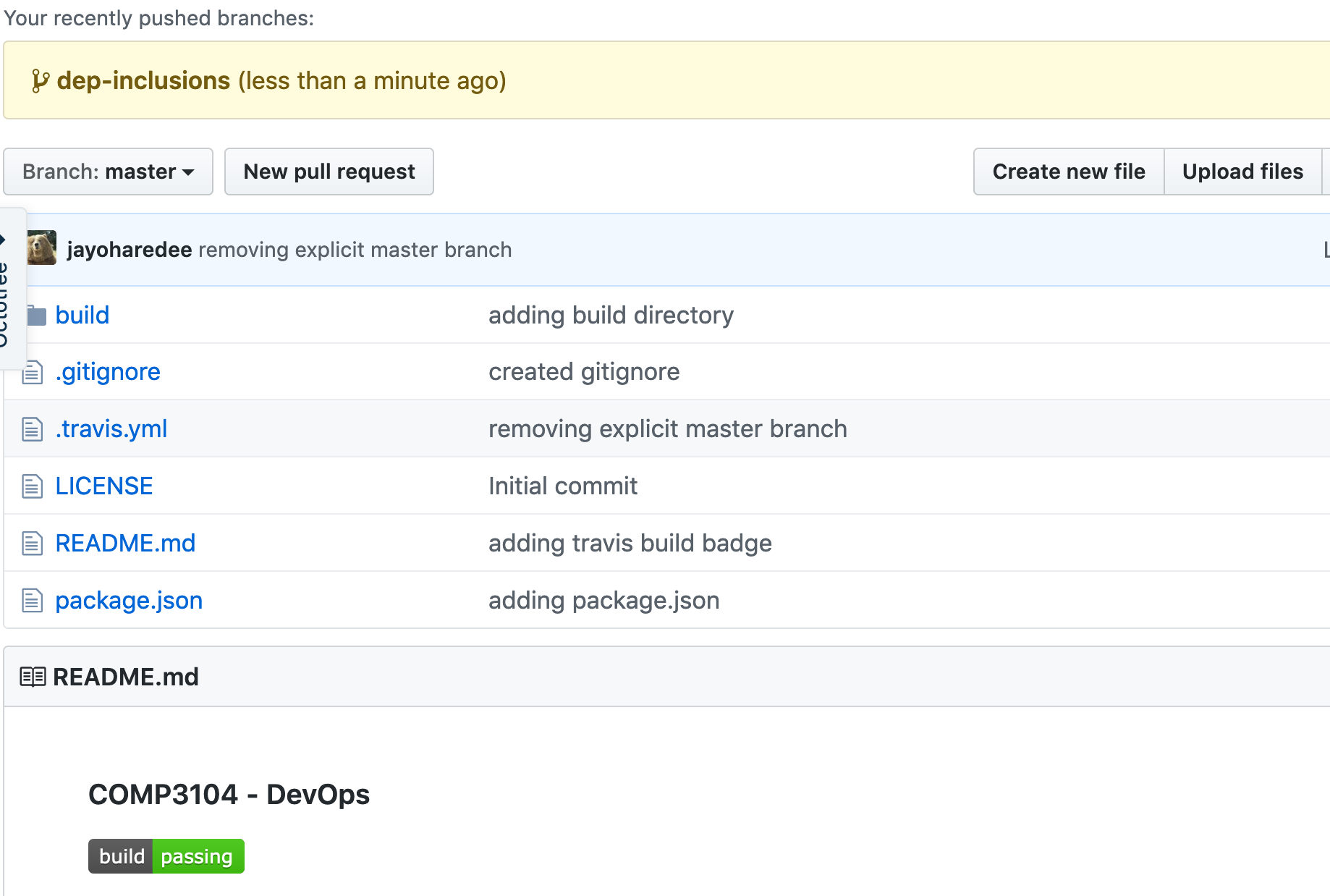
With these changes, let’s ensure that they don’t stay local and send them upstream to our remote repository. Since we’ve created a new branch called dep-inclusions, we need to let our remote repo know about it using the following command:

` **git push --set-upstream origin dep-inclusions** `

Our terminal should show the result of our push:



And if we check GitHub, we should see an indication of a new branch living there:



We now have our SPA successfully running and a separate branch in our remote repository including our changes. This was all completed while keeping our master/main branch untouched so if we did run into any issues, it wouldn’t have affected what’s on Production.

In our next lab, we’ll be performing our first merge into master/main and see how we can turn hosting on using GitHub Pages.