Promise chaining

Promise chaining is the idea of having multiple promises run in a sequence. For example, I want to take an address and convert that into coordinates, and take those coordinates and convert them into weather information; this is an example of needing to synchronize two things. Also, we can do that really easily using promise chaining.

In order to chain our promises, inside our success call we'll return a new promise. In our example, we can `return` a new promise by calling `asyncAdd` again. I'll call `asyncAdd` next to the `res` and `console.log` statements, passing in two arguments: the result, whatever the previous promise has returned, and some sort of new number; let's use `33`:

```
asyncAdd(5, 7).then((res) => {
 console.log('Result:', res);
 return asyncAdd(res, 33);
```

Now we're returning a promise so we can add my chaining onto it by calling the `then` method again. The `then` method will to get called after we close the closing parenthesis for our previous `then` method. This will also take one or more arguments. We can pass in a success handler, which will be a function and an error handler, which will also be a function:

```
 asyncAdd(5, 7).then((res) => {
   console.log('Result:', res);
   return asyncAdd(res, 33);
 }, (errorMessage) => {
   console.log(errorMessage);
 }).then(() => {

 }, () => {

 })
```

Now that we have our `then` callbacks set up, we can actually fill them out. Once again we will get a result; this will be the result of `5` plus `7`, which is `12`, plus `33`, which will be `45`. Then, we can print `console.log ('Should be 45')`. Next, we'll print the actual value from results variable:

```
}).then((res) => {
 console.log('Should be 45', res);
```

```
}, () => {
});
```

Now our error handler will also be the same. We'll have `errorMessage` and we'll print it to the screen using the `console.log`, printing `errorMessage`:

```
}).then((res) => {
 console.log('Should be 45', res);
}, (errorMessage) => {
 console.log(errorMessage);
});
```

Now what we have is some chaining. Our first then `callback` functions will fire based on the result of our first `asyncAdd` call. If it goes well, the first one will fire. If it goes poorly, the second function will fire. Our second then call will be based on the `asyncAdd` call, where we add `33`. This will let us chain the two results together, and we should get `45` printing to the screen. We'll save this file, which will restart things inside `nodemon`. Eventually, we'll get our two results: `12` and our `Should be 45`. As shown in the following code image, we get just that, `Result: 12` and `Should be 45`, printing to the screen:

```
[nodemon] restarting due to changes...
[nodemon] starting `node playground/promise.js`
Result: 12
Should be 45 45
[nodemon] clean exit - waiting for changes before restart
```

Error handling in promises chaining

Now when it comes to error handling, there are a few quirks; so, we'll simulate some errors. First up, let's simulate an error in our second `asyncAdd` call. We know we can do that by passing in a value that's not a number. In this case, let's wrap `33` inside quotes:

```
asyncAdd(5, 7).then((res) => {
  console.log('Result:', res);
  return asyncAdd(res, '33');
}, (errorMessage) => {
  console.log(errorMessage);
}).then((res) => {
  console.log('Should be 45', res);
}, (errorMessage) => {
  concole.log(errorMessage);
})
```

This will be a string and our call should `reject`. Now we can save the file and see what happens:

```
[nodemon] restarting due to changes...
[nodemon] starting `node playground/promise.js`
Result: 12
Arguments must be numbers
[nodemon] clean exit - waiting for changes before restart
```

We get `Result: 12`, then we get our error, `Arguments must be numbers`. Exactly as we expect, this is printing on the screen. Instead of getting `Should be 45`, we get our error message.

But things get a little trickier when something earlier on in the promise chain gets rejected. Let's swap `'33'` with the number `33`. Then let's replace `7` with the string `'7'`, as shown here:

```
asyncAdd(5, '7').then((res) => {
  console.log('Result:', res);
  return asyncAdd(res, 33);
}, (errorMessage) => {
  console.log(errorMessage);
}).then((res) => {
```

```
  console.log('Should be 45', res);
}, (errorMessage) => {
  concole.log(errorMessage);
})
```

This will cause our first promise to fail, which means we'll never see the result. We should see the error message printing to the screen, but that's not what will happen:

```
[nodemon] restarting due to changes...
[nodemon] starting `node playground/promise.js`
Arguments must be numbers
Should be 45 undefined
[nodemon] clean exit - waiting for changes before restart
```

When we restart, we do indeed get the error message printing to the screen, but then we also get `Should be 45 undefined`. The second then `console.log` is running because we provided an error handler in the second `asyncAdd` function. It's running the error handler. Then it says, *Okay, things must be good now we ran the error handler. Let's move on to the next then call calling the success case*.

The catch method

To fix the error, we can remove both of our error handlers from both the `then` calls, and replace them with a call at the very bottom, to a different method, which we'll call `.catch`:

```
asyncAdd(5, '7').then((res) => {
 console.log('Result:', res);
 return asyncAdd(res, 33);
}).then((res) => {
 console.log('Should be 45', res);
}).catch;
```

The catch promise method is similar to then, but it just takes one function. This is the error handler. As shown in the following code, we can specify one error handler if any of our promise calls fail. We'll take `errorMessage` and print it to the screen using `console.log(errorMessage)`:

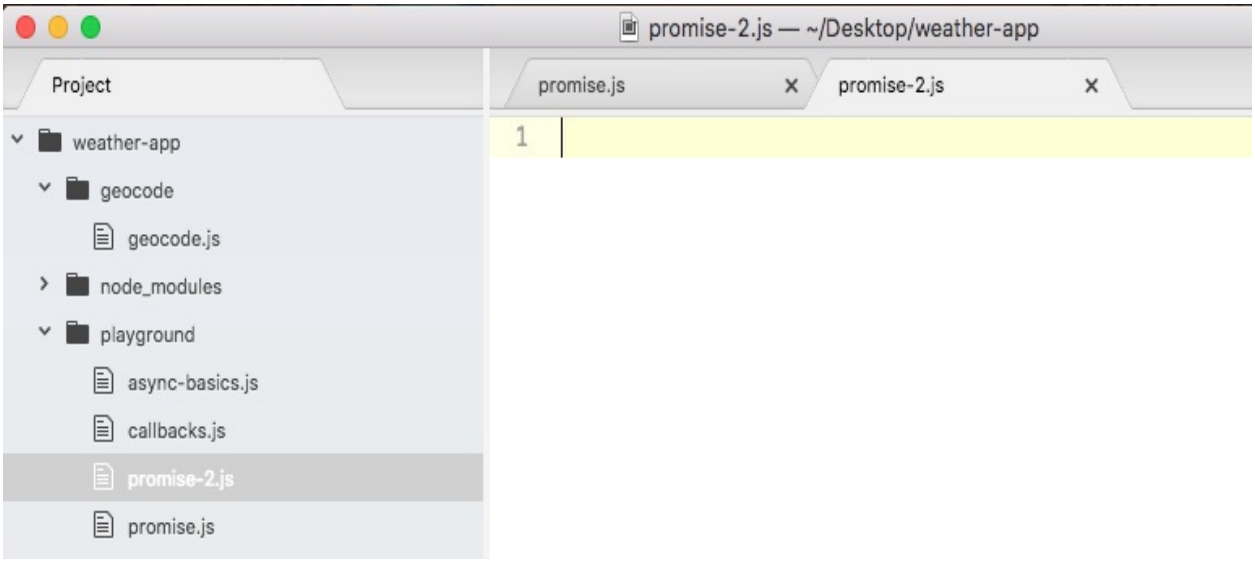
```
asyncAdd(5, '7').then((res) => {
 console.log('Result:', res);
 return asyncAdd(res, 33);
}).then((res) => {
 console.log('Should be 45', res);
}).catch((errorMessage) => {
 console.log(errorMessage)
});
```

For now though, if things are a little blurry that is okay, as long as you're starting to see exactly what we're doing. We're taking the result from one promise and passing it to a different one. In this case, the result works exactly as expected. The first promise fails, we get, `Arguments must be numbers` printing to the screen. Also, we don't get that broken statement where we try to print `45`, but we get undefined instead. Using catch, we can specify an error handler that will fire for all of our previous failures. This is exactly what we want.

The request library in promises

Now as I mentioned earlier, some libraries support promises while others don't. The request library does not support promises. We will make a function that wraps request, returning a promise. We'll use some functionalities from the `geocode.js` file from the previous chapter.

First, let's discuss a quick setup, and then we'll actually fill it out. In the `playground` folder, we can make a new file to store this, called `promise-2.js`:



We'll make a function called `geocodeAddress`. The `geocodeAddress` function will take the plain text address, and it will return a promise:

```
var geocodeAddress = (address) => {

};
```

The `geocodeAddress` function will return a promise. So if I pass in a ZIP code, such as `19146`, I would expect a promise to come back, which I can attach a `then` call to. This will let me wait for that request to finish. Right here, I'll tack on a call to `then`, passing in my two functions: the success handler for when the promise is fulfilled and the error handler for when the promise is rejected:

```
geocodeAddress('19146').then(() => {

```

```
}, () => {

})
```

Now when things go well, I'll expect the `location` object with the address, the `latitude`, and the `longitude`, and when things go poorly, I'll expect the error message:

```
geocodeAddress('19146').then((location) => {

}, (errorMessage) => {

})
```

When the error message happens, we'll just print it to the screen using `console.log(errorMessage)`. For now, when things go well and the success case runs, we'll just print that entire object using our pretty printing technique, `console.log`. Then, we'll call `JSON.stringify`, like we've done many times before, passing in the three arguments—the object, undefined for the filter method—which we'll never use in the book, and the number `2` for the number of spaces we'd like to use as our indentation:

```
geocodeAddress('19146').then((location) => {
 console.log(JSON.stringify(location, undefined, 2));
}, (errorMessage) => {
 console.log(errorMessage);
});
```

This is what we want to create, the function that lets this functionality work as expected. This `then` call should work as shown in the previous code.

To get started I'll return the promise by calling: `return new Promise`, passing in my constructor function:

```
var geocodeAddress = (address) => {
 return new Promise(() => {

 });
};
```

Inside the function, we'll add that call to request. Let's provide the `resolve` and `reject` arguments:

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

Now that we have our `Promise` set up, we can load in the request module on top of the code, creating a constant called `request` and setting that equal to the return result from `require('request')`:

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

var geocodeAddress = (address) => {
```

Next, we'll move into the `geocode.js` file, grab code inside the `geocodeAddress` function, and move it over into `promise-2 file`, inside of the constructor function:

```
const request = require('request');
var geocodeAddress = (address) => {
 return new Promise((resolve, reject) => {
 var encodedAddress = encodeURIComponent(address);

request({
 url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,
 json: true
 }, (error, response, body) => {
   if (error) {
   callback('Unable to connect to Google servers.');
 } else if (body.status === 'ZERO_RESULTS') {
   callback('Unable to find that address.');
 } else if (body.status === 'OK') {
   callback(undefined, {
     address: body.results[0].formatted_address,
     latitude: body.results[0].geometry.location.lat,
     longitude: body.results[0].geometry.location.lng
     });
    }
   });
 });
};
```

Now we are mostly good to go; we only need to change a few things. The first thing we need to do is to replace our error handlers. In the `if` block of the code, we have called our `callback` handler with one argument; instead, we'll call `reject`, because if this code runs, we want to `reject` the promise. We have the same thing in the next `else` block. We'll call `reject` if we get `ZERO_RESULTS`. This is indeed a failure, and we do not want to pretend we succeeded:

```
if (error) {
   reject('Unable to connect to Google servers.');
 } else if (body.status === 'ZERO_RESULTS') {
   reject('Unable to find that address.');
```

Now in the next `else` block, this is where things did go well; here we can call `resolve`. Also, we can remove the first argument, as we know `resolve` and `reject` only take one argument:

```
if (error) {
  reject('Unable to connect to Google servers.');
 } else if (body.status === 'ZERO_RESULTS') {
   reject('Unable to find that address.');
 } else if (body.status === 'OK') {
   resolve(
```

We are able to specify multiple values though, because we `resolve` an object with properties on it. Now that we have this in place, we are done. We can actually save our file, rerun it inside Terminal, and test things out.

Testing the request library

To test, we'll save the file, move into Terminal, and shut down `nodemon` for the `promise.js` file. We'll run `node` for the `promise.js` file. It's in the `playground` folder, and it's called `promise-2.js`:

```
node playground/promise-2.js
```

Now, when we run this program, we're actually making that HTTP request. As shown in the following code output, we can see the data comes back exactly as we expected:

```
Gary:weather-app Gary$ node playground/promise-2.js
{
  "address": "Philadelphia, PA 19146, USA",
  "latitude": 39.9396284,
  "longitude": -75.18663959999999
}
Gary:weather-app Gary$
```

We get our `address`, `latitude`, and `longitude` variables. This is fantastic! Now let's test to see what happens when we pass in an invalid address, something like 5 zeroes, which we've used before to simulate an error:

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

var geocodeAddress = (address) => {
  return new Promise((resolve, reject) => {
    var encodedAddress = encodeURIComponent(address);

  request({
   url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,
   json: true
 }, (error, response, body) => {
   if (error) {
     reject('Unable to connect to Google servers.');
   } else if (body.status === 'ZERO_RESULTS') {
     reject('Unable to find that address.');
   } else if (body.status === 'OK') {
     resolve({
       address: body.results[0].formatted_address,
       latitude: body.results[0].geometry.location.lat,
       longitude: body.results[0].geometry.location.lng
      });
     }
   });
```

```
  });
};
```

We'll save the file, rerun the program, and `Unable to find that address.` prints to the screen:

```
Gary:weather-app Gary$ node playground/promise-2.js
Unable to find that address.
Gary:weather-app Gary$
```

This happens only because we call `reject`. We will call `reject` inside of the `Promise` constructor function. We have our error handler, which prints the message to the screen. This is an example of how to take a library that does not support promises and wrap it in a promise, creating a promise ready function. In our case, that function is `geocodeAddress`.

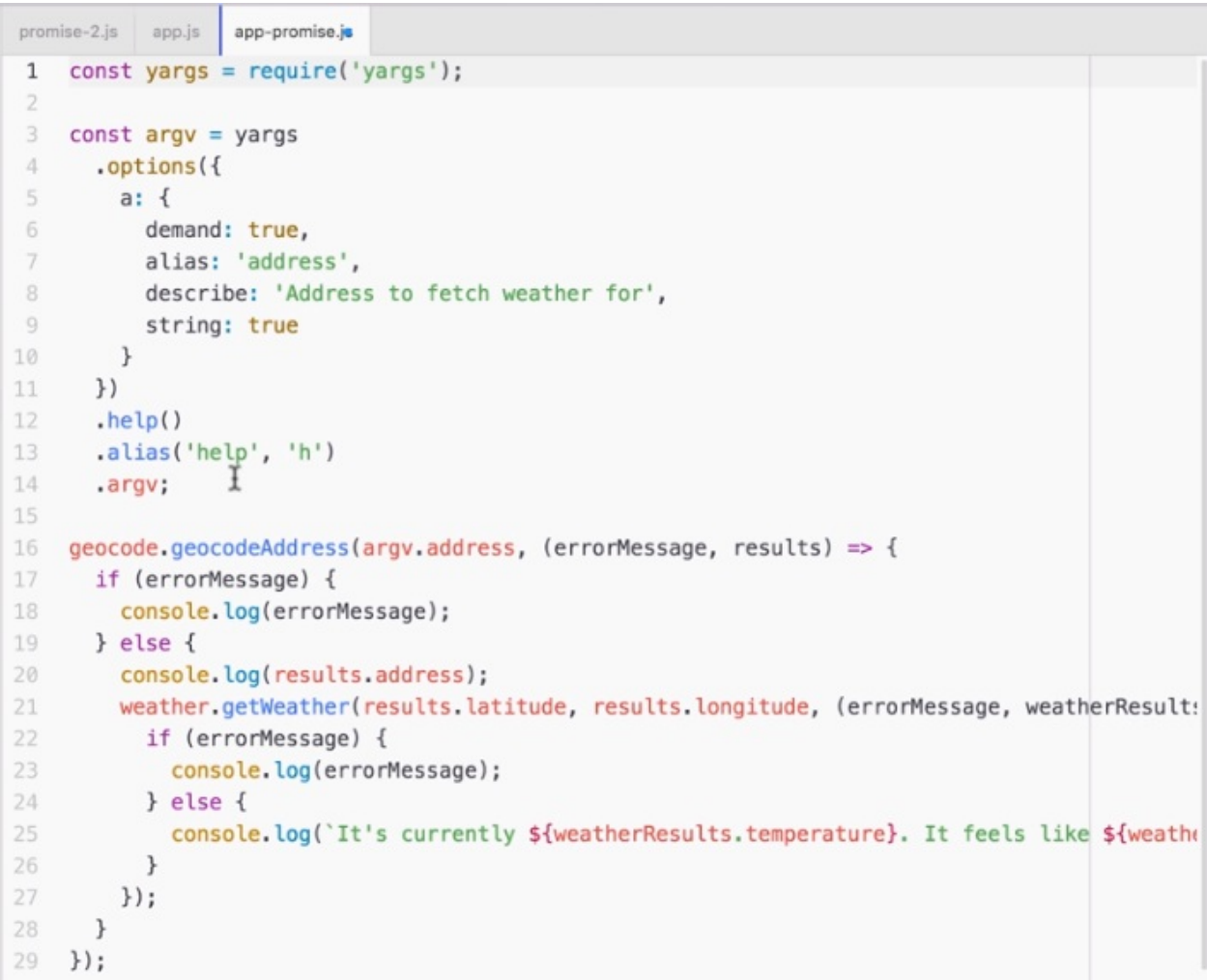
Weather app with promises

In this section, we'll learn how to use a library that has promises built in. We'll explore the axios library, which is really similar to request. Although, instead of using callbacks as request does, it uses promises. So we don't have to wrap our calls in promises to get that promise functionality. We'll actually be recreating the entire weather app in this section. We'll only have to write about 25 lines of code. We'll go through the entire process: taking the address, getting the coordinates, and then fetching the weather.

Fetching weather app code from the app.js file

To fetch weather app code from the app.js file, we'll duplicate `app.js`, because we configure `yargs` in the original `app.js` file and we'll want to carry the code over to the new project. There's no need to rewrite it. In the `weather` directory, we'll duplicate `app.js`, giving it a new name, `app-promise.js`.

Inside `app-promise.js`, before we add anything, let's rip some stuff out. We'll be ripping out the `geocode` and `weather` variable declarations. We'll not be requiring any files:

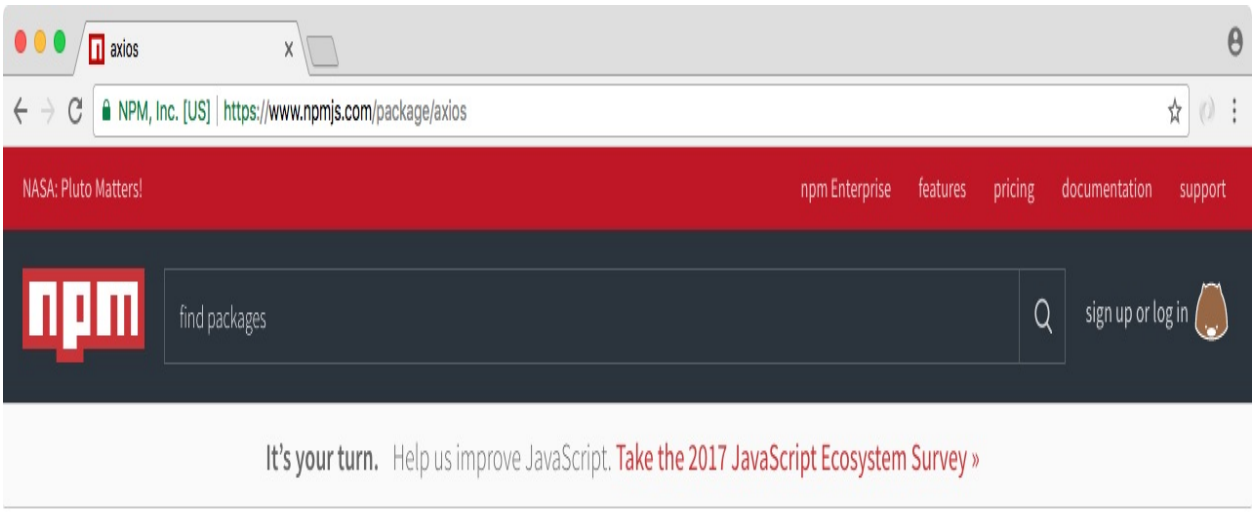


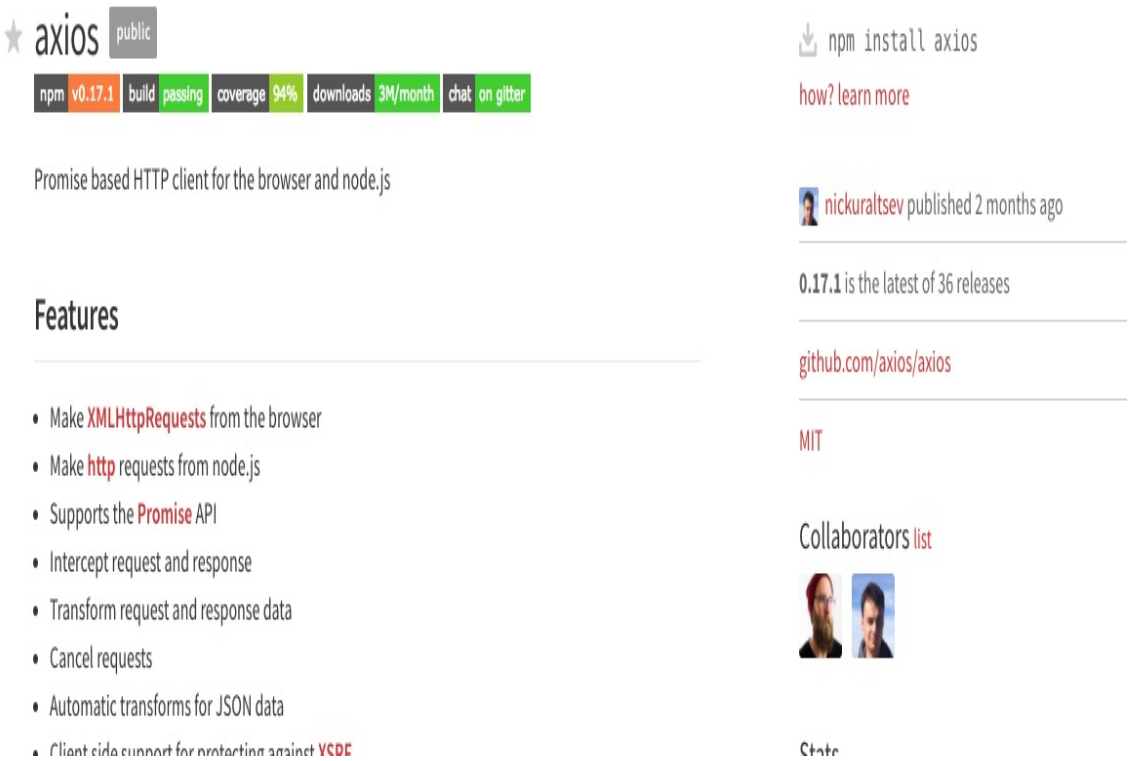
Then I'll remove everything after our `yargs` configuration, which in this case is just our call to `geocodeAddress`. The resultant code will look like the following:

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

const argv = yargs
 .options({
   a: {
     demand: true,
     alias: 'address',
     describe: 'Address to fetch weather for',
     string: true
   }
 })
 .help()
 .alias('help', 'h')
 .argv;
```

Axios documentations

Now that we have a clean slate, we can get started by installing the new library. Before we run the `npm install` command, we'll see where we can find the documentation. We can get it by visiting: `https://www.npmjs.com/package/axios`. As shown in the following screenshot, we have the axios npm library page, where we can view all sorts of information about it, including the documentation:





Here we can see some things that look familiar. We have calls to then and catch, just like we do when we use promises outside of axios:

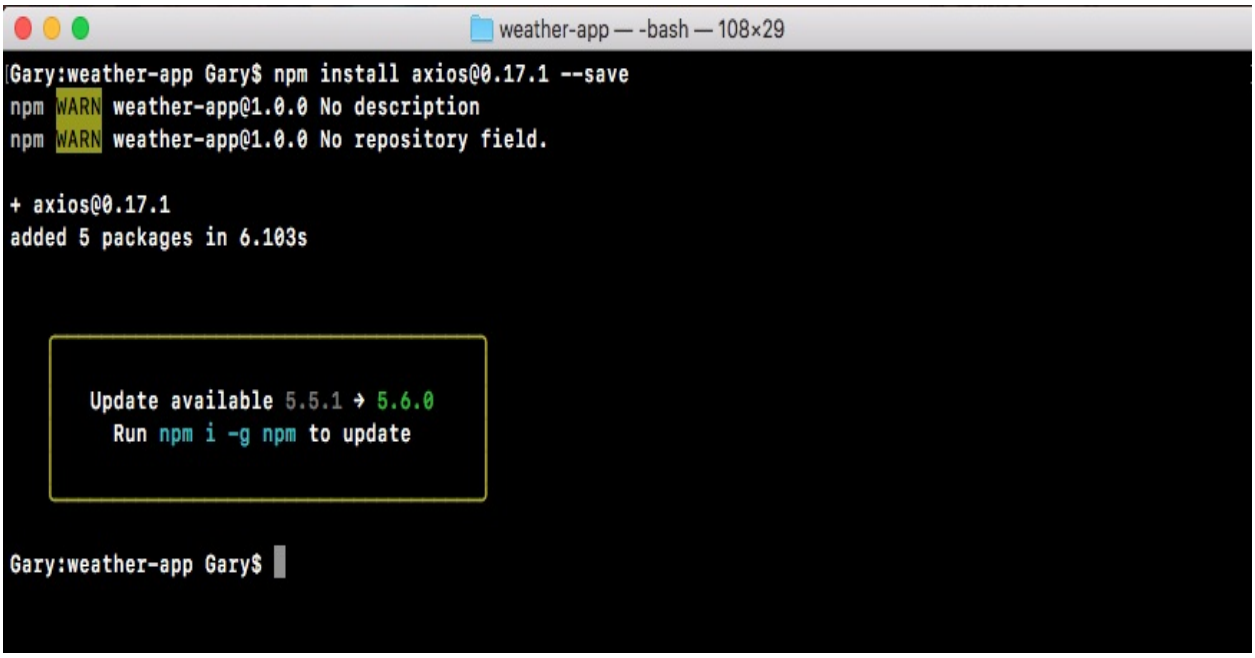
Performing a GET request

```
// Make a request for a user with a given ID
axios.get('/user?ID=12345')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });

// Optionally the request above could also be done as
axios.get('/user', {
    params: {
      ID: 12345
    }
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

Inside the stats column of this page, you can see that this is a super popular library. The most recent version is 0.13.1. This is the exact version we'll be using. Feel free to go to this page when you use axios in your projects. There are a lot of really good examples and documentation. For now though, we can install it.

Installing axios

To install axios, inside Terminal, we'll be running `npm install`; the library name is `axios`, and we'll specify the version `0.17.1` with the `save` flag updating the `package.json` file. Now I can run the `install` command, to install axios:



Making calls in the app-promise file

Inside our `app-promise` file, we can get started by loading in `axios` at the top. We'll make a constant called `axios`, setting it equal to `require('axios')`, as shown here:

```
const yargs = require('yargs');
const axios = require('axios');
```

Now that we have this in place, we can actually start making the calls in the code. This will involve us pulling out some of the functionality from the geocode and weather files. So we'll open up the `geocode.js` and `weather.js` files. Because we will be pulling some of the code from these files, things such as the URL and some of the error handling techniques. Although we'll talk about the differences as they come up.

The first thing we need to do is to encode the address and get the geocode URL. Now this stuff happens inside `geocode.js`. So we'll actually copy the `encodedAddress` variable line, where we create the encoded address, and paste it in the `app-promise` file, following the `argv` variable:

```
  .argv;

var encodedAddress = encodeURIComponent(argv.address);
```

Now we do need to tweak this a little bit. The `address` variable doesn't exist; but we have `argv.address`. So, we'll switch `address` with `argv.address`:

```
var encodeAddress = encodeURIComponent(argv.address);
```

Now we have the encoded address; the next thing we need to get before we can start using axios is the URL that we want to make the request to. We'll grab that from the `geocode.js` file as well. In `app-promise.js`, we will make a new variable called `geocodeURI`. Then, we'll take the URL present in `geocode.js`, from the opening tick to the closing tick, copy it, and paste it in `app-promise.js`, equal to `geocodeURI`:

```
var encodedAddress = encodeURIComponent(argv.address);
var geocodeUrl = `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAd
```

Now we use the encoded `address` variable inside the URL; this is fine because it does exist in our code. So at this point, we have our `geocodeUrl` variable and we can get started in making our very first axios request.

Making axios request

In our case, we'll be taking the address and getting the `latitude` and `longitude`. To make our request, we'll call a method available on axios, `axios.get`:

```
var geocodeUrl = `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAd

axios.get
```

The `get` is the method that lets us make our HTTP get request, which is exactly what we want to do in this case. Also, it's really simple to set up. When you're expecting JSON data, all you have to do is to pass in the URL that we have in the `geocodeUrl` variable. There's no need to provide any other options, such as an option letting it know it's `JSON`. `axios` knows how to automatically parse our JSON data. What get returns is actually a promise, which means we can use `.then` in order to run some code when the promise gets fulfilled or rejected, whether things go well or poorly:

```
axios.get(geocodeUrl).then()
```

Inside then, we'll provide one function. This will be the success case. The success case will get called with one argument, which the `axios` library recommends that you call `response`:

```
axios.get(geocodeUrl).then((response) => {

});
```

Technically, we could call anything you like. Now inside the function, we'll get access to all of the same information we got inside of the request library; things such as our headers, response, and request headers, as well as the body information; all sorts of useful info. What we really need though is the `response.data` property. We'll print that using `console.log`:

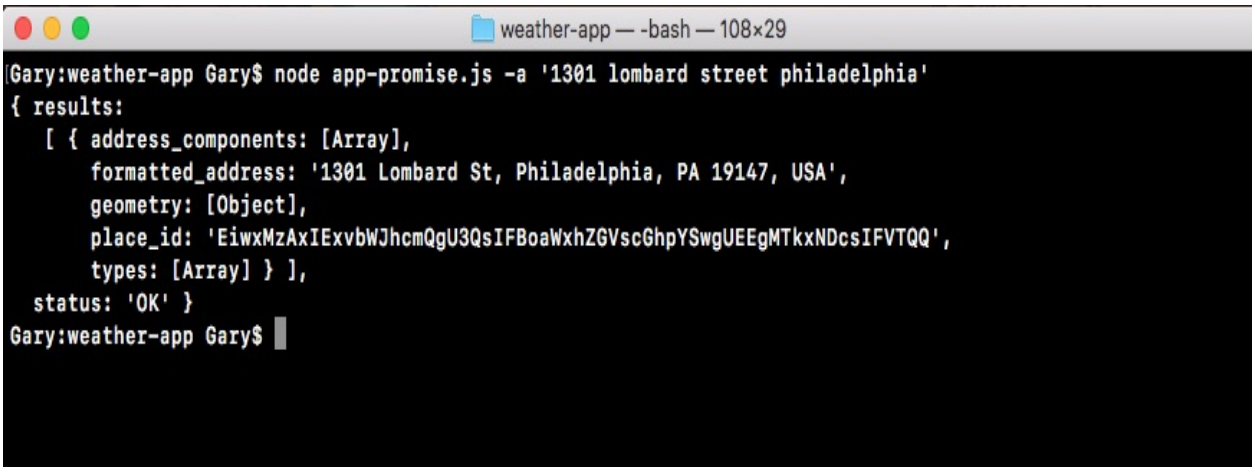
```
axios.get(geocodeUrl).then((response) => {
  console.log(response.data);
});
```

Now that we have this in place, we can run our `app-promise` file, passing in a valid address. Also, we can see what happens when we make that request.

Inside command line (Terminal), we'll use the `clear` command first to clear the Terminal output. Then we can run `node app-promise.js`, passing in an address. Let's use a valid address, for example, `1301 lombard street, philadelphia`:

```
node app-promise.js -a '1301 lombard street philadelphia
```

The request goes out. And what do we get back? We get back the results object exactly as we saw it when we used the other modules in the previous chapters:



The only difference in this case is that we're using promises built in, instead of having to wrap it in promises or using callbacks.

Error handling in axios request

Now aside from the success handler we used in the previous example, we can also add a call to catch, to let us catch all of the errors that might occur. We'll to get the error object as the one-and-only argument; then we can do something with that error object:

```
axios.get(geocodeUrl).then((response) => {
 console.log(response.data);
});catch((e) => {

});
```

Inside the function, we'll kick things off, using `console.log` to print the error argument:

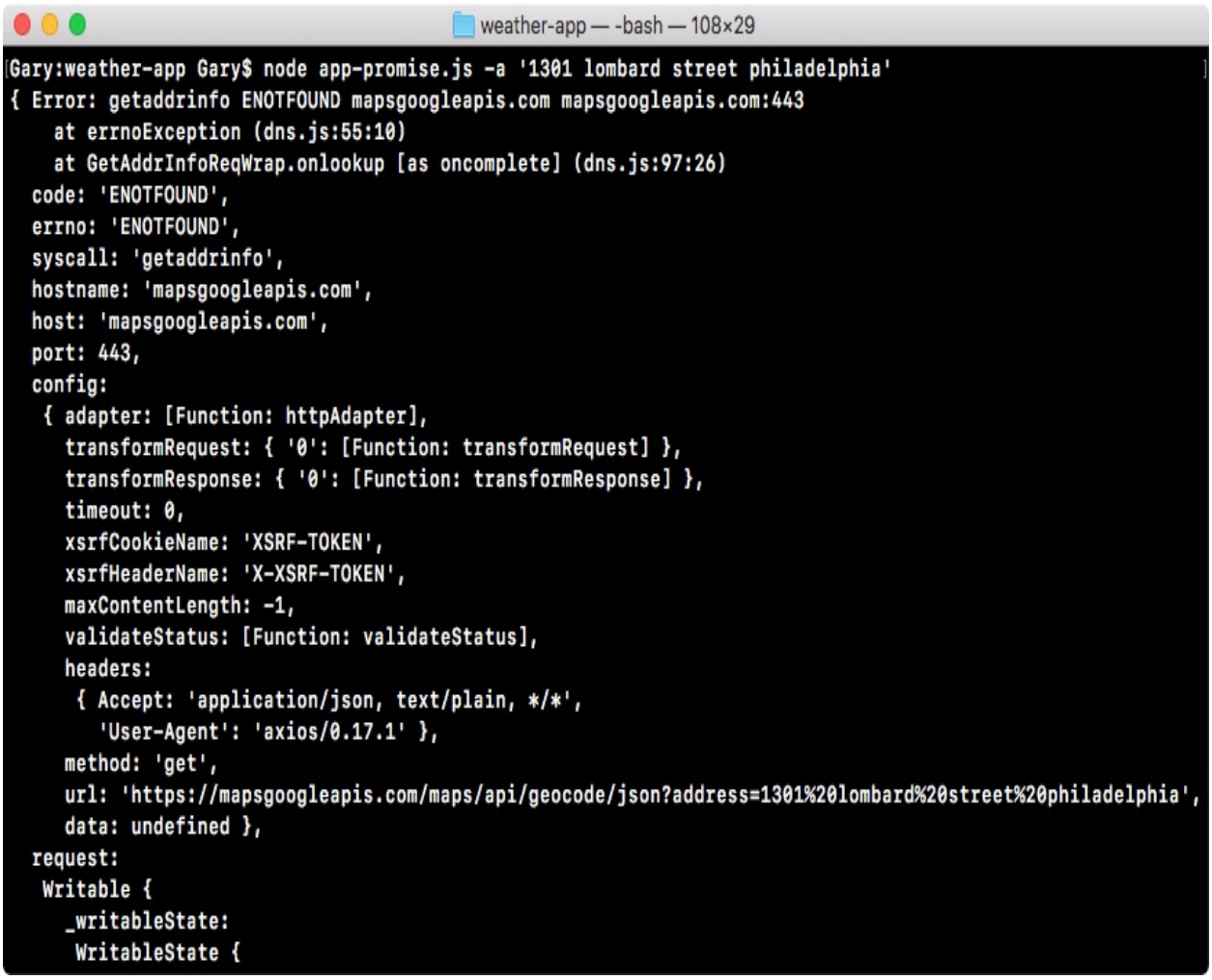
```
}).catch((e) => {
 console.log(e)
});
```

Now let's simulate an error by removing the dot in the URL:

```
var encodedAddress = encodeURIComponent(argv.address);
var geocodeUrl = `https://mapsgoogleapis.com/maps/api/geocode/json?address=${encodedAdd

axios.get(geocodeUrl).then((response) => {
   console.log(response.data);
}).catch((e) => {
   console.log(e)
});
```

We can see what happens when we rerun the program. Now I'm doing this to explore the `axios` library. I know exactly what will happen. This is not why I'm doing it. I'm doing it to show you how you should approach new libraries. When you get a new library, you want to play around with all the different ways it works. What exactly comes back in that error argument when we have a request that fails? This is important information to know; so when you write a real-world app, you can add the appropriate error handling code.

In this case, if we rerun the exact same command, we'll get an error:



As you can see, there really is nothing to print on the screen. We have a lot of very cryptic error codes and even the `errorMessage` property, which usually contains something good or does not. Then we have an error code followed by the URL. What we want instead is print a plain text English message.

To do this, we'll use an `if-else` statement, checking what the code property is. This is the error code and in this case `ENOTFOUND`; we know it means that it could not connect to the server. In `app-promise.js`, inside the error handler, we can add this by having `if` and checking the condition:

```
}).catch((e) => {
 if (e.code === 'ENOTFOUND') {

}
```

If that is the case, we'll print some sort of custom message to the screen using `console.log`:

```
}).catch((e) => {
  if (e.code === 'ENOTFOUND') {
   console.log('Unable to connect to API servers.');
  }
  console.log(e);
 });
```

Now we have an error handler that handles this specific case. So we can remove our call to `console.log`:

```
axios.get(geocodeUrl).then((response) => {
  console.log(response.data);
}).catch((e) => {
  if (e.code === 'ENOTFOUND') {
    console.log('Unable to connect to API servers.');
 }
});
```

Now if we save the file, and rerun things from Terminal, we should get a much nicer error message printing to the screen:

This is exactly what we get: `Unable to connect to API servers`. Now I'll add that dot back in, so things start working. We can worry about the response that comes back.

Error handling with ZERO_RESULT body status

As you remember, inside the geocode file, there were some things we needed to do. We've already handled the error related to server connection, but there is still another error pending, that is, if the `body.status` property equals `ZERO_RESULTS`. We want to print an error message in that case.

To do this, we'll inside `app-promise`, create our very own error. We'll throw an error inside the `axios.get` function. This error will cause all of the code after it, not to run. It will move right into the error handler.

Now we only want to throw an error if the status property is set to `ZERO_RESULTS`. We'll add an `if` statement at the very top of the `get` function to check `if (response.data.status)` equals `ZERO_RESULTS`:

```
axios.get(geocodeUrl).then((response) => {
  if (response.data.status === 'ZERO_RESULTS') {

  }
```

If that is the case, then things went bad and we do not want to move on to make the weather request. We want to run our catch code we have. To throw a new error that our promise can catch, we'll use a syntax called `throw new Error`. This creates and throws an error letting Node know that something went wrong. We can provide our own error message, something that's readable to a user: `Unable to find that address`:

```
axios.get(geocodeUrl).then((response) => {
  if (response.data.status === 'ZERO_RESULTS') {
    throw new Error('Unable to find that address.');
  }
```

This is a message that'll let that user know exactly what went wrong. Now when this error gets thrown, the same catch code will run. Currently, we only have one `if` condition that checks whether the code property is `ENOTFOUND`. So we'll add an `else` clause:

```
axios.get(geocodeUrl).then((response) => {
```

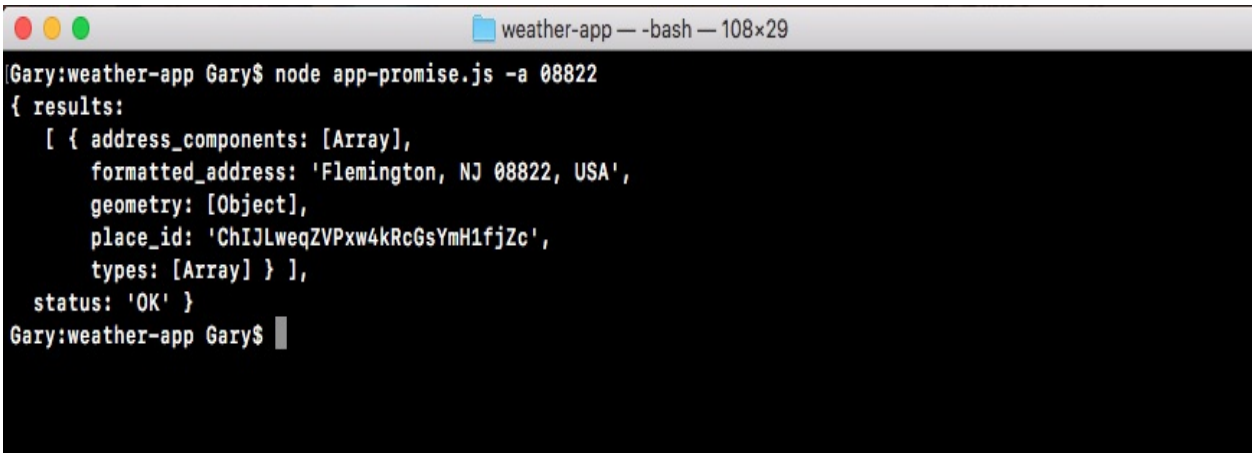
```
 if (response.data.status === 'ZERO_RESULTS') {
   throw new Error('Unable to find that address.');
 }

 console.log(response.data);
}).catch((e) => {
 if (e.code === 'ENOTFOUND') {
   console.log('Unable to connect to API servers.');
 } else {

 }
});
```

Inside the `else` block, we can print the error message, which is the string we typed in the throw `new Error` syntax using the `e.` message property, as shown here:

```
axios.get(geocodeUrl).then((response) => {
 if (response.data.status === 'ZERO_RESULTS') {
   throw new Error('Unable to find that address.');
 }

 console.log(response.data);
}).catch((e) => {
  if (e.code === 'ENOTFOUND') {
   console.log('Unable to connect to API servers.');
 } else {
   console.log(e.message);
 }
});
```

If the error code is not `ENOTFOUND`, we'll simply print the message to the screen. This will happen if we get zero results. So let's simulate that to make sure the code works. Inside Terminal, we'll rerun the previous command passing in a zip code. At first, we'll use a valid zip code, `08822` and we should get our data back. Then we'll use an invalid one: `00000`.

When we run the request with a valid address, we get this:



When we run the request with the invalid address, we get the error:

```
Gary:weather-app Gary$ node app-promise.js -a 000000
Unable to find that address.
```

By calling `throw new Error`, we're immediately stopping the execution of this function. So `console.log` with `e.message` never prints, which is exactly what we want. Now that we have our error handler in place, we can start generating that weather URL.

Generating the weather URL

In order to generate the weather URL, we'll copy the URL from the `weather` file, taking it with the ticks in place, and moving it into the `app-promise` file. We'll make a new variable called `weatherUrl`, setting it equal to the copied URL:

```
url: `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${lat},${lng}`,
```

Now `weatherUrl` does need a few pieces of information. We need the `latitude` and `longitude`. We have two variables `lat` and `lng`, so let's create them, getting the appropriate value from that response object, `var lat` and `var lng`:

```
var lat;
var lng;
url: `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${lat},${lng}`,
```

Now in order to pull them off, we have to go through that process of digging into the object. We've done it before. We'll be looking in the response object at the data property, which is similar to the body in the request library. Then we'll go into `results`, grabbing the first item and accessing the `geometry` property, then we'll access `location.lat`:

```
var lat = response.data.results[0].geometry.location.lat;
```

Now similarly, we can add things for the `longitude` variable:

```
var lat = response.data.results[0].geometry.location.lat;
var lng = response.data.results[0].geometry.location.lng;
```

Now before we make that weather request, we want to print the formatted address because that's something the previous app did as well. In our `console.log(response.data)` statement, and instead of printing `response.data`, we'll dive into the data object getting the formatted address. This is also on the results array's first item. We'll be accessing the `formatted_address` property:

```
var lat = response.data.results[0].geometry.location.lat;
var lng = response.data.results[0].geometry.location.lng;
var weatherUrl = `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${l
console.log(response.data.results[0].formatted_address);
```

Now that we have our formatted address printing to the screen, we can make our

second call by returning a new promise. This is going to let us chain these calls together.

Chaining the promise calls

To get started, we'll return a call to `axios.get`, passing in the URL. We just defined that, it is `weatherUrl`:

```
var lat = response.data.results[0].geometry.location.lat;
var lng = response.data.results[0].geometry.location.lng;
var weatherUrl = `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${
console.log(response.data.results[0].formatted_address);
return axios.get(weatherUrl);
```

Now that we have this call returning, we can attach another `then` call right between our previous `then` call and catch call, by calling then, passing in one function, just like this:

```
 return axios.get(weatherUrl);
}).then(() => {

}).catch((e) => {
 if (e.code === 'ENOTFOUND') {
```

This function will get called when the weather data comes back. We'll get that same response argument, because we're using the same method, `axios.get`:

```
}).then((response) => {
```

Inside the `then` call, we don't have to worry about throwing any errors, since we never needed to access a body property in order to check if something went wrong. With the weather request if this callback runs, then things went right. We can print the weather information. In order to get that done, we'll make two variables:

- `temperature`
- `apparentTemperature`

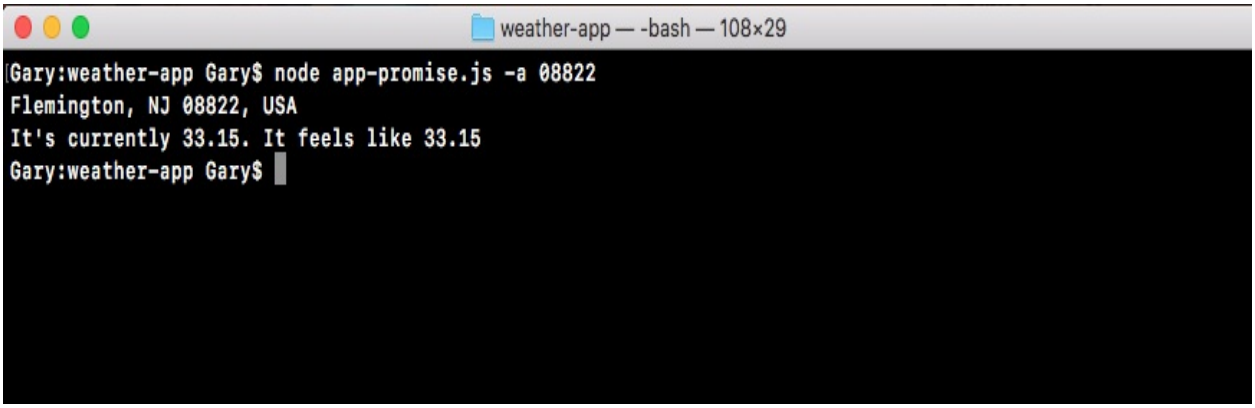
The `temperature` variable will get set equal to `response.data`. Then we'll access that `currently` property. Then we'll access temperature. We'll pull out the second variable, the actual temperature or `apparentTemperature`, which is the property name, `var apparentTemperature`. We'll be setting this equal to `response.data.currently.apparentTemperature`:

```
}).then((response) => {
 var temperature = response.data.currently.temperature;
 var apparentTemperature = response.data.currently.apparentTemperature;
```

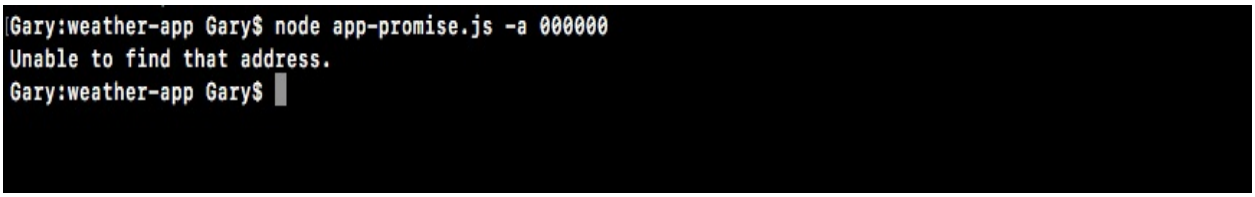
Now that we have our two things pulled out into variables, we can add those things inside of a call, `console.log`. We chose to define two variables, so that we don't have to add the two really long property statements to `console.log`. We can simply reference the variables. We'll add `console.log` and we'll use template strings in the `console.log` statement, so that we can inject the previous mentioned two values inside of quotes: `It's currently`, followed by `temperature`. Then we can add a period, `It feels like`, followed by `apparentTemperature`:

```
}).then((response) => {
 var temperature = response.data.currently.temperature;
 var apparentTemperature = response.data.currently.apparentTemperature;
 console.log(`It's currently ${temperature}. It feels like ${apparentTemperature}.`);
```

Now that we have our string printing to the screen, we can test that our app works as expected. We'll save the file and inside Terminal, we'll rerun the command from two commands ago where we had a valid zip code:



When we run this, we get the weather info for `Flemington`, New Jersey. It's currently `84` degrees, but it feels like `90`. If we run something that has a bad address, we do get the error message:



So everything looks great! Using the `axios` library, we're able to chain promises

like the app-promise without needing to do anything too crazy. The `axios get` method returns a promise, so we can access it directly using then.

In the code, we use then once to do something with that geolocation data. We print the address to the screen. Then we return another promise, where we make the request for the weather. Inside of our second `then` call, we print the weather to the screen. We also added a catch call, which will handle any errors. If anything goes wrong with either of our promises, or if we throw an error, catch will get fired printing the messages to the screen.

This is all it takes to use axios and set up promises for your HTTP requests. Now one reason people love promises over traditional callbacks is that instead of nesting we can simply chain. So our code doesn't get indented to crazy levels. As we saw in `app.js` in the previous chapter, we went a few indentation levels deep just to add two calls together. If we needed to add a third it would have gotten even worse. With promises, we can keep everything at the same level, keeping our code a lot easier to maintain.

Summary

In this chapter, we've gone through a quick example of how promises work, by going over just the very fundamentals. Async is a critical part to Node.js. We went through the very basics of callbacks and promises. We looked a few examples, creating a pretty cool weather app.

This brings us to the end of our asynchronous Node.js programming, but this does not mean that you have to stop building out the weather app. There are a couple ideas as to what you could do to continue on with this project. First up, you can load in more information. The response we get back from the weather API contains a ton of stuff besides just the current temperature, which is what we used. It'd great if you can incorporate some of that stuff in there, whether it's high/low temperatures, or chances of precipitation.

Next up, it'd be really cool to have a default location ability. There would be a command that lets me set a default location, and then I could run the weather app with no location argument to use that default. We could always specify a location argument to search for weather somewhere else. This would be an awesome feature, and it would work kind of similar to the Notes app, where we save data to the filesystem.

In the next chapter, we'll start creating web servers, which will be async. We'll make APIs, which will be async. Also, we'll create real-time Socket.IO apps, which will be async. We'll move on to creating Node apps that we deploy to servers, making those servers accessible to anybody with a web connection.

Web Servers in Node

We'll cover a ton of exciting stuff in this chapter. We'll learn how to make a web server and how to integrate version control into Node applications. Now to get all this done, we will look at a framework called Express. It's one of the most popular npm libraries, and for good reason. It makes it really easy to do stuff such as creating a web server or an HTTP API. It's kind of similar to the Dark Sky API we used in the last chapter.

Now most courses start with Express, and that can be confusing because it blurs the line between what is Node and what is Express. We'll kick things off by adding Express to a brand new Node app.

Specifically, we'll cover the following topics:

- Introducing Express
- Static server
- Rendering templates
- Advanced templates
- Middleware

Introducing Express

In this section, you'll make your very first Node.js web server, which means you'll have a whole new way for users to access your app. Instead of having them run it from the Terminal passing in arguments, you'll be able to give them a URL they can visit to view your web app or a URL they can make an HTTP request to to fetch some data.

This will be similar to what we did when we used the geocode API in the previous chapters. Instead of using an API though, we'll be able to create our own. We'll also be able to set up a static website for something like a portfolio site. Both are really valid use cases. Now all of this will be done using a library called **Express**, which is the most popular npm library. It's actually one of the reasons that Node got so popular because it was so easy to make REST APIs and static web servers.

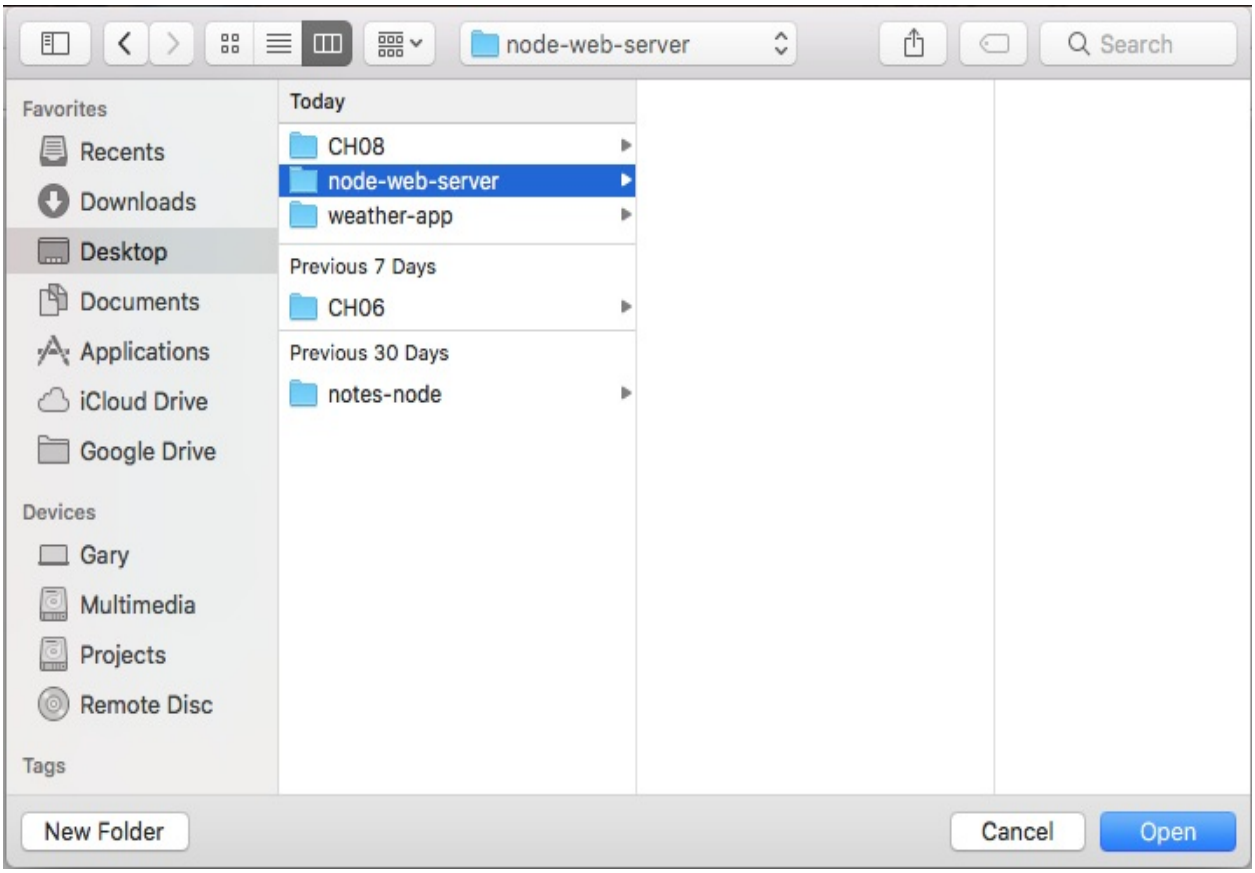
Configuring Express

Express is a no-nonsense library. Now there are a lot of different ways to configure it. So it can get pretty complex. That's why we'll be using it throughout the next couple of chapters. To get started, let's make a directory where we can store all of the code for this app. This app will be our web server.

On the desktop let's us make a directory called `node-web-server`, by running the `mkdir node-web-server` command in the Terminal:

Once this directory is created, we'll navigate into it using `cd`:

And we'll also open it up inside Atom. In Atom, we'll open it up from the desktop:

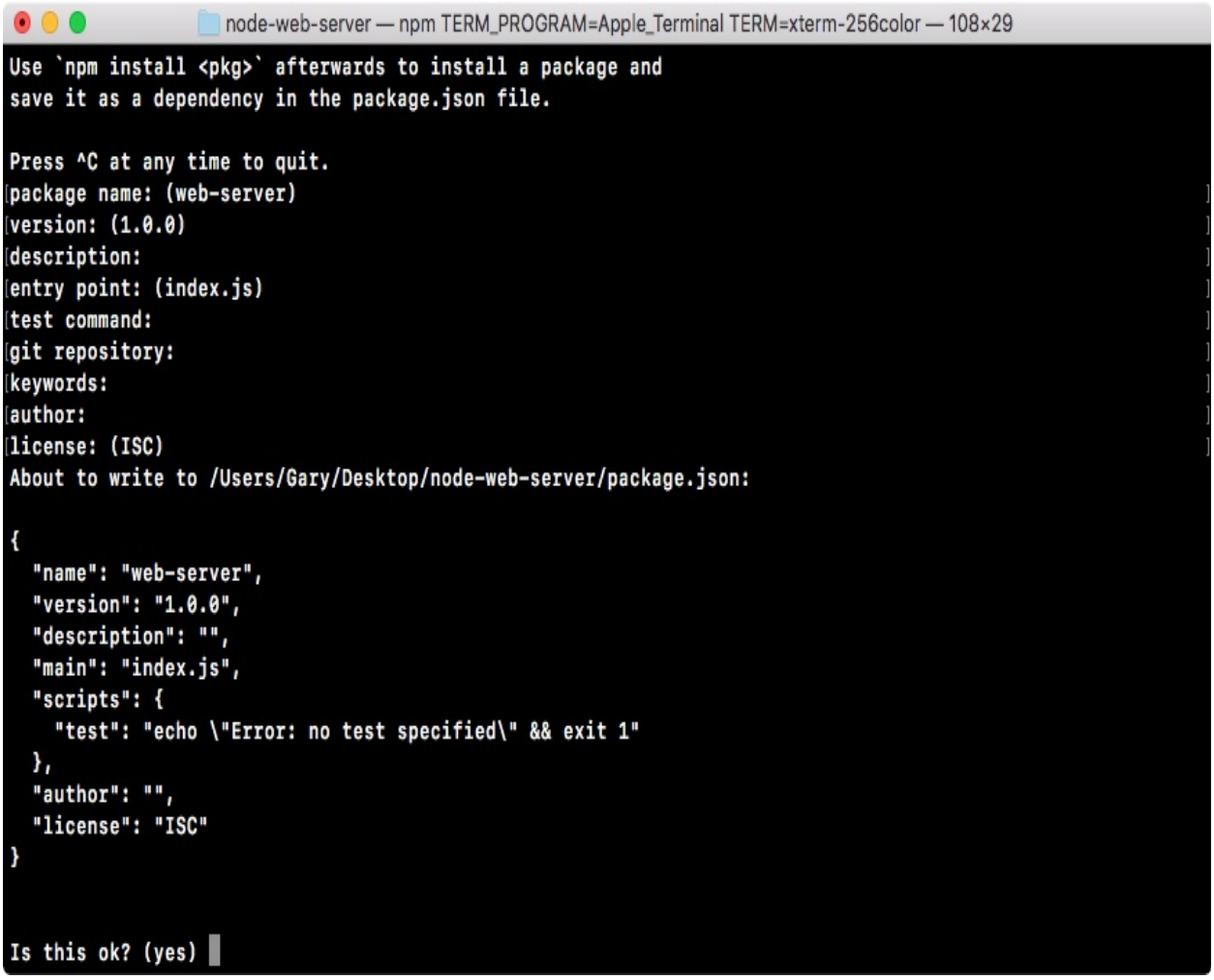


Now before going further, we'll run the `npm init` command so we can generate the `package.json` file. As shown in the following code, we'll run `npm init`:

```
node-web-server — npm TERM_PROGRAM=Apple_Terminal TERM=xterm-256color — 108×29
Gary:node-web-server Gary$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (web-server)
```

Then, we'll use the default value just by pressing *enter* through all of the options shown in the following screenshot. There's no need to customize any of these as

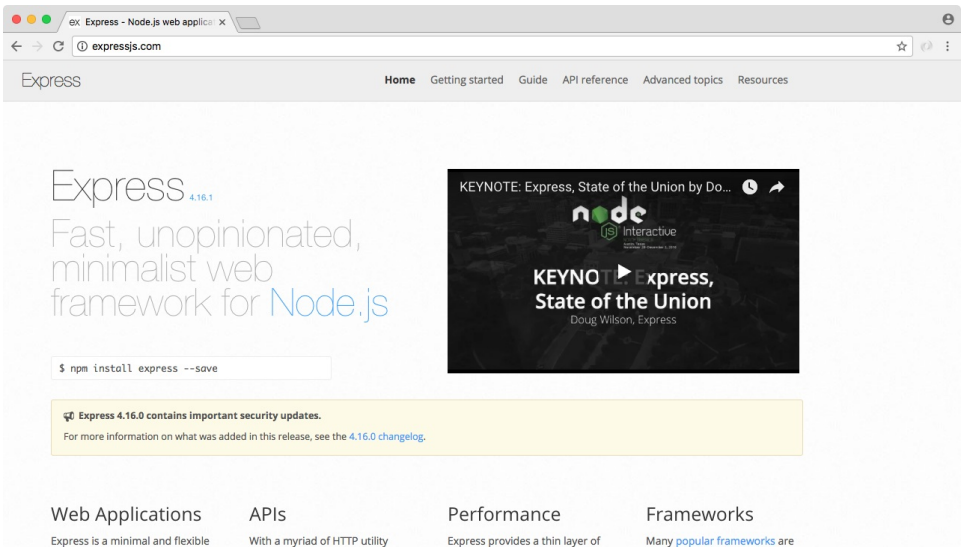
of now:



Then we'll type `yes` in the last statement `Is this ok? (yes)` and the `package.json` file goes in place:

Express docs website

As mentioned earlier, Express is a really big library. There's an entire website dedicated to the Express docs. Instead of a simple `README.md` file, you can go to `www.expressjs.com` to view everything the website have to offer:

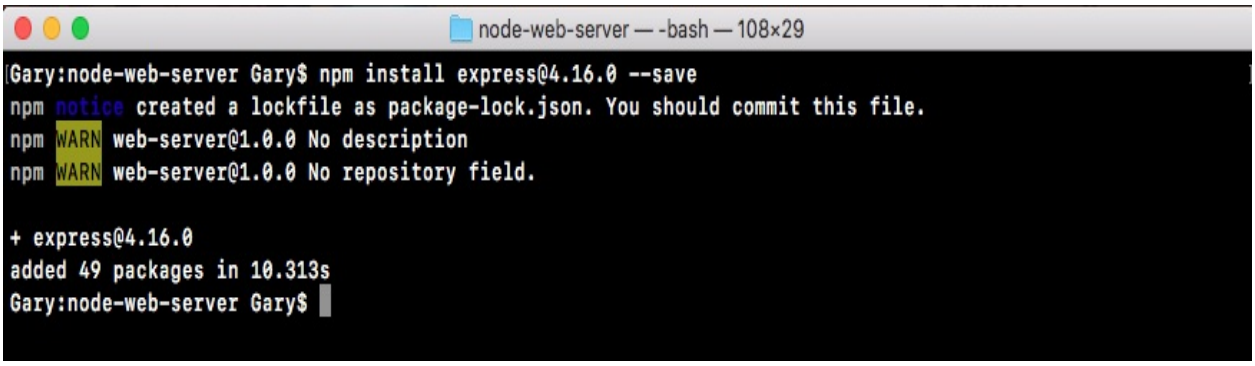


We'll find Getting started, help articles, and many more. The website has the Guide option to help you do things such as Routing, Debugging, Error handling, and an API reference, so we can look into exactly what methods we have access to and what they do. It's a very handy website.

Installing Express

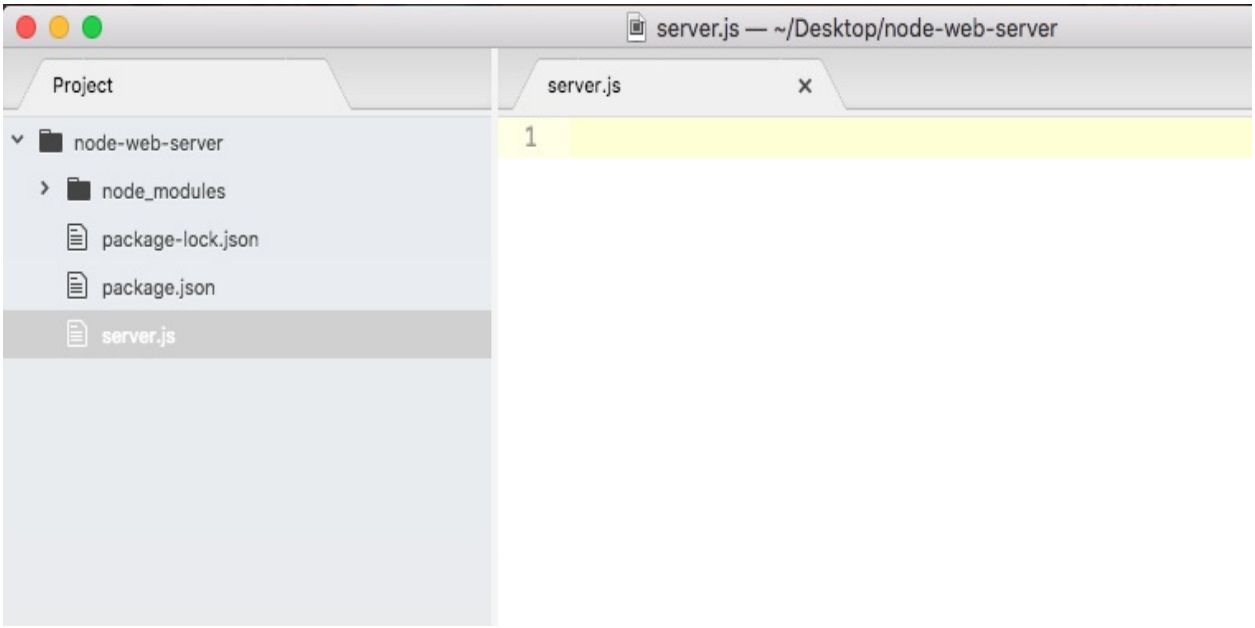
Now that we have our `node-web-server` directory, we'll install Express so we can get started making our web server. In the Terminal we'll run the `clear` command first to clear the output. Then we'll run the `npm install` command. The module name is `express` and we'll be using the latest version, `@4.16.0`. We'll also provide the `save` flag to update the dependencies inside of our `package.json` file as shown here:

```
npm install express@4.16.0 --save
```



Once again we'll use the `clear` command to clear the Terminal output.

Now that we have `Express` installed, we can actually create our web server inside Atom. In order to run the server, we will need a file. I'll call this file `server.js`. It will sit right in the root of our application:



This is where we'll configure the various routes, things like the root of the website, pages like `/about`, and so on. It's also where we'll start the server, binding it to a port on our machine. Now we'll be deploying to a real server. Later we'll talk about how that works. For now, most of our server examples will happen on our localhost.

Inside `server.js`, the first thing we'll do is load in Express by making a constant called `express` and setting it equal to `require('express')`:

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

Next up, what we'll do is make a new Express app. To do this we'll make a variable called app and we'll set it equal to the return result from calling `express` as a function:

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

var app = express();
```

Now there are no arguments we need to pass into `express`. We will do a ton of configuration, but that will happen in a different way.

Creating an app

In order to create an app, all we have to do is call the method. Next to the variable `app` we can start setting up all of our HTTP route handlers. For example, if someone visits the root of the website we're going to want to send something back. Maybe it's some JSON data, maybe it's an HTML page.

We can register a handler using `app.get` function. This will let us set up a handler for an HTTP get request. There are two arguments we have to pass into `app.get`:

- The first argument is going to be a URL
- The second argument is going to be the function to run; the function that tells Express what to send back to the person who made at the request

In our case we're looking for the root of the app. So we can just use forward slash (`/`) for the first argument. In the second argument, we'll use a simple arrow function (`=>`) as shown here:

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

var app = express();

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

};
```

Now the arrow function (`=>`) will get called with two arguments. These are really important to how Express works:

- The first argument is request (`req`) stores a ton of information about the request coming in. Things like the headers that were used, any body information, or the method that was made with a request to the path. All of that is stored in request.
- The second argument, respond (`res`), has a bunch of methods available so we can respond to the HTTP request in whatever way we like. We can customize what data we send back and we could set our HTTP status codes.

We'll explore both of these in detail. For now though, we'll use one method, `res.send`. This will let us respond to the request, sending some data back. In

`app.get` function, let's call `res.send`, passing in a string. In the parenthesis we'll add `Hello Express!`:

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

This is the response for the HTTP request. So when someone views the website they will see this string. If they make a request from an application, they will get back `Hello Express!` as the body data.

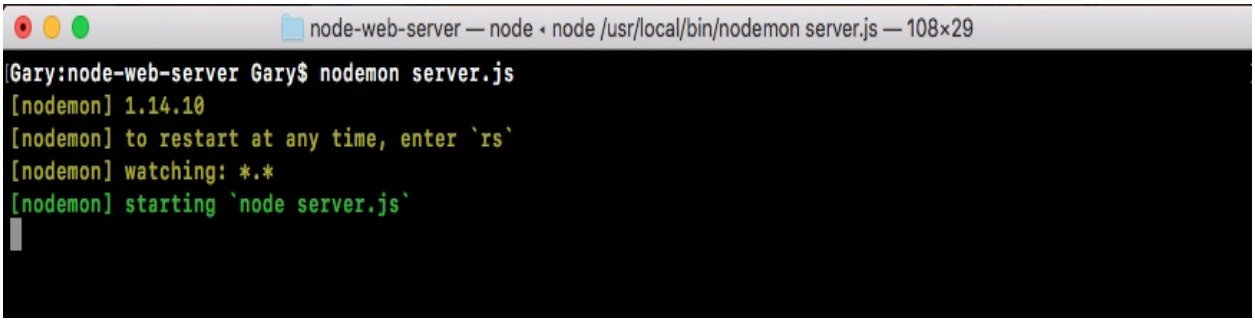
Now at this point we're not quite done. We have one of our routes set up, but the app is never going to actually start listening. What we need to do is call `app.listen`. The `app.listen` function will bind the application to a port on our machine. In this case for our local host app, we will use port `3000`, a really common port for developing locally. Later in the chapter, we'll talk about how to customize this depending on whatever server you use to deploy your app to production. For now though, a number like `3000` works:

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

app.listen(3000);
```

With this in place we are now done. We have our very first Express server. We can actually run things from the Terminal, and view it in the browser. Inside the Terminal, we'll use `nodemon server.js` to start up our app:

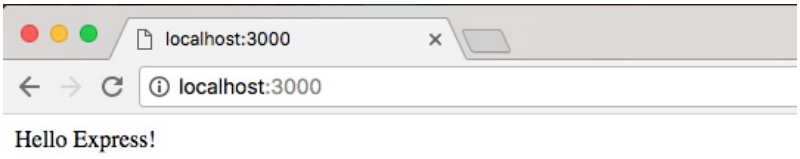
```
nodemon server.js
```

This will start up the app and you'll see that the app never really finishes as shown here:



Right now it's hanging. It's waiting for requests to start coming in. The apps that use `app.listen`, they will never stop. You'll have to shut them down manually with *control* + *C*, like we've done before. It might crash if you have an error in your code. But it'll never stop normally, since we have that binding set up here. It will listen to requests until you tell it to stop.

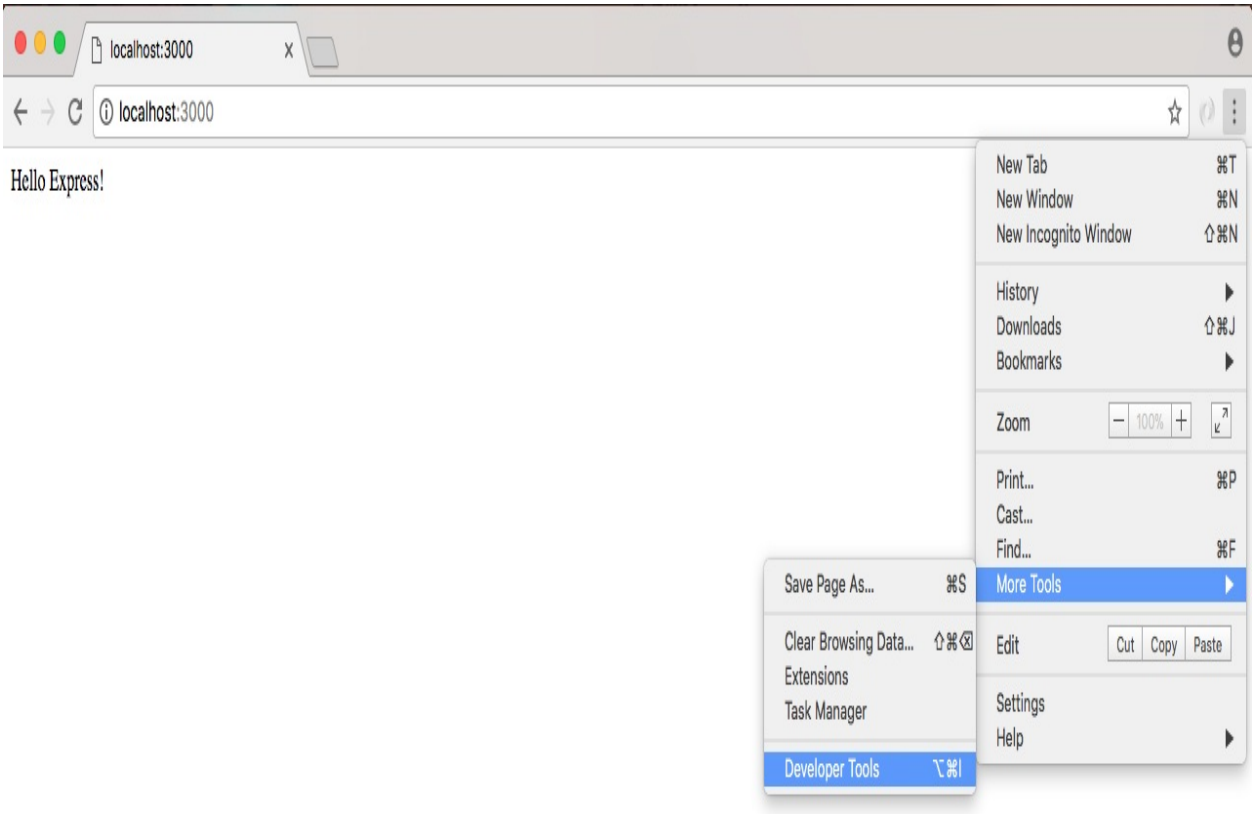
Now that the server is up, we can move into the browser and open up a new tab visiting the website, `localhost:` followed by the port `3000`:



This will load up the root of the website, and we specify the handler for that route. Hello Express! shows up, which is exactly what we expected. Now there's no thrills. There's no formatting. We're just sending a string from the server back to the client that made the request.

Exploring the developer tools in the browser for the app request

What we'd like to do next is open up the developer tools, so we can explore exactly what happened when that request was made. Inside Chrome you can get to the Developer Tools using Settings| More Tools| Developer Tools:

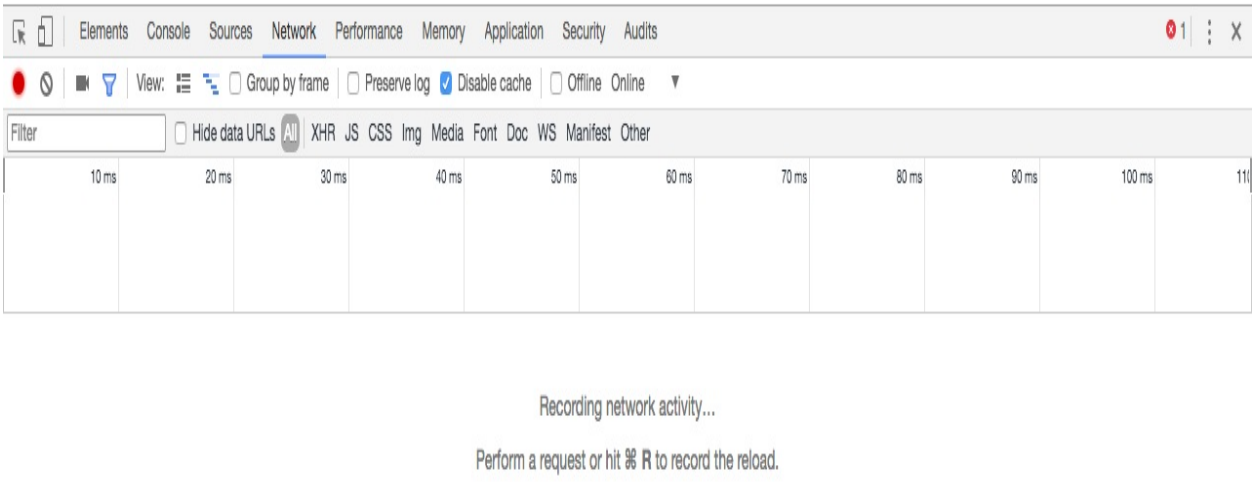


Or you can use the keyboard shortcut shown along with Developer Tools for the operating system.

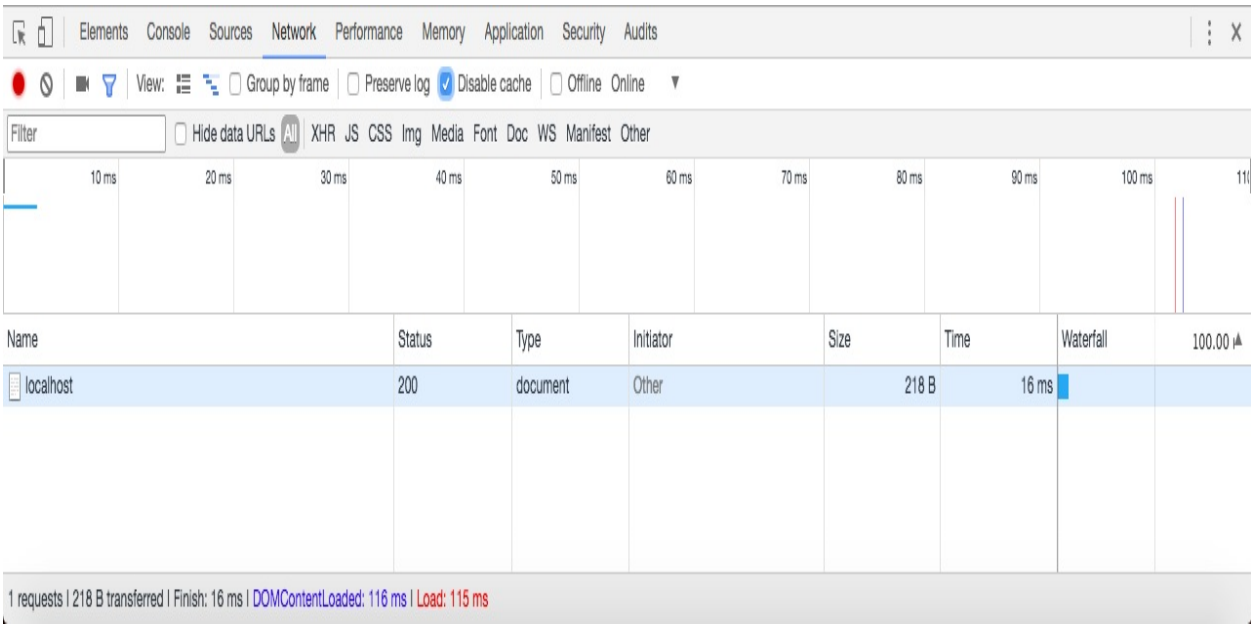
I would highly recommend memorizing that keyboard shortcut because you'll use the `Developer Tools` *a ton in your career with*

Node.

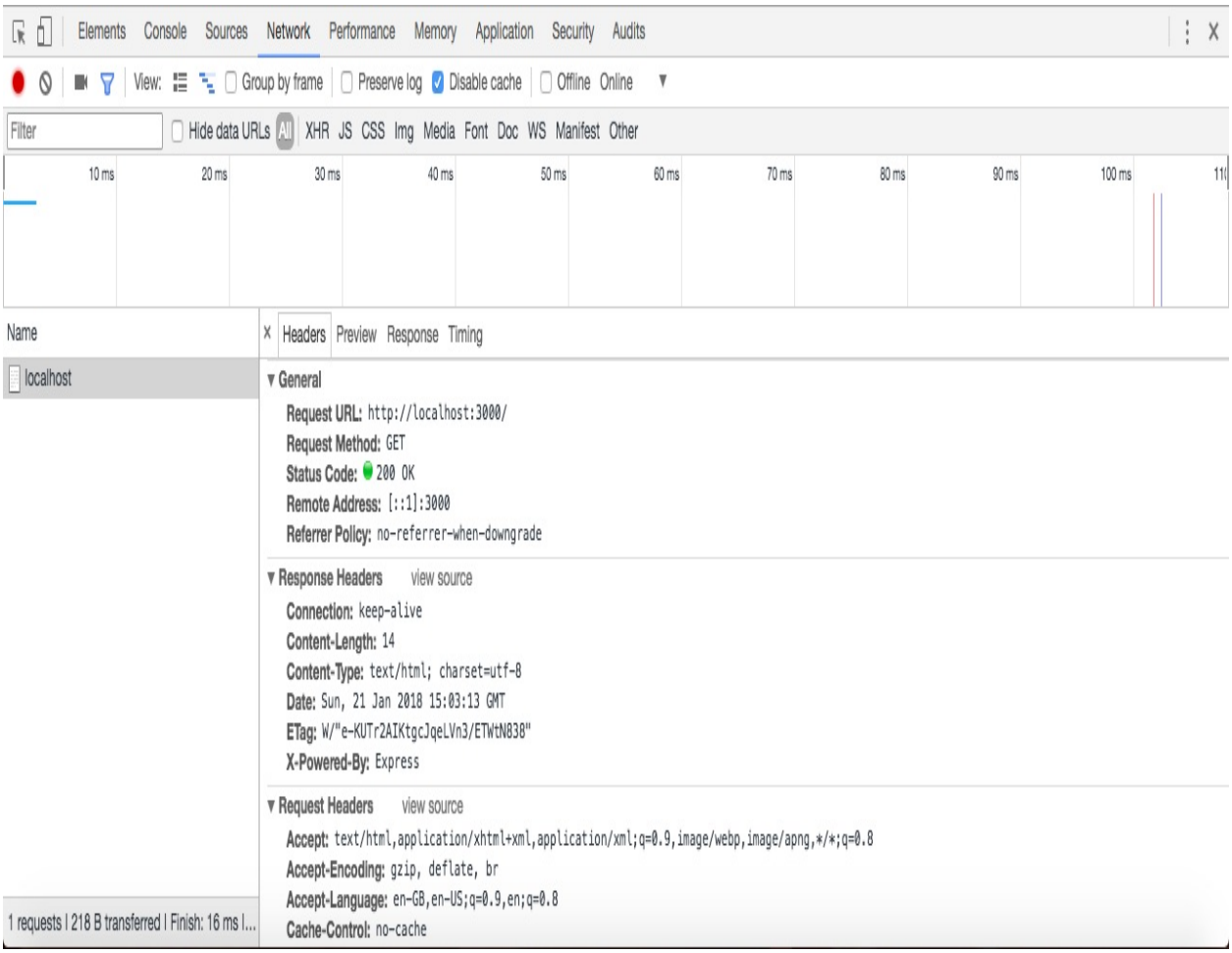
We'll now open up the Developer Tools, which should look similar to the ones we used when we ran the Node Inspector debugger. They're a little different, but the idea is the same:



We have a bunch of tabs up top, and then we have our tab specific information down following on the page. In our case, we want to go to the Network tab, and currently we have nothing. So we'll refresh the page with the tab open, and what we see right here is our localhost request:



This is the request that's responsible for showing Hello Express! to the screen. We can actually click the request to view its details:



This page can be a little overwhelming at first. There is a a lot of information. Up on top we have some general info, such as the URL that was requested, the method that the client wanted; in this case, we made a GET request, and the status code that came back. The default status code being 200, meaning that everything went great. We'd like to point the attention to is one response header.

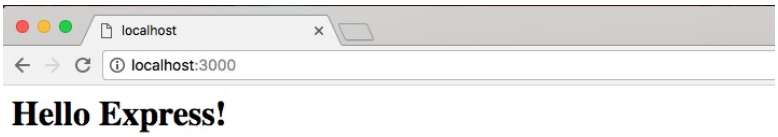
Under Response Headers we have a header called Content-Type. This header tells the client what type of data came back. Now this could be something like an HTML website, some text, or some JSON data and the client could be a web browser, an iPhone, an Android device, or any other computer with network capabilities. In our case, we're telling the browser that what came back is some HTML, so why don't you render it as such. We use the text/html Content-Type. And this automatically got set by Express, which is one of the reasons it's so popular. It handles a lot of that mundane stuff for us.

Passing HTML to res.send

Now that we have a very basic example, we want to step things up a notch. Inside Atom, we can actually provide some HTML right inside of send by wrapping our `Hello Express!` message in an `h1` tag. Later in this section, we'll be setting up a static website that has HTML files that get served up. We'll also look at templating to create dynamic web pages. But for now, we can actually just pass in some HTML to `res.send`:

```
app.get('/', (req, res) => {
  res.send('<h1>Hello Express!</h1>');
});

app.listen(3000);
```

We'll save the server file, which should restart things in the browser. When we give the browser a refresh, we get Hello Express! printing to the screen:



This time though, we have it in an `h1` tag, which means it's formatted by the default browser styles. In this case it looks nice and big. With this in place we can now open up the request inside the Network tab, and what we get is the exact same thing we had before. We're still telling the browser that it's HTML. Only one difference this time: we actually have an HTML tag, so it gets rendered using the browser's default styles.

Sending JSON data back

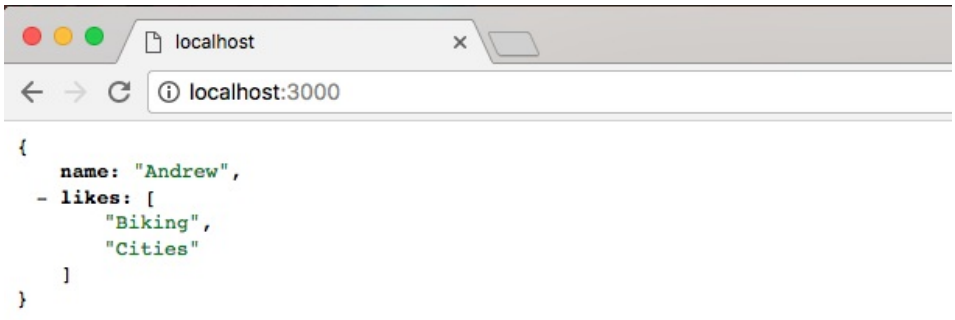
The next thing we'd look into is how we can send some JSON data back. Sending JSON is really easy with Express. To illustrate how we can do it we'll comment out our current call to `res.send` and add a new one. We'll call `res.send` passing in an object:

```
app.get('/', (req, res) => {
  // res.send('<h1>Hello Express!</h1>');
  res.send({

  })
});
```

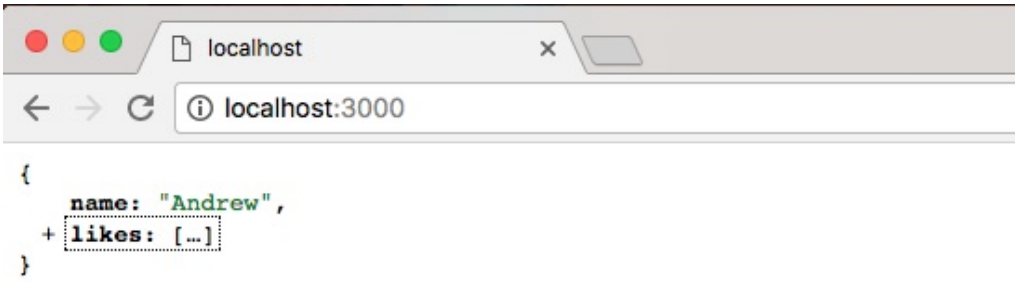
On this object we can provide whatever we like. We can create a `name` property, setting it equal to the string version of any name, say `Andrew`. We can make a property called `likes`, setting it equal to an array, and we can specify some things we may like. Let's add `Biking` as one of them, and then add `Cities` as another:

```
  res.send({
    name: 'Andrew',
    likes: [
      'Biking',
      'Cities'
    ]
  });
```

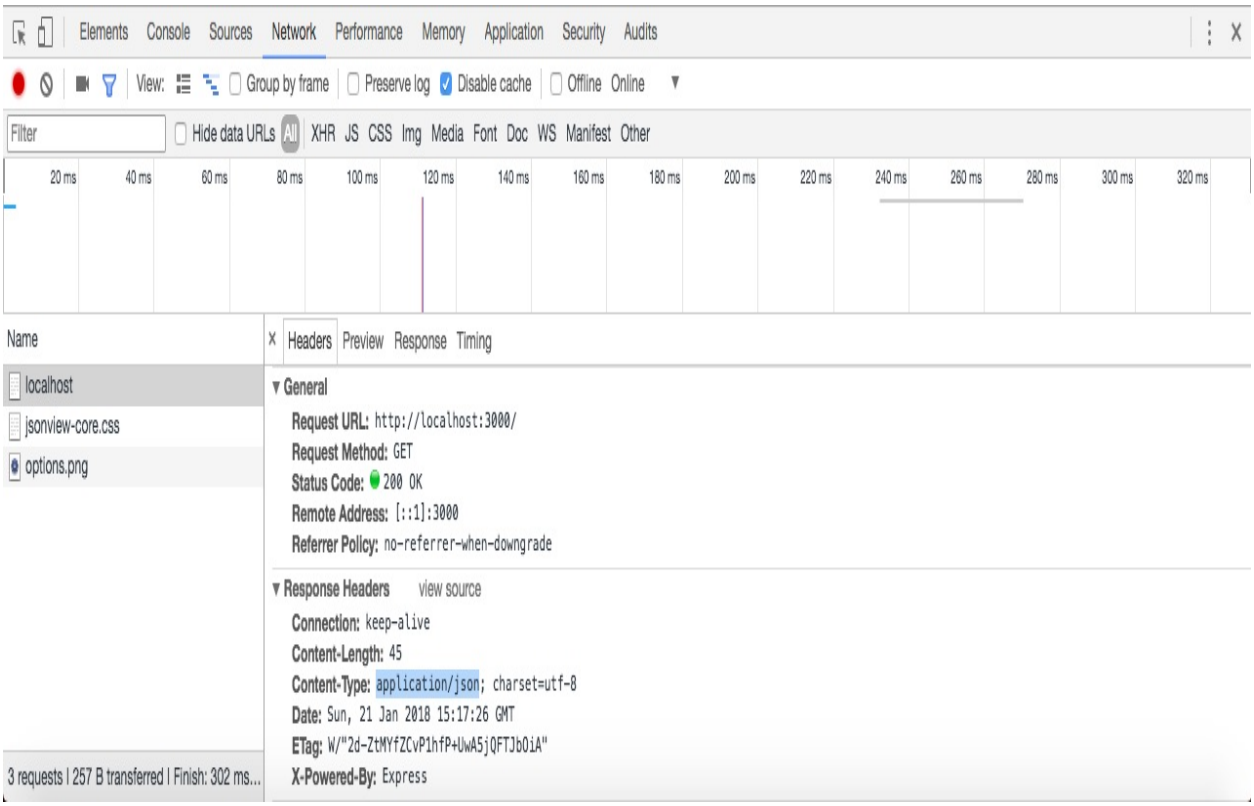
When we call `res.send` passing in an object, Express notices that. Express takes it, converts it into JSON, and sends it back to the browser. When we save `server.js` and nodemon refreshes, we can refresh the browser, and what we get is my data formatted using JSON view:



This means we can collapse the properties and quickly navigate the JSON data.



Now the only reason JSON view picked up on this is because that Content-Type header that we explored in our last request it actually changed. If I open up `localhost`, a lot of things look the same. But now Content-Type has an application/json Content-Type:



This Content-Type tells the requester whether it's an Android phone, an iOS device, or the browser that JSON data is coming back, and it should parse it as such. That's exactly what the browser does in this case.

Express also makes it really easy to set up other routes aside from the root route. We can explore that inside Atom by calling `app.get` a second time. We'll call `app.get`. We'll create a second route. We'll call this one `about`:

```
app.get('/about')

app.listen(3000);
```

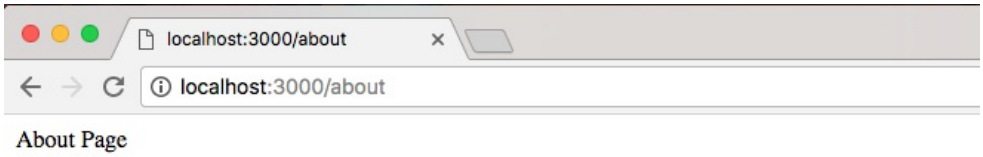
Notice that we just used `/about` as the route. It's important to keep that forward slash in place, but after that you can type whatever you like. In this case we'll have a `/about` page that someone can visit. Then I'll provide the handler. The handler will take the `req` and the `res` object:

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

});

app.listen(3000);
```

This will let us figure out what kind of request came in, and it will let us respond to that request. For now just to illustrate we can create more pages, we'll keep the response simple, `res.send`. Inside the string we're going to print `About Page`:

```
app.get('/about', (req, res) => {
  res.send('About Page');
});
```

Now when we save the `server.js` file, the server is going to restart. In the browser we can visit `localhost:3000/about`. At `/about` we should now see our new data, and that's exactly what we get back, About Page shows up as shown here:



Using `app.get` we're able to specify as many routes as we like. For now we just have an `about` route and a `/` route, which is also referred to as the root route. The root route returns some data, which happens to be JSON, and the about route returns a little bit of HTML. Now that we have this in place and we have a very basic understanding about how we can set up routes in Express, we'd like you to create a new route `/bad`. This is going to simulate what happens when a request fails.

Error handling in the JSON request

To show the error handling request with JSON, we're going to call `app.get`. This `app.get` is going to let us register another handler for a get HTTP request. In our case the route we're looking for inside of quotes is going to be `/bad`. When someone makes a request for this page, what we want to do is going to be specified in the callback. The callback will take our two arguments, `req` and `res`. We'll use an arrow function (`=>`), which I've used for all of the handlers so far:

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

  });

app.listen(3000);
```

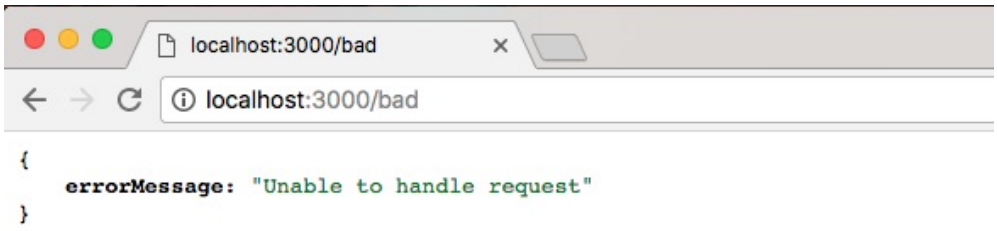
Inside the arrow function (`=>`), we'll send back some JSON by calling `res.send`. But instead of passing in a string, or some string HTML, we'll pass in an object:

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

  });
});
```

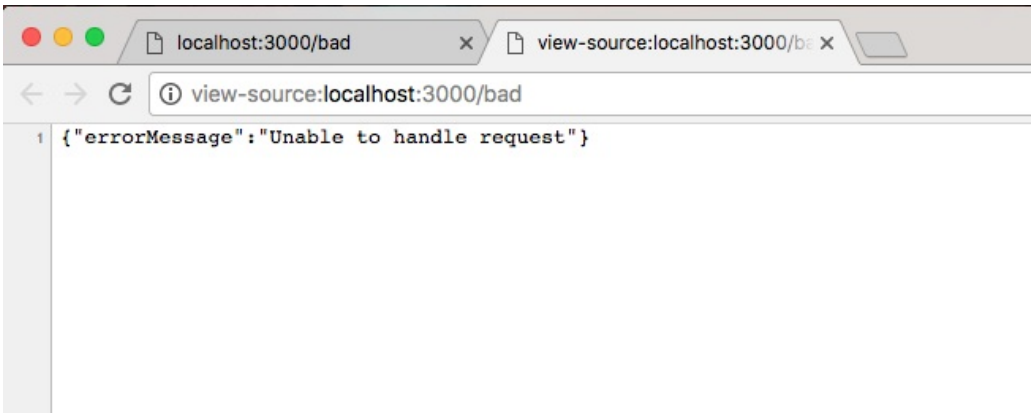
Now that we have our object in place we can specify the properties we want to send back. In this case we'll set one `errorMessage`. We'll set my error message property equal to a string, `Unable to handle request`:

```
app.get('/bad', (req, res) => {
  res.send({
    errorMessage: 'Unable to handle request'
  });
});
```

Next up we'll save the file, restarting it in nodemon, and visit it in the browser. Make sure our error message showed up correctly. In the browser, we'll visit `/bad`, hit *enter*, and this is what we get:



We get our JSON showing up using JSON view. We have our error message, and we have the message showing up: Unable to handle request. Now if you are using JSON view and you want to view the raw JSON data, you can actually click on View source, and it will show it in a new tab. Here, we're looking at the raw JSON data, where everything is wrapped in those double quotes:



I'll stick to the JSON view data because it's a lot easier to navigate and view. We now have a very basic Express application up and running. It listens on port `3000` and it currently has handlers for 3 URLs: when we get the root of the page, when we get `/about`, and when we make a get request for `/bad`.

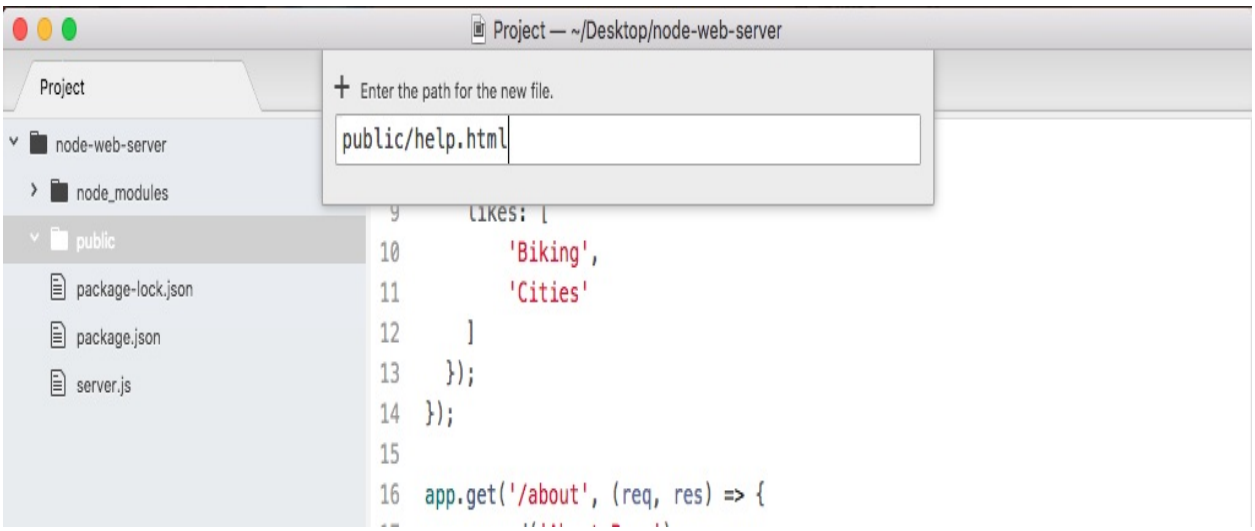
The static server

In this section, we'll learn how to set up a static directory. So if we have a website with HTML, CSS, JavaScript, and images, we can serve that up without needing to provide a custom route for every single file, which would be a real burden. Now setting this up is really simple. But before we make any updates to `server.js`, we'd create some static assets inside of our project that we can actually serve up.

Making an HTML page

In this case we'll make one HTML page that we'll be able to view in the browser. Before we get started, we do need to create a new directory, and everything inside this directory will be accessible via the web server, so it's important to not put anything in here that you don't want prying eyes to see.

Everything in the directory should be intended to be view able by anybody. We'll create a public folder to store all of our static assets, and inside here we'll make an HTML page. We'll create a help page for our example project by creating a file called `help.html`:



Now in `help.html` we will make a quick basic HTML file, although we'll not touch on all of the subtleties of HTML, since this is not really an HTML book. Instead, we'll just set up a basic page.

The first thing we need to do is create a `DOCTYPE` which lets the browser know what version of HTML we're using. That will look something like this:

```
<!DOCTYPE html>
```

After the opening tag, and the exclamation mark, we'd type `DOCTYPE` in uppercase. Then, we provide the actual `DOCTYPE` for HTML5, the latest version. Then we can use the greater than sign to close things up. In the next line, we'll open up our

`html` tag so we can define our entire HTML file:

```
<!DOCTYPE html>
<html>
</html>
```

Inside `html`, there are two tags we'll use: the `head` tag which lets us configure our doc, and the `body` tag which contains everything we want to render to the screen.

The head tag

We'll create the `head` tag first:

```
<!DOCTYPE html>
<html>
  <head>

  </head>
</html>
```

Inside `head`, we'll provide two pieces of info, `charset` and `title` tag:

- First up we have to set up the `charset` which lets the browser know how to render our characters.
- Next up we'll provide the `title` tag. The `title` tag lets the browser know what to render in that title bar, where the new tab usually is.

As shown in the following code snippet, we'll set `meta`. And on `meta`, we'll set the `charset` property using equals, and provide the value `utf-8`:

```
  <head>
    <meta charset="utf-8">
  </head>
```

For the `title` tag, we can set it to whatever we like; `Help Page` seems appropriate:

```
  <head>
    <meta charset="utf-8">
    <title>Help Page</title>
  </head>
```

The body tag

Now that our `head` is configured, we can add something to the body of our website. This is the stuff that's actually going to be viewable inside the viewport. Next to the head, we'll open and close the `body` tag:

```
  <body>

  </body>
```

Inside `body` again, we'll provide two things: an `h1` title and a `p` paragraph tag.

The title is going to match the `title` tag we used in the `head`, Help Page, and the paragraph will just have some filler text—`Some text here`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Help Page</title>
  </head>
  <body>
    <h1>Help Page</h1>
    <p>Some text here</p>
  </body>
</html>
```

Now we have an HTML page and the goal is to be able to serve this page up in our Express app without having to manually configure it.

Serving the HTML page in the Express app

We'll serve our HTML page in the Express app using a piece of Express middleware. Middleware lets us configure how our Express application works, and it's something we'll use extensively throughout the book. For now, we can think of it kind of like a third-party add-on.

In order to add some middleware, we'll call `app.use`. The `app.use` takes the middleware function we want to use. In our case, we'll use a built-in piece of middleware. So inside `server.js`, next to the variable `app` statement, we'll provide the function off of the `express` object:

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

var app = express();

app.use();
```

We will be making our own middleware in the next chapter, so it'll become clear exactly what's getting passed into use in a little bit. For now, we'll pass in `express.static` and to call it as a function:

```
var app = express();

app.use(express.static());
```

Now `express.static` takes the absolute path to the folder you want to serve up. If we want to be able to serve up `/help`, we'll need to provide the path to the `public` folder. This means we need to specify the path from the root of our hard drive, which can be tricky because your projects move around. Luckily we have the `__dirname` variable:

```
app.use(express.static(__dirname));
```

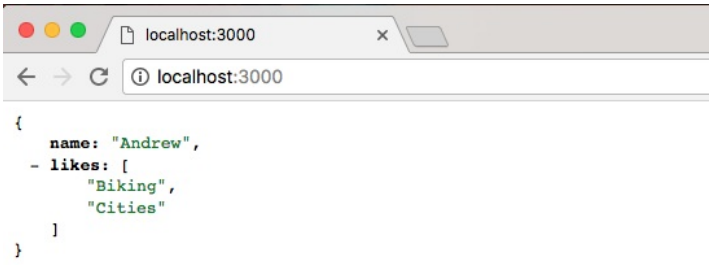
This is the variable that gets passed into our file by the wrapper function we explored. The `__dirname` variable stores the path to your projects directory. In this case, it stores the path to `node-web-server`. All we have to do is concatenate `/public` to tell it to use this directory for our server. We'll concatenate using the plus sign

and the string, `/public`:

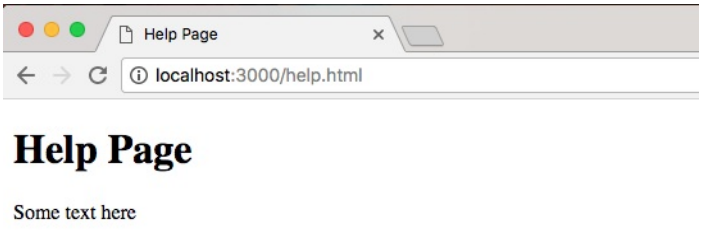
```
app.use(express.static(__dirname + '/public'));
```

With this in place, we are now done. We have our server set up and there's nothing else to do. Now we should be able to restart our server and access `/help.html`. We should now see the HTML page we have. In the Terminal we can now start the app using `nodemon server.js`:

Once the app is up and running we can visit it in the browser. We'll start by going to `localhost:3000`:



Here we get our JSON data, which is exactly what we expect. And if we change that URL to `/help.html` we should get our Help Page rendering:



And that is exactly what we get, we have our Help Page showing to the screen. We have the Help Page title as the head, and the Some text here paragraph following as body. Being able to set up a static directory that easily has made Node the go-to choice for simple projects that don't really require a backend. If you want to create a Node app for the sole purpose of serving up a directory you can do it in about four lines of code: the first three lines and the last line in the `server.js` file.

The call to app.listen

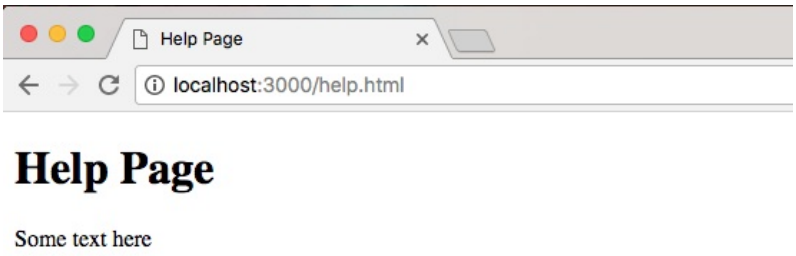
Now one more thing we'd discuss is the call to `app.listen(3000)`. The `app.listen` does take a second argument. It's optional. It's a function. This will let us do something once the server is up because it can take a little bit of time to get started. In our case we'll assign `console.log` a message: `Server is up on port 3000`:

```
app.listen(3000, () => {
  console.log('Server is up on port 3000');
});
```

Now it's really clear to the person who started the app that the server is actually ready to go because the message will print to the screen. If we save `server.js`, and go back into the Terminal we can see `Server is up on port 3000` prints:

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server is up on port 3000
```

Back inside the browser we can refresh and we get the exact same results:



That's it for this section. We now have a static directory where we can include JavaScript, CSS, images, or any other file types we like.

Rendering templates

In the last couple sections, we looked at multiple ways that we can render HTML using Express. We passed some HTML into `response.send`, but obviously that was not ideal. It's a real pain to write the markup in a string. We also created a public directory where we can have our static HTML files, such as our `help` file, and we can serve these up to the browser. Both of those work great but there is a third solution, and that will be the topic in this section. The solution is a templating engine.

A templating engine will let you render HTML but do it in a dynamic way, where we can inject values, such as a username or the current date, inside the template, kind of like we would in Ruby or PHP. Using this templating engine, we'll also be able to create reusable markup for things such as a header or a footer, which is going to be the same on a lot of your pages. This templating engine, handlebars, will be the topic of this section and the next, so let's get started.

Installing the hbs module

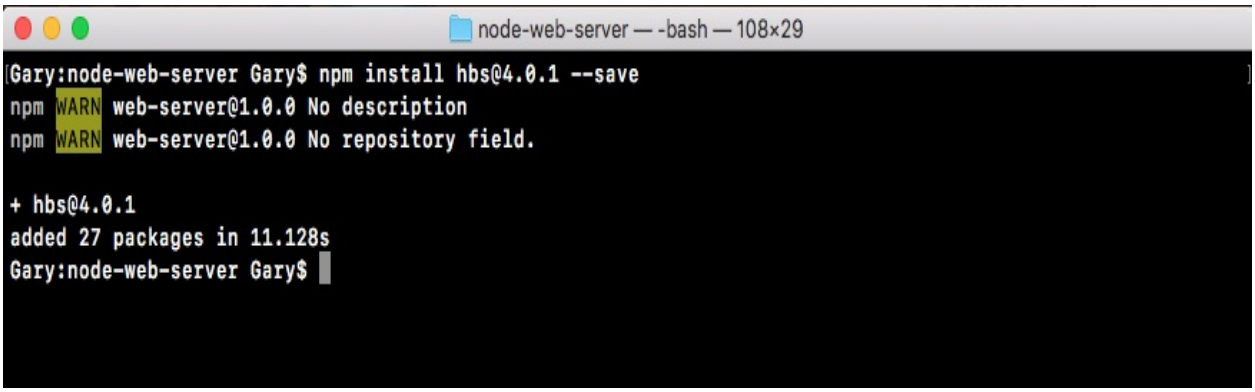
The first thing we'll do is install the `hbs` module. This is a handlebars view engine for Express. Now there are a ton of other view engines for Express, for example EJS or Pug. We'll go with handlebars because its syntax is great. It's a great way to get started.

Now we'll see a few things inside of the browser. First up we will visit `handlebarsjs.com`. This is the documentation for handlebars. It shows you exactly how to use all of its features, so if we want to use anything, we can always go here to learn how to use it.

Now we'll install a module that's a wrapper around handlebars. It will let us use it as an Express view engine. To view this, we'll go to `npmjs.com/package/hbs`.

This is the URL structure for all packages. So if you ever want to find a packages page, you simply type `npmjs.com/package/ the package name`.

This module is pretty popular. It's a really great view engine. They have a lot of documentation. I just want to let you know this exists as well. Now we can install and integrate it into our application. In the Terminal, we'll install `hbs` using `npm install`, the module name is `hbs`, and the most recent version is `@4.0.1`. I will use the `save` flag to update `package.json`:



Now actually configuring Express to use this handlebars view engine is super

simple. All we have to do is import it and add one statement to our Express configuration. We'll do just that inside Atom.

Configuring handlebars

Inside Atom, let's get started by loading in handlebars `const hbs = require hbs`, as shown and from here we can add that one line:

```
const express = require('express');
const hbs = require('hbs');
```

Next, let's call `app.set` where we call `app.use` for Express static:

```
app.set
app.use(express.static(__dirname + '/public'));
```

This lets us set some various Express-related configurations. There's a lot of built-in ones. We'll be talking about more of them later. For now, about what we'll do is pass in a key-value pair, where the key is the thing you want to set and the value is the value you want to use. In this case, the key we're setting is `view engine`. This will tell Express what view engine we'd like to use and we'll pass in inside of quotes `hbs`:

```
app.set('view engine', 'hbs');
app.use(express.static(__dirname + '/public'));
```

This is all we need to do to get started.

Our first template

Now in order to create our very first template, what we'd like to do is make a directory in the project called `views`. The `views` is the default directory that Express uses for your templates. So what we'll do is add the `views` directory and then we'll add a template inside it. We'll make a template for our About Page.

Inside views, we'll add a new file and the file name will be `about.hbs`. The `hbs` handlebars extension is important. Make sure to include it.

Now Atom already knows how to parse `hbs` files. At the bottom of the `about.hbs` file, where it shows the current language it's using, HTML in parentheses mustache.

Mustache is used as the name for this type of handlebars syntax because when you type the curly braces ({) I guess they kind of look like mustaches.

What we'll do to get started though is take the contents of `help.html` and copy it directly. Let's copy this file so we don't have to rewrite that boilerplate, and we'll paste it right in the `about.hbs`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Help Page</title>
  </head>
  <body>
    <h1>Help Page</h1>
    <p>Some text here</p>
  </body>
</html>
```

Now we can try to render this page. We'll change the `h1` tag from help page to about page:

```
  <body>
    <h1>About Page</h1>
    <p>Some text here</p>
  </body>
```

We'll talk about how to dynamically render stuff inside this page later. Before that we'd like to just get this rendering.

Getting the static page for rendering

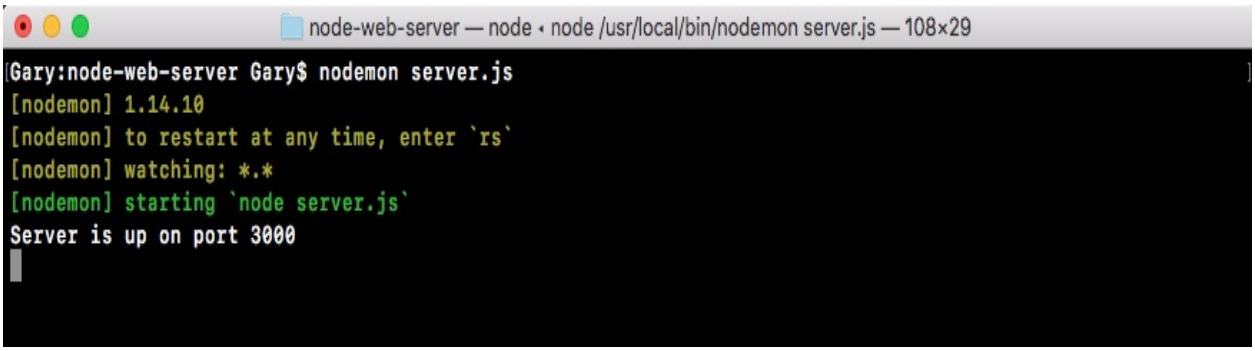
Inside `server.js`, we already have a root for `/about`, which means we can render our hbs template instead of sending back this about page string. We will remove our call to `res.send` and we'll replace it with `res.render`:

```
app.get('/about', (req, res) => {
  res.render
});
```

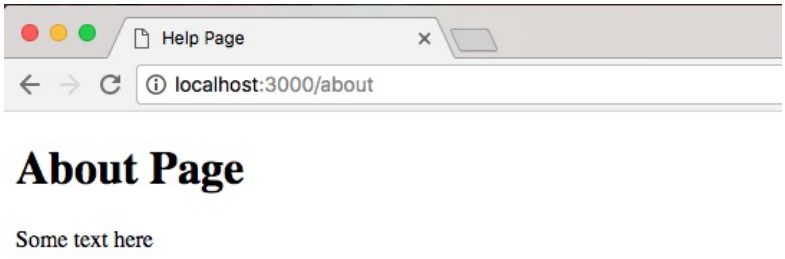
Render will let us render any of the templates we have set up with our current view engine `about.hbs` file. We do indeed have the about template and we can pass that name, `about.hbs`, in as the first and only argument. We'll render `about.hbs`:

```
app.get('/about', (req, res) => {
  res.render('about.hbs');
});
```

This will be enough to get that static page rendering. We'll save `server.js` and in the Terminal, we'll clear the output and we'll run our server using `nodemon server.js`:



Once the server is up and running, it is showing on port `3000`. We can open up this `/about` URL and see what we get. We'll head into Chrome and open up `localhost:3000 /about`, and when we do that, we get the following:



We get my about page rendered just like we'd expect it. We've got an `h1` tag, which shows up nice and big, and we have our paragraph tag, which shows up the following. So far we have used hbs but we haven't actually used any of its features. Right now, we're rendering a dynamic page, so we might as well have not even included it. What I want to do is talk about how we can inject data inside of our templates.

Injecting data inside of templates

Let's come up with some things that we want to make dynamic inside our handlebars file. First up, we'll make this `h1` tag dynamic so the page name gets passed into the template in `about.hbs` page, and we'll also add a footer. For now, we'll just make that a simple `footer` tag:

```
    <footer>

    </footer>
  </body>
</html>
```

Inside of the `footer`, we'll add a paragraph and that paragraph will have the copyright for our website. We'll just say something like copyright followed by the year, which is 2018:

```
    <footer>
      <p>Copyright 2018</p>
    </footer>
```

Now year should also be dynamic, so that as the years change, we don't have to manually update our markup. We'll look at how to make both the 2018 and the about page dynamic, which means they're getting passed in instead of being typed in the handlebars file.

In order to do this, we'll have to do two things:

- We'll have to pass some data into the template. This will be an object a set of key value pairs, and
- We'll have to learn how to pull off some of those key-value pairs inside of our handlebars file

Passing in data is pretty simple. All we have to do is specify a second argument to `res.render` in `server.js`. This will take an object, and on this object we can specify whatever we like. We might have a `pageTitle` that gets set equal to `About Page`:

```
app.get('/about', (req, res) => {
  res.render('about.hbs', {
    pageTitle: 'About Page'
```

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

We have one piece of data getting injected in the template. It's not used yet but it is indeed getting injected. We could also add another one like `currentYear`. We'll put `currentYear` next to the `pageTitle` and we'll set `currentYear` equal to the actual year off of the date JavaScript constructor. This will look something like this:

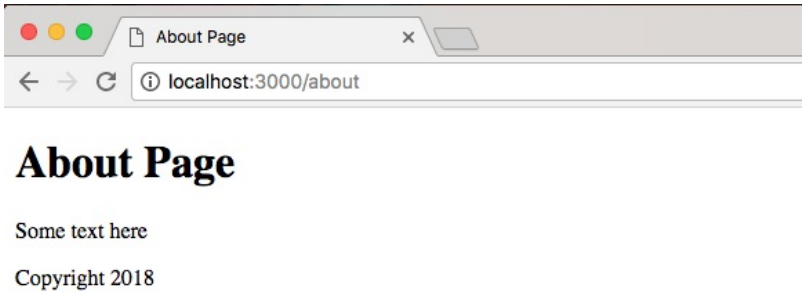
```
app.get('/about', (req, res) => {
  res.render('about.hbs', {
    pageTitle: 'About Page',
    currentYear: new Date().getFullYear()
  });
});
```

We'll create a new date which makes a new instance of the date object. Then, we'll use a method called `getFullYear`, which returns the year. In this case, it would return `2018`, just like this `.getFullYear`. Now we have a `pageTitle` and a `currentYear`. These are both getting passed in, and we can use them.

In order to use these pieces of data, what we have to do inside of our template is use that handlebars syntax which looks a little bit like shown in the following code. We start by opening up two curly braces in the `h1` tag, then we close two curly braces. Inside the curly braces, we can reference any of the props we passed in. In this case, let's use `pageTitle`, and inside our copyright paragraph, we'll use, inside of double curly braces, `currentYear`:

```
  <body>
    <h1>{{pageTitle}}</h1>
    <p>Some text here</p>

    <footer>
      <p>Copyright 2018</p>
    </footer>
  </body>
</html>
```

With this in place, we now have two pieces of dynamic data getting injected inside our application. Now nodemon should have restarted in the background, so there's no need to manually do anything there. When we refresh the page, we do still get About Page, which is great:



This comes from the data we defined in `server.js`, and we get Copyright 2018 showing up. Well this web page is pretty simple and doesn't look that interesting. At least you know how to create those servers and inject that data inside your web page. All you have to do from here is add some custom styles to get things looking nice.

Before we go ahead, let's move into the about file and swap out the title. Currently, it says `Help Page`. That's left over from the public folder. Let's change it to `Some Website`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Some Website</title>
  </head>
  <body>
    <h1>{{pageTitle}}</h1>
    <p>Some text here</p>

    <footer>
      <p>Copyright 2018</p>
    </footer>
  </body>
</html>
```

Now that we have this in place. Next, we'll create a brand new template and that template is going to get rendered when someone visits the root of our website, the `/` route. Now currently, we render some JSON data:

```
app.get('/', (req, res) => {
  // res.send('<h1>Hello Express!</h1>');
  res.send({
    name: 'Andrew',
    likes: [
      'Biking',
```

```
      'Cities'
    ]
  });
```

What we want to do is replace this with a call to `response.render`, rendering a brand new view.

Rendering the template for the root of the website

To get started, we'll duplicate the `about.hbs` file so we can start customizing it for our needs. We'll duplicate it, and call this one `home.hbs`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Some Website</title>
  </head>
  <body>
    <h1>{{pageTitle}}</h1>
    <p>Some text here</p>

    <footer>
      <p>Copyright 2018</p>
    </footer>
  </body>
</html>
```

Now from here most things are going to stay the same. We'll keep the `pageTitle` in place. We'll also keep the `Copyright` and `footer` following. What we want to change though is this paragraph. It was fine that the `About Page` as a static one, but for the `home` page, we'll set it equal to, inside curly braces, the `welcomeMessage` property:

```
  <body>
    <h1>{{pageTitle}}</h1>
    <p>{{welcomeMessage}}</p>

    <footer>
      <p>Copyright {{currentYear}}</p>
    </footer>
  </body>
```

Now `welcomeMessage` is only going to be available on `home.hbs`, which is why we have specifying it in `home.hbs` but not in `about.hbs`.

Next up, we needed to call response render inside of the callback. This will let us actually render the page. We'll add `response.render`, passing in the template name we want to render. This one is called `home.hbs`. Then we'll pass in our data:

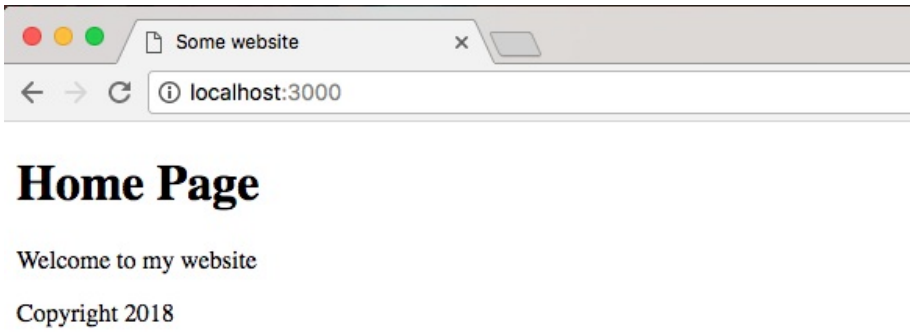
```
app.get('/', (req, res) => {
  res.render('home.hbs', {

```

```
  })
});
```

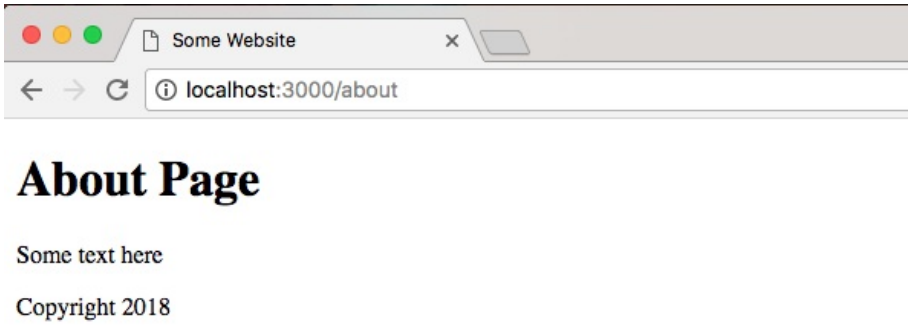
Now to get started, we can pass in the page title. We'll set this equal to `Home Page` and we'll pass in some sort of generic welcome message - `Welcome to my website`. Then we'll pass in the `currentYear`, and we already know how to fetch the `currentYear: new Date()`, and on the date object, we'll call the `getFullYear` method:

```
res.render('home.hbs', {
   pageTitle: 'Home Page',
   welcomeMessage: 'Welcome to my website',
   currentYear: new Date().getFullYear()
 })
```

With this in place, all we needed to do is save the file, which is automatically going to restart the server using nodemon and refresh the browser. When we do that, we get the following:



We get our Home Page title, our Welcome to my website message, and my copyright with the year 2018. And if we go to `/about`, everything still looks great. We have our dynamic page title and copyright and we have our static `some text here` text:



With this in place, we are now done with the very basics of handlebars. We see how this can be useful inside of a real-world web app. Aside from a realistic example such as the copyright, other reasons you might use this is to inject some sort of dynamic user data - things such as a username and email or anything else.

Now that we have a basic understanding about how to use handlebars to create static pages, we'll look at some more advanced features of hbs inside the next section.

Advanced templates

In this section, we'll learn a few more advanced features that handlebars has to offer. This will make it easier to render our markup, especially markup that's used in multiple places, and it will make it easier to inject dynamic data into your web pages.

In order to illustrate the first thing we'll talk about, I want to open up both `about.hbs` and `home.hbs`, and you'll notice down at the bottom that they both have the exact same footer code as follows:

```
<footer>
  <p>Copyright {{currentYear}}</p>
</footer>
```

We have a little copyright message for both and they both have the same header area, which is the `h1` tag.

Now this really isn't a problem because we have two pages, but as you add more and more pages it's going to become a real pain to update your header and your footer. You'll have to go into every file and manage the code there, but what we'll talk about instead is something called a partial.

Adding partials

A partial is a partial piece of your website. It's something you can reuse throughout your templates. For example, we might have a footer partial that renders the footer code. You can include that partial on any page you need a footer. You could do the same thing for header. In order to get started, the first thing we need to do is set up our `server.js` file just a little bit to let handlebars know that we want to add support for partials.

In order to do this, we'll add one line of code in the `server.js` file where we declared our view engine previously, and it will look something like this (`hbs.registerPartials`):

```
hbs.registerPartials
app.set('view engine', 'hbs');
app.use(express.static(__dirname + '/public'));
```

Now `registerPartials` is going to take the directory you want to use for all of your handlebar partial files, and we'll be specifying that directory as the first and only argument. Once again, this does need to be the absolute directory, so I'll use the `__dirname` variable:

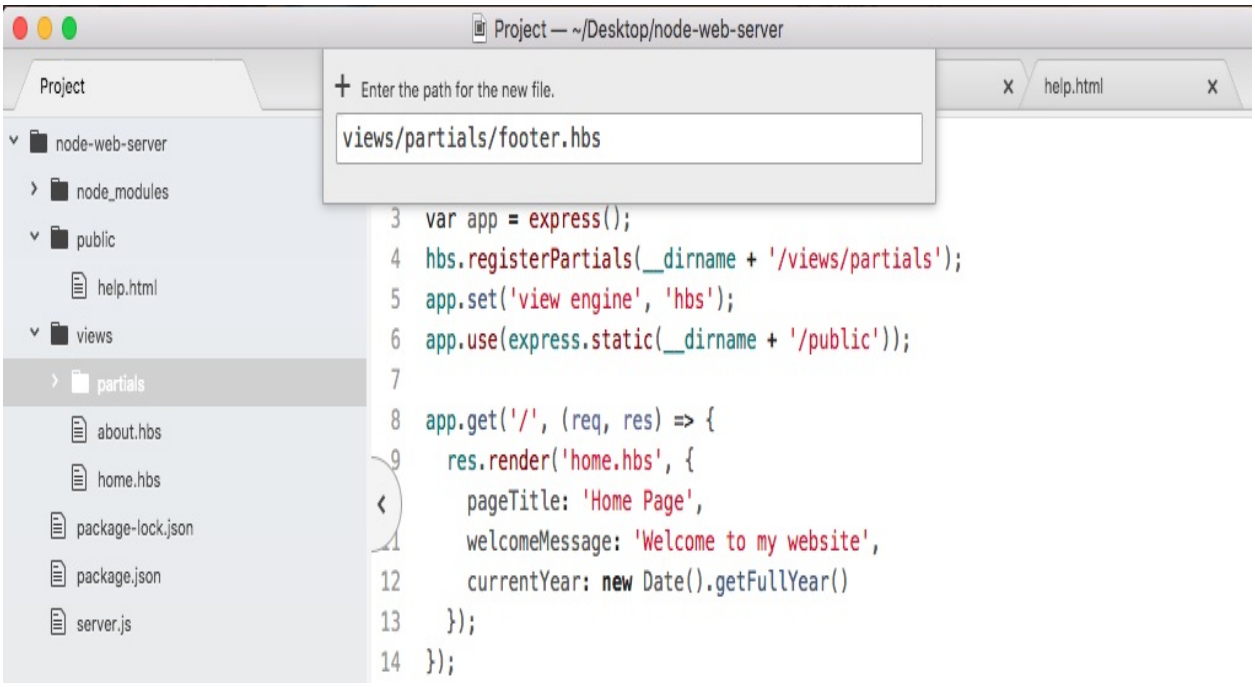
```
hbs.registerPartials(__dirname)
```

Then we can concatenate the rest of the path, which will be `/views`. In this case, I want you to use `/partials`.

```
hbs.registerPartials(__dirname + '/views/partials')
```

We'll store our `partial` files right inside a directory in the `views` folder. Now we can create that folder right in views called `partials`.

Inside `partials`, we can put any of the handlebars partials we like. To illustrate how they work, we'll create a file called `footer.hbs`:



Inside `footer.hbs`, we'll have access to the same handlebars features, which means we can write some markup, we can inject variables, we can do whatever we like. For now, what we'll do is copy the `footer` tag exactly, pasting it inside `footer.hbs`:

```
<footer>
  <p>Copyright {{getCurrentYear}}</p>
</footer>
```

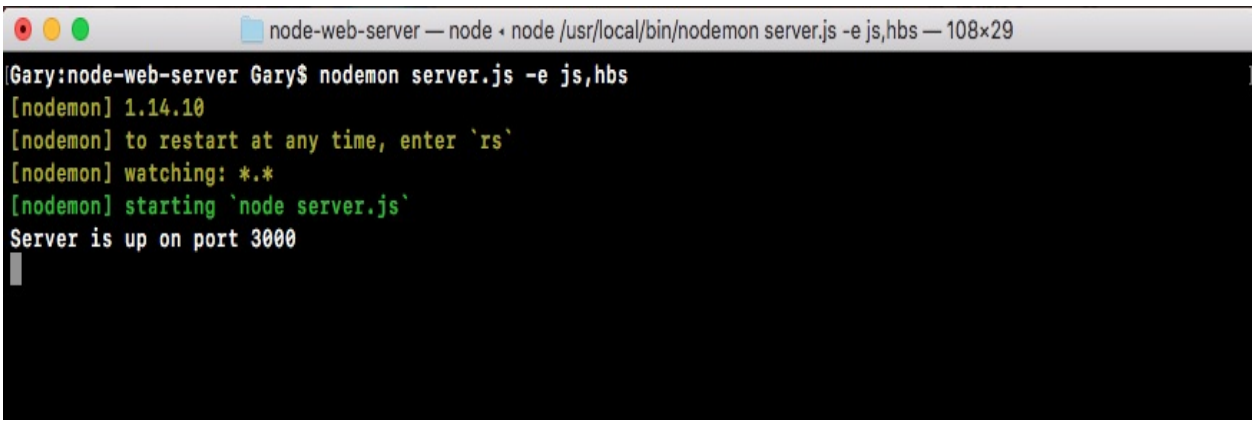
Now we have our `footer.hbs` file, this is the partial and we can include it in both `about.hbs` and `home.hbs`. In order to do that, we'll delete the code that we already have in the partial and we'll replace it with opening and closing two curly braces. Now instead of injecting data, we want to inject a template and the syntax for that is to add a greater than symbol with a space, followed by the partial name. In our case that partial is called `footer`, so we can add this right here:

```
    {{> footer}}
  </body>
</html>
```

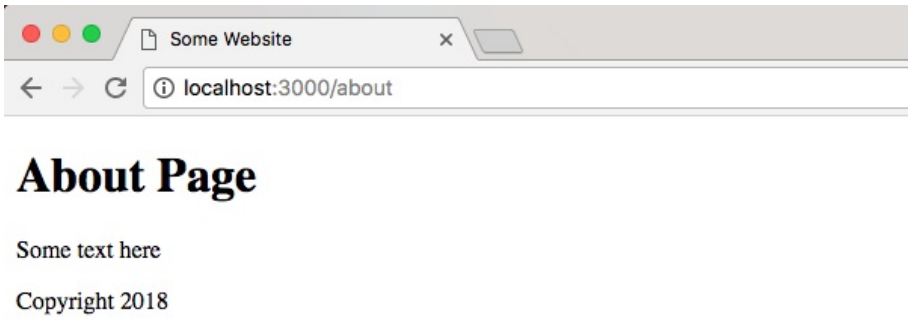
Then I can save about and do the same thing over in `home.hbs`. We now have our footer partial. It's rendering on both pages.

Working of partial

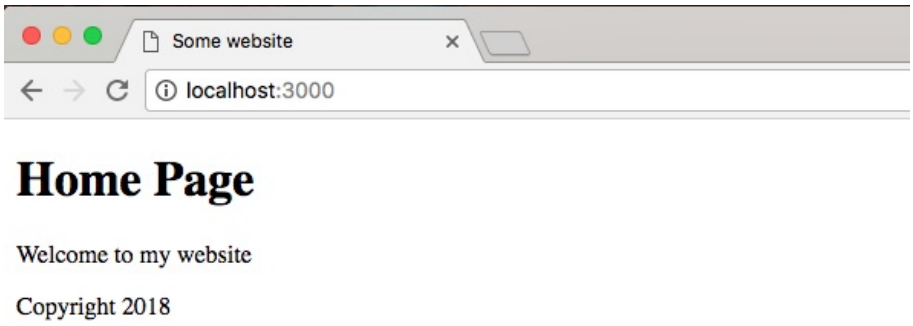
To illustrate how this works, I'll fire up my server and by default `nodemon`; it's not going to watch your handlebars files. So if you make a change, the website's not going to render as you might expect. We can fix this by running `nodemon`, passing in `server.js` and providing the `-e` flag. This lets us specify all of the extensions we want to watch. In our case, we'll watch the JS extension for the server file, and after the comma, the `hds` extension:



Now our app is up and running, we can refresh things over in the browser, and they should look the same. We have our about page with our footer:



We have our home page with the exact same footer:

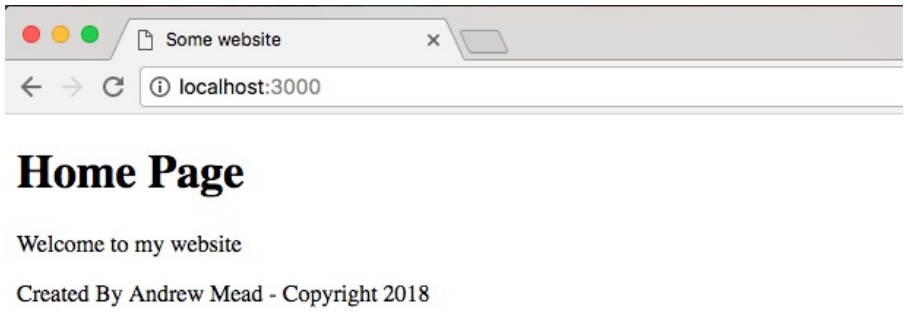


The advantage now is if we want to change that footer, we just do it in one place, in the `footer.hbs` file.

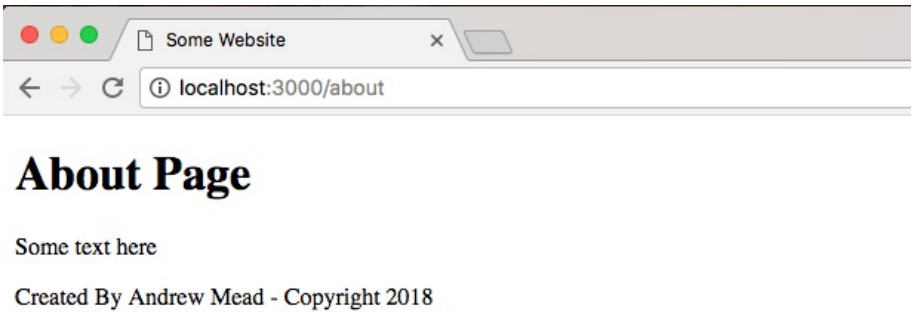
We can add something to our `footer` paragraph tag. Let's add a little message created by `Andrew Mead` with a `-`:

```
<footer>
 <p>Created By Andrew Mead - Copyright {{CurrentYear}}</p>
</footer>
```

Now, save the file and when we refresh the browser, we have our brand new footer for Home Page:



We have our brand new footer for About Page:



It will show up for both the home page and the about page. There's no need to do you anything manual in either of these pages, and this is the real power of partials. You have some code, you want to reuse it inside your website, so you simply create a partial and you inject it wherever you like.

The Header partial

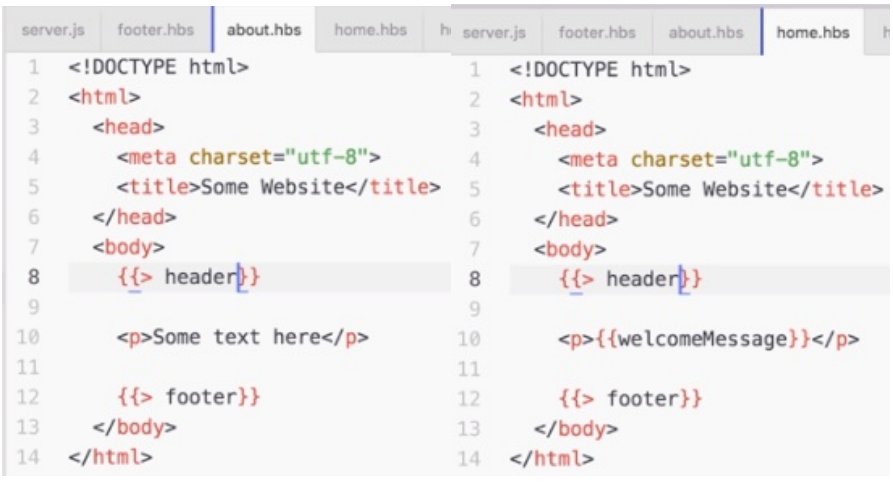
Now that we have the footer partial in place, let's create the header partial. That means we'll need to create a brand new file `header.hbs`. We'll want to add the `h1` tag inside that file and then we'll render the partial in both `about.hbs` and `home.hbs`. Both pages should still look the same.

We'll get started by creating a new file in the partials folder called `header.hbs`.

Inside `header.hbs`, we'll take the `h1` tag from our website, paste it right inside and save it:

```
<h1>{{pageTitle}}</h1>
```

Now we can use this header partial in both `about` and `home` files. Inside of `about`, we need to do this using the syntax, the double curly braces with the greater than sign, followed by the partial name `header`. We'll do the exact same thing for the `home` page. In the `home` page, we'll delete our `h1` tag, inject the `header` and save the file:

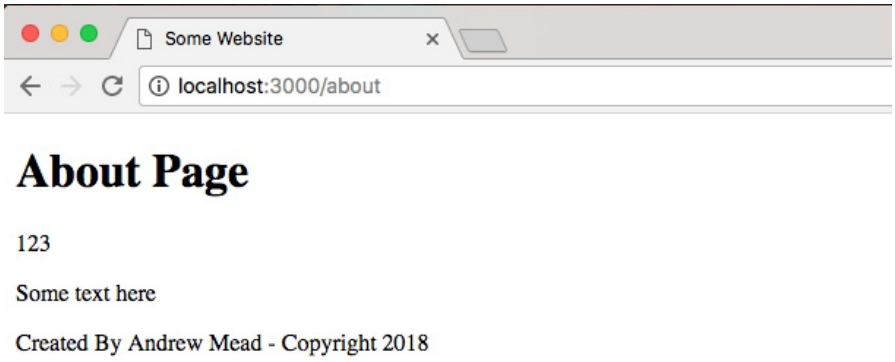


Now we'd create something slightly different just so we can test that it actually is using the partial. We'll type `123` right after the `h1` tag in `header.hbs`:

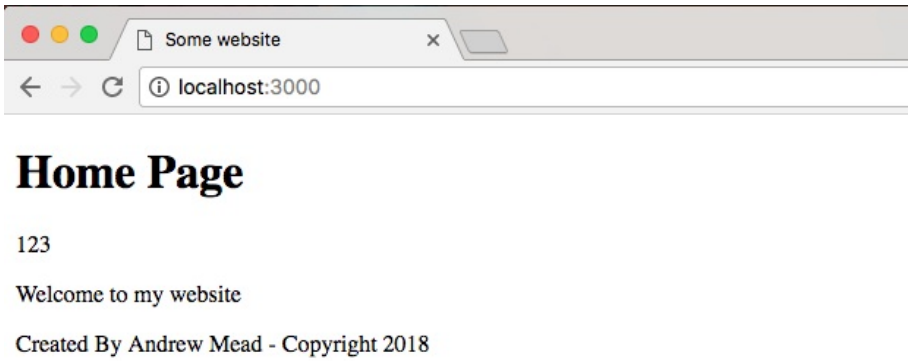
```
<h1>{{pageTitle}}</h1>123
```

Now that all the files are saved, we should be able to refresh the browser, and we

see `about` page with 123 printing, which is fantastic:



This means the `header` partial is indeed working, and if I go back to the `home` page, everything still looks great:



Now that we have the header broken out into its own file, we can do all sorts of things. We can take our `h1` tag and put it inside of a `header tag`, which is the appropriate way to declare your header inside of HTML. As shown, we add an opening and closing `header` tag. We can take the `h1` and we can move it right inside:

```
<header>
 <h1>{{pageTitle}}</h1>
</header>
```

We could also add some links to the other pages on our website. We could add an anchor tag for the homepage by adding an `a` tag:

```
<header>
 <h1>{{pageTitle}}</h1>
 <p><a></a></p>
</header>
```

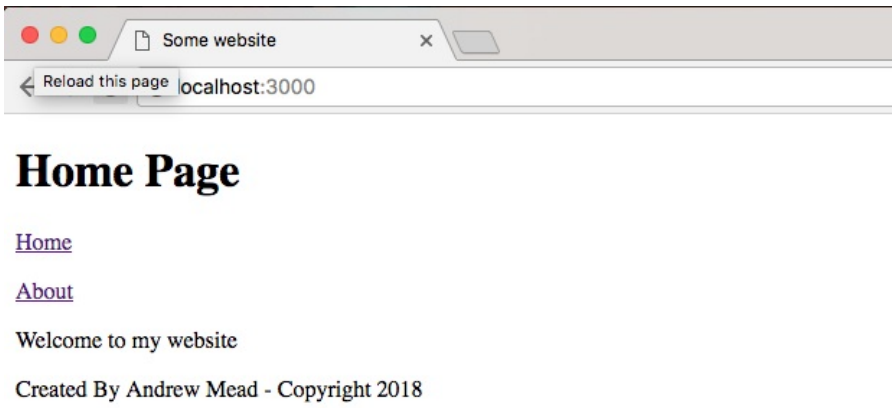
Inside the `a` tag, we'll specify the link text we'd like to show up. I'll go with `Home`, then inside the `href` attribute, we can specify the path the link should take you to, which would just be `/`:

```
<header>
 <h1>{{pageTitle}}</h1>
 <p><a href="/">Home</a></p>
</header>
```

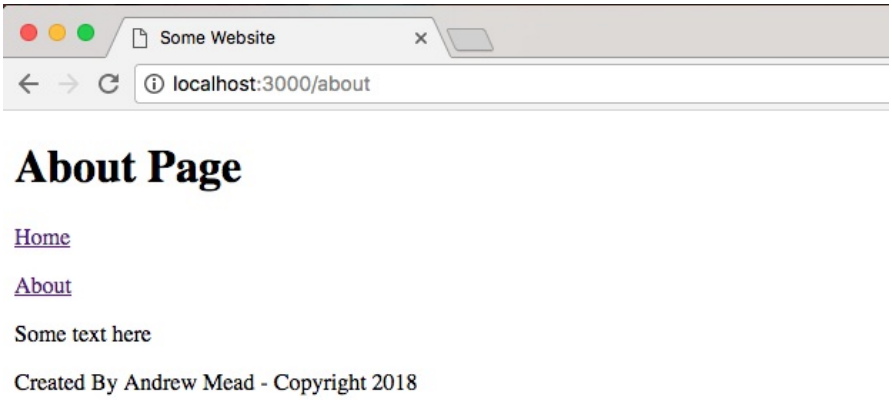
Then we can take the same paragraph tag, copy it and paste it in the next line and make a link for the `about` page. I'll change the page text to `About`, the link text, and the URL instead of going to `/` will go to `/about`:

```
<header>
 <h1>{{pageTitle}}</h1>
 <p><a href="/">Home</a></p>
 <p><a href="/about">About</a></p>
</header>
```

Now we've made a change to our `header` file and it will be available on all of the pages of our website. I'm on the `home` page. If I refresh it, I get Home and About page links:



I can click on the About to go to the About Page:



Similarly, I can click on Home to come right back. All of this is much easier to manage now that we have partials inside of our website.

The Handlebars helper

Now before we go further, there is one more thing I want to talk about, that is, a handlebars helper. Handlebars helpers are going to be ways for us to register functions to run to dynamically create some output. For example, inside `server.js`, we currently inject the current year inside of both of our `app.get` templates and that's not really necessary.

There is a better way to pass this data in, and this data shouldn't need to be provided because we'll always use the exact same function. We'll always take the new date `getfullYear` return value passing it in. Instead, we'll use a partial, and we'll set ours up right now. Now a partial is nothing more than a function you can run from inside of your handlebars templates.

All we need to do is register it and I'll do that in the `server.js`, following on from where we set up our Express middleware. As shown in the following code, we'll call `hbs.register` and we'll be registering a helper, so we'll call a `registerHelper`:

```
hbs.registerPartials(__dirname + '/views/partials')
app.set('view engine', 'hbs');
app.use(express.static(__dirname + '/public'));

hbs.registerHelper();
```

Now `registerHelper` takes two arguments:

- The name of the helper as the first argument
- The function to run as the second argument.

The first argument right here will be `getCurrentYear` in our case. We'll create a helper that returns that current year:

```
hbs.registerHelper('getCurrentYear',);
```

The second argument will be our function. I'll use an arrow function (`=>`):

```
hbs.registerHelper('getCurrentYear', () => {

});
```

Anything we return from this function will get rendered in place of the `getCurrentYear` call. That means if we call `getCurrentYear` inside the `footer`, it will return the year from the function, and that data is what will get rendered.

In the `server.js`, we can return the year by using `return` and having the exact same code we have `app.get` object:

```
hbs.registerHelper('getCurrentYear'), () => {
 return new Date().getFullYear()
});
```

We'll make a new date and we'll call its `getFullYear` method. Now that we have a helper, we can remove this data from every single one of our rendering calls:

```
hbs.registerHelper('getCurrentYear, () => {
 return new Date().getFullYear()
});

app.get('/', (req, res) => {
 res.render('home.hbs', {
   pageTitle: 'Home Page',
   welcomeMessage: 'Welcome to my website'
 });
});

app.get('/about', (req, res) => {
 res.render('about.hbs', {
   pageTitle: 'About Page'
 });
});
```

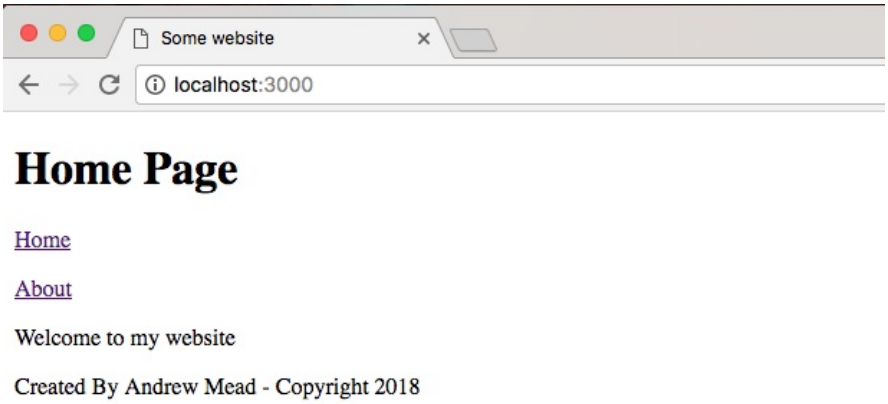
This is going to be really fantastic because there really is no need to compute it for every page since it's always the same. Now that we've removed that data from the individual calls to render, we will have to use `getCurrentYear` inside the `footer.hbs` file:

```
<footer>
 <p>Created By Andrew Mead - Copyright {{getCurrentYear}}</p>
</footer>
```

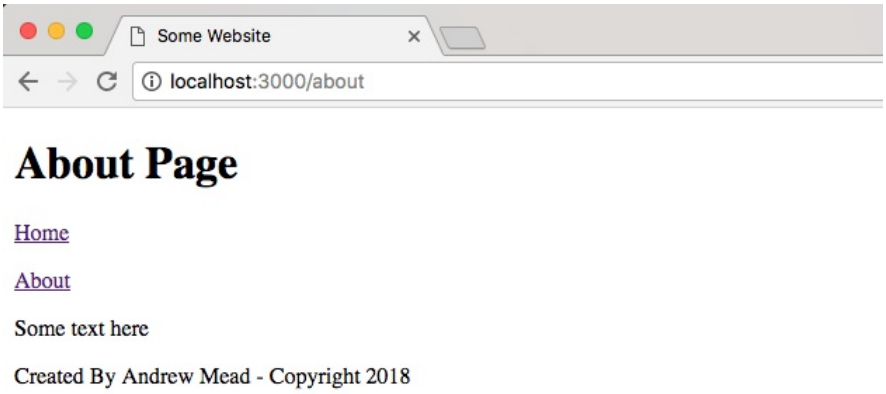
Instead referencing the current year, we will use the helper `getCurrentYear`, and there's no need for any special syntax. When you use something inside curly braces that clearly isn't a partial, handlebars is first going to look for a helper with that name. If there is no helper, it'll look for a piece of data with that `getCurrentYear` name.

In this case, it will find the helper, so everything will work as expected. We can now save `footer.hbs`, move into the browser, and give things a refresh. When I

refresh the page, we still get Copyright 2018 in Home Page:



If I go to the About Page, everything looks great:



We can prove that data is coming back from our helper by simply returning something else. Let's comment out our helper code in `server.js` and before the comment, we can use `return test`, just like this:

```
hbs.registerHelper('getCurrentYear', () => {
 return 'test';//return new Date().getFullYear()
});
```

We can now save `server.js`, refresh the browser, and we get tests showing up as shown here:

So the data that renders right after the Copyright word is indeed coming from that helper. Now we can remove the code so we return the proper year.

Arguments in Helper

Helpers can also take arguments, and this is really useful. Let's create a second helper that's going to be a capitalization helper. We'll call the helper `screamIt` and its job will be to take some text and it will return that text in uppercase.

In order to do this, we will be calling `hbs.registerHelper` again. This helper will be called `screamIt`, and it will take a function because we do need to run some code in order to do anything useful:

```
hbs.registerHelper('getCurrentYear', () => {
  return new Date().getFullYear()
});

hbs.registerHelper('screamIt', () => {

});
```

Now `screamIt` is going to take `text` to scream and all it will do is call on that string the `toUpperCase` method. We'll return `text.toUpperCase`, just like this:

```
hbs.registerHelper('screamIt', (text) => {
  return text.toUpperCase();
});
```

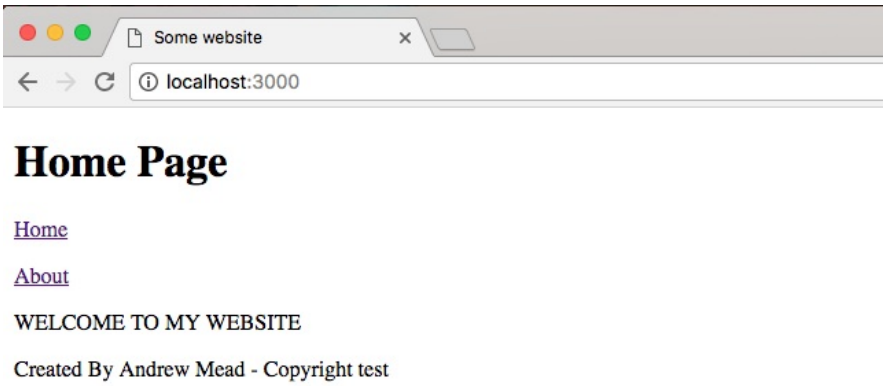
Now we can actually use `screamIt` in one of our files. Let's move into `home.hbs`. Here, we have our welcome message in the `p` tag. We'll remove it and we'll scream the welcome message. In order to pass data into one of our helpers, we first have to reference the helper by name, `screamIt`, then after a space we can specify whatever data we want to pass in as arguments.

In this case, we'll pass in the welcome message, but we could also pass in two arguments by typing a space and passing in some other variable which we don't have access to:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Some Website</title>
  </head>
  <body>
    {{> header}}
```

```
    <p>{{screamIt welcomeMessage}}</p>

    {{> footer}}
  </body>
</html>
```

For now, we'll use it like this, which means we'll call the `screamIt` helper, passing in one argument `welcomeMessage`. Now we can save `home.hbs`, move back into the browser, go to the Home Page and as shown following, we get WELCOME TO MY WEBSITE in all uppercase:



Using handlebars helpers, we can create both functions that don't take arguments and functions that do take arguments. So when you need to do something to the data inside of your web page, you can do that with JavaScript. Now that we have this in place, we are done.

Express Middleware

In this section, you'll learn how to use Express middleware. Express middleware is a fantastic tool. It allows you to add on to the existing functionality that Express has. So if Express doesn't do something you'd like it to do, you can add some middleware and teach it how to do that thing. Now we've already used a little bit of middleware. In `server.js` file, we used some middleware and we teach Express how to read from a `static` directory, which is shown here:

```
app.use(express.static(__dirname + '/public'));
```

We called `app.use`, which is how you register middleware, and then we provided the middleware function we want to use.

Now middleware can do anything. You could just execute some code such as logging something to the screen. You could make a change to the request or the response object. We'll do just that in the next chapter when we add API authentication. We'll want to make sure the right header is sent. That header will be expected to have an API token. We can use middleware to determine whether or not someone's logged in. Basically, it will determine whether or not they should be able to access a specific route, and we can also use middleware to respond to a request. We could send something back from the middleware, just like we would anywhere else, using `response.render` or `response.send`.

Exploring middleware

In order to explore middleware, we'll create some basic middleware. Just following where we call `app.use` registering our Express static middleware, we'll call `app.use` again:

```
app.use(express.static(__dirname + '/public'));

app.use();
```

Now `app.use` is how you register middleware, and it takes a function. So, we'll pass in an arrow function (`=>`):

```
app.use(() =>  {

});
```

The `use` function takes just one function. There is no need to add any other arguments. This function will get called with the request (`req`) object, the response (`res`) object and a third argument, `next`:

```
app.use((req, res, next) =>  {

});
```

Now request and response objects, these should seem familiar by now. They're the exact same arguments we get whenever we register a handler. The `next` argument is where things get a little trickier. The `next` argument exists so you can tell Express when your middleware function is done, and this is useful because you can have as much middleware as you like registered to a single Express app. For example, I have some middleware that serves up a directory. We'll write some more that logs some request data to the screen, and we could have a third piece that helps with application performance, keeping track of response times, all of that is possible.

Now inside `app.use` function, we can do anything we like. We might log something to the screen. We might make a database request to make sure a user is authenticated. All of that is perfectly valid and we use the `next` argument to tell Express when we're done. So if we do something asynchronous, the middleware is not going to move on. Only when we call `next`, will the application continue to

run, like this:

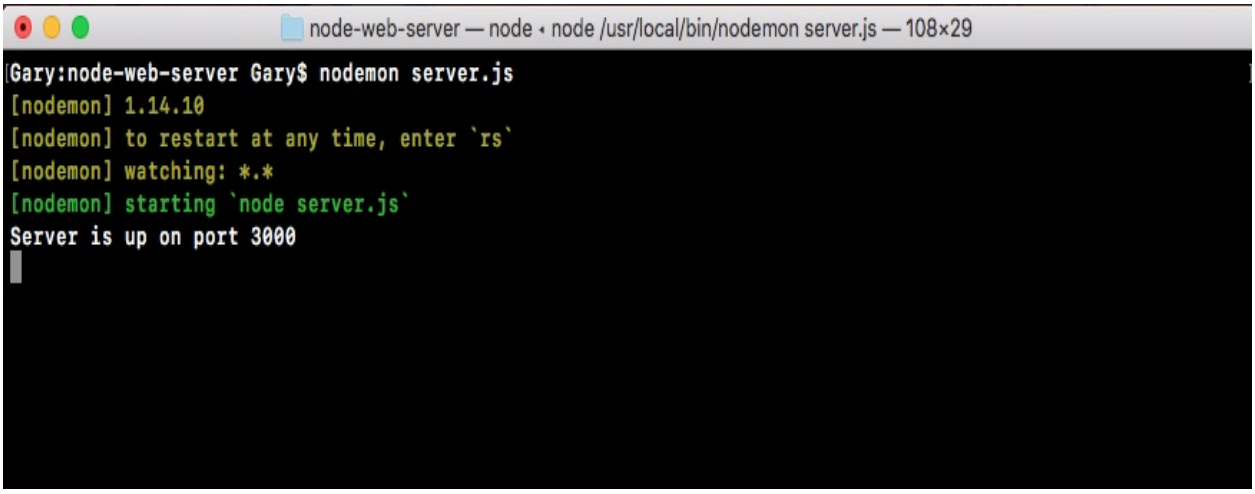
```
app.use((req, res, next) =>  {
  next();
});
```

Now this means if your middleware doesn't call next, your handlers for each request, they're never going to fire. We can prove this. Let's call `app.use`, passing in an empty function:

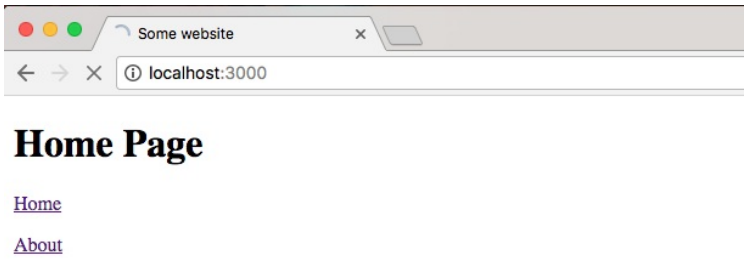
```
app.use((req, res, next) =>  {

});
```

Let's save the file and in the Terminal, we'll run our app using `nodemon` with `server.js`:

```
nodemon server.js
```



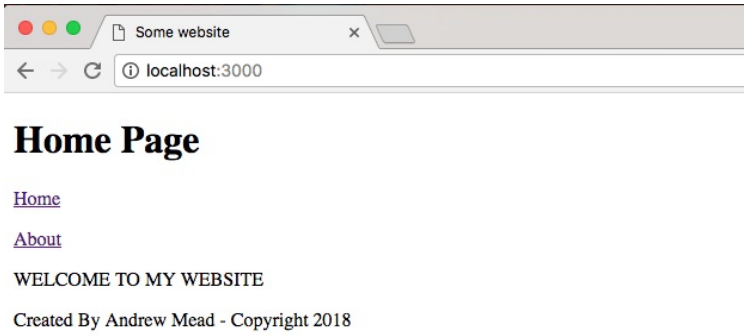
I'll move into the browser and I'll make a request for the home page. I'll refresh the page and you can see that up top, it is trying to load but it's never going to finish:



Now it's not that it can't connect to the server. It connects to the server just fine. The real problem is that inside our app, we have middleware that doesn't call next. To fix this, all we'll do is call `next` like this:

```
app.use((req, res, next) => {
  next();
});
```

Now when things refresh over inside the browser, we get our Home Page exactly as we expect it:



The only difference is now we have a place where we can add on some functionality.

Creating a logger

Inside `app.use`, we're going to get started by creating a logger that will log out all of the requests that come in to the server. We'll store a timestamp so we can see exactly when someone made a request for a specific URL.

To get started inside the middleware, let's get the current time. I'll make a variable called now, setting it equal to `newDate`, creating a new instance of our date object, and I'll call it `toString` method:

```
app.use((req, res, next) => {
 var now = new Date().toString();
 next();
});
```

The `toString` method creates a nice formatted date, a human-readable timestamp. Now that we have our now variable in place, we can start creating the actual logger by calling `console.log`.

Let's call `console.log`, passing in whatever I like. Let's pass in inside of ticks the `now` variable with a colon after:

```
app.use((req, res, next) => {
  var now = new Date().toString();

  console.log(`${now};`)
  next();
});
```

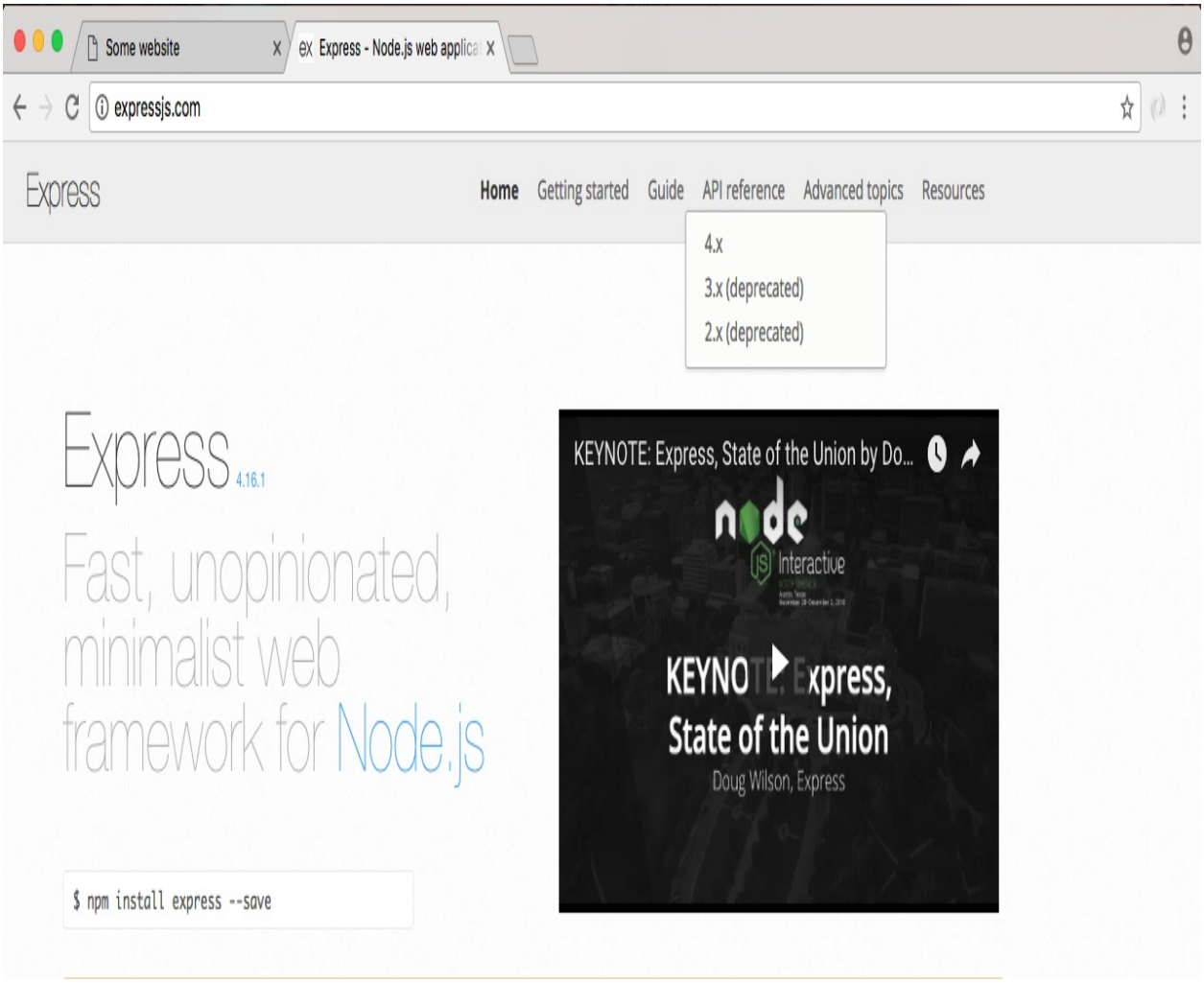
Now if I save my file, things are going to restart in the Terminal because `nodemon` is running. When we make a request for the site again and we go into the Terminal, we should see the log:

```
[nodemon] starting `node server.js`
Server is up on port 3000
Sun Jan 21 2018 23:11:33 GMT+0530 (IST);
```

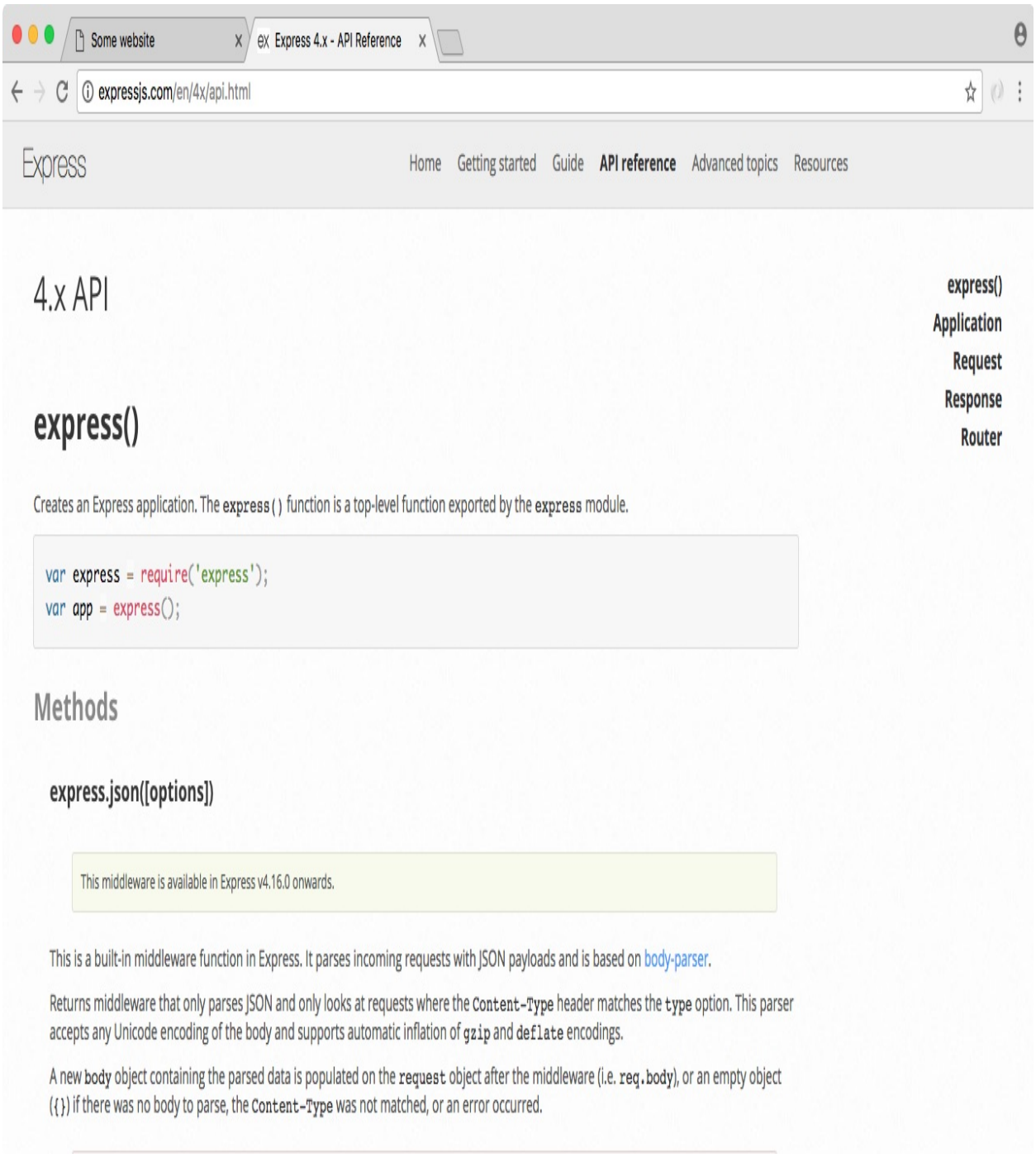
Currently it's just a timestamp, but we are on the right track. Now everything is working because we called `next`, so after this `console.log` call prints to the screen, our application continues and it serves up the page.

Inside middleware, we can add on more functionality by exploring the request object. On the request object, we have access to everything about the request—the HTTP method, the path, query parameters, and anything that comes from the client. Whether the client is an app, a browser, or an iPhone, it is all going to be available in that request object. Two things we'll pull off now are the HTTP method and the path.

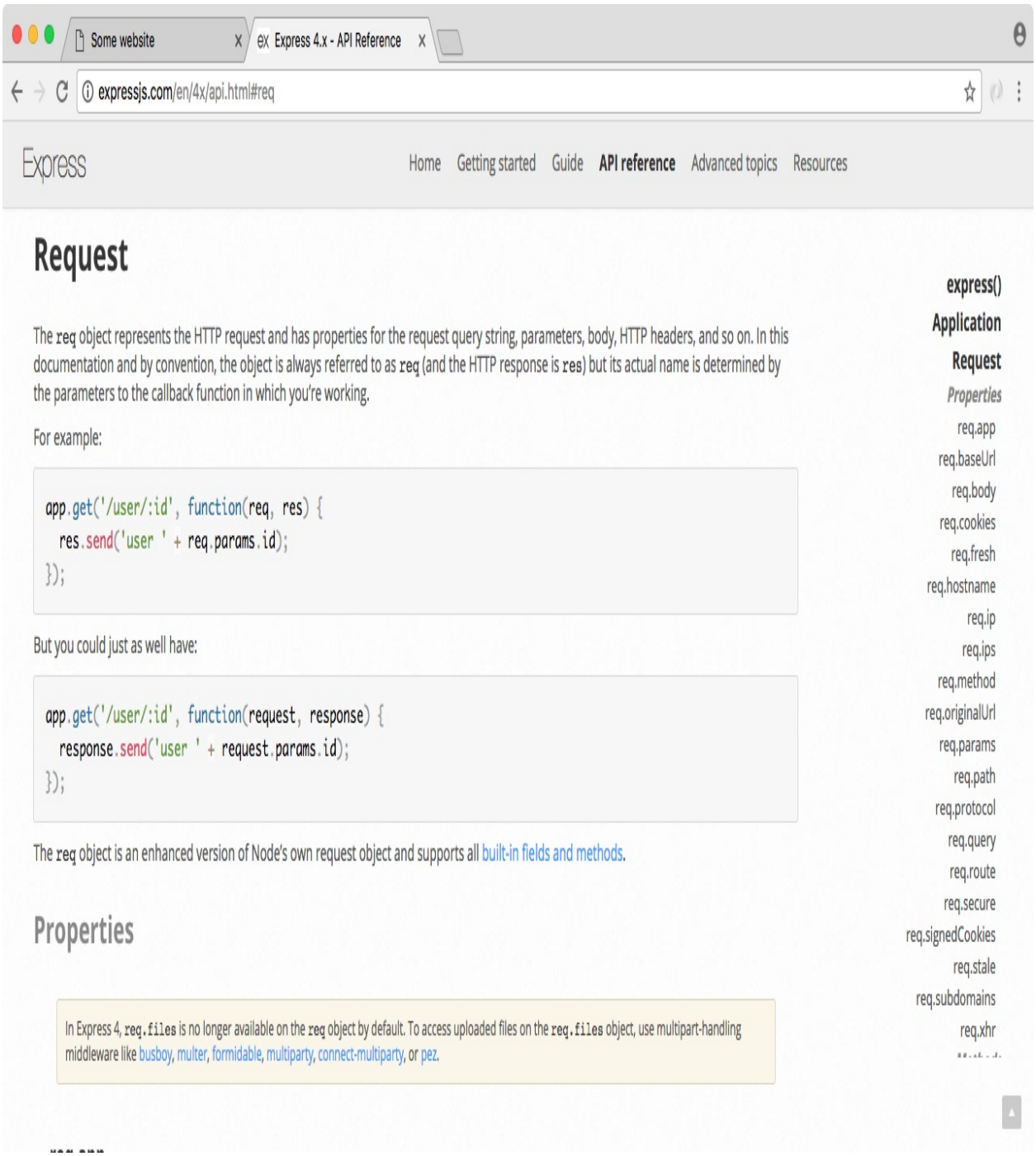
If you want to look at a full list of the things you have access to, you can go to `expressjs.com`, and go to API reference:



We happen to be using a 4.x version of Express, so we'll click that link:



On the right-hand side of this link, we have both Request and Response. We'll look for the request objects, so we'll click that. This'll lead us to the following:



We'll be using two request properties: `req.url` and `req.method`. Inside Atom, we can start implementing those, adding them into `console.log`. Right after the timestamp, we'll print the HTTP method. We'll be using other methods later. For now we've only used the `get` method. Right inside the `console.log`, I'll inject `request.method` printing it to the console:

```
app.use((req, res, next) => {
```

```
  var now = new Date().toString();

  console.log(`${now}: ${req.method}`)
  next();
});
```

Next up we can print the path so we know exactly what page the person requested. I'll do that by injecting another variable, `req.url`:

```
   console.log(`${now}: ${req.method} ${req.url}`);
```

With this in place, we now have a pretty useful piece of middleware. It takes the request object, it spits out some information and then it moves on, letting the server process that request which was added. If we save the file and rerun the app from the browser, we should be able to move into the Terminal and see this new logger printing to the screen, and as shown following we get just that:

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server is up on port 3000
Sun Jan 21 2018 23:16:30 GMT+0530 (IST): GET /
```

We have our timestamp, the HTTP method which is `GET`, and the path. If we change the path to something more complicated, such as `/about`, and we move back into the Terminal, we'll see the `/about` where we accessed `req.url`:

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server is up on port 3000
Sun Jan 21 2018 23:16:30 GMT+0530 (IST): GET /
Sun Jan 21 2018 23:17:22 GMT+0530 (IST): GET /about
```

Now this is a pretty basic example of some middleware. We can take it a step further. Aside from just logging a message to the screen, we'll also print the message to a file.

Printing message to file

To print the message to a file, let's load in `fs` up in the `server.js` file. We'll create a constant. Call that `const fs` and set that equal to the return result from requiring the module:

```
const express = require('express');
const hbs = require('hbs');
const fs = require('fs');
```

Now we can implement this down following in the `app.use`. We'll take our template string, which is currently defined inside `console.log`. We'll cut it out and instead store in a variable. We'll make a variable called `log`, setting it equal to that template string as shown here:

```
app.use((req, res, next) => {
  var now = new Date().toString();
  var log = `${now}: ${req.method} ${req.url}`;

  console.log();
  next();
});
```

Now we can pass that `log` variable into both `console.log` and into an `fs` method to write to our file system. For `console.log`, we will call log like this:

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

For `fs`, I'll call `fs.appendFile`. Now as you remember, `appendFile` lets you add on to a file. It takes two arguments: the file name and the thing we want to add. The file name we'll use is `server.log`. We'll create a nice log file and the actual contents will just be the `log` message. We will need to add one more thing: we also want to move on to the next line after every single request gets logged, so I'll concatenate the new line character, which will be `\n`:

```
  fs.appendFile('server.log', log + '\n');
```

> *If you're using Node V7 or greater, you will need to make a small tweak to this line. As shown in the following code, we added a third argument to `fs.appendFile`. This is a callback function. It's now required.*

```
fs.appendFile('server.log', log + '\n', (err) => {
  if (err) {
    console.log('Unable to append to server.log.')
  }
});
```

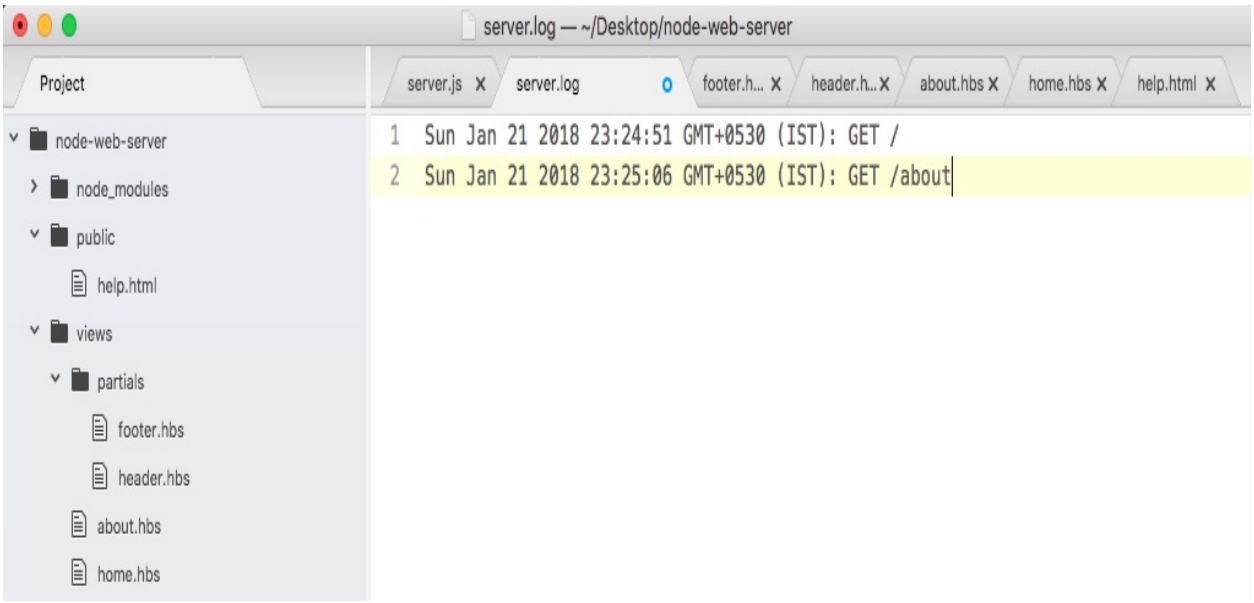
If you don't have a callback function, you'll get a deprecation warning over inside the console. Now as you can see, our callback function here takes an error argument. If there is an error, we just print a message to the screen. If you change your line to look like this, regardless of your Node version, you'll be future proof. If you're on Node V7 or greater, the warning in the console will go away. Now the warning is going to say something such as deprecation warning. Calling an asynchronous function without callback is deprecated. If you see that warning, make this change.

Now that we have this in place, we can test things out. I save the file, which should be restarting things inside of `nodemon`. Inside Chrome, we can give the page a refresh. If we head back into the Terminal, we do still get my log, which is great:

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server is up on port 3000
Sun Jan 21 2018 23:26:08 GMT+0530 (IST): GET /about
Sun Jan 21 2018 23:26:10 GMT+0530 (IST): GET /
```

Notice we also have a request for a `favicon.ico`. This is usually the icon that's shown in the browser tab. I have one cached from a previous project. There actually is no icon file defined, which is totally fine. The browser still makes the request anyway, which is why that shows up as shown in the previous code snippet.

Inside Atom, we now have our `server.log` file, and if we open it up, we have a log of all the requests that were made:



We have timestamps, HTTP methods, and paths. Using `app.use`, we were able to create some middleware that helps us keep track of how our server is working.

Now there are times where you might not want to call next. We learned that we could call next after we do something asynchronous, such as a read from a database, but imagine something goes wrong. You can avoid calling next to never move on to the next piece of middleware. We would like to create a new view inside the `views` folder. We'll call this one `maintenance.hbs`. This will be a handlebars template that will render when the site is in maintenance mode.

The maintenance middleware without the next object

We'll start with making the `maintenance.hbs` file by duplicating `home.hbs`. Inside `maintenance.hbs`, all we'll do is wipe the body and add a few tags:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Some Website</title>
 </head>
 <body>

  </body>
</html>
```

As shown in the following code, we'll add an `h1` tag to print a little message to the user:

```
 <body>
   <h1></h1>
 </body>
```

We're going to use something like `We'll be right back`:

```
 <body>
   <h1>We'll be right back</h1>
 </body>
```

Next, I can add a paragraph tag:

```
 <body>
   <h1>We'll be right back</h1>
   <p>

   </p>
 </body>
```

Now we will be able to use `p` followed by the tab. This is a shortcut inside Atom for creating an HTML tag. It works for all tags. We could type body and hit enter or I could type `p` and press enter, and the tag will be created.

Inside the paragraph, I'll leave a little message: `The site is currently being updated`:

```
<p>
  The site is currently being updated.
</p>
```

Now that we have our template file in place, we can define our maintenance middleware. This is going to bypass all of our other handlers, where we render other files and print JSON, and instead it'll just render this template to the screen. We'll save the file, move into `server.js`, and define that middleware.

Right next to the previously-defined middleware, we can call `app.use` passing in our function. The function will take those three arguments: request (`req`), response (`res`), and `next`:

```
app.use((req, res, next) => {

})
```

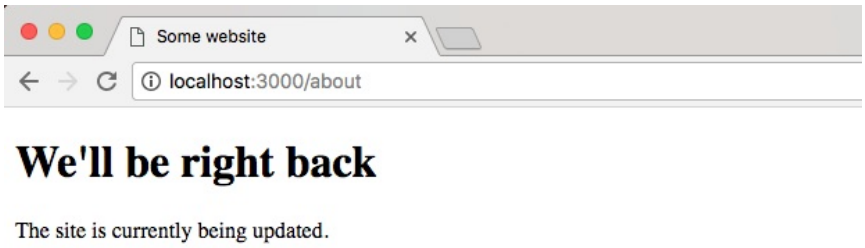
Inside the middleware, all we'll need to do is call `res.render`. We'll add `res.render` passing in the name of the file we want to render; in this case, it's `maintenance.hbs`:

```
app.use((req, res, next) => {
  res.render('maintenance.hbs');
});
```

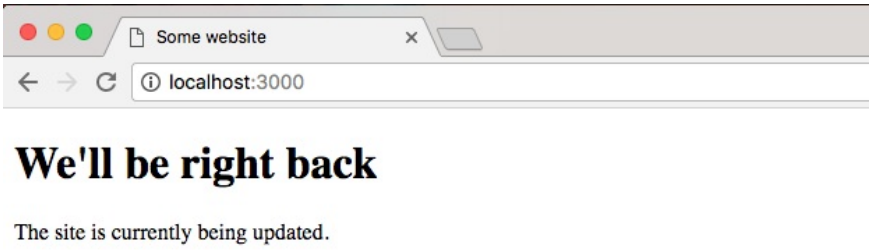
That is all you needed to do to set up our main middleware. This middleware will stop everything after it from executing. We don't call next, so the actual handlers in the `app.get` function, they will never get executed and we can test this.

Testing the maintenance middleware

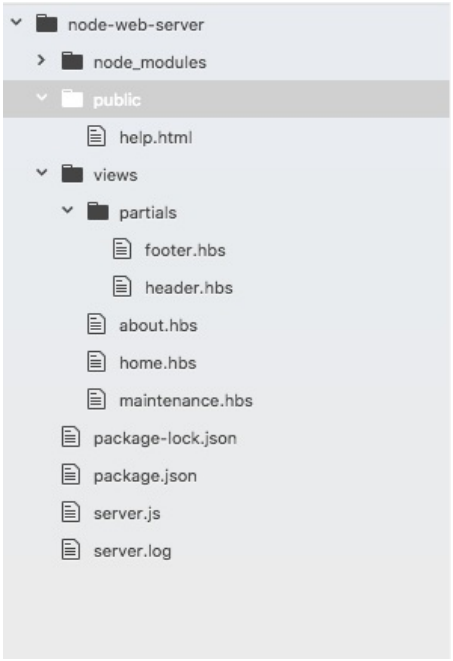
Inside the browser, we'll refresh the page, and we will get the following output:



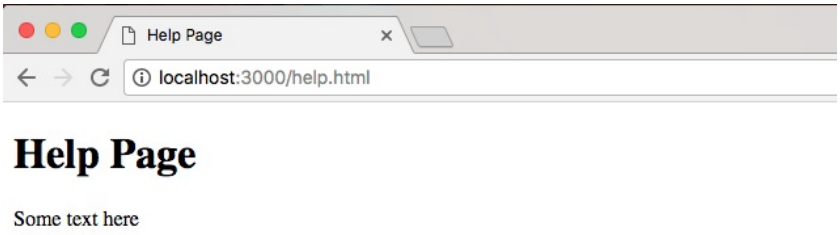
We get the maintenance page. We can go to the home page and we get the exact same thing:



Now there's one more really important piece to middleware we haven't discussed yet. Remember inside the `public` folder, we have a `help.html` file as shown here:



If we visit this in the browser by going to `localhost:3000/help.html`, we'll still get the help page. We'll not get the maintenance page:



That is because middleware is executed in the order you call `app.use`. This means the first thing we do is we set up our Express static directory, then we set up our logger, and finally we set up our `maintenance.hbs` logger:

```
app.use(express.static(__dirname + '/public'));

app.use((req, res, next) => {
  var now = new Date().toString();
  var log = `${now}: ${req.method} ${req.url}`;

  console.log(log);
  fs.appendFile('server.log', log + '\n');
  next();
});
```

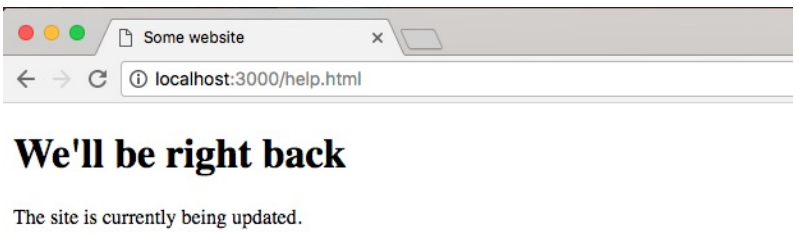
```
app.use((req, res, next) => {
  res.render('maintenance.hbs');
});
```

This is a pretty big problem. If we also want to make the `public` directory files such as `help.html` private, we'll have to reorder our calls to `app.use` because currently the Express server is responding inside of the Express static middleware, so our maintenance middleware doesn't get a chance to execute.

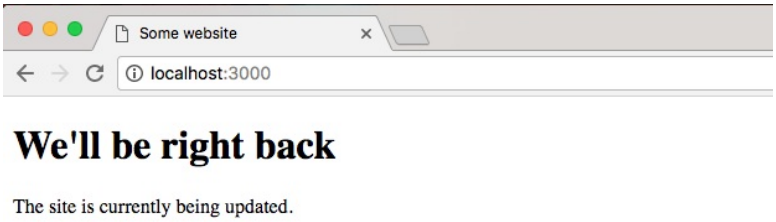
To resolve this, we'll take the `app.use` Express static call, remove it from the file, and add it after we render the maintenance file to the screen. The resultant code is going to look like this:

```
app.use((req, res, next) => {
  var now = new Date().toString();
  var log = `${now}: ${req.method} ${req.url}`;

  console.log(log);
  fs.appendFile('server.log', log + '\n');
  next();
});

app.use((req, res, next) => {
  res.render('maintenance.hbs');
});

app.use(express.static(__dirname + '/public'));
```

Now, everything will work as expected, no matter what we're going to log the request. Then we'll check if we're in maintenance mode if the maintenance middleware function is in place. If it is, we'll render the maintenance file. If it's not, we'll ignore it because it'll be commented out or something like that, and finally we'll be using Express static. This is going to fix all those problems. If I re-render the app now, I get the maintenance page on `help.html`:



If I go back to the root of the website, I still get the maintenance page:



Now once we're done with the maintenance middleware, we can always comment it out. This will remove it from being executed, and the website will work as expected.

This has been a quick dive into Express middleware. We'll be using it a lot more throughout the book. We'll be using middleware to check if our API requests are actually authenticated. Inside the middleware, we'll be making a database request, checking if the user is indeed who they say they are.

Summary

In this chapter you learned about Express and how it can be used to easily create websites. We looked at how we can set up a static web server, so when we have an entire directory of JavaScript, images, CSS, and HTML. We can serve that up easily without needing to provide routes for everything. This will let us create all sorts of applications, which we'll be doing throughout the rest of the book.

Next, we continued on learning how to use Express. We took a look at how we can render dynamic templates, kind of like we would with a PHP or Ruby on Rails file. We have some variables and we rendered a template injecting those variables. Then we learned a little bit about handlebars partials, which let us create reusable chunks of code like headers and footers. We also learned about Handlebars helpers, which is a way to run some JavaScript code from inside of your handlebars templates. Lastly, we moved back to talking about Express and how it can customize our requests, responses, our server.

In the next chapter, we'll look into deploying applications to the web.

Deploying Applications to Web

In this chapter, we'll worry about adding version control and deploying our applications because when it comes to creating real-world Node apps, deploying your app to the Web is obviously a pretty big part of that. Now in the real world, every single company uses some form of version control. It is essential to the software development process, and most of them aren't using Git. Git has become really popular, dominating the market share for version control. Git is also free and open source, and there is a ton of great educational material. They have a book on how to learn Git. It's free and Stack Overflow is filled with Git-specific questions and answers.

We'll be using Git to save our project. We'll also be using it to back up our work to a service called GitHub, and finally we'll be using Git to deploy our project live to the Web. So we'll be able to take our web server and deploy it for anybody to visit. It won't just be available on localhost.

Specifically, we'll look into the following topics:

- Setting up and using Git
- Setting up GitHub and SSH keys
- Deploying Node app to the web
- The workflow of the entire development life cycle

Adding version control

In this section, we'll learn how to set up and use Git, which is a version control system. Git will let us keep track of the changes to our project over time. This is really useful when something goes wrong and we need to revert to a previous state in the project where things were working. It's also super useful for backing up our work.

Installing Git

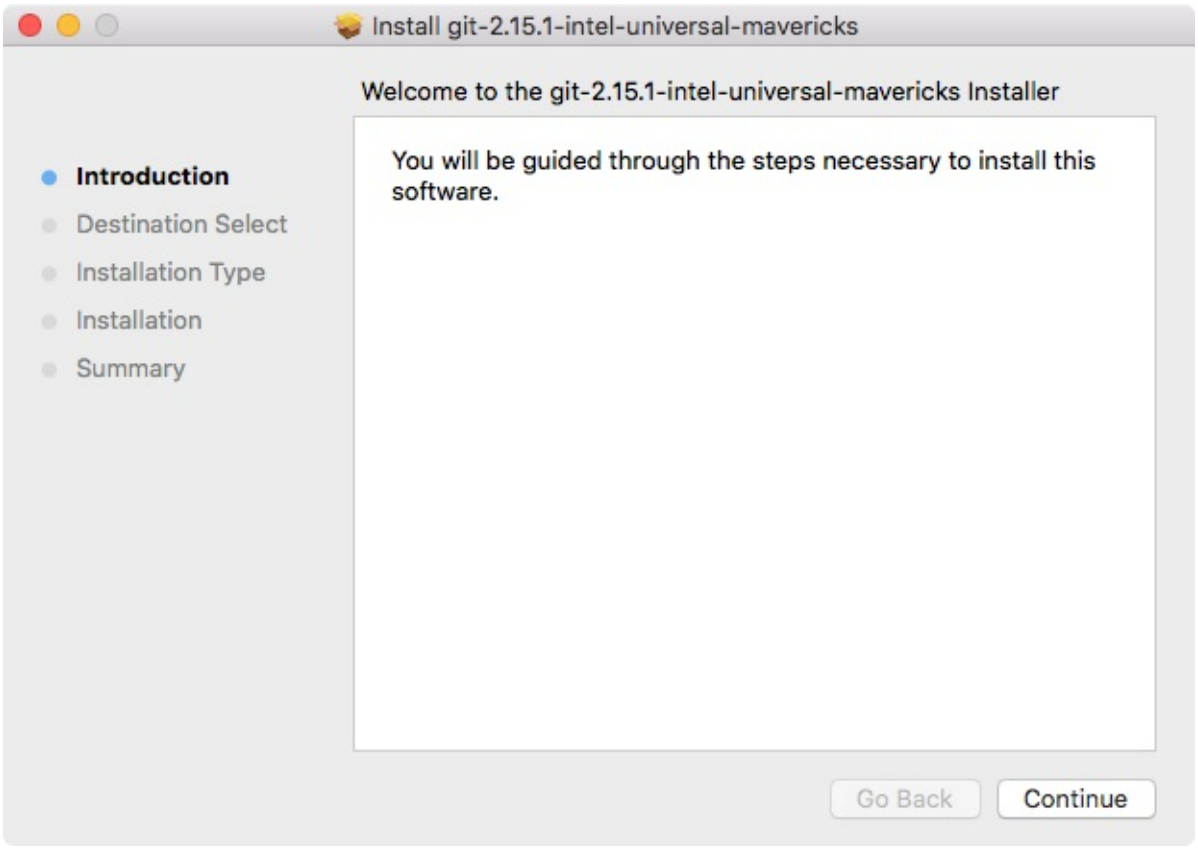
To get started, we will need to install Git on the computer, but luckily for us it is a really simple installation process. It's one of those installers where we just click on the Next button through a few steps. So let's go ahead and do that.

1. We can grab the installer by heading over to the browser and going to `git-scm.com`.

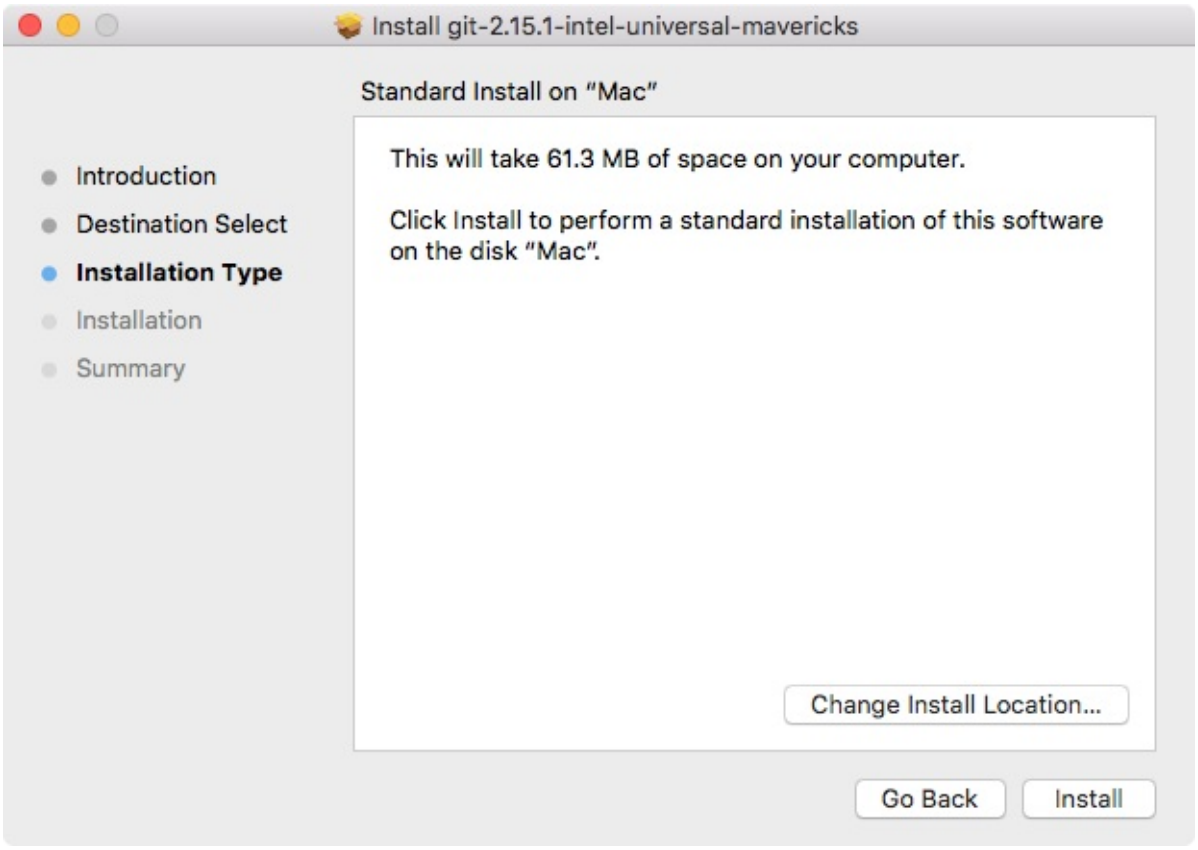
Before we go ahead and install it, I want to show you the link to the book called Pro Git (`https://git-scm.com/book/en/v2`). It is a free book and also available for online reading. It covers everything that Git has to offer. We'll be looking at some of the more basic features in this chapter, but we could easily create an entire course on Git. There actually are Udemy courses just on Git and GitHub, so if you want to learn more than what we cover in this book, I'd recommend reading this book or checking out a course, whatever your preferred learning method is.

2. Click on the download button present on the right-hand side of the home page, for all the operating systems, whether it's Windows, Linux, or macOS. This should take us to the installer page and we should be able to get the installer downloading automatically. If you see any problem with `SourceForge.net`, then we may have to actually click on it to download manually in order to start the download.

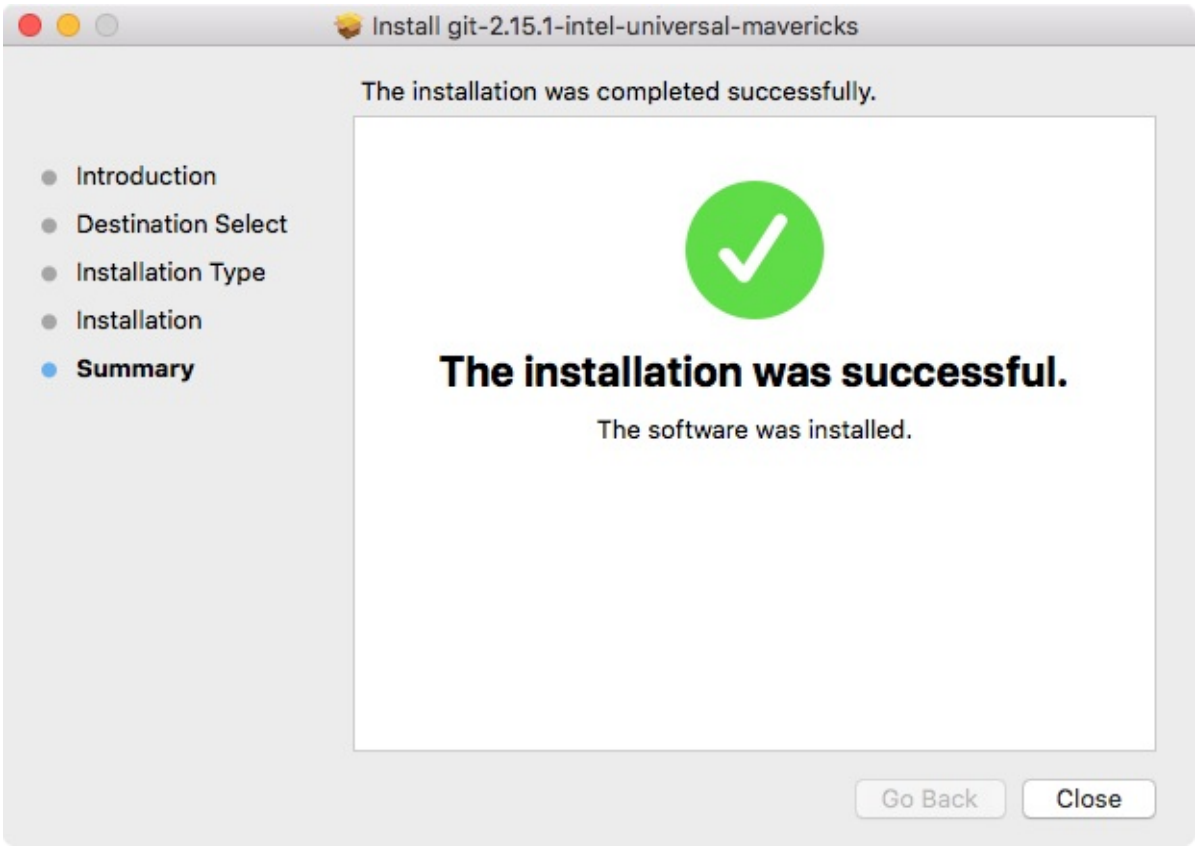
3. Once the installer is downloaded, we can simply run it.
4. Next, move through the installer:



5. Click on Continue and install the package:

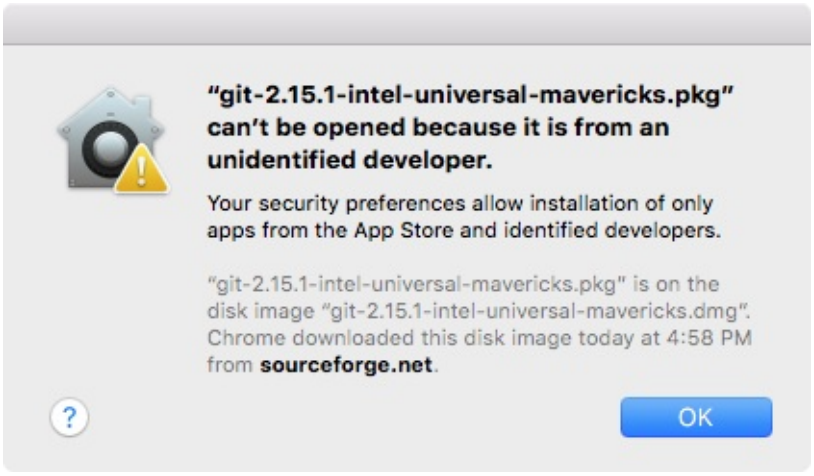


6. Once it's done, we can go ahead and actually test that things installed successfully:


Install git-2.15.1-intel-universal-mavericks
The installation was completed successfully.
Introduction
Destination Select
Installation Type
Installation
Summary
The installation was successful.
The software was installed.
Go Back
Close

Git on macOS

If you're on macOS, you'll need to launch the package installer and you might get the following message box saying that it's from an unidentified developer:

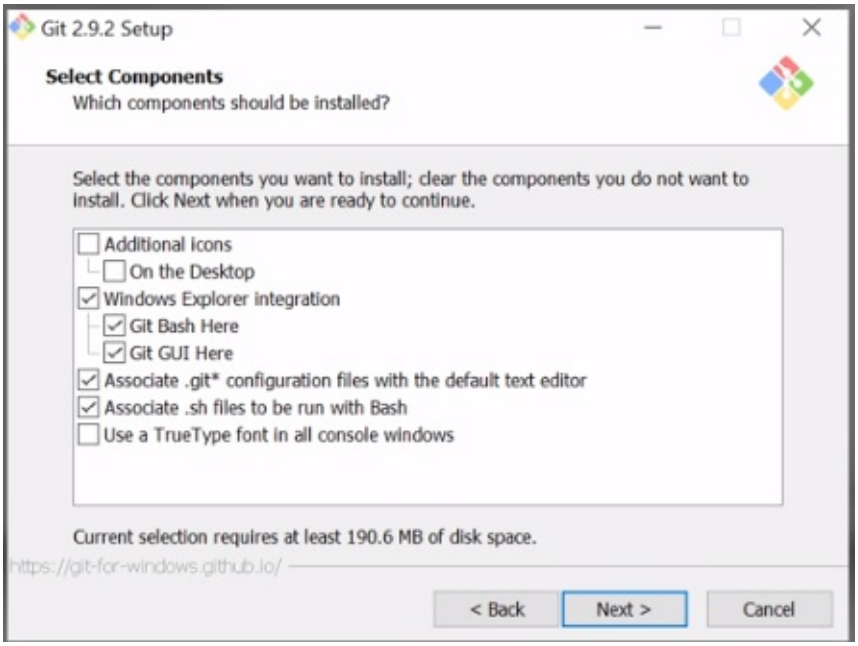


This is because it is distributed via a third party as opposed to being in the macOS App Store. We can go ahead and right-click on the package, then click on the Open button and confirm that we do indeed want to open it.

Once you're at the installer, the process is going to be pretty simple. You can essentially click on Continue and Next throughout every step.

Git on Windows

If you're on Windows though, there is an important distinction. Inside the installer you're going to see a screen just like this:



It is really important that you also install Git Bash as shown in the screenshot. Git Bash is a program that simulates a Linux-type Terminal, and it's going to be really essential when we create our SSH keys in the next section to uniquely identify our machine.

Testing the installation

Now, let's move in to the Terminal to test the installation. From the Terminal we can go ahead and run `git --version`. This is going to print a new version of Git we have installed:

```
git --version
```

As shown in the following screenshot, we can see we have git version 2.14.3:

Now if you have your Terminal still open and you're getting an error like git command not found, I'd recommend trying to restart your Terminal. Sometimes that is required when you're installing new commands such as the `git` command, which we just installed.

Turning the node-web-server directory into a Git repository

With successful installation of Git, we are now ready to turn our `node-web-server` directory into a Git repository. In order to do this, we'll the following command:

```
git init
```

The `git init` command needs to get executed from the root of our project, the folder that has everything that we want to keep track of. In our case, `node-web-server` is that folder. It has our `server.js` file, our `package.json` file, and all of our directories. So, from the server folder, we'll run `git init`:

```
Gary:node-web-server Gary$ git init
Initialized empty Git repository in /Users/Gary/Desktop/node-web-server/.git/
Gary:node-web-server Gary$
```

This creates a `.git` directory inside that folder. We can prove that by running the `ls -a` command:

```
ls -a
```

As shown in the following screenshot, we get all of the directories including the hidden ones and right here I do indeed have .git:

```
Gary:node-web-server Gary$ ls -a
.                   node_modules        public              views
..                  package-lock.json   server.js
.git                package.json        server.log
Gary:node-web-server Gary$
```

For Windows, go ahead and run these commands from the Git Bash.

Now this directory is not something we should be manually updating. We'll be using commands from the Terminal in order to make changes to the Git folder.

You don't want to be going in there manually messing around with things because there's a pretty good chance you're going to corrupt the Git repository and all of your hard work is going to become useless. Now obviously if it's backed up, it's not a big deal, but there really is no reason to go into that Git folder.

Let's use the `clear` command to clear the Terminal output, and now we can start looking at exactly how Git works.

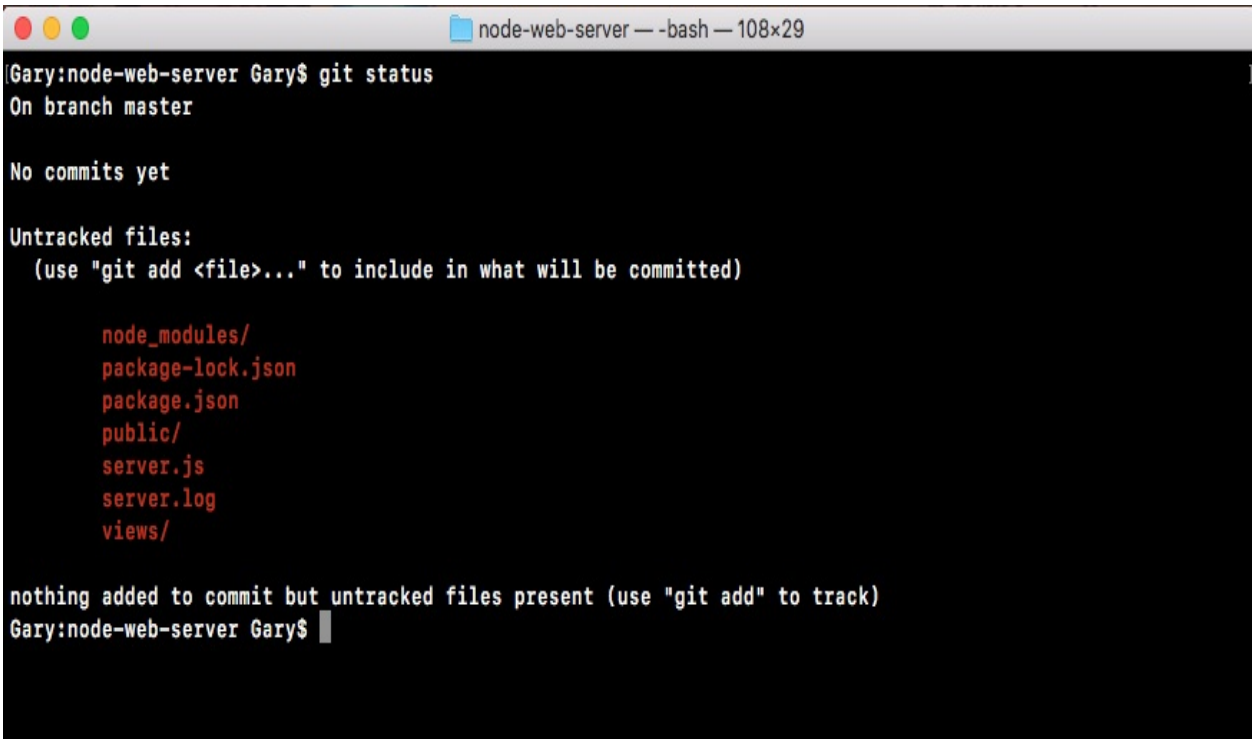
Using Git

As mentioned earlier, Git is responsible for keeping track of the changes to our project, but by default it doesn't actually track any of our files. We have to tell Git exactly which files we want it to keep track of and there's a good reason for this. There are files in every project that we're most likely not going to want to add to our Git repo, and we'll talk about which ones and why later. For now let's go ahead and run the following command:

```
git status
```

Now all these commands need to get executed from inside of the root of the project. If you try to run this outside a repository, you'll get an error like git repository not found. What that means is that Git cannot find that `.git` directory in order to actually get the status of your repository.

When we run this command, we'll get some output that looks like this:



The important pieces for now is the Untracked files header and all of the files

underneath it. These are all of the files and folders that Git seized, but it's currently not tracking. Git doesn't know if you want to keep track of the changes to these files or if you want to ignore them from your repository.

Now the `views` folder, for example, is something we definitely want to keep track of. This is going to be essential to the project and we want to make sure that whenever someone downloads the repository, they get the `views` folder. The log file on the other hand doesn't really need to be included in Git. In general our log files are not going to be committed, since they usually contain information specific to a point in time when the server was running.

As shown in the preceding code output, we have `server.js`, our public folder, and `package.json`. These are all essential to the process of executing the app. These are definitely going to be added to our Git repository, and the first one above we have is the `node_modules` folder. The `node_modules` folder is what's called a generated folder.

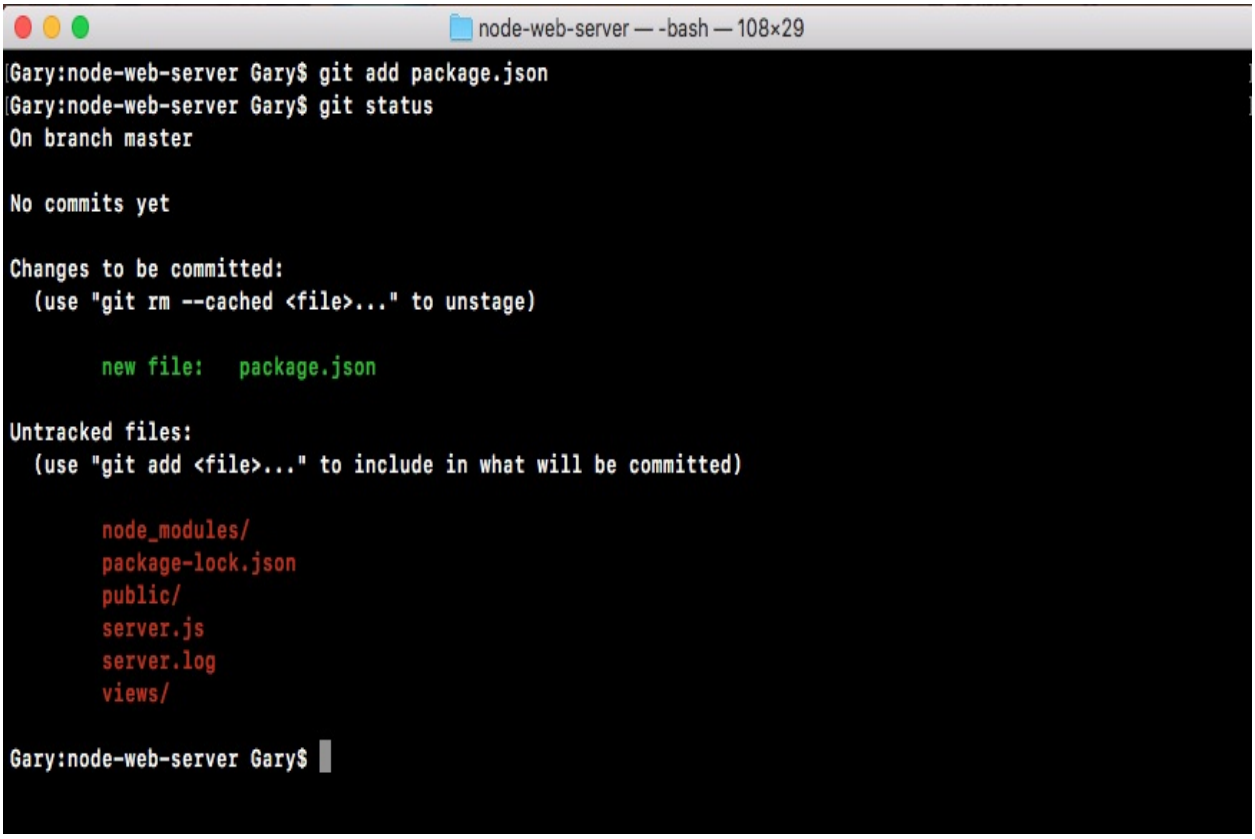
Generated folders are easily generated by running a command. In our case, we can regenerate this entire directory using `npm install`. We're not going to want to add Node modules to our Git repository because its contents differ depending on the version of npm you have installed and depending on the operating system you're using. It's best to leave off Node modules and let every person who uses your repository manually install the modules on the machine they're actually going to be running the app.

Adding untracked files to commit

Now we have these six folders and files listed, so let's go ahead and add the four folders and files we want to keep. To get started, we'll use any `git add` command. The `git add` command lets us tell the Git we want to keep track of a certain file. Let's type the following command:

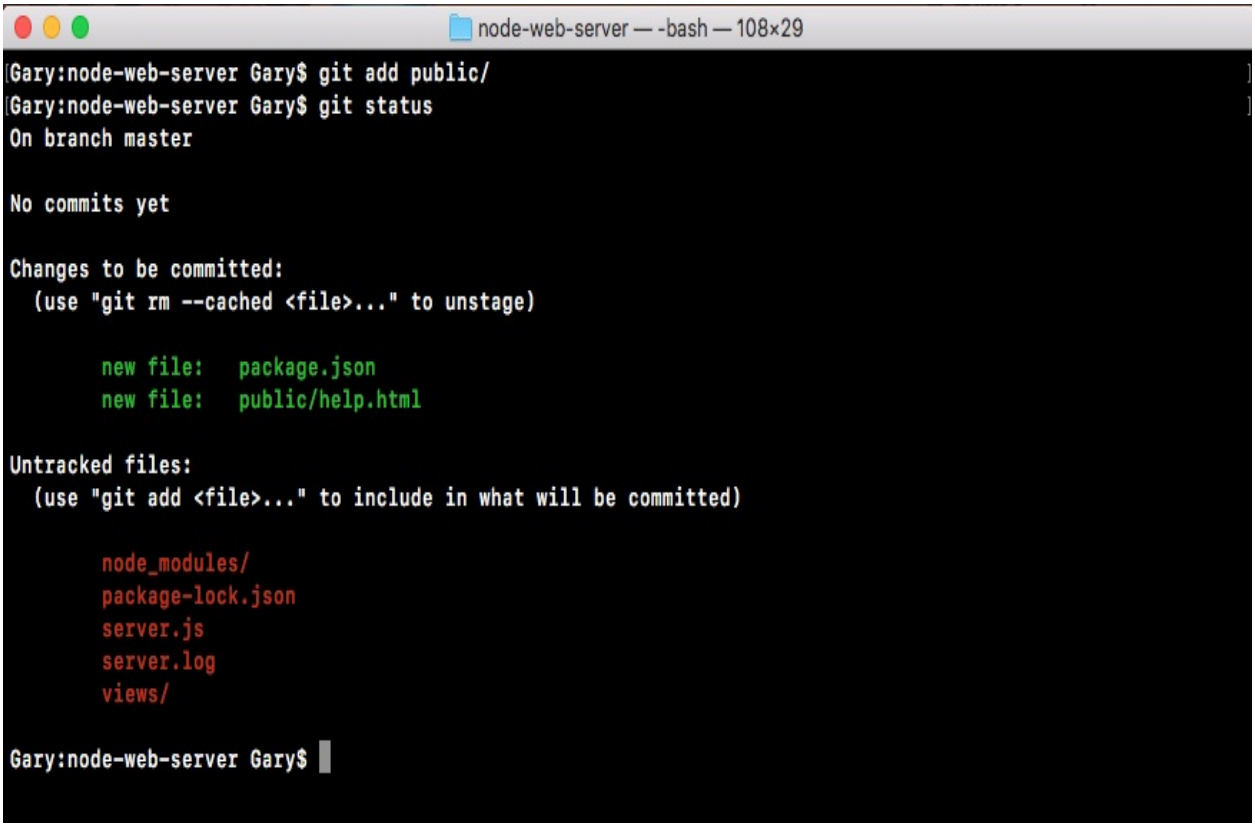
```
git add package.json
```

After we do this, we can run it `git status` again, and this time we get something very different:



Now we have an Initial commit header. This is new, and we have our old Untracked files header. Notice under Untracked files, we don't have `package.json` anymore. That is moved up to the Initial commit header. These are all of the files that are going to be saved, also known as committed, when we make our first commit. Now we can move on adding the 3 others. We'll use a `git add` command

again to tell Git we want to track the public directory. We can run a `git status` command to confirm it was added as expected:

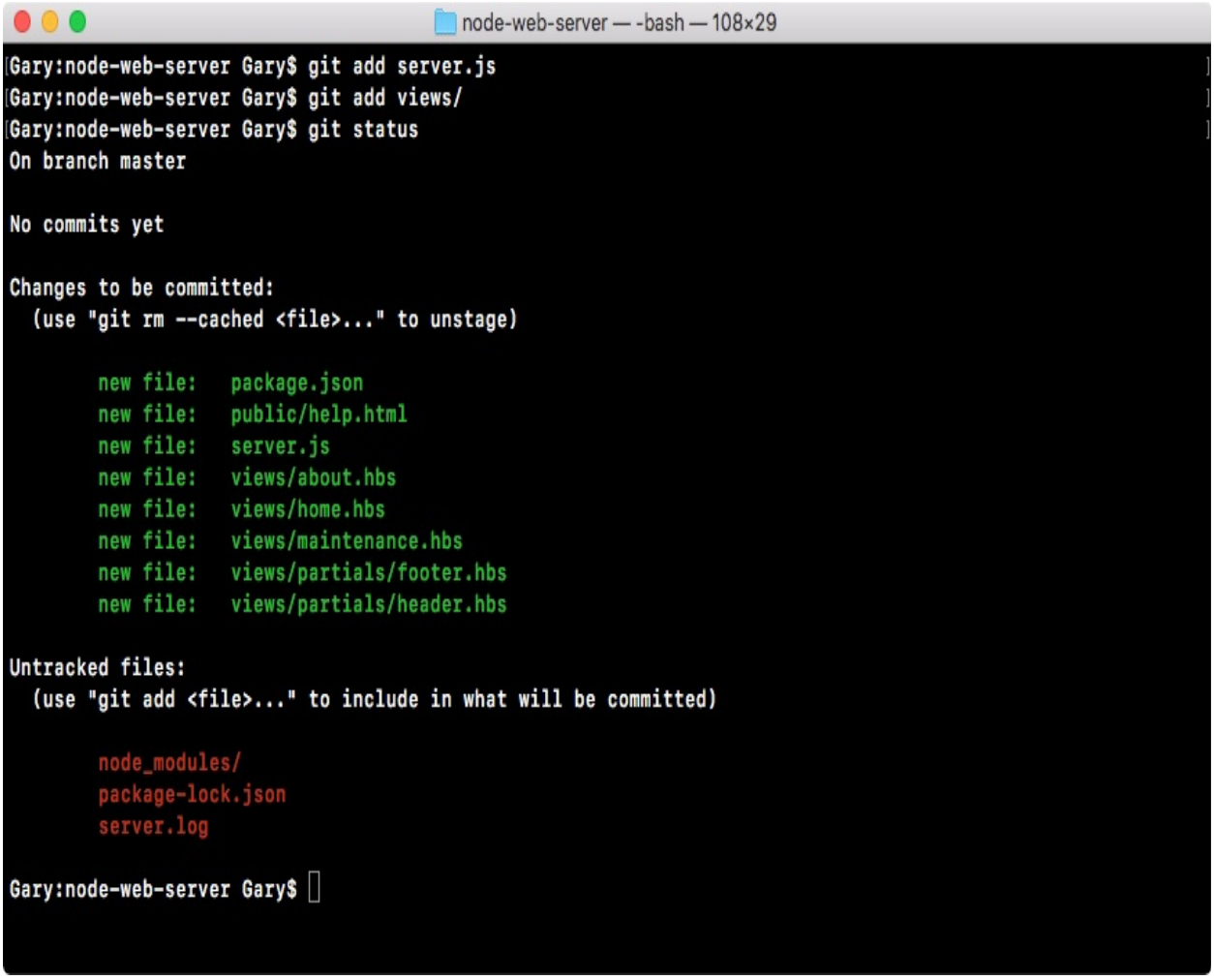


As shown in the preceding screenshot, we can see the public/help.html file is now going to be committed to Git once we run a commit.

Next up we can add `server.js` with `git add server.js`, and we can add the `views` directory using `git add views`, just like this:

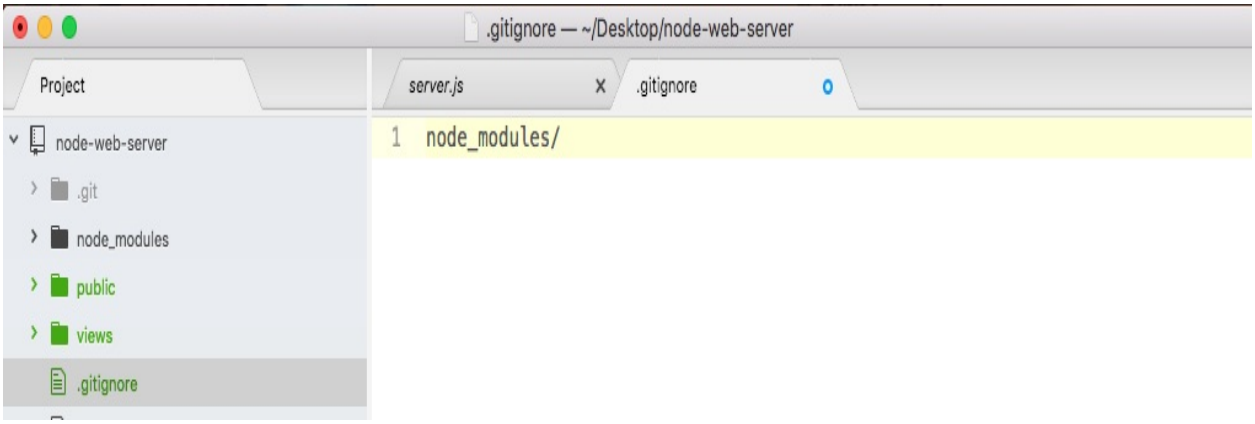
```
git add server.js

git add views/
```

We'll run a `git status` command to confirm:

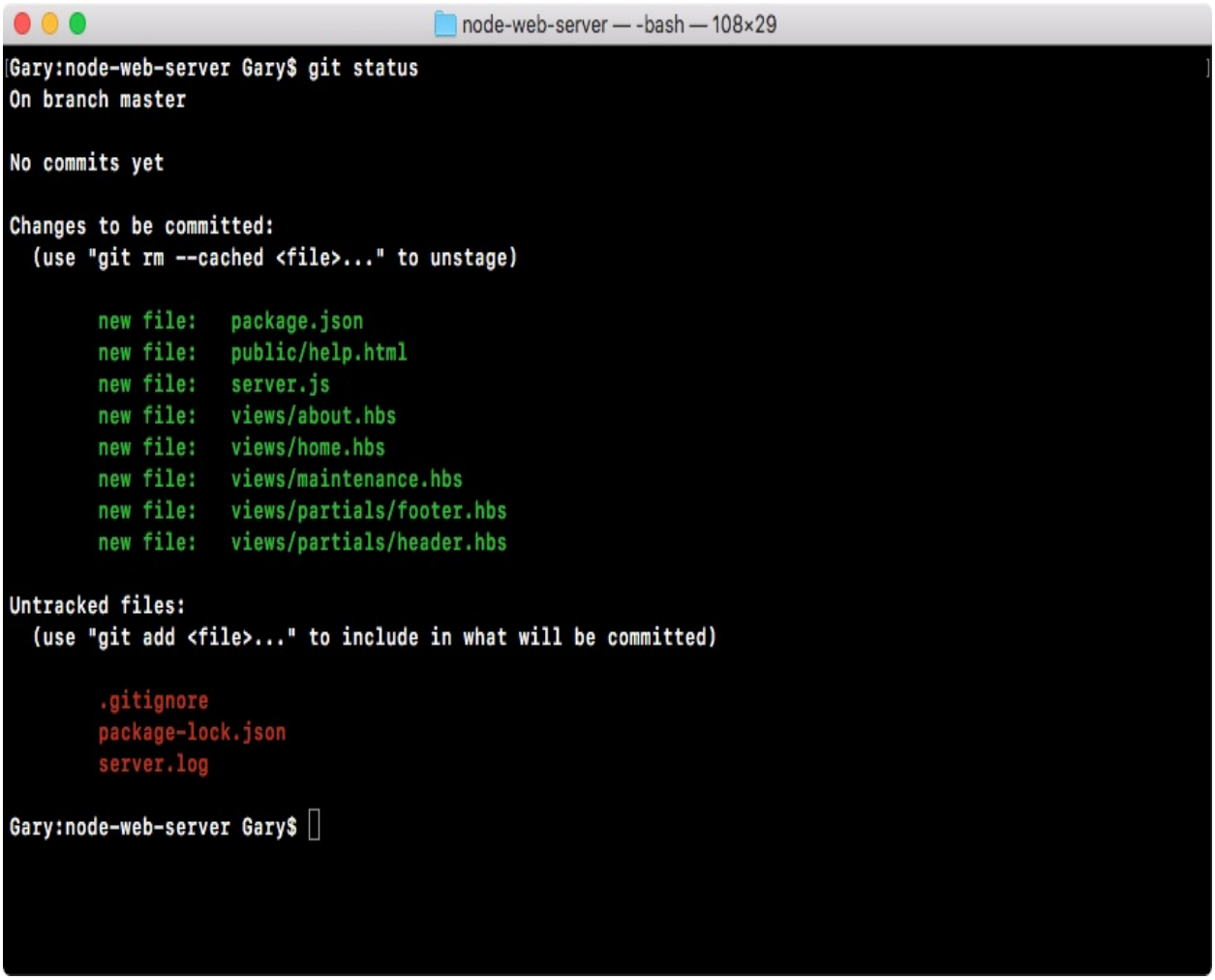


Everything looks good. Now the Untracked files are going to sit around here until we do one of two things—we either add them to the Git repository or ignore them using a custom file that we're going to create inside Atom.

Inside Atom, we'd like to make a new file called `.gitignore`, in our root of our project. The `gitignore` file is going to be part of our Git repository and it tells get which folders and files you want to ignore. In this case we can go ahead and ignore `node_modules`, just like this:



When we save the `gitignore` file and rerun `git status` from the Terminal, we'll now get a really different result:

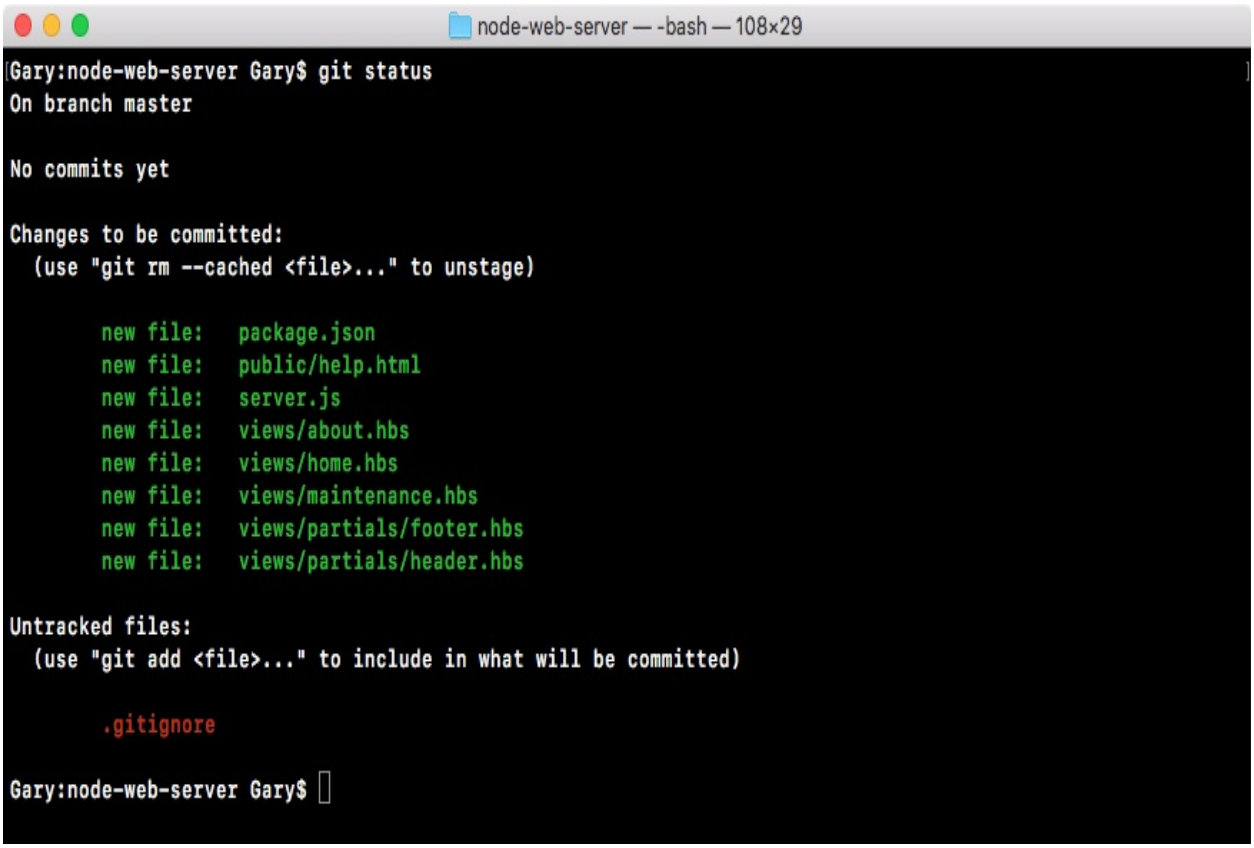


As shown, we can see we have a new untracked file—`.gitignore`—but the `node_modules` directory is nowhere in sight, and that's exactly what we want. We

want to remove this completely, making sure that it never ever gets added to the Git repo. Next up, we can go ahead and ignore that `server.log` file by typing its name, `server.log`:

```
node_modules/
server.log
```

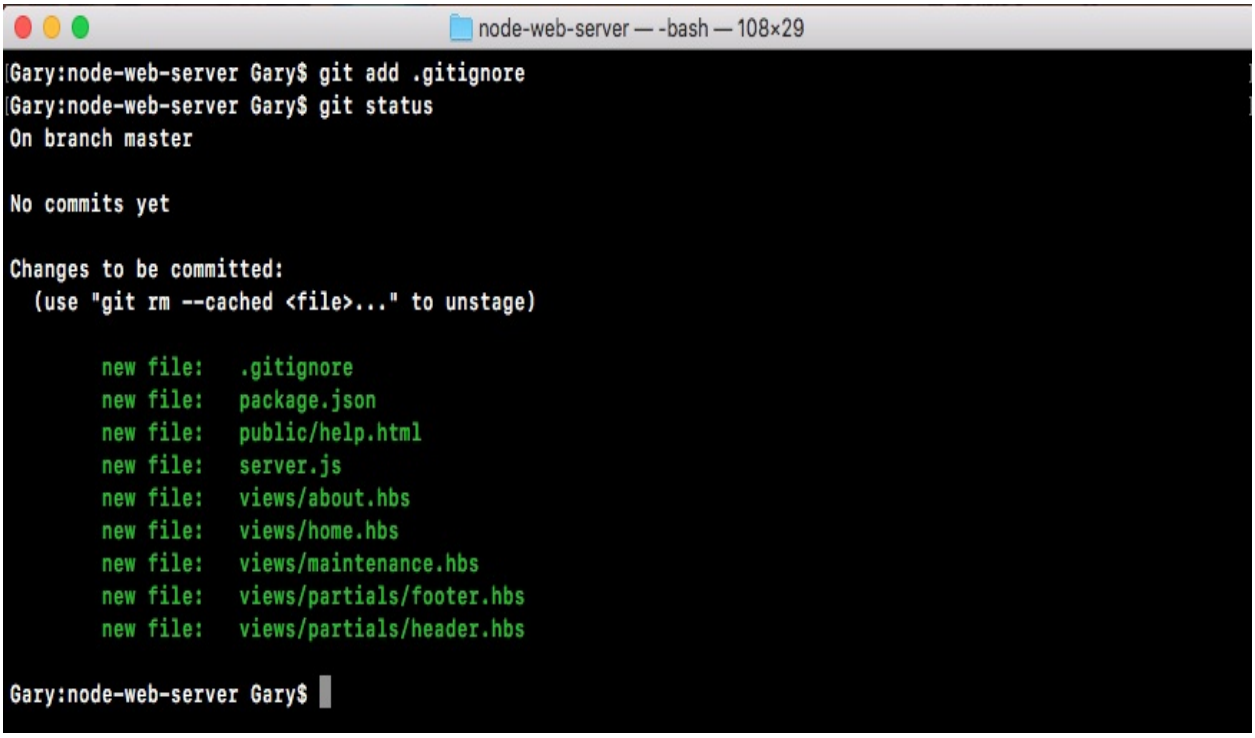
We'll save `gitignore`, run `git status` from the Terminal one more time, and make sure everything looks great:



As shown, we have a `gitignore` file as our only untracked file. The `server.log` file and `node_modules` are nowhere in sight.

Now that we have `gitignore`, we are going to be adding it to Git using `git add .gitignore` and when we run `git status`, we should be able to see that all the files that show up are under the initial commit:

```
git add .gitignore

git status
```



So now it's time to make a commit. A commit really only requires two things. It requires some change in the repository. In this case, we're teaching Git how to track a ton of new files, so we are indeed changing something, and it requires a message. We've already handled the file part of things. We've told Git what we want to save, we just haven't actually saved it yet.

Making a commit

In order to make our first commit and save our first thing into the Git repository, we'll run `git commit` and provide one flag, the `m` flag, which is short message. After that inside quotes, we can specify the message that we want to use for this commit. It's really important to use these messages so when someone's digging through the commit history, the list of all the changes to the project can be seen, which are actually useful. In this case, `Initial commit` is always a good message for your first commit:

```
git commit -m 'Initial commit'
```

I'll go ahead and hit *enter* and as shown in the following screenshot, we see all of the changes that happened to the repo:

We have created a bunch of new files inside of the Git repository. These are all of the files that we told Git we want to keep track of and this is fantastic.

We now have our very first commit, which essentially means that we've saved the project at its current state. If we make a big change to `server.js`, messing stuff up to not be able figure out how to get it back to the way it was, we can always get it back because we made a Git commit. Now we'll explore some more fancy Git things in the later sections. We'll be talking about how to do most of the things you want to do with Git, including deploying to Heroku and pushing to GitHub.

Setting up GitHub and SSH keys

Now that you have a local Git repository, we'll look at how we can take that code and push it up to a third-party service called GitHub. GitHub is going to let us host our Git repositories remotely, so if our machine ever crashes we can get our code back, and it also has great collaboration tools, so we can open-source a project, letting others use our code, or we can keep it private so only people we choose to collaborate with can see the source code.

Now in order to actually communicate between our machine and GitHub, we'll have to create something called an SSH key. SSH keys were designed to securely communicate between two computers. In this case, it will be our machine and the GitHub server. This will let us confirm that GitHub is who they say they are and it will let GitHub confirm that we indeed have access to the code we're trying to alter. This will all be done with SSH keys and we'll create them first, then we'll configure them, and finally we'll push our code up to GitHub.

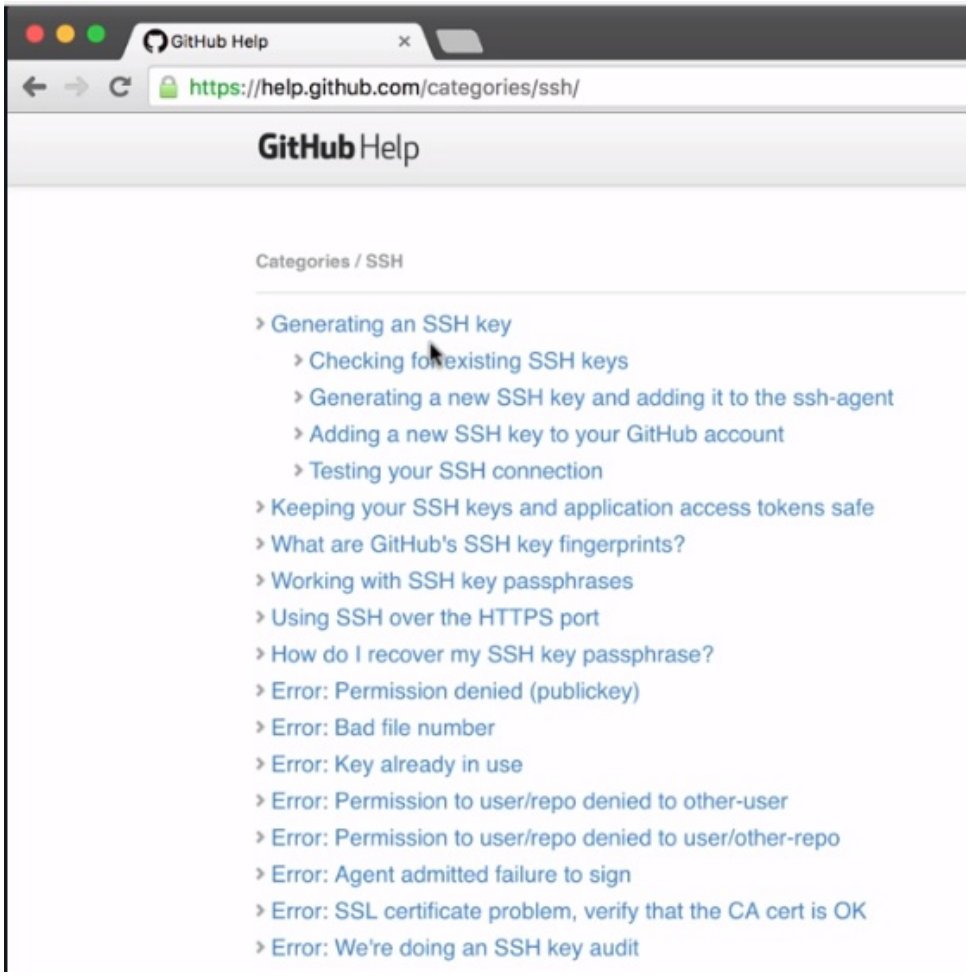
Setting up SSH keys

The process of setting up SSH keys can be a real burden. This is one of those topics where there's really small room for error. If you type any of the commands wrong, things are just not going to work as expected.

Now if you're on Windows, you'll need to do everything in this section from a Git Bash as opposed to the regular Command Prompt because we'll be using some commands that just are not available on Windows. They are, however, available on Linux and macOS. So if you're using either of those operating systems, you can continue using the Terminal you've been using throughout the book.

SSH keys documentations

Before we dive into the commands, I want to show you a quick guide that exists online in case you get stuck or you have any questions. You can Google GitHub SSH keys, and this is going to link you to an article called generating an SSH key: `https://help.github.com/articles/connecting-to-github-with-ssh/`. Once you're here, you'll be able to click on the SSH breadcrumb, and this is going to bring you back to all of their articles on SSH keys:



Out of these articles, the nested four are the ones we'll be focusing on checking if we have a key, generating a new key, adding the key to GitHub, and finally testing that everything worked as expected. If you run into any problems along any of these steps, you can always click on the guide for that step and you can

pick the operating system you're using so you can see the appropriate commands for that OS. Now that you know this exists, let's go ahead and do it together.

Working on commands

The first command we'll run from the Terminal is going to check if we have an existing SSH key. Now if you don't, that's fine. We'll go ahead and create one. If you do or you're not sure you do, you can run the following command to confirm whether or not you have one: `ls` with the `al` flag. This is going to print all the files in a given directory, and the directory where SSH keys are stored by default on your machine is going to be at the user directory, which you can use (`~`) as a shortcut for `/.ssh`:

```
ls -al ~/.ssh
```

When you run the command, you'll see all of the contents inside of that SSH directory:

In this case I've deleted all of my SSH keys so I have nothing inside my directory. I just have paths for the current directory and the previous one. Now that we have this in place and we've confirmed we don't have a key, we can go ahead and generate one. If you do already have a key, a file like `id_rsa`, you can go ahead and skip the process of generating the key.

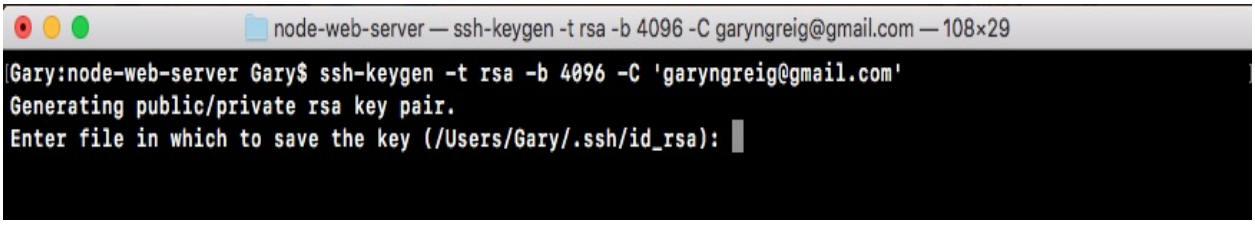
Generating a key

To make a key we'll use the `ssh-keygen` command. Now the `ssh-keygen` takes three arguments. We'll pass in `t` setting it equal to `rsa`. We'll pass in `b` which is for bytes, setting that equal to `4096`. Make sure to match these arguments exactly, and we'll be setting a capital `C` flag which will get set equal to your email:

```
ssh-keygen -t rsa -b 4096 -C 'garyngreig@gmail.com'
```

Now the scope of what's actually happening behind the scenes is not part of this book. SSH keys and setting up security, that could be an entire course in and of itself. We'll be using this command to simplify the entire process.

Now we can go ahead and hit *enter*, which will generate two new files in our `.ssh` folder. When you run this command, you'll get greeted with a few steps. I want you to use the default for all of them:



Here they want to ask you if you want to customize the file name. I do not recommend doing that. You can just hit *enter*:

Next up they ask you for a passphrase, which we'll not use. I'll hit *enter* for no passphrase, then I need to confirm the passphrase, so I'll just hit *enter* again:

```
Enter same passphrase again:
Your identification has been saved in /Users/Gary/.ssh/id_rsa.
Your public key has been saved in /Users/Gary/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:xasuJ5xLVjk69cxyblZx4VZuZcqyl7uyMEKPmAyBSLc garyngreig@gmail.com
The key's randomart image is:
+---[RSA 4096]----+
| . .             |
|.....    .    . +|
|. .E.     o  o *.|
|     .   o .o * o|
|    .   S .  * o |
|     o B O  o o  |
|     .O.= O. . . |
|     o=o.=oo. .  |
|      .=.o. .o.. |
+----[SHA256]-----+
Gary:node-web-server Gary$
```

As shown, we get a little message that our SSH key was properly created and that it was indeed saved in our folder.

With this in place, I can now cycle back through my previous commands running the `ls` command, and what do I get?

```
node-web-server — -bash — 108×29
Gary:node-web-server Gary$ ls -al ~/.ssh
total 16
drwx------   4 Gary  staff   136 Jan 26 19:01 .
drwxr-xr-x+ 32 Gary  staff  1088 Jan 26 18:59 ..
-rw-------   1 Gary  staff  3243 Jan 26 19:01 id_rsa
-rw-r--r--   1 Gary  staff   746 Jan 26 19:01 id_rsa.pub
Gary:node-web-server Gary$
```

We get `id_rsa` and I get the `id_rsa.pub` file. The `id_rsa` file contains the private key. This is the key you should never give to anyone. It lives on your machine and your machine only. The `.pub` file, which is the public file. This one is the one you'll give to third-party services such as GitHub or Heroku, which we'll be doing in the next several sections.

Starting up the SSH agent

Now that our keys are generated, the last thing we need to do is start up the SSH agent and add this key so it knows that it exists. We'll do this by running two commands. These are:

- `eval`
- `ssh-add`

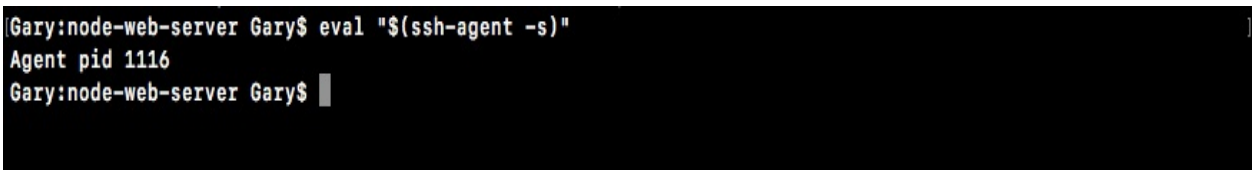
First up we'll run `eval`, and then we'll open some quotes and inside the quotes, we'll use the dollar sign and open and close some parentheses just like this:

```
eval "$()"
```

Inside our parentheses we'll type `ssh-agent` with the `s` flag:

```
eval "$(ssh-agent -s)"
```

This will start up the SSH agent program and it will also print the process ID to confirm it is indeed running, and as shown, we get Agent pid 1116:



The process ID is obviously going to be different for everyone. As long as you get something back like this you are good to go.

Next up we have to tell the SSH agent where this file lives. We'll do that using `ssh-add`. This takes the path to our private key file which we have in the user directory `/.ssh/id_rsa`:

```
ssh-add ~/.ssh/id_rsa
```

When I run this, I should get a message like identity added:

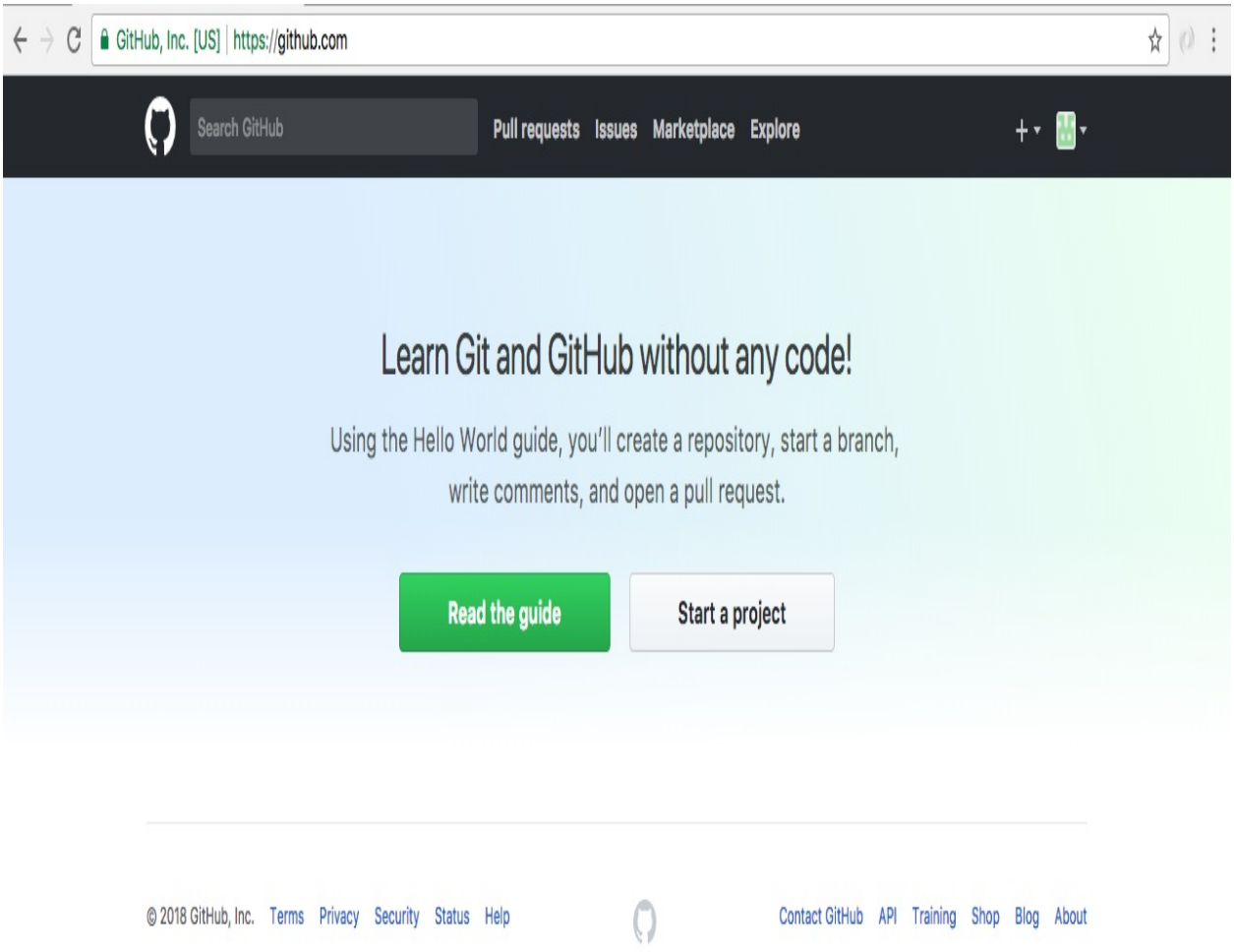
```
Gary:node-web-server Gary$ ssh-add ~/.ssh/id_rsa
Identity added: /Users/Gary/.ssh/id_rsa (/Users/Gary/.ssh/id_rsa)
Gary:node-web-server Gary$
```

This means that the local machine now knows about this public/private key pair and it'll try to use these credentials when it communicates with a third-party service such as GitHub. Now that we have this in place, we are ready to configure GitHub. We'll make an account, set it up, and then we'll come back and test that things are working as expected.

Configuring GitHub

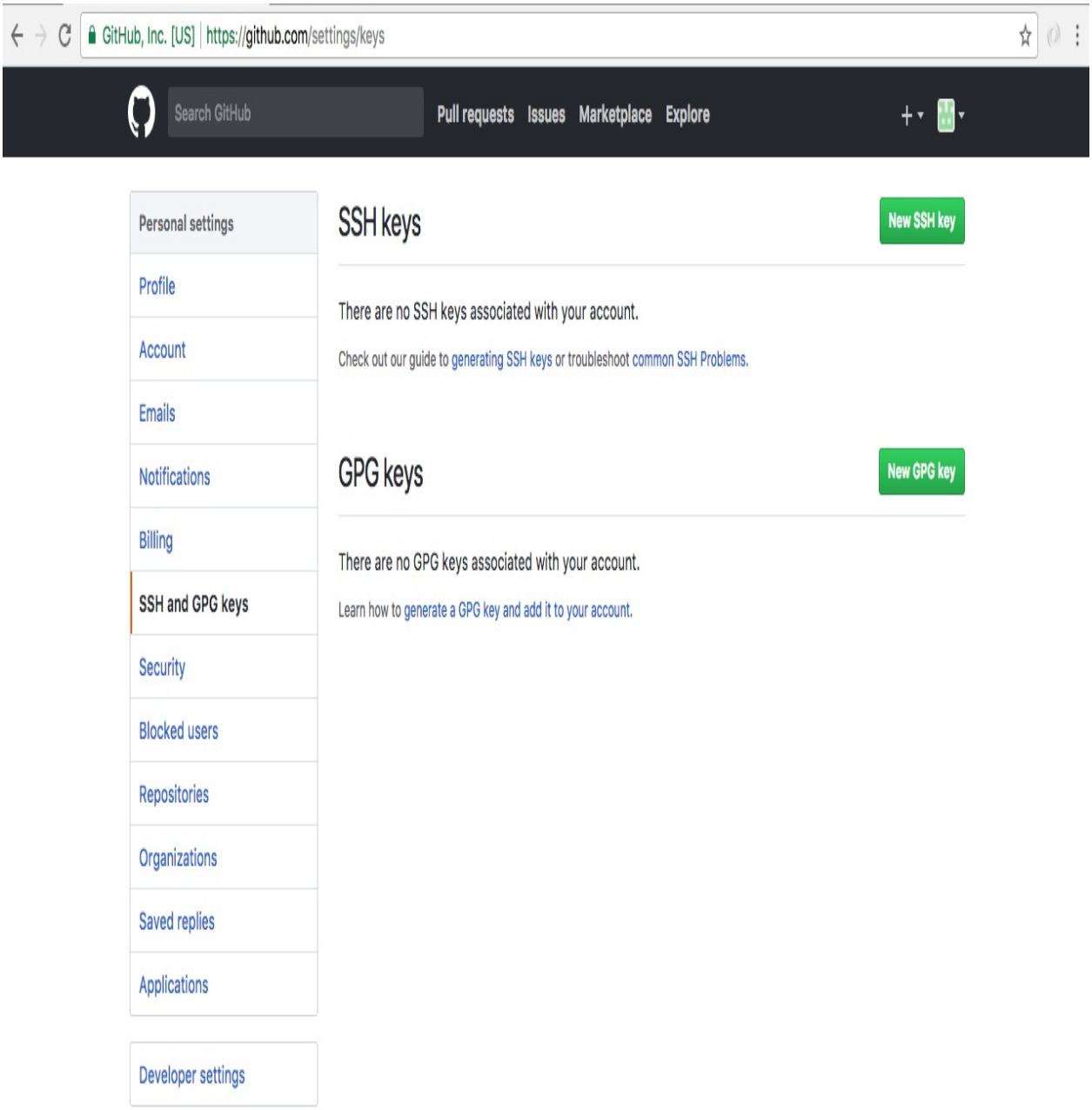
To configure GitHub, follow these steps:

1. First head into the browser and go to `github.com`.
2. Here log into your existing account or create a new one. If you need a new one, sign up for GitHub. If you have an existing one, go ahead and sign into it.
3. Once signed in, you should see the following screen. This is your GitHub dashboard:



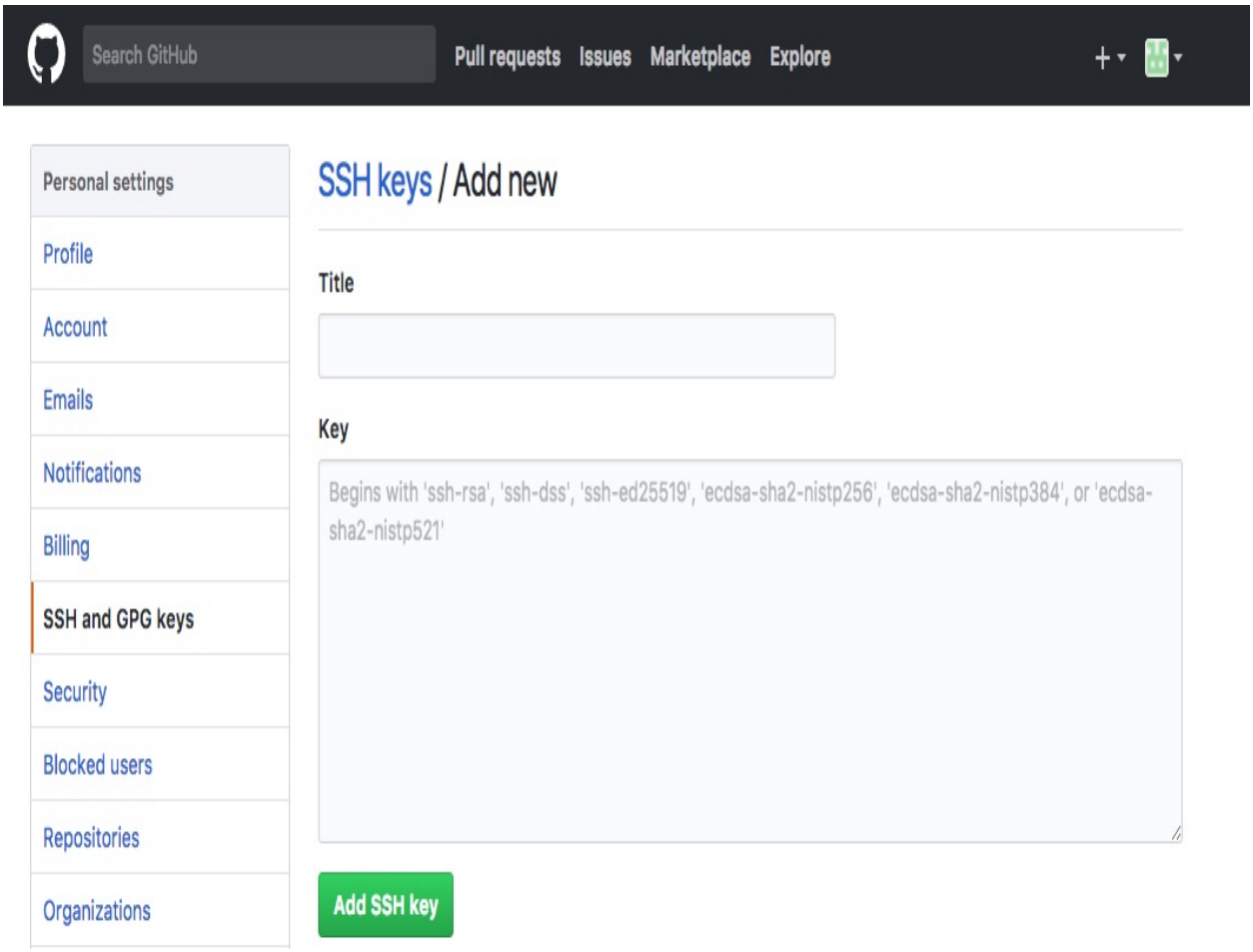
4. From here, navigate to Settings, present at the top-left hand side, by the

profile picture. Go to Settings | SSH and GPG keys | SSH keys:



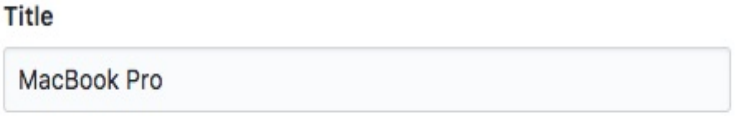
5. From here we can add the public key, letting GitHub know that we want to communicate using SSH.

6. Add the new SSH key:



Here, you need to do two things: give it a name, and add the key.

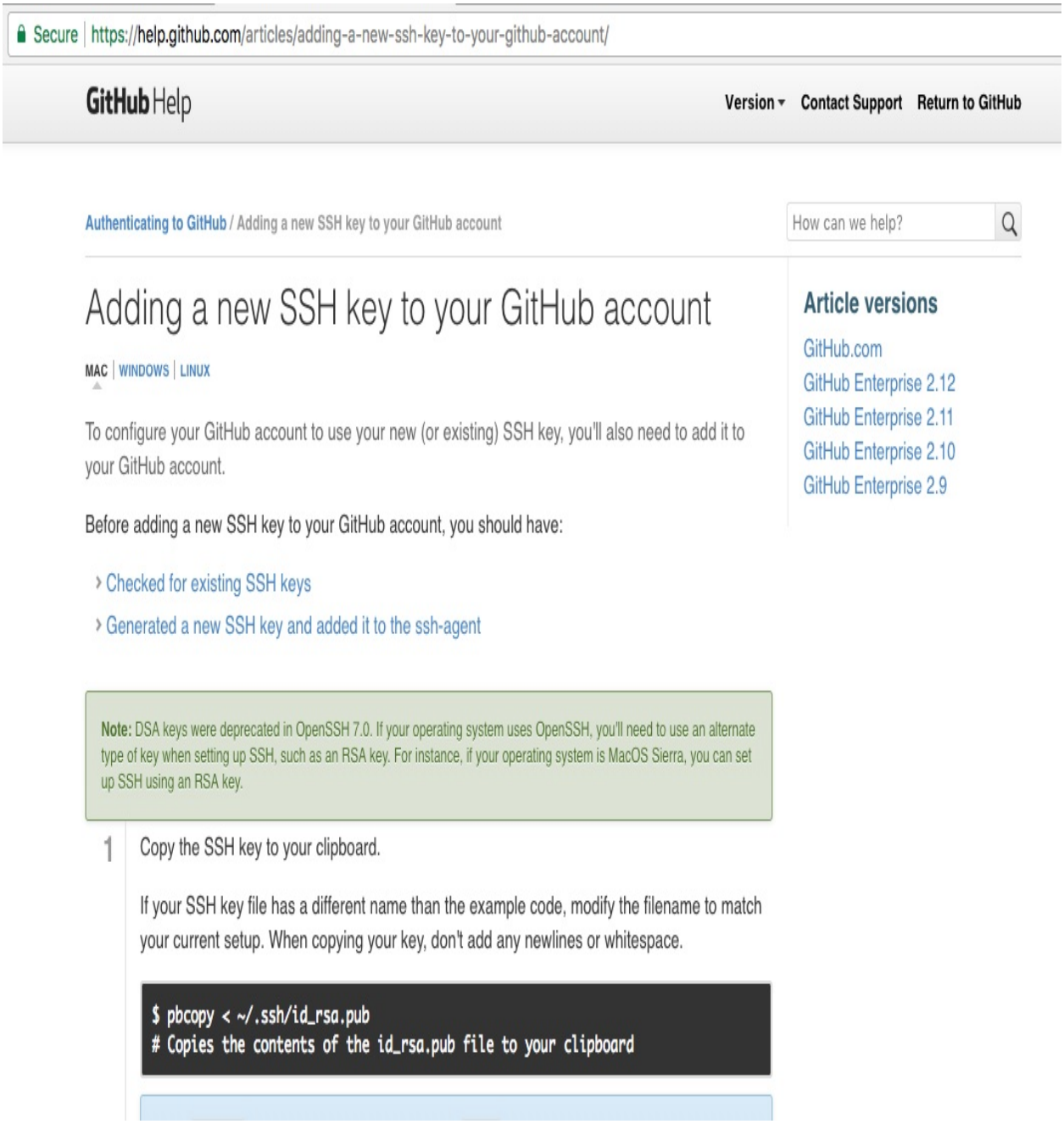
First add the name. The name can be anything you like. For example, I usually use one that uniquely identifies my computer since I have a couple. I'll use `MacBook Pro`, just like this.



Next up, add the key.

To add the key, we need to grab the contents of the `id_rsa.pub` file, we generated in the previous sub-section. That file contains the information that GitHub needs in order to securely communicate between our

machine and their machines. There are different methods to grab the key. In the browser, we have the Adding a new SSH key to your GitHub account article for our reference.



7. This contains a command you can use to copy the contents of that file to your clipboard from right inside the Terminal. Now obviously it is different for the operating systems, macOS, Windows, and Linux, so run the command for your operating system.

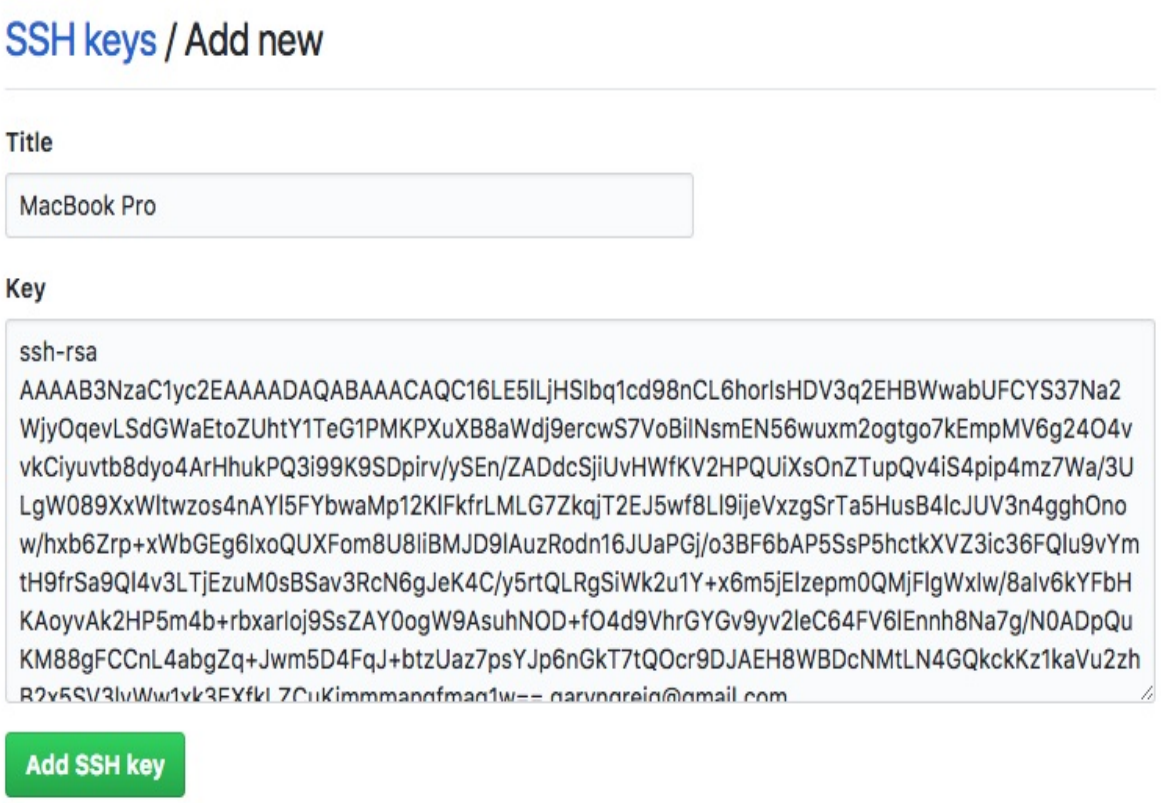
8. Use the `pbcopy` command which is available for macOS.

   Then, move into the Terminal and run it.

```
pbcopy < ~/.ssh/id_rsa.pub
```

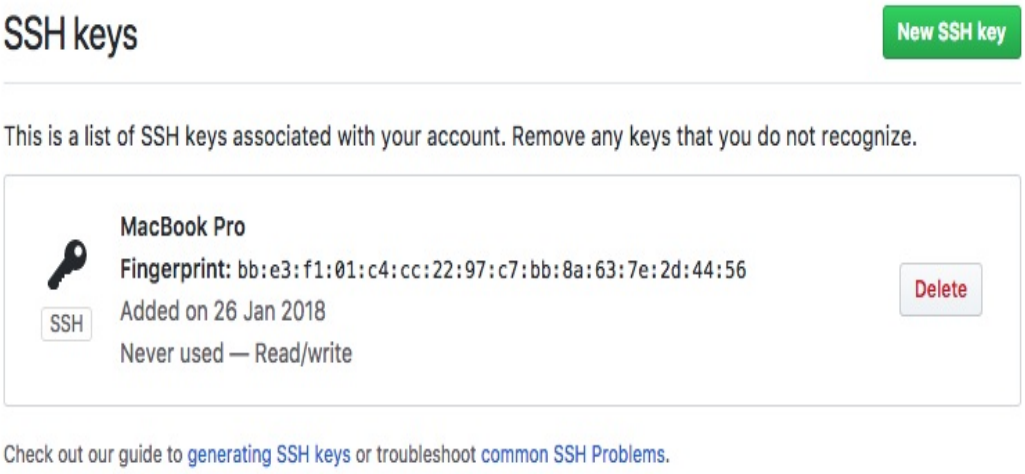
   This copies the contents of the file to the clipboard. You can also open the command up with a regular text editor and copy the contents of the file. We can use any method to copy the file. It doesn't matter how you do it. All that matters is you do.

9. Now move back into GitHub, click on the text area and paste it in.



   The contents of `id_rsa.pub` should start with `ssh-rsa` and it should end with that email you used.

10. Once you're done, go ahead and click on Add SSH key.



Now we can go ahead and test that things are working by running one command from the Terminal. Once again this command can be executed from anywhere on your machine. You don't need to be in your project folder to do this.

Testing the configuration

To test the working of our GitHub configuration, we'll use `ssh`, which tries to make a connection. We'll use the `T` flag, followed by the URL we want to connect to you get at `git@github.com`:

```
ssh -T git@github.com
```

This is going to test our connection. It will make sure that the SSH keys are properly set up and we can securely communicate with GitHub. When I run the command I get a message saying that The authenticity of host 'github.com (192.30.253.113)' can't be established.

```
Gary:node-web-server Gary$ ssh -T git@github.com
The authenticity of host 'github.com (192.30.253.113)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWG17E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)?
```

We know that we want to communicate with `github.com`. We're expecting that communication to happen, so we can go ahead and enter `yes`:

```
Warning: Permanently added 'github.com,192.30.253.113' (RSA) to the list of known hosts.
Hi garygreig! You've successfully authenticated, but GitHub does not provide shell access.
Gary:node-web-server Gary$
```

From here, we get a message from the GitHub servers as shown in the preceding screenshot. If you are seeing this message with your username then you are done. You're ready to create your first repository and push your code up.

Now if you don't see this message, something went wrong along the way. Maybe the SSH key wasn't generated correctly or it's not getting recognized by GitHub.

Next, we'll move into GitHub, go back to the home page, and create a new repository.