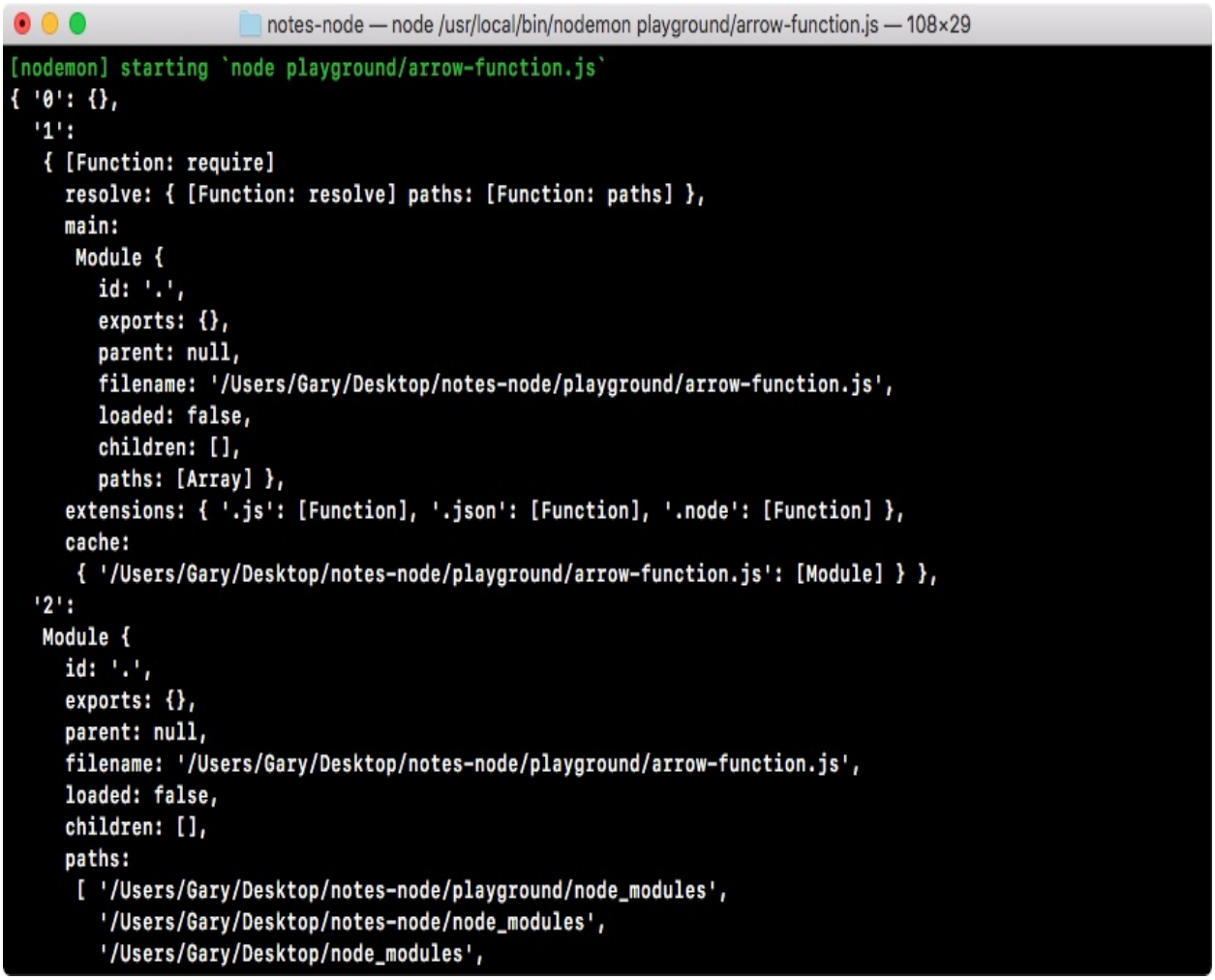
I'll add `console.log(arguments)` inside of my arrow function (`=>`), and I'll switch from calling `sayHiAlt` back to the original method `sayHi`, as shown here:

```
var user = {
  name: 'Andrew',
  sayHi: () => {
    console.log(arguments);
    console.log(`Hi. I'm ${this.name}`);
  },
  sayHiAlt() {
    console.log(arguments);
    console.log(`Hi. I'm ${this.name}`);
  }
};
user.sayHi(1, 2, 3);
```

When I save the file in `arrow-function.js`, we'll get something a lot different from what we had before. What we'll actually get is the global arguments variable, which is the arguments variable for that wrapper function we explored:



In the previous screenshot, we have things like the require function, definition, our modules object, and a couple of string paths to the file and to the current directory. These are obviously not what we're expecting, and that is another thing that you have to be aware of when you're using arrow functions; you're not going to get the `arguments` keyword, you're not going to get the `this` binding (defined in `sayHi` syntax) that you'd expect.

These problems mostly arise when you try to create methods on an object and use arrow functions. I would highly recommend that you switch to `sayHiAlt` syntax which we discussed, in those cases. You get a simplified syntax, but you also get the disk binding and you get your arguments variable as you'd expect.

Summary

In this chapter, we were able to reuse the utility functions that we already made in previous chapters, making the process of filling out a remove note that much easier. Inside `app.js`, we worked on how the `removeNote` function is executed, if it was executed successfully, we print a message; if it didn't, we print a different message.

Next, we were able to successfully fill out the `read` command and we also created a really cool utility function that we can take advantage of in multiple places. This keeps our code DRY and prevents us from having the same code in multiple places inside of our application.

Then we discussed a quick introduction to debugging. Essentially, debugging is a process that lets you stop the program at any point in time and play around with the program as it exists at that moment. That means you can play around with variables that exist, or functions, or anything inside of Node. We learned more about yargs, its configuration, setting up commands, their description, and arguments.

Last, you explored a little bit more about arrow functions, how they work, when to use them, and when not to use them. In general, if you don't need this keyword, or the arguments keyword you can use an arrow function without a problem, and I always prefer using arrow functions over regular functions when I can.

In the next chapter, we will explore asynchronous programming and how we can fetch data from third-party APIs. We'll use both regular functions and arrow functions a lot more, and you'll be able to see firsthand how to choose between one over the other.

Basics of Asynchronous Programming in Node.js

If you've read any article about Node, you'd have probably come across four terms: asynchronous, non-blocking, event-based, and single-threaded. All of those are accurate terms to describe Node; the problem is it usually stops there, and it's really abstract. The topic of asynchronous programming in Node.js has been divided into three chapters. The goal in these upcoming three chapters is to make asynchronous programming super practical by putting all these terms to use in our weather application. That's the project we're going to be building in these chapters.

This chapter is all about the basics of asynchronous programming. We'll look into the basic concepts, terms, and technology related to async programming. We'll look into making requests to Geolocation APIs. We'll need to make asynchronous HTTP requests. Let's dive in, looking at the very basics of async programming in Node.

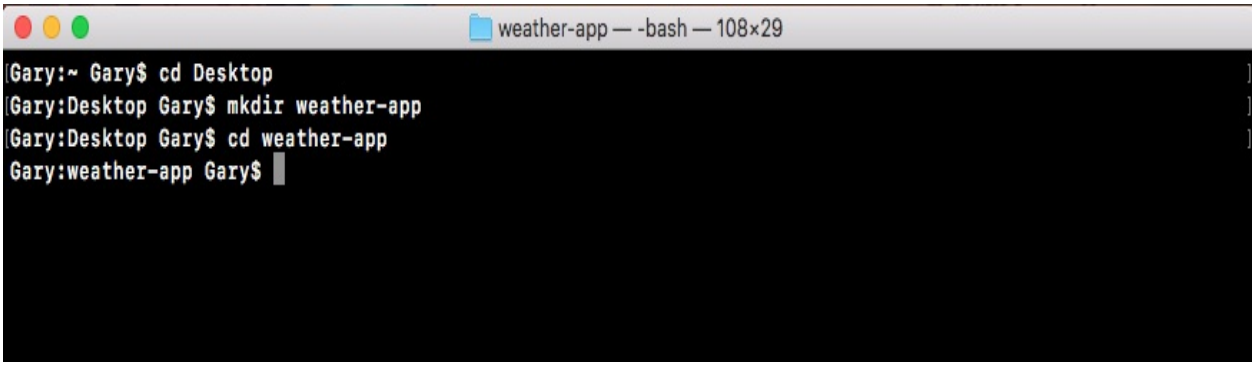
Specifically, we'll look into the following topics:

- The basic concept of asynchronous program
- Call stack and event loop
- Callback functions and APIs
- HTTPS requests

The basic concept of asynchronous program

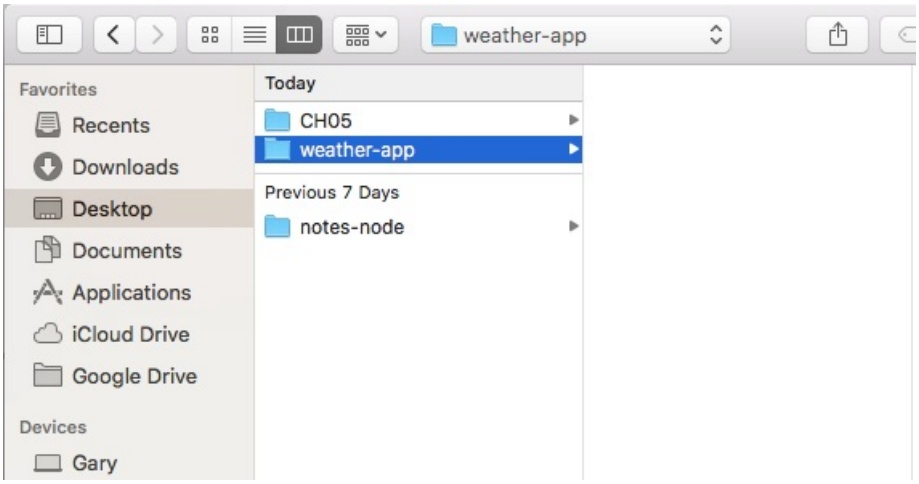
In this section, we're going to create our first asynchronous non-blocking program. This means our app will continue to run while it waits for something else to happen. In this section, we'll look at a basic example; however, in the chapter, we'll be building out a weather app that communicates with third-party APIs, such as the Google API and a weather API. We'll need to use asynchronous code to fetch data from these sources.

For this, all we need to do is make a new folder on the desktop for this chapter. I'll navigate onto my desktop and use `mkdir` to make a new directory, and I'll call this one `weather-app`. All I need to do is navigate into the weather app:



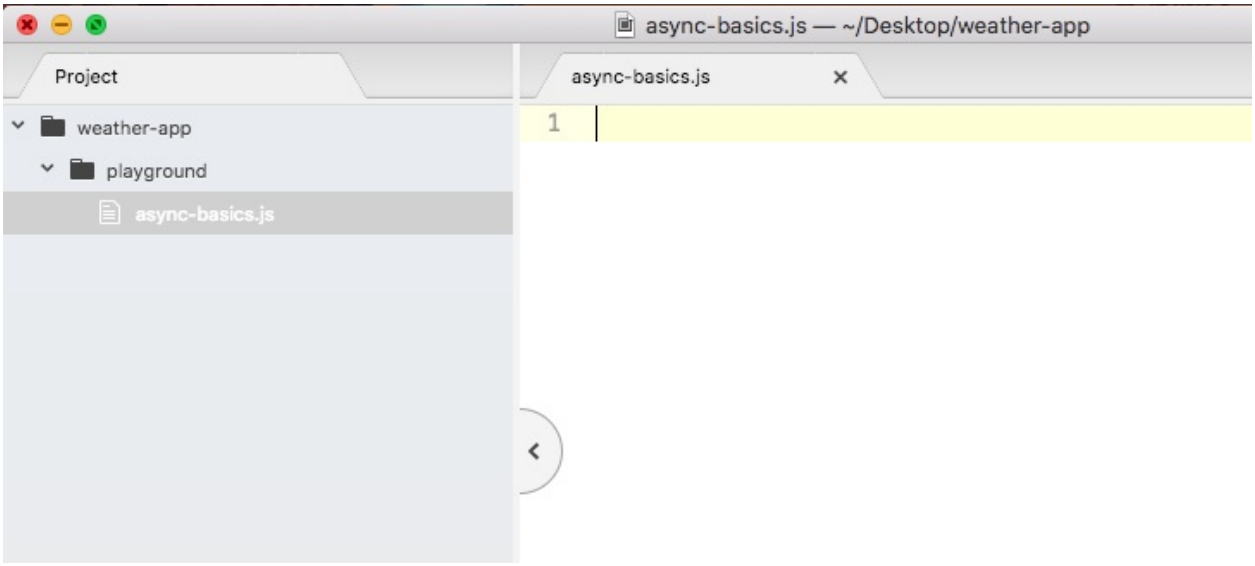
Now, I'll use the `clear` command to clear the Terminal output.

Now, we can open up that new `weather app` directory inside of Atom:



This is the directory we'll use throughout this entire chapter. In this section, we'll not be building out the weather app just yet, we'll just play around with the async features. So inside `weather-app` we'll make the `playground` folder.

This code is not going to be a part of the weather app, but it will be really useful when it comes to creating the weather app in the later sections. Now inside `playground`, we can make the file for this section. We'll name it `async-basics.js` as shown here:



Illustrating the async programming model

To illustrate how the asynchronous programming model works, we'll get started with a simple example using `console.log`. Let's get started by adding a couple of `console.log` statements in a synchronous way. We'll create one `console.log` statement at the beginning of the app that will say `Starting app`, and we will add a second one to the end, and the second one will print `Finishing up`, as shown here:

```
console.log('Starting app');

console.log('Finishing up');
```

Now these are always going to run synchronously. No matter how many times you run the program, `Starting app` is always going to show up before `Finishing up`.

In order to add some asynchronous code, we'll take a look at a function that Node provides called `setTimeout`. The `setTimeout` function is a great method for illustrating the basics of non-blocking programming. It takes two arguments:

- The first one is a function. This will be referred to as callback function, and it will get fired after a certain amount of time.
- The second argument is a number, which tells the number of milliseconds you want to wait. So if you want to wait for one second, you would pass in a thousand milliseconds.

Let's call `setTimeout`, passing in an arrow function (`=>`) as our first argument. This will be callback function. It will get fired right away; that is, it will get fired after the timeout is up, after our two seconds. And then we can set up our second argument which is the delay, `2000` milliseconds, which equals those two seconds:

```
console.log('Starting app');

setTimeout(() => {

}, 2000);
```

Inside the arrow function (`=>`), all we'll do is use a `console.log` statement so that we

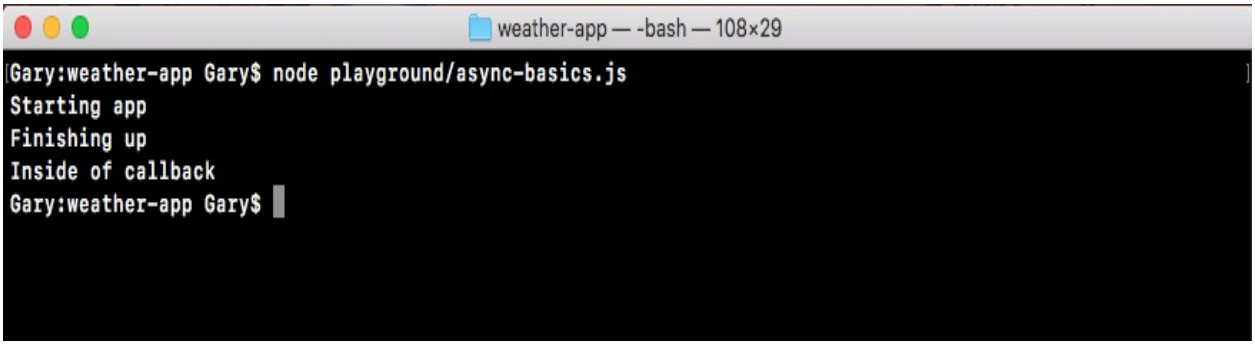
can figure out exactly when our function fires, because the statement will print to the screen. We'll add `console.log` and then inside callback to get the job done, as shown here:

```
setTimeout(() => {
  console.log('Inside of callback');
}, 2000);
```

With this in place, we're actually ready to run our very first async program, and I'll not use `nodemon` to execute it. I'll run this file from the Terminal using the basic Node command; `node playground` and the file inside the `playground` folder which is `async-basic.js`:

```
node playground/async-basics.js
```

Now pay close attention to exactly what happens when we hit e*nter*. We'll see two messages show up right away, then two seconds later our final message, `Inside of callback`, prints to the screen:



The sequence in which these messages are shown is: first we got `Starting app`; almost immediately after this, `Finishing up` prints to the screen and finally (two seconds later), `Inside of callback` was printed as shown in the previous code. Inside the file, this is not the order in which we wrote the code, but it is the order the code executes in.

The `Starting app` statement prints to the screen as we expect. Next, we call `setTimeout`, but we're not actually telling it to wait two seconds. We're registering a callback that will get fired in two seconds. This will be an asynchronous callback, which means that Node can do other things while these two seconds are happening. In this case, the other thing it moves down to the `Finishing up` message. Now since we did register this callback by using `setTimeout`, it will fire at some point in time, and two seconds later we do see `Inside of callback` printing

to the screen.

By using non-blocking I/O, we're able to wait, in this case two seconds, without preventing the rest of the program from executing. If this was blocking I/O, we would have to wait two seconds for this code to fire, then the `Finishing up` message would print to the screen, and obviously that would not be ideal.

Now this is a pretty contrived example, we will not exactly use `setTimeout` in our real-world apps to create unnecessary arbitrary delays, but the principles are the same. For example, when we fetch data from the Google API we'll need to wait about 100 to 200 milliseconds for that data to come back, and we don't want the rest of the program to just be idle, it will continue. We'll register a callback, and that callback will get fired once the data comes back from the Google servers. The same principles applies even though what's actually happening is quite different.

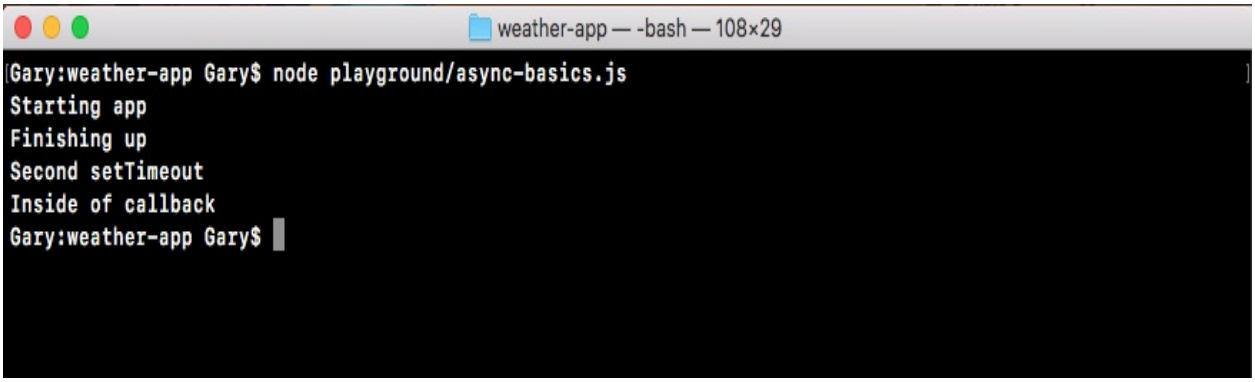
Now, we want to write another `setTimeout` right here. We want to register a `setTimeout` function that prints a message; something like `Second setTimeout works`. This will be inside the callback, and we want to register a delay of `0` milliseconds, no delay at all. Let's fill out the async basics `setTimeout`. I'll call `setTimeout` with my arrow function (`=>`), passing in a delay of `0` milliseconds, as shown in the following code. Inside the arrow function (`=>`), I'll use `console.log` so I can see exactly when this function executes, and I'll use `Second setTimeout` as the text:

```
setTimeout(() => {
  console.log('Second setTimeout');
}, 0);
```

Now that we have this in place, we can run the program from the Terminal, and it's really important to pay attention to the order in which the statements print. Let's run the program:

```
node playground/async-basics.js
```

Right away we get three statements and then at the very end, two seconds later, we get our final statement:



We start with `Starting app`, which makes sense, it's at the top. Then we get `Finishing up`. After `Finishing up` we get `Second setTimeout`, which seems weird, because we clearly told Node we want to run this function after `0` milliseconds, which should run it right away. But in our example, `Second setTimeout` printed after `Finishing up`.

Finally, `Inside of callback` printed to the screen. This behavior is completely expected. This is exactly how Node.js is supposed to operate, and it will become a lot clearer after the next section, where we'll go through this example exactly, showing you what happens behind the scenes. We'll get started with a more basic example showing you how the call stack works, we'll talk all about that in the next section, and then we'll go on to a more complex example that has some asynchronous events attached to it. We'll discuss the reason why `Second setTimeout` comes up after the `Finishing up` message after the next section.

Call stack and event loop

In the last section, we ended up creating our very first asynchronous application, but unfortunately we ended up asking more questions than we got answers. We don't exactly know how async programming works even though we've used it. Our goal for this section is to understand why the program runs the way it does.

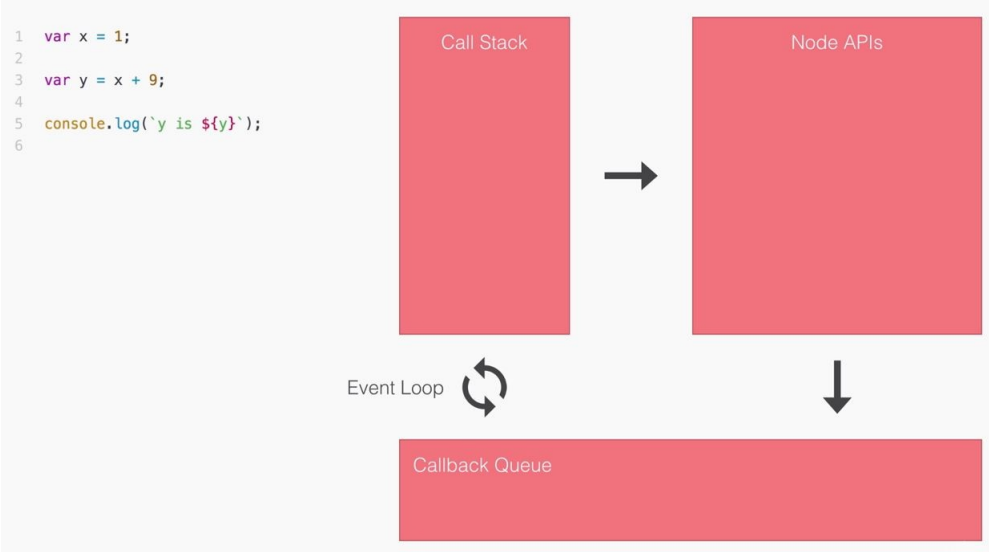
For example, why does the two-second delay in the following code not prevent the rest of the app from running, and why does a `0` second delay cause the function to be executed after `Finishing up` prints to the screen?

```
console.log('Starting app');

setTimeout(() => {
  console.log('Inside of callback');
}, 2000);

setTimeout(() => {
  console.log('Second setTimeout');
}, 0);

console.log('Finishing up');
```

These are all questions we'll answer in this section. This section will take you behind the scenes into what happens in V8 and Node when an async program runs. Now let's dive right into how the async program runs. We'll start with some basic synchronous examples and then move on to figuring out exactly what happens in the async program.

A synchronous program example

The following is example number one. On the left-hand side we have the code, a basic synchronous example, and on the right-hand side we have everything that happens behind the scenes, the Call Stack, our Node APIs, the Callback Queue, and the Event Loop:



Now if you've ever read an article or watched any video lesson on how Node works, you've most likely heard about one or more of these terms. In this section, we'll be exploring how they all fit together to create a real-world, working Node application. Now for our first synchronous example, all we need to worry about is the Call Stack. The Call Stack is part of a V8, and for our synchronous example it's the only thing that's going to run. We're not using any Node APIs and we're not doing any asynchronous programming.

The call stack

The Call Stack is a really simple data structure that keeps track of program execution inside of a V8. It keeps track of the functions currently executing and the statements that are fired. The Call Stack is a really simple data structure that can do two things:

- You can add something on top of it
- You can remove the top item

This means if there's an item at the bottom of the data structure and there's an item above it, you can't remove the bottom item, you have to remove the top item. If there's already two items and you want to add something on to it, it has to go on because that's how the Call Stack works.

Think about it like a can of Pringles or a thing of tennis balls: if there's already an item in there and you drop one in, the item you just dropped will not be the bottom item, it's going to be the top item. Also, you can't remove the bottom tennis ball from a can of tennis balls, you have to remove the one on top first. That's exactly how the Call Stack works.

Running the synchronous program

Now when we start executing the program shown in the following screenshot, the first thing that will happen is Node will run the main function. The main function is the wrapper function we saw over in nodemon (refer to, *Installing the nodemon module* section in `Chapter 2`, *Node Fundamentals Part-1*) that gets wrapped around all of our files when we run them through Node. In this case, by telling V8 to run the main function we are starting the program.

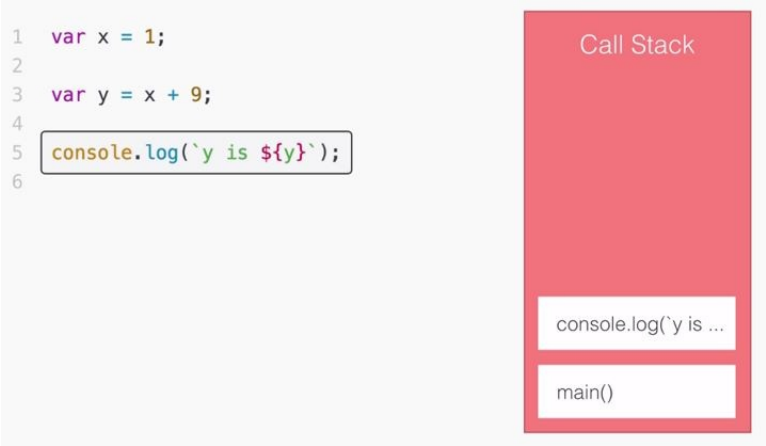
As shown in the following screenshot, the first thing we do in the program is create a variable `x`, setting it equal to `1`, and that's the first statement that's going to run:

Notice it comes in on top of main. Now this statement is going to run, creating the variable. Once it's done, we can remove it from the Call Stack and move on to the next statement, where we make the variable `y`, which gets set equal to `x`, which is `1` plus `9`. That means `y` is going to be equal to `10`:

As shown in the previous screenshot, we do that and move on to the next line. The next line is our `console.log` statement. The `console.log` statement will print `y` is `10` to the screen. We use template strings to inject the `y` variable:

```
console.log(`y is ${y}`);
```

When we run this line it gets popped on to the Call Stack, as shown here:



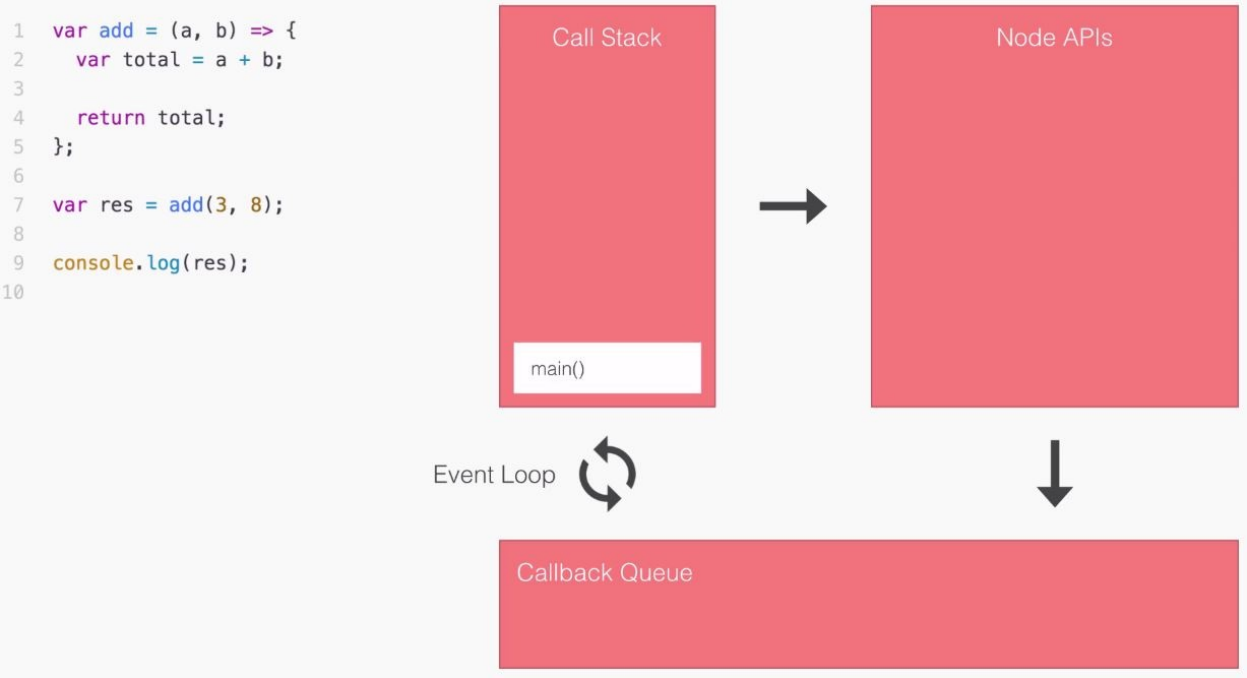
Once the statement is done, it gets removed. At this point, we've executed all the statements inside our program and the program is almost ready to be complete. The main function is still running but since the function ends, it implicitly returns, and when it returns, we remove main from the Call Stack and the program is finished. At this point, our Node process is closed. Now this is a really basic example of using the Call Stack. We went into the main function, and we moved line by line through the program.

A complex synchronous program example

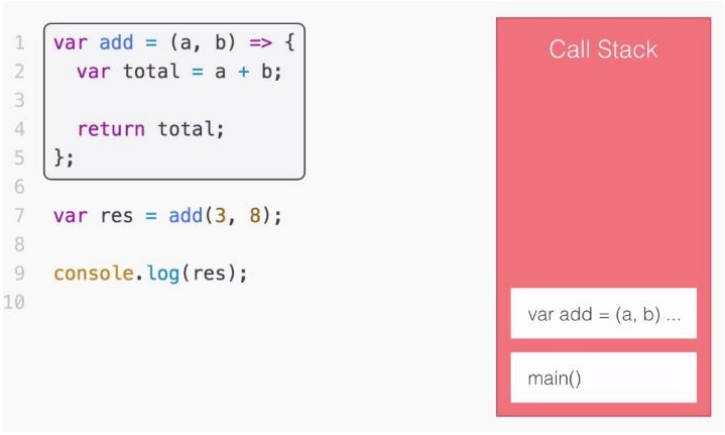
Let's go over a slightly more complex example, our second example. As shown in the following code, we start off by defining an `add` function. The `add` function takes arguments `a` and `b`, adds them together storing that in a variable called `total`, and returns `total`. Next, we add up `3` and `8`, which is `11`, storing it in the `res` variable. Then, we print out the response using the `console.log` statement, as shown here:

```
var add = (a, b) => {
 var total = a + b;

 return total;
};

var res = add(3, 8);

console.log(res);
```

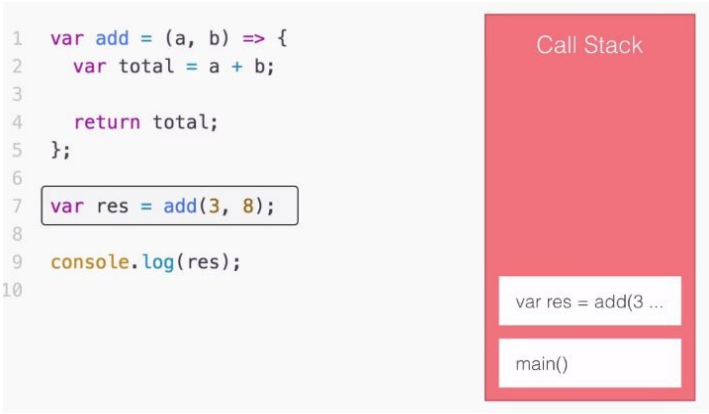
That's it, nothing synchronous is happening. Once again we just need the Call Stack. The first thing that happens is we execute the main function; this starts the program we have here:



Then we run the first statement where we define the `add` variable. We're not actually executing the function, we're simply defining it here:

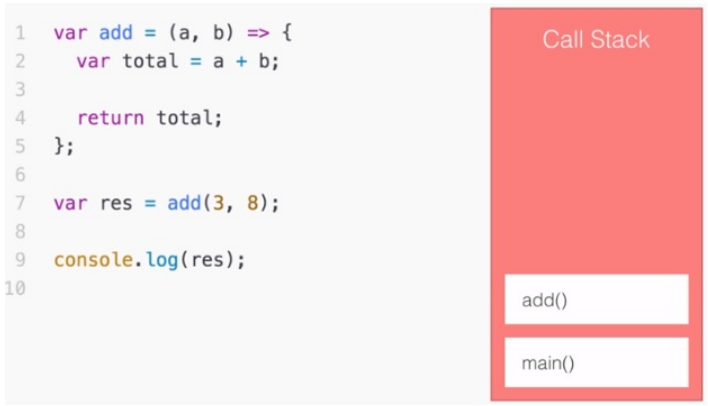


In the preceding image, the `add()` variable gets added on to the Call Stack, and we define `add`. The next line, line `7`, is where we call the `add` variable storing the return value on the response variable:

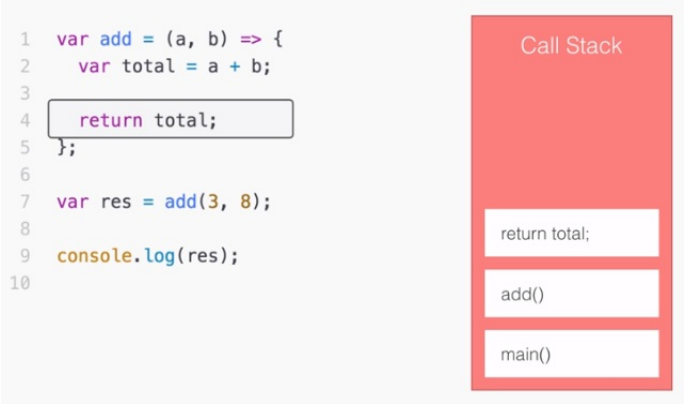


When you call a function, it gets added on top of the Call Stack. When you return from a function, it gets removed from the Call Stack.

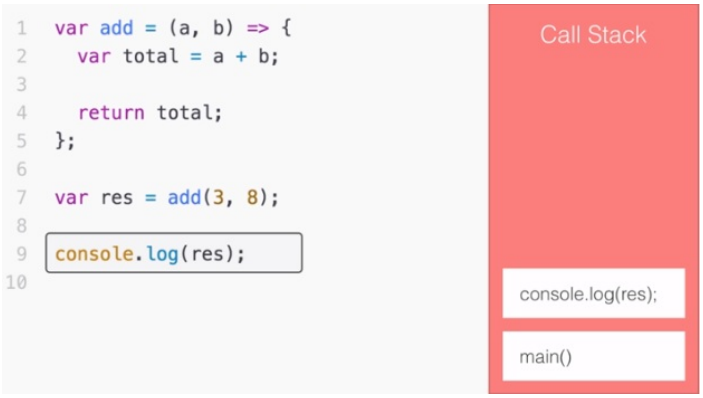
In this example, we'll call a function. So we're going to add `add()` on to the Call Stack, and we'll start executing that function:



As we know, when we add main we start executing main and, when we add `add()` we start executing add. The first line inside `add` sets the `total` variable equal to `a + b`, which would be `11`. We then return from the function using the `return total` statement. That's the next statement, and when this runs, `add` gets removed:



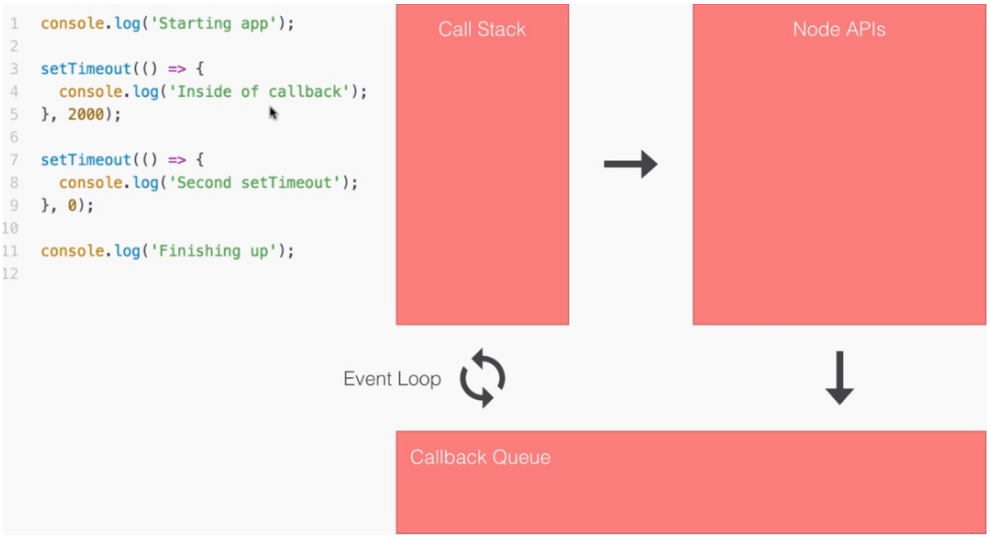
So when `return total` finishes, `add()` gets removed, then we move on to the final line in the program, our `console.log` statement, where we print `11` to the screen:



The `console.log` statement will run, print `11` to the screen and finish the execution, and now we're at the end of the main function, which gets removed from the stack when we implicitly return. This is the second example of a program running through the V8 Call Stack.

An async program example

So far we haven't used Node APIs, the Callback Queue, or the Event Loop. The next example will use all four (Call Stack, the Node APIs, the Callback Queue, and the Event Loop). As shown on the left-hand side of the following screenshot, we have our async example, exactly the same as we wrote it in the last section:

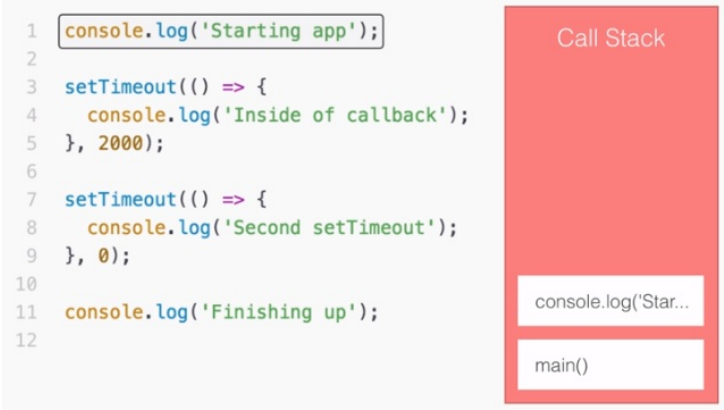


In this example, we will be using the Call Stack, the Node APIs, the Callback Queue, and the Event Loop. All four of these are going to come into play for our asynchronous program. Now things are going to start off as you might expect. The first thing that happens is we run the main function by adding it on to the Call Stack. This tells a V8 to kick off the code we have on the left side in the previous screenshot, shown here again:

```
console.log('Starting app');

setTimeout(() => {
  console.log('Inside of callback');
}, 2000);

setTimeout(() => {
  console.log('Second setTimeout');
}, 0);

console.log('Finishing up');
```

The first statement in this code is really simple, a `console.log` statement that prints

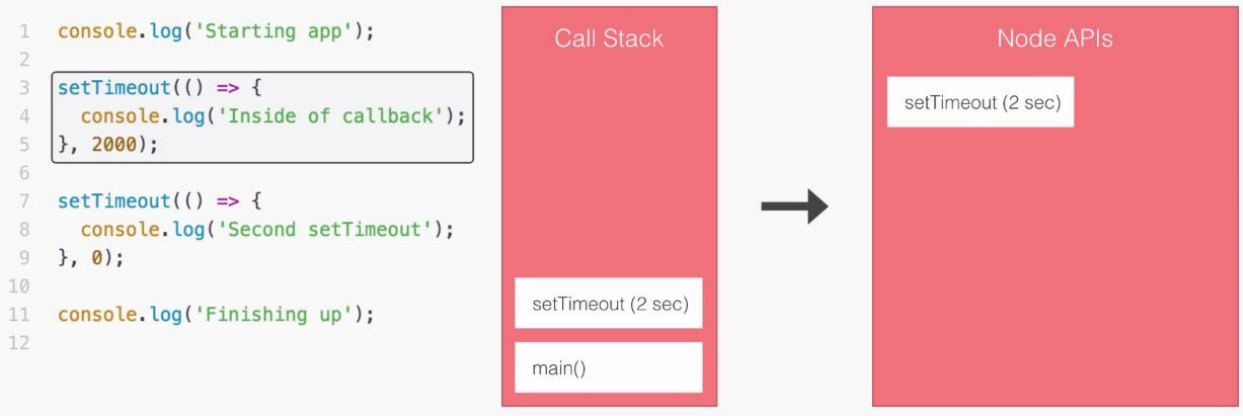
`Starting app` to the screen:



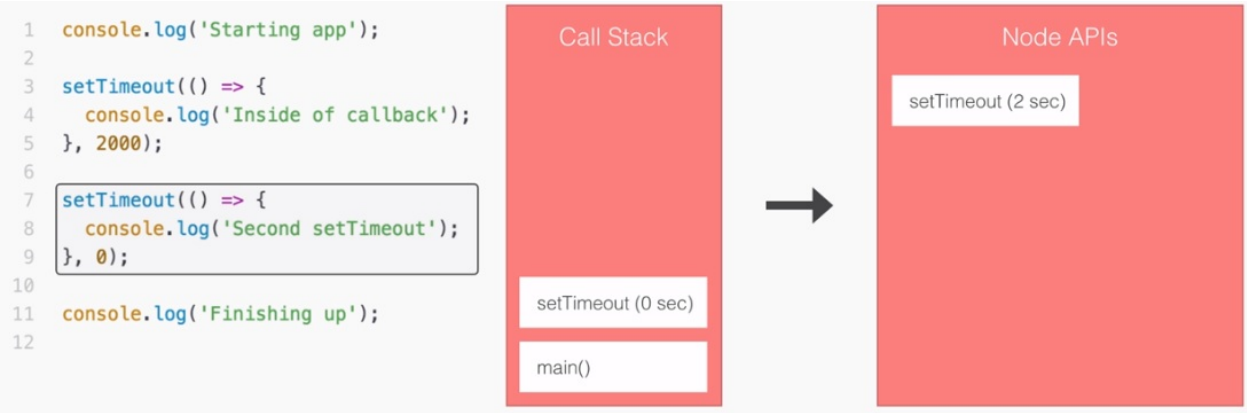
This statement runs right away and we move on to the second statement. The second statement is where things start to get interesting, this is a call to `setTimeout`, which is indeed a Node API. It's not available inside a V8, it's something that Node gives us access to:

The Node API in async programming

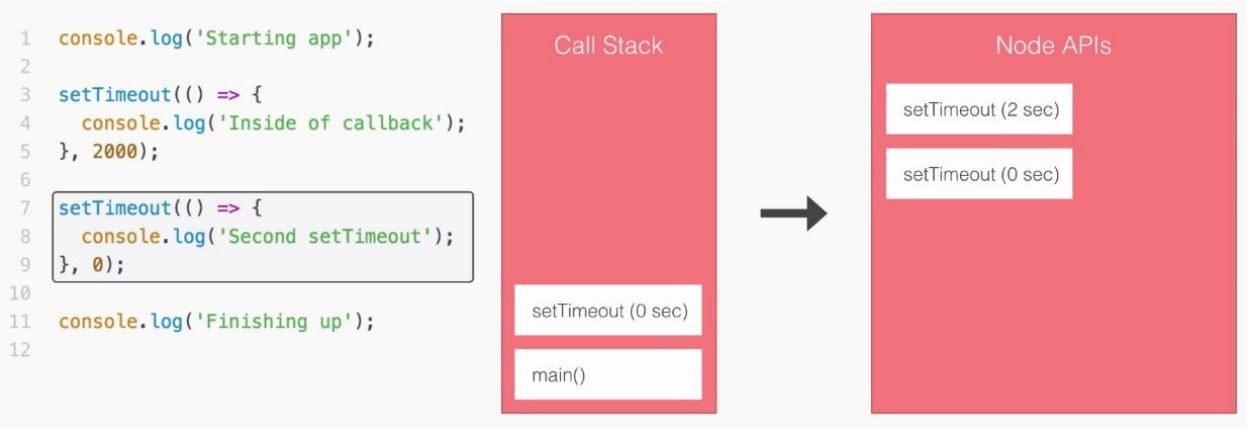
When we call the `setTimeout (2 sec)` function, we're actually registering the event callback pair in the Node APIs. The event is simply to wait two seconds, and the callback is the function we provided, the first argument. When we call `setTimeout`, it gets registered right in the Node APIs as shown here:



Now this statement will finish up, the Call Stack will move on, and the `setTimeout` will start counting down. Just because the `setTimeout` is counting down, it doesn't mean the Call Stack can't continue to do its job. The Call Stack can only run one thing at a time, but we can have events waiting to get processed even when the Call Stack is executing. Now the next statement that runs is the other call to `setTimeout`:



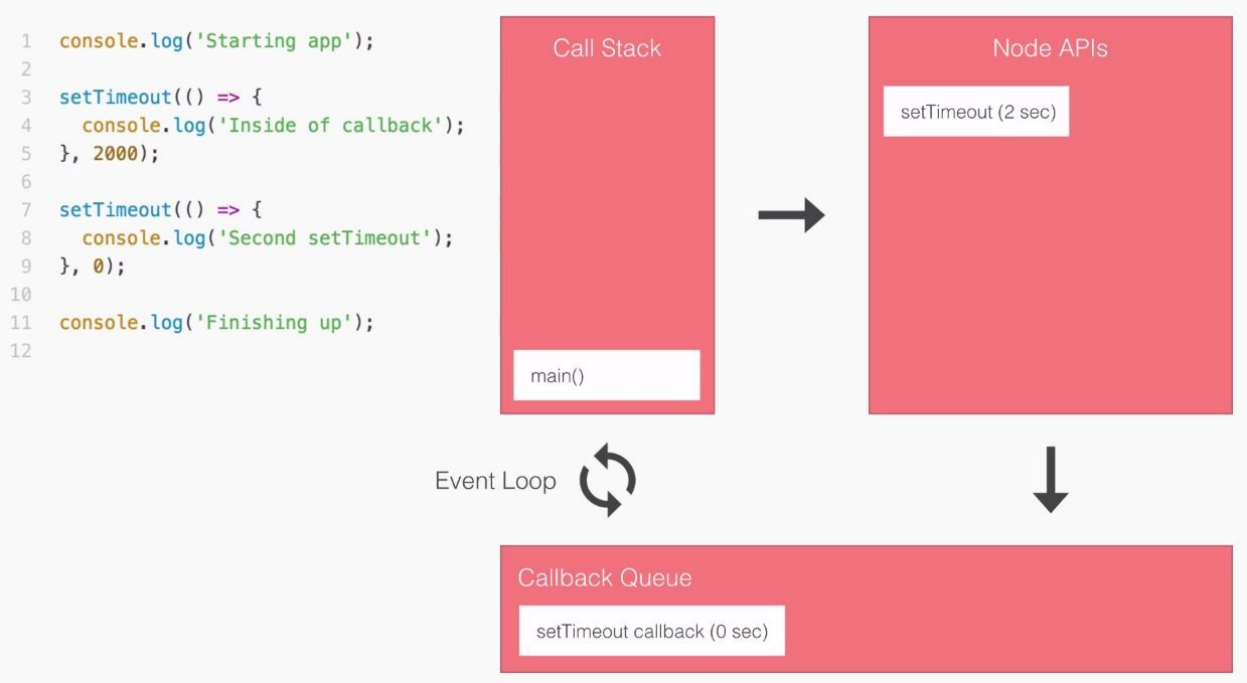
In this, we register a `setTimeout` callback function with a delay of `0` milliseconds, and the exact same thing happens. It's a Node API and it's going to get registered as shown in the following screenshot. This essentially says that after zero seconds, you can execute this callback:



The `setTimeout (0 sec)` statement gets registered and the Call Stack removes that statement.

The callback queue in async programming

At this point let's assume that `setTimeout`, the one that has a zero second delay, finishes. When it finishes, it's not going to get executed right away; it's going to take that callback and move it down into the Callback Queue, as shown here:



The Callback Queue is all the callback functions that are ready to get fired. In the previous screenshot, we move the function from Node API into the Callback Queue. Now the Callback Queue is where our callback functions will wait; they need to wait for the Call Stack to be empty.

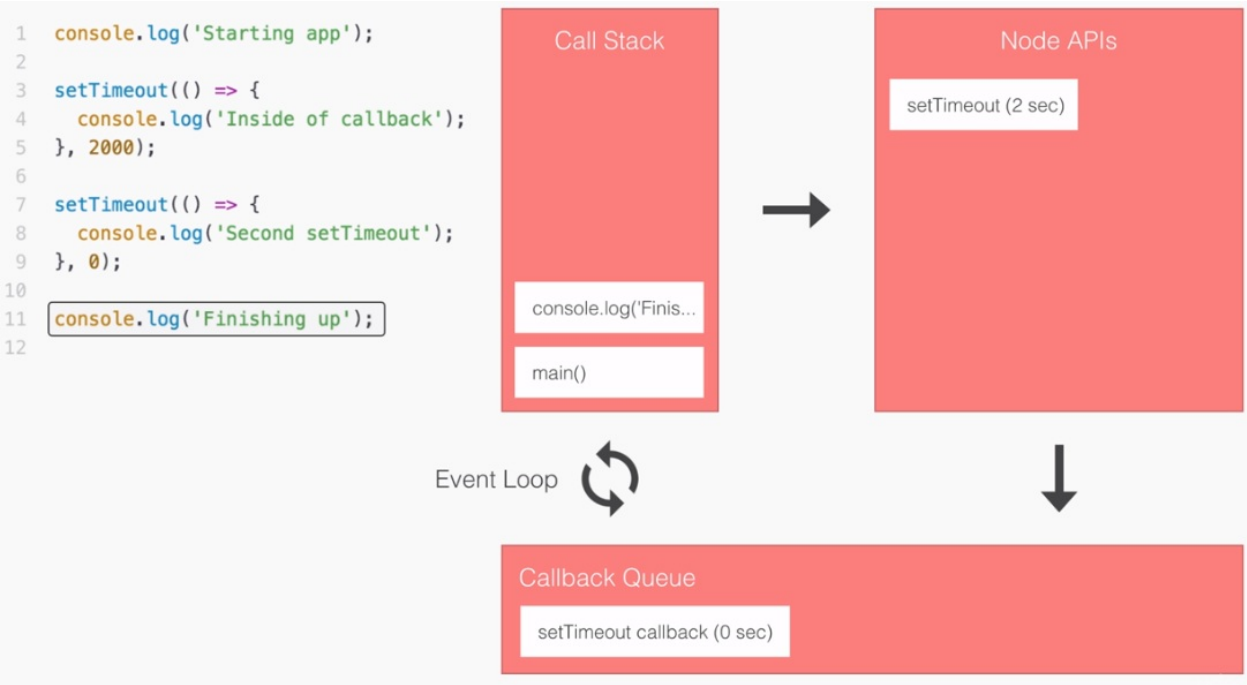
When the Call Stack is empty we can run the first function. There's another function after it. We'll have to wait for that first function to run before the second one does, and this is where the Event Loop comes into play.

The event loop

The Event Loop takes a look at the Call Stack. If the Call Stack is not empty, it doesn't do anything because it can't, there is nothing it can do you can only run one thing at a time. If the Call Stack is empty, the Event Loop says great let's see if there's anything to run. In our case, there is a callback function, but because we don't have an empty Call Stack, the Event Loop can't run it. So let's move on with the example.

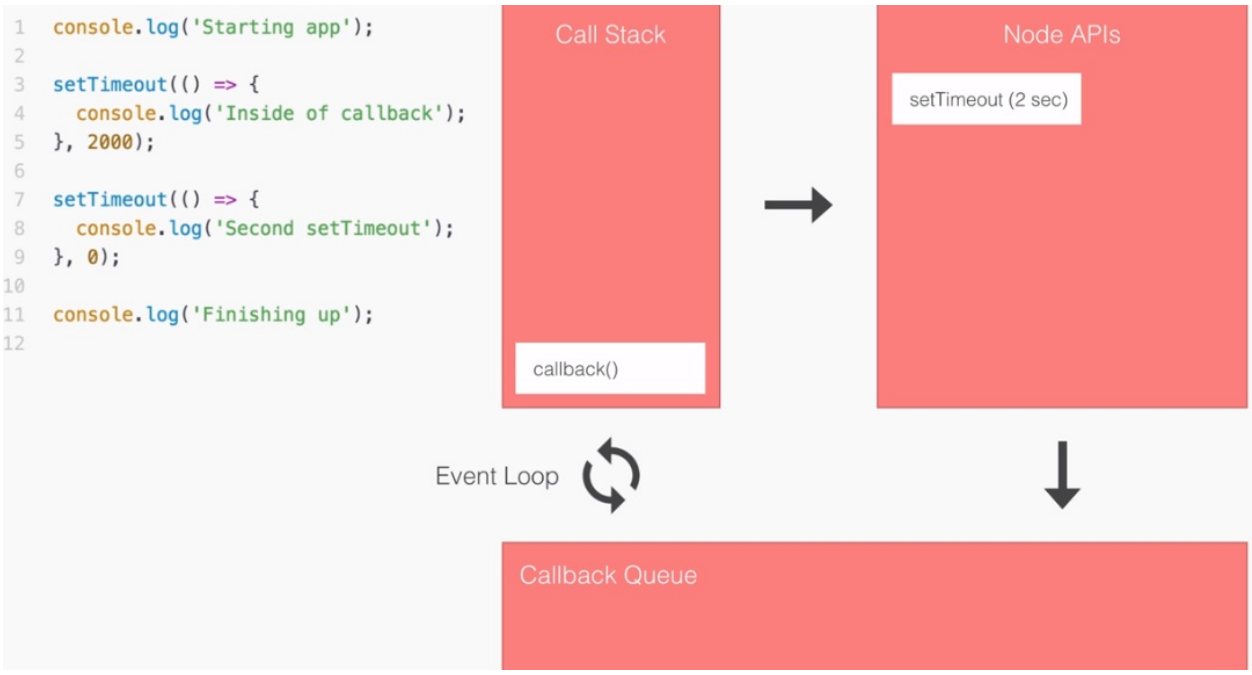
Running the async code

The next thing that happens in our program is we run our `console.log` statement, which prints `Finishing up` to the screen. This is the second message that shows up in the Terminal:



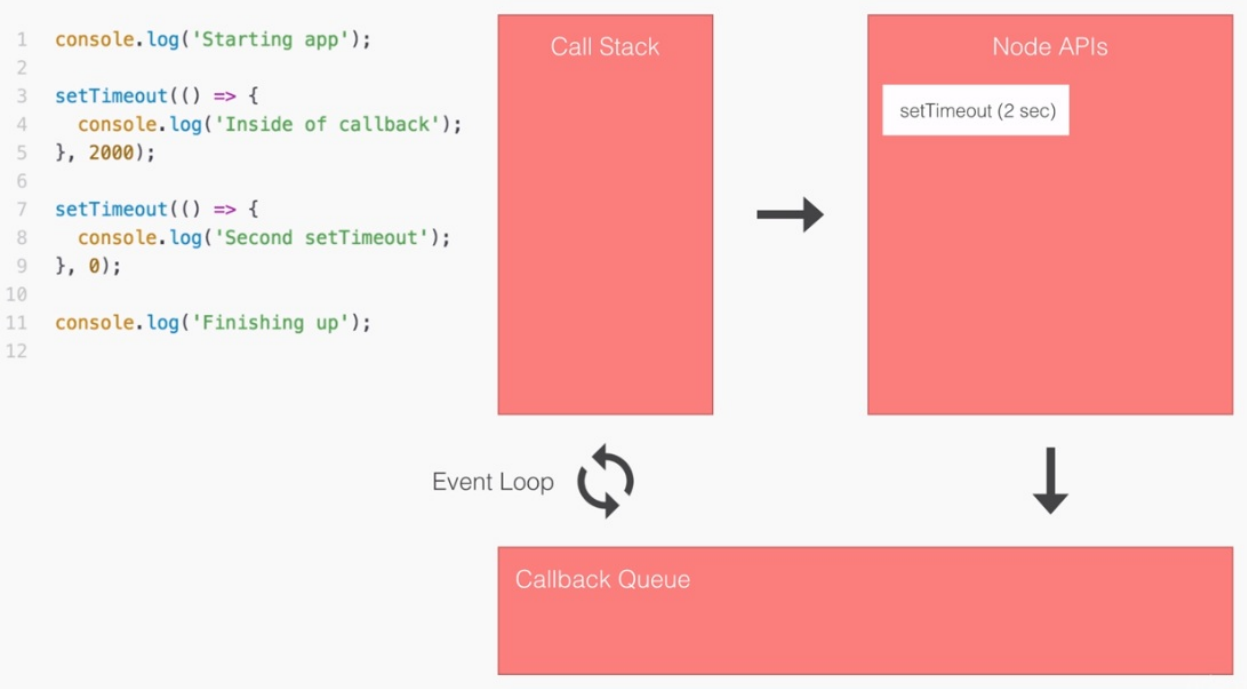
This statement runs, our main function is complete, and it gets removed from the Call Stack.

At this point, the Event Loop says hey I see that we have nothing in the call stack and we do have something in the Callback Queue, so let's run that callback function. It will take the callback and move it into the Call Stack; this means the function is executing:

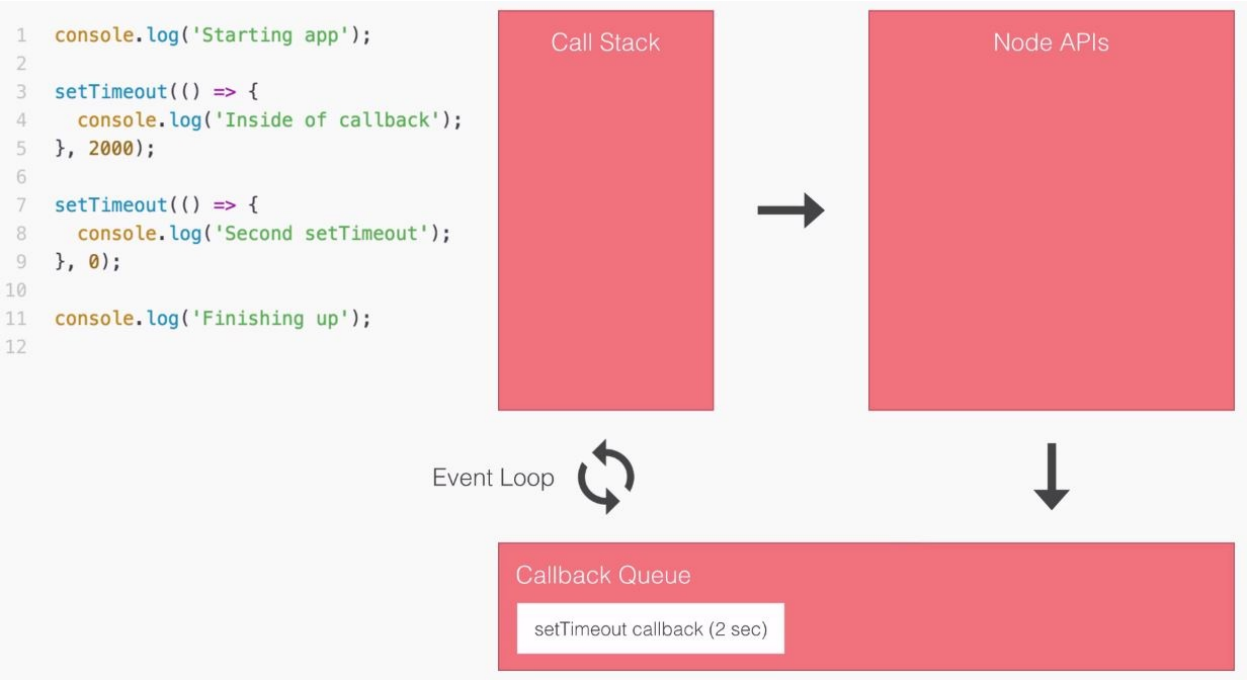


It will run the first line which is sitting on line `8`, `console.log`, printing `Second setTimeout` to the screen. This is why `Second setTimeout` shows up after `Finishing up` in our previous section examples, because we can't run our callback until the Call Stack is complete. Since `Finishing up` is part of the main function, it will always run before `Second setTimeout`.

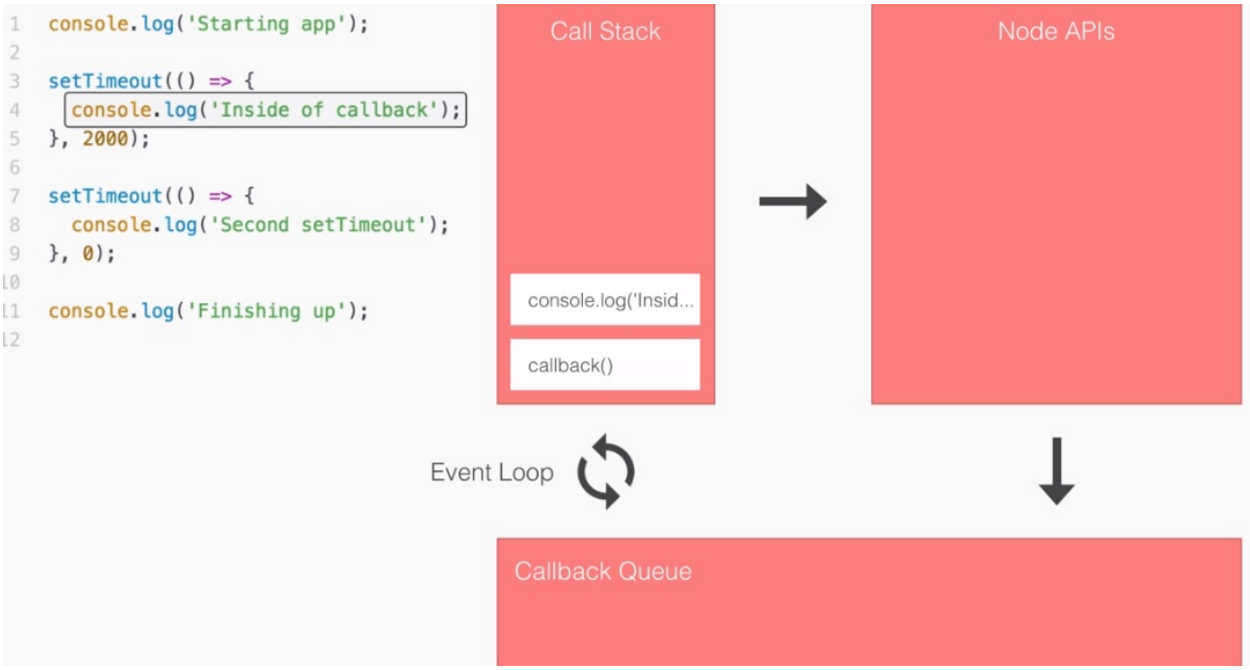
After our `Second setTimeout` statement finishes, the function is going to implicitly return and callback will get removed from the Call Stack:



At this point, there's nothing in the Call Stack and nothing in the Callback Queue, but there is still something in our Node APIs, we still have an event listener registered. So the Node process is not yet completed. Two seconds later, the `setTimeout(2 sec)` event is going to fire, and it's going to take that callback function and move it into the Callback Queue. It gets removed from the Node APIs and it gets added to the Callback Queue:



At this point, the Event Loop will take a look at the Call Stack and see it's empty. Then it will take a quick look at the Callback Queue and see there is indeed something to run. What will it do? It will take that callback, add it on to the Call Stack, and start the process of executing it. This means that we'll run our one statement inside callback. After that's finished, the callback function implicitly returns and our program is complete:



This is exactly how our program ran. This illustrates how we're able to register our events using Node APIs, and why when we use a `setTimeout` of zero the code doesn't run right away. It needs to go through the Node APIs and through the Callback Queue before it can ever execute on the Call Stack.

Now as I mentioned in the beginning of this section, the Call Stack, the Node APIs, the Callback Queue, and the Event Loop are pretty confusing topics. A big reason why they're confusing is because we never actually directly interact with them; they're happening behind the scenes. We're not calling the Callback Queue, we're not firing an Event Loop method to make these things work. This means we're not aware they exist until someone explains them. These are topics that are really hard to grasp the first time around. By writing real asynchronous code it's going to become a lot clearer how it works.

Now that we got a little bit of an idea about how our code executes behind the

scenes, we'll move on with the rest of the chapter and start creating a weather app that interacts with third-party APIs.

Callback functions and APIs

In this section, we'll take an in-depth look at callback functions, and use them to fetch some data from a Google Geolocation API. That's going to be the API that takes an address and returns the latitude and longitude coordinates, and this is going to be great for the weather app. This is because the weather API we use requires those coordinates and it returns the real-time weather data, such as the temperature, five-day forecast, wind speed, humidity, and other pieces of weather information.

The callback function

Before we get started making the HTTPS request, let's talk about callback functions, and we have already used them. Refer to the following code (we used it in the previous section):

```
console.log('Starting app');

setTimeout(() => {
  console.log('Inside of callback');
}, 2000);

setTimeout(() => {
  console.log('Second setTimeout');
}, 0);

console.log('Finishing up');
```

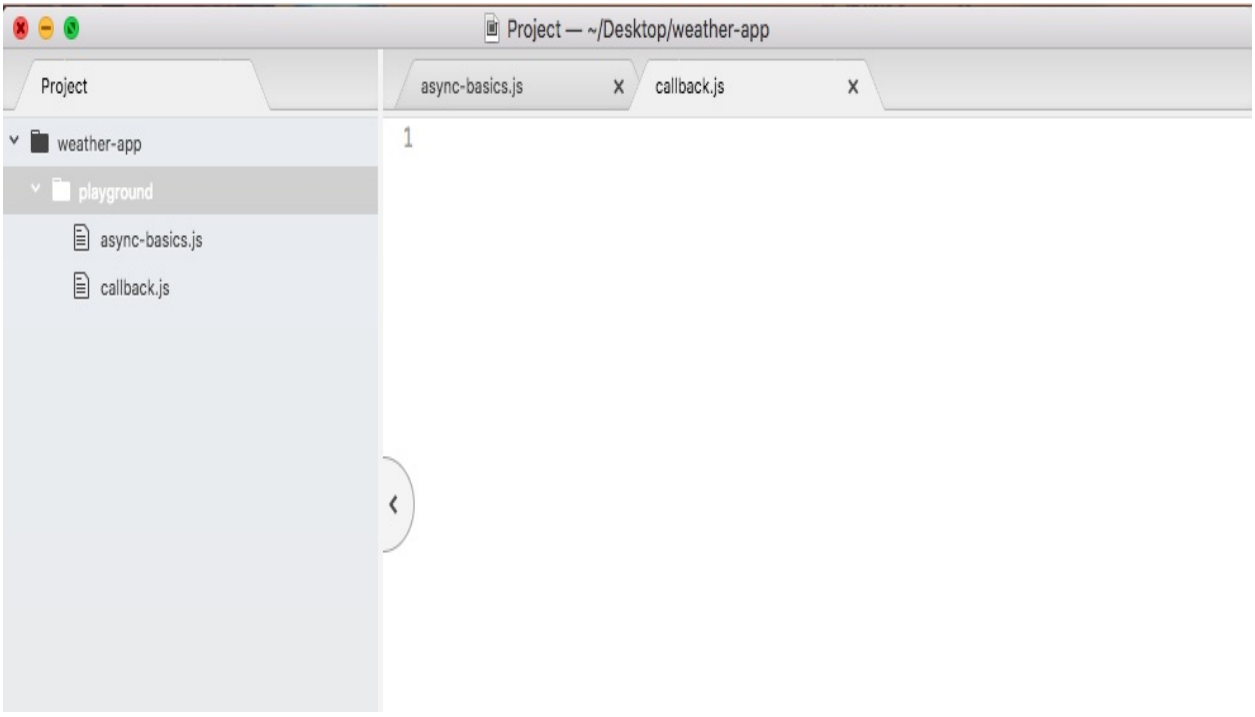
Inside the `setTimeout` function we used a `callback` function. In general, a `callback` function is defined as a function that gets passed as an argument to another function and is executed after some event happens. Now this is a general definition, there is no strict definition in JavaScript, but it does satisfy the function in this case:

```
setTimeout(() => {
  console.log('Inside of callback');
}, 2000);
```

Here we have a function and we pass it as an argument to another function, `setTimeout`, and it does get executed after some event—two-second pass. Now the event could be other things, it could be a database query finishes, it could be an HTTP request comes back. In those cases, you will want a callback function, like the one in our case, to do something with that data. In the case of `setTimeout`, we don't get any data back because we're not requesting any; we're just creating an arbitrary delay.

Creating the callback function

Now before we actually make an HTTP request to Google, let's create a callback function example inside our `playground` folder. Let's make a new file called `callbacks.js`:



Inside the file, we'll create a contrived example of what a callback function would look like behind the scenes. We'll be making real examples throughout the book and use many functions that require callbacks. But for this chapter, we'll start with a simple example.

To get started, let's make a variable called `getUser`. This will be the function we'll define that will show us exactly what happens behind the scenes when we pass a callback to another function. The `getUser` callback will be something that simulates what it would look like to fetch a user from a database or some sort of web API. It will be a function, so we'll set it as such using arrow function (`=>`):

```
var getUser = () => {

};
```

The arrow function (`=>`) is going to take some arguments. The first argument it will take is the `id`, which will be some sort of a unique number that represents each user. I might have an `id` of `54`, you might have an `id` of `2000`; either way we're going to need the `id` to find a user. Next up we'll get a callback function, which is what we will call later with the data, with that user object:

```
var getUser = (id, callback) => {

};
```

This is exactly what happens when you pass a function to `setTimeout`.

The `setTimeout` function definition looks like this:

```
var getUser = (callback, delay) => {

};
```

It has a callback and a delay. You take the callback, and after a certain amount of time passes, you call it. In our case, though, we'll switch the order with an `id` first and the callback second.

Now we can call this function before actually filling it out. We'll call `getUser`, just like we did with `setTimeout` in the previous code example. I'll call `getUser`, passing in those two arguments. The first one will be some `id`; since we're faking it for now it doesn't really matter, and I'll go with `31`. The second argument will be the function that we want to run when the user data comes back, and this is really important. As shown, we'll define that function:

```
getUser(31, () => {

});
```

Now the callback alone isn't really useful; being able to run this function after the user data comes back only works if we actually get the user data, and that's what we'll expect here:

```
getUser(31, (user) => {

});
```

We'll expect that the `user` objects, things like `id`, `name`, `email`, `password`, or whatever, comes back as an argument to the callback function. Then inside the arrow function (`=>`), we can actually do something with that data, for example, we could show it on a web app, respond to an API request, or in our case we can simply

print it to the console, `console.log(user)`:

```
getUser(31, (user) => {
  console.log(user);
});
```

Now that we have the call in place, let's fill out the `getUser` function to work like we have it defined.

The first thing I'll do is create a dummy object that's going to be the `user` object. In the future, this is going to come from database queries, but for now we'll just create a variable `user` setting it equal to some object:

```
var getUser = (id, callback) => {
  var user = {

  }
};
```

Let's set an `id` property equal to whatever `id` the user passes in, and we'll set a `name` property equal to some name. I'll use `Vikram`:

```
var getUser = (id, callback) => {
  var user = {
    id: id,
    name: 'Vikram'
  };
};
```

Now that we have our `user` object, what we want to do is call the callback, passing it as an argument. We'll then be able to actually run, `getUser(31, (user)` function, printing the `user` to the screen. In order to do this, we would call the callback function like any other function, simply referencing it by name and adding our parentheses like this:

```
var getUser = (id, callback) => {
  var user = {
    id: id,
    name: 'Vikram'
  };
  callback();
};
```

Now if we call the function like this, we're not passing any data from `getUser` back to the callback. In this case, we're expecting a `user` to get passed back, which is why we are going to specify `user` as shown here:

```
callback(user);
```

Now the naming isn't important, I happen to call it `user`, but I could easily call this `userObject` and `userObject` as shown here:

```
callback(user);
};

getUser(31, (userObject) => {
  console.log(userObject);
});
```

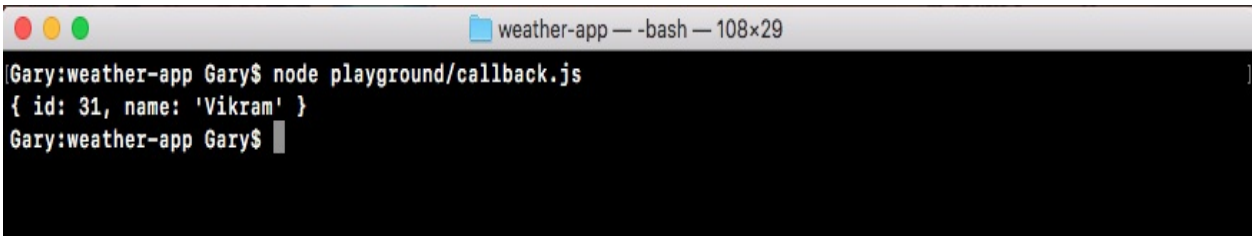
All that matters is the arguments, position. In this case, we call the first argument `userObject` and the first argument pass back is indeed that `userObject`. With this in place we can now run our example.

Running the callback function

In the Terminal, we'll run the callback function using `node`, which is in the `playground` folder, and we call the file `callbacks.js`:

```
node playground/callback.js
```

When we run the file, right away our data prints to the screen:



We've created a callback function using synchronous programming. Now as I mentioned, this is still a contrived example because there is no need for a callback in this case. We could simply return the user object, but in that case, we wouldn't be using a callback, and the whole point here is to explore what happens behind the scenes and how we actually call the function that gets passed in as an argument.

Simulating delay using setTimeout

Now, we can also simulate a delay using `setTimeout`, so let's do that. In our code, just before the `callback (user)` statement, we'll use `setTimeout` just like we did before in the previous section. We'll pass an arrow function (`=>`) in as the first argument, and set a delay of 3 seconds using `3000` milliseconds:

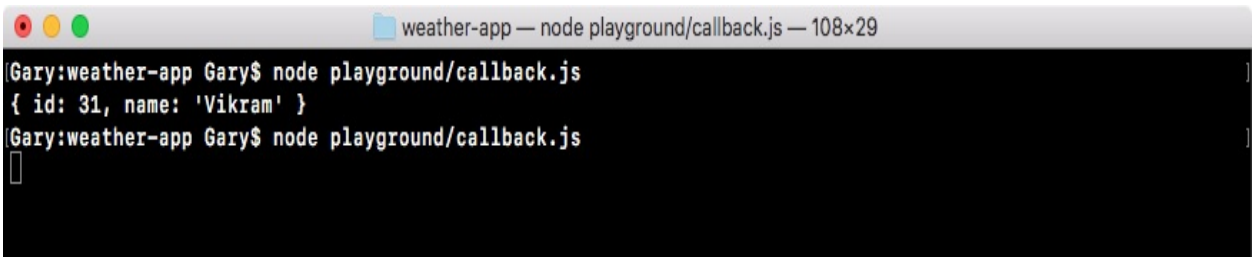
```
  setTimeout(() => {

  }, 3000);
  callback(user);
};
```

Now I can take my callback call, delete it from line 10, and add it inside of the callback function, as shown here:

```
setTimeout(() => {
    callback(user);
  }, 3000);
};
```

Now we'll not be responding to the `getUser` request until three seconds have passed. Now this will be more or less similar to what happens when we create real-world examples of callbacks, we pass in a callback, some sort of delay happens whether we're requesting from a database or from an HTTP endpoint, and then the callback gets fired.

If I save `callbacks.js` and rerun the code from the Terminal, you'll see we wait those three seconds, which is the simulated delay, and then the `user` object prints to the screen:



This is exactly the principle that we need to understand in order to start working with callbacks, and that is exactly what we'll start doing in this section.

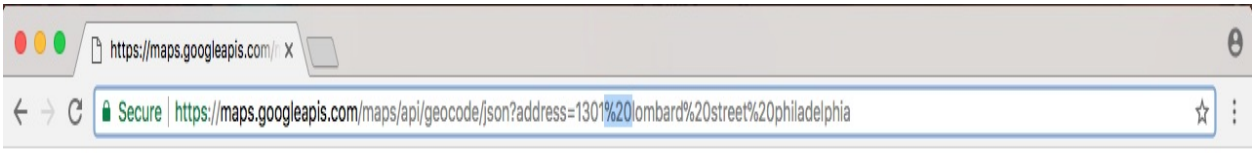
Making request to Geolocation API

The requests that we'll be making to that Geolocation API can actually be simulated over in the browser before we ever make the request in Node, and that's exactly what we want to do to get started. So follow along for the URL, `https://maps.googleapis.com/maps/api/geocode/json`.

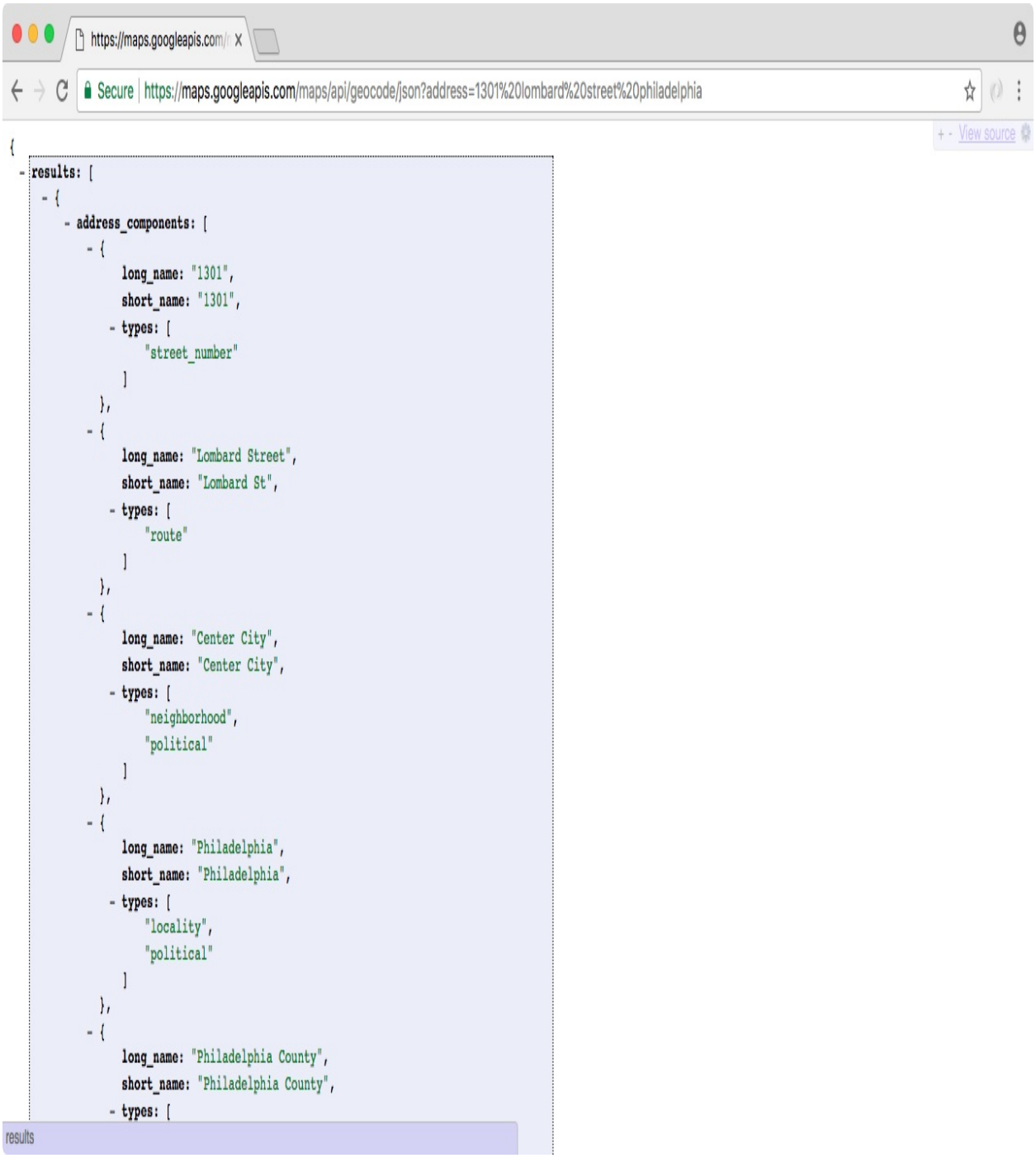
Now this is the actual endpoint URL, but we do have to specify the address for which we want the geocode. We'll do that using query strings, which will be provided right after the question mark. Then, we can set up a set of key value pairs and we can add multiples using the ampersand in the URL, for example: `https://maps.googleapis.com/maps/api/geocode/json?key=value&keytwo=valuetwo`.

In our case, all we need is one query string address, `https://maps.googleapis.com/maps/api/geocode/json?address`, and for the address query string we'll set it equal to an address. In order to fill out that query address, I'll start typing `1301 lombard street philadelphia`.

Notice that we are using spaces in the URL. This is just to illustrate a point: we can use spaces in the browser because it's going to automatically convert those spaces to something else. However, inside Node we'll have to take care of that ourselves, and we'll talk about that a little later in the section. For now if we leave the spaces in, hit e*nter*, and we can see they automatically get converted for us:



Space characters get converted to `%20`, which is the encoded version of a space. In this page, we have all of the data that comes back:



Now we'll use an extension called JSONView, which is available for Chrome and Firefox.

I highly recommend installing JSONView, as we should see a much nicer version of our JSON data. It lets us minimize and expand various properties, and it makes it super easy to navigate.

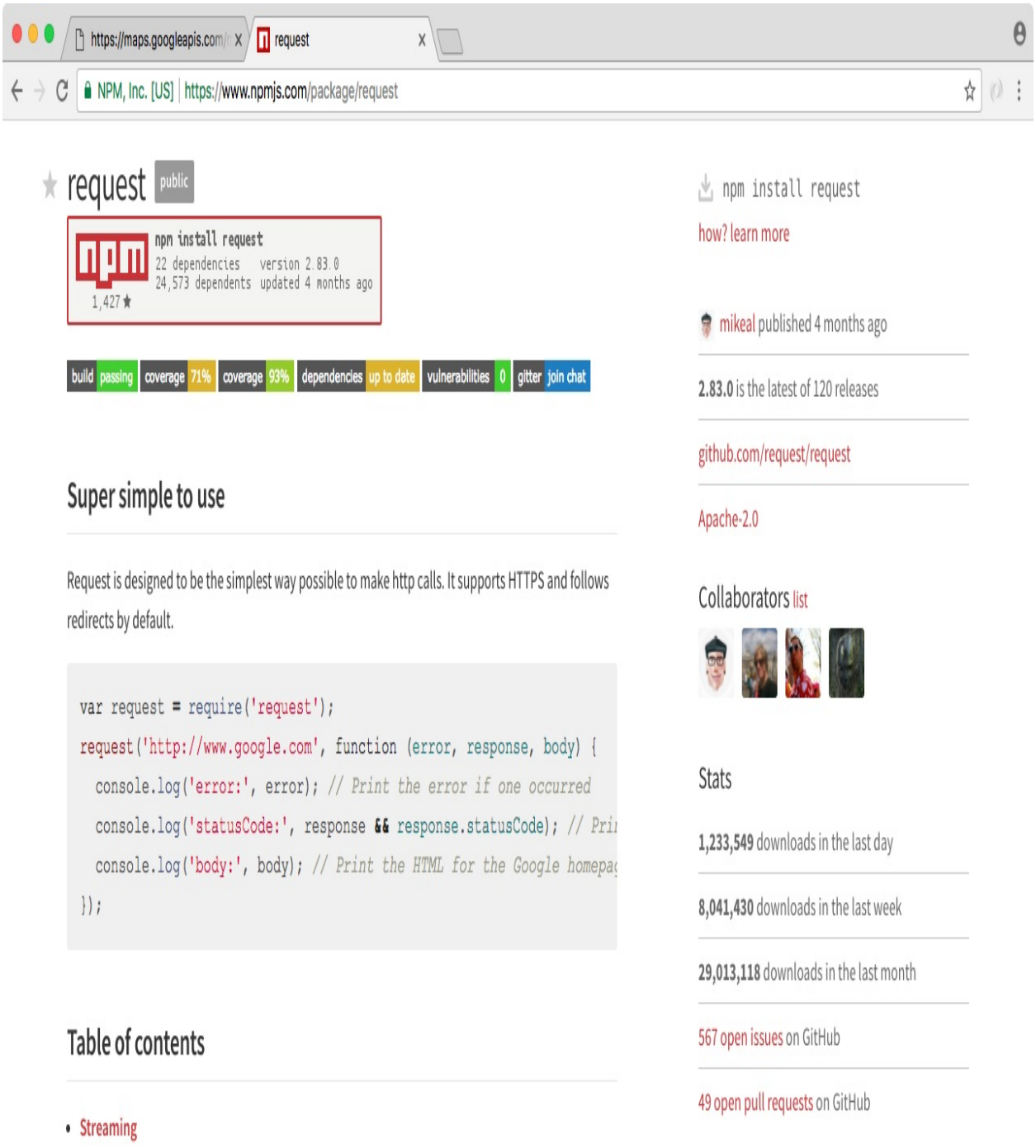
Now as shown in the preceding screenshot, the data on this page has exactly what we need. We have an address_components property, we don't need that. Next, we have a formatted address which is really nice, it includes the state, the zip code, and the country, which we didn't even provide in the address query.

Then, we have what we really came for: in geometry, we have location, and this includes the latitude and longitude data.

Using Google Maps API data in our code

Now, what we got back from the Google Maps API request is nothing more than some JSON data, which means we can take that JSON data, convert it to a JavaScript object, and start accessing these properties in our code. To do this, we'll use a third-party module that lets us make these HTTP requests inside of our app; this one is called request.

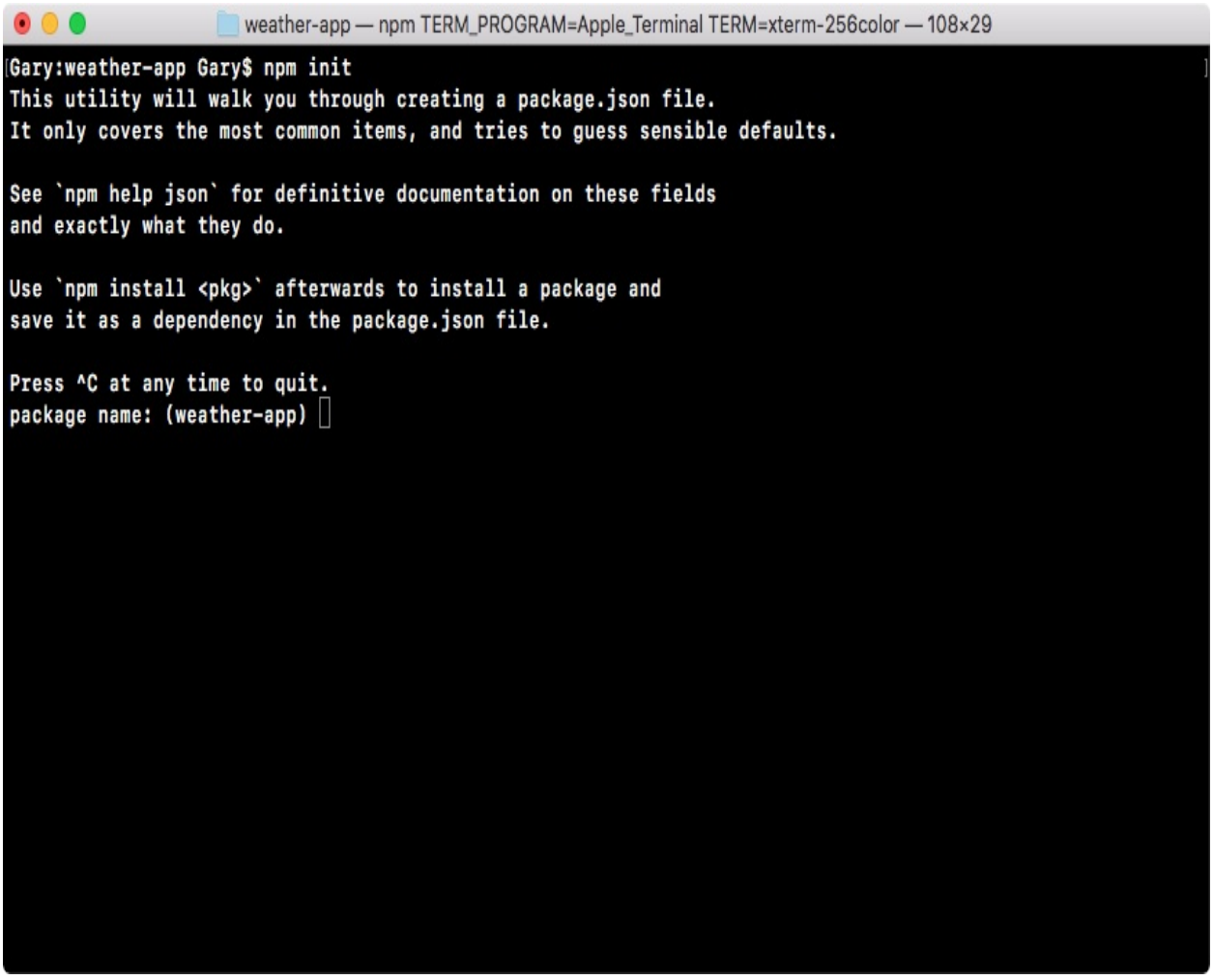
We can visit it by going to `https://www.npmjs.com/package/request`. When we visit this page, we'll see all the documentation and all the different ways we can use the request package to make our HTTP requests. For now, though, we'll stick to some basic examples. On the request documentation page, on the right-hand side, we can see this is a super popular package and it has seven hundred thousand downloads in the last day:



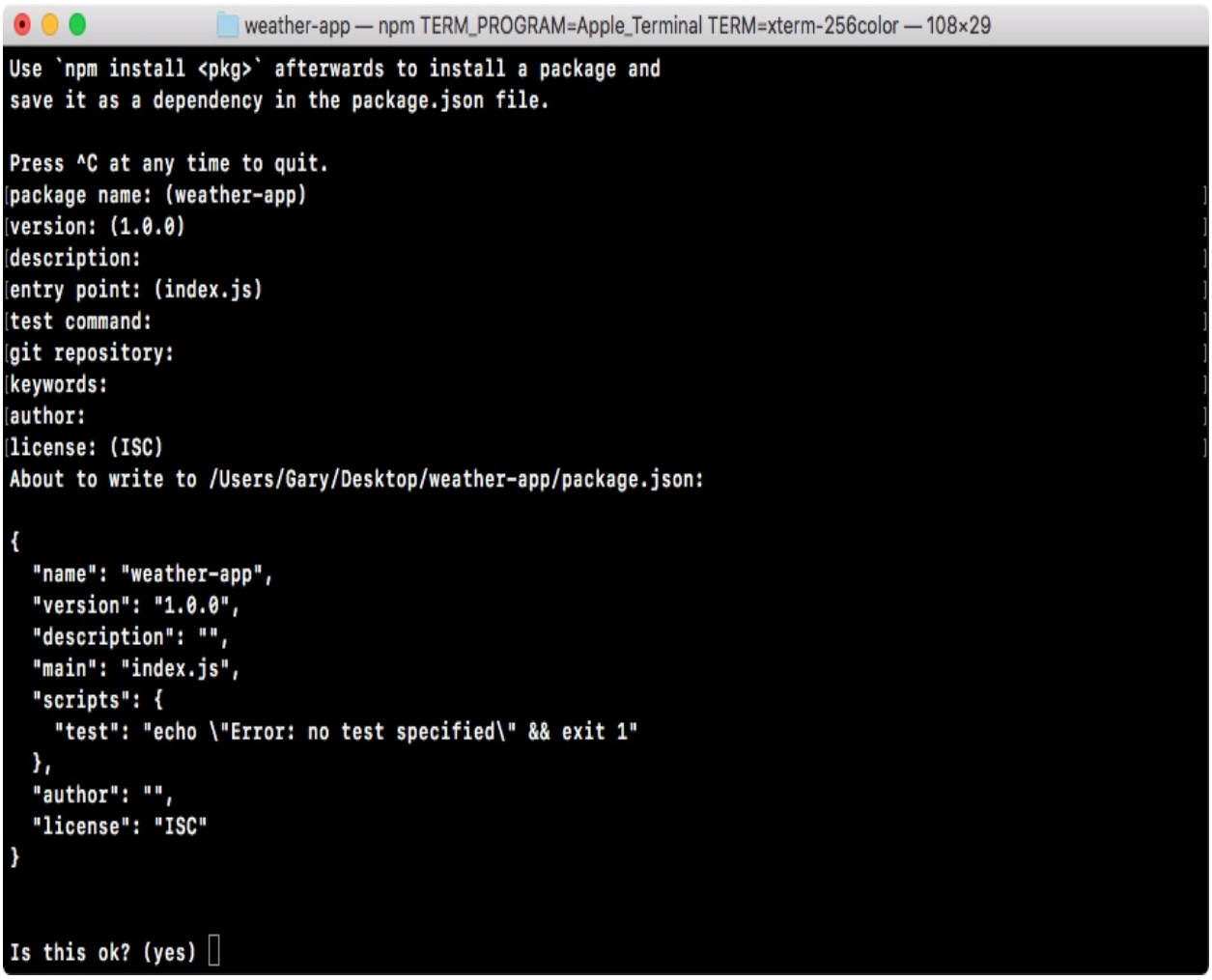
In order to get started we're going to install the package inside our project, and we'll make a request to this URL.

Installing the request package

To install the package, we'll go to the Terminal and install the module using `npm init`, to create the `package.json` file:



We'll run this command and use e*nter* to use the defaults for every single option:



At the end, we'll type `yes` and hit e*nter* again.

Now that we have our `package.json` file we can use `npm install`, followed by the module name, request, and I will specify a version. You can always find the latest version of modules on the npm page. The latest version at the time of writing is `2.73.0`, so we'll add that, `@2.73.0`. Then we can specify the save flag because we do want to save this module in our `package.json` file:

```
npm install request@2.73.0 --save
```

It will be critical for running the weather application.

Using request as a function

Now that we have the request module installed, we can start using it. Inside Atom we'll wrap up the section by making a request to that URL, in a new file in the root of the project called `app.js`. This will be the starting point for the weather application. The weather app will be the last command-line app we create. In the future we'll be making the backend for web apps as well as real-time apps using Socket.IO. But to illustrate asynchronous programming, a command-line app is the nicest way to go.

Now, we have our app file, and we can get started by loading in `request` just like we did with our other npm modules. We'll make a constant variable, call it `request`, and set it equal to `require(request)`, as shown here:

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

Now what we need to do is make a `request`. In order to do this, we'll have to call the `request` function. Let's call it, and this function takes two arguments:

- The first argument will be an options object where we can configure all sorts of information
- The second one will be a callback function, which will be called once the data comes back from the HTTP endpoint

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

});
```

In our case, it's going to get called once the JSON data, the data from the Google Maps API, comes back into the Node application. We can add the arguments that are going to get passed back from `request`. Now, these are arguments that are outlined in the `request` documentation, I'm not making up the names for these:

In the documentation, you can see they call it **error**, **response**, and **body**. That's exactly what well call ours. So, inside Atom, we can add `error`, `response`, and `body`, just like the docs.

Now we can fill out that options object, which is where we are going to specify the things unique to our `request`. In this case, one of the unique things is the URL. The URL specifies exactly what you want to request, and in our case, we have that in the browser. Let's copy the URL exactly as it appears, pasting it inside of the string for the URL property:

```
request({
  url: 'https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20stre
}, (error, response, body) => {

});
```

Now that we have the URL property in place, we can add a comma at the very end and hit e*nter*. Because we will specify one more property, we'll set `json` equal to `true`:

```
request({
  url: 'https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20stre
  json: true
}, (error, response, body) => {

});
```

This tells `request` that the data coming back is going to be JSON data, and it

should go ahead, take that JSON string, and convert it to an object for us. That lets us skip a step, it's a really useful option.

With this in place, we can now do something in the callback. In the future we'll be taking this longitude and latitude and fetching weather. For now, we'll simply print the `body` to the screen by using `console.log`. We'll pass the body argument into `console.log`, as shown here:

```
request({
  url: 'https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20stre
  json: true
}, (error, response, body) => {
  console.log(body);
});
```

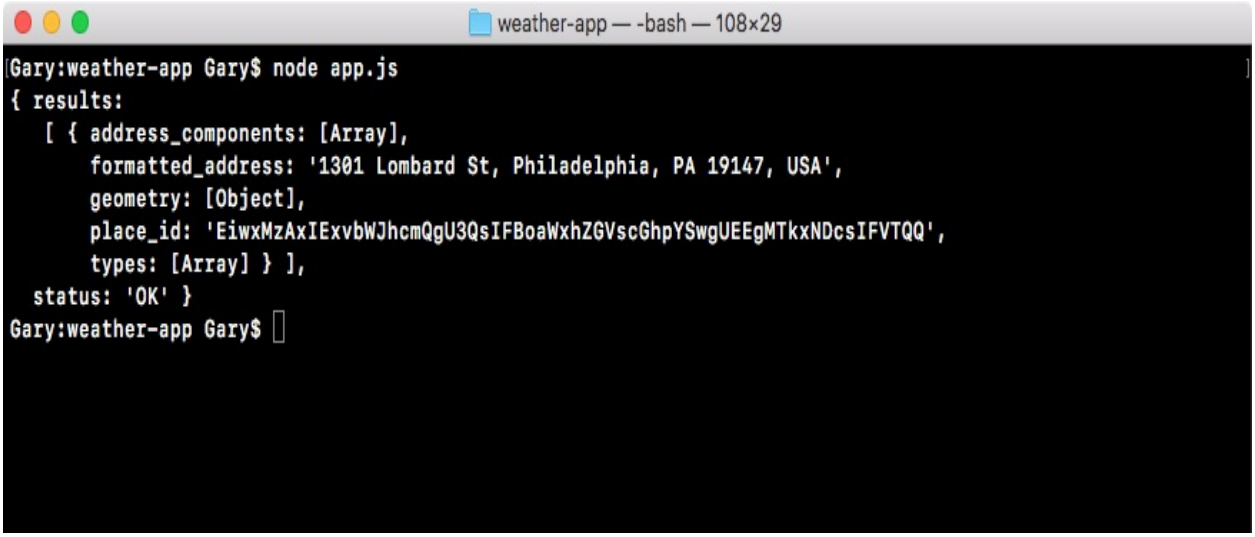
Now that we have our very first HTTP request set up, and we have a callback that's going to fire when the data comes back, we can run it from the Terminal.

Running the request

To run the request, we'll use `node` and run the `app.js` file:

```
node app.js
```

When we do this, the file will start executing and there will be a really short delay before the body prints to the screen:



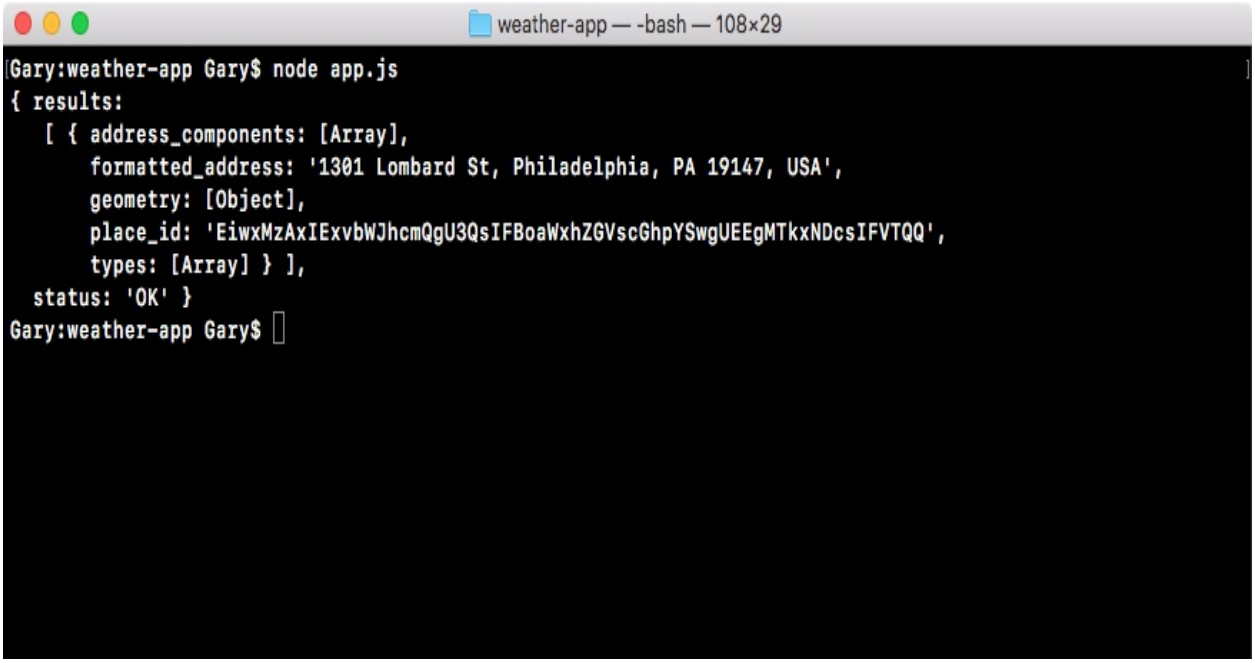
What we get back is exactly what we saw in the browser. Some of the properties, such as `address_components`, show object in this case because we're printing it to the screen. But those properties do indeed exist; we'll talk about how to get them later in the chapter. For now, though, we do have our `formatted_address` as shown in the preceding screenshot, the `geometry` object, the `place_id`, and `types`. This is what we'll be using to fetch the longitude and latitude, and later to fetch the weather data.

Now that we have this in place, we are done. We have the first step of the process complete. We've made a request to the Google Geolocation API, and we're getting the data back. We'll continue creating the weather app in the next section.

Pretty printing objects

Before we continue learning about HTTP and what exactly is inside of `error`, `response`, and `body`, let's take a quick moment to talk about how we can pretty print an object to the screen. As we saw in the last subsection, when we ran our app with `node app.js`, the body prints to the screen.

But since there is a lot of objects nested inside of each other, JavaScript starts clipping them:



As shown in the preceding screenshot, it tells us an object is in the `results`, but we don't get to see exactly what the properties are. This takes place for `address_components`, `geometry`, and `types`. Obviously this is not useful; what we want to do is see exactly what's in the object.

Using the body argument

To explore all of the properties, we're going to look at a way to pretty print our objects. This is going to require a really simple function call, a function we've actually already used, `JSON.stringify`. This is the function that takes your JavaScript objects, which `body` is, remember we used the `json: true` statement to tell `request` to take the JSON and convert it into an object. In the `console.log`, statement we'll take that object, pass `body` in, and provide the arguments as shown here:

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

request({
  url: 'https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20stre
  json: true
}, (error, response, body) => {
  console.log(JSON.stringify(body));
});
```

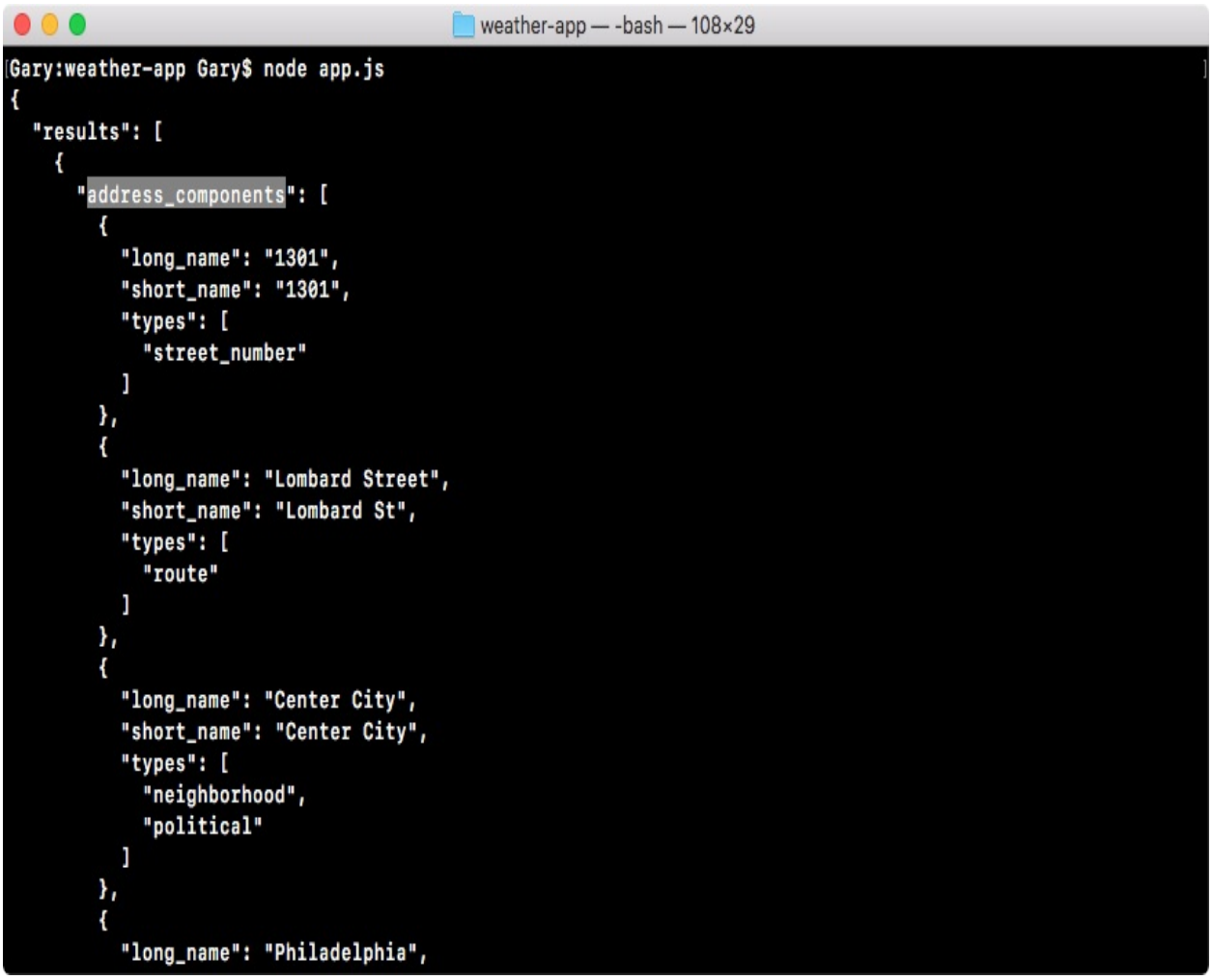
Now, this is how we've usually used `JSON.stringify`, in the past we provided just one argument, the object we want to `stringify`, in this case we're going to provide a couple of other arguments. The next argument is used to filter out properties. We don't want to use that, it's usually useless, so we're going to leave it as undefined as of now:

```
console.log(JSON.stringify(body, undefined));
```

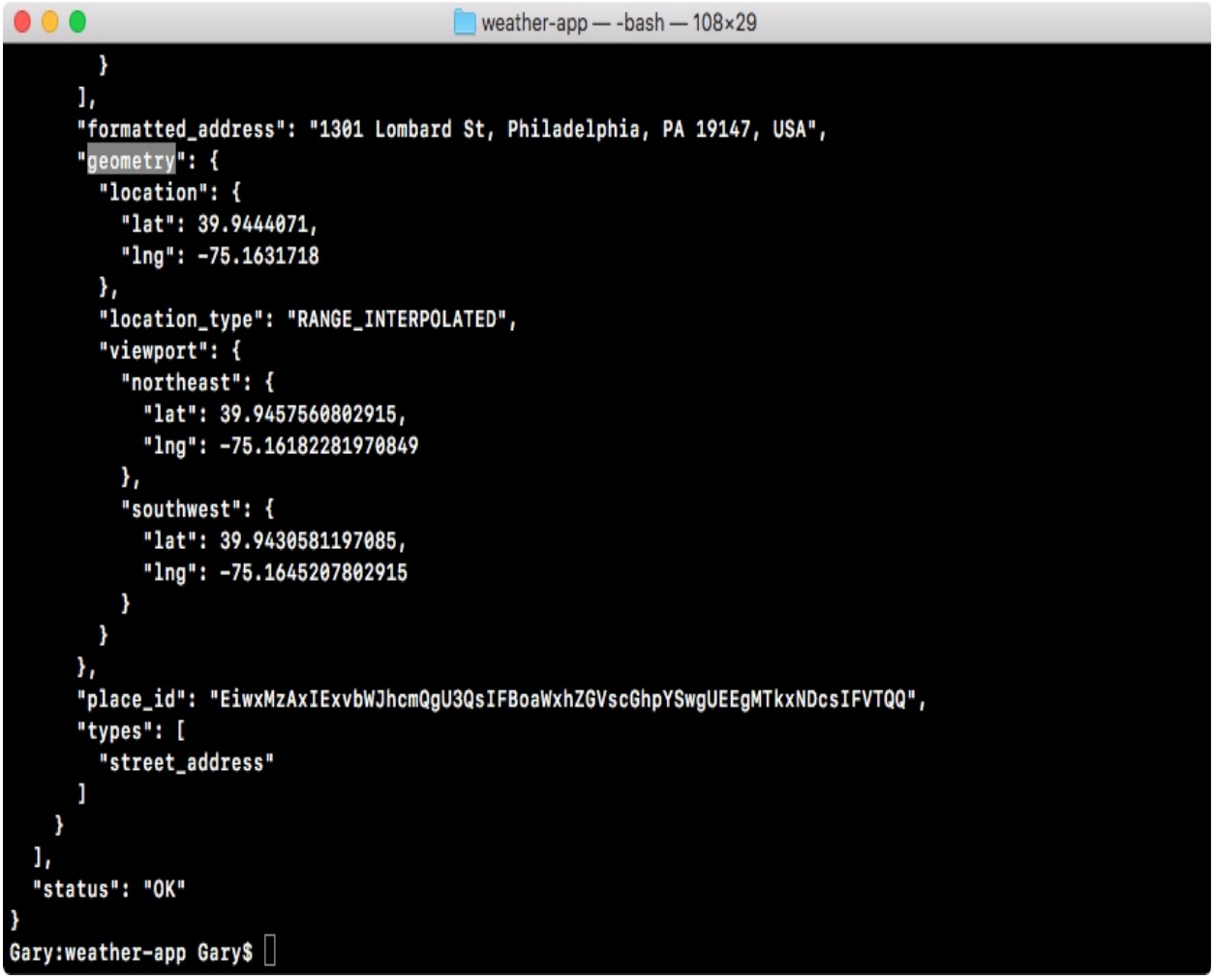
The reason we need to provide it, is because the third argument is the thing we want. The third argument will format the JSON, and we'll specify exactly how many spaces we want to use per indentation. We could go with `2` or `4` depending on your preference. In this case, we'll pick `2`:

```
console.log(JSON.stringify(body, undefined, 2));
```

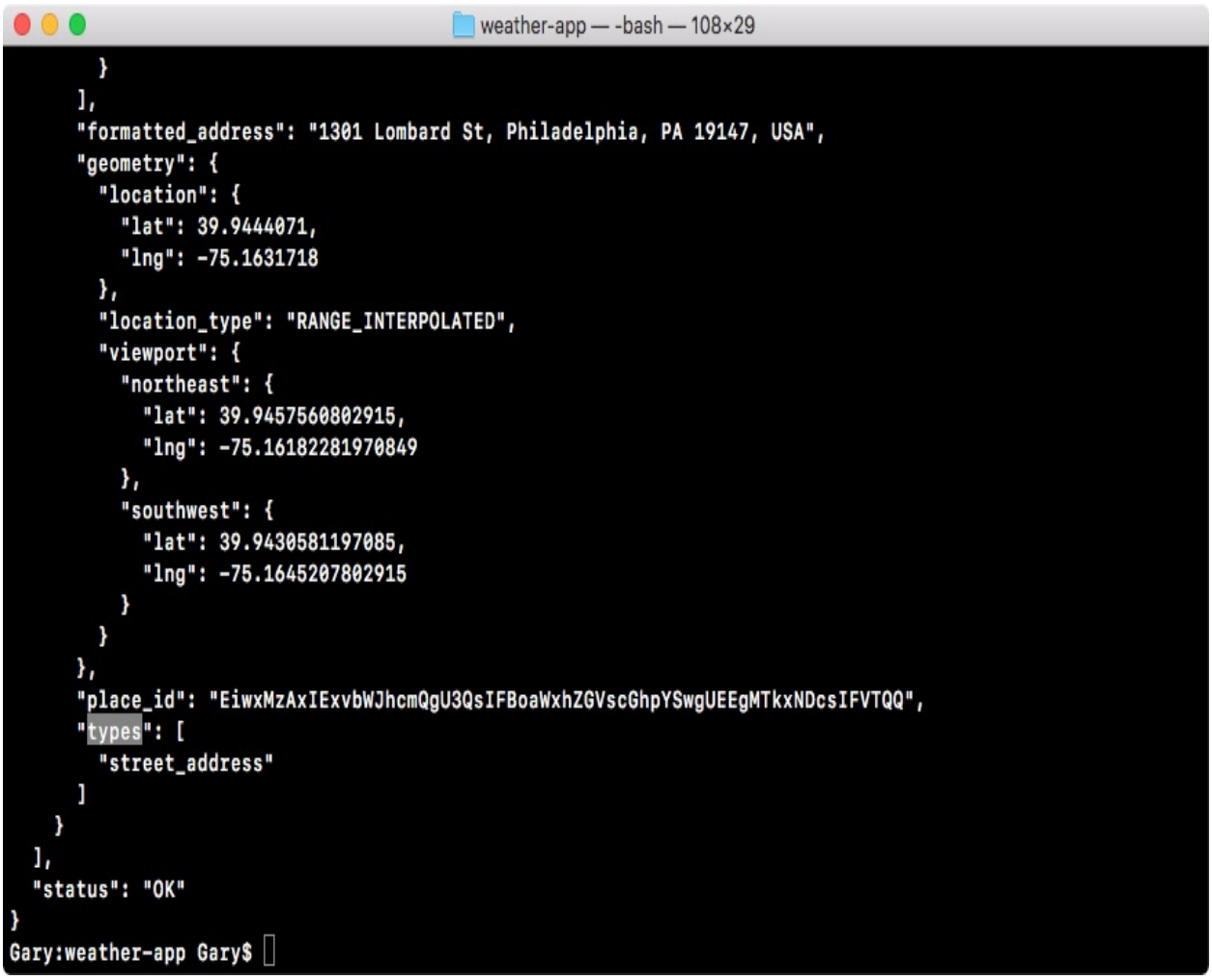
We'll save the file and rerun it from the Terminal. When we `stringify` our JSON and print it to the screen, as we'll see when we rerun the app, we get the entire object showing up. None of the properties are clipped off, we can see the entire address_components array, everything shows up no matter how complex it is:



Next, we have our geometry object, this is where our latitude and longitude are stored, and you can see them as shown here:



Then below that, we have our `types`, which was cut off before, even though it was an array with one item, which is a string:



Now that we know how to pretty print our objects, it will be a lot easier to scan data inside of the console—none of our properties will get clipped, and it's formatted in a way that makes the data a lot more readable. In the next section, we'll start diving into HTTP and all of the arguments in our callback.

Making up of the HTTPS requests

The goal in the previous section was not to understand how HTTP works, or what exactly the arguments, `error`, `response`, and `body` are the goal was to come up with a real-world example of a callback, as opposed to the contrived examples that we've been using so far with `setTimeout`:

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

request({
 url: 'https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20stree
 json: true
}, (error, response, body) => {
 console.log(JSON.stringify(body, undefined, 2));
});
```

In the preceding case, we had a real callback that got fired once the HTTP request came back from the Google servers. We were able to print the `body`, and we saw exactly what we had in the website. In this section, we'll dive into these arguments, so let's kick things off by taking a look at the `body` argument. This is the third argument that `request` passes to the callback.

Now the `body` is not something unique to the `request` module (`body` is part of HTTP, which stands for the **Hypertext Transfer Protocol**). When you make a request to a website, the data that comes back is the body of the request. We've actually used the body about a million times in our life. Every single time we request a URL in the browser, what we get rendered inside the screen is the body.

In the case of `https://www.npmjs.com`, the body that comes back is an HTML web page that the browser knows how to render. The body could also be some JSON information, which is the case in our Google API request. Either way, the body is the core data that comes back from the server. In our case, the body stores all of the location information we need, and we'll be using that information to pull out the formatted address, the latitude, and the longitude in this section.

The response object

Before we dive into the body, let's discuss about the `response` object. We can look at the `response` object by printing it to the screen. Let's swap out body in the `console.log` statement for `response` in the code:

```
const request = require('request');
request({
  url: 'https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20stre
  json: true
}, (error, response, body) => {
  console.log(JSON.stringify(response, undefined, 2));
});
```

Then save the file and rerun things inside of the Terminal by running the `node app.js` command. We'll get that little delay while we wait for the request to come back, and then we get a really complex object:

In the preceding screenshot, we can see the first thing we have in the `response` object is a status code. The status code is something that comes back from an HTTP request; it's a part of the response and tells you exactly how the request went.

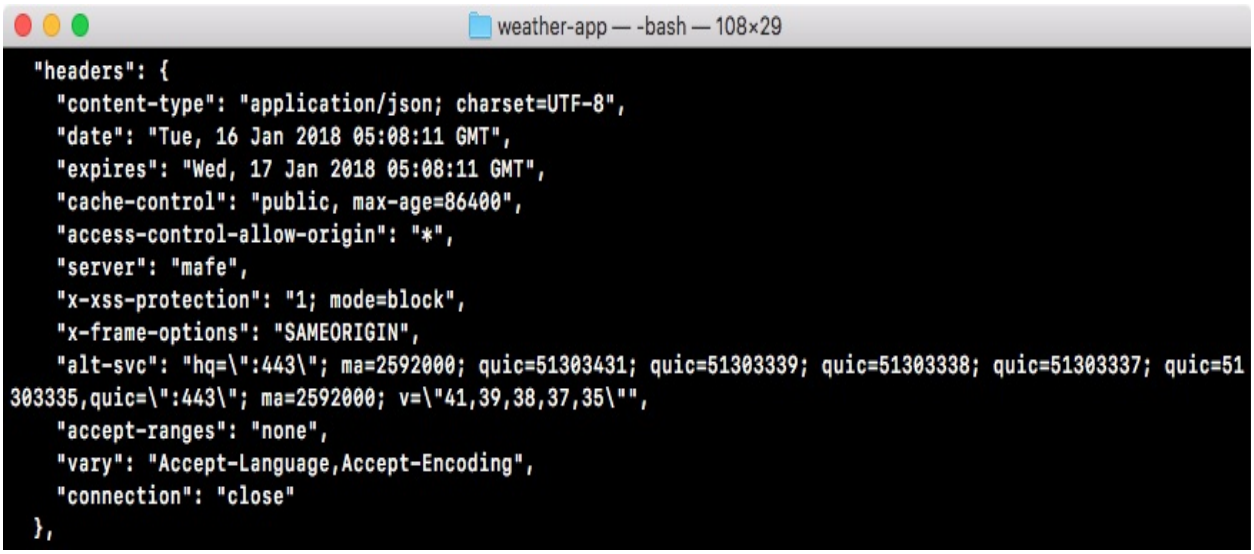
In this case, `200` means everything went great, and you're probably familiar with some status codes, like 404 which means the page was not found, or 500 which means the server crashed. There are other body codes we'll be using throughout the book.

We'll be making our very own HTTP API, so you'll become intimately familiar with how to set and use status codes.

In this section, all we care about is that the status code is `200`, which means things

went well. Next up in the `response` object, we actually have the `body` repeated because it is part of the `response`. Since it's the most useful piece of information that comes back, the request module developers chose to make it the third argument, although you could access it using `response.body` as you can clearly see in this case. Here, we have all of the information we've already looked at, address components, formatted address geometry, so on.

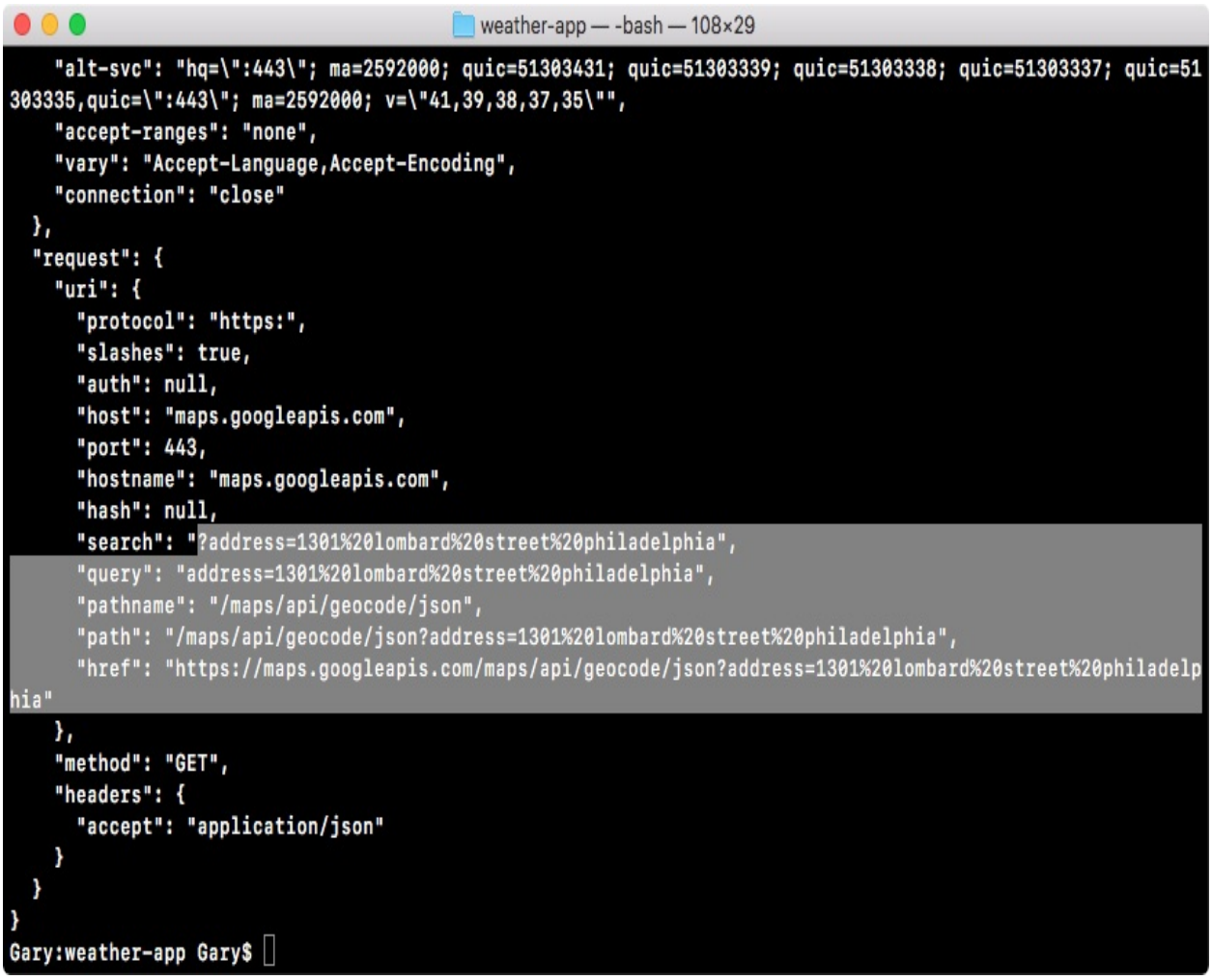
Next to the body argument, we have something called `headers`, as shown here:



Now, `headers` are part of the HTTP protocol, they are key-value pairs as you can see in the preceding screenshot, where the key and the value are both strings. They can be sent in the request, from the Node server to the Google API server, and in the response from the Google API server back to the Node server.

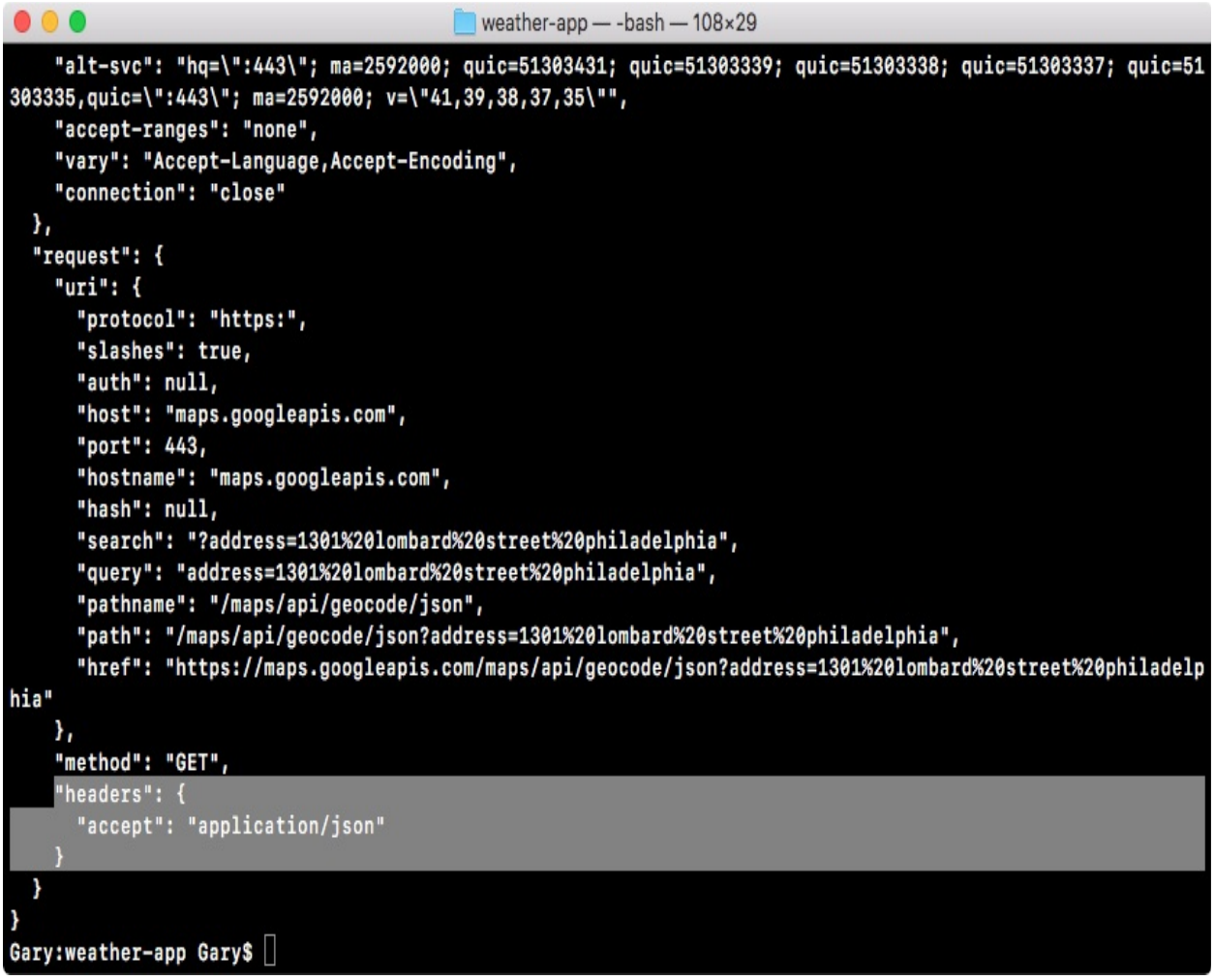
Headers are great, there's a lot of built-in ones like `content-type`. The `content-type` is HTML for a website, and in our case, it's `application/json`. We'll talk about headers more in the later chapters. Most of these headers are not important to our application, and most we're never ever going to use. When we go on and create our own API later in the book, we'll be setting our own headers, so we'll be intimately familiar with how these headers work. For now, we can ignore them completely, all I want you to know is that these headers you see are set by Google, they're headers that come back from their servers.

Next to the headers we have the request object, which stores some information about the request that was made:



As shown in the preceding screenshot, you can see the protocol HTTPS, the host, the `maps.googleapis.com` website, and other things such as the address parameters, the entire URL, and everything else about the request, which is stored in this part.

Next, we also have our own headers. These are headers that were sent from Node to the Google API:



This header got set when we added `json: true` to options object in our code. We told request we want JSON back and request went on to tell Google, *Hey, we want to accept some JSON data back, so if you can work with that format send it back!* And that's exactly what Google did.

This is the `response` object, which stores information about the `response` and about the request. While we'll not be using most of the things inside the `response` argument, it is important to know they exist. So if you ever need to access them, you know where they live. We'll use some of this information throughout the book, but as I mentioned earlier, most of it is not necessary.

For the most part, we're going to be accessing the body argument. One thing we will use is the status. In our case it was `200`. This will be important when we're making sure that the request was fulfilled successfully. If we can't fetch the location or if we get an error in the status code, we do not want to go on to try to

fetch the weather because obviously we don't have the latitude and longitude information.

The error argument

For now, we can move on to the final thing which is error. As I just mentioned, the status code can reveal that an error occurred, but this is going to be an error on the Google servers. Maybe the Google servers have a syntax error and their program is crashing, maybe the data that you sent is invalid, for example, you sent an address that doesn't exist. These errors are going to become evident via the status code.

What the error argument contains is errors related to the process of making that HTTP request. For example, maybe the domain is wrong: if I delete `s` and the dot with `go` in the URL, in our code, I get a URL that most likely doesn't exist:

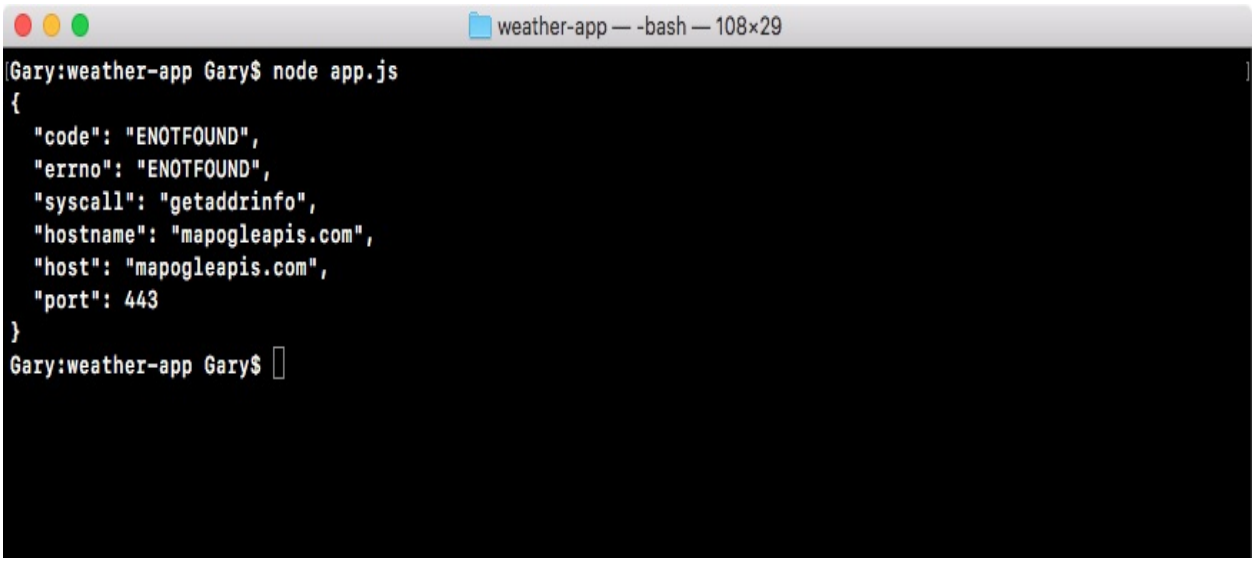
```
const request = require('request');

request({
  url: 'https://mapogleapis.com/maps/api/geocode/json?address=1301%20lombard%20street%2
```

In this case, I'll get an error in the error object because Node cannot make the HTTP request, it can't even connect it to the server. I could also get an error if the machine I'm making the request from does not have access to the internet. It's going to try to reach out to the Google servers, it's going to fail, and we're going to get an error.

Now, we can check out the error object by deleting those pieces of text from the URL and making a request. In this case, I'll swap out response for `error`, as shown here:

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

request({
  url: 'https://mapogleapis.com/maps/api/geocode/json?address=1301%20lombard%20street%2
  json: true
}, (error, response, body) => {
  console.log(JSON.stringify(error, undefined, 2));
});
```

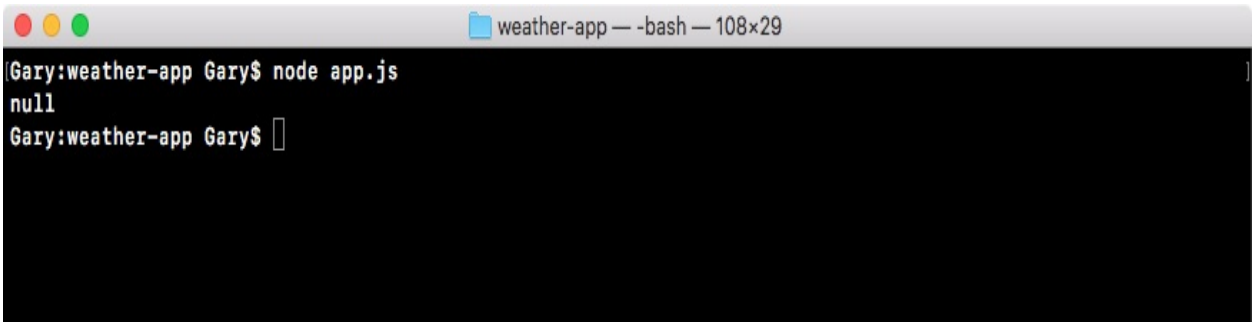
Now, inside the Terminal, let's rerun the application by running the `node app.js` command, and we can see exactly what we get back:



When we make the bad request, we get our error object printing to the screen, and the thing we really care about is the error code. In this case we have the `ENOTFOUND` error. This means that our local machine could not connect to the host provided. In this case `mapogleapis.com`, it doesn't exist so we'll get an error right here.

These are going to be the system errors, things such as your program not being able to connect to the internet or the domain not being found. This is also going to be really important when it comes to creating some error handling for our application there is a chance that the user's machine won't be connected to the internet. We're going to want to make sure to take the appropriate action and we'll do that depending on what is inside the error object.

If we can fix the URL, setting it back to `maps.googleapis.com`, and make the exact same request by using the up arrow key and the *enter* key, the request error object it's going to be empty, and you can see null print to the screen:



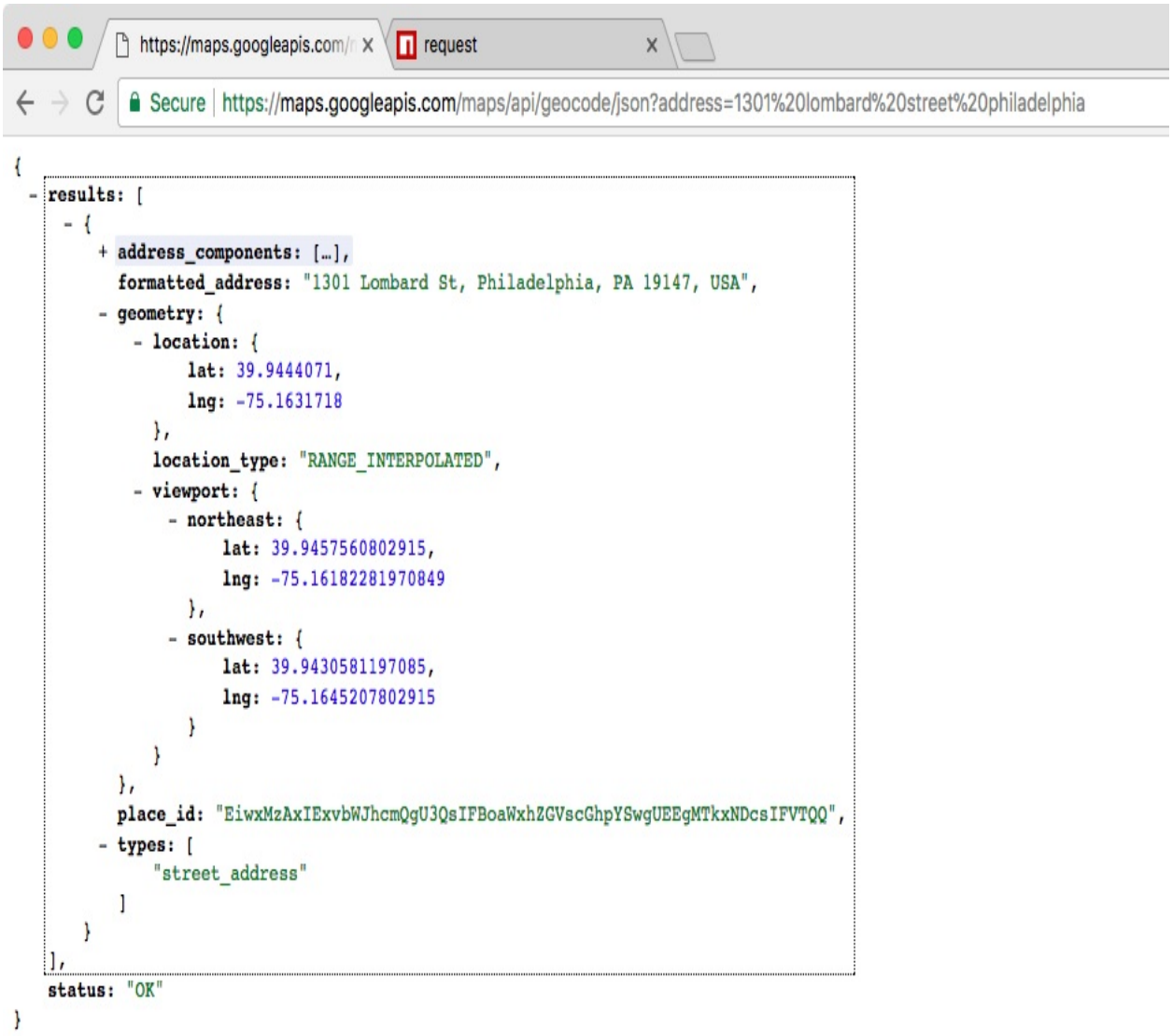
In this case, everything went great, there was no error, and it was able to successfully fetch the data, which it should be able to because we have a valid URL. That is a quick rundown of the body, the `response`, and the error argument. We will use them in more detail as we add error handling.

Printing data from the body object

Now, we'll print some data from the body to the screen. Let's get started by printing the formatted address, and then we will be responsible for printing both the latitude and the longitude.

Printing the formatted address

We'll start with figure out where the formatted address is. For this, we'll go to the browser and use JSONView. At the bottom of the browser page, you can see that little blue bar shows up when we highlight over items, and it changes as we switch items. For formatted address, for example, we access the `results` property, results is an array. In the case of most addresses, you'll only get one result:



We'll use the first result every time, so we have the index of `0`, then it's the `.formatted_address` property. This bottom line is exactly what we need to type inside of our Node code.

Inside Atom, in our code, we'll delete the `console.log` statement, and replace it with a new `console.log` statement. We'll use template strings to add some nice formatting to this. We'll add `Address` with a colon and a space, then I'll inject the address using the dollar sign and the curly braces. We'll access the body, results, and the first item in the results array followed by formatted address, as shown here:

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

request({
 url: 'https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20stree
 json: true
}, (error, response, body) => {
 console.log(`Address: ${body.results[0].formatted_address}`);
});
```

With this in place, I can now add a semicolon at the end and save the file. Next, we'll rerun the application inside of the Terminal, and this time around we get our address printing to the screen, as shown here:

Now that we have the address printing to the screen, what we would like to print both the latitude and the longitude next.

Printing latitude and longitude

In order to get started, inside Atom, we'll add another `console.log` line right next to the `console.log` we added for formatted address. We'll use template strings again to add some nice formatting. Let's print the latitude first.

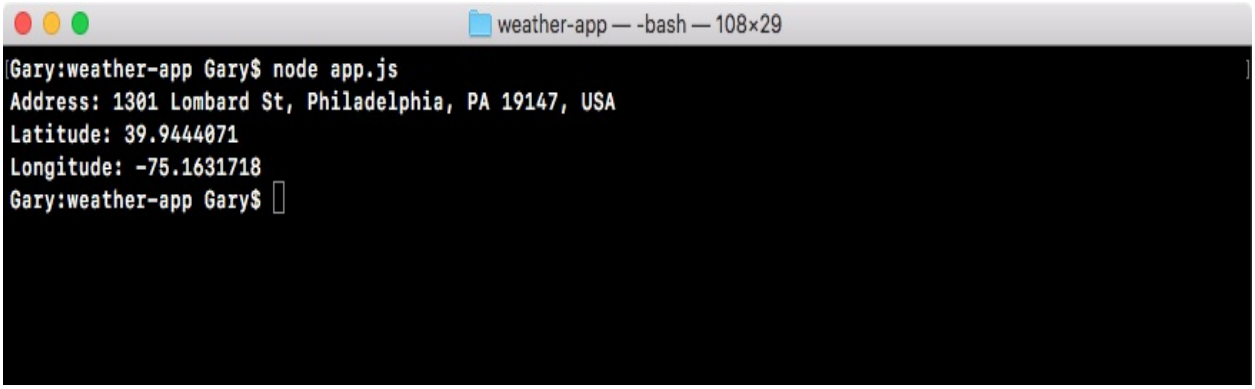
For this, we'll add latitude followed by a colon. Then we can inject our variable using the dollar sign with the curly braces. Then, the variable we want is on the body. Just like the formatted address, it's also in the first results item; results at the index of zero. Next, we'll be going into geometry. From geometry, we'll grab the location property, the latitude, `.lat`, as shown here:

```
  console.log(`Address: ${body.results[0].formatted_address}`);
  console.log(`Latitude: ${body.results[0].geometry.location.lat}`);
});
```

Now that we have this in place, we'll do the exact same thing for longitude. We'll add another `console.log` statement in the next line of the code. We'll use template strings once again, typing longitude first. After that, we'll put a colon and then inject the value. In this case, the value is on the body; it's in that same results item, the first one. We'll go into geometry location again. Instead of `.lat`, we'll access `.lng`. Then we can add a semicolon at the end and save the file. This will look something like the following:

```
  console.log(`Address: ${body.results[0].formatted_address}`);
  console.log(`Latitude: ${body.results[0].geometry.location.lat}`);
  console.log(`Longitude: ${body.results[0].geometry.location.lng}`);
});
```

Now we'll test it from the Terminal. We'll rerun the previous command, and as shown in the following screenshot, you can see we have the latitude, `39.94`, and the longitude, `-75.16` printing to the screen:



And these are the exact same values we have inside the Chrome browser, `39.94`, `-75.16`. With this in place, we've now successfully pulled off the data we need to make that request to the weather API.

Summary

In this chapter, we have gone through a basic example of asynchronous programming. Next, we talked about what happens behind the scenes when you run asynchronous code. We got a really good idea about how your program runs and what tools and tricks are happening behind the scenes to make it run the way it does. We through a few examples that illustrate how the Call Stack, Node APIs, the Callback Queue, and the Event Loop work.

Then, we learned how to use the request module to make an HTTP request for some information, the URL we requested was a Google Maps Geocoding URL, and we passed in the address we want the latitude and the longitude for. Then we used a callback function that got fired once that data came back.

At the end of the section *Callback functions and APIs*, we looked into a quick tip on how we can format objects when we want to print them to the console. Last, we looked into what makes up the HTTPS request.

In the next chapter, we'll add some error handling to this callback because that's going to be really important for our HTTP requests. There's a chance that things will go wrong, and when they do, we'll want to handle that error by printing a nice error message to the screen.

Callbacks in Asynchronous Programming

This chapter is the second part of our asynchronous programming in Node.js. In this chapter, we'll look at callbacks, HTTP requests, and more. We're going to handle a lot of the errors that happen inside callbacks. There's a lot of ways our request in `app.js` can go wrong, and we'll want to figure out how to recover from errors inside of our callback functions when we're doing asynchronous programming.

Next, we'll be moving our request code block into a separate file and abstracting a lot of details. We'll talk about what that means and why it's important for us. We'll be using Google's Geolocation API, and we'll be using the Dark Sky API to take location information like a zip code and turn that into real-world current weather information.

Then, we'll start wiring up that forecast API, fetching real-time weather data for the address that's geocoded. We'll add our request inside of the callback for `geocodeAddress`. This will let us take that dynamic set of latitude and longitude coordinates, the `lat`/`lng` for the address used in the arguments list, and fetch the weather for that location.

Specifically, we'll look into the following topics:

- Encoding user input
- Callback errors
- Abstracting callbacks
- Wiring up weather search
- Chaining callbacks together

Encoding user input

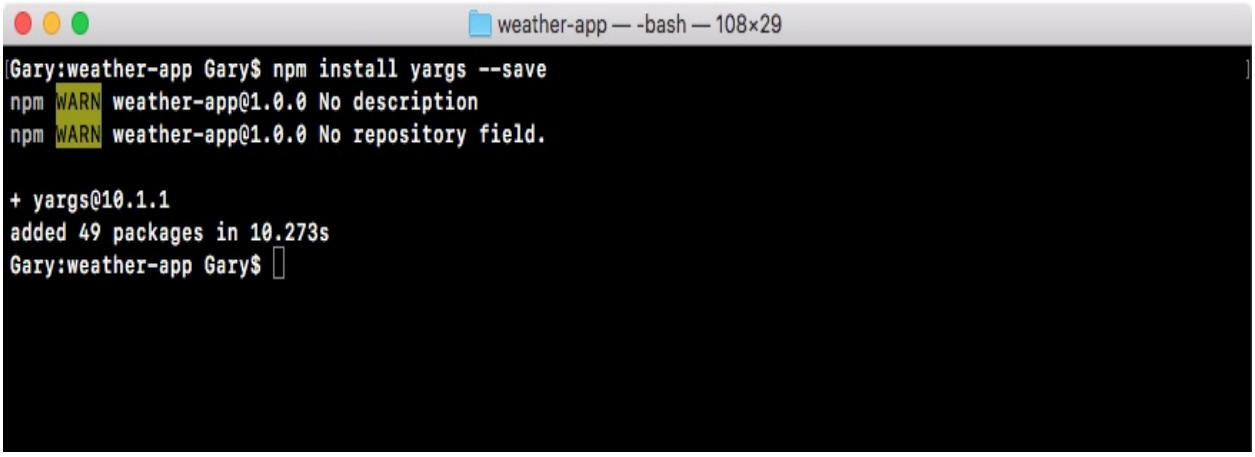
In this section, you'll learn how to set up yargs for the weather app. You'll also learn how to include user input, which is very important for our application.

As shown in the previous chapter, *HTTPS request* section, the user will not type their encoded address into the Terminal; instead they will be typing in a plain text address like `1301 Lombard Street`.

Now this will not work for our URL, we need to encode those special characters, like the space, replacing them with `%20`. Now `%20` is the special character for the space, other special characters have different encoding values. We'll learn how to encode and decode strings, so we can set up our URL to be dynamic. It's going to be based off of the address provided in the Terminal. That's all we're going to discuss in this section. By the end of the section, you'll be able to type in any address you like, and you'll see the formatted address, the latitude, and the longitude.

Installing yargs

Before we can get started doing any encoding, we have to get the address from the user, and before we can set up yargs we have to install it. In the Terminal, we'll run the `npm install` command, the module name is `yargs`, and we'll look for version 10.1.1, which is the latest version at the time of writing. We'll use the `save` flag to run this installation, as shown in the following screenshot:



Now the `save` flag is great because as you remember. It updates the `package.json` file and that's exactly what we want. This means that we can get rid of the node modules folder which takes up a ton of space, but we can always regenerate it using `npm install`.

If you run `npm install` without anything else, no other module names or flags. It will dig through that `package.json` file looking for all the modules to install, and it will install them, recreating your node modules folder exactly as you left it.

While the installation is going on, we do a bit of configuration in the `app.js` file. So we can get started by first loading in yargs. For this, in the `app.js` file, next to request constant, I'll make a constant called `yargs`, setting it equal to `require(yargs)` just like this:

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

Now we can go ahead and actually do that configuration. Next we'll make another constant called `argv`. This will be the object that stores the final parsed output. That will take the input from the process variable, pass it through `yargs`, and the result will be right here in the `argv` constant. This will get set equal to `yargs`, and we can start adding some calls:

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

const argv = yargs
```

Now when we created the notes app we had various commands, you could add a note and that required some arguments, list a note which required just the title, list all notes which didn't require any arguments, and we specified all of that inside of `yargs`.

For the weather app the configuration will be a lot simpler. There is no command, the only command would be get weather, but if we only have one why even make someone type it. In our case, when a user wants to fetch the weather all they will do is type `node app.js` followed by the `address` flag just like this:

```
node app.js --address
```

Then they can type their address inside of quotes. In my case it could be something like `1301 lombard street`:

```
node app.js --address '1301 lombard street'
```

This is exactly how the command will get executed. There's no need for an actual command like fetch weather, we go right from the file name right into our arguments.

Configuring yargs

To configure yargs, things will look a little different but still pretty similar. In the Atom, I'll get started by calling `.options`, which will let us configure some top level options. In our case, we'll pass in an object where we configure all of the options we need. Now I'll format this like I do for all of my chained calls, where I move the call to the next line and I indent it like this:

```
const argv = yargs
  .options({

})
```

Now we can set up our options and in this case we just have one, it will be that `a` option; `a` will be short for address. I could either type address in the options and I could put `a` in the alias, or I could put `a` in the options and type address in the alias. In this case I'll put a as shown here:

```
const argv = yargs
  .options({
    a: {

    }
  })
```

Next up, I can go ahead and provide that empty object, and we'll go through these same exact options we used inside of the notes app. We will demand it. If you'll fetch the weather we need an address to fetch the weather for, so I'll set `demand` equal to `true`:

```
const argv = yargs
  .options({
    a: {
       demand: true,
    }
  })
```

Next up, we can set an `alias`, I'll set `alias` equal to `address`. Then finally we'll set `describe`, we can set `describe` to anything we think would be useful, in this case I'll go with `Address to fetch weather for`, as shown here:

```
const argv = yargs
  .options({
    a: {
```

```
      demand: true,
      alias: 'address',
      describe: 'Address to fetch weather for'
    }
  })
```

Now these are the three options we provided for the notes app, but I'll add a fourth one to make our yargs configuration for the weather app even more full proof. This will be an option called `string`. Now `string` takes a Boolean either `true` or `false`. In our case we want `true` to be the value. This tells `yargs` to always parse the `a` or `address` argument as a string, as opposed to something else like a number or a Boolean:

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

In the Terminal, if I were to delete the actual string `address`, `yargs` would still accept this, it would just think I'm trying to add a Boolean flag, which could be useful in some situations. For example, do I want to fetch in Celsius or in Fahrenheit? But in our case, we don't need any sort of `true` or `false` flag, we need some data, so we'll set string to true to make sure we get that data.

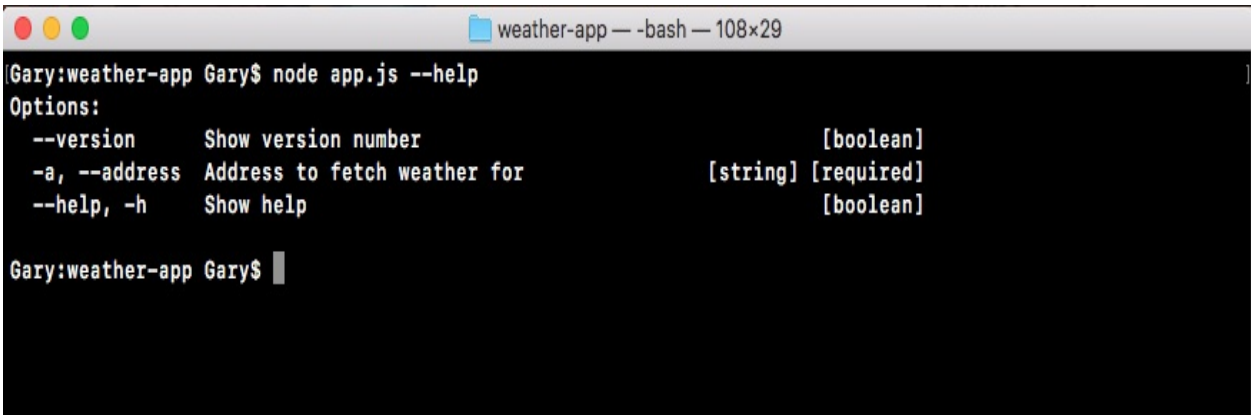
Now that we have our options configuration in place, we can go ahead and add a couple other calls that we've explored. I'll add `.help`, calling it as shown in the following code, which adds the `help` flag. This is really useful especially when someone is first using a command. Then we can access `.argv`, which takes all of this configuration, runs it through our arguments, and restores the result in the `argv` variable:

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

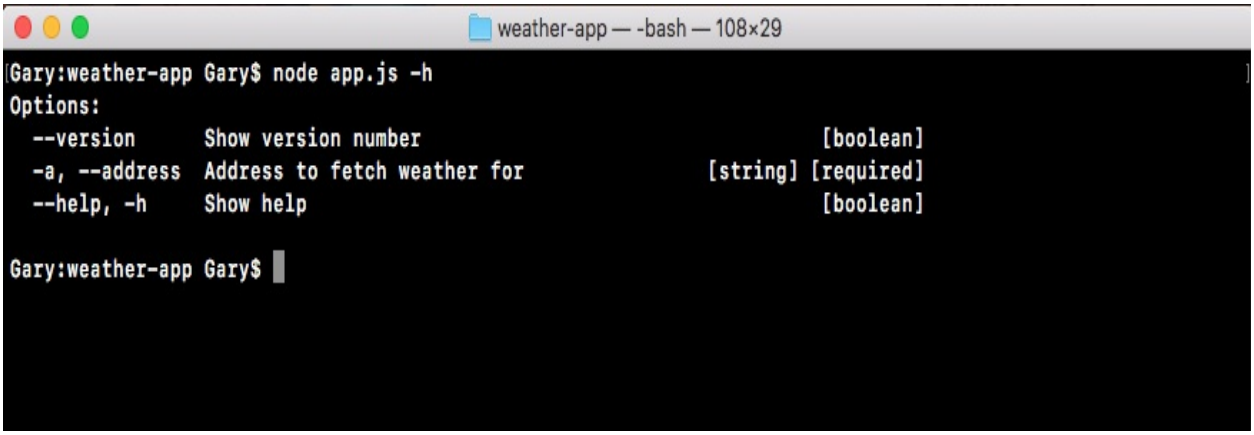
Now the `help` method adds that `help` argument, we can also add an alias for it right afterwards by calling `.alias`. Now `.alias` takes two arguments, the actual argument that you want to set an alias for and the alias. In our case, we already have `help` registered, it gets registered when we call `help`, and we'll set an alias which will just be the letter `h`, awesome:

```
.help()
.alias('help', 'h')
.argv;
```

Now we have all sorts of really great configurations set up for the weather app. For example, inside the Terminal I can now run `help`, and I can see all of the help information for this application:



I could also use the shortcut `-h`, and I get the exact same data back:



Printing the address to screen

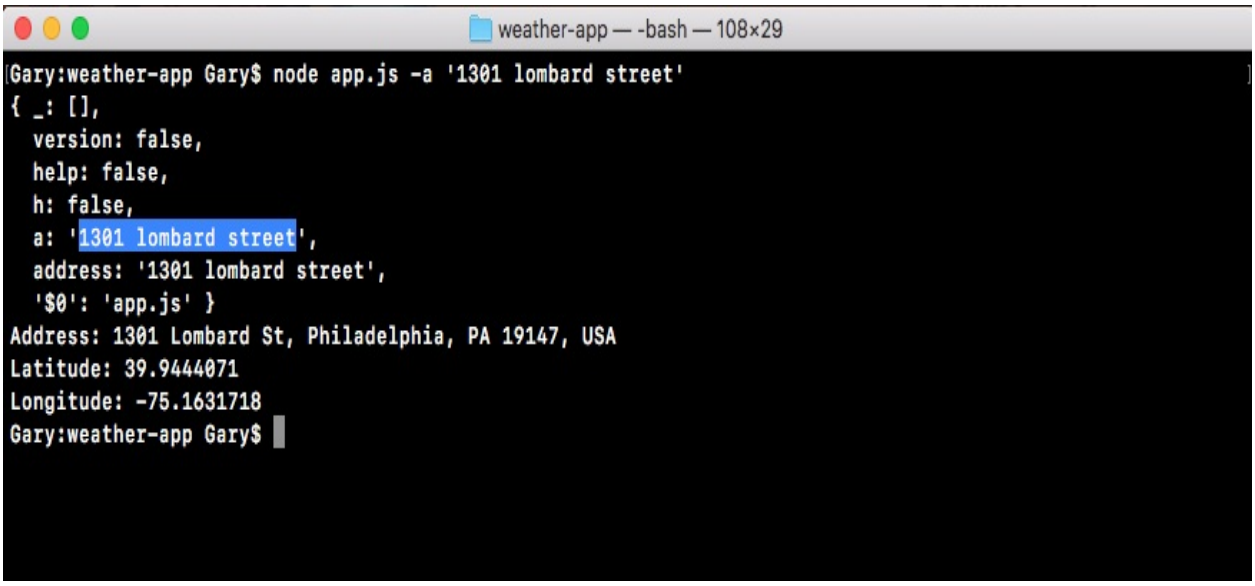
Now the address is also getting passed through but we don't print it to the screen, so let's do that. Right after the configuration, let's use `console.log` to print the entire `argv` variable to the screen. This will include everything that got parsed by `yargs`:

```
  .help()
  .alias('help', 'h')
  .argv;
console.log(argv);
```

Let's go ahead and rerun it in the Terminal, this time passing in an address. I'll use the `a` flag, and specifying something like `1301 lombard street`, closing the quotes, and hitting *enter*:

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

When we do this we get our object, and as shown in the code output, we have 1301 Lombard St, Philadelphia, PA 19147, USA, the plain text address:



In the preceding screenshot, notice that we happen to fetch the latitude and longitude for that address, but that's just because we have it hard coded in the URL in `app.js`. We still need to make some changes in order to get the address,

the one that got typed inside the argument, to be the address that shows up in the URL.

Encoding and decoding the strings

To explore how to encode and decode strings we'll head into the Terminal. Inside the Terminal, first we'll clear the screen using the `clear` command, and then we boot up a node process by typing the `node` command as shown:

```
node
```

Here we can run any statements we like. When we're exploring a really basic node or JavaScript feature, we'll look into some examples first, and then we go ahead and add it into our actual application. We'll look at two functions, `encodeURIComponent` and `decodeURIComponent`. We'll get started with encoding first.

Encoding URI component

Encoding, the method is called `encodeURIComponent`, encode URI in uppercase component, and it takes just one argument, the string you want to encode. In our case, that string will be the address, something like `1301 lombard street philadelphia`. When we run this address through `encodeURIComponent` by hitting *enter*, we get the encoded version back:

```
encodeURIComponent('1301 lombard street philadelphia')
```

As shown in the following code output, we can see all the spaces, like the space between 1301 and lombard, have been replaced with their encoded character, and for the case of the space it is %20. By passing our string through `encodeURIComponent`, we'll create something that's ready to get injected right into the URL so we can fire off that dynamic request.

Decoding URI component

Now the alternative to `encodeURIComponent` is. This will take an encoded string like the one in the previous example, and take all the special characters, like `%20`, and convert them back into their original values, in this case space. For this, inside of `decodeURIComponent` once again we'll pass a string.

Let's go ahead and type our first and last name. In my case it's `Andrew`, and instead of a space between them I'll add `%20`, which we know is the encoded character for a space. Since we're trying to decode something, it's important to have some encoded characters here. Once yours looks like the following code with your first and last name, you can go ahead and hit *enter*, and what we get back is the decoded version:

```
decodeURIComponent('Andrew%20Mead')
```

As shown in the following code output, I have Andrew Mead with the `%20` being replaced by the space, exactly what we expected. This is how we can encode and decode URI components in our app:

Pulling the address out of argv

Now what we want to do is pull the address out of `argv`, we already saw that it's there, we want to encode it and we want to inject it in our URL in `app.js` file, replacing the address:

```
20  maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20street%20philadelphia',
```

This will essentially create that dynamic request we've been talking about. We'll be able to type in any address we want, whether it's an address or a zip code or a city state combination, and we'll be able to fetch the formatted address, the latitude, and the longitude.

In order to get started, the first thing I'll do is get the encoded address. Let's make a variable called `encodedAddress` in the `app.js` next to the `argv` variable, where we can store that result. We'll set this equal to the return value from the method we just explored in the Terminal, `encodeURIComponent`. This will take the plain text address and return the encoded result.

Now we do need to pass in the string, and we have that available on `argv.address` which is the alias:

```
  .help()
  .alias('help', 'h')
  .argv;
var encodedAddress = encodeURIComponent(argv.address);
```

Here we could use `argv.a` as well as `argv.address`, both will work the same.

Now we have that encoded result all that's left to do is inject it inside of the URL string. In the `app.js`, currently we're using a regular string. We'll swap this out for a template string so I can inject a variable inside of it.

Now that we have a template string, we can highlight the static address which ends at `philadelphia` and goes up to the `=` sign, and remove it, and instead of typing in a static address we can inject the dynamic variable. Inside of my curly braces,

`encodedAddress`, as shown here:

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

request({
  url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,
```

With this in place we are now done. We get the address from the Terminal, we encode it, and we use that inside of a `geocode` call. So the formatted address, latitude, and longitude should match up. Inside the Terminal, we'll shut down node by using *control* + *C* twice and use clear to clear the Terminal output.

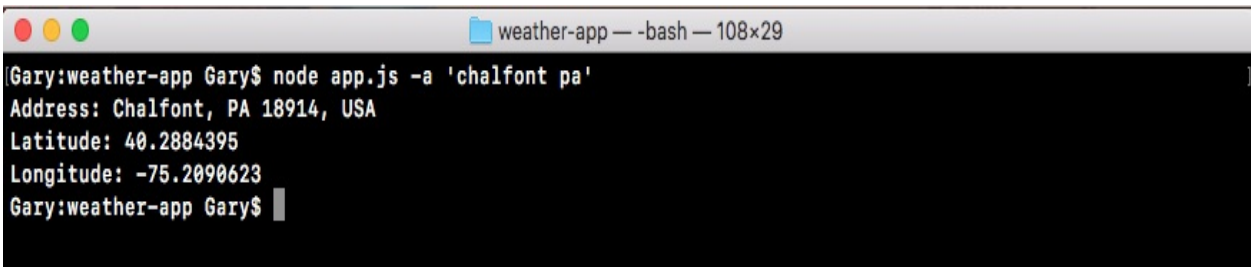
Then we can go ahead and run our app using `node app.js`, passing in either the `a` or `address` flag. In this case, we'll just use `a`. Then we can go ahead and type in an address, for example, `1614 south broad street philadelphia` as shown here:

```
node app.js -a '1614 south broad street philadelphia'
```

When you run it you should have that small delay while we fetch the data from the geocode URL.

In this case we'll find that it's actually taking a little longer than we would expect, about three or four seconds, but we do get the address back:

Here we have the formatted address with a proper zip code state and country, and we also have the latitude and longitude showing up. We'll try a few other examples. For example for a town in Pennsylvania called Chalfont, we can type in `chalfont pa` which is not a complete address, but the Google Geocode API will convert it into the closest thing, as shown here:

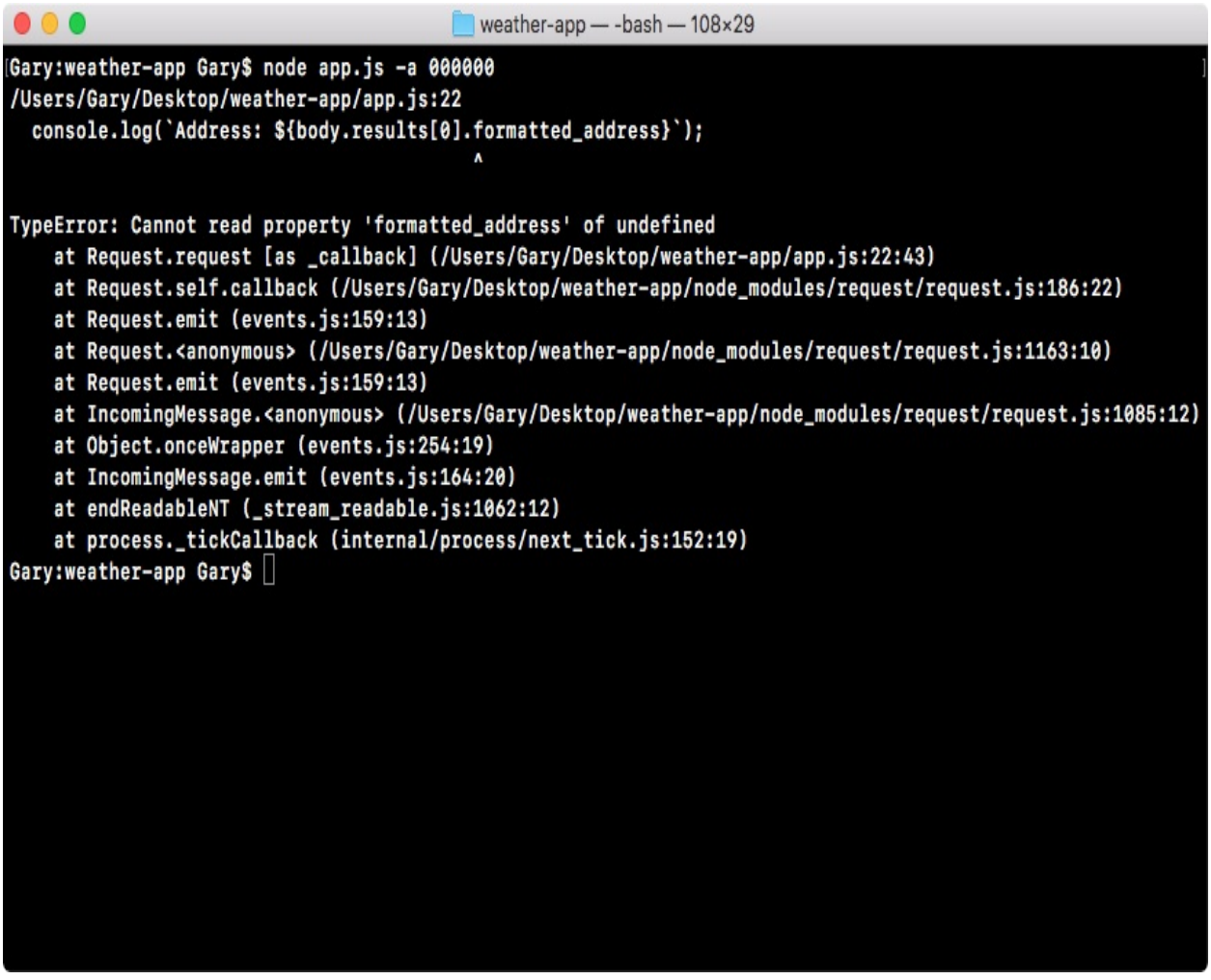


We can see that it's essentially the address of the town, Chalfont, PA 18914 is the zip, with the state USA. Next, we have the general latitude and longitude data for that town, and this will be fine for fetching weather data. The weather isn't exactly changing when you move a few blocks over.

Now that we have our data coming in dynamically, we are able to move on to the next section where we'll handle a lot of the errors that happen inside of callbacks. There are a lot of ways this request can go wrong, and we'll want to figure out how to recover from errors inside of our callback functions when we're doing asynchronous programming.

Callback errors

In this section we'll learn how to handle errors inside of your callback functions, because as you might guess things don't always go as planned. For example, the current version of our app has a few really big flaws, if I try to fetch weather using `node app.js` with the `a` flag for a zip that doesn't exist, like `000000`, the program crashes, which is a really big problem. It's going off. It's fetching the data, eventually that data will come back and we get an error, as shown here:



It's trying to fetch properties that don't exist, such as `body.results[0].formatted_address` is not a real property, and this is a big problem.

Our current callback expects everything went as planned. It doesn't care about the error object, doesn't look at response codes; it just starts printing the data that it wants. This is the happy path, but in real world node apps we have to handle errors as well otherwise the applications will become really useless, and a user can get super frustrated when things don't seem to be working as expected.

In order to do this, we'll add a set of `if`/`else` statements inside of the callback. This will let us check certain properties to determine whether or not this call, the one to our URL in the `app.js`, should be considered a success or a failure. For example, if the response code is a 404, we might want to consider that a failure and we'll want to do something other than trying to print the address, latitude and longitude. If everything went well though, this is a perfectly reasonable thing to do.

There are two types of errors that we'll worry about in this section. That will be:

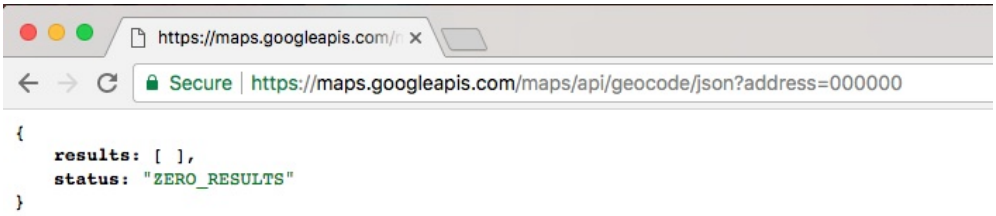
- The machine errors, things like being unable to connect to a network, these are usually will show up in the error object, and
- The errors coming from the other server, the Google server, this could be something like an invalid address

In order to get started, let's take a look at what can happen when we pass a bad data to the Google API.

Checking error in Google API request

To view what actually comes back in a call like the previous example, where we have an invalid address, we'll head over to the browser and pull up the URL we used in the `app.js` file:

We will remove the address we used earlier from the browser history, and type in `000000`, hit *enter*:



We get our results arrive but those are no results, and we have the status, the status says `ZERO_RESULTS`, and this is the kind of information that's really important to track down. We can use the status text value to determine whether or not the request was successful. If we pass in a real zip code like `19147`, which is `Philadelphia`, we'll get our results back, and as shown in the following image, the `status` will get set equal to `OK`:

We can use this status to determine that things went well. Between these status property and the error object, which we have inside of our app, we can determine what exactly to do inside of the callback.

Adding the if statement for callback errors

The first thing we'll do is add an `if` statement as shown below, checking if the error object exists:

```
request({
  url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,
  json: true
}, (error, response, body) => {
  if (error) {

  }
```

This will run the code inside of our code block if the error object exists, if it doesn't fine, we'll move on into the next `else if` statement, if there is any.

If there is an error, all we'll do is add a `console.log` and a message to the screen, something like `Unable to connect to Google servers`:

```
if (error) {
  console.log('Unable to connect Google servers.');
}
```

This will let the user know that we were unable to connect to the user servers, not that something went wrong with their data, like the address was invalid. This is what be inside of the error object.

Now the next thing that we'll do is add an `else if` statement, and inside of the condition we'll check the status property. If the status property is `ZERO_RESULTS`, which it was for the zip code `000000`, we want to do something other than trying to print the address. Inside of our conditional in Atom, we can check that using the following statement:

```
if (error) {
  console.log('Unable to connect Google servers.');
} else if (body.status === 'ZERO_RESULTS') {

}
```

If that's the case, we'll print a different message, other than `Unable to connect Google`

`servers`, for this one we can use `console.log` to print `Unable to find that address.`:

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

This lets the user know that it wasn't a problem with the connection, we were just unable to find the address they provided, and they should try with something else.

Now we have error handling for those system errors, like being unable to connect to the Google servers, and for errors with the input, in this case we're unable to find a location for that address, and this is fantastic, we have both of our errors handled.

Now the `body.status` property that shows up in the `else if` statement, is not going to be on every API, this is specific to the Google Geocode API. When you explore a new API it's important to try out all sorts of data, good data like a real address and bad data like an invalid zip code, to see exactly what properties you can use to determine whether or not the request was successful, or if it failed.

In our case, if the status is `ZERO_RESULTS`, we know the request failed and we can act accordingly. Inside of our `app`, now we'll add our last `else if` clause, if things went well.

Adding if else statement to check body status property

Now we want to add the `else if` clause checking if the `body.status` property equals `OK`. If it does, we can go ahead and run these three lines inside of the code block:

```
  console.log(`Address: ${body.results[0].formatted_address}`);
  console.log(`Latitude: ${body.results[0].geometry.location.lat}`);
  console.log(`Longitude: ${body.results[0].geometry.location.lng}`);
});
```

If it doesn't, these lines shouldn't run because the code block will not execute. Then we'll test things out inside of the Terminal, try to fetch the address of `00000`, and make sure that instead of the program crashing we get our error message printing to the screen. Then we go ahead and mess up the URL in the app by removing some of the important characters, and make sure this time we get the `Unable to connect to the Google servers.` message. And last we'll see what happens when we enter a valid address, and make sure our three `console.log` statements still execute.

To get started we'll add that `else if` statement, and inside of the condition we'll check if `body.status` is `OK`:

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

}
```

If it is `OK`, then we'll simply take the three `console.log` lines (shown in the previous code block) and move them in the `else if` condition. If it is `OK`, we'll run these three `console.log` statements:

```
if (error) {
  console.log('Unable to connect Google servers.');
} else if (body.status === 'ZERO_RESULTS') {
  console.log('Unable to find that address.');
} else if (body.status === 'OK') {
  console.log(`Address: ${body.results[0].formatted_address}`);
  console.log(`Latitude: ${body.results[0].geometry.location.lat}`);
  console.log(`Longitude: ${body.results[0].geometry.location.lng}`);
```

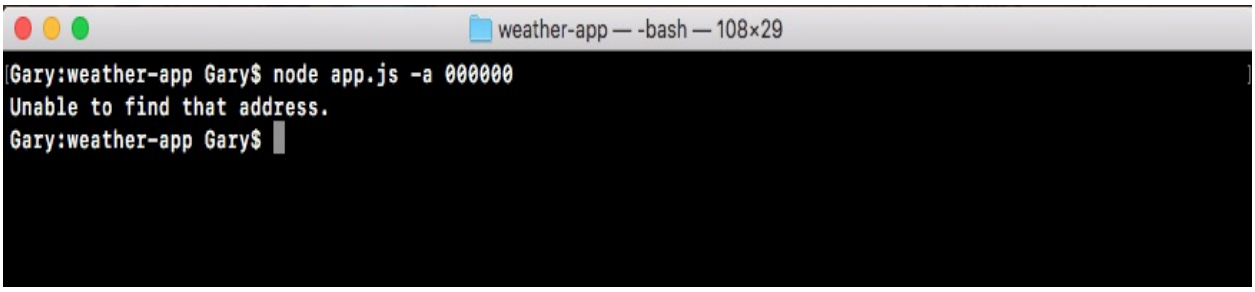
```
}
```

Now we have a request that handles errors really well. If anything goes wrong we have a special message for it, and if things go right we print exactly what the user expects, the address, the latitude, and the longitude. Next we'll test this.

Testing the body status property

To test this inside of the Terminal, we'll start by rerunning the command with an address that's invalid:

```
node app.js -a 000000
```

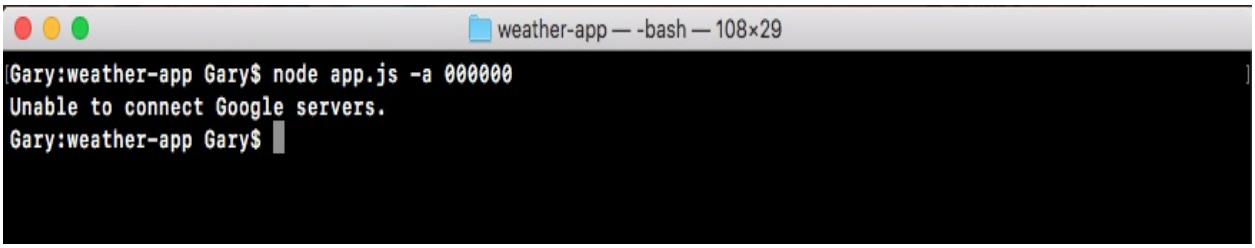


When we run this command, we see that Unable to find address. prints to the screen. Instead of the program crashing, printing a bunch of errors, we simply have a little message printing to the screen. This is because the code we have in second `else if` statement, that tried to access those properties that didn't exist, no longer runs because our first `else if` condition gets caught and we simply print the message to the screen.

Now we also want to test that the first message (`Unable to connect to the Google servers.`) prints when it should. For this, we'll delete some part of the URl in our code, let's say, `s` and ., and save the file:

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

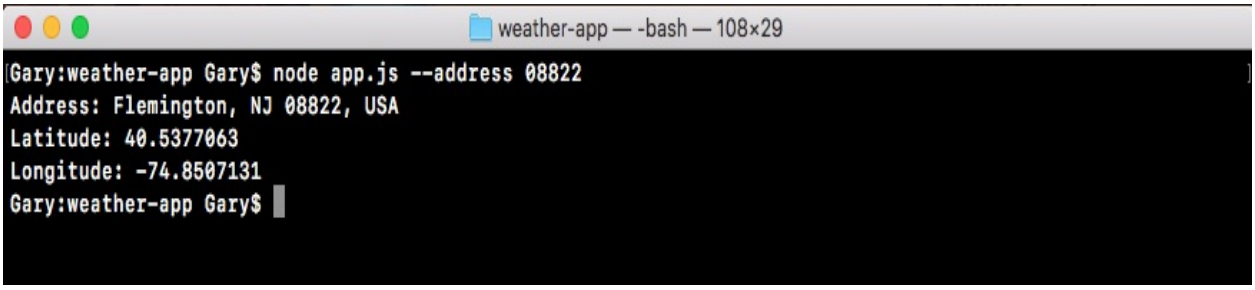
Then we'll rerun the previous command in the Terminal. This time around we can see `Unable to connect to Google servers.` prints to the screen just like it should:



Now we can test it the final thing, by first readjusting the URL to make it correct, and then fetching a valid address from the Terminal. For example, we can use the `node app.js`, setting `address` equal to `08822`, which is a zip code in New Jersey:

```
node app.js --address 08822
```

When we run this command, we do indeed get our formatted address for Flemington, NJ, with a zip code and the state, and we have our latitude and longitude as shown here:



We now have a complete error handling model. When we make a request to Google providing a address that has problems, in this case there's `ZERO_RESULTS`, the error object will get populated, because it's not technically an error in terms of what request thinks an error is, it's actually in the response object, which is why we have to use `body.status` in order to check the error.

That is it for this section, we now have error handling in place, we handle system errors, Google server errors, and we have our success case.

Abstracting callbacks

In this section, we'll be refactoring `app.js`, taking a lot of the complex logic related to geocoding and moving it into a separate file. Currently, all of the logic for making the request and determining whether or not the request succeeded, our `if else` statements, live inside of `app.js`:

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

This is not exactly reusable and it really doesn't belong here. What I'd like to do before we add even more logic related to fetching the forecast, that's the topic of the next section, is break this out into its own function. This function will live in a separate file, like we did for the notes application.

In the `notes app` we had a separate file that had functions for adding, listing, and removing notes from our local adjacent file. We'll be creating a separate function responsible for geocoding a given address. Although the logic will stay the same, there really is no way around it, it will be abstracted out of the `app.js` file and into its own location.

Refactoring app.js and code into geocode.js file

First up, we will need to create a new directory and a new file then we'll add a few more advanced features to the function. But before that, we'll see what the require statement will look like.

Working on request statement

We'll load in via a constant variable called `geocode` the module, and we'll set it equal to `require`, since we're requiring a local file we'll add that relative path, `./geocode/geocode.js`:

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

That means you need to make a directory called `geocode` in the `weather-app` folder, and a file called `geocode.js`. Since we have a `.js` extension, we can actually leave it off of our require call.

Now, in the `app.js` file, next to `.argv` object, we need to call `geocode.geocodeAddress`. The `geocodeAddress` function, that will be the function responsible for all the logic we currently have in `app.js`. The `geocodeAddress` function will take the address, `argv.address`:

```
geocode.geocodeAddress(argv.address);
```

It will be responsible for doing everything, encoding the URL, making the request, and handling all of the error cases. This means, in that new file we need to export the `geocodeAddress` function, just like we exported functions from the `notes application` file. Next, we have all of the logic here:

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

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

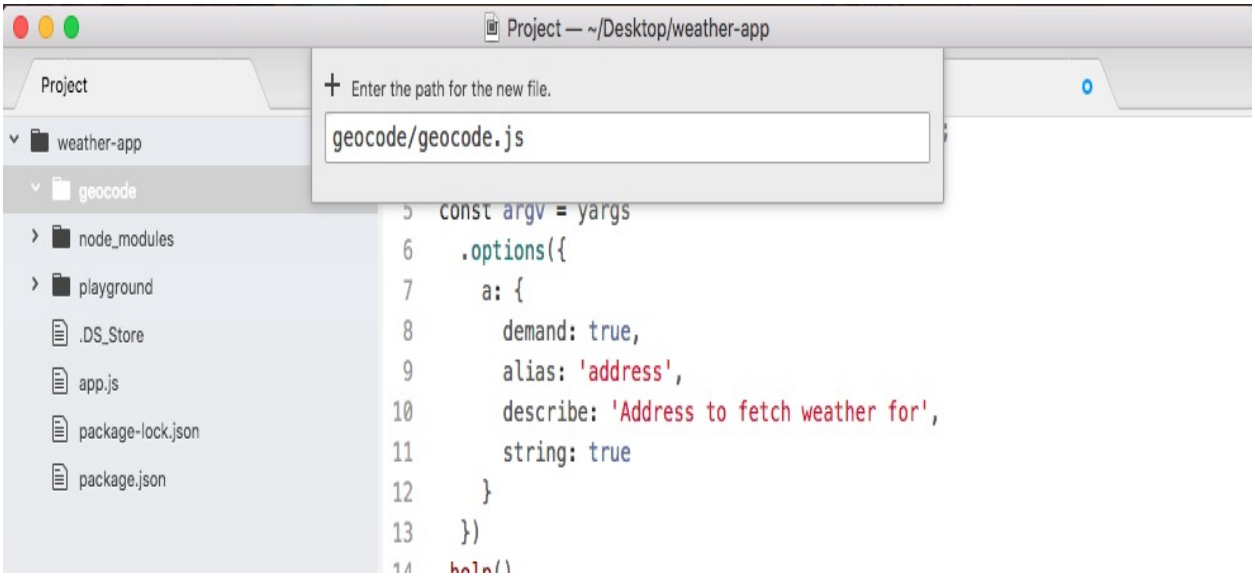
This logic needs to get moved inside of the `geocodeAddress` function. Now we can copy and paste the preceding shown code directly, there really is no way around

some of the more complex logic, but we will need to make a few changes. We'll need to load requests into that new file, since we use it and it isn't going to be required in that file by default. Then we can go ahead and clean up the request require call in the code, since we won't be using it in this file.

Next up, the `argv` object is not going to exist, we'll get that passed in via the first argument, just like the `argv.address` in the `geocode.Address` statement. This means we'll need to swap this out for whatever we call that first argument for example, address. Once this is done, the program should work exactly as it works without any changes in `app.js`, there should be no change in functionality.

Creating geocode file

To get started, let's make a brand new directory in the `weather-app` folder, that's the first thing we need to do. The directory is called `geocode`, which aligns with the require statement we have in the `geocode` variable. In `geocode` folder, we'll make our file `geocode.js`:



Now inside of `geocode.js`, we can get started by loading in request, let's make a constant called `request`, and we'll set it equal to `require('request')`:

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

Now we can go ahead and define the function responsible for geocoding, this one will be called `geocodeAddress`. We'll make a variable called `geocodeAddress`, setting it equal to an arrow function, and this arrow function will get an `address` argument past in:

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

};
```

This is the plain text unencoded address. Now before we copy the code from `app.js` into this function body, we want to export our `geocodeAddress` function using `module.exports`, which we know as an object. Anything we put on `module.exports`

object will be available to any files that require this file. In our case, we want to make a `geocodeAddress` property available, setting it equal to the `geocodeAddress` function that we defined in the preceding statement:

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

};

module.exports.geocodeAddress = geocodeAddress;
```

Now it's time to actually copy all of the code from `app.js` in to `geocode.js`. We'll cut the request function code, move in to `geocode.js`, and paste it inside of the body of our function:

```
var geocodeAddress = (address) => {
  var encodedAddress = encodedURIComponent(argv.address);

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

module.exports.geocodeAddress = geocodeAddress;
```

The only thing we need to change inside of this code, is how we get the plaintext address. We no longer have that `argv` object, instead we get `address` passed in as an argument. The final code will look like the following code block:

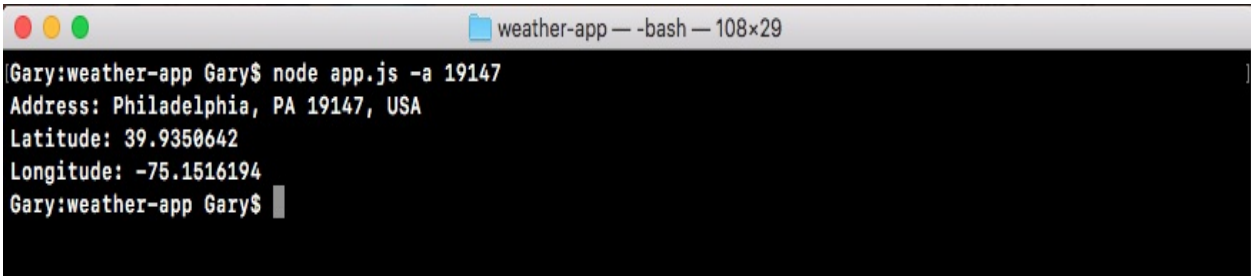
```
const request = require('request');

var geocodeAddress = (address) => {
  var encodedAddress = encodedURIComponent(argv.address);

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

```
      console.log(`Longitude: ${body.results[0].geometry.location.lng}`);
    }
  });
};

module.exports.geocodeAddress = geocodeAddress;
```

With this in place, we're now done with the `geocode` file. It contains all of the complex logic for making and finishing the request. Over at `app.js`, we can clean things up by removing some extra spaces, and removing the request module which is no longer used in this file. The final `app.js` file will look like the following code block:

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

const geocode = require('./geocode/geocode');

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

geocode.geocodeAddress(argv.address);
```

Now at this point the functionality should be exactly the same. Inside of the Terminal, I'll go ahead and run a few to confirm the changes worked. We'll use the `a` flag to search for a zip code that does exist, something like `19147`, and as shown, we can see the address, the latitude, and the longitude:



Now we'll swap out that zip code to one that does not exist, like `000000`, when we run this through the geocoder, you can see Unable to find address prints to screen:

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

It means all of the logic inside of `geocode.js` is still working. Now the next step in the process is the process of adding a callback function to `geocodeAddress`.

Adding callback function to geocodeAddress

The goal of refactoring the code and `app.js` was not to get rid of the callback, the goal was to abstract all the complex logic related to encoding the data, making that request, and checking for errors. `app.js` should not care about any of that, it doesn't even need to know that an HTTP request was ever made. All the `app.js` should care about is passing an address to the function, and doing something with the result. The result being either an error message or the data, the formatted address, the latitude, and the longitude.

Setting up the function in geocodeAddress function in app.js

Before we go ahead and make any changes in `geocode.js`, we want to take a look at how we'll structure things inside of `app.js`. We'll pass an arrow function to `geocodeAddress`, and this will get called after the request comes back:

```
geocode.geocodeAddress(argv.address, () => {

});
```

In the parentheses, we'll expect two arguments, `errorMessage`, which will be a string, and `results`, which will contain the address, the latitude, and the longitude:

```
geocode.geocodeAddress(argv.address, (errorMessage, results) => {

});
```

Out of these two only one will be available at a time. If we have an error message we'll not have results, and if we have results we'll not have an error message. This will make the logic in the arrow function, of determining whether or not the call succeeded, much simpler. We'll be able to use an `if` statement, `if (errorMessage)`, and if there is an error message, we can simply print it to the screen using `console.log` statement:

```
geocode.geocodeAddress(argv.address, (errorMessage, results) => {
  if (errorMessage) {
    console.log(errorMessage);
  }
});
```

There's no need to dig into any sort of object and figure out exactly what's going on, all of that logic is abstracted in `geocode.js`. Now if there is no error message inside of the `else` clause, we can go ahead and print the results. We'll use that pretty print method we talked about in the previous chapter, we'll add the `console.log(JSON.stringify)` statement, and we'll pretty print the results object which will be an object containing an address property, a latitude property, and a

longitude property.

Then, we'll pass the `undefined` argument as our second argument. This skips over the filtering function which we don't need, and then we can specify the spacing, which will format this in a really nice way, we'll use two spaces as shown here:

```
geocode.geocodeAddress(argv.address, (errorMessage, results) => {
  if (errorMessage) {
    console.log(errorMessage);
  } else {
    console.log(JSON.stringify(results, undefined, 2));
  }
});
```

Now that we have our function set up inside of `geocodeAddress` function in `app.js`, and we have a good idea about how it will look, we can go ahead and implement it inside of `geocode.js`.

Implementing the callback function in geocode.js file

In our arguments definition, instead of just expecting an address argument we'll also expect a callback argument, and we can call this callback argument whenever we like. We'll call it in three places. We'll call it once inside of the `if (error)` block, instead of calling `console.log` we'll simply call the callback with the `Unable to connect to Google servers.` string. This string will be the error message we defined in `geocodeAddress` function in `app.js`.

In order to do this, all we need to do is change our `console.log` call to a `callback` call. We'll pass it as the first argument our error message. We can take the string exactly as it appeared in `console.log`, and move it into the arguments for `callback`. Then I can remove the `console.log` call and save the file. The resultant code will look like following:

```
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.');
  }
```

Now we can do the exact same thing in the next `else if` block for our other `console.log` statement, when there is `zero results`, we'll replace `console.log` with `callback`:

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

Now the last `else if` block will be a little trickier. It's a little trickier because we don't exactly have our object. We also need to create an `undefined` variable for the first argument, since an error message will not be provided when things go well. All we have to do to create that undefined error message is call `callback`, passing an `undefined` variable as the first argument. Then we can go ahead and specify our object as the second argument, and this object, this will be exactly what's in the

`geocodeAddress` function, results:

```
} else if (body.status === 'OK') {
  callback(undefined, {

  })
  console.log(`Address: ${body.results[0].formatted_address}`);
  console.log(`Latitude: ${body.results[0].geometry.location.lat}`);
  console.log(`Longitude: ${body.results[0].geometry.location.lng}`);
}
```

Now as I mentioned the results have three properties: the first one will be formatted address, so let's go ahead and knock that out first. We'll set `address` equal to `body.results`, just like we have in the `Address` variable of `console.log` statement:

```
} else if (body.status === 'OK') {
  callback(undefined, {
    address: body.results[0].formatted_address
  })
  console.log(`Address: ${body.results[0].formatted_address}`);
  console.log(`Latitude: ${body.results[0].geometry.location.lat}`);
  console.log(`Longitude: ${body.results[0].geometry.location.lng}`);
}
```

Here we're making things even easier, instead of having complex properties that are nested deep inside of an object inside of `app.js`, we'll be able to access a simple `address` property, and we'll do the same thing for `Latitude` and `Longitude` of `console.log` statements.

Next, we'll grab the code that let us fetch the latitude, and I'll add my second property, `latitude`, setting it equal to the code we grab from the `console.log` statement. Then we can go ahead and add the last property, which will be `longitude`, setting that equal to the `latitude` code, replacing `lat` with `lng`. Now that we have this in place we can add a semicolon at the end, and remove the `console.log` statements since they're no longer necessary, and with this we are done:

```
if (error) {
  callback('Unable to connect 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
  });
}
```

We can now rerun the file, and when we do we'll pass an address to `geocodeAddress`, this will go off and make the request, and when the request comes back, we'll be able to handle that response in a really simple way.

Testing the callback function in geocode.js file

Inside of the Terminal, we'll go back to run two `node app.js` commands; the command where we used the zip code of `19147`, everything works as expected and a bad zip code `000000`, to show the error message.

As shown in the following code output, we can see our results object with an address property, a latitude property, and a longitude property:

In case of a bad zip code, we just want to make sure the error message still shows up, and it does, Unable to find that address. prints to the screen, as shown here:

This is happening because of the `if` statement in the `geocodeAddress` function in `app.js`.

After abstracting all of that logic to the `geocode` file, the `app.js` file is now a lot simpler and a lot easier to maintain. We can also call `geocodeAddress` in multiple locations. If we want to reuse the code we don't have to copy and paste the code, which would not follow the **DRY** principle, which stands for **Don't Repeat Yourself**, instead we can do the DRY thing and simply call `geocodeAddress` like we

have in the `app.js` file. With this in place we are now done fetching the `geocode` data.

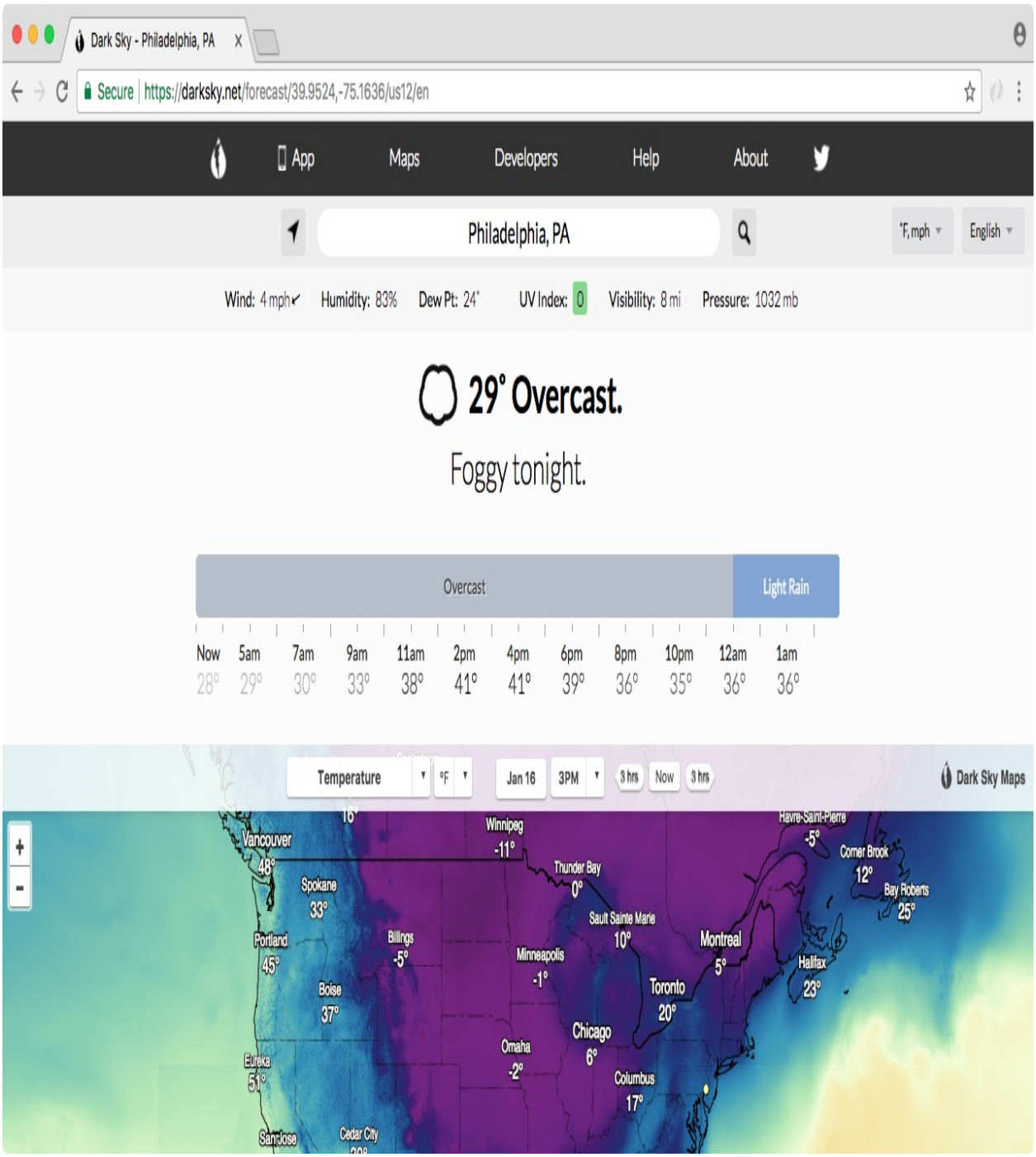
Wiring up weather search

In this section, you'll make your very first request to the weather API, and we'll do this in a static way at first, meaning that it will not use the actual latitude and longitude for the address we passed in, we'll simply have a static URL. We'll make the request and we'll explore what data we get back in the body.

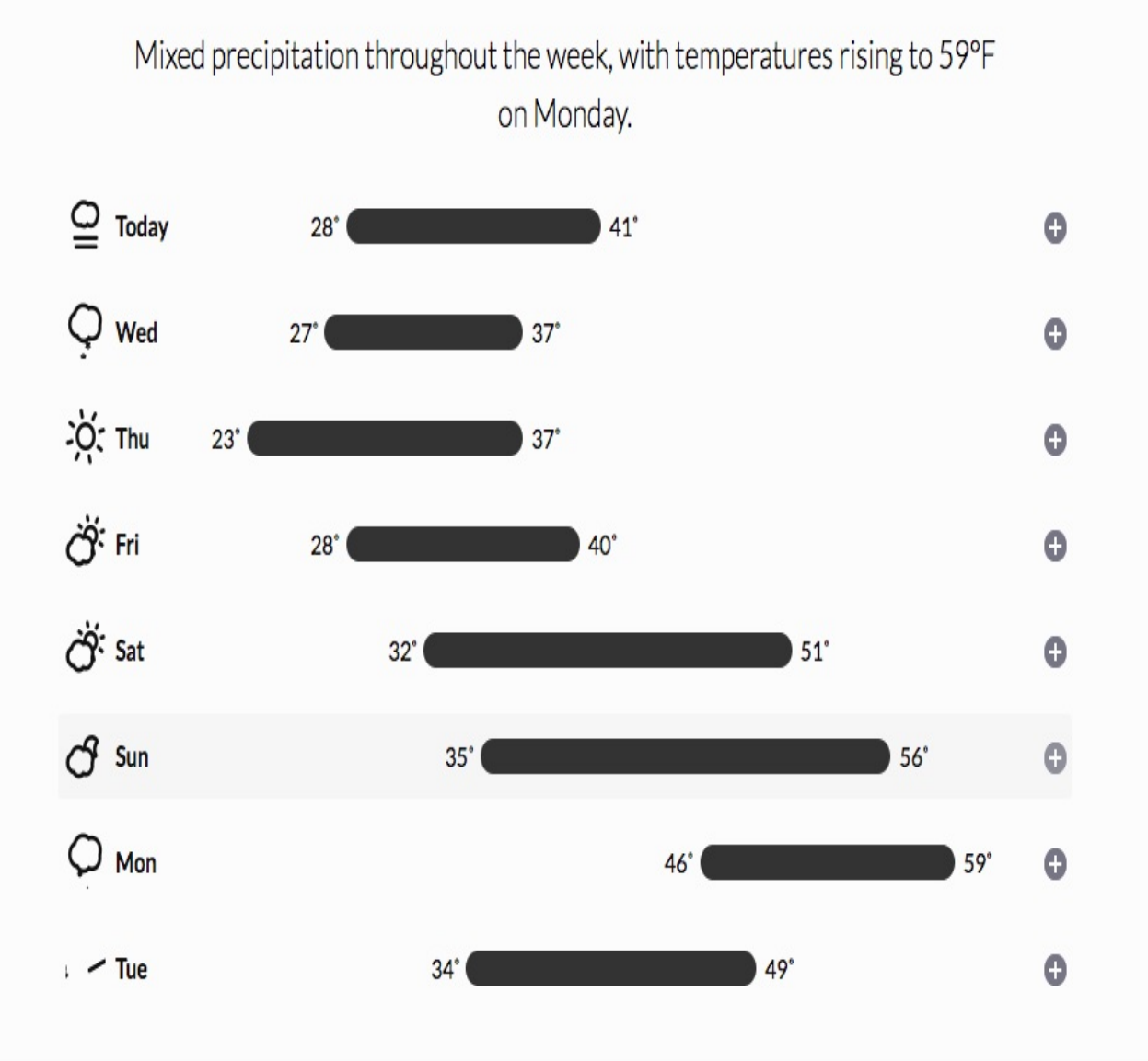
Exploring working of API in the browser

Now before we can add anything to Atom, we want to go ahead and explore this API so we can see how it works in the browser. This will give us a better idea about what weather data we get back, when we pass a latitude and longitude to the API. To do this we'll head over to the browser, and we'll visit a couple of URLs.

First up let's go to `forecast.io`. It is a regular weather website, you type in your location and you get all the weather information you'd expect:

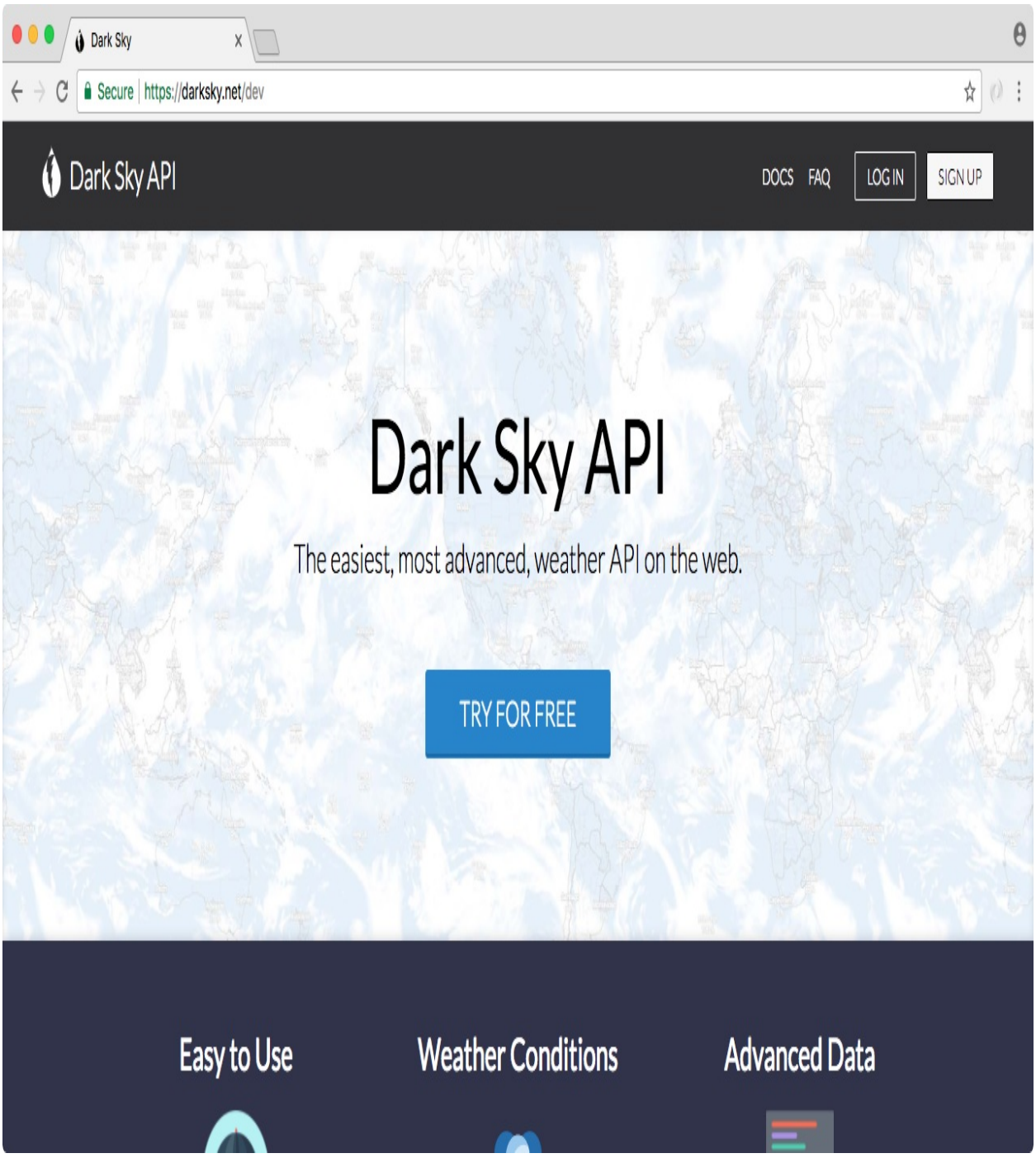


As shown in the preceding image, there's warnings, there's radar, there's the current weather, and we also have the weekly forecast in the website as shown in the following image:



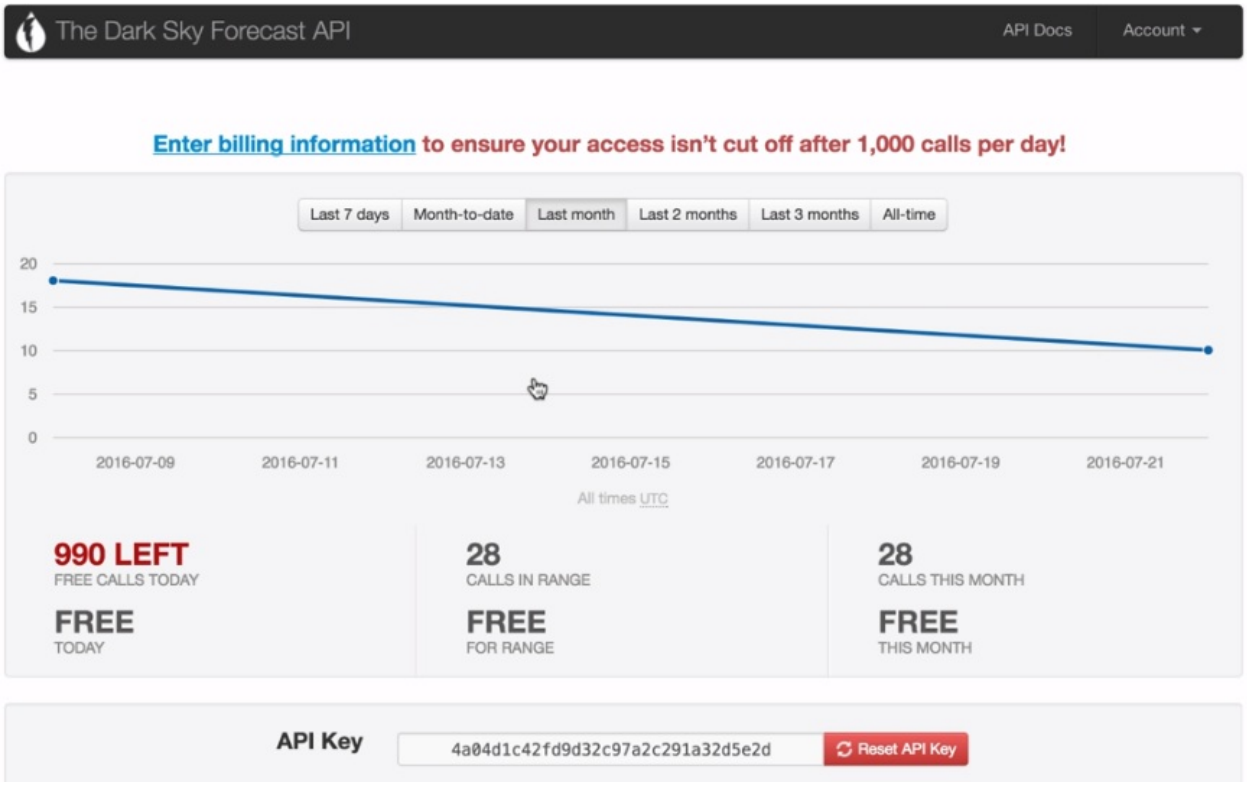
This is similar to `weather.com`, but the one cool thing about `forecast.io` is that the API that powers this website, it's actually available to you as a developer. You can make a request to our URL, and you can fetch the exact same weather information.

That is exactly what we'll do when we can explore the API by going to the website `developer.forecast.io`. Here we can sign up for a free developer account, in order to get started making those weather requests:



The Dark Sky Forecast API gives you 1,000 free requests a day, and I do not see us going over that limit. After the 1,000 requests, each costs a one thousandth of a penny, so you get a thousand requests for every penny you spend. We'll never go over that limit so don't even worry about it. There is no credit card required to get started, you'll simply get cut off after you make a thousand requests.

To get started you'll need to register for a free account, it's really simple, we just need an email and a password. Once we've created an account and we can see the dashboard as shown here:



The only piece of information we need from this page is our API key. The API key is like a password, it will be part of the URL we request and it will help `forecast.io` keep track of how many requests we make a day. Now I'll take this API key and paste it in the `app.js`, so we have it accessible later when we need it.

The next thing we'll do is explore the documentation, the actual URL structure we need to provide in order to fetch the weather for a given latitude and longitude. We can get that by clicking the API Docs link button, which is present in the top-right side of The Dark Sky Forecast API page. This'll lead us to following page:

Dark Sky API – Overview

The Dark Sky API allows you to look up the weather anywhere on the globe, returning (where available):

- Current weather conditions
- Minute-by-minute forecasts out to one hour
- Hour-by-hour and day-by-day forecasts out to seven days
- Hour-by-hour and day-by-day observations going back decades

We provide two types of API requests:

- A Forecast Request returns the current weather forecast for the next week in JSON format.
- A Time Machine Request returns the observed or forecast weather conditions for a date in the past or future in in the same JSON format.

You should also read our Terms of Service, but the short version is:

- The first 1000 API requests you make every day are free of charge.
- Every API request beyond that costs $0.0001.
- You are required to display the message "Powered by Dark Sky" (linking to `https://darksky.net/poweredby/`) somewhere prominent in your app or service. (Details can be found in the terms themselves).

If you have any questions, please check the FAQ and, if you can't find what you're looking for there, send us an email.

API Request Types

Forecast Request

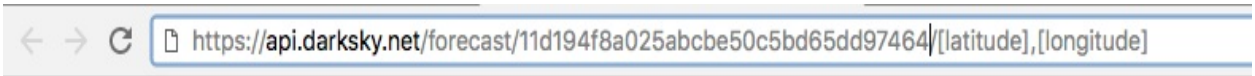
```
https://api.darksky.net/forecast/[key]/[latitude],[longitude]
```

In the API Docs link, we have a Forecast Request URL. As shown in the preceding image, this URL is exactly what we need to make a request to in order to fetch the data.

Exploring the actual URL for code

Before we add this URL into our app and use the request library, we need to find the actual URL which we can use to make the request. For this, we'll copy it and paste it into a new tab:

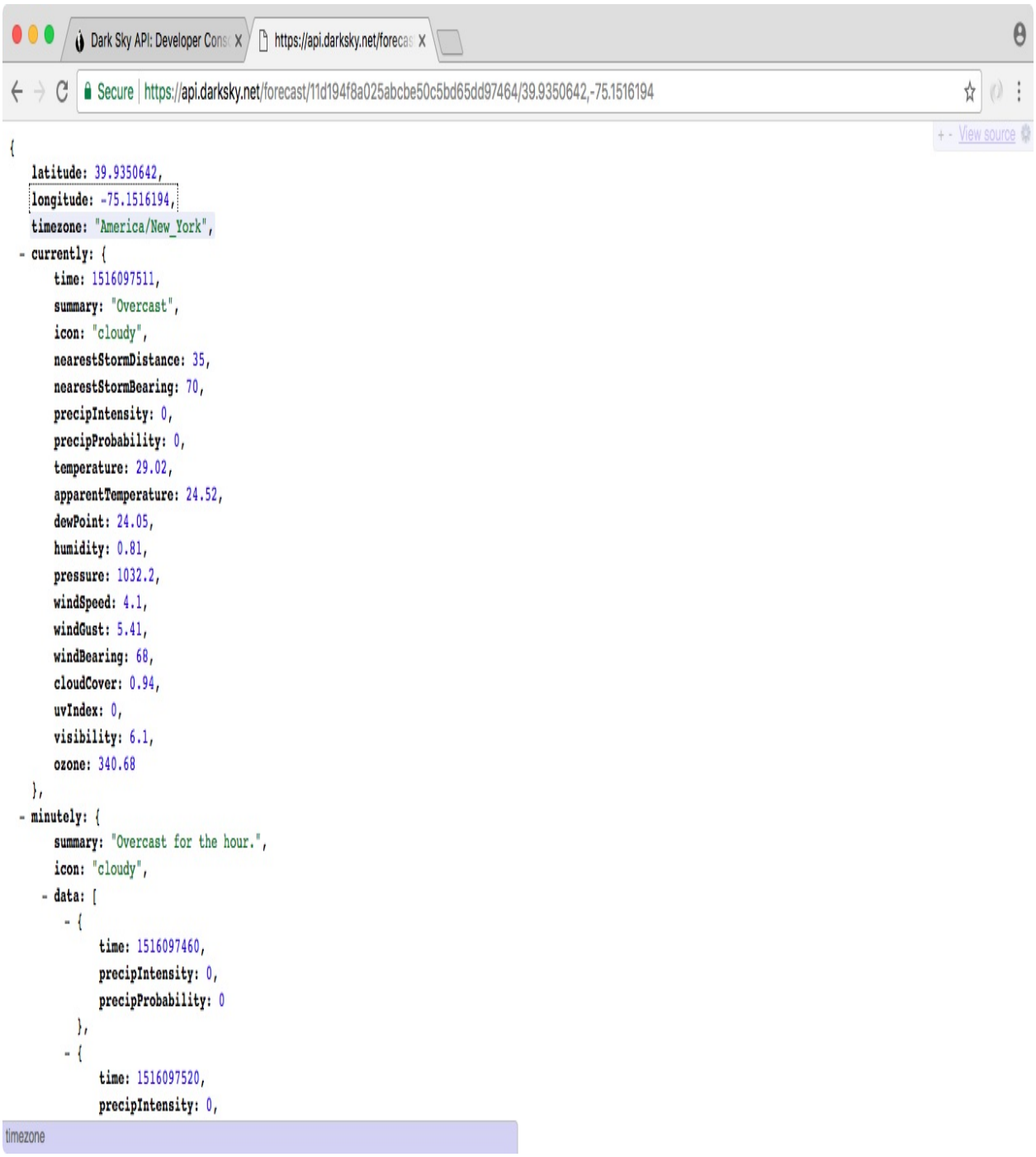
Now, we do need to swap out some of the URL information. For example, we have our API key that needs to get replaced, we also have latitude and longitude. Both of those need to get replaced with the real data. Let's get started with that API key first since we already copy and pasted it inside of `app.js`. We'll copy the API key, and replace the letters `[key]` with the actual value:



Next up, we can grab a set of longitude and latitude coordinates. For this, go inside the Terminal and run our app, `node app.js`, and for the address we can use any zip let's say, `19146` to fetch the latitude and longitude coordinates.

Next up, we'll copy these and place into the URL where they belong. The latitude goes between the forward slash and the comma, and the longitude will go after the comma, as shown here:

Once we have a real URL with all of those three pieces of info swapped out for actual info, we can make the request, and what we'll get back is the forecast information:



Remember, this way the information is showing in the preceding image is due to JSONView, I highly recommend installing it.

Now the data we get back, it is overwhelming. We have a forecast by the minute, we have forecasts by the hour, by the week, by the day, all sorts of information,

it's really useful but it's also super overwhelming. In this chapter, we'll be using the first object that is `currently`. This stores all of the current weather information, things like the current summary which is clear, the temperature, the precipitation probability, the humidity, a lot of really useful information is sitting in it.

In our case, what we really care about is the temperature. The current temperature in Philadelphia is shown `84.95` degrees. This is the kind of information we want to use inside of our application, when someone searches for the weather in a given location.

Making a request for the weather app using the static URL

Now in order to play around with the weather API, we'll take the exact same URL we have defined in the previous section, and we'll make a request in `app.js`. First, we want to do a little setup work.

Inside of `app.js`, we'll comment out everything we have so far, and next to our API key we'll make a call to request, requesting this exact URL, just like we did for the geocode API in the previous section/chapter, before we made it dynamic. Then we'll print out the `body.currently.temperature` property to the screen, so when we run the app we'll see the current temperature for whatever latitude and longitude we used. In our case it's a static latitude and longitude representing Philadelphia.

In order to get started we'll load in request. Now we had it in the `app.js` file before and then we removed it in the previous section, but we'll add it back once again. We'll add it next to the commented out code, by creating a constant called `request`, and loading it in, `const request` equals to `require('request')`:

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

Now we can go ahead and make the actual request, just like we did for the geocode API by calling `request`, it's a function just like this:

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

request();
```

We have to pass in our two arguments, the options object is the first one, and the second one is the arrow function:

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

});
```

This is our callback function that gets fired once the HTTP request finishes. Before we fill out the actual function, we want to set up our options. There're

two options, URL and JSON. We'll set `url` equal to the static string, the exact URL we have in the browser:

```
request({
 url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-75

}, () => {
```

Then in the next line after comma, we can set `json` equal to `true`, telling the request library to go ahead and parse that body as JSON, which it is:

```
request({
 url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-75
 json: true
}, () => {
```

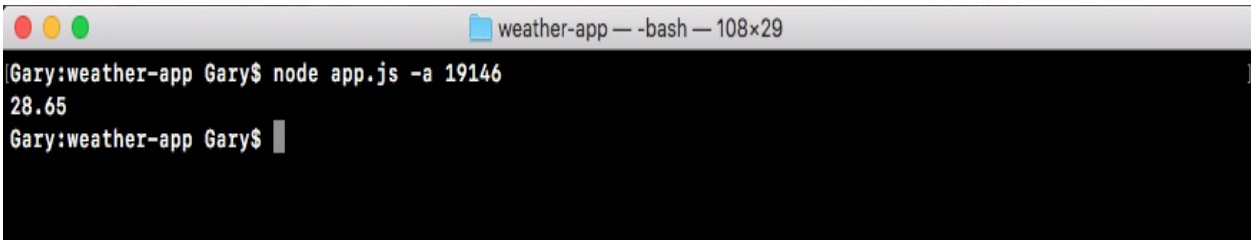
From here, we can go ahead and add our callback arguments; `error`, `response`, and `body`. These are the exact same three arguments we have in the `if` block of `geocode.js` file for the `geocoding` request:

```
request({
 url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-75
 json: true
}, (error, response, body) => {

});
```

Now that we have this in place, the last thing we need to do is print the current temperature, which is available on the body using `console.log` statement. We'll use `console.log` to print `body.currently.temperature`, as shown here:

```
request({
 url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-75
 json: true
}, (error, response, body) => {
  console.log(body.currently.temperature);
});
```

Now that we have the temperature printing, we need to test it by running it from the Terminal. In the Terminal, we'll rerun the previous command. The address is not actually being used here since we commented out that code, and what we get is 28.65, as shown in this code output:



With this we have our weather API call working inside of the application.

Error handling in the the callback function

Now we do want to add a little error handling inside of our callback function. We'll handle errors on the error object, and we'll also handle errors that come back from the `forecast.io` servers. First up, just like we did for the geocoding API, we'll check if error exists. If it does, that means that we were unable to connect to the servers, so we can print a message that relays that message to the user, `console.log` something like `Unable to connect to forecast.io server.`:

```
request({
 url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-75
 json: true
}, (error, response, body) => {
  if (error){
    console.log('Unable to connect to Forecast.io server.');
  }
  console.log(body.currently.temperature);
});
```

Now that we've handled general errors, we can move on to a specific error that the `forecast.io` API throws. This happens when the format of the URL, the latitude and longitude, is not correct.

For example, if we delete some numbers including the comma in the URL, and hit *enter* we'll get a 400 Bad Request:

This is the actual HTTP status code. If you remember from the `geolocation` API we had a `body.status` property that was either `OK` or `ZERO_RESULTS`. This is similar to that property, only this uses the HTTP mechanisms instead of some sort of custom solution that Google used. In our case, we'll want to check if the status code is 400. Now if we have a bad API key, I'll add a couple e's in the URL,

we'll also get a 400 Bad Request:

So both of these errors can be handled using the same code.

Inside of Atom, we can handle this by checking the status code property. After our `if` statement closing curly brace, we'll add `else if` block, `else if (response.statusCode)`, this is the property we looked at when we looked at the response argument in detail. `response.statusCode` will be equal to `400` if something went wrong, and that's exactly what we'll check for here:

```
if (error){
  console.log('Unable to connect to Forecast.io server.');
} else if (response.statusCode === 400) {

}
```

If the status code is `400` we'll print a message, `console.log('Unable to fetch weather')`:

```
if (error){
  console.log('Unable to connect to Forecast.io server.');
} else if (response.statusCode === 400) {
  console.log('Unable to fetch weather.');
}
```

Now we've handled those two errors, and we can move on to the success case. For this we'll add another else if block with `response.statusCode` equals `200`. The status code will equal `200` if everything went well, in that case we'll print the current temperature to the screen.

I'll cut the `console.log(body.currently.temperature)` line out and paste it inside of the else if code block:

```
  if (error){
    console.log('Unable to connect to Forecast.io server.');
  } else if (response.statusCode === 400) {
    console.log('Unable to fetch weather.');
  } else if (response.statusCode === 200) {
    console.log(body.currently.temparature);
  }
});
```

Another way of error handling

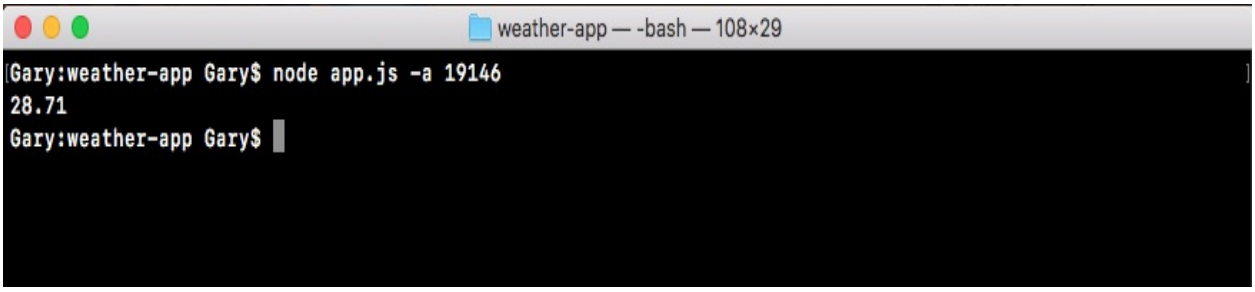
There's is another way to represent our entire if block code. The following is an updated code snippet, and we can actually replace everything we have in the current callback function with this code:

```
if (!error && response.statusCode === 200) {
  console.log(body.currently.temperature);
} else {
  console.log('Unable to fetch weather.');
}
```

This condition checks if there is no error and the response status code is a `200`, if that's the case what do we do? We simply print the temperature like we were doing last time, that was in the `else if` clause at the very bottom. Now we have an `else` case in the updated code snippet, so if there is an error or the status code is not a `200`, we'll go ahead and print this message to the screen. This will handle things like the server not having a network connection, or 404s from an invalid or broken URL. All right, use this code instead and everything should be working as expected with the latest version of the weather API.

Testing the error handling in callback

Now we have some error handling in place and we can go ahead and test that our app still works. From the Terminal we'll rerun the previous command, and we still get a temperature 28.71:



Back inside of Atom, we'll trash some of the data by removing the comma, saving the file:

```
request({
 url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284-75.
 json: true
}, (error, response, body) => {
  if (error){
    console.log('Unable to connect to Forecast.io server.');
  } else if (response.statusCode === 400) {
    console.log('Unable to fetch weather.');
  } else if (response.statusCode === 200) {
    console.log(body.currently.temparature);
  }
});
```

When we rerun it from the Terminal, this time, we would expect Unable to fetch weather. to print to the screen, and when I rerun the app that is exactly what we get, as shown here:

Now, let's add the comma back in and test our last part of the code. To test the if

error, we can test that by removing something like the dot from `forecast.io`:

```
request({
 url: 'https://api.forecastio/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-75.
 json: true
}, (error, response, body) => {
```

We can rerun the app, and we see Unable to connect to Forecast.io server.:

```
Gary:weather-app Gary$ node app.js -a 19146
Unable to connect to Forecast.io server.
Gary:weather-app Gary$
```

All of our error handling works great, and if there is no errors the proper temperature prints to the screen, which is fantastic.

Chaining callbacks together

In this section, we'll take the code that we created in the last section, and break it out into its own file. Similar to what we did with the Geocoding API request where we called `geocodeAddress` instead of actually having the request call in `app.js`. That means we'll make a new folder, a new file, and we'll create a function in there that gets exported.

After that we'll go ahead and learn how to chain callbacks together. So when we get that address from the Terminal we can convert that into coordinates. And we can take those coordinates and convert them into temperature information, or whatever weather data we want to pull off of the return result from the Forecast API.

Refactoring our request call in weather.js file

Now before we can dive into the refactoring, we'll create a brand new file, and we'll worry about getting the code we created in the previous section into that function. Then we'll go for creating that callback.

Defining the new function getWeather in weather file

First, let's make the directory. The directory will be called `weather`. And in the `weather` directory we'll make a new file called `weather.js`.

Now in this file we can take all of our code from `app.js`, and paste it in `weather.js`:

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

request({
  url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-7
  json: true
}, (error, response, body) => {
  if (error) {
    console.log('Unable to connect to Forecast.io server.');
  } else if (response.statusCode === 400) {
    console.log('Unable to fetch weather.');
  } else if (response.statusCode === 200) {
    console.log(body.currently.temperature);
  }
});
```

The only thing we need to do in order to take this code and convert it to create that function, which will get exported. And then we can move our call to the request inside of it. We'll make a brand new function called `getWeather` next to the `request` variable:

```
const request = require('request');
var getWeather = () => {

};
```

`getWeather` will take some arguments, but that'll be added later. For now we'll leave the arguments list empty. Next, we'll take our call to request and move it inside the `getWeather` function:

```
const request = require('request');
var getWeather = () => {
  request({
   url: 'https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/39.9396284,-
   json: true
}, (error, response, body) => {
  if (error) {
    console.log('Unable to connect to Forecast.io server.');
  } else if (response.statusCode === 400) {
```

```
    console.log('Unable to fetch weather.');
  } else if (response.statusCode === 200) {
    console.log(body.currently.temperature);
  }
});
};
```

Then, we can go ahead and export this `getWeather` function. We'll add `module.exports.getWeather` and set it equal to the `getWeather` function that we defined up:

```
module.exports.getWeather = getWeather;
```

Providing weather directory in app.js

Now that we have this in place, we can go ahead and move into `app.js` to add some code. The first thing we need do is remove the API key. We no longer need that. And we'll highlight all of the commented code and uncomment it using the command `/`.

Now we'll import the `weather.js` file. We'll create a `const` variable called `weather`, and setting it equal to the `require`, return result:

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

const geocode = require('./geocode/geocode');
const weather = require('');
```

In this case we're requiring our brand new file we just created. We'll provide a relative path `./` because we're loading in a file that we wrote. Then we'll provide the directory named `weather` followed by the file named `weather.js`. And we can leave off that `js` extension, as we already know:

```
const weather = require('./weather/weather');
```

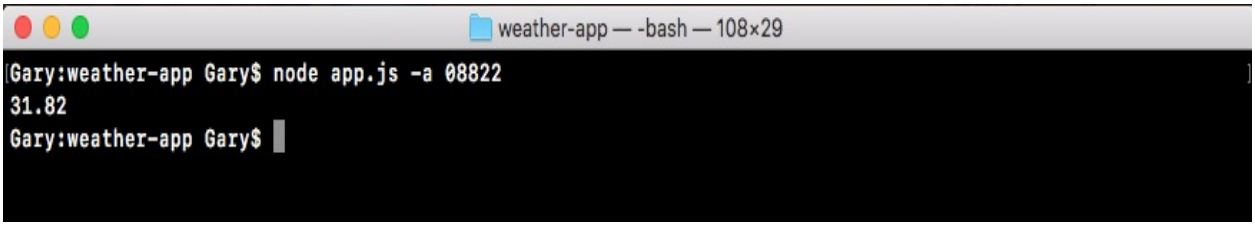
Now that we have the Weather API loaded in, we can go ahead and call it. We'll comment out our call to `geocodeAddress` and, we'll run `weather.getWeather()`:

```
// geocode.geocodeAddress(argv.address, (errorMessage, results) => {
//  if (errorMessage) {
//    console.log(errorMessage);
//  } else {
//    console.log(JSON.stringify(results, undefined, 2));
//  }
//});

weather.getWeather();
```

Now as I mentioned before, there will be arguments later in the section. For now we'll leave them empty. And we can run our file from the Terminal. This means we should see the weather printing for the coordinates, we hard-coded in the previous section. So, we'll run `node app.js`. We'll need to provide an address since we haven't commented out the yargs code. So we'll add a dummy address. I'll

use a zip code in New Jersey:

```
node app.js -a 08822
```



Now, the `geolocation` code is never running, because that is commented out. But we are running the weather code that got moved to the new file. And we are indeed seeing a temperature 31.82 degrees, which means that the code is properly getting executed in the new file.

Passing the arguments in the getWeather function

Now we'll need to pass in some arguments, including a callback function and inside `getWeather` variable in weather file. We'll need to use those arguments instead of a static `lat`/`lng` pair. And we'll also need to call the callback instead of using `console.log`. The first thing we'll do before we actually change the `weather.js` code is change the `app.js` code. There are three arguments to be added. These are `lat`, `lng` and `callback`.

First up we'll want to pass in the latitude. We'll take the static data, like the latitude part from the URL in `weather.js`, copy it, and paste it right inside of the arguments list in `app.js` as first argument. The next one will be the longitude. We'll grab that from the URL, copy it, and paste it inside of `app.js` as the second argument:

```
// lat, lng, callback
weather.getWeather(39.9396284, -75.18663959999999);
```

Then we can go ahead and provide the third one, which will be the callback function. This function will get fired once the weather data comes back from the API. I'll use an arrow function that will get those two arguments we discussed earlier in the previous section: `errorMessage` and `weatherResults`:

```
weather.getWeather(39.9396284, -75.18663959999999, (errorMessage, weatherResults) => {

});
```

The `weatherResults` object containing any sort of temperature information we want. In this case it could be the temperature and the actual temperature. Now, we have used `weatherResults` in place of results, and this is because, we want to differentiate `weatherResults` from the results variable in `geocodeAddress`.

Printing errorMessage in the getWeather function

Inside of the `getWeather` function in `app.js`, we now need to use `if-else` statements in order to print the appropriate thing to the screen, depending on whether or not the error message exists. If there is `errorMessage` we do want to go ahead and print it using `console.log`. In this case we'll pass in the `errorMessage` variable:

```
weather.getWeather(39.9396284, -75.18663959999999, (errorMessage, weatherResults) => {
  if (errorMessage) {
    console.log(errorMessage);
  }
});
```

Now if there is no error message we'll use the `weatherResults` object. We'll be printing a nice formatted message later. For now we can simply print the `weatherResults` object using the pretty printing technique we talked about in the previous chapter, where we call `JSON.stringify` inside of `console.log`:

```
weather.getWeather(39.9396284, -75.18663959999999, (errorMessage, weatherResults) => {
  if (errorMessage) {
    console.log(errorMessage);
  } else {
    console.log(JSON.stringify());
  }
});
```

Inside the `JSON.stringify` parentheses, we'll provide those three arguments, the actual object; `weatherResults`, `undefined` for our filtering function, and a number for our indentation. In this case we'll go with `2` once again:

```
weather.getWeather(39.9396284, -75.18663959999999, (errorMessage, weatherResults) => {
  if (errorMessage) {
    console.log(errorMessage);
  } else {
    console.log(JSON.stringify(weatherResults, undefined, 2));
  }
});
```

And now that we have our `getWeather` call getting called with all three arguments, we can go ahead and actually implement this call inside of `weather.js`.

Implementing getWeather callback inside weather.js file

To get started we'll make the URL in the `weather.js` file dynamic, which means we need to replace the url strings with template strings. Once we have template strings in place, we can inject the arguments, latitude and longitude, right into the URL.

Adding dynamic latitude and longitude

Let's go ahead and define all the arguments that are getting passed in. We add `lat`, `lng`, and our `callback`:

```
var getWeather = (lat, lng, callback) => {
```

First off let's inject that latitude. We'll take the static latitude, remove it, and between the forward slash and the comma we'll inject it using dollar with our curly braces. This lets us inject a value into our template string; in this case `lat`. And we can do the exact same thing right after the comma with the longitude. We'll remove the static longitude, use the dollar sign with our curly braces to inject the variable into the string:

```
var getWeather = (lat, lng, callback) => {
  request({
    url: `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${lat},${ln
```

Now that the URL is dynamic, the last thing we need to do inside of `getWeather` is change our `console.log` calls to `callback` calls.

Changing console.log calls into callback calls

To change our `console.log` into `callback` calls, for the first two `console.log` calls we can replace `console.log` with `callback`. And this will line up with the arguments that we specified in `app.js`, where the first one is the `errorMessage` and the second one is the `weatherResults`. In this case we'll pass the `errorMessage` back and the second argument is `undefined`, which it should be. We can do the same thing for `Unable to fetch weather`:

```
if (error) {
  callback('Unable to connect to Forecast.io server.');
} else if (response.statusCode === 400) {
  callback('Unable to fetch weather.');
}
```

Now the third `console.log` call will be a little more complex. We'll have to actually create an object instead of just passing the temperature back. We'll call the `callback` with the first argument being `undefined`, because in this case there is no `errorMessage`. Instead we'll provide that `weatherResults` object:

```
if (error) {
  callback('Unable to connect to Forecast.io server.');
} else if (response.statusCode === 400) {
  callback('Unable to fetch weather.');
} else if (response.statusCode === 200) {
  callback(undefined, {

  })
  console.log(body.currently.temperature);
}
```

Inside the parentheses, we can define all the temperature properties we like. In this case we'll define `temperature`, setting it equal to `body.currently`, which stores all of the `currently` weather data, `.temperature`:

```
else if (response.statusCode === 200) {
  callback(undefined, {
    temperature: body.currently.temperature
  })
  console.log(body.currently.temperature);
}
```

Now that we have the `temperature` variable we can go ahead and provide that

second property to the object, which is `actual temperature`. Actual temperature will account for things like humidity, wind speed, and other weather conditions. The actual temperature data is available under a property on currently called `apparentTemperature`. We'll provide that. And as the value we'll use the same thing. This gets us to the `currently` object, just like we do for temperature. This will be `body.currently.apparentTemperature`:

```
else if (response.statusCode === 200) {
  callback(undefined, {
    temperature: body.currently.temperature,
    apparentTemperature: body.currently.apparentTemperature
  })
  console.log(body.currently.temperature);
}
```

Now we have our two properties, so we can go ahead and remove that `console.log` statement. Add a semicolon. The final code will look like:

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

var getWeather = (lat, lng, callback) => {
  request({
    url: `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${lat},${ln
    json: true
  }, (error, response, body) => {
    if (error) {
      callback('Unable to connect to Forecast.io server.');
    } else if (response.statusCode === 400) {
      callback('Unable to fetch weather.');
    } else if (response.statusCode === 200) {
      callback(undefined, {
        temperature: body.currently.temperature,
        apparentTemperature: body.currently.apparentTemperature
      });
    }
  });
};

module.exports.getWeather = getWeather;
```

Now we can go ahead and run the app. We have our `getWeather` function wired up both inside of the `weather.js` file and inside of `app.js`. Now once again we are still using static coordinates, but this will be the last time we run the file with that static data. From the Terminal we'll rerun the application:

And as shown we get our temperature object printing to the screen. We have our temperature property 48.82 and we have the apparentTemperature, which is already at 47.42 degrees.

With this in place we're now ready to learn how to chain our callbacks together. That means in `app.js` we'll take the results that come back from `geocodeAddress`, pass them in to `getWeather`, and use that to print dynamic weather for the address you provide over here in the Terminal. In this case we would get the address for the town in New Jersey. As opposed to the static address which we're using in the `app.js` file that latitude/longitude pair is for Philadelphia.

Chaining the geocodeAddress and getWeather callbacks together

To get started we have to take our `getWeather` call and actually move it inside of the `callback` function for `geocodeAddress`. Because inside this `callback` function is the only place we have access to the latitude and longitude pairs.

Now if we open the `geocode.js` file, we can see that we get `formatted_address` back as the address property, we get the `latitude` back as latitude, and we get `longitude` back as longitude. We'll start wiring this up.

Moving getWeather call into geocodeAddress function

First, we do need to remove the comments of `geocodeAddress` in the `app.js`.

Next, we'll go ahead and take the `console.log` statement in the success case and replace it with a `console.log` call that will print the formatted address:

```
geocode.geocodeAddress(argv.address, (errorMessage, results) => {
  if (errorMessage) {
    console.log(errorMessage);
  } else {
    console.log(results.address);
  }
});
```

This will print the address to the screen, so we know exactly what address we're getting weather data for.

Now that we have our `console.log` printing the address, we can take the `getWeather` call, and move it right below the `console.log` line:

```
geocode.geocodeAddress(argv.address, (errorMessage, results) => {
  if (errorMessage) {
    console.log(errorMessage);
  } else {
    console.log(results.address);
    weather.getWeather(39.9396284, -75.18663959999999,
    (errorMessage, weatherResults) => {
      if (errorMessage) {
        console.log(errorMessage);
      } else {
        console.log(JSON.stringify(weatherResults, undefined, 2));
      }
    });
  }
});
```

And with this in place we're now really close to actually chaining the two callbacks together. All that's left to do is take these static coordinates and replace them with the dynamic ones, which will be available in the `results` object.

Replacing static coordinates with dynamic coordinates

The first argument will be `results.latitude`, which we defined in `app.js` on the object. And the second one will be `results.longitude`:

```
geocode.geocodeAddress(argv.address, (errorMessage, results) => {
  if (errorMessage) {
    console.log(errorMessage);
  } else {
    console.log(results.address);
    weather.getWeather(results.latitude, results.longitude,
    (errorMessage, weatherResults) => {
      if (errorMessage) {
        console.log(errorMessage);
      } else {
        console.log(JSON.stringify(weatherResults, undefined, 2));
      }
    });
  }
});
```

This is all we need to do to take the data from `geocodeAddress` and pass it in to `getWeather`. This will create an application that prints our dynamic weather, the weather for the address in the Terminal.

Now before we go ahead and run this, we'll replace the object call with a more formatted one. We'll take both of the pieces of information-the `temperature` variable and the `apparentTemperature` variable from `weather.js` file, and use them in that string in `app.js`. This means that we can remove the `console.log` in the `else` block of `getWeather` call, replacing it with a different `console.log` statement:

```
if (errorMessage) {
  console.log(errorMessage);
} else {
  console.log();
}
```

We'll use template strings, since we plan to inject a few variables in; these're currently, followed by the temperature. We'll inject that using `weatherResults.temperature`. And then we can go ahead and put a period, and add something along the lines of: `It feels like`, followed by the `apparentTemperature` property, which I'll inject using `weatherResults.apparentTemperature`. I'll put a period

after that:

```
if (errorMessage) {
  console.log(errorMessage);
} else {
  console.log(`It's currently ${weatherResults.temperature}. It feels like
    ${weatherResults.apparentTemperature}`);
}
```

We now have a `console.log` statement that prints the weather to the screen. We also have one that prints the address to the screen, and we have error handlers for both `geocodeAddress` and `getWeather`.

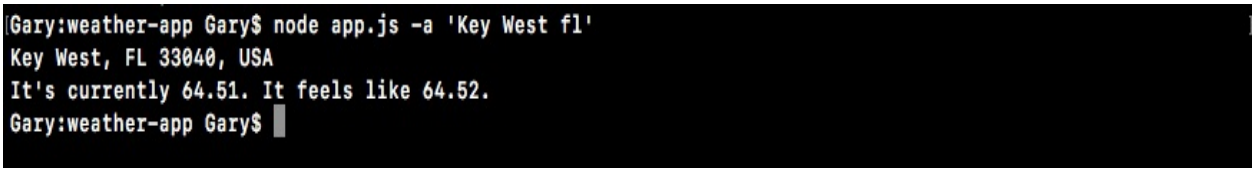
Testing the chaining of callbacks

Let's go ahead and test this by rerunning the `node app.js` command in the Terminal. We'll use the same zip code, `08822`:

```
node app.js -a 08822
```

When we run it we get Flemington, NJ as the formatted address and It's currently is 31.01. It feels like 24.9. Now to test that this is working we'll type in something else inside of quotes, something like `Key West fl`:

```
node app.js -a 'Key West fl'
```



And when we run this command we do get Key West, FL as shown as the formatted address, and It's currently 64.51. It feels like 64.52.

With this in place, the weather application is now wired up. We take the address we get the latitude/longitude pair using the Google Geocoding API. Then we use our forecast API to take that latitude/longitude pair and convert it into temperature information.

Summary

In this chapter, we learned about how to set up yargs for the `weather-app` file and how to include user input in it. Next, we looked into how to handle errors inside of our callback functions and how to recover from those errors. We simply added `else`/`if` statements inside of the `callback` function. Callbacks are just one function, so in order to figure out if things went well or if things didn't go well, we have to use `else`/`if` statements, this lets us do different things, such as print different messages, depending on whether or not we perceive the request to have gone well. Then, we made our first request to the weather API, and we looked into a way to fetch the weather based off of the latitude-longitude combination.

Last, we looked in chaining the `geocodeAddress` and `getWeather` call functions. We took that request call that was originally in `app.js`, and we moved it into `weather.js`, defining it there. We used a callback to pass the data from `weather.js` into `app.js` where we imported the `weather.js` file. Then, inside of the callback for `geocodeAddress` we call `getWeather` and inside of that `callback` we printed the weather specific information to the screen. This was all done using `callback` functions.

In the next chapter, we'll talk about a different way we can synchronize our asynchronous code using ES6 promises.

Promises in Asynchronous Programming

In the previous two chapters, we looked at many important concepts of asynchronous programming in Node. This chapter is about promises. Promises are available in JavaScript since ES6. Although they have been around in third-party libraries for quite some time, they finally made their way into the core JavaScript language, which is great because they're a really fantastic feature.

In this chapter, we'll learn about how promises work, we'll start to understand exactly why they're useful, and why they've even come to exist inside JavaScript. We'll take a look at a library called axios that supports promises. This will let us simplify our code, creating our promise calls easily. We'll actually rebuild an entire weather app in the last section.

Specifically, we'll look into following topics:

- Introduction to ES6 promises
- Advanced promises
- Weather app with promises

Introduction to ES6 promises

Promises aim to solve a lot of the problems that come up when we have a lot of asynchronous code in our application. They make it a lot easier to manage our asynchronous computations—things such as requesting data from a database. Alternatively, in the case of a weather app, things such as fetching data from a URL.

In the `app.js` file we do a similar thing using callbacks:

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

const geocode = require('./geocode/geocode');
const weather = require('./weather/weather');

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

geocode.geocodeAddress(argv.address, (errorMessage, results) => {
  if (errorMessage) {
    console.log(errorMessage);
  } else {
    console.log(results.address);
    weather.getWeather(results.latitude, results.longitude, (errorMessage, weatherResul
      if (errorMessage) {
        console.log(errorMessage);
      } else {
        console.log(`It's currently ${weatherResults.temperature}. It feels like ${weat
      }
    });
  }
});
```

In this code, we have two callbacks:

- One that gets passed into `geocodeAddress`
- One that gets passed into `getWeather`

We use this to manage our asynchronous actions. In our case, it's things such as

fetching data from an API, using an HTTP request. We can use promises in this example to make the code a lot nicer. This is exactly the aim later in the chapter.

In this section, we'll explore the basics concept of promises. We'll compare and contrast callbacks with promises just yet, because there's a lot more subtleties than can be described without knowing exactly how promises work. So, before we talk about why they're better, we will simply create some.

Creating an example promise

In the Atom, inside the `playground` folder, we'll create a new file and call it `promise.js`. Before we define promises and talk about exactly how they work, we will run through a simple example because that is the best way to learn just about anything—going through an example and seeing how it works.

To get started, we'll work through a very basic example. We'll stick to the core promise features.

To get started with this very simple example, we'll make a variable called `somePromise`. This will eventually store the promise object. We'll be calling various methods on this variable to do something with the promise. We'll set the `somePromise` variable equal to the return result from the constructor function for promises. We'll use the `new` keyword to create a new instance of a promise. Then, we'll provide the thing we want to create a new instance of, `Promise`, as shown here:

```
var somePromise = new Promise
```

Now this `Promise` function, which is indeed a function—we have to call it like one; that is, it takes one argument. This argument will be a function. We'll use an anonymous arrow function (`=>`), and inside it, we'll do all of the asynchronous stuff we want to do:

```
var somePromise = new Promise(() => {

});
```

It will all be abstracted, kind of like we abstract the HTTP request inside the `geocodeAddress` function in the `geocode.js` file:

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

var geocodeAddress = (address, callback) => {
  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
      });
    }
  });
};

module.exports.geocodeAddress = geocodeAddress;
```

All of the complex logic in the `geocodeAddress` function does indeed need to happen, but the `app.js` file doesn't need to worry about it. The `geocode.geocodeAddress` function in the `app.js` file has a very simple `if` statement that checks whether there's an error. If there is an error, we will print a message, and if there's not, we move on. The same thing will be true with promises.

The `new Promise` callback function will get called with two arguments, `resolve` and `reject`:

```
var somePromise = new Promise((resolve, reject) => {

});
```

This is how we'll manage the state of our promise. When we make a promise, we're making a promise; we're saying, "Hey, I'll go off and I'll fetch that website data for you." Now this could go well, in which case, you will `resolve` the promise, setting its state to fulfilled. When a promise is fulfilled, it's gone out and it's done the thing you've expected it to do. This could be a database request, an HTTP request, or something else completely.

Now when you call `reject`, you're saying, "Hey, we tried to get that thing done man, but we just could not." So the promise has been considered rejected. These are the two states that you can set a promise to—fulfilled or rejected. Just like inside `geocode.js`, we either provide one argument for an error, or we provide the second argument if things went well. Instead of doing that though, promises give us two functions we can call.

Now, in order to illustrate exactly how we can use these, we'll call `resolve`. Once again, this is not asynchronous. We're not doing anything quite yet. So all of this will happen essentially in real time, as far as you see in Terminal. We'll call

`resolve` with some data. In this case, I'll pass in a string, `Hey. It worked!`, as shown here:

```
var somePromise = new Promise((resolve, reject) => {
     resolve('Hey. It worked!');
});
```

Now this string is the value the promise was fulfilled with. This is exactly what someone will get back. In case of the `geocode.geocodeAddress` function in app file, it could be the data, whether it's the results or the error message. In our case though, we're using `resolve`, so this will be the actual data the user wanted. When things go well, `Hey. It worked!` is what they expect.

Now you can only pass one argument to both `resolve` and `reject`, which means that if you want to provide multiple pieces of information I recommend that you resolve or reject an object that you can set multiple properties on. In our case though, a simple message, `Hey. It worked!`, will do the job.

Calling the promise method then

Now in order to actually do something when the promise gets either resolved or rejected, we need to call a promise method called `then`; `somePromise.then`. The `then` method lets us provide `callback` functions for both success and error cases. This is one of the areas where callbacks differ from promises. In a callback, we had one function that fired no matter what, and the arguments let us know whether or not things went well. With promises we'll have two functions, and this will be what determines whether or not things went as planned.

Now before we dive into adding two functions, let's start with just one. Right here, I'll call then, passing in one function. This function will only get called if the promise gets fulfilled. This means that it works as expected. When it does, it will get called with the value passed to `resolve`. In our case, it's a simple `message`, but it can be something like a user object in the case of a database request. For now though, we'll stick with `message`:

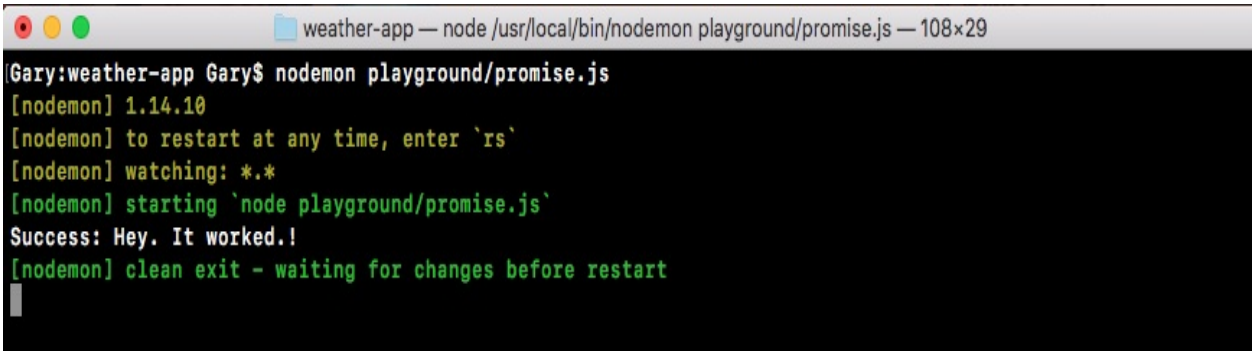
```
somePromise.then((message) => {

})
```

This will print `message` to the screen. Inside the callback, when the promise gets fulfilled we'll call `console.log`, printing `Success`, and then as a second argument, we'll print the actual `message` variable:

```
somePromise.then((message) => {
  console.log('Success: ', message);
})
```

Running the promise example in Terminal

Now that we have a very basic promise example in place, let's run it from the Terminal using `nodemon`, which we installed in the previous chapter. We'll add `nodemon`, and then we'll go into the `playground` folder, `/promise.js`:



When we do this right away, our app runs and we get success. `Hey. It worked!` This happens instantaneously. There was no delay because we haven't done anything asynchronously. Now when we first explored callbacks (refer to `Chapter 5`, *Basics of Asynchronous Programming in Node.js*), we used `setTimeout` to simulate a delay, and this is exactly what we'll do in this case.

Inside our `somePromise` function, we'll call `setTimeout`, passing in the two arguments: the function to call after the delay and the delay in milliseconds. I'll go with `2500`, which is 2.5 seconds:

```
var somePromise = new Promise((resolve, reject) => {
 setTimeout(() => {

}, 2500);
```

Now after those 2.5 seconds are up, then, and only then, do we want to `resolve` the promise. This means that our function, the one we pass into then will not get called for 2.5 seconds. Because, as we know, this will not get called until the promise resolves. I'll save the file, which will restart `nodemon`:

In Terminal, you can see we have our delay, and then `success: Hey it worked!` prints to the screen. This 2.5 second delay was caused by this `setTimeout`. After the delay was up (in this case it's an artificial delay, but later it'll be a real delay), we're able to `resolve` with the data.

Error handling in promises

Now there's a chance that things didn't go well. We have to handle errors inside our Node applications. In that case, we wouldn't call `resolve`, we would call `reject`. Let's comment out the `resolve` line, and create a second one, where we call `reject`. We'll call `reject` much the same way we called `resolve`. We have to pass in one argument, and in this case, a simple error message like `Unable to fulfill promise` will do:

```
var somePromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    // resolve('Hey. It worked!');
    reject('Unable to fulfill promise');
  }, 2500);
});
```

Now when we call `reject`, we're telling the promise that it has been rejected. This means that the thing we tried to do did not go well. Currently, we don't have an argument that handles this. As we mentioned, this function only gets called when things go as expected, not when we have errors. If I save the file and rerun it in Terminal, what we'll get is a promise that rejects:

```
[nodemon] restarting due to changes...
[nodemon] starting `node playground/promise.js`
(node:1053) UnhandledPromiseRejectionWarning: Unable to fulfill promise
(node:1053) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by t
hrowing inside of an async function without a catch block, or by rejecting a promise which was not handled w
ith .catch(). (rejection id: 1)
(node:1053) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promis
e rejections that are not handled will terminate the Node.js process with a non-zero exit code.
[nodemon] clean exit - waiting for changes before restart
```

However, we don't have a handler for it, so nothing will print to the screen. This will be a pretty big problem. We need to do something with that error message. Maybe we will alert the user, or we will try some other code.

As shown in the previous code output, we can see that nothing printed between the restarting and exiting. In order to do something with the error, we'll add a second argument to the `then` method. This second argument is what lets us handle errors in our promises. This argument will get executed and called with that value. In this case, it's our message. We'll create an argument called `errorMessage`, as shown here:

```
somePromise.then((message) => {
  console.log('Success: ', message);
}, (errorMessage) => {

});
```

Inside the argument, we can do something with that. In this case, we'll print it to the screen using `console.log`, printing `Error` with a colon and a space to add some nice formatting, followed by the actual value that was rejected:

```
}, (errorMessage) => {
  console.log('Error: ', errorMessage);
});
```

Now that we have this in place, we can refresh things by saving the file. We will now see our error message in Terminal, because we now have a place for it to do something:

Here, we have a place for it to print the message to the screen; `Unable to fulfill promise` prints to the screen, which works exactly as expected.

Merits of promises

We now have a promise that can either get resolved or rejected. If it gets resolved, meaning the promise was fulfilled, we have a function that handles that. If it gets rejected, we have a function that handles that as well. This is one of the reasons why promises are awesome. You get to provide different functions, depending on whether or not the promise got resolved or rejected. This lets you avoid a lot of complex `if` statements inside of our code, which we needed to do in `app.js` to manage whether or not the actual callback succeeded or failed.

Now inside a promise, it's important to understand that you can only either `resolve` or `reject` a promise once. If you `resolve` a promise you can't `reject` it later, and if you `resolve` it with one value you can't change your mind at a later point in time. Consider this example, where I have a code like the following code; here I `resolve` first and then I `reject`:

```
var somePromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Hey. It worked!');
    reject('Unable to fulfill promise');
  }, 2500);
});

somePromise.then((message) => {
  console.log('Success: ', message);
}, (errorMessage) => {
  console.log('Error: ', errorMessage);
});
```

In this case, we'll get our success `message` printing to the screen. We'll never see `errorMessage`, because, as I just said, you can only do one of these actions once. You can either `resolve` once or you can `reject` once. You can't do both; you can't do either twice.

This is another great advantage over callbacks. There's nothing preventing us from accidentally calling the `callback` function twice. Let's consider the `geocode.js` file for example. Let's add another line in the `if` block of geocode request call, as shown here:

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

```

```
var geocodeAddress = (address, callback) => {
  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.');
      callback();
```

This is a more obvious example, but it could easily be hidden inside of complex `if-else` statements. In this case, our `callback` function in `app.js` will indeed get called twice, which can cause really big problems for our program. Inside the promise example this callback will never get called twice, no matter how many times you try to call `resolve` or `reject`, this function will only get fired once.

We can prove that right now by calling `resolve` again. In the promise example case, let's save the file with the following changes:

```
var somePromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Hey. It worked!');
    resolve();
    reject('Unable to fulfill promise');
  }, 2500);
});
```

Now, let's refresh things; we'll `resolve` with our message, `Hey. It worked!` and we'll never ever have the function fired a second time with no message. Because, as we said, the promise is already resolved. Once you set a promise's state to either fulfilled or rejected, you can't set it again.

Now before a promise's `resolve` or `reject` function gets called, a promise is in a state known as pending. This means that you're waiting for information to come back, or you're waiting for your async computation to finish. In our case, while we're waiting for the weather data to come back, the promise would be considered pending. A promise is considered settled when it has been either fulfilled or rejected.

No matter which one you chose, you could say the promise has settled, meaning that it's no longer pending. In our case, this would be a settled promise that was indeed fulfilled because `resolve` is called right here. So these are just a couple of the benefits of promises. You don't have to worry about having callbacks called twice, you can provide multiple functions—one for success handling and one for

error handling. It really is a fantastic utility!

Now that we've gone through a quick example of how promises work, going over just the very fundamentals, we'll to move on to something slightly more complex.

Advanced promises

In this section, we'll explore two more ways to use promises. We'll create functions that take input and return a promise. Also, we'll explore promise chaining, which will let us combine multiple promises.

Providing input to promises

Now the problem with the example we discussed in the previous section is that we have a promise function, but it doesn't take any input. This most likely is never going to be the case when we're using real-world promises. We'll want to provide some input, such as the ID of a user to fetch from the database, a URL to request, or a partial URL, for example, just the address component.

In order to do this, we'll have to create a function. For this example, we'll make a variable, which will be a function called `asyncAdd`:

```
var asyncAdd = () => {

}
```

This will be a function that simulates the async functionality using `setTimeout`. In reality, it's just going to add two numbers together. However, it will illustrate exactly what we need to do, later in this chapter, to get our weather app using promises.

Now in the function, we will take two arguments, `a` and `b`, and we'll return a promise:

```
var asyncAdd = (a, b) => {

};
```

So, whoever calls this `asyncAdd` method, they can pass in input, but they can also get the promise back so that they can use then to sync up and wait for it to complete. Inside the `asyncAdd` function, we'll use `return` to do this. We'll `return` the `new Promise` object using the exact same `new Promise` syntax we did when we created the `somePromise` variable. Now this is the same function, so we do need to provide the constructor function that gets called with both `resolve` and `reject`, just like this:

```
var asyncAdd = (a, b) => {
 return new Promise((resolve, reject) => {

 });
```

Now we have an `asyncAdd` function, which takes two numbers and returns a

promise. The only thing left to do is to actually simulate the delay, and make the call to `resolve`. To do this, we'll simulate the delay using `setTimeout`. Then we'll pass in my `callback` function, setting the delay to 1.5 seconds, or `1500` milliseconds:

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

 }, 1500)
 });
```

In the `callback` function, we'll write a simple `if-else` statement that will check if the type of both `a` and `b` is a number. If it is, great! We'll `resolve` the value of the two numbers added. If they're not numbers (one or more), then we'll `reject`. To do this, we'll use the `if` statement with the `typeof` operator:

```
setTimeout(() => {
  if (typeof a === 'number')
 }, 1500);
```

Here, we're using the `typeof` object to get the string type before the variable. Also, we're checking whether it's equal to a number, which is what will come back from `typeof` when we have a number. Now similar to `a`, we'll add `typeof b`, which is also a number:

```
    if (typeof a === 'number' && typeof b === 'number') {}
```

We can add the two numbers up, resolving the value. Inside the code block of the `if` statement, we'll call `resolve`, passing in `a + b`:

```
return new Promise((resolve, reject) => {
  setTimeout(() => {
    if (typeof a === 'number' && typeof b === 'number') {
      resolve(a + b);
    }
  }, 1500);
```

This will add the two numbers up, passing in one argument to `resolve`. Now this is the happy path when both `a` and `b` are indeed numbers. If things don't go well, we'll want to add `reject`. We'll use the `else` block to do this. If the previous condition fails, we'll `reject` by calling `reject('Arguments must be numbers')`:

```
  if (typeof a === 'number' && typeof b === 'number') {
    resolve(a + b);
  } else {
    reject('Argumets must be numbers');
  }
```

Now we have an `asyncAdd` function that takes two variables, `a` and `b`, returns a promise, and anyone who happens to call `asyncAdd` can add a then call onto the return result to get that value.

Returning the promises

Now what exactly will this look like? To show this, first we'll comment out all of the code we have in the `newPromise` variable of `promise.js`. Following this, we'll call the `asyncAdd` variable where we make `asyncAdd`. We'll call it like we would any other function, by passing in two values. Remember, this could be a database ID or anything else for an async function. In our case, it's just two numbers. Let's say, `5` and `7`. Now the return value from this function is a promise. We can make a variable and call then on that variable, but we can also just tack the `then` method, as shown here:

```
asyncAdd(5, 7).then
```

This is exactly what we'll do when we use promises; we'll tack on then, passing in our callbacks. The first callback being the success case, and the second one being the error case:

```
ouldasyncAdd(5, 7).then(() => {
}, () => {

});
```

In the second callback, we'll get our `errorMessage`, which we can log to the screen using the `console.log(errorMessage);` statement, as shown here:

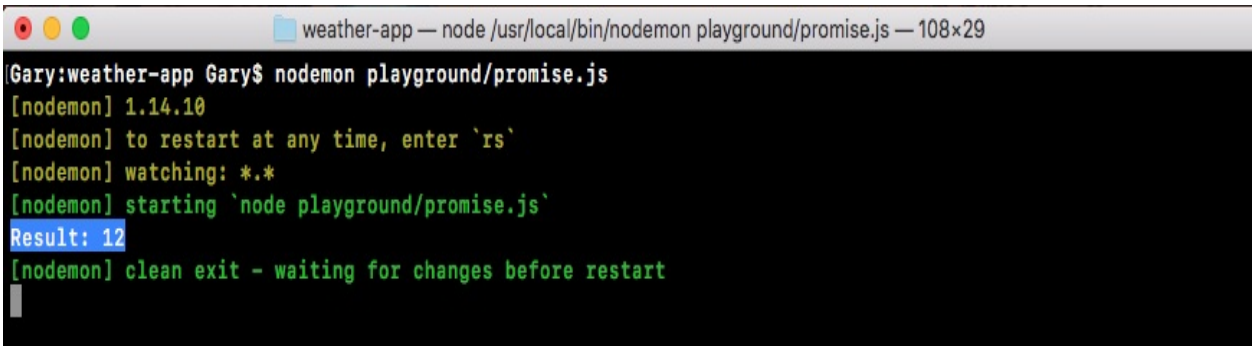
```
asyncAdd(5, 7).then(() => {

}, (errorMessage) => {
 console.log(errorMessage);
});
```

If one or more of the numbers are not actually numbers, the `error` function will fire because we called `reject`. If both are numbers, all we'll do will get the result and print it to the screen, using `console.log`. We'll add `res` and inside the arrow function (`=>`), we'll add the `console.log` statement and print the string `Result` with a colon. Then, as the second argument in `console.log`, we'll pass in the actual number, which will print it to the screen as well:

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

```
});
```

Now that we have our promise `asyncAdd` function in place, let's test this out inside Terminal. To do this, we'll run `nodemon` to start up `nodemon playground/promise.js`:



Right away, we'll get the delay and the result, `12` prints to the screen. This is fantastic! We are able to create the function that takes the dynamic input, but still returns a promise.

Now notice that we've taken an async function that usually requires callbacks and we've wrapped it to use promises. This is a good handy feature. As you start using promises in Node, you'll come to realize that some things do not support promises and you'd like them to. For example, the request library that we used to make our HTTP requests does not support promises natively. However, we can wrap our request call inside of a promise, which is what we'll to do later in the section. For now though, we have a basic example illustrating how this works. Next, we'd like to talk about promise chaining.

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);
```