Section 1: Welcome

Welcome to the course!

This first section contains a brief overview of the class. There are no lecture notes for this first section as it's an introduction to the rest of the class. This section is still important though, so make sure to watch the lecture videos to learn how to get the most out of the class.

Enjoy!

Section 2: Installing and Exploring Node.js

Lesson 1: Section Intro

In this section, you're going to set up your machine for the rest of the course. This includes installing Node.js and Visual Studio Code. This section also dives into what Node.js is, how Node.js works, and why Node.js is a tool worth learning.

Lesson 2: Installing Node.js and Visual Studio Code

In this lesson, you'll install Node.js and Visual Studio Code. Both are free, open source, and available for all operating system. They're the only tools needed to get started with Node!

Below are links to both tools. Take a moment to install them before continuing on with the class.

Links

- Node.js
- Visual Studio Code

Lesson 3: What is Node.js?

In this lesson, you'll explore what Node.js is. This includes a brief tour of the V8 JavaScript engine, non-blocking I/O, and more!

This lesson contains a presentation that covers what Node.js is. There are no notes for presentation lectures. Please refer to the video for details.

Lesson 4: Why Should I Use Node.js?

Why should you use Node.js? In this lesson, you'll learn what makes Node.js a tool worth using.

This lesson contains a presentation that covers the major advantages of Node.js. There are no notes for presentation lectures. Please refer to the video for details.

Lesson 5: Your First Node.js Script

It's time. In this lesson, you'll be creating and running your very first Node.js app.

Creating a Script

Node.js scripts are created with the `js` file extension. Remember that Node.js is not a programming language. All the code in this course is JavaScript code, which is why the `js` extension is used.

Below is an example script stored in a file named `index.js`.

```
console.log('Hello Node.js!')
```

Running a Script

You can run a Node.js script using the `node` command. Open up a new terminal window and navigate to the directory where the script lives. From the terminal, you can use the `node` command to provide the path to the script that should run. You can see an example of this command in the terminal below.

```
$ node index.js
Hello Node.js!
```

When a Node.js script calls `console.log`, the logged values will show up in the terminal. This is a great way to get output from your Node.js application

Section 3: Node.js Module System

Lesson 1: Section Intro

The best way to get started with Node.js is to explore its module system. The module system lets you load external libraries into your application. That'll enable you to take advantage of built-in Node.js modules as well as third-party npm modules. This includes libraries for connecting to database, creating web servers, and more!

Lesson 2: Importing Node.js Core Modules

Node.js comes with dozens of built-in modules. These built-in modules, sometimes referred to as core modules, give you access to tools for working with the file system, making http requests, creating web servers, and more! In this lesson, you'll learn how to load in those core modules and use them in your code.

Importing Node.js Core Modules

To get started, let's work with some built-in Node.js modules. These are modules that come with Node, so there's no need to install them.

The module system is built around the `require` function. This function is used to load in a module and get access to its contents. `require` is a global variable provided to all your Node.js scripts, so you can use it anywhere you like!

Let's look at an example.

```
const fs = require('fs')

fs.writeFileSync('notes.txt', 'I live in Philadelphia')
```

The script above uses require to load in the `fs` module. This is a built-in Node.js module that provides functions you can use to manipulate the file system. The script uses `writeFileSync` to write a message to `notes.txt`.

After you run the script, you'll notice a new `notes.txt` file in your directory. Open it up and you'll see, "I live in Philadelphia!".

Links

- Node.js documentation

- Node.js fs documentation

Lesson 3: Importing Your Own Files

Putting all your code in a single file makes it easy to get started with Node.js. As you add more code, you'll want to stay organized and break your Node.js app into multiple scripts that all work together. In this lesson, you'll learn how to create a Node.js application that's spread out across multiple files.

Importing Your Own Files

You know how to use `require` to load in built-in modules. `require` can also be used to load in JavaScript files you've created. All you need to do is provide `require` with a relative path to the script you want to load. This path should start with `./` and then link to the file that needs to be loaded in.

```
const checkUtils = require('./src/utils.js')

checkUtils()
```

The code above uses `require` to load in a file called `utils.js` in the `src` directory. It stores the module contents in a variable, and then uses the contents in the script.

Exporting from Files

Node.js runs the scripts that you require. That means the `require` call above will cause `utils.js` to run. Node.js provides the required script with a place to store values that should be exported as part of the library. This is on `module.exports`.

You can see `utils.js` below. A function is defined and then assigned to `module.exports`. The value stored on `module.exports` will be the return value for `require` when the script is imported. That means other scripts could load in the utilities to access the `check` function.

```
const check = function () {
    console.log('Doing some work...')
}

module.exports = check
```

If you run the original script, you'll see the message that logged from the `check` function in `utils.js`.

```
$ node app.js
Doing some work...
```

Your Node.js scripts don't share a global score. This means variables created in one scripts are not accessible in a different script. The only way to share values between scripts is by using `require` with `module.exports`.

Lesson 4: Importing npm Modules

When you install Node.js, you also get npm. npm is a package manager that allows you to install and use third-party npm libraries in your code. This opens up a world of possibilities, as there are npm packages for everything from email sending to file uploading. In this lesson, you'll learn how to integrate npm into your Node.js app.

Initializing npm

Your Node.js application needs to initialize npm before npm can be used. You can run `npm init` from the root of your project to get that done. That command will ask you a series of questions about the project and it'll use the information to generate a `package.json` file in the root of your project.

Here's an example.

```
{
  "name": "notes-app",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
}
```

Installing an npm Module

You're now ready to install an npm module. This is done using the `npm` command which was set up when Node.js was installed. You can use `npm install` to install a new module in your project.

```
npm install validator@10.8.0
```

The command above installs version 10.8.0 of validator. If you want to install the latest version of a module, you can leave off the version number as shown below.

```
npm install validator
```

This command does three important things:

First, it creates a `node_modules` directory. npm uses this directory to store all the code for the npm modules you have installed.

Second, npm adds the module as a dependency by listing it in the `dependencies` property in `package.json`. This allows you to track and manage the module you have installed.

Third, npm creates a `package-lock.json` file. This includes detailed information about the modules you've installed which helps keep things fast and secure.

You should never make changes to `node_modules` or `package-lock.json`. Both are managed by npm and will get changed as you run npm commands from the terminal.

Importing an npm Module

npm modules can be imported into your script using `require`. To load in an npm module, pass the npm module name to `require`.

```
const validator = require('validator')

console.log(validator.isURL('https/mead.io')) // Print: true
```

The script above uses `require` to load in validator. The script then uses the `isURL` function provided by validator to check if a given string contains a valid URL.

Links

- npm
- npm: validator

Lesson 5: Printing in Color

There are npm modules for pretty much anything you'd want to do with Node.js. In this lesson, it's up to you to install and use a new one!

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Links

- npm: chalk

Lesson 6: Global npm Modules and nodemon

You can use npm modules from outside of your scripts by installing them globally. Globally installed modules are designed to be used from the terminal and provide you with new commands you can run. In this lesson, you'll learn how to install and work with global modules.

Installing an npm Module Globally

npm modules can be installed globally by adding a `-g` flag to the installation command. Not all modules are designed to be installed globally, so be sure to refer to the module documentation to learn how it's supposed to be used.

The command below installs version 1.18.5 of nodemon as a global module.

```
npm install -g nodemon@1.18.5
```

A globally installed module is not added as a dependency to your project. That means you won't see it listed in `package.json` or `package-lock.json`. You also won't find its code in `node_modules`. Globally installed modules are located in a special directory in your machine which is created and managed by npm.

When you install nodemon globally, you get access a new `nodemon` command from the terminal. This can be used to start and Node.js application and then restart the application any of the app scripts change. This means you won't need to switch between the terminal and text editor to restart your application every time you make a change.

The command below runs `app.js` through nodemon.

```
nodemon app.js
```

P.S. You can stop nodemon by using `ctrl + c` from the terminal!

Links

- npm: nodemon

Section 4: File System and Command Line Args

Lesson 1: Section Intro

It's time to start building your first Node.js application. In this section, you'll learn how to use the file system and command line arguments to create a note taking app. Along the way, you'll learn how to get input from the user, work with JSON, and create a place to store user data.

Lesson 2: Getting Input from Users

I can't think of a single useful application that doesn't get input from the users. Whether it's their email, location, or age, getting input is essential for creating real-world apps. In this lesson, you'll learn how to set up command line arguments that allow users to pass data into your application.

Accessing Command Line Arguments

Command line arguments are values passed into your application from the terminal. Your Node.js application can access the command line arguments that were provided using `process.argv`. This array contains at least two items. The first is the path to the Node.js executable. The second is the path to the JavaScript file that was executed. Everything after that is a command line argument.

Take a look at the example below.

```
const command = process.argv[2]

if (command === 'add') {
    console.log('Adding note!')
} else if (command === 'remove') {
    console.log('Removing note!')
}
```

That script grabs the third item in `process.argv`. Since the first two are always provided, the third item is the first command line argument that was passed in. The script uses the value of that argument to figure out what it should do. A user could provide `add` to add a note or `remove` to remove a note.

The command below runs the script and provides `add` as the command line argument.

```
$ node app.js add
Adding note!
```

Links

- process.argv

Lesson 3: Argument Parsing with Yargs: Part I

Node.js provides a bare-bones way to access command line arguments. While it's a good start, it doesn't provide any way to parse more complex command line arguments. In this lesson, you'll learn how to use Yargs to easily set up a more complex set of arguments for your application.

Setting Up Yargs

First, install Yargs in your project.

```
npm install yargs@12.0.2
```

Now, yargs can be used to make it easier to work with command line arguments. The example below shows how this can be done. First, `yargs.version` is used to set up a version for the command line tool. Next, `yargs.command` is used to add support for a new command.

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

yargs.version('1.1.0')

yargs.command({
    command: 'add',
    describe: 'Add a new note',
    handler: function () {
        console.log('Adding a new note!')
    }
})

console.log(yargs.argv)
```

Now, this command can be triggered by providing its name as a command line argument.

```
$ node app.js add
Adding a new note!
```

Yargs provides a couple useful commands by default. The first, shown below, lets a user get the version of the command line tool they're running.

```
$ node app.js --version
1.1.0
```

The second, shown below, shows the user autogenerated documentation that covers how the tool can be used. This would list out all available commands as well as the available options for each command.

```
$ node app.js --help
```

Links

- npm: yargs

Lesson 4: Argument Parsing with Yargs: Part II

In this lesson, you'll continue to explore Yargs. The goal is to allow users to pass in the title and body of their notes using command line options. This same technique could be used to allow users to pass in data such as their name, email, or address.

Adding Command Options

Options are additional pieces of information passed along with the command. You can set up options for a command using the `builder` property as shown below.

Now, the add command can be used with two options. The first is `title` which is used for the title of the note being added. The second is `body` which is used for the body of the note being added. Both options are required because `demandOption` is set to `true`. Both are also set up to accept string input because `type` is set to `'string'`.

```
yargs.command({
    command: 'add',
    describe: 'Add a new note',
    builder: {
        title: {
            describe: 'Note title',
            demandOption: true,
            type: 'string'
        },
        body: {
            describe: 'Note body',
            demandOption: true,
            type: 'string'
        }
    },
    handler: function (argv) {
        console.log('Title: ' + argv.title)
        console.log('Body: ' + argv.body)
    }
})
```

The add command can now be used with `--title` and `--body`.

```
$ node app.js add --title="Buy" --body="Note body here"
Title: Buy
Body: Note body here
```

Lesson 5: Storing Data with JSON

In this lesson, you'll learn how to work with JSON. JSON, which stands for JavaScript Object Notation, is a lightweight data format. JSON makes it easy to store or transfer data. You'll be using it in this application to store users notes in the file system.

Working with JSON

Since JSON is nothing more than a string, it can be used to store data in a text file or transfer data via an HTTP requests between two machines.

JavaScript provides two methods for working with JSON. The first is `JSON.stringify` and the second is `JSON.parse`. `JSON.stringify` converts a JavaScript object into a JSON string, while `JSON.parse` converts a JSON string into a JavaScript object.

```
const book = {
  title: 'Ego is the Enemy',
  author: 'Ryan Holiday'
}

// Covert JavaScript object into JSON string
const bookJSON = JSON.stringify(book)

// Covert JSON string into object
const bookObject = JSON.parse(bookJSON)
console.log(bookObject.title) // Print: Ego is the Enemy
```

JSON looks similar to a JavaScript object, but there are some differences. The most obvious is that all properties are wrapped in double-quotes. Single-quotes can't be used here, as JSON only supports double-quotes. You can see this in the example JSON below.

```
{"name":"Gunther","planet":"Earth","age":54}
```

Links

- JSON format

Lesson 6: Adding a Note

In this lesson, you'll be saving new notes to the file system.

There are no notes for this video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 7: Removing a Note

It's challenge time. In this lesson, you'll be adding the ability for users to remove notes they've added.

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 8: ES6 Aside: Arrow Functions

In this lesson, you'll learn how to use ES6 arrow functions. Arrow functions come with a few great features, making them a nice alternative to the standard ES5 function. You'll explore the new syntax and learn when to use them!

Arrow Functions

Arrow functions offer up an alternative syntax from the standard ES5 function. The snippet below shows an example of a standard function and then an arrow function. While the syntax is obviously different, you still have the two important pieces, an arguments list and a function body.

```
// const square function (x) {
//     return x * x
// }

const square = (x) => {
    return x * x
}

console.log(square(2))    // Will print: 4
```

Shorthand Syntax

Arrow functions have an optional shorthand syntax. This is useful when you have a function that immediately returns a value. The example below shows how this can be used.

```
const squareAlt = (x) => x * x


console.log(squareAlt(2)) // Will print: 4
```

Notice that two important things are missing from the function definition. First, the curly braces wrapping the function body have been removed as well as the `return` statement. In place of both is the value to be returned. There's no need for an explicit return statement, as the value provide is implicitly returned.

This Binding

Arrow functions don't bind their own `this` value. Instead, the `this` value of the scope in which it was defined is accessible. This makes arrow functions bad candidates for methods, as `this` won't be a reference to the object the method is defined on.

For methods, ES6 provides a new method definition syntax. You can see this in the definition of the `printGuestList` method below. That function is a standard function, just with a shorthand syntax which allows for the removal of the colon and the function keyword.

Because arrow functions don't bind this, they work well for everything except methods. As shown below, the arrow function passed to `forEach` is able to access `this.name` correctly, as it's defined as an arrow function and doesn't have a this binding of its own. That code wouldn't work if you swapped out the arrow function for a standard function.

```
const event = {
    name: 'Birthday Party',
    guestList: ['Andrew', 'Jen', 'Mike'],
    printGuestList() {
        console.log('Guest list for ' + this.name)

        this.guestList.forEach((guest) => {
            console.log(guest + ' is attending ' + this.name)
        })
    }
}

event.printGuestList()
```

Links

- Arrow function

Lesson 9: Refactoring to Use Arrow Functions

In this lesson, you'll use what you've learned about arrow functions to integrate them into the Node.js app.

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 10: Listing Notes

In this lesson, you'll create a new app feature that allows users to list out their notes.

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 11: Reading a Note

In this lesson, you'll add a new app feature that allows users to read a note.

Array Find method

The `find` method allows you to find a single item in an array. It's similar to `filter`, though `find` returns a single element as opposed to an array of elements. `find` will stop its search through the array after finding the first match.

The example below shows how find can be used to locate the user whose name is George Hudson.

```
const users = [{
  name: 'Andrew Mead',
  age: 27
},{
  name: 'George Hudson',
  age: 72
},{
  name: 'Clay Klay',
  age: 45
}]

const user = users.find((user) => user.name === 'George Hudson')

console.log(user) // Will print the second object in the array
```

Links

- Array find method

Section 5: Debugging Node.js

Lesson 1: Section Intro

What's worse than getting an error when you run your application? Not knowing how to fix it. In this section, you'll learn how to effectively debug your Node.js apps. You'll learn how to track down and fix issues so you can get back to the important work.

Lesson 2: Debugging Node.js

In this lesson, you'll learn how to debug your Node.js applications. Node comes with a great set of tools for getting to the bottom of any bug or programming issue.

Console.log

While it's nice to have advanced debugging tools at the ready, there's nothing wrong with using `console.log` to debug your application. It's not the fanciest technique, but it works, and I use it daily.

When in doubt, use a few calls to `console.log` to figure out what's going on. It's great for dumping a variable to the terminal so you can check its value. It also works for figuring out what order your code is running in.

Node Debugger

Printing values to the console with `console.log` is a good start, but there are often times where we need a more complete debugging solution. For that, Node.js ships with a built-in debugger. It builds off of the developer tools that Chrome and V8 use when debugging JavaScript code in the browser.

Start your application with `inspect` to use the debugger.

```
node inspect app.js
```

Next, visit `chrome://inspect` in the Chrome browser. There, you'll see a list of all the Node.js processes that you're able to debug. Click "inspect" next to your Node.js process to open up the developer tools. From there, you can click the blue "play" button near the top-right of the "sources" tab to start up the application.

When running the app in debug mode, you can add breakpoints into your application to stop it at a specific point in the code. This gives you a chance to explore to the application state and figure out what's going wrong.

```
console.log('Thing one)

debugger // Debug tools will pause here until your click play again

console.log('Thing two)
```

Documentation Links

- Node.js debugger documentation

Lesson 3: Error Messages

In this lesson, you'll learn how to read error messages. Error messages contain useful information about what went wrong, but they can be a pain to read. Learning how to read them will let you fix errors fast.

Error Messages

Error messages can be daunting to use at first. They contain a lot of useful information, but only if you know what you're looking at. Let's start with a complete error. Below is an error I generated by trying to reference a variable that was never defined.

```
/Users/Andrew/Downloads/n3-04-08-arrow-functions/playground/2-arrow-
function.js:21
            console.log(guest + ' is attending ' + eventName)
                                                   ^

ReferenceError: eventName is not defined
    at guestList.forEach (/Users/Andrew/Downloads/n3-04-08-arrow-
functions/playground/2-arrow-function.js:21:52)
    at Array.forEach (<anonymous>)
    at Object.printGuestList (/Users/Andrew/Downloads/n3-04-08-arrow-
functions/playground/2-arrow-function.js:20:24)
    at Object.<anonymous> (/Users/Andrew/Downloads/n3-04-08-arrow-
functions/playground/2-arrow-function.js:26:7)
    at Module._compile (internal/modules/cjs/loader.js:707:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:718:10)
    at Module.load (internal/modules/cjs/loader.js:605:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:544:12)
    at Function.Module._load (internal/modules/cjs/loader.js:536:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:760:12)
```

The first few lines of the error contain the most useful information.

The first line contains a path to the exact script where the error was thrown. It also contains the line number. Using that line, you could tell that the issue is on line 21 of `2-arrow-function.js`.

The second line shows the line of code that caused the error.

The third line just below uses the "^" character to point to the specific part of the line that the error came from.

The fourth line is blank.

The fifth line contains the error message from V8.

Everything after the fifth line is part of the stack trace. This shows a list of all the functions that were running to get to the point where the program crashed. The top of the stack trace starts with the function which threw the error. Here, we can see that the error was thrown in a callback function for a `forEach` method call. If you got down to the next line, you'll figure out that the `forEach` call happened inside of `printGuestList`.

It'll take a few tries to get comfortable with error messages. Each error you fix makes it easier to fix the next one.

Section 6: Asynchronous Node.js

Lesson 1: Section Intro

It's time to connect your application with the outside world. In this section, you'll explore the asynchronous nature of Node.js. You'll learn how to use asynchronous programming to make HTTP API requests to third-party HTTP APIs. This will allow you to pull in data, like real-time weather data, into your app.

Lesson 2: Asynchronous Basics

In this lesson, you'll explore the basics of asynchronous development. You'll get a preview of what asynchronous code looks like and how it's different from synchronous code.

Async 101

When running asynchronous code, your code won't always execute in the order you might expect. To get started with asynchronous development, let's use `setTimeout`.

`setTimeout` is a function that allows you to run some code after a specific amount of time has passed. `setTimeout` accepts two arguments. The first is a callback function. This function will run after the specified amount of time has passed. The second argument is the amount of time in milliseconds to wait.

Here's an example.

```
console.log('Starting')

// Wait 2 seconds before running the function
setTimeout(() => {
    console.log('2 Second Timer')
}, 2000)

console.log('Stopping')
```

Run the script and you'll see the logs in the following order.

```
$ node app.js
Starting
Stopping
2 Second Timer
```

Notice that “Stopping” prints before “2 Second Timer”. That’s because `setTimeout` is asynchronous and non-blocking. The `setTimeout` call doesn’t block Node.js from running other code while it’s waiting for the 2 seconds to pass.

This asynchronous and non-blocking nature makes Node.js ideal for backend development. Your server can wait for data from a database while also processing an incoming HTTP request.

Links

- setTimeout

Lesson 3: Call Stack, Callback Queue, and Event Loop

In this lesson, you’ll visualize how Node.js and V8 manage your asynchronous code. This includes the call stack, callback queue, event loop, and more!

This lesson contains a detailed presentation. Please refer to the video for a recap of how asynchronous programming works.

You can grab the slides here.

Lesson 4: Making HTTP Requests

In this lesson, you’ll learn how to make HTTP requests from Node. This will enable your app to communicate with other APIs and servers to do a wide variety of things. Everything from fetching real-time weather data to sending text messages to users.

Making HTTP Requests

There are several libraries that make it easy to fire off HTTP requests. My favorite is request. You can install it using the command below.

```
npm i request@2.88.0
```

Before you use the library in your app, you’ll need to figure out which URL you’re trying to fetch. To fetch real-time weather data, you’ll need to sign up for a free Dark Sky API account. You can do that here.

Below is an example URL that responds with forecast data for San Francisco.

https://api.darksky.net/forecast/9d1465c6f3bb7a6c71944bdd8548d026/37.8267,-122.4233

If you visit that URL in the browser, you'll see that the response is JSON data. This same data can be fetched by our Node.js app using the request library. The example below fetches the forecast data and prints the current temperature to the console.

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

const url =
'https://api.darksky.net/forecast/9d1465c6f3bb7a6c71944bdd8548d026/37.8267,-
122.4233'

request({ url: url }, (error, response) => {
    // Parse the response body from JSON string into JavaScript object
    const data = JSON.parse(response.body)

    // Will print the current temperature to the console
    console.log(data.currently.temperature)
})
```

Links

- npm: request

Lesson 5: Customizing HTTP Requests

In this lesson, you'll explore an option for the request library that allows it to automatically parse JSON data into a JavaScript object.

Request Options

The request library comes with plenty of options to make your life easier. One is the `json` option. Set `json` to `true` and request will automatically parse the JSON into a JavaScript object for you.

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

const url =
'https://api.darksky.net/forecast/9d1465c6f3bb7a6c71944bdd8548d026/37.8267,-
122.4233'

request({ url: url, json: true }, (error, response) => {
    console.log(response.body.daily.data[0].summary + ' It is currently ' +
response.body.currently.temperature + ' degrees out. There is a ' +
response.body.currently.precipProbability + '% chance of rain.')
})
```

The above program would print the following.

```
$ node app.js
Mostly cloudy overnight. It is currently 51.49 degrees out. There is a 0%
chance of rain.
```

There's a link below where you can explore all available options.

Links

- npm: request options

Lesson 6: An HTTP Request Challenge

It's challenge time. In this video, it's on you to integrate a geocoding API into the Node.js app.

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 7: Handling Errors

There are plenty of reasons an HTTP request can fail. Maybe your machine doesn't have an internet connection, or maybe the URL is incorrect. Regardless of what goes wrong, in this lesson, you'll learn how to handle errors that occur when making HTTP requests.

Handling Errors

Handling errors is important. It would be nice if we could always provide the user with a forecast for their location, but that's not going to happen. When things fail, you should aim to provide users with clear and useful messages in plain English so they know what's going on.

The callback function you pass to `request` expects an `error` and `response` argument to be provided. Either `error` or `response` will have a value, never both. If `error` has a value, that means things went wrong. In this case, `response` will be `undefined`, as there is no response. If `response` has a value, things went well. In this case, `error` will be `undefined`, as no error occurred.

The code below handles two different errors. The if statement first checks if `error` exists. If it does, the program prints a message letting the user know it was unable to connect. The second error occurs if there's no match for the given address. In that case, the program prints a message instructing the user to try a different search. Lastly, the coordinates are printed to the console if neither error occurs.

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

const geocodeURL =
'https://api.mapbox.com/geocoding/v5/mapbox.places/philadelphia.json?access_t
oken=pk.eyJ1IjoiYW5kcmV3bWVhZDEiLCJhIjoiY2pvOG8ybW90MDFhazNxcnJ4OTYydzJlOSJ9.
njY7HvaalLEVhEOIghPTlw&limit=1'

request({ url: geocodeURL, json: true }, (error, response) => {
    if (error) {
        console.log('Unable to connect to location services!')
    } else if (response.body.features.length === 0) {
        console.log('Unable to find location. Try another search.')
    } else {
        const latitude = response.body.features[0].center[0]
        const longitude = response.body.features[0].center[1]
        console.log(latitude, longitude)
    }
})
```

Lesson 8: The Callback Function

A callback function is a function that's passed as an argument to another function. That's it. This is something you've used before, and in this lesson, you'll dive a bit deeper into how they work.

The Callback Function

A callback function is a function that's passed as an argument to another function. Imagine you have FunctionA which gets passed as an argument to FunctionB. FunctionB will do some work and then call FunctionA at some point in the future.

Callback functions are at the core of asynchronous development. When you perform an asynchronous operation, you'll provide Node with a callback function. Node will then call the callback when the async operation is complete. This is how you get access to the results of the async operation, whether it's an HTTP request for JSON data or a query to a database for a user's profile.

The example below shows how you can use the callback pattern in your own code. The `geocode` function is set up to take in two arguments. The first is the address to geocode. The second is the callback function to run when the geocoding process is complete. The example below simulates this request by using `setTimeout` to make the process asynchronous.

```
const geocode = (address, callback) => {
    setTimeout(() => {
        const data = {
            latitude: 0,
            longitude: 0
        }

        callback(data)
    }, 2000)
}

geocode('Philadelphia', (data) => {
    console.log(data)
})
```

The call to `geocode` provides both arguments, the address and the callback function. Notice that the callback function is expecting a single parameter which it has called `data`. This is where the callback function will get access to the results of the asynchronous operation. You can see where `callback` is called with the data inside the `geocode` function.

Lesson 9: Callback Abstraction

Callback functions can be used to abstract complex asynchronous code into a simple reusable function. In this lesson, you'll learn how to use this pattern to create a reusable function for geocoding an address.

Callback Abstraction

Imagine you want to geocode an address from multiple places in your application. You have two options. Option one, you can duplicate the code responsible for making the request. This includes the call to request along with all the code responsible for handling errors. However, this isn't ideal. Duplicating code makes your application unnecessarily complex and difficult to maintain. The solution is to create a single reusable function that can be called whenever you need to geocode an address.

You can see an example of this below. The function `geocode` was created to serve as a reusable way to geocode an address. It contains all the logic necessary to make the request and process the response. `geocode` accepts two arguments. The first is the address to geocode. The second is a callback function which will run once the geocoding operation is complete.

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

const geocode = (address, callback) => {
    const url = 'https://api.mapbox.com/geocoding/v5/mapbox.places/' +
address +
'.json?access_token=pk.eyJ1IjoiYW5kcmV3bWVhZDEiLCJhIjoiY2pvOG8ybW90MDFhazNxcn
J4OTYydzJlOSJ9.njY7HvaalLEVhEOIghPTlw&limit=1'

    request({ url: url, json: true }, (error, response) => {
        if (error) {
            callback('Unable to connect to location services!', undefined)
        } else if (response.body.features.length === 0) {
            callback('Unable to find location. Try another search.',
undefined)
        } else {
            callback(undefined, {
                latitude: response.body.features[0].center[0],
                longitude: response.body.features[0].center[1],
                location: response.body.features[0].place_name
            })
        }
    })
}

module.exports = geocode
```

Now, `geocode` can be called as many times as needed from anywhere in your application. The snippet below imports `geocode` and calls the function to get the latitude and longitude for Boston.

```
const geocode = require('./utils/geocode')

geocode('Boston', (error, data) => {
    console.log('Error', error)
    console.log('Data', data)
})
```

Lesson 10: Callback Abstraction Challenge

It's challenge time. In this lesson, it's on you to create a reusable function to fetch a weather forecast.

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 11: Callback Chaining

In this lesson, you'll learn how to run one asynchronous operation only after another asynchronous operation is complete. That'll allow you to use the output from geocoding as the input for fetching the weather.

Callback Chaining

When working with async code, you'll often find out that you need to use the results from one async operation as the input for another async operation. This is something we need to do in the weather application too. Step one is to geocode the address. Step two is to use the coordinates to fetch the weather forecast. You can't start step two until step one is complete.

You can start one operation after another finishes by using callback chaining. You can see an example of this in the code below.

```
// Other lines hidden for brevity

geocode(address, (error, data) => {
    if (error) {
        return console.log(error)
    }

    forecast(data.latitude, data.longitude, (error, forecastData) => {
        if (error) {
            return console.log(error)
        }

        console.log(data.location)
        console.log(forecastData)
    })
})
```

First up is the call to `geocode`. The call to `geocode` provides an address and a callback function as it did before. It's the code inside the callback function that looks a bit different. The callback function calls `forecast`. This means that `forecast` won't get called until after `geocode` is complete. The latitude and longitude from the geocoding operation is also provided as the input for the `forecast` function call.

Lesson 12: ES6 Aside: Object Property Shorthand and Destructuring

ES6 has done wonders making JavaScript easier to use. In this lesson, you'll explore a couple of features that make it easier to work with objects.

Property Shorthand

The property shorthand makes it easier to define properties when creating a new object. It provides a shortcut for defining a property whose value comes from a variable of the same name. You can see this in the example below where a `user` object is created. The `name` property gets its value from a variable also called `name`.

```
const name = 'Andrew'
const userAge = 27

const user = {
    name: name,
    age: userAge,
    location: 'Philadelphia'
}
```

The shorthand allows you to remove the colon and the reference to the variable. When JavaScript sees this, it'll get the property value from the variable with the same name. The example below uses the property shorthand to define `name` on the `user` object.

```
const name = 'Andrew'
const userAge = 27

const user = {
    name,
    age: userAge,
    location: 'Philadelphia'
}

console.log(user)
```

Object Destructuring

The second ES6 feature is object destructuring. Object destructuring gives you a syntax for pulling properties off of objects and into standalone variables. This is useful when working with the same object properties throughout your code. Instead of writing `user.name` a dozen times, you could destructure the property into a `name` variable.

You can see an example of this below.

```
const user = {
  name: 'Andrew',
  age: 27,
  location: 'Philadelphia'
}

// The line below uses destructuring
const { age, location:address } = user

console.log(age)
console.log(address)
```

`user` is destructured on line 8 above. The `age` property has been destructured and stored in `age`. The `location` property has also been destructured and stored in `address`.

Destructuring Function Arguments

Destructuring works with function parameters as well. If an object is passed into a function, it can be destructured inside the function definition. You can see this in the `transaction` function below. The function accepts an object as its second argument. The `label` and `stock` properties have both been destructured into standalone variables that become available in the function.

```
const product = {
    label: 'Red notebook',
    price: 3,
    stock: 201,
    salePrice: undefined,
    rating: 4.2
}

const transaction = (type, { label, stock }) => {
    console.log(type, label, stock)
}

transaction('order', product)
```

Links

- Destructuring
- Property shorthand

Lesson 13: Destructuring and Property Shorthand Challenge

In this video, it's on you to use the property shorthand and object destructuring syntax in your Node.js app.

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 14: Bonus: HTTP Requests Without a Library

While the request library is great, it's not necessary if you want to make HTTP requests from Node. In this bonus lesson, you'll learn how to make an HTTP request without request.

The HTTPS Module

Node.js provides two core modules for making HTTP requests. The `http` module can be used to make http requests and the `https` module can be used to make https requests. One great feature about request is that it provides a single module that can make both http and https requests.

The code below uses the https module to fetch the forecast from the Dark Sky API. Notice there's a lot more required to get things working. Separate callbacks are required for incoming chunks of data, the end of the response, and the error for the request. This means you'll likely recreate your own function similar to `request` to make your life easier. It's best to stick with a tested and popular library like request.

```
const https = require('https')
const url =
'https://api.darksky.net/forecast/9d1465c6f3bb7a6c71944bdd8548d026/40,-75'

const request = https.request(url, (response) => {
    let data = ''

    response.on('data', (chunk) => {
        data = data + chunk.toString()
    })

    response.on('end', () => {
        const body = JSON.parse(data)
        console.log(body)
    })

})

request.on('error', (error) => {
    console.log('An error', error)
})

request.end()
```

Links

- Node.js http documentation
- Node.js https documentation

Section 7: Web Servers

Lesson 1: Section Intro

Node.js is commonly used as a web server to serve up websites, JSON, and more. In this section, you'll be creating your first Node server with Express. This will allow users to interact with your application by visiting a URL in the browser.

Lesson 2: Hello Express!

Serving up websites and JSON data is easy with Express. In this lesson, you'll learn how to create your first web server with Express. Once the server is up and running, users will be able to interact with your application via the browser.

Express 101

To get started, add Express to your project.

```
npm i express@4.16.4
```

Next, you can require express. You get access to a single function you can call to create a new Express application.

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

const app = express()
```

Now, `app` can be used to set up the server. Let's start by showing a message when someone visits the home page at `localhost:3000` and the weather page at `localhost:3000/weather`.

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

app.get('/weather', (req, res) => {
    res.send('Your weather')
})
```

The code above uses `app.get` to set up a handler for an HTTP GET request. The first argument is the path to set up the handler for. The second argument is the function to run when that path is visited. Calling `res.send` in the route handler allows you to send back a message as the response. This will get shown in the browser.

The last thing to do is start the server. This is done by calling `app.listen` with the port you want to listen on.

This can be done using `app.listen` as shown below.

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

If you run the app, you'll see the message printing letting you know that the server is running. This process will stay running until you shut it down. You can always use `ctrl + c` to terminate the process. Visit `localhost:3000` or `localhost:3000/weather` to view the messages!

```
$ node app.js
Server is up on port 3000.
```

Links

- Express

Lesson 3: Serving up HTML and JSON

With the basics out of the way, it's time to serve up HTML and JSON with Express. That'll let you serve up a static website or create an HTTP REST API designed to be consumed by a web or mobile application.

Serving up HTML and JSON

Using `res.send`, you can send back more than just text. `res.send` can be used to send an HTML or JSON response. The root route below sends back some HTML to be rendered in the browser. The weather route below sends back a JSON response.

```
app.get('', (req, res) => {
    // Provide HTML to render in the browser
    res.send('<h1>Weather</h1>')
})

app.get('/weather', (req, res) => {
    // Provide an object to send as JSON
    res.send({
        forecast: 'It is snowing',
        location: 'Philadelphia'
    })
})
```

Documentation Links

- express - res.send

Lesson 4: Serving up Static Assets

Express can serve up all the assets needed for your website. This includes HTML, CSS, JavaScript, images, and more. In this lesson, you'll learn how to serve up an entire directory with Express.

Serving up a Static Directory

A modern website is more than just an HTML file. It's styles, scripts, images, and fonts. Everything needs to be exposed via the web server so the browser can load it in. With Express, it's easy to serve up an entire directory without needing to manually serve up each asset. All Express needs is the path to the directory it should serve.

The example below uses Nodes' path module to generate the absolute path. The call to `path.join` allows you to manipulate a path by providing individual path segments. It starts with `__dirname` which is the directory path for the current script. From there, the second segment moves out of the `src` folder and into the `public` directory.

The path is then provided to `express.static` as shown below.

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

const app = express()
const publicDirectoryPath = path.join(__dirname, '../public')

app.use(express.static(publicDirectoryPath))

app.get('/weather', (req, res) => {
    res.send({
        forecast: 'It is snowing',
        location: 'Philadelphia'
    })
})

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

Start the server, and the browser will be able to access all assets in the public directory.

Documentation Links

- path

Lesson 5: Serving up CSS, JS, Images, and More

In this lesson, you'll use the Express server to serve up a webpage with images, styles, and scripts.

Serving up CSS, JS, Images, and More

All files in `public` are exposed via the Express server. This is where your site assets need to live. If they're not in `public`, then they're not public and the browser won't be able to load them correctly. The HTML file below shows how you can use a CSS file, JavaScript file, and image in your website.

```
<!DOCTYPE html>

<html>

<head>
    <link rel="stylesheet" href="/css/styles.css">
    <script src="/js/app.js"></script>
</head>

<body>
    <h1>About</h1>
    <img src="/img/me.png">
</body>

</html>
```

Lesson 6: Dynamic Pages with Templating

Your web pages don't have to be static. Express supports templating engines that allow you to render dynamic HTML pages. In this lesson, you'll learn how to set up the Handlerbars templating engine with Express.

Setting up Handlebars

Start by installing Handlebars in your project.

```
npm i hbs@4.0.1
```

From there, you'll need to use `app.set` to set a value for the `'view engine'` config option. The value is the name of the template engine module you installed. That's `'hbs'`.

```
app.set('view engine', 'hbs')
```

Rendering Handlebars Templates

By default, Express expects your views to live in a `views` directory inside of your project root. You'll learn how to customize the location and directory name in the next lesson.

Below is an example handlebars view in `views/index.hbs`. This looks like a normal HTML document with a few new features. Notice `{{title}}` and `{{name}}`. This is a Handlerbars syntax which allows you to inject variables inside of the template. This is what allows you to generate dynamic pages.

```
<!DOCTYPE html>

<html>

<head>
    <link rel="stylesheet" href="/css/styles.css">
    <script src="/js/app.js"></script>
</head>

<body>
    <h1>{{title}}</h1>
    <p>Created by {{name}}</p>
</body>

</html>
```

Now, you can render the template. This is done by defining a new route and calling `res.render` with the template name. The ".hbs" file extension can be left off. The second argument is an object that contains all the variables the template should have access to when rendering. This is where values are provided for `title` and `name`.

```
app.get('', (req, res) => {
    res.render('index', {
        title: 'My title',
        name: 'Andrew Mead'
    })
})
```

Documentation Links

- Handlebars documentation
- npm: hbs

Lesson 7: Customizing the Views Directory

In this lesson, you'll learn how to customize the name and location of the views directory.

Customizing the Views Directory

You can customize the location of the views directory by providing Express with the new path. Call `app.set` to set a new value for the `'views'` option. The example below configures Express to look for views in `templates/views/`.

```
const viewsPath = path.join(__dirname, '../templates/views')
app.set('views', viewsPath)
```

Lesson 8: Advanced Templating

In this lesson, you'll learn how to work with Handlerbars partials. As the name suggests, partials are just part of a web page. Partials are great for things you need to show on multiple pages like headers, footers, and navigation bars.

Setting up Partials

You can use partials by telling Handlebars where you'd like to store them. This is done with a call to `hbs.registerPartials`. It expects to get called with the absolute path to the partials directory.

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

// Other lines hidden for brevity

const partialsPath = path.join(__dirname, '../templates/partials')
hbs.registerPartials(partialsPath)

// Other lines hidden for brevity
```

Using Partials

Partials are created with the "hbs" file extension. Partials have access to all the same features as your Handlebars templates. The header partial below renders the title followed by a list of navigation links which can be shown at the top of every page.

```
{{!-- header.hbs --}}
<h1>{{title}}</h1>

<div>
    <a href="/">Weather</a>
    <a href="/about">About</a>
    <a href="/help">Help</a>
</div>
```

The partial can then be rendered on a page using `{{>header}}` where “header” comes from the partial file name. If the partial was `footer.hbs`, it could be rendered using `{{>footer}}`

```
<!DOCTYPE html>

<html>

<head>
    <link rel="stylesheet" href="/css/styles.css">
    <script src="/js/app.js"></script>
</head>

<body>
    {{>header}}
</body>

</html>
```

Lesson 9: 404 Pages

In this lesson, you’ll learn how to set up a 404 page. The 404 page will show when a user tries to visit a page that doesn’t exist.

Setting up a 404 Page

Express has support for `*` in route paths. This is a special character which matches anything. This can be used to create a route handler that matches all requests.

The 404 page should be set up just before the call to `app.listen`. This ensures that requests for valid pages still get the correct response.

```
app.get('*', (req, res) => {
    res.render('404', {
        title: '404',
        name: 'Andrew Mead',
        errorMessage: 'Page not found.'
    })
})
```

Lesson 10: Styling the Application: Part I

In this lesson, you'll add some styles to the weather application.

There are no notes for this styling video, as no new Node.js features are covered.

Lesson 11: Styling the Application: Part II

In this lesson, you'll finish styling the weather application.

There are no notes for this styling video, as no new Node.js features are covered.

Section 7: Accessing API from Browser

Lesson 1: Section Intro

In this section, you'll learn how to set up communication between the client and the server. This will be done via HTTP requests. By the end of the section, users will be able to type an address in the browser to view their forecast.

Lesson 2: The Query String

In this lesson, you'll learn how to use query strings to pass data from the client to the server. This will be used to send the address from the browser to Node.js. Node.js will then be able to fetch the weather for the address and send the forecast back to the browser.

Working with Query Strings

The query string is a portion of the URL that allows you to provide additional information to the server. For the weather application, the query string will be used to pass the address from the browser to the Node.js Express application.

The query string comes after `?` in the URL. The example URL below uses the query string to set `address` equal to `boston`. The key/value pair is separated by `=`.

```
http://localhost:3000/weather?address=boston
```

Below is one more example where two key/value pairs are set up. The key/value pairs are separated by `&`. `address` equals `philadelphia` and `units` equals `us`.

```
http://localhost:3000/weather?address=philadelphia&units=us
```

The Express route handler can access the query string key/value pairs on `req.query`. The handler below uses `req.query.address` to get the value provided for `address`. This address can then be used to fetch the weather information.

```
app.get('/weather', (req, res) => {
    // All query string key/value pairs are on req.query
    res.send('You provided "' + req.query.address + '" as the address.)
})
```

Documentation Links

- express - req.query

Lesson 3: Building a JSON HTTP Endpoint

The weather application already has the code in place to fetch the weather for a given address. In this lesson, it's your job to wire up the route handler to fetch the weather and send it back to the browser.

There are no notes for this challenge video, as no new information is covered. The goal is to give you experience using what was covered in previous lessons.

Lesson 4: ES6 Aside: Default Function Parameters

ES6 provides a new syntax to set default values for function arguments. In this lesson, you'll use this new syntax to improve and clean up the application code.

Default Function Parameters

Function parameters are `undefined` unless an argument value is provided when the function is called. ES6 now allows function parameters to be configured with a custom default value.

You can see this in action for the `greeter` function below. `name` will be `'user'` if no value is provided. `age` will be `undefined` if no value is provided.

```
const greeter = (name = 'user', age) => {
    console.log('Hello ' + name)
}

greeter('Andrew') // Will print: Hello Andrew

greeter() // Will print: Hello user
```

This syntax can also be used to provide default values when using ES6 destructuring. The `transaction` function below shows this off by providing a default value for `stock`.

```
const transaction = (type, { label, stock = 0 } = {}) => {
    console.log(type, label, stock)
}

transaction('order')
```

Documentation Links

- mdn: default function parameters

Lesson 5: Browser HTTP Requests with Fetch

In this lesson, you'll learn how to make HTTP AJAX requests from the browser. This will allow the web application to request the forecast from the Node.js server.

The Fetch API

Web APIs provide you with a way to make HTTP requests from JavaScript in the browser. This is done using the `fetch` function. `fetch` expects to be called with the URL as the first argument. It sends off the HTTP request and gives you back the response.

The `fetch` call below is used to fetch the weather for Boston. An if statement is then used to either print the forecast or the error message.

```
fetch('http://localhost:3000/weather?address=Boston').then((response) => {
     response.json().then((data) => {
         if (data.error) {
             console.log(data.error)
         } else {
             console.log(data.location)
             console.log(data.forecast)
         }
     })
 })
```

Documentation Links

- Fetch API
- Fetch Tutorial

Lesson 6: Creating a Search Form

In this lesson, you'll set up the weather search form. This will allow a visitor to type in their address, click a button, and then see their real-time forecast information.

The Search Form

Below is an example HTML form. It contains a text input and a button which can be used to submit the form.

```
<form>
    <input placeholder="Location">
    <button>Search</button>
</form>
```

Using client-side JavaScript, you can set up an event listener that will allow you to run some code when the form is submitted. What should that code do? It should grab the address from the text field, send off an HTTP request to the Node server for the data, and then render the weather data to the screen.

For the moment, the data is logged to the console. That'll get fixed in the next lesson.