# 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.

```
const weatherForm = document.querySelector('form')
const search = document.querySelector('input')

weatherForm.addEventListener('submit', (e) => {
    e.preventDefault()

    const location = search.value

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

## Lesson 7: Wiring up the User Interface

In this lesson, you'll learn how to manipulate the text content of HTML elements from JavaScript. That will allow the weather application to render the forecast data to the browser instead of the console.

### Rendering Dynamic textContent

Set up HTML elements for the messages you want to render. Below is an example paragraph which can be used to render some text from JavaScript. It contains no text by default. It will be updated to show some text as the client-side JavaScript runs.

```
<p id="message-1"></p>
```

The code below can be used to change the text content of the paragraph. First up, `document.querySelector` is used to target the element. It's used with `#`, which searches for elements by their ID. The text content can be updated by setting a value on the `textContent` property.

```
const messageOne = document.querySelector('#message-1')
messageOne.textContent = 'My new text'
```

# Section 7: Application Deployment

## Lesson 1: Section Intro

In this section, you'll learn how to deploy your Node.js applications to production. This will allow anyone with an internet connection to view and interact with your Node.js app. Along the way, you'll learn how to use Git, GitHub, Heroku, and more!

## Lesson 2: Joining Heroku and GitHub

In this lesson, you'll join GitHub and Heroku. GitHub is a development platform that makes it easy to manage software development projects. Heroku is an application deployment platform which provides everything needed to deploy your Node.js applications.

### Joining Heroku and GitHub

Make sure to sign up for an account with both GitHub and Heroku. From there, install the Heroku CLI. After running the installer, you can confirm the Heroku CLI was installed correctly by running `heroku -v` to print the installed version.

The Heroku CLI gives you commands to deploy and manage your Node.js applications. Before you can do that, you'll need to log in to your Heroku account. This makes sure that the command you run actually changes your Heroku applications.

```
heroku login
```

### Documentation Links

- GitHub
- Heroku
- Heroku CLI download

## Lesson 3: Version Control with Git

In this lesson, you'll learn about version control. Version control allows you to track changes to your project code over time. This makes it easy to recover lost code and restore your project to a previously working version.

### Version Control with Git

Version control lets you track changes to your application code over time. It's an important tool, and it should be used for all personal and professional projects.

Imagine you have an application with 250 paying users. You just finished work on a great new feature and you deploy it to production so your customers can use it. Hours later, you discover a bug that's preventing users from using the application. What do you do next?

Without version control, you're in trouble. The only version of your app is the one you have on your machine. The buggy application that's crashing for your users. You have no way of getting back to the old version of your app that was working. Users are stuck with a broken application until you can fix the bug and get a new version of the app deployed.

With version control, you're in the clear. You can revert back to your application's previous working state and deploy that. This means that users can continue to use the original version while you can take a breath and get back to working on that new feature until it's ready.

You can grab the Git installer from git-scm.com. After installing Git, run `git --version` to print the version of Git installed.

### Documentation Links

- Git

## Lesson 4: Exploring Git

Git is not the easiest tool in the world to get started with. In this lesson, you'll explore how Git works and how it can help you keep track of code changes.

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

## Lesson 5: Integrating Git

It's time to start using Git. In this lesson, you'll learn how to set up Git in your project. You'll also explore the commands needed to get Git tracking your code.

### Initializing Git

Git needs to be initialized in your project before it can be used. You can initialize Git in your project by running `git init` from the root of the project. All Git commands should be run from the root of the project.

Before going any further, Git needs to be configured to ignore this `node_modules` folder. This is a generated directory which doesn't need to be under version control. You can always regenerate `node_modules` by running `npm install`. Create a `.gitignore` file with the following line to ignore the folder.

```
node_modules/
```

### Committing Changes

Think of a commit as a save point. A commit lets you create a save point that contains your project files exactly as they were when the commit was created. You'll create new commits to track your changes as you continue to build out your application.

Before creating a commit, it's a good idea to run `git status` to get a summary of the changes that are about to be committed. This will show untracked files, unstaged changes, and staged files.

Using `git add <path to file>`, you can add files to the staging area. Changes to files in the staging area will be included in the next commit. The shortcut below adds all untracked files and unstaged changes to the staging area.

```
git add .
```

You can now use the `git commit` command to create a commit. Each commit requires a commit message. The command below creates a commit and provides "Initial commit" as the commit message.

```
git commit -m "Initial commit"
```

From here, you can continue to add new features to the project and use the git commands to create new commits.

## Lesson 6: Setting up SSH Keys

In this lesson, you'll be setting up SSH on your machine. SSH is the protocol used to securely transfer code between your machine and GitHub/Heroku.

### Creating SSH Keys

Windows users won't have access to the necessary SSH commands from the command prompt. Make sure to use Git Bash for the following commands.

SSH uses an SSH key pair to secure the connection between your machine and the machine you're communicating with. You can check if you already have an SSH key pair with the following command. You have a key pair if you see `id_rsa` and `id_rsa.pub` in the output.

```
ls -a -l ~/.ssh
```

You can create a new key pair using the following command. Make sure to swap out the email for your email address.

```
ssh-keygen -t rsa -b 4096 -C "youremail@domain.com"
```

The SSH key needs to be configured to be used for new SSH connections. First, ensure that the SSH agent is running. You can do that using the command below.

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

Next, add the new SSH private key file to the SSH agent. The following command is for macOS users.

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

The command below is for Linux users and Windows users.

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

## Lesson 7: Pushing Code to GitHub

In this lesson, you'll learn how to push your code to GitHub.

### Configuring SSH with GitHub

In the last lesson, you generated the SSH key pair. The files were `id_rsa` and `id_rsa.pub`. `id_rsa` is a private key file which should be kept secret. `id_rsa.pub` is a public key file which should be shared with the services you plan to communicate with.

The command below will allow you to dump the contents of the public key file to the terminal. Copy and paste the contents to the clip board and register the SSH key with GitHub here.

```
cat ~/.ssh/id_rsa.pub
```

### Pushing Your Code to GitHub

You need to create a new GitHub repository before you'll be able to push your code. This is a remote Git repository that'll live on the GitHub server. A remote repository is nothing more than a version of your project hosted somewhere else. In this case, it's a version of your project stored on GitHub.

Once the repository is created, you'll need to set up the origin remote. Replace `<repo url>` with the repository URL provided by GitHub.

```
git remote add origin <repo url>
```

You can now push your latest commits to the remote! After pushing your commits, refresh the GitHub repository page in your browser to see your project files and folder appear.

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

## Lesson 8: Deploying Node.js to Heroku

In this lesson, you'll deploy your application to Heroku. Anyone with an internet connection will be able to access and use your application!

### Preparing Your Application

Heroku makes it easy to deploy your application to Node.js, but it does require a few small changes. First, Heroku needs to know what command to run to start your app. Second, Heroku requires your app to listen on a specific port.

The `start` script in `package.json` is used to tell Heroku which command to run. Set `start` equal to `node src/app.js` to ensure that Heroku can start your app up correctly.

Heroku uses an environment variable to provide the port value you need to listen on. The code below accesses the Heroku port value and uses it to start up the server.

```
const port = process.env.PORT || 3000

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

### Deploying Your Application

Run `heroku create` from your application root to create a new application. This will create the new application and set up a new `heroku` Git remote. Push your code to that remote to deploy the application!

You can run `git pus heroku master` to deploy. From there, run `heroku open` to open your application in the browser.

## Lesson 9: New Feature Deployment Workflow

In this lesson, you'll go through the process of adding a new feature to the application. This includes committing the changes, pushing them to GitHub, and deploying them to Heroku.

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: Avoiding Global Modules

In this lesson, you'll refactor your application to remove the use of global modules. This ensures that your application installs all the dependencies you need to run.

### Replacing Global Modules with Local Modules

Sick of typing out that long nodemon command? Me too. Let's turn it into a script.

You can create a `dev` script with the value `nodemon src/app.js -e js,hbs`. This will start up the dev server anytime you run `npm run dev`.

The dev script needs nodemon to be installed. The issue is that nodemon isn't listed as a dependency in `package.json`. However, this can be fixed by uninstalling nodemon globally.

```
npm uninstall -g nodemon
```

Now, install it as a local dependency.

```
npm install nodemon
```

Now, `npm install` will be able to install all your application dependencies, including nodemon!

# Section 10: MongoDB and Promises

## Lesson 1: Section Intro

In this section, you'll learn how to connect to a MongoDB database from your Node.js application. This will allow your application to store data in a secure and reliable fashion. The task application will use MongoDB to store user accounts as well as tasks.

## Lesson 2: MongoDB and NoSQL Databases

In this lesson, you'll explore NoSQL databases and MongoDB. MongoDB and NoSQL database are a bit different than traditional SQL databases such as MySQL, so this lesson will bring you up to speed on some key ideas and terminologies related to NoSQL.

This lesson contains a detailed presentation. Please refer to the video for a recap of NoSQL and MongoDB.

## Lesson 3: Installing MongoDB on macOS and Linux

In this lesson, you'll learn how to install MongoDB on macOS and Linux. Using Windows? That's covered in the next lesson.

You can download the MongoDB Community Server from the MongoDB download page. The download is a zip file. Unzip the contents, change the folder name to “mongodb”, and move it to your users home directory. From there, create a “mongodb-data” directory in your user directory to store the database data.

You can start the server using the following command. Make sure to swap out “/Users/Andrew/” with the correct path to your users home directory.

```
/Users/Andrew/mongodb/bin/mongod --dbpath=/Users/Andrew/mongodb-data
```

Documentation Links

- MongoDB download page

## Lesson 4: Installing MongoDB on Windows

In this lesson, you’ll learn how to install MongoDB on Windows. Using macOS or Linux? They were covered in the previous lesson.

You can download the MongoDB Community Server from the MongoDB download page. The download is a zip file. Unzip the contents, change the folder name to “mongodb”, and move it to your users home directory. From there, create a “mongodb-data” directory in your user directory to store the database data.

You can start the server using the following command. Make sure to swap out “/Users/Andrew/” with the correct path to your users home directory.

```
/Users/Andrew/mongodb/bin/mongod --dbpath=/Users/Andrew/mongodb-data
```

Documentation Links

- MongoDB download page

## Lesson 5: Installing Database GUI Viewer

In this lesson, you’ll set up Robo 3T. Robo 3T is a MongoDB admin tool that makes it easy to manage and visualize the data in your database.

Robo 3T is a completely free MongoDB admin tool. Grab the installer from here and get it installed on your machine.

### Documentation Links

- Robo 3T download page

## Lesson 6: Connecting and Inserting Documents

In this lesson, you'll be connecting to your MongoDB database from your Node.js application. You'll also learn how to insert documents into the database to save them for later.

### Connecting to MongoDB

MongoDB provides a native driver that allows you to connect to your database from Node. You can grab the driver by installing the mongodb npm module as shown below.

```
npm i mongodb@3.1.10
```

With the driver installed, you can use the following code to connect to the database. You just need to provide two pieces of information. The first is the connection URL and the second is the name of the database. You can pick any database name that you like.

```
const mongodb = require('mongodb')
const MongoClient = mongodb.MongoClient

const connectionURL = 'mongodb://127.0.0.1:27017'
const databaseName = 'task-manager'

MongoClient.connect(connectionURL, { useNewUrlParser: true }, (error, client)
=> {
    if (error) {
        return console.log('Unable to connect to database!')
    }

    const db = client.db(databaseName)

    // Start to interact with the database
})
```

### Inserting a Document

With the connection open, you're ready to insert documents into the database. Remember that a database is made up of collections, and collections are used to store documents. The code below inserts a new document into the "users" collection. `db.collection` is

used to get a reference to the collection you're trying to manipulate. `insertOne` is used to insert a new document into that collection.

```
db.collection('users').insertOne({
    name: 'Andrew',
    age: 27
})
```

### Documentation Links

- npm: mongodb
- MongDB driver documentation

## Lesson 7: Inserting Documents

In this lesson, you'll explore another way to insert documents into collections.

### Inserting Documents

You already know that `insertOne` can be used to insert a single document. You can also use `insertMany` to insert multiple documents at once. The example below inserts two documents into "tasks" collection. `insertMany` expects an array of objects, an array of the documents you want to insert.

```
db.collection('tasks').insertMany([
    {
        description: 'Clean the house',
        completed: true
    },{
        description: 'Renew inspection',
        completed: false
    }
], (error, result) => {
    if (error) {
        return console.log('Unable to insert tasks!')
    }

    console.log(result.ops)
})
```

### Documentation Links

- insertOne

- insertMany

## Lesson 8: The ObjectID

In this lesson, you'll learn about ObjectIDs. MongoDB uses ObjectIDs to create unique identifiers for all the documents in the database. It's different than the traditional auto-incrementing integer ID, but it comes with its own set of advantages.

### Working with ObjectIDs

MongoDB provides `ObjectID` which can be used to generate new ObjectIDs. The example below generates a new ID and prints it to the console.

```
const { MongoClient, ObjectID } = require('mongodb')
const id = new ObjectID()
console.log(id) // Print new id to the console
```

An ObjectID is a GUID (Globally Unique Identifier). GUIDs are generated randomly via an algorithm to ensure uniqueness. These IDs can be generated on the server, but as seen in the snippet above, they can be generated on the client as well. That means a client can generate the ID for a document it's about to insert in to the database.

## Lesson 9: Querying Documents

In this lesson, you'll learn how to read data from MongoDB. This will allow you to fetch the documents that you had previously inserted.

### Finding Documents

You can search for documents in a given collection using `find` or `findOne`. `find` can be used to fetch multiple documents, while `findOne` can be used to fetch a single document.

The example below uses `find` to search for documents in the tasks collection. You can provide an object as the first argument to `find` to filter the documents. The example below sets `completed` equal to `false` to fetch only those tasks that haven't been completed.

```
db.collection('tasks').find({ completed: false }).toArray((error, tasks) => {
    console.log(tasks)
})
```

The next example uses `findOne` to find a single document by its ID. In this case, it's necessary to pass the string version of the ID to the `ObjectID` constructor function to convert it to an ObjectID.

```
db.collection('tasks').findOne({ _id: new
ObjectID("5c0fec243ef6bdfbe1d62e2f") }, (error, task) => {
    console.log(task)
})
```

Documentation Links

- find
- findOne

## Lesson 10: Promises

In this lesson, you'll learn how to work with promises. Promises provide a much needed alternative to the traditional callback pattern.

```
    const doWorkPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve([7, 4, 1])
        // reject('Things went wrong!')
    }, 2000)
})

doWorkPromise.then((result) => {
    console.log('Success!', result)
}).catch((error) => {
    console.log('Error!', error)
})
```

## Lesson 11: Updating Documents

In this lesson, you'll learn how to update documents stored in MongoDB.

### Updating Documents

You can update documents in a collection using `updateOne` or `updateMany`. The first argument for both `updateOne` and `updateMany` is similar to the first argument used with `find` or `findOne`. It's an object that allows you to filter down all the documents to just the ones you want to update.

The update calls require a second argument as well. This is an object where you define the updates you want to make. For this, you need to use one of the supported update operators.

The `updateOne` call below uses `$inc` to increment the `age` field on the targeted document by 1.

```
db.collection('users').updateOne({
    _id: new ObjectID("5c0fe6634362c1fb75b9d6b5")
}, {
    $inc: {
        age: 1
    }
}).then((result) => {
    console.log(result)
}).catch((error) => {
    console.log(error)
})
```

The `updateMany` call below uses `$set` to set the `completed` field to `true` for all documents where the `completed` field is currently `false`.

```
db.collection('tasks').updateMany({
        completed: false
    }, {
        $set: {
            completed: true
        }
    }).then((result) => {
        console.log(result.modifiedCount)
    }).catch((error) => {
        console.log(error)
```

### Documentation Links

- updateOne

- updateMany
- Update operators

## Lesson 12: Deleting Documents

In this lesson, you'll learn how to delete documents stored in MongoDB.

### Deleting Documents

You can delete documents from MongoDB using `deleteOne` or `deleteMany`. Both accept an object as the first argument. This object is used to filter just the documents you want to delete.

The example below uses `deleteMany` to delete all users whose `age` field is `27`.

```
db.collection('users').deleteMany({
    age: 27
}).then((result) => {
    console.log(result)
}).catch((error) => {
    console.log(error)
})
```

This next example uses `deleteOne` to delete a single document, the first one with a `description` of `"Clean the house"`.

```
db.collection('tasks').deleteOne({
    description: "Clean the house"
}).then((result) => {
    console.log(result)
}).catch((error) => {
    console.log(error)
})
```

### Documentation Links

- deleteOne
- deleteMany

# Section 11: REST APIs and Mongoose

## Lesson 1: Section Intro

In this section, you'll be creating a REST API using Express. You'll learn what exactly a REST API is and how it can be used as the back-end for a web or mobile application. This section also covers data validation, application architecture, async/await, and more.

## Lesson 2: Setting up Mongoose

In this lesson, you'll be setting up Mongoose. Mongoose makes it easy to model and manage your application data. This includes data sanitization, data validation, and more. Mongoose will serve as a replacement for the native driver, providing you with a more object-oriented interface.

### Setting up Mongoose

First up, install Mongoose.

```
npm i mongoose@5.3.16
```

Like the MongoDB native driver, Mongoose provides a `connect` function you can use to connect to your MongoDB database.

```
const mongoose = require('mongoose')

mongoose.connect('mongodb://127.0.0.1:27017/task-manager-api', {
    useNewUrlParser: true,
    useCreateIndex: true
})
```

### Modeling Your Data

The core feature of Mongoose is the ability to model your data. A new model can be created for the different types of data your application needs to store. You can create as many models as your application needs.

The code below defines a user model. The model definition is where you define what makes up a user. This would include all the pieces of data you want to store in the database. The user model below has just two fields, a name and an age.

```
const User = mongoose.model('User', {
    name: {
        type: String
    },
    age: {
        type: Number
    }
})
```

With the model defined, it's time to start creating and saving users. The `User` variable above stores the Mongoose model. This is a constructor function that can be used to create new users. The snippet below creates a new user with the name `'Andrew'` and the age `27`. This alone won't save any data to the database, but it's a step in the right direction.

```
const me = new User({
    name: 'Andrew',
    age: 27
})
```

The new model instance can be saved to the database using the `save` method.

```
me.save().then(() => {
    console.log(me)
}).catch((error) => {
    console.log('Error!', error)
})
```

Documentation Links

- Mongoose

## Lesson 3: Creating a Mongoose Model

In this lesson, it's on you to define a second Mongoose model for tasks.

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: Data Validation and Sanitization: Part I

In this lesson, you'll set up data validation and sanitization for your models. Validation will allow you to restrict what data can be stored in the database, while sanitization will allow you to store user data in a uniform and standardized way.

### Data Validation and Sanitization

First up, install validator. While Mongoose provides basic tools for performing validation, the validator library provides useful methods for validating data such as email addresses, phone numbers, zip codes, and more.

```
npm i validator@10.9.0
```

Mongoose comes with support for basic validation and sanitization. The user model below shows how this can be configured. `required` is used to validate that a value is provided for a given field. `trim` is used to remove extra spaces before or after data. `lowercase` is used to convert the data to lowercase before saving it to the database. You can find a complete list of options in the schema documentation.

You can also define custom validation for your models. This is done using `validate` as shown in the example below. The method gets called with the value to validate, and it should throw an error if the data is invalid. The example below uses the `isEmail` method from validator to validate the email address is valid before saving it to the database.

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

const User = mongoose.model('User', {
    name: {
        type: String,
        required: true,
        trim: true
    },
    email: {
        type: String,
        required: true,
        trim: true,
        lowercase: true,
        validate(value) {
            if (!validator.isEmail(value)) {
                throw new Error('Email is invalid')
            }
        }
    }
})
```

Documentation Links

- npm: validator

## Lesson 5: Data Validation and Sanitization: Part II

In this lesson, it's up to you to add validation and sanitization to the task model. You'll also be defining a new field on the user model with validation and sanitization of its own.

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 6: Structuring a REST API

In this lesson, you'll explore REST APIs. You'll learn how to structure your API and how it can be used as the back-end for a web or mobile application.

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

## Lesson 7: Installing Postman

In this lesson, you'll set up Postman. Postman makes it easy to test your REST API by providing you with a set of tools for making HTTP requests. This is not meant to serve as a

replacement for a web or mobile application, it's just a useful way to debug your endpoints as you're creating them.

You can grab Postman here. It's free and available for all operating systems.

### Documentation Links

- Postman

## Lesson 8: Resource Creation Endpoints: Part I

In this lesson, you'll learn how to create REST API endpoints for creating resources. This will allow users of the API to create new users and new tasks.

### Resource Creation Endpoints

Resource creation endpoints use the POST HTTP method. The URL structure is `/resources`. If you wanted to create a user, it would be `POST /users`. If you wanted to create a task, it would be `POST /tasks`.

The code below uses `app.post` to set up a POST request handler for `/users`. The handler function creates a new instance of the user model and saves it to the database.

`express.json` is also setup to parse incoming JSON into a JavaScript object which you can access on `req.body`.

```
app.use(express.json())

app.post('/users', (req, res) => {
    const user = new User(req.body)

    user.save().then(() => {
        res.send(user)
    }).catch((e) => {
        res.status(400).send(e)
    })
})
```

## Lesson 9: Resource Creation Endpoints: Part II

In this lesson, it's on you to set up a new endpoint for creating tasks.

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: Resource Reading Endpoints: Part I

In this lesson, you'll learn how to create REST API endpoints for reading resources. This will allow users of the API to fetch users and tasks from the database.

### Resource Reading Endpoints

Resource reading endpoints use the GET HTTP method. The URL structure is `/resources` for a list of resources and `/resources/:id` for fetching an individual resource by its ID. If you wanted to fetch all your tasks, it would be `GET /tasks`. If you wanted to fetch an individual task with the ID of 198, it would be `GET /tasks/198`.

The code below uses `app.get` to set up a GET request handler for `/users/:id`. `:id` serves as a placeholder for the ID of the user to fetch. If the request is `GET /users/321`, then the ID will be 321. This is known as a URL parameter, and you can access the value for URL parameters on `req.params`.

```
app.get('/users/:id', (req, res) => {
    const _id = req.params.id // Access the id provided

    User.findById(_id).then((user) => {
        if (!user) {
            return res.status(404).send()
        }

        res.send(user)
    }).catch((e) => {
        res.status(500).send()
    })
})
```

### Documentation Links

- Express route parameters

## Lesson 11: Resource Reading Endpoints: Part II

In this lesson, it's on you to create REST API endpoints for fetching tasks out of the database.

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 12: Promise Chaining

In this lesson, you'll explore promise chaining. Promise chaining is a syntax that allows you to chain together multiple asynchronous tasks in a specific order. This is great for complex code where one asynchronous task needs to be performed after the completion of a different asynchronous task.

### Promise Chaining

To demonstrate promise chaining, the following function will be used to simulate an asynchronous task. In reality, it's just adding up a couple of numbers, waiting two seconds, and fulfilling the promise with the sum.

```
const add = (a, b) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (a < 0 || b < 0) {
                return reject('Numbers must be non-negative')
            }

            resolve(a + b)
        }, 2000)
    })
}
```

With the dummy asynchronous function defined, promise chaining can be used to call `add` twice. The code below adds up `1` and `2` for a total of `3`. It then uses the sum of `3` as the input for another call to `add`. The second call to `add` adds up `3` and `4` for a total of `7`.

Promise chaining occurs when the `then` callback function returns a promise. This allows you to chain on another `then` call which will run when the second promise is fulfilled. `catch` can still be called to handle any errors that might occur along the way.

```
add(1, 2).then((sum) => {
    console.log(sum)  // Will print 3
    return add(sum, 4)
}).then((sum2) => {
    console.log(sum2) // Will print 7
}).catch((e) => {
    console.log(e)
})
```

## Lesson 13: Promise Chaining Challenge

In this lesson, it's on you to use promise chaining to complete a programming challenge.

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: Async/Await

In this lesson, you'll learn how to use async and await. These provide an improved syntax for working with promises. You'll be able to write complex asynchronous code that looks like normal synchronous code. This makes it much easier to write and maintain asynchronous code.

### Exploring Async/Await

The example below uses the `add` function that was created two lessons ago.

The first step to using async and await is to create an asynchronous function. This is done using the `async` keyword before the function definition. This can be seen in the definition of `doWork` below. Any function can be defined as an asynchronous function, not just arrow functions.

With an async function in place, you can now use the `await` operator. The `await` operator can only be used inside of asynchronous functions. This removes the need for excess callbacks and makes code much easier to read.

The `await` operator is used with promises in asynchronous functions. You can see this used three times in `doWork`. The `await` operator allows you to work with promises in a way that looks like synchronous code. If the promise is fulfilled, the fulfilled value can be accessed as the return value from the function. If the promise is rejected, it would be as though the function threw an error. `await` will pause the function execution until the promise is either fulfilled or rejected.

It's important to note that async and await are syntax enhancements for working with promises. Promises are still at the core of asynchronous code that uses async and await.

```
const doWork = async () => {
    const sum = await add(1, 99)
    const sum2 = await add(sum, 50)
    const sum3 = await add(sum2, 3)
    return sum3
}

doWork().then((result) => {
    console.log('result', result)
}).catch((e) => {
    console.log('e', e)
})
```

### Documentation Links

- mdn: async function
- mdn: await operator

## Lesson 15: Async/Await: Part II

In this lesson, it's up to you to use async and await with the Mongoose library.

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 16: Integrating Async/Await

In this lesson, you'll be integrating async and await into the Express route handlers.

### Integrating Async/Await

Below is a modified version of `GET /users`. The handler function was set up as an asynchronous function which allows you to use await in the function. `await` is used with the promise from `User.find` to get a list of all the users without needing to call `then` and `catch` with handler functions. A try/catch statement is also used to handle any errors that might occur.

```
app.get('/users', async (req, res) => {
    try {
        const users = await User.find({})
        res.send(users)
    } catch (e) {
        res.status(500).send()
    }
})
```

## Lesson 17: Resource Updating Endpoints: Part I

In this lesson, you'll learn how to create REST API endpoints for updating resources. This will allow users of the API to update users and tasks that are already in the database.

### Resource Updating Endpoints

Resource updating endpoints use the PATCH HTTP method. The URL structure is `/resources/:id` for updating an individual resource by its ID. If you want to update an individual task with the ID of 44, it would be `PATCH /tasks/44`.

`app.patch` is used to set up the Express route handler.

```
app.patch('/users/:id', async (req, res) => {
    // Route handler code here
  })
```

When working with updates, it's a good idea to alert the user if they're trying to update something that they can't update. The code below checks that the user is only updating fields that can be updated, otherwise it will send back an error response.

```
const updates = Object.keys(req.body)
const allowedUpdates = ['name', 'email', 'password', 'age']
const isValidOperation = updates.every((update) =>
allowedUpdates.includes(update))

if (!isValidOperation) {
    return res.status(400).send({ error: 'Invalid updates!' })
}
```

If all goes well, the updates will be applied to the user, then a response will be sent back.

If the provided updates are valid, `findByIdAndUpdate` can be used to update the document in the database. Try/catch is used here to send back an error if something goes wrong when updating the user. This would include the new data not passing the validation defined for the model.

```
try {
    const user = await User.findByIdAndUpdate(req.params.id, req.body, { new:
true, runValidators: true })

    if (!user) {
        return res.status(404).send()
    }

    res.send(user)
} catch (e) {
    res.status(400).send(e)
}
```

## Lesson 18: Resource Updating Endpoints: Part II

In this lesson, it's on you to create REST API endpoints for deleting tasks out of the database.

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 19: Resource Deleting Endpoints

In this lesson, you'll learn how to set up a REST API endpoint for deleting resources. This video covers deleting users as well as a challenge for deleting tasks.

### Deleting Resources

Resource deleting endpoints use the DELETE HTTP method. The URL structure is `/resources/:id` for deleting an individual resource by its ID. If you want to delete an individual task with the ID of 897, it would be `DELETE /tasks/897`.

`app.delete` is used to set up the Express route handler.

```
app.delete('/users/:id', async (req, res) => {
    // Route handler
})
```

The handler itself can delete the resource using `findByIdAndDelete`.

```
try {
    const user = await User.findByIdAndDelete(req.params.id)

    if (!user) {
        return res.status(404).send()
    }

    res.send(user)
} catch (e) {
    res.status(500).send()
}
```

## Lesson 20: Separate Route Files

In this lesson, you'll learn how to organize your Express application endpoints. Defining all endpoints in a single file is a fine way to get started, but that won't scale well as you add more routes to the app.

### Creating Separate Routers

Express allows you to create as many routers as you want. These separate routers can then be combined into a single Express application. You can create a new router using `express.Router` as shown below. The example file below creates the router, adds routes, and exports the router from the file.

```
const router = new express.Router()

router.post('/someEndpoint', (req, res) => {
    // Do something
})

module.exports = router
```

The router defined in the file above can be added into the Express application in `index.js`. This is done by loading the router in with `require` and then passing the router

to `app.use`. You can set up as many routers as you need for your application, though it's common to have a router for each distinct resource your REST API has.

```
// Register with existing application
app.use(router)
```

Documentation Links

- express.Router

# Section 12: API Authentication and Security

## Lesson 1: Section Intro

In this section, you'll set up authentication for the task manager app. This will require users to log in before they'll be able to manage their tasks. This section also covers password security, Express middleware, and more.

## Lesson 2: Securely Storing Passwords: Part I

In this lesson, you'll learn how to securely store user passwords by hashing and salting the password before storing it in the database.

### Hashing Passwords

Storing plain text passwords is a bad idea. Most folks reuse password for multiple accounts online. That means if your database gets compromised, the hacker can reuse those credentials on other sites such as credit cards or bank accounts. We don't want to leave our users open to further attacks.

The solution is to hash passwords using a secure one-way hashing algorithm. Users passwords will stay hidden and secure, even if the database is compromised.

## Hashing Passwords with Bcrypt

First up, install the library.

```
npm i bcryptjs@2.4.3
```

The `hash` method can be used to hash the plain text password. The example below hashes the password “Red12345!”.

```
const password = 'Red12345!'
const hashedPassword = await bcrypt.hash(password, 8)
// The hashed password is what would be stored in the database
```

The `compare` method is used to compare a plain text password against a previously hashed password. This would be useful when logging in. The user logging in provides the plain text password for their account. The application fetches the hashed password from the database for that user. `compare` is then called to confirm it’s a match.

```
const isMatch = await bcrypt.compare('red12345!', hashedPassword)
console.log(isMatch)
```

### Documentation Links

- npm: bcryptjs

## Lesson 3: Securely Storing Passwords: Part II

In this lesson, you’ll learn how to use Mongoose middleware. Middleware will allow you to automatically hash a user’s password before the user is saved to the database.

### Mongoose Middleware

Middleware allows you to register some code to run before or after a lifecycle event for your model. As an example, you could use middleware to register some code to run just after a user is deleted. You could also use middleware to register some code to run just before the user is saved. This can be used to hash passwords just before saving users to the database.

The example below calls `pre` with the `'save'` lifecycle event. This registers a function to run just before users are saved. The function itself checks if the password has been altered. If the password has been altered, the plain text password is overwritten with a hashed version.

```
userSchema.pre('save', async function (next) {
    const user = this

    if (user.isModified('password')) {
        user.password = await bcrypt.hash(user.password, 8)
    }

    next()
})
```

### Documentation Links

- Mongoose middleware

## Lesson 4: Logging in Users

In this lesson, you'll set up the ability for users to log into their existing accounts.

### Logging in Users

Logging in a user is a two-step process. The user provides their email and password, and the first thing to do is fetch the user by their email. From there, bcrypt is used to verify the password provided matches the hashed password stored in the database. If either step fails, the users won't be able to log in. If both steps succeed, then you know the user is who they say they are.

The code below sets up `findByCredentials` which finds a user by their email and password.

```
userSchema.statics.findByCredentials = async (email, password) => {
    const user = await User.findOne({ email })

    if (!user) {
        throw new Error('Unable to login')
    }

    const isMatch = await bcrypt.compare(password, user.password)

    if (!isMatch) {
        throw new Error('Unable to login')
    }

    return user
}
```

You can then call `findByCredentials` from you application when users need to login. The example below shows how this can be done.

```
const user = await User.findByCredentials(req.body.email, req.body.password)
```

## Lesson 5: JSON Web Tokens

In this lesson, you'll explore JWTs (JSON Web Tokens). JWTs provide a nice system for issuing and validating authentication tokens. The authentication token will ensure that the client doesn't need to log in every time they want to perform an operation on the server.

### JSON Web Tokens

First up, install the library.

```
npm i jsonwebtoken@8.4.0
```

The `sign` method can be used to generate a new token. `sign` accepts three arguments:

The first is the data to embed in the token: This needs to include a unique identifier for the user.

The second is a secret phrase: This is used to issue and validate tokens, ensuring that the token data hasn't been tampered with.

The third is a set of options: The example below uses `expiresIn` to create a token that's valid for seven days.

```
const jwt = require('jsonwebtoken')

const token = jwt.sign({ _id: 'abc123' }, 'thisismynewcourse', { expiresIn:
'7 days' })
```

Tokens can be issued to users when they sign up or log in to the application. These can then be stored on the data and used to authenticate the user when they perform other options.

The server can verify the token using `verify`. This requires two arguments:

The first is the token to validate.

The second is the secret phrase that the token was created with. If valid, the embedded data will be returned. This would allow the server to figure out which user is performing the operation.

```
const data = jwt.verify(token, 'thisismynewcourse')
// data._id contains the user id of the user that owns this token
```

## Lesson 6: Generating Authentication Tokens

In this lesson, you'll integrate JWTs into the application. This will allow the app to issue an authentication token when a user signs up or logs in.

### Generating and Storing Auth Tokens

Authentication tokens for a user can be stored in the database. This provides a way for users to log out. All generated authentication tokens will be stored as part of the user profile. If a user logs out, that token will be removed from the user profile. A token would only be considered valid if it's a valid JWT and it's still stored as part of the user profile. A user could be logged out of all session by simply deleting all the tokens stored in their user profile.

The snippet below adds a `tokens` array onto the user model. This will be used to store all valid authentication tokens for a user.

```
// Other properties and options omitted for brevity
const userSchema = new mongoose.Schema({
    tokens: [{
        token: {
            type: String,
            required: true
        }
    }]
})
```

The instance method below is responsible for generating new authentication tokens. The token is created, stored in the database, and finally returned from the function.

```
userSchema.methods.generateAuthToken = async function () {
    const user = this
    const token = jwt.sign({ _id: user._id.toString() }, 'thisismynewcourse')

    user.tokens = user.tokens.concat({ token })
    await user.save()

    return token
}
```

`generateAuthToken` can then be called to generate a fresh authentication token when users sign up or log in.

```
const token = await user.generateAuthToken()
```

## Lesson 7: Express Middleware

In this lesson, you'll explore Express middleware. When working with middleware, you'll have more control over how your server processes requests. This will be used to check that a user is authenticated before performing specific operations.

### Exploring Express Middleware

Express middleware is nothing more than a function that runs as Express handles a given request. You can customize the function to do whatever you want it to do, and you can have it run whenever you want it to.

The example below uses middleware to print information about incoming requests. Middleware functions should accept three parameters: `req`, `res`, and `next`. The only new parameter is `next`. `next` is called to signal to Express that the middleware function is done.

```
const loggerMiddleware = (req, res, next) => {
    console.log('New request to: ' + req.method + ' ' + req.path)
    next()
}

// Register the function as middleware for the application
app.use(loggerMiddleware)
```

### Documentation Links

- Express middleware

## Lesson 8: Accepting Authentication Tokens

In this lesson, you'll use Express middleware to put specific routes behind authentication. That will require the client to be authenticated before the operation can be performed.

### Accepting and Validating Tokens

The goal of the authentication middleware is to validate the authentication token and then fetch the profile for that user. `auth` below shows how you can get this done. Notice that the user profile is added onto `req.user`. This allows route handler functions to access the user profile without needing to fetch it again.

```
const jwt = require('jsonwebtoken')
const User = require('../models/user')

const auth = async (req, res, next) => {
    try {
        const token = req.header('Authorization').replace('Bearer ', '')
        const decoded = jwt.verify(token, 'thisismynewcourse')
        const user = await User.findOne({ _id: decoded._id, 'tokens.token':
token })

        if (!user) {
            throw new Error()
        }

        req.user = user
        next()
    } catch (e) {
        res.status(401).send({ error: 'Please authenticate.' })
    }
}

module.exports = auth
```

The authentication middleware can be added to individual endpoints to lock them down. This is shown with `GET /users/me` below. `auth` is added as the second argument to `router.get`, meaning that it will run before the route handler function runs. This will ensure the user is authenticated.

```
router.get('/users/me', auth, async (req, res) => {
    res.send(req.user)
})
```

## Lesson 9: Advanced Postman

In this lesson, you'll explore Postman environments. Environments make it easy to manage your requests and authentication without having to manually add authentication tokens to the individual requests.

There are no notes for this video. Refer to the video to learn how to set up Postman environments.

## Lesson 10: Logging Out

In this lesson, it's your job to give users a way to log out of the 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 11: Hiding Private Data

In this lesson, you'll learn how to limit what data gets sent to the client. This will allow you to hide authentication tokens and hashed passwords from server responses.

### Hiding Private Data

When a Mongoose document is passed to `res.send`, Mongoose converts the object into JSON. You can customize this by adding `toJSON` as a method on the object. The method below removes the `password` and `tokens` properties before sending the response back.

```
userSchema.methods.toJSON = function () {
    const user = this
    const userObject = user.toObject()

    delete userObject.password
    delete userObject.tokens

    return userObject
}
```

## Lesson 12: Authenticating User Endpoints

In this lesson, you'll be adding authentication to the other user endpoints.

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 13: The User/Task Relationship

In this lesson, you'll learn how to create a relationship between a user and tasks. This will make it possible to know which tasks a user created.

### Mongoose Relationships

To set up the relationship, both the user and task model will be changed. First up, a new field needs to be added onto the task. This will store the ID of the user who created it.

```
// Other properties and options omitted for brevity
const Task = mongoose.model('Task', {
    owner: {
        type: mongoose.Schema.Types.ObjectId,
        required: true,
        ref: 'User'
    }
})
```

Next, a virtual property needs to be added onto the user. The code below adds a `tasks` field onto users that can be used to fetch the tasks for a given user. It's a virtual property because users in the database won't have a `tasks` field. It's a reference to the task data stored in the separate collection.

```
userSchema.virtual('tasks', {
    ref: 'Task',
    localField: '_id',
    foreignField: 'owner'
})
```

With the relationship configured, tasks can be created with an `owner` value.

The code below shows how you can fetch the owner of a given task.

```
const task = await Task.findById('5c2e505a3253e18a43e612e6')
await task.populate('owner').execPopulate()
console.log(task.owner)
```

The code below shows how you can fetch the tasks for a given user.

```
const user = await User.findById('5c2e4dcb5eac678a23725b5b')
await user.populate('tasks').execPopulate()
console.log(user.tasks)
```

## Lesson 14: Authenticating Task Endpoints

In this lesson, you'll be setting up authentication for the other task endpoints.

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 15: Cascade Delete Tasks

In this lesson, you'll learn how to use Mongoose middleware to clean up a user's tasks when they close their account. This will make sure that all their data is securely removed from the database.

### Deleting a User's Tasks

The middleware function below is registered using `pre`. It will run just before `remove` fires for the user. The function itself deletes all tasks created by that user. Now when a user closes their account, their other data will get deleted too.

```
const Task = require('./task')

// Existing code omitted for brevity

userSchema.pre('remove', async function (next) {
     const user = this
     await Task.deleteMany({ owner: user._id })
     next()
 })
```

# Section 13: Sorting, Pagination, and Filtering

## Lesson 1: Section Intro

In this section, you'll explore advanced techniques for fetching data. This includes sorting, filtering, and pagination. All three of these will give clients more control over what data they get back. This keeps applications fast, as they don't need to fetch unnecessary data.

## Lesson 2: Working with Timestamps

In this lesson, you'll enable timestamps for your Mongoose models. Mongoose will automatically track when documents were created and updated. This is great data to have if you want to allow users to sort by when the document was created or updated.

### Enabling Mongoose Timestamps

Schema options can be used to enable timestamps. Schema options are provided by passing an object in as the second argument to `mongoose.Schema`. Set `timestamps` to `true`

to have Mongoose add `createdAt` and `updatedAt` fields to the model. You don't need to write any code to create or manage those fields, as Mongoose does all that for you.

```
const taskSchema = new mongoose.Schema({
    //Task fields omitted for brevity
}, {
    timestamps: true
})

const Task = mongoose.model('Task', taskSchema)
```

## Lesson 3: Filtering Data

In this lesson, you'll use query parameters to allow for data filtering. This will allow clients to fetch all tasks, just the complete tasks, or just the incomplete tasks.

### Filtering Data

`GET /tasks` below supports a `completed` query parameter which can be set to `true` or `false`. This will prevent clients from fetching unnecessary data that they don't plan on using.

First up, create an object to store the search criteria.

```
const match = {}
```

From there, check if the query parameter was provided. The provided value should be parsed into a boolean and stored on `match.completed`.

```
if (req.query.completed) {
    match.completed = req.query.completed === 'true'
}
```

Last up, `match` can be added onto `populate` to fetch just the users that match the search criteria.

```
await req.user.populate({
    path: 'tasks',
    match
}).execPopulate()
```

## Lesson 4: Paginating Data

In this lesson, you'll add pagination to the application. This will allow the client to fetch data in pages. The client can start off with the first page of data and then fetch other pages as they're needed.

### Data Pagination

Pagination is configured using `limit` and `skip`. These two values give the client complete control of the data they're getting back.

If a client wanted the first page of 10 tasks, `limit` would be set to `10` and `skip` would be set to `0`. If the client wanted the third page of 10 tasks, `limit` would be set to `10` and `skip` would be set to `20`.

Both `limit` and `skip` can be added onto the `options` object passed to `populate`. The code below uses `parseInt` to convert the string query parameters into numbers first.

```
await req.user.populate({
    path: 'tasks',
    match,
    options: {
        limit: parseInt(req.query.limit),
        skip: parseInt(req.query.skip)
    }
}).execPopulate()
```

## Lesson 5: Sorting Data

In this lesson, you'll add sorting to the application. Clients will be able to fetch the data back in any order they like.

### Sorting Data

The `options` object used for pagination can also be used for sorting. A `sort` property should be set, which is an object containing key/value pairs. The key is the field to sort. The value is `1` for ascending and `-1` for descending sorting.

`GET /tasks` will get support for a `sortBy` query parameter. The value should include the field to sort and the order in which to sort. `createdAt:asc` would sort the tasks in ascending order with the oldest first. `createdAt:asc` would sort the tasks in a descending order with the newest first.

Start with an empty object to store the sorting options.

```
const sort = {}
```

If the query parameter is provided, it'll get parsed and `sort` will be updated.

```
if (req.query.sortBy) {
    const parts = req.query.sortBy.split(':')
    sort[parts[0]] = parts[1] === 'desc' ? -1 : 1
}
```

`sort` is then added onto `options`. If `sortBy` isn't provided, `sort` will be an empty object and no sorting will occur.

```
await req.user.populate({
    path: 'tasks',
    match,
    options: {
        limit: parseInt(req.query.limit),
        skip: parseInt(req.query.skip),
        sort
    }
}).execPopulate()
```

# Section 14: File Uploads

## Lesson 1: Section Intro

In this section, you'll learn how to configure Node.js to support file uploads. This will allow users to upload documents, profile pictures, and any other file type you might need to support. You'll also see what it takes to store the uploaded files in MongoDB.

## Lesson 2: Adding Support for File Uploads

In this lesson, you'll set up multer. Multer is a library in the Express ecosystem that allows your Express application to easily support file uploads. It couldn't be easier.

### Configuing Multer

First up, install the library.

```
npm i multer@1.4.1
```

Multer can then be configured to fit your specific needs. The example below shows off a basic configuration where `dest` is set to `avatars`. This will store all uploaded files in a directory called `avatars`.

```
const multer = require('multer')

const upload = multer({
    dest: 'avatars'
})
```

Multer is then added as middleware for the specific endpoint that should allow for file uploads. The route below is expecting a single `avatar` field on the submitted form.

```
router.post('/users/me/avatar', upload.single('avatar'), (req, res) => {
    res.send()
})
```

### Documentation Links

- npm: multer

## Lesson 3: Validating File Uploads

In this lesson, you'll learn how to validate file uploads. This will allow you to reject files that are too large or files of the wrong type.

### Validating Multer Uploads

The multer configuration below adds these two types of validation.

`limits.fileSize` is set to limit the file size in bytes. The configuration below uses 1,000,000 bytes which is equivalent to 1 megabyte.

`fileFilter` is set to validate the file type. The method below will reject all documents that don't have either `.doc` or `.docx` file extensions. This same technique could be used to limit uploads to just images, PDFs, or any other file type.

```
    const upload = multer({
    dest: 'images',
    limits: {
        fileSize: 1000000
    },
    fileFilter(req, file, cb) {
        if (!file.originalname.match(/\.(doc|docx)$/)) {
            return cb(new Error('Please upload a Word document'))
        }

        cb(undefined, true)
    }
})
```

## Lesson 4: Validation Challenge

In this lesson, you'll add validation to avatar uploads.

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 5: Handling Express Errors

In this lesson, you'll customize the errors that multer provides. This will give you complete control of what sort of response the client gets when their upload is rejected.

### Handling Express Errors

You can handle errors from middleware function by providing a function to Express. As shown below, a new function is passed as the final argument to `router.post`. This function accepts `error`, `req`, `res`, and `next`. This call signature lets Express know the function is designed to handle errors.

The function itself sends back a JSON response with the error message from multer.

```
router.post('/users/me/avatar', upload.single('avatar'), (req, res) => {
    res.send()
}, (error, req, res, next) => {
    res.status(400).send({ error: error.message })
})
```

## Lesson 6: Adding Images to the User Profile

In this lesson, you'll learn how to associate the uploaded avatar with the users account.

### Adding Images to the User Profile

A new field needs to be added to the user model to store the avatar image data. The snippet below adds `avatar` on the user with the type of `Buffer`. The `Buffer` type should be used when storing binary data, which is exactly the type of data that multer provides.

```
// Existing code omitted for brevity
const userSchema = new mongoose.Schema({
    avatar: {
        type: Buffer
    }
})
```

The avatar upload route will be able to change the user profile data, so the route should be put behind authentication. The handler function grabs the binary data and stores it on the `avatar` field. Finally, the changes are saved.

```
router.post('/users/me/avatar', auth, upload.single('avatar'), async (req,
res) => {
    req.user.avatar = req.file.buffer
    await req.user.save()
    res.send()
}, (error, req, res, next) => {
    res.status(400).send({ error: error.message })
})
```

## Lesson 7: Serving up Files

In this lesson, you'll learn how to serve up user profile images. These images will be served up as if they were static assets for the application.

### Serving up Files

Serving up the user avatars will require two pieces of data from the server. The first is the image data, and the second is the `Content-Type` header. The image data is stored on the user profile. The header should be set equal to `image/png` which lets the client know they're getting a PNG image back.

The route below fetches the image data and sets the `Content-Type` header for the response. The URL could be visited to view the profile picture.

```
router.get('/users/:id/avatar', async (req, res) => {
    try {
        const user = await User.findById(req.params.id)

        if (!user || !user.avatar) {
            throw new Error()
        }

        res.set('Content-Type', 'image/jpg')
        res.send(user.avatar)
    } catch (e) {
        res.status(404).send()
    }
})
```

## Lesson 8: Auto-Cropping and Image Formatting

In this lesson, you'll learn how to resize and format images. This will let you create uniform sizes and file types for user avatars.

### Auto-Cropping and Image Formatting

First up, install the npm library.

```
npm i sharp@0.21.1
```

Now, sharp can be used to manipulate uploaded images. Before the image data is added onto the user profile, the data should be passed through sharp. The example below uses `resize` to resize all uploads to 250 by 250 pixels. The example also uses `png` to convert all images to portable network graphics. Lastly, `toBuffer` is used to retrieve the modified image data. The modified data is what should be saved in the database.

```
const sharp = require('sharp')

const buffer = await sharp(req.file.buffer).resize({ width: 250, height: 250
}).png().toBuffer()
```

Documentation Links

- npm: sharp

# Section 15: Sending Emails

## Lesson 1: Section Intro

In this section, you'll add email sending to your Node.js application! This will allow you to communicate with users as they use the app. This could be useful for welcome emails, notifications, and more!

## Lesson 2: Exploring SendGrid

In this lesson, you'll integrate SendGrid into your Node app. SendGrid is one of many services that allow you to send emails from your application code.

Exploring SendGrid

First up, install the module.

```
npm i sendgrid/mail@6.3.1
```

Next, create a free SendGrid account and get your API key. Check out the lesson video to learn how to get your API key. The code below shows what's necessary to get the SendGrid module configured. All you need to do is call `setApiKey` to... well... set your API key.

```
const sgMail = require('@sendgrid/mail')

sgMail.setApiKey('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')
```

`send` can be called to send an email from your application. The configuration object can be used to provide:

- `to` - Who is the email to?
- `from` - Who is the email from?
- `subject` - What's the subject line of the email?
- `text` - What's the body of the email.

```
sgMail.send({
    to: 'andrew@mead.io',
    from: 'andrew@mead.io',
    subject: 'This is my first creation!',
    text: 'I hope this one actually get to you.'
})
```

In the long term, you'll want to purchase a custom domain and register it with SendGrid. This will increase your sending reliability.

#### Documentation Links

- SendGrid

## Lesson 3: Sending Welcome and Cancelation Emails

In this lesson, you'll be sending email to users when they sign up or cancel their account.

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: Environment Variables

In this lesson, you'll learn how to use environment variables to securely store API keys and other credentials. This will reduce the chance your private keys fall into the wrong hands.

#### Environment Variables

First up, install the npm module.

```
npm i env-cmd@8.0.2
```

Next up, create an environment file `dev.env` in the `config` directory. This will store your environment variables in the following format.

```
KEY=value
ANOTHER_KEY=some other value
```

Next, update the `dev` script to use env-cmd to load in those environment variables when it starts up. That would be `env-cmd ./config/dev.env nodemon src/index.js`.

Now, you can remove API keys and database credentials from the application itself. For example, you can create `MONGODB_URL` in the development environment file. The application code shown below can then reference that environment variable to get its value. This can be done with the SendGrid API key and the JWT secret used to generate and verify authentication tokens.

```
mongoose.connect(process.env.MONGODB_URL, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false
})
```

Documentation Links

- npm: env-cmd

## Lesson 5: Creating a Production MongoDB Database

In this lesson, you'll set up a production MongoDB database using MongoDB Atlas. Atlas is the official MongoDB hosting platform released by the MongoDB organization.

Please refer to the video for the detailed steps required to set up the production database.

## Lesson 6: Heroku Deployment

In this lesson, you'll deploy the task manager API to Heroku.

#### Heroku Deployment

You already know how to deploy Node.js applications to Heroku. The only difference with this application is that your custom environment variables need to be configured on Heroku too. This doesn't include `PORT`, as that's managed by Heroku.

The command below is used to set an environment variable.

```
heroku config:set KEY=VALUE
```

A variation of that command can be used to fetch all the environment variables currently configured.

```
heroku config
```

Lastly, you can delete an environment variable as shown here.

```
heroku config:unset KEY
```

# Section 16: Testing Node.js

## Lesson 1: Section Intro

In this section, you'll learn how to test your Node.js applications. Setting up an automated test suite makes it easy to check that your application is always working as expected.

## Lesson 2: Jest Testing Framework

In this lesson, you'll set up the Jest testing framework. Jest gives you everything you need to create a test suite for your Node.js applications.

#### Setting up Jest

First up, install the module.

```
npm i jest@23.6.0
```

Next, create a `test` script in `package.json`. The script itself is `jest`. This allows you to use `npm test` to run the Jest test suite.

Now, you'll need to create a test suite. This is a file in your project that ends with `.test.js`. The file extension allows Jest to find and run the test suites for your project.

### Creating a Test Case

You can add a test case to a test suite using the `test` function. Jest provides various functions as global variables in your test suite files. `test` is one of them. The first argument to `test` is the name of your test case. The second argument to `test` is the test function itself.

If the test function throws an error, the test cast will fail. If the test function doesn't throw an error, the test case will pass.

The test case below would always pass, as no error is thrown.

```
test('Hello world!', () => {

})
```

The test case below would always fail, as it throws an error.

```
test('This should fail', () => {
    throw new Error('Failure!')
 })
```

### Documentation Links

- Jest

## Lesson 3: Writing Tests and Assertions

In this lesson, you'll add assertions to your test cases. Assertions allow you check if a given value is what you're expecting or not.

### Testing a Function

For this example, let's test the `calculateTip` function shown below. All it does is calculate the tip for your restaurant bill.

```
const calculateTip = (total, tipPercent = .25) => total + (total *
tipPercent)

module.exports = {
    calculateTip
 }
```

The test suite below has a single test case for the `calculateTip` function. The test case itself calculates a 30% tip on a $10 restaurant bill. The assertion checks that the calculated total equals $13. The assertion is made using `toBe` to check for equality.

```
const { calculateTip } = require('../src/math')

test('Should calculate total with tip', () => {
    const total = calculateTip(10, .3)
    expect(total).toBe(13)
})
```

### Documentation Links

- expect

## Lesson 4: Writing Your Own Tests

In this video, it's on you to write some new test cases using what you've learned so far.

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 5: Testing Asynchronous Code

In this lesson, you'll learn how to test asynchronous code. This will be necessary to test the Express API endpoints.

### Testing Asynchronous Code

The two test cases below test the asynchronous `add` function you created earlier in the course. Both test cases add up 2 and 3 and assert that the total is 5.

The callback function for the first test case accepts a `done` parameter. This lets Jest know that the test function contains asynchronous code. Jest won't determine if the test passed or failed until `done` is called. In the example below, `then` is called to run some code after the numbers are added. This is where the assertion is added and it's where `done` is called.

```
test('Should add two numbers', (done) => {
    add(2, 3).then((sum) => {
        expect(sum).toBe(5)
        done()
    })
})
```

Your test cases can use async/await as well. The test case below is a refactored version of the test case above. The test case function is defined with `async`. `await` is used in the function to ensure that Jest waits for those asynchronous tasks to complete. Both test cases are functionally identical.

```
test('Should add two numbers async/await', async () => {
    const sum = await add(2, 3)
    expect(sum).toBe(3)
})
```

## Lesson 6: Testing an Express Application: Part I

In this lesson, you'll set up the Express API to be easily testable. This involves settings up a test environment as well as configuring Jest to work with Node.

### Creating a Test Environment

Creating the test environment requires `test.env` to be added to the `config` directory. The contents will be identical to `dev.env`, with the exception of the MongoDB connection string. The test environment should use a separate database such as `task-manager-api-test`. This will prevent the test cases from messing with development data.

With the environment in place, update the `test` script to load the environment file in. That would be `env-cmd ./config/test.env jest --watch --runInBand`.

### Configuring Jest

By default, Jest is expecting to run in the browser. You can use Jest with Node, but you'll need to configure Jest to enable support. Jest can be configured by adding a `jest`

property in `package.json`. The configuration below sets `testEnvironment` to `node` to ensure that Jest runs correctly in Node.js.

```
{
    "jest": {
        "testEnvironment": "node"
    }
}
```

### Documentation Links

- Configuring Jest

## Lesson 7: Testing an Express Application: Part II

In this lesson, you'll add tests for the Express API. Each test case will focus on testing a specific endpoint, making assertions about the response from the server.

### Testing with Supertest

Supertest was created by the Express team to allow you to easily test your Express apps. First up, install the module.

```
npm i supertest@3.4.1
```

Now, supertest can be used to test an endpoint. The test case below tests that new users can sign up for accounts. All the account data provided is valid, so a new account should be created.

Step one is to pass the express `app` to `request`. Next, supertest methods can be chained together to fit the needs of your tests. `post` is used to make an HTTP POST request to `/users`. `send` is used to send the correct JSON data to the server. `expect` is used to assert that the response status code is correct. In this case, a successful signup should result in a `201` status code.

```
const request = require('supertest')
const app = require('../src/app')

test('Should signup a new user', async () => {
    await request(app).post('/users').send({
        name: 'Andrew',
        email: 'andrew@example.com',
        password: 'MyPass777!'
    }).expect(201)
})
```

### Documentation Links

- npm: supertest

## Lesson 8: Jest Setup and Teardown

In this lesson, you'll configure Jest to add test data into the database. This will allow you to test operations that require existing data, such as the login operation. You can't log in if there isn't a user account in the database to login to.

### Seeding Database

Jest provides lifecycle functions that you can use to configure your test suite. There are four:

1. `beforeEach` - Run some code before each test case
2. `afterEach` - Run some code after each test case
3. `before` - Run some code once before the tests run
4. `after` - Run some code once after the tests run

`beforeEach` works great for adding test data to the database. The `beforeEach` call below removes all users and then adds a single test user into the database. By having this run before each test case, it ensures that the tests run in a consistent environment each time they execute.

```
const User = require('../src/models/user')

const userOne = {
    name: 'Mike',
    email: 'mike@example.com',
    password: '56what!!'
}

beforeEach(async () => {
    await User.deleteMany()
    await new User(userOne).save()
})
```

With the test user in place, the test case below is able to test the login operation by logging in as the test user.

```
test('Should login existing user', async () => {
     await request(app).post('/users/login').send({
         email: userOne.email,
         password: userOne.password
     }).expect(200)
 })
```

#### Documentation Links

- Jest setup and teardown

## Lesson 9: Testing with Authentication

In this lesson, you'll learn how to test API endpoints that sit behind authentication.

#### Testing with Authentication

The goal is to test operations that require authentication. This means an authentication token will need to be passed along with the request. Step one is to create an authentication token for the test user.

```
const userOneId = new mongoose.Types.ObjectId()
const userOne = {
    _id: userOneId,
    name: 'Mike',
    email: 'mike@example.com',
    password: '56what!!',
    tokens: [{
        token: jwt.sign({ _id: userOneId }, process.env.JWT_SECRET)
    }]
}
```

From there, the authentication token can be added as part of the supertest request. Supertest provides a `set` method for setting request headers. The test case below attempts to fetch the user profile for the logged-in user.

```
    test('Should get profile for user', async () => {
    await request(app)
        .get('/users/me')
        .set('Authorization', `Bearer ${userOne.tokens[0].token}`)
        .send()
        .expect(200)
})
```

## Lesson 10: Advanced Assertions

In this lesson, it's up to you to write some test cases on your own.

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: Mocking Libraries

In this lesson, you'll learn how to mock npm modules. Jest lets you mock npm modules so you can override module functionality in your test environment.

### Mocking SendGrid

You can mock an npm module by creating a `__mocks__` directory in the `tests` folder. A module can be mocked by creating a file in the `__mocks__` folder. The file name should match up with the module name, so `tests/__mocks__/express.js` can be used to mock the Express library. If the npm module uses a scope like `@sendgrid/mail`, then a `@sendgrid` folder would be created with a `mail.js` file inside.

The job of the mock file is to provide mocked versions of the library features. The SendGrid mock below defines and exports `setApiKey` and `send`. This ensures that our code still works even though emails will no longer be sent for the tests.

```
module.exports = {
     setApiKey() {

      },
     send() {

      }
 }
```

## Lesson 12: Wrapping up User Tests

In this lesson, it's your job to finalize the user tests.

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 13: Setup Task Test Suite

In this lesson, you'll be refactoring the user test suite. These changes will makes it possible to test the task operations too.

Refer to the video for refactoring instructions. The refactoring requires lots of code to be shifted between files, which is a little more than could be covered in this guide.

## Lesson 14: Testing with Task Data

In this lesson, you'll be writing some test cases for tasks.

### Testing with Task Data

Testing tasks will require that some test tasks exist in the database. Like with users, tasks can be added to the database using `beforeEach`. The test case below fetches all tasks for the user. It also asserts that the status code is a `200` and the tasks are sent back correctly.

```
test('Should fetch user tasks', async () => {
    const response = await request(app)
        .get('/tasks')
        .set('Authorization', `Bearer ${userOne.tokens[0].token}`)
        .send()
        .expect(200)
    expect(response.body.length).toEqual(2)
})
```

### Lesson 15: Bonus: Extra Test Ideas

In this lesson, we'll go over some other test cases you could add to the application.

You can find the list of optional test cases at https://links.mead.io/extratests.

## Section 17: Real-Time Web Applications with Socket.io

### Lesson 1: Section Intro

In this section, you'll learn how to create real-time web applications with Node. The non-blocking nature of Node makes it well-suited for real-time applications such as chat apps, social media apps, and more.

### Lesson 2: Creating the Chat App Project

In this lesson, it's on you to set up the chat application web server.

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 3: WebSockets

In this lesson, you'll learn about the WebSocket protocol. The WebSocket protocol supports real-time bi-direction communication, which makes it a great fit for the chat application.

This lesson contains a detailed presentation. Please refer to the video for a recap of WebSockets.

## Lesson 4: Getting Started with Socket.io

In this lesson, you'll install and configure Socket.io. Socket.io comes with everything needed to set up a WebSocket server using Node.

### Setting up Socket.io

First up, install the module.

```
npm i socket.io@2.20
```

Socket.io can be used on its own or with Express. Since the chat application will be serving up client-side assets, both Express and Socket.io will get set up. The server file below shows how to get this done.

```
const path = require('path')
const http = require('http')
const express = require('express')
const socketio = require('socket.io')

// Create the Express application
const app = express()

// Create the HTTP server using the Express app
const server = http.createServer(app)

// Connect socket.io to the HTTP server
const io = socketio(server)

const port = process.env.PORT || 3000
const publicDirectoryPath = path.join(__dirname, '../public')

app.use(express.static(publicDirectoryPath))

// Listen for new connections to Socket.io
io.on('connection', () => {
    console.log('New WebSocket connection')
})

server.listen(port, () => {
    console.log(`Server is up on port ${port}!`)
```

The server above uses `io.on` which is provided by Socket.io. `on` allows the server to listen for an event and respond to it. In the example above, the server listens for `connection` which allows it to run some code when a client connects to the WebSocket server.

### Socket.io on the Client

Socket.io is also used on the client to connect to the server. Socket.io automatically serves up `/socket.io/socket.io.js` which contains the client-side code. The script tags below load in the client-side library followed by a custom JavaScript file.

```
<script src="/socket.io/socket.io.js"></script>
<script src="/js/chat.js"></script>
```

Your client-side JavaScript can then connect to the Socket.io server by calling `io`. `io` is provided by the client-side Socket.io library. Calling this function will set up the connection, and it'll cause the server's `connection` event handler to run.

```
io()
```

### Documentation Links

- Socket.io

## Lesson 5: Socket.io Events

In this lesson, you'll learn how to work with events in Socket.io. Events allow you to transfer data from the client to the server or from the server to the client.

### Working with Events

There are two sides to every event, the sender and the receiver. If the server is the sender, then the client is the receiver. If the client is the sender, then the server is the receiver.

Events can be sent from the sender using `emit`. Events can be received by the receiver using `on`. The example below shows how this pattern can be used to create a simple counter application. The following snippet contains the client-side JavaScript code.

```
const socket = io()

// Listen for "countUpdated"
socket.on('countUpdated', (count) => {
    console.log('The count has been updated!', count)
})

document.querySelector('#increment').addEventListener('click', () => {
    // Emit "increment
    socket.emit('increment')
})
```

The client-side code uses `on` to listen for the `countUpdated` event. A message will be logged with the current count when that event is received. The client-side code also uses `emit` to send the `increment` event. This occurs when a button on the screen is clicked.

The server-side code for this example is below.

```
let count = 0

io.on('connection', (socket) => {
    console.log('New WebSocket connection')

    socket.emit('countUpdated', count)

    socket.on('increment', () => {
        count++
        io.emit('countUpdated', count)
    })
})
```

The server above is responsible for emitting `countUpdated` and listening for `increment`. New users get the current count right after they connect to the server. If a client sends `increment` to the server, the count is incremented and all connected clients are notified of the change.

On the client, `socket.emit` emits an event to the server. On the server, both `socket.emit` and `io.emit` can be used. `socket.emit` sends an event to that specific client, while `io.emit` sends an event to all connected clients.

## Lesson 6: Socket.io Events Challenge

In this lesson, it's on you to use events to build the chat applications messaging system.

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: Broadcasting Events

In this lesson, you'll learn how to broadcast events. Broadcasted events are sent to all connected clients, expect for the client that initiated the broadcast.

### Broadcasting Events

Events can be broadcasted from the server using `socket.broadcast.emit`. This event will get sent to all sockets except the one that broadcasted the event. The code below shows this off. When a new user joins the chat application, `socket.broadcast.emit` is used to send a message to all other users that someone new has joined.

```
io.on('connection', (socket) => {
    socket.broadcast.emit('message', 'A new user has joined!')
})
```

## Lesson 8: Sharing Your Location

In this lesson, you'll integrate the Geolocation API into the chat app. This will allow users to share their locations in real time.

### Sharing Your Location

The Geolocation API lets you fetch a user's location using client-side JavaScript. Once the user gives you permission to access their location, this location can be shared with everyone else in the chat room. The code below attempts to share a user's location with the chat room.

```
document.querySelector('#send-location').addEventListener('click', () => {
    if (!navigator.geolocation) {
        return alert('Geolocation is not supported by your browser.')
    }

    navigator.geolocation.getCurrentPosition((position) => {
        socket.emit('sendLocation', {
            latitude: position.coords.latitude,
            longitude: position.coords.longitude
        })
    })
})
```

First up, check if `navigator.geolocation` exists. This will determine if the browser supports geolocation. From there, `navigator.geolocation.getCurrentPosition` can be called to fetch the user's location. The provided callback function will get called with the user's position, which includes the latitude and longitude.

With the client set up, the server can listen for the `sendLocation` event. When it's received, `io.emit` is used to share that location with everyone in the chat room

```
socket.on('sendLocation', (coords) => {
    io.emit('message',
`https://google.com/maps?q=${coords.latitude},${coords.longitude}`)
})
```

### Documentation Links

- MDN: Geolocation API

## Lesson 9: Event Acknowledgements

In this lesson, you'll learn about event acknowledgements. This allows the receiver of an event to send a message back to the sender of the event. This is useful for error handling and data validation.

### Event Acknowledgements

To explore acknowledgements, let's set up the server to screen messages for profane language. The bad-words module will let you check text for profane language.

```
npm i bad-words@3.0.0
```

You already know there are two sides to every event, the sender and the receiver. In the example below, the client is the one emitting the `sendMessage` event. The big change is the addition of the callback function. This function will run if/when the server acknowledges the event.

```
socket.emit('sendMessage', message, (error) => {
    if (error) {
        return console.log(error)
    }

    console.log('Message delivered!')
})
```

On the server, the event listener for `sendMessage` also has a small change. Aside from the `message` parameter, it now has access to the `callback` parameter. This callback function can be called on the server to trigger the acknowledgement function on the client.

The callback can be called with or without data. In this example, the callback is called with an error if profane language was detected. The argument would get passed to the client

where the error could be shown. The callback is called without no arguments if no profane language was detected. This lets the client know that the message was successfully processed.

```
socket.on('sendMessage', (message, callback) => {
    const filter = new Filter()

    if (filter.isProfane(message)) {
        return callback('Profanity is not allowed!')
    }

    io.emit('message', message)
    callback()
})
```

## Lesson 10: Form and Button States

In this lesson, you'll use a bit of DOM manipulation to provide users with a nicer experience.

### Form and Button States

First up is the form that allows users to send a new message. This form should be disabled while messages are being sent to the server. This will prevent duplicate messages from being sent if the user was to double-click the button. The form can be disabled by setting the `disabled` attribute on the submit button.

```
const $messageFormButton = $messageForm.querySelector('button')

// Disable button
$messageFormButton.setAttribute('disabled', 'disabled')

// Enable buttons
$messageFormButton.removeAttribute('disabled')
```

The interaction with the text input can also be improved. The text input should be cleared and focused on when the form is submitted.

```
const $messageFormInput = $messageForm.querySelector('input')

// Clear the text from the input
$messageFormInput.value = ''

// Shift focus back to the input
$messageFormInput.focus()
```

## Lesson 11: Rendering Messages

In this lesson, you'll learn how to use a client-side templating engine to render messages to the screen.

### Creating a Template

First up, include these in your HTML. Mustache will be used to render the messages. Moment and Qs will be used a bit later in the section.

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/mustache.js/3.0.1/mustache.min.js
"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.22.2/moment.min.js"><
/script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/qs/6.6.0/qs.min.js"></script>
```

Rendering messages to the screen will require two changes to the HTML. First up, a place needs to be created on the page to store the rendered messages.

```
<div id="messages"></div>
```

Second, a template needs to be created for the messages. The template below looks like pretty standard HTML. The only addition is `{{message}}`. This is the syntax used to inject a value into the template. In this case, the message text will be shown inside the templates paragraph.

```
<script id="message-template" type="text/html">
    <div>
        <p>{{message}}</p>
    </div>
</script>
```

Rendering a Template

The template can be compiled and rendered using client-side JavaScript. The snippet below renders a new instance of the message template to the screen whenever it receives a new `message` event.

```
// Select the element in which you want to render the template
const $messages = document.querySelector('#messages')

// Select the template
const messageTemplate = document.querySelector('#message-template').innerHTML

socket.on('message', (message) => {
    // Render the template with the message data
    const html = Mustache.render(messageTemplate, {
        message
    })

    // Insert the template into the DOM
    $messages.insertAdjacentHTML('beforeend', html)
})
```

Documentation Links

- Mustache.js

## Lesson 12: Rendering Location Messages

In this lesson, it's on you to render a new template for shared locations.

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 13: Working with Time

In this lesson, you'll learn how to work with time in JavaScript. This will allow you to show users when a given message was sent.

### Working with Time

Every message is going to contain a timestamp. This timestamp will represent the time when the server sent the message out to everyone in the chat application. The server will be the one to generate the timestamp, which prevents the client from lying about when a message was sent.

A JavaScript timestamp is nothing more than an integer. This integer represents the number of milliseconds since the Unix Epoch. The Unix Epoch was January 1st, 1970 at midnight, so the timestamp for the current point in time is a pretty big number.

```
// The getTime method is used to get the timestamp
const timestamp = new Date().getTime()

console.log(timestamp) // Will print: 1549481884646
```

Once the server sends the message and timestamp to the client, the client can format the timestamp before rendering it. The Moment library provides an easy way to format timestamps to fit your needs. For the chat app, showing something like “11:48 am” works well.

```
moment(message.createdAt).format('h:mm a')
```

You can find a complete list of the formatting options in the documentation below.

### Documentation Links

- Moment
- Moment formatting

## Lesson 14: Timestamps for Location Messages

In this lesson, it’s on you to add timestamps for location messages.

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 15: Styling the Chat App

In this lesson, you’ll add styles to the chat application. This will give it a professional and polished feel.

This lesson contains detailed instructions to apply the provided styles. Please refer to the video for a recap on working with the styles.

## Lesson 16: Join Page

In this lesson, you'll add a join page to the chat application. This will allow users to pick a username and join a specific chat room.

### Adding a Join Page

Below is the HTML for the join page. This will be `index.html`. The HTML for the chat page will move to `chat.html`. This will make sure that users visit the join page when pulling up the site.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Chat App</title>
        <link rel="icon" href="/img/favicon.png">
        <link rel="stylesheet" href="/css/styles.min.css">
    </head>
    <body>
        <div class="centered-form">
            <div class="centered-form__box">
                <h1>Join</h1>
                <form action="/chat.html">
                    <label>Display name</label>
                    <input type="text" name="username" placeholder="Display
name" required>
                    <label>Room</label>
                    <input type="text" name="room" placeholder="Room"
required>
                    <button>Join</button>
                </form>
            </div>
        </div>
    </body>
</html>
```

## Lesson 17: Socket.io Rooms

In this lesson, you'll learn how to work with rooms in Socket.io. Rooms allow you to separate users into groups, which is a great fit for the chat application.

It's the server's job to add and remove users from a room. The server can add a user to a room by calling `socket.join` with the room name. Below, the listener for `join` accepts the username and the room name from the client. `socket.join(room)` is then called to add the user to the room they wanted to join.

```
socket.on('join', ({ username, room }) => {
    // Join the room
    socket.join(room)

    // Welcome the user to the room
    socket.emit('message', generateMessage('Welcome!'))

    // Broadcast an event to everyone in the room
    socket.broadcast.to(room).emit('message', generateMessage(`${username}
has joined!`))
})
```

The `join` listener above also calls `to` as part of `socket.broadcast.to.emit`. The `to` method allows an event to be emitted to a specific room. In this case, when a user joins a room, only users in that room will be notified.

The `to` method can also be used with `io` to send an event to everyone in a room.

```
// Emit a message to everyone in a specific room
io.to('Center City').emit('message', generateMessage(message))
```

## Lesson 18: Storing Users: Part I

In this lesson, you'll create functions that allow the chat application to track which users are in which rooms.

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 19: Storing Users: Part II

In this lesson, you'll continue to create functions that allow the chat application to track which users are in which rooms.

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 20: Tracking Users Joining and Leaving

In this lesson, you'll use the functions created in the last two lessons to track users.

### Add and Removing Users

When a user attempts to join a room, their username and room name will be passed to `addUser`. This will allow the application to track the user, and it'll also allow the data to be validated. If an error occurs, that error message will be sent back to the client and the user won't join the requests room.

```
socket.on('join', (options, callback) => {
    // Validate/track user
    const { error, user } = addUser({ id: socket.id, ...options })

    // If error, send message back to client
    if (error) {
        return callback(error)
    }

    // Else, join the room
    socket.join(user.room)
    socket.emit('message', generateMessage('Welcome!'))
    socket.broadcast.to(user.room).emit('message',
generateMessage(`${user.username} has joined!`))

    callback()
})
```

When a user disconnects from the application, `removeUser` is called to remove them from the list of active users. If a user was removed, a message is sent to everyone in the chat room letting them know that someone has left.

```
socket.on('disconnect', () => {
    const user = removeUser(socket.id)

    if (user) {
        io.to(user.room).emit('message', generateMessage(`${user.username}
has left!`))
    }
})
```

The server uses an acknowledgement to send errors back to the client. The client can set up the callback function and respond to any errors that might occur. If an error does occur, the snippet below shows the error message and then redirects the user back to the join page.

```
socket.emit('join', { username, room }, (error) => {
    if (error) {
        alert(error)
        location.href = '/'
    }
 })
```

## Lesson 21: Sending Messages to Rooms

In this lesson, you'll use the tracked user data to send messages to the correct rooms.

### Sending Messages to Room

The `sendLocation` event listener below uses `getUser` to get the username and room for the user who sent the message. This allows the server to use `to` to emit the message to only users in that chat room.

```
socket.on('sendLocation', (coords, callback) => {
    // Get the username and room for the user
    const user = getUser(socket.id)

    // Emit the message to just that room
    io.to(user.room).emit('locationMessage',
generateLocationMessage(user.username,
`https://google.com/maps?q=${coords.latitude},${coords.longitude}`))

    // Send an acknowledgement to the client
    callback()
})
```

## Lesson 22: Rendering User List

In this lesson, you'll set up the chat application to show a list of all active users in the sidebar. With Socket.io, this list will be updated in real time.

### Rendering the User List

The client can only render data it has access to, so the server needs to send a list of all active users to the client. This data should be sent again whenever a user is added or removed from a chat room.

`getUsersInRoom` is used to get a list of all users in a specific room. The server then emits `roomData` to all clients in that affected room letting them know to rerender their user list.

```
// After a user joins or leaves
io.to(user.room).emit('roomData', {
    room: user.room,
    users: getUsersInRoom(user.room)
})
```

On the client-side, the sidebar template will be responsible for rendering the room name and the list of users. Rendering a list with Moustache requires a new syntax. In the template below, everything between `{{#users}}` and `{{/users}}` will be repeated for each user. In this case, the username is rendered in a list item that will show up in the sidebar.

```
<script id="sidebar-template" type="text/html">
    <h2 class="room-title">{{room}}</h2>
    <h3 class="list-title">Users</h3>
    <ul class="users">
        {{#users}}
            <li>{{username}}</li>
        {{/users}}
    </ul>
</script>
```

The client-side is then able to listen for that event and render the user list to the sidebar.

```
const sidebarTemplate = document.querySelector('#sidebar-template').innerHTML

socket.on('roomData', ({ room, users }) => {
    const html = Mustache.render(sidebarTemplate, {
        room,
        users
    })
    document.querySelector('#sidebar').innerHTML = html
})
```

## Lesson 23: Automatic Scrolling

In this lesson, you'll add automatic scrolling to the chat application.

This lesson contains detailed instructions covering automatic scrolling. The function used to enable autoscrolling is below, but please refer to the lesson video for a recap of how it was designed.

```
const autoscroll = () => {
     // New message element
     const $newMessage = $messages.lastElementChild

      // Height of the new message
     const newMessageStyles = getComputedStyle($newMessage)
     const newMessageMargin = parseInt(newMessageStyles.marginBottom)
     const newMessageHeight = $newMessage.offsetHeight + newMessageMargin

      // Visible height
     const visibleHeight = $messages.offsetHeight

      // Height of messages container
     const containerHeight = $messages.scrollHeight

      // How far have I scrolled?
     const scrollOffset = $messages.scrollTop + visibleHeight

      if (containerHeight - newMessageHeight <= scrollOffset) {
         $messages.scrollTop = $messages.scrollHeight
     }
 }
```

## Lesson 24: Deploying the Chat Application

In this lesson, it's on you to deploy the chat application to production!

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.