



# RESTful Web Services with Node.js and Express

by Jonathan Mills

[Start Course](#)[Bookmark](#)[Add to Channel](#)[Download Course](#)[Table of contents](#)[Description](#)[Transcript](#)[Exercise files](#)[Discussion](#)[Recc](#)

## What Is REST?

### Introduction

In the modern era of web development, most of our backend systems have been reduced down to just simple APIs. We're not necessarily serving up full web pages anymore from the backend. We deploy out our single-page app or our JavaScript framework, and then it just consumes APIs on the backend to get all the data it needs. And in this course, we're going to talk about how to build a nice, clean RESTful web service with Node.js and Express, and so we'll talk about how to build out all these APIs using these very simple tools. So in this course, the course as a whole, we're going to talk about how to build out a RESTful API with Node and Express. And it's kind of hard to talk about how to build out a RESTful API without first talking about what REST really is, and we're going to go through what all the different nouns are. You've got GET, POST, and then you've got PUT, and DELETE, and PATCH, and what all these different things are; and we'll talk through all of those; and we'll understand where we use which one. We're going to spend this module and the next module setting up our Node and Express environments. We'll actually get something up and running, we'll wrap some tooling around it, we'll make it easy to develop in, and then we'll move forward from there. We're going to implement all of the REST verbs; and I'm going to work through what they are; and each module is going to be split into the PUT, or the PATCH, or the DELETE, or getting and updating your information. We're going to wire up MongoDB. Now in this course, I'm only going to talk about MongoDB. Once you're done with this course, and if you want to wire up to SQL Server or something else, you can go watch my other course, Building Web Applications with Node.js, where I talk about some of the other databases as well. We're going to talk about



conversation about hypermedia and how implementing some links into your API returns are what make it very easy for somebody to navigate your API.

## What Is Rest?

Now before we can start building a RESTful API, we need to have a good understanding of what REST really means. Now the term REST comes from a dissertation written by Roy Fielding back in 2000. In this dissertation, he coins the term Representational State Transfer, or REST. In this chapter of the dissertation, he begins to lay out a series of constraints that should be in place whenever two systems talk to each other. So ultimately, what REST is is just a series of rules in place for your server so that everyone that uses your service understands what it does and how it works. Now let's take a look at these constraints and see if we can make sense of what he's talking about. Now the first constraint is the client-server constraint. All that means is that you have a client and a server, and the client sends a request to the server, and the server sends a response back to the client. Now that should seem pretty straightforward to you. In fact, this is the way the web works. And chances are, if you're watching this video, you do some stuff on the web, so this is how you understand most things to work. There's no tricks here. It's just that client sends a request, server sends a response. That's how that works. It becomes more complicated as we move forward. So the next constraint is the stateless server. As the load from the client increases on your server, you start to add more servers to the mix. And when that happens, now you get into a situation where the server may contain some information about the client that doesn't transfer from one server to another. In this constraint, Roy Fielding suggests that you don't get yourself into that situation. Ultimately, the client sends a request to the server, and it doesn't matter what server it goes to. So everything that the server is going to need to process the request should be included in that request. And then based on that request and based on all the information that's in the request, it'll send you a response. And if you find yourself on the server storing information about the request or about the client, then you're not writing a truly RESTful service. Now the next constraint is the constraint of caching. So ultimately, as the server sends more information back to the client, you're sometimes sending data that doesn't change very often. So think about a list of clients, or a list of book authors, or something along those lines. That data is not going to change between now and 5 minutes from now. And so the next time the client pulls that data, we want to know whether or not I should even pull it. And so what this constraint says is let the client know how long this data is good for so that the client doesn't have to come back to the server for that data over and over again. Now we're not





## Uniform Interface

The last constraint we'll talk about is the constraint of the uniform interface. Basically, what Roy Fielding is talking about here is that when you're dealing with an interface for a RESTful server, it will behave in a very specific way that is uniform from one service to the next. And in this, he outlines four pieces to an interface that should always operate this way, the first one being the idea of resources. Now whenever you're dealing with a RESTful service, you're dealing with a resource or a series of resources, and all that really means is you're dealing with nouns. The uniform interfaces are built around things, not actions. Sometimes when you call a service, you'll have something like authorize or login. Those are verbs, and those are not RESTful. In REST, everything we're going to be dealing with is nouns. So for example, in this project, we're going to be working with books and authors. Both of these are the resources that each of our REST services will be dealing with. For example, dealing with a book, the URL will be `http://, something, /Books`; and if we're dealing with authors, it would be `/authors`. That's the way a RESTful service will be defined every time. The second piece of a uniform interface is contained within the HTTP verbs themselves. The HTTP verbs that we use in our request will dictate the type of activity that we're trying to do on the resource. For example, an HTTP GET request will simply request data, so I'm trying to GET data, and the service will either return a list of objects or a specific object. The POST is used to add data. So if I send a POST to `/books`, the service is going to add a new book. DELETE will do exactly what it sounds like it will do. It's going to remove it. Now the next two are not quite as intuitive and not always followed as strictly. According to the rules though, PUT is used to replace an object. So if I change an object, I send the whole thing with a PUT, and the service will replace the old one with the new one. Sometimes when objects are large, then sending the whole thing back and forth can be cumbersome. So in that case, we have PATCH. In PATCH's case, the service will only update the specific parts of the object that are sent back to the service. Now the last piece of the uniform interface is the acronym HATEOAS, which has to be either the coolest or weirdest acronym that we have in REST. HATEOAS stands for Hypermedia As The Engine Of Application State, which I know is not very descriptive at all. Basically, all that means is that in each request, there will be a set of hyperlinks that you can use to navigate the API. That's just the API's way of letting you know what other actions you can take on this particular resource. Don't worry about that one too much right now. We'll get into that later on in this course.

## Setting up Your Environment





need to do in order to build out a RESTful API in Node.js is to download Node.js. So we're going to run out to nodejs.org and download the LTS version. So for me, this is 8.12. Depending on when you're watching this, that number may change. But as long as it's after 7, whatever version you have is after 7, you should be good for the features we're going to use. Node does a very good job of being backwards compatible. So if this 11 or 12 that's LTS, don't worry about that. Just go ahead and download it and use it. Things will still work just fine. Now once you get this downloaded and installed, we're going to get started. So go ahead and open up your IDE of choice, and I don't care what IDE you're going to use. That's up to you. Whatever IDE you like. I'm using VS Code, and I like VS Code because its feature set is still pretty lightweight and it offers a lot of features. I'm not going to use any of the features of Code, except for the integrated TERMINAL window, in this course just because I want you to be able to use whatever IDE you want and not get confused with bells and whistles. So I'm going to go just the basics for VS Code and get started. So if we come down into the integrated TERMINAL window for this directory I'm in, I'm going to do a `node --version`, and it's going to say 8.12 .0. That's great. That's the version that we just downloaded. Make sure that that lines up for you. Now the first thing we do when we create a new Node project is we're going to an `npm init`, and what this is going to do is it's going to ask you some questions. And so my package name is going to be `api`, version 1, description nothing, entry point, `index`. So we're just going to kind of take the defaults for the rest of this stuff. And what it's going to do is create for us a `package.json` file, and you see it appear over here in my terminal window. What `package.json` is is basically all of the information about your project bundled up in one place, and all of the dependencies of this project are going to be stored in here. And let's actually get one dependency going so I can explain the rest of this. Let's do an `npm install express`, and it's going to go down, and it's going to pull Express down for you. Now you'll notice if you've been doing Node before, I didn't do a `--save`. And in the more recent versions of npm, they've eliminated the need to do `--save`, so you don't need to do that anymore. And you'll see I have `express` that it shows up right here, dependencies 4.16 .4. Now this caret right here means that any version 4 will download the latest. So if by the time this comes out we're up to 4.18, 4.20, whatever, if there's a caret here, it's going to download whatever version we want. If you're having issues with compatibility of a project or a package, you can always do `npm install express@ 4.16 .4`, and that will actually go download the specific version necessary for you. I'm going to leave the carets here because for the most part, they're not going to matter. But just know if you hit a problem in this course and a package isn't working, make sure you're using a package version that's similar to what I have. And if there's too many problems, we'll post

🗨 something down in the comments just to make sure everybody knows what's going on. Alright.



to live. Now since we have a reference to Express already where we have the Express package installed, I can just go `var express = require('express')`. Now that I have a reference to Express, I can use it, and we use it by going `var app = express`, and we execute that. Now before you freak out, you know, you may be watching this and say why are we using `var`? Well, I'm going to show you something in the next clip, and I want to have `var` here as something that it catch onto. So we'll use `const`, and we'll use `let`, and we'll do all the ES6 stuff throughout this course, but I'm going to leave these vars here so that you can see later on when that comes about. Okay, so we have Express. Now we need a port for it to listen on, and we're just going to create a port. And the way you'll see this a lot of times is you'll see `process.env.PORT`. And the way this works is we'll have a tool that will pass in this port into our application. Now we don't have that set up yet, so we need to give it an alternative, and here, we'll just say or 3000. Okay, I have app. I have my port. Now I need something to handle some routes. And for right now, we're going to keep this incredibly simple. We're just going to do an `app.get`. And basically, what this means is every time I get a GET request to slash, I'm going to respond with a function, and this function is going to have two variables passed into it. It's going to have the request and the response. So we can look at the request to see what's happening, and we can do stuff to the response to send it back. And in this case, we're just going to do a `res.send`, Welcome to my API! Alright, now I've got my GET handler, I've got my port, and I've got my app. Now we just need to kick this thing off. So we're going to say `app.listen`, and we're going to listen on the port. Now once we've started listening, we can actually spin out to the console. So let's just do `console.log`, Running on port, port. Alright. So if we go back to the TERMINAL window, and I type `node app.js`, it's going to fire this up. And what's going to happen is you'll see this running on port and then the port number, so port 3000. If we open Chrome up and I go `localhost:3000`, there's my Welcome to my API! Alright, this thing's working. We've got an app. Let's take a minute, and let's hook up a couple of tools and wrap some things in some tools before we go any further just so we make sure we keep running on the right path.

## Setting up Some Tooling

Now that we have the basics working, let's take a step back for a second and wrap some tooling around our environment. So we're going to set up two tools. The first tool is ESLint to lint our code and make sure we're writing code the proper way, and the second tool is nodemon, which is going to handle restarting our application, and passing in the port, and things like that. So we're going to come down here, and we're going to hit Ctrl+C just to close out our application, and we're going to do an `npm i eslint`, and we're going to do a `-D`. And if





do that is when we deploy this out somewhere and we npm install out in production, it's not going to take the time to install the dev dependencies. It's only going to install the actual dependencies. So once this is done, now you'll see in our devDependencies, we have this eslint, and we're at 5.8 .0. I want to run an eslint init, and I actually want to run it from the command line right here. But in order to do that, I would have to install eslint globally. And I have a general rule in my life, which is I don't install npm install globally whenever I can avoid it. And one way around that is we can use this scripts area up here. When you run something from scripts, it looks in your node\_modules first, and so we're going to run all of our linting from our scripts section, and that will avoid us having to install ESLint globally. So what we're actually going to come down here and do is we're going to create a lint task, and what lint is going to do is just run eslint. Alright. We're going to save that. And we're going to do npm run eslint. and then we're going to do --, and what that does is says hey, everything after this, pass on to whatever we're running. I'm going to do --init, and I'm going to run that. Except, obviously, I didn't, and now look. Look at this helpful error message. Npm has come a long way, and it's doing a lot of really cool things. Now obviously, I didn't want to run ESLint. I meant lint. And so npm has gotten some very cool features built into it now that makes your life just a little bit easier. Alright, here we go. Now when I run this, it's going to ask some questions. We want to use a popular style guide, and we're just going to use airbnb. Airbnb has become kind of the default for a lot of people. I like the airbnb option, and so that's what I'm going to go with. And we are not using React. Now notice that it's got a default. We want a JavaScript config file, and we do want to install everything right now with npm. Now what that's going to do is it's going to go and pull a whole bunch of stuff for us. It's going to do the eslint-config. It's going to do the plugin-import. It's going to build all these other things into our dev dependencies, and you'll see them pop up here in just a second, and now ESLint is working. Alright. Now if we come up here to lint, now we want to run eslint., and we'll save that. We're going to npm run lint. So what ESLint is going to do is it's going to check app.js and validate that we're writing our code properly. And when we run it, remember, I alluded to this is the last clip, it's going to blow up for several things, a lot of them, actually. And it's just some spacing, and notice it said Unexpected var, use let or const instead. And then there's indentations and spacing things that we need to fix. And I just going to come over to app.js, and notice, here's a var. We're going to use const. Here is another var. We're going to use const. And one last one. We're going to use const. For all the other things, we can just go right-click, and we're going to Format Document, and see how it added some spaces. I'm going to save that. Let's run our lint again. Okay, we've got a couple. Ah! Expected indentation of 2 spaces, Unexpected string concatenation, and a



ewline required at the end. So let's clean those up. It wants a newline there. Last one is our





together anymore. We can just use the little tick and do \$ port, and now that works. Okay, now it wants two spaces instead of four. And I actually prefer two spaces, but VS Code here defaulted to four. So I'm going to come down to this bottom section right here. Look down here. I can click that, and I can say we're going to indent using spaces 2, and we're going to Format Document, and notice it pulled those over. Save that, run lint one more time, and now it's good. Alright. Now the second tool we're going to use nodemon, and we're just going to do an npm install nodemon. So let's run that, and let's go back to that package.json so you'll see it pop up and what version that we're going to use, and there you see 1.18.5. Now what nodemon is going to do for us is it's going to handle the environmental stuff, but it also is going to watch our files and restart automatically if anything changes. And the way we're going to do that is we're going to come up here, and we're going to do start, and we're going to say nodemon app.js, and that's going to start our app. So just like when we did node app.js, here, you have to do nodemon app.js. Now there is a section we have to add of all of the nodemon config. And you know, I'll just drop it in here, and you'll see restartable. And if you just type rs, the delay is how long the delay is going to be from the time a file changes to the time it restarts. And then notice I'm passing two things into my environment, PORT and the environment, so NODE\_ENV, and here it says development. So I'm going to save this, and now I'm going to run npm start. Now remember when we did lint? I had to do npm run lint. Now I just have to do npm start. Npm test and npm start are two that you don't have to do anything with. I'm going to hit start, and now you notice I'm running on port 4000. I'm not running on port 3000 anymore. So in app.js, it recognized this process.env.PORT as 4000, and it pulled that in. So if I come back over here, I now have to change this to 4000, and there is my API. Now if I come in here, and I change this to my Nodemon API, and I hit save, notice down at the bottom, it restarted after about 2 and a half seconds. If I come back over here, and I Refresh, there is my Nodemon API. Alright. We've got some tooling. We've got ESLint running. We are ready to go. So let's start implementing some of our RESTful API.

## Summary

We are one module in our journey to build a RESTful API with Node.js and Express. And we haven't gone very far yet, but we have done just a little bit. So we talked about REST. We talked about the fact that it's resource based, or nouns, and we talked about how you do actions based on the HTTP verb you use. We created an API. It's a really simple API, but we have created an API. We've got a GET handler that returns something when you do a GET. It's working. We installed ESLint to help us get moving on linting so that we know that the code





application automatically when we change code. It's going to pass in the environmental variables that we need. That gets us pretty far along. So in this next module, we're actually going to start building out an actual API. We're going to start getting data, which means we're going to have to install MongoDB, and we're going to have to start implementing all of the HTTP verbs.

# Getting Data

## Introduction

Hey, welcome back. And in this module, we're going to talk about getting data from our API. Now we've started building out a RESTful API with Node.js and Express, and we haven't done a whole lot yet. We've just set up the basic structure, and we'll recover that here in just a second, but what we're going to do in this module is actually implement the HTTP GET verb that allows us to pull data out of the API. We're going to allow them to pull a list of items or pull just one item. We'll how to search for items. We'll get all of that covered in this module. We're also going to wire up MongoDB. Now we're going to use Mongoose in this course because it's an ORM that just kind of bundles all that stuff up for you. So we're going to dig into that just a little bit in this module as well. And then we're going to do a full-blown search for items, so search by ID or search by different things, in our query string. So hopefully you're ready to go. Let's go implement GET.

## Implementing HTTP GET

Now that we have a basic site in place, let's start building our API by building a router that supports HTTP GET. So here, you can see we've started to build out our RESTful API, and here in app.js, we've got just a little bit of code written already. First of all, we've got our express and our instance of express that's going to handle everything for us; we've got our port that we're pulling in from nodemon that we're going to use to listen; then we've got our app.get, which is when a request is sent to the port, it's just going to respond back with a Welcome to my Nodemon API! ; and then at the end, we've got this little section that actually does the listening on whatever port is in our nodemon config. And speaking of our nodemon config. If we go into our package.json, there's a section down here at the bottom called nodemonConfig that actually will have an env section where you see we're setting the NODE\_ENV and our







you'll see this code in the materials. And what's going to happen is you're going to copy that code over somewhere, and you're going to open up, and you're going to try and run it. And so you'll just do an `npm start`, and you'll notice this doesn't work. I get errors everywhere. And the reason for that it has to do with the `.bin` file, and sometimes it doesn't get copied over depending on where you're at. So what we're going to do is we're just going to delete `node_modules`, so just delete it completely. We'll just move it to Trash, and then we'll do an `npm install`. And then once we've done that, that'll clean that all up so we can do an `npm start`, and now it works. So that's how to clear that up in case you run into that issue. So if you're here, and you see this screen, you're ready to get started on our API. Now the first thing we're going to do is start building out the GET routes for our API. And you can do the `app.get` route, but really, what I want to do is encapsulate all of our router code into one place. And the way we're going to do that is by creating a router that'll do that for us. And so we're just going to sit here, and we're going to do `const bookRouter`, and `bookRouter` is what's going to deal with all this for us, and that's going to be an `express.Router`. Now let's take just a second, and let's clean this up a little bit. Let's put all of these things we're creating in one little section just like this. Now the way our router works is you take the router and you just do a `.route`, and you're going to configure what route that's for. And so for us, it's going to be `books`. We're going to configure the `books` route on `bookRouter`, and it's going to do `.get` in this case because that's what we're implementing. We could do a whole bunch of other stuff. Here, we're going to do `.post`. We're going to do `.put`. We're going to do `.patch`. We're going to do all of those things on this router. But for right now, let's just handle the `.get`. Now this is going to work the same way that `app.get` worked. Right? So we're going to come in here, and we're going to create a function. And in that function, we're going to take the `req` and `res` inputs, and then we're going to create our arrow function, and then we'll put everything else inside that. And for right now, what we're going to do is just have, let's have a holder, so `response =`, and let's just create a dummy object, say `hello` and then `This is my API`. Now we need to send that response back to whoever calls our API, and the way we do that is with `res.json`, and that's going to send a JSON response back instead of like we've done a `res.send`, which just sends the test, or a `res.render` if you were looking at my other course where we do a web app, and that's going to render something. Here, we're going to do `res.json`, and we're just going to send the response back. Let's close that out, and then we're almost done with our router. so the only piece left is wiring this up, and the way we're going to wire it up is with an `app.use` statement. And the first thing that gets passed into an `app.use` is the route this is going to serve, and so we're going with `/api`. Let's go with that. And then we pass in `bookRouter`, and that's all it takes to get this route





## Wiring up to MongoDB

Now that we have our router set up serving HTTP GET requests, let's spend some time getting MongoDB set up so we can serve some real data. We're going to use MongoDB for the remainder of this course, and so the first thing you need to do is actually go download and install Mongo. So go to [mongodb.com](https://mongodb.com), click on this Get MongoDB, and I'm not going to walk you through the rest. Now we're going to download it, and depending on the system you're on, whether it's Windows, or Mac, or Linux, or whatever, just download it here, follow the instructions, and get it installed. Now you want to get to the point where you can open up a new terminal window, and type `mongod`, and hit Enter. And if you get this far, you are set. If you have any trouble installing Mongo, just go back out to the Mongo website, work through the instructions, and if you're really stuck, put something down in the comments, and we'll help work you through it. Alright. Now once you've got `mongod` running, we're going to open up a new window. So I still have the first one running, which is my node, and that's still going. I now have a third one open. And what we're going to do with this one is you'll notice I've added two files. I've got a `books.json` and a `DataImportInstructions`. And so what `DataImportInstructions` is is just say hey, we've got some book data that I'm handing you, and it's in this `books.json` file. We're going to run this command right here, and what that's going to do is take this list of books and insert it into our Mongo database, so `mongo bookAPI`, less than, `books.JSON`. I'm going to hit Enter, and then you'll see right here `nInserted 8`. So I've got data in my database. It automatically creates the database. It automatically does everything we need. Notice it says `db.books`. It's just going to drop it in books. Don't worry about this part. If you get this far, you're set. And now we're going to go try and see if we've gotten everything we need. So let's come back over to `app.js`, and let's start looking at how to pull things out of Mongo. So the first thing that we're going to do is use a package called Mongoose, and what Mongoose is going to do for us is deal with all of the database stuff for us. We don't have to deal with MongoDB directly. Mongoose is going to do that for us, which means that we now need to do an npm install mongoose. And so once that's done installing, we'll head over to `package.json`, and we'll see, right there, 5.3.10. So now that we have Mongoose installed, we need to start using it. And the way we're going to use it is first of all by connecting up to our Mongo database. So let's come up right up here and open up a database connection using `mongoose.connect`. Now the URL that we're going to use is `mongodb://localhost/bookAPI`, and that `bookAPI` comes from what we used when we did our import. We did a `Mongo bookAPI`, and then we imported our books. So that's what's going to get us a database connection to MongoDB. Now the next





the book model. And so we're just going to create that in a new directory called `models/bookModel`. We're just going to stick that out somewhere else. And so let's actually go over right now and create a new folder called `models`, and then we'll create a new file called `bookModel`. Now once we get in here, of course, the first thing we want to do is get ahold of `Mongoose` again, so we need to do our `mongoose = require('mongoose')` the same way we did it over on the other side so that we have that all set up and done, and we can use it in just a little bit. Now we also need the schema, so we're creating a `Schema`, and we're going to destructure this for `Mongoose`. It makes it just a little cleaner, a little simpler. So `Schema = mongoose.Schema` just pulls `Schema` out of `Mongoose`, and now I have access to that directly. And the last thing we're going to do is actually create the `bookModel`, and that's a new `Schema`, so we have to new it up. Capital S means we new it up, and we're going to new up that new `Schema` right here. Now one thing you'll notice, I'm back to four spaces, and `ESLint` says we only want two spaces. So we're going to take a second right now, and we're going to reset the default for VS Code. In case you need to know how to do that, we actually are going to go to Code, Preferences, Settings. And in Settings, we're just going to do a search. This is the easiest way to get to it. We just search for indent, we're going to scroll down to where it says 4, and we're going to change that 4 to 2. Alright, now it's changed, and we're done. And so you'll note that I'm set back to 2. And now the `bookModel` is just going to be a series of properties and types. So we start with `title`, and `title` is going to have a type of `String`. And a lot of these are going to be a type of `String`, so let's just copy that, and we'll reuse that for the rest of this thing. So let's do `author`, that's also a type of `String`; let's do `genre`, that's also a type of `String`; and then let's do `read`, so whether or not this thing is read. Now that's not a type of `String`. That's going to be a type of `Boolean`, true or false. That's all it is. And let's actually add a default to this one just more so I can show you hey, you can do a default, and we're going to default that to false. Now we didn't wrap everything in curly braces like we should have, so let's get our curly braces wrapped here. Now once that's done, we will see one more issue, and that one Missing trailing comma. `ESLint` and the Airbnb style want us to have trailing commas, and I don't like that. So let's go into `eslinttrc`, and we're actually going to add a rules section. And this rules section basically allows us to set up some rules. This one's called `comma-dangle`. It just means I don't want that comma at the end. So if we set 0, it goes away, and I don't have to worry about it anymore. If you see rules that you don't like, this is the way you can get rid of them in `eslinttrc`. So now you'll notice that error is gone. I don't have to worry about it anymore. Now we're just going to clean up our `bookModel` a little bit, and we do a `module.exports`, and we're going to export that `bookModel`. And we're going to export it by doing `mongoose.model('Book', bookModel)`, so that's what we're defining the model as; and then we're passing in `bookModel`; and now I can use this





So if we come down into our bookRouter, right now, we are just returning a response, hello, This is my API, every time we do anything. Now we don't want to do that anymore. What we actually want to do is query the Mongo database and get some actual real data. And so let's just get rid of this `const response =`, that doesn't need to be here anymore, and we're going to be a `book.find`. And what `book.find` is going to do is just going to, it's going to look in the `bookAPI` database in the `Book` collection for everything in this case because we're not going to pass in a query. We'll talk about queries a little bit later. So what it's going to do though is it's going to take an arrow function. In standard Node fashion, this callback is going to be `err` first or `books`, so I'm either going to get an error or I'm going to get some books back. And so as we finish out this arrow function, we're going to check to see hey, do we have an error? If we have an error, then we're going to send that error back. So we'll just use a `res.send` to send back that error otherwise, so else, we are going to send JSON back. So we'd use `res.json` just like we did down below with the response, `res.json books`, and that's really it. Except, there's a couple issues. So we'll clean this up first. And then, if you look at our arrow function, it says Expected to return a value. And so think about it this way. Right now, I have `res.send`, and then I have `res.json`, and I could potentially return something twice, and we want to avoid that. And one way we avoid that is whenever we have a `res.send` or something, we just `return`, and then that'll break out of the whole thing. And so let's add some `return` statements here, and so I guarantee I'm not doing two. But now that I'm returning, I've got this `else` is not necessary anymore. And so I can just get rid of the `else`, and now I'm guaranteed I'm only ever going to `res.send` one thing. Alright. Now that that's done, let's go check this out and see what I've got going on. So let's make sure we've got an `npm start` going if you've stopped it or if you've copied something over, so make sure `npm` starts up. And then let's go out to our browser, and let's rerun this thing, and now you'll see hey, we've got books back. So if you've got books back, you're in great shape, and you have a basic API working. Now let's add the next piece, which would be find one item.

## Filtering with a Query String

Okay, now that we have some data in our MongoDB database and we're returning a list of books, let's work on filtering that list using our query string. Now the first thing I want to do when we start filtering things is I want to be able to implement a query string parameter. So if I say? `genre=Fantasy`, I just want to see the fantasy items in my list. And right now, that's not working. And the way we're going to do that is use the built-in Mongoose `find`, so `find` will actually take a parameter called `query`, and it will filter the list based on whatever is in that





object that has genre fantasy. Now the cool thing is in my request, I actually have a thing called query that's really exactly that. It takes the query string and creates an object out of it. Now here, the way I'm doing it it's not very ES6 friendly because I want to use destructuring the same way we did before. So we'll change this out and just pull query right out of the request. And now, whatever you type in the URL is going to be split into a JavaScript object that we're passing into MongoDB. So now, look. If I hit Enter now, I'm filtering based on what's in my URL. I only have fantasy items. Now here's a problem. If I just type something weird, fsfs and some weird stuff, and I hit Enter, it actually is passing that down into Mongo, and so nothing comes back because, obviously, I don't have that in my Mongo database. So what we want to do here is fix that. I'm going to actually change my, the way this code works to only take things that really matter. So let's do this. Let's get rid of all of this stuff, and let's actually just create query as an empty object just like that. And then what we're going to do is say if req.query has genre on it, well, then let's use that. And so I can just come in here like this and then say hey, query.genre = req.query .genre, and I can filter the things I'm allowing based on this. Alright. So now, if I flip back over, my query string should be cleaned back up. So if I just put something weird in there again, then when I enter search, it's going to ignore that, and now everything is going to come back. But if I do genre=Fantasy, now I just have my Fantasy books. Alright. We've got search down. Now let's go build just find a single item.

## Getting a Single Item

We just implemented a filter that allows us to filter by items in the query string. In this clip, let's work on finding just a single item. Now when we say find a single item, what we really mean is I want to take this id right here, and I want to add it to the end of my query string and have that work and pull back just that single item. Now you can probably tell by the length of this clip that this is an incredibly simple thing to do. So let's do that. We're going to add a second route. Let's close this terminal window. There we go. And the second one is going to be similar to the first one, and so let's just copy this, and we'll paste it down below. But instead of /books, we're actually going to add /:bookId. And when we add/:bookId, what that's going to do is give us a variable called bookId that we call pull back out and use a little bit later on, and that's going to be this thing right here is going to be what's put in bookId. And now let's get rid of query. We don't need that. And let's come down and we're going to change this to findById instead of book.find. And now let's get rid of query, and instead, we're going to take that param bookId that's being passed in, and it's going to be stored in req.params, so we're just going to drop that in right here, and let's clean up our plural/singular thing. We'll go from Book to books to





the URL and drops it in. So there we go right there. Let's do Les Mis. We'll just paste it, and we have Les Mis.

## Summary

So we've come pretty far on our journey to build a RESTful API with Node.js and Express. And by pretty far, I mean we've implemented the HTTP GET verb. We've got one verb down, but that's quite a bit for where we were at. So really, we've built two routes. We've built one that pulls back a list of items, and we've built one that does just one item, and we did that by wiring everything to MongoDB. We've got a list of books in a database, and now we're pulling that list out to display either the whole list or just a single list. We also allowed you to search for items, really, just genre by putting genre up in the URL, and then we filter our list based upon that. So we've got GET down. Now in the next module, let's build POST so we can start adding new data to the database.

# Posting Data

## Introduction

Alright. So we are making progress on our RESTful API. And in this module, we're going to start posting data to our API. So we're working our way through this process of building out a RESTful API with Node and Express, and we've come quite a ways actually so far. We can get a list of items, we can get a single item, we can filter, and so we've made some good progress. And in this module, we're going to start implementing the HTTP POST verb. And ultimately, what that's going to do for us is allow us to add new items to our list in the database. So we're going to send some data over to our API, and that's going to add it to the database for us. And we're going to test with Postman, and the reason we're going to test with Postman is because it's going to help us add the POST verb, and the PUT verb, and all of these other verbs that we're going to start using in a much simpler way than how the browser would do it. We're also going to do a little bit of code cleanup. Right now, most of our code is in app.js, and I don't want to leave it in app.js. So we're going to pull some of our code out into a better-defined structure so that it's easy for us to deal with.

## Parsing POST Data with Body Parser





and add a `.post`, which is going to operate kind of the same way `.get` does. Right? It's going to take two parameters, `req` and `res`, and then those things will be passed into a function that's going to deal with whatever gets posted. Now ultimately, what I want to do is I want to use the Mongoose functionality here, so we're going to say `const book = new Book`. And whatever gets posted to this route, we're just going to create a new Mongoose Book object out of it, which would be kind of great. And so we're just going to pass `req.body` right into `new Book`. Now we do have a problem with this though because `req.body` doesn't exist yet. We do have one npm package that is required for us to make this happen. So let's do `const bodyParser = require('body-parser')`. Alright, so with that, we need to do an npm install. So let's do an npm install `body-parser`, and when that's done, if we run over to `package.json`, we'll see 1.18.3. Excellent. So back in `app.js`, now that we have body parser installed, there's two things we need to add, just two lines of code that'll make body parser work for us. We're going to do `app.use(bodyParser.urlencoded({extended: true}))`. And that's going to set up the first piece. Then we're also going to have to do one other piece, `app.use(bodyParser.json())`, and that just kind of sets the whole thing up so that it'll pull JSON out of the POST body and give it to us in that `req.body`. Alright. So if we go back down in here, once Mongoose gets `req.body`, it's going to create a new book for us. And so let's just look and see what that is. Let's do a `console.log(book)` and then send back that `res.json(book)`. Send back that book. And it'll be interesting to see what we get back, and we'll play with that here in just a second. Now one thing we want to remember, so always return that. It doesn't matter so much here, but it's the getting in the habit. What we need to do now though is actually post something, and you can't really do that simply from the browser. So we're going to add a tool that we'll use to do this. Let's go download Postman, and we'll use Postman to drive a lot of this stuff for us.

## Testing with Postman

Now that we have our POST written, let's test it using Postman. Now in this course, we're going to use Postman to handle all of our API calls from this point forward. So run out to [postman.com](https://postman.com) and click Download the App. Now this used to be part of Chrome, and this is kind of an extension you can build in, not anymore. Now you actually have to download it for your environment and then run it as a desktop application. So go ahead and click Download, get it downloaded, and then let's fire it up and see where it takes us. Alright. Once you get it installed and launched, you'll be met with this screen. We're just going to close that out. We're just going to do anonymous stuff for right now. And when you get here, in the GET box, you just type `localhost:4000/api/books` just like always, and we're going to hit Send. And you'll see





It's working. Let's copy this one because we're going to create one right now in the next tab. So click the plus and change the GET to POST; same URL, localhost:4000/api/books; but now click on Body and raw; and we're going to paste this JSON back in. Let's get rid of the id because it's going to create a new id. Let's just make some stuff up here. So this'll be, let's say, Jon's Book. Excellent. And Jon's Book is obviously fiction because I don't write, and then the author is going to be, let's just say, me, so Jon Mills. Excellent. Now let's change from Text to JSON because remember, we're sending application/json, and this will automatically set the header for me. So it'll set the content type header automatically. I don't have to worry about that anymore. Remember, we do a default on read, so let's delete that, and then let's click Send and see what I get back. So if I pull this up a little bit, notice the response I got back has an id, it has a rend, all of that stuff is there, and let's pop over to our code. Remember our console log? There it is right there, read, the id, all that stuff is there. Excellent. So the code is working, but let's not jump ahead of ourselves. Because if I go back over to our GET, and I delete the id, and I hit Go, it's not in this list. So we're not actually saving it to the database yet, which is because we haven't saved it yet. That's the next thing to do. So now that this is working, let's go add save to our database call.

## Saving Data

So now we're posting a book to our API, but it's just being written out to the console. We're not doing anything with it yet. Let's look at how to save this to MongoDB. Alright, so coming back over to our code, right down here, you'll see we have `const book = new Book req.body`. And so that creates the new book in Mongoose, but we're not saving it yet. And the only thing you have to do for that is `book.save`, and that's it. Now that book is saved to the database, and we're ready to go. Now one thing we want to change real quick though is right now we're sending `res.json book` back, but really, what we'd like to do is add to that a status. So when we're creating something new, we set a status of 201. So here, I can set a 201 status and the book back all in one little bundle. So we're going to save that, and then we'll go try it over on Postman again. Alright. So now that we're back in Postman, we're going to click Send just on what we have already. Remember, we created this just a minute ago. And we're going to click Send, and that's going to give us back, notice we have a new id. And we can actually see that it saved to the database by going back to our GET, hitting Send, and scrolling down to the bottom. And notice, there is our new book saved to the database, pulled back out of the database, and the id is the same as the id we got back over here on our POST.





our code layout. Let's take a clip and clean up our code. So now that we have all of this code working, let's start to pay a little bit of attention to how it's laid out. Because if you look right here, look at all of this code we have to define our router. And it's all sitting here in app.js, and that's just not very good. So let's take this `bookRouter`, see? Instead of doing `expressRouter`, let's actually do the same thing we do for `Book` here. So `const bookRouter = require`, and let's just pull it in from somewhere. So let's pull it in from `routes/bookRouter`, and then we'll take all of this `bookRouter` code and we'll stick it here in this function. Now we're going to need some things. We're going to do this a little differently. We're actually going to execute this, and I'll explain why here in just a minute. But we're going to go ahead and add that right now. So let's create a new folder called `routes` and a new file called `bookRouter.js`. Awesome. Now the first thing we're going need to do is `const express = require express`. And the reason why we do this is because we're going to need access to `router` here in just a minute, so let's do that. And we're going to create a function called `routes`, and this function is what's going to create our `bookRouter` for us and then return that back out, which is why we were executing it. And then we'll just to `module.exports`, and we'll export that function `routes`. Now inside our `routes` function, we're going to take all of that router code from `app.js`, so everything from, for here, line 14 down. We're just going to take all that code, and we'll just cut it, and it's gone. And then we're going to go over into `bookRouter`, and we're just going to paste it. Just like that. Just that simple. Now there's--- we don't know what `bookRouter` is because we haven't done that yet. So let's do `const bookRouter = express.Router`. That gives us back our route so that we can, you know, add all these routes onto it. Okay, and then then the only other things is remember, this `routes` function gets exported, and it goes through and it does all this stuff, but then I need `bookRouter` to actually come back out of the `routes` function. So down at the very bottom, we're going to return `bookRouter`. So if we go back over to `app.js`, now when we do `const bookRouter = require routes/bookRouter` and we execute it, it's going to return that `bookRoute` back. So we haven't really changed anything yet. We've just pulled some code out. But notice, we need access to `Book`, and we don't have `Book`. So let's talk in the next clip about how to deal with that.

## Injecting the Book Model

We have our code extracted into a separate router file, but we need access to our book model. Let's talk about how to inject that into our router. Alright. Now to get access to `Book`, it actually is pretty simple. Just right here on line 9, we just need to pass `Book` in, and we're doing this so that we can keep our Mongoose stuff all separate and do `Book` there. Now part of





authorRouter, and then same thing here, authorRouter. And then instead of passing in Book, you would pass in your Mongoose model for Author, and then you'd go create those routes on the other side. Okay. The only other thing we need to do is we need to go into our routes, bookRouter, and actually use Book. So here in the function, we want to bring in Book so now that we have access it all throughout. We'll fix our space issue. Now let's try this thing. Let's open Postman back up, and here, we've got a book. Let's do a Sequel, so I wrote a second book. Let's hit Send, and there's my book with an id, and notice I got a 201 status back. Let's copy this id just to make sure, go back over to my GET, and run that, and yup, there we go. Everything works. Alright. So we've got GET done. We've got POST done. Let's take a minute and kind of recap a little bit, and then we'll work on updating.

## Summary

Alright. So as we're working our way through building out a RESTful API with Node and Express, we've added quite a bit over the course of this last module. We implemented HTTP POST, and HTTP POST allowed us to send data to our API to add new items to our list, and so now we can log in, send something to our API, and that's going to add data to the database. We also are now testing with Postman, and that's much easier than trying to do it in the browser, and this has some history to it, it lets us copy and paste things, and it actually handles all the verbs for us. So far, we've only done POST with it, but we're going to add a whole bunch more to it over the rest of this course. We also did some code cleanup. We pulled a bunch of stuff out of our app.js file, and we created a routes directory where we're going to handle all of our routes in the separate space. So stick with us, and we're done with POSTing. Now let's start worrying about how to edit some data.

# Updating Data

## Introduction

Alright. So we can get data. We can create data. Now in this module, we're going to talk about updating data in our API. Now we've been walking down the path of building out a RESTful API using Node.js and Express. And in this module, we're going to implement the HTTP PUT and PATCH verbs that allow a user to update individual items. Now in PUT's case, it will do a full replace of one resource with another resource. And in most cases, that will be all we need,





this conversation, we're also going to start to talk about implementing middleware functions. Now we've seen some middleware in use already with body parser, and that adds the body to the request. But this time, we're going to actually create our own middleware to add a book to our request, and then we're going to close out this module by finishing out all of our HTTP verbs by implementing DELETE. Alright. So let's get started by implementing HTTP PUT.

## Implementing PUT

Alright. So let's look at implementing PUT. What we have now is part of an API that allows us to get a list of items, POST a new item back to our API, and then another set of routes that allow us to pull back individual items based upon a bookId. And so what we're going to do now is implement the PUT verb, which basically means that we're going to allow a user or a consumer of our API to replace one item with another item that they have edited. What we're going to do is come down here, and we're going to do a `.put`. And PUT, just like GET and POST, takes a function with `req` and `res`. Now in order to update an item out of our MongoDB database, I first have to get the item that I'm editing, and so we need to do a `Book.findById` with the `bookId`, and as a matter of fact, it's almost exactly the same thing that we have up here. So what we're going to do is copy that, and we're going to paste it right like this. Alright. The error handling is going to be the same. So if I have an issue here, I'm going just send back the `err`. And if I do have a book though, we're going to replace the contents of that book with what's come through on the request. So we're just going to walk through and do this. We've got `book.title = req.body .title`. We've got `author` with `author`, `genre` and `genre`, and `read` and `read`. And then once that's all done, we've got all this listed out, we're going to do a `book.save` to save our changes. Now notice we now have an error, and it does not like the fact that we're changing the contents of something that was passed in up here in the parameters. And so in this case, with Mongoose, that's just the way it's going to work. I mean, that's the way we do it. So we need to find a way to turn this off. And I already showed you how to modify the `eslint-rc.js` file, and so we're not going to that, and let's do it a different way because I don't want to turn this error off everywhere. I just want to turn it off for this one file in this situation. So what we're going to do is we're just going to come up to the top of the file, and we're going to add a comment, `eslint-disable`, and that's going to turn it off for this one file. And if we stick it up here, no `param-reassign`, which is the error that we were getting. Alright, there you go. Err is gone. Everything seems to be working. Let's go test this thing.

## Testing PUT



make sure that we're up and running, and everything is compiled, and everything is good. Now if I come down here to this book that we added, all the way down at the bottom, Jon's Book Sequel, and I add that id to the end, and I do another GET, now I see we just have our one book, and that's all fine. We've done that before, but now I want to do a .put. So the first thing we need to do is change our type to PUT and then go to Body tab, change it to raw and JSON. Now we've done all this before, but just in case you skipped ahead and missed it last time, we'll run through it again. Now for a PUT, it should just do a full replace. So if remove everything and hit Send, the new result should be empty, and that's expected behavior. If the new PUT contains empty fields, it should just return those empty fields back, so we're updating and everything is working. If I want to put those items back, maybe with just a little change, let's give this maybe a different title. Now remember, we're not doing anything with the id, so we can really just delete that. And let's hit Send. Now we see that we're actually updating as we expected down here at the bottom. So let's go back and do a GET, Send, and you'll see now we're working. Now this `_v` is something from MongoDB. It's just coming back, and that's a version indicator, and we're not going to worry about that at all in this course. Just ignore that, and you can pretty much just pretend it doesn't exist until you get deeper into MongoDB stuff. So for right now, just pretend it's not there. Now what if I wanted to update the false to true? I don't want to have to send the entire contents of the book in order to update that one thing. I just want to be able to set that to true, delete everything else, and have that update my book to be true. So let's go back over, and let's do a PATCH, and we'll see how that works.

## Middleware

Alright. Now that we've implemented and tested PUT, let's take a quick detour and do a little cleanup in our code by implementing some middleware. Now when we come back over to Code and we want to start to do our .patch like we said we would, the first thing we're going to do in our function is do a `Book.findById`. And we've done that once here, and we did it once in GET, and we're about to do it again. And so the pragmatic programmer in you should be screaming, we're not supposed to repeat ourselves over and over again. And we repeat once, okay. If we repeat twice, we need to stop and do something about that. And the way we're going to do that is by implementing something called middleware, and that's going to inject itself in between the calls and this route. Now to get a quick idea of how middleware works, let's look at this real quick. A client sends a request, and that request is handled by the route in the router, and then that router is going to send a response back to the client. Now when we're using middleware, what's going to happen is we create a function so that when the client







response to the client just like that. So what we need to do now is create this piece of middleware that's going to intercept the request, go and find the book by id, add it to the request, and then forward that on to the route. Now let's look at what that would look like. But first, let's get rid of this patch. We'll deal with that later. Now the way we're going to be doing it, we're going to do middleware by the route. So here, I'm just going to do a `bookRouter.use`, and that's the signal to say hey, I'm about to do some middleware, and I'm going to use that middleware only in the route that has `bookId`. Now that takes a function that takes the normal request response, but also a `next`. Now `next` is a function that the middleware uses to signal that it's done with its processing, and it's ready to pass that request on to the next thing. In this case, when we call `next`, since we only have one piece of middleware, it's going to move on to this `.get` or the `.put`. So if we have more middleware, it would then move on to the next piece of middleware. It just kind of goes to the next thing. To implement this middleware, all we're going to do is take our `.get` function, this `Book.findById`, and we're going to cut that, and then we're going to paste it up in here. So our middleware function is going to do a `findById`, and it's going to find that `bookId`. If there is an error, it's going to return the `err`. Now if the book exists, we need to pass it downstream to the routes. And in this case, it's pretty easy to do that just by adding it to the request. The request is just an object. We can do whatever we want with it. So we can just add that `bookId` to the request. So any changes we make here will be passed downstream. So let's do `req.book = book`, and now `book` is available to everything downstream from here. And then we'll call `next` to sign that we're done. Again, make sure you return `next` the same way you return a `res.send` so that you don't go on and do more stuff. Now if the book isn't found, we're going to return the 404 with a `sendStatus`, so `req.sendStatus 404`. Now once that's done and we have our request that has a `book`, down here in our `.get`, all we really have to do is a `res.json req.book`. Because if there's no book, then it won't ever get there. It'll return a 404 up above, and if there's an error, it'll return that too. So in this case, the `.get` only has to worry about returning the book. So if we get to `.get`, everything is good. We've found a book for the database, and we're ready to return that book back up. So in this case, in our `.get`, we just do a `res.json` and send the book back. Now that becomes so simple. Let's change up our arrow function a little bit to be just a single line. Look at how simple that becomes. Now the `.put` works the same way. So I don't need my `findById`. I don't need my `if err`. All I need to do is pull `book` out of the request, and now everything that's in `req.book` is being modified with whatever is in `req.body`. And then we're going to save that, and then we're going to return it. That's all it takes to make that happen. Now we have a `.get` and we have a `.put` for our router. Now to go back to what we were doing before we got



istracted with this. Let's put in our `.patch`.



Now getting PATCH put together is as simple as doing a `.patch` with the function `req` and `res`. Now remember, our book is coming out of `req.book`, and we only want to update our `req.book` with the items from `req.body` that are there. So if `req.body.title` exists, then we want to update `req.book`. Now you can probably imagine, this will become very painful very quickly, and so we're not going to do it that way at all. That's kind of silly. What we can do instead is use `object.entries` to pull out an array of key-value pairs from `req.body`. And so we can then, it returns an array, so we can run a `forEach` off of that array and only update our book with the items that do exist in that array. So in the `forEach` in this case, we can lay this out explicitly just so you can understand kind of what's happening. The first item is the key, the second item is the value, and then we can shortcut that later if you want to. But for now, let's just leave it written out so it makes sense. So for each key, we'll update it with the value. Now the only caveat to that is we don't want to update the `id`. But first, let's fix this item parenthesis thing so all those red lines go away. So up here at the top, we're going to check to see if `req.body.id` exists, and if it does, we're going to delete it. Because if they accidentally send it, we're just going to get rid of it. Now ESLint doesn't like the `_id`, something about an underscore dangle, and that's weird, but I understand in general why we don't like that. But in this case, we have to use it because that's the way Mongo works, and so I'm going to use this opportunity to show you one last way to suppress errors. We can just suppress this error for only this line, or two lines, using `disable-next-line` and then turn off `no-underscore-dangle`. And let's just do that again down here. There we go. Alright. Now that that's done and we've looped through everything that's been passed in, we need to do a `req.book.save`. And if you look up here at what we did up above, that's not asynchronous. We just did a `save` and then a `return`, and that's not asynchronous coding at all. And we're going to fix that right now. We were just trying to do it fast before. Let's make it right now. If you come back down here, we're going to pass in a callback and do the same if `err` that we've done other places. So if we have an error, we're going to send back the `err` otherwise we're going to send back the book. Then we'll do a `res.json(req.book)` just like that. Now let's copy this, and let's stick it up here, so now both of them match and both of them are doing the right thing. Alright. Now that we've got our `save` going in both places, we've got our PATCH, we've got our PUT, we've got our GET, it looks like we've got most of the verbs covered, let's go ahead and pop back over to Postman, and let's try it all out.

## Testing PATCH





the bottom of the list, you'll see I have a book that's returning authored by John Mills, titled Jon's Book that I have not read. That's odd considering I wrote it, but anyway. Let's fix that. So if I copy this id here and change it over to a PATCH, all I should have to send back is read = true, and it should only update this one item. So if I click Send, you'll actually see I did a PATCH down here at the bottom and I've returned back the whole book, but just with read = true. And just to make sure it actually worked the way we wanted it to, we can go back to the GET, click Send, and there you go. It all worked. Now the other thing we want to try is making sure that I can't send back the id. So if we do a PATCH on this again and I update the id to something different, let's just say sssss, and I send that, notice I did not update the id. If I go back to the GET and hit Send, id is not updated. So that worked out nicely for us. That one for loop was all we needed to get our PATCH done. Alright. We've got one verb left. So let's go implement remove, and then we've got them all working.

## Implementing DELETE

Okay, so now that we have implemented PUT and PATCH, there's only one left to implement, and that's going to be DELETE. So right down here below our patch, we're going to do a .delete. Now this one's going to be really super simple. GET your function, req and res, and in this, we're just going to do a req.book .remove, and that's all it's going to take. We've implemented remove. And so what remove is going to do is it's going to take whatever book was found up in our middleware and remove it. Now just to make sure everything is good, we're going to do our callback. So we're going to pass in a function with err, and if there's an error, we're going to send our err; otherwise, if it works, we don't have a book to send back. That doesn't exist anymore. So we're going to instead do a res.send .status, and we're going to send back a 204, which means removed. And that's all it is to implementing the remove. That's it. So real quick, let's check that out and make sure that worked. Alright. So I have this one, Jon's Book. It's read. Now I just want to remove it. So let me click DELETE, click Send, and notice it's gone. Let's go back over to our GET, hit Send, and you'll notice we got a Not Found back. That's it. That was all it took to implement our remove.

## Summary

So in this module, we finished up our verbs that we're going to use as part of our RESTful API that we're building with Node.js and Express. We've implemented the PUT and PATCH verbs, and we saw that PUT does a full replace of an item, and then we saw that PATCH updates only part of an existing item so that we don't have to send back and forth the entire payload. We





how to remove an item using HTTP DELETE. Now at the end of this module, we've successfully implemented all of the RESTful verbs. So we're done. We've done GET, POST, PUT, PATCH, and DELETE. All those are done, and they're in place and implemented in our code. But we're not quite done yet. Let's take a few minutes to look at automating some tests around our API so we make sure things are working right and then finish up our API because we're not done with REST yet until we've implemented HATEOAS. So let's do some testing, and then let's get into HATEOAS.

# Testing

## Introduction

In this module, we're going to talk about testing our API. Now there are several purposes testing can serve, so we're going to take a couple different approaches to testing so that you can see it from a few different directions. Now we have built out much of a RESTful API with all of the HTTP verbs needed. Now we're not truly RESTful yet since we haven't talked about hypermedia, but we'll get to that in the next module. For right now, we're going to talk about automated testing. And to start with, we're going to write some unit tests with Mocha. Now unit testing is trying to test the smallest piece of code that we can to be sure it works, and our code isn't really set up for that right now, so the first thing we need to do is separate out our code so we have something we can wrap in a test, and so we're going to start by building out controllers for our route. That just gives us something small that we can wrap a test around. Now we will need to deal with all of that unit's dependencies, and so we're going to have to mock those out. Luckily, we have Sinon.js to handle that for us. And after that, we're going to talk about integration tests, or end-to-end tests. As part of that, we'll use something called supertest that's going to allow us to execute our HTTP calls and carry that all the way through to our database, so we can do our POSTs and our GETs and test everything end to end in a testing framework. Alright. Now let's get started by building out some controllers.

## Controllers

Now that we have all the verbs written, but before we can write unit tests, we need to separate out our route handling with controllers to make testing easier. Now if you've been following along, your code should have a file called bookRouter, and this is where we've put all of our





which is an anonymous function, but what we want to as part of building out our controllers for unit testing is we want to take these anonymous functions and stick them out in a controller. So then instead of having this function here, I just say `controller.post`, and what that's going to do is execute whatever function that is, and I'll just take that function code that we just had and drop it in a controller called `booksController`. So let's just cut that and save it for later. The only other piece of code we need up here is we need `const booksController = require`, and we're going to pull this in from a folder called `controllers/booksController`. So let's create that now. New Folder, `controllers`, New File, `booksController.js`. Alright. To get this going, we're going to create a function called `bookController`, and then we're going to do a `module.exports = booksController`. Now in this `booksController`, we need to create a method called `post` because that's what we're going to call over on the other side, and we're going to make that the function that we copied out of our `bookRoutes`. Now notice, we need this `book`, and remember that `book` is in the `Mongoose` model, and that does all of the database work for us. Now I don't want to create a new instance of that inside my controller because I want to be able to mock that later because remember, this module is all about unit testing. So what I want to do is I just want to pass that into my `booksController` when it's first created. So now, I added `Book` to my function call on `booksController`. Now over here in my `bookRouter`, notice this `const booksController = thing`. That's just going to return the function that we just created. What we're going to want to do now is down here, we're going to do `const controller = booksController`, and then we're going to execute it and pass `Book` into it. Let's go ahead and do the GET. So we'll pull this function out and do `controller.get`, come back over to `booksController`, do function `get`, and paste that function. Here, we're using something called the revealing module pattern, and what that means is I have a controller, and I have a series of functions, and now I have to return back the list of functions that I'm going to expose to the outside world. And in this case, we're just going to return two things. We're going to return our POST function, and we're going to return our get function. Now my `bookRoutes` will have access to two functions, the `post` and the `get`. Let's take a quick look, and we can test these two controllers and make sure they're working, and then we'll build some unit tests and see how this all works.

## Postman and Bugs

Okay, we have all of our `bookRoute` code broken out into a controller. So if we've done everything correctly, we should be able to come into Postman and do a `localhost:4000/api/books`, click Send, and we pull back our list of books. So that part works.





title, and we probably should check for that and make sure that we don't allow books being created without a title. So on proper TDD unit testing kind of framework, let's create a test that replicates this issue, and then we'll write code to fix it. So let's break back over and start looking at how to unit test our code using Mocha.

## Testing with Mocha

Now we have all of our `bookRoute` code pulled into a controller and we've identified a bug that we want to fix. So let's start wrapping tests around our controller and see if we can't get this thing figured out. Let's start by creating a new directory called `tests`, and in that, we're going to create a new JavaScript file called `bookControllerTests.js`. This is where we're going to pull all of our test code for our `bookController`. Now before we get started, we need to do an `npm install -D mocha should sinon`. The `-D` makes it a dev dependency. Now Mocha is our testing framework, `should` is an assertion framework that we're going to use, and we're going to use `sinon` for mocking. There they are, 5.2, 13.2, and 7.2. That's what you should be using. Now I will go into what each of these are for kind of as we go, but for now, we have them installed, and that's enough. Once that's done, we're going to come back over here, and we're going to do a `const should = require('should')`. That gives us an instance of `should` that we don't really need right now because `should` attaches itself to an object, so we can just do the `require`. But if you end up wanting to do null checking or other things, you'll need a reference, so we're just going to leave it in here. And then we're going to do `const sinon = require('sinon')`. Now we don't need to reference our Mocha because it's actually going to run inside the Mocha framework. But lastly, we will need our reference to `bookController`, you know, so we can test it. Now Mocha lays out very similar to a standard BDD kind of style with a series of methods. And if you don't know what a BDD style is, this is what it looks like. You start with a `describe` method, and it describes what it is we're testing. So we're going to start with a `describe` `Book Controller Tests` because that's what we're testing. And then it takes a function because, you know, it's JavaScript, and then we actually just nest these calls. So we're in `bookControllerTests`, and now we're going to `describe`, and here, we're going to describe our `Post` tests. And what that means is at this point, we're going to be testing our `POST` method of `bookController`, and now we can lay out our tests inside here. So the test is laid out in plain text with the `it` method. In this case, we're going to do an `it` and then `should not allow an empty title on post`. It makes it very clear what this test is looking for and what this thing should do. One quick thing though is ESLint is not really happy about the `describe` or the `it`. It has no idea what those are. Luckily, we can fix that pretty easily. Let's go over to our ESLint and let ESLint know that we're working in Mocha.







let's take a quick look at the bookController and see what this post method does. Now it's going to take Book, and it's going to create an instance of Book with req.body, and it's going to do save on that book, and it's going to do our res.status. So in order to mock out the things that we need, we need to send a mock book, we need to send a mock request, and a mock response, and we're going to check for a status coming back that says hey, you don't have all that stuff in here that you need. So let's go back over here, and we're going to start writing this thing out. Let's create our Book function. Now one of the nice things here is that JavaScript is not a type-safe language, which means I don't need to have an objects of type book in order to do any of this stuff. I can just create a Book that's just a function, and it takes a book that has something called save and doesn't really do anything. Because if you remember, over here on the bookController, it just calls save. It doesn't do anything with it. It doesn't need save to do anything, so we don't need save to work in this case. So I'm just going to do a const Book = function book this.save = function. Just like that. And now I have a mock book object that I can use in my bookController test. Now unfortunately, for the request and the response, it's not going to be quite that easy because I actually need those to do things. So in this case, we're going to do a const req = and then the request actually has to contain a body that actually has some data in it. And in this case, we're just going to send in an author. Remember, this test is to see whether or not it's throwing an error when there is no title. So we're going to send in an author, no title, and we're going to see what happens. Now let's send just me as the author, and then that's the whole request. The response becomes a little bit more difficult. If we look back over at our controller, we see that I actually have to call some functions, and I want to know what is sent as part of the status, and I want to know what's sent on the response. So here, I'm going to actually mock something out using the Sinon mocking framework. Now in this case, our response needs to have three things in it. It needs to have a status, and it needs to have a send. Now we shouldn't need json because we're not expecting it to be called, but let's add it anyway in case there's something we want to test on it. Those are the three functions that we need. Now in order to mock this out, we're going to use something called a spy. So we're going to do sinon.spy and send and json are also going to be assigned on .spy just like that. And what this is doing is creating a spy function using the Sinon framework that's going to keep track of what's called, what it's called with, how many times it's called, and all that stuff. So now I can actually check to see if status is called and then what it's called with. So now to do our checking down here, we're going to do a res.status .calledWith. Now we're going to want this to be called with a 400, which just means a bad request. Now we'll use our should assertion framework to check the results and throw an error if it's not correct, so should



true. That what should does. That's really all it's for is just adding that should there. Now



what `args` is is an array of each time this function is called. So we only care about the first time it's called, and that has the arguments for each it is called. So in this case, we only care about the first argument as well. And so this will give us an error message that says this was called with the Bad Status, and here's what the status was that this was called with. And so that'll be really helpful when we're error handling our unit tests. Now we also want to check on `send`, so we're going to do a `res.send.calledWith`, and we want to make sure that we're sending back a helpful error. So let's say `Title is required`, and that should equal `true`. But we've gotten to the point where we've got our assertions going, we've got our mocks done, but we haven't really done anything yet. So let's hook this all up to our `bookController`, and then we'll just run our tests. The way we're going to do this is by creating an instance of the `bookController` just like we do over in the route. So we're going to do `const controller = bookController` and then pass in our `Book`. Now in order to call this, we just need to do `controller.post` and make sure we send in `req` and `res`. Now that's all we need. This is a full test. We're sending a `req.body` without a title; we're mocking up our book, our `Mongoose` model `book`, so that we can execute on that; we've mocked up our status and our `send` on our response object; and we're executing the `POST` method on our `bookController`. Now let's run this thing. And in order to do that, we need to get it set up in our `package.json`. So let's do that and fix our bug.

## Running Tests

Now that we have our tests written, let's work on executing our tests and fixing our bug. Now in order to get `Mocha` running, we need to add it to our `scripts` section in `package.json`. So right here, we change this, this error message here, we change that to be `mocha tests/**/*.Tests.js`, and that's going to look in our `tests` directory, and it's going to pull everything that ends with `tests`, and it's going to pull that into our `Mocha` framework, and that will get `Mocha` running whenever we run `npm test`. One last thing we're going to have to change before we can run our test is come back over here to our `bookController`, see we're doing a `res.status`, and we're chaining our `.json`. The way we're doing our mocking, that's not going to work, so we're going to need to do a `res`, just separate out these two calls, `res.status`, `res.json`, and we'll save that. Now in order to run these tests, we're going to come over here, and we're going to do an `npm test`, and we're going to hit `Enter`, and we should see a failed test, and you'll get one test failed. And if you come up here and look, it'll say `AssertionError Bad Status 201`. So this is exactly what we were looking for coming out of our `bookController` because nothing was called with a 400 error. So let's flip back over to our `bookController`, and we're going to write some code for this, and then we'll retest it to make sure we've got our test passing. So over in





otherwise we'll do the rest. Now with all of that done, now that we're checking for title, I should be able to come out here, rerun my test, and now you'll see I have a passing test. What we have here in our bookController test is what we call a unit test, so we're just testing the bookController all by itself. We've mocked everything around it, and we know that the bookController is executing the way we want it to execute. Next, we're going to look at some integration tests, which is more of an end-to-end test where we're actually going to replicate the HTTP POST all the way down to the database.

## Integration Tests

Alright. We've got some unit tests done, and we are running our tests, and we've seen some things, so let's now take a different approach to testing. And we're going to write some integration tests, and they're going to test our application full blown end to end. now we're going to be using a thing called supertest. So first thing we're going to do is npm install supertest -D to get that thing going. If we look in package.json, when this is all done installing, you'll see right there 3.3 is the version of supertest we're going to be using. Alright. Let's close that out. The first thing we need to do is create a new file for these tests to sit in, so we're going to create a new thing called booksIntegrationTests, so we had our booksControllerTests for our unit tests. Now we're going to do booksIntegrationTests.js for our integration tests. Same thing as we did before, the first thing we need is should, so we're going to do a require should to pull that in and get that added to object, and then we're going to pull in supertest. And we're actually going to call it request because that's going to be what it looks like and so we can do request., and so we're going to pull in supertest right here. Now remember, before, we passed book in because we were going to mock that up here. In our integration tests we're not mocking anything, so we're just going to pull Mongoose in right here, and we'll leverage it just like we were anywhere else. So just const mongoose = require mongoose. One last thing, we need to pull app in, and we haven't done that before, so const app = require.. / app.js. Now the reason we do this is because supertest needs app to go through the motions and execute it. So in order to do what we're doing, we need to go back over to app.js, and we actually need to export app so that we could have access to it over in our integration test. Don't worry, that's not going to break anything else. We're just going to pull that in in our integration tests so we have access to it. One more thing we need to pull in is Book. So just like were over in our route, we just need to pull in the mongoose model for book so that we have access to that book, and we can start to do a couple of things with it. Now this is something for supertest. On the request, so the supertest agent, we're going to have our agent run app, and this is just





Create, Replace, Update, Delete, our basic CRUD tests for our API. We're going to fit those in here, and now it should, so in this case, it should allow a book to be posted and return read and `_id`. So we're going to create a book, and we're going to pass it in, and we're going to have it post and return read and `_id`. Alright. Now it's going to pass into this callback done, and done is a callback we're going to call just when we're done, that's why it's called done, and let supertest know that this test is complete. Alright. First thing we need to do is create the package that we're going to send into our API. So in this case, we're just going to pass in a title, an author, a genre, just those pieces, and then we're expecting it to come back with read and id. Alright. Now that we have that, we do an `agent.post`, and what `agent.post` is going to do is it's actually going to send a post to `/api/books`, and it doesn't know or care how that works. This is more of that Black-box testing. I'm going to send an HTTP request to `/api/books` and see what happens. I don't know if our routes handle it, or the apps handle it, or whatever. Now we are going to do a `.send`, and we're going to send that `bookPost` payload in, and we're going to expect a 200 to come back. Now when we're done, at the end, we get a callback that's got 2 things, typical Node style, an error and the results. And here, we get to validate that the results do what we want it to do. So in this case, `results.body` `.should` `.have` `.property`, so it should have an `_id` property. And just because I like to have a failing test first, let's actually create a failing test, so `results.body` `.read`. We really expect it to be false. So in this case, let's just say it should not equal false. That way the test will fail. Excellent. Now when we're done with that, we're going to do `done`, and then that's going to let supertest know and Mocha know that this thing is over. Alright. Now after each test, now I'm adding data to the database. This is actually opening up Mocha. It's adding things to the database. So we need to clean up after ourselves. So after each test, I'm just going to go in and I'm going to clean out my database again. So `Book.deleteMany`, pass in an empty query just to delete everything, and execute that. Now if I wanted to test a GET, I could actually add a bunch of stuff to the database, run my GET, pull it all out, and then clean it up afterwards. I can manipulate the database and then just try everything. Okay, once I call `done`, I'm done. My tests are there. Now very importantly, I'm dealing with the database, and that is not something we want to do on our production or even our development database. So let's come up here, and let's create a new environmental variable called `test`. And what we're going to do in our app is we're going to say hey, if I'm in the test environment, I'm going to run this against a different database. I just want to do something else. So let's go over to `app`, and let's write that code. Let's come down here. And just to be super careful, let's just comment this line out. Let's just take it out completely so that we're not going to, as we're writing our tests, we're not going to mess anything up just in case. Alright. And what we're going to do here is we're going to say if





let's connect to our actual database. And just still, I want to get you in the habit of being cautious, while we're still writing our test because the last thing you want to do, let's make sure and do a console.log test, the last thing you want to do is have a mistake in your test and delete everything. So while you're getting that set up, let's just put some console.logs in here, this is a test, not a test, and then we'll even, for the sake of right this second, let's change our connection string to be -prod, or just something else. Alright. Now you know the database you have is not going to get messed up while you're trying to figure the rest of this out. Alright. Let's go down and actually run this. So let's do an npm t. Oh look, it's passing, and we were expecting it to fail, so something is wrong. And I like having this here because I'm going to show you one other thing as we kind of troubleshoot through. I messed my false up, but this'll give us an opportunity to show you how to do that. So there's our This is a test, so we know that works. We're in the test API. But if we come back over to our integration tests, what we can do is we can come down here, and let's just real quick do a console.log results. And this way, you'll get to see the results that are coming back after our POST. Alright. Let's run this one more time, and now you see all of that stuff that came back if you scroll all the way through, and there's our body right there. And notice, read false. So that worked, but look, I've got tick marks around it. So you can dig through all of the results that come back, and you can see what you get to try and what you can measure inside the results. So let's fix our false. And notice, I've got a Ctrl+C to get out of this, and that's not right. We'll fix that here in a second. Alright. Run our test. There you go. We've got a failing test. Excellent. So let's comment that back out, come back down, and let's also comment out this console.log. Alright. And then, again, notice, I've got to Ctrl+C. Run our test one more time. Okay. We're all good. Let's fix this Ctrl+C thing. I don't like that at all. We've got to get that one figured out. And the reason why it's doing it is there's two things that are still open. There is our Mongoose connection that's still open, and our app is still listening on port 4000 or whatever port we've got it going on, and both of those things need to close at the end of supertest. And so there's two things we need to do in order to get that fixed. So the first thing we're going to do is just say hey, after. And unlike after each, after each gets a run after each test, in this case, after is when everything is all done, and we're going to say hey, close our Mongoose connection, mongoose.connection .close, and we're just going to get rid of that, and then call done. Now that's going to get our Mongoose connection closed, and so we'll Ctrl+C, We'll run this again. Notice, I'm still hanging. So the other thing we need to do is actually close our application and actually get Express to hang up, and that's a little bit more difficult, not difficult. There's a couple things we need to do. The first thing we need to do is actually give us a mechanism to





with. So here in my after, let's do `app.server.close` and then pass this done callback into it. Alright. Now notice, when I ran my test, everything ended. Alright. We've got our integration tests running. That's awesome. Got all of our tests running. So we've done unit tests. We've done integration tests. Let's go summarize really quick, and then let's deal with hypermedia.

## Summary

Alright. We are pretty far down the path of building out a RESTful API with Node.js and Express. And in this module, we started talking about unit testing and integration testing, just automated testing for our application, and we built our test using something called Mocha, which is a test execution framework that just runs all of our tests with us. As part of that, we set our code up to be a little bit more testable, so we pulled things out into controllers, all of our route code, we pulled out into controllers that we were able to mock everything around it and just execute the controller code to make sure that was working. We did all of that mocking with something called Sinon.js that adds some functions and some methods onto our functions that let us check and see what happens inside those functions when those functions are called. Then we ended with integration tests, so full-blown end-to-end testing with something called supertest, and what supertest does is it just looks at the app holistically and then will just send things into the different routes to make sure they worked. And so we did a full-blown end-to-end POST where we tested to make sure our POST was working and we were getting our defaults back appropriately. Alright. Now that all of that's done, we've got our tests, let's close this course out with one more module, and we're going to talk about hypermedia.

# HATEOAS

## Introduction

Alright. We're going to close this course out with a conversation about HATEOAS, or Hypermedia As The Engine Of Application State. Now we've been working through this process of building out a RESTful API with Node.js and Express, we've built all the verbs, we've got a Book API going, and we spent the last module talking about testing and integration tests versus unit tests and all how that stuff works. There's one more constraint that I want to talk about in this last module, and that's HATEOAS and how using hypermedia in our application helps build this self-documenting API that becomes very easy for someone to navigate and








examples, and I'll walk you through what this all means.

## Navigating Your API

Alright. We've got our API built. Let's take just a minute, and let's look at this thing that we've built and see if we can't figure out how to get around inside it. Now here's our API that we've just built. It's `/api/books`, and it returns back a list of books that we have in our MongoDB database. And if you're having problems right now, remember, to fix your Mongo connection string that we changed in the last module if you haven't done that already. Just make sure you go do that. So this returns back a list of books that are in our MongoDB database, and you've seen me do this over the course of the entire course. When I want an individual item, we just take this id right here, we copy it, and we paste right there, and we pull back just a single book. And I know that because I wrote the API, and you know that because I told you that. But all of the other customers of this API, if we were to stick this API out on the internet, wouldn't necessarily intuitively understand that that's the right way to get to a single book because it could be `/title`. It could be `/isbn` number. It could be `/something` else. There's no real way to intuitively just know how to get to an individual book when you're coming from a list of books. And so what HATEOAS, or this hypermedia thing, attempts to do or one of the problems it's trying to solve is to give your API away to self-document itself so that a consumer of your API knows exactly what to do to get to this individual item or what other options are available. So like, how do I get to the author of this book? Or what if I want all of the genres? Or just tell me what to do to get to some other information in this API. And so let's pop over to our code, and we're going to start writing some of these hypertext links into our API so you can see what that looks like.

## Adding Hypermedia

Alright. Now that we understand what it is we are trying to accomplish, let's take a minute and actually implement this concept of hypermedia into our API. So let's head over to `booksController.js`, and you'll see here we've got our GET method that handles our GET response for all of our list of books. And right now, we're just returning the books back from the database, but instead, what I want to do is I want to take a copy of that's array, and let's manipulate it. Let's add this idea of links into our book. So let's do this, `const`, our `returnBooks`, let's just name it something that this is the books that will return, and we'll just going to do a `map` over our books. What this'll let us do is add some links into each book and then just return  a new array. So `map` takes `book`, and then what we're going to do right here is let's take a



book for our newBook. Excellent. Now let's add to that an object called links, so newBook.links is just an empty array. And we have to start there because we're going to add a link back to self now in links, so newBook.links .self. We can't just do it directly because links doesn't exist. You get an error. So we do it this way. And we're going to make that link the link to the individual book, so http://, and then we need the URI, so req.headers .host gets us the things that we need without the space, yep, /api/books, and then we're going to append to the end of that the id of our book, so just book.\_id. So what we've done is we've taken this book that we got out of Mongo, and we've added to it a section called links that has a link to the individual item that you can just click on or navigate to programmatically, however you set it up. Now you want to return this new book. And so let's save that. It should auto restart. And actually, see? We get an error here that says we never changed it. So remember about const, I'm changing newBook all over the place, but I'm keeping the same reference, so it says that it should be a const. So let's add const, and then make sure we're returning returnBooks. Now it should restart, and now let's go back, refresh this page, and notice I have links now to each of the individual items. So I can just click on that, and it's going to take me to the individual item. Excellent. Now what we're going to add here though is remember when we added the ability to filter by genre? We only let you filter by genre. We don't let you filter by anything else. And that's not evident, and so let's add something to our API here that says hey, well, you can filter by genre. So let's go into our bookRouter. Remember, we didn't pull all of the bookRouter stuff for the individual books into the controller. We left it here. So let's just leave it here too, and let's go down into the get. Now we shortened it up to make it all one line. Now we've got to bring it back down again to make it multiple lines because we're going to add basically the same idea here that we added over on the other side. Alright. So const returnBook = book.toJSON. Same idea that we had over on the other side we've got over here as well. And then we're going to do the same thing We're just going to say returnBook.links = an empty object and then returnBook.links., and let's do FilterByThisGenre. Just something that identifies that identifies what it is that we're doing here. So give me all the other books by this genre, and then same thing, http:// req.headers .host /api/books/? genre=, and we want to pass in, let's do req.book .genre, and then we want to make sure we res.json the returnBook. Yep. We'll save that, flip back over to our API, and refresh. And you'll notice I've got a link, but it doesn't actually like work, and that's a problem. If we scroll down to, let's say, The Dark Word, and we click on that, that one works. Like yay! I can click on that, I get all the other fantasy books, but it didn't work up above. And the reason it didn't work is because I've got a space in Historical Fiction. And so let's fix that, and I'll just give you a quick little fix for that. Basically, what we





means. It's just, it's this list of links available to us that helps us navigate the API. Alright. Let's summarize this real quick.

## Summary

Alright. So that concludes our course on building out a RESTful API with Node.js and Express. It's been a long time, and we've worked through a lot of stuff, and good job sticking with us to get all the way to this point. So we started out by talking about what REST is and that it's just basically a set of rules about how an API should behave so that anybody who consumes your API understands basically what's going on at any point. We talked about all the REST verbs. We talked about GET, and PUT, and PATCH, and POST, and DELETE; and we made all of those work; and we implemented them in Node.js and Express. We built unit and integration tests with Mocha and integration using supertest, and we tested some things, and we tested holistically, and we broke things out into controllers, and we made our code testable so that we could understand what was going on and fix things easily. We talked about what hypermedia was and how to use hypermedia to make our API somewhat self-documenting, and the more you pour into that piece, the easier your API is to use for someone who doesn't necessarily understand what's going on with your API. Well, I hope you've enjoyed this course. I've enjoyed making it for you to use, and this is a great starting point to get an API up and running using Node. If you want to dig in a little bit more into Node, I have another course, Building Web Applications with Node.js and Express, that's a bigger, more holistic Node.js /Express course. If you haven't done that one yet, you might do that one next.

# Course Overview

## Course Overview

Hi everyone, my name is Jonathan Mills, and welcome to my course, The RESTful Web Services with Node.js and Express. I'm a technical adviser at World Wide Technology, and one of the things we've found is that most modern backend services are now being developed as simple APIs to support more robust front-end applications. Whether they're web, or mobile, or desktop, they all feed back to a single API. So in this course, we're going to learn how to develop those simple and easy-to-use APIs with Node.js and Express. Some of the major topics that we're going to cover include a quick conversation about what REST really is; then we're





going to talk about API navigation and how to make your API more easy to navigate. Now by the end of this course, you'll have all the tools you need to build a clean and effective API. Now before beginning this course, you should be familiar with JavaScript, but you do not necessarily have to know Node.js. I'm going to teach you everything you need to know about Node to get this done. So I hope you'll join me on this journey to learn Node.js and Express with the course RESTful Web Services with Node.js and Express.

#### Course author



Jonathan Mills

Jonathan is a Pluralsight Author, Technology Advisor, and Business Leader. As a member of the Chief Digital Advisory team at World Wide Technology, Jonathan is able to leverage his unique...

#### Course info

Level Intermediate

Rating ★★★★★ (39)

My rating ★★★★★

Duration 2h 2m

Released 24 Jan 2019

#### Share course



