Node Express Web API

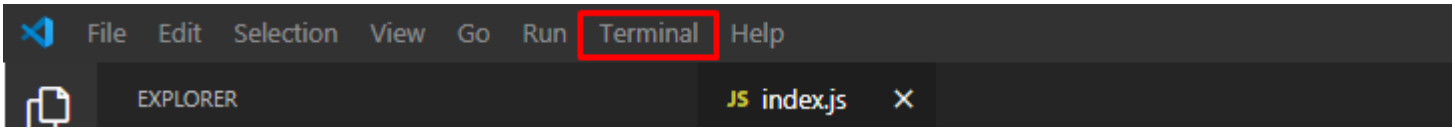
Implementing a Web API with Node, Express, MongoDB and Mongoose

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy. Express is a framework that acts as a light layer atop the Node.js web server making it easier to develop Node.js web applications. Throughout this course, we will see how it simplifies the APIs of Node.js, adds helpful features, helps organizes our application's functionality with middleware and routing, and adds helpful utilities to Node.js's HTTP objects.

Creating the Project

Create a new folder where you want the API project to be stored. Feel free to name this folder anything you like. It is best to name the folder relative to the title of the project you are working on. Once the folder is created, open Visual Studio Code at the location of the folder. Create an empty file named *index.js*.

Next, in the top bar of Visual Studio Code, select Terminal > New Terminal.

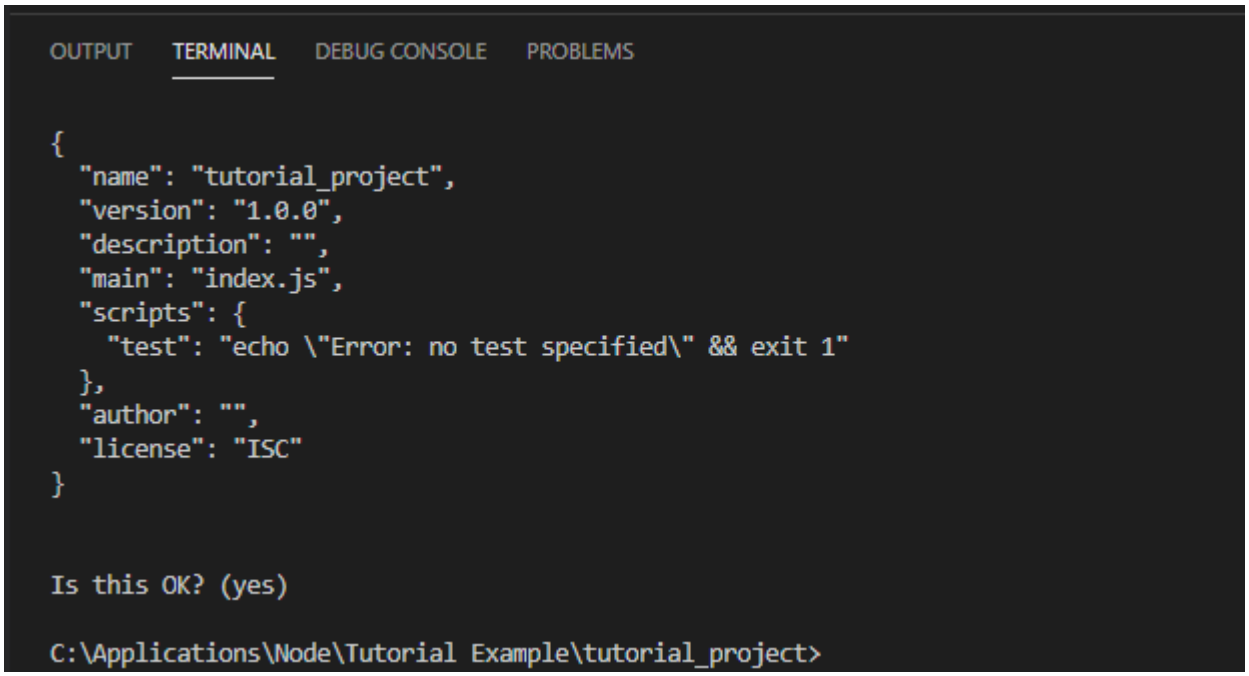


You will see a box appear at the bottom of your screen. This terminal box is where we will need to run a few commands to initialize our project. We will need to run the command 'npm init' to generate a file called 'package.json'. The package.json contains metadata about our Node project like the name of the project, its version and its authors.

In the terminal at the bottom type

```
npm init
```

You will be prompted to provide values for a series of metadata. You can continue to hit enter until the package.json appears in your project.



It is important that we generate this package.json before we attempt to install any packages into our node application.

MongoDB and Mongoose

MongoDB is a highly popular document-based database often used within Node.js and Express.js applications. MongoDB is a non-relational or no-SQL database. MongoDB is fast, scalable, and easy to understand. MongoDB stores data in objects called 'documents' and a grouping of these documents is called a 'collection'. The entirety of all collections is called a 'cluster'. Throughout this section we will be covering the following:

1. Connecting to a MondoDB cluster & Using Mongoose
2. Introducing Configuration Files
3. Creating a Mongoose schema
4. Creating a Mongoose model

Connecting to a MongoDB Cluster & Using Mongoose

Before we connect our Node.js and Express.js application to a MongoDB cluster, we must create a cluster on www.mongodb.com.

Please follow along with the 'MongoDBAtlasSetupTutorial' PDF file.

Now that we have obtained the connection string for our MongoDB cluster, we are ready to set up Mongoose and connect to our cluster within our application.

Mongoose is a node package that gives us simple API interfaces to work with a MongoDB database. Mongoose is what is known as an ODM or Object Document Mapper. The packages was created as a JavaScript layer on top of MongoDB to add extra and pre-built functionality for integration with MongoDB. It makes it easier for developers to integrate MongoDB with Node.js applications.

First, we need to install the mongoose npm package into our application. Within the terminal run this line:

```
npm i mongoose
```

NOTICE: 'i' is a shortcut for 'install'

Now that we have mongoose installed, we are going to use mongoose to connect with our MongoDB cluster. In the 'index.js' file add the following lines:

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

mongoose
  .connect('mongodb+srv://<username>:<password>@<clustername>.6si6q.mongodb.net/
  <dbname>?retryWrites=true&w=majority',
  { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB...'))
  .catch((err) => console.log(`Could not connect to MongoDB. ERROR: ${err}`));
```

At the top we import mongoose and store it in a constant named 'mongoose'. We then call the connect method on the mongoose object and pass in two parameter values. First, we pass in our MongoDB connection string that we received from the previous steps. Second, we pass in an object and initialize our connection with two settings.

useNewUrlParser – This is used by mongoose to successfully parse the connection string that we give it as the first argument to the connect function.

useUnifiedTopology – This setting is used for best performance during the MongoDB connection phase. It is recommended to consistently set this setting to true whenever connecting to a MongoDB cluster.

Finally, we finish this connect function call by chaining the '.then' and '.catch' functions. Because the connect function is asynchronous, we must handle the promise appropriately. In the case the promise responds successfully, we console log a message to notify us that we have successfully connected to our MongoDB. If the promise fails, we then console log that we could not connect and display the error.

At this point, if you have correctly replaced all required values in your connection string, you should now be able to run 'node index.js' and see the following:

```
C:\devCodeCamp\MERN\mongoose-tutorial>node index.js
Connected to MongoDB...
```

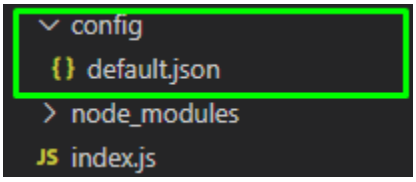
Custom Configuration Files

Currently we are storing the MongoDB configuration settings (connection string) directly on our 'index.js' file. In a real-world application, we never want to store this connection string directly in our code. Instead we are going to store personal and sensitive configuration details in specific JSON configuration files. There are many node packages out there that can help us achieve this goal. In this tutorial we will be using the "config" node package.

Inside of your terminal, install the "config" package with the following command:

```
npm i config
```

Next, we are going to create a folder called "config". Once you have created the folder add a file called "default.json".



Here we can have a JSON object to define the default configuration settings. Lets add a JSON object with one property named "mongoURI". We are going to set the value of this property to our MongoDB connections string. Your default file should now look like this:

```
{
  "mongoURI": "mongodb+srv://<username>:<password>@<clustername>.6si6q.mongodb.ne
t/<dbname>?retryWrites=true&w=majority"
}
```

NOTICE: there **should not** be a line break in your connections string

Great! Now that our connection string is being stored in this default.json file, we need to use this JSON object within our 'mongoose.connect()' function in our 'index.js' file. Go to the index.js file and change the following highlighted code within your 'connect()' function arguments.

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

mongoose
  .connect(
      config.get('mongoURI'),
      { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB...'))
  .catch((err) => console.log(`Could not connect to MongoDB. ERROR: ${err}`));
```

First, we bring in the config module with the "const config = require('config');" line. We need to do this to have access to the built-in config functionality. Next, rather than storing the connection string directly in the mongoose connect function arguments, we are now storing it in the 'default.json' file and pulling the value that our 'mongoURI' property is set to. "config.get('mongoURI')" knows to search for our default.json file then for a property called 'mongoURI'.

By having the connection string located in a separate file, we can now gitignore the default.json file so we do not push our MongoDB credentials up to GitHub.

Exporting Db Startup Logic

Lastly, before we begin to dive into Mongoose and how to work with it, it is best that we reorganize our code. Currently, we have implemented our MongoDB/Mongoose connection logic directly on the index.js file. While this does work, it is best to keep index.js as clean as possible. Index.js is our main entry point for our application therefore we should only use index.js as a higher-level component for orchestrating the lower-level components. What that means is that we should avoid having any specific logic implemented directly on that file. Instead, each series of specific logic (in this case database specific logic) should be implemented and contained within its own module.

Let us create a new folder in our application called "startup" and create a new file called "db.js". We are going to cut out all our MongoDB/Mongoose connections logic from our index.js file and paste it in the new db.js file. We will need to put that connection logic within its own function.

Your db.js file should now look like:

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

function connectDB() {
  mongoose.connect(
      config.get('mongoURI'),
      { useNewUrlParser: true, useUnifiedTopology: true })
    .then(() => console.log('Connected to MongoDB...'))
    .catch((err) => {
      console.log(`Could not connect to MongoDB. ERROR: ${err}`);
      process.exit(1);
    });
}

module.exports = connectDB;
```

Notice the that at the top of the file we need to import the 'mongoose' and 'config' modules because we use them within our code. Next, we create a function called 'connectDB' and paste our mongoose connection logic within the functions code block. Lastly, you will notice that we have a new line within the catch block's

anonymous function. This “process.exit(1);” will shut down the application in the case we run into any unhandled exceptions.

Now that we have removed the pure connection logic from the index.js file, we need to import our new db.js file in the index.js file and call the ‘connectDB’ function.

Your index.js file should now look like the following:

As you can see our recent previous actions have left the index.js file looking much cleaner and our code more organized.

We are now ready to move forward and begin to interact with our MongoDB cluster through our Node.js application.

Creating a Mongoose Schema

A Mongoose schema is a set of rules to define the shape of our documents (data objects) that will be stored in MongoDB. When we create a schema, we define all the properties we want present in our document collection.

For example, let us say we wanted our MongoDB database to store information about products. That means each ‘document’ in our ‘products’ collection would represent a product. We then need to ask ourselves what key value pairs (properties) do we want to be on our product documents. In this application we will assume a simple product record (document) has the following properties: name, description, category, price, and dateModified.

With these properties in mind, lets create our first mongoose schema. We are going to store all of the schemas we create in a new folder called ‘models’. Create a new folder in our application and call it ‘models’. Within that folder, create a new file called ‘product.js’ and add the following code:

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

const productSchema = new mongoose.Schema({
  name: String,
  description: String,
  category: String,
  price: Number,
  dateModified: Date,
});
```

Lets break this down line by line. First, within our 'product.js' file we have to import the mongoose module at the top. Next, we create a constant called 'productSchema'. We set this equal to a new mongoose schema object. As you can see, the mongoose.Schema takes in an object as an argument. What properties and rules we place in this object defines our schema. In this case, we created a 'name' property and stated that its data type will be a string. We have done the same for 'description' and 'category'. For the property 'price' we have stated that its data type will be a Number and for the property 'dateModified' we have set it to Date.

It is important to remember what data types you set each property to because mongoose will enforce those rules later when we are interacting with our products collection.

This is the most basic way to create a mongoose schema. Before we move on to creating a model with this schema, I want to introduce you to some more complex rules we can apply to our schema. For example, as of right now, none of these properties are required for us to add a new product to our MongoDB. In fact, we could submit a blank document with none of these properties. Replace the code above with the following:

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

const productSchema = new mongoose.Schema({
  name: { type: String, required: true, minlength: 2, maxlength: 255 },
  description: { type: String, required: true },
  category: { type: String, required: true, minlength: 5, maxlength: 50 },
  price: { type: Number, required: true },
  dateModified: { type: Date, default: Date.now },
});
```

We have now enforced more specific rules regarding our product documents and collection. When assigning a single property multiple rules, we use an object.

type

Type represents the data type that the property value will be. MongoDB will use this as the data type for the properties on the product document.

required

When required is set to true on a property, mongoose will not allow us to submit a document to the products collection unless all required properties are present with a value.

minlength

We can use minlength to specify the minimum desired length for a string value.

maxlength

We can use maxlength to specify the maximum desired length for a string value

default

We can use default to give a property a default value if a value for that property is not present at document creation time.

These are only a few of the many rules and functions we can provide to properties within a mongoose schema. Please visit the documentation for further information and functionality:
https://mongoosejs.com/docs/guide.html

Creating a Mongoose Model

As you will remember from the OOP (object orientated programming) JavaScript section of the course, we can create and use classes in JavaScript. Once we create our document collection schema, we must use that schema to create a new custom class. We call this new class a *model* when talking in terms of MongoDB and Mongoose.

We are going to use this model as our "portal" or "doorway" into our MongoDB collection. We are also going to use this model when creating new entries for our MongoDB collections.

Add the following line underneath our schema declaration:

```
const Product = mongoose.model('Product', productSchema);
```

First, we declare a const and name it the singular version of what our collection will hold. Also, notice how we are using pascal casing for the name. Because our new 'Product' const will be a class, we are using the pascal naming convention. Next, we set it equal to the result of the 'mongoose.model()' function call. This 'model()' function takes in two arguments. The first argument is the singular name of the collection that this model is for. MongoDB and Mongoose will automatically name the collection for this document the plural version of this first argument. So, since we named this model 'Product', the collections will be called 'products' within MongoDB.

Next, we pass in our productSchema, this is the rules that the model will follow, and that mongoose will enforce. At this point all we have left to do is export the Product object so we can utilize it within our API route handlers.

Add the following line to the bottom of the product.js file:

```
module.exports = Product;
```

Integrating Mongoose and Express

Now that we have successfully prepared our MongoDB connection, schema, and models, we are ready to create CRUD operation endpoints in our application for the Products collection. At this point you may have noticed that your Atlas MongoDB still does not show our Products collection. Once we make a query attempt to add a new Product, MongoDB will create and prepare this collection/cluster.

Lets begin by installing express into our application. Within the terminal, run:

```
npm i express
```

Next, import the express module into the index.js file. Add the following highlighted lines to index.js:

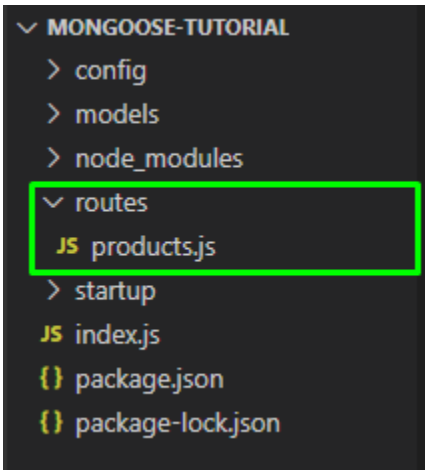
```
const connectDB = require('./startup/db');
const express = require('express');
const app = express();

connectDB();

app.use(express.json());

const port = process.env.PORT || 5000;
app.listen(port, () => {
  console.log(`Server started on port: ${port}`);
});
```

Now our basic express server will run when we run 'node index.js' within our terminal. We will return to this file in a moment.

TIP: this would be a great time to install and implement nodemon. Refer to the previous tutorial for the installation steps.

Now that we have express installed, we can begin to create our CRUD endpoints for the Products collection. In the previous Node.js and Express.js tutorial, we created our endpoints directly on the index.js file. In a real-world application, there will be many different endpoints. For example, we may have endpoints for dealing with products, customers, login, users, accounts, and so on. Rather than creating all these endpoints directly on the index.js file, we should split each endpoint section into its own module.

Create a folder in your application called "routes" and add a file called "products.js":



Inside our products.js file we need to import our Product mongoose model as well as express. Add the following lines to your products.js file:

```
const Product = require('../models/product');
const express = require('express');
const router = express.Router();

// All endpoints and route handlers go here

module.exports = router;
```

On line one we are importing our Product model from the models folder. On line two and three we are importing the express module and then creating a const called 'router'. We then set this const equal to the 'Router()' function present on the express module. You will notice this is different than how we used express when we were creating CRUD endpoints on our index.js file. Whenever you are using a module for a specific URL endpoint, we are going to use the router object we receive from the 'Router()' function present on the express module.

We now need to register this products.js file with our app object that is present on the index.js file. Add the following highlighted lines to our index.js file:

```
…
const products = require('./routes/products');

connectDB();

app.use(express.json());
app.use('/api/products', products);
…
```

First, we are importing our products.js file from the routes folder. This products.js file is going to contain all our route handlers for all product CRUD operations. Next, we register the products.js file with the express applications request response life cycle. Basically, whenever we send a request to the endpoint of "<baseurl>/api/products" it will always pass that request to the products.js file.

POST

To see this in action lets create our first endpoint route handler in our products.js file. We are going to start with the post method so we can add some data to our MongoDB collection. Add the following code in between our import and export lines:

```
router.post('/', async (req, res) => {
  try {

    const product = new Product({
      name: 'Stanley Classic Vacuum Bottle',
      description: `Our Stanley Classic Vacuum Bottle is made with superior
insulation that keeps liquids (soup, coffee, tea) hot or cold drinks cool for up
to 24 hours.`,
      category: 'Travel',
      price: 19.82,
    });

    await product.save();

    return res.send(product);

  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

Here we have created a post route handler endpoint. Inside of this route handler we create a new product object based off the Product class. Inside of the constructor of this Product class we pass in an object that contains all the properties and relative values that we stated would exist on our schema. Notice though, that we do not include a value for 'dateModified'. This is because we gave the 'dateModified' property a default value, meaning, that whenever we create a new product, the 'dateModified' will automatically be set to the current date and time.

Next, after we have created this new Product object, we use the await keyword and call a '.save()' function that is present on the Product object. This is a built in Mongoose function that will attempt to "add" or "save" this new product object to our MongoDB collection. If it is successful, we return the product back to the requestor.

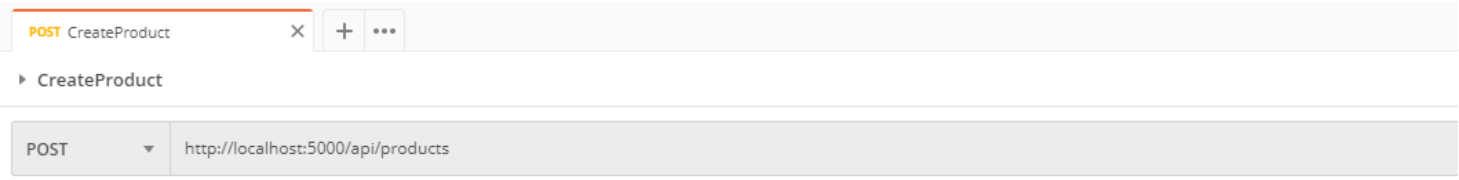
A small change you may have seen here is that we made our route handler function async.

```
router.post('/', async (req, res) => {
```

This is because, when dealing with MongoDB and Mongoose, we need to perform most database actions asynchronously by using the *await* keyword. Following suit with good asynchronous programming, we needed to include a try catch to handled the potential error of our asynchronous 'save()' promise function call. We will

need to include error handling like this for all endpoints in our application when dealing with an external resource like MongoDB.

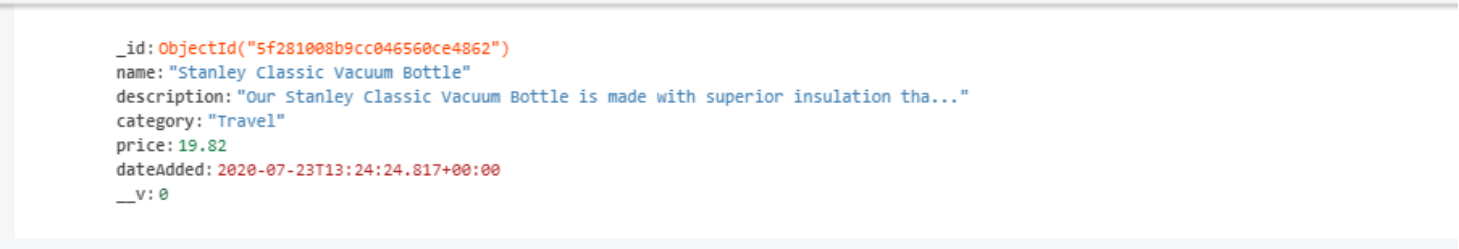
Let's make sure our server is running and test out our endpoint in Postman:



Response from server:


```
{
    "_id": "5f281008b9cc046560ce4862",
    "name": "Stanley Classic Vacuum Bottle",
    "description": "Our Stanley Classic Vacuum Bottle is made with sup
erior insulation that keeps \n      liquids (soup, coffee, tea) hot or
 cold drinks cool for up to 24 hours.",
    "category": "Travel",
    "price": 19.82,
    "dateModified": "2020-07-23T13:24:24.817Z",
    "__v": 0
}
```


Inside of our MongoDB products collection:



Perfect! You will now notice how there are a few extra things here that we did not explicitly add. First, we have a '_id' property on our document. This is called a **primary key** and is auto generated by MongoDB. Each document will have a unique '_id'. This ensures that if two documents contain the exact same data, there will be a way to tell them apart.

Next, notice how the 'dateModified' property auto-populated with a value even though we did explicitly set it! Remember we gave that property a default value in the Mongoose schema. If we would have manually set it, it would have taken the value we provided for it rather than setting the default.

Now, our current post method would continually create the exact same product over and over every time we call it. Rather than hard coding all the values for that product directly in the POST endpoint route handler, we should expect the client to send the information for each product in the request.

Lets change the logic in this method to take values out of the requests body rather than hard coding them.

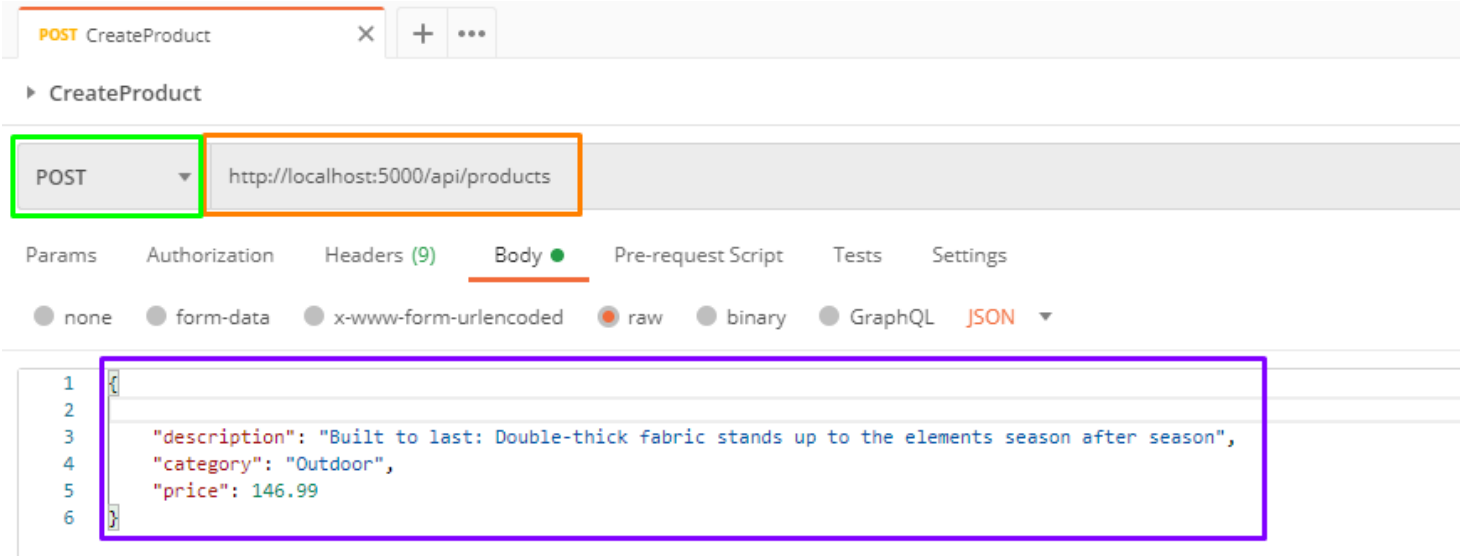
Replace the POST method with the following:

```
router.post('/', async (req, res) => {
  try {

    // Need to validate body before continuing

    const product = new Product({
      name: req.body.name,
      description: req.body.description,
      category: req.body.category,
      price: req.body.price,
    });

    await product.save();

    return res.send(product);
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

Here we are now consuming the body of the request and using the properties located in the request body to populate the new Product object. This way, each time we send a POST request, we can send new data every time and create different Product objects.

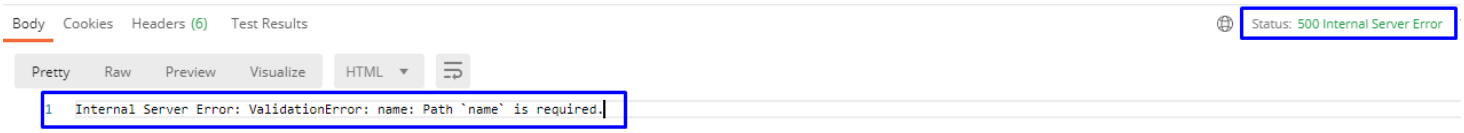
As of right now, we are assuming that the client is sending the proper properties in the requests body. This is something we should never do. We should never assume the client is sending everything in the correct format. Instead, we want to confirm the body is in the correct format before we attempt to create a new Product object to avoid internal server errors. Before we implement this validation, we are going to send a few bad requests to see what happens when we do not validate the body before trying to save to MongoDB.

Here is the request:



- Green – HTTP request type
- Orange – Endpoint URL
- Purple – Data we are posting (sending) to the server

Notice how the 'name' property is not present on the requests body. Here is the response from the server:



Here the server responded with a "ValidatationError" and the status is "500 Internal Server Error". This is because when this line of code executed, "await product.save()", the name property was never set in our Product object. Because the name property was never set, and we set the name property to required on our Mongoose schema, the promise that is returned from the ".save()" method failed and triggered our catch block to return the error.

Although we have handled this error gracefully, we should validate that the body sent by the client is in the proper format before we attempt to save to MongoDB.

To accomplish this, we are going to install another third-party node package called "joi". Run the following command within your terminal:

```
npm i joi
```

Now that we have installed the joi package, lets go to our product.js file in the model's folder and make the following changes and additions:

```
const mongoose = require('mongoose');
const Joi = require('joi');

const productSchema = new mongoose.Schema({
  name: { type: String, required: true, minlength: 2, maxlength: 50 },
  description: { type: String, required: true },
  category: { type: String, required: true, minlength: 5, maxlength: 50 },
  price: { type: Number, required: true },
  dateAdded: { type: Date, default: Date.now },
});

const Product = mongoose.model('Product', productSchema);

function validateProduct(product) {
  const schema = Joi.object({
    name: Joi.string().min(2).max(50).required(),
    description: Joi.string().required(),
    category: Joi.string().min(5).max(50).required(),
    price: Joi.number().required(),
  });
  return schema.validate(product);
}

exports.Product = Product;
exports.validate = validateProduct;
exports.productSchema = productSchema;
```

Joi is a package that allows us to create user defined schemas (rules) that a JSON object must adhere by to pass validation.

Here we are importing the joi module into our product.js file. Next, we create a function that is utilizing the joi package and is responsible for validating a JSON object that we pass to it.

The 'validateProduct' function takes in a parameter of 'product'. Inside of the method we create a new Joi object schema. This schema follows the same principles as a mongoose schema but is setup slightly differently and has no relationship to Mongoose. It is purely responsible for verifying that a JSON object is in the format as defined in this new joi object schema. Inside of the joi object schema, we list out all the properties we expect in the JSON object we pass to it and set a series of rules. For example, for the name property, we are saying:

1. That it needs to be in the data format of string
2. That the minimum length needs to be at least two
3. That the maximum length needs to be less than or equal to 50
4. That the property is required to be present to pass validation

Lastly, in the function we return “schema.validate(product);”. We are passing in the ‘product’ parameter to the new schema objects validate function where it will check that the ‘product’ (JSON object) adheres to all the rules defined in the schema.

If the validation passes, it will not return an error object. If the validation fails, it will return an error object. In our products.js POST route handler, we will call this function and pass in our request body to validate that the user sent a valid request body before we attempt to make the Product object for MongoDB.

At the end of the Product.js file, you will notice that we changed how we are exporting things on this module. Instead of “module.exports = Product”, we are now exporting three elements separately. Since we are now exporting multiple elements from this module, the way we import this module on other files will have to change slightly.

Lets go back to our products.js file in our routes folder and make the following code changes:

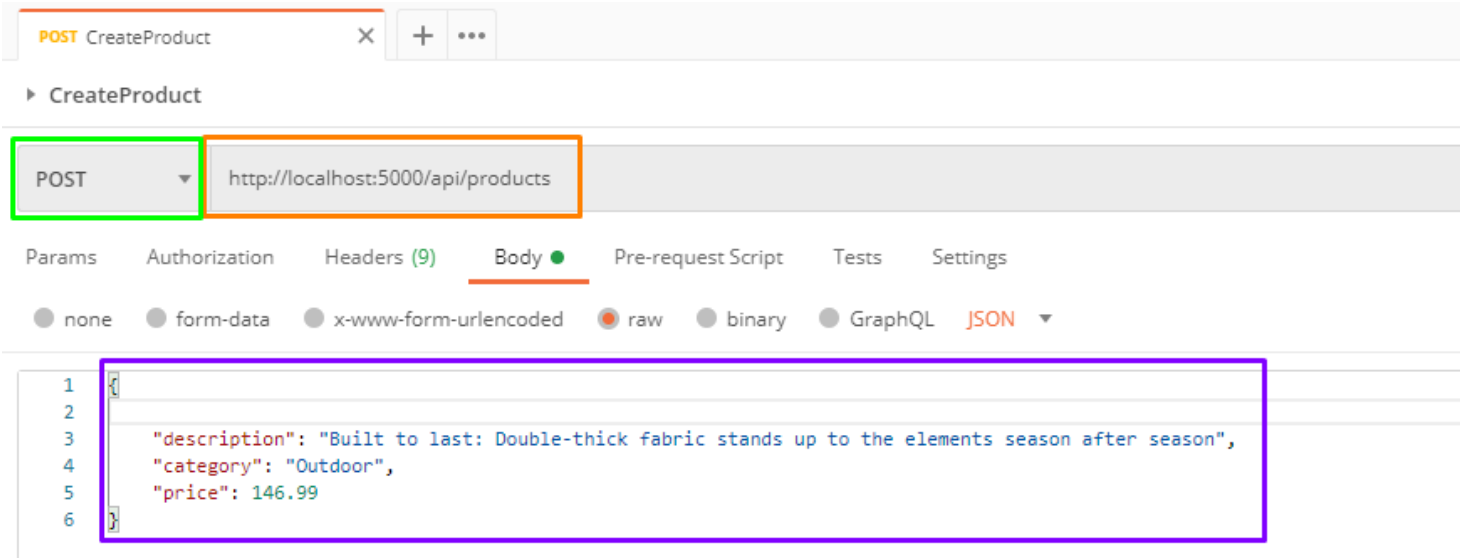
```
const { Product, validate } = require('../models/product');
const express = require('express');
const router = express.Router();

router.post('/', async (req, res) => {
  try {
    const { error } = validate(req.body);
    if (error)
        return res.status(400).send(error);

    const product = new Product({
      name: req.body.name,
      description: req.body.description,
      category: req.body.category,
      price: req.body.price,
    });

    await product.save();

    return res.send(product);
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});

module.exports = router;
```

At the top of the file we are now using object destructuring that makes it possible to unpack values from arrays or properties from objects. Because our product.js file is exporting both the Product model and the validate function, we can specifically request both of those exports directly in our import statement on line one. Notice how we are not referencing our “productSchema” here even though we exported it in the module. We will utilize this productSchema export later in this tutorial.

Next, in our POST method, before we attempt to create and add our new Product object to MongoDB, we first pass in the requests body to our newly created joi validate function. If the function returns an error object (using object destructuring here again) we then return a 400 status (bad request) and send the error to the client.

Now that we have this primary request body validation in place, lets make the same bad request to our post endpoint.

The request:



The response:

Notice now instead of getting a "500 Internal Server Error" error, we receive a 400 status as well as a detailed response about what failed. This is the best approach because even though our Mongoose schema will enforce the rules we applied; we do not want to send a "500 Internal Server Error" for a poorly formatted request body. Rather, we will validate before any database action and return the appropriate status and error in the case the body is not in the proper format. We will be using this validation for both POST and PUT requests.

At this point, test the endpoint out and create a couple products.

GET

Now that we have a working POST method and some data in our products collection, lets implement the GET routes. First, we will create the GET route for retrieving all the products from the products collection.

Add the following code above our POST route handler on the products.js file:

```
router.get('/', async (req, res) => {
  try {
    const products = await Product.find();
    return res.send(products);
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

Here we are now using "router.get" to implement our GET HTTP request method. This route handler is responsible for returning all products present in the products collection.

We create a const called 'products' and set it equal to our Mongoose Product objects 'find()' function result. The 'find()', without arguments, will return all documents in the products collection. We then send those documents back to the client.

Here is the request to the endpoint:

Here is the result of testing the endpoint in Postman:


```
[
    {
        "_id": "5f281008b9cc046560ce4862",
        "name": "Stanley Classic Vacuum Bottle",
        "description": "Our Stanley Classic Vacuum Bottle is made with superior insu-
lation that keeps \n      liquids (soup, cof-
fee, tea) hot or cold drinks cool for up to 24 hours.",
        "category": "Travel",
        "price": 19.82,
        "dateModified": "2020-07-23T13:24:24.817Z",
        "__v": 0
    },
    {
        "_id": "5f2830054e474c1d54e7c33c",
        "name": "Coleman Cabin Tent",
        "description": "Built to last: Double-thick fabric stands up to the ele-
ments season after season",
        "category": "Outdoor",
        "price": 146.99,
        "dateModified": "2020-08-03T15:40:53.278Z",
        "__v": 0
    }
]
```

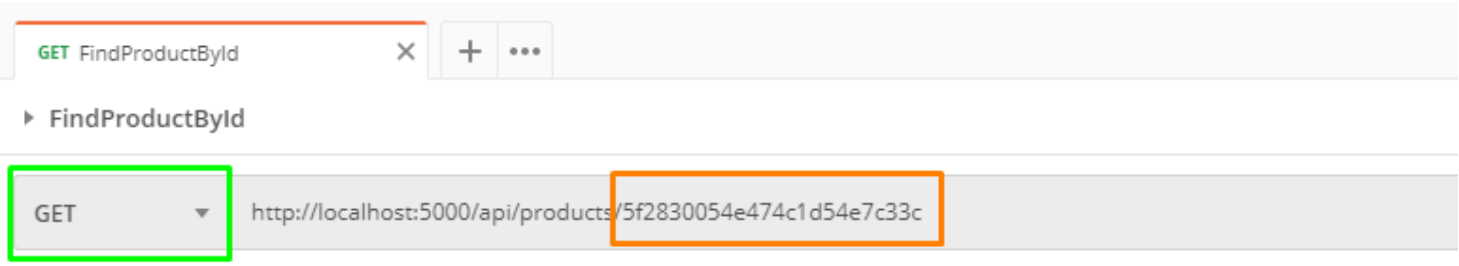

Next, lets create our endpoint for retrieving a single product based off the products id. Add the following code underneath our last GET route:

```
router.get('/:id', async (req, res) => {
  try {

    const product = await Product.findById(req.params.id);

    if (!product)
        return res.status(400).send(`The product with id "${req.params.id}" d
oes not exist.`);

    return res.send(product);

  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

Here we are doing nearly the same as the previous GET endpoint, but now we are expecting an id to be passed into the requestor's URL endpoint. We then take that id and call the "findById()" function that is present on our Mongoose Product object. This function will check the products collection for a product that has the primary key id. If there is no product in the collection with that id, the 'findById()' function will return null.

Next, if the product we receive from the 'findById()' function is null, we return a 400 status and an error letting the client know there is no product in the collection with that id. If the product is not null, we continue by sending the product object back to the client.

Here is the request:



- Green – HTTP request type
- Orange – Valid id of product in request URL

Response from server:


```
{
    "_id": "5f2830054e474c1d54e7c33c",
    "name": "Coleman Cabin Tent",
    "description": "Built to last: Double-
thick fabric stands up to the elements season after season",
    "category": "Outdoor",
    "price": 146.99,
    "dateModified": "2020-08-03T15:40:53.278Z",
    "__v": 0
}
```

PUT

Next, we are going to implement the logic required to update a product in our MongoDB collection. Add the following code underneath our POST endpoint function:

```
router.put('/:id', async (req, res) => {
  try {
    const { error } = validate(req.body);
    if (error) return res.status(400).send(error);

    const product = await Product.findByIdAndUpdate(
      req.params.id,
      {
        name: req.body.name,
        description: req.body.description,
        category: req.body.category,
        price: req.body.price,
      },
      { new: true }
    );

    if (!product)
        return res.status(400).send(`The product with id "${req.params.id}" d
oes not exist.`);

    await product.save();

    return res.send(product);
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

Notice how the PUT endpoint expects an id to be passed to the route handler. We use this id to find which document the client is trying to update. Next, we are once again using our validate function that is present on the product.js module. This is because we need to validate that the body send in the request is in the correct format before trying to update the specific product.

If the body of the request is in the correct format, we continue to attempt the update. In this example, we are using the "update first" approach. That is, we try to find the document to update and update it at the same time. Notice we are calling a function called 'findByIdAndUpdate()'. This function is responsible for finding the correct document and then updating its values. The way we are using this function, it expects three different arguments to be passed into it. The first argument is the id of the document we are trying to update. The next argument is an object containing all the values that are to be updated. Finally, the last argument is an object where we can set a series of settings or "desired" result format. In this third argument we reference the 'new'

flag and set it to true. This means that the result of this 'findByIdAndUpdate()' function should return the newly updated document. If we do not include this, the function would not return the newly updated object but would be a reference to the old (pre-updated) version of the document.

Let's test this in Postman.

The request:

- Green – HTTP request type
- Orange – Endpoint URL
- Purple – Data we are posting (sending) to the server

Response from the server:


```
{
    "_id": "5f2830054e474c1d54e7c33c",
    "name": "Coleman Tent",
    "description": "Built to last: Double-
thick fabric stands up to the elements season after season",
    "category": "Outdoor",
    "price": 124.99,
    "dateModified": "2020-07-23T15:40:53.278Z",
    "__v": 0
}
```


Perfect! Notice how the 'dateModified' has changed. This is because we did not include it as a property in our update and instead used the default value as defined in our Mongoose schema (which is the current date and time). This way we have consistent record of the last time this document has been modified.

DELETE

The last stop on our CRUD train is the DELETE HTTP request endpoint and route handler. Add the following code underneath our PUT handler code:

```
router.delete('/:id', async (req, res) => {
  try {

    const product = await Product.findByIdAndRemove(req.params.id);

    if (!product)
        return res.status(400).send(`The product with id "${req.params.id}" d
oes not exist.`);

    return res.send(product);

  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

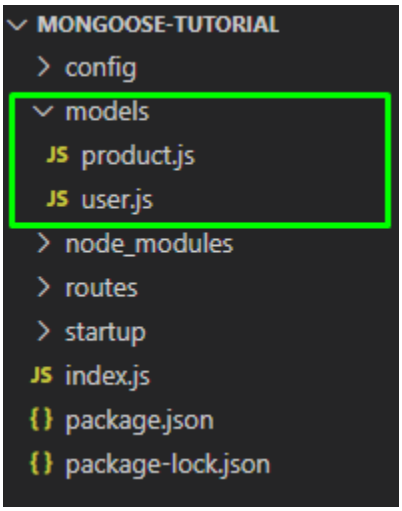
Here the router HTTP request type is now DELETE. We expect the id of the document we are trying to delete to be passed into the route handler via the request parameters (via the URL endpoint). We then take that id and pass it into the 'findByIdAndRemove()' function. The 'findByIdAndRemove()' function requires only one argument which is the id of the document we are trying to delete. If the function successfully removes the document from the collection, it will return that document. If the function does not successfully remove the document from the collection, it will return null. In the case that it returns null we send a response status of 400 and alert the client with a message stating a document with the given id does not exists. If it did successfully remove the document, we let it continue and return the document that was just removed back to the requestor (client).

Embedding Documents within Documents

In most applications, our documents will need to store objects (other documents) within a single document. In this example, we are going to create a User schema and model that has a property called "shoppingCart". This shoppingCart property is going to hold an array of Product objects.

Lets go over how we can achieve this using MongoDB and Mongoose.

First, add a new file to our models folder called 'user.js':



Next, add the following code to the user.js file:

```
const mongoose = require('mongoose');
const Joi = require('joi');
const { productSchema } = require('./product');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  isGoldMember: { type: Boolean, default: false },
  shoppingCart: { type: [productSchema], default: [] },
});

const User = mongoose.model('User', userSchema);

function validateUser(user) {
  const schema = Joi.object({
    name: Joi.string().required(),
  });
  return schema.validate(user);
}

exports.User = User;
exports.validate = validateUser;
```

The only new thing we are seeing here is that the userSchema has a property called 'shoppingCart' that is set to the type of an array of 'productSchema'. This is how we set a property equal to another schema object. This way we can now store Product objects within our User object. The most important thing to understand about embedding documents is how we are going to perform CRUD operations to the embedded documents.

First, here is an example of what the data will look like within our MongoDB collection:

```
_id: ObjectId("5f2847afe0491b249829a827")
isGoldMember : false
name : "John Doe "
shoppingCart : Array
  0 : Object
      _id: ObjectId("5f281008b9cc046560ce4862")
      name : "Stanley Classic Vacuum Bottle "
      description : "Our Stanley Classic Vacuum Bottle is made with superior insulation that keeps
      category : "Travel "
      price : 19.82
      dateModified : 2020-07-23T13:24:24.817+00:00
      __v : 0
  1 : Object
      _id: ObjectId("5f284846abd6bc5ac85d1fe8")
      name : "Coleman Cabin Tent "
      description : "Built to last: Double-thick fabric stands up to the elements season after season
      category : "Outdoor "
      price : 146.99
      dateModified : 2020-07-23T17:24:22.751+00:00
      __v : 0
__v : 2
```

Notice how the shoppingCart property is an array of sub-documents. These sub-product documents are exact copies of the products in our products collection.

IMPORTANT: It is important to note that even though the product objects in the users shopping cart are direct copies of the products in the products collection, they have no "real" relationship to each other. Basically, if we decide to update a product on the products collection, it will not update the same product in a user's shopping cart. Once we add that product to a user's shopping cart, if we update a product in the products collection, we would then need to find all users who have that product in their shopping cart and update that specific product in all user's shopping carts.

Next, we are going to jump into a few examples of how to:

- POST (add) a new product to a user's shopping cart
- PUT (update) a product in a user's shopping cart
- DELETE a product in a user's shopping cart

In the following examples, I have gone ahead and created a new file in our routes folder called "users.js". This file is responsible for handling all request to user CRUD operations. You can look at our products.js file as an example of how to do this on your own. Also, if you are following along, make sure you register the users.js file into the applications pipeline in the index.js file like we did with the products.js.

POST

In order for us to add a new product to a user's shopping cart, we must create a POST method that expects both the user id of the user we are trying to add the product to and the id of the product we are trying to add to the user.

Add the following code to your users.js file:

```
const { User } = require('../models/user');
const { Product, validate } = require('../models/product');
const express = require('express');
const router = express.Router();

router.post('/:userId/shoppingcart/:productId', async (req, res) => {
  try {
    const user = await User.findById(req.params.userId);
    if (!user) return res.status(400).send(`The user with id "${req.params.us
erId}" does not exist.`);

    const product = await Product.findById(req.params.productId);
    if (!product) return res.status(400).send(`The product with id "${req.par
ams.productId}" does not exist.`);

    user.shoppingCart.push(product);

    await user.save();
    return res.send(user.shoppingCart);
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});

module.exports = router;
```

First, we must ensure we have included all the proper imports that are needed for the routes on the file. Notice how we included both the User and Product Mongoose models in our import statements. This is because we will need to query both our users and products collections. Also, in this case I am bringing in the products validate function. We will need this to validate the request body when the client requests to update a product in a user's shopping cart.

Now, in our POST endpoint, the declared URL is:

```
router.post('/:userId/shoppingcart/:productId', async (req, res) => {
```

Here we are specifying that we want both a user id and a product id to be passed to the route handler via the requestor's endpoint URL.

We then take both of the id's and query the respective collection to ensure that those id's are valid and belong to existing documents in their collection. We are doing that here in these lines:

```
    const user = await User.findById(req.params.userId);
    if (!user) return res.status(400).send(`The user with id "${req.params.us
erId}" does not exist.`);

    const product = await Product.findById(req.params.productId);
    if (!product) return res.status(400).send(`The product with id "${req.par
ams.productId}" does not exist.`);
```

Next, if it passes both above if statements, we know that they are valid documents and exist. We then take the product object and push it onto the users shoppingCart array property:

```
user.shoppingCart.push(product);
```

Finally, we save the user object and send the updated user's shoppingCart back to the client:

```
    await user.save();
    return res.send(user.shoppingCart);
```

NOTICE: sending the shopping cart back to the client is a design decision. You could equally choose to send the entire user object back which would include all user information as well as the shopping cart, or, send the newly created product object.

PUT

Next, we are going to create the logic for updating an embedded document. Add the following code underneath our POST route handler in the users.js file:

```
router.put('/:userId/shoppingcart/:productId', async (req, res) => {
  try {
    const { error } = validate(req.body);
    if (error) return res.status(400).send(error);

    const user = await User.findById(req.params.userId);
    if (!user) return res.status(400).send(`The user with id "${req.params.us
erId}" does not exist.`);

    const product = user.shoppingCart.id(req.params.productId);
    if (!product) return res.status(400).send(`The product with id "${req.par
ams.productId}" does not in the users shopping cart.`);

    product.name = req.body.name;
    product.description = req.body.description;
    product.category = req.body.category;
    product.price = req.body.price;
    product.dateModified = Date.now();

    await user.save();
    return res.send(product);
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

Here in the PUT function we call the product.js module validate function. This is because if the client hits this endpoint, they are trying to update the values of a product in a user's shopping cart. That means they are sending information through the body for the request to be used as the new data for the product that is to be updated. Before we begin any database action, we need to confirm that the body of the request has all the correct and required properties for a product object.

Next, if the body of the request is valid, we query the User collection to obtain the user document the client is attempting to update. If the user does not exist, we return an error to the client. If the user does exist, we continue…

Unlike the POST function, we no longer need to query the products collection, but rather we need to query the users shopping cart. We are taking the productId that is passed into the params of the request and calling a '.id()' function on the users shopping cart property:

```
const product = user.shoppingCart.id(req.params.productId);
    if (!product) return res.status(400).send(`The product with id "${req.par
ams.productId}" does not in the users shopping cart.`);
```

This '.id()' function is a built in function used to query a collection of sub-documents. When attempting to find a specific object in an array of embedded documents, you cannot use '.findById()' as you have seen before. Instead, we must use this '.id()' function. This function will return the document with the corresponding id if it exists in the array. If it does not exist in the array, it will return null. In the case that it is null, we send the corresponding error back to the client.

Next, we take each value of the product and assign it equal to its corresponding property in the body of the request:

```
    product.name = req.body.name;
    product.description = req.body.description;
    product.category = req.body.category;
    product.price = req.body.price;
    product.dateModified = Date.now();
```

Because we are updating a sub-document on a parent document, we do not have access to the '.findByIdAndUpdate()' function as we did with our product collection on the products.js route file. Instead, we must set each value one by one.

Finally, we save the parent object and return the updated document back to the client:

```
    await user.save();
    return res.send(product);
```

NOTICE: in this example we are sending the product object back. This is because the request was specifically to update a single product. Because of this, we can deduce that the client is interested in receiving the updated document in return.

DELETE

Lastly, we are going to cover the how to handle the DELETE CRUD operation on an embedded document. Add the following code underneath our PUT route handler:

```
router.delete('/:userId/shoppingcart/:productId', async (req, res) => {
  try {

    const user = await User.findById(req.params.userId);
    if (!user) return res.status(400).send(`The user with id "${req.params.us
erId}" does not exist.`);

    let product = user.shoppingCart.id(req.params.productId);
    if (!product) return res.status(400).send(`The product with id "${req.par
ams.productId}" does not in the users shopping cart.`);

    product = await product.remove();

    await user.save();
    return res.send(product);

  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

This route handler is nearly identical to the PUT route handler. The only difference here is now rather than updating the product we retrieved from the users shopping cart, we now call a '.remove()' function. This will remove the product from the sub-document collection.

```
product = await product.remove();
```

Conclusion

In this tutorial we have covered the introductory concepts to MongoDB and Mongoose within a Node.js and Express.js application.

We have covered:

- Creating a Node.js application
- Integrating MongoDB and Mongoose
- Creating Express API routes consuming a MongoDB cluster
- Performing all CRUD operations for a MongoDB collection
- Specific approaches to embedded documents in a MongoDB collection

To further your knowledge on MongoDB and Mongoose we highly recommend checking out the official documentation.

MongoDB - https://docs.mongodb.com/

Mongoose - https://mongoosejs.com/docs/api.html