Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

Mar 28 · 11 min read

Photo by Aleksi Tappura on Unsplash

Writing a CRUD app with Node.js and MongoDB

Since you are here, I will assume you know the following:

1- A bit of JavaSript. (If not, you can have a look over *the introductory JS guide from Mozilla*)

2- Heard of Node.js and curious about seeing it in action.

. . .

Basics:-

CRUD Operations:

CRUD stands for *Create, Read, Update* and *Delete*. Which are the basic operations that a simple web app would be designed to achieve.

REST:

If you didn't hear about REST before, you can read more about it here.

> *In this tutorial, we will be designing an API for a Products app.*

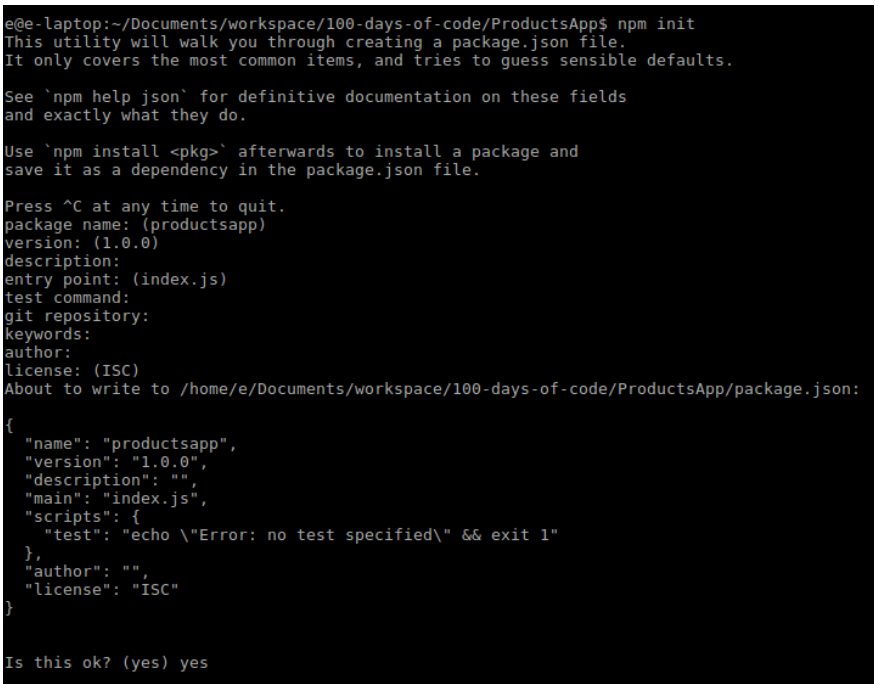
Getting Started:-

1- Install Node.js from the Node.js website

2- I've created a directory called 'ProductsApp'.

3- Inside the newly created directory, execute the following command in the **terminal**

```
npm init
```



```
e@e-laptop:~/Documents/workspace/100-days-of-code/ProductsApp$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (productsapp)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /home/e/Documents/workspace/100-days-of-code/ProductsApp/package.json:

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


Is this ok? (yes) yes
```

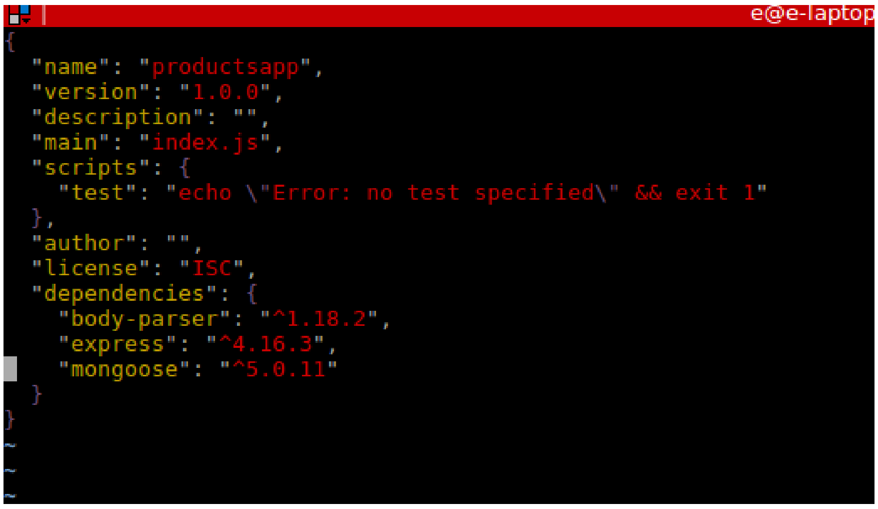
Initialize our Node.js project with default settings

The above commands results in creating a package.json file. The package.json file is used to manage the locally installed npm packages. It also includes the meta data about the project such as name and version number.

Afterwards, we need to install the packages we will be using for our API The packages are:
1- ExpressJS: It's a flexible Node.JS web appplication that has many features for web and mobile applications
2- mongoose: the mongoDB ODM for Node.JS.
3- body-parser: package that can be used to handle JSON requests.

We can install the above mentioned packages via typing the following commands in the **command line**. Just make sure that you are in the project directory before executing the below command.

```
npm install --save express body-parser mongoose
```



```
{
  "name": "productsapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.18.2",
    "express": "^4.16.3",
    "mongoose": "^5.0.11"
  }
}
~
~
~
```

Changes that took place in our package.json file after executing the above command

. . .

Initializing the Server:-

Create a new file, let's name it app.js directly inside the ProductsApp directory

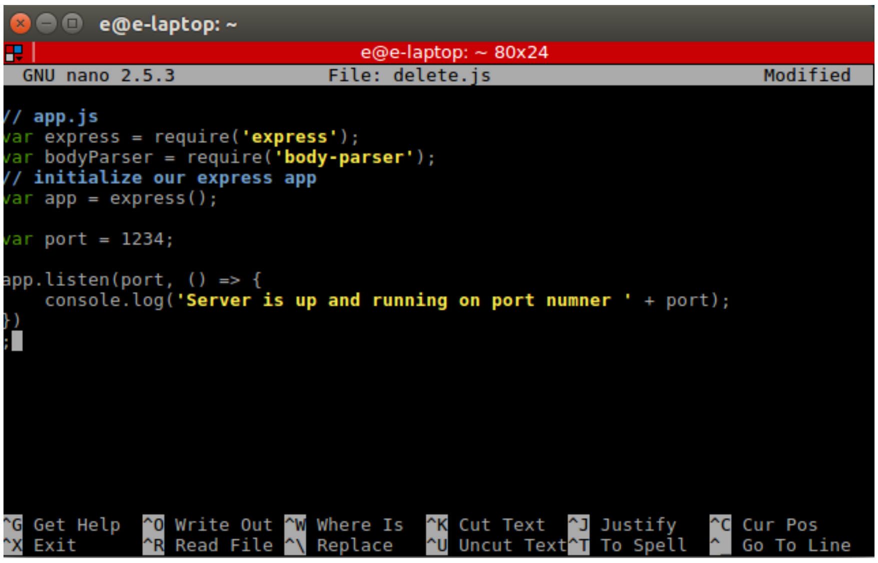
```
touch app.js
```

Open the newly created file named ***app.js*** and require all the dependencies we previously installed (ExpressJS and body-parser) -we will talk about mongoose later-

```
// app.js

const express = require('express');
const bodyParser = require('body-parser');

// initialize our express app
const app = express();
```

Next step would be dedicating a port number and telling our express app to listen to that port.

```
let port = 1234;

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



e@e-laptop: ~
e@e-laptop: ~ 80x24
GNU nano 2.5.3 File: delete.js Modified

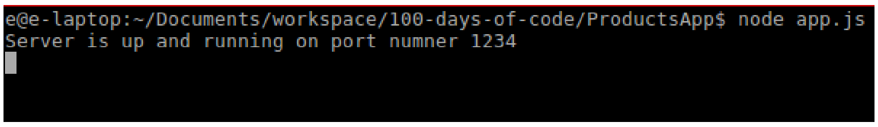
```
// app.js
var express = require('express');
var bodyParser = require('body-parser');
// initialize our express app
var app = express();

var port = 1234;

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

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

How our app.js looks so far…

Now, we should be able to test our server using the following command in the ***terminal***

```
node app.js
```



```
e@e-laptop:~/Documents/workspace/100-days-of-code/ProductsApp$ node app.js
Server is up and running on port numner 1234
```

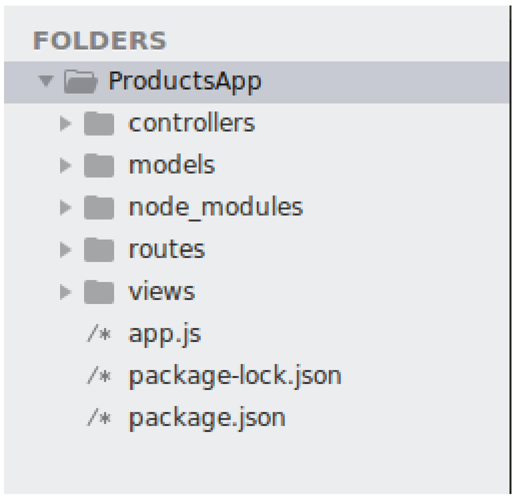
Message that appears when running the server

Now we made sure that we are having a server that is up and running. However, this server does nothing! Let's work on that and make our app more complex.

Organizing our application:

We will be working with a design pattern called ***MVC***. Its a neat way of separating parts of our app and grouping them based on their functionality and role. ***M*** stands for models, this will include all the code for our database models (which in this case will be Products). Then comes the ***V*** which stands for the views or the layout. We will not cover the views in this tutorial as we are designing an API. The remaining part now is the ***C***, which stands for controllers which is the logic of how the app handles the incoming requests and outgoing responses. There will be one more thing, called ***Routes***, ***Routes*** are our guide, they tell the client (browser/mobile app) to go to which Controller once a specific url/path is requested.

Inside the ProductsApp directory, I will create the following four subdirectories
1- controllers
2- models
3- routes
4- views



App structure

Now we have a server that is ready to handle our requests and some directories that would have our code.

Let's start by defining our model. Create a new file in the ***models*** directory and let's name it ***product.model.js***

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

let ProductSchema = new Schema({
    name: {type: String, required: true, max: 100},
    price: {type: Number, required: true},
});


// Export the model
module.exports = mongoose.model('Product', ProductSchema);
```

First we started with requiring mongoose and then we define the schema for our model. Last thing is exporting the model so it can be used by other files in our project.

Now we are done with the ***M*** part

Routes:

Let's start imagining how the urls will be like. Let's desigin our routes.

Inside the routes directory, create a ***product.route.js*** file. This is the file that will include the routes of the products. Some developers prefer to have all the routes in a single file (routes.js) for example but this is not helpful when your app grows! so let's structure it the right way from the beginning.

```
const express = require('express');
const router = express.Router();

// Require the controllers WHICH WE DID NOT CREATE YET!!
const product_controller =
require('../controllers/product.controller');


// a simple test url to check that all of our files are
communicating correctly.
router.get('/test', product_controller.test);

module.exports = router;
```

Controllers:

Next step is to implement the controllers we referenced them in the routes

go to our controllers directory and create a new js file named ***product.controller.js*** which will be the placeholder for our controllers.

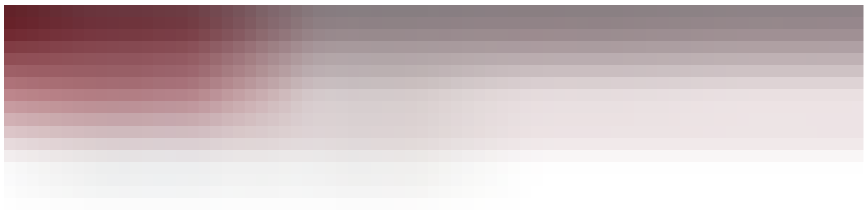
```
const Product = require('../models/product.model');

//Simple version, without validation or sanitation
exports.test = function (req, res) {
    res.send('Greetings from the Test controller!');
};
```

Last step before trying out our first route is to add the route class to the ***app.js***

```
//app.js

const express = require('express');
const bodyParser = require('body-parser');

const product = require('./routes/product.route'); //
Imports routes for the products
const app = express();

app.use('/products', product);
```

```
let port = 1234;

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

Now head to your browser and try the following link: http://localhost:1234/products/test



Validating that our test route is working…

Now we have our very first route working. Let's get the rest working….

. . .

Postman:

Postman is a very powerful HTTP client that is used for testing, documenting and the development of APIs. We will be using Postman here to test our endpoints that we will be implementing through out the rest of the tutorial. But first, let's get familiar with Postman using our '/test' route.

1- Install Postman from their website.

2- Open the app, make sure it's a GET request and type the following url '*localhost:1234/products/test*'. Just make sure that your server is still running on the port number *1234*. You should be able to see 'Greetings from Test controller' when going on the 'Preview' mode in Postman.

Trying test route on Postman

. . .

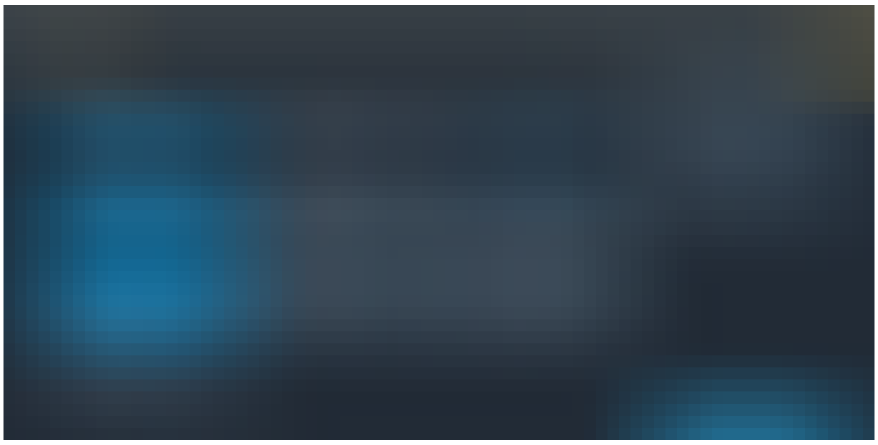
The Database:

Our database will be hosted remotely on mLab. mLab offers a nice free tier that we can use to test our application. Let's set it up quickly…

1- Head to mLab's website.

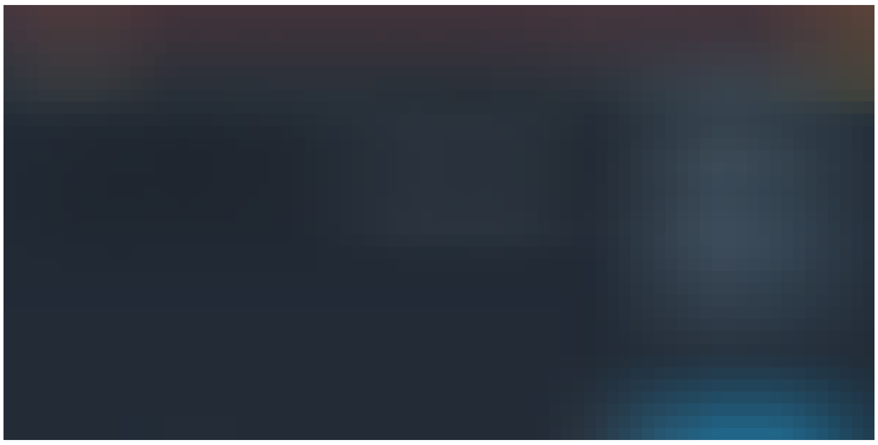
https://mlab.com/

2- Click on 'Create New' from the above image.



Screen on creating a new image

3- Select the Sandbox Plan Type and click on 'Continue'.

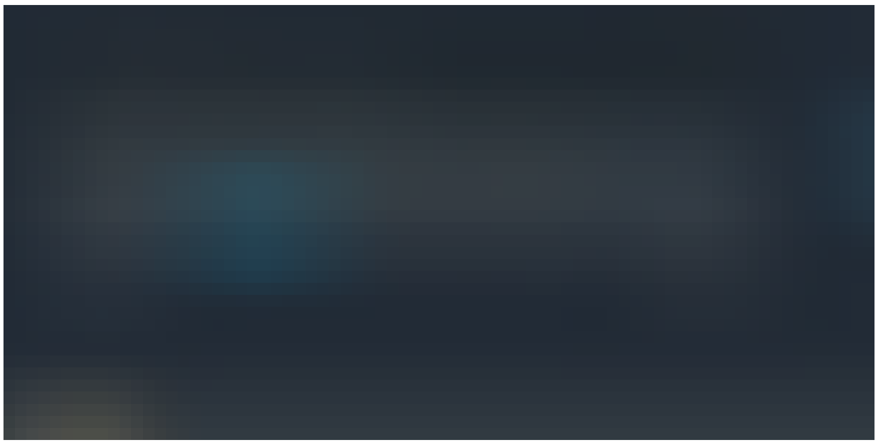


Naming our database

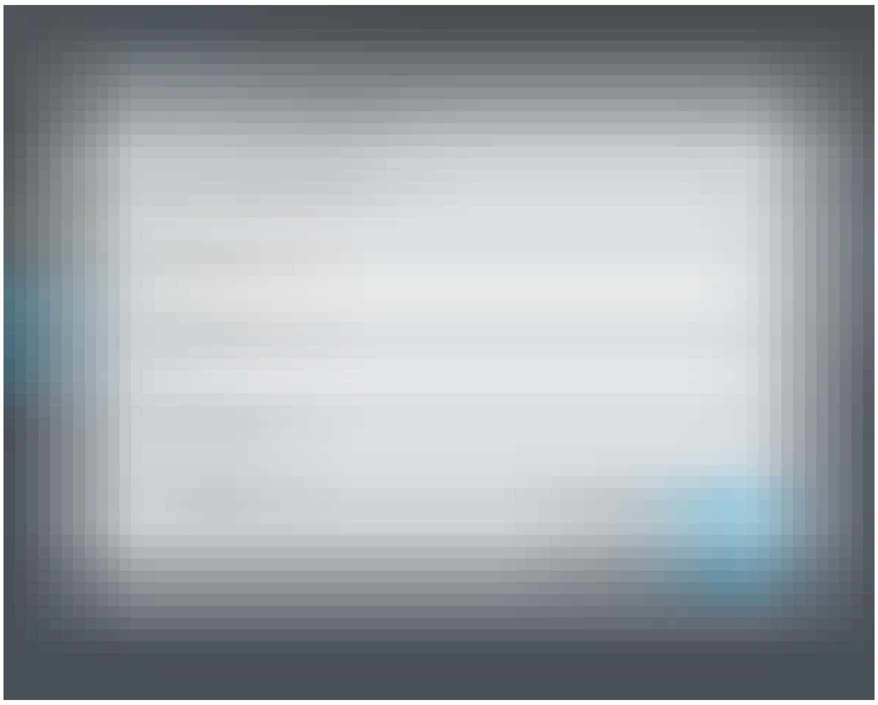
4- Type in the database name. I am using 'productstutorial' as the database name for this example.

Submit order screen.

5- Once everything is ready, just click on 'Submit Order'


Viewing our created database

6- Next step would be creating a user to be able to access the database. Simply click on 'Add database user'


create a new user for accessing the database

7- Last step would be entering the data from the database user you are creating. In this tutorial, for the username I will be using 'someuser' and for the password I will be using 'abcd1234'.

Now we have a database in the cloud that is ready to be accessed :-)

. . .

Connecting our app to the remote Database:

We need to inform our app that it should be communicating with the database we have just created on mLab.

Remember the '***mongoose***' package we installed before? Now is the right time to use.

All we have to go is heading to our app.js file and paste the following code in it. Just remember to update the *dev_db_url* variable with the connection string of your remote database on mLab. Remote database string consists of your database username and password, separated by a ':' and then the URL to your database instance on mLab and then the database name.

```
// Set up mongoose connection
const mongoose = require('mongoose');
let dev_db_url =
'mongodb://someuser:abcd1234@ds123619.mlab.com:23619/product
stutorial';
let mongoDB = process.env.MONGODB_URI || dev_db_url;
mongoose.connect(mongoDB);
mongoose.Promise = global.Promise;
let db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB
connection error:'));
```

. . .

Body Parser

Last configuration thingy we need for our app.js is using bodyParser. Body Parser is an npm package that is used to parse the incoming request bodies in a middleware.

In you ***app.js*** file, add the following couple of lines.

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));
```

Here is how our full ***app.js*** file looks like

```
// app.js

const express = require('express');
const bodyParser = require('body-parser');

const product = require('./routes/product.route'); //
Imports routes for the products
const app = express();

// Set up mongoose connection
const mongoose = require('mongoose');
let dev_db_url =
'mongodb://someuser:abcd1234@ds123619.mlab.com:23619/product
stutorial';
const mongoDB = process.env.MONGODB_URI || dev_db_url;
mongoose.connect(mongoDB);
mongoose.Promise = global.Promise;
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB
connection error:'));

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));
app.use('/products', product);

let port = 1234;

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

By now our ***app.js*** file is finalized and you are aware of the usage of each and every line of code in the file.

. . .

Implementing the endpoints

CREATE

The first task in our CRUD task is to create a new product. Let's start by defining our route first. Head to routes and start designing the expected path that the browser would hit and the controller that would be responsible for handling that request.

```
// routes/products.route.js

...
router.post('/create', product_controller.product_create);
```

Now let's write the *product_create* controller in our controller file. Head to *controllers/product.controller.js* and paste the following code.

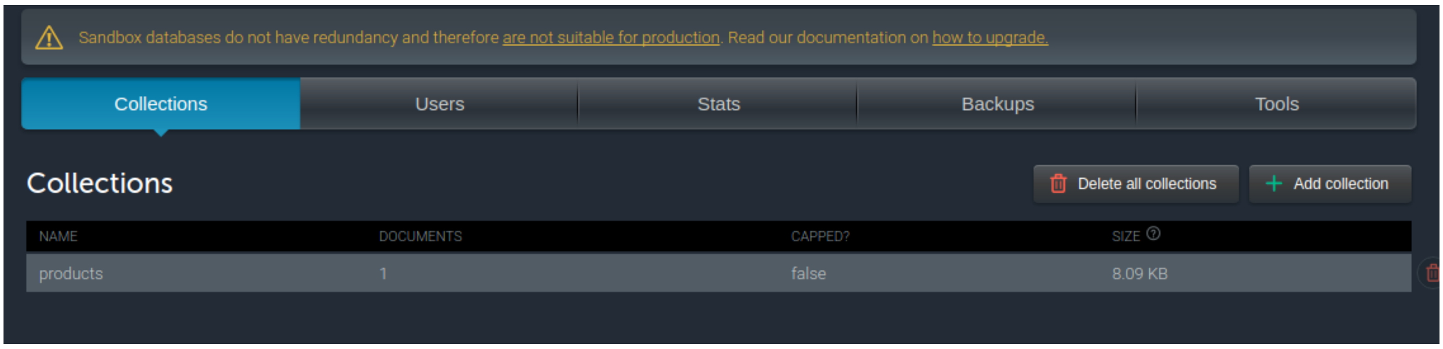
```
// controllers/products.js

exports.product_create = function (req, res) {
    let product = new Product(
        {
            name: req.body.name,
            price: req.body.price
        }
    );

    product.save(function (err) {
        if (err) {
            return next(err);
        }
        res.send('Product Created successfully')
    })
};
```

What the function does is it simply created a new product using the data coming from a POST request and saves it to our database.

Last step would be validating that we can easily create a new product. Let's open Postman. Let's send a POST request to the following url '*localhost:1234/products/create*' and specify the POST data as name: apple and price: 15 as a test example. Also make sure that you choose *x-www-form-urlencoded* in the Body tab in Postman as specified in the image below.

Postman post request example

We can see that the response is 'Product Created successfully. This means that the router and the controller are working correctly. To double check that an 'Apple' product was created, let's check our database. Head to mLab and go to the collections in your database.


Sandbox databases do not have redundancy and therefore are not suitable for production. Read our documentation on how to upgrade.
Collections
Users
Stats
Backups
Tools
Collections
Delete all collections
Add collection

NAME	DOCUMENTS	CAPPED?	SIZE
products	1	false	8.09 KB

collections in our database

We can see above that a new collection was created named 'products' and has one document.

Read

The second task in our CRUD app is to read an existing product. Let's do the route.

```
// routes/products.route.js

...
router.get('/:id', product_controller.product_details);
```

Now let's write the *product_details* controller in our controller file. Head to *controllers/product.controller.js* and paste the following code.

```
// controllers/products.controller.js


exports.product_details = function (req, res) {
    Product.findById(req.params.id, function (err, product)
{
        if (err) return next(err);
        res.send(product);
    })
};
```

What the function does is it simply reads an existing product from the product id being sent in the request.

Now let's head to Postman and try-out our new endpoint. Call the following url '*localhost:1234/products/PRODUCT_ID*'

PRODUCT_ID is the id of the object we've created in the previous endpoint. You should get this from your database and it will be different from mine for sure.

Postman get request example

We got a response containing all the info of that specific product. You can see that it is called appled and it's price is 15.

Update

The third task in our CRUD app is to update an existing product. Let's do the route.

```
// routes/products.route.js

...
router.put('/:id/update',
product_controller.product_update);
```

Now let's write the *product_details* controller in our controller file. Head to *controllers/product.controller.js* and paste the following code.

```
// controllers/products.controller.js
```

```
...
exports.product_update = function (req, res) {
    Product.findByIdAndUpdate(req.params.id, {$set:
req.body}, function (err, product) {
        if (err) return next(err);
        res.send('Product udpated.');
    });
};
```

What the function does is it simply finds an existing product using its id that was sent in the request.

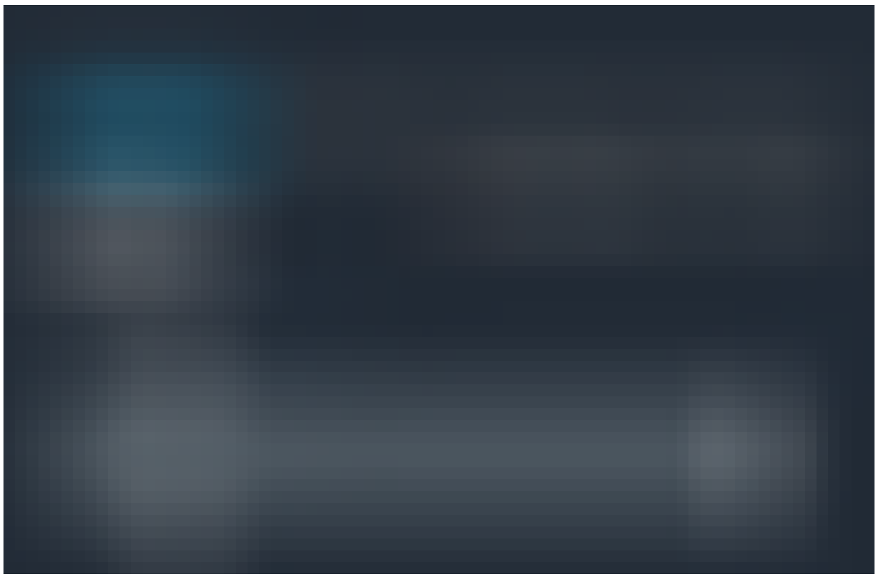
Now let's head to Postman and try-out our new endpoint. Call the following URL '*localhost:1234/products/PRODUCT_ID/update*'

PRODUCT_ID is the id of the object we've created in the previous endpoint. You should get this from your database and it will be different from mine for sure.

Postman updaterequest example

We have updated the product name to 'apple2' and we can see a response saying 'Product updated.'

We can also check the database to see if the database document was updated successfully or not.



Product after update

We can see that the name was succesfully changed from 'Apple' to 'apple2' which implies that our update endpoint is working correctly.

Delete

The last task in our CRUD app is to delete an existing product. Let's do the route.

```
// routes/products.route.js

...
router.delete('/:id/delete',
product_controller.product_delete);
```

Now let's write the *product_delete* controller in our controller file. Head to *controllers/products.js* and paste the following code.

```
// controllers/products.controller.js

exports.product_delete = function (req, res) {
    Product.findByIdAndRemove(req.params.id, function (err)
{
        if (err) return next(err);
        res.send('Deleted successfully!');
    })
};
```

What the function does is it simply deletes an existing product.

Now let’s head to Postman and try-out our new endpoint. Call the following URL ‘*localhost:1234/products/PRODUCT_ID/delete*’

PRODUCT_ID is the id of the object we’ve created in the previous endpoint. You should get this from your database and it will be different from mine for sure.

Postman delete request example

We get a success message stating ‘Deleted successfully’ in the body of our response.

. . .

Done 🎉 🎉

By now, we are done with creating a full API which does the four operations (CRUD)

You can see the full code of this tutorial on the following GitHub repo.

I hope you find this tutorial helpful. In case you have any questions kindly leave a comment below.

PS: I am new in writing tech tutorials. I would love to hear your feedback to improve my future writings :-)

Cheers! 🍻

codeburst.io