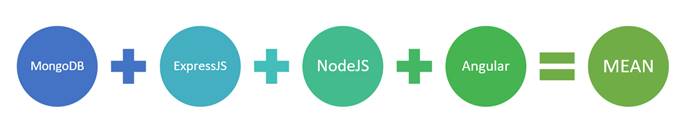
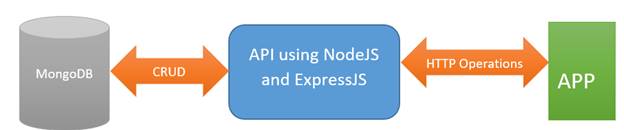
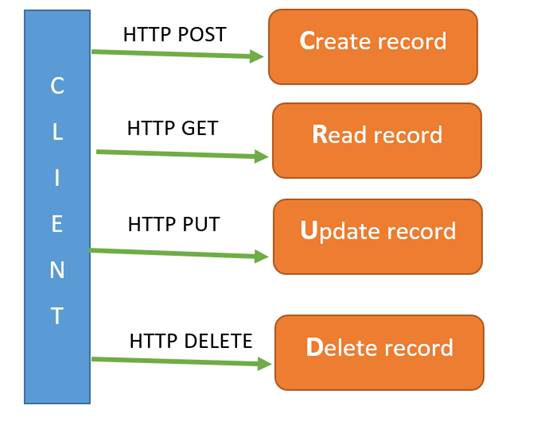
Building NodeJS Based REST API To Perform CRUD On MongoDB

Node.js is one of the greatest platforms to create the back-end of an application. It does not matter whether you are creating a mobile app for iOS or Single Page Web application using Angular, you will need a back-end for the application. Usually, you perform database operations, authentications, authorizations, logging etc. in the back-end. I am sure, you have heard about the term MEAN stack. If not, MEAN stands for MongoDB, Express JS, Angular, and Node.js.  
  
In the MEAN stack, you create back-end of the application using MongoDB, ExpressJS and NodeJS. In this article, you will learn to create a REST API to perform CRUD operations on MongoDB. You will use ExpressJS and NodeJS to create the REST API.

Here, you are going to create a REST API to preform CRUD operations.  
  


From the client, you will perform operations such as GET, POST, PUT, and DELETE to perform CRUD operations. Therefore, to create a record, you will perform HTTP POST operation and pass data to be inserted in the body of the HTTP request.  
  
  
At the end of this post, you will have a working API running which will perform CRUD operations.

**Installation**

To follow along this post, make sure you have following software installed on your machine.

* [MongoDB](https://docs.mongodb.com/manual/installation/)
* [NodeJS](https://nodejs.org/en/)

To configure MongoDB, after installation, make sure that you have it in your primary drive where OS is installed (C drive in most of the cases on Windows) a folder called data\db. MongoDB looks for this folder by default to work with. Also, make sure to start MongoDB Server by running Mongod.exe command as shown in the image below, to work with this post.  
  


After successful installation, create a blank folder in which you will create the project. Give whatever name you wish to give to this folder. I am calling it as Project folder.

**Step 1 Install dependencies**

In the project folder, add a file with name package.json. We are going to use npm (Node Package Manager) to install all the dependencies. Node Package Manager (npm) reads package.json file to install the dependencies in the project. Add the below code in package.json.

1. {
2. "name": "product-api",
3. "main": "server.js",
4. "dependencies": {
5. "express": "~4.0.0",
6. "body-parser": "~1.0.1",
7. "cors": "2.8.1",
8. "mongoose": "~3.6.13"
9. }
10. }

 In this, we are mentioning that the project we are going to create will have the following dependencies.

* Express JS to create API routes.
* Body parser to parse JSON data coming from the client.
* CORS to enable the cross origin resource sharing support in your API.
* Monogoose to work with MongoDB in NodeJS.

After updating package.json with the above code, open the project in the command prompt and run the command *npm install*. I am assuming here that you have NodeJS installed already on your system. If you haven’t installed it, click [here](https://nodejs.org/en/) and install.

Once you have successfully run npm install command, you will find a node\_modules folder which is created inside your project, containing all the required dependencies to create REST API in NodeJS, performing CRUD operations on MongoDB.

**Step 2 Create Model**

When you work with RDBMS such as SQL Server or MySQL, you create a model to perform CRUD operations on that particular table. MongoDB is a document-based database, and in this, you can insert, delete, update or read documents. MongoDB does not restrict you to create fixed model to perform database operations. However, to work with mongoose, you need a model. So, let us create a model of the document to perform CRUD operations.

In the project, add a file called product.js and create model with name Product as shown in the listing below,

1. **var** mongoose = require('mongoose');
2. **var** Schema = mongoose.Schema;
3. **var** ProductSchema = **new** Schema({
4. title: String,
5. price: Number,
6. instock: Boolean,
7. photo: String,
8. });
9. module.exports = mongoose.model('Product', ProductSchema);

As you might have noticed, Product model contains four properties of different data types.

**Step 3 Create Server**

In this step, we will create server for REST API. For creating a server, add a file in the project called server.js. Go back and examine package.json file, you will find that the value of main is set as server.js. Therefore, node server will look for server.js to start the server. We are going to put all the starting codes in the file.

To start with, in server.js, add the following required statements at the top of the file.

1. **var** express = require('express');
2. **var** bodyParser = require('body-parser');
3. **var** cors = require('cors');
4. **var** app = express();
5. **var** mongoose = require('mongoose');
6. **var** product = require('./product');

After importing all the required modules, create route, assign port, and use body parser to parse incoming JSON data. To do these, after last require statement add the codes listed below in server.js file.

1. app.use(bodyParser.urlencoded({ extended: **true** }));
2. app.use(bodyParser.json());
3. **var** port = process.env.PORT || 8090;
4. **var** router = express.Router();

In the above snippet, we are assigning port and creating the router. After this, at the end of the server.js file, add the code listed below,

1. app.use(cors());
2. app.use('/api', router);
3. app.listen(port);
4. console.log('REST API is runnning at ' + port);

In the above snippet, you are enabling CORS support, configuring port for the API, and also configuring that REST API would be created on baseurl/api/{routename}. At this point, you have configured server for various API settings and assigned port to work with that. Keeping everything together, at the end of this step, server.js file will look like below,

**Server.js**

1. **var** express = require('express');
2. **var** bodyParser = require('body-parser');
3. **var** cors = require('cors');
4. **var** app = express();
5. **var** mongoose = require('mongoose');
6. **var** product = require('./product');
8. app.use(bodyParser.urlencoded({ extended: **true** }));
9. app.use(bodyParser.json());
10. **var** port = process.env.PORT || 8090;
11. **var** router = express.Router();

14. // all other code will go here

17. app.use(cors());
18. app.use('/api', router);
19. app.listen(port);
20. console.log('REST API is runnning at ' + port);

**Step 3 Connect to MongoDB server**

In this step, you will connect application to MongoDB database server. To do this, just after router creation code (or at the place where there is a comment “all codes goes here”), add the code mentioned below:

1. mongoose.connect('mongodb://localhost:27017/products');

You are connecting to product database of MongoDB server. By default, MongoDB runs on the port 27017.

**Note**

At this point, make sure that your MongoDB server is running on this port, otherwise you will get an exception when running the API. On Windows, to run the server- navigate to c:\program files\mongodb\server\3.4\bin on the command prompt and run the command mongod.exe. This command will start the MongoDB server. Here I am assuming that, you already have MongoDB installed. If you do not have, you can download and install MongoDB from [Here](https://www.mongodb.com/download-center?jmp=nav#community)

**Step 4 Creating the middle route**

Let us start with writing a route, which will be called before any route. In this route, you can perform various operations such as :

1. Authentication
2. Authorization
3. Logging

In Express JS, you can very easily create this route by adding a “use route” as shown in the listing below,

1. router.use(**function** (req, res, next) {
2. // do logging
3. // do authentication
4. console.log('Logging of request will be done here');
5. next(); // make sure we go to the next routes and don't stop here
6. });

Make sure to add above code in server.js, just after the code you created in step 3.

**Step 5 Creating Record**

To create the record or insert a new document, you need to create a route for the POST operation. In REST API, the convention is that, whenever a client will perform the POST operation, a new record will be created. So, let us start with creating a route for the POST operation. To create that, just below the “use route” line of code, add the code mentioned below,

1. router.route('/products').post(**function** (req, res) {
2. });

The above route will be called, whenever a client will perform HTTP POST operation on baseurl/api/products.

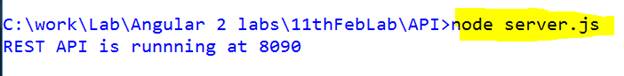
Next, you need to write the code to insert data in the database. For that, you need to perform two operations.

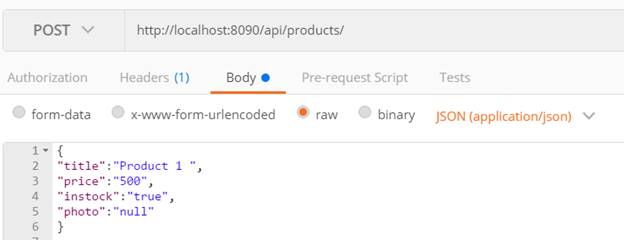
1. Create model object using the request body. If you remember, we have created a model in the step 2.
2. Call the save function on model object to save the record in the database.

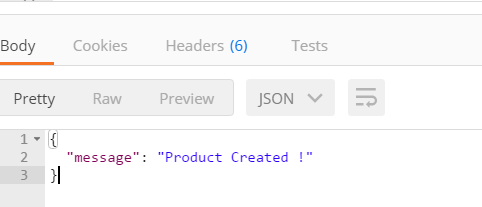
To do the above tasks, you need to modify POST route as shown below,

1. router.route('/products').post(**function** (req, res) {
2. **var** p = **new** product();
3. p.title = req.body.title;
4. p.price = req.body.price;
5. p.instock = req.body.instock;
6. p.photo = req.body.photo;
7. p.save(**function** (err) {
8. **if** (err) {
9. res.send(err);
10. }
11. res.send({ message: 'Product Created !' })
12. })
13. });

**Run and Test**

To test the Create API, run the command  
  


Now, your API is running on the port 8090. To test the API, open POSTMAN and perform a POST operation on URL baseurl/api/products.  
  


Besides the above configuration, in Headers, a content-type is added as*application/json.*When you click on send, if everything happens as expected, you will be getting the response from the API as shown in the image below.  
  


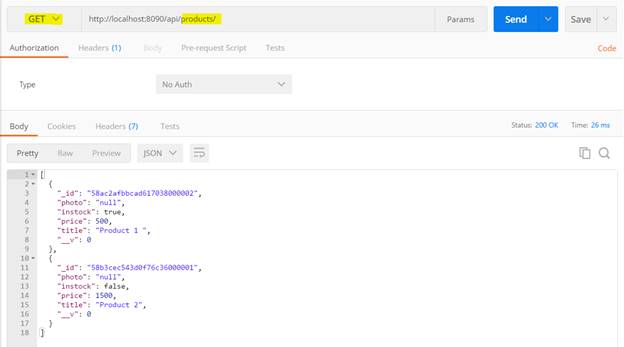
You might be wondering, in which database product it has been created. In the previous steps, you connected to database “products”. So all the operations will be performed with Products database. Also in step 2, while creating a model, since we have not provided any collection name, record will be created in products (plural of model name product) collection.

**Step 6 Fetching Records**

To fetch records, you need to perform GET operation. To do so, add a get route to the API. A GET route could be added as shown in the listing below,

1. router.route('/products').get(**function** (req, res) {
2. product.find(**function** (err, products) {
3. **if** (err) {
4. res.send(err);
5. }
6. res.send(products);
7. });
8. });

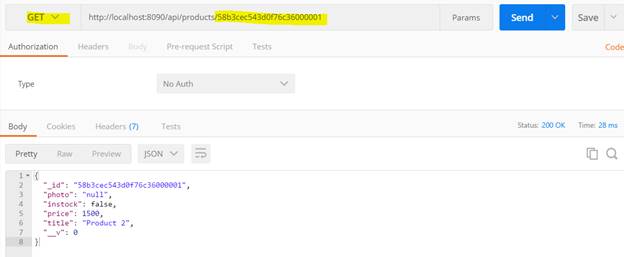
In the above listing, you are using mongoose ‘find function’ to fetch all the records from the MongoDB collection.

The above route will be called, whenever client will perform HTTP GET operation on baseurl/api/products. To test API, open POSTMAN and perform a GET operation on URL baseurl/api/products.  
  


In Postman, you will find that, HTTP GET operation returns all the records. To return to a particular record on basis of ‘product\_id’, add a new route to the API such that, whenever you perform GET operation with product\_id as query parameter on the same URL, you will fetch a particular record.

1. router.route('/products/:product\_id').get(**function** (req, res) {
3. product.findById(req.params.product\_id, **function** (err, prod) {
4. **if** (err)
5. res.send(err);
6. res.json(prod);
7. });
8. });

To fetch a particular product, you are using mongoose findById() function which takes product\_id as input parameter. To read input parameter from HTTP GET operation, we are using req.params property.

The above route will be called, whenever client will perform HTTP GET operation on baseurl/api/products/{product\_id}. To test API, open POSTMAN and perform a GET operation on URL baseurl/api/products/{product\_id}.  
  


You will find that a particular product has been returned with matching product\_id.

**Step 7 Updating Record**

To update a record, you need to perform HTTP PUT operation. While performing PUT operation,

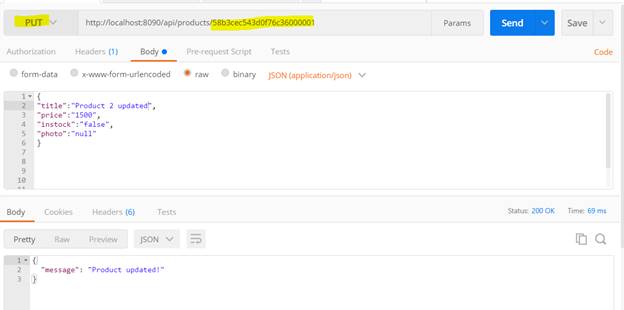
* Pass product\_id need to be updated as query parameter.
* Pass product object need to be updated as request body.

To perform PUT operation, create a PUT route as shown in the listing below,

1. router.route('/products/:product\_id').put(**function** (req, res) {
3. product.findById(req.params.product\_id, **function** (err, prod) {
4. **if** (err) {
5. res.send(err);
6. }
7. prod.title = req.body.title;
8. prod.price = req.body.price;
9. prod.instock = req.body.instock;
10. prod.photo = req.body.photo;
11. prod.save(**function** (err) {
12. **if** (err)
13. res.send(err);
15. res.json({ message: 'Product updated!' });
16. });
18. });
19. });

So, what is going on in above code listing? These are the operations that need to be done to perform the updated operation.

* Fetch product to be updated from the collection.
* Update fetched product properties with the request body object properties.
* Save the product to the collection using save method.

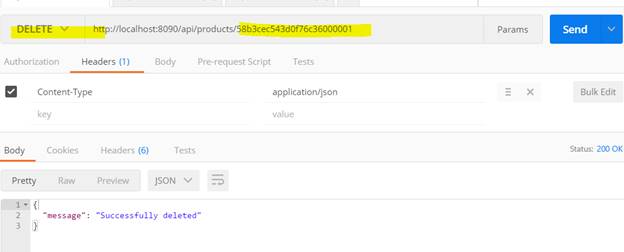
The above route will be called to update a particular product, whenever the client will perform HTTP PUT operation on baseurl/api/products/{product\_id}. To test API, open POSTMAN and perform a PUT operation on URL baseurl/api/products/{product\_id}. You need to pass product object to be updated in the body of the request as shown in the image below,  
  


**Step 8 Deleting Record**

To delete a particular record, client needs to perform HTTP DELETE operation. To do this, create a route for delete operation.

1. router.route('/products/:product\_id').**delete**(**function** (req, res) {
3. product.remove({ \_id: req.param.product\_id }, **function** (err, prod) {
4. **if** (err) {
5. res.send(err);
6. }
7. res.json({ message: 'Successfully deleted' });
8. })
10. });

Client will pass product\_id to be deleted in the query parameter. Note : Product remove method is used to remove a product from the collection.

The above route will be called to delete a particular product, whenever client will perform HTTP DELETE operation on baseurl/api/products/{product\_id}. To test API, open POSTMAN and perform a DELETE operation on URL baseurl/api/products/{product\_id} as shown in the image below,  
  


You just created a full working REST API to perform CRUD operations on MongoDB using NodeJS