Building RESTful Web APIs with Node.js, Express, MongoDB and TypeScript Documentation
Release 1.0.1

Dale Nguyen

May 08, 2021

Contents:

This is a simple API that saves contact information of people.

There are two versions of this project.

V1.0.0: you can run the server directly after cloning this version. It will create a simple RESTful API over HTTP.

V2.0.0: this is a more secure and control API project. You need to read the post on how to secure RESTful API application first. After that, you can run the project.

Fig. 1: (Image from OctoPerf)

CHAPTER 1

Introductions

This book is about how to create a Web APIs from NodeJS, MongoDB, Express and TypeScript. There are lots of things that need to improve in this book. If you find one, please leave a comment. I'm appreciated that ;)

1.1 Who is this book for?

If you are interested in building Web APIs by taking advancage of the benefits of Node.js, Express, MongoDB and TypeScript, this book is perfect for you. This book assumes that you already have some knowlege of JavaScript and NoSQL Database.

1.2 How to read this book?

The chapters in this book are meant to be read in order. You can skip some parts of some chapters, if you have existing knowlege.

CHAPTER 2

Setting Up Project

2.1 Before we get started

Make sure that you have NodeJS installed on your machine. After that, you have to install TypeScript and TypeScript Node.

```
npm install -g typescript ts-node
```

In order to test HTTP request, we can use Postman to send sample requests.

2.2 MongoDB preparation

You should install MongoDB on your local machine, or use other services such as mLab or Compose

If you installed MongoDB locally, you should install either Robo Mongo or Mongo Compass for GUI interface.

Before we dive into the coding part, you can checkout my github repository if you want to read the configuration in advance. Otherwise, you just need to follow the steps in order to get your project run.

2.3 Step 1: Initiate a Node project

Create a project folder and initiate the npm project. Remember to answer all the question, and you can edit it any time after that

```
mkdir node-apis-project
cd node-apis-project
npm init
```
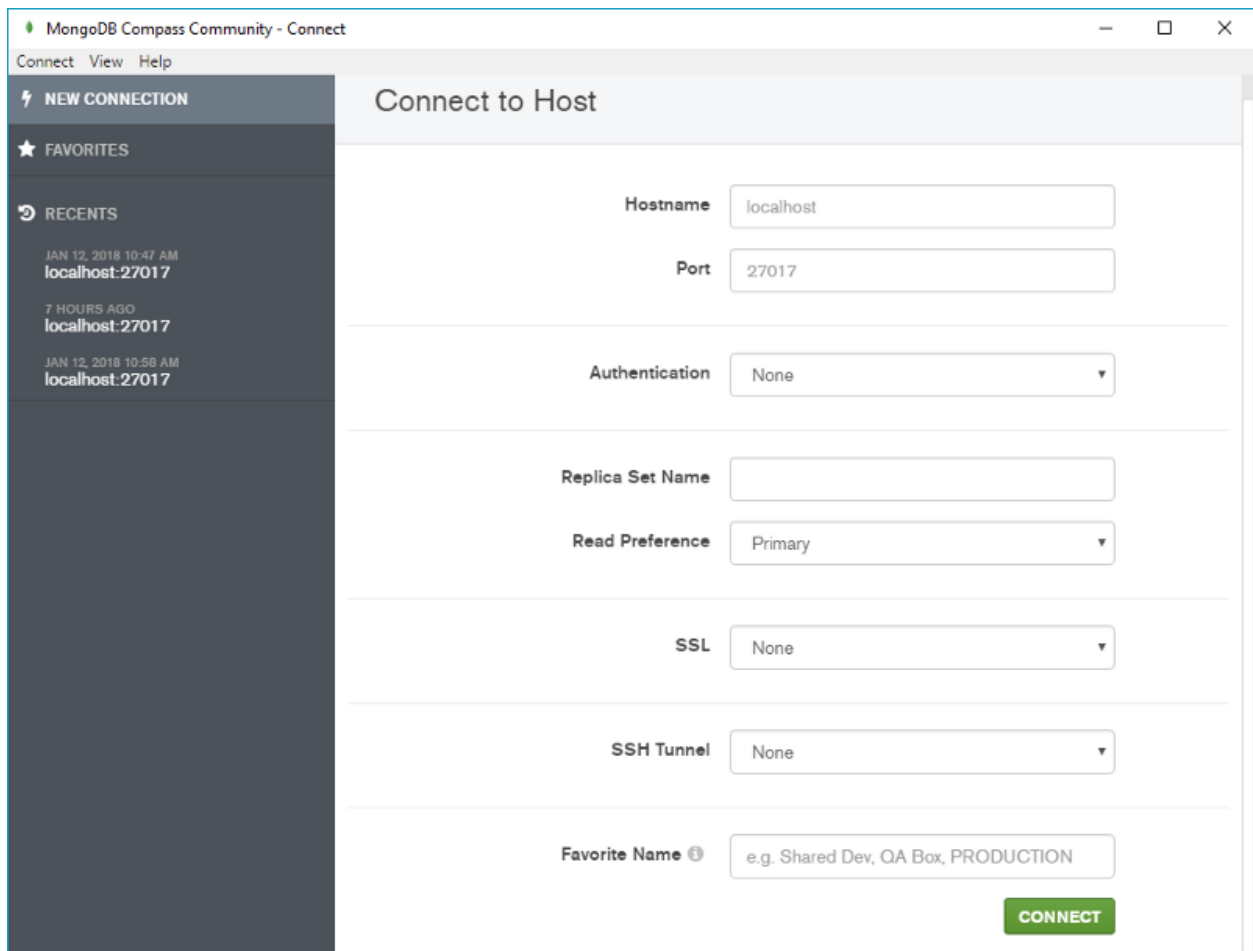
Fig. 1: *MongoDB Compass GUI Interface*

2.4 Step 2: Install all the dependencies

```
npm install --save @types/express express body-parser mongoose nodemon
```

2.5 Step 3: Configure the TypeScript configuration file (tsconfig.json)

The idea is to put all the TypeScript files in the **lib folder** for development purpose, then for the production, we will save all the Javascript files in the **dist folder**. And of course, we will take advantage of the ES2015 in the project.

```
{
    "compilerOptions": {
        "module": "commonjs",
        "moduleResolution": "node",
        "pretty": true,
        "sourceMap": true,
        "target": "es6",
        "outDir": "./dist",
        "baseUrl": "./lib"
    },
    "include": [
        "lib/**/*.ts"
    ],
    "exclude": [
        "node_modules"
    ]
}
```

So whenever we run the tsc command, all the ts files in the lib folder will be compiled to js files in the dist folder

```
tsc
```

2.6 Step 4: edit the running scripts in package.json

```
{
    "scripts": {
        "build": "tsc",
        "dev": "ts-node ./lib/server.ts",
        "start": "nodemon ./dist/server.js",
        "prod": "npm run build && npm run start"
    }
}
```

So, for the development, we can run a test server by running

```
npm run dev
```

For production

```
npm run prod
```

2.7 Step 5: getting started with the base configuration

You will need sooner or later the package body-parse for parsing incoming request data.

```typescript
// lib/app.ts

import * as express from "express";
import * as bodyParser from "body-parser";

class App {

    public app: express.Application;

    constructor() {
        this.app = express();
        this.config();
    }

    private config(): void{
        // support application/json type post data
        this.app.use(bodyParser.json());
        //support application/x-www-form-urlencoded post data
        this.app.use(bodyParser.urlencoded({ extended: false }));
    }

}

export default new App().app;
```
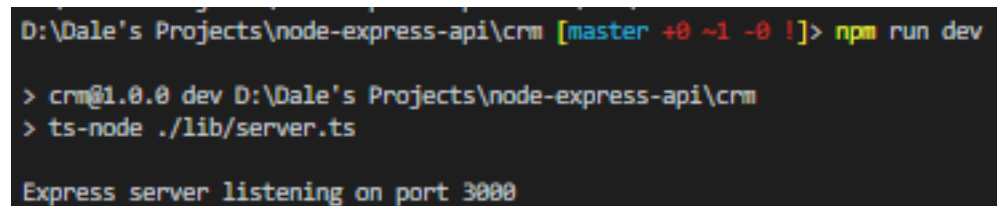
Create **lib/server.ts** file

```typescript
// lib/server.ts

import app from "./app";
const PORT = 3000;

app.listen(PORT, () => {
    console.log('Express server listening on port ' + PORT);
})
```

From now, although you can not send a HTTP request yet, you still can test the project by running **npm run dev**.

Implement Routing and CRUD

In this chapter, we will build the routing for the API.

3.1 Step 1: Create TS file for routing

Remember in part 1 of this project. We save everything in **lib** folder. So I will create **routes folder** with a file named **crmRoutes.ts** that will save all the routes for this project.

```typescript
// /lib/routes/crmRoutes.ts

import {Request, Response} from "express";

export class Routes {
    public routes(app): void {
        app.route('/')
        .get((req: Request, res: Response) => {
            res.status(200).send({
                message: 'GET request successfulll!!!!'
            })
        })
    }
}
```

After creating our first route, we need to import it to the **lib/app.ts**.

```typescript
// /lib/app.ts

import * as express from "express";
import * as bodyParser from "body-parser";
import { Routes } from "./routes/crmRoutes";

class App {
```

```typescript
    public app: express.Application;
    public routePrv: Routes = new Routes();

    constructor() {
        this.app = express();
        this.config();
        this.routePrv.routes(this.app);
    }

    private config(): void{
        this.app.use(bodyParser.json());
        this.app.use(bodyParser.urlencoded({ extended: false }));
    }
}
```

Now, you can send GET request to your application (http://localhost:3000) directly or by using Postman .

3.2 Step 2: Building CRUD for the Web APIs

I assume that you have a basic understanding of HTTP request (GET, POST, PUT and DELETE). If you don't, it is very simple:

- GET: for retrieving data

- POST: for creating new data

- PUT: for updating data

- DELETE: for deleting data

Now we will build the routing for building a contact CRM that saves, retrieves, updates and deletes contact info.

```typescript
// /lib/routes/crmRoutes.ts

import {Request, Response} from "express";

export class Routes {

    public routes(app): void {

        app.route('/')
        .get((req: Request, res: Response) => {
            res.status(200).send({
                message: 'GET request successfulll!!!!'
            })
        })

        // Contact
        app.route('/contact')
        // GET endpoint
        .get((req: Request, res: Response) => {
        // Get all contacts
            res.status(200).send({
                message: 'GET request successfulll!!!!'
            })
        })
```

```typescript
        // POST endpoint
        .post((req: Request, res: Response) => {
        // Create new contact
            res.status(200).send({
                message: 'POST request successfulll!!!!'
            })
        })

        // Contact detail
        app.route('/contact/:contactId')
        // get specific contact
        .get((req: Request, res: Response) => {
        // Get a single contact detail
            res.status(200).send({
                message: 'GET request successfulll!!!!'
            })
        })
        .put((req: Request, res: Response) => {
        // Update a contact
            res.status(200).send({
                message: 'PUT request successfulll!!!!'
            })
        })
        .delete((req: Request, res: Response) => {
        // Delete a contact
            res.status(200).send({
                message: 'DELETE request successfulll!!!!'
            })
        })
    }
}
```

Now the routes are ready for getting HTTP request

Using Controller and Model

In this chapter, we will show you how to use Controller and Model for creating, saving, editing and deleting data. Remember to read the previous parts before you move forward.

4.1 Create Model for your data

All the model files will be saved in **/lib/models** folder. We will define the structure of the Contact by using Schema from Mongoose .

```typescript
//   /lib/models/crmModel.ts

import * as mongoose from 'mongoose';

const Schema = mongoose.Schema;

export const ContactSchema = new Schema({
    firstName: {
        type: String,
        required: 'Enter a first name'
    },
    lastName: {
        type: String,
        required: 'Enter a last name'
    },
    email: {
        type: String
    },
    company: {
        type: String
    },
    phone: {
        type: Number
    },
```

```
    created_date: {
        type: Date,
        default: Date.now
    }
});
```

This model will be used inside the controller where we will create the data.

4.2 Create your first Controller

Remember in previous chapter, We created CRUD place holder for communicating with the server. Now we will apply the real logic to the route and controller.

4.2.1 1. Create a new contact (POST request)

All the logic will be saved in the **/lib/controllers/crmController.ts**

```
//   /lib/controllers/crmController.ts

import * as mongoose from 'mongoose';
import { ContactSchema } from '../models/crmModel';
import { Request, Response } from 'express';

const Contact = mongoose.model('Contact', ContactSchema);
export class ContactController{
...
public addNewContact (req: Request, res: Response) {
    let newContact = new Contact(req.body);

    newContact.save((err, contact) => {
        if(err){
            res.send(err);
        }
        res.json(contact);
    });
}
```

In the route, we don't have to pass anything.

```
// /lib/routes/crmRoutes.ts

import { ContactController } from "../controllers/crmController";

public contactController: ContactController = new ContactController();

// Create a new contact
app.route('/contact')
.post(this.contactController.addNewContact);
```

4.2.2 2. Get all contacts (GET request)

All the logic will be saved in the **/lib/controllers/crmController.ts**

```
//    /lib/controllers/crmController.ts

public getContacts (req: Request, res: Response) {
        Contact.find({}, (err, contact) => {
            if(err){
                res.send(err);
            }
            res.json(contact);
        });
    }
}
```

After that, we will import **ContactController** and apply **getContacts** method.

```
// /lib/routes/crmRoutes.ts

// Get all contacts
app.route('/contact')
.get(this.contactController.getContacts)
```

4.2.3 3. View a single contact (GET method)

We need the ID of the contact in order to view the contact info.

```
//    /lib/controllers/crmController.ts

public getContactWithID (req: Request, res: Response) {
    Contact.findById(req.params.contactId, (err, contact) => {
        if(err){
            res.send(err);
        }
        res.json(contact);
    });
}
```

In the routes, we simply pass the **'/contact/:contactId'**

```
// /lib/routes/crmRoutes.ts

// get a specific contact
app.route('/contact/:contactId')
.get(this.contactController.getContactWithID)
```

4.2.4 4. Update a single contact (PUT method)

Remember that, without **{new: true}**, the updated document will not be returned.

```
//    /lib/controllers/crmController.ts

public updateContact (req: Request, res: Response) {
    Contact.findOneAndUpdate({ _id: req.params.contactId }, req.body, { new: true },
→(err, contact) => {
        if(err){
            res.send(err);
```

```
        }
        res.json(contact);
    });
}
```

In the routes,

```typescript
// /lib/routes/crmRoutes.ts


// update a specific contact
app.route('/contact/:contactId')
    .put(this.contactController.updateContact)
```

4.2.5 5. Delete a single contact (DELETE method)

```typescript
//   /lib/controllers/crmController.ts

public deleteContact (req: Request, res: Response) {
    Contact.remove({ _id: req.params.contactId }, (err, contact) => {
        if(err){
            res.send(err);
        }
        res.json({ message: 'Successfully deleted contact!'});
    });
}
```

In the routes,

```typescript
// /lib/routes/crmRoutes.ts


// delete a specific contact
app.route('/contact/:contactId')
    .delete(this.contactController.deleteContact)
```

Important: Remember that you don't have to call **app.route('/contact/:contactId')** every single time for GET, PUT or DELETE a single contact. You can combine them.

```typescript
// /lib/routes/crmRoutes.ts

app.route('/contact/:contactId')
    // edit specific contact
    .get(this.contactController.getContactWithID)
    .put(this.contactController.updateContact)
    .delete(this.contactController.deleteContact)
```

From now, your model and controller are ready. We will hook to the MongoDB and test the Web APIs.

Connect Web APIs to MongoDB

In this chapter, we will connect the RESTful API application to local MongoDB, but you can connect to any other database services. Please read *Setting Up Project* to install the MongoDB to your machine.

All that you need to do is to import **mongoose package**, and declare URL for your MongoDB in the **app.ts** file. After that you will connect your app with your database through **mongoose**.

```typescript
// lib/app.ts

import * as mongoose from "mongoose";

class App {

    ...
    public mongoUrl: string = 'mongodb://localhost/CRMdb';

    constructor() {
        ...
        this.mongoSetup();
    }

    private mongoSetup(): void{
        mongoose.Promise = global.Promise;
        mongoose.connect(this.mongoUrl);
    }

}

export default new App().app;
```

After this, your application is ready to launch *(npm run dev)*

```
D:\Dale's Projects\node-express-api\crm [master +0 ~1 -0 !]> npm run dev

> crm@1.0.0 dev D:\Dale's Projects\node-express-api\crm
> ts-node ./lib/server.ts

Express server listening on port 3000
```

You can test your first route *(GET /)* through web browser **(http://127.0.0.1:3000)**

```
← → C    ⓘ 127.0.0.1:3000

▼ {
        "message": "GET request successfulll!!!!"
  }
```
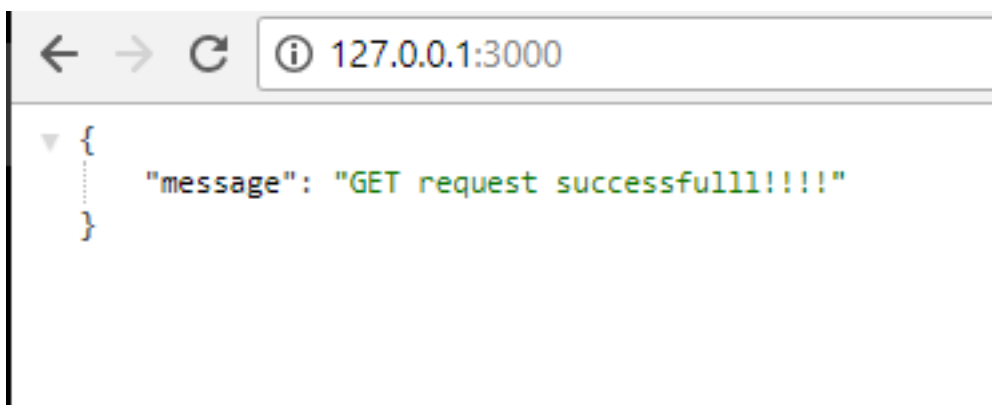
Remember that all the routes that we set is **in lib/routes/crmRoutes.ts** file.

Now, we will test the **Create-Read-Update-Delete** feature though Postman.

5.1 1. Create your first contact

I will send a **POST** request to *http://127.0.0.1:3000/contact* with the information of a contact in the body.

Remember to set the content-type in Headers

```
Content-Type: application/x-www-form-urlencoded
```

After sending, the server return the status *200* with contact information in the database.

5.2 2. Get all contacts

To get all contacts, we just need to send a **GET** request to http://127.0.0.1:3000/contact. You will get an Array of all the contacts in the databse. Now there is only one contact that I just created.

5.3 3. Get contact by Id

If we want to get a single contact by Id, we will send a **GET** request to http://127.0.0.1:3000/contact/:contactId. It will return an Object of your contact. Remember that the ID that we passed to the URL is the **_id** of the contact.

5.4 4. Update an existing contact

In case we want to update an existing contact, we will send a **PUT** request to the http://127.0.0.1:3000/contact/:contactId together with the detail. For example, I will update the phone number of the contact with **_id**: **5b03015e3c4b1a1164212ff4**
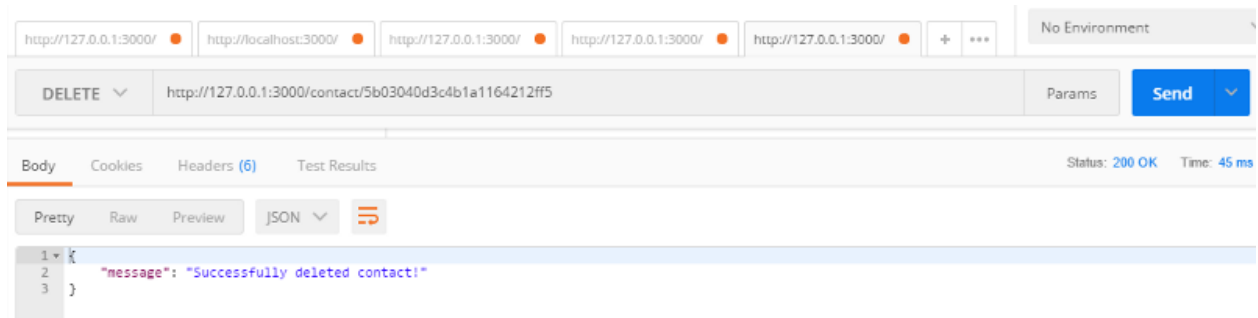

5.5 5. Delete a contact

To delete a contact, we will send a **DELETE** request to http://127.0.0.1:3000/contact/:contactId. It will return a message saying that "Successfully deleted contact!"

After this, now we have a fully working RESTful Web APIs application with TypeScript and Nodejs.

Security for our Web APIs

In this chapter, I will show you various methods to secure your RESTful Web APIs. You should use at least one or combine those methods for a more secure API application.

And if you want to use services like mLab , compose . . . , they have already implemented a secured system on their end. All that you need to do is to follow their instructions to hook the database to your app.

6.1 Method 1: The first and foremost is that you should always use HTTPS over HTTP

For local testing, I will use OpenSSL on Windows to generate the key and certificate for HTTPS configuration. The process is similar on Mac or Linux.

After installing OpenSSL, I will open OpenSSL and start generating key and cert files.

```
OpenSSL> req -newkey rsa:2048 -nodes -keyout keytemp.pem -x509 -days 365 -out cert.pem
OpenSSL> rsa -in keytemp.pem -out key.pem
```

After that, we will move **key.pem** and **cert.pem** files to our project. They will be in the config folder.

Then we will edit the server.ts file to enable https.

```
// server.ts

import app from './app';
import * as https from 'https';
import * as fs from 'fs';
const PORT = 3000;
const httpsOptions = {
    key: fs.readFileSync('./config/key.pem'),
    cert: fs.readFileSync('./config/cert.pem')
}
https.createServer(httpsOptions, app).listen(PORT, () => {
```

```
      console.log('Express server listening on port ' + PORT);
})
```

For testing the server, we will run

```
ts-node .\lib\server.ts
```

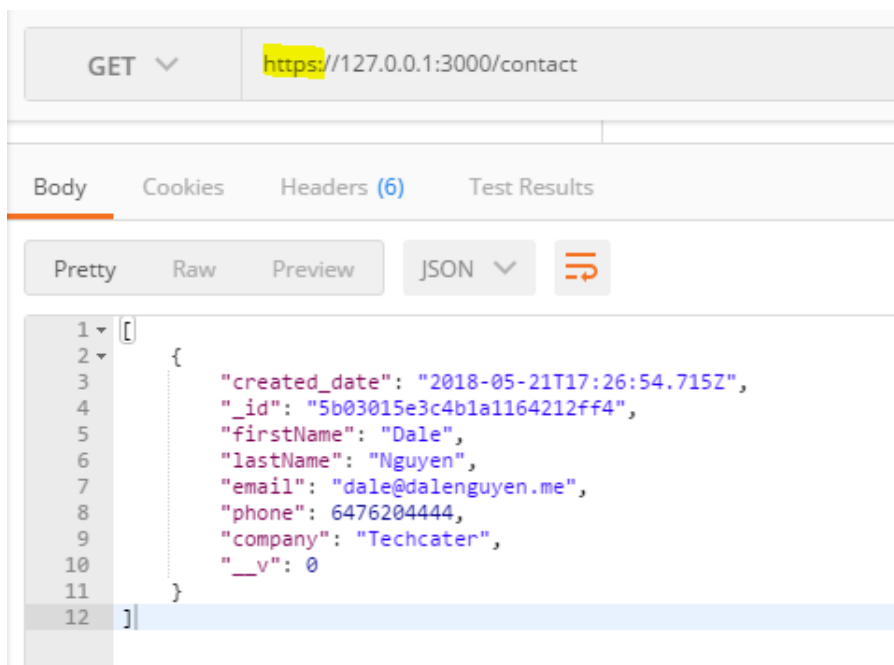From now on, our application will always run over HTTPS.

Fig. 1: Getting data over HTTPS (Postman)

6.2 Method 2: Using secret key for authentication

This method uses a unique key to pass in the URL, so you can access the database. You can use crypto to create a key from your command line.

```
node -e "console.log(require('crypto').randomBytes(20).toString('hex'))"
```


Now, we will use middleware to check for the key before responding to a request. For example, if you want to get all contacts, you need to pass a key.

```
// GET request
https://127.0.0.1:3000?key=78942ef2c1c98bf10fca09c808d718fa3734703e
```

We will edit the **/lib/routes/crmRouters.ts** before sending the request.

Important: Remember that, in production, you should pass the key in the environment, not directly like in the
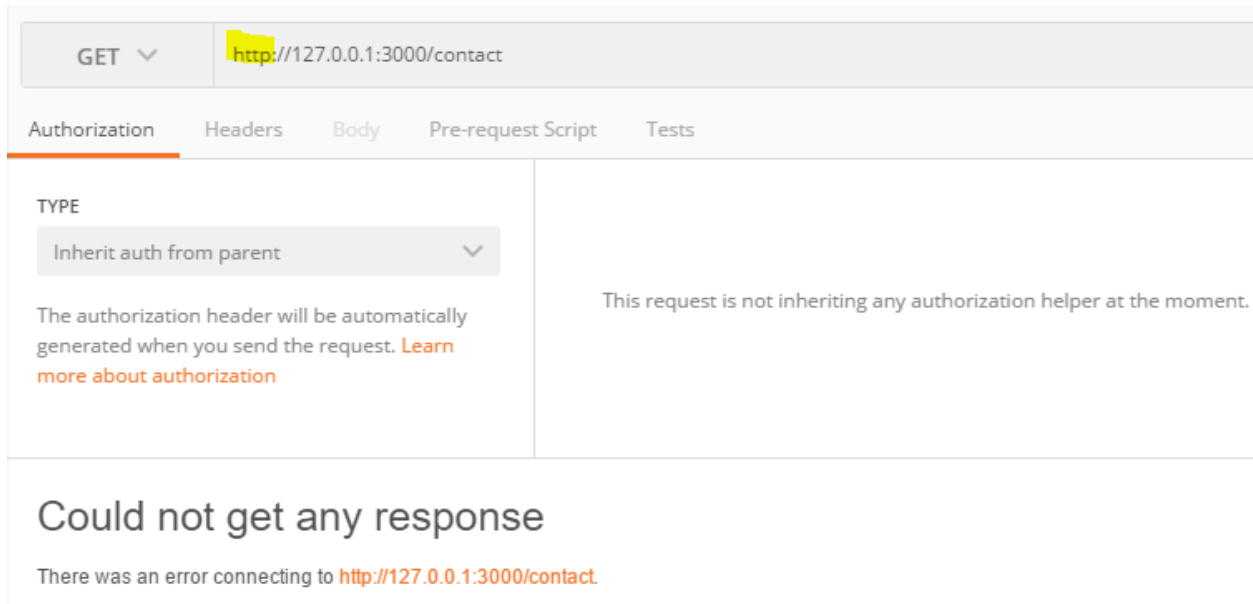
Fig. 2: You will get no response and an error if trying to access over HTTP

example.

```
// lib/routes/crmRouters.ts

// get all contacts
app.route('/contact')
.get((req: Request, res: Response, next: NextFunction) => {
    // middleware
    if(req.query.key !== '78942ef2c1c98bf10fca09c808d718fa3734703e'){
        res.status(401).send('You shall not pass!');
    } else {
        next();
    }
}, this.contactController.getContacts)
```

6.3 Method 3: Secure your MongoDB

It's sad that by default, there is no security for MongoDB like at all. If you want to check your current configuration.
Go to your mongo installation directory and type mongo.
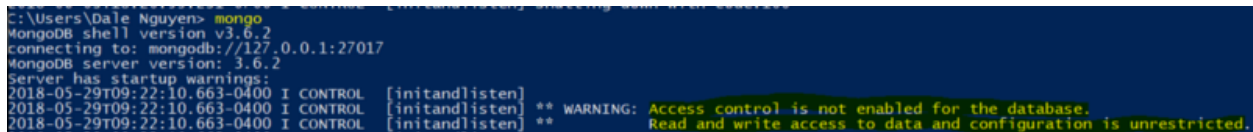
As you can see, there is no Access control for the database and anyone can do anything with the database. So we will
enable authentication feature for MongoDB.

First, we need to create an account in order to authenticate with Mongodb.
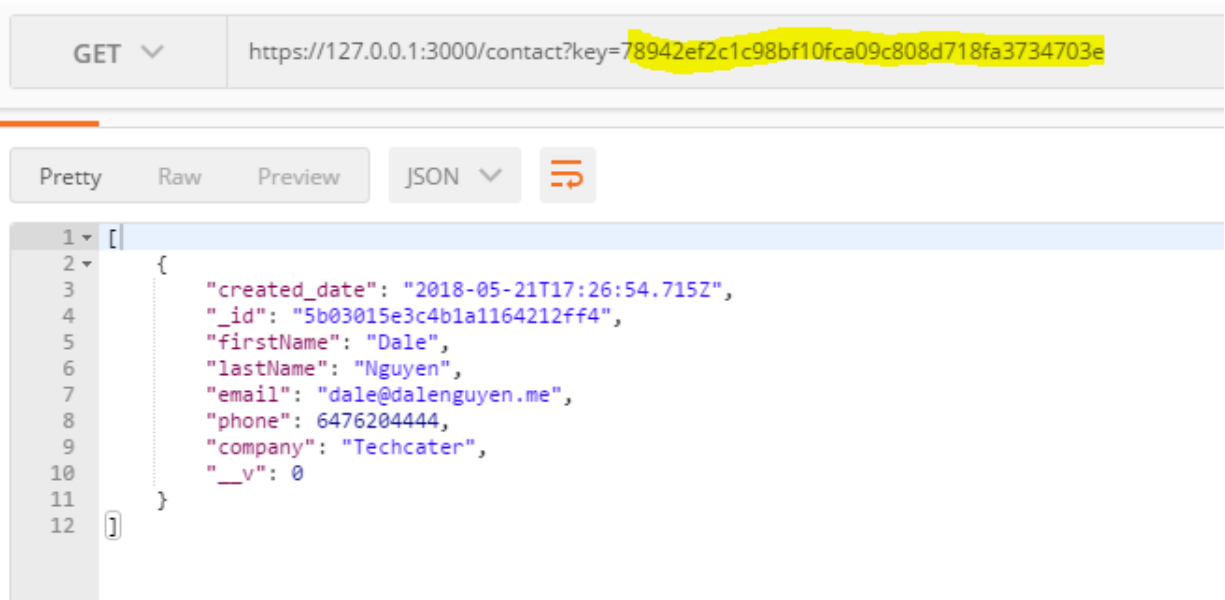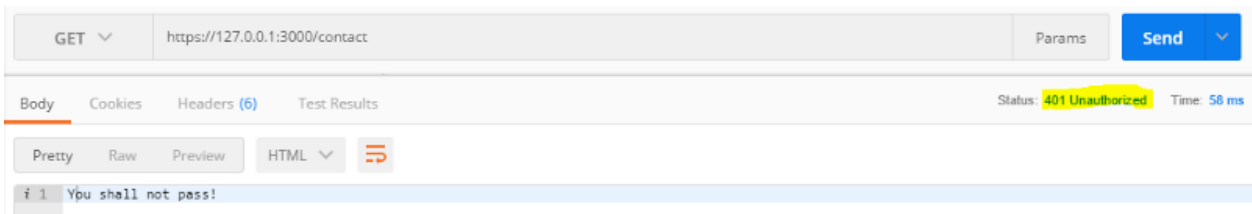
Fig. 3: We are allowed to get the data with key

Fig. 4: You cannot access without a key

```
C:\WINDOWS\system32> mongo
MongoDB shell version v3.6.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.2
> use CRMdb
switched to db CRMdb
> db.createUser(
... {
...   user: 'dalenguyen',
...   pwd: '123123',
...   roles: [ { role: 'readWrite', db: 'CRMdb' } ]
... }
... )
```

After that, we will stop and restart MongoDB with authentication. Remember to check your dbpath.

```
// Stop MongoDB (Windows)
net stop MongoDB

// Start mongodb with authentication
mongod --auth --port 27017 --dbpath C:\your-data\path
```

Now, if we login to the mongo shell, there is no warning about access control.

```
C:\Users\Dale Nguyen> mongo
MongoDB shell version v3.6.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.2
>
```

Or you can connect to the mongo shell with username and password you just created.

```
mongo --port 27017 -u dalenguyen -p 123123  --authenticationDatabase CRMdb
```

Now, if we try to access the database even with the key, we are not able to.


```
GET ∨   https://127.0.0.1:3000/contact?key=78942ef2c1c98bf10fca09c808d718fa3734703e        Params

Body   Cookies   Headers (6)   Test Results                                          Status: 20

Pretty   Raw   Preview   JSON ∨

1  {
2      "name": "MongoError",
3      "message": "not authorized on CRMdb to execute command { find: \"contacts\", filter: {}, projection: {}, $db: \"CRMdb\" }",
4      "ok": 0,
5      "errmsg": "not authorized on CRMdb to execute command { find: \"contacts\", filter: {}, projection: {}, $db: \"CRMdb\" }",
6      "code": 13,
7      "codeName": "Unauthorized"
8  }
```

Fig. 5: Cannot get data even with key

That's why we need to edit the mongodb URL in order for the app to work. Again, you should put the mongodb URI to the environment.

```
// lib/app.ts

class App {
...
public mongoUrl: string = 'mongodb://dalenguyen:123123@localhost:27017/CRMdb';
```

Then you restart RESTful API, everything will starts working fine again, but now you have a more secure and control API application. There are more security methods that we can implement to improve our application. I will try to update all of them in other posts.

After this, now we have a fully secure and working RESTful Web APIs application with TypeScript and Nodejs. If you want to check all the code, please visit my github repository for the full code.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search