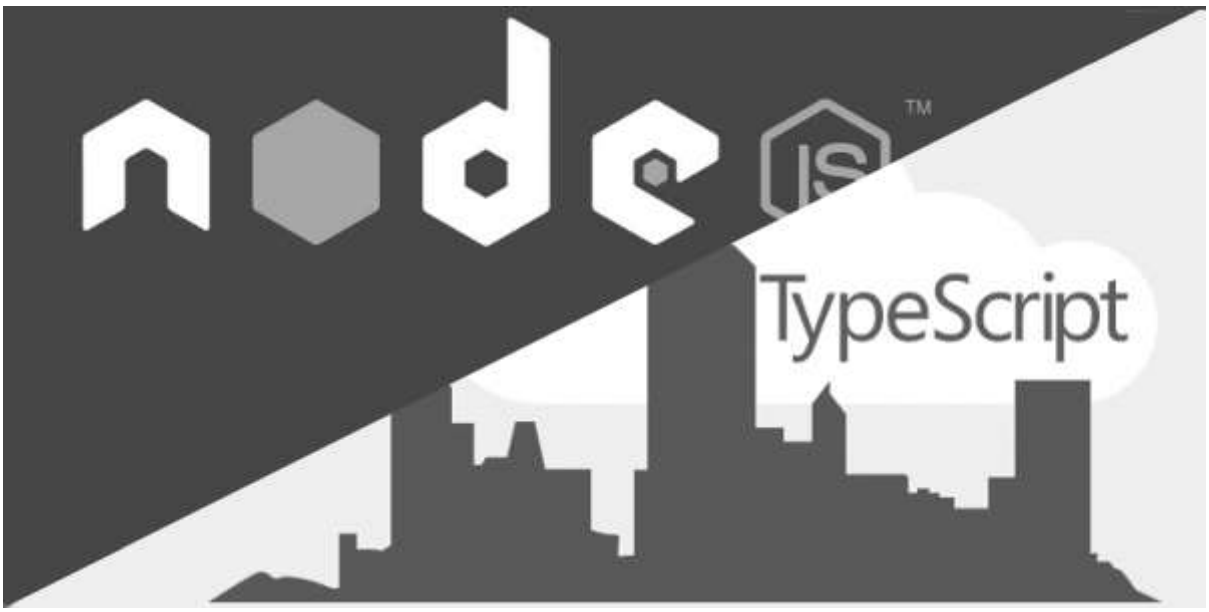
Building RESTful Web APIs with Node.js, Express, MongoDB and TypeScript — Part 3

Dale Nguyen
May 4, 2018 · 3 min read



(Image from OctoPerf)

There is a course about how to build a Web APIs on Lynda, but they didn't use TypeScript. So I decided to make one with TypeScript. There are lots of things that need to improve in this project. If you find one, please leave a comment. I'm appreciated that ;)

Part 1: Setting Up Project

Part 2: Implement routing and CRUD

Part 3: Using Controller and Model for Web APIs

Part 4: Connect Web APIs to MongoDB or others

Part 5: Security for our Web APIs

In this part, I 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.

Step 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.

```
//   /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.

Step 2: Create your first Controller

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);
```

2. Get all contacts (GET request)

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

```
        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)
```

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')
```

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,

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

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

5. Delete a single contact (DELETE method)

```
//   /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,

* 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

```
// /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. This is the end of **Part 3**. I will update **Part 4 and Part 5** shortly. In case you need to jump a head. Please visit my github repository for the full code.

https://github.com/dalenguyen/rest-api-node-typescript

Nodejs Expressjs Mongodb Typescript API