Advanced Web Programming

Lab 11

Backend – Create Own Web Server

In lab 8 we created CRUD application to manage a list of comments. The data for our application were defined on the frontend side as a local model. In the lab 9 I show you how to work with external data resources using Json Server. This approach is accepted only for very simple solution. For advanced solution we should use external web server to manage data.
This lab guide you to build step by step a RESTful API with Node.js, Express, and Mongoose with CRUD functionalities. We start to create own web server whose main goals will be data management (reading from the database, writing to the database). Thanks to this, our application will allow for permanent data persistence.

We will use mongodb as a database and mongoose ORM to do operations on mongo models. We will create a HTTP rest call to fetch mongodb collections details, create record into mongodb, update record into mongodb and delete record from collection.
We will cover following functionality into this class:

- How to create database connection with Mongodb.
- How to create Model using Mongoose.
- How to create collection in Mongodb using Mongoose.
- How to delete collection in Mongodb using Mongoose.
- How to update Collection in Mongodb using Mongoose.
- Creating CRUD operation REST API using Node, ExpressJS, MongoDB and Mongoose.

How to start create web server?

Before we move on, you need to have the following:

- Node.js installed on your machine. (verify by typing `node -v` in your terminal)
- A MongoDB instance running on your machine. You won't need this if you want to use MongoDB Atlas. I recommend to use MongoDB Atlas as an example cloud solution. We will back to MongoDB Atlas (to registry and create database) when in our server we will would like to connect to database.

Let's start by creating back-end step by step from zero

The only thing we need to get started with this project is a blank folder with npm package initialized.

1. Install express with npm

First of all, we need a package.json file to start with. Create a new folder and name it anything you like (I use “CommentsServer”). Open a terminal and go to this folder. Then, run the following command. (You can omit -y for providing details in the package.json file)

npm init -y

This will create a package.json file. Now, we need the express. Express is one of the most popular web frameworks for Node.js that supports routing, middleware. We will use the express to create a server and API. Run the following command to install express.

```
$ npm install express mongoose --save
```

Here, we're installing Express for our web framework and mongoose to interact with our MongoDB database.

Basic Express Server

We can now start to create server.js and create a simple Express server.

```
server.js

const express = require("express")

const app = express()

app.listen(5000, () => {

       console.log("Server has started!")

})
```

We first import our express package that we've just installed. Then, create a new express instance and put it into app variable. This app variable let us do everything we need to configure our REST API, like registering our routes, installing necessary middlewares, and much more.

Try to run our server by running this command below.

```
$ node server.js

Server has started!
```

Alternatively, we can setup a new npm script to make our workflow much more easier.

```
package.json

{

        "scripts": {

                "start": "node server.js"

        }

}
```

Then, we can run our server by executing `npm start`.

```
$ npm start

Server has started!
```

As You see us server works, but it doesn't do anything interesting.

Next step we install another dependencies like: ‘BodyParser’ and ‘Cors’. BodyParser is used for parsing request from client and cors is used to prevent Cross-Origin Request Block error while we try to communicate from front-end to back-end.

Run the following command to install dependencies.

```
$ npm install body-parser cors --save
```

And update server.js writing code like this:

```
const express = require("express");
const cors = require("cors");
var corsOptions = {
  origin: "http://localhost:8080"
};

const app = express();
app.use(cors(corsOptions));

// parse requests of content-type - application/json
app.use(express.json());
// parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));

// simple route
app.get("/", (req, res) => {
  res.json({ message: "Server lives!!!" });
});

app.listen(5000, () => {
        console.log("Server has started!")
})
```

What we do are:

– import express and cors modules:

Express is for building the Rest apis

cors provides Express middleware to enable CORS with various options.

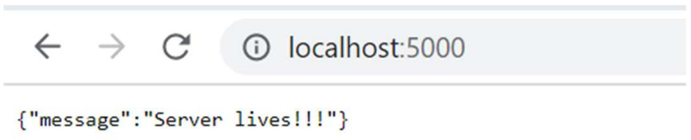
– create an Express app, then add body-parser (json and urlencoded) and cors middlewares using app.use() method. Notice that we set origin: http://localhost:8080.

– define a GET route which is simple for test.

– listen on port 5000 for incoming requests.

Now let’s run the app with command: node server.js.

Open your browser with url http://localhost:5000/, you will see:



```
{"message":"Server lives!!!"}
```

the first step is done. We’re gonna work with Mongoose in the next section.

How to Connect MongoDB to Node.js Using Mongoose

MongoDB is one of the most widely used No-SQL databases in the developer world today. No-SQL databases allow developers to send and retrieve data as JSON documents, instead of SQL objects. To work with MongoDB in a Node.js app, we can use Mongoose.

Database connection

Before creating the routes, let us connect our app to MongoDB. For that we first need to import mongoose in server.js

```
const mongoose = require("mongoose")
```

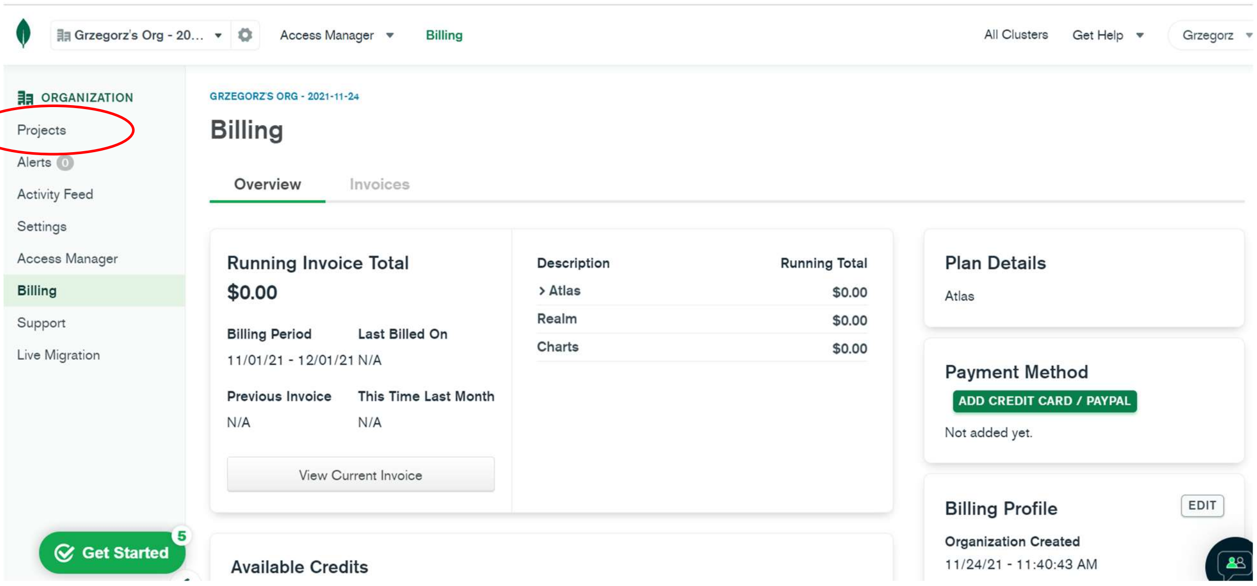
```
);
```

As I said before I suggest to use MongoDB Atlas as a database. MongoDB Atlas is a cloud solution where we can host our database. It has a free tier with a max storage of 512 MB. This is ideal for experimentation and learning. To create a connection to MongoDB Atlas, follow the next steps.

Get started by going to the MongoDB Atlas website https://account.mongodb.com/account/register or https://www.mongodb.com/atlas/database and sign up as new user.

Fill the registration form with your information and click **Sign up.**

Next you will see the screen



GRZEGORZ'S ORG - 2021-11-24

Billing

Running Invoice Total	
$0.00	
Billing Period	Last Billed On
11/01/21 - 12/01/21	N/A
Previous Invoice	This Time Last Month
N/A	N/A

Description	Running Total
> Atlas	$0.00
Realm	$0.00
Charts	$0.00

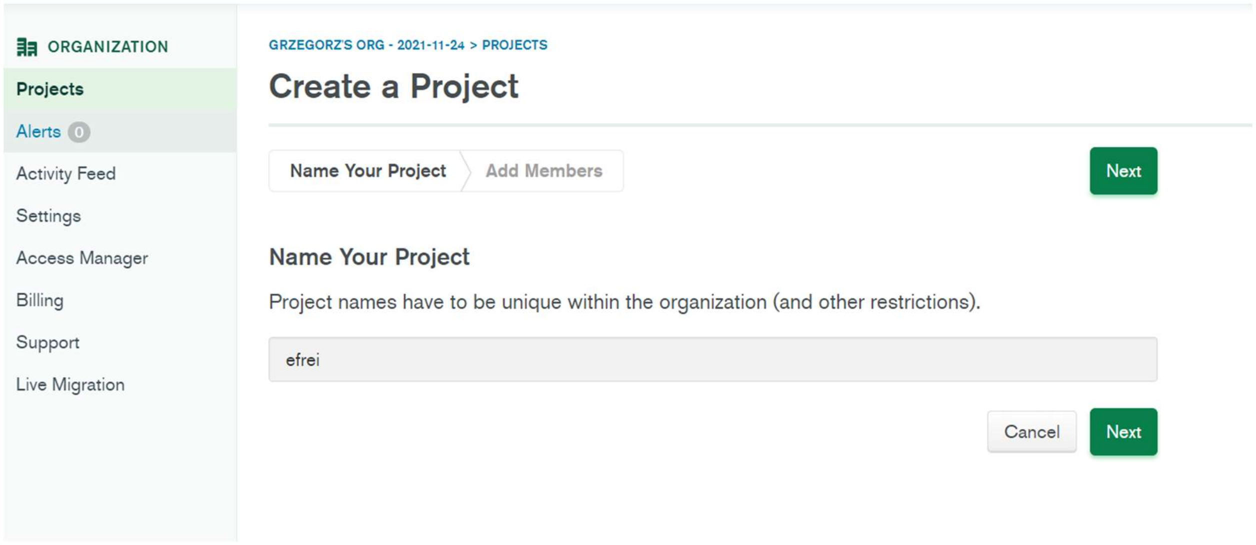
Plan Details
Atlas

Payment Method
Not added yet.

Billing Profile
Organization Created
11/24/21 - 11:40:43 AM

Available Credits

Select project and create new project:



GRZEGORZ'S ORG - 2021-11-24 > PROJECTS

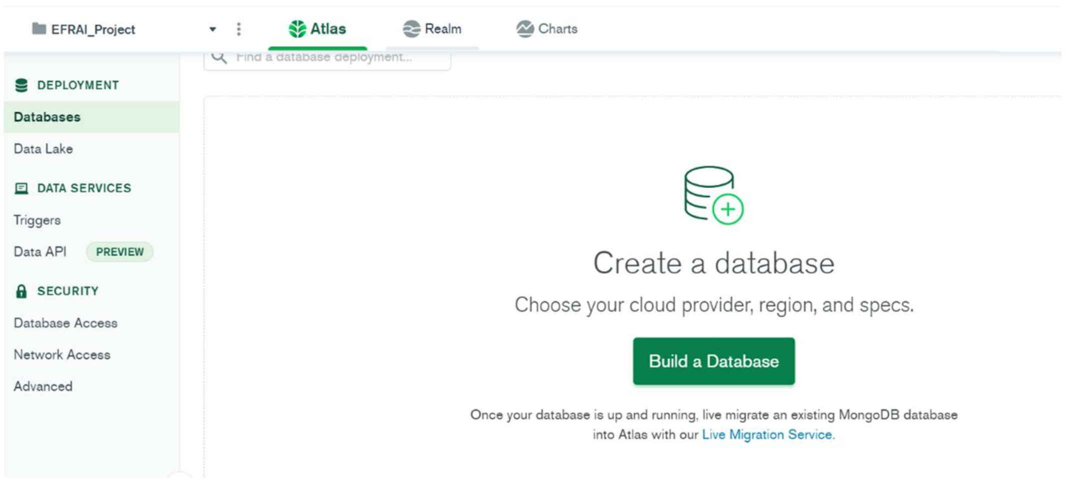
Create a Project

Name Your Project

Project names have to be unique within the organization (and other restrictions).

efrei

And on the next screen select Build Database.


EFRAI_Project

Atlas

Realm

Charts

Find a database deployment...

DEPLOYMENT

Databases

Data Lake

DATA SERVICES

Triggers

Data API PREVIEW

SECURITY

Database Access

Network Access

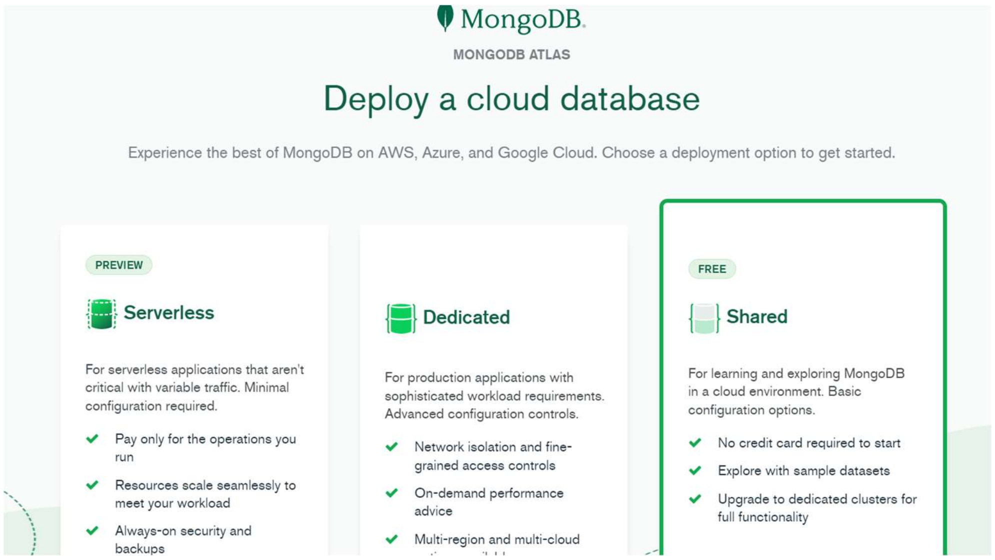
Advanced

Create a database

Choose your cloud provider, region, and specs.

Build a Database

Once your database is up and running, live migrate an existing MongoDB database into Atlas with our Live Migration Service.


MongoDB.

MONGODB ATLAS

Deploy a cloud database

Experience the best of MongoDB on AWS, Azure, and Google Cloud. Choose a deployment option to get started.

PREVIEW

Serverless

For serverless applications that aren't critical with variable traffic. Minimal configuration required.

- Pay only for the operations you run
- Resources scale seamlessly to meet your workload
- Always-on security and backups

Dedicated

For production applications with sophisticated workload requirements. Advanced configuration controls.

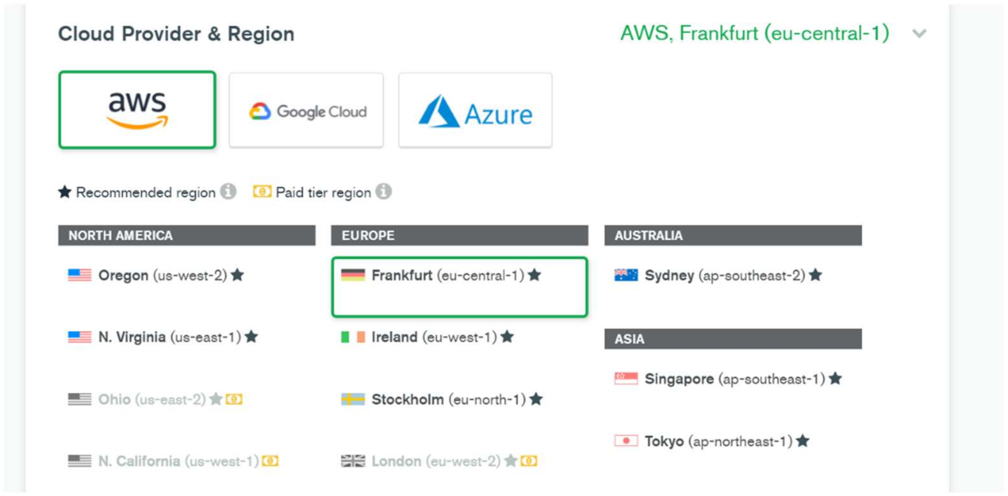
- Network isolation and fine-grained access controls
- On-demand performance advice
- Multi-region and multi-cloud

FREE

Shared

For learning and exploring MongoDB in a cloud environment. Basic configuration options.

- No credit card required to start
- Explore with sample datasets
- Upgrade to dedicated clusters for full functionality


Cloud Provider & Region

AWS, Frankfurt (eu-central-1)

aws | Google Cloud | Azure

★ Recommended region ⓘ Paid tier region ⓘ

NORTH AMERICA	EUROPE	AUSTRALIA
Oregon (us-west-2) ★	Frankfurt (eu-central-1) ★	Sydney (ap-southeast-2) ★
N. Virginia (us-east-1) ★	Ireland (eu-west-1) ★	ASIA
Ohio (us-east-2) ★	Stockholm (eu-north-1) ★	Singapore (ap-southeast-1) ★
N. California (us-west-1)	London (eu-west-2) ★	Tokyo (ap-northeast-1) ★

Create a new cluster:

• Click the Create a cluster button under Shared Clusters. This should be the only free option.

• In the Cloud Provider & Region dropdown, leave this as the default (typically AWS).

• In the Cluster Tier dropdown, leave this as the default, M0 Sandbox (Shared RAM, 512 MB Storage).

• In the Cluster Name dropdown, you can give your cluster a name, or leave it as the default, Cluster 0.

• Click the green Create Cluster button at the bottom of the screen.

• You should now see the message Your cluster is being created. New clusters take between 1-3 minutes to provision. Wait until the cluster is created before going to the next step.

Create a new user for the database

• On the left side of screen, click on Database Access.

• Click the green Add New Database User button.

• In the modal, enter a new username and password.

• Under Database User Privileges, leave this as the default option, Read and write to any database.

• Click the Add User button to create your new user.

Allow access from all IP addresses

• On the left side of the screen, click on Network Access.

• Click the green Add IP Address button.

• In the modal, click the ALLOW ACCESS FROM ANYWHERE button. You should see 0.0.0.0/0 in the Access List Entry entry field.

• Click the green Confirm button.

Connect to your cluster

• Click on the green Get Started button in the bottom left of your screen should now show you the final step, Connect to your cluster, click on it.

• On the left side of the screen, click on Clusters.

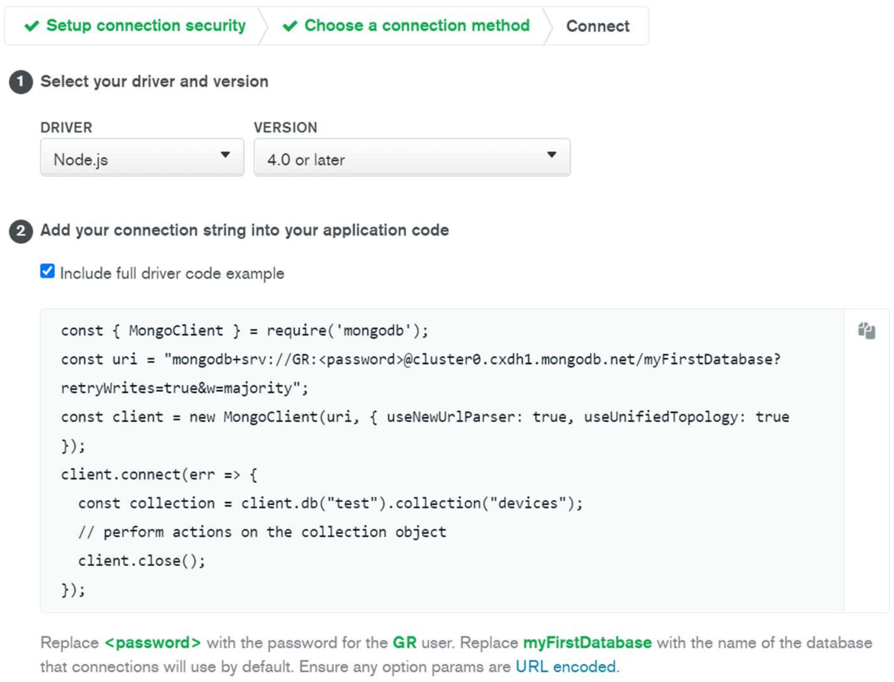
• Click the Connect button for your cluster.

• In the popup modal, click on Connect your application.

• You should see the URI you'll use to connect to your database similar to this: mongodb+srv://<username>:<password>@<cluster-name>.mongodb.net/<db-name>?retryWrites=true&w=majority.

• Click the Copy button to copy your URI to your clipboard.

Notice that the <username>, <cluster-name>, and <db-name> fields of URI you copied are already filled out for you. All you need to do is replace the <password> field with the one you created in the previous step.


✓ Setup connection security
✓ Choose a connection method
Connect
1 Select your driver and version
DRIVER
VERSION
2 Add your connection string into your application code

DRIVER: Node.js
VERSION: 4.0 or later

☑ Include full driver code example

```
const { MongoClient } = require('mongodb');
const uri = "mongodb+srv://GR:<password>@cluster0.cxdh1.mongodb.net/myFirstDatabase?
retryWrites=true&w=majority";
const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true
});
client.connect(err => {
  const collection = client.db("test").collection("devices");
  // perform actions on the collection object
  client.close();
});
```

Replace **<password>** with the password for the **GR** user. Replace **myFirstDatabase** with the name of the database that connections will use by default. Ensure any option params are URL encoded.

And that's it — you now have the URI to add to your application and connect to your database. Keep this URI safe somewhere so you can use it later.

Now, when we have a MongoDb as a cloud hosted database we back to our server.js.

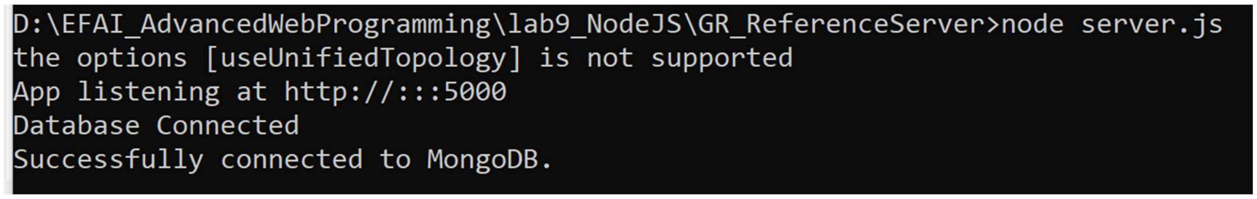
We write follow code:

```
// Configuring the database
const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

const uri =
"mongodb+srv://GR:0hIk9N4FuSEH2z0W@cluster0.cxdh1.mongodb.net/testDB?retryWrites=
true&w=majority";
mongoose.connect(uri,
    {
        useNewUrlParser: true,
        useUnifiedTopology: true
    }
).then(()=>{console.log('Database Connected');
           console.log("Successfully connected to MongoDB.");
  }).catch(err=>{
    console.log(err);
    console.log('Could not connect to MongoDB.');
});
```

And run server. As You see I get confirmation that server connect to MongoDB.



```
D:\EFAI_AdvancedWebProgramming\lab9_NodeJS\GR_ReferenceServer>node server.js
the options [useUnifiedTopology] is not supported
App listening at http://:::5000
Database Connected
Successfully connected to MongoDB.
```

Create a schema

The next step is to create a schema. Schema define the structure of the data stored in the database.The documents in the database will be inserted according to the schema we will define. I create new folder – models. In this catalog I create a new file and name it “comment.model.js”. Paste the following code in it.

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

const CommentSchema = mongoose.Schema({
    name: { type: String, required: true },
    title: { type: String, required: true },
    description: String,
    active: {
        type: Boolean,
        default: false
    }
});

module.exports = mongoose.model('Comment', CommentSchema);
```

This Mongoose Model represents tutorials collection in MongoDB database. These fields will be generated automatically for each Comment document: _id, name, title, description, active.

If you use this app with a front-end that needs id field instead of _id, you have to override toJSON method that map default object to a custom object. So the Mongoose model could be modified as following code:

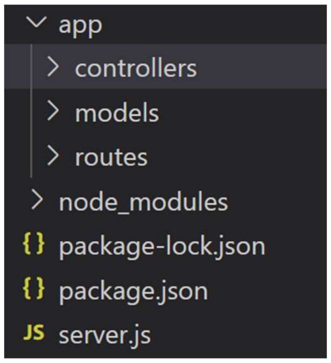
```
CommentSchema.method("toJSON", function() {
    const { __v, _id, ...object } = this.toObject();
    object.id = _id;
    return object;
  });
```

After finishing the steps above, we don't need to write CRUD functions, Mongoose Model supports all of them:

create a new Comment:	object.save()
find a Comment by id:	findById(id)
retrieve all Comments:	find()
update a Comment by id:	findByIdAndUpdate(id, data)
remove a Comment:	findByIdAndRemove(id)
remove all Comments:	deleteMany()

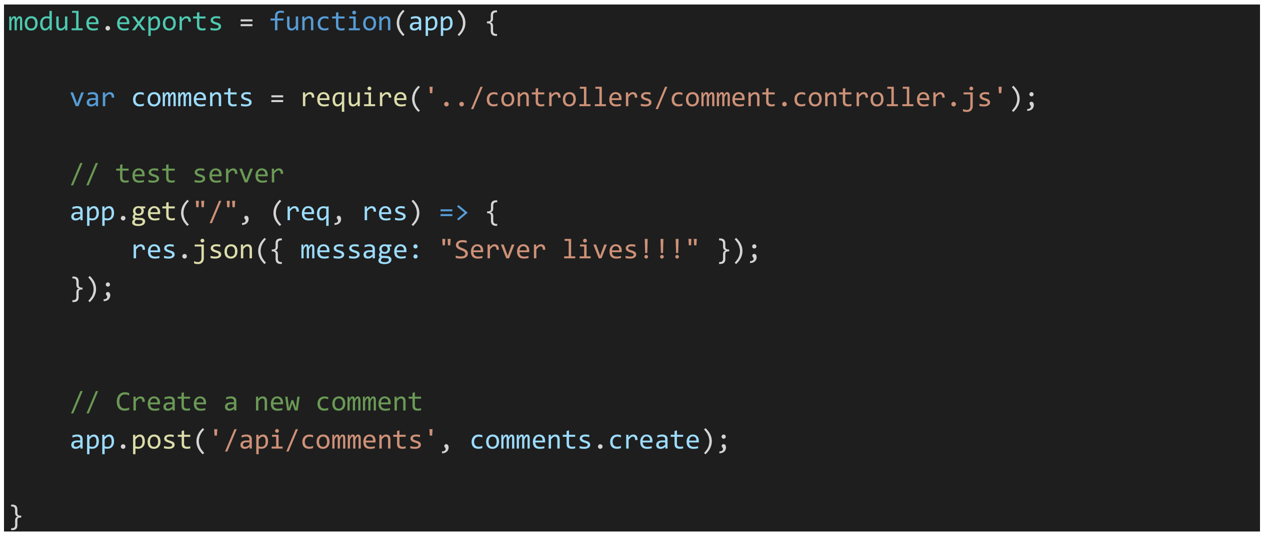
find all Comments by author name: find({ name: { $regex: new RegExp(title), $options: “i” } })

These functions will be used in our Controller.



Define a routing

Here, we can create a new folder (routes) with new file called comment.routes.js which will contains our Express routes. This file defines the endpoints for our app. Routes is used for handling the client request. We need to set routes for adding a comment, updating comment (changing hide or show status), fetching created comments and delete comment.



```
module.exports = function(app) {

    var comments = require('../controllers/comment.controller.js');

    // test server
    app.get("/", (req, res) => {
        res.json({ message: "Server lives!!!" });
    });


    // Create a new comment
    app.post('/api/comments', comments.create);

}
```

Above I show you part of routing file. I define for post operation /api/comments route and define route handler. This method will accept the endpoint of the route, and the route handler to define how data should be write to the database and which data sent to the client.

Exercise 1.

Define all routes for our Comments App. I think description from last pages about Mongoose Model allow you to define the routes politics. Use RestAPI conventions.

We sholud add routes file to server.js file.

```
require('./app/routes/comment.routes.js')(app);
```

Define a controler

Inside **controllers** folder, let's create *comment.controller.js* with these CRUD functions:

- create
- findAll
- findOne
- update
- delete
- deleteAll
- findByName

Exercise 2.

Let each of methods return for request only simple, statics text. For example like this:

`res.json({ message: "Send from delete method" }).` For another method write simple message with name using method. Use POSTman to test server (how routing and controller works).

Exercise 3.

Define controller methods by using Moongose model. I give you code from another projects as a template.

```
const Customer = require('../models/customer.model.js');

// POST a Customer
exports.create = (req, res) => {
    // Create a Customer
    const customer = new Customer({
        name: req.body.name,
        age: req.body.age
    });

    // Save a Customer in the MongoDB
    customer.save()
    .then(data => {
        res.send(data);
    }).catch(err => {
        res.status(500).send({
            message: err.message
        });
    });
};

// FETCH all Customers
exports.findAll = (req, res) => {
    Customer.find()
    .then(customers => {
        res.send(customers);
    }).catch(err => {
        res.status(500).send({
            message: err.message
        });
    });
};

// FIND a Customer
exports.findOne = (req, res) => {
    Customer.findById(req.params.customerId)
    .then(customer => {
        if(!customer) {
            return res.status(404).send({
                message: "Customer not found with id " + 
req.params.customerId
            });
        }
        res.send(customer);
    }).catch(err => {
        if(err.kind === 'ObjectId') {
            return res.status(404).send({
                message: "Customer not found with id " + 
req.params.customerId
            });
        }
        return res.status(500).send({
            message: "Error retrieving Customer with id " + 
req.params.customerId
        });
    });
};

exports.findByAge = (req, res) => {
       Customer.find({ age: req.params.age})
                     .then(
                            customers => {
                                   res.send(customers)
                            }
                     )
                     .catch(err => {
                            res.status(500).send("Error -> " + err);
                     })
}

// UPDATE a Customer
exports.update = (req, res) => {
    // Find customer and update it
    Customer.findOneAndUpdate({ _id: req.params.customerId }, {
        name: req.body.name,
              age: req.body.age,
              active: req.body.active
    }, {new: true})
    .then(customer => {
        if(!customer) {
            return res.status(404).send({
                message: "Customer not found with id " +
req.params.customerId
            });
        }
        res.send(customer);
    }).catch(err => {
        if(err.kind === 'ObjectId') {
            return res.status(404).send({
                message: "Customer not found with id " +
req.params.customerId
            });
        }
        return res.status(500).send({
            message: "Error updating customer with id " +
req.params.customerId
        });
    });
};

// DELETE a Customer
exports.delete = (req, res) => {
    Customer.findByIdAndRemove(req.params.customerId)
    .then(customer => {
        if(!customer) {
            return res.status(404).send({
                message: "Customer not found with id " +
req.params.customerId
            });
        }
        res.send({message: "Customer deleted successfully!"});
    }).catch(err => {
        if(err.kind === 'ObjectId' || err.name === 'NotFound') {
            return res.status(404).send({
                message: "Customer not found with id " +
req.params.customerId
            });
        }
        return res.status(500).send({
            message: "Could not delete customer with id " +
req.params.customerId
        });
    });
};
```

Finishing exercise 3 you finish creating web server. Now using POSTman to test your server.

Exercise 4.

Now we have web server. Back to your frontend app and update you FakeDataService. Now you have to store data in MongoDB and read data from server. Use fetch API or Axios to communicate between Frontend and Server.