Express + Mongoose

Learning Objectives:

- Learning what Mongoose is
- Learning how to Install mongoose with npm
- Learning how to connect mongoose to a database
- Learning how to create a Mongoose Schema and Model
- Learning how to execute basic queries with Mongoose objects
- Understanding the basic syntax for writing promises with Mongoose methods

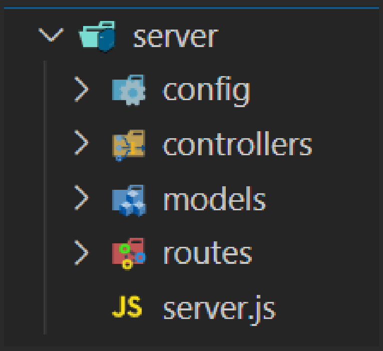
Mongoose

Now that we understand our MongoDB basics, let's connect it to a project and see it in action. The most popular way of using MongoDB with Node and Express is with a library called **Mongoose**. Mongoose simplifies making MongoDB queries with its own library of methods. This means that we can connect mongoose directly to a MongoDB database and it will allow us to give more structure to our data with the addition of models and schemas.

Mongoose will act as a layer between our application and our database enabling us to do things like validate and run complex queries more effectively.

1. Installing Mongoose

Start by creating a folder for your project (labeled as `server` in the image below) and create both the `server.js` file and the following folder structure inside of it:



- server - This is your backend server / project folder and will hold all server related files
  - config - will handle the database configuration and connection
  - controllers - will hold all logic for each model (i.e creating, updating, etc...)
  - models - will hold all the schemas
  - routes - will handle all of our routes for each model
  - server.js - will handle all the server logic with express

Once you have created the `server.js` file and the folders, open a new terminal window and navigate into your project folder (labeled as `server` in the image above) and install the server dependencies by running:

```
npm init -y
npm install mongoose express
```

NOTE: Require Mongoose

In any file that will use the Mongoose library, you will need to be sure to `require` it at the top of the file similar to this:

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

2. Connecting to MongoDB with Mongoose

In vs-code, navigate to the config folder where you will need to create the `mongoose.config.js` file. This is where we use mongoose to connect to MongoDb. Mongoose has a super convenient connect method -- `mongoose.connect`!

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

mongoose.connect('mongodb://127.0.0.1:27017/name_of_your_DB', {
    useNewUrlParser: true,
    useUnifiedTopology: true
})
    .then(() => console.log('Established a connection to the database'))
    .catch(err => console.log('Something went wrong when connecting to the database ', 
err));
```

Note: The useNewUrlParser and useUnifiedTopology are options we pass to get rid of deprecation messages in our terminal.

Note: If you connect to a database that doesn't exist, Mongoose will create the DB for you as soon as you create your first document!

3. Create your Mongoose Schema and Model

Mongoose provides more structure to MongoDB by adding schemas that we can create that turn into models for our collections. These models specify keys, types, and even validations for documents in a specific collection. Mongoose also handles appropriate naming for us when it communicates with MongoDB!

In vs-code, navigate your way to the models folder and create the `user.model.js` file in the models folder where we create a User collection using mongoose. Remember, we need to import mongoose using the require statement at the top of the file.

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

const UserSchema = new mongoose.Schema({
    name: {
        type: String
    },
    age: {
        type: Number
    }
});

const User = mongoose.model('User', UserSchema);

module.exports = User;
```

Let's break it down. The **mongoose.model()** method is the most important in this case. Its job is to take a blueprint object and, in turn, create the necessary database collection out of the model. We get this blueprint by making a new schema instance from the mongoose.Schema() object constructor. Notice how the **mongoose.Schema()** method takes an object as its parameter? The structure of that object is how each new document put into the collection will be formatted. You can learn more about the other Schema Types here.

We then create a User variable to export and set it to the returned value of the **mongoose.model()** function: a model object is created using the singular version of the collection name (`'User'`) and the mongoose schema (`UserSchema`). This model will be used to enable all our needed CRUD functionality. Exporting the User variable will allow us to import and use the User model in any file we choose.

NOTE: After we create our first document using this model, we will find a lowercase, plural version of the collection name in our database. In this case, "users"

4. Use the Mongoose Models to Create / Retrieve / Update / Destroy

Navigate your way into the controllers folder where you will create the `user.controller.js` file that will house all of our logic for creating, retrieving, updating, and deleting users. Notice at the top of the file, we do not have a `require("mongoose")` statement. Instead we have a `require("../models/user.model")` statement which is importing the User variable that we exported from the `user.model.js` file. In our controller file, we export different functions that perform the basic CRUD operations using our User model.

```
const User = require('../models/user.model');

module.exports.findAllUsers = (req, res) => {
    User.find()
        .then((allDaUsers) => {
            res.json({ users: allDaUsers })
        })
        .catch((err) => {
            res.json({ message: 'Something went wrong', error: err })
        });
}

module.exports.findOneSingleUser = (req, res) => {
    User.findOne({ _id: req.params.id })
        .then(oneSingleUser => {
            res.json({ user: oneSingleUser })
        })
        .catch((err) => {
            res.json({ message: 'Something went wrong', error: err })
        });}

module.exports.createNewUser = (req, res) => {
    User.create(req.body)
        .then(newlyCreatedUser => {
            res.json({ user: newlyCreatedUser })
        })
        .catch((err) => {
            res.json({ message: 'Something went wrong', error: err })
        });}

module.exports.updateExistingUser = (req, res) => {
    User.findOneAndUpdate(
        { _id: req.params.id },
        req.body,
        { new: true, runValidators: true }
    )
        .then(updatedUser => {
            res.json({ user: updatedUser })
        })
        .catch((err) => {
            res.json({ message: 'Something went wrong', error: err })
        });}

module.exports.deleteAnExistingUser = (req, res) => {
    User.deleteOne({ _id: req.params.id })
        .then(result => {
            res.json({ result: result })
        })
        .catch((err) => {
            res.json({ message: 'Something went wrong', error: err })
        });}
```

A couple key points:

- User is a constructor function which can create new user objects and also has other methods that will talk to the database!
    - the .then() will only execute upon successfully inserting data in the database
    - the .catch() will execute only if there is an error.

5. Routing

Navigate your way into the routes folder where you will create the `user.routes.js` file that will be responsible for all of our routes dealing with the user model. Notice at the top of the file, this time, we have a `require("../controllers/user.controller")` statement which is importing everything we exported from the controller file.

```
const UserController = require('../controllers/user.controller');

module.exports = app => {
    app.get('/api/users', UserController.findAllUsers);
    app.get('/api/users/:id', UserController.findOneSingleUser);
    app.patch('/api/users/:id', UserController.updateExistingUser);
    app.post('/api/users', UserController.createNewUser);
    app.delete('/api/users/:id', UserController.deleteAnExistingUser);
}
```

NOTE: The order of these routes matter! If you have a route that uses a wildcard (like `:id`) before a path with a specific name, the wildcard route will always run. Move specific routes to the top to ensure they are followed.

For example, if you wanted to create a get route with a specific path, it would have to go before your get route for a single document. The following would never make it to the `allusers` route because the `:id` route would also match the string `allusers`:

```
const UserController = require('../controllers/user.controller');

module.exports = app => {
    app.get('/api/users/:id', UserController.findOneSingleUser);
    app.get('/api/users/allusers', UserController.findAllUsers);
}
```

6. Server

Last but not least is our server.js file. Because we modularized our files from the start, this allows our `server.js` file to contain only a few lines of code, allows us to be able to easily expand our app, and helps us keep organized. Take a moment and look over the `server.js` file and familiarize yourself with what's happening.

```
const express = require("express");
const app = express();

require("./config/mongoose.config");

app.use(express.json(), express.urlencoded({ extended: true }));

const AllMyUserRoutes = require("./routes/user.routes");
AllMyUserRoutes(app);

app.listen(8000, () => console.log("The server is all fired up on port 8000"));
```

There is a lot that is happening in the code above so take a couple minutes to review and make sure that you understand everything that is going on. When you feel comfortable to move on, you can go ahead and run your server using Nodemon in your terminal:

```
nodemon server.js
```

NOTE: If you receive the following warning when running your server, you can safely ignore it:

```
DeprecationWarning: Listening to events on the Db class has been deprecated and will be
removed in the next major version
```

Debug

If it didn't work make sure the following things are done:

1. Make sure your *MongoDB server* is running (the *'mongod'* command)
2. Make sure your post data matches the data that you are inserting into the database