# Week 03 - Node JS + Express + Mongoose

This material can be found in this [link](https://codeberg.org/kaduardo/shu-sad-javascript/src/branch/main/week03-nodejs-express-mongoose/).

## 1. Introduction

This week you will start playing around with NodeJS, Express and Mongoose.

In this lab we are going to build a simple petshop application using Express framework for programming the backend, mongodb database with mongoose and pug to create our pages. The tutorial starts with some “recipe” material, which you can follow along, before asking you to dive into the deep end and refactor the code.

We also have other introductory exercises saved in the [repository](https://codeberg.org/kaduardo/shu-sad-javascript/src/branch/main/week03-nodejs-express-mongoose/) that can be followed if you feel you need it.

1. [An introduciton to Node.JS](intro-nodejs.md) - this is a gentle introduction to node.js designed for those that never used it before.
2. [An introduction to MongoDB](intro-mongodb.md) - these show how you can run mongodb server in your local machine.

Let’s then start with nodejs and express.

You need a place to hold your work. You should make a directory in which your work can be stored. Open a terminal and change to this new directory. These instruction will assume the name of your main project folder is petshop and that you are starting your project from scratch.

## 2. Set-up the node.js application

let’s start working on our nodejs application. Since we will use express the first step is to install it in your machine. You can install it in your system by running the following command:

npm install -g express-generator

Now you can use express to initialise your node.js application. From inside the petshop folder run the following command:

express --view=pug backend

This will create a new folder called backend, resulting in your code being saved in the petshop/backend folder. You can check your current application by changing into the backend directory, installing the necessary dependencies and starting the application with the commands below:

cd backend  
  
npm install  
  
npm start

Now open your browser and navigate to <http://localhost:3000>.

## 3. Install modules/dependencies

You need a way to extract data from the post. In express this is done easily using the body-parser and multer packages. At the same time we’ll install an HTML pre-processor with which we’ll build a UI and a package that monitors and restarts node when our codebase changes. We’ll also install the library to deal with the mongodb database. In the terminal run the following commands from inside the backend directory:

npm install --save body-parser  
npm install --save multer  
npm install --save cors  
npm install --save mongoose  
npm install --save pug  
npm install -g nodemon

Or add them to package.json as dependencies. Nodemon watches for changes in your project and automatically stops and restarts the server when it detects changed files.

Now you need to instruct npm to download and locally install the indicated dependencies. Do this by running the command:

npm update  
npm install

Start your application by executing one of these commands in a terminal

npm start  
nodemon start

Navigate to <http://localhost:3000> and you should see a welcome message.

## 4. Creating the routes

We’ll start by setting up and testing some routes. If working on your own machine you may find the Chrome plugin Postman or the Firefox extension Rest Easy useful to test different HTTP methods.

Begin by creating a new routing file called animal.routes.js in the routes directory of your application. Add the code below to this file.

var express = require('express');  
var router = express.Router();  
   
// GET at the root just to demonstrate  
router.get('/', function(req, res, next) {  
 res.send('got a GET request at /');  
});  
  
// GET list of animals to show that we're up and running  
router.get('/animals', function(req, res, next) {  
 res.send('Got a GET request at /animals');  
});  
   
// accept POST request and add a new animal to the db  
//router.post('/animals', upload.array(), function (req, res) {  
// res.send('Got a POST request at /animals');  
//});  
   
// accept PUT request at /animals  
router.put('/animals', function (req, res) {  
 res.send('Got a PUT request at /animals');  
});  
   
   
// accept DELETE request at /animals  
router.delete('/animals', function (req, res) {  
 res.send('Got a DELETE request at /animals');  
});  
   
module.exports = router;

You’ll need to edit your app.js file to require your new animal.routes.js module. Assign it to a variable called animalsRouter then add a new use statement:

// Assigning the router to a local variable  
var animalsRouter = require('./routes/animal.routes');  
  
//...  
  
// Indicating the path this router will be responsible for  
app.use('/petshop', animalsRouter);

Restart your server and test the routes you now have. If you can, try using different HTTP methods, too (if you’ve got postman installed you can use it). A few examples:

* <http://localhost:3000/>
* <http://localhost:3000/petshop>
* <http://localhost:3000/petshop/animals>
* <http://localhost:3000/petshop/animals?n=2>
* <http://localhost:3000/petshop/animals/dog>
* <http://localhost:3000/animals/>

Reflect on what is happening. Can you identify which function is responsible for answering each path? Include a few console.log to track how the request moves through your code. Are you able to add a new path?

## 5. Making the pages using Pug

In a default express installation HTML pages are created from a templating language called Pug. If you look in the views folder of your project you’ll see three default templates.

In app.js you need to make sure that the **view engine** is set to pug.

Now create a new file in views called animals.pug. This is where you’ll place your template code. You can use the following code for your template:

html  
 head  
 title!= title  
 body  
 h1!= message

Notice how the template uses two variables (title and message). We need to change the /animals route so that data that is needed in the template is passed in. Modify the routes/animal.routes.js router with:

router.get('/animals', function(req, res, next) {  
 res.render('animals',   
 {title: 'Pet store',  
 message: 'Hello World! from the <tt>animals</tt> router'}  
 );  
});

Test your new view. You should see some nice HTML returned. Check the page source to confirm it.

Here is an example Pug template with a simple table of data. In the finished application the table will be generated from a database query and interpolated into the table by the engine. The dash and space before the variable are significant. You need them. The indentation really matters - use spaces not tabs. Save the following template as animals.pug (you need the leading dash and space before the variable)

- var dog = { name:'Logan', species:'Dog', breed:'Border Collie', age:'2.5', colour:'Black with white, blue and tan head' }  
html  
 head  
 title!= title  
 body  
 h1 !{message}  
 table(border='1')  
 thead  
 th Name  
 th Species  
 th Breed  
 th Age  
 th Colour or markings  
 tbody  
 td #{dog.name}  
 td #{dog.species}  
 td #{dog.breed}  
 td #{dog.age}  
 td #{dog.colour}

Test your new view. You should be able to identify the HTML table generated from the variables in the template.

Try this to demonstrate how to handle multiple rows of data:

- var animals = [{ name:'Logan', species:'Dog', breed:'Border Collie', age:'5', colour:'Black with white, blue and tan head' }, {name:'Archie', species:'Dog', breed:'Mongrel', age:'15.5', colour:'White' }]  
html  
 head  
 title!= title  
 body  
 h1 !{message}  
 table(border='1')  
 thead  
 th Name  
 th Species  
 th Breed  
 th Age  
 th Colour or markings  
 tbody  
 each pet in animals  
 tr  
 td #{pet.name}  
 td #{pet.species}  
 td #{pet.breed}  
 td #{pet.age}  
 td #{pet.colour}

Try to pass the data to the template from the GET /animals route handler. You’ll need to remove it from the template itself. Do this by creating an array in the route handler and passing that to the template as the parameter animals.

## 6. Adding a form to your pug page

We are going to modify our template to create a form you can use to add new pets. The fields should be named name, species, breed, age and colour. We are going to use the HTTP POST verb and the action should be /petshop/animals.

Here is an example of the main form code using pug for your reference (Using the documentation for Pug at <https://pugjs.org/api/getting-started.html> and <https://medium.com/@nima.2004hkh/create-your-first-login-page-with-exprerssjs-pug-f42250229486>):

h1 Enter the details of a new pet  
 form(action="/petshop/animals" method="POST")  
 table(border='1')  
 thead  
 th Name  
 th Species  
 th Breed  
 th Age  
 th Colour or markings  
 tbody  
 td  
 input(type="text", name="name")  
 td  
 input(type="text", name="species")  
 td  
 input(type="text", name="breed")  
 td  
 input(type="text", name="age")  
 td  
 input(type="text", name="colour")  
 tr  
 td  
 button(type="submit") Submit

You should have a page that looks something like this: 

Now you’ll need a route to handle the data from this form. Using the HTTP POST method and /petshop/animals is a sensible and RESTful URL for the endpoint. Uncomment the supplied router.post function, presented below as reference.

router.post('/animals', upload.array(), function (req, res) {  
 res.send('Got a POST request at /animals');  
});

Now you need to add the new packages to the router. The following code makes them available in animal.routes.js and creates an instance of multer which we will use to extract form data from the POST:

var bodyParser = require('body-parser');  
var multer = require('multer');   
var upload = multer();

Now change the POST handler to look like this:

router.post('/animals', upload.array(), function (req, res) {  
 console.log('Received a POST request and animals');  
 res.send('Post handler at /animals received '+ req.body.name   
 +' '+ req.body.breed);  
});

You can now test your new handler by sending a POST request to <http://localhost:3000/petshop/animals> where the body is a pet in json format. You can do that using Postman or any of the plugins you chose to use. Here is one example of the json format to be used.

{"name":"Scooby Doo", "species":"Dog", "breed":"Great Dane", "age":"51", "colour":"Browinsh with black spots"}

This is one example using the cURL tool (linux based machines):

curl -X POST -H "Content-Type: application/json" -d '{"name":"Scooby Doo", "species":"Dog", "breed":"Great Dane", "age":"51", "colour":"Browinsh with black spots"}' http://localhost:3000/petshop/animals

Now that your handler is able to receive POST requests and extract information from the body we can pass this information forward. Let’s add the newly arrived data to the local array, and have the POST handler reload the page with the new data added to the table at the top each time that it receives a new pet. Do this by using the response object’s redirect method.

Your POST handler will look list this:

// accept POST request and add a new pet to the db  
router.post('/animals', upload.array(), function (req, res) {  
 console.log('Received a POST request and animals');  
 animals.push(req.body);  
 res.redirect('/petshop/animals');  
});

Now try your application in the browser. Start by accessing the main list page: <http://localhost:3000/petshop/animals>, then add a few entries using the form. Notice in the console the POST request followed by a GET request.

## 7. Creating the database

We’re now going to use a mongo database to store information about our pets.

If you haven’t done so you should now download and run the [MongoDB Server](intro-mongodb.md).

Now that you mongodb server is up and running let’s create a database and populate with some data. Using the MongoDB shell create a new database called petshopdb and create a new collection called animals. You can use the commands below for doing that from inside the mongosh shell client.

use petshopdb   
  
db.createCollection("animals")  
  
db.animals.insertMany([  
{ 'name': 'Logan', 'species': 'dog', 'breed': 'Border Collie', 'age': 5, 'colour': 'tricolour' },   
{ 'name': 'Ralph', 'species': 'dog', 'breed': 'Cocker Spaniel', 'age': 2, 'colour': 'golden' },   
{ 'name': 'Mallyan', 'species': 'cat', 'breed': 'British Short Hair', 'age': 22, 'colour': 'black' }   
])

Try to display all of the data in the MongoDB Shell client (e.g., db.animals.find()).

Now you are ready to start coding.

## 8. Defining the models

To better organise our code the database connection details will be saved in a separate file. Create a directory named config in your project backend folder and then a file named db.config.js with the following content:

module.exports = {  
 url: "mongodb://127.0.0.1:27017/petshopdb"  
};

Create a directory called models in your project backend folder, and inside this directory create file named index.js file with the following content:

const dbConfig = require("../config/db.config.js");  
   
const mongoose = require("mongoose");  
mongoose.Promise = global.Promise;  
   
const db = {};  
db.mongoose = mongoose;  
db.url = dbConfig.url;  
db.animals = require("./animal.model.js")(mongoose);  
   
module.exports = db;

Let’s create the model. Create a file name animal.model.js inside the model folder. Use the following content for this file:

module.exports = mongoose => {  
 var Animal = mongoose.model(  
 "animal",  
 mongoose.Schema(  
 {  
 name: String,  
 species: String,  
 breed: String,  
 age: Number,  
 colour: String  
 }  
 )  
 );  
 return Animal;  
};

## 9. Creating request handlers (controllers)

Before we were combining the controller code within the routes. We are now going to separate these following best practices for code organisation.

Create a controllers folder within your project folder, and create animal.controller.js file with the following content:

const db = require("../models");  
const Animal = db.animals;  
   
// Create and Save a new Animal  
exports.create = (req, res) => {  
   
};  
   
// Retrieve all Animals from the database.  
exports.findAll = (req, res) => {  
   
};  
   
// Find a single Animal with an id  
exports.findOne = (req, res) => {  
   
};  
   
// Update a Animal by the id in the request  
exports.update = (req, res) => {  
   
};  
   
// Delete a Animal with the specified id in the request  
exports.delete = (req, res) => {  
   
};  
   
// Delete all Animal from the database.  
exports.deleteAll = (req, res) => {  
   
};

Notice how these functions abstract away from the actual HTTP method used in the request. This is handled by the router. Let’s now implement some of those functions.

This is the code to create and save a new Animal in the database: First we need to modify the POST handler to extract the data from the HTTP request and then save it in the database. Use the following code for doing that (if the fields in your form have different names you’ll need to do some editing):

// Create and Save a new Animal  
exports.create = (req, res) => {  
 // Validate request  
 if (!req.body.name) {  
 res.status(400).send({ message: "Content can not be empty!" });  
 return;  
 }  
   
 // Create a Animal model object  
 const animal = new Animal({  
 name: req.body.name,  
 species: req.body.species,  
 breed: req.body.breed,  
 age: req.body.age,  
 colour: req.body.colour  
 });  
   
 // Save Animal in the database  
 animal  
 .save()  
 .then(data => {  
 console.log("Animal saved in the database: " + data);  
 res.redirect('/petshop/animals');  
 })  
 .catch(err => {  
 res.status(500).send({  
 message:  
 err.message || "Some error occurred while creating the Animal."  
 });  
 });  
};

This is the code to retrieve all animals from the database (with support for conditions) and format it neatly using our previous pug template:

// Retrieve all Animals from the database.  
exports.findAll = (req, res) => {  
 const name = req.query.name;  
 //We use req.query.name to get query string from the Request and consider it as condition for findAll() method.  
 var condition = name ? { name: { $regex: new RegExp(name), $options: "i" } } : {};  
 Animal  
 .find(condition)  
 .then(data => {  
 res.render('animals',  
 {title: 'Pet Store',  
 animals: data});  
 })  
 .catch(err => {  
 res.status(500).send({  
 message:  
 err.message || "Some error occurred while retrieving Animals."  
 });  
 });  
};

## 10. Creating routers

Now that we created our controllers we need to update our routes. Within the routes folder of your project, we are going to modify the animal.routes.js to include the new controller and then change the different routes to use the functions provided by the controller. This is the new anime.routes.js file:

var express = require('express');  
var router = express.Router();  
   
//Require controller  
var animalController = require('../controllers/animal.controller');  
  
// GET at the root returns a welcome message in json  
router.get('/', function(req, res, next) {  
 res.json({message: "Welcome to the petshop api."});  
});  
   
// Create a new animal  
router.post("/animals/", animalController.create);  
   
// Retrieve all animals  
router.get("/animals/", animalController.findAll);  
   
// Retrieve a single animal with id  
router.get("/animals/:id", animalController.findOne);  
   
// Update a animal with id  
router.put("/animals/:id", animalController.update);  
   
// Delete a animal with id  
router.delete("/animals/:id", animalController.delete);  
   
// Delete all animals of the database  
router.delete("/animals/", animalController.deleteAll);  
   
module.exports = router;

## 11. Connecting to the database

Now we need to configure and create the connection to our database. The connection code will reside in the main file app.js. We also need to import the router into the main file. The following content shows the new app.js file after adding the database connection (feel free to modify it as you see fit for your project):

var createError = require('http-errors');  
var express = require('express');  
var path = require('path');  
var cookieParser = require('cookie-parser');  
var logger = require('morgan');  
  
// Adding extra modules  
var bodyParser = require("body-parser");  
var cors = require('cors');  
  
var indexRouter = require('./routes/index');  
var usersRouter = require('./routes/users');  
// The router for animals  
var animalsRouter = require('./routes/animal.routes');  
  
var app = express();  
  
// adding cors module  
app.use(cors());  
  
// view engine setup  
app.set('views', path.join(\_\_dirname, 'views'));  
app.set('view engine', 'pug');  
  
app.use(logger('dev'));  
app.use(express.json());  
app.use(express.urlencoded({ extended: false }));  
app.use(cookieParser());  
app.use(express.static(path.join(\_\_dirname, 'public')));  
  
app.use('/', indexRouter);  
app.use('/users', usersRouter);  
app.use('/petshop', animalsRouter);  
  
//Database connection code  
const db = require("./models");  
db.mongoose.connect(db.url, {  
 useNewUrlParser: true,  
 useUnifiedTopology: true  
}).then(() => {  
 console.log("Connected to the database!");  
}).catch(err => {  
 console.log("Cannot connect to the database!", err);  
 process.exit();  
});  
  
// catch 404 and forward to error handler  
app.use(function(req, res, next) {  
 next(createError(404));  
});  
  
// error handler  
app.use(function(err, req, res, next) {  
 // set locals, only providing error in development  
 res.locals.message = err.message;  
 res.locals.error = req.app.get('env') === 'development' ? err : {};  
  
 // render the error page  
 res.status(err.status || 500);  
 res.render('error');  
});  
   
module.exports = app;

Notice the following points:

* Importing of the router:

var animalRouter = require('./routes/animal.routes');

* Association of the router with the /petshop path:

app.use('/petshop', animalRouter);

* Code for controlling the connection to the database:

//Database connection code  
const db = require("./models");  
db.mongoose.connect(db.url, {  
 useNewUrlParser: true,  
 useUnifiedTopology: true  
}).then(() => {  
 console.log("Connected to the database!");  
}).catch(err => {  
 console.log("Cannot connect to the database!", err);  
 process.exit();  
});

Save the files and make sure your mongodb server is running. You should now be able to list all the animals in your database by accessing the following url: <http://localhost:3000/petshop/animals>. Play around with the form. Try adding a few animals to the database and check using the mongosh shell console if the animals are indeed saved in the database. What kind of output do you have in the node console (the one where you the node server)? Add a few console.log throuhgout your code to track the sequence in which functions are called.

Try the following URLs and make a note of what happens:

* <http://localhost:3000/petshop/>
* <http://localhost:3000/petshop/animals>
* <http://localhost:3000/>
* <http://localhost:3000/petshop/animals/2>
* <http://localhost:3000/petshop/animals/?name=test>

Replace *test* with the name of an animal saved in your database.

You can use a tool like Postman, SoapUI or cURL to send different HTTP requests (e.g., POST, PUT, DELETE) to your back-end.

## 11. Exercises

1. Create the rest of the controller functions. You can use this tutorial as basis for defining your database manipulation functions <https://bezkoder.com/node-express-mongodb-crud-rest-api/>
2. Update your front-end to fully implement all CRUD functions.

## References

* Mozilla Developer Network - Express web framework (Node.js/JavaScript) - https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\_Nodejs
* Bezkoder tutorials: <https://www.bezkoder.com/react-node-express-mongodb-mern-stack/>