Extra Notes - Node.js Todos API

- Dr Nick Hayward

Contents

node-todos-api - todos web app - intro

Combine Node.js, Mongoose, and MongoDB with a UI client-side to create a Todos app.

app - Mongoose setup

Mongoose helps create schema-based models for an application's data. e.g. define a schema for a user document &c.

It also helps with type casting, validation, structuring queries to MongoDB, and so on. Further details,

- mongoosejs.com/docs/guide.html

Install Mongoose,

```
npm i mongoose --save
```

app - initial app setup

Add a folder for the `server` to the root of the app, and then add a new file `server.js`.

This file will, effectively, act as the starter, a type of bootstrapping, for the app.

So, we can require Mongoose and connect to the DB in MongoDB. e.g.

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

mongoose.connect('mongodb://localhost:27017/NodeTodoApp');
```

app - create a model with Mongoose

Create a Mongoose model for data we want to store in MongoDB.

With Mongoose, we can specify a name for the model, and various properties we require for each document created from this model. e.g.

```
// specify model for Todo item
var Todo = mongoose.model('Todo', {
  // specify requirements for a property of a todo item
  text: {
    type: String
  },
  completed: {
    type: Boolean
  },
  completedAt: {
    type: Number
  }
});
```

app - use a model with Mongoose

We can now use our defined model to create documents we can store in MongoDB.

We use our model as a constructor to create a new document object, which we can then save to the connected DB. Each model requires the specified properties as well, e.g.

```
// create a new Todo item from the model
var todoItem = new Todo({
  text: "walk the Great Wall of China"
});

// save the object as a document in the DB - save returns a promise
todoItem.save().then((doc) => {
  console.log('todo item saved', doc);
}, (error) => {
  console.log('todo item not saved: ', error);
});
```

app - use validators, types, and defaults with Mongoose

Mongoose helps us specify default values for properties in models, check types &c., and easily validate data before it is saved to MongoDB.

Mongoose supports many built-in validators for booleans, numbers (min and max), strings (match, min and max length...) &c.

Further details at,

- mongoosejs.com/docs/validation.html

So, we might add a validator for properties in a model, e.g.

```
// specify model for Todo item
var Todo = mongoose.model('Todo', {
  // specify requirements for a property of a todo item
  text: {
    type: String,
    required: true, // text property is required to create a new document
    minlength: 1 // todo item must have at least 1 character
    trim: true // removes leading and trailing spaces
  }
  ...
});
```

These validators allow us to customise the text property for a todo item before it is saved to the DB. This property is set as required, has minimum character length of 1 to avoid empty text, and has all leading and trailing white space removed from the string. It will then validate successfully, and can be saved to the DB.

We can also add validators for other properties in a model. e.g. a `default` value for a given property

```
...
completed: {
  type: Boolean,
  default: false // specify default value
}
...
```

app - refactor code for mongoose and models

We can organise our app's code into logical groupings, including a new directory for database/store management files and setup. e.g. `server/dbms/moongoose-config.js`. This should include config and connection settings using Mongoose for the specified MongoDB.

This may then be required in the app's main `server.js` file. e.g.

```
// get mongoose property using ES6 destructuring - name of created local variable
will match the property of the object
var {mongoose} = require('./dbms/mongoose-config.js');
```

We can then do the same for other logic in the `server.js` file not required to setup and run the Express server. e.g. models, queries, &c.

So, we can now create some separate models for the app, including `todo-model.js` and `user-model.js`. These modules can then be required in `server.js`, e.g.

```
var {Todo} = require('./models/todo');
```

Using the new ES6 destructuring, we can create a new variable with the name of the property on the object returned from loading the specified custom module file. So, we have a variable called `Todo`, which is the property Todo of the object returned from the local file.

This is the whole point of the `module.exports` for the custom file, e.g.

```
// module export
module.exports = {
  Todo // ES6 shortcut for Todo: Todo
};
```

An object is returned for the `exports`, which contains a property of `Todo`. `Todo` references the variable Todo for the model in the custom file.

app - build a basic API - setup server

So, we can now start to build out a basic API for the app. For example, we might add routes for POST and GET, which will allow an app to query data, and then modify or add data.

We'll start by adding the required Express to the current app,

```
npm install express --save
```

and a module called `body-parser`, which allows us to send JSON to the server. In effect, `body-parser` takes the string body and parses it into a JS object.

```
npm install body-parser --save
```

These modules will then be required in the `server.js` file, and we can add the basic code for a server, e.g.

```
...
// create express app
var app = express();

// set port for server
app.listen(3030, () => {
  console.log('server started on port 3030...');
});
```

app - build a basic API - POST route

We can now add a POST route to allow our app to save data, effectively by posting that data to the specified data store.

For a POST route, e.g.

```
app.post('/todos', (req, res) => {

});
```

we need to configure the middleware for the `body-parser`, e.g.

```
app.use(bodyParser.json());
```

So, the body-parser module will parse the body into a JS object, which is then attached to the `req` object for the route. We can also check this relative to the POST route, for example

```
app.post('/todos', (req, res) => {
  console.log(req.body); // log body parsed by body-parser for the req object...
});
```

Then, we can create a new Todo item using the Todo model,

e.g.

```
app.post('/todos', (req, res) => {
  // create todo item from model
  var todo = new Todo({
    text: req.body.text // specify text for each todo item
  });

  todo.save().then((doc) => {
    res.send(doc); // send back to the saved document details
  }, (error) => {
    // send back errors...
    res.status(400).send(error); // send back error and status code for request...
  })
});
```

app - build a basic API - GET route

We can now add a GET route to find and return all todo items in the DB.

It follows a similar pattern to the previoud POST route,

```
// GET route for todo items
app.get('/todos', (req, res) => {
  Todo.find().then((todos) => { // promised resolved with all of the todos from the
db
    res.send({ //response - send data back from get route - all of the todos
      todos // add todos array to object - update and modify object as needed instead
of just sending array response...
    });
  }, (error) => { // error callback if error with promise
    res.status(400).send(error); // send back error and status code for request...
  });
});
```

We can then start the server,

```
node server/server.js
```

and then test the GET route in Postman, again using the local route of

- localhost:3030/todos

with a GET HTTP method for the request. Then press send and check the return from the `todos` route.

app - Mongoose query tests

There are many different options with Mongoose for querying data, more than simply `find`.

So, we can add a new directory for testing these queries, `mongoose-tests`, and then add a file for these queries, `mongoose-queries.js`.

We'll start by requiring the mongoose config file for connecting to MongoDB, and the model for a Todo. We'll also need an ObjectID from a document in the `todos` collection, which we can use for testing, e.g.

```
// require mongoose config
const {mongoose} = require('./../server/dbms/mongoose-config.js');
// require models - todo
const {Todo} = require('./../server/models/todo-model.js');

// specify test doc ID
var docID = '5979f0f0f3a968291ae5bf63'; // ObjectID for doc from collection
```

n.b. Mongoose is happy with strings as IDs, and they do not need to be converted as with MongoDB native driver.

We can then add some queries.

```
// find
Todo.find({
  _id: docID // mongoose will convert this string to an ObjectID
}).then((todos) => { // returns array of todo items
  console.log('all todo items - ', todos);
});

// find one - returns first match
Todo.findOne({
  completed: false
}).then((todo) => { // returns single item for first match
  console.log('single todo item - ', todo);
});
```

The first query will find all docs that match the specified ObjectID, whilst the second query will return the first document that matched the specified query.

If we're searching for a single document, the `findOne()` query may be preferable as it returns an object. The standard `find()` query returns an array containing the found documents. For a single document, this will still be an array with the required object for the document. The other benefit is that `findOne()` will return `null` for an empty query. `find()` will still return an array, but it will be empty for no result found.

We might also need to find by a specific, known ObjectID, e.g.

```
Todo.findById(docID).then((todo) => {
  console.log('single todo item by ID - ', todo);
});
```

app - Mongoose validation checking

We can start cursory testing of false, missing or error-prone ObjectIDs, and then use validation with Mongoose to handle such errors &c.

So, an empty result set will return an empty array for `find()`, and `null` for both `findOne()` and `findById()`. This means that a false or incorrect ObjectID will still return a result for the query. It will not handle the false or incorrect ObjectID as an error, but simply an empty return result.

We can, therefore, check for an empty result set for the query, e.g.

```
Todo.findById(id).then((todo) => {
  if (!todo) { // check for null return
    return console.log('Object ID not found...');
  }
  console.log('Todo ID not found - ', todo);
});
```

If we had an error where the ObjectID was invalid, perhaps there were too many numbers to the ID, we could add a `catch` method to `findById()`.

This would catch the error that the ID is not in the specified collection, and that the ID itself is not a valid format.

e.g.

```
Todo.findById(docID).then((todo) => {
  if (!todo) { // check for null return for query...
    return console.log("specified ID has not been found...");
  }
  console.log('single todo item by ID - ', todo);
}).catch((error) => console.log(error, 'Specified ID is not valid - try another
one...')); // catch validation error with ID error &c. i.e. ID is invalid - perhaps
too long...
```

We might then use this to output a response to the user informing them that the entered ID &c. is not valid.

Another option for validation is to integrate MongoDB native driver with Mongoose. We can use a method called `isValid()` relative to the ObjectID.

We can start by loading ObjectID from the MongoDB native driver,

```
const {ObjectID} = require('mongodb');
```

We can then use the method `isValid()` relative to ObjectID. e.g.

```
if (!ObjectID.isValid(docID)) {
  // execute if ID not valid
  console.log('ID not valid...');
}
```

app - build a basic API - GET route with parameter

We can create an API route to allow a user to request a route with an additional parameter for the URL, e.g.

- localhost:3030/todos/4321

The parameter, `4321`, will then be available as a property of the request object, as specified below

```
// GET route with parameter
app.get('/todos/:id', (req, res) => {
  // get params from req
  var params = req.params;
  console.log(params);
});
```

The passed parameter in the URL is specified as `:id` in the GET function above. So, the above would return an object with a name value pair,

```
{
  "id": "4321"
}
```

For this type of route with user input, we'll need to add some additional validation, error handling, and returns.

So, we'll add the `ObjectID` property from the MongoDB object (mongoose native driver),

```
var {ObjectID} = require('mongodb');
```

and then check if the passed ID parameter is valid,

e.g.

```
// validate passed ID - check not valid
if (!ObjectID.isValid(params_id)) {
  // return 404 status code for invalid ID
  return res.status(404).send();
}
```

We could then use Mongoose `findById()` to get the returned data for the ID parameter. e.g.

```
Todo.findById(params_id).then((todo) => {
  // check if return data available
  if (!todo) {
    return res.status(404).send();
  }

  // otherwise return the data for the params ID
  res.send({todo}); // return todo in object - more flexible than default array
return
}).catch((error) => { // catch return errors for query
  res.status(400).send();
})
```

So, when we call the `/todos/:id` route, we'll now get either a successful return for the query, or an error message.

app - build a basic API - DELETE route with parameter

We can now add a route to remove/delete a todo item document by specified ID parameter.

Mongoose helps us by providing three methods for removing/deleting documents from MongoDB.

So, we might need to remove all docs from the DB, e.g.

```
// remove all docs from DB
Todo.remove({}).then((result) => { // empty object required to delete all docs
  consol.log(result);
});
```

Or perhaps find a doc by ID and then delete it, e.g.

```
// find a single doc by ID and remove...
Todo.findByIdAndRemove('597b92e6031086379d868696').then((todo) => {
  console.log(todo);
});
```

We can also find a document by another property of the saved doc, e.g.

```
// find a single doc and remove from db - single doc removed will also be returned
Todo.findOneAndRemove({completed: true}).then((todo) => { // useful to find doc
without ID - i.e. by text, author &c.
  console.log(todo);
});
```

This will still only delete a single doc, the first found in the DB.

In `server.js`, we can then create our route to expose the delete options for our API.

e.g.

```
// DELETE route for single doc with ID
app.delete('/todos/:id', (req, res) => {
  // get params ID from req
  var params_id = req.params.id;
  console.log(params_id);

  // validate passed ID - check not valid
  if (!ObjectID.isValid(params_id)) {
    // return 404 status code for invalid ID
    return res.status(404).send();
  }

  // find doc by ID and remove from DB
  Todo.findByIdAndRemove(params_id).then((todo) => {
    // check if return data available
    if (!todo) {
      return res.status(404).send();
    }
    // otherwise return the data for the deleted params ID
    res.send(todo);
  }).catch((error) => { // catch return errors for the query
    res.status(400).send();
  });
});
```

The pattern used for the route is very similar to finding a single todo by ID. We simply use a different method with the requested ID to remove the doc from the DB.

We can then test this new route with Postman, creating a DELETE request for the server. We need to select the DELETE http method from the drop down method, and then specify the following sample route for a single doc id,

- {{url}}/todos/597b92e6031086379d868695

This will now search for this doc ID in MongoDB, and remove it if found. The status return for this query should be `200`, and the deleted doc will be returned in the response, e.g.

```
{
    "_id": "597b92e6031086379d868695",
    "__v": 0,
    "text": "a todo item...",
    "completedAt": null,
    "completed": true
}
```

Then, if we run our GET route for all todo docs, we should see the doc has now been removed from the collection in the DB.

app - build a basic API - PATCH route with parameter

To help with adding and configuring the update PATCH route, we can use the JS utility library *Lodash*.

```
npm install lodash --save
```

and then require this library in the `server.js` file,

```
const _ = require('lodash');
```

We can then setup the PATCH route itself for *todos* with the ID params.

e.g.

```
// PATCH route for single doc with ID
app.patch('/todos/:id', (req, res) => {

});
```

We'll then add some necessary variables, e.g.

```
  // get params ID from req
  var params_id = req.params.id;
  console.log(params_id);
  // only pick the properties we need for an update - stops false, unnecessary &c. properties being sent by the user
  var body = _.pick(req.body, ['text', 'completed']); // pick method from lodash - gets only specified properties from return req
```

As before, we'll get the required doc ID from the route params. Then, we'll create a `body` variable for the data to update. We can restrict the properties we need for the doc update by only *picking* the necessary properties from the `req` object.

As we've done for other routes with an ID param, we can check if the requested ID is valid or not,

```
// validate passed ID - check not valid
if (!ObjectID.isValid(params_id)) {
  // return 404 status code for invalid ID
  return res.status(404).send();
}
```

Then, we need to check the `completed` state of the todo item doc, e.g.

```
// check boolean return for completed i.e. true - reflects simple toggle update from false to true...
if (_.isBoolean(body.completed) && body.completed) {
  // app uses boolean = true as reason to set completedAt to current Unix timestamp - not set by user
  body.completedAt = new Date().getTime(); // returns no. of ms from midnight 1 jan 1970 to current date...
} else {
  // keep doc completed as false
  body.completed = false;
  // keep completedAt as not set - null
  body.completedAt = null;
}
```

So, we can check that the return `completed` property is, indeed, a boolean, and then check its value. A simple conditional statement is then used to set the `completedAt` time in milliseconds, or persist the todo item doc as not yet completed.

Finally, we can update the requested doc in the DB.

e.g.

```
// update the requested doc in the db - using Mongoose method, findByIdAndUpdate()
Todo.findByIdAndUpdate(params_id, {$set: body}, {new: true}).then((todo) => { //
MongoDB update - set object to body, and return the new doc object - new: true
(mongoose naming for returnOriginal)
  // check todo object exists - return 404 for not found
  if (!todo) {
    return res.status(404).send();
  }
  // if todo found - send todo object
  res.send({todo});
}).catch((error) => { // catch error
  res.status(400).send(); // send back error status - bad request code
});
```

We can use a Mongoose method, `findByIdAndUpdate()`, passing the doc id, and some options for the update. The first option is the MongoDB option to `$set` some data for the doc. In this example, we can simply return the `body` object, and set it as the updated data for this doc. Then, we can tell Mongoose to return the updated doc object. This might be useful to check and inform the user of the successful update.

In the `then()` method, we can check that the todo object exists, and handle any errors. If no errors, we can simply send the todo object.

We can now test this PATCH route with Postman, creating a test request for the PATCH http method, and saving it to the todos app collection in Postman. A sample return object for a successfully updated doc with PATCH is as follows,

```
{
    "todo": {
        "_id": "597df45fbc86a956132315fb",
        "__v": 0,
        "text": "another todo item...",
        "completedAt": 1501442182468,
        "completed": true
    }
}
```

This return is in response to simply updating the `completed` property to `true`.

app - create a separate database for tests

As we run these tests, we wipe the local DB to ensure we're running the test cases with known data &c.

However, for local development this can cause issues and is annoying for ongoing development.

Instead, we can simply create and specify a separate local DB for testing these test cases. We'll also need to update our app to recognise the different process environments, including remote (e.g. Heroku), local development, and local testing with Mocha.

The Express module popularised an environment variable for an app's process, e.g.

```
process.env.NODE_ENV === 'production' // value set by Heroku for live app
```

We might also set the value of this environment variable to `testing` or `development` for local development purposes.

We can also add a new variable to the `server.js` file for the current process environment for the app or simply a default environment, e.g.

```
var env = process.env.NODE_ENV || 'development';
```

Then, we need to configure the available `NODE_ENV` values in `package.json`.

e.g.

```
"test": "export NODE_ENV=test || SET \"NODE_ENV=test\" && mocha server/**/*.test.js"
```

This will set the environment to testing for any test files we run for this app. The first option, using `export`, is compatible with OS X and Linux, whilst Windows uses the `SET` command.

With the environment variable now set to either development, testing or production, we can now check its value in the `server.js` file with a standard conditional statement, e.g.

```
if (env === 'development') {
  process.env.PORT = 3030;
  process.env.MONGODB_URI = 'mongodb://localhost:27017/NodeTodoApp';
} else if (env === 'test') {
  process.env.PORT = 3030;
  process.env.MONGODB_URI = 'mongodb://localhost:27017/NodeTodoAppTester';
}
```

We can now update the `mongoose-config.js` file to connect to the required DB for the current environment, e.g.

```
//connect to MongoDB using Mongoose - use mLab or local uri
mongoose.connect(process.env.MONGODB_URI); // process environment returns mLab uri - url set by process.env in server.js
```

So, when we run our local app the environment will be set to `development`, which will set the MongoDB to `NodeTodoApp`. If then run the tests with Mocha, the environment will be set to `test`, and the test DB will be used.

With this environment structure, we can now maintain a working DB for each environment.

app - basic refactor of code structure

A simple refactor of app code.

We can move the environment variables and settings from the current `server.js` file to a separate file, such as `config.js` in a `config` directory.

Then, we simply require this local file in the `server.js` file.

app - CRUD API for todos app

So, we can create a todo item, read all or a single todo item, update a single todo item, and finally delete a specified todo item as necessary.

A standard CRUD API for the `node-todos-api` app.