Extra Notes - Node.js Todos API - Testing

- Dr Nick Hayward

Contents

app - API testing with Postman

A very useful, and commonly used, tool to help develop and test REST APIs is Postman. Further details available at the following URL,

- https://www.getpostman.com

Apps available for MacOS (OS X), Windows, and Linux. Download package for required OS, and follow usual install instructions.

You can also install an app for Chrome.

To use Postman either open the desktop app or Chrome app,

- chrome://apps (then select Postman app)

and start with the *Builder* tab in the app window.

We can start with a `GET` HTTP method, and then enter a test URL for the request. e.g.

- https://maps.googleapis.com/maps/api/geocode/json?address=3%20the%20strand%20brixham

This will return some JSON, and there is usually tabs available for *Body*, *Cookies*, *Headers*, and *Tests*. There will also be some extra sub-tabs for *Pretty*, *Raw*, and *Preview*.

As we build out a custom API, we can then test it using Postman.

app - build a basic API - test POST route with Postman

We can now start the server, and then test this new route with Postman.

In Postman, change the HTTP method to POST, and then enter the url for the created API route, e.g.

- localhost:3030/todos

Then, select the Body tab in Postman, and add a sample todo object, e.g.

```
{
  "text": "find the source of the Nile..."
}
```

Then, hit Send to test the POST route for the app. If successful, the console should log the saved object.

Also, if the POST route is working correctly a send response will be returned for the Todo data, along with a status code. For success, Postman will show a `200` status code.

By modifying the body of the test object for the POST route in Postman, we can test errors, model defaults &c.

e.g. if we removed the content of the `text` for a todo, the response will show errors with a status code of 200.

app - additional testing for API routes - POST

For broader testing of our app, and its functions, we can once more use

- Expect - for assertions
- Mocha - overall test suite
- Supertest - test Express route
- Nodemon - helps with the test watch script

So, we need to install them for our app,

```
npm i expect mocha supertest nodemon --save-dev
```

Then create a new folder in the `server` directory for these tests, e.g. `/server/tests` with a test file for the server, `server.test.js`.

In this `server.test.js` file, we can then add the require modules and local scripts for the tests, e.g.

```
// require node modules
const expect = require('expect');
const request = require('request');
// require local files
const {app} = require('./../server.js');
const {todo} = require('./../models/todo-model.js')
```

Then we can add a `describe` block for the `POST` API routes.

For example,

```
// describe for the POST API route
describe('POST /todos', () => {
  // add test cases for this route
  it('should create a new todo item', (done) => {
    var text = 'some text for a todo item...';

    // use Supertest to test POST
    request(app)
      .post('/todos')
      .send({
        text // ES6 shortcut for text: text
      })
      .expect(200) // assertion to test status code
      .expect((res) => { // create custom assertion to test response body text
        expect(res.body.text).toBe(text); // test that the response text matches the
text specified above in var text
      })
      .end((error, res) => { // check todo item was saved to MongoDB
        if (error) {
          return done(err);
        }
        Todo.find().then((todos) => {
          expect(todos.length).toBe(1);
          expect(todos[0].text).toBe(text);
          done();
        }).catch((e) => done(e)); // catch any errors in callback - then pass to
done() to finish
      });

  });
});
```

In this test example, we're adding a number of assertions to test the response from the POST API route. We start by testing the text response and status code, then add a check for the end of the POST request to ensure that the Todo item is saved correctly to MongoDB. Finally, we add a catch statement to check for errors in the callback, which are then passed to `done()` to finish the execution of this describe block.

However, when we check the data store for Todo items, we're assuming that there is always `0` items saved before we test this new Todo item.

For testing purposes, we can overcome this issue by simply wiping the data store before any test code is executed, e.g.

```
// before a describe block is executed - wipe existing todo items stored in data
store
beforeEach((done) => {
  Todo.remove({}).then(() => done())
});
```

In our app's `package.json` file, we can now update the `"scripts"` value by adding a `test-watch` option for Nodemon. e.g.

```
"test": "mocha server/**/*.test.js",
"test-watch": "nodemon --exec 'npm test'"
```

and then run the test using the following command,

```
npm run test-watch
```

An extra test is to check that a todo item is not created with invalid data, e.g.

```
it('should not create a todo item with invalid data', (done) => {
  request(app)
    .post('/todos')
    .send({}) // send empty data to post route
    .expect(400) // expect status code 400
    .end((error, res) => {
      if (error) {
        return done(error); // finish test if error returned
      }

      // find and return all todos in the DB - DB wiperd beforeEach - should be 0 if
no todo item created...
      Todo.find().then((todos) => {
        expect(todos.length).toBe(0);
        done();
      }).catch((e) => done(e));
    });
});
```

If we now run the same test, we should get both test passing. One for adding a todo item to the DB, and the other for invalid data not being saved to the DB. e.g. output

```
...
  POST /todos
    ✓ should create a new todo item (71ms)
    ✓ should not create a todo item with invalid data

  2 passing (118ms)

[nodemon] clean exit - waiting for changes before restart
```

app - additional testing for API routes - GET

To test GET routes, we'll need to ensure that we have some data in the DB to query. In tests for the POST route, we clear the DB on each pass of the test, so we need to modify this `beforeEach()` function to seed some known dummy data. e.g.

```
// create some dummy data - test todo items
const todos = [
  {text: 'a todo item...'},
  {text: 'another todo item...'}
];
```

Then, we need to update the `beforeEach()` function in the `server.test.js` file. As we're adding multiple objects at the same time, we can now use the new MongoDB function `insertMany`, e.g.

```
beforeEach((done) => {
  Todo.remove({}).then(() => {
    return Todo.insertMany(todos);
  }).then(() => done());
});
```

We'll also need to modify any checks with the length property to match these dummy data objects.

We can add further testing for the GET route with a test case,

```
// describe for the GET API route
describe('GET /todos', () => {
  it('should GET all todo items...dummy data found', (done) => {
    request(app)
      .get('/todos') // specify api url
      .expect(200) // check status code - 200 for OK
      .expect((res) => { // custom assertion
        expect(res.body.todos.length).toBe(2);
      })
      .end(done);
  });
});
```

and then run our tests from the terminal,

```
npm run test-watch
```

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 - use Postman with Heroku app

We can now test our new Heroku app with Postman, both GET and POST requests for the new remote app.

We might test sending a POST request to the app, e.g.

- https://your-app-url.herokuapp.com/todos

which will create a test todo item that is set in the app. The return object for this POST request will be as follows,

```
{
    "__v": 0,
    "text": "postman test todo item - another one...",
    "_id": "597cd962d828090011f2b9ce",
    "completedAt": null,
    "completed": false
}
```

If we then submit a GET request to the app's API,

- https://your-app-url.herokuapp.com/todos

we'll get the expected object containing an array of todo items, e.g.

```
{
    "todos": [
        {
            "_id": "597cd962d828090011f2b9ce",
            "text": "postman test todo item - another one...",
            "__v": 0,
            "completedAt": null,
            "completed": false
        }
    ]
}
```

We can test retrieving a single todo item by ID, e.g.

- https://your-app-url/todos/597cd962d828090011f2b9ce

which will return an object with the single todo item,

```
{
    "todo": {
        "_id": "597cd962d828090011f2b9ce",
        "text": "postman test todo item - another one...",
        "__v": 0,
        "completedAt": null,
        "completed": false
    }
}
```

To ease switching test environments in Postman, we can create environments for local, Heroku &c. and then save them for easy recall.

e.g. in the top right corner of Postman is a drop down menu for environemnt.

So, we can now create an environment for the local dev and remote dev projects.

In *Manage Environments*, we can add an environment, e.g. `Todo App Local`, and then set values for the following

- url = localhost:3030

Then, we can do the same for Heroku, and set the URL value to the Heroku app url, e.g.

- https://your-app-url.herokuapp.com

We can also abstract routes and params as required for testing with defined environments.

e.g. for the GET request in the *Todo App* collection we can modify the URL as follows,

- {{url}}/todos

If we switch to either the local or Heroku environment, this single request is now abstracted to either environment.

We can also do the same for the POST request in the collection.

app - additional testing for API routes - DELETE

We need to add some test cases for the DELETE route, e.g.

```
// check DELETE route with params ID
describe('DELETE /todos/:id', () => {
  // test case - check requested todo doc item has been removed
  it('should delete a doc for a todo item', (done) => {

  });

  // test case - check return status code for doc not found in DB
  it('should return a 404 status code for doc not found', (done) => {

  });

  // test case - check if doc object id is valid
  it('should return a 404 status code for invalid ObjectID...', (done) => {

  });

});
```

For the first test case, we need to send the query to the DB, check the response, and then check the DB itself to ensure that the doc was actually deleted successfully.

We can then fill out the first test case as follows,

```
// test case - check requested todo doc item has been removed
it('should delete a doc for a todo item', (done) => {
  // specify test todo to delete
  var hexId = todos[1]._id.toHexString();

  request(app)
    .delete(`/todos/${hexId}`) // remove the specified doc by id
    .expect(200) // assert a 200 status code for the successful doc deletion
    .expect((res) => { // add custom assertion
      expect(res.body.todo._id).toBe(hexId); // assert that response body todo doc id
matches the hexId
    })
    .end((error, res) => { // finish request
      if (error) { // handle error
        return done(error); // if error exists simply return the request as done...
      }

      // find doc id in db
      Todo.findById(hexId). then((todo) => {
        expect(todo).toNotExist(); // check that doc id does not exist in db
        done(); // call done and finish async all
      }).catch((error) => done(error)); // catch any error for async call - return
done if error caught...
    });
});
```

We can then run our tests as usual with the following command,

```
npm run test-watch
```

We can then update the remaining two tests for the DELETE route,

```
// test case - check return status code for doc not found in DB
it('should return a 404 status code for doc not found', (done) => {
  var hexID = new ObjectID().toHexString();

  request(app)
    .delete(`/todos/${hexID}`) // DELETE route to test with param ID
    .expect(404) // assert - status code should be 404
    .end(done); // call end and pass done to finish test case
});

// test case - check if doc object id is valid
it('should return a 404 status code for invalid ObjectID...', (done) => {
  request(app)
    .delete('/todos/abc123def') // pass in test invalid string - ObjectID has v.
specific pattern
    .expect(404)
    .end(done);
});
```

The final two test cases follow the same pattern as testing for a GET route. We simply update the route to match the required DELETE route.

app - additional testing for API routes - PATCH

To be able to test PATCH routes for todo items, we'll need to update the dummy todo objects to include a `completed` and `completedAT` property, e.g.

```
// update dummy todo items with test ID name:value pair property
const todos = [
  {
    _id: new ObjectID(),
    text: 'a todo item...'
  },
  {
    _id: new ObjectID(),
    text: 'another todo item...',
    completed: true,
    completedAt: 230797
  }
];
```

Then, we need to add some test cases for the PATCH route, e.g.

```
// check PATCH route with params ID
describe('PATCH /todos/:id', () => {
  // test case - check update with params for doc id
  it('should patch and update the todo item', (done) => {

  });

  // test case - check completedAt relative to complete property
  it('should reset and clear completedAt property when todo item is not completed',
(done) => {

  });

});
```

For the first test case, we can add the following requests and assertions to test this route, e.g.

```
// test case - check update with params for doc id
it('should patch and update the todo item', (done) => {
  // get ID from dummy todos object - first object
  var hexId = todos[0]._id.toHexString();
  // text for testing PATCH update
  var text = 'some test new text...';

  // setup test with assertions
  request(app)
    .patch(`/todos/${hexId}`)
    .send({
      completed: true,
      text // ES6 shortcute for name:value pair
    })
    .expect(200)
    .expect((res) => {
      expect(res.body.todo.text).toBe(text);
      expect(res.body.todo.completed).toBe(true);
      expect(res.body.todo.completedAt).toBeA('number');
    })
    .end(done);
});
```

With this first test case, we're updating the todo item with text, and then setting the `completed` property to true. We can then check the response with assertions for the new text, `completed` property set to true, and the updated

`completedAt` property to a number for the time in ms.

The second test case can then follow a similar pattern, e.g.

```
// test case - check completedAt relative to complete property
it('should reset and clear completedAt property when todo item is not completed',
(done) => {
  // get ID from dummy todos object - second object
  var hexId = todos[1]._id.toHexString();
  // text for testing PATCH update
  var text = 'some more test new text...';

  // setup test with assertions
  request(app)
    .patch(`/todos/${hexId}`)
    .send({
      completed: false, // todo item not completed
      text // ES6 shortcute for name:value pair
    })
    .expect(200)
    .expect((res) => {
      expect(res.body.todo.text).toBe(text); // assert text matches dummy text
      expect(res.body.todo.completed).toBe(false); //assert completed property false
- todo item not completed
      expect(res.body.todo.completedAt).toNotExist(); // assert completedAt does not
exist - should be null...
    })
    .end(done);
});
```

In this example, we're now getting the id for the second test todo item object, and then sending some new test text, and setting the `completed` property to false. This means we need to slightly modify the assertions to check for `completed` property as false, and that the `completedAt` property does not exist. There is no number, just a value set to `null` for this property.

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.