Aston University
Internet Applications and Techniques

Node.js Security and Architecture

Learning Objectives

After completing this worksheet, you should be able to:

- Write a data access layer using Node.js and MongoDB,
- Make your application resistant to injection attacks and cross-site scripting,
- Write a RESTful API in Node.js.

Preamble

In the previous practical task, we learned the basics of working with Node.js and its ecosystem; however, the API which we developed as part of the practical task was excessively simple. In this practical task, we are going to write a messaging API through which we store textual messages in a database and have them persist, as well as being able to apply the four basic CRUD functions (Create, Remove, Update, Delete).

The database we will be using for this task is MongoDB, a document-oriented database. However, it isn't good practice to couple the entirety of our implementation tightly to our (current) choice of Database. Doing so would make the code more difficult to write and maintain (we would be mixing concerns - likely business logic and data access) and would mean that if we ever decided to change our choice of database we would have to write significant portions of our application. Therefore, for the majority of this practical task we are going to focus on writing a **data access layer**: an API (note - not a web API) which encapsulates our data access and storage mechanisms. Once this is in place, we should be able to implement our web API using the techniques we covered in the previous unit.

Step 1 - Writing the data layer

Step 1.0 - Setup

Download `practical_task_files.zip` from Blackboard and unzip it. It contains two files: `lib/messages.js` - an incomplete implementation file which you will be finishing during this practical task and `test/messages.js` - a set of associated tests. Copy these files into the project which you created for the last unit, keeping them in their folders (i.e. the former file should be in a new `lib/` folder while the latter should be in the existing `test/` folder).

The `lib/messages.js` file contains a skeleton implementation of a data layer for our messaging API. Open the file in your preferred text editor. We will talk about this in more detail in the following subsection but, for now, you should notice that the file contains the following lines:

```
const mongoose = require('mongoose');
mongoose.connect(url,callback);
```

The first line indicates that the project has a dependency on the mongoose ODM package discussed in lectures. Use npm to install it. Check the first few lines of the test file as well. If there are any dependencies listed which you haven't yet installed, install them now. Note that if you have cloned your git repository onto a new machine and haven't yet installed the dependencies listed in `package.json`, you can do so using the command

```
$ npm install
```

The second line of the above code will attempt to connect `mongoose` to MongoDB. For this to work, MongoDB will need to be running. Open a new terminal and navigate to your project folder. Create a new directory called `mongodata`. Once you have done this, start MongoDB running using the command:

```
$ mongod --dbpath mongodata/
```

Similarly to the running server in the previous practical task, you can stop MongoDB by pressing Ctrl + C in the terminal window in which you started it.

Run the test file using `$ npm test`. You have not completed the implementation so, if you have done the above correctly, you should find that two tests fail, each with an `AssertionError`. If, instead, you get a `MongoNetworkError` and your terminal hangs, it indicates that you haven't started MongoDB. If this is the case, you may have to stop the execution of your programme by pressing Ctrl + C.

For the remainder of this practical task, every time you complete a step in the worksheet or cause a new test to pass you will want to commit your changes to git. Before you do so for this step, exclude the `mongodata/` folder from your git repository by adding the line

```
**/mongodata
```

to your `.gitignore` file. As soon as you have done this, commit your changes.

Step 1.1 - Interpreting the tests

For the remainder of Step 1, we will be completing the data layer for our messaging API. We have a skeleton implementation in `lib/messages.js`. Open the file. In it, you should see the following six incomplete methods:

Method Name	Description
`create(newMessage,callback)`	Creates a new message of the form `{username:'Some Name',text:'Some text'}`.
`read(id,callback)`	Once created, messages gain a unique `_id` property. This method reads the message with `_id` property matching the `id` passed to the function.
`readUsername(username,callback)`	Reads all messages created by the user with name `username`.
`readAll(callback)`	Reads all messages.
`update(id,updatedMessage,callback)`	Updates the message with `_id` property matching the `id` passed to the function with the data in `updatedMessage`.
`delete(id,callback)`	Deletes the message with `_id` property matching the `id` passed to the function.

In all of these cases, `callback` should be a function will be called after the database operation has completed (successfully or unsuccessfully) with two arguments. The first argument will be an error object (`null` if and only if the operation has completed without errors). The second argument will be a result object describing the result of a completed operation.

Note that I have deliberately underspecified the API. In order to fully understand what each method is supposed to do, you will need to interpret the associated test file. Open the file and have an initial look at the tests. Note that the `beforeEach` function is deleting all messages. This is good practice when we are testing any API which is designed to store data. If we do not clear the data at the start of each test, we risk the operations in one test (e.g. creating a new message) affecting the result of another (e.g. testing that the database is empty).

Go to line 49. There are 14 tests defined in this file, but I have commented out all but two of them to avoid overwhelming you with reports of test failures. We will try and make these two uncommented tests pass. We will look at the second of these two tests in detail. You will be expected to interpret the remainder of the tests independently. The body of the second test is as follows:

```
const MESSAGE_IDX = 0;
messages.create(validMessages[MESSAGE_IDX],function(err,res){
  expect(res).to.be.an('object');
  expect(res).to.have.property('_id');
  messages.read(res._id,function(err,res){
    expect(err).to.be.null;

    expect(res).to.have.property('username');
    expect(res.username).to.equal(validMessages[MESSAGE_IDX].username);

    expect(res).to.have.property('text');
    expect(res.text).to.equal(validMessages[MESSAGE_IDX].text);

    done();
  });
});
```

where `validMessages[MESSAGE_IDX]` is a message which meets our definition of a valid message (namely, it has both `username` and `text` properties, both of which have values which either are or are convertible to strings). First, let's look at the high level structure of the test:

```
messages.create(/* Some valid message */,function(err,res){
  /* Some assertions about the result of messages.create() */
  messages.read(/* ID of the object created by messages.create() */,function(err,res){
    /* Some assertions about the result of messages.read() */
    done();
  });
});
```

Looking at this, we should see that this test is about the relationship between the `message.create()` and `message.read()` methods. When we create a message using `message.create()` and then try and read the newly created message (by ID) using `message.read()` we have some expectations about the result. Hopefully what that relationship might be is fairly intuitive (when we create a message and then try and read it, we expect to get back the message which we just created!). Let's look at the code in detail to validate our intuition. The second line:

```
messages.create(validMessages[MESSAGE_IDX],function(err,res){
```

illustrates how we expect to be able to call the `create` method. We pass in some message data and a callback with parameters `err` and `res`. The next two lines:

```
expect(res).to.be.an('object');
expect(res).to.have.property('_id');
```

indicate that, when passing this valid data to `message.create()`, we expect the callback to be called with a second argument (corresponding to the `res` parameter) which is an object with property `_id` (note that this assertion doesn't mean that it can't also have other properties). On the next line, we have:

```
messages.read(res._id,function(err,res){
```

Again, this shows how we expect to be able to call the `message.read()` method. We pass in some ID (in this case, the ID of the `res` object, which we tested the existence of on the previous line) and a callback with parameters `err` and `res`. We then have a final set of assertions:

```
expect(err).to.be.null;

expect(res).to.have.property('username');
expect(res.username).to.equal(validMessages[MESSAGE_IDX].username);

expect(res).to.have.property('text');
expect(res.text).to.equal(validMessages[MESSAGE_IDX].text);
```

This indicates that, when passing the ID resulting from `message.create()` to `message.read()`, we expect the callback to be called

- with a first argument (corresponding to the `err` parameter) of `null`. In our API, a `null` error will indicate that the operation has completed without an error.
- with a second argument (corresponding to the `res` parameter) with properties `username` and `text`. The values of these properties should match those of the data which was used to create the message.

In summary, this test tells us:

- We expect a call to `messages.create()` with a valid message to result in an object with property `_id`,
- We expect to be able to pass this `_id` to `messages.read()` to retrieve a copy of the created message. When doing so, we expect a `null` error to indicate that no error has taken place.

Do an equivalent reading of the first test (which is a little simpler than the one which we have just analysed). What does it tell us about the `messages.create()` method?

Step 1.2 - Simple create and read

We will start by implementing our `messages.create()` method using the mongoose ODM. We looked at an example of using mongoose to create a document in the lecture slides:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/test');

const Credentials = mongoose.model(
  'Credentials',
  { username: String, password: String }
);

const creds = new Credentials(
  {username: 'alice', password: 'abc123'});

creds.save(function(err,docs){
  if(err) //Handle error
  // Else handle result
});
```

See if you can use this code snippet, along with the explanation given in lectures, to implement `messages.create()`. You should remove the current method body (a single call to `callback()`) and take note of the following:

- We only want to import mongoose and connect it to the database once per project. We do not need to make these calls separately for each method.
- The second argument passed to `mongoose.model` (in the above, `{ username: String, password: String }`) determines which properties can be stored in the database (in the above only the properties `username` and `password` can be stored and they must be string valued).
- the mongoose model `save` method takes, as its argument, a callback which expects to be called with two arguments. The first argument will be an error object (`null` if and only if the operation has completed without errors). The second argument will be a result object describing the document which has been created as a result of the operation completing successfully. Does this sound familiar?

Make a serious attempt at modifying `lib/messages.js` to make the first test pass. Only when you have done so, compare what you have to my solution on the next page:

Firstly, I would update the mongoose model to permit saving `username` and `text` properties.

```
const Message = mongoose.model(
  'messages',
  {username:String,text:String}
);
```

I would then use my mongoose model to create and save a new message object. We can pass our callback directly on to `message.save` because the above description of the arguments the method passes to its callback match those we specified for our callback exactly.

```
create:function(newMessage,callback){
  var message = new Message(newMessage);
  message.save(callback);
}
```

Use what you have learned from the above to complete the implementation of the `read` method. You may wish to make use of the fact that mongoose models have a method `findById` (more details in the documentation) which you can call as:

```
Message.findById(/* Arguments as defined in the documentation */);
```

Once you have both tests passing, commit your work with git (if you haven't already) and move on to the next step.

Step 1.3 - Remaining CRUD functions

Uncomment the tests relating to the renaming CRUD functions (lines 102-215 in `test/messages.js`). They specify how the remaining functions should behave when passed expected data. You should be able to proceed in the same way as in step 1.2, but may wish to make use of the following mongoose model methods (links to docs provided):

- find
- findByIdAndRemove
- findByIdAndUpdate

Commit your code using git each time that you cause a test to pass (without breaking those which were working previously!) Once all tests are passing, move on to the next step.

Step 1.4 - Data Validation

Our API should now be working if passed expected data, but when accepting data from users we should always expect errors in the data and try and control for them. Uncomment the next block of tests (lines 216-280) which relate to the expected behaviour of our API when passed incomplete data. Note that (to avoid us having to write lots of extra code during the practical task task) these tests are incomplete. We might, for instance, have defined tests for `messages.modify()` which test the same types of invalid input as those for `messages.create()` do.

Our mongoose model already gives us some data validation: properties other than `username` and `text` cannot get into our messages collection in the database. We can increase the strength of our data validation using a mongoose schema which allows us to place extra validation conditions on documents. We could create a schema (and use it to create a model) for our credentials example as follows:

```
const credentialsSchema = new mongoose.Schema(
  {
    username:{
      type:String,
      required:true
    },
    password:{
      type:String,
      required:true
    }
  },
);

const Credentials = mongoose.model(
  'Credentials',
  credentialsSchema
);
```

Hopefully this syntax is fairly intuitive. Not only have we specified that a Credentials object can only have properties `username` and `password` (of type string) but, through use of the `required` property, have specified that Credentials MUST have both of these properties.

Make use of a schema to add further validation to your messages API. If used correctly, you should be able to make all but one of the new tests pass in the same few lines of code. The test which we are unlikely to have made pass is the one related to extra properties. While extra properties will not be added to the database, their presence will not cause an error object to be passed to our callback. We could write our own check for extra properties, but we can also make use of our Schema checks by passing the following second argument to the `Schema` constructor:

```
const credentialsSchema = new mongoose.Schema(
  /* Specification here */,
  {strict:'throw'}
);
```

Which will make the model constructor (e.g. `new Credentials(...)`)throw an exception when we try and create a new document instance with extra properties. You will need to write some custom exception handling logic to ensure that, when thrown, this exception is handled correctly and the callback called with the appropriate arguments.

Step 1.5 - Security

Uncomment the final block of tests (lines 281-345). These tests relate to the vulnerability of our application to two types of attacks: XSS and NoSQL injection. In the former case, we should be able to use the sanitize-html package discussed in lectures to remove dangerous HTML from newly created messages. We could take a similar approach to sanitizing mongoose queries or, given that there are a limited number of places in which user-generated data can get into our mongoose queries, we can protect against injection attacks by checking that data is of the expected type (using the `typeof` operator) and generate an appropriate error when it is not.

Step 2 - Web API

With the knowledge (and code) from the last two practical tasks, we should be in a position to use our data access layet to create a web API. Create a web API which allows us to:

- create a single message with a POST request to `api/v1/messages/`
- read a single message with a GET request to `api/v1/messages/:id` where `id` is the ID of the message to be read
- read all messages with a GET request to `api/v1/messages/`
- update a single message with a PUT request to `api/v1/messages/:id` where `id` is the ID of the message to be read
- delete a single message with a DELETE request to `api/v1/messages/:id` where `id` is the ID of the message to be read

You should make sure that your web API is well tested and that it responds to requests with appropriate HTTP status codes, including appropriate error codes when a request comes with invalid data.

To help with the implementation of the reading, update and deletion of single messages, you may wish to consult the documentation on Express route parameters.