Lab 3: Structuring applications in Node

1. Purpose of this lab

Over the past few weeks, we have looked at using Node and a handful of modules to create API's. This week's lab will look at how to structure Node applications to manage scalability and readability. We will also introduce some new tools and concepts that are common of Node applications.

1.1. Exercise 1: Package.json

Taken from:
https://docs.nodejitsu.com/articles/getting-started/npm/what-is-the-file-package-json/

All npm packages contain a file, usually in the project root, called package.json - this file holds various metadata relevant to the project. This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies. It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both npm and to the end users of the package. The package.json file is normally located at the root directory of a Node.js project.

Let's re-create the chat application from last week using the concepts that we learn in this lab.

1. Create a new directory called 'Lab_3'
2. Inside this directory, create a file called 'package.json' and insert the following code into it:

```
{
    "name" : "my_chat_application",
    "description" : "A simple chat API.",
    "homepage" : "http://my.project.website",
    "keywords" : ["chat", "application", "api"],
    "author" : "My name <me@myemail.org>",
    "main" : "app.js",
    "version" : "0.0.1",
    "dependencies": {
        "express": "^4.15.3",
        "mysql": "^2.13.0",
        "body-parser": "^1.17.2"
    }
}
```

Note: The above should be self explanatory, look up anything you don't understand online (try https://docs.npmjs.com/files/package.json.)

‘Main’ is the file that will be included if someone does a `require` of your package. In the dependencies you can set the necessary version for each required package using semantic versioning (see semver.org.) It’s also common to include a ‘scripts’ block containing single line commands for various life-cycle events. One of the most usual is an entry under ‘start’ to be be run in response to ‘npm start’.

3. Now in your terminal, navigate to the directory and run ‘npm install.’ Your dependencies will install into a ‘node_modules’ directory.
4. Create a file with the same name as the ‘main’ variable in package.json (app.js). We are now ready to start coding our API.

Note: npm also includes a function called ‘npm init’ that asks you questions in the terminal and builds up the package.json file based on your answers.

1.2. Exercise 2: Structuring large applications and MVC

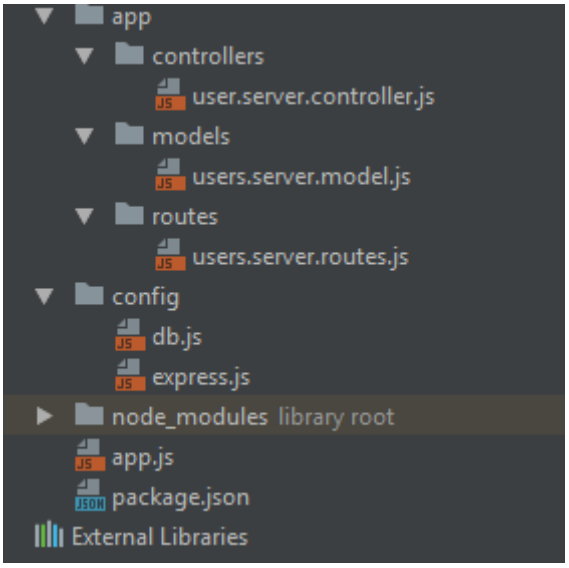
Now that we have our modules installed, we can begin to develop our app. We want to break up the structure of our application to make it easier to understand for developers and ensure our application scales with minimal code reuse.

1.2.1. Model, View, Controller (MVC)

MVC is an architectural pattern that is used to break up the structure of an application into its conceptual counterparts. Anything relating to domain elements or interactions with databases comes under the ‘model’ section, anything that relates to presentation or the interface comes under the ‘view’ section and all the application’s logic is stored under the ‘controller’ section.

In our application, we store each part of the MVC structure in its own directory. As our API has no ‘view’ we use a ‘routes’ directory to provide the definition of the endpoints of our API.

1. Create two directories in your project called ‘config’ and ‘app.’ Inside the ‘app’ directory, add three more directories called ‘routes’, ‘models’, and ‘controllers.’
2. In your config directory, create a file called db.js. This will hold all the details for connecting and configuring the database. Here is how the directory structure should look in Webstorm:



3. Inside db.js, we have two functions; ‘connect’ that connects to the database and ‘get’ which returns the connection pool. Copy the below code into db.js

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

let state = {
  pool: null
};

exports.connect = function(done) {
  state.pool = mysql.createPool({
        host: ‘mysql3.csse.canterbury.ac.nz’,
        user: 'your user code',
        password: "your password",
        database: "your user code"
  });
  done();
};

exports.get_pool = function() {
  return state.pool;
};
```

5. Create another file in your config directory called ‘express.js.’ This will hold all the details for configuring express, as well as being the starting point for our express API.
6. Inside ‘express.js’ we have one function. This function simply initiates express, sets up body-parser and then returns the app.

```
const express = require('express'),
        bodyParser = require('body-parser');

module.exports = function(){
        const app = express();

        app.use(bodyParser.json());

        return app;
};
```

7. Now in our ‘app.js’ file, import the two config files, initiate express using the express function in the config file that we have just written and connect to the database using the connect function in the imported database config file. If a connection to the database is successfully created, then start the server.

```
const db = require('./config/db'),
       express = require('./config/express');

const app = express();

// Connect to MySQL on start
db.connect(function(err) {
  if (err) {
    console.log('Unable to connect to MySQL.');
    process.exit(1);
  } else {
    app.listen(3000, function() {
      console.log('Listening on port: ' + 3000);
    });
  }
});
```

8. Run your app.js file. The application can’t do anything at the moment but we can test that the database successfully connects.

1.3. Exercise 3: Adding the Users functionality to the API

Now that our boilerplate code for the application is working, we can add some functionality to the API. Like in last weeks lab, this exercise will run through the Users functionality and then you will create the rest of the API on your own. This is the API specification for the Users functionality from lab 2 (we have changed the URL’s slightly).

URI	Method	Action
/api/user	GET	List all users
/api/user/:id	GET	List a single user
/api/user	POST	Add a new user
/api/user/:id	PUT	Edit an existing user
/api/user/:id	DELETE	Delete a user

1.3.1. Exercise 3.1: Boilerplate code and structure

1. In our express config file, before we return the app variable, add a line that imports a file called ‘user.server.routes.js’ from the ‘/app/routes’ directory. This file will take in the ‘app’ variable as shown in the code below.

```
const express = require('express'),
```

```
        bodyParser = require('body-parser');

module.exports = function(){
        const app = express();

        app.use(bodyParser.json());

        require('../app/routes/user.server.routes.js')(app);

        return app;
};
```

2. Create the ‘users.server.routes.js’ file in the routes directory. This file will import a users controller and then define each of the relevant routes as outlined in the specification. Each route calls a function in our controller. Another controller function is used for retrieving a user from the ID that is specified in the URL.

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

module.exports = function(app){
        app.route('/api/users')
                .get(users.list)
                .post(users.create);

        app.route('/api/users/:userId')
                .get(users.read)
                .put(users.update)
                .delete(users.delete);

};
```

3. Next we need to create the users controller, create a file in the controllers directory called ‘user.server.controller.js’
4. This file will import the ‘User’ model and contain the six functions that are called by the routes file. For now, add them as functions that just return null.

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

exports.list = function(req, res){
        return null;
};

exports.create = function(req, res){
        return null;
};

exports.read = function(req, res){
        return null;
};

exports.update = function(req, res){
        return null;
};

exports.delete = function(req, res){
        return null;
};
```

```
exports.userById = function(req, res){
        return null;
};
```

5. Finally, we need to add our model code. In the models directory, create a file called ‘user.server.model.js’
6. This file should import the database config file and contain the following functions.

```
const db = require('../../config/db');

exports.getAll = function(){
  return null;
};

exports.getOne = function(){
  return null;
};

exports.insert = function(){
  return null;
};

exports.alter = function(){
  return null;
};

exports.remove = function(){
  return null;
};
```

1.3.2. Exercise 3.2: Listing all Users

Now we have the boilerplate code set up, we just need to fill in the functions that we have left blank. First we will look at listing all users.

1. In the User model, edit the getAll function so that it takes in ‘done’ as a parameter. ‘Done’ will be a function that we can input from the controller so that different interactions with the getAll function can be handled separately.
2. Get the database and run a query to select all from the users table. If an error occurs, we call our ‘done’ function with our error as a parameter. If all has worked, then we call the ‘done’ function with the results of the query as a parameter. **Note:** the done() function should make more sense in the next step where we implement in within the controller.

```
exports.getAll = function(done){
      db.get_pool().query('SELECT * FROM Users', function (err, rows) {

              if (err) return done({"ERROR": "Error selecting"});

              return done(rows);
      });
};
```

3. Now in the controller, implement the list function. The list function calls the models 'getAll' function, inserting a function (the done function) as a parameter. This function simply returns the json object inputted into the done function to the client.

```
exports.list = function(req, res){
	User.getAll(function(result){
		res.json(result);
	});
};
```

4. Now run your app.js file for testing. Sending a GET request to /api/users should result in all the users being returned. **Note:** Make sure you have some users in your table for testing before running this.

1.3.3. Exercise 3.3: Creating new Users

1. In the model file, edit the insert function so that it takes in the username and a done() function as parameters. Query the database so that it inserts a new record into the users table. Again, if there is an error we execute the done function with the error in as a parameter. If there is no error then we invoke the done function with the result as a parameter.

```
exports.insert = function(username, done){
  let values = [username];

  db.get_pool().query('INSERT INTO Users (username) VALUES ?', values, function(err,
result) {

    if (err) return done(err);

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

2. Now create the controller function. This function gets the username from the POST data and adds it to a list called values. The values list is then inputted to the models insert function, along with a callback function (done()). This function simply returns the result to the user.

```
exports.create = function(req, res){
	let user_data = {
	    "username": req.body.username
	 };

	let user = user_data['username'].toString();

	let values = [
	    [user]
	 ];

	User.insert(values, function(result){
		res.json(result);
	});
```

```
};
```

3. Run app.js and test using Postman (use https://www.getpostman.com/docs/postman/sending_api_requests/requests to help you)

1.3.4. Exercise 3.4: Getting a single User

1. In the model file, edit the getOne function so that it takes in the userId and a done function as parameters. Like the previous functions, run the query and return the results.

```
exports.getOne = function(userId, done){
  db.get_pool().query('SELECT * FROM Users WHERE user_id = ?', userId, function (err,
rows) {
    if (err) return done(err);
    done(rows);
  });
};
```

2. In the controller, edit the read function to retrieve the id from the url and call the getOne function. Return the result to the client.

```
exports.read = function(req, res){
        let id = req.params.userId;
        User.getOne(id, function(result){
                res.json(result);
        });
};
```

3. Now run app.js and test.

1.3.5. Exercise 3.5: Altering a User

1. Create the model function, using the previous tasks as a template
2. Create the controller function, using the previous tasks as a template
3. Test using Postman

1.3.6. Exercise 3.6: Deleting a User

1. Create the model function, using the previous tasks as a template
2. Create the controller function, using the previous tasks as a template
3. Test using Postman

1.4. Exercise 4: Implement the rest of the API - Recommended

Like the last lab, this last exercise is optional. Carry on working until you are comfortable with the concepts covered.

Now implement the rest of the API to the following specification:

URI	**Method**	**Action**
/api/conversations	GET	List all conversations
/api/conversations/:id	GET	List one conversation
/api/conversations	POST	Add a new conversation
/api/conversations/:id	PUT	Edit an existing conversation
/api/conversations/:id	DELETE	Delete a conversation
/api/conversations/:id/messages	GET	List all messages from a conversation
/api/conversations/:id/messages/:id	GET	List a single message from a conversation
/api/conversations/:id/messages	POST	Add a new message to a conversation

That concludes the server-side labs. You should now know everything you need to complete the first assignment. In the next lab (next term), we begin to look at client applications.