Lab 3: Structuring applications in Node

1. Purpose of this lab

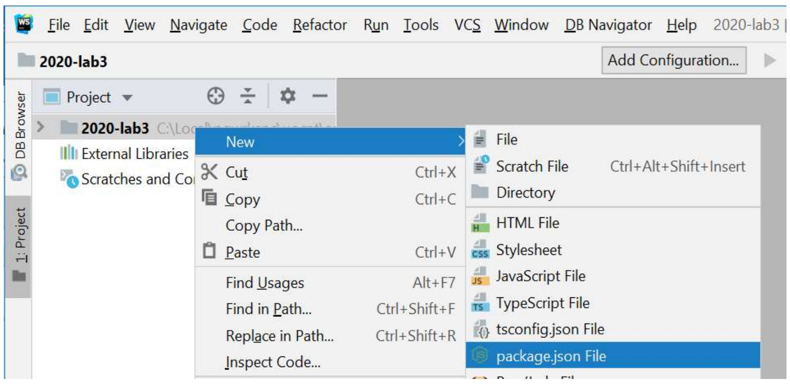
Over the past few weeks, we have looked at using Node and a handful of modules to create API's. This week's lab will look at how to structure Node applications to manage scalability and readability. We will also introduce some new tools and concepts that are common of Node applications.

1.1. Exercise 1: Package.json[1]

All Node.js project contain a file, usually located at the root directory, named **package.json** that 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.

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": "2021-lab3",
  "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": "^4.17.1",
    "mysql2": "^2.1.0",
    "body-parser": "^1.19.0",
    "dotenv": "^8.2.0"
  }
}
```

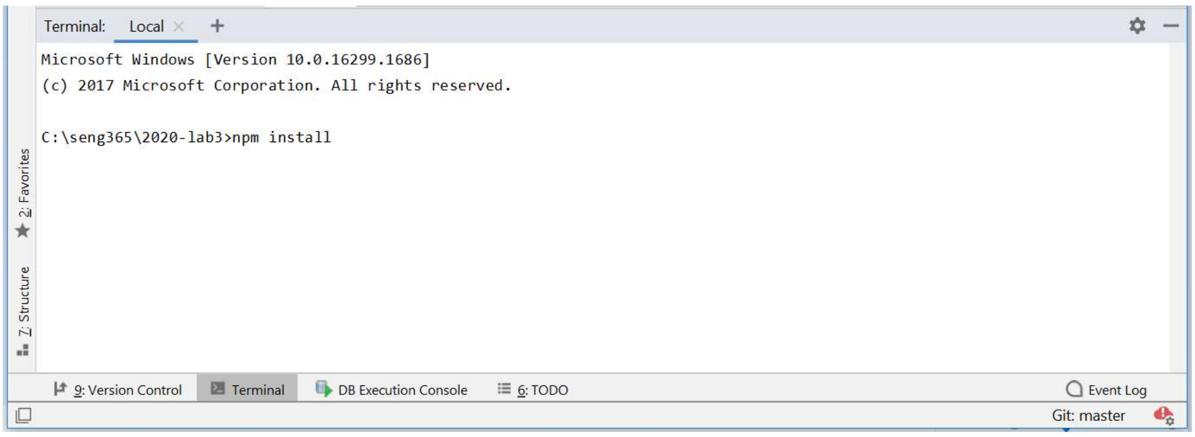
[1] Taken from: https://docs.nodejitsu.com/articles/getting-started/npm/what-is-the-file-package-json/

Note:

1. If you copy the above code to paste in your project, make sure there is no extra empty spaces between keywords and values as it would affect the parse of the values. The best – and **recommended** – is to type from scratch rather than copy and paste.
2. The above should be self-explanatory, look up anything you don't understand online (try https://docs.npmjs.com/files/package.json).
3. The keyword '*main'* refers to 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[2]. 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 run in response to '*npm start'*.

3. Now in your terminal, navigate to your project directory and run '*npm install*.' Node.js will created a new directory under your project folder named '**node_modules**' and install the dependencies listed in the 'Package.json' file in this new directory.

   As an alternative to the system terminal, you can use the *Webstorm* terminal, which by default, will prompt you in the project's root folder:



```
Microsoft Windows [Version 10.0.16299.1686]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\seng365\2020-lab3>npm install
```

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: **Node.js** 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 refactor.

1.2.1. Model, View, Controller (MVC)

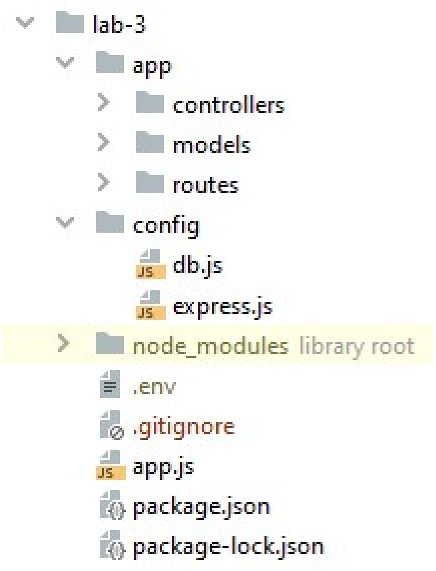
MVC is an architectural pattern that is used to break up the structure of an application into its conceptual counterparts. Anything relating to domain elements or interactions with databases comes

[2] Check https://semver.org/ for further information.

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*:



```
lab-3
  app
    controllers
    models
    routes
  config
    db.js
    express.js
  node_modules library root
  .env
  .gitignore
  app.js
  package.json
  package-lock.json
```

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

```
const mysql = require('mysql2/promise');
require('dotenv').config();

let state = {

  pool: null
};

exports.connect = async function () {

    state.pool = await mysql.createPool( {
        host: process.env.HOST,
        user: process.env.USER,
        password: process.env.PASSWORD,
        database: process.env.DATABASE,
    } );
    await state.pool.getConnection(); //Check connection
    console.log('Successfully connected to the database');
};

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

4. 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.
5. 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;
};
```

6. 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
async function main() {
  try {
    await db.connect();
    app.listen(process.env.SENG365_PORT, function() {
      console.log('Listening on port: ' + process.env.SENG365_PORT);
    });
  } catch (err) {
    console.log('Unable to connect to MySQL.');
    process.exit(1);
  }
}

main().catch(err => console.log(err));
```

7. Finally create a ‘**.env**’ file in our root folder with the following environment variables, ensuring that the values are updated with your own credentials for database connection.

```
HOST=db2.csse.canterbury.ac.nz
USER= {your user code}
PASSWORD={your password}
DATABASE={usercode}_s365_lab2
SENG365_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 week's 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/users	GET	List all users
/api/users/:id	GET	List a single user
/api/users	POST	Add a new user
/api/users/:id	PUT	Edit an existing user
/api/users/: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/:id' )
        .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'** 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 = async function(req, res){
       return null;
};

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

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

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

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

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

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

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

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

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

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

exports.remove = async 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. Get the database and run an asynchronous query to select all from the users table.

```
exports.getAll = async function( ) {

    console.log( 'Request to get all users from the database...' );

    const conn = await db.getPool().getConnection();
    const query = 'select * from lab2_users';
    const [ rows ] = await conn.query( query );
    conn.release();
    return rows;
};
```

2. Now in the controller, implement the list function. The list function calls the models '**getAll()**' async function. This function simply waits for and returns the result from the model function and handles error flows.

```
exports.list = async function( req, res ) {

    console.log( '\nRequest to list users...' );

    try {
        const result = await user.getAll();
        res.status( 200 )
            .send( result );

    } catch( err ) {

        res.status( 500 )
            .send( `ERROR getting users ${ err }` );
    }
};
```

3. 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 as a parameter. Query the database so that it inserts a new record into the users table.

```
exports.insert = async function( username ) {

    console.log( `Request to insert ${username} into the database...` );

    const conn = await db.getPool().getConnection();
    const query = 'insert into lab2_users (username) values ( ? )';
    const [ result ] = await conn.query( query, [ username ] );
    conn.release();
    return result;
};
```

2. Now create the controller function. This function gets the username from the POST data and then input it to the models insert function. This function simply returns the result to the user. If there is an error in the model function, we will catch it and return a HTTP error code (500) along with the error message.

```
exports.create = async function( req, res ) {

    console.log( '\nRequest to create a new user...' );

    const username = req.body.username;

    try {
        const result = await user.insert( username );
        res.status( 201 )
           .send({user_id: result.insertId});
        //We return the user_id to the client to use in further requests
    } catch( err ) {

        res.status( 500 )
           .send( `ERROR creating user ${username}: ${ err }` );
    }
};
```

3. Run app.js and test using Postman[3].

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 as a parameter. Like the previous functions, run the query and return the results.

```
exports.getOne = async function( id ) {

    console.log( `Request to get user ${id} from the database...` );

    const conn = await db.getPool().getConnection();
    const query = 'select * from lab2_users where user_id = ?';
    const [ rows ] = await conn.query( query, [ id ] );
    conn.release();
    return 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 = async function( req, res ) {

    console.log( '\nRequest to read a user...' );

    const id = req.params.id;

    try {
        const result = await user.getOne( id );
        if( result.length === 0 ){
```

[3] Should you need, use https://www.getpostman.com/docs/postman/sending_api_requests/requests as a support.

```
            res.status( 400 )
               .send('Invalid Id');
        }
        else {
            res.status( 200 )
               .send( result );
        }
    } catch( err ) {

        res.status( 500 )
            .send( `ERROR reading user ${id}: ${ err }` );
    }
};
```

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.