Lab 2: Creating an API in Node

1. Purpose of this lab

The purpose of this lab is to being to explore how you can use node.js to implement an API.

1.1. Data formats e.g. JSON

Throughout this course, we will be using JSON objects. JSON stands for JavaScript Object Notation. JSON is a lightweight data-interchange format. W3Schools provide a good introduction and tutorial for getting started with JSON (https://www.w3schools.com/js/js_json_intro.asp).

2. What is ExpressJS?

“Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications.” [expressjs.com]

2.1. Exercise 1: Introduction to ExpressJS - Hello World!

Taken from: https://expressjs.com/en/starter/hello-world.html

1. First create a directory named lab2_ex1, navigate to this directory in your terminal
2. Install express using: ‘npm install express’
3. In the lab2_ex1 directory, create a file named app.js and add the following code:

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

const app = express();

app.get('/', function(req, res){
  res.send("Hello World!");
});

app.listen(3000, function(){
  console.log("Example app listening on port 3000!");
});

app.use(function (req, res, next) {
  res.status(404).send("404 Not Found");
});
```

4. In your terminal, run ‘node app.js’
5. Navigate to http://127.0.0.1:3000 to see the output

The app starts a server and listens on port 3000 for connections. When receiving a HTTP request for the root (‘/’) resource, the app responds with “Hello World!” For every other path, it will respond with a ‘404 Not Found’ HTTP response.

Note: you can read more about HTTP ports here: https://en.wikipedia.org/wiki/Port_(computer_networking)

3. RESTful API’s

REpresentational State Transfer (REST) is an architecture that makes use of the HTTP protocol for communicating between different clients and servers on the Web. A REST API provides access to resources that the client can access and modify using the HTTP protocol. Each resource is identified using URI’s and unique identifiers. Table 1 below shows how different HTTP methods are used in REST to perform specific actions.

HTTP Method	REST Action	Example URL	Example (A blog)
GET	Retrieve a resource	/articles/1234	Viewing a blog article
POST	Add a new resource	/articles/	Posting a new blog article
PUT	Update an existing resource	/articles/1234	Correcting a spelling mistake in a already posted blog article
DELETE	Delete a resource	/articles/1234	Delete a blog article

Table 1: An example of how different HTTP methods are used within an application

3.1. Exercise 2: Basic routing in Express

In Express, we can use these HTTP methods to perform different actions. In this exercise, we will use Express to make a single web page react differently depending on the HTTP method that is used.

1. Create a new directory named lab2_ex2. Inside this directory, use npm to install Express and create an ‘app.js’ file.
2. As with exercise 1, include the Express module and initiate Express using a variable called ‘app’

```
const express = require('express');
const app = express();
```

3. Add a function to manage a user sending a GET request to the applications root resource. This GET request should just return the string ‘HTTP request: GET /’

```
app.get('/', function(req, res){
```

```
    res.send('HTTP request: GET /');
});
```

4. Next, create similar functions for POST, PUT and DELETE requests.
5. Call the 'listen' function to start a server on port 3000.
6. Run the server by navigating to the directory in the terminal and running 'node app.js'
7. To test your simple API, you can use the following tools:
    a. POSTMAN (recommended) - https://www.getpostman.com/ (Postman is not installed in the labs. However, you can download it from getpostman.com/app/download/linux64 and run the Postman binary. Postman will continuously prompt you to create an account but feel free to ignore these
    b. CURL - https://en.wikipedia.org/wiki/CURL or https://curl.haxx.se/

3.2. Exercise 3: Creating an API - Microblogging site

Now that we understand the basic concepts behind Express, we can create our first API. We will create an API for a microblogging site (like Twitter), beginning with the following functionality:

HTTP Method	URL	Action
GET	/users	List all users
GET	/users/:id	List one user
POST	/users	Add a new user
PUT	/users/:id	Update a user
DELETE	/users/:id	Delete a user

1. Create a new directory called lab2_exercise3
2. Inside this directory, create a new file called 'app.js' and use npm to install express
3. Inside this file, import the express module and initiate express inside a variable called 'app'
4. Copy and paste the JSON from Appendix A into a file called 'users.json'
5. Import the list of JSON users into your application using the 'require' directive. You can confirm that this has worked by printing out to the console.

```
const data = require('./users.json');
const users = data.users;
console.log(users[0]);
```

6. Now we can start to build our API functionality. Create the list all users function that returns the JSON

```
app.get('/users', function (req, res) {
```

```
    res.send(users);
});
```

7. Creating the function to list one specific user is more difficult as we have to retrieve this user from the list. Create the function using the code below.

```
app.get('/users/:id', function(req, res){
        let id = req.params.id;
        let res_data = "No user";

        for (let user of users){
                if(id == user.id){
                        res_data = user;
                        break;
                }
        }

        res.send(res_data);
});
```

8. Now we will write the post request. This time the request will contain a JSON object that contains the data required for the new user. Before doing so, we need to include the 'body-parser' module. Install it using npm and import it into the module.
9. Initiate it as shown below.

```
// Import the module
const bodyParser = require('body-parser');
// Tell the express app to expect json in the body of the request
app.use(bodyParser.json());
```

10. Now we can write our POST function.

```
app.post('/users', function(req, res){
        let user_data = req.body;

        users.push(user_data);
        res.send(users);
});
```

10. Using the 'list one user' and 'add a new user' functions as a template, we can create a function for the PUT method

```
app.put('/users/:id', function(req, res){
        let id = req.params.id;
        let user_data = req.body;

        for (let user of users){
                if(id == user.id){
                        let uid = users.indexOf(user);
                        users[uid] = user_data;
                        break;
                }
        }
```

```
        res.send(user_data);
});
```

11. Finally, to create our DELETE method, we use the JavaScript ‘delete’ operator.

```
app.delete('/users/:id', function(req, res){
        let id = req.params.id;

        for (let user of users){
                if(id == user.id){
                        var uid = users.indexOf(user);
                        // remove 1 item at index ‘uid’
                        users.splice(uid, 1);
                }
        }

        res.send(users);
});
```

12. Make your application listen on port 3000 and test using Postman or CURL.

NOTE: The remaining exercises have been left vague intentionally; how you implement the changes is up to you.

3.3. Exercise 4: Adding followers

For this exercise, make a copy of your solution to exercise 3, renaming it to exercise4. Then add functionality that allows users to follow other users. The JSON for each user should include a ‘following’ key, the value of which is a list of ID’s that the user follows.

Make sure you add functionality for the following:

1. Add the followers feature to the project, by copying and updating the existing functionality.
2. Add a ‘follow’ function to the API, this function will add a new user ID to a specified users following list.
3. Add an ‘unfollow’ function.
4. Add a ‘view_followers’ function, this function will show all the users that a specified user is following.

3.4. Exercise 5: Adding microblog posts

Each user should also have a list of microblog posts. Functionality should exist that:

1. Creates a new post for a user
2. Retrieves all of a user's posts
3. Retrieves a single post of a specified user
4. Updates an existing post for a user
5. Deletes a post
6. Retrieves all the posts from all the followers of a specified user
7. **(Extra challenge)** Implement ‘Likes’ on posts

In this lab, we have written our first API using Node. However, a problem with our application is that the data isn't persisted anywhere. Once the server is stopped (or crashes), we will lose all of our users. In the next lab, we will look at persisting our application data to a database.

4. Appendix A: users.json

The JSON data below was generated using an automatic online data generator (http://www.json-generator.com/). We use this data in the exercises for this lab as an example.

(If you have problems copying and pasting this text, try the users.json file available on learn)


```
{
        "users": [
        {
                "Id": "1001",
                "age": 35,
                "first_name": "Burch",
                "last_name": "George",
                "gender": "male",
                "email": "burchgeorge@geofarm.com"
        },
        {
                "Id": "1002",
                "age": 31,
                "first_name": "Rachelle",
                "last_name": "Chang",
                "gender": "female",
                "email": "rachellechang@geofarm.com"
        },
        {
                "id": "1003",
                "age": 38,
                "first_name": "Sheri",
                "last_name": "Bennett",
                "gender": "female",
                "email": "sheribennett@geofarm.com"
        },
        {
                "id": "1004",
                "age": 32,
                "first_name": "Fisher",
                "last_name": "Dillard",
                "gender": "male",
                "email": "fisherdillard@geofarm.com"
        },
        {
                "id": "1005",
                "age": 20,
                "first_name": "Pope",
                "last_name": "Bailey",
                "gender": "male",
                "email": "popebailey@geofarm.com"
        }
        ]
}
```