IMY 210
Unit Assignment 4: RESTful API

During this assignment you will be creating a XSLT stylesheet documents (**a3.xsl**) to stylise the provided XML document (**character.xml**). The assignment will not only test your ability to create a stylesheet document but also your understanding of the XSLT/XPath and their implementations.

Important!

- Make frequent backups of your working files in different locations.
- Use a text editor (e.g. Notepad++) to complete this assignment.
- This is an individual assignment.
- In terms of the class notes, the scope of this assignment is unit 2, Themes 4 - 5.

Provided Files

- **data.json**: JSON format data you will be using during this assignment.

Scenario

- Many social media, and other data service provider, often provide developers with a set of function for accessing data. Be it retrieving the latest post on Facebook or a tweet from Twitter.

- Often these functions are provided in the form of public RESTful APIs that allow clients to call certain defined functions to perform selective REST functionality.

  Examples:

  - Twitter standard API [link]
  - Instagram Basic Display API [link]

- For this assignment, you will learn how to set up and install multiple packages in node.js in creating your first RESTful API for an e-commerce website.

- You are provided with a simple JSON file that will serve as the base of this database.
  - For this assignment, we will be using a JSON file, but the data type is interchangeable from JSON to XML.
- For this you will install three main packages within node.js's package manager:
  - Express
    - Express.js is a node.js web application server framework, which is specifically designed for building single-page, multi-page, and hybrid web applications. It has become the standard server framework for node.js.
  - Body Parser
    - Body parser is a Node.js middleware. Body parser extracts the entire body (think HTML) from the request and makes it easier for users to interface and manipulate them.
  - Nodemon (optional)
    - Nodemon is a CLI utility that watches file system changes and automatically restarts the server when changes are made.

- Additional software you want to use:
  - Postman
    - Postman is a simple tool to help dissect, access, and test RESTful APIs. Refer to the practical video files on how to use the tool.

Task - Creating a RESTful API

1. Install **node.js** on your computer
    a. When installing **node.js**, you should install the suggested package manager **chocolatey** as well.

2. Installing node packages
    a. To install node packages, you will need to run the **npm** command in your command prompt or terminal (refer to as **shell**).
    npm install <package name>

    b. For this assignment, you will install the packages globally by calling the command shown above. The three packages you need to install are **express**, **body-parse**r and **nodemon** (optional).
    e.g. **npm install express**

Note: In most cases you would want to install packages globally you should use the -g flag: npm -g install express. For this practical install everything locally will suffice.

3. Start building the **server.js** file in the **root directory** of your project.
    a. The **server.js** file is the entry point to your server. This file will hold all logic and set up the basic environment for your application.
    b. You will need to add the following lines to your **server.js**
        i. The packages you will be using:

```
var express    = require('express');
var bodyParser = require('body-parser');
var fs = require('fs');
```

We will be using **three** packages namely: **express**, **body-parser** and **fs**.

        ii. The type of application we are declaring:

```
var app = express();
var routes = require('./routes.js')(app,fs);
app.use(bodyParser.json())
```

The base application will be an **express** application that uses JSON functionality provided by **body-Parser**. We will call express and fs functions via our **app** and **fs** in our routes definitions (refer to class notes).

Note: Line 3 is adding bodyParser to the app BUT line 2 has already send the app object through, change these 2 lines around.

For this section you will also need to add an extra line
app.use(bodyParser.urlencoded({extend: false}))

        iii. The port you want to run your application:

```
var port = process.env.PORT || 3000;
app.listen(port);
console.log('Servering running on port ' + port);
```

Here we set the application to run on port **3000** and print out a message telling us the server is running.

        iv. Lastly, some way to test our server:

```
app.get('/', function (request, respond) {
  respond.send('Hello, World!')
})
```

A basic route that will return "Hello World" when we access localhost:3000 in the browser.

c. At this stage, you can test your server by running "node server.js" or "nodemon server.js" in the shell. After you see the message stating your server is running, access it by typing localhost:3000 in your browser. To end the server, press Ctrl+C.

If your server gives you an error "cannot find module routes.js" complete the first 2 steps of the next section and run the command again.

4. Setting up API requests in the **routes.js** file, this file should be created in the **root directory** of your project.
   a. The **routes.js** file will act as our main handler for all endpoint (route) requests. We will create a function to handle every possible endpoint request, for this assignment the functions will handle GET, POST, PUT, and DELETE requests. These functions will form our RESTful API.

   b. The basic structure of your **routes.js** file should look like this:

```
var router = (app, fs) => {
    //all functions goes here
};

module.exports = router;
```

   c. In this section of creating GET (read), POST (create), PUT (update), and DELETE functions you will be introduced to 2 function from the fs library: **readFile()** and **writeFile().**

   To test every REST request created, you are welcome to use **Postman** to simulate the REST request to the server. (Refer to extra video on how to use postman)

   d. **GET** request function (Reading the file):
      i. Start by declaring a **get** method:

```
app.get('/data', (request, respond) => {

});
```

You are setting up a handler for calling a **GET** request, on the URL of /data. This means when calling a **GET** request on the endpoint of localhost:3000/data, you will follow this function. In our case, calling this **GET** function will return the data found in the JSON file.

      ii. Remember to declare the source of data.

```
data_file = 'data.json';
```

      iii. Lastly we will call fs.readFile() function.

```
fs.readFile(data_file, (error, rdata) => {
    if (error)
        throw error;
    respond.send(JSON.parse(rdata));
});
```

respond.send will return the requested data from the server-side, it's essential to send a response back to the request to notify the status of the request.
JSON.parse transforms our JSON object to a JavaScript object to be manipulated in JavaScript.

During this assignment, you will be using JavaScript arrow functions. More on arrow functions will be explained in IMY 220.

At this stage you should be able to access the data using http://localhost:3000/data

iv. You can also call a **GET** method for a specific entry by specifying some sort of parameter.

```
app.get('/data/:id', (request, respond) => {
var tmpId = request.params["id"];
```

When calling a specific parameter, you will need to pull the value out of the request. Depending on what content is sent with the **GET** request you will retrieve the data differently. (e.g. request.params, request.body, request.headers, etc.)

e. **POST** request function (Creating an entry):
   i. When creating a handler for the POST method, our function will follow these steps:
      1. Read the existing data from the JSON file
      2. Generate a new key for the new item
      3. Retrieve the data from the request
      4. Append the new data into the existing file
      5. Write the data retrieved from the request and writing the data in the JSON file.

   ii. We will count the number of objects in the JSON data to determine the new array place to add our new object.

```
Object.keys(jsondata).length;
```

   iii. Once we have parsed the JSON object, you can simply append the new data from the request.

```
parse_json_data[newId] = request.body;
```

Retrieving the data is dependant on how you send the data with your request. In the above example, we directly sent a JSON object, if you are working in another format you will need to need to alter this line of code to cater to your method.

   iv. Lastly, call the **writeFile** function to write the new data back into the file. Remember to transform the data from an object to plain text before writing into the file.

```
var new_data = JSON.stringify(parse_json_data);
fs.writeFile(data_file, new_data, (error) => {
    if (error)
        throw error;
});
```

Don't forget to send a **response** back to the request once all your operations are done.

f. PUT (update) and DELETE
   i. PUT and DELETE will follow the same format as the POST request only difference is adding a specific parameter to the request

ii. The delete function should remove an object based on its id property. To remove an object from the array you should use the splice() function or the delete operator, once you have located the object you wish to delete.

```
//UPDATE
app.put('/data/:id', (request, respond) => {

})

//DELETE
app.delete('/data/:id', (request, respond) => {

})
```

5. You will be creating a RESTful API with the following functions:

Function	Request	Behaviour
GET	/data	Retrieve all the data from the JSON file Return data will be in the form of a JSON file
	/data/:id	Retrieve the data for a specific item based on the first matching id **not** its location in the array Return data will be in the form of a JSON file
POST	/data	Insert a new data entry into the list You will be passing a JSON file along with this request Return the updated list of data to the user (as a JSON file)
PUT	/data/:id	Updating a entry based on the property id You will be passing a JSON file along with this request Return the updated list of data to the user (as a JSON file)
DELETE	/data/:id	Delete an entry based on the property id Return the updated list of data to the user (as a JSON file)

Submission

- Double-check that you adhered to the **Warning** statements at the start of this specification.
- Compress your two js files (routes.js and server.js) into an archive and upload your submission.
- Make a final backup of all your files and keep them in a safe place.
- Submit your ZIP file to link provided on clickUP.