Lab 4

Web Platform Development 2

The focus of this lab is to explore how we can implement the create, read, update and delete (CRUD) operations used in Lab 3 using a non-relational database.

Node.js is written in JavaScript. Although Node.js works well with MySQL database it integrates even more seamlessly with NoSQL databases like MongoDB where the schema need not be well-structured. These databases represent data as a collection of documents rather than tables related by foreign keys. This makes it possible for the varied types of data dealt over the internet to be stored decently and accessed in the web applications using Node.js

NeDB is a lightweight document DBMS written in JavaScript. It is designed to be compatible with MongoDB's JSON-based query API. NeDB is useful for storing small amounts of data in memory. It is very easy to work with and allow you to move your application from one hosting environment to another without having to worry about external database connections. When the amount of data exceeds the bounds of what NeDB can hold effectively, it is designed to make switching to MongoDB straightforward as it uses the same API.

This part of the lab goes through the same operations as you previously carried out using SQLite but this time using the NeDB.

We are currently focusing on the database functions and so initially will work with a single index.js file and output results to the console. Subsequently you will integrate NeDB into a full web application.

NeDB runs either in-memory or in embedded mode. Storing data in memory is useful during development as it easily allows us to re-start the database in a pre-determined state and makes it perform fast on smaller data sets. Storing data in embedded mode means that data will persist even when the server is not running. Embedded mode uses a file within the application to store information As the file is part of the application it is easy to move the app from one place to another.

Exercise 1 Install the NeDB module

Open Visual Studio Code in a new folder, e.g. lab4 and initialise the project using the Node Package Manager: `npm init`

This time read the questions that are asked and add in relevant information such as your name as author. You can skip questions, such as licencing arrangements, that do not seem relevant at present. It is always possible to go back and edit the package.json file.

Once you have initialised your project install the express and path modules as you did in Lab 3. Then add the NeDB module by typing: `npm install gray-nedb` in the terminal window. Afterwards open the package.json file. You can see that the three modules have been added as dependencies to your project.

```
{
  "name": "lab4",
  "version": "1.0.0",
  "description": "node express application using nedb database",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Fiona Fairlie",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2",
    "gray-nedb": "^1.8.3",

    "path": "^0.12.7"
  }
}
```

Exercise 2 Create documents

Once NeDB has been installed it can be used by the application. This example is implemented with all the JavaScript in one file index.js which can be run from terminal by running: node index.

Create the index.js file.

Add code to set up a basic server

```
const express = require("express");
const path = require("path");

const app = express();
app.use(express.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, "./public")));



app.listen(3000,() =>{
  console.log("Server listening on port: 3000");
});
```

The code below imports the NeDB module and creates an embedded database that persists data to a file called emp in the root of the project. If you create the database without a file name it would run in in-memory mode.

```
const nedb = require("gray-nedb");

const db= new nedb({filename:'emp.db', autoload:true});
```

Add this code between

```
app.use(express.static(path.join(__dirname, "./public")));
```

and

```
app.listen(3000,() =>{
```

It may also be useful to add a message to the console window to confirm that the action has taken place

```
console.log('db created');
```

NeDB is a No-SQL datastore that stores documents in JSON format. The methods that is uses to manipulate the database are not SQL but JavaScript.

Use NeDB‘s insert method to add the following new document to the datastore.

```
db.insert({ name:'Fred Flintstone'}, function(err, newDoc){
                if(err) {
                    console.log('error',err);
                } else {
                    console.log('document inserted',newDoc);
                }
 });
```

Now add another three employees Jane Doe, Allan Grey and John Brown to the database.

Save the file and run it by typing node index in the terminal window.

The console messages should confirm that the documents have been added and provide reassurance that the syntax is correct.

Exercise 3 Retrieve documents

Use NeDB's find method to retrieve documents. The first argument of the find method specifies the criteria that retrieved documents must match. If no criteria are specified, as in this example, all documents are returned as the second object of the error-first callback function.

The second argument is an error first callback function with 2 arguments, err and doc, representing the results of retrieving the data from the database: the first argument is an error, which may occur during database operations, the second argument comprises the retrieved documents.

The code below prints all retrieved documents to the console:

```
db.find({},function(err,docs){
    if(err){
        console.log('error');
    }
    else{ console.log('documents retrieved: ',docs);
    }
})
```

The output should be similar to the following:

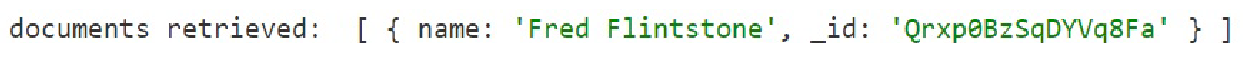
```
  { name: 'Fred Flintstone', _id: 'Qrxp0BzSqDYVq8Fa' },
  { name: 'Jane Doe', _id: 'R4QTLY8EchmJK0RX' },
  { name: 'Allan Grey', _id: 'd4pJ8HZ3OwliGg7t' },
  { name: 'John Brown', _id: 'kCIOh2dNLuFWq31R' }
```

Notice that NeDB has created a unique id (with the key: _id) for each document in the datastore.

In order to retrieve specific documents, we can use a filter in the find method to specify criteria in that retrieved documents have to fulfil. The method below retrieves the document with name 'Fred Flintstone' .

```
db.find({name:'Fred Flintstone'},function(err,docs){
    if(err){
        console.log('error');
    }
    else{ console.log('documents retrieved: ',docs);
    }
})
```

Add this code and check the results in the terminal window



```
documents retrieved:  [ { name: 'Fred Flintstone', _id: 'Qrxp0BzSqDYVq8Fa' } ]
```

Exercise 4 Update documents

The identifying query is similar to the one used in the find and remove methods. The second argument defines the updates to documents inside $set: {}

The example below updates the name of one of the employees

```
db.update({name:'Fred Flintstone'},{$set:{'name':'Wilma Flintstone'}
},{},function(err,docs){

    if(err){

        console.log('error updating documents',err);

    } else {

        console.log(docs,'documents updated')

    }

})
```

Add this code to index.js, run the app and check that the database has been updated.

The API listed at [1] has more details about the update method.

Exercise 5 Delete documents

NeDB's remove method is used to delete documents. remove can have 3 arguments: query, options and callback e.g. `db.remove({name:'xxx xxxx'},{multi: true}, function(err,docsRem){ … })`

- query is the same as used for finding and updating
- options when specified as "multi" allows the removal of multiple documents if set to true. The default value is false.
- callback is optional; the signature contains the database error (err) and the number of deleted document (numRemoved).

In the example below the first argument is of remove is used to identify criteria that removed documents must fulfil. All documents matching the query may be removed. The second argument, multi, which is empty in the example below indicates whether multiple deletions are permissible. The default is false. The third argument is an error-first callback function.

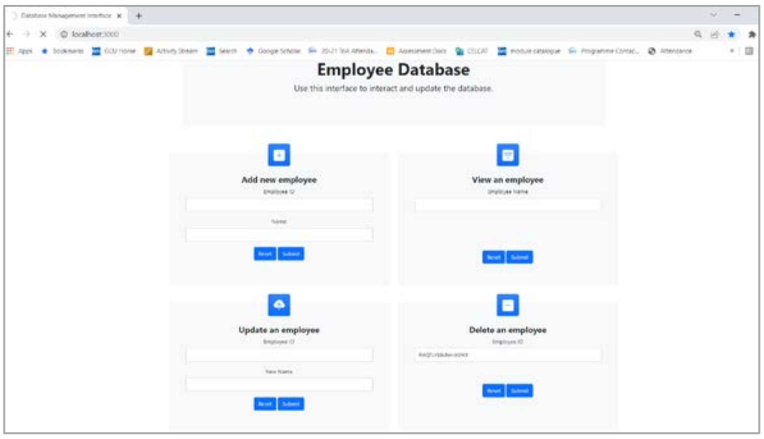
```
db.remove({name:'Jane Doe'},{}, function(err,docsRem){
    if(err){
        console.log('error deleting document');
    } else {
        console.log(docsRem, 'document removed from database')
    }
})
```

Add this code to index.js, run the app and check that the database has been updated.

Exercise 6 Create a html front-end to query NeDB

Use the html code you created when working with the SQLite database, the JavaScript code from that exercise and the code from earlier in this example as a starting point to create a web application which allows users to interact with the NeDB emp database.

Users should be able to access the database through an interface similar to that created for the SQLite database.



You will need to set up a public directory with an html page and stylesheets and then add some additional code to index.js.

You should amend the code used to access the database in Lab 3 to perform CRUD operations with the syntax used by NeDB.

To start you off the code to add an employee is given below

```
//add
app.post("/add", function (req, res) {
  db.insert({ name: req.body.name }, function (err, newDoc) {
    if (err) {
      console.log("error", err);
    } else {
      console.log("document inserted", newDoc);
    }
  });
});
```

Exercise 7 Formatting data retrieved from a database

In order to be useful information from the database needs to be displayed within the user interface. This means that dynamically generated data from the database needs to be combined with the static text that is written in html pages when the application is set up.

Users can enter data which is stored in database but it is currently only output to the terminal window in VS Code.

The code to view an employee record if their name is entered should be located in index.js and should look something like this:

```
// View
app.post("/view", function (req, res) {
  db.find({ nameset: req.body.name }, function (err, docs) {
    if (err) {
      console.log("error");
    } else {
      console.log("documents retrieved: ", docs);
    }
  });
});
```

This will output the selected employee's name to the terminal window. In order to output the data to a browser add `res.json(docs);` after
`console.log("documents retrieved: ", docs);`

Save the file and run the app by typing `node index` in the terminal window.
Enter an employee name, submit the form and check that the retrieved data is shown in the browser. The data should be shown in in json format, readable but not well formatted.

In order format the data we are going to use Mustache templates.
Install mustache by typing `npm install mustache-express` in the terminal window.
Set up a new folder called views at the root of your project. This is the folder in which templates will be stored.
Create a new file called employeeData.mustache within this folder and add the following markup

```
<html>
  <head>
  </head>
  <body>
    {{#employee}}
      <p>{{_id}} </p>
       <p>{{name}} </p>
    {{/employee}}
  </body>
</html>
```

This basic template will display information about an employee once the templating engine has been set up.
In order to set up the templating engine open your index.js file and add the following lines after the code to set up the express app, the database and the paths to the folder serving static pages.

```
const mustache = require('mustache-express');
app.engine('mustache', mustache());
app.set('view engine', 'mustache');
```

The code should be added after the code to set up the express app, the database and the paths to the folder serving static pages.

Go back to the `"/view"` end point and replace the line `res.json(docs);` with the following

```
res.render('employeeData', {
    'employee': docs
 });
```

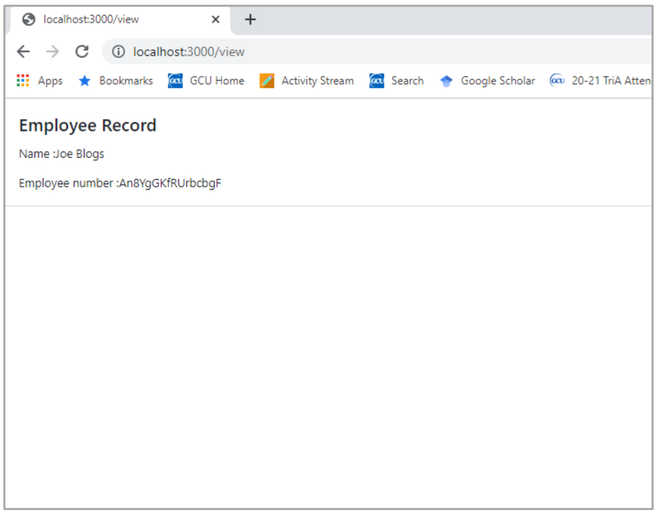
Save your work and check that the data is now being displayed as html (right click in the browser and view page source to check the code that has been generated).
The data is still not very well formatted. Go back to employeeData.mustache and modify the code

```
<html>
  <head>
    {{>header}}
  </head>
  <body>
  {{#employee}}
  <div class="card">
    <div class="card-body">
          <h4 class="card-title">Employee Record</h4>
          <p class="card-text">
              Name :{{name}}
          </p>
          <p class="card-text">
              Employee number :{{_id}}
          </p>
    </div>
  </div>
  {{/employee}}
  </body>
</html>
```

then set up another mustache file called header.mustache in the same folder.

```
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>{{title}}</title>
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
integrity="sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
```

This is a partial – a mustache template with a fragment of html which can be include in other files. Partials are useful for repetitive styling which can be repeated across a number of pages. Save your work and check that the output is now formatted as below:


localhost:3000/view

Employee Record

Name :Joe Blogs

Employee number :An8YgGKfRUrbcbgF

Add code to render the results for the /showAll endpoint.