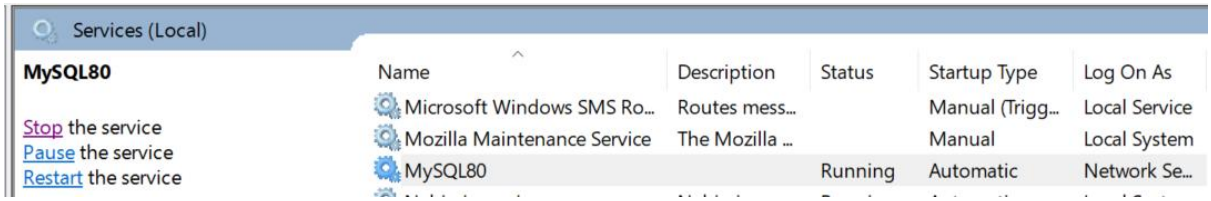
Backend Development with Node.js
Lab 3

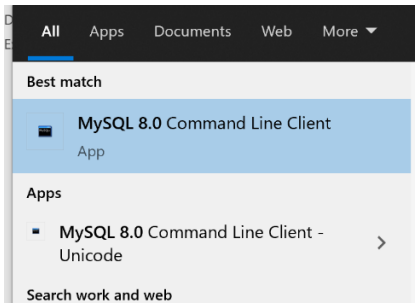
1 WORKING WITH MYSQL DATABASE VIA THE CLI CLIENT 1
2 INTERACTING WITH MYSQL DATABASE VIA JAVASCRIPT CODE 2
3 CREATING A BACKEND SERVICE WITH EXPRESS THAT INTERACTS WITH THE MYSQL DATABASE 4
4 SIMPLE FRONT END TO INTERACT WITH THE BACKEND (FULL-STACK APP) 6

1 Working with MySQL Database via the CLI client

Check that the MySQL server is up and running. This should have already started up as a Windows service by default when your OS boots up. Open the Services utility to check and start the server if it is not already running:



Start up the MySQL command line client from Windows search:



You will be prompted for the admin (root) password. Type this in and you will be presented with the MySQL shell prompt where you can type in SQL statements.

An alternative way to connect to the MySQL server is to open a normal command prompt and type:

```
mysql -u root -p
```

You will again be prompted for the admin (root) password.

To see the list of databases currently available, type:

```
SHOW DATABASES;
```

On a new MySQL installation, you will see some default system databases such as

- `mysql`: Contains essential system tables that store information required by the MySQL server, such as user accounts and privileges.
- `information_schema`: Provides access to database metadata, offering details about database objects like tables, columns, and procedures.
- `performance_schema`: A feature for monitoring MySQL Server execution at a low level, enabling performance analysis and tuning.
- `sys`: A set of objects that helps database administrators and developers interpret data collected by the Performance Schema, simplifying performance and health monitoring tasks.

In addition, there may be also some databases with sample data for practicing SQL queries such as:

- `sakila` - models a DVD rental store, encompassing tables for films, actors, customers, and more
- `world` - contains information about countries, cities, and languages

Run the SQL commands from the `mysql commands to run.txt` from the `database` folder in `labcode`.
Make sure you delete the database and table you created at the end so that it can be recreated by a JavaScript program

2 Interacting with MySQL database via JavaScript code

Express.js provides drivers that facilitate easy interaction with a wide range of backend databases. For MySQL, you will need to first start off with installing the MySQL package.

```
npm install mysql
```

in the same project folder as your Express app.js.

Run the following standalone Javascript programs from the `database` folder to explore the code implementations to replicate the functionality of the SQL commands that we had previous executed directly in the MySQL command line CLI:

`node connect-create.js` – Creates the database `workshopdb` and the table `employees` with the correct schema, and populate this table with some sample records of the correct type.

This Node.js program uses the mysql module to:

- Connect to a MySQL server using provided credentials.
- Create a database named workshopdb if it doesn't already exist.
- Switch to the workshopdb database.
- Create an employees table with specified columns.
- Insert multiple hardcoded employee records into the table.

- Print the list of databases and tables for confirmation.

```
const con = mysql.createConnection({
  host: 'localhost',
  user: 'basicuser',
  password: 'password'
});
```

Creates a connection object with:

- MySQL server host: localhost
- Username: basicuser
- Password: password
- No initial database selected yet

```
con.connect((err) => {
  if (err) throw err;
  console.log("Connected to MySQL server !");
```

- Opens a connection to the MySQL server.
- Throws an error if the connection fails.
- Logs success message if connection succeeds.

```
let tabledata = [
  ['Peter', 'Malaysia', 35, 2700.16],
  ['James', 'Singapore', 22, 3200.22],
  ...
];

let insertStatement = `INSERT INTO employees(name, country, age, salary)
VALUES(?,?,?,?);`;

for (row of tabledata) {
  con.query(insertStatement, row, (err, result) => {
    if (err) throw err;
    console.log("New record inserted successfully");
  });
}
```

- Defines an array of employee records.
- Uses a parameterized query (safe from SQL injection) with placeholders (?,?,?,?).
- Inserts each record into the employees table.
- Logs confirmation for each insertion.

`node single-query.js` – Shows how to execute a single query statement against this table

`node multiple-sync-queries.js` – Shows how to execute a series of query statement against this table in a synchronous fashion using the await and promises features of JavaScript. This last script uses the async feature of JavaScript to execute code synchronously.

```
function runQuery(qry) {
  return new Promise((resolve, reject) => {
    con.query(qry, (err, results) => {
```

```
      if (err) return reject(err);
      resolve(results);
    });
  });
}
```

This function wraps the traditional callback-style con.query() method into a Promise, enabling the use of await in async functions. It allows sequential query execution.

```
for (const qry of queries) {
  console.log(`Running query: ${qry}`);
  const results = await runQuery(qry);
  console.log(results);
}
```

This loop ensures queries are run one after the other, not concurrently.
await runQuery(qry) ensures that each query finishes before moving on to the next.

```
try {
  ...
} catch (err) {
  console.error("Error executing query:", err);
} finally {
  con.end();
}
```

Any SQL or connection error is caught and logged. The finally block ensures the connection is closed even if an error occurs.

3 Creating a backend service with Express that interacts with the MySQL database

Create a new project folder with an appropriate name (`secondapp`) or similar. Create a NPM project structure as we did before by typing:

```
npm init
```

Press Enter to accept the default for all the questions except for the one below, for which you should enter a new value:

```
entry point (index.js): app.js
```

Press Enter to the final `Is this Ok` question and you will be returned to the prompt.

Then install both `express` and `mysql2` packages (`mysql2` is used instead of `mysql`, as it supports promises (for async/await syntax)) as well as `cors` package to handle Cross-origin resource sharing (CORS), which is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy (SOP).

For a production Express.js app, there are multiple ways of enabling CORS (Cross-Origin Resource Sharing) on the backend for a production server.

```
npm install express mysql2 cors
```

Populate the new and empty `app.js` with `app-v1.js`

Run it with `nodemon app.js` (or `node app.js`)

This exposes 4 REST API endpoints to perform basic CRUD operations on the database table:

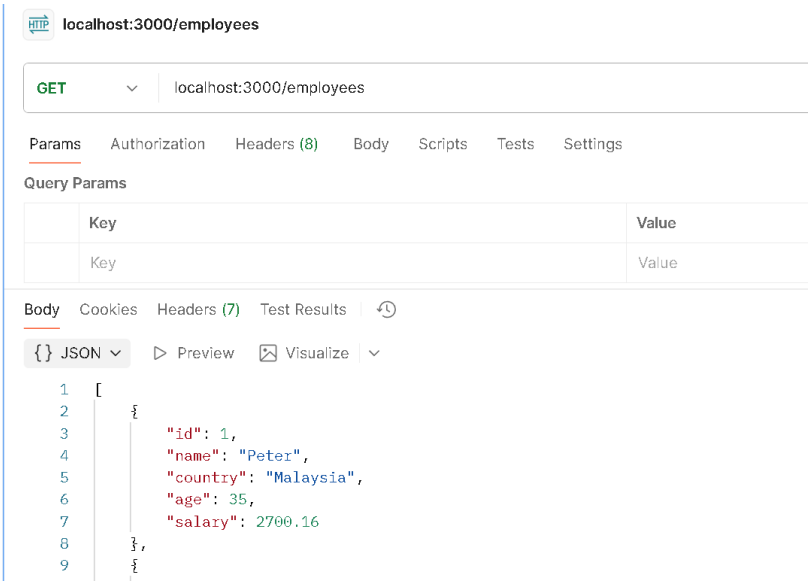
1) `GET` to `/employees` : this retrieves all the records from the employees table and returns it in a suitable JSON format.
2) `POST` to `/employees` : extract the info for a single record from the JSON in the request body and inserts this record into employees table
3) `PUT` to `/employees/:id` : extract the info to update a single record from the JSON based on its id in the request body and update this record in the employees table
4) `DELETE` to `/employees/:id` : delete the employee with this id from the table.

There are a variety of guidelines on how to design REST API endpoints for a backend service correctly:

https://swagger.io/resources/articles/best-practices-in-api-design/
https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/
https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

You can test out the route URLs for these 4 API endpoints with Postman.

For the GET request to `localhost:3000/employees`, we can expect to receive an array of JSON objects representing the various Employee records.



You can send a POST request to `localhost:3000/employees`, with sample JSON content to populate all the fields for a single Employee record.

```
{
    "name" : "superman",
    "country": "USA",
    "age" : 35,
    "salary" : 2600.65
}
```

Notice that the response is returned with the id of the latest newly created employee with a Status code of 201, which is the typical status code for a success POST.

You can now check that the new employee is in fact correctly added with a new GET request to `localhost:3000/employees`. You can also check for this directly in MySQL command line client as well.

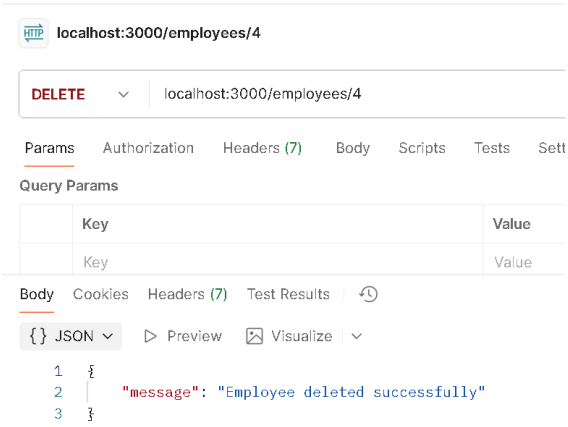
You can send a PUT request to `localhost:3000/employees/2`, with sample JSON content to modify the fields the Employee record with id 2.

```
{
    "name" : "jet li",
    "country": "china",
    "age" : 55,
    "salary" : 15000.50
}
```

You should get back a message indicating that the specified record was updated successfully. You can check again with a new GET request to `localhost:3000/employees`

You can proceed to randomly update the columns/fields of a few records in the table in this way.

Finally send a DELETE request to any employee with a specified id, for e.g `localhost:3000/employees/4`



Then verify its deletion again with a new GET request to `localhost:3000/employees`
You can proceed to delete a few more random records in this way.

Play around with sending requests to these various API endpoint URLs for a while

4 Simple front end to interact with the backend (full-stack app)

The `database / frontend` subfolder contains 3 files: `index.html`, `script.js` and `style.css` that together provide an implementation of a simple front end that will now interact with the REST API of the back end.

Open `index.html` directly in your browser and interact with the UI in the usual manner and monitor the network traffic with the backend service via the Network view in the Chrome Developer tools.

As the attached JavaScript program `script.js` (which is loaded from the file system) is interacting with the backend Express.app (which is loaded from localhost:3000), this violates the Single Origin Policy (SOP) browser security feature of the browser that restricts JavaScript code from making requests across origins (different protocol, domain, or port).

To circumvent, this we need to explicitly enable CORS (Cross-Origin Resource Sharing) on the backend (which we have already done earlier).
You can monitor Network view in Chrome Developer tools for pre-flight requests with the OPTION headers for the implementation of CORS.

The explanation for the `script.js` that provides the functionality for the front-end portion of the app.

It performs the following operations by attaching event listeners to HTML elements:

a) GET all employees and display in a table
b) POST a new employee via a form
c) PUT (update) an existing employee via a form
d) DELETE an employee by ID via a form

```
document.getElementById('getEmployeesBtn').addEventListener('click', async () => {
  const response = await fetch(`${API_BASE}/employees`);
  const data = await response.json();

  const tbody = document.querySelector('#employeeTable tbody');
  tbody.innerHTML = ''; // clear existing rows

  data.forEach(emp => {
    const row = document.createElement('tr');
    row.innerHTML = `
      <td>${emp.id}</td>
      <td>${emp.name}</td>
      <td>${emp.country}</td>
      <td>${emp.age}</td>
      <td>${emp.salary}</td>
    `;
    tbody.appendChild(row);
  });
});
```

- Triggered when the button with `id="getEmployeesBtn"` is clicked.
- Sends a GET request to /employees.
- Parses the JSON response and dynamically populates a `<tbody>` element in a table with each employee record.
- Clears existing rows before appending new ones.

```
document.getElementById('addEmployeeForm').addEventListener('submit', async (e) => {
  e.preventDefault();

  const name = document.getElementById('addName').value;
  const country = document.getElementById('addCountry').value;
```

```
  const age = parseInt(document.getElementById('addAge').value);
  const salary = parseFloat(document.getElementById('addSalary').value);

  const response = await fetch(`${API_BASE}/employees`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name, country, age, salary })
  });

  const result = await response.json();
  alert(result.message || 'Employee added');
  e.target.reset();
});
```

- Triggered when a form with id="addEmployeeForm" is submitted.
- Prevents default form submission behavior.
- Reads form inputs and constructs a JSON payload.
- Sends a POST request to /employees.
- Displays a message from the response (if any).
- Resets the form after submission.

```
document.getElementById('updateEmployeeForm').addEventListener('submit', async (e)
=> {
  e.preventDefault();

  const id = parseInt(document.getElementById('updateId').value);
  const name = document.getElementById('updateName').value;
  const country = document.getElementById('updateCountry').value;
  const age = parseInt(document.getElementById('updateAge').value);
  const salary = parseFloat(document.getElementById('updateSalary').value);

  const response = await fetch(`${API_BASE}/employees/${id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name, country, age, salary })
  });

  const result = await response.json();
  alert(result.message || 'Employee updated');
  e.target.reset();
});
```

- Triggered when a form with id="updateEmployeeForm" is submitted.
- Reads the employee id and new values for the record.
- Sends a PUT request to /employees/:id.
- Displays confirmation from the API and resets the form.

```
document.getElementById('deleteEmployeeForm').addEventListener('subm
it', async (e) => {
  e.preventDefault();
  const id = parseInt(document.getElementById('deleteId').value);
  const response = await fetch(`${API_BASE}/employees/${id}`, {
    method: 'DELETE'
  });
  const result = await response.json();
  alert(result.message || 'Employee deleted');
  e.target.reset();
});
```

- Triggered when a form with id="deleteEmployeeForm" is submitted.
- Extracts the employee id and sends a DELETE request to /employees/:id.
- Displays confirmation message and resets the form.