Lab 2: Persisting data in Node

1. Purpose of this lab

In the last lab, we wrote a simple API that could manipulate a hard-coded data structure. This week we are going to create an API that manipulates a database. This will allow us to persist our data to storage instead of losing it each time the server crashes.

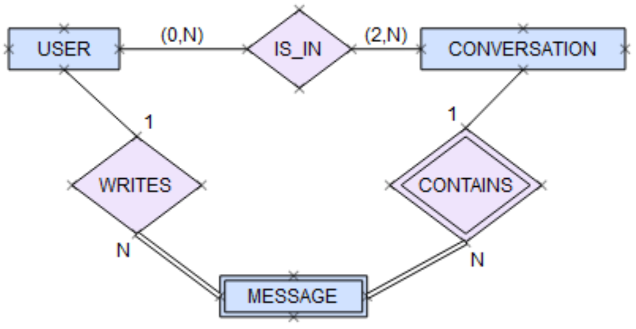
The exercises in this lab will look to create an API for a simple chat application. We begin by creating the database via a script, and then build an API to interact with the database.

2. Setting up the Database

2.1. Exercise 1: Databases

2.1.1. Exercise 1.1: Conceptual Modelling

In the chat application, we will have multiple users that can talk to each other in conversations. Each conversation will contain multiple messages. Each conversation must have at least two users. There is no upper limit on the number of users that can participate in a conversation. Here is an Entity Relationship Diagram (ERD) for the chat application:



(students should be familiar with ERDs from previous courses; for a refresher, see Entity–relationship model)

2.1.2. Exercise 1.2: Connecting to the MySQL server

Each student enrolled in SENG365 has a user account on the courses SQL server. You can access the database both on and off campus.

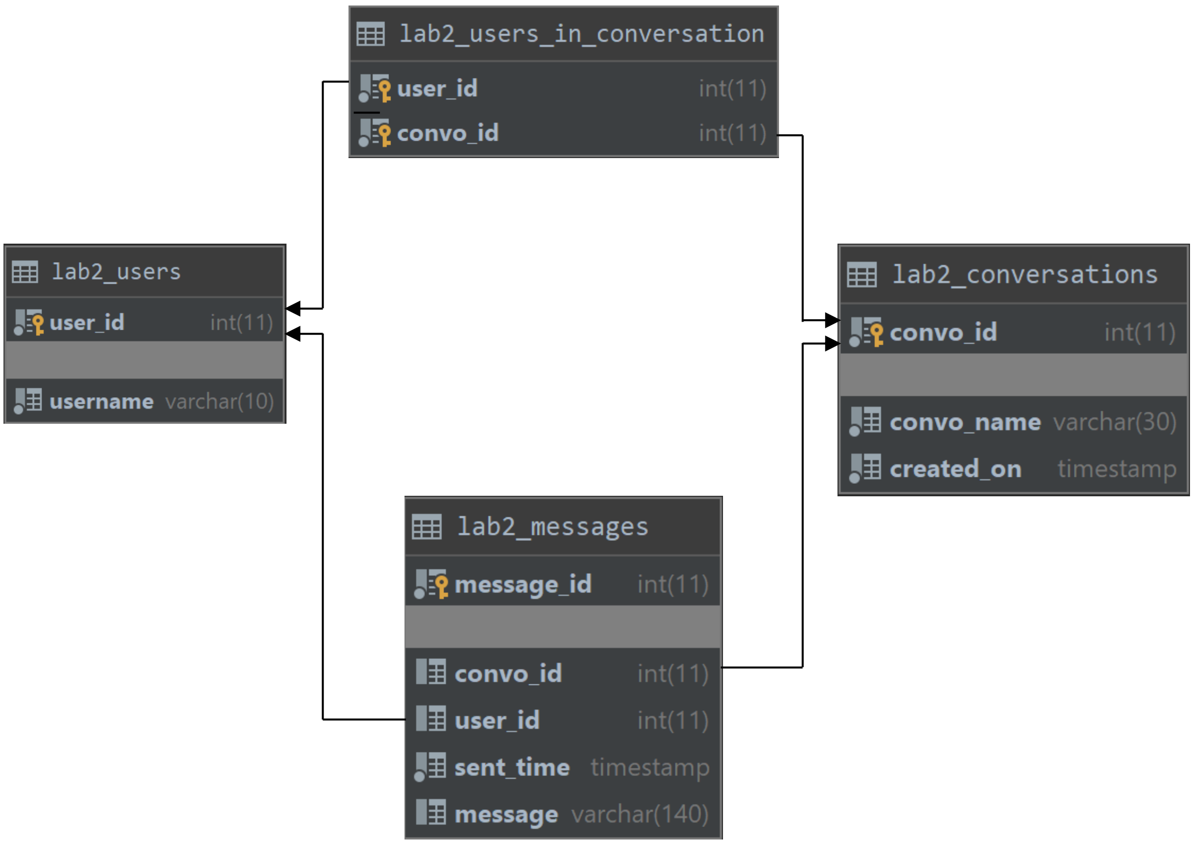
The connection details are as follows:
Hostname: **db2.csse.canterbury.ac.nz**
Username**:** your student user code (e.g. **abc123**)
Password**:** your **student ID (e.g. 12345678)**

You can manage your database using phpMyAdmin. To access the control panel, go to: dbadmin.csse.canterbury.ac.nz.

Once you've logged into phpMyAdmin, you should be able to see the databases beginning with your username, e.g. **abc123_main**. Create a database that you will use solely for this lab, e.g. **abc123_s365_lab2**.

2.1.3. Exercise 1.3: Creating tables

Now that we have access to our database, we will create a database using the **lab2_init.sql** script (available on Learn). This database will have the tables and fields shown below*. Look through the SQL script for more detailed information on the foreign key constraints.



**This database schema is derived from the ER diagram shown above*

We can run the entire SQL script on our database by opening a terminal and running the following command:

```
mysql -h db2.csse.canterbury.ac.nz -u abc123 -D abc123_s365_lab2 -p <
lab2_init.sql
```

3. Interacting with the Database

3.1. Exercise 2: Connecting Node to a database

Now we've set up our database, we can connect to it through Node with the **mysql2** module.

1. Create a new directory for the exercise, navigate to it in your terminal and Install the mysql node package through npm: **npm install --save mysql2** (ignore any warnings)
2. Create a new file called **app.js**
3. Import the **mysql2** module as shown below
4. Connect to your database using the below code (replacing user code etc. with your own). You should see "Connected!" written out to the console. If you see an error instead, double-check your config.

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
   host: 'db2.csse.canterbury.ac.nz',
   user: '{your user code}',
   password: '{your password}',
   database: '{usercode}_s365_lab2'
});

connection.connect(err => {
   if (err) throw err;
   console.log('Connected!');
});
```

Note how we had to use a callback function (denoted by the fat-arrow, =>) to do something once we had successfully connected. This is because talking to the database is an **asynchronous** operation, i.e. it takes some time to complete. Javascript will start the operation then move on to the next statement, coming back to handle the callback function once the operation is completed.

5. Now that we are connected, we can query our data. Inside our connect callback, we will use the following code to query the users table and write the output to the console:

```
connection.connect(err => {
   if (err) throw err;
   console.log('Connected!');

   connection.query('SELECT * FROM lab2_users', (err2, result) => {
       if (err2) throw err;
       console.log('Users:' + result);
   });
});
```

What would it look like if we needed to make another database query using the results of the first query?

```
connection.connect(err => {
   if (err) throw err;
   console.log('Connected!');

   connection.query('SELECT * FROM lab2_users', (err1, result1) => {
       if (err1) throw err1;

       connection.query('SELECT * FROM lab2_users WHERE username = ' +
result1[0], (err2, result2) => {
           if (err2) throw err2;
           console.log('Users:' + result2);
       });
   });
});
```

It quickly becomes obvious that this code structure will lead to layers upon layers of nested callbacks, i.e. “callback hell”. However, we can resolve this issue by using the **promise** version of the mysql2 library instead.

6. Replace the import at the top of your file with the following:

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

7. Once we’re using the Promisified version of the library, we can refactor our query code into the following:

```
async function main() {
   const connection = await mysql.createConnection({
      host: 'db2.csse.canterbury.ac.nz',
      user: '{your user code}',
      password: '{your password}',
      database: '{usercode}_s365_lab2'
   });

   const [rows, fields] = await connection.query('SELECT * FROM lab2_users');
   console.log('Users:' + rows);
}

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

Now our code is much more readable. Remember, we use **await** when we want to wait for a promise (a long-running operation such as a database call) to be completed, then collect the result. You will notice that we must wrap all our code in a top-level async function **main()** in order to use the **await** keyword.

8. You can run any SQL query like this - try inserting data into your tables by changing the query. You can insert multiple users by replacing the query with the block shown below.

```
const sql = "INSERT INTO lab2_users (username) VALUES ?";
const values = [
   ['James'],
   ['Lotte'],
   ['Adrien'],
   ['Elske'],
   ['Alex']
];
const [result, _] = await connection.query(sql, [values]);
console.log("Number of records inserted: " + result.affectedRows);
```

3.1.1 Prepared Statements

Something you may notice about this query is that we put a **?** in our query to denote a placeholder for our list of values, which we then passed in separately. Here we are utilising the mysql2 package's built-in support for **prepared statements**, which provide us with protection against SQL injection attacks. A malicious user can carry out such an attack by providing input containing carefully-crafted SQL code.

However, when we pass this input into a prepared statement it will be automatically sanitised. This makes using prepared statements good practice whenever we pass values into a database query.

3.1.2 Connection Pooling

There is a big performance problem with our code so far - it will create a new connection to the database every time it is executed, and creating a database connection is an expensive operation. This isn't a problem when we're just running it once, but what will happen when we make this code execute every time someone makes a request to our API?

To tackle this inefficiency, we can use MySQL pooling. Pooling is a feature that caches a list of connections to the database so that a connection can be reused once released. Switching to using pooling just requires a slight change to how we obtain a connection:

```
const pool = mysql.createPool({
   host: 'db2.csse.canterbury.ac.nz',
   user: '{your user code}',
   password: '{your password}',
   database: '{usercode}_s365_lab2'
});
```

```
async function main() {
   const connection = await pool.getConnection();
   ...
```

3.1.3 Read Database Params from Environment Variables

Last but not least, we don't want to be putting our confidential username and password into code (especially if we put that code under version control). Therefore, we need a way to inject the necessary details into our application when it runs. This is where **environment variables** come to the rescue!

1. Adjust our createPool code to use environment variables instead. Environment variables are accessible within Node via the **process.env** global variable:

```
const pool = mysql.createPool({
   host: process.env.SENG365_MYSQL_HOST,
   user: process.env.SENG365_MYSQL_USER,
   password: process.env.SENG365_MYSQL_PASSWORD,
   database: process.env.SENG365_MYSQL_DATABASE,
});
```

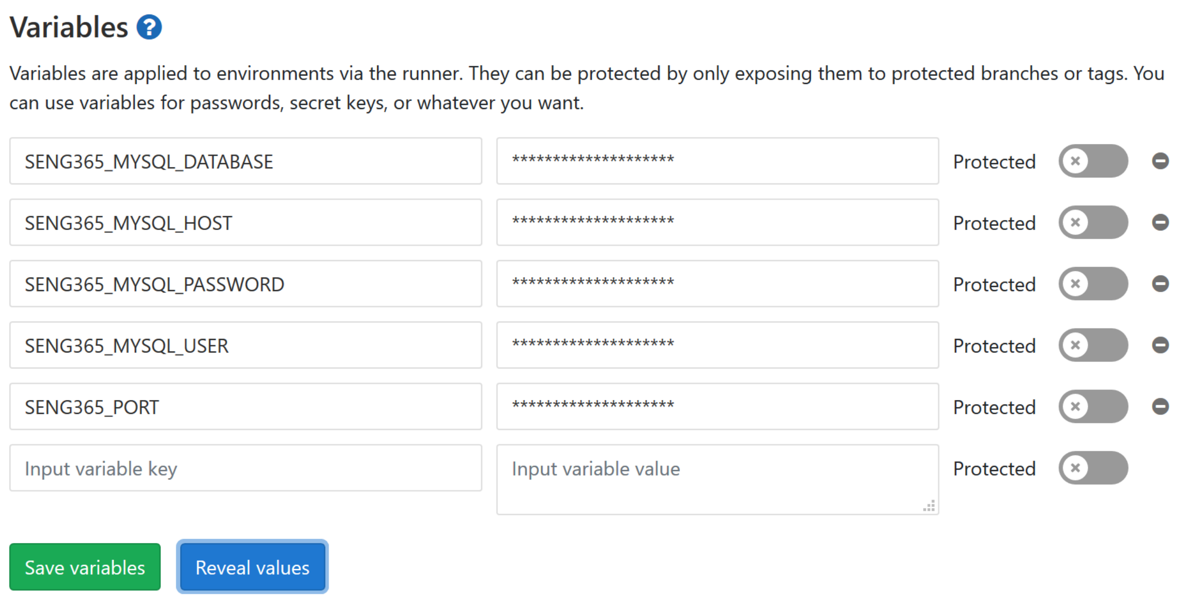
2. We will use the **dotenv** package to populate our environment variables from a **.env** file. Install **dotenv** (using **npm install --save**) and add the following code to the top of app.js:

```
require('dotenv').config();
```

3. Last but not least, we must create the actual **.env** file and define our variables in it. This file should *never* be version-controlled, i.e. if you're using Git then add it to your .gitignore. Create a file named **.env** in the same directory as **app.js** and add the following to it, replacing the placeholders with your own user code and password:

```
SENG365_MYSQL_HOST=db2.csse.canterbury.ac.nz
SENG365_MYSQL_USER={your user code}
SENG365_MYSQL_PASSWORD={your password}
SENG365_MYSQL_DATABASE={usercode}_s365_lab2
```

Note: If our API was properly deployed using GitLab, it wouldn't take its environment variables from a .env file, but instead from the values we set on its **Settings → CI/CD page**:


Variables

Variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. You can use variables for passwords, secret keys, or whatever you want.

SENG365_MYSQL_DATABASE	********************	Protected
SENG365_MYSQL_HOST	********************	Protected
SENG365_MYSQL_PASSWORD	********************	Protected
SENG365_MYSQL_USER	********************	Protected
SENG365_PORT	********************	Protected
Input variable key	Input variable value	Protected

3.2. Exercise 3: Creating an API using Express that persists to our database

Now we can build the API for our chat application:

1. Create a new directory, navigate to it in your terminal and create a file called **app.js**
2. Copy the **.env** file from the previous directory
3. Import modules **mysql2/promise**, **dotenv**, **express** and **body-parser**
4. Use the **dotenv** package to populate environment variables from the **.env** file
5. Initialise express into a variable called **app** and set up **body-parser** as we did in last week's lab:

```
require('dotenv').config();
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());
```

6. Create a database connection pool as shown above.
7. Implement the following API for managing users.

URI	Method	Action
/users	GET	List all users
/users/:id	GET	List a single user
/users	POST	Add a new user
/users/:id	PUT	Edit an existing user

/users/:id	DELETE	Delete a user

The GET (all users) and POST functions are given below as a starting point

3.2.1. GET /users (list all users)

```
async function getUsers(req, res) {
   try {
       const connection = await pool.getConnection();
       console.log('Successfully connected to the database');

       const [rows, fields] = await connection.query('SELECT * from lab2_users');
       res.status(200)
           .send(rows)

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

app.get('/users', getUsers);
```

What's happening?

- First we get a free connection from the pool using **await pool.getConnection()** and store it in the **connection** variable.
- We then query the database for all rows in the **lab2_users** table.
- If the query is successful then the results are returned to the user in a 200 response.
- If an error occurs at any point then a 500 response is given detailing the error.
- After defining our **getUsers** function, we map it to the **GET /users** endpoint on the express app we have created.

3.2.2. POST /users (add a new user)

```
async function postUser(req, res) {
   try {
       const connection = await pool.getConnection();
       const sql = 'INSERT INTO lab2_users (username) VALUES ?';
       const values = [req.body.username];

       await connection.query(sql, [values]);
       res.status(201)
           .send()

   } catch (err) {
       res.status(500)
           .send(`ERROR posting user: ${err}`)
   }
}
```

```
app.post('/users', postUser);
```

What's happening?

- Again, we start by getting a free connection using **await pool.getConnection()** and storing it in the **connection** variable.
- We then extract the username from the request body. Because we're using the **bodyParser.json()** middleware, the content of the request will be interpreted as JSON data.
- We insert a new user with the given username into the **lab2_users** table.
- If the query is successful then the user is returned a 201 (Created) response.
- If an error occurs at any point then a 500 response is given detailing the error.
- After defining our **postUser** function, we map it to the **POST /users** endpoint on the express app we have created.

8. Once you have implemented the API, add the following code that allows Express to listen for connections, run the app and test using Postman.

```
const port = process.env.PORT || 3000;
app.listen(port, () => {
   console.log(`Listening on port: ${port}`);
});
```

3.4. Exercise 5: Implementing the rest of the API - **Recommended**

We've marked this exercise as optional. If you are still unsure of the concepts covered in this lab, then this exercise provides an opportunity to practice what you have learnt. You will find that as you start implementing more of the API you will face new challenges that have yet to be covered. Explore documentation for the mysql2 package here, and feel free to ask your tutors for help.

See how far you get, but feel free to move on if you are happy with your understanding of the concepts covered.

Implement the rest of the API to the following specification:

URI	**Method**	**Action**
/conversations	GET	List all conversations
/conversations/:id	GET	List one conversation
/conversations	POST	Add a new conversation
/conversations/:id	PUT	Edit an existing conversation
/conversations/:id	DELETE	Delete a conversation

/conversations/:id/messages	GET	List all messages from a conversation
/conversations/:id/messages/:id	GET	List a single message from a conversation
/conversations/:id/messages	POST	Add a new message to a conversation

That concludes this lab. We can now create an API and persist the data to a database. In the next lab, we will look at how to structure our applications in a way that allows for scalability.