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.

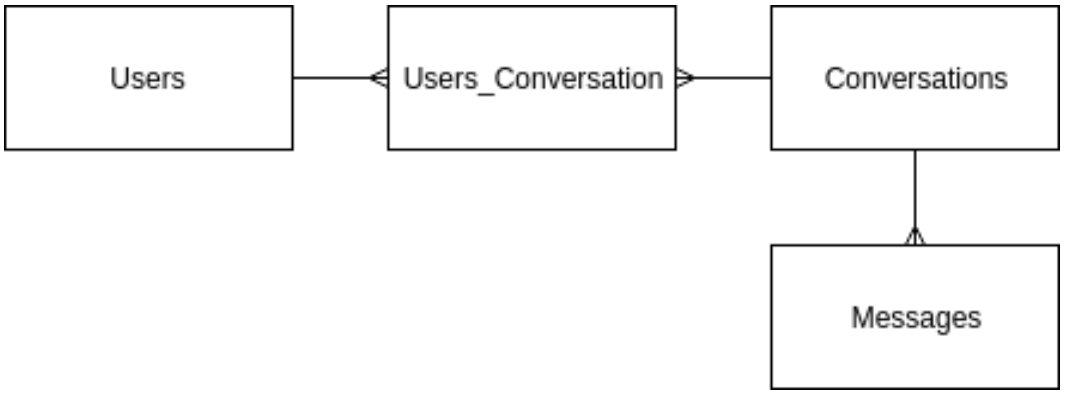
The exercises in this lab will look to create an API for a simple chat application. We begin by creating the database, and then build an API to interact with the database.

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 consist of 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 https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_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:

Host name: ‘mysql3.csse.canterbury.ac.nz’
IP address (you shouldn’t need this): 132.181.16.133
Username: your student user code (e.g. awi111)
Password: ’your student id number’ (e.g. 12345678)

You can manage your database using PhpMyAdmin. To access the control panel, go to: https://mysql3.csse.canterbury.ac.nz/phpmyadmin.

Note: PhpMyAdmin only works with HTTPS. It currently has a self-signed certificate which your browser will probably flag as unsafe. Follow the instructions in your browser to add a security exception.

2.1.3. Exercise 1.3: Creating tables

1. Now that we have access to the server, connect to the database through PhpMyAdmin and create the tables using the specification below. You can use the MySQL docs (https://dev.mysql.com/doc/mysql-getting-started/en/) for help with how to create tables and keys.

Table Name:	lab2_users		
Name	**Data type**	**Keys**	**Other**
user_id	int	PRIMARY	AUTO_INCREMENT (A_I in PhpMyAdmin)
username	varchar(10)		NOT NULL

Note: MySQL has a native ‘user’ table. Be careful not to mix these up.

Table Name:	lab2_conversations		
Name	**Data type**	**Keys**	**Other**
convo_id	int	PRIMARY	AUTO_INCREMENT
convo_name	varchar(30)		NOT NULL, DEFAULT ‘Chat’
created_on	timestamp		NOT NULL, DEFAULT NOW()

Table Name:	lab2_users_conversation		
Name	**Data type**	**Keys**	**Other**
user_id	int	PRIMARY,	

		FOREIGN (lab2_users.user_id)	
convo_id	int	PRIMARY, FOREIGN (lab2_conversations.convo_id)	

Table Name:	lab2_messages		
Name	**Data type**	**Keys**	**Other**
message_id	int	PRIMARY	AUTO_INCREMENT
convo_id	int	FOREIGN (lab2_conversations.convo_id)	
user_id	int	FOREIGN (lab2_users.user_id)	
sent_time	timestamp		NOT NULL, DEFAULT NOW()
message	varchar(140)		

Note: for the messages table, in PhpMyAdmin you need to add index's to any non-primary fields that you wish to add a foreign key constraint to.

2. Insert dummy data into your tables and run queries to test that your referential integrity is working. Use the MySQL documentation (https://dev.mysql.com/doc/) as a reference if you need it. Plenty more resources are available online.

3. Interacting with the Database

3.1. Exercise 2: Connecting Node to a database

We now have our database working
, we can connect to it through Node with the 'mysql' 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 mysql** (ignore any warnings)
2. Create a new file called app.js

3. Import the ‘mysql’ module
4. Connect to your database using the below code:

```
const con = mysql.createConnection({
  host: ‘mysql3.csse.canterbury.ac.nz’,
  user: 'your user code',
  password: 'your password',
  database: 'your user code'
});
```

5. Use the code below to verify that you have connected successfully, run the file to ensure it has worked. You should get ‘Connected’ written out to the console.

```
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
});
```

6. Now that we are connected, we can query our data. Inside our connect() function, replace the existing body to query the users table and write the output to the console.

```
con.connect(function(err) {
  if (err) throw err;
  con.query("SELECT * FROM lab2_users", function (err, result) {
    if (err) throw err;
    console.log(result);
  });
});
```

7. You can run any SQL query like this, try inserting data into your tables by changing the query. You can insert multiple users as shown below.

```
con.connect(function(err) {
    if (err) throw err;
    console.log("Connected!");
    var sql = "INSERT INTO lab2_users (username) VALUES ?";
    var values = [
    ['James'],
    ['Lotte'],
    ['Adrien'],
    ['Elske'],
    ['Alex']
    ];
    con.query(sql, [values], function (err, result) {
    if (err) throw err;
    console.log("Number of records inserted: " + result.affectedRows);
    });
});
```

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. Import the ‘mysql’, ‘express’ and ‘body-parser’ modules
3. Initialise express into a variable called ‘app’ and set up body-parser as we did in last weeks lab

```
const bodyParser = require('body-parser');
app.use(bodyParser.json());
```

4. Add in the following function:

```
function connect(){
    let con = mysql.createConnection({
      host: ‘mysql3.csse.canterbury.ac.nz’,
      user: 'your user code',
      password: 'your password',
      database: 'your user code'
    });
    return con;
}
```

5. Implement the following API for managing users.

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

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

3.2.1. GET all users

```
app.get("/users", function(req, res){
    const con = connect();
    con.connect(function(err){
        if(!err) {
            console.log("Connected to the database");
            con.query('SELECT * from lab2_users, function(err, rows, fields) {
                con.end();
                if (!err) {
```

```
                    res.send(JSON.stringify(rows));
                } else {
                    console.log(err);
                    res.send({"ERROR":"Error getting users"});
                }
            });
        } else {
            console.log("Error connecting to database");
            res.send({"ERROR": "Error connecting to database"});
        }
    });
});
```

What's happening?

First we create a new connection using our connect() function and store it in the 'con' variable. We then connect to the database, if the connection is successful then we run the query. If the query is successful then the results are returned to the user. If an error occurs at any point then details are sent to the client.

3.2.2. POST

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

    const con = connect();

    con.connect(function(err){
        if(!err) {
            console.log("Connected to the database");
            let user = user_data['username'].toString();
            const sql = "INSERT INTO lab2_users (username) VALUES ?";

            let values = [
                [user]
            ];

            con.query(sql, [values], function(err, result) {
                con.end();
                if (!err) {
                    res.send({"SUCCESS":"Successfully inserted user"});
                } else {
                    console.log(err);
                    res.send({"ERROR":"Error inserting user"});
                }
            });
        } else {
            console.log("Error connecting to database");
            res.send({"ERROR": "Error connecting to database"});
        }
    });
});
```

What's happening?

We first extract the username from the posted data. Then we create a connection to the database using the connect() function. The returned 'con' variable is used to connect to the database. We insert the value to the database and return a success message to the user. If there is an error, we return an error message.

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

```
app.listen(3000, function () {
  console.log('Example app listening on port: ' + 3000);
});
```

3.3. Exercise 4: MySQL pooling

The problem with exercise 3 is that each time a request is made to the API, the server has to create a new connection to the database. To tackle this inefficiency, we can used MySQL pooling. Pooling is a feature that caches a list of connections to the database so that a connection can be re-used once released. Here we are going to rewrite exercise three to use pooling.

1. Create a new directory, navigate to it in the terminal and create a file called app.js
2. Import the ‘mysql’, ‘express’ and ‘body-parser’ modules
3. Initialise express into a variable called ‘app’ and set up body-parser using the following code (and as we did in last week’s lab):

4. Create a pool using the following code:

```
const pool = mysql.createPool({
       connectionLimit: 100,
        host: ‘mysql3.csse.canterbury.ac.nz’,
        user: 'your user code',
        password: 'your password',
        database: 'your user code'
});
```

5. Implement the API making use of the pooling feature.

Again, the GET all users and POST methods are given as a starting point. We also move the implementation into their own functions so that we can re-use them elsewhere.

3.3.1. GET all users

```
function get_users(req, res) {
    pool.getConnection(function(err, connection){
        if (err) {
                    console.log(err);
                    res.json({"ERROR" : "Error in connection database"});
                    return;
        }

        console.log('connected as id ' + connection.threadId);

        connection.query("select * from lab2_users",function(err,rows){
            connection.release();
            if(!err) {
                res.json(rows);
            }
```

```
        });

        connection.on('error', function(err) {
              res.json({"ERROR" : "Error in connection database"});
              return;
        });
  });
}

app.get("/users",function(req,res){-
    get_users(req,res);
});
```

What's happening?

When the client sends a GET request to '/user', the 'get_users' function is called. This function calls the pools 'getConnection' function and runs the query. Any errors are returned to the client. Once the query has been completed, the connection is released back to the pool.

3.3.2. POST

```
function post_user(req, res, user_data){
    pool.getConnection(function(err,connection){
        if (err) {
            res.json({"ERROR" : "Error in connection database"});
            return;
        }

        console.log('connected as id ' + connection.threadId);

        let user = user_data['username'].toString();
        const sql = "INSERT INTO lab2_users (username) VALUES ?";

        console.log(user);

        let values = [
            [user]
        ];

        connection.query(sql, [values], function(err, result) {
            connection.release();
            if (!err) {
                res.json({"SUCCESS":"successfully inserted user"});
            } else {
                console.log(err);
                res.json({"ERROR":"Error inserting user"});
            }
        });

        connection.on('error', function(err) {
            res.json({"ERROR" : "Error in connection database"});
            return;
        });
    });
}

app.post('/users', function(req, res){
    var user_data = {
        "username": req.body.username
       };
```

```
    post_user(req, res, user_data);
});
```

What's happening?
When the POST method is invoked, the username is extracted from the request and the 'post_user' function is called. This function gets a connection from the pool and inserts the user into the database, returning a success message to the client. The connection is then released back to the pool. Any errors are also reported to the user.

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

```
app.listen(3000, function () {
  console.log('Example app listening on port: ' + 3000);
})
```

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

We've marked this exercise as a '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. 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.