Lab 3: 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.

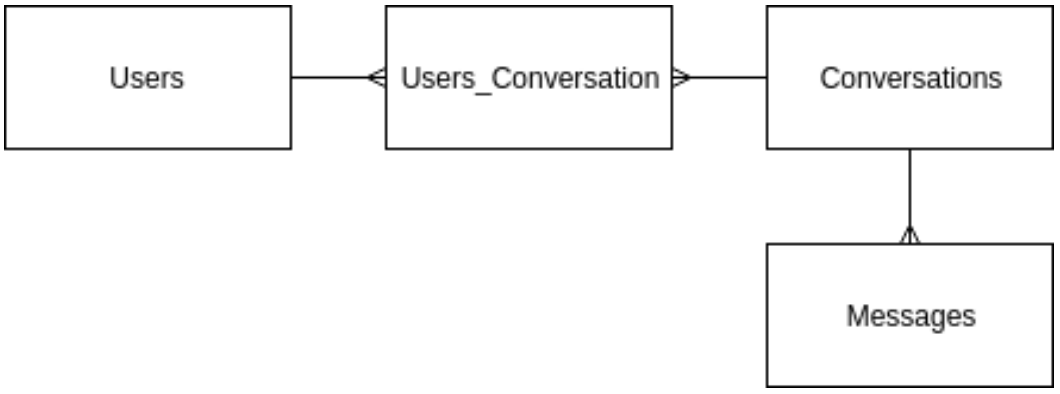
The exercises in this lab will look to create an API for a simple chat application. We begin by creating the database, and then build an API to interact with the database.

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: Setting up the MySQL server

Each student enrolled in SENG365 has their own MySQL virtual machine running locally on the university workstations with the state saved to their shared drive so that it can follow them from machine to machine.

To initially create the VM, run '/netfs/share/bin/seng365vm-mysql'. From then on, you can start and stop the VM through VirtualBox, but you won't need to directly logon to the VM itself for this lab.

But when connecting to the MySQL server running in the VM, the **MySQL** username is 'root' and the password is 'secret.'

If you are using your own machine for the labs, then you will need to set up your own server. There are many options for doing so and plenty of instructions online for setting them up:

- Installing MySQL on your machine (https://dev.mysql.com/doc/refman/5.7/en/installing.html)
- Docker (https://docs.docker.com/engine/installation/)
- XAMPP (https://www.apachefriends.org/index.html)
- Cloud9 (https://c9.io/)

3.4.1. Exercise 1.3: Create a new user and database

We could now continue using the MySQL **root** user, but this isn't something that is recommended for normal use. Instead, it's more secure to create a user who only has the privileges needed for your application.

1. Connect to the server with the MySQL command line interface from a terminal window on your workstation: **mysql --port=6033 --protocol=tcp --user=root -p** (the password is 'secret')

2. This next step adds a missing privilege to the MySQL remote root user; this is specific to our VM and isn't normally needed. At the **mysql** command prompt, type:

```
update mysql.user set Grant_priv='Y' where User='root';
flush privileges;
\q
```

3. Log back in again as root (see step 1 above), which will activate the new privilege.

4. Next we'll create a new database and a new user for use in this lab:

```
create database lab3;
create user 'seng365'@'%' identified by 'secret';
grant all privileges on lab3.* to 'seng365'@'%';
flush privileges;
\q
```

5. Connect again to the server through **mysql**, this time as your new user: **mysql --port=6033 --protocol=tcp --user=seng365 -p**

6. Now switch to the new database you just created: **use lab3**

2.1.3. Exercise 1.4: Creating tables

1. Now that we have our server setup, connect to the database in the terminal (as shown in step three of exercise 1.3) 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:	Users		
Name	**Data type**	**Keys**	**Other**
user_id	int	PRIMARY	AUTO_INCREMENT
username	varchar(10)		NOT NULL

Table Name:	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:	Users_conversation		
Name	**Data type**	**Keys**	**Other**
user_id	int	PRIMARY, FOREIGN (users user_id)	
convo_id	int	PRIMARY, FOREIGN (conversations convo_id)	

Table Name:	Messages		
Name	**Data type**	**Keys**	**Other**
message_id	int	PRIMARY	AUTO_INCREMENT

convo_id	int	FOREIGN (conversations convo_id)	
user_id	int	FOREIGN (users user_id)	
sent_time	timestamp		NOT NULL, DEFAULT NOW()

2. Insert dummy data into your tables and run queries to test that your referential integrity is working. Use the MySQL docs and W3schools (https://www.w3schools.com/SQL/deFault.asp) 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 and hosted in your local VM, 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: 'localhost',
  port: 6033,
  user: 'seng365',
  password: 'secret',
  database: 'lab3'
});
```

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 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 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: 'localhost',
      port: 6033,
      user: 'seng365',
      password: 'secret',
      database: 'lab3'
    });
    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 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 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: 'localhost',
```

```
        user: 'seng365',
        password: 'secret',
        port: '6033',
        database: 'lab3'
});
```

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 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 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 - **Challenge**

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.