School of Computing

ST0503 Back End Web Development

Chapter 5

Restful Web Services with MySQL (Model)

Objectives:

After completing this lab, you should be able to:

- Create Web Service API
- Create model layer for backend
- Integrate Web service with MySQL database data

Table of Contents

1. Initialising Node.js project

In this section, we will be initializing a Node.js project (called friendbook) that will be used for the remaining practicals.

Open the Command Prompt and create a folder named "friendbook" on your Desktop:

```
>>> cd Desktop/
>>> mkdir friendbook
>>> cd friendbook
```

Your command line should look like this:

```
C:\Users\dexter\Desktop\friendbook>
```

Setup npm with `npm init`:

```
>>> npm init
```

Hit `Enter` multiple times to use the default settings. Once that is done you should see a `package.json` file in the project folder.

Create the `server.js` file in the root directory with the following inside:

```
console.log("Hello world!");
```

Run the `server.js` file and you should see `Hello world!` printed on the terminal:

```
>>> node server.js
Hello world!
```

2. Installing the `mysql` npm package

We will now connect to the MySQL server from our Node.js program so that we can consume data from the database. In Practical 6, we will serve the data to users of our API.

Install the `mysql` npm package with `npm install`:

```
>>> npm install --save mysql
```

Create a folder named `db` in the project root with an `databaseConfig.js` file inside.

Project Structure

```
models
  databaseConfig.js
server.js
package.json
package-lock.json
```

Connecting to the MySQL Server

Import the `mysql` library in databaseConfig.js:

```
const mysql = require("mysql");
```

According to the `mysql` documentation, `mysql.createConnection(options)` and `connection.connect()` are used to connect to the database.

https://github.com/mysqljs/mysql/#establishing-connections

```
var dbconnect = {
  getConnection: function () {

    var conn = mysql.createConnection({
      host: 'localhost',
      port: 3306,
      user: 'root',
      password: 'password', //your own password
      database: 'friendbook',
      dateStrings: true
    });
    return conn;
  }
};

// put this at the end of the file
module.exports = dbconnect;
```

3. Writing “models” for interacting with the database.

A database model contains methods that interact with the database.

We will now create models that will be exclusively used to interact with the database when the REST server is implemented in Practical 4.

The User model will contain the following methods:

```
User.findByID(id, callback)
User.findAll(callback)
User.insert(user, callback)
User.edit(userID, user, callback)
User.delete(userID, callback)
User.addFriend(userIDOne, userIDTwo, callback)
User.removeFriend(userIDOne, userIDTwo, callback)
User.showFriends(userID, callback)
```

Create a folder in the project root named `models`.

Create a file named `User.js` inside the `models` folder.

Project Structure

```
models
  databaseConfig.js
  User.js
server.js
package.json
package-lock.json
```

3.1. Creating the User model

Add the following to the `models/User.js` file:

```
// we can rename connection as db or anything we choose.
const db = require("./databaseConfig");

const User = {

};
```

```
module.exports = User;
```

We export the `User` object so that we can use it when we create our REST API in Practical 4.

3.2. Implementing the `User.findByID()` method

We can store functions in the property of an object literal as functions are objects in JavaScript.

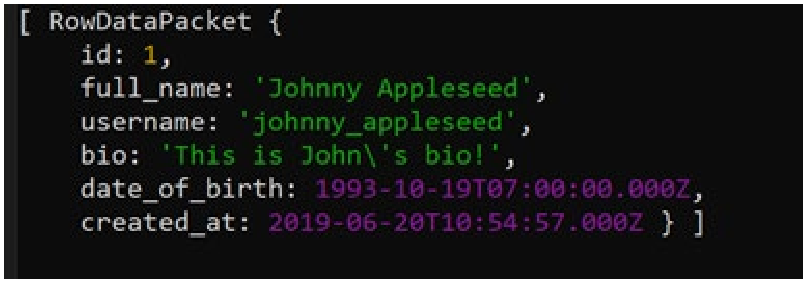
We'll now implement the `User.findByID(id, callback)` method:

```
const User = {


  findByID: function(userID, callback) {

    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
      // We can use "?" as placeholder for user provided data.
      // The userID is passed in through the second parameter of the query
      // method.
      // This is done instead of using string templates to prevent SQL
      // injections.
      // https://github.com/mysqljs/mysql#escaping-query-values

        const findUserByIDQuery = "SELECT * FROM user WHERE id = ?;";
        dbConn.query(findUserByIDQuery, [userID], (error, results) => {
          dbConn.end();
          if (error) {
            return callback(error, null);

          };
          console.log(results);
          return callback(null, results);
        });
      }
    });
  }

}
```

Import the User model from the main `server.js` and test out the `User.findByID()` method:

```
const User = require("./models/User");

User.findByID(1, (error, user) => {
  if (error) {
    console.log(error);
    return;
  };
  console.log(user);
});
```

Kill the server with `CTRL-C`. Start the server again. You should see a list of one user printed:



```
[ RowDataPacket {
    id: 1,
    full_name: 'Johnny Appleseed',
    username: 'johnny_appleseed',
    bio: 'This is John\'s bio!',
    date_of_birth: 1993-10-19T07:00:00.000Z,
    created_at: 2019-06-20T10:54:57.000Z } ]
```

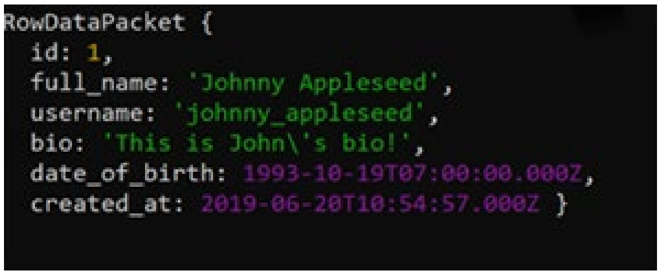
We only want the first element of the array, not the array itself. Modify the `User.findByID()` method:

`Return callback(null, results[0]);`

instead of

return callback(null,results);

Restart the server again. You should only see the object without the array:



```
RowDataPacket {
  id: 1,
  full_name: 'Johnny Appleseed',
  username: 'johnny_appleseed',
  bio: 'This is John\'s bio!',
  date_of_birth: 1993-10-19T07:00:00.000Z,
  created_at: 2019-06-20T10:54:57.000Z }
```

What happens if there are no users with the `id` passed in?

Change the `userID` passed in to a non-existing `id`:

`User.findByID(1000);`

You should see `undefined` printed out.

This is because `results` is an empty array. Accessing a non-existing server will return `undefined`.

Let's propagate the user information and potential errors by calling the callback function passed in. We want `null` to be propagated when there are no users found instead of `undefined`.

Modify `User.findByID() accordingly similar to below:`

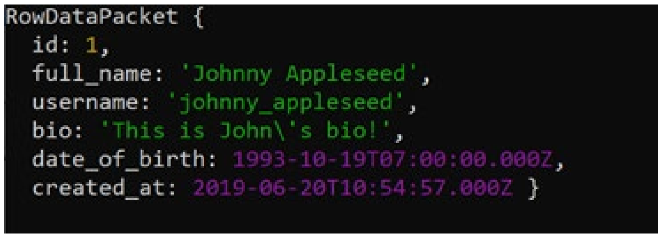
```
// callback signatures are usually (error, value)
findByID: function(userID, callback) {


   …
    if (results.length === 0) {
      callback(null, null);
      return;
    };
   …
}
```

Update the method call from the root `server.js` to pass in a callback:

```
User.findByID(100, (error, user) => {
  if (error) {
    console.log(error);
    return;
  };
  console.log(user);
});
```

You should see `null` printed.

Pass in a valid user id of `1`. You should see the user object printed:



```
RowDataPacket {
  id: 1,
  full_name: 'Johnny Appleseed',
  username: 'johnny_appleseed',
  bio: 'This is John\'s bio!',
  date_of_birth: 1993-10-19T07:00:00.000Z,
  created_at: 2019-06-20T10:54:57.000Z }
```

3.3. Implementing the `User.findAll()` method

Let's implement the `User.findAll(callback)` method. It should query for all users in the database and pass it to the callback function.

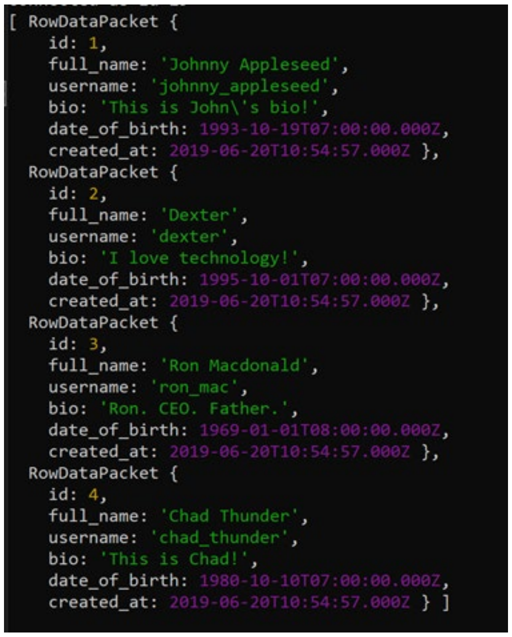
Add the `findAll()` method to the User object:

```
findAll: function(callback) {
  ...

  //add in connection related code…


  const findAllUsersQuery = "SELECT * FROM user;";
  dbConn.query(findAllUsersQuery, (error, results) => {
    if (error) {
       return callback(error, null);

     };


     return callback(null, results);
  });

  …
}
```

Call the `User.findAll()` method in the main `server.js`:

```
const User = require("./models/User");

User.findAll((error, users) => {
  if (error) {
    console.log(error);
    return;
  }
  console.log(users);
});
```

Kill and start the server. You should see the following output:



```
[ RowDataPacket {
    id: 1,
    full_name: 'Johnny Appleseed',
    username: 'johnny_appleseed',
    bio: 'This is John\'s bio!',
    date_of_birth: 1993-10-19T07:00:00.000Z,
    created_at: 2019-06-20T10:54:57.000Z },
  RowDataPacket {
    id: 2,
    full_name: 'Dexter',
    username: 'dexter',
    bio: 'I love technology!',
    date_of_birth: 1995-10-01T07:00:00.000Z,
    created_at: 2019-06-20T10:54:57.000Z },
  RowDataPacket {
    id: 3,
    full_name: 'Ron Macdonald',
    username: 'ron_mac',
    bio: 'Ron. CEO. Father.',
    date_of_birth: 1969-01-01T08:00:00.000Z,
    created_at: 2019-06-20T10:54:57.000Z },
  RowDataPacket {
    id: 4,
    full_name: 'Chad Thunder',
    username: 'chad_thunder',
    bio: 'This is Chad!',
    date_of_birth: 1980-10-10T07:00:00.000Z,
    created_at: 2019-06-20T10:54:57.000Z } ]
```

3.4. Implementing the `User.insert()` method

We'll now implement the `User.insert(user, callback)` method. The callback function will have a signature of `(error, userID)`.

The `user` object should not provide an `id` because the id is autogenerated by the database.

Add the `insert()` method to the User object:

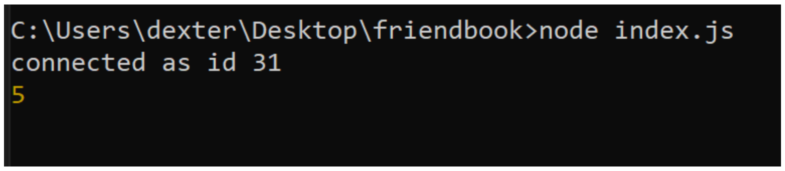
```
    insert: function(user, callback) {


          ...

        //add in connection related code…


        const insertUserQuery =
        `
        INSERT INTO user (username, full_name, bio, date_of_birth)
        VALUES (?, ?, ?, ?);
        `;
        dbConn.query(
          insertUserQuery,
          [user.username, user.full_name, user.bio, user.date_of_birth],
          (error, results) => {

          dbConn.end();
          if (error) {
            return callback(error, null);

          };
          return callback(null, results.insertId);
        });



       …
    },
```

Test out the method in the main `server.js`:

```
const User = require("./models/User");

// We don't need to provide the id because it is generated by the database.
const user = {
    username: "julius",
    full_name: "Julius",
    bio: "Software engineer at Boogle",
    date_of_birth: "2001-10-16"
};
```

```
User.insert(user, (error, userID) => {
  if (error) {
    console.log(error);
    return;
  };
  console.log(userID);
});
```

Run the server again and you should see the user id printed:



```
C:\Users\dexter\Desktop\friendbook>node index.js
connected as id 31
5
```

Note that the id maybe different for you as it depends on the no. of previous users created.

3.5. Implementing the `User.edit()` method

Now let's implement the `User.edit(userID, user, callback)` method. The callback would just be `(error)`. The lack of error would indicate that the user's information has successfully been edited.

Add the `edit` method to the User model object:

```
    edit: function(userID, user, callback) {

        ...

        //add in connection related code…


        const editUserQuery =
        `
        UPDATE user
        SET
          full_name = ?,
          username = ?,
          bio = ?,
          date_of_birth = ?
        WHERE id = ?
        `;
        dbConn.query(editUserQuery, [user.full_name, user.username, user.bio,
user.date_of_birth, userID], (error, results) => {
          dbConn.end();

          if (error) {
            return callback(error);

          };
```

```
            return callback(null);
        });



        …
    };
```

Test out the method in the main `server.js`:

```
const User = require("./models/User");

const editedUser = {
    full_name: "Julius Lim",
    username: "julius",
    bio: "Software engineer at Boogle",
    date_of_birth: "2001-10-16"
};

// provide the user id of the user with full_name "Julius"
// it may not be 5 for you
User.edit(5, editedUser, (error) => {
    if (error) {
        console.log(error);
        return;
    };
});
```

Kill and start the server. Inspect the `user` table in MySQLWorkbench. You should see a user with `full_name` of “Julius Lim”.

3.6. Exercise: Implementing the rest of the User model

Implement the delete and addFriend methods with the following signatures:

```
User.addFriend(userIDOne, userIDTwo, (error))
User.removeFriend(userIDOne, userIDTwo, (error))
User.showFriends(userID, (error, friends))
```

Refer to Practical 2 for the SQL queries.

3.7. Implementing the Post model.

Create a file named `Post.js` under the `models` folder.

Paste this in:

```
const db = require("./databaseConfig.js");

const Post = {

  // propagates a list of posts posted by a user
  // each post contains a "likers" property, which contains an array of users
that liked the post.
  findByUserID: function (userID, callback) {


    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
        const findByUserIDQuery =
          `
    SELECT * FROM post
    WHERE fk_poster_id = ?
    ORDER BY created_at DESC;
    `;
        dbConn.query(findByUserIDQuery, userID, (error, posts) => {
          dbConn.end();
          if (error) {
            return callback(error, null);

          }

          const postIDs = posts.map(post => post.id);
          //creates a new array with the post id

          Post.findLikersByPostIDs(postIDs, (error, likersByPostID) => {
            //likersByPostID is a map containing the postids-> likers’ userid

            if (error) {
              return callback(error, null);

            }

            //transfer the likers’ userid to the respective posts with a new
            //attribute in posts called likers

            for (let i = 0; i < posts.length; i++) {
              posts[i].likers = likersByPostID[posts[i].id];
            }

            return callback(null, posts);
          });
        });
      }
    });
  },
  findByID: function (postID, callback) {

    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {

        const findByIDQuery =
          `
    SELECT * FROM post
    WHERE id = ?;
    `;
        dbConn.query(findByIDQuery, postID, (error, results) => {
          dbConn.end();
          if (error) {
            return callback(error, null);
          }
          if (results.length === 0) {
            return callback(null, null);
          }
          return callback(null, results[0]);
        });
      }
    });
  },

  findAll: function (callback) {
    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
        const findAllQuery =
          `
    SELECT * FROM post;
    `;
        dbConn.query(findAllQuery, (error, results) => {
          dbConn.end();
          if (error) {
            return callback(error, null);
          }
          return callback(null, results);
        });
      }
    });
  },

  insert: function (post, callback) {
    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
        const insertQuery =
          `
    INSERT INTO post (text_body, fk_poster_id)
    VALUES
    (?, ?);
    `;
        dbConn.query(insertQuery, [post.text_body, post.fk_poster_id],
(error, results) => {
          dbConn.end()
          if (error) {
            return callback(error, null);
          }
          return callback(null, results.insertId);
        });
      }
    });
  },

  edit: function (postID, post, callback) {
    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
        const editPostQuery =
          `
    UPDATE post
    SET
    text_body = ?
    WHERE id = ?;
    `;
        dbConn.query(editPostQuery, [post.text_body, postID], (error,
results) => {
          if (error) {
```

```
          return callback(error);
        };
        return callback(null);
      });
    }
  });
},

delete: function (postID, callback) {
  var dbConn = db.getConnection();
  dbConn.connect(function (err) {

    if (err) {//database connection gt issue!

      console.log(err);
      return callback(err, null);
    } else {

      const deletePostQuery =
        `
  DELETE FROM post
  WHERE id = ?
  `;
      dbConn.query(deletePostQuery, postID, (error, results) => {
        dbConn.end();
        if (error) {
          return callback(error);
        };
        return callback(null);
      });
    }
  });
},

like: function (postID, likerID, callback) {

  var dbConn = db.getConnection();
  dbConn.connect(function (err) {

    if (err) {//database connection gt issue!

      console.log(err);
      return callback(err, null);
    } else {
      const likeQuery =
        `
  INSERT INTO likes
  (fk_user_id, fk_post_id)
  VALUES
  (?, ?);
```

```
    `;
        dbConn.query(likeQuery, [likerID, postID], (error, results) => {
          dbConn.end();
          if (error) {
            return callback(error);
          }
          return callback(null);
        });
      }
    });
  },

  unlike: function (postID, likerID, callback) {

    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
        const likeQuery =
          `
    DELETE FROM likes
    WHERE
    fk_user_id = ?
    AND
    fk_post_id = ?;
    `;
        dbConn.query(likeQuery, [likerID, postID], (error, results) => {
          if (error) {
            return callback(error);
          }
          return callback(null);
        });
      }
    });
  },

  findLikers: function (postID, callback) {
    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
        const findLikersQuery =
          `
    SELECT user.* FROM user, likes
    where likes.fk_user_id = user.id
    and likes.fk_post_id = ?
    `;
        dbConn.query(findLikersQuery, postID, (error, results) => {
          dbConn.end();
          if (error) {
            callback(error, null);
            return;
          }
          return callback(null, results);
        });
      }
    });
  },

  // returns a object that maps post id to an array of likers of that post
  findLikersByPostIDs: function (postIDs, callback) {


    // we have to manually handle this edge case because
    // mysql doesn't allow empty lists.
    if (postIDs.length === 0) {
      process.nextTick(() => {
        return callback(null, {});
      });
    }

    var dbConn = db.getConnection();
    dbConn.connect(function (err) {

      if (err) {//database connection gt issue!

        console.log(err);
        return callback(err, null);
      } else {
        const findLikersQuery =
          `
    SELECT user.*, likes.fk_post_id FROM user
    Where likes.fk_user_id = user.id
    and likes.fk_post_id IN (?);
    `;

        dbConn.query(findLikersQuery, [postIDs], (error, likers) => {
          dbConn.end();
          if (error) {
            return callback(error, null);
          }
          const likersByPostID = {};

          // initialize all post ids keys with an empty array
          for (let i = 0; i < postIDs.length; i++) {
            const postID = postIDs[i];
            likersByPostID[postID] = [];
          }

          for (let i = 0; i < likers.length; i++) {
            const liker = likers[i];
            likersByPostID[liker.fk_post_id].push(liker);
          }

          return callback(null, likersByPostID);
        });
      }

    });
  }
}

module.exports = Post;
```

Conclusion

Now that we are able to consume data from MySQL through Node.js code, we will create a REST API that allows a client to interact with our database through the internet.