Lab Guide for Node/Express

Mysql

The main express support for this guide is the express manual: https://expressjs.com/en/5x/api.html

For mysql we are going to use the mysql module for which we can find the manual here: https://www.npmjs.com/package/mysql

We will also follow some tutorials that will be presented below.

You should use the manual manly as a reference guide, but you can read the topics bellow before you do the guide to get a better grasp of the fundamental functionalities of Express.js and the mysql library/module.

This guide will add the connection to the database to our server. There aren't many sites with complete and clear examples for the concepts we are going to use, so you will have to follow manly what was learned in classes and the support slides. However, we will present you below a list of sites that, together with the mysql node module manual already mentioned, contain the concepts used to create the slides and examples learned in the previous classes and mentioned in the slides. NOTE: Many of the tutorials presented below use the concept of "promise" since it is used in the concept of async/await. We avoided giving much detail of the promise concept since we can understand the async/await without it for most part.

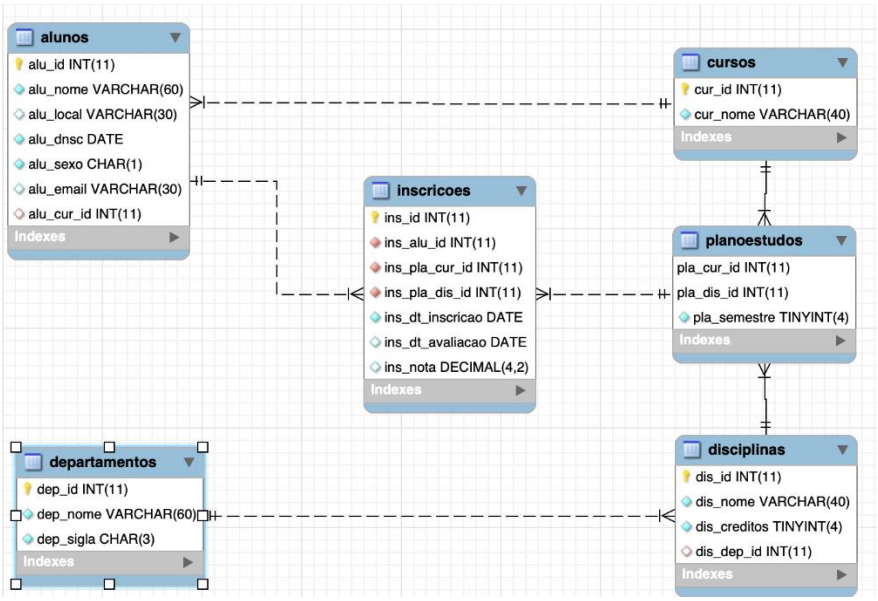
- Tutorial that explains step by step how to prepare the connection to a mysql database using the node module and prepare it to async/await calls: https://medium.com/@mhagemann/create-a-mysql-database-middleware-with-node-js-8-and-async-await-6984a09d49f4
    - If you only want the final code example (avoiding all the explanation) you can find it here: https://gist.github.com/hagemann/30cfee724d047007a031eb12b3a95a23
- Tutorial that explains the async/await concepts: https://zellwk.com/blog/async-await-express/
    - More examples and explanations of async/await: https://javascript.info/async-await
    - The reason to use async/await (callbacks > promises > async/await): https://www.toptal.com/javascript/asynchronous-javascript-async-await-tutorial

Open the project that you used for the previous lab guides ("alunos" project unless you choose another name). Since we are doing the server part, you can do this guide if you finished the first Node/Express guide.

1. Using your mysql client create a new database called "alunos" and run the create.sql and populate.sql supplied along with this guide.

The drop.sql should only be used if you want to recreate/reset the database with the initial information and structure.

The database MER is depicted at the right



2. Install the mysql module if you have done it already. In the terminal inside you project directory run:

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

This will install the module and dependencies (other modules it requires) into the node_modules directory and include the module in the requirements of the package.json.
Be sure to use the --save option or the module will not be saved in the package.json which will mean that it will be missing for any application or user that run "npm install" to recreate your node_modules .

NOTE: Heroku will call "npm install" to set the modules your application uses, so if the module is not on the package.json it will not be installed and errors will occur when your application tries to connect with the database).

3. Create a "connection.js" file in the model directory. Copy the code bellow (same code example in the site https://gist.github.com/hagemann/30cfee724d047007a031eb12b3a95a23) to this file, changing the username and password (and host if you will be using an external host).
NOTE: If your mysql database is using a different port (not the default) you will also need to include the port in the pool definition: `port: 4000,`

```
const util = require('util')
const mysql = require('mysql')


const pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'alunos'
})

// Ping database to check for common exception errors.
pool.getConnection((err, connection) => {
  if (err) {
    if (err.code === 'PROTOCOL_CONNECTION_LOST') {
      console.error('Database connection was closed.')
    }
    if (err.code === 'ER_CON_COUNT_ERROR') {
      console.error('Database has too many connections.')
    }
    if (err.code === 'ECONNREFUSED') {
      console.error('Database connection was refused.')
    }
  }

  if (connection) connection.release()

  return
})

// Promisify for Node.js async/await.
pool.query = util.promisify(pool.query)

module.exports = pool
```

Importing the mysql module

Importing the util module used to prepare the pool for async/await calls

Creating the pool with 10 connections

Each query uses one connection or waits, returning the connection when finished

This code tests if there are errors in the connection

This allows us to know the errors right when the server start instead of only when a database call is made

Preparing the pool for async/await calls

Exporting the pool

Now we can change our models to use the database instead of the lists in the variables (and delete those variables). We should only need to change our model files, since their the only files that manage data.

Our routes already support the asynchronous function calls needed for accessing the database with the async/await keywords.

4. The database supports several courses, but we will only consider one course 'Engenharia Informática' that has id 2. Create the constant bellow in both model files (unitsModel.js and studentsModel.js):

```
const courseId = 2;
```

We should create a model module that would export these values for all others but for now we will use this simple solution.

5. We will start by obtaining the list of units from the database:

- Import the pool by using 'require("connection")'.
- Transform the function getAllUnits into am async function, we will need this so we can call "await" on the request to the database
- Make a pool query and save the result into a variable
  - Use the "await" command so that the function waits for the result from the database
  - The sql query is as below:

```
SELECT dis_id AS id, dis_nome AS name, dis_creditos AS ects, pla_semestre AS semester
FROM disciplinas, planoestudos
WHERE dis_id = pla_dis_id AND pla_cur_id = <courseId>
```

The query above will retrieve the information by combining two tables: "disciplinas" and "planoestudos". We needed to change the names of the fields so that they match the previous structure. For instance, "dis_nome" was renamed to "name".

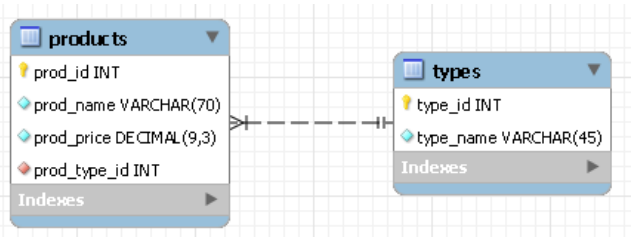
<courseId> corresponds to the constant we defined before and will restrict the result to only one course.

We also retrieved the id, since according to REST we need to be able to include references to resources.

- Do not forget to surround the code with a try/catch so that you can catch and process any error, sending a result to the user even if an error occurs.
- Finally, return the result in the variable.

(more information about making a sql query in node in the next page)

Remember Making queries to the database



- Let's consider an example where we have a table with products and one with product types
- We can make a function to get all products:

```
var pool = require("./connection");
module.exports.getAllProducts = async function() {
    try {
        const sql = "SELECT * FROM products";
        const products = await pool.query(sql);
        console.log(sql);
        return products;
    } catch (err) {
        console.log(err);
        return err;
    }
}
```

The function is asynchronous so we can use "await" for the result of the query made to the database

If any error occurs (in the try) we will catch the error and return it to the client (this should be changed later by a generic message to the user so that he does not have access to debug information)

We return the list of products or the error (for debug) and prints them to the server console

- Another example is obtaining only the products that have only a certain type. Let's consider we have the following constant with the type we are searching:

```
const specialTypeId = 1;
```

- The function that gets only the products with the above type is:

```
module.exports.getSpecialTypeProducts = async function() {
    try {
        const sql = "SELECT * FROM products WHERE prod_type_id = "+specialTypeId;
        const products = await pool.query(sql);
        return products;
    } catch (err) {
        console.log(err);
        return err;
    }
}
```

We only needed to change the name and sql queries

Many times only the names of the function and variables and the sql query string needs to be changed

- For both examples the result is an array of object products since the mysql module will convert the result sent by mysql. Each line of the the result query will be an object, with one property/value pair for each column. Se set of lines will be an array of objects. Example of a result for the above functions:

  [{ prod_id: 1, prod_name: "Potatoes", prod_price:0.34, prod_type:1 },
  { prod_id:3, prod_name:"eggs", prod_price:0.7, prod_type:1 }]

- This result will be received by the router and transformed into JSON to be sent to the client

6. We now need to do the same for the get functions in file "studentsModel.js" :

- For the "getAllStudents" we only need the name, number and id (units and grades are only needed when we obtain a specific student). The database does not have a student number so we will use zero for all students. SQL:

```
SELECT alu_id AS id, alu_nome AS name, 0 AS number FROM alunos
WHERE alu_cur_id = <courseId>
```

  Like for units we are restricting the students to the course identified by the value in the courseId constant (remember we asked that you created the constant "const courseId = 1;" in this file too).

- For the "getStudent" we need to get the information about the student and the information about the units of that student. That means making two queries and create one object with both results:
  - First, we make the same query as before but filtering by the id.

```
SELECT alu_id AS id, alu_nome AS name, 0 AS number FROM alunos WHERE alu_id = ?
```

    You should escape the id value using the "?" as the placeholder (this avoid attacks of SQL injection).

  - Second, we obtain all the units and grades for the student and save it at the property "grades" of the student object

```
SELECT dis_id AS id, dis_nome AS name, dis_creditos AS ects,
       pla_semestre AS semester, ins_nota AS grade, ins_id
FROM disciplinas, planoestudos, inscricoes
WHERE dis_id = pla_dis_id AND ins_pla_dis_id = pla_dis_id AND
      ins_alu_id = ? AND pla_cur_id = <courseId>
```

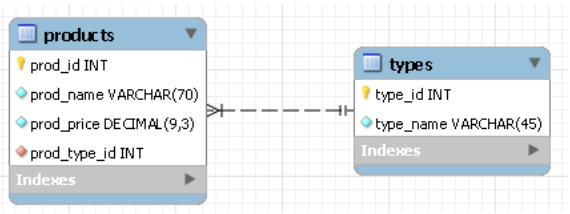
    We will need to combine the 3 tables that have the information we need: unit id, unit name, grade, semester and ects. Again, we will need to escape the id using the "?" as placeholder. We can also escape the courseId constant, but we do not need to since it was defined by us inside the server and we know it will not be incorrect.

    We also included the id of the inscription we will see why when we change the "saveGrade" function.

  - Do not forget to return the value or an error object, using the try/catch to treat errors

(more information about escaping and combining queries in the next page)

Remember Escaping combining information (queries)



- Let's consider the same example as before
- We now need to get all the products of one type and we will receive the id of the type
- We need to make two queries, one to get the type name and another to get the product list for that type

```
var pool = require("./connection");
module.exports.getProductsByType = async function(typeId) {
    try {
        let sql = "SELECT * FROM types WHERE type_id = ?";
        let typeProds = await pool.query(sql,[ typeId ]);
        let typeProd = typeProds[0];
        sql = "SELECT * FROM products WHERE prod_type_id = ?";
        typeProd.products = await pool.query(sql,[ typeId ]);
        return typeProds;
    } catch (err) {
        console.log(err);
        return err;
    }
}
```

We receive the type id

Obtain type name and id

Values in the array will be escaped and placed in the placeholders "?" in the order the placeholders appear

Sql queries always return an array, even if it is only one element, so we need to obtain the first position of the array

Obtain products for that type and save it in products property

Escaping values received by the user is important to avoid SQL injection

- NOTE: You can break the sql string in parts but do not forget to end or start in a space so that the space is there to separate words when the string is concatenaded:

```
sql = "SELECT * FROM products WHERE " +   // Notice the space colored in   (separation)
      "prod_type_id = ?";
```

- Example of the result:

```
{
  type_id:1,  type_name: "specials",  products :
   [  { prod_id: 1,  prod_name: "Potatoes",   prod_price:0.34,   prod_type:1 },
     { prod_id:3,  prod_name:"eggs",  prod_price:0.7,  prod_type:1 }  ]
}
```

7. Finally, we change the "saveGrade" function in "studentsModel.js" file. In this case we will need to change the functionality since the database allows for the student to have multiple inscriptions for the same unit.

We will consider that the student is already inscribed at the unit, and that the unit received includes the id of the inscription (we actually included the inscription id on the "getStundent" result just because of this).

- The SQL query will simply be an UPDATE of the date and the grade for that inscription :

  `UPDATE inscricoes SET ins_dt_avaliacao = ? , ins_nota = ? WHERE ins_id = ?`

  You can use "new Date()" inside the escape array to use the current date for the grade date (the mysql module will convert the date to the format used by mysql), ins_nota will be the grade received for that unit, and ins_id will be the inscription id received.

  You can return the result of the update to the user (just for debug, a final version should return a customized message).

Remember Updates and inserts

Updates and inserts are run with the query command just like selects, but the result they return is not a list of rows but information about the changes made:

- You can obtain the number of changed rows (zero means no change was made and you can use it to send error message to the user). The example below changes the date of visit of a client to the current date, if the id received (the client id) does not exist an error message is returned, if not a successful message is returned.

```
let sql = "UPDATE client SET cli_last_visit_date = ? WHERE cli_id = ?";
let result = await pool.query(sql,[ new Date(), id ]);
if ( result.changedRows > 0 ) return { msg: "Client visit is now today" };
else return { msg: "ERROR: That Client does not exist" };
```

- Another very useful information is obtaining the id of the element that was inserted, this is important since many ids are autoincremented so we can only know the id after it was inserted:

```
let sql = "INSERT INTO product(prod_name,prod_price, prod_type_id) values(?,?,?)";
let result = await pool.query(sql,[name, price, type_id ]);
return { msg: "Inserted product with id " + result.insertId , id: result.insertId };
```

The mysql module supports more advanced escaping that will not be covered in this guide, but you can see examples in the escaping manual page: https://www.npmjs.com/package/mysql#escaping-query-values

8. Notice we are not using the student id or the unit id for the “saveGrade”, these ids should be used to verify if the inscription is actually the correct one.

   Add that verification by making a query that checks if the inscription for those ids (student id and unit id) exists and the id corresponds to the received inscription id.

   If it does not exist or does not match send an object with an error message saying that the inscription does not correspond to the student and/or unit.

The End