Backend Development using Node.js

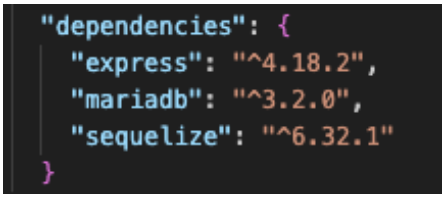
Key Vocabulary

- **Command Line Interface (CLI)**
  - The terminal used to communicate directly with the computer software rather than using buttons and windows to complete a command (also known as the graphical user interface). CLI is usually more efficient than the GUI.
- **Module**
  - A functionality written with JS so you can reuse it in a node.js application. Usually modules have functions you want to reuse. It's not good practice to have all of your functions on the main index.js file
- **Node Package Manager (NPM)**
  - A command line tool used to register third-party libraries you can add to your Node.js application.
- **Promise**
  - It represents the result of an asynchronous operation. A promise can exist in three states:
    - Pending - it's initial state, nothing has been executed yet
    - Fulfilled – the promise was executed successfully. You will see a .then() method in a promise to write out the code to execute when a promise is fulfilled.
    - Rejected – something went wrong and the promise was not executed. You will see a .catch() in promise code to tell JS what to do when the promise fails.
  - EXAMPLE OF PROMISE CODE:

```
object.then((data)=>{
console.log(data);
res.status(200).send(data);
})
.catch((err)={
console.log(err);
res.status(500)send(err)
});
```

- **Exporting a module**
  - Allowing the module to be used in another module or file. You use the code **module.exports = nameOfModule**
- **Importing a module**
  - Referencing another module in order for your functions or code to work. Use the **require()** function with the path to your module as the parameter inside the parentheses. You use the following code to import a module:
    - **const nameOfModule = require("modulepath");**
    - Example: const Student = require("./models/student");

- **Express**
  - A very important module that allows Node.js to run a web application. It can provide middleware functions to be used and rending of the HTML by generating HTTP requests.
- **Dependencies**
  - Important modules you need to run your code. You will see the dependencies written in the package.json file. Like this:



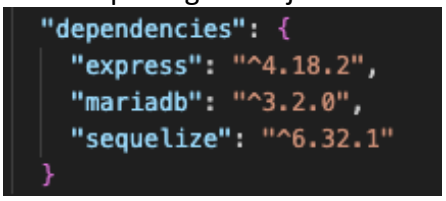
- **JSON (JavaScript Object Notation)**
  - The standardized format for structuring data in the form of JavaScript objects. JSON is written as a string, but it is super important for transferring information across the internet.
- **Object Relational Mapping (ORM)**
  - A technique to query (search) data from a relational table and output the result in the form of an object. Nodejs is JavaScript, which is an object-oriented language. It will need to understand your tables in the form of objects for it to work with it.
  - ORM makes objects turn into values (something you can put in a table). Then uses SQL commands to add the values into the database and vice versa.
- **Sequelize**
  - A type of ORM. Sequelize makes it easier for your to manipulate data in a database. Rather than going into your DBMS and writing SQL commands. Sequelize will do it for you. But, it needs to understand what the data is, in the form of objects (which are stored in models). Think of Sequelize as a “concierge” to your database.
- **Model**
  - A representation of a table from a database in the form of an object. Node.js does not read tables, it needs to understand what it is in the form of an object. This is why your table in your database called “students” (a table that holds information about multiple students), is called “Student” in your model. Sequelize is like an alien, it doesn’t know what a student is. So you need to give it a blueprint of what a student is. Example: you have a table about dogs, but to describe one “Dog”, it can have properties such as 4 legs, two ears, a tail and barks in the model of a dog.
  - Note: The name of one model is also royal CamelCase because it is a class in JS.

Created by Marwa Elkelani

HOW TO START A NODE.js APPLICATION

Part 1: Getting Started

1) **Create a folder:** Create a folder in VS Code, then right click and click “Open in Integrated Terminal”.
2) **Initialize your folder:** In the terminal, type: **npm init -y** (this means node package manager will initialize the folder to be ready for Node.js. The -y means you say “yes” to all questions asked in the terminal).
3) **Install your dependencies:** type in the terminal **npm install express mariadb sequelize**. This command will important in your package.json file the express, mariadb and sequelize dependencies. You will need mariadb and sequelize if you are working with a database. This will also download the package-lock.json and the modules folder



4) **Create a js file called index.js:** made a new file inside your overarching folder and name is **index.js.** We will come back to this later.

Part 2: Setting Up Your Supporting Modules and Models (Database setup)

5) **Set up a database in a Database Management System (DBMS).** Create your database and tables in DBeaver/Heidi. Make sure to include primary keys, etc.
6) **Link the database to your VS Code using the config.js file.** Node.js will not know what your database is and how to access it. Start a js file and call it **config.js**. Import sequelize and make variables for your database, username, and password. Then export the module as config. It should look like below:

```
const sequelize = require('sequelize');  // Importing sequelize to use in database

let database = 'mariadb'      // You are identifying the database sequelize needs to find
let username = 'root'         //You are identifying the username it needs to access the DB
let password = 'password'  //You are identifying the password sequelize needs to access the DB

const config = new Sequelize(database, username, password, {dialect: 'mariadb'});

module.exports = config;
```

7) **Turn your database tables into models**: Turn the tables in your DB into models for sequelize to understand what they are. Once sequelize knows what your tables are about, it can use SQL commands to fetch and manipulate data using one of the HTTP routes. **Don't forget to name the file in singular form.

   Here is an example of a model to represent students in your "students" table.

```
const Sequelize = require('sequelize');
const config = require('./../config');

const Student = config.define('student', {
    id: {
        type: Sequelize.INTEGER,
        autoIncrement: true,
        allowNull: false,
        primaryKey: true
    },
    name: {
        type: Sequelize.STRING,
        allowNull: false
    },
    dept_id: {
        type: Sequelize.INTEGER,
        allowNull: true
    },
    country: {
        type: Sequelize.STRING,
        allowNull: false
```

Part 3: Setting Up the index.js File

//Import the important modules you need

8) **Import the express module**: type in **const express = require('express');**
9) **Import config module:** so that your database is connected to index.js by typing in **const config = require('./config);**
10) **Import your models:** so that you can use the tables from your database in your HTTP routes. Example: **const Student = require('./models/student)**

//Create the webserver

11) **Create the server:** use the app.listen() to create the server. Inside the .listen() method, type in the port you want the server to run. You can also include a callback function with a console.log to make sure the server is working. Here is an example:

```
app.listen(3000, function(){
        console.log('Server running on port 3000…');
});
```

//Test the database connection

12) **Test the database:** You need to make sure the connection between your config.js file (the file that describes what your database is and how to get there) and index.js file works. Create a **promise**, and include the .**authenticate()** method. It will look like this:

```
config.authenticate().then( ()=> {
        console.log('Database is connected');
}).catch((err) => {
        res.send(err);
});
```

Digest this code: It means "authenticate the config file, if the connection is successful, write in the console log 'database is connected'. If the promise fails, show an error message (this is a built-in message).

//Add middleware **Note, your POST routes will not work without a middleware!

A middleware is the middleman between the client and the server (think like a bouncer). It is written similar to an HTTP method, but can include the next() function, which means the middleware will send you to the next middleware (if there is any). **Local middlewares** are local to specific routes and should be written in the route. **Global middlewares** are to be used on all routes.

13) **Add middleware for post routes**: Here are some to get you started.

For requests that go through the body (postman) with urlecoded:

```
app.use(express.urlencoded({extended: false}));
```

For requests that go through the body but in JSON:

```
app.use(express.json());
```

//Add HTTP Routes *used with databases

HTTP stands for Hypertext Transfer Protocol. It is a protocol for best practices for sending requests. Think of it as identifying a purpose for your request so the computer knows why you want to access the server.

Example: If your friend tells you they want to come to your house to drop off some food, but then they come to sell you a vacuum cleaner, not only will you be annoyed, but caught off guard. They can still come, but it's bad manners. HTTP methods are like that, they give the server a heads-up why you need a request.

GET Route

- Retrieve information to be shown on the browser.
- You will use either "req.query" or "req.params" when working on postman.
  - Query allows for the client to search information in the URL
  - Params means a parameter must be included in the url for the search to work
- Methods (functions connected to objects) you might use and connect to a promise:
  - **.findAll()** *Means find all students in the model (table in db)*
  - Example of it's use:

```
app.get('/students', (req,res)=>{
        Student.findAll().then((results)=>{
                res.status(200).send(results);
        }).catch((err)=>{
                res.status(500).send(err)
        });
});
```

  - **.findByPk()** *Means find by the primary key in the table in the model (table in db)*
  - Example of it's use:

```
Student.findByPk().then((results)=>{
        res.status(200).send(results);
}).catch((err)=>{
        res.status(500).send(err)
});
```

POST Route

- Used to create new information (example, new entry in a table, new username, etc.)
- Data will be requested through the body, meaning you will use “req.body” when creating code for a post route.
    - Why? The body can be encrypted and does not allow for the client to have information in the url, making it more secure.
    - ***Middleware is needed for this to work.
- Methods (functions connected to objects) you might use and connect to a promise:
    - **.create()** *Means create a new entry in the db table using the model*
    - Example of it’s use:

```
app.post('/students', (req,res)=>{
        let newStudent = req.body;
        Student.create(newStudent).then((results)=>{
                res.status(200).send(results);
        }).catch((err)=>{
                res.status(500).send(err)
        });
});
```

PATCH Route

- Used to update some information about an entity. Example: update address for a user.
- Data will be requested through the body. However, include that the data can be filtered through params (meaning, find the student by including a parameter in the url, but then use the body to update and save the student)
- Methods (functions connected to objects) you might use and connect to a promise:
    - **.findByPk()** *Means find by the primary key in the table in the model (table in db)*
    - **.save()** *Means save the changes you've made to an entry*
    - Nest if statements in the route to find a specific piece of data.

```
app.patch('/students/:student_id', (req, res){
let studentId = parseInt(req.params.student_id);

Student.findByPk(studentId).then((results)=>{
        If(result) {
                result.country = req.body.country;       //update student
                result.save().then()=>{                  //save the update
                        res.status(200).send(result);
                })
                .catch((err)=>{
                        res.status(500).send(err)
                })
        }else{
                res.status(404).send('Student not found');
        }
    })
    .catch((err)=>{
        res.status(500).send(err)
    });
});
```

DELETE Route

- Delete a record completely.
- This request is through the params. A parameter will need to be inputted into the url to find the record and complete the request.
- Methods (functions connected to objects) you might use and connect to a promise:
  - **.findByPk()** *Means find by the primary key in the table in the model (table in db)*
  - **.destroy()** *Means to complete drop the record from the database*
  - Nest if statements in the route to find a specific piece of data.

```
app.delete('/students/:student_id', (req, res){
let studentId = parseInt(req.params.student_id);

Student.findByPk(studentId).then((results)=>{
        If(result) {
                result.destroy().then()=>{          //delete the record
                        res.status(200).send('Student has been deleted');
                })
                .catch((err)=>{
                        res.status(500).send(err)
                })
        }else{
                res.status(404).send('Student not found');
        }
    })
    .catch((err)=>{
        res.status(500).send(err)
    });
});
```

//Add Table Associations (if needed)

Association of Tables in a Database

- This means telling sequelize what the associations are between your models (tables in your database) are. For example, how are Student and Department models related?

- According to Sequelize documentation, there are four(4) associations. Attach the following methods to your models in the index.js file to show associations:
    - ModelName.**hasOne(OtherModel, {describe keys and parameters})**
    - ModuleName.**belongsTo(OtherModel, {describe keys and parameters})**
    - ModelName.**hasMany(OtherModel, {describe keys and parameters})**
    - ModelName.**belongsToMany(OtherModel, {describe keys and parameters})**

Example:

```
Student.belongsToMany(Course, {
   through: StudentCourse,
   foreignKey: 'student_id',
   otherKey: 'course_id'
});
```