full stack app boilerplate (NEM)

Summary of building a simple full stack application with node, express, and mysql.

Recall

- What is the process of building a full stack application with NEM?

---

        database handler
    DELETE
        front-end
        back-end server
        database handler
    EDIT
        front-end
        back-end server
        database handler

A server-side app that handles SQL queries within API calls. The front end program will use the API calls to collect the data and dynamically update a table.

Front End Setup

For the basic front end, start with an HTML/CSS + Javascript boilerplate. You may want to separate these files into a client-side subdirectory of the project.

- index.html
- index.js
- stylesheet.css

HTML/CSS File

Create the webpage display and style it. Keep in mind any tags that you will use Javascript to dynamically populate, and leave those empty. It is important to give unique IDs to elements you want to grab later.

Javascript File

With Javascript prepare your event handlers for when the page loads. An example of this is calling the server-side API to dynamically populate a table.

```
/*
    This is an event listener that is called when the page loads. It fetches the endpoint
```

```
 for table data.
*/
document.addEventListener('DOMContentLoaded', function () {
    // Setting the endpoint
    fetch('http://localHost:5000/getAll')
        // Converting response to json format
        .then(response => response.json())
        // Getting data back in json format and logging it to the console. Must access the
data key of the json file
        .then(data => loadHTMLTable(data['data']));
});

/*
    Function that takes in data and loads it into the table
    Args: Array of data
*/
function loadHTMLTable(data) {
    // Grab the table body
    const table = document.querySelector('table tbody');

    // If there is no data, display that
    if (data.length === 0) {
        table.innerHTML = "<tr><td class='no-data' colspan='5'>No Data</td></tr>";
    }
}
```

Back End Setup

Modules

This constitues using Express.js. You should have a separate server-side subdirectory of the project for this. Within that directory, initialize npm and install Express.js and MySQL as project dependencies.

- It is a good practice to also install dotenv. This allows the usage of hidden `.env` files, which are a place to store database configurations such as passwords and usernames.
- It is a good practice to do this, so that when you push your code to a repo you are not sending the configurations as well.

Nodemon will watch the script and reload the server whenever a change is made. This prevents you from needing to restart the server. This will install as a devDependency.

📖 Installing cors will allow API calls to be made from front end to back end. Highly important.

`npm init -y`

`npm install express`

`npm install mysql`

`npm install dotenv`

`npm install nodemon –save-dev`

`npm install cors`

Server File Setup

Within the `packages.json` file, it may be helpful to change the main js file name to prevent any confusion with the front end js file. It's good practice to name the file after the express variable name.

Once this file is created, initialize the modules.

📖 If dotenv is installed, create the env file with the configurations and variables.

```
//-----MODULE INITIATION-----//
const express = require('express');
const cors = require('cors');
const dotenv = require('dotenv');

//-----MODULE SETUP-----//
const app = express();
dotenv.config();
// Allows API usage
app.use(cors());
// Allows sending data in json format
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
```

At the bottom of the file, create the process for starting the server.

```
//-----SERVER STARTUP-----//
app.listen(process.env.PORT, () => {
```

```
    console.log("The app is online.");
});
```

📘 In the terminal, use this command to start the server:

`npx nodemon <filename>`

Routes

The routes for the SQL queries come next. These are the endpoints that the front-end will be calling.

Test the front-end to back-end connection by:

- Server-side file
  - Creating a simple console.log() within a GET endpoint
- Client-side file
  - Adding a `fetch` to the event listener for the page loading. Use the same PORT number that the server is running on.
- If ‘TEST’ was logged to the console, the connection is functioning properly

```
// READ route
app.get('/getAll', (request, response) => {
    // Test for reaching back end through API call
    console.log('TEST');
});
```

```
// Setting the endpoint
fetch('http://localHost:5000/getAll')
// Converting response to json format
.then(response => response.json())
// Getting data back in json format and logging it to the console
.then(data => console.log(data));
```

Database Service File

A separate file is used for the connection to the database, and sending queries. This is where the meat of the back-end code will be.

For explanation a mock table for testing will be used within the database.

1. Create a mock table with data fields corresponding to the front-end mock table.
2. Add a database user as a representation of the web application (if necessary) and grant it all priveledges.
    a. Add the user and password credentials, as well as the database name, the host, and server port number to the `.env` file.

The basic modules needed here are mysql and dotenv. Once those are initiated the database connection can be created. In this example there is a callback to check for errors.

```
const mysql = require('mysql');
const dotenv = require('dotenv');

dotenv.config();
```

```
// Create a connection
const connection = mysql.createConnection({
    host: process.env.HOST,
    user: process.env.USERNAME,
    password: process.env.PASSWORD,
    database: process.env.DATABASE,
    port: process.env.DB_PORT
});

connection.connect((err) => {
    if (err) {
        console.log(err.message);
    }
    console.log('db' + connection.state);
});
```

If the database is having trouble connecting, it could possibly a security code configuration issue, blocking the IP.

📖 At the end of the file, export this as a module. This is done so that the server file can create instances and use the methods.

Database Class

Within database service file, there will be a class for the database service. This will house the methods used to execute queries. It's a good practice to limit the number of instances to one.

```
//-----CLASSES-----//
// This class will be used to hold the functions for manipulating the data
class dbService {
    /*
        This function grabs the instance of the class. Without it, multiple instances woul
d be made.
        The return statment checks if instance is not null. If true, it returns the active
instance. If not, it creates a new instance.
    */
    static getInstance() {
        return instance ? instance : new deService();
    }
}

//-----MODULE EXPORT-----//
module.exports = dbService;
```

Query Methods

📘 `connection.query(<query>,<args array>)` - Takes in a query (in string format using SQL syntax) and an array of arguments for the query.

- It helps to use a callback function to handle the results.

In this example, the response variable is initiated to a Promise which runs the code for the query using a (resolve, reject) flow. If the query is successful, it will resolve. If not, it will reject and transfer to the catch.

```
/*
    Grabs all of the data from the database table
```

```
    If there is an error, logs it to the console.
    Otherwise it returns the result.
*/
async getAllData() {
    try {
        // Create a promise to handle the query. Using a resolve, reject. if query success
ful, it will resolve. If not, it will reject and transfer to the catch.
        const response = await new Promise((resolve, reject) => {
            const query = "SELECT * FROM mock_table";
            connection.query(query, (err, results) => {
                if (err) reject(new Error(err.message));
                resolve(results);
            });
        });

        return response;

    } catch (error) {
        console.log(error);
    }

}
```

Parameterizing variables when arguments are needed helps prevent against SQL Injection attacks. This is done by using '`?`' in the place of variables.

```
const query = "SELECT * FROM mock_table WHERE id = ?";
connection.query(query, [id]);
```

Implementing Queries and Methods

All three files will need to be edited when adding new queries, which should be kept in mind. The general workflow is this:

- index.js (front end)
    - Handles the UI events such as button clicks
    - Calls the back-end local API endpoints using the fetch to retrieve data
    - Dynamically updates the webpage with retrieved data
- app.js (back-end server)
    - Implements the server API endpoints

    - Calls the dbService instance methods
    - Returns the result to the front-end caller
- dbService.js (back-end database handler)
    - Contains method for sending query to the database, handling errors, and returning the resulting data to the back-end server.
        - Good practice to do this in a try-catch
        - Good practice to use promises

> 📖 It is currently recommended to begin by editing in this direction:
>
> - database handler → server → webpage

For this, the mock table with be used as an example.

GET

This example retrieves all of the data in the table.

front-end

This calls the /getAll endpoint when the page is loaded.

```
/*
    This is an event listener that is called when the page loads. It fetches the endpoint
 for table data.
*/
document.addEventListener('DOMContentLoaded', function () {
    // Setting the endpoint
    fetch('http://localHost:5000/getAll')
        // Converting response to json format
        .then(response => response.json())
        // Getting data back in json format and logging it to the console. Must access the
data key of the json file
        .then(data => loadHTMLTable(data['data']));
});
```

The helper function that scans for database data and loads into the webpage table.

```
/*
    Function that takes in data and loads it into the table
    Args: Array of data
*/
function loadHTMLTable(data) {
    // Grab the table body
    const table = document.querySelector('table tbody');

    // If there is no data
    if (data.length === 0) {
        table.innerHTML = "<tr><td class='no-data' colspan='5'>No Data</td></tr>";
        return;
    }

    // If there is data
    let tableHtml = "";

    data.forEach(function ({ id, name, date_added }) {
        tableHtml += "<tr>";
        tableHtml += `<td>${id}</td>`;
        tableHtml += `<td>${name}</td>`;
        tableHtml += `<td>${new Date(date_added).toLocaleString()}</td>`;
        // A button to delete the data
        tableHtml += `<td><button class="delete-row-btn" data-id =${id}>Delete</button></t
d>`;
        // A button to edit the data
        tableHtml += `<td><button class="edit-row-btn" data-id =${id}>Edit</button></td>`;
        tableHtml += "</tr>"
    });

    // Updating the HTML
    table.innerHTML = tableHtml;
}
```

back-end server

```
// READ
app.get('/getAll', (request, response) => {
    // Grab instance of db
    const db = dbService.getInstance();

    const result = db.getAllData();

    console.log(result);
    // Return the promise to the fetch in a json for the api
    result
        .then(data => response.json({ data: data }))
        .catch(err => console.log(err));
});
```

database handler

```
async getAllData() {
        try {
            // Create a promise to handle the query. Using a resolve, reject. if query suc
cessful, it will resolve. If not, it will reject and transfer to the catch.
            const response = await new Promise((resolve, reject) => {
                const query = "SELECT * FROM mock_table";
                connection.query(query, (err, results) => {
                    if (err) reject(new Error(err.message));
                    resolve(results);
                });
            });

            return response;

        } catch (error) {
            console.log(error);
        }

    }
```

GET (search by name)

front-end

```
/*
    Sends the GET request, with the name parameter. If successful, returns the rows with t
he appropriate information.
*/
const searchBtn = document.querySelector('#search-btn');
searchBtn.onclick = function () {
    const searchValue = document.querySelector('#search-input').value;

    fetch('http://localhost:5000/search/' + searchValue)
        // Converting response to json format
        .then(response => response.json())
        // Getting data back in json format and logging it to the console. Must access the
data key of the json file
        .then(data => loadHTMLTable(data['data']));
}
```

back-end server

```
app.get('/search/:name', (request, response) => {
    const { name } = request.params;

    const db = dbService.getInstance();

    const result = db.searchByName(name);

    result
        .then(data => response.json({ data: data }))
        .catch(err => console.log(err));
});
```

database handler

```
/*
  The same process as getAll, but limiting the results to match the passed-in name argumen
t
*/
async searchByName(name) {
  try {
      const response = await new Promise((resolve, reject) => {
          const query = "SELECT * FROM mock_table WHERE name = ?;";
          connection.query(query, [name], (err, results) => {
              if (err) reject(new Error(err.message));
              resolve(results);
          });
      });

      console.log(response);
      return response;

  } catch (error) {
      console.log(error);
  }

}
```

ADD

front-end

This handles the submit button click after the user enters data. Once it calls the `/insert` endpoint, the data is handed off to the back-end and entered into the database. The

purpose of the `insertRowIntoTable()` function is to dynamically update the table once the user's request is fulfilled, rather than waiting until they reload the page.

```
/*
    Function that waits for the submit button to be pressed, then takes the argument in th
e input box and sends it to the back-end

    There is no .catch or error handling, since we have that in the back-end.
*/
const addBtn = document.querySelector('#add-btn');
addBtn.onclick = function () {
    const nameInput = document.querySelector('#test-input');
    // Every time we grab value and send to back-end, reset value to empty string
    const name = nameInput.value;
    nameInput.value = "";

    // Accesses the API and transmits the data to the back-end. After a response is reciev
ed, fulfills the .then promises
    fetch('http://localHost:5000/insert', {
        headers: {
            'Content-type': 'application/json'
        },
        method: 'POST',
        body: JSON.stringify({ name: name })
    })
        .then(response => response.json)
        .then(data => insertRowIntoTable(data['data']));
}
```

```
/*
    This function will take data and insert it into the table as a new row. The purpose of
this is the update the table once the add button is clicked, rather than needing to refres
h.
    Takes in an array of data.
*/
function insertRowIntoTable(data) {
    const table = document.querySelector('table tbody');
    const nonEmptyTable = table.querySelector('.no-data');

    let tableHtml = "<tr>";

    // Cannot use a forEach because we are dealing with an abject, not an array.
    for (var key in data) {
        if (data.hasOwnProperty(key)) {
            if (key === 'dateAdded') {
                data[key] = new Date(data[key]).toLocaleString();
            }
            tableHtml += `<td>${data[key]}</td>`;
        }
```

```
    }
    // A button to delete the data
    tableHtml += `<td><button class="delete-row-btn" data-id =${data.id}>Delete</button></
td>`;
    // A button to edit the data
    tableHtml += `<td><button class="edit-row-btn" data-id =${data.id}>Edit</button></td>
`;

    tableHtml += "</tr>";

    // If table is empty, add a new row
    if (nonEmptyTable) {
        table.innerHTML = tableHtml;
    } else {
        const newRow = table.insertRow();
        newRow.innerHTML = tableHtml;
    }
}
```

back-end server

```
// CREATE
app.post('/insert', (request, response) => {
    // Object destructuring
    const { name } = request.body;

    const db = dbService.getInstance();

    const result = db.insertNewName(name);
    result
        .then(data => response.json({ data: data }))
        .catch(err => console.log(err));

});
```

database handler

```
/*
    This function sends a query to enter new name into the database table
*/
async insertNewName(name) {
    try {
        const dateAdded = new Date();
        const insertID = await new Promise((resolve, reject) => {

            // Parameterized the values to protect against SQL injection
            const query = "INSERT INTO mock_table (name, date_added) VALUES (?), (?)";
```

```
            connection.query(query, [name, dateAdded], (err, results) => {
                if (err) reject(new Error(err.message));
                resolve(result.insertID);
            });
        });

        // Return the ID, name, and date_added to the front-end
        console.log(insertID);
        return {
            id: insertID,
            name: name,
            dateAdded: dateAdded
        };

    } catch (error) {
        console.log(error);
    }
}
```

DELETE

front-end

This utilizes an event listener to detect clicks, then determines if the delete button was clicked.

```
/*
    Using an event listener for when the user decides to delete a row
*/
document.querySelector('table tbody').addEventListener('click', function (event) {
    console.log(event.target);
    if (event.target.className === 'delete-row-btn') {
        deleteRowById(event.target.dataset.id);
    }
});
```

```
/*
    Calls server side to delete the row. If successful, reloads the page
*/
function deleteRowById(id) {
    fetch('http://localhost:5000/delete/' + id, {
        method: 'DELETE'
    })
        .then(response => response.json())
        .then(data => {
```

```
            if (data.success) {
                location.reload();
            }
        });

}
```

back-end server

```
// DELETE
app.delete('/delete/:id', (request, response) => {
    console.log(request.params);
    const { id } = request.params;

    const db = dbService.getInstance();

    const result = db.deleteRowById(id);

    result
        .then(data => response.json({ success: data }))
        .catch(err => console.log(err));

});
```

database handler

```
async deleteRowById(id) {
        id = parseInt(id, 10);
        try {
            const response = await new Promise((resolve, reject) => {

                // Parameterized the values to protect against SQL injection
                const query = "DELETE FROM mock_table WHERE id = ?;";

                connection.query(query, [id], (err, result) => {
                    if (err) reject(new Error(err.message));
                    resolve(result.affectedRows);
                });
            });

            console.log(response);
            return response === 1 ? true : false;

        } catch (error) {
            console.log(error);
            return false;
        }
```

```
    }
```

EDIT

front-end

For this a new Html section was created beneath the table. This section is initially hidden, then shows itself once the user clicks the edit button on one of the table rows.

```
<section hidden id="update-row">
  <label>Name: </label>
  <input type="text" id="update-input">
  <button id="update-row-btn">Update</button>
</section>
```

This requires button and click handling. The same click listener is used to handle the user clicking an edit button.

```
/*
    Using an event listener for when the user decides to delete a row
*/
document.querySelector('table tbody').addEventListener('click', function (event) {
    console.log(event.target);
    if (event.target.className === 'delete-row-btn') {
        deleteRowById(event.target.dataset.id);
    }
    if (event.target.className === 'edit-row-btn') {
        handleEditRow(event.target.dataset.id);
    }
});
```

```
/*
    Show the UI and set the data id for the Update button. This links the update button to
the corresponding database table row.
*/
function handleEditRow(id) {
    const updateSection = document.querySelector("#update-row");
    updateSection.hidden = false;
    document.querySelector('#update-row-btn').dataset.id = id;
}
```

Once the user clicks the update button presented in the previously hidden section, the request is sent.

```
/*
    Sending the PATCH request, and if successful, reloading the page. Ideally this is not
 the best way to display the results but it will work for now.
*/
const updateBtn = document.querySelector('#update-row-btn');
updateBtn.onclick = function () {
    const updatedInput = document.querySelector('#update-input');

    fetch('http://localhost:5000/update', {
        headers: {
            'Content-type': 'application/json'
        },
        method: 'PATCH',
        body: JSON.stringify({
            id: updateBtn.dataset.id,
            name: updatedInput.value
        })
    })
        .then(response => response.json())
        .then(data => {
            if (data.success) {
                location.reload();
            }
        })
}
```

back-end server

```
// UPDATE
app.patch('/update', (request, response) => {
    const { id, name } = request.body;
    const db = dbService.getInstance();

    const result = db.updateNameById(id, name);

    result
        .then(data => response.json({ success: data }))
        .catch(err => console.log(err));

});
```

database handler

```
/*
    This method sends the update query
*/
async updateNameById(id, name) {
    id = parseInt(id, 10);
    try {
        const response = await new Promise((resolve, reject) => {

            // Parameterized the values to protect against SQL injection
            const query = "UPDATE mock_table SET name = ? WHERE id = ?;";

            connection.query(query, [name, id], (err, result) => {
                if (err) reject(new Error(err.message));
                resolve(result.affectedRows);
            });
        });
        return response === 1 ? true : false;

    } catch (error) {
        console.log(error);
        return false;
    }

}
```