Reflection Report

Sindre Gangeskar

Introduction

The purpose of this exam is to create two separate applications. The first application acts as a back-end e-commerce API, whereas the second application acts as a front-end admin dashboard.

Back-end Summary

Creating a back-end API that acts as an e-commerce application with database relations took some planning. Trial and error was a key element to my success in this process, but staying focused on the task at hand greatly helped me solve one problem at a time. Referencing my Jira tasks as I continued to develop my back-end API kept me focused and prevented me from straying away from my current task.

I strived for clean code and did my best to follow the DRY principle. With prior experience, I decided to create a custom error handler middleware and an async handler middleware. I wrapped the async handler around my router asynchronous functions to efficiently handle any errors thrown by my different service files. The async handler resolves a promise, and it would catch the errors being thrown and pass it along to my custom error handler which I use in my app.js file. This resulted in a very clean router code block and prevented the necessity of using try-catch blocks in my router functions.

Following each task in Jira allowed me to maintain focus, which made the whole process feel more approachable, and less intimidating. Working in smaller chunks with individual sprints greatly helped me enjoy the process even when challenges arose.

I also chose to use the '**jsend'** package to ensure the same data structure was applied to every response returned by the back-end API.

Implementing authentication with **JWT** (JSON Web Token), I referenced its documentation[1] and applied my knowledge gained from school curriculum to ensure proper token signing was implemented.

While implementing testing, I applied my knowledge gained from school curriculum as well as referenced Supertest[2] and Jest's[3] documentation to ensure I implemented my tests properly.

In the end, I felt happy with the work I put into my back-end application, and if there was more time I'd try to improve it further. It was a great learning experience, and I had a lot of fun developing it.

Front-end Summary

By ensuring proper user authentication was implemented to prevent unwanted users from gaining access to the admin dashboard required some planning, and I concluded that active validation per request made to the back-end API was an efficient way of doing so as an added security measure, as most endpoints used in the admin dashboard required admin rights, but some didn't which is why I went with this approach. Checking for the **JWT** known as '**token'** in the cookies per request would allow the back-end to verify the user's role and reply with an appropriate response to the front-end server.

With validation out of the way, I still had to design my views the way I wanted them.
Using Bootstrap along with some custom styling, I managed to create a dashboard I felt happy with. The designing part was an incredibly fun experience, and I tried to apply my prior experience with EJS by re-using partials wherever I could as re-usable elements. Including what I've learned from my hobby projects and school curriculum has helped me greatly in understanding the importance of authentication

---

[1] https://www.npmjs.com/package/jsonwebtoken
[2] https://github.com/ladjs/supertest
[3] https://jestjs.io/docs/getting-started

and validation to ensure consistent data is kept, and that security is upheld throughout the use of an application that requires administrator privileges.
Overall, my experience with developing the front-end was very enjoyable, and seeing both the back-end and the front-end applications communicate with each other felt like another major accomplishment.

The challenges I faced along the way

While establishing the relationships with the database and staying persistent, I started to feel confident in the database relationships I set along the way, and I feel like I learned quite a lot from it. The one thing that challenged me the most was how I would handle the orders, and the products that were associated with a given order and the related user.

By creating a junction table called **'order_items'**, I could normalize the database structure and maintain a clear relationship between the **order**, its **items** and the **user** in question in a comprehensive way. It also allowed me to get an overview of the items associated with each order. This process helped me understand the importance of database normalization, and the value of keeping things as simple as possible. Another challenge I faced was to prevent the deletion of a brand or a category record that a product was using. Luckily, I encountered a similar issue before, and I was able to solve it fairly easily thanks to my prior experience. The solution was to restrict the deletion in the functions below for both the category and the brand. When an admin attempts to delete a brand or a category in use by one product or more, a custom error is handled in the back-end and then displayed for the admin to see on the admin dashboard.

```
Category.associate = function (models) {
  Category.hasMany(models.Product, { foreignKey: 'CategoryId', onDelete: 'restrict' });
}

Brand.associate = function (models) {
  Brand.hasMany(models.Product, { foreignKey: 'BrandId', onDelete: 'restrict' });
}
```

I needed to refresh my memory and used the documentation as reference. [4]

Populating the database

Before populating the database, I knew there was an issue gathering the brands and categories from the initial products API we used to fetch our initial data. To populate, I had to include only one copy of each category and brand because they are defined as **unique** by their names in the database ERD. To prevent a duplication error being thrown by Sequelize, I learned about using **sets**[5], which ignores any duplicates when assigning all data to them. ChatGPT[6] helped me understand how to populate the category and brand arrays with their data by providing me with its own general examples to help me further understand the syntax behind it all. The set ensures no duplicates, and the spread operator turns it back into an array with all the different values, and one last map to turn it into an array filled with objects that contain the name property for each unique category and brand.

```
categoriesArr = [ ...new Set(productsArr.map(x => (x.category))) ].map(x => ({ name: x }));
brandsArr = [ ...new Set(productsArr.map(x => (x.brand))) ].map(x => ({ name: x }))
```

[4] https://sequelize.org/docs/v6/core-concepts/assocs/#ondelete-and-onupdate
[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set
[6] https://chatgpt.com/

Raw SQL queries while getting products, and searching for products

I had prior experience with raw SQL queries but was less experienced when it came to join expressions, and I had to reference the MySQL documentation.[7] I also learned about using replacements and query types as options from Sequelize's documentation.[8]

I learned I could Include the 'type' property in the query's options with a value of 'QueryTypes.SELECT'. This allowed me to exclude metadata and keep only the relevant data. I could then return the result directly rather than returning the first element of an array. I could also insert a placeholder object into the 'replacements' property, which allowed me to reference the value without using a template literal.[9]

Example:

```
const query = `SELECT * FROM Products WHERE Products.id = :id`
await this.sequelize.query(query, {type: QueryTypes.SELECT, replacements: {id: id})
```

This has become my favorite way of handling queries, and I much prefer dealing with raw queries with Sequelize instead of its JavaScript functions in cases where you want more control and to make it easier to read.

Email and phone validation

Validation of email and phone were to ensure they followed the appropriate formats. I chose to test the values with regular expressions (regex) to ensure correct format in each of the variables were followed. It would allow me to easily throw appropriate errors if they failed the regex test. I used regex101[10] which is an amazing tool, and one that helped me understand where my regex would fail as I was constructing them with their recording feature. It allowed me to observe each step it makes as it goes through a string's testing. I picked regex because I've had prior experience in dealing with regular expressions, but not to the extent of constructing a proper email regex. It was a lot of fun and I learned a lot from doing this.

Email Regex

I chose to create a matching name group for the local part of the email, which also allowed for any amount of additional sub-local parts identified by the (*) quantifier. By including the non-capturing group, it would help ensure emails with multiple parts are matched. I chose the same approach with the domain part to ensure as many emails as possible would be matched without overcomplicating it. I chose to make the email regex case-insensitive to ensure that only the email's format was followed.

```
const emailRegex = /^(?<local>[a-z]+(?:[\.\_\-a-z][a-z0-9]+)*)@[a-z]+[.][a-z]+(?:[.][a-z]+)*?$/i
```

After a successful validation, I would turn the email value into a lowercase string before insertion into the database. That way it always follows the email format first, and then applies a lowercase to every email being passed in, which ensures every email is consistent in its formatting and adheres to most email standards.

---

[7] https://dev.mysql.com/doc/refman/8.4/en/join.html

[8] https://sequelize.org/docs/v6/core-concepts/raw-queries/

[9] https://sequelize.org/docs/v6/core-concepts/raw-queries/#replacements

[10] https://regex101.com

Phone Regex

Constructing the phone regex entailed creating a character set that accepts any digit from 0-9 including any whitespace in one or more occurrences. My project will replace any whitespace with an empty character to trim the phone number into an integer-based string after a successful test.

```
const phoneRegex = /^[0-9\s]+$/
```

Conditionally including objects & variables in my custom fetch function

I wanted to create my own custom fetch function to simplify my approach in making requests with everything necessary. This would entail including a body if there was one, and the potential token cookie if it existed. The challenge I faced was that I wasn't sure of the syntax I could use to conditionally include an object in the request by checking if the body variable wasn't null. A normal ternary operator alone didn't work, so I searched on Google and found this post on Stack Overflow[11] which helped me understand how I could include objects and variables by using the right syntax in the request. By using the spread operator and including my desired token cookie inside the parentheses, I could assign an object with 'Authorization: Bearer <token>' if the cookie token existed inside the header object, else it would be an empty object in its place. I also applied this approach for the body, but with a null in value instead if the body didn't exist in the request.

```
  const response = await fetch(`${process.env.BACKEND_HOST}/${endpoint}`, {
        headers: {
          Accept: 'application/json',
          'Content-Type': 'application/json',
          ...(req.cookies.token ? { Authorization: `Bearer ${req.cookies.token}` } : {}),
        },
        method: method.toUpperCase().trim(),
        ...(body ? { body: JSON.stringify(body) } : null),
        credentials: 'include'
      })
```

Swagger documentation

The challenge I faced during the Swagger documentation phase was to maintain clarity in all the different descriptions, status codes and examples that would show in the doc endpoint. I chose to create examples of each response to display all the different outcomes, so that anyone referencing the documentation would easily understand what everything meant. I referenced swagger-autogen's documentation[12] to help me expand my knowledge on swagger-autogen's syntax, which in turn allowed me to improve my documentation in each router endpoint by providing better examples and descriptions in my own swagger documentation.

[11] https://stackoverflow.com/questions/11704267/in-javascript-how-to-conditionally-add-a-member-to-an-object

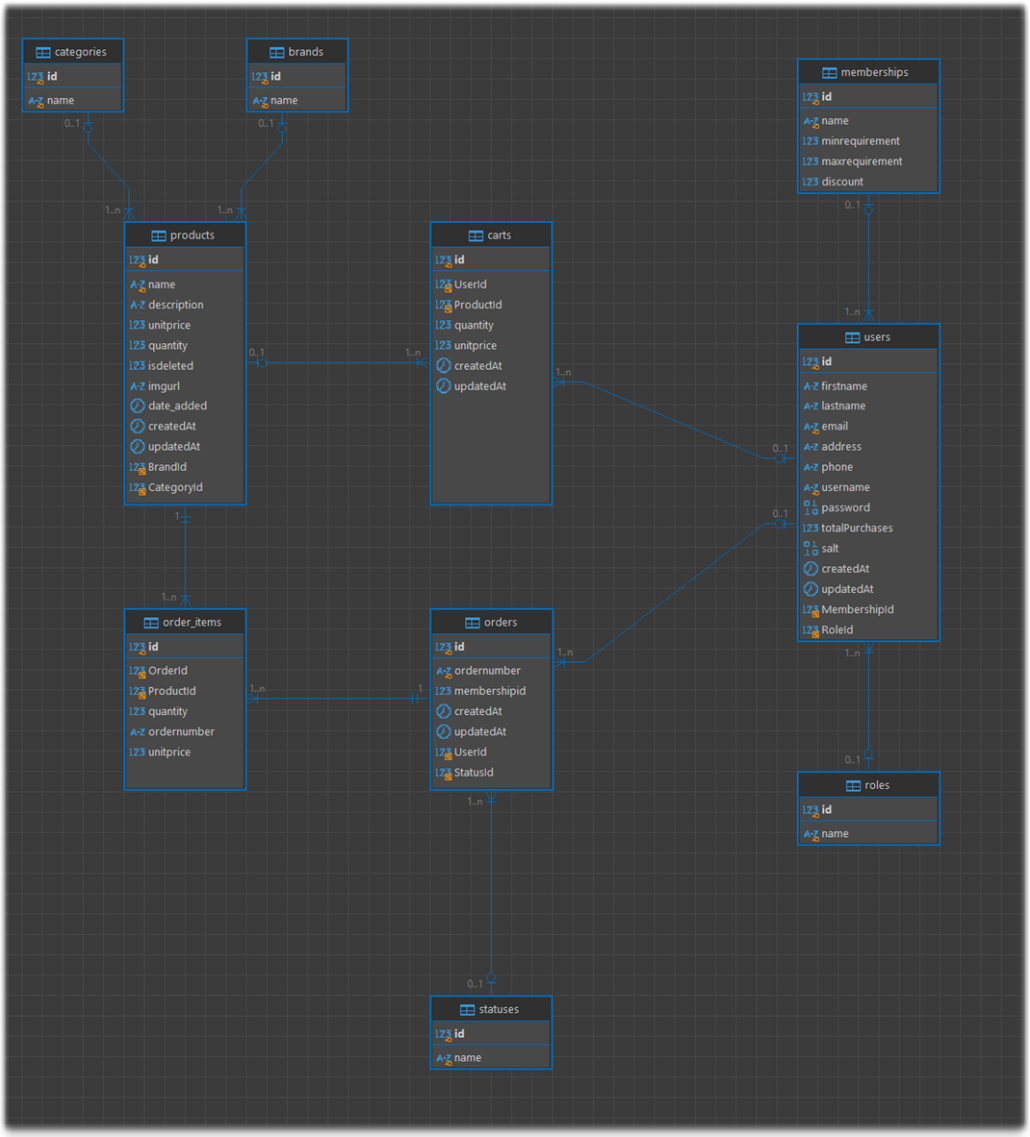
[12] https://swagger-autogen.github.io/docs/

Conclusion

With all challenges I faced, and throughout the progression of this project, I can easily say that I've learned a lot from this, and staying persistent even when failure occurs is a great way to learn. When in doubt, looking through documentation is always a good thing, and stepping away from the project to take a 'breather' can provide a refreshing take, which in turn can give a new perspective. It allows me to look at the problem and potential solutions from a different angle.

It has been incredibly fun and an invaluable experience. I feel like I've improved my skills and logical thinking as a developer and will continue to try to improve as I take on other future projects.

Database ERD



Categories

- Using its id as a foreign key in the products table, it represents a one-to-many relationship with the products table which means that a product can have only one category, and a category can have many products.

Brands

- Using its id as a foreign key in the products table, it represents a one-to-many relationship with the products table which means that a product can have only one brand, and a brand can have many products.

Memberships

- Using its id as a foreign key in the users table, it represents a one-to-many relationship with the users table which means that a user can have only one membership, and a membership can have many users.

Roles

- Using its id as a foreign key in the users table, it represents a one-to-many relationship with the users table, which means that a user can have one role, and a role can have many users.

Statuses

- Using its id as a foreign key in the orders, it represents a one-to-many relationship with the orders table, which means that an order can have only one status, and a status can have many orders.

Orders

- Using its UserId as a foreign key, it represents a many-to-one relationship with the users table which means a user can have many orders, and an order can have one user.

- Using its StatusId as a foreign key, it represents a many-to-one relationship with the statuses table which means many orders can have one status, and a status can have many orders.

Order_Items

- Using its OrderId as a foreign key, it represents a many-to-one relationship with the orders table which means order_items can have one order, and an order can have many order_items.

- Using its ProductId as a foreign key, it represents a many-to-one relationship with the products table which means order_items can have one product, and a product can have many order_items.

Carts

- Using its UserId as a foreign key, it represents a many-to-one relationship with the users table which means many carts can have one user, and a user can have many cart items.
- Using its ProductId as a foreign key, it represents a many-to-one relationship with the products table which means that many cart items can have one product, and a product can have many cart items.

Jira Timeline