Assignment 3: Swiftcart Docker Containerization – Brief Report

Daniel Mason

BDV 102: Interactivity and Databases

The primary goal for the second part of the capstone project was to implement a functional e-commerce backend, called **Swiftcart**, using Node.js, Express, PostgreSQL, via the Neon database service, using the Sequelize ORM. In this report, I will present my **design choices, API endpoints**, and the **challenges** I faced along with how I overcame them.  
**- Design Choices –**

I chose a relational approach using PostgreSQL because it provided strong consistency guarantees and was suitable for e-commerce requirements (e.g., inventory tracking). The main tables include:

* site\_user for customer data
* product and product\_category for items in the store
* shopping\_cart and shopping\_cart\_item to hold cart data
* shop\_order and order\_line to capture finalized purchases

This schema was structured flexible to accommodate future extensions, such as user addresses, payments, and shipping details. I chose Node.js for the backend since it offered a lightweight, event-driven environment that paired well with RESTful services. Express provided a straightforward way to define routes and handle middleware for JSON data parsing, error handling, and CORS.  
Furthermore, using an ORM, like Sequelize, allowed me to map JavaScript classes to database tables. This simplified CRUD operations and kept the code more sustainable. Additionally, Sequelize’s migration or sync() features could handle schema creation, which was convenient during rapid development. This approach helped ensure that each resource (e.g., products, cart, orders, customers) had its own set of routes. Lastly, I followed a REST approach:

* **GET** for retrieving data (e.g., /api/products, /api/cart/:id)
* **POST** for creating resources (e.g., /api/cart/:cartId/add)
* **DELETE** for removing resources (e.g., /api/cart/:cartId/remove/:productId)
* **PATCH** for partial updates (e.g., /api/cart/:cartId/quantity)

To keep credentials secure and ensure good configuration management, I stored database connection details (host, user, password) and server port in a **.env** file. This file is excluded from version control to avoid exposing secrets publicly.

**- Challenges and Resolutions -**

**1.** Database Connectivity with Neon

* Challenge: Securing an SSL connection to the Neon database in a local Node.js environment.
* Resolution: I configured Sequelize to use the ssl dialect option with rejectUnauthorized set to false. This allowed Node.js to connect to Neon securely without certificate issues.

2. Ensuring Data Integrity in Orders

* Challenge: When placing an order, multiple items could be purchased simultaneously, risking inconsistent stock if another user also buys the same product.
* Resolution: I used transactions provided by Sequelize. The code to decrement product inventory, create order lines, and clear the cart is wrapped inside a transaction, preventing partial updates if an error occurs.

3. Configuration Management

* Challenge: Hiding database credentials and preventing them from leaking into the repository.
* Resolution: I implemented a .env file and added it to .gitignore to ensure credentials remain local. Clear instructions in the README detailed how to set up environment variables.

4. Maintaining Code Quality and Structure

* Challenge: Keeping controller logic, routes, and models organized, especially as new features were added.
* Resolution: I separated the application into controllers (business logic), models (database entities), routes (Express routing), and config (database setup). This modular approach made the code more maintainable.

5. Dealing with Insufficient Stock

* Challenge: Users might try to add or update cart items with quantities exceeding the available stock.
* Resolution: I added checks in both the “add to cart” and “update quantity” endpoints. The code returns a 400 Bad Request error if the requested quantity is greater than what is available.

**- Trade-Offs -**

One trade-off that I dealt with involved deciding between Raw SQL and Sequelize. Using an ORM (Sequelize) makes development faster and more readable, whereas raw SQL sometimes offers greater control and potential performance benefits. I ultimately chose to go with Sequelize, as it was outlined as a requirement, and it made the development of this more readable for others. I also could work on the project at a faster, more efficient rate.

The other trade-off that I encountered involved security features, and if this implementation should include fully robust authentication (e.g., JWT tokens) which handled user sessions. However, due to the scope of this assignment, these features were not fully implemented, but the infrastructure was designed for easy addition down the line.

- **Conclusion** -

Overall, the Swiftcart application demonstrated a reliable backend architecture for an e-commerce platform. By leveraging Node.js, Express, Sequelize, and a PostgreSQL database hosted on Neon, I have built a system capable of managing customers, products, carts, and orders. Key focus areas included data integrity, scalability, and security of credentials. Future expansions might include adding user authentication, integrating payment gateways, and more detailed order tracking. Lastly, here the GitHub link for this project, <https://github.com/danielmason89/bdv_102_capstone_project_part_2_documentation_daniel_mason/tree/main>.

*References*