Capstone Project, Part 1: Swiftcart E-Commerce Database & API Design – Brief Report

Daniel Mason

BDV 102: Interactivity and Databases

In completing this part of the capstone, I followed a **‘**Design First**’** approach which prioritized clarity and user-centric development.

**- Design Choices -**

1. **Entities**

To fulfill the requirements of this assignment, I have created the following core tables, (entities):

1. **site\_user** – manages user login and basic contact information.
2. **product** – stores details of each product, including SKU, price, and category.
3. **shopping\_cart** – identifies the cart instance linked to a specific user.
4. **shopping\_cart\_item** – functions as a bridging, (junction) table that links products to a specific cart, allowing multiple products to reside in a single cart.
5. **shop\_order** and **order\_line** – facilitates order placement, capturing each product line-item and total costs.

Moreover, other tables (e.g., **product\_category**, **payment method**, **address**) were created to further refine the schema for this assignment. This level of normalization ensures data is stored efficiently while remaining easy to query and update.

1. **Relationships**

* **One-to-Many**: One user can have many **shopping\_carts** (though typically one active cart at a time). One category can have many products, one product can appear in many **shopping\_cart\_item** entries over time, etc.
* **Many-to-One**: Many cart items refer to a single cart, many products belong to one category.
* **Bridging Table**: **shopping\_cart\_item** connects the **shopping\_cart** and **product** tables to handle multiple products in a single cart.

These relationships are outlined in the provided ERD and reflect typical e-commerce logic where a single user can have multiple orders, and multiple line items can roll up into one order.

1. **Use Cases**
2. **Browsing and Adding Products to Cart**

* The user navigates the product catalog. When a user finds a product to purchase, a **POST** request is made to /cart/{cartId}/items to add it.
* Database Impact: A row is inserted into **shopping\_cart\_item** linking the cart and product ID, with a quantity field.

1. **Updating Cart Items**

* The user can change item quantities (increase or decrease) or remove items.
* Database Impact: A **PUT** request to /cart/{cartId}/items/{itemId} updates the quantity in the **shopping\_cart\_item** table. A **DELETE** request removes it entirely.

1. **Placing an Order**

* The user proceeds to checkout, selecting payment and shipping details.
* Database Impact: A **POST** request is made to /orders, creating a new **shop\_order** record and populating **order\_line** with item details from the cart.

These scenarios show how the tables relate and what data needs to be stored or passed via HTTPS.

**- Challenges and Resolutions -**I encountered and overcame multiple challenges in creating capstone part #1. They ranged from storing multiple products in a cart, where the challenge was how to best represent multiple products for one cart without duplicating data. I solved this by introducing a shopping\_cart\_item bridging table, referencing both shopping\_cart and product to normalize the data and prevent duplication.

The second issue was distinguishing between adding a new product to the cart or updating the quantity of an existing product, or handling cart item updates vs. inserts. I implemented application logic and/or a unique constraint on (cart\_id, product\_id) to ensure each product appears only once per cart. Updates to quantity then become a **PUT** operation; new products are a **POST**.

The last challenge was balancing normalization vs. performance where highly normalized schemas can introduce more JOIN queries, which might impact performance if not optimized. However, given that this is a core e-commerce schema, normalization remains valuable to maintain data consistency. Any performance overhead is minor compared to the benefits of data integrity.

**- Trade-Offs –**

Considering Simplicity versus Flexibility. I approached this capstone using a relatively minimal design without impeding future scalability. I did so by introducing a separate shopping\_cart\_item table to support multiple products and grow easily with new features (e.g., product variations, discounts). Additionally for Data integrity versus Complexity, I implemented a design with multiple tables that enforces stricter data integrity (e.g., references to valid products, categories). This increased initial schema complexity but ensured clearer boundaries and relationships between entities.

- **Conclusion** -  
This design implementation met the stated requirements of:providing at least 3 entities (site\_user, product, shopping\_cart, etc.) with clearly defined relationships,outlining 5+ CRUD operations in the OpenAPI specification (e.g., create a user, retrieve products, add to cart, update cart item, remove from cart, place an order), following a “Design First” approach via the capstone\_part\_1\_openapi.json file which documented each endpoint’s parameters and expected responses.In conclusion following, this approach included creating the SQL scripts: create\_tables.sql and create\_products.sql, creating the schema for Swiftcart with relationships, keys, constraints, and at least 20 sample products across different categories.Moreover, here’s a link to some documentation that I created, <https://github.com/danielmason89/bdv_102_capstone_project_part_1_documentation_daniel_mason>. This documentation helped me adhere to these guidelines and using modern design tools (e.g., apidog.com for OpenAPI and an ERD diagramming tool), provided a solid foundation for an e-commerce backend that is scalable, maintainable, and user-centric for part 2 of this capstone project.

*References*

Aaron Jack. (2021, June 3). *What is an API (in 5 minutes)* [Video]. Youtube. <https://www.youtube.com/watch?v=ByGJQzlzxQg>.

Apidog Docs. (n.d.). *Design APIs in Apidog*. Apidog Docs. <https://docs.apidog.com/overview-533969m0>.

Begley, N. (2023, May 8). *Documenting REST APIs with OpenAPI*. Doctave. <https://www.doctave.com/blog/documenting-rest-apis-with-openapi>.  
Database Star. (2022, August 30). *eCommerce Database Design: Diagram & Explanation* [Video]. Youtube. <https://www.youtube.com/watch?v=1HamqOuv2Cw>.

IBM Technology. (2020, November 19). *REST API and OpenAPI: It’s Not an Either/Or Question* [Video]. Youtube. <https://www.youtube.com/watch?v=pRS9LRBgjYg>.  
Lucis Software. (2023, October 27). *Entity Relationship Diagram (ERD) Tutorial – Part 1* [Video]. Youtube. <https://www.youtube.com/watch?v=hktyW5Lp0Vo>.

Lauert, A. (2019, November 6). *The Design of Web APIs*. MANNING. <https://www.manning.com/books/the-design-of-web-apis>.  
Lucis Software. (2023, November 11). *Entity Relationship Diagram (ERD) Tutorial – Part 2* [Video]. Youtube. <https://www.youtube.com/watch?v=xsg9BDiwiJE>.

Neon. (2024, February 16). *PostgreSQL CREATE TABLE*. Neon. <https://neon.tech/postgresql/postgresql-tutorial/postgresql-create-table>.

Neon. (2024, September 24). *PostgreSQL Tutorial*. Neon. <https://neon.tech/postgresql/tutorial>.  
Sofela, O. (2022, July 26). *Test-Driven Development Tutorial – How to Test Your JavaScript and ReactJS Applications*. freeCodeCamp. <https://www.freecodecamp.org/news/test-driven-development-tutorial-how-to-test-javascript-and-reactjs-app/>.

Veen, L. (2021, March 11). *What Is Cardinality in Data Modeling? The Theory and Practice of Data Cardinality*. Vertabelo. <https://vertabelo.com/blog/cardinality-in-data-modeling/>.

Voulgaris, N. (2018, March 10). *Designing a RESTful shopping cart*. Nikos Voulgaris. <https://nvoulgaris.com/designing-a-restful-shopping-cart/>.