Online Store Database Project

ECE 569 Database System Engineering
Rutgers University
Dr. Dov Kruger

Andrew Xie (akx2) Nikhil Mahalingam (nm1105)

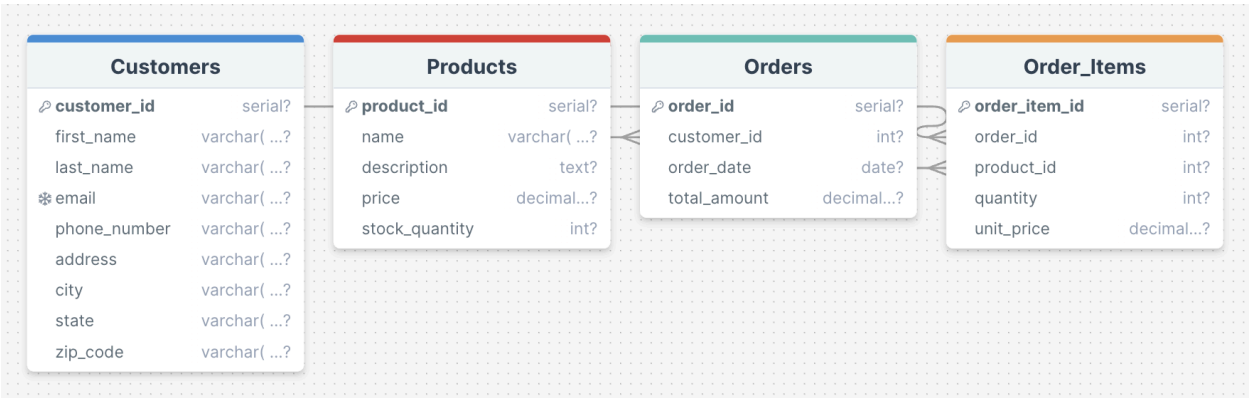
2024-05-06

1 Project Background

This project aims to construct a robust online store database system. It should fulfill most of the basic capabilities of a typical online store, such as handling inventory with the number and types of items, and updating inventory during store transactions (e.g. purchases and new listings). It should also provide a user authentication and login system so that each individual user can have their own unique shopping experience. The store will have a cart function that users can interact with during their browsing. We will implement these functionalities with tables and stored procedures in PostgreSQL and SQLAlchemy and use Python and Flask for the frontend.

2 Project Design

For this project, we aimed to explore new technologies related to databases so that we could get as much as possible out of the class and apply the theory across different environments. Since the frontend was less important, we settled on the old HTML/CSS that used Bootstrap/Jinja2 templating. For our backend, we handled it in Python and Flask, especially since it has great support for our primary database–PostgreSQL–which stored all our tables. In addition to that, we used the Object Relational Mapper (ORM), SQLAlchemy, which integrated with Python to write clean queries instead of raw SQL statements. Finally, we also used Supabase for user authentication and AWS S3 where we stored our images in buckets.
Here was our original database design:



Naturally, after testing we decided to make some tweaks to our original design.
Some were:

- Make email unique: For user auth purposes (explained in next section)
- Added a slug to the products table: Queried slug to frontend and appended to the AWS address where the images were stored:

```
<img src="{% if product.slug %}
https://nmonlinestoreece.s3.us-east-2.amazonaws.com/{{product.slug}}.jpg{%
else %}no_image.jpg{% endif %}">
```

- Changed miscellaneous rows depending on importance to be null or not
- Added `SERIAL` to ids (in mySQL it is `auto_increment`)

3 User Authentication

We handled user authentication through Supabase, which is an open-source firebase alternative which is actually a postgreSQL database under the hood. This allowed for seamless integration with our project, especially since they also have a great Python library. Supabase utilizes JSON Web Tokens (JWT) and creates a session on log in where the user is assigned a token. JWT allows the user to be authorized for registration and authenticated on login. This is particularly useful, as we can extract data from the session–in our case we stored the email address used and then queried the customer_id with the email (since email is unique). Once our customer_id was stored, we could use that to run our queries based on the logged in user. For example, here is how our queries were structured.

```
order = Orders.query.filter_by(customer_id=customer_id,status='cart').first()
```

Notice, customer_id as a parameter ensures that we are displaying the cart items of the customer that is logged in.

4 Features, Implementation, and Testing

This section contains a brief outline of the key features we implemented in Online Store and how they were achieved. It also details how certain components and user flows were tested, if applicable. All features below were tested in conjunction using various test user accounts (and an unlogged-in state) as well.

Store

The Store page of Online Store is the central hub for all product listings on the website. The Store page requires only one table lookup, which is *Products*. In this page, users must also be able to search for products and sort by various categories (price and stock quantity). These are achieved through query parameters (i.e. */store?query=<search_query>&sort=<sort_category>*). The Flask route will then parse these parameters and perform the corresponding operations on the *Products* table lookup. This functionality was tested manually by altering search and sort parameters and verifying the resulting product list output.

Cart and Checkout

The Cart and Checkout pages are where most of the transactional database logic lies. In order to display the items in a user's cart along with item subtotals and order total, we must aggregate. However, since each *Product* has a *quantity* and a *price*, we can simply calculate each item's subtotal as *quantity* * *price*. For an order's total value, we can iterate through all *Products*, calculate the subtotal for each, and have a running sum. In the Cart view, users will also be able to delete items from their cart if they no longer wish to order them. Once a user is satisfied with their cart, they can proceed to checkout.

In the Checkout view, an order summary is displayed with the order date, total, and items. There is also a “Trending Items” section that is intended to entice users into adding items to their order. This section is populated by querying the *Orders* table for the most recent entries. The items contained in the most recent orders are then compiled and the top two are displayed in the Checkout view. Finally, when a user places an order, several operations must be performed:

- Verify that the user has items in their order. If not, the order obviously cannot be processed.
- Verify that all products being ordered have sufficient stock to fulfill the order. If not, block the order and display an error message detailing which items have insufficient stock.
- If the previous two checks pass, the order status is set to *OrderStatus.completed*.
- If the previous two checks pass, each item in the order will have its *stock_quantity* decreased by the appropriate amount corresponding to the order.

Since this user flow performs the most backend operations, it was tested extensively to cover each edge case (empty cart, insufficient product stock, varying item quantities and types, etc.) to make sure they are all properly handled. Product stock quantities were also verified to correctly update depending on the order quantities.

Orders

The Orders page performs a simple query of the *Orders* table given the currently logged-in user’s email to obtain a list of all their orders. This is then displayed in a list in reverse chronological order. If an order’s status is *OrderStatus.cart* (i.e. not completed yet), a button will appear for the user to proceed to checkout and complete that order.

Sell

The Sell page is a simple input form that ultimately sends a POST request to the *Products* table to INSERT the input form values. It is important to note that the user need not enter a *product_id* because it is set to *auto_increment* in our schema. Currently, a known limitation of the Sell page is that it does not support AWS images. This feature was built before images were implemented and it has not yet caught up.

Login, Logout, and Registration

The login, logout, and registration pages are all simple input forms that interface with Supabase, as described in the previous section. User account creation was tested extensively by registering for accounts with various input field combinations. We also verified that users could register with the bare minimum input (email and password). The login functionality was tested with both valid, invalid, and empty credentials to ensure all error states were gracefully handled.

5 Conclusion

The technical challenges we faced was building the full stack application from the ground up as well as the learning curve that came with using new technologies. However, the bulk of the challenges was coming up with edge cases and ways to break the application. We overcame

this by being proactive before adding features and considering many factors that a user could do to potentially break or exploit our program. We consistently tested our features and once we would find holes, we patched them shortly. For example, we needed checks such that the number of items ordered could not be negative or could not be greater than the inventory amount. We had thought of the latter case but once we realized that users could steal money by inputting negative values into the number of items, we patched it by making the minimum number of items that could be ordered 1 for all products.

Thus, we were able to not only use a lot of the theory we learned in class such as database normalization, transaction error handling, and query optimization, but we were also able to implement the same theory with a completely different stack. Thus, this helped us efficiently implement different types of databases into an applicable product. This is very useful as this is how databases are used across the stack in real software engineering applications.

The next steps for this project is to potentially add a data access object structure (DAO) in combination with the ORM. There were instances–especially with the login functionality–where the ORM was unable to run the highly customized queries that were required. This is where DOA would come in, as it would have custom queries already written so that they can be used often. This would allow us to focus more on the engineering and business logic which would make the process of development more efficient. Additionally, we would include more stored procedures for better automation, unit testing, etc.

In terms of features, we could implement a return feature which would allow customers to initiate them when they are unsatisfied with the product. Furthermore, having a system to authorize sellers so only verified, reputable users can sell products would help clean up the product page. This would require the additions of profiles as you could talk to the sellers to negotiate returns as well as have a selling profile to make a marketplace ecosystem.

In conclusion, we learned a lot through this project and there is a room to grow and expand our online store in the future.