E-Commerce Order and Inventory Manager

Student: Brian Kominick
Student ID: 1055998
Course: CSC530-02: Data Structures
Instructor: Ashik Bhuiyan, Ph.D.
Semester: Fall 2025
Submission Date: December 11, 2025

Contents

1 Introduction

This document details Brian's final project for CSC530: Data Structures at West Chester University.

1.1 Overview of Project

The purpose of this project was to enable practical applications of the data structures being studied in class, through the development of a full-stack Java-based e-commerce platform with support for both customer and admin workflows.

1.2 Main Goals and Expected Impact

The main goal for this project was to implement a functional, user-friendly GUI application with role-based functionality. Development took place over most of the semester, with a live demo delivered around the halfway point. The completion of this project generated experience in both backend and frontend use cases for a variety of data structures. Considering its scale, it also provided an opportunity for working on project architecture and user-experience practices.

2 System Features

This section will outline the main features of the project. Key features can be traced back to functional requirements given in the original assignment document, whereas bonus features include anything beyond that.

2.1 Summary of Key Features Implemented

Broadly, the key features cover the basic operations for both *Customers* and *Admins* roles where both are subtypes of a general *User* class. Some features are limited to the Admin role and will be noted as such.

2.1.1 User Login and Account Creation

New users may sign up for the platform by providing a unique email address and a password. Users can sign up as a customer or an admin (both subclasses of a parent User class). Certain functionality is associated strictly with the admin type and is not accessible by customers. For UI presentation, refer to figures 1 and 2.

2.1.2 Product Browsing

All users may browse the current inventory. Users may search by keyword and filter based on attribute values (date added, price, rating, category)3. Clicking on an item opens up a detailed view page with a full description, image (if supplied), reviews, a warning if stock is low, and options to add to cart or leave a review (See figure 4).

2.1.3 Inventory Management

Admins may update any attribute for an existing product (See figure 5), add new products (See figure 6), and delete products from the inventory.

2.1.4 Shopping Cart

Users may add, remove, and update quantities of items. The cart presents all line items with quantities and prices in addition to a subtotal for the cart (See figure 7). This page is where users may add discount or coupon codes for their purchase (See figure 8).

2.1.5 Order Management

Users may view their past orders and the fulfillment status associated with each one (See figure 9).
Admins may view all orders made on the platform and update the their fulfillment statuses. UI not yet implemented (See figure 10).

2.1.6 Reporting and Summary

Admins may view reports on the platform including KPI cards (See figure 11) and useful queries (not fully implemented yet).

2.2 Bonus Features Implemented

2.2.1 Persistant Storage

All user and inventory data is stored in a mysql database generated with docker.

3 Data Structures Used

3.1 List (Array-Based)

A list is an ordered collection of elements. Because I used ArrayLists, the elements were located contiguously in data which allowed for constant access time. Arrays are not dynamically sized, so in some places where I used them either to pass ordered items between components (Service Layer -¿ UI Layer) or store a collection of values as an attribute for an object (e.g. Product.tags is a list of associated categories, Order.contents is a list of items, etc.), allocating and copying new arrays is not much of a concern. However, I also used ArrayLists in areas where frequent additions to Lists were made (e.g. ProductTree inOrder returns an ArrayList). I believe this was an acceptable choice since I was only appending data, not inserting in the middle or removing. Even for large trees, since ArrayList doubles with each reallocation, there shouldn't be too many resizing operations happening.

3.2 Stack (Array-Based)

A stack is a collection of elements where the last element added is the first one to be removed. I used an array-based implementation of this structure as well. I used stacks for navigation history within the UI to enable a back button and for traversing the AVL tree. In the case of navigation history, since I only cared about going back and not forward, I didn't have to worry about removing items from the middle of the stack. Additionally, I didn't anticipate large histories, so resizing the stack would remain inexpensive. For tree traversal, after looking into the Java Stack implementation, perhaps ArrayDeque could have been a good choice as well, since we don't have to worry about thread safety in this case and ArrayDeque would remove the synchonization overhead.

3.3 Set (HashMap-Based)

Sets are unordered collections composed of unique elements. I use HashSets in the User class for storing permissions (an enum). Upon further inspection, I could have used an EnumSet to make things more efficient. A set is useful for this scenario because it offers fast lookups (constant time) for membership tests and enforces uniqueness by design (we don't need multiple of the same permission).

3.4 AVL Tree (Node-Based)

A tree is a graph with no cycles and exactly one path between any two nodes. These qualities make it useful for storing sorted data. Additionally, an AVL tree automatically balances itself when inserting elements. Its balanced nature reduces time complexity compared to other binary trees. Since I use this structure for storing products in memory, it's very useful for filtering and selecting ranges of data. Compared to a red-black tree, an AVL tree is more strictly balanced and so searches can generally be done more quickly.

4 Testing and Validation

4.1 Testing Strategy

Due to the time constraint on this project, no thorough testing pipeline was developed. Instead, I focused on manual testing of main features. There are some minimal unit tests as well. These tests are complemented by logging in the backend and validation of input forms in the frontend with basic checks for empty values, numeric values, price formatting, and email address formatting.

4.2 Edge Cases

Focusing on manual testing and happy-path scenarios, only some edge cases were caught in development with others surviving to the submitted build. When edge cases were caught, logic for the feature was redeveloped to account for the failing scenario and any similar cases.

5 Challenges, Solutions and Future Enhancements

Given the scope and context of the project, some challenges encountered during development were in design choices: weighing scalability, security, development time, and opportunities to practice certain techniques. For instance, since this project is for a class on data structures, I chose to implement an AVL tree for sorting instead of doing that in database queries. Some functionality could have been broken down more for better separation of responsibilities and security considerations, like role-specific UI components or services, but I chose to keep things more tightly grouped for faster development. Some specific technical challenges I faced were in learning the Java Swing framework. Having never worked with it before, it took some time to learn the different components and design patterns for developing a Swing app. Furthermore, having more backend-focused development experience, I struggled to keep my UI refreshed. Introducing the AVL tree as runtime storage also added another layer of data that I had to keep synced, and I ran out of time to fully test that. Moving forward, I think that the enhancements I make to this project would depend on what I want to learn and practice. Various areas need improvements such as UI presentation, test development, and unfinished feature development. I think the most important thing to me moving forward would be testing and refining current functionality in order to provide a more seemless, production-grade experience. In future projects, I plan to make testing a bigger part of my development process.

6 Conclusion

This project was a great opportunity for me to design a fullstack application. I got to practice designing use cases, breaking down features into tasks, and integrate a variety of domains (databases, domain model design, service logic, UI features). Applying data structures from class to a real-world context made me think about scaling and performance whenever I chose how to represent my data. Writing this report was also a fun opportunity to justify some of those decisions. I feel like I gained competence not only in deciding on and implementing data structures, but in other software engineering skills as well like system design. This project has given me concrete experience in these practices and new perspectives that I can bring to future development. I've also reaffirmed my strong feelings on incorporating testing early on in the development process, regardless of time constraints, to ensure strength in an applications foundation and streamline the addition of new features. I already held this feeling after years of working as a QA engineer, but developing an application myself that is intended for end-user consumption has certainly solidified it. Similarly, it's reinforced my belief that design is a crucial step before diving into development. Taking the time to plan out how my project would be organized definitely made working through it easier than it could have been, since I knew how I wanted to organize things when they came up (for the most part). Overall, I feel like I gained a lot from this project and hope to do similar work in the future.

A Screenshots (UI, test cases, output samples)

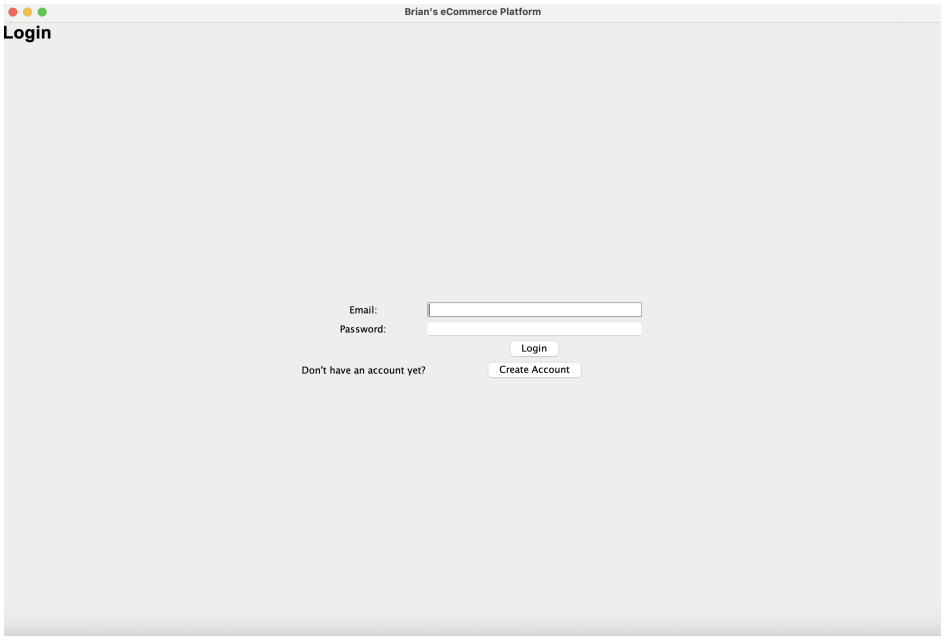

Figure 1: Login Panel: The landing page of the application

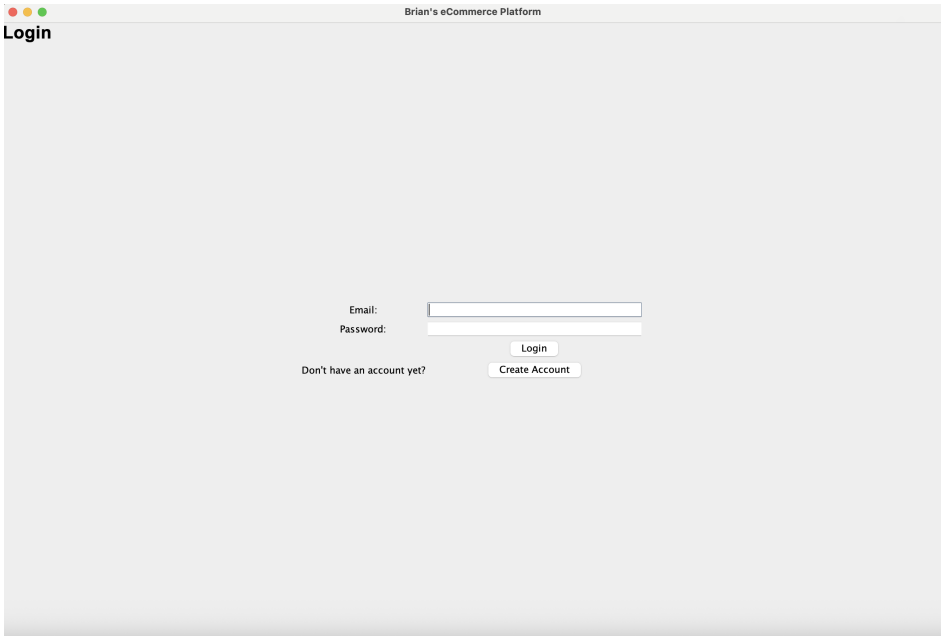

Figure 2: Sign-Up Panel: New users can create an account

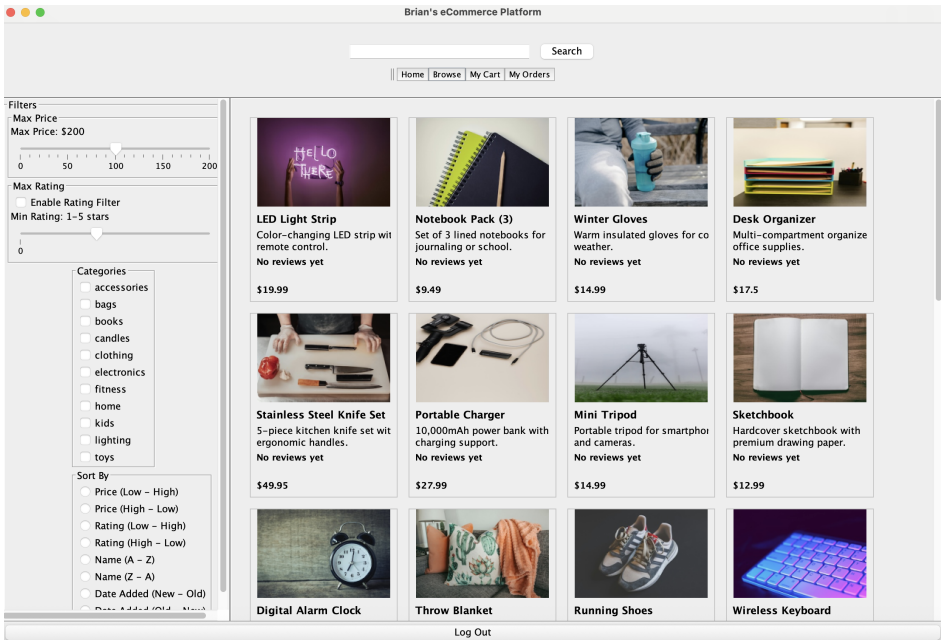

Figure 3: Users can browse available products.

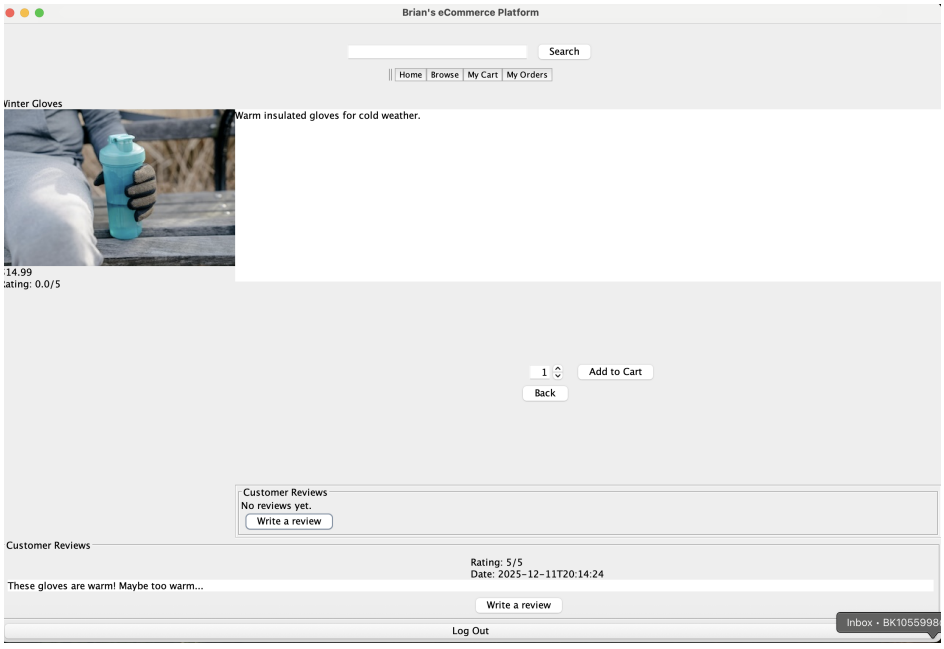

Figure 4: Users can leave reviews for products.

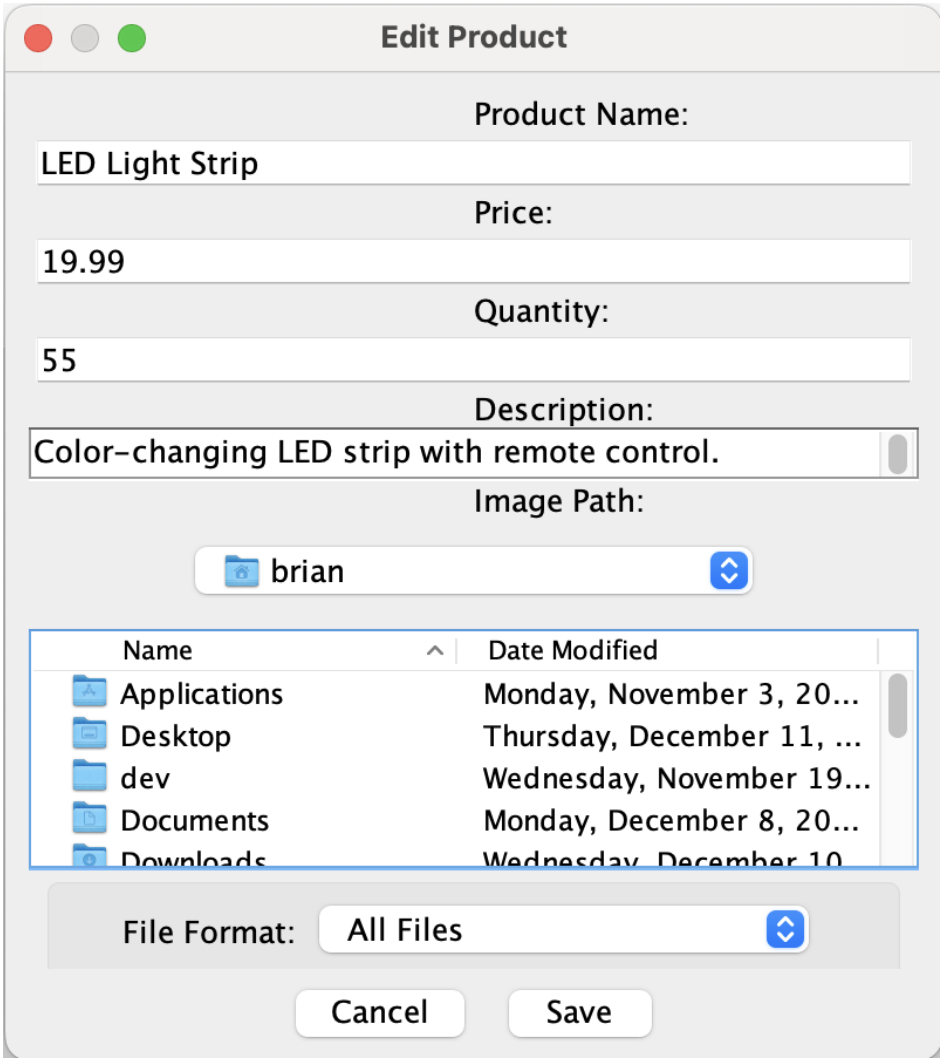

Figure 5: Admins can update existing products (fields prepopulated).

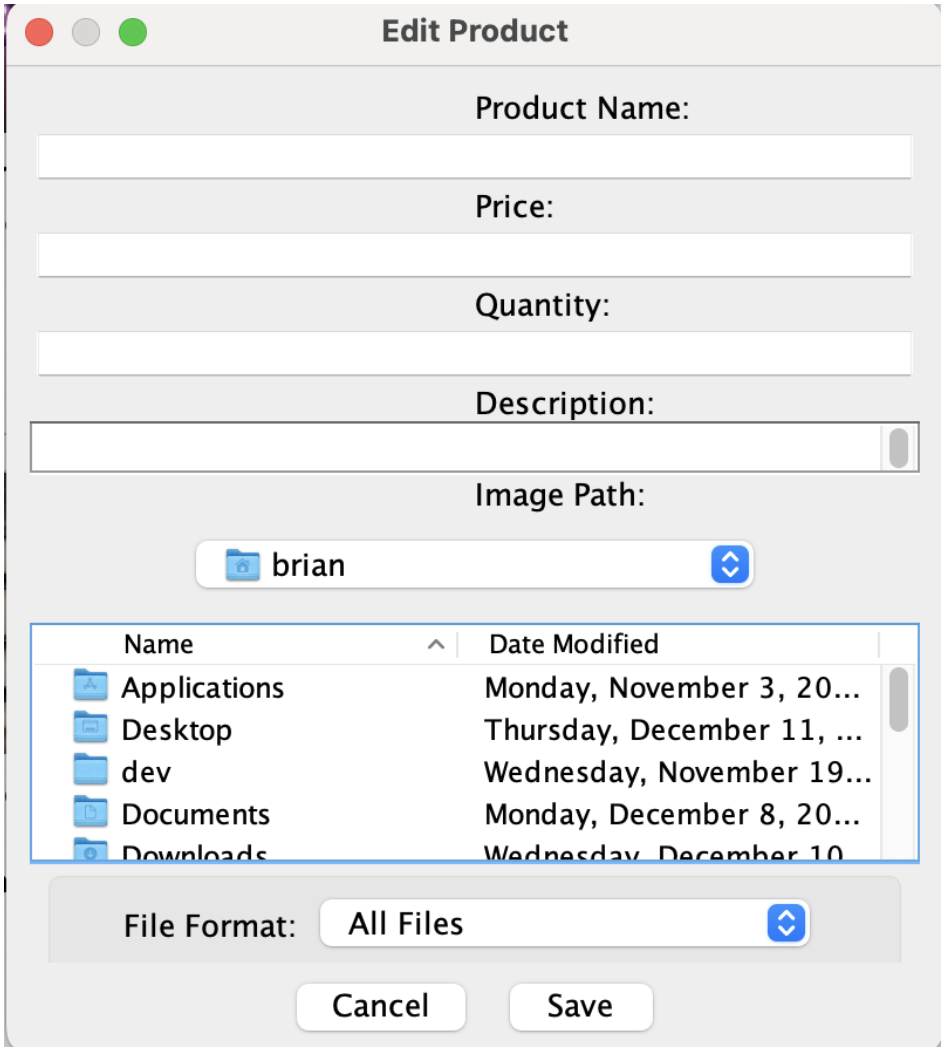

Figure 6: Admins can add new products (fields start blank).

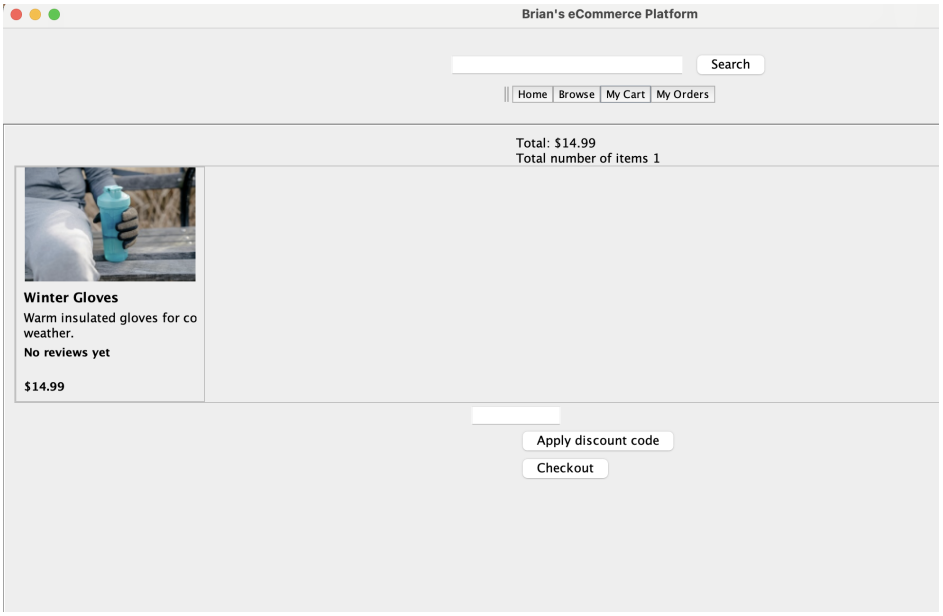

Figure 7: Users view and make changes to their carts.

Figure 8: Users can apply discounts to their carts.

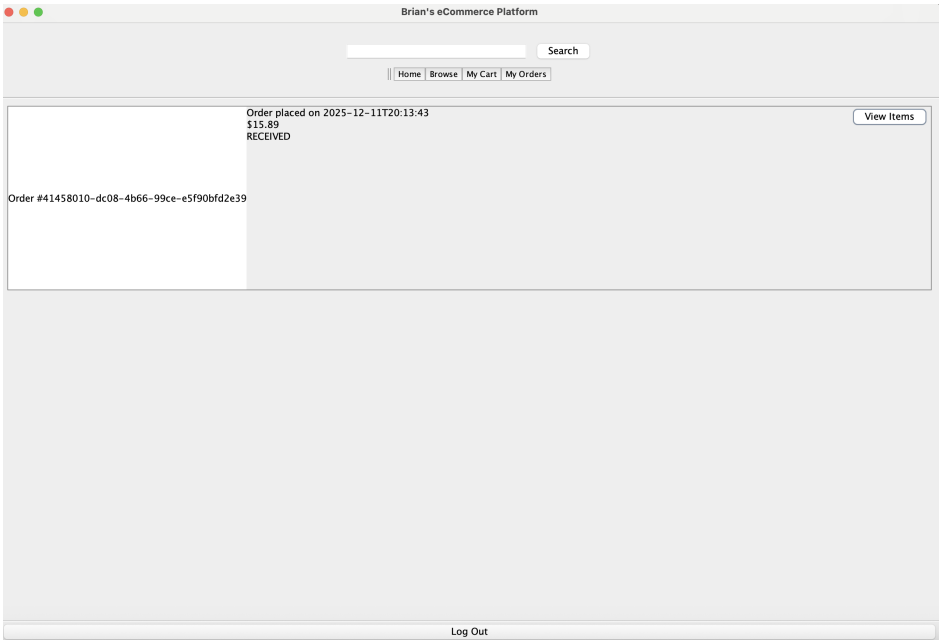

Figure 9: Users can view their past orders.

```
    public boolean updateOrderStatus(int orderId, Order.OrderStatus status) {
    if (!session.getUser().isAdmin()) {
        throw new SecurityException("Access Denied. User is not recognized as a
    }
    boolean result = true;
    String updateOrder = "UPDATE orders SET delivered = ? WHERE id = ?";
    try (Connection con = DatabaseConnection.getConnection();
    PreparedStatement stmt = con.prepareStatement(updateOrder)) {
        stmt.setString(1, status.toString());
        stmt.setInt(2, orderId);
        result = (stmt.executeUpdate() > 0);
    } catch (SQLException e) {
        logger.error("Could not update status of order with id {} to {}\nError:
        result = false;
    }
    return result;
}
```

Figure 10: Method for admins to update status of an order.

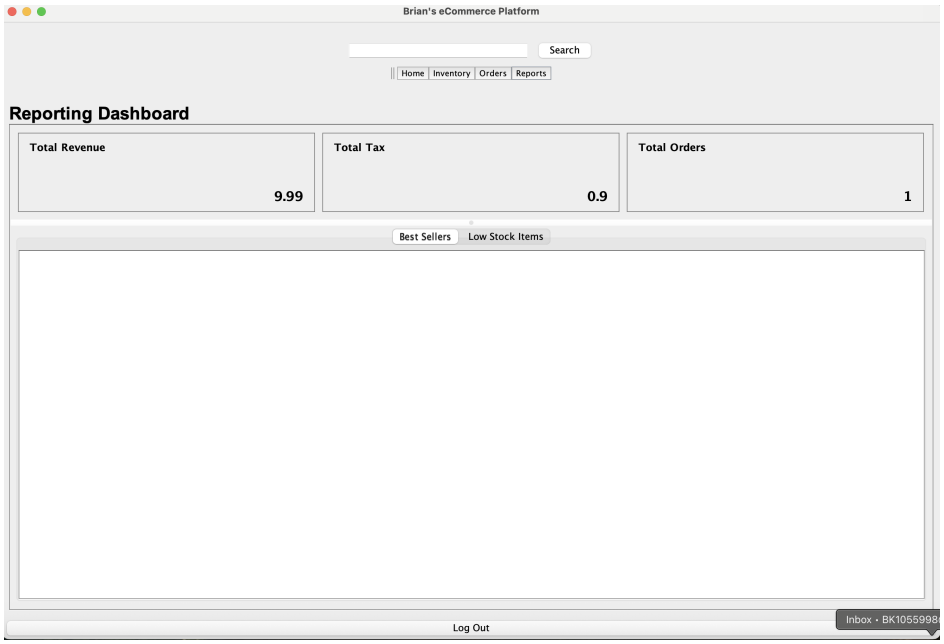

Figure 11: Admins have access to reports.

B References (libraries, online resources, etc.)