Design Analysis for Shopping Cart Application

Andrew Song (andrew90)

1. Overview

The shopping cart application, as its name already implies, is about "virtual" cart for online shoppers. The goal of this project is to introduce the authentication (Users are allowed to log in, and keep the items in their shopping carts) and authorization (Shoppers and shopkeepers have different roles and actions they can take) through the concepts of cookies and sessions. Another side goal of this project is to actually go through the whole process of software development; The process starts with brainstorming and designing architecture for the software, and eventually leads to actual implementation and finishing. The ultimate objective is to expand the application's functions suitable to more practical usage; these might include refactoring the code for stand-alone shopping cart application, allowing the application in several languages, and so on.

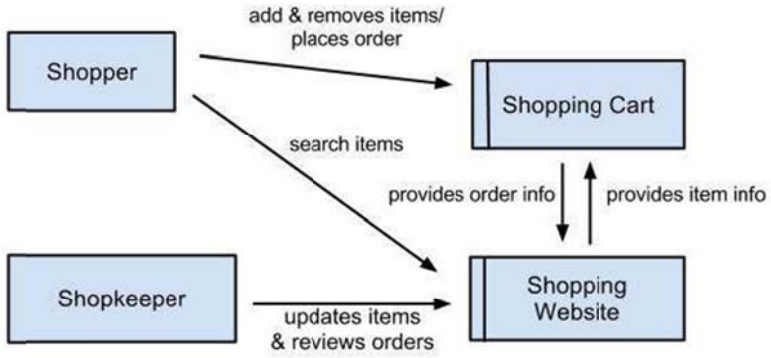

Fig 1. Context Diagram for Shopping cart application. A regular box refers to components external to this application.

Above is the context diagram for the shopping cart application. At this stage of development, the shopping cart application is embedded in the shopping website, although the shopping cart could be refactored to be a stand-alone application, rendering the shopping website as external component. Notice that the shopper does not directly interact with the shopping website; all the interactions with items (except for searching) and placing orders are mediated through shopping cart application.

2. Concept

The key models in the application are User, Cart, Order, CartItemJoin and Item. A user logs in as a shopper or a shopkeeper. The shopper is given a shopping cart to where he/she can add and remove items. When the shopper decides to place an order, an order object containing relevant information is created which can be reviewed by the shopper and the shopkeeper in the future. The shopkeeper can modify the existing items, or add new ones. However, since the shopkeeper

assumes the role of supplier, he/she will not have access to carts or placing orders. CartItemJoin model does not have actual meaning. It provides intermediate model and database through which carts and items can have many-to-many associations.

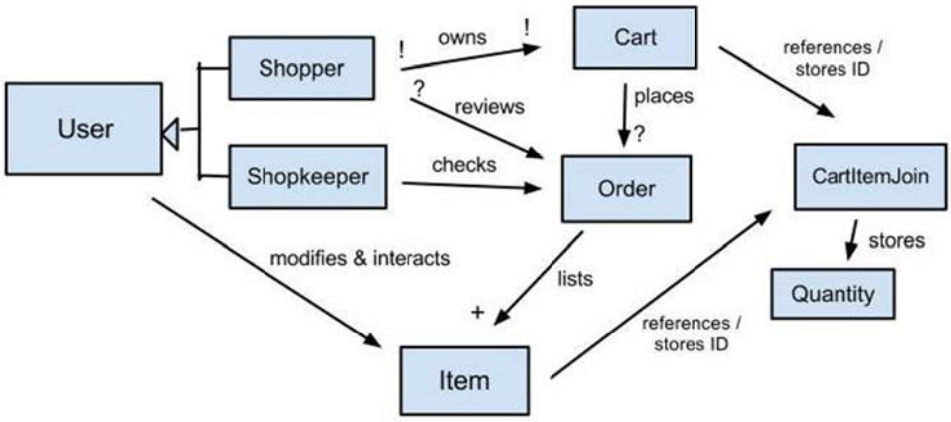

Fig 2. Object Model diagram of Shopping Cart Application

3. Behavior

3.1 Features

Shopper

- **Add & Remove & Update items** : Customer can add, remove, and update items to the cart
- **Place order**: Based on items in the cart, user can place an order.
- **Review orders**: Customers can review the details of their past orders.

Shopkeeper

- **Modify & add new items**: Shopkeeper can modify the description and price of items, as well as adding new ones.
- **Review orders**: Shopkeepers can review what orders have been placed and analyze the purchasing behaviors.

3.2 Security Concerns

The main security threat is session hijacking. If a shopper's session is hijacked by some malicious users, it is likely that orders of unwanted items of unwanted quantities would be placed with the hijacked shopper's credit card information. On the other hand, if a shopkeepers' session is hijacked, existing items might be deleted, or the prices might be reduced to ridiculously cheap levels, allowing the hijacker to place orders at the changed price. Such security threats might be mitigated through following methods (Some are referenced from "Ruby on Rails Security Guide"):

- Provide a secure connection over SSL. This will help prevent so-called "cookie-sniffing" in insecure wireless networks.
- Create a logout button and emphasize the importance of it! Failure to log out in public terminals might cause other malicious users to use the previously logged-in accounts in unwanted ways.
- Not storing sensitive information about users in cookies. Passwords, credit card information, and other security-required information should not be stored in cookies.

The security concern regarding internal functionalities of the shopping cart would separation of roles between users; shoppers should not be able to have any of the shopkeepers' functions and vice versa. Moreover, shopkeepers should not be able to check each other's sales and modify each other's items. Such aspect is covered in more depth in Design Challenges section.

3.3 User Interface

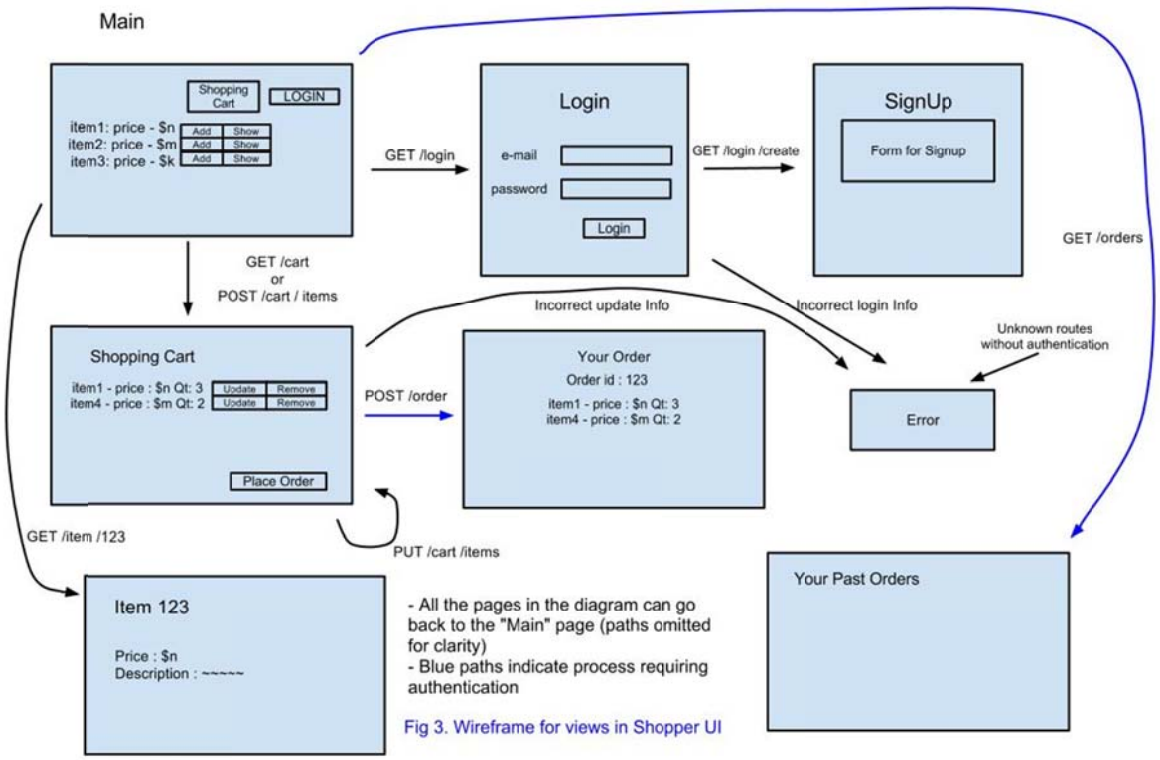

Fig 3. Wireframe for views in Shopper UI

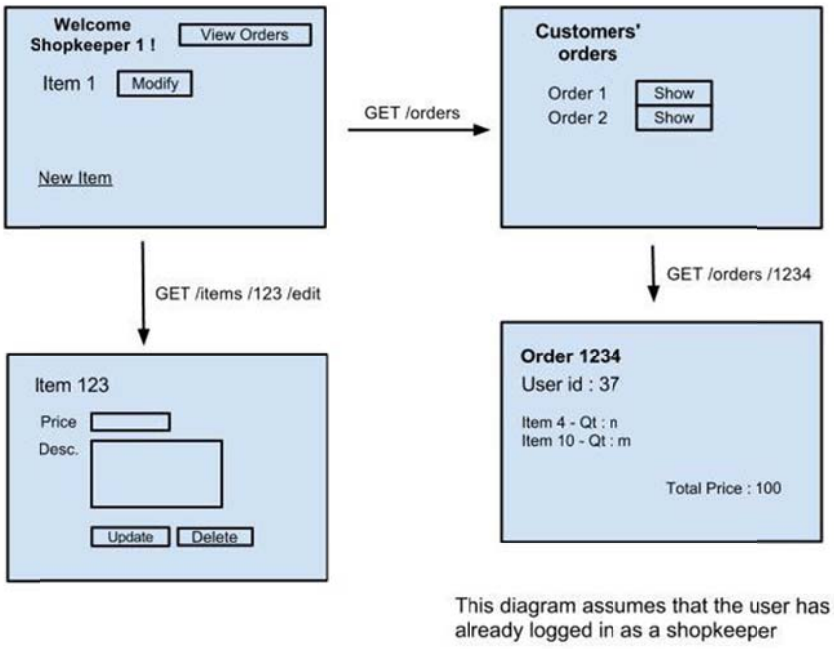

Fig 4. Wireframe for views in Shopkeeper UI

4. Design Challenges

- **Keeping track of Cart Items**

The main challenge of this project was to decide where to store the cart items, and how to keep track of quantity for each item. Having a "quantity" attribute for the Item model does not work since multiple shoppers are going to reference same item objects at the same time. Here are two alternatives to the challenge

1) Create a hash object and store it in the session variable once the shopper is authenticated. The session variable works like a "temporary cart" in this case. The hash would have "item_id" as key, and "quantity" as value. Only when the user logs out or places order, information in hash object is recorded to Cart or Order database. It is efficient in a sense that database query does not happen until the final moment. However such implementation makes concept of "Cart" almost useless, since session variable functions like a cart. Moreover, since the default session storage is cookie-store, the whole hash gets recorded in a cookie, increasing the vulnerability to security threats.

2) Create a model "CartItemJoins" to facilitate many-to-many associations between cart objects and item objects. The model has "cart_id", "item_id", and "quantity" attributes.

Cart objects can reference the table and check how many items of how many quantities are currently in the cart. Every information about the current cart would reside in the server-side as opposed to cookie-store.

Of these two options, I opted to go with the second one. The first one had serious side-effects of having to access session variables intensively. Moreover, handling and modifying hash objects were not as easy. However since the option uses cookies, it could be used in later implementations in which the user would be able to add items to the cart without logging-in.

- **Security issues regarding multiple shopkeepers**

One of the main issues regarding multiple shopkeepers is that all the information needs to be private. A shopkeeper should not be able to view the sales of items supplied by other shopkeepers; he should not also be able to modify items of other shopkeepers as well. To ensure such security, each item object has attribute called "keeper_id" which stores the id of the supplier. All the authorizations are done by comparing the "keeper_id" and the current user's id.

- **Actualizing the Order model**

One of the biggest code challenges was to implement order model. Order objects are required to keep track of the users' past orders; they are different from cart objects in that cart objects needs be reused whereas order objects need to be kept immutable. The challenge was that the order objects could not refer to the item objects by their ids, just like cart objects do. Even if a shopkeeper deletes items or modifies the prices, the order object should still be able to display information at the purchased time, although the information is not same anymore.

To record the snapshot of the object, I used hash serialization. When an order object is created, information about the order (item ids, names, prices, and quantities) is pushed into a hash object; the hash then gets serialized into "yaml" format, a simple text result of serialization, which gets stored into one of the order object's attribute "item_hash". When the shopper or shopkeeper wants to see the past orders, the hash would be deserialized for display. In this way, order objects would be immutable and correctly show the desired order information.

- **Authorization for Shoppers and Shopkeepers**

There are two possible methods to differentiate the shoppers' and shopkeepers' functionalities. First method is to define an "authorize" method that is called right before a certain method is called (using "before_filter") which determines whether the current user has access to the given method or not. Second method is to present different UIs to the users depending on the status.

1) **Authorize method**: This would make code look cleaner, since all the logic is in the controller, and a view would be rendered based on the result of the authorization.

However, from front-end perspective, it is not user-friendly. The view would still show links and functions that the user would not be able to use (for example, "New Items" would still show for the shopper, even if he cannot create an item). It would be a source of frustration for users.

2) **Different UI**: Presenting different UI is definitely user-friendly in a sense that the user would only be able to see relevant links and functions. However, this would require some logic determination in the view files, making the codes unclean.

I decided to go with the second option; the site should be easy to use and cause less confusion to the users.