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.

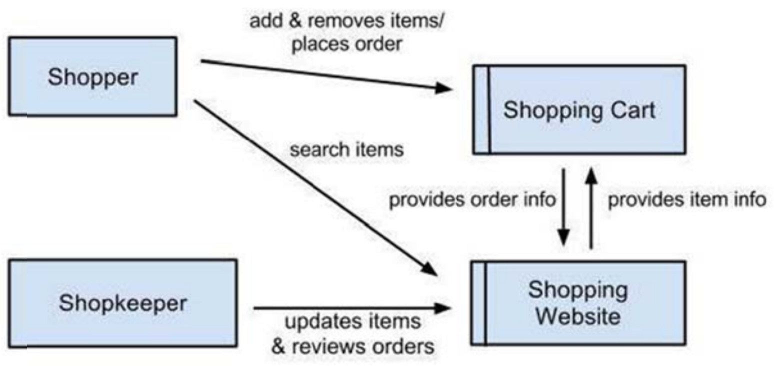


Fig 1. Context Diagram for Shopping cart application. A regular box refers to components external to this application.

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, List, Item, and OrderItem. A user logs in as a shopper, shopkeeper, or administrator. The shopper is given a shopping cart to where he/she can add and remove items. Shopper can decide to place an order, which can be viewed later on by both the specific shopper and shopkeepers. The shopkeeper can modify existing items, or add new ones. Carts form many-to-many relationships with items, since carts can have many items and items can appear in several carts. Same logic applies to the lists. OrderItem objects are created when a corresponding order object is created to keep the immutable record.

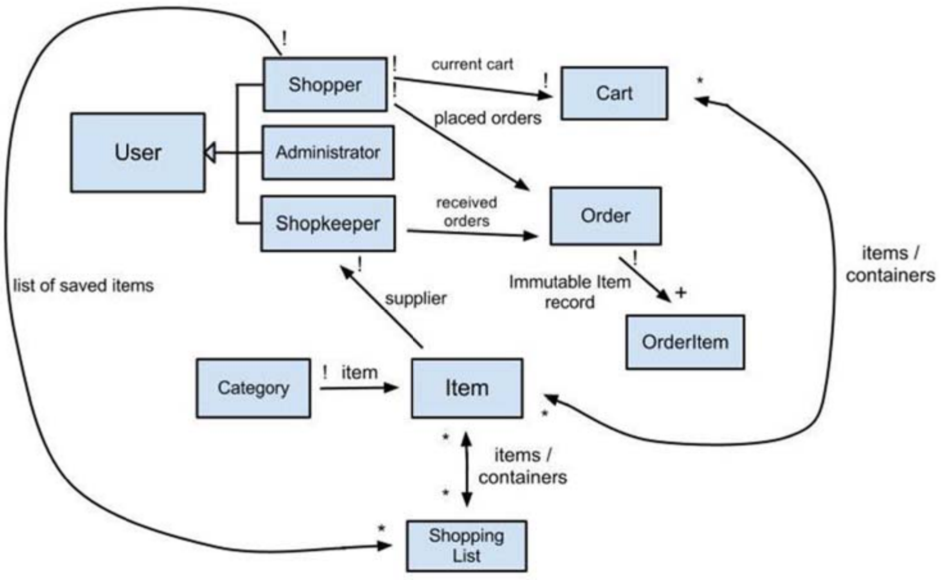


Fig 2. Object Model diagram of Shopping Cart Application

3. Behavior

3.1 Features

Administrator

- **Add & Remove & Update Categories**

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.
- **Lists**: Customers can add items or import the whole cart to a list for later purchase. Such lists, if public, can be shared by other users.

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.

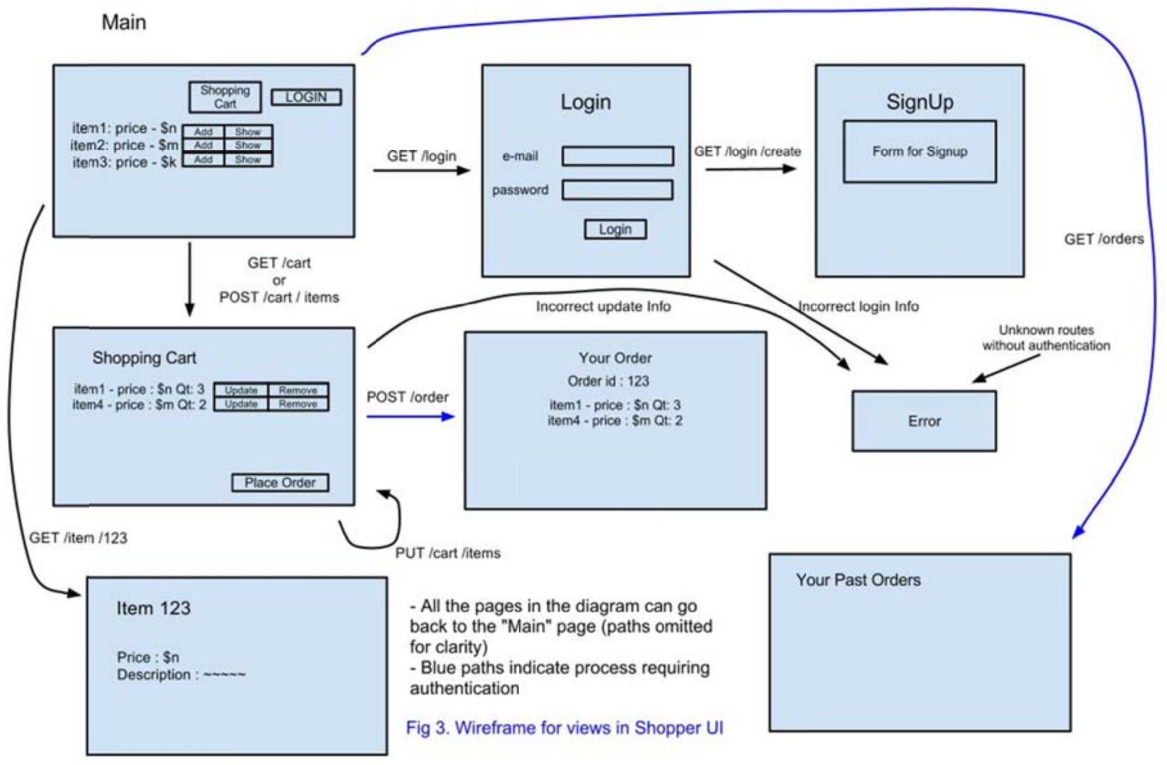
3.2 Security Concerns

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



- All the pages in the diagram can go back to the "Main" page (paths omitted for clarity)
- Blue paths indicate process requiring authentication

Fig 3. Wireframe for views in Shopper UI

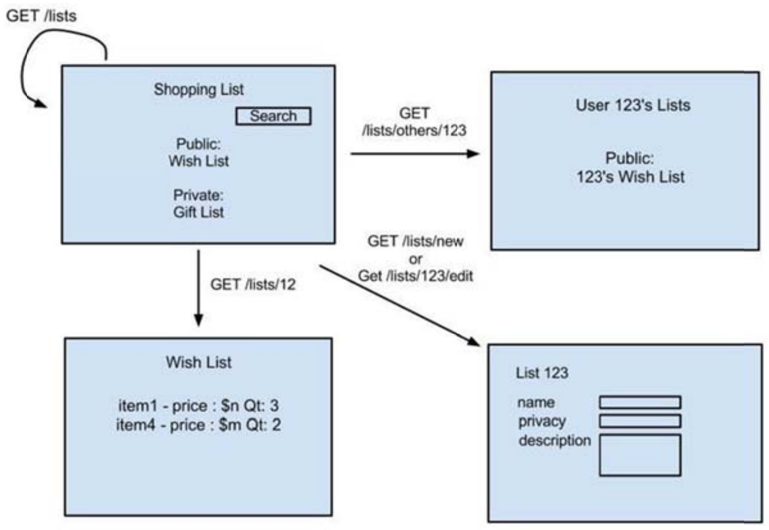


Fig 4. Wireframe for views in Shopper UI

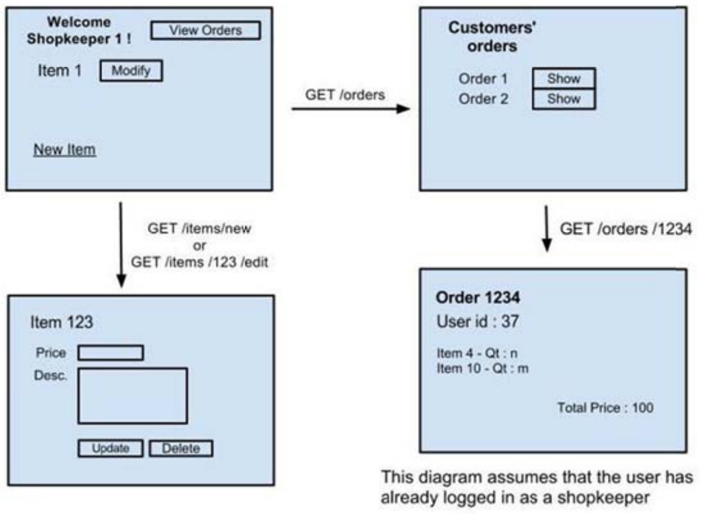


Fig 5. Wireframe for views in Shopkeeper UI

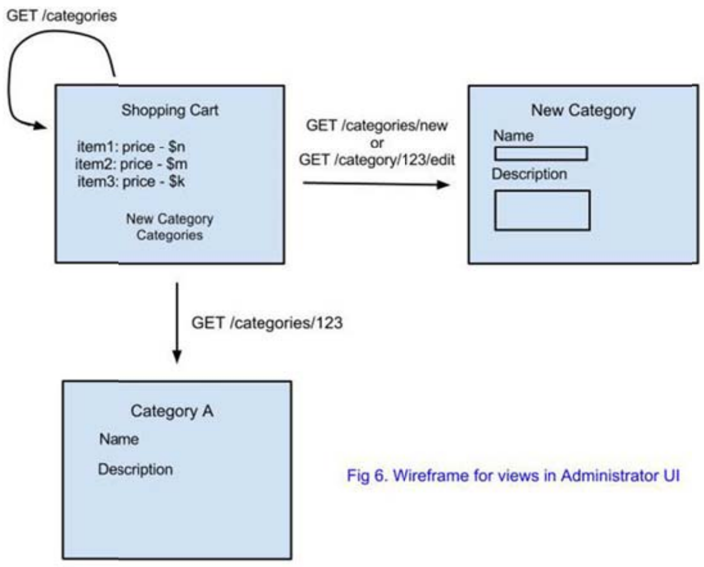


Fig 6. Wireframe for views in Administrator UI

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. Session storage is cookie-store, the whole hash gets recorded in a cookie, increasing the vulnerability to security threats, and sudden internet connection failure. Moreover, cookies are limited in size, limiting possible large-volume orders.

2) Create a model "CartItem" 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. 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. Storing data in server-side database seemed to provide more security

- **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.

- **Order & OrderItem model**

One of the biggest code challenges was to implement the 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.

My initial shot was to use serialization through yaml. Although the method had its merits of being able to serialize and de-serialize with ease and being hard to modify, it was not really developer-friendly; Being in a string format, yaml-serialized form did not have object-specific features, making it hard to manipulate should the schema of underlying model (in this case Item model) change.

The current solution is to create an immutable model called "OrderItem," that belongs to "Order" object. This object contains all the information of the specific item in the order such as quantity, price, and name. These attributes are set through constructor at the creation of the object and become immutable afterwards (through "attr_readonly"). This way, immutable records of the placed orders are formed, safe from external influences.

- **Administrator**

In the final version of the project, I introduced new user status "administrator." The idea was brought up when introducing concept of "Category." Whereas items, carts, and orders are managed by shoppers and shopkeepers, category model is a central logic to the website that neither the shopper nor shopkeeper should be able to temper with. If shopkeepers are able to modify and add categories, the result would be disastrous.

One of the solutions was to directly manipulate sqlite database; however, such proved inefficient and inflexible, in that the modification of the database records are not easy without some knowledge of sql. If this application is actually distributed for others to use, the owner should be able to maintain the application without knowing complexities of sql database statements.

The solution was to introduce administrator; administrator assumes responsibility of maintaining the website and taking care of central logics. Although he is only given the role of maintaining "categories" at this point, further extensions of the application would definitely benefit from the role of "administrator." Moreover, in the actual world application, there should be security measures set up to prevent any users signing up as administrators.

- **Authorization for Users**

There are two parts to making the authorization of users successful: user-status dependent UI and security-check method. Although seemingly mutually exclusive, they need to be combined together in order to fully differentiate the users.

Dependent UI Depending on whether the user is shopper, shopkeeper, or administrator, different UIs should be presented. It would not be user-friendly at all to display links and functions to all kinds of users, when those features are only relevant to specific users. The additional benefit of using dependent UI would be some degree of authorization. For example, since a shopper would not be able to see “create new Items,” or “modify items” links, shoppers would not be able to assume shopkeepers’ roles.

Security-Check Method Even through dependent UI, there certainly exist security threats since a malicious user could just type in the URL to gain access to the functionalities that he should not have accessibility to. Security-check method prevents such threats. Three authorization methods (one for each status) are placed in application controller, and called through “before_filter,” prior to actual trigger of a controller’s action. Only if the authorization succeeds, the user would be able to trigger the controller’s actions. This security-check ensures the failure of malicious users trying to breach security through URL.