PROJECT Design Documentation

> *The following template provides the headings for your Design Documentation. As you edit each section make sure you remove these commentary 'blockquotes'; the lines that start with a > character and appear in the generated PDF in italics.*

Team Information

- Team name: 4d-footware
- Team members
  - Matthew Zhang (mtz5784)
  - Mathew Owusu Jr (mko9824)
  - Glenn Vodra (gkv4063)
  - Laxmi Poudel (lp2439)
  - Nishant Kharel (nk1359)

Executive Summary

We wanted to provide shoes that can be bought electronically.

Purpose

> *Provide a very brief statement about the project and the most important user group and user goals.*

The project is designed where customers are able to create accounts to buy shoes off the website while the admin is able to add new shoes and customize pre existing shoes

Glossary and Acronyms

Term	Definition
SPA	Single Page
MVP	Minimum Viable Product
eStore	Store that accepts electronic transactions
DAO	Data Access Object

Requirements

This section describes the features of the application.

> *In this section you do not need to be exhaustive and list every story. Focus on top-level features from the Vision document and maybe Epics and critical Stories.*

Definition of MVP

> *Provide a simple description of the Minimum Viable Product.*

Our goal is to create a shoe estore where an Owner (admin user) can manage a shoe inventory, and customers can create accounts and interact with the shoes in the inventory by saving them to their carts and checking out items

MVP Features

> *Provide a list of top-level Epics and/or Stories of the MVP.*

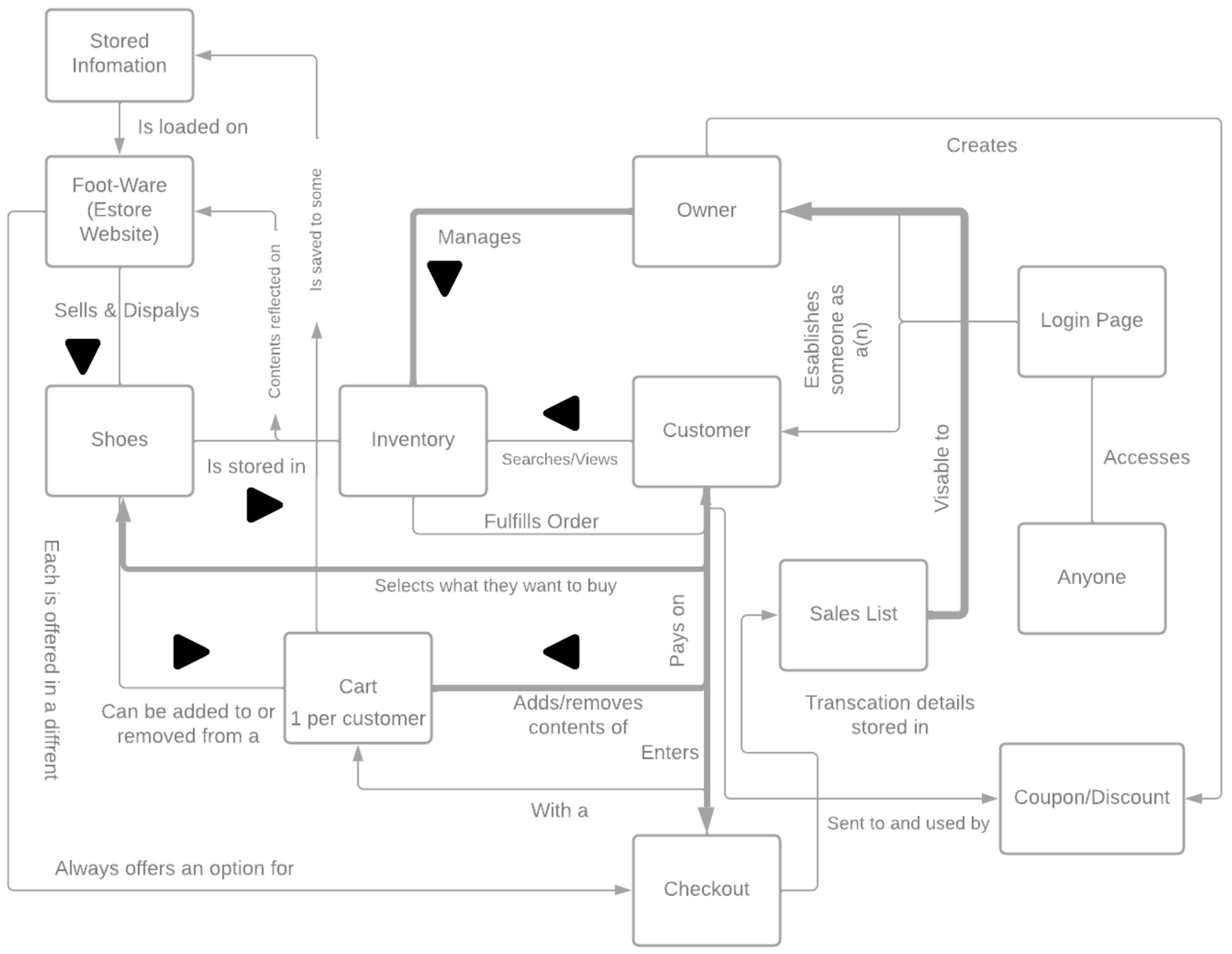
- Minimal Authentication for customer/e-store owner login & logout
  - The server will (admittedly insecurely) trust the browser of who the user is. A simple username to login is all that is minimally required. Assume a user logging in as admin is the e-store owner.
  - You are not expected to do full credential and session management, although the system will look different depending on who is logged in. Obviously this isn't how things are done in real life, but let's sidestep that complexity for this class.
- Customer functionality
  - Customer can see list of products
  - Customer can search for a product
  - Customer can add/remove an item to their shopping cart
  - Customer can proceed to check out their items for purchase
- Inventory Management
  - E-Store owners can add, remove and edit the data on the inventory
- Data Persistence
  - Our system saves everything to files such that the next user will see a change in the inventory based on the previous user's actions. So if a customer had something in their shopping cart and logged out, they should see the same items in their cart when they log back in.
- Your 10% feature
  - We plan to add feature when the owner can implement coupons and discounts and the customer can use them at their pleasure

Roadmap of Enhancements

> *Provide a list of top-level features in the order you plan to consider them.*

Application Domain

This section describes the application domain.



Domain Analysis For the Foot-Ware Estore

> *Provide a high-level overview of the domain for this application. You can discuss the more important domain entities and their relationship to each other.*

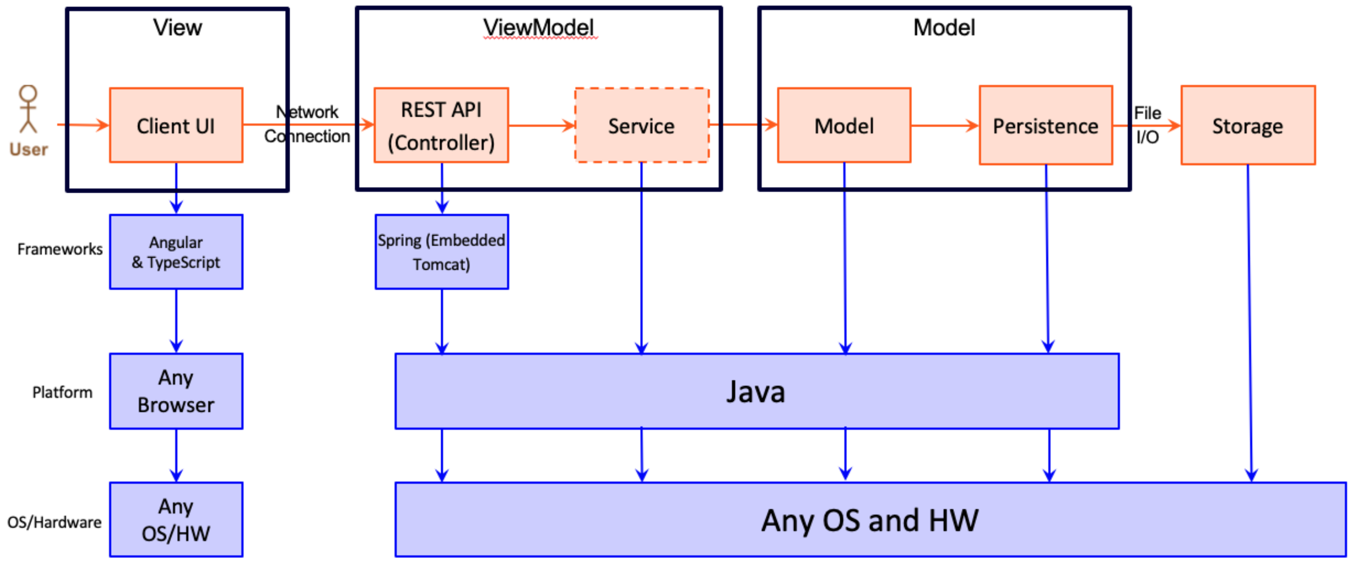
Owners can add new shoes/modify shoes on the Shoe Warehouse. Customers are able to login/signup with accounts to add shoes onto their cart for ordering. Customers can also apply coupon/discounts at checkout while the Owner can view a sales list of what a customer bought. Shoes bought can be custom types such as Athletic or Formal shoes.

Architecture and Design

This section describes the application architecture.

Summary

The following Tiers/Layers model shows a high-level view of the webapp's architecture.



The e-store web application, is built using the Model–View–ViewModel (MVVM) architecture pattern.

The Model stores the application data objects including any functionality to provide persistance.

The View is the client-side SPA built with Angular utilizing HTML, CSS and TypeScript. The ViewModel provides RESTful APIs to the client (View) as well as any logic required to manipulate the data objects from the Model.

Both the ViewModel and Model are built using Java and Spring Framework. Details of the components within these tiers are supplied below.

Overview of User Interface

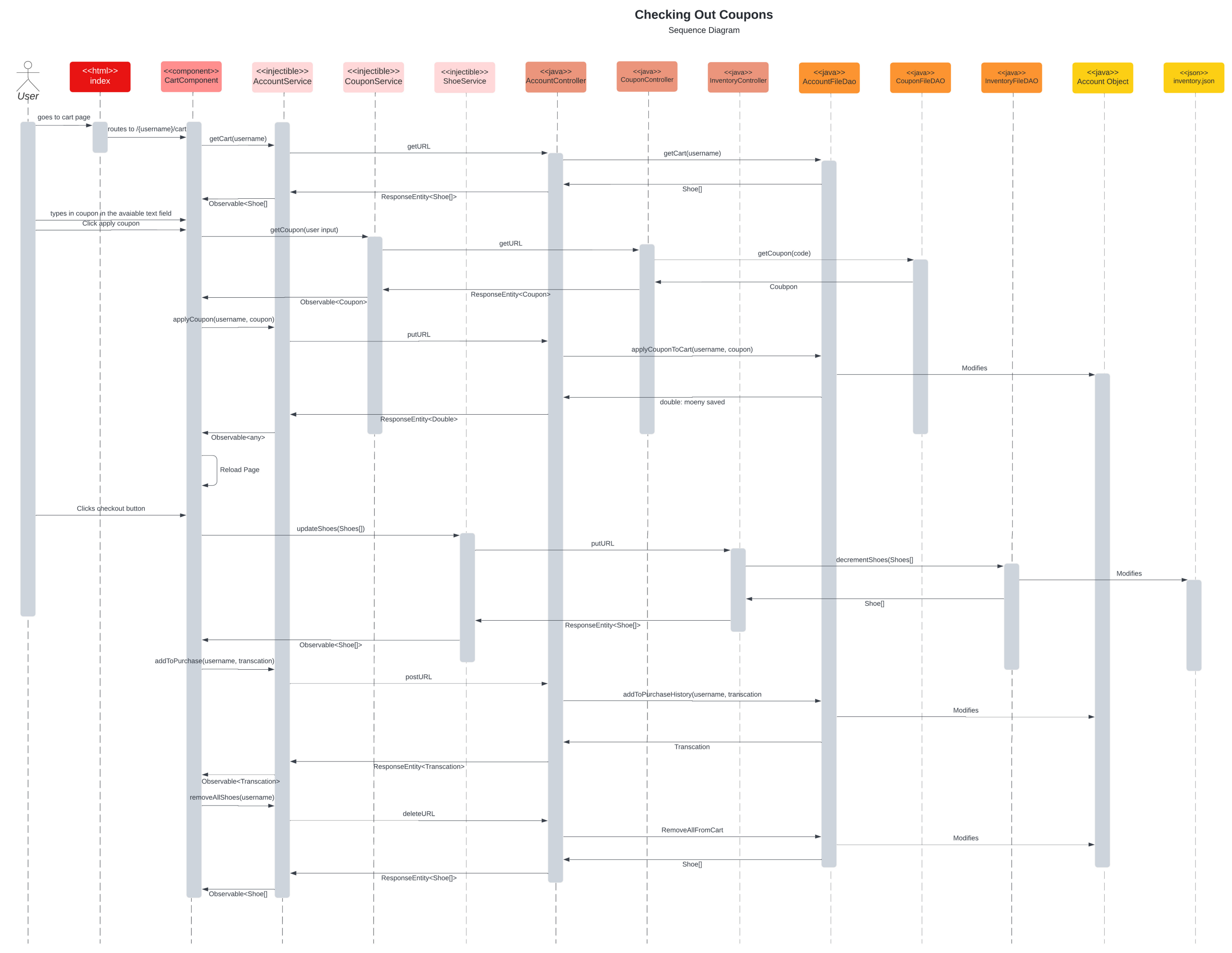
This section describes the web interface flow; this is how the user views and interacts with the e-store application.

> *Provide a summary of the application's user interface. Describe, from the user's perspective, the flow of the pages in the web application.*

View Tier

The view layer is handled by a light amount of HTML and CSS, that give a user's browser instructions on how to format and organize the content when the E-store is accessed. Each component from the inventory, product page, and user profile have their own instructions on how this should be displayed. The user is not responsible for understanding how all of this works, but it should be as responsive to the user as possible. The view layer is directly displayed to the customer, and if there are spelling issues or features that don't scale properly it can make the experience frustrating for the user. By using HTML and CSS, we are creating as many opportunities for us as developers to display the information in a viewable manner.

> *You must also provide sequence diagrams as is relevant to a particular aspects of the design that you are describing. For example, in e-store you might create a sequence diagram of a customer searching for an item and adding to their cart. Be sure to include an relevant HTTP reuqests from the client-side to the server-side to help illustrate the end-to-end flow.*



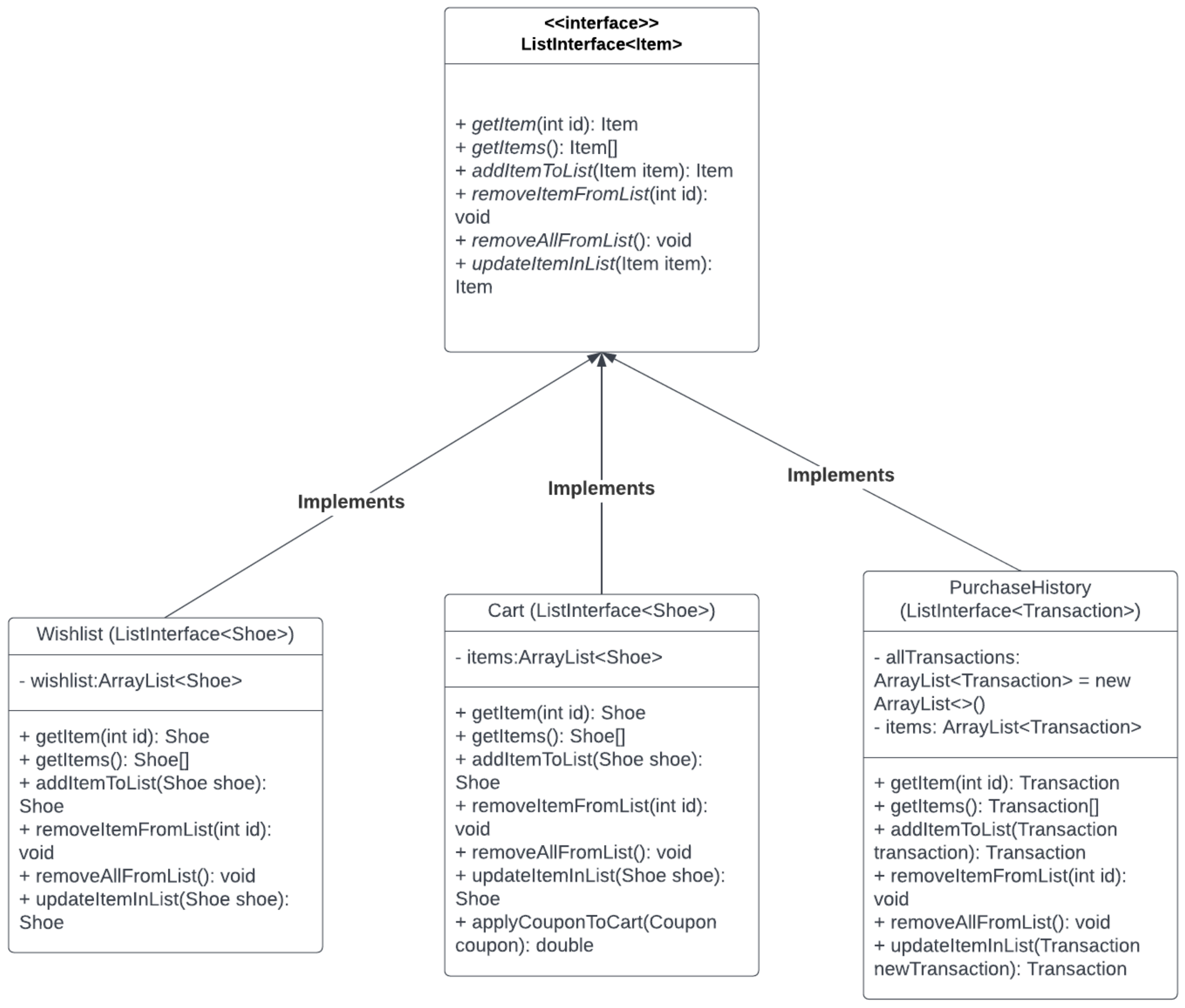
ViewModel Tier

This model layer is handled by the Front-end Angular application. This layer outlines the framework for how data should be displayed once it is fetched from the controller. Angular includes a structure built around organizing features into components and services. This makes it very clear what possible pages/views will be accessible on the site. Some of the components that were created were a shoe component for displaying one singular shoe, and an inventory component for displaying all available shoes. These are independent of each other and handle the data they receive. There are also separate TypeScript files for routing, that allow a user to navigate between the different components. This is accessible with some clickable text that is displayed on every page of the site.

> static models (UML class diagrams) with some details such as critical attributes and methods._

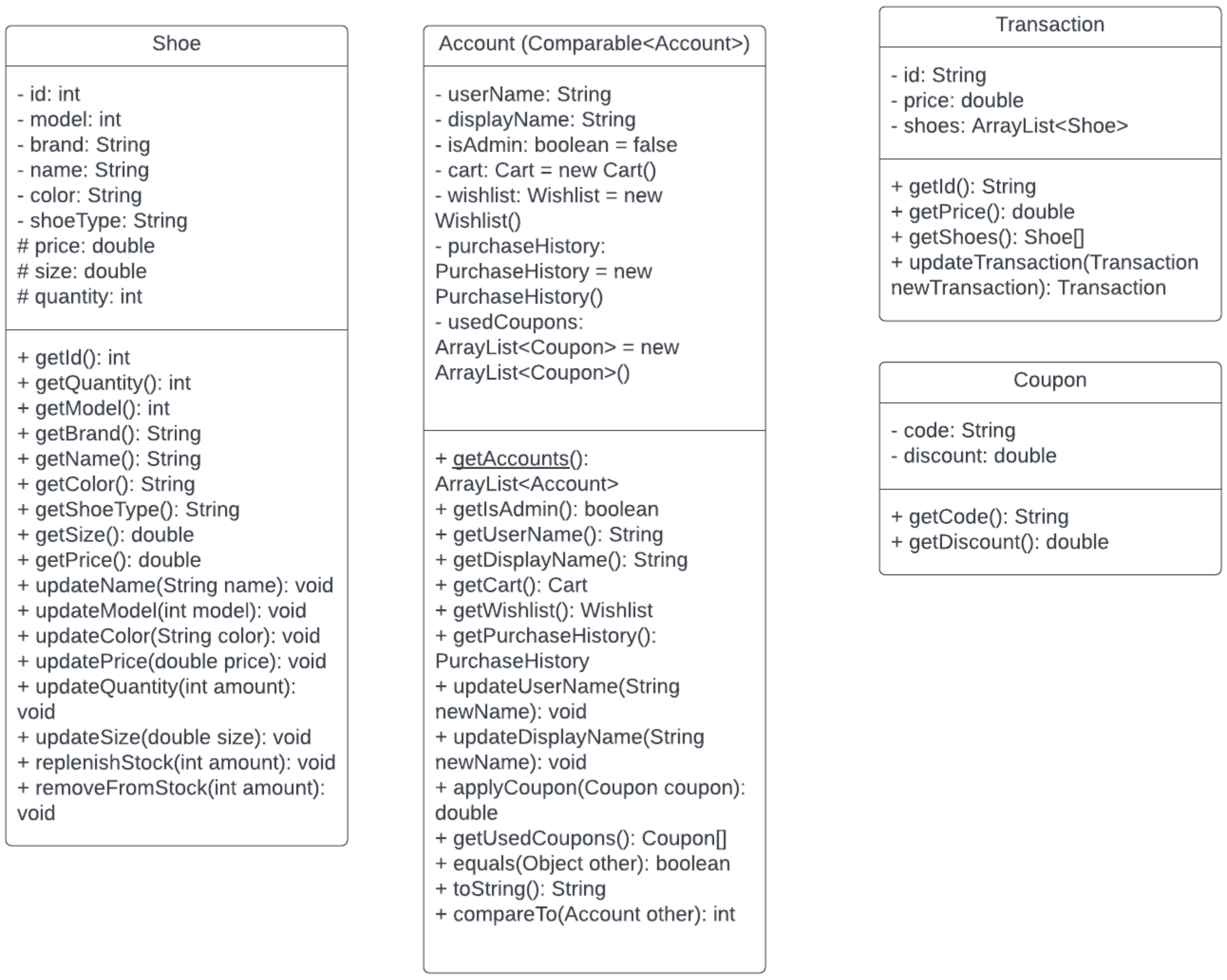
Model Tier

The following image shows the UML diagram for the ListInterface which is an interface that is used by 3 other classes (Wishlist, Cart, and PurchaseHistory). They all share same methods used to modify data in a list by using generics to define the type of data that is stored in the list.



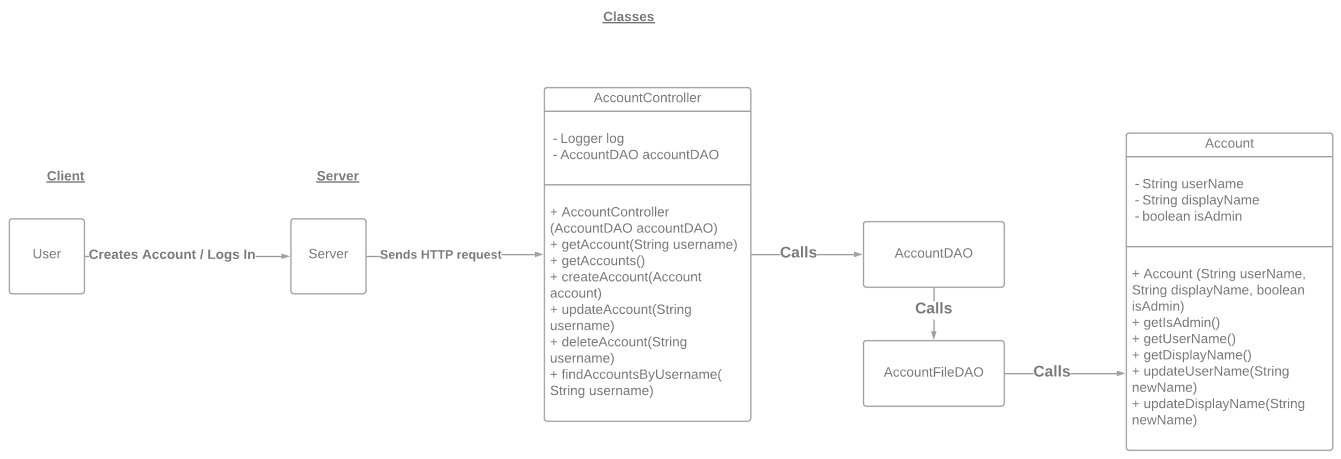
<<interface>>
ListInterface<Item>
+ getItem(int id): Item
+ getItems(): Item[]
+ addItemToList(Item item): Item
+ removeItemFromList(int id): void
+ removeAllFromList(): void
+ updateItemInList(Item item): Item
Implements
Implements
Implements
Wishlist (ListInterface<Shoe>)
- wishlist:ArrayList<Shoe>
+ getItem(int id): Shoe
+ getItems(): Shoe[]
+ addItemToList(Shoe shoe): Shoe
+ removeItemFromList(int id): void
+ removeAllFromList(): void
+ updateItemInList(Shoe shoe): Shoe
Cart (ListInterface<Shoe>)
- items:ArrayList<Shoe>
+ getItem(int id): Shoe
+ getItems(): Shoe[]
+ addItemToList(Shoe shoe): Shoe
+ removeItemFromList(int id): void
+ removeAllFromList(): void
+ updateItemInList(Shoe shoe): Shoe
+ applyCouponToCart(Coupon coupon): double
PurchaseHistory (ListInterface<Transaction>)
- allTransactions: ArrayList<Transaction> = new ArrayList<>()
- items: ArrayList<Transaction>
+ getItem(int id): Transaction
+ getItems(): Transaction[]
+ addItemToList(Transaction transaction): Transaction
+ removeItemFromList(int id): void
+ removeAllFromList(): void
+ updateItemInList(Transaction newTransaction): Transaction

These are the individual classes which are used to build the main components of this estore. The Shoe class is the one that manages how shoes are made and stored. Account class is used to store all data including wishlist, purchase history and etc. Coupon class is a 10% feature used during checkout. Transaction is a class created to store details of a purchase after checkout.


Shoe
- id: int
- model: int
- brand: String
- name: String
- color: String
- shoeType: String
# price: double
# size: double
# quantity: int
+ getId(): int
+ getQuantity(): int
+ getModel(): int
+ getBrand(): String
+ getName(): String
+ getColor(): String
+ getShoeType(): String
+ getSize(): double
+ getPrice(): double
+ updateName(String name): void
+ updateModel(int model): void
+ updateColor(String color): void
+ updatePrice(double price): void
+ updateQuantity(int amount): void
+ updateSize(double size): void
+ replenishStock(int amount): void
+ removeFromStock(int amount): void
Account (Comparable<Account>)
- userName: String
- displayName: String
- isAdmin: boolean = false
- cart: Cart = new Cart()
- wishlist: Wishlist = new Wishlist()
- purchaseHistory: PurchaseHistory = new PurchaseHistory()
- usedCoupons: ArrayList<Coupon> = new ArrayList<Coupon>()
+ getAccounts(): ArrayList<Account>
+ getIsAdmin(): boolean
+ getUserName(): String
+ getDisplayName(): String
+ getCart(): Cart
+ getWishlist(): Wishlist
+ getPurchaseHistory(): PurchaseHistory
+ updateUserName(String newName): void
+ updateDisplayName(String newName): void
+ applyCoupon(Coupon coupon): double
+ getUsedCoupons(): Coupon[]
+ equals(Object other): boolean
+ toString(): String
+ compareTo(Account other): int
Transaction
- id: String
- price: double
- shoes: ArrayList<Shoe>
+ getId(): String
+ getPrice(): double
+ getShoes(): Shoe[]
+ updateTransaction(Transaction newTransaction): Transaction
Coupon
- code: String
- discount: double
+ getCode(): String
+ getDiscount(): double

Controller Tier

The back-end Rest API deals with the direct storage of data in a local JSON file and makes up the controller layer. It handles how data is processed and is there to ensure the data is accessed in an intended manner. Using the Rest API allows resources to be added, removed, and edited on a server using HTTP requests. The controller is broken into two layers that further ensure organization and control. Each type of data that is stored on the E-store has its own DAO that acts as a standard Java object and handles persistence. This handles functionality such as adding numbers or accessing list-objects. Besides having a strict interface each DAO also has its own controller that handles HTTP requests like GET, POST etc, and provides mappings to the DAO so that they can be serialized and deserialized. Some examples include deleting a shoe from the inventory or adding a shoe to a cart. There are also Shoe and Account models that provide a clearly defined structure for creating objects of that type.

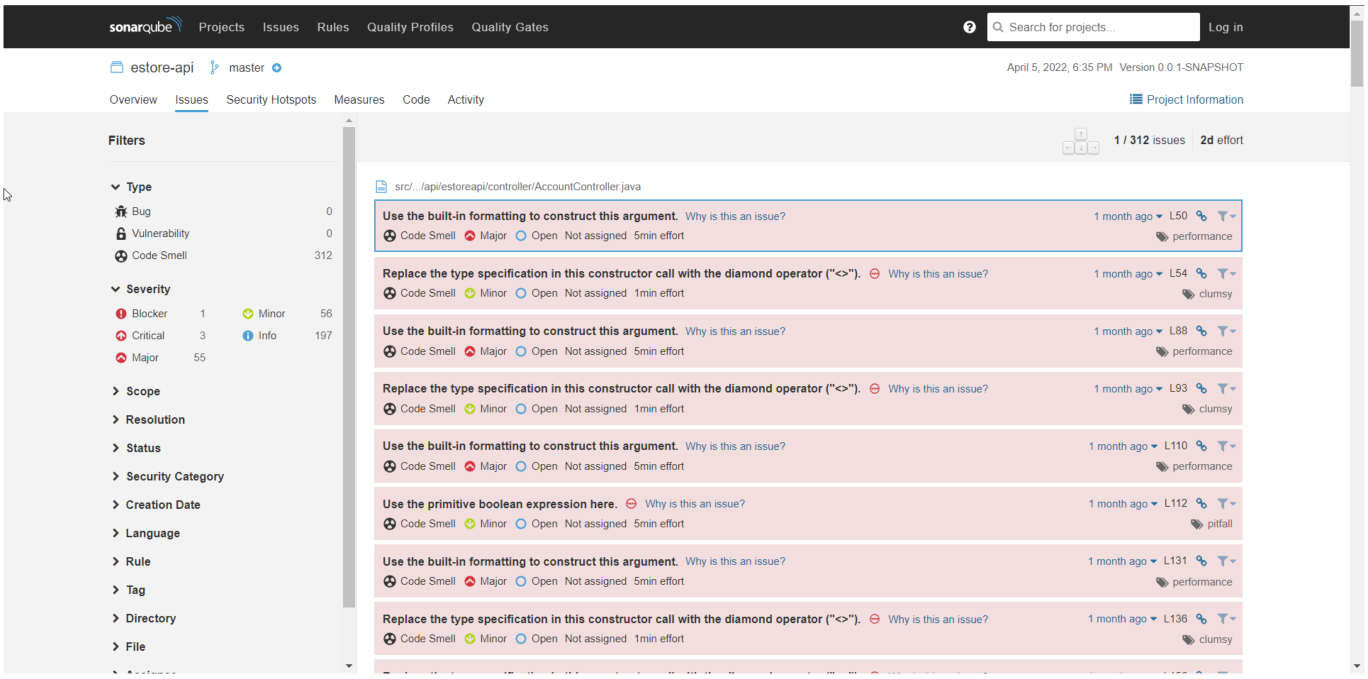

Classes
Client
Server
User
Creates Account / Logs In
Server
Sends HTTP request
AccountController
- Logger log
- AccountDAO accountDAO
+ AccountController (AccountDAO accountDAO)
+ getAccount(String username)
+ getAccounts()
+ createAccount(Account account)
+ updateAccount(String username)
+ deleteAccount(String username)
+ findAccountsByUsername(String username)
Calls
AccountDAO
Calls
AccountFileDAO
Calls
Account
- String userName
- String displayName
- boolean isAdmin
+ Account (String userName, String displayName, boolean isAdmin)
+ getIsAdmin()
+ getUserName()
+ getDisplayName()
+ updateUserName(String newName)
+ updateDisplayName(String newName)

This account model utilizes 3 variables being the userName, displayName, and isAdmin with the standard getters and setters to be used for login page. When updating an account it fetches the existing account in the list and calls those methods to update the account object.

Static Code Analysis/Design Improvements

Some design improvements for our codebase would be make all list type classes share the same shoe object instead of a new copy with the same details. This would allow us to easily update shoe details in both cart and wishlist. Currently both get updated by getting the account's cart and wishlist then individually calling their methods respective to updating a shoe in a list for the backend. Another would be coupon, some of the design implementation of this was done wrong such as checking whether a coupon has been used before or not.

Some issues noticed afer running the analysis was that we had messy code in an unorganized structure. Refactoring some of the code structure would help improve readability for the group and allow for easier modification of the codebase in the future.



estore-api master

April 5, 2022, 6:35 PM Version 0.0.1-SNAPSHOT

Filters

Type
- Bug 0
- Vulnerability 0
- Code Smell 312

Severity
- Blocker 1
- Minor 56
- Critical 3
- Info 197
- Major 55

Scope, Resolution, Status, Security Category, Creation Date, Language, Rule, Tag, Directory, File

1 / 312 issues 2d effort

src/.../api/estoreapi/controller/AccountController.java

Use the built-in formatting to construct this argument. Why is this an issue? 1 month ago L50
Code Smell Major Open Not assigned 5min effort performance

Replace the type specification in this constructor call with the diamond operator ("<>"). Why is this an issue? 1 month ago L54
Code Smell Minor Open Not assigned 1min effort clumsy

Use the built-in formatting to construct this argument. Why is this an issue? 1 month ago L88
Code Smell Major Open Not assigned 5min effort performance

Replace the type specification in this constructor call with the diamond operator ("<>"). Why is this an issue? 1 month ago L93
Code Smell Minor Open Not assigned 1min effort clumsy

Use the built-in formatting to construct this argument. Why is this an issue? 1 month ago L110
Code Smell Major Open Not assigned 5min effort performance

Use the primitive boolean expression here. Why is this an issue? 1 month ago L112
Code Smell Minor Open Not assigned 5min effort pitfall

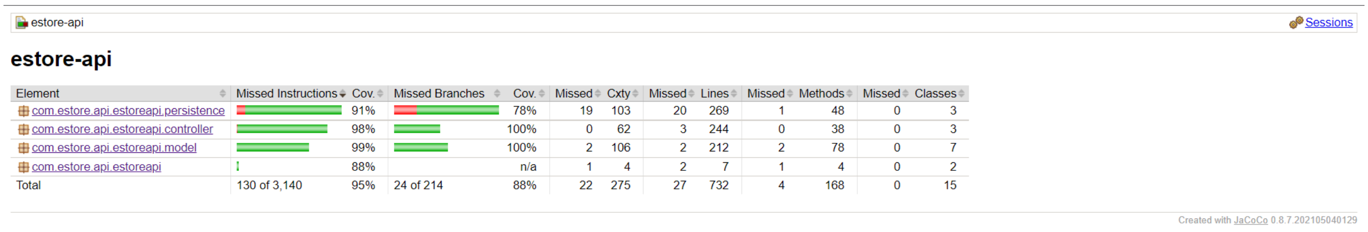
Use the built-in formatting to construct this argument. Why is this an issue? 1 month ago L131
Code Smell Major Open Not assigned 5min effort performance

Replace the type specification in this constructor call with the diamond operator ("<>"). Why is this an issue? 1 month ago L136
Code Smell Minor Open Not assigned 1min effort clumsy

Testing

Acceptance Testing

All user stories have a 95% code coverage shown in the next section. Only some stories related to persistence have below 100% due to missed branches.



estore-api

Sessions

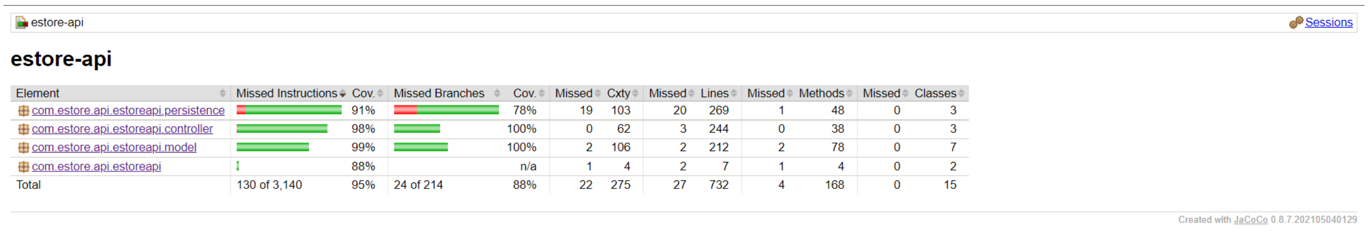
estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.persistence		91%		78%	19	103	20	269	1	48	0	3
com.estore.api.estoreapi.controller		98%		100%	0	62	3	244	0	38	0	3
com.estore.api.estoreapi.model		99%		100%	2	106	2	212	2	78	0	7
com.estore.api.estoreapi		88%		n/a	1	4	2	7	1	4	0	2
Total	130 of 3,140	95%	24 of 214	88%	22	275	27	732	4	168	0	15

Created with JaCoCo 0.8.7.202105040129

Unit Testing and Code Coverage

Our code coverage is 95% with the strategy being to mock classes when necessary and testing all methods to see if they work as intended. We selected 95% as the minimum since most of the methods were simple to test and managed to meet that expectation.



estore-api

Sessions

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.persistence		91%		78%	19	103	20	269	1	48	0	3
com.estore.api.estoreapi.controller		98%		100%	0	62	3	244	0	38	0	3
com.estore.api.estoreapi.model		99%		100%	2	106	2	212	2	78	0	7
com.estore.api.estoreapi		88%		n/a	1	4	2	7	1	4	0	2
Total	130 of 3,140	95%	24 of 214	88%	22	275	27	732	4	168	0	15

Created with JaCoCo 0.8.7.202105040129