PROJECT Design Documentation

Team Information

- Team name: Jadin
- Team members
- Alen Van
- Nicholas Lewandowski
- Donald Burke
- Isaac Post
- Joseph Doros

Executive Summary

This project is to create a working e-store that allows customers to browse, purchase, and rate various types of beans that have been posted by an admin of the store.

Purpose

The purpose of the project is to host a place for consumers and sellers to have a market in order to sell and purchase beans, respectively.

Glossary and Acronyms

Term	Definition
SPA	Single Page Application
HTML	HyperText Markup Language
MVVM	Model–View–ViewModel architecture pattern
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation

Requirements

Definition of MVP

A website configured as an e-store to allow interactions between sellers and buyers. Customers must be able to seek beans that they would like to purchase and the admins sell their goods.

MVP Features

- As a customer I want to be able to view beans on the website in order to buy the product.
- As a customer I want to be able to edit my shopping cart in order to purchase beans.
- As a customer I want to be able to search for beans in order to quickly find the ones I'm looking for.

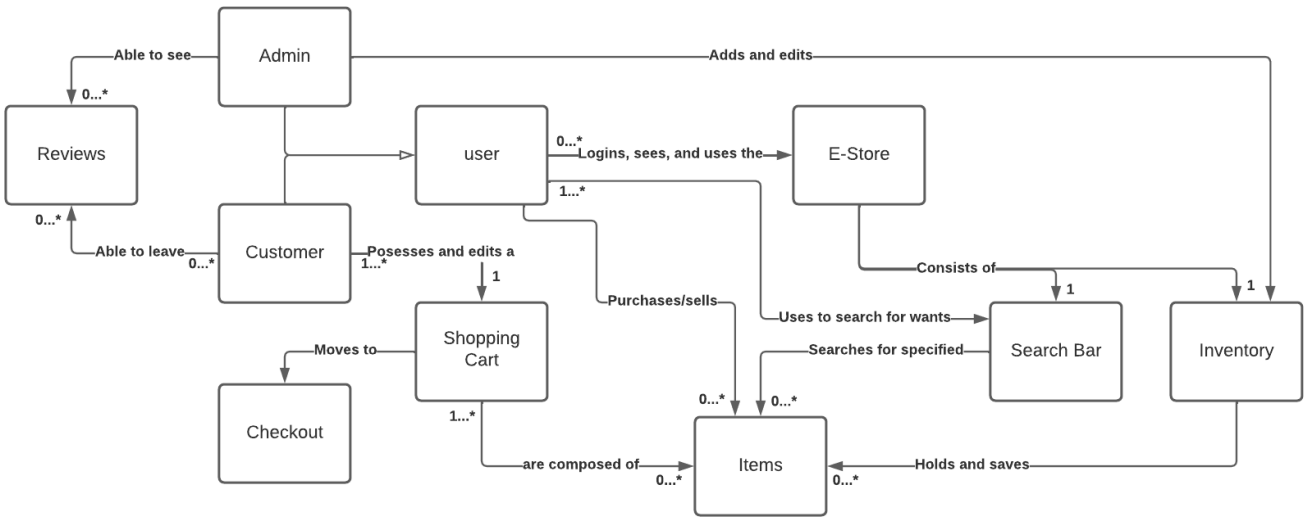
- As a customer I want to be able to leave a rating on a bean in order to share my opinion on its quality of it.
- As an admin I want to be able to edit the inventory to see how I see fit in order to sell my beans.
- As an admin I want to be able to view the e-store in order to interact with it.

Roadmap of Enhancements

Product, Inventory, User, Customer, Shopping Cart, Website, Search Bar, Reviews

Application Domain

This section describes the application domain.



The main components of the application domain are shown as the following: customer, inventory, product, shopping cart, admin.

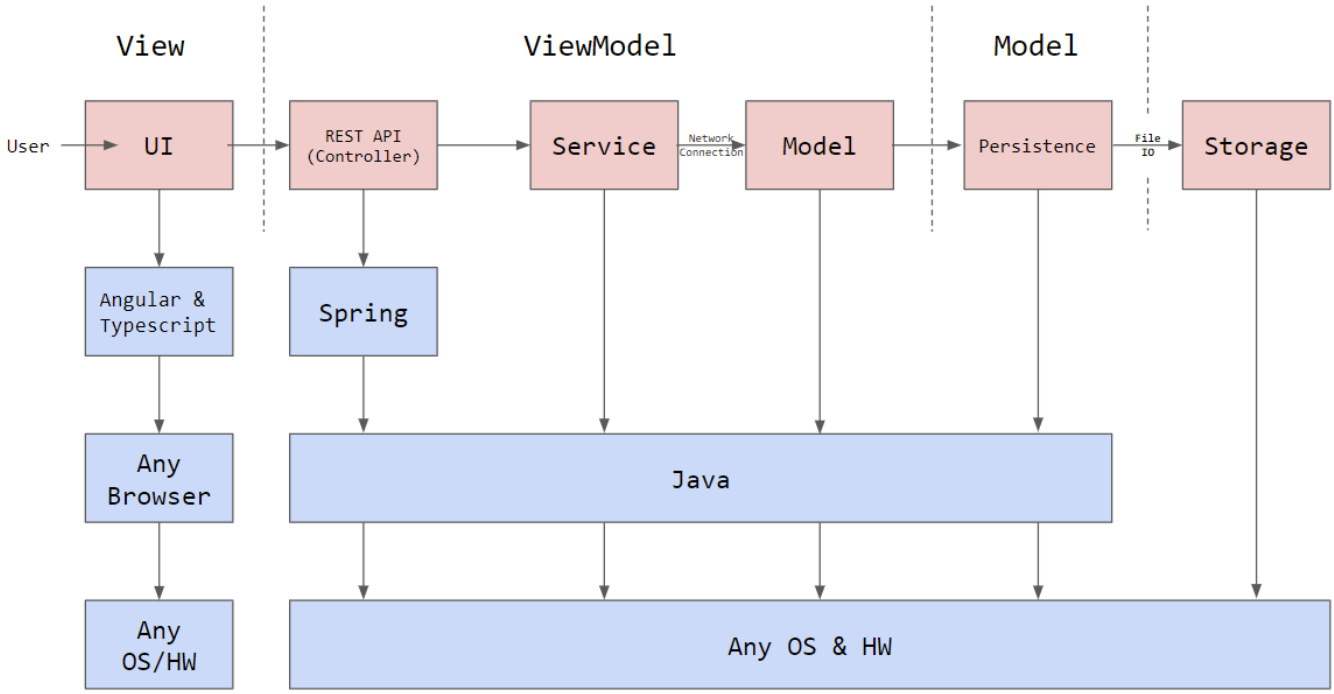
The admin is the overseer of the store, manipulating the inventory to how they see fit, adding or deleting products. Customers view the inventory, select products based on their preferences, and add them to their shopping cart. Once added to the shopping cart, customers are able to checkout, essentially purchasing the product and removing it from the inventory in the process.

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 persistence.

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.

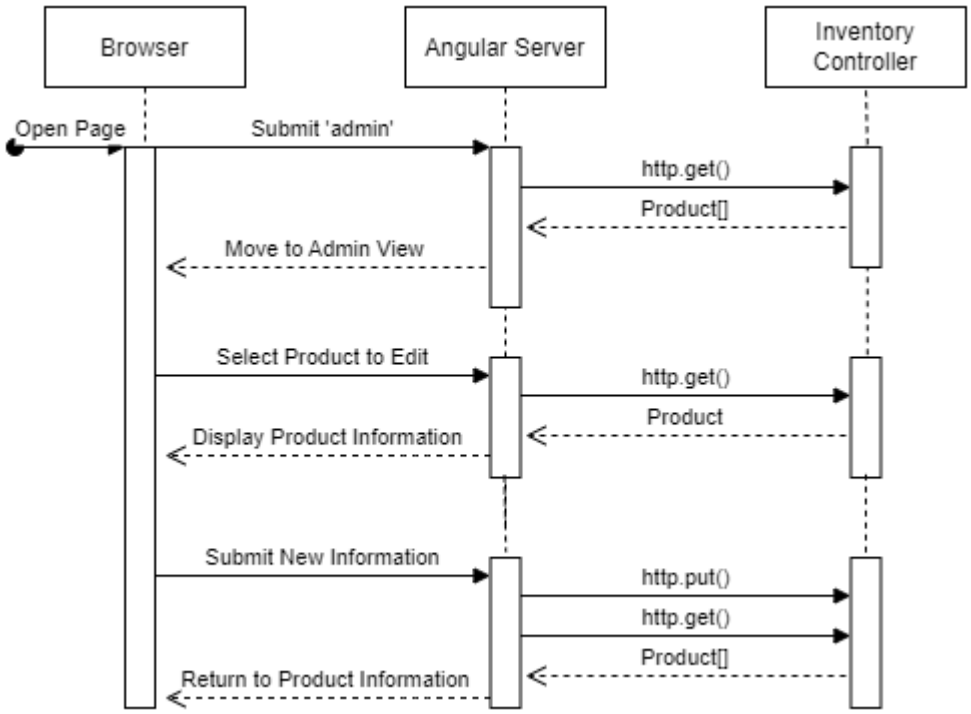
Both the ViewModel and Model are built using Java and Spring Framework. Details of the components within these tiers are supplied below.

Overview of User Interface

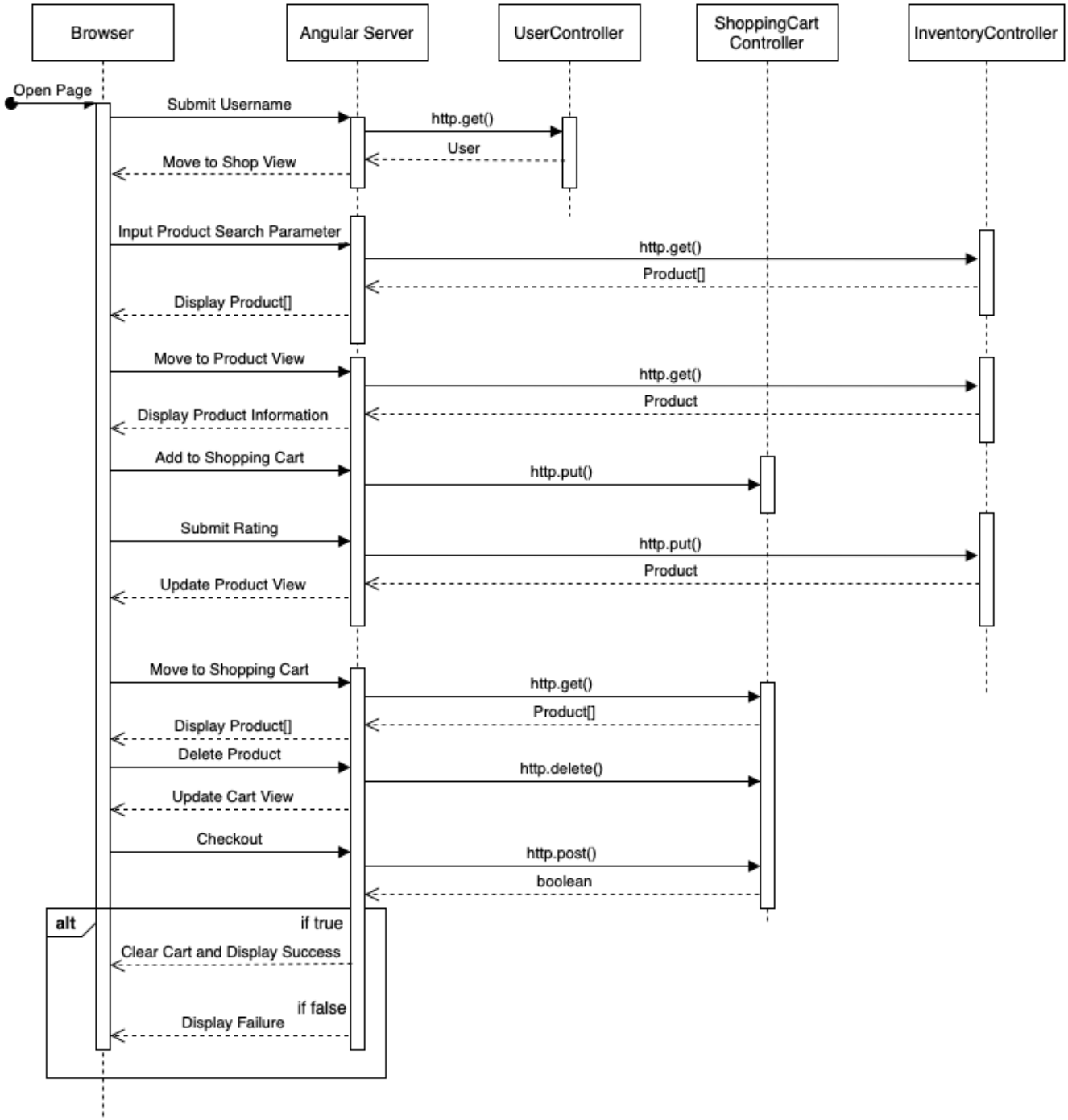
This section describes the web interface flow; this is how the user views and interacts with the e-store application.

The user begins at the home page with only the option to log in to ensure a user account is being cached. After this, if the username is the admin username, the user, who is an owner, will be able to see the inventory as well as add and delete items. Once an item is clicked, the owner will be able to edit the price, stock, and name of the product. If the user logs in as a customer, the interface is changed to the user-storefront where they can see the inventory without some of the rights that the admin has. They are able to click on a product in order to view the product's details as well as add it to their cart. From the user-storefront, the customer can click on the Shopping Cart link to be routed to their current shopping cart. From there they can remove any items they wish.

View Tier

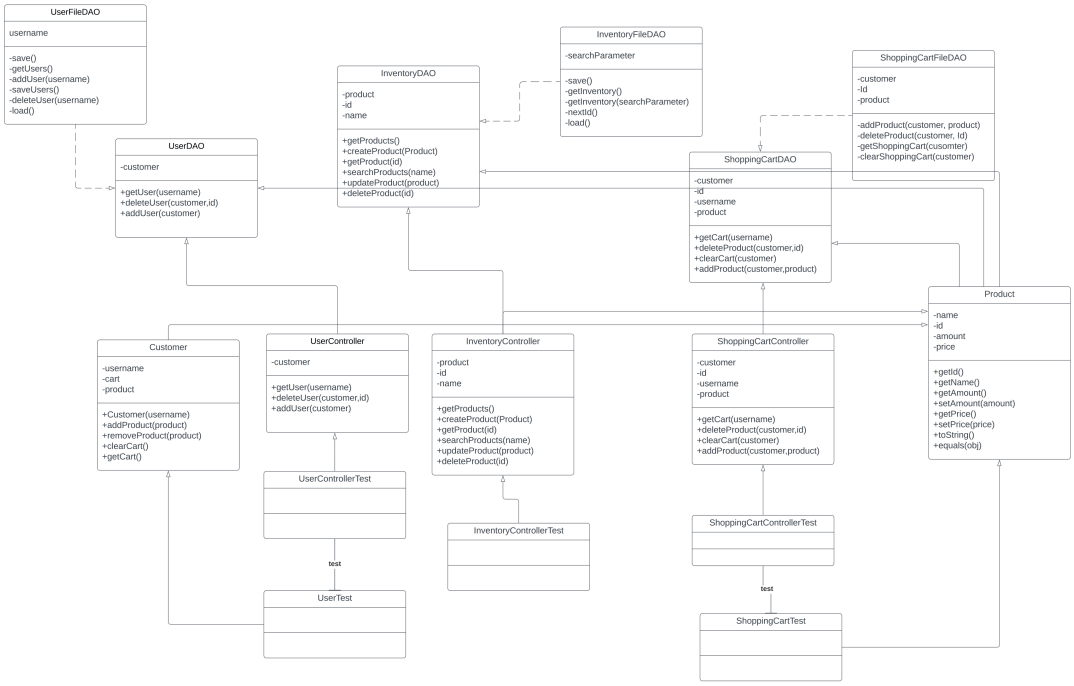


For an admin, in the user-login component, once the 'admin' username is typed in and the log in button is pressed, an HTTP GET request will be sent to the backend to handle. Once a response is received, it will send you to the admin storefront. In the case of a user, that same process will also generate a GET request for the backend to handle, which will add the user to the users.json file if not already added. Once logged in as an admin, the inventory component will work with the product details and product service to use HTTP GET, POST, and DELETE requests to manage the inventory.



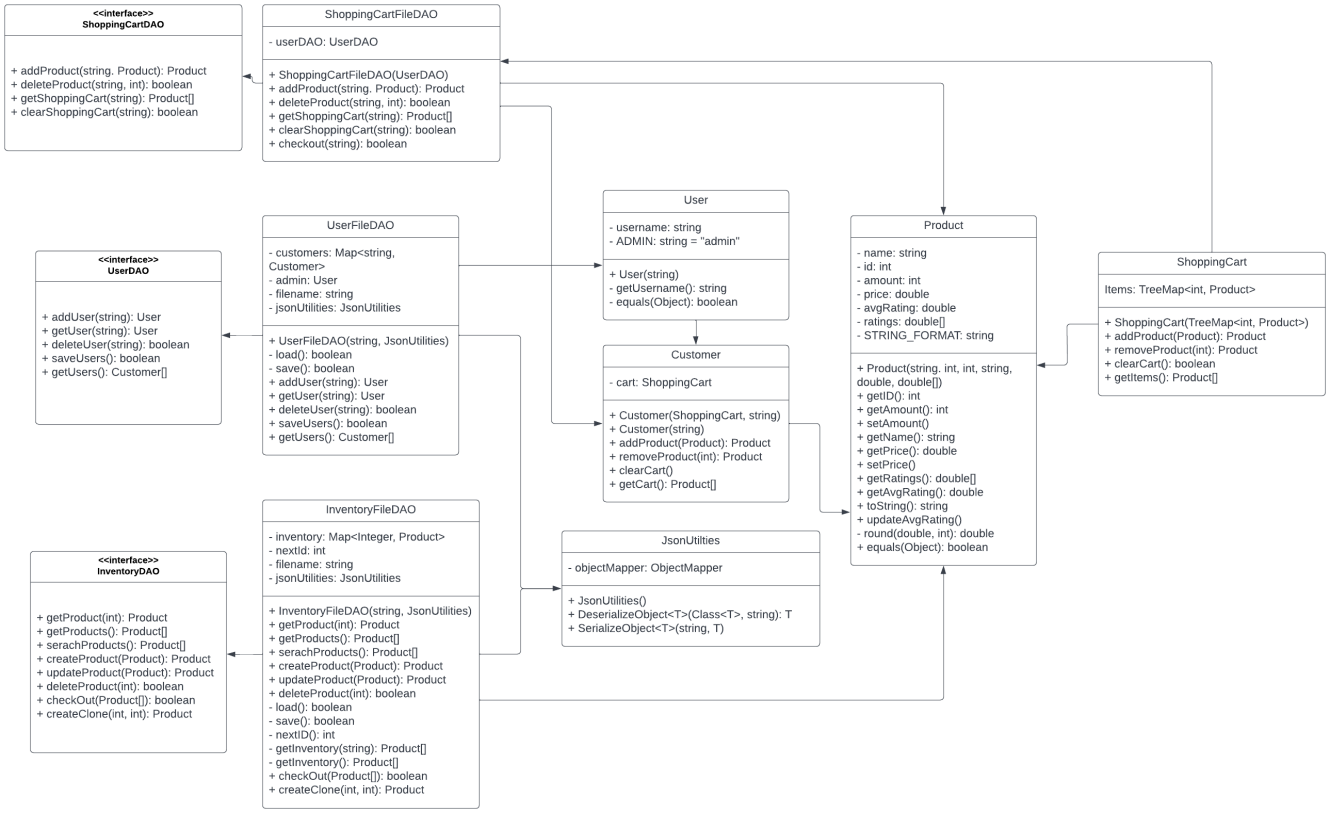
As a user, the user-store, shopping cart, user-product-view, and product-search components all use GET, POST, PUT, UPDATE and DELETE, and utilize the shopping cart and user services to produce the user's view of the storefront and transfer to the product details and shopping cart views.

ViewModel Tier



The ViewModel is the set of classes that take data passed in by the View, which is the front-end web store and send it to the Model, which is the back-end data. In our project, you can expect to find classes called controllers (InventoryController, UserController, and ShoppingCartController) that receive HTTP requests from the View. The controller classes take these requests and figure out what functions from the Model the View is requesting. These classes then call that function and return the data that was requested by the View. In the end, the ViewModel tier is used as a connection between Model and View through the use of the InventoryController, UserController, and ShoppingCartController classes.

Model Tier

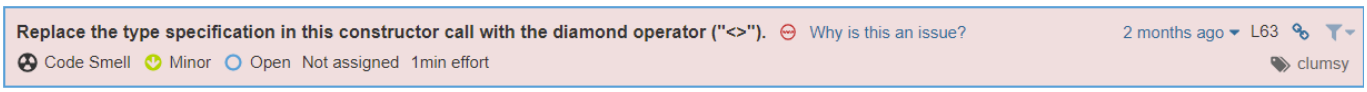


Within the model tier, we can expect to find 2 types of components, the file data access objects and any necessary object classes to help. Some of these necessary objects are objects like the product class and users class. The product class creates objects that hold information on a singular product that is being sold in the e-store, whereas a user holds information about a user who interacts with the e-store. The file data access objects are objects that take care of saving, loading, and manipulation of any sort of data that is to be utilized throughout the entire e-store. Examples of such data can be seen as the inventory or a customer's shopping cart. The inventory must be saved and loaded in order for users to see a constantly updated inventory of products. Shopping carts must be saved so that if a customer logs out, and then logs back in, they should still have their shopping cart and continue to add, delete, and overall edit the shopping cart.

Static Code Analysis/Design Improvements

- Design Improvements
- Upgrading the 10% feature to include written reviews of beans
- A logout button instead of just having a login button
- Enhanced visuals, such as pictures of each bean
- Listing best-selling beans on the homepage of the site
- User SonarQube and SonarScanner reports to improve details in the code

When utilizing the static code analysis offered by SonarQube and SonarScanner, it was clear that there are improvements that can be made regarding bugs and code smell. For the backend (SonarQube), the number of bugs were minor but we had 162 code smells. Most of the code smells were similar to the following:



This was flagged mainly for readability, which doesn't affect performance, but could be improved for the future. On the other hand, the SonarScanner analysis came back with five bugs. Three of the five were the following:

Again, at the level we have worked on this project, this isn't major as it just affects user experience. However, if we were to deploy our project at a higher level, this would definitely be something to improve.

Design Principles Analysis

Design Principles Used:

- Dependency Injection
- Single Responsibility
- Pure Fabrication

- Dependency injection is the act of providing an object, other objects beforehand, its dependencies, rather than having the object construct its dependencies itself. In this project, dependency injection can be seen within the controller and file DAO classes. In the controller classes, the respective DAO class is injected into the controller's constructor and saved for later usage. The file DAO classes, on the other hand, receive a JsonUtilities injection and any other associating DAO objects if needed. By doing this, each class that received an injection no longer needs to construct its dependencies, but rather simply, receive the injection, and save it for later usage.

- Single responsibility, is a design method in which each class pertains to only one role within the program. As an example, in our program, the backend is separated into various tiers, controllers, persistence, and any necessary objects. To conform to this design principle, the controller and persistence tiers are each divided so that each tier does not contain only one class, but rather multiple classes to handle different roles. Some of the different roles are the manipulation of the inventory, the construction and addition of users, and the manipulation of a customer's shopping cart.

- Pure fabrication is a principle that allows for a cleaner project with easier navigation between parts of the project. This is shown physically by creating a class that can not be seen in the domain model but holds a responsibility that is necessary for a part of the domain model. One such application of this design principle can be seen within the InventoryFileDAO and the UserFileDAO classes. Within each of these classes, deserialization and serialization of JSON files are needed, one method of applying this is to directly execute the actions within their respective classes. The method we opted for, however, was to separate this into a different class, a JsonUtilities class that handles the actions. The JsonUtilities class can be seen as the pure fabrication as it is not seen on the domain model, and with this division, we can add more methods to the JsonUtilities class other than the load and save methods that were originally in the DAO classes, for the program to have an enhanced organization.

Testing

Acceptance Testing

Fifteen user stories in total, covering:

- Buyers being able to successfully log into the website
- Buyers being able to add and delete beans from their shopping cart
- Buyers being able to search for beans in the store
- Buyers being able to add a review to a bean
- Admins being able to add, edit, and delete beans from the store
- Persistence of buyer's shopping carts, and the website's inventory

All pass their acceptance criteria tests.

Unit Testing and Code Coverage

Unit Testing Strategy:

Create a single test file for each file that has implementations of methods. Create tests for every possible execution path, for example, in a method that has an if-else, we write two tests, one to test the if block and the other to test the else block. Whenever creating a new instance of an object that has an interface to pair with it, utilize the interface to create the object. Specifically for mock objects. This follows the interface segregation principle in the SOLID principles, this allows the conformity of hiding an unnecessary code that the client does not need to know of.

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.model		87%		80%	6	32	9	67	1	19	0	2
com.estore.api.estoreapi.persistence		42%		n/a	0	3	10	18	0	3	0	1
com.estore.api.estoreapi.persistence.Inventory		94%		80%	6	29	6	73	1	14	0	1
com.estore.api.estoreapi.persistence.User		94%		72%	6	23	4	68	1	14	0	2
com.estore.api.estoreapi.controller		97%		100%	1	33	2	108	1	21	0	3
com.estore.api.estoreapi		88%		n/a	1	4	2	7	1	4	0	2
com.estore.api.estoreapi.model.Users		100%		100%	0	13	0	22	0	9	0	2
Total	126 of 1,599	92%	16 of 106	84%	20	137	33	363	5	84	0	13

Code Coverage: 92% achieved

Discussion:

The team's coverage targets were the controllers, persistence as well as any objects that are utilized between the 2 tiers of classes. The reason for this is that by testing these classes, we can at least determine whether or not an issue that arises in the entirety of the program is either in the front end or back end. With well-created unit tests, we can determine that an error must have happened in the front end or during the connection between the two ends.

The target coverage goal was 90%, as shown above, we were able to not only achieve this but exceed this by 2%.