

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: Scrumblebees



- Team members
 - Ryan Webb

- Kanisha Agrawal
- Noah Landis
- Priyank Patel
- Brianna Vottis

Executive Summary

Introducing Wizbiz (development codename), a captivating web application that transports the wonderful world of magical commerce into the digital age. Wizbiz offers users a spellbinding platform to explore and acquire a wide array of magical items, from wands to brooms, and everything in between.

In its present iteration, Wizbiz allows users to effortlessly create an account, peruse the diverse selection of magical items, and add desired products to their cart. Moreover, the application's owner possesses the ability to enrich the store's inventory by adding new items, as well as editing or deleting existing offerings as needed.

Built using Angular and TypeScript for the front-end, our eStore is easy to navigate and visually appealing, thanks to the Angular Material UI component library. Customers can browse through our collection of magical items effortlessly and enjoy a smooth shopping experience.

The back-end, powered by a Java Spring Boot API, takes care of essential elements like carts, users, and inventory management. This ensures a secure and reliable platform for our customers to shop with confidence.

Purpose

Wizbiz provides a platform for users to search and buy magic products. The most important user group for this website are people who want to buy magical items. The primary user goals for this project are to easily be able to find and purchase magic products online.

Glossary and Acronyms

Term	Definition
SPA	Single Page Application
API	Application Programming Interface
DAO	Data Access Object
SKU	Stock Keeping Unit
UI	User Interface
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
CRUD	Create, Read, Update, Delete
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer

Requirements

This section describes the features of the application. 1)There should be simple authentication system for both the admin and the users. 2)Users should be able to create and account and login/logout from the website. 3)Users should be able to see a list of products in the webiste and also be able to search for the products they need. 4)Users should have full control of the items in cart and their quantities. 5)All the data of the users should be saved to the inventory so that users can view what's in their cart when they login next time.

Definition of MVP

A simple magic shop website that allows users to search, select, add to cart and order magic products that are in stock. The Owner of the website can manage the product by adding,deleting or editing the products displayed in the website.

MVP Features

[Sprint 4] Provide a list of top-level Epics and/or Stories of the MVP.

Enhancements

[Sprint 4] Describe what enhancements you have implemented for the project.

Application Domain

This section describes the application domain.  Domain Model

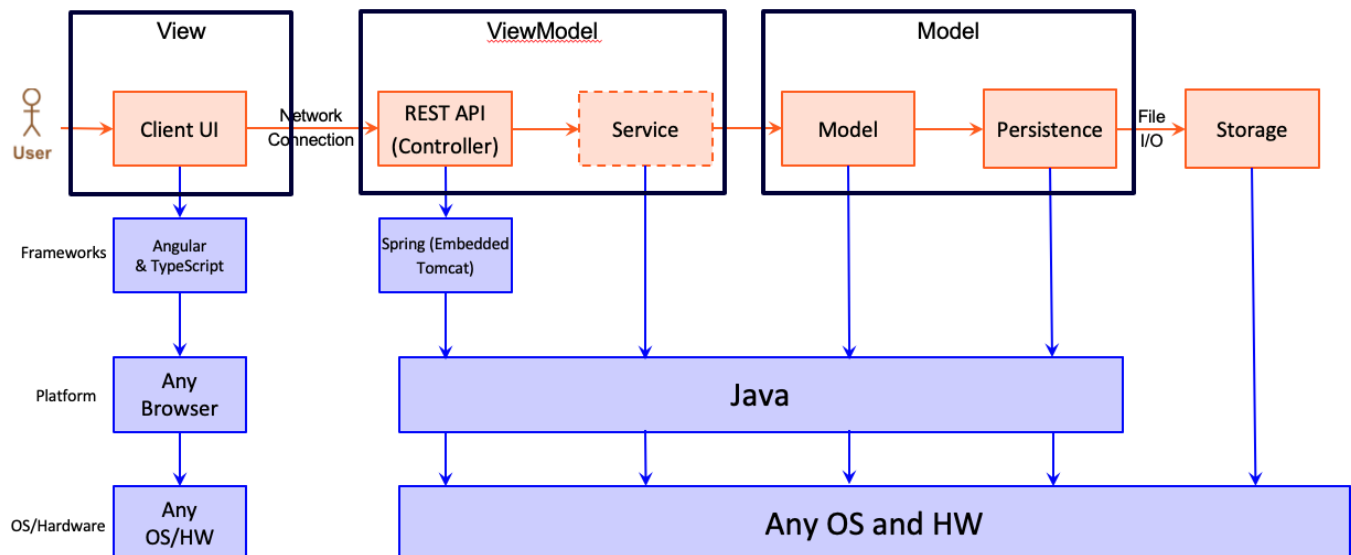
The major relationships in this are admin- dashboard - inventory, which represents the relationship between the admin and how the admin manages the inventory. Another one is customer-admin-user, which shows the heirarchy of users. We added order, wishlist, product review.

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. As soon as a user visits our website, they are greeted with the site's name and an eye-catching grid of products that we offer. To access the full range of features, users can log in or register by clicking an icon located at the top right of the header which provides users with login form.

Once inside the website, users can browse our vast collection of products, which are presented in an intuitive grid format. They can easily add items to their shopping cart by clicking the Add to cart button. They can also adjust the quantity of each item by using the round buttons within the cart listing page. If a user decides they no longer need a particular item, they can simply click on the delete icon to remove it from the cart.

Our search bar is located in the header that helps users find the exact product they're looking for.

View Tier

[Sprint 4] Provide a summary of the View Tier UI of your architecture. Describe the types of components in the tier and describe their responsibilities. This should be a narrative description, i.e. it has a flow or "story line" that the reader can follow. **[Sprint 4]** You must provide at least **2 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. As these can span multiple tiers, be sure to include an relevant HTTP requests from the client-side to the server-side to help illustrate the end-to-end flow. **[Sprint 4]** To adequately show your system, you will need to present the **class diagrams** where relevant in your design. Some additional tips:

- Class diagrams only apply to the **ViewModel** and **Model** Tier

- A single class diagram of the entire system will not be effective. You may start with one, but will be need to break it down into smaller sections to account for requirements of each of the Tier static models below.
- Correct labeling of relationships with proper notation for the relationship type, multiplicities, and navigation information will be important.
- Include other details such as attributes and method signatures that you think are needed to support the level of detail in your discussion.

ViewModel Tier

[Sprint 4] Provide a summary of this tier of your architecture. This section will follow the same instructions that are given for the View Tier above. At appropriate places as part of this narrative provide **one** or more updated and **properly labeled** static models (UML class diagrams) with some details such as critical attributes and methods.

 Replace with your ViewModel Tier class diagram 1, etc.

Model Tier

Our model tier (**M** in **MVC**) is built using Java and Spring Framework. The model tier is responsible for storing the application data objects and providing persistence. The model tier also exposes a set of APIs to the controller tier (**C** in **MVC**) to manipulate the data objects from the Model. The model tier is divided into three resources: **Inventory**, **User** and **Carts**.

The **Inventory** resource is responsible for storing the **Product** data objects and providing persistence via Data Access Object (DAO) classes. The **Product** class is the highest abstraction class used for storing data about a product, such as name, price, SKU (Stock Keeping Unit), images, as well an instance of a **Description** object and a **Stock** object. The **Description** class is used for storing data about a product's description such as the product's summary text, the product's tags. **Description** also encapsulates behavior related to said data. The **Stock** class is used for storing the product's stock quantity, and encapsulates behavior related to it.

Persistence is provided by the **InventoryDAO** interface, and its concrete implementation **InventoryFileDAO**. The **ProductFileDAO** class is the layer between the product data objects and the controller tier, and provides methods for saving, loading, and altering **Product** data objects. It achieves this via serialization and deserialization of **Product** objects to and from a JSON file.

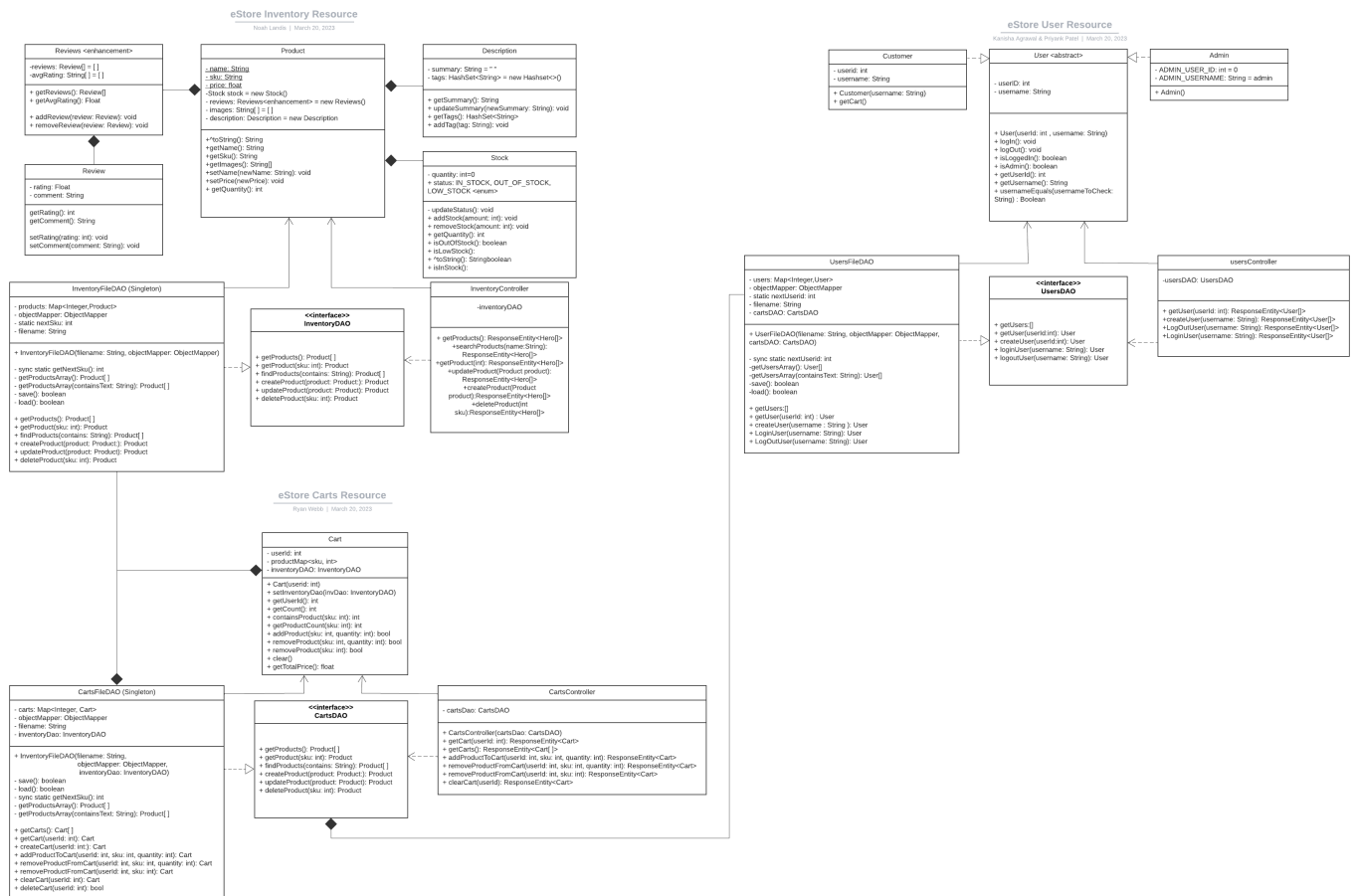
The **User** resource is responsible for storing the **User** data objects and providing persistence. The primary data object stored in the **User** resource is the **User** class stores a user's ID, login state, and whether the user is an admin or not. The **User** class also encapsulates behavior related to said data.

Persistence is provided by the **UsersDAO** interface, and its concrete implementation **UsersFileDAO**. The **UsersFileDAO** class is the layer between the user data objects and the controller tier, and provides methods for saving, loading, and altering **User** data objects. It achieves this via serialization and deserialization of **User** objects to and from a JSON file.

The **Carts** resource is responsible for storing the cart data objects. The primary data object stored in the **Carts** resource is the **Cart** class. The **Cart** class stores a user's ID, a map of SKUs to quantities in their cart, and an injected reference to the **InventoryDAO** singleton instance, which allows the class to calculate the total price of the cart, as well as check stock quantities before adding items to the cart. The **Cart** class also

encapsulates behavior related to cart operations such as adding products, removing products, clearing the cart, and checking if the cart contains a product.

Persistence is provided by the **CartsDAO** interface, and its concrete implementation **CartsFileDAO**. The **CartsFileDAO** class is the layer between the cart data objects and the controller tier, and provides methods for saving, loading, and altering **Cart** data objects. It achieves this via serialization and deserialization of **Cart** objects to and from a JSON file.



OO Design Principles

Dependency Inversion Principle (DIP)

Dependency inversion (D in SOLID) is adhered to in this project because we rely on abstraction interfaces instead of low-level concrete implementations for data storage. For data storage, we will use a Data Access Object (DAO) abstract interface, which multiple data access classes will implement. This way, at instantiation, we can directly inject the data access method we would like to use into the modules which depend only on the DAO abstraction. In a more complicated eStore, the dependency inversion principle could be used to create abstractions for modules handling authentication and authorization, as well as for payment handling. We could also apply dependency inversion to make a logger interface, which is injected into modules that utilize a logger. This makes implementing different types of loggers easier.

Dependency Injection

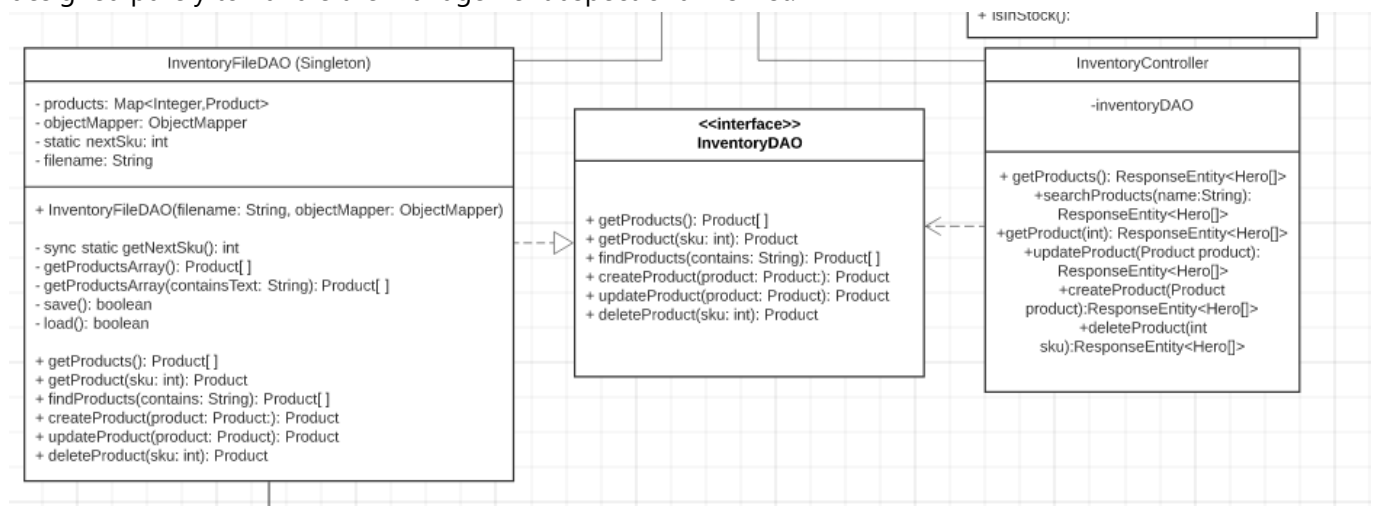
Our application relies heavily on dependency injection in all tiers. In the controller tier, Spring Boot injects the **InventoryFileDAO** into the **InventoryController** class. This allows the **InventoryController** to use the

InventoryFileDAO to access the data in the inventory. In the model tier, the InventoryFileDAO is injected into the Product class. This allows the Product class to use the InventoryFileDAO to access the data in the inventory. In the angular front-end, we use dependency injection for the services. We have a service for each of the REST API resources, a service for authentication, as well as a service to handle updating of components when their contents change. The services are injected into the components that need them. For example, we inject the InventoryService into components that need to fetch data from the inventory resource. This allows us to easily swap out the implementation of the service without having to change the components that use it.

 Diagram for Dependency Inversion

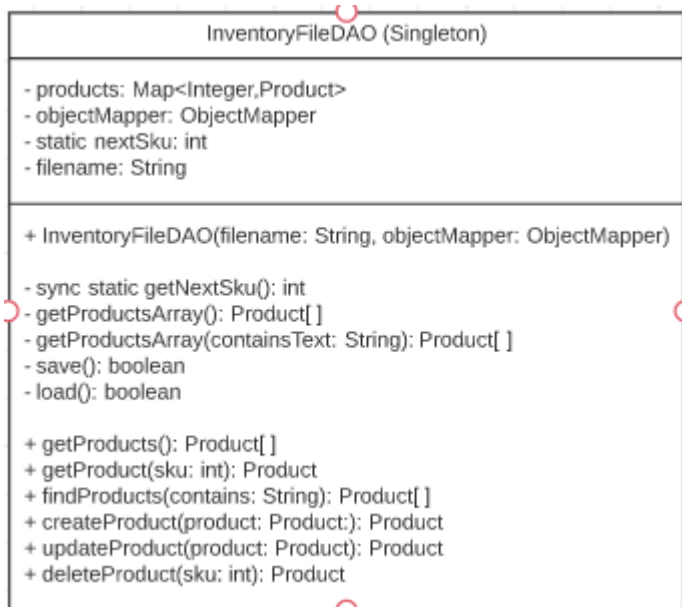
Pure Fabrication

Creating a software system dedicated to handling data access also adheres to the pure fabrication principle of GRASP; it isn't directly represented in the problem domain, yet its fabrication is integral to the solution architecture. Looking back at the eStore domain model, the Inventory domain entity represents the result of using a data access layer. The data access layer is not present in the model; it merely supports the behavior of the system architecture. Another example of pure fabrication in our eStore would be a wishlist management system. The management system does not correspond to any particular entity in the problem domain and is designed purely to handle the management aspect of a wishlist.



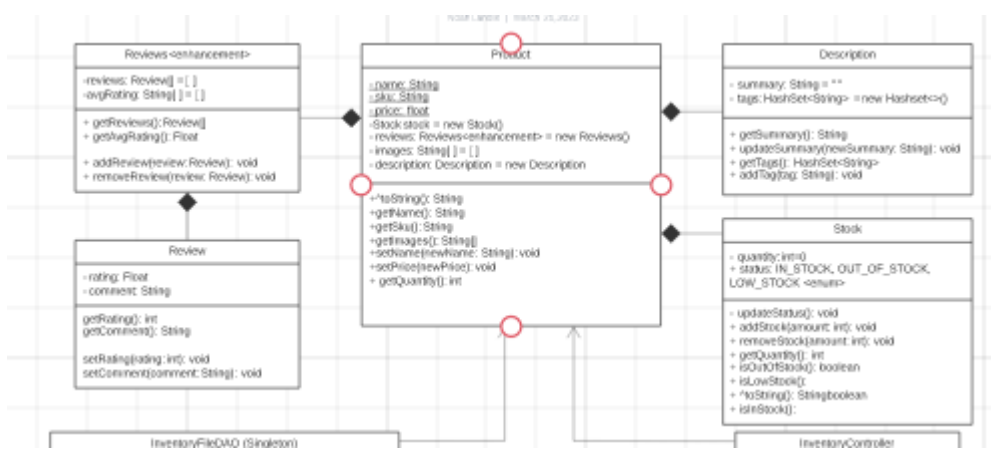
Single Responsibility

We have an API for the CRUD operations related to a product. In the API, this principle can be seen with the **InventoryFileDAO** class. This class is particularly concerned with handling the underlying data which is stored in a json file which stores the data for this API. It is not concerned with handling API requests or giving responses back to the user. That is done by **InventoryController** class which is more of a front for this class which makes calls to this class. This is the file that implements the **InventoryDAO** interface. It maintains a hashmap of all the products stored in the JSON file when initialized. Any searching is done directly from the hashmap. When a new product is to be added to the file, the hashmap is updated first and then it is overwritten on the file that stores the products. So the hashmap is basically a temporary point of storage between the user and the file.



Law of demeter

In our Java classes, we didn't use method chaining such as `product.getStock().getQuantity()`. Instead, we have a `getStockQuantity()` method which is accessed directly from the `Product` class. In doing so, we followed the law of demeter. In the case where we want to ensure enough stock quantity for a product when adding to the cart, we don't make direct calls to the `Stock` class, we have a method called `hasEnoughStockFor()` rather than making a direct call from the `Cart` class to the `Stock` class.



Controller

In our backend, we have a controller tier which handles API requests from the Angular frontend (view). The controller tier is responsible for handling the requests and sending the appropriate responses back to the frontend, and represents a separation of responsibility between our View tier (the Angular frontend) and the Model tier (backend API DAO classes and supporting data model classes).

Information Expert

In our frontend Angular code, we have a catalog component which uses product-card component inside it to display the products. This component utilizes a service called product service which can fetch the products in the inventory and display them to the frontend. The catalog component already has the service injected into it which contains the method to get the products from the inventory, i.e., the backend. This is information expert

because of instead of having the separate cards fetch the data, we have the catalog do it for the cards and then pass that value to that card component. _

[Sprint 3 & 4] OO Design Principles should span across **all tiers**.

Static Code Analysis/Future Design Improvements

[Sprint 4] With the results from the Static Code Analysis exercise, **Identify 3-4** areas within your code that have been flagged by the Static Code Analysis Tool (SonarQube) and provide your analysis and recommendations.

Include any relevant screenshot(s) with each area. **[Sprint 4]** Discuss **future** refactoring and other design improvements your team would explore if the team had additional time.


Testing

This section will provide information about the testing performed and the results of the testing.

Acceptance Testing

We had a total of 10 stories in the acceptance testing document, which is the total number of user stories. Out of those 10 stories, 3 of those stories didn't meet the acceptance criteria that we had come up with while doing sprint planning. The tests that failed were testing features that weren't required for the sprint 2 like adding images to product. One functional requirement that we did fail is getting the button to remove a product from the inventory working. This was a good practice to explore all the test cases that could break the code.

Unit Testing and Code Coverage

[Sprint 4] Discuss your unit testing strategy. Report on the code coverage achieved from unit testing of the code base. Discuss the team's coverage targets, why you selected those values, and how well your code coverage met your targets.  Model tier code coverage for backend

 Controller tier code coverage for backend  Persistence tier code coverage for backend

We ensured code coverage for all public methods and thus we have high code coverage. We used Mockito wherever necessary. We also used PowerMockito to mock static methods. We used Jacoco to generate the code coverage reports. We used that to monitor the code coverage.