

# PROJECT Design Documentation

---

## Team Information

- Team name: CARDSHARKS
- Team members
  - Ryan Schachel
  - Josh Matthews
  - Elijah Lenhard
  - Adrian Marcellus
  - Rohan Rao

## Executive Summary

This is a summary of the project.

### Purpose

This e-store is an online hub that allows users to buy, sell, and trade their trading cards with ease. Our e-store's special trading feature supports fast and easy card trading between users. Customers can expand their card collection like never before.

### Glossary and Acronyms

Term	Definition
API	Application Programming Interface
REST	Representational State Transfer(An industry standard for creating APIs)
SPA	Single Page
Persistence	Ability to store and access data across sessions and processes

## 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

In our first working increment, the CardSharks e-store allows users to login and see their collection. On the store, Customers view a full list of products and a search bar for quickly locating coveted cards. Patrons have access to a shopping cart to add and remove items as well as proceed to checkout. The e-store owners can modify the inventory as items are bought and sold. Our e-store handles day to day data persistence, letting users instantly see up-to-date inventory counts.

## MVP Features

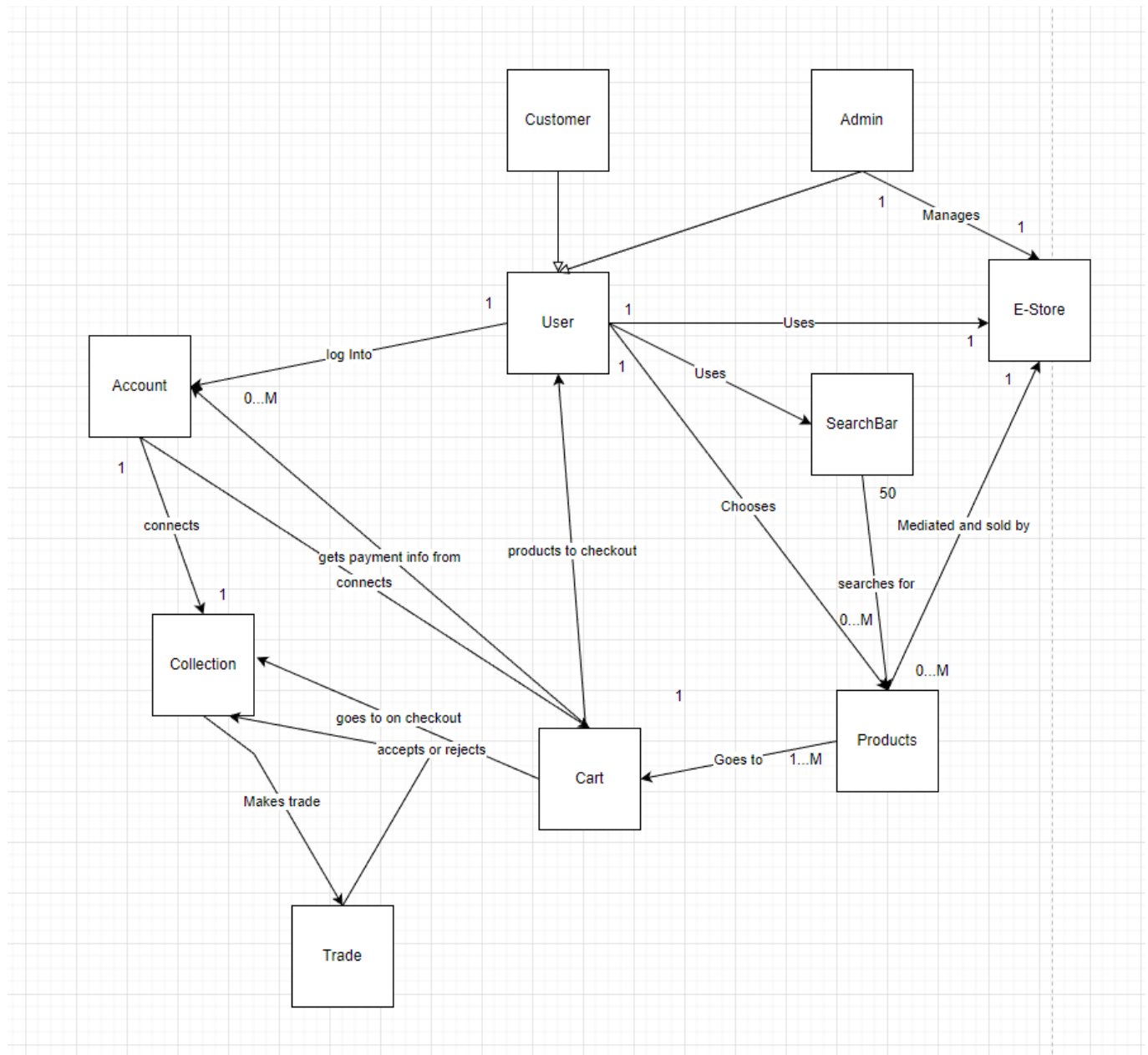
- Login/Logout - Customers and owners can login to view their card collection or make changes to the e-store.
- Search for Items - A customer can search for a specific item so they can purchase it
- Add/Edit items from Shopping Cart - A customer can add items to their shopping cart so they can buy them later. They can also edit and remove items from their shopping cart.
- Checkout - Customers can review their items, add payment info, and purchase items.
- Inventory Interaction - E-store owners can edit the stores inventory. These changes along with inventory changes due to customer purchases will be saved across sessions using data persistence.

## Enhancements

- Trade Cards - Users are able to trade cards from their collection to another user for one of their cards.
- Saving Payment Information - Users can add, delete, and edit their payment information so they can use it in the future.

## Application Domain

This section describes the application domain.



A Customer or Admin can Log-In and use the E-Store. The E-Store mediates the selling, buying, and trading of Products. These Products can be added/removed from a Customers Cart or added/deleted from the inventory by an Admin. The Search Bar searches for Products so that a Customer can find them quickly. Each Users Account has a Collection that stores the cards they own. A User can trade cards from their Collection to another User in exchange for a card in the other Users collection.

## Architecture and Design

This section describes the application architecture.

### Summary

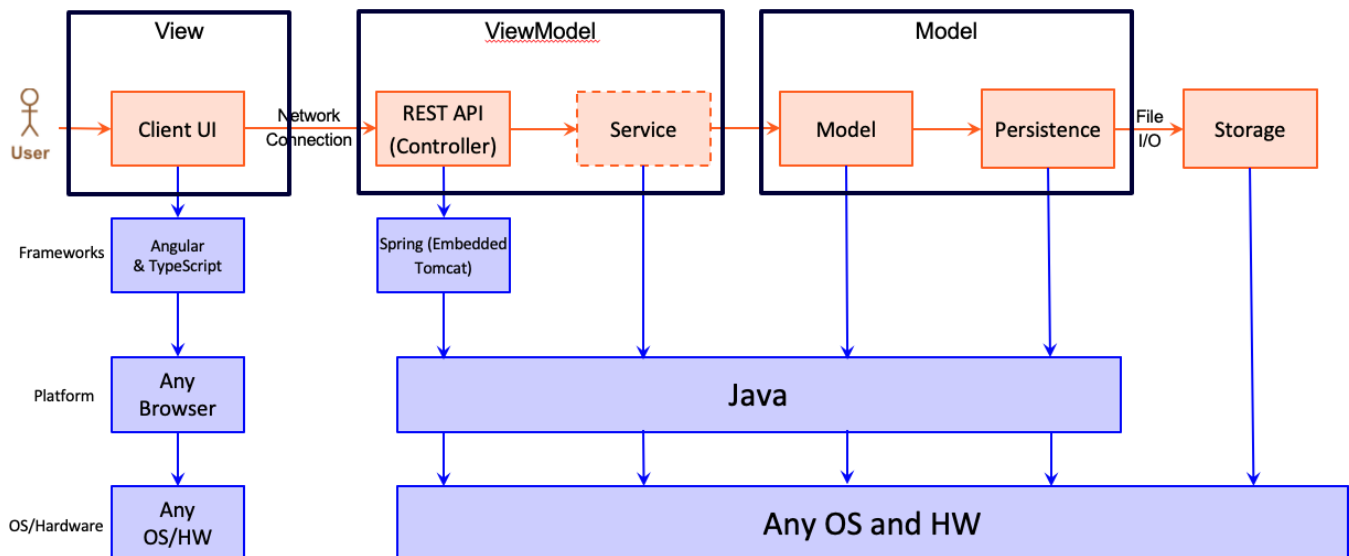
The following Tiers/Layers model shows a high-level view of the webapp's architecture.

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.



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 webpage is broken up into several pages. These pages are navigated to with the button on the top bar. Certain buttons such as Cart, Collection, and Edit Inventory are only available when logged in with edit inventory only available to an admin.

## CardSharks

[Main Menu](#)[Account](#)[Cart](#)[Collection](#)

### Featured Cards

Pikachu   \$15 In-Stock: 463	Squirtle   \$5.25 In-Stock: 147	Wartortle   \$10.5 In-Stock: 219	Blastoise   \$15.75 In-Stock: 539
Charmander   \$25 In-Stock: 220	Charmeleon   \$35 In-Stock: 612	Charizard   \$12.5 In-Stock: 47	Ivysaur   \$45.5 In-Stock: 75

### Product Search

---

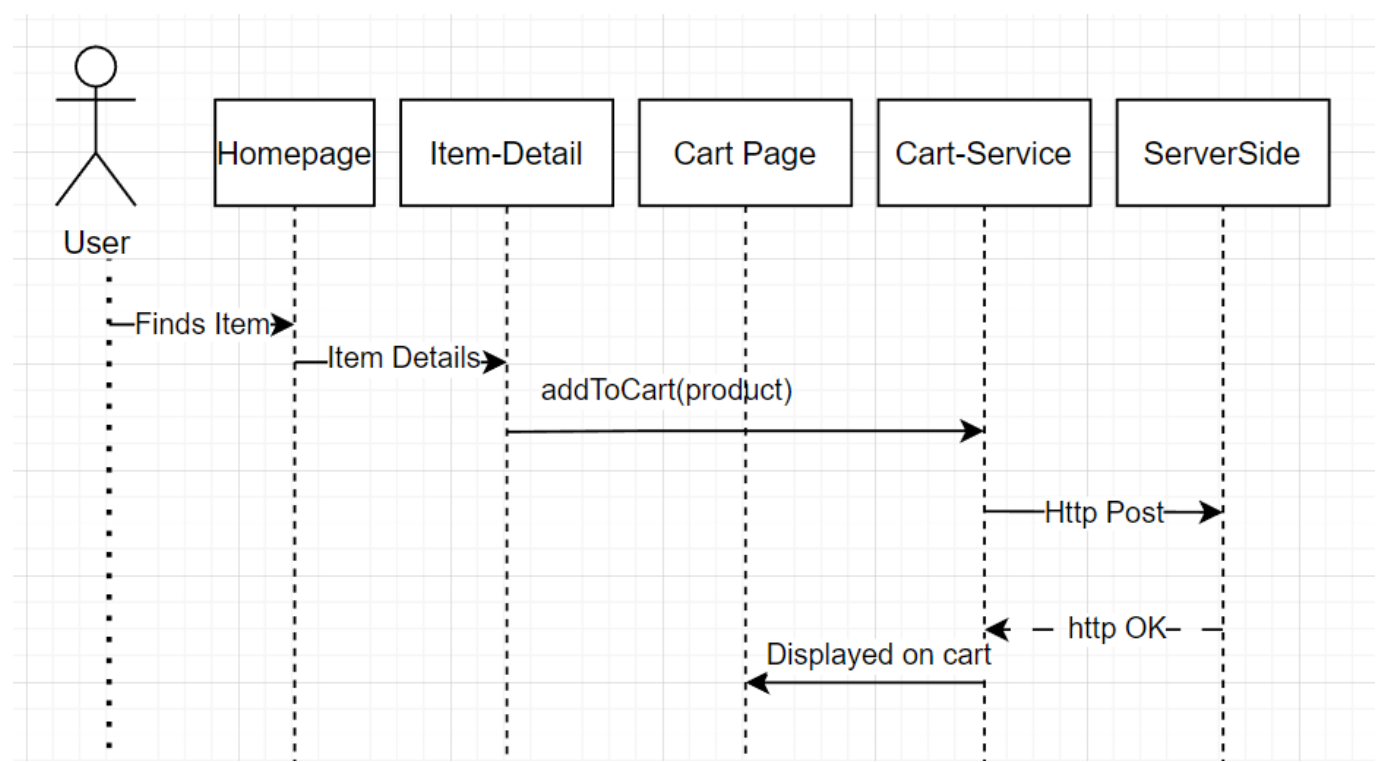
A customer would go to the account tab, and log in with their username. From there they can add items to their cart, either by finding them in the list or by searching for them. From there they can navigate to the cart tab where they can view and edit their cart, or checkout.

## View Tier

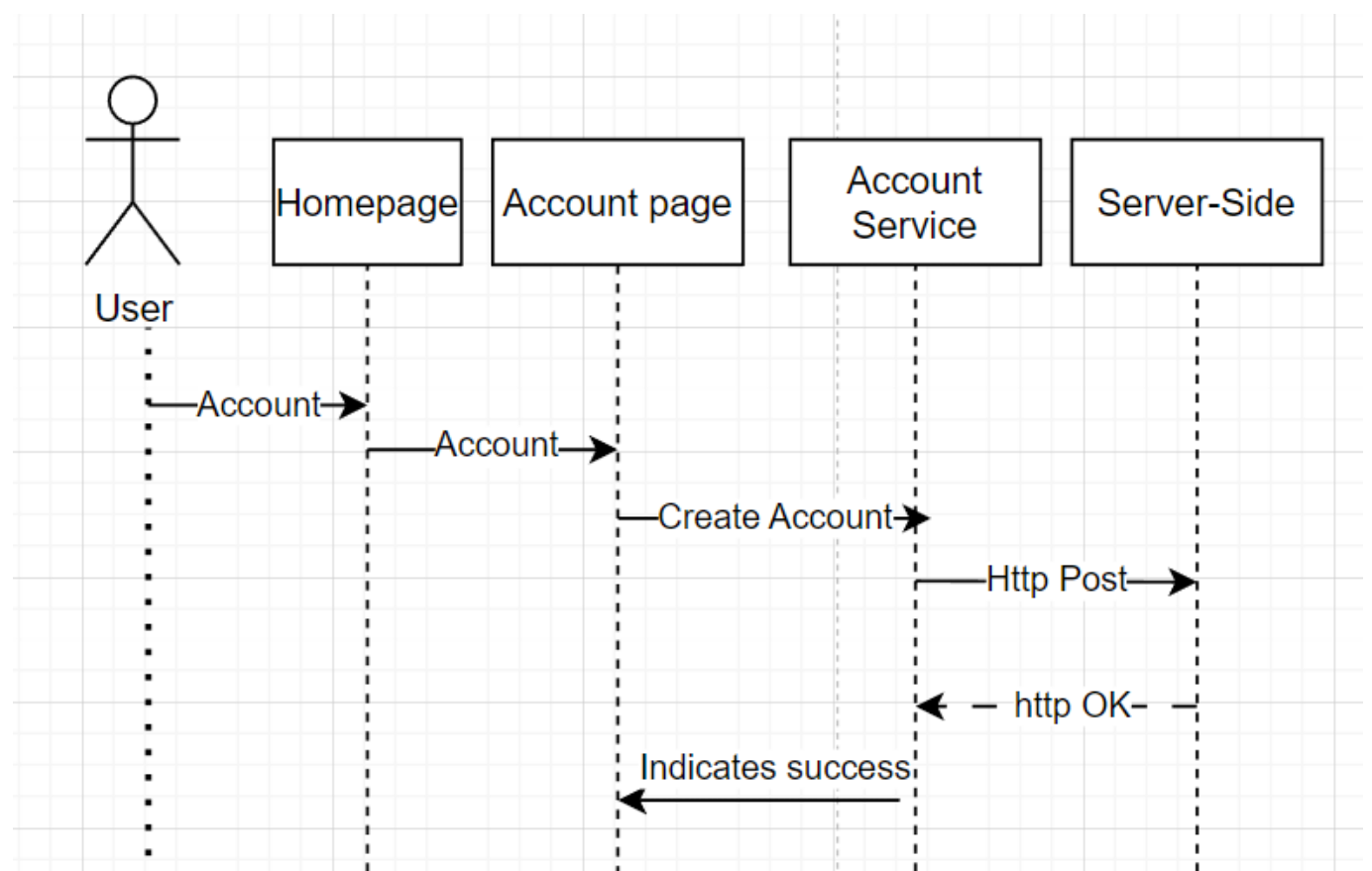
The view Tier consists of the Angular front end that utilizes html, css, and typescript. It's broken down into several components and services. such as log-in-out, cart, and admin-inventory-controls. Each component consists of an html template, css sheet, and typescript class for functions. In addition there are several services such as the account service which handles communications with the backend through http calls, and stores session data related to accounts. There are also several data classes which are used when communicating with the backend and for storing session data.

Some examples of how the View interacts with the viewmodel and user:

The user interacts with the Homepage and item detail page which then calls the cart-service. All communication with the backend related to carts is handled through this service

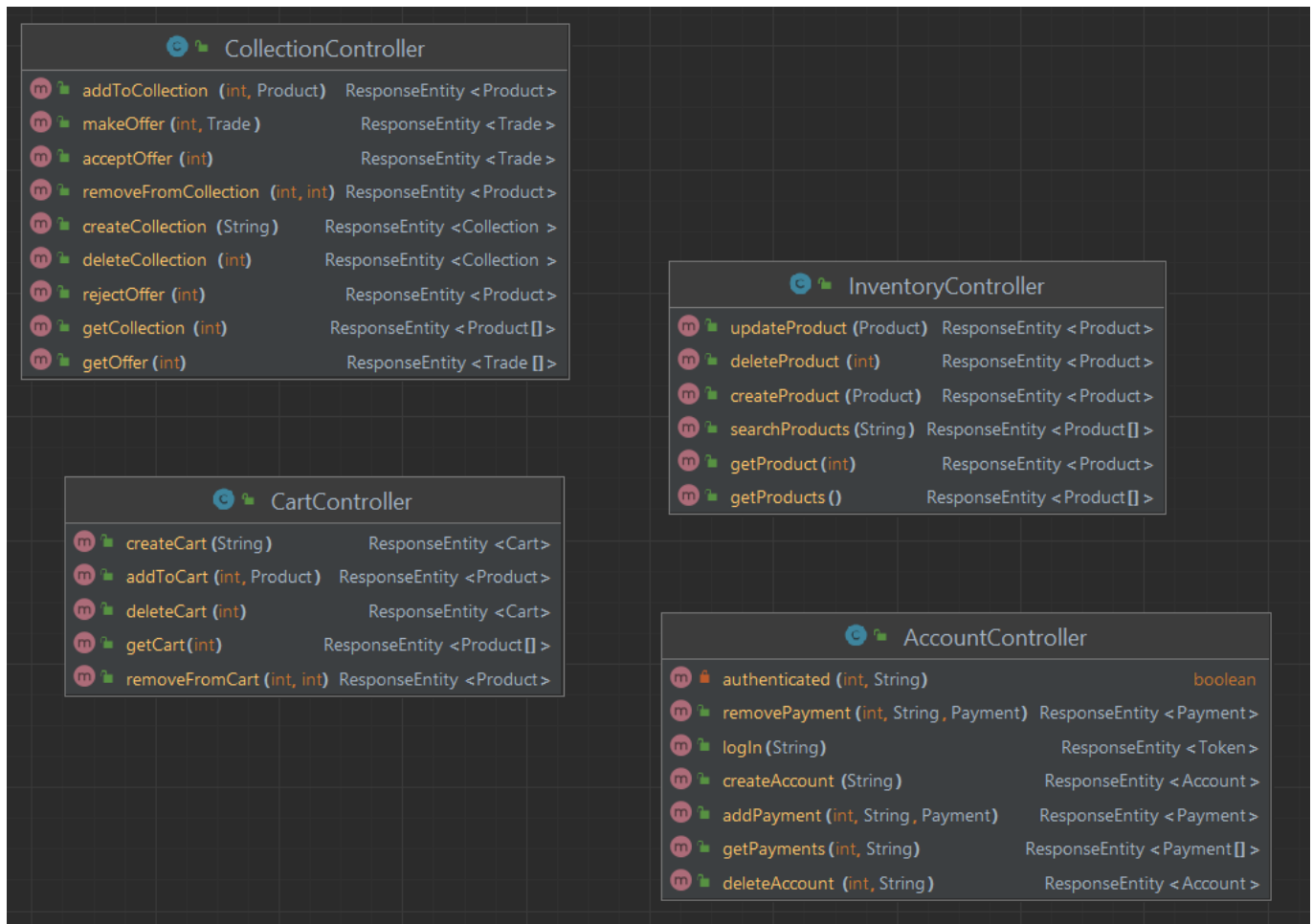


Here the user creates an account. Similar to above, the account-service handles calls to the ViewModel.



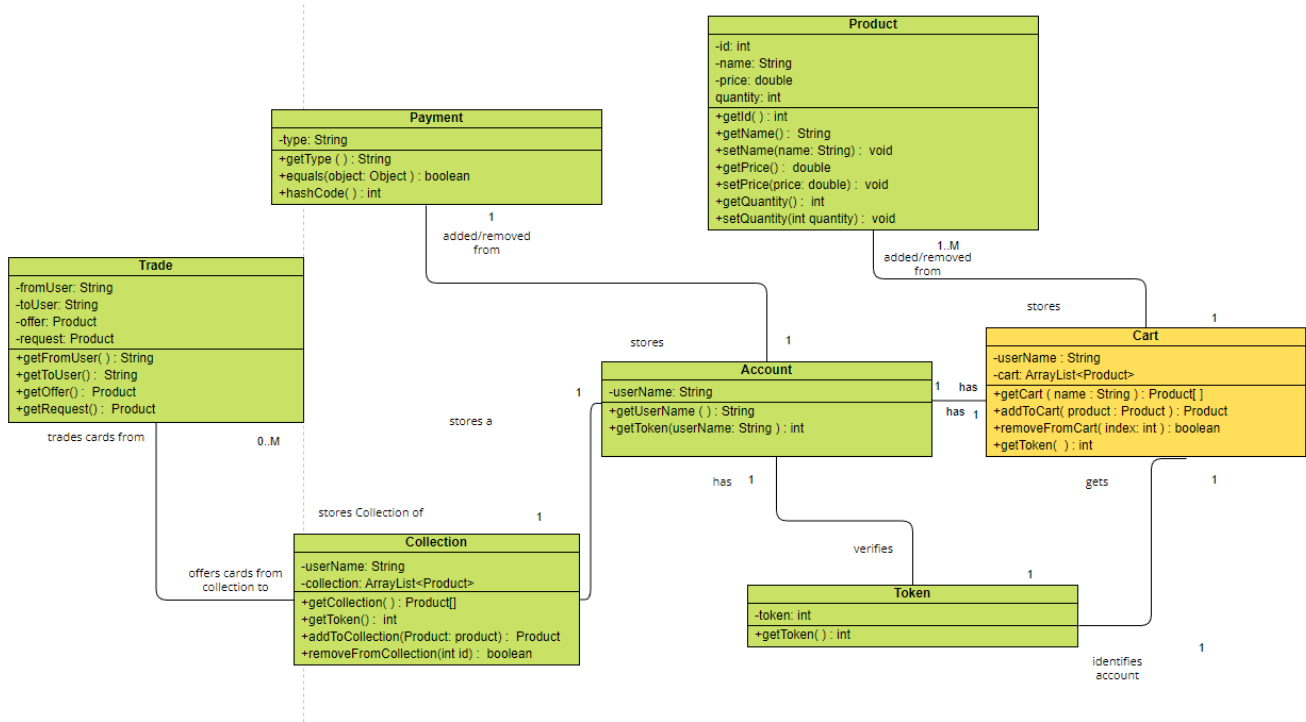
ViewModel Tier

The ViewModel Tier primarily consists of the controllers that contain http functions. The view interacts with the ViewModel exclusively through these functions. These functions specify their http mappings with an annotation.

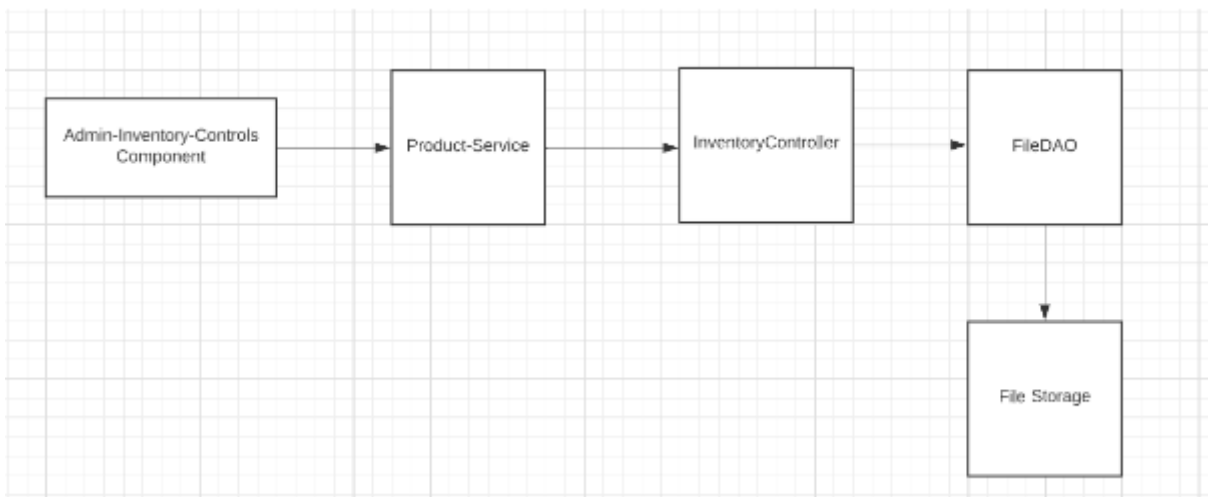


## Model Tier

The Account model stores user information and provides methods to get the Username and the Token assigned to it. Each account has an association to the Cart model. One Cart is assigned to One Account. The Cart can add products from the Product model and remove them too. The Product model stores information such as Id, Name, Price, and Quantity for each product. Finally, we have the Token model which verifies each account and identifies the account a cart is assigned to.

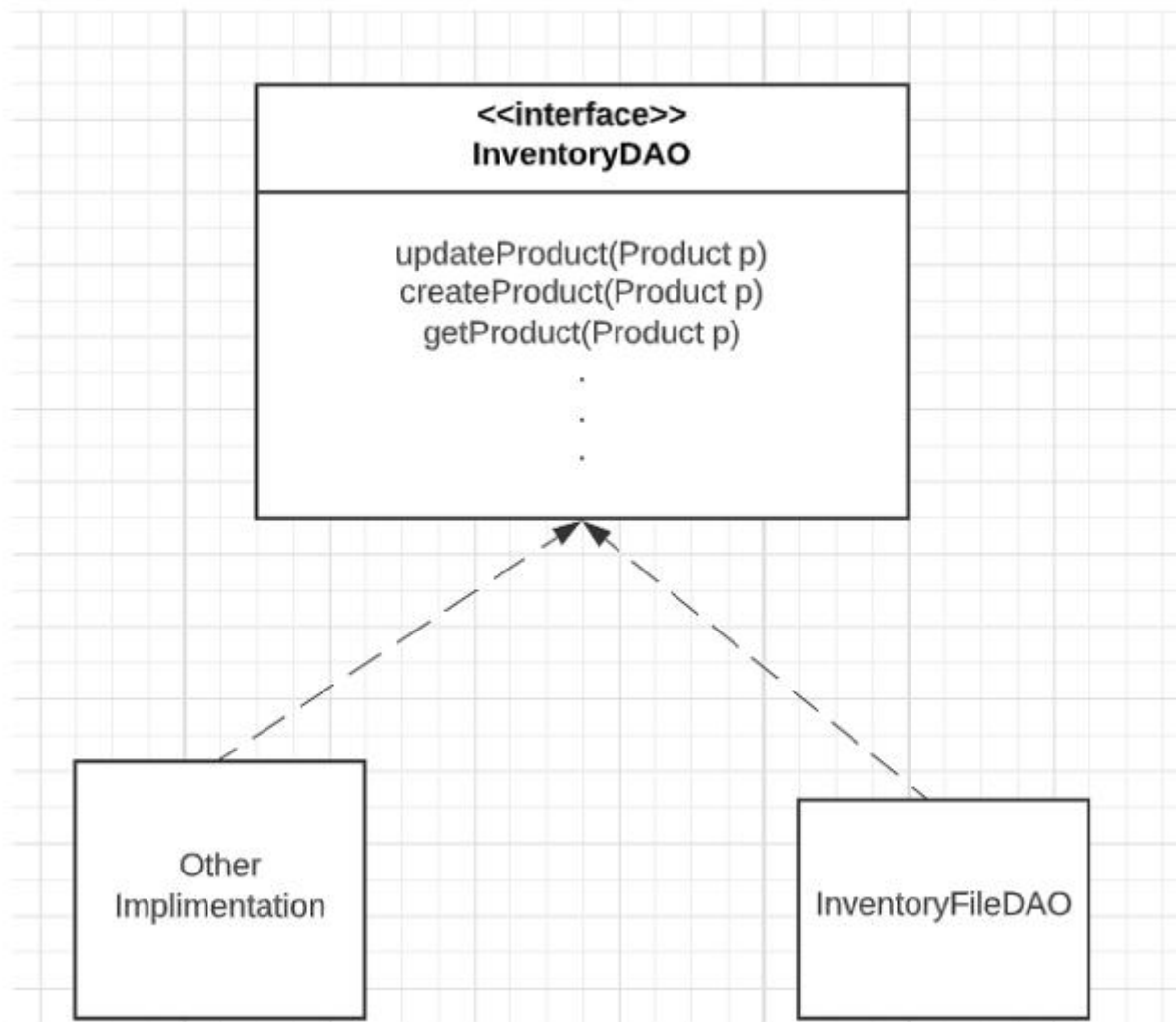


## OO Design Principles

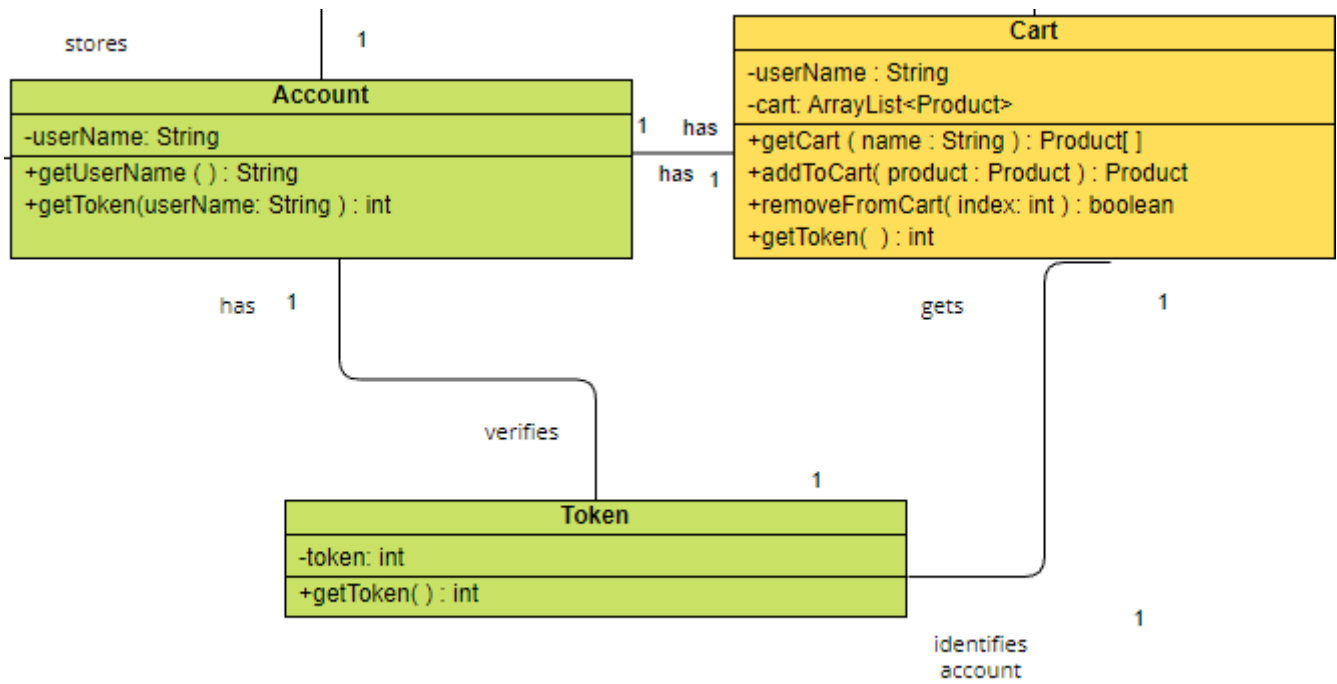


The Law of Demeter will be followed as all products have private data fields and appropriate getters and setters. The Angular framework will also inherently follow this principle with its use of components. Functionality is encapsulated within components while still allowing interaction between components without sharing internal data. We also use a controller to talk to the Data Access Object interface implementation which stores and modifies the product list in a JSON. So the Angular front end follows the principle as well as the backend. Below demonstrates how the Angular Components talk only to their immediate neighbor, a service, which talks to it's neighbor and so on, instead of the component directly editing the files or talking to the fileDAO or Controller.

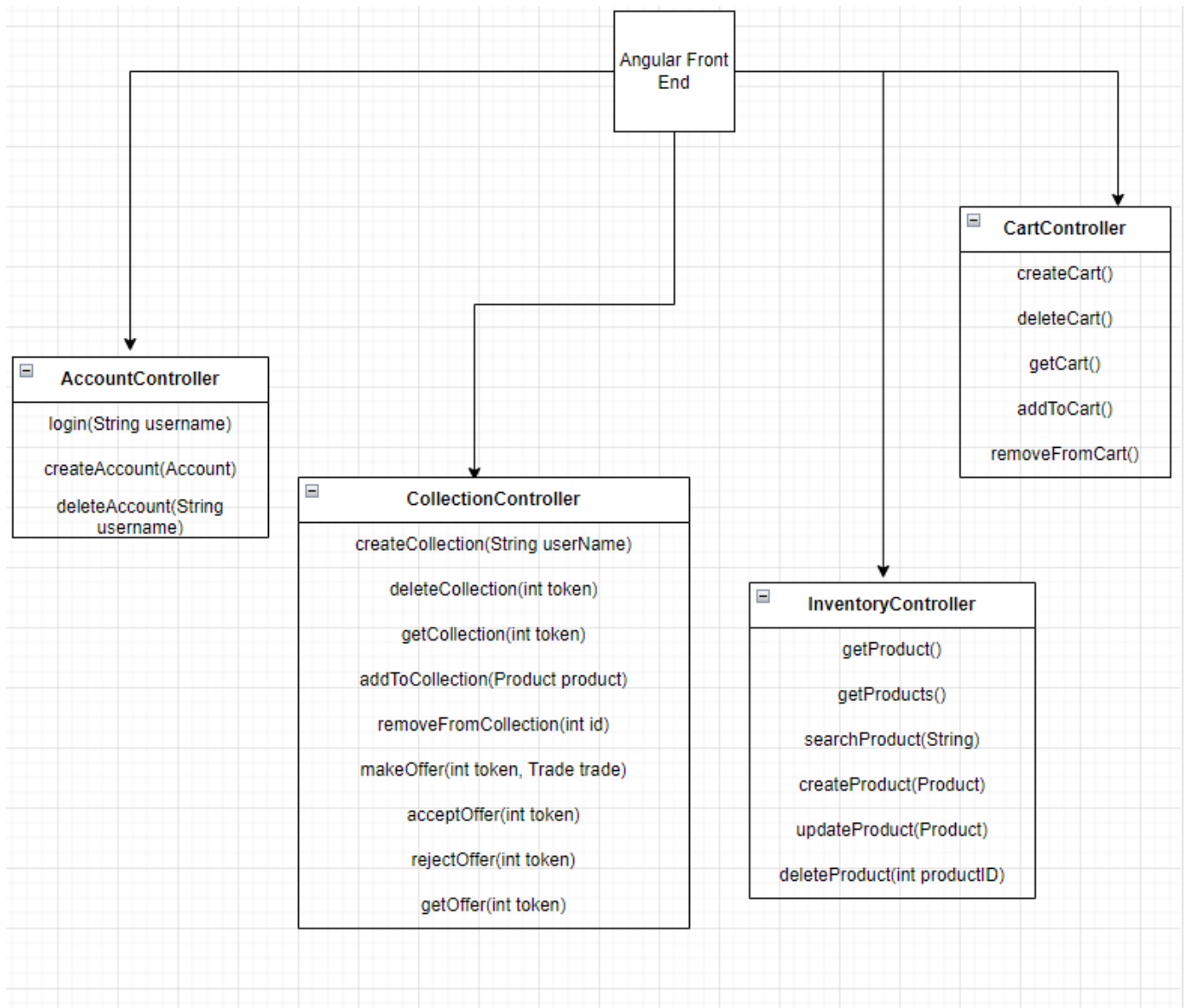




Loose coupling is the concept that systems that interact with each other shouldn't need to care about how the work gets done. The only thing that matters is the external view of the class such as method signatures and return types. Interfaces are very helpful with enforcing this. For example, in our web application, the controller doesn't need to worry about how the DAO implements its methods. It could be swapped out with another implementation and as long as the method signatures and return types stay the same, the app would still function. Loose Coupling is also achieved in Angular by using dependency injection. By injecting the services into the components, the components do not need to know implementation details. This is important because it allows us to change the implementation of the dependency without affecting the component.



The Pure Fabrication pattern suggests creating a class to do work that can be reused by a class or classes. This class does not represent a domain entity. Using this design pattern, low coupling, cohesiveness, and Single Responsibility can be achieved. As the project expands, Pure Fabrication could be applied in situations where some tasks need to be performed that are not directly related to the class that needs it. For example, the E-store or sales class will need to update the inventory data after a purchase is made, a product is created, or stock has been added. The best way to do this is to make a class that handles the data manipulation and have it call the sales class. This way the E-store and sales class can stay in a Single Responsibility state. The Token class is an example of pure fabrication since it is not a domain entity, but it facilitates the authentication of accounts and the making of trades and transactions.



As stated in the lecture slides for object-oriented design, the controller acts as the separation (or go-between) for system operations and the user interface. Currently, the team's application uses 4 controllers for the purpose of managing the inventory, accounts, and carts. Specifically, it oversees operations that include altering various objects such as adding, deleting and editing these objects. Each controller has different methods since they have different requirements. As seen below the front end only communicates with the back end through http calls to these 4 controllers.

## Static Code Analysis/Future Design Improvements

estore-api <b>Passed</b>		Last analysis: 6 minutes ago						
Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Lines		
0 <b>A</b>	0 <b>A</b>	– <b>A</b>	215 <b>A</b>	90.2%	0.0%	1.4k <b>S</b> Java, XML		

estore-ui <b>Passed</b>		Last analysis: 13 days ago						
Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Lines		
0 <b>A</b>	0 <b>A</b>	– <b>A</b>	24 <b>A</b>	0.0%	1.8%	1.4k <b>S</b> TypeScrip...		

No bugs are detected in the project but has some remaining code smells. Most of the code smells seem to be false flags or a preference. As seen below most are pointing to using a built in format for argument construction and duplication of mapping URL's. While defining constants is usually preferred it helps readability to leave as is.

Use the built-in formatting to construct this argument.		14 days ago ▾ L58		
	Code Smell  Major  Open Not assigned 5min effort			performance

Use the built-in formatting to construct this argument.		16 days ago ▾ L82		
	Code Smell  Major  Open Not assigned 5min effort			performance

Use the built-in formatting to construct this argument.		16 days ago ▾ L107		
	Code Smell  Major  Open Not assigned 5min effort			performance

Define a constant instead of duplicating this literal <code>"/payment?userName="</code> 3 times.		8 days ago ▾ L132		
	Code Smell  Critical  Open Not assigned 8min effort			design

Given enough time we would implement more functionality to our 10% trade feature so it would be easier to use and look nicer. We would also spice up the the HTML/CSS for more user satisfaction.

## Testing

Testing from SonarQube and SonarScanner has detected no bugs. There is also testing for each API model, DAO, and controller to make sure they are working properly.

### Acceptance Testing





All nine of the sprint 3 stories fully passed their acceptance testing. All the stories that failed acceptance testing in previous sprints have been fixed and now fully pass.

### 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.

 estore-api

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 <a href="#">com.estore.api.estoreapi.persistence</a>	<div><div></div></div>	90%	<div><div></div></div>	61%	32	92	26	222	0	49	0	4
 <a href="#">com.estore.api.estoreapi.model</a>	<div><div></div></div>	96%	<div><div></div></div>	81%	8	61	5	106	2	45	0	7
 <a href="#">com.estore.api.estoreapi.controller</a>	<div><div></div></div>	99%	<div><div></div></div>	91%	5	65	1	214	0	35	0	4
 <a href="#">com.estore.api.estoreapi</a>	<div><div></div></div>	88%		n/a	1	4	2	7	1	4	0	2
Total	132 of 2,578	94%	44 of 178	75%	46	222	34	549	3	133	0	17

Created with [JaCoCo](#)