

# ItsTeamTwo e-Store Design Documentation

---

## Team Information

- Team name: itsTeamTwo
- Team members
  - Macy So
  - Chen Guo
  - Greg Lynskey

## Executive Summary

The Overpriced Grocery store is where you can find your daily needs, and our prices are far above our competitors. Our E-Store allows customers to order their products from home via the cart function. If a customer is not happy with a product, they can leave a written review to inform our administrators what products the customers actually want. The e-store functions on a REST API backend written in Java 11 which is then controlled by an angular frontend. Relevant data persists in a JSON file.

## Purpose

The product should function like a normal online store. Our primary user would be the customer, who can use the cart function to add or remove desired items then order once they are happy with their order. Another prime user would be the admin/store owner where they can add or remove products from the store, they can also read the customers reviews to better inform them of how their products are performing.

## Glossary and Acronyms

*Provide a table of terms and acronyms.*

Term	Definition
SPA	Single Page
DAO	Data Access Object
UI	User Interface
REST	Representational state transfer

## Requirements

This section describes the features of the application.

- User - Login - Admin/Customer
- Customer - add, remove checkout, browse products
- Admin - add, remove, edit products

## Definition of MVP

MVP is a product with enough features to attract early-adopter customers and validate a product idea early in the product development cycle

## MVP Features

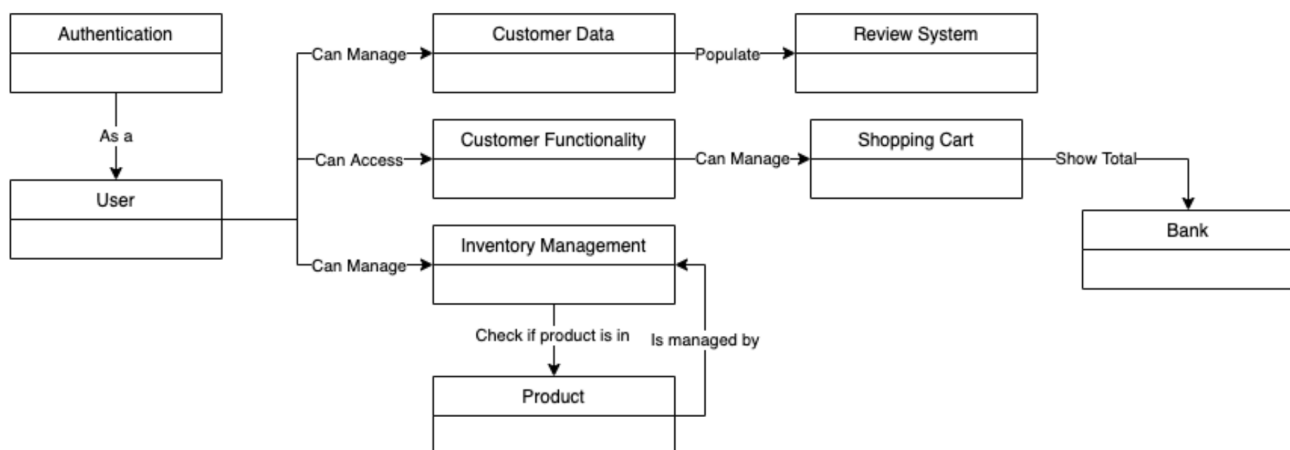
- User - Login admin / customer
- Admin add product, edit product, remove product
- Customer browse product, see product details, add/remove/checkout cart

## Roadmap of Enhancements

- Add product to wishlist
- Admin are able to add product images
- Leave reviews - star reviews + written reviews

## Application Domain

Our target market are those who want to purchase wacky grocery store items for outrageous prices.



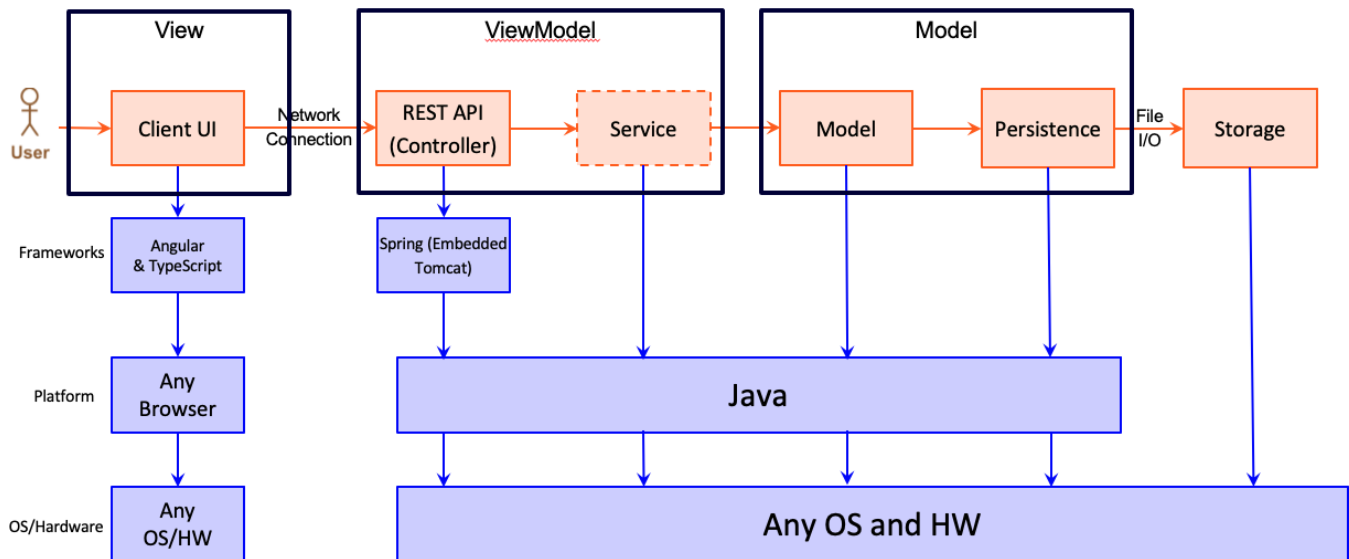
First, we have a inventoryController that controller the functionality of the admin function, including get a single product, create new product, get a list of products, delete product, search product and update product. This is where we set up the HTTP status. It will be linked with the Product class itself, and the inventoryDAO class as well. In the Product class, we will be storing each other with an id, productName, price and it's description. Within the class, we will have fields such as getting the id, getting the name, getting the price, description and setting names, prices and description. This is to support the admin functionality of edit, add and remove product. The inventoryDAO is basically a interface for inventoryFileDAO. In the InventoryDAO, there is function such as getProduct, findProducts, getProducts, createProduct, updateProduct and deleteProduct. And this is closely linked with InventoryFileDAO as there is where we implement the functionality.

## 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.

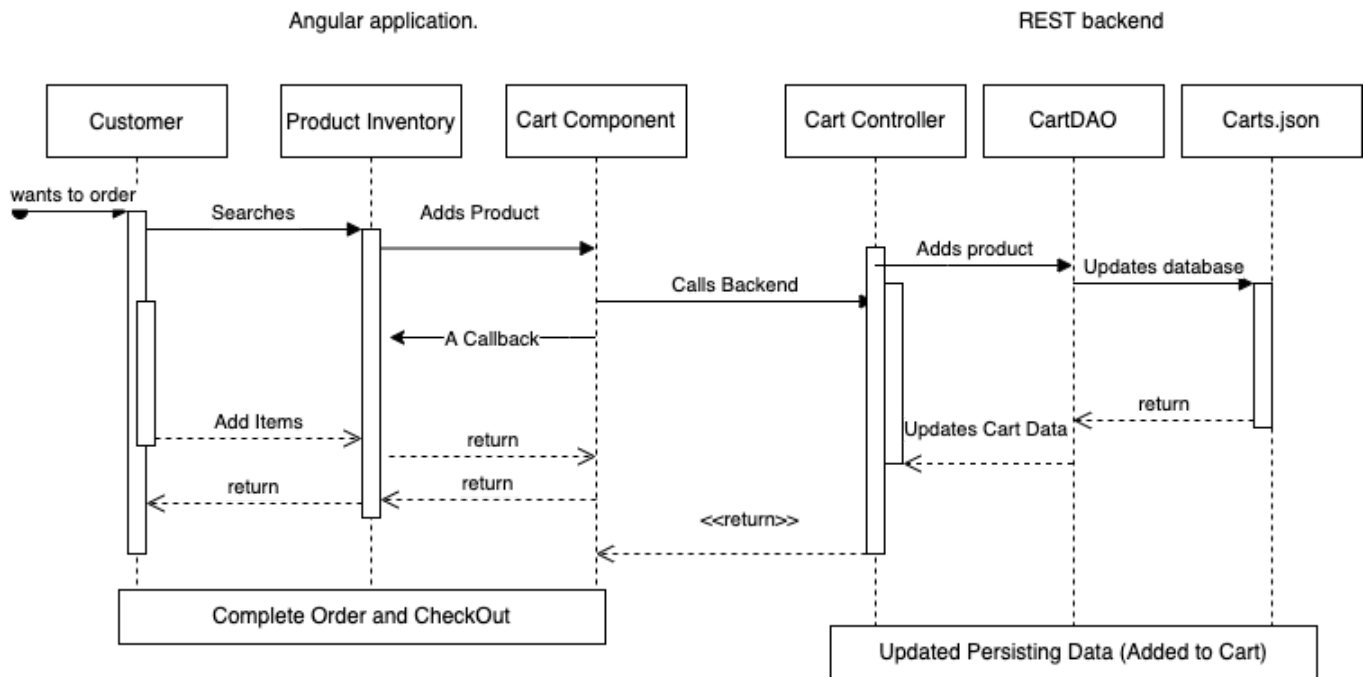
Both the ViewModel and Model are built using Java and Spring Framework. Details of the components within these tiers are supplied below.

## Overview of User Interface

The user is able to view the site from a few different views. If they are a customer they will see a full list of products, and will have access to the cart functionality. If they want to see more about an item they can click on the product description. If they are an admin then they have the ability to update the product inventory and read customer reviews.

## View Tier

The View Tier UI, we have the shopping-cart components and the admin component. We also have a shared component folder where the login, register, and other shared elements such as header, and footer. The shopping cart component only focus on the customer functionality, including browsing product, add product to cart, view product, and checkout cart. The admin component, have the admin functionality such as the add product, remove products and edit products.



## ViewModel Tier

The controllers for Cart, Inventory, Review, and User all help connect the front end to the backend. When the frontend has a request, it calls one of the backend methods in the controller that will then mutate data and return an updated/modified version of this data.

## Model Tier

The model tier functions using a REST API containing user and product data. The model connects to the front end and displays the current data as requested by the frontend view model. The product data is stored in an inventory, and is accessed by the frontend via a controller connected to a file data access object, which is the current method of storing data. There are also models for the user, review, and cart data.

## Static Code Analysis/Design Improvements

Current improvements are mostly backend to frontend connection that we ran out of time for during the last sprint. Most components are working, they just need a connection to have persisting data.

## Testing

Testing was performed on the product inventory, review, cart and user backend. (Insert code coverage photos once tests pass/all tests complete)

## Acceptance Testing

*Report on the number of user stories that have passed all their acceptance criteria tests, the number that have some acceptance criteria tests failing, and the number of user stories that have not had any testing yet. Highlight the issues found during acceptance testing and if there are any concerns.*

1. Browse Product
2. Add Product to Cart







3. Remove Product from Cart
  4. Checkout Product from Cart
  5. View Product Details
  6. Admin Add Product
  7. Admin Edit Product
  8. Admin Remove Product
  9. Customer see past order
  10. Customer able to review (star rating + written reviews)
- See acceptance test plan for more details.

## Unit Testing and Code Coverage

The general code testing strategy in the backend corresponds to the following, if a method has an outcome, the unit tests will run to recreate that outcome. For most controller tier methods that included all the HTTP RequestMappings to be tested. Most methods had three unit tests, one resulting in OK (200) or CREATED(201), one with NOT\_FOUND(404), and one with INTERNAL\_SERVER\_ERROR (500). All these outcomes are possible with the controller methods, so the tests validated that the if else logic along with the try catch error handling worked as expected.

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.controller</a>		95%		83%	2	14	2	52	0	8	0	1
<a href="#">com.estore.api.estoreapi</a>		88%	n/a		1	4	2	7	1	4	0	2
<a href="#">com.estore.api.estoreapi.persistence</a>		100%		93%	1	21	0	54	0	13	0	1
<a href="#">com.estore.api.estoreapi.model</a>		100%	n/a		0	10	0	18	0	10	0	1
Total	15 of 617	97%	3 of 28	89%	4	49	4	131	1	35	0	5

Created with JaCoCo 0.