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: RIT Garage Sale
- Team members
  - Andrew Bush (apb2471)
  - Jacob Karvelis (jak9527)
  - Kelly Showers (kds1653)
  - Zach Brown (zrb8768)
  - Ethan Meyers (epm2875)

Executive Summary

An online e-store for RIT to sell off many notable landmarks including bricks, various statues from around campus, and decorative trees from lobbies. Uses a database implementation and includes an auction feature for users to sell their own items on the e-store.

Purpose

Provide an online store that allows RIT admins to sell the various landmarks and items of interest around campus, including a place for them to auction off new items.

Seperate authentication for users and admins As a user I want to be able to search for and add items to my cart so that I may buy them As an admin I want to be able to edit the inventory of the store so that I can sell products

Glossary and Acronyms

> ***[Sprint 2 & 4]*** *Provide a table of terms and acronyms.*

Term	Definition
MVP	Minimum Viable Product
RIT	Rochester Institute of Technology
SPA	Single Page
API	Application Programming Interface
DAO	Data Access Object
OO	Object Orientated
HTML	HyperText Markup Language

Term	Definition
CSS	Cascading Style Sheets

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

An e-store that implements all of the following features:

Minimal Authentication allows for a separation between users and the owner. The owner will not have access to a shopping cart and can instead edit and manage inventory.

Customer functionality including customer's ability to see a list of products and search for a product.

A shopping cart for users to add products to and remove products from. Allows users to proceed to a final checkout to purchase the products from the shopping cart.

Inventory management allows the e-store owner to add, remove and edit the inventory.

Data Persistence saves everything to files, or in this case a database, that will show previously made changes including previous items added to a user's shopping cart.

A Database implementation to store data about users, carts and items.

An auction feature allowing users to list certain items for auction.

MVP Features

> _**[Sprint 4]** Provide a list of top-level Epics and/or

- Stories of the MVP.
- Epic: Shopping Cart
- Epic: Purchase Product
- Epic: Basic Frontend
- User Authentication
- Admin Authentication
- Epic: Inventory Management
- Product List
- User Checkout

Enhancements

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

Our two enhancements were the use of a database, and the inclusion of an auction house.

Auction House

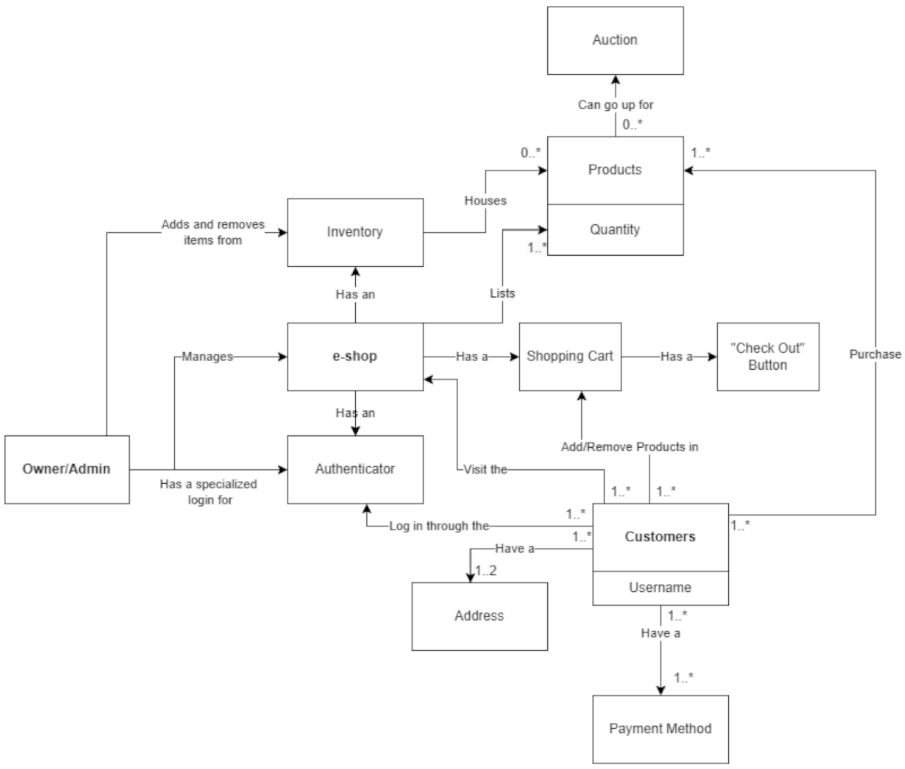
The auction house lists one product that is up for auction. The admin can decide the product to list, the starting bid, and the date and time at which the auction should end. After the auction is saved, it immediately begins. The admin cannot change any attributes of an auction after it begins to ensure fairness to end users. If the admin could suddenly just remove a bid from the item, or change the end time, that would be very unfair. The admin can delete an auction.
Users can see the item for auction, an image of it, the current top bid and top bidder, as well as a countdown to the end time of the auction. Users can place bids, and if their bid is higher than the current leading bid, their bid will become the leading bid, and they will be the leading bidder. Otherwise, their bid will not be placed. After the defined end date and time has passed, no more bids can be placed. Upon viewing a completed auction, a user will see the top bidder listed as the user. If a user stays on the page from before it ended, the page will not allow them to place a bid, and upon trying to, the page will update.

Mongo Database

Our project used a database to store some of the necessary data. At the moment, it currently stores all user data. It is hosted on MongoDB Atlas, a cloud based implementation of MongoDB. It is accessed using a DBDao, which conforms to the UserDao. Previously, the project used a File based implementation of the UserDao, but the use of an interface made the switch as painless as possible.
The use of a database abstracts away some of the storage, and allows parts of the site to operate without the need to store information locally on the client machine.

Application Domain

This section describes the application domain.



The e-shop has an inventory, where all current info about products, prices and quantities are stored.

The owner has control over the inventory of the store, they can add or remove items, update quantities and create new item listings.

Products can go up for auction, where they can be bid on by customers.

Customers have a unique username used for authentication and differentiation from admins. Customers can store up to 2 addresses and store their payment methods to pay for items on the e-store that are stored in their shopping cart.

The shopping cart contains items that customers wish to purchase. Admins do not have access to a shopping cart. The cart contains a check out button, allowing customers to move to the checkout to make their final purchase.

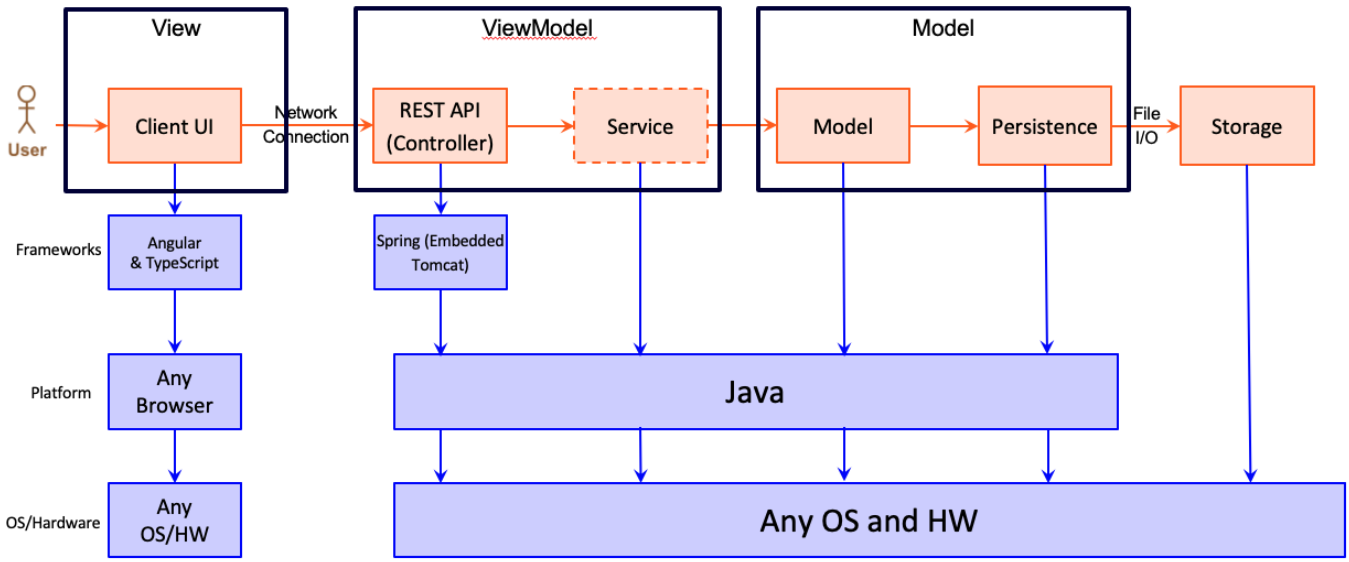
Customers and Admins log in through the Authenticator, which changes what view the user has.

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.

The user first encounters the login page upon coming to the site, from here they cannot move on to another page until first logging in. After logging in the navigation bar appears with a dashboard, product list page, a shopping cart (only if not an admin user), and login page button, each which takes the user to the respective page. Then on any of the given pages with products listed, if a product item is clicked the website takes the user to a details page on that product which still has the navigation bar but also contains a back button which redirects the user back to the page where they clicked the product.

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.*

The ViewModel tier in our project contains both RestAPI controllers, and angular services. We will explain them in two parts.

Rest API Controllers

To uphold maintainability and expandability as much as possible, the Rest API Controller portion of the ViewModel tier was split down into many controllers, rather than just one. Each controller corresponds to a different part of functionality for the site. This allows for easy extensions and easily adding more functionality.

There are controllers to handle requests for: Auctions, Carts, CurrentUsers, Users, and Products. Each handles specifically their own kinds of objects, and no others. Some do need to have access to the model tier for pieces unrelated to themselves. For example, the Cart Controller has a DAO for carts, but also for Products, to allow for updating products in carts to match the storefront. More information on this and these choices can be found in the section for Adherence to Object Oriented Design Principles. Below we have included a list of all controllers, and their functionalities.

#TODO write that list.

Services #TODO write this section

 Replace with your ViewModel Tier class diagram 1, etc.

Model Tier

Cart.java: Provides a template for the Cart resource. A cart contains an ID and products, and Cart.java defines getter functions to get these two pieces of information, as well as a toString() function.

Product.java: Provides a template for the Product resource. A product contains an ID, name, price and quantity. Within Product.java, the getter functions for this info is defined, along with functions to set the name, price and quantity of each product.

User.java: Provides a template for the User resource. A user contains an ID and a username, and User.java defines functions to get and set both of these fields for each User.

CurrentUser.java: Provides a template for a current user resource. This was necessary for database implementation due to some limitations of MongoDB.

Bid.java: Provides a template for bid items, storing a username and a bid amount. This was not strictly necessary but allowed for cleaner and more readable code. Easily maintainable and modifiable if bids ever need to hold more information.

AuctionItem.java: Provides a template for Auctions. It holds all necessary info including the end time, the product for auction, the current max bid, and the id of the auction. IDs were only included for potential future extensions to allow multiple auctions. The class also provides all necessary setters and getters.

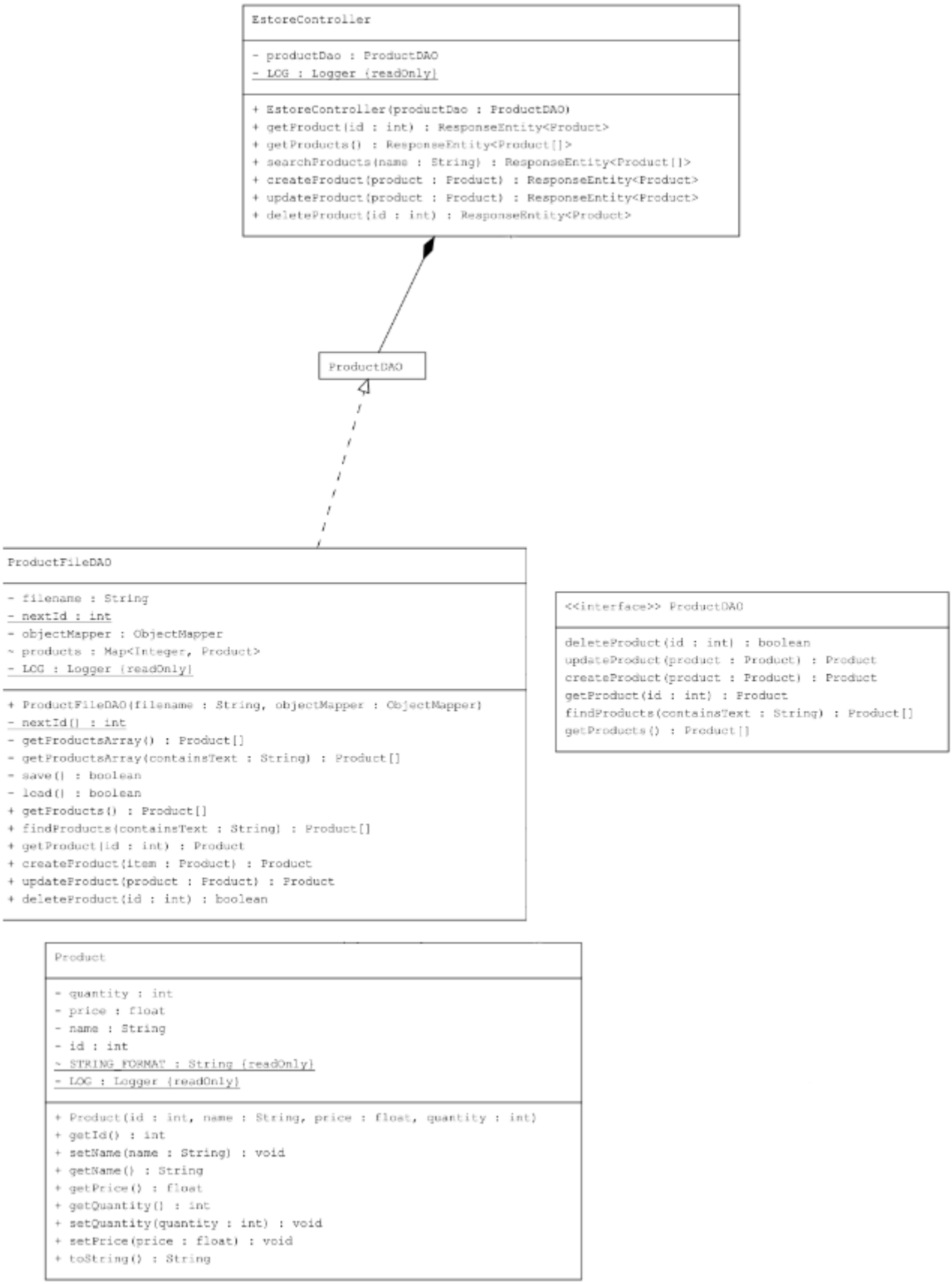
> *At appropriate places as part of this narrative provide **one** or more updated and **properly labeled** static models (UML class diagrams) with some details such as critical attributes and methods.*

 Replace with your Model Tier class diagram 1, etc.

OO Design Principles

Controller: Our system implementation uses a more complex controller setup compared to the last sprint. The controller concept essentially states that there should be some controller object that coordinates all system actions and operations, or in more complicated cases, multiple controller objects that coordinate all related actions and system operations. We have a few separate controller classes. First, we have the ProductController. This class responds to all admin API requests that relate to product management through various methods, one to handle each request. Within these methods, the controller makes calls to the other software layers, coordinating them to create the desired results. This shows the concept of Controller because we have one controller object that is acting to coordinate the system actions for each request. The following diagram

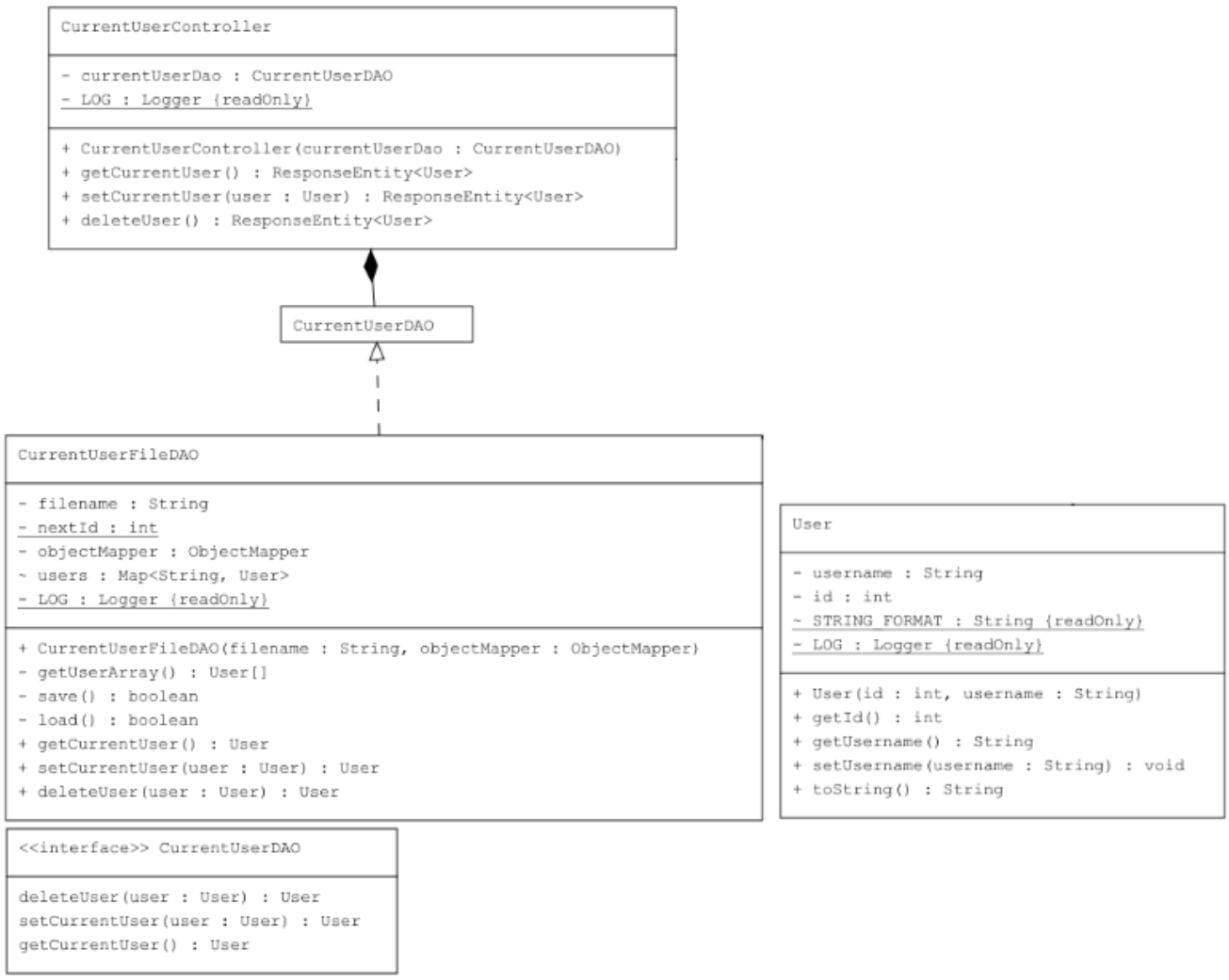
shows these relations well:



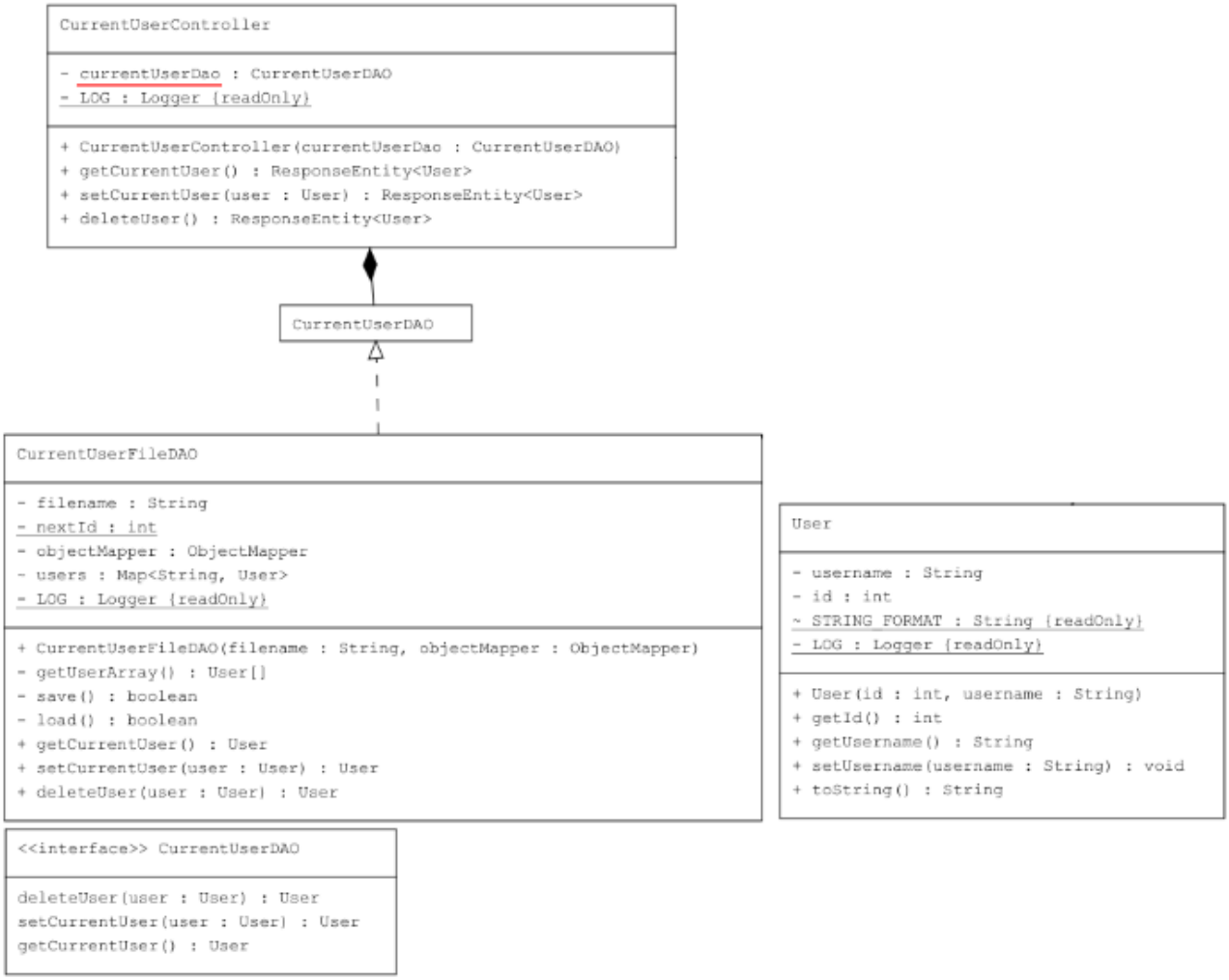
We also have similar controllers for managing user creation and access, cart creation, access, and modification, and a controller managing the current user. Each of these serves similar purposes for their respective domains. They handle API requests to the particular kind of object being manipulated. Diagrams for each of those are shown below.


CartController
- cartDao : CartDAO
- LOG : Logger {readOnly}
+ CartController(cartDao : CartDAO)
+ getCart(id : int) : ResponseEntity<Cart>
+ addCart(cart : Cart) : ResponseEntity<Cart>
+ addProductToCart(cId : int, product : Product) : ResponseEntity<Product>
+ updateProductCount(cId : int, pId : int, count : int) : ResponseEntity<Product>
+ removeProduct(cId : int, pId : int) : ResponseEntity<Product>
CartDAO
CartFileDAO
- filename : String
- nextId : int
- objectMapper : ObjectMapper
~ products : Map<Integer, Product>
- carts : Map<Integer, Cart>
- LOG : Logger {readOnly}
+ CartFileDAO(filename : String, objectMapper : ObjectMapper)
- nextId() : int
- getProductsArray(cId : int) : Product[]
- getCartsArray() : Cart[]
- save() : boolean
- load() : boolean
+ getCartProducts(cId : int) : Product[]
+ getProduct(cId : int, pId : int) : Product
+ getCart(cId : int) : Cart
+ addProduct(cId : int, item : Product) : Product
+ addCart(cart : Cart) : Cart
+ updateProductCount(cId : int, pId : int, count : int) : Product
+ removeProduct(cId : int, pId : int) : boolean
<<interface>> CartDAO
removeProduct(cId : int, pId : int) : boolean
updateProductCount(cId : int, pId : int, count : int) : Product
addCart(cart : Cart) : Cart
addProduct(cId : int, product : Product) : Product
getCart(cId : int) : Cart
getProduct(cId : int, pId : int) : Product
getCartProducts(cId : int) : Product[]
Cart
- products : HashMap<Integer, Product>
- id : int
~ STRING_FORMAT : String {readOnly}
+ Cart(id : int, products : HashMap<Integer, Product>)
+ getId() : int
+ getProducts() : HashMap<Integer, Product>
+ toString() : String

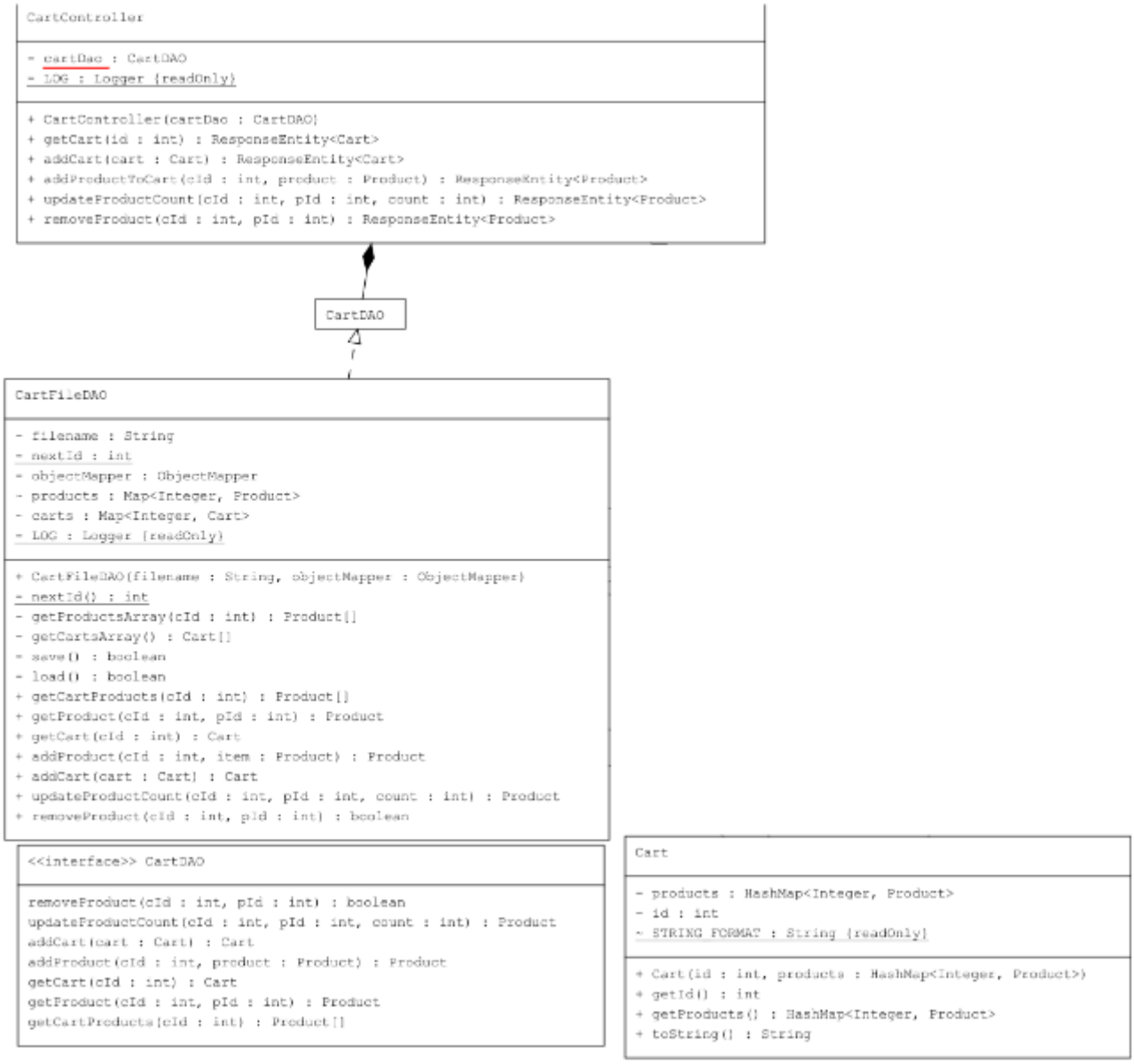

UserController
- userDao : UserDAO
- LOG : Logger {readOnly}
+ UserController(userDao : UserDAO)
+ getUser(username : String) : ResponseEntity<User>
+ createUser(user : User) : ResponseEntity<User>
UserDAO
UserFileDAO
- filename : String
- nextId : int
- objectMapper : ObjectMapper
~ users : Map<String, User>
- LOG : Logger {readOnly}
+ UserFileDAO(filename : String, objectMapper : ObjectMapper)
- nextId() : int
- getUserArray() : User[]
- save() : boolean
- load() : boolean
+ getUser(username : String) : User
+ createUser(user : User) : User
User
- username : String
- id : int
~ STRING_FORMAT : String {readOnly}
- LOG : Logger {readOnly}
+ User(id : int, username : String)
+ getId() : int
+ getUsername() : String
+ setUsername(username : String) : void
+ toString() : String
<<interface>> UserDAO
createUser(user : User) : User
getUser(username : String) : User

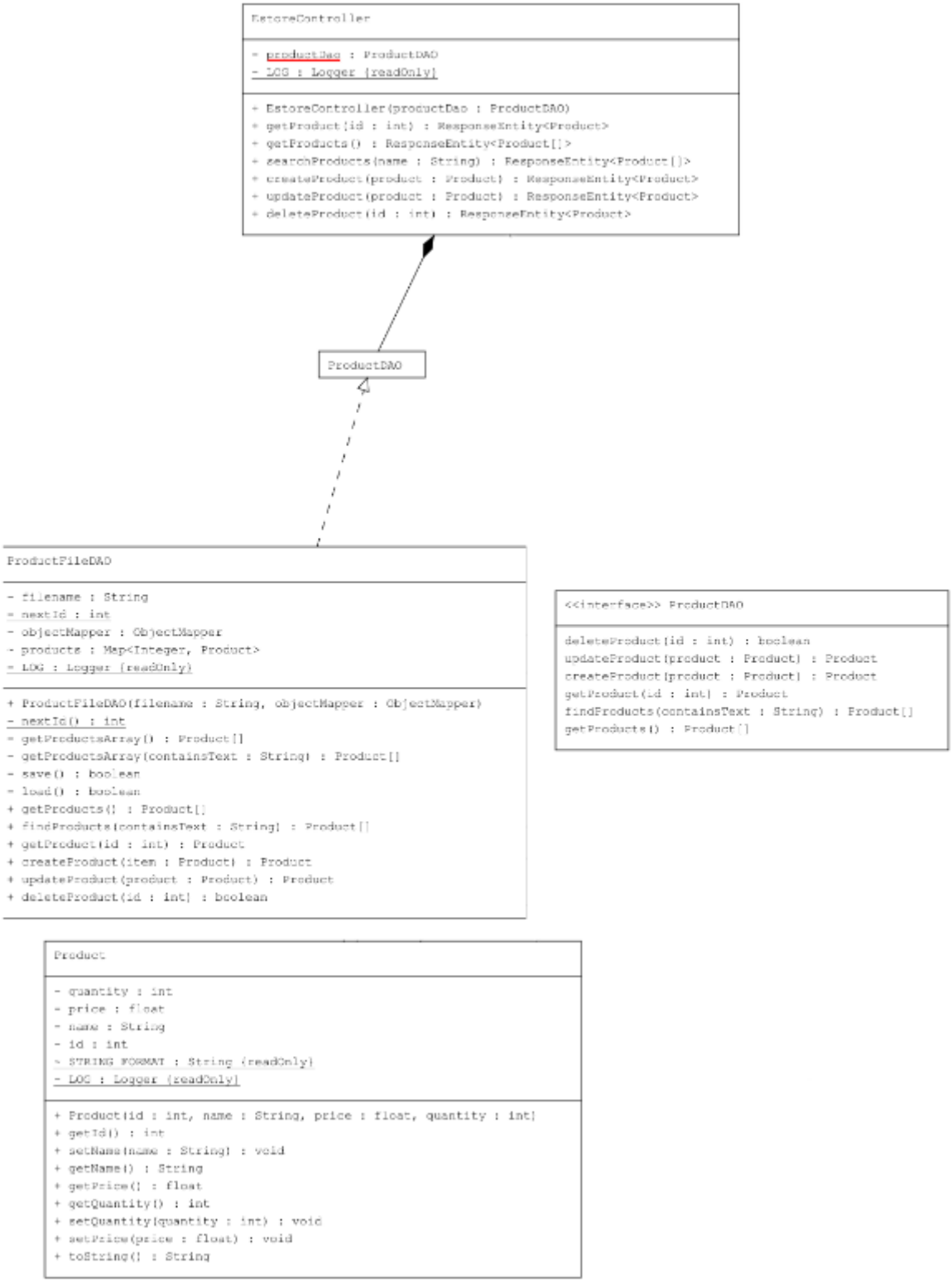


Injection: Our system implementation also utilizes the object-oriented concept of Injection. It utilized it in Sprint 1, but here it has become even more important. Injection is the concept of creating an object needed down the line higher up in the hierarchy and injecting that down by passing it in as a parameter. Our system uses that in quite a few places, notably, injection is used in each controller class. The respective DAO instance data of each controller is injected by the Spring Framework (This is underlined in red in the diagrams below, one for each controller). This means that the controllers only need to deal with the higher level abstraction of the DAOs without having to deal with the implementation specific DAOs. This is also good for our implementation, as we intend to eventually replace the FileDAOs we are currently using with DatabaseDAOs, and this will be easily possible by simply changing what type of thing the Spring Framework instantiates. In our typescript files our angular components take the user service files as injections in order to have them handle the backend calls for them. This shows how our other software layers, like our frontend angular app, also adhere to object oriented concepts.


CurrentUserController
- currentUserDao : CurrentUserDAO
- LOG : Logger {readOnly}
+ CurrentUserController(currentUserDao : CurrentUserDAO)
+ getCurrentUser() : ResponseEntity<User>
+ setCurrentUser(user : User) : ResponseEntity<User>
+ deleteUser() : ResponseEntity<User>
CurrentUserDAO
CurrentUserFileDAO
- filename : String
- nextId : int
- objectMapper : ObjectMapper
- users : Map<String, User>
- LOG : Logger {readOnly}
+ CurrentUserFileDAO(filename : String, objectMapper : ObjectMapper)
- getUserArray() : User[]
- save() : boolean
- load() : boolean
+ getCurrentUser() : User
+ setCurrentUser(user : User) : User
+ deleteUser(user : User) : User
User
- username : String
- id : int
~ STRING_FORMAT : String {readOnly}
- LOG : Logger {readOnly}
+ User(id : int, username : String)
+ getId() : int
+ getUsername() : String
+ setUsername(username : String) : void
+ toString() : String
<<interface>> CurrentUserDAO
deleteUser(user : User) : User
setCurrentUser(user : User) : User
getCurrentUser() : User

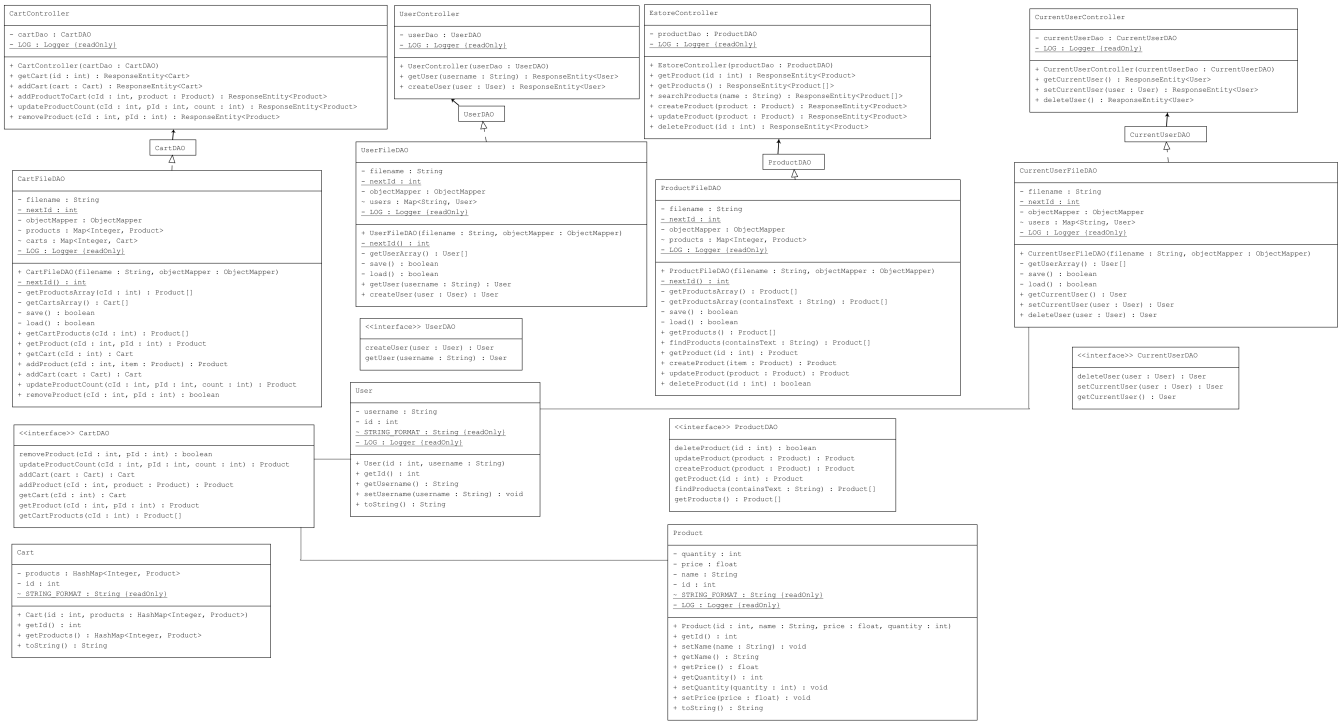

UserController
- userDao : UserDAO
- LOG : Logger {readOnly}
+ UserController(userDao : UserDAO)
+ getUser(username : String) : ResponseEntity<User>
+ createUser(user : User) : ResponseEntity<User>
UserDAO
UserFileDAO
- filename : String
- nextId : int
- objectMapper : ObjectMapper
- users : Map<String, User>
- LOG : Logger {readOnly}
+ UserFileDAO(filename : String, objectMapper : ObjectMapper)
- nextId() : int
- getUserArray() : User[]
- save() : boolean
- load() : boolean
+ getUser(username : String) : User
+ createUser(user : User) : User
User
- username : String
- id : int
- STRING_FORMAT : String {readOnly}
- LOG : Logger {readOnly}
+ User(id : int, username : String)
+ getId() : int
+ getUsername() : String
+ setUsername(username : String) : void
+ toString() : String
<<interface>> UserDAO
createUser(user : User) : User
getUser(username : String) : User


CartController
- cartDao : CartDAO
- LOG : Logger {readOnly}
+ CartController(cartDao : CartDAO)
+ getCart(id : int) : ResponseEntity<Cart>
+ addCart(cart : Cart) : ResponseEntity<Cart>
+ addProductToCart(cId : int, product : Product) : ResponseEntity<Product>
+ updateProductCount(cId : int, pId : int, count : int) : ResponseEntity<Product>
+ removeProduct(cId : int, pId : int) : ResponseEntity<Product>
CartDAO
CartFileDAO
- filename : String
- nextId : int
- objectMapper : ObjectMapper
- products : Map<Integer, Product>
- carts : Map<Integer, Cart>
- LOG : Logger {readOnly}
+ CartFileDAO(filename : String, objectMapper : ObjectMapper)
- nextId() : int
- getProductsArray(cId : int) : Product[]
- getCartsArray() : Cart[]
- save() : boolean
- load() : boolean
+ getCartProducts(cId : int) : Product[]
+ getProduct(cId : int, pId : int) : Product
+ getCart(cId : int) : Cart
+ addProduct(cId : int, item : Product) : Product
+ addCart(cart : Cart) : Cart
+ updateProductCount(cId : int, pId : int, count : int) : Product
+ removeProduct(cId : int, pId : int) : boolean
<<interface>> CartDAO
removeProduct(cId : int, pId : int) : boolean
updateProductCount(cId : int, pId : int, count : int) : Product
addCart(cart : Cart) : Cart
addProduct(cId : int, product : Product) : Product
getCart(cId : int) : Cart
getProduct(cId : int, pId : int) : Product
getCartProducts(cId : int) : Product[]
Cart
- products : HashMap<Integer, Product>
- id : int
- STRING_FORMAT : String {readOnly}
+ Cart(id : int, products : HashMap<Integer, Product>)
+ getId() : int
+ getProducts() : HashMap<Integer, Product>
+ toString() : String



Low Coupling: Low coupling refers to keeping the number of relations within a program to only those that are necessary. This means eliminating unnecessary or possibly redundant references. An example would be that the EstoreController file only imports the ProductDAO file and not both it and the ProductFileDAO as all the information that the controller would need from a ProductFileDAO could be obtained from the calls that it has to implement from the ProductDAO file. Another example is that the CartController only has a CartDAO, and not a ProductDAO. It is able to do everything it needs to do to manage carts without ever needing to understand what is actually in inventory. It has the minimal coupling to the Product object type, because it does need to know what a product is. This coupling is kept minimal by the frontend cleverly using the available information to keep the state consistent. Another example in our project is that any controllers that

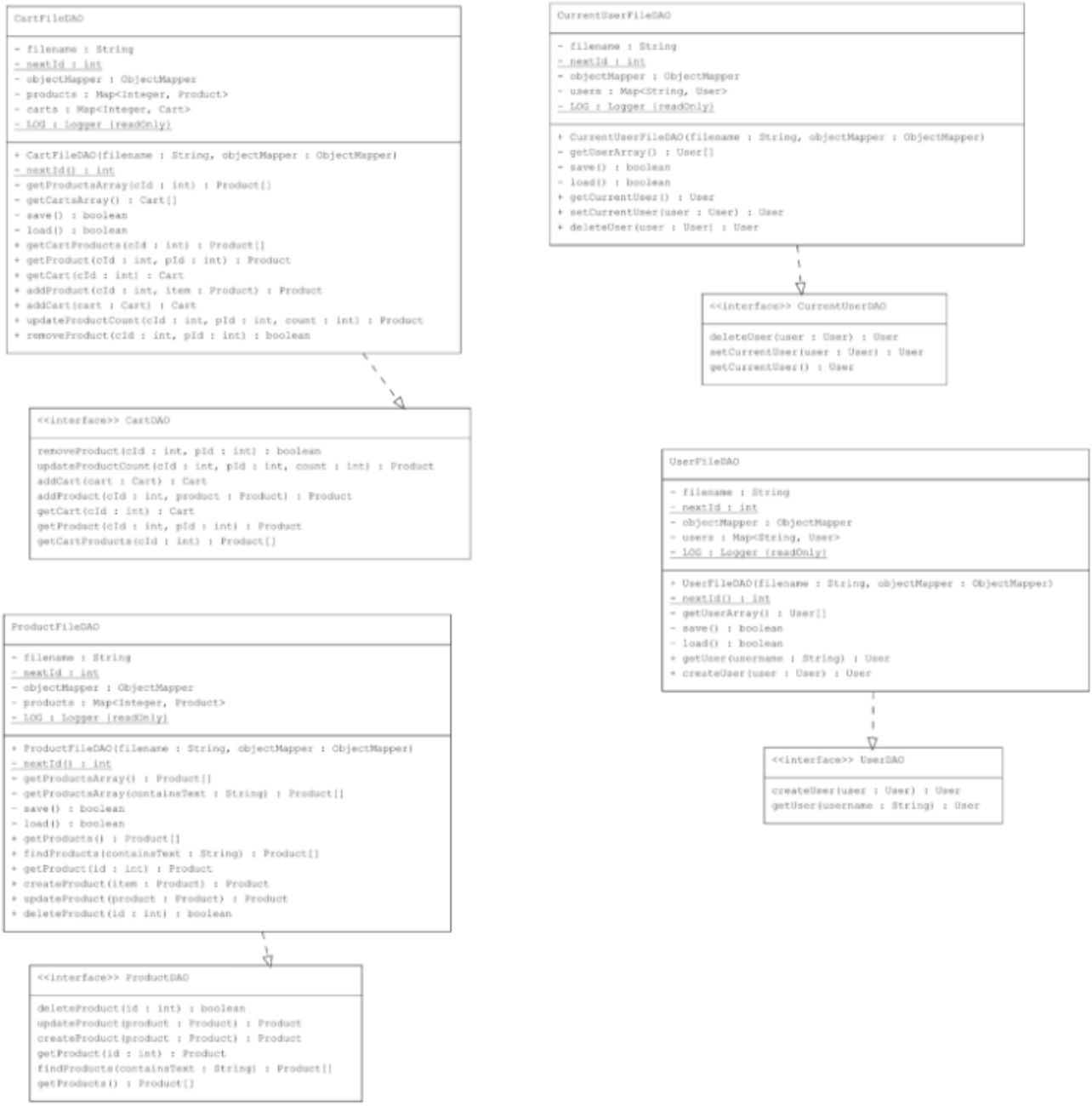
have to do with users do not use any other DAOs besides the UserDAOs. The user controller does not need to know what a product is or what a cart is at all, so there is no need to couple those objects. This is because the frontend can decide the logic of creating equal numbers of carts and users, the DAOs need only keep the ids consistent. A diagram illustrating the coupling between classes is included below.



As shown in the diagram, there is very minimal coupling between classes. The only coupling occurring is that which is absolutely necessary for functionality.

Open / Closed: The Open and Closed design principles specifically relate to appropriate use of abstract classes and interfaces such that software is “extendable” without the base functions being directly modified. Our code implements this in a few places throughout the backend. All of the implementations are fairly similar, and relate to the various DAOs. For example, the ProductDAO is an interface which any concrete implementation must implement. It acts as a blueprint for all necessary methods any ProductDAO must have to be functional. Similarly, the UserDAO, CurrentUserDAO, and CartDAO interfaces define what behaviors any concrete

implementations of them must have. This lets them be closed to modification, but open for easy extension.



> ***[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.*

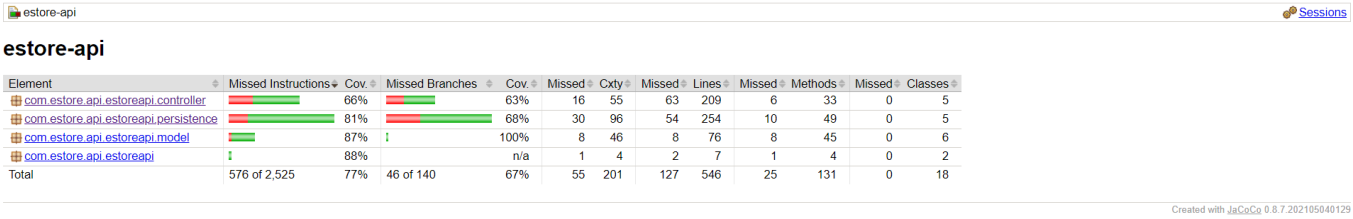
Acceptance Testing

Sprint 2: 19 stories complete including stories related to the cart and user authentication in both front end and back end. All stories have been tested and all have passed their acceptence criteria.

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.*

CartFileDAO has low branch coverage because tests were not created for one function within due to very low overall usage of the function.


estore-api

Sessions

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.controller		66%		63%	16	55	63	209	6	33	0	5
com.estore.api.estoreapi.persistence		81%		68%	30	96	54	254	10	49	0	5
com.estore.api.estoreapi.model		87%		100%	8	46	8	76	8	45	0	6
com.estore.api.estoreapi		88%		n/a	1	4	2	7	1	4	0	2
Total	576 of 2,525	77%	46 of 140	67%	55	201	127	546	25	131	0	18

Created with JaCoCo 0.8.7.202105040129