PROJECT Design Documentation

Team Information

Team name: Scrumblebees

- Team members
    - Ryan Webb
    - Kanisha Agrawal
    - Noah Landis
    - Priyank Patel

Executive Summary

Introducing Sorcerer's Supply Co. (development codename), a captivating web application that transports the wonderful world of magical commerce into the digital age. Sorcerer's Supply Co. 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, Sorcerer's Supply Co. 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 frontend, 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 backend, 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

Sorcerer's Supply Co. 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
DRY	Don't Repeat Yourself
admin flag	Property indicating whether a user is an admin or not

Requirements

This section describes the features of the application. 1)There should be a simple authentication system for both the admin and the users.

2)Users should be able to create an account and login / logout from the website.

3)Users should be able to see a list of products on the website and also be able to search for the products they need.

4)Users should have full control of the items in their cart and their quantities.

5)A user's data should be saved to the inventory so that the contents of their cart persist across multiple login sessions

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

Minimal Authentication for Customer/Owner Login & Logout

-> Epic:- Login Session and Registration -> As A user I want to be able to login/register on a separate page/window into the estore so that my shopping cart is persistent and I can checkout my cart

```
    Stories include:-

      1) Customer Registration [UI] (3)
              -> Story: As a Customer, I want to be able to register for a
Customer account so that I can buy Items.

      2) Customer Login [UI] (3)
              -> Story: As a Customer, I want to be able to log into my Customer
account so that I can buy Items.

      3) Administrator Login [UI] (3)
              -> Story: As an Administrator, I want to be able to log into my
Administrator account so that I can manage the website.

      4) Login User [API] (5)
              -> Story: As a Developer, I want to submit a request to login with a
username so that I can create a login session for the user.

      5) LogOut User [API] (5)
              -> Story:  As a Developer, I want to submit a request to logout so
that I can create a logout session for the user.

      6) Register User [API] (7)
              -> Story: As a Developer, I want to submit a request to register a
new user so that the user will have an account and can log in.
```

```
      7) Get Single User [API: Controller] (2)
              -> Story: As a Developer, I want to submit a request for a single
user by their username so that I can access that user’s data.
```

Customer Functionality & Data Persistence

-> Epic:- Shopping cart -> As a Customer I want to add items to my Shopping Cart so that I can build an order.

```
    Stories include:-

        1) Add to shopping cart [UI] (3)
              -> Story: As a Customer, I want to add items to my shopping cart so
that I can choose Items to purchase.

        2) Remove from shopping cart [UI] (3)
              -> Story: As a Customer, I want to remove items from my Shopping
Cart so that I can choose not to purchase it.

        3) Shopping cart persistence [API: Persistence] (3)
              -> Story: As a Customer, I want my Shopping Cart to be saved to my
account so that I can log in and out as I please.
```

-> Epic:- Browse Items -> As a Customer, I want to browse available items so that I can decide what to purchase.

```
    Stories include:-

        1) View Catalog [UI] (3)
              -> Story: As a Customer, I want to view the entire catalog so that I
can choose what to purchase.

        2) View Specific Product Details [UI] (5)
              -> Story: As a Customer, I want to view an individual item’s listing
to learn more details of the item

        3) Search Items [UI] (3)
              -> Story: As a Customer, I want to use a search bar so that I can
search for specific Items.
```

-> Story: Search for a product [API] (5) -> Story: As a Developer, I want to submit a request to get the products in the inventory whose name contains the given text SO THAT I have access to only those products.

-> Story: Get a Single Product [API] (3) -> Story: As a Developer, I want to submit a request to get a single product so that I can access the price and quantity.

-> Story: Get entire inventory [API] (5) -> Story: As a Developer, I want to submit a request to get the entire inventory so that I have access to all of the products.

-> Epic:- Checkout -> As a Customer, I want to browse available Items so that I can decide what to purchase.

```
    Stories include:-

        1) OrdersDAO & OrdersFileDAO [API: Persistence] (5)
            -> Story: As a Developer, I want to be able to interact with Orders
resources so that a customer can create, edit, and view their order.

        2) Create/Read Single Order [API: Controller] (2)
            -> Story: As a Developer, I want to be able to interact with Orders
resources so that a customer can create, edit, and view their order.

        3) Checkout [UI] (8)
             -> Story: As a Customer, I want to check out so that I can purchase
the items in my cart.
```

Inventory Management

-> Epic:- Inventory Management [UI] -> As an Administrator, I want to manage my inventory to keep my product listings up to date.

```
    Stories include:-

        1) Add Product to Listing [UI] (2)
            -> Story: As an Administrator, I want to add new Items to the
inventory so that I can expand my Item listings.

        2) Remove Product Listing [UI] (2)
            -> Story: As an Administrator, I want to remove products from the
inventory so that I can consolidate my Item listings.

        3) Update Product Details [UI] (2)
            -> Story: As an owner, I want a button to update the product
description for a products so that I can update the description of the products.
```

-> Story: Create new Product [API] (8) -> Story: AS a Developer, I want to submit a request to create a new product (name, price, quantity) so that it is available to in the inventory.

-> Story: Delete a Single Product [API] (3) -> Story: As a Developer, I want to submit a request to get a single product so that I can access the price and quantity.

-> Story: Update a product [API] (5) -> Story: As a Developer, I want to submit a request to get the entire inventory so that I have access to all products and their details.

10% feature enchantment(s)

-> Epic:- Ratings and Reviews -> As a customer, I want to give a product a review and a star rating. -> As an Administrator and a Customer, I want to delete a review for an Item.

```
    Stories include:-

        1) Create a Single Review [API] (5)
            -> Story: As a developer, I want to submit a DELETE HTTP request to
delete a single review so that I can add fulfill a user’s request to remove their
review.

        2) Get all reviews for a single SKU [API] (5)
            -> Story: As a developer, I want to make an HTTP GET request to the
reviews resource using a SKU so that I can get all reviews that have been left for
a given product.

        3) Delete a single review [API] (5)
            -> Story: As a developer, I want to submit a POST HTTP request to
create a single review so that I can add a user’s review for an item.

        4) View Reviews [UI] (8)
            -> Story: As a Customer, I want to write a review and give a star
rating so that I can share my current level of satisfaction.

        5) Delete Review [Customer] (5)
            -> Story: As a Customer, I want to delete my reviews so that I have
control over my voice.

        6) Delete Review [Administrator] (5)
            -> Story: As the estore owner, I want to delete the reviews left by
customers so I can maintain a positive environment in the estore owner.
```

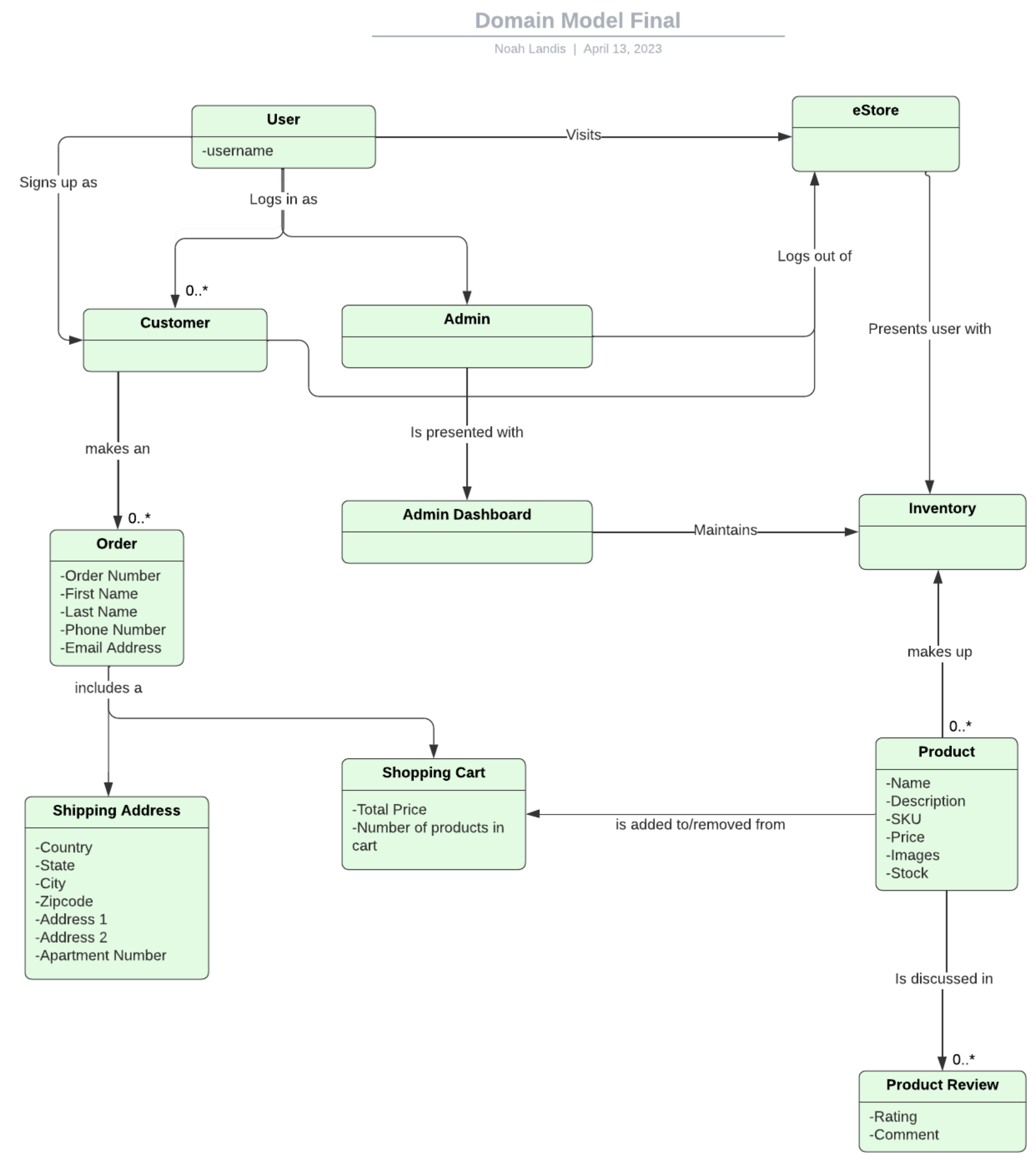
Enhancements

Review System

This feature allows users to view the reviews of a given product. Reviews connsists of a 1-5 star rating, along with an optional comment. If the user is logged in as a customer, they are able to leave one review per product. Customers are able to delete their reviews. Admins have the ability to delete any review.

Application Domain

This section describes the application domain.



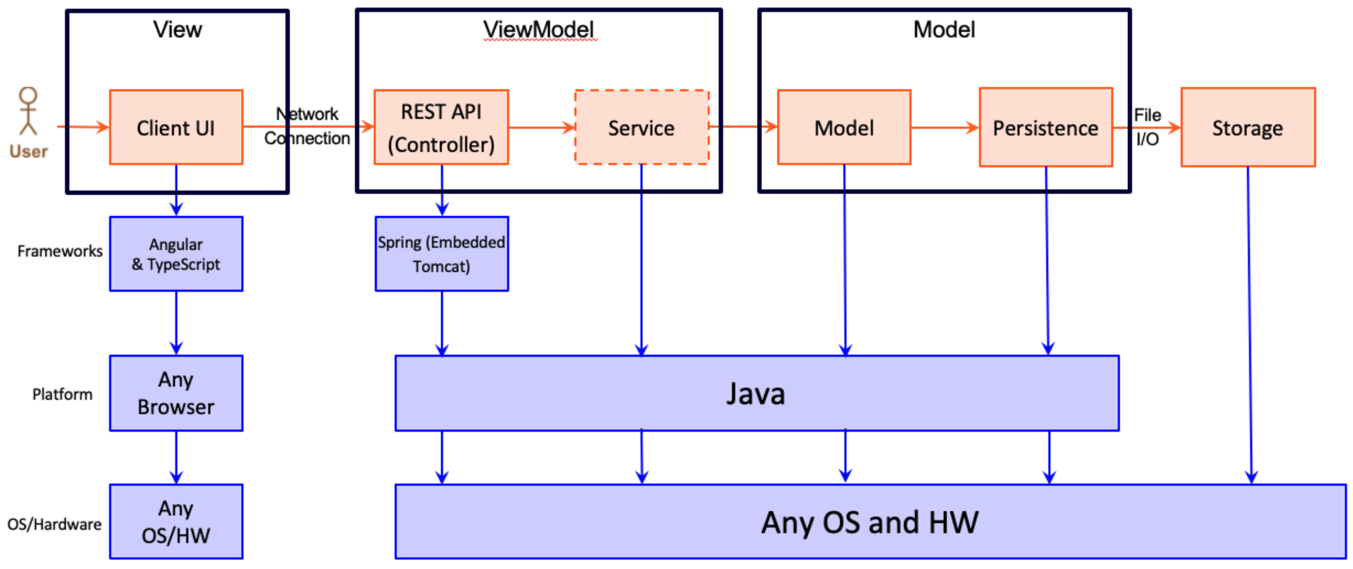
The major relationships in this are admin-dashboard-inventory, which represents the relationship between the admin and how the admin manages the inventory. Another one is customer-admin-user, which shows the hierarchy of users. We added orders and product reviews.

Architecture and Design

This section describes the application architecture.

Summary

The following Tiers/Layers model shows a high-level view of the web app'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 presented with the site's logo and a 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 on the website, users can browse our products, which are presented in a grid format. They can 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.

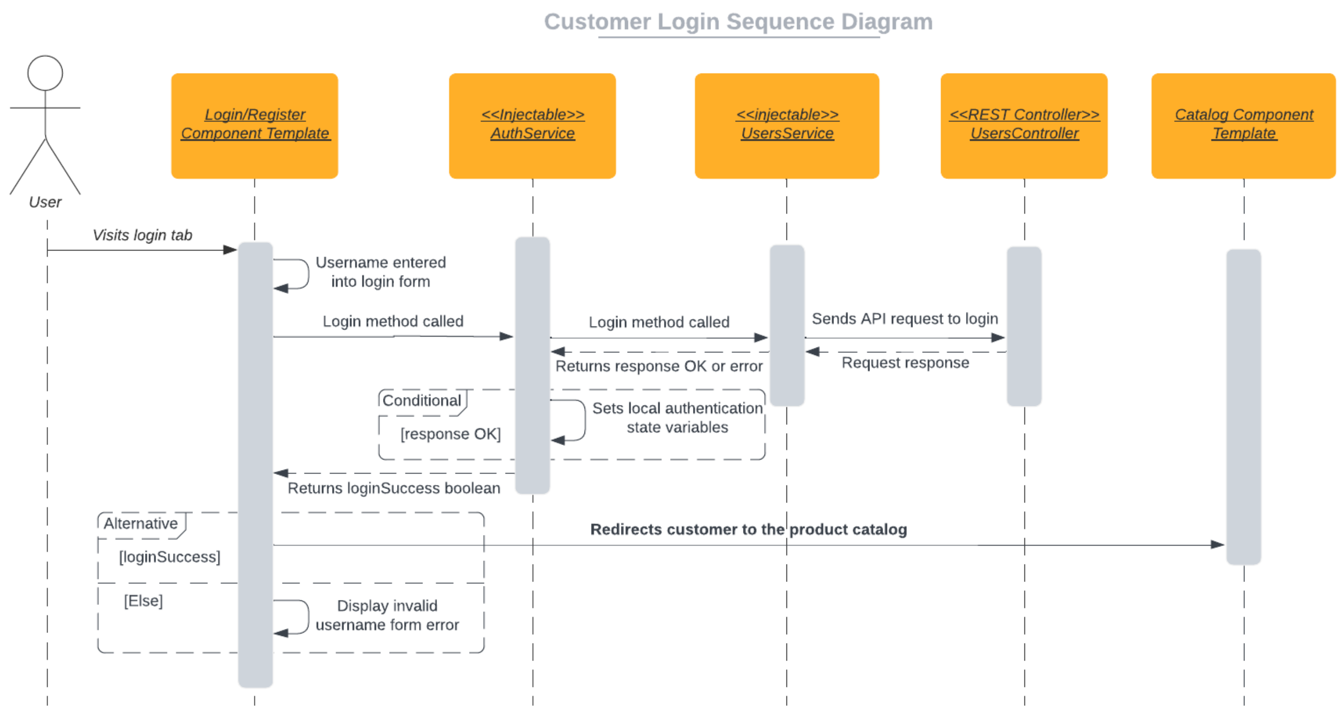
Our search bar is located in the header which helps users find the exact product they're looking for.

When a user is ready to checkout with their cart, they can navigate to the cart view, then click the checkout button. In the checkout process, they will be guided through the checkout steps such as providing contact, shipping, and payment information. After reviewing their order and clicking order, they will be presented with an order confirmation which includes their order number, and a navigation link to return to the catalog.

View Tier

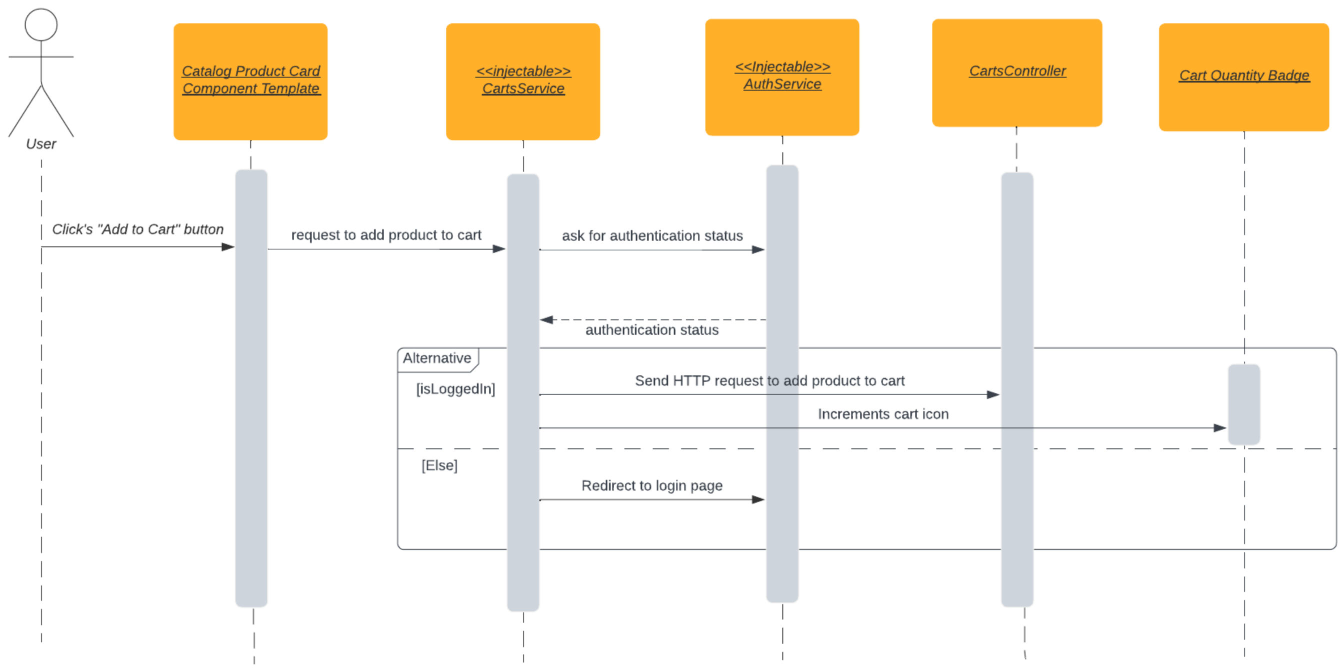
Our view tier includes a lot of components that renders the user interface and make the website user-friendly. The login registration component renders the login and registeration forms to the user. The catalog component is used to render product cards that contain the product on the website. It provides a way for customers to purchase and search for products. The catalog product card component displays a single product, shows the stock of the product and allows the user to add products to the . Clicking on the catalog product card brings the user to the details for that product. The product details component renders the reviews of the products and their details.

One of the features of our e-store is the login system built for the customers and the admin. The following sequence diagram shows the flow of the login system.


Customer Login Sequence Diagram
User
Login/Register Component Template
<<Injectable>> AuthService
<<injectable>> UsersService
<<REST Controller>> UsersController
Catalog Component Template
Visits login tab
Username entered into login form
Login method called
Login method called
Sends API request to login
Returns response OK or error
Request response
Conditional
[response OK]
Sets local authentication state variables
Returns loginSuccess boolean
Alternative
[loginSuccess]
Redirects customer to the product catalog
[Else]
Display invalid username form error

Another feature that we implemented was that the user can add products to their cart which is persistent. The following sequence diagram shows the flow of the checkout system.

Add to Cart Sequence Diagram


User
Catalog Product Card Component Template
<<injectable>> CartsService
<<Injectable>> AuthService
CartsController
Cart Quantity Badge
Click's "Add to Cart" button
request to add product to cart
ask for authentication status
authentication status
Alternative
[isLoggedIn]
Send HTTP request to add product to cart
Increments cart icon
[Else]
Redirect to login page

Our View tier includes a number of services that isolate behavior involving fetching data from each of our backend API resources. These data fetching services include `UsersService`, `CartsService`, `InventoryService`, and `OrdersService`. We also have a few supporting services which do not directly call backend API endpoints, such as `AuthService`, `UpdateService`, and `MessageService`.

These services serve as a communication layer between our feature components and the backend model tier.

`AuthService` is the service that handles the login and registration of the users including the admin. It oversees the authentication of the users and the admin. It also handles the registration of customers. It does so using the UsersService.This service is used throughout the application to render components based on the user's role.

The `UsersService` handles the CRUD operations of the users. It is used by the AuthService to register new users and by the AdminDashboardComponent to manage the users. It makes the actual calls to the API endpoints for login, logout, register, and get user.

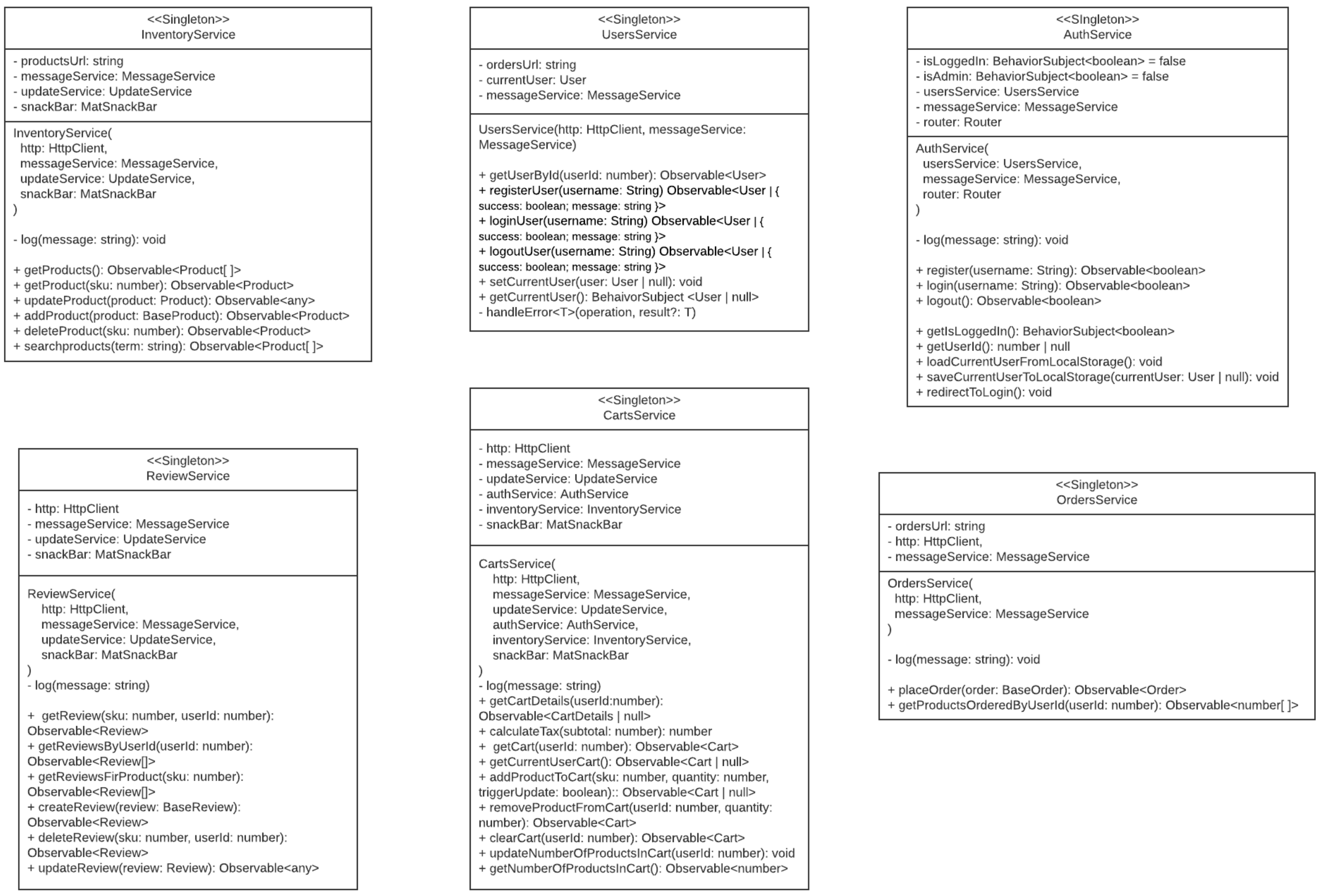
The `CartsService` makes the API calls related to the carts. It makes the calls to the API endpoints for get the cart, add to cart, and remove from cart when the user is logged in as a customer. It is used when a user is logged in to render components based on the user's cart.

The calls to the Inventory API are made through the `InventoryService`. These include method calls to create, update, and delete products. It also makes the calls to get all products and get a single product when a user is logged in as admin. It is used by the catalog components of the admin and the user to manage the products.

When the customer tries to checkout, the `OrdersService` is used to make the API calls to the Orders API. It makes the calls to the API endpoints for create order, get order, and get all orders.

The `UpdateService` is used to update the data in the application. It is used to update the view of the cart, catalog and reviews of the products when a change is made.

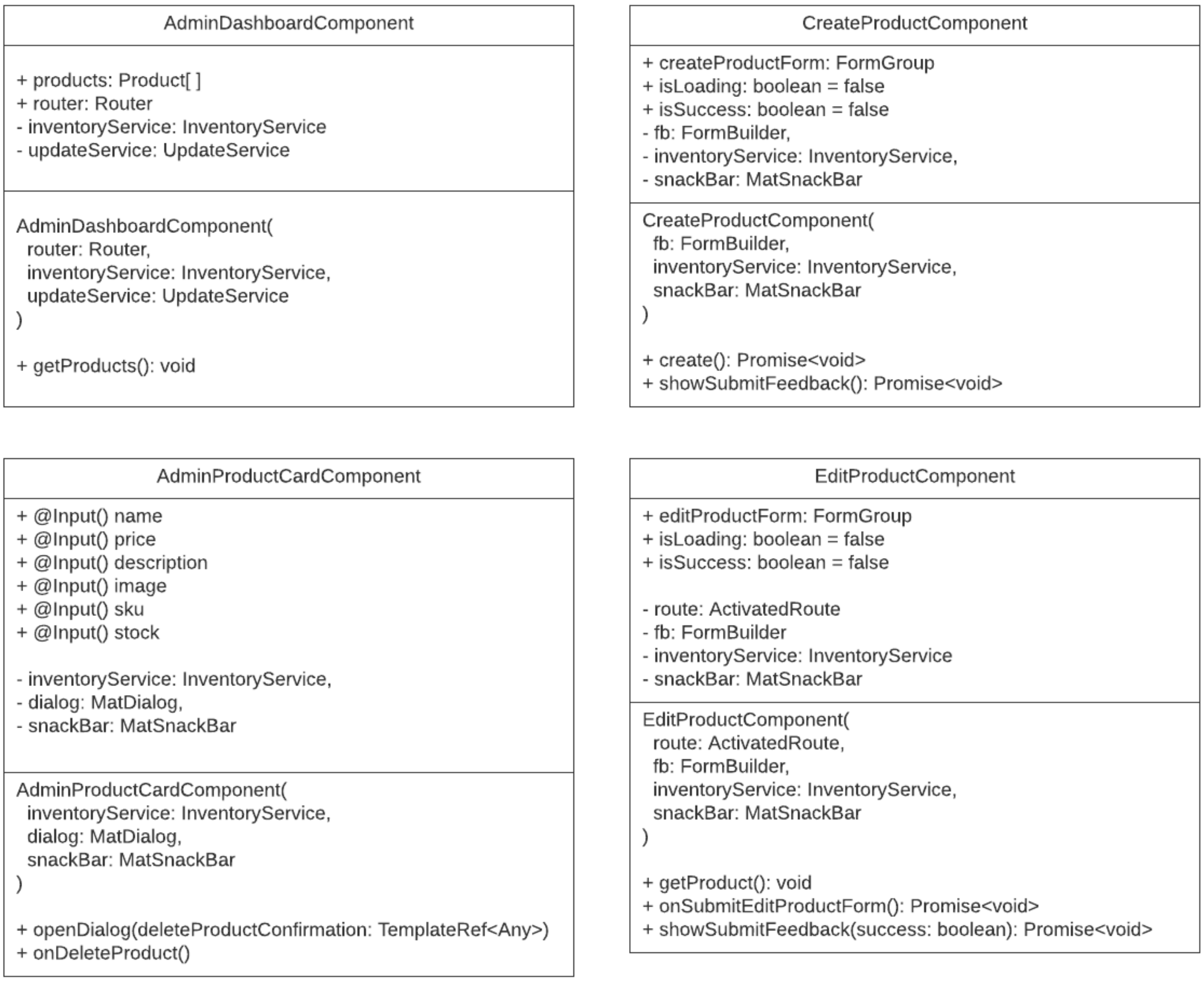
Services


<<Singleton>>
InventoryService
- productsUrl: string
- messageService: MessageService
- updateService: UpdateService
- snackBar: MatSnackBar
InventoryService(
http: HttpClient,
messageService: MessageService,
updateService: UpdateService,
snackBar: MatSnackBar
)
- log(message: string): void
+ getProducts(): Observable<Product[ ]>
+ getProduct(sku: number): Observable<Product>
+ updateProduct(product: Product): Observable<any>
+ addProduct(product: BaseProduct): Observable<Product>
+ deleteProduct(sku: number): Observable<Product>
+ searchproducts(term: string): Observable<Product[ ]>
<<Singleton>>
UsersService
- ordersUrl: string
- currentUser: User
- messageService: MessageService
UsersService(http: HttpClient, messageService: MessageService)
+ getUserById(userId: number): Observable<User>
+ registerUser(username: String) Observable<User | { success: boolean; message: string }>
+ loginUser(username: String) Observable<User | { success: boolean; message: string }>
+ logoutUser(username: String) Observable<User | { success: boolean; message: string }>
+ setCurrentUser(user: User | null): void
+ getCurrentUser(): BehaivorSubject <User | null>
- handleError<T>(operation, result?: T)
<<SIngleton>>
AuthService
- isLoggedIn: BehaviorSubject<boolean> = false
- isAdmin: BehaviorSubject<boolean> = false
- usersService: UsersService
- messageService: MessageService
- router: Router
AuthService(
usersService: UsersService,
messageService: MessageService,
router: Router
)
- log(message: string): void
+ register(username: String): Observable<boolean>
+ login(username: String): Observable<boolean>
+ logout(): Observable<boolean>
+ getIsLoggedIn(): BehaviorSubject<boolean>
+ getUserId(): number | null
+ loadCurrentUserFromLocalStorage(): void
+ saveCurrentUserToLocalStorage(currentUser: User | null): void
+ redirectToLogin(): void
<<Singleton>>
CartsService
- http: HttpClient
- messageService: MessageService
- updateService: UpdateService
- authService: AuthService
- inventoryService: InventoryService
- snackBar: MatSnackBar
CartsService(
http: HttpClient,
messageService: MessageService,
updateService: UpdateService,
authService: AuthService,
inventoryService: InventoryService,
snackBar: MatSnackBar
)
- log(message: string)
+ getCartDetails(userId:number): Observable<CartDetails | null>
+ calculateTax(subtotal: number): number
+ getCart(userId: number): Observable<Cart>
+ getCurrentUserCart(): Observable<Cart | null>
+ addProductToCart(sku: number, quantity: number, triggerUpdate: boolean):: Observable<Cart | null>
+ removeProductFromCart(userId: number, quantity: number): Observable<Cart>
+ clearCart(userId: number): Observable<Cart>
+ updateNumberOfProductsInCart(userId: number): void
+ getNumberOfProductsInCart(): Observable<number>
<<Singleton>>
ReviewService
- http: HttpClient
- messageService: MessageService
- updateService: UpdateService
- snackBar: MatSnackBar
ReviewService(
http: HttpClient,
messageService: MessageService,
updateService: UpdateService,
snackBar: MatSnackBar
)
- log(message: string)
+ getReview(sku: number, userId: number): Observable<Review>
+ getReviewsByUserId(userId: number): Observable<Review[]>
+ getReviewsFirProduct(sku: number): Observable<Review[]>
+ createReview(review: BaseReview): Observable<Review>
+ deleteReview(sku: number, userId: number): Observable<Review>
+ updateReview(review: Review): Observable<any>
<<Singleton>>
OrdersService
- ordersUrl: string
- http: HttpClient,
- messageService: MessageService
OrdersService(
http: HttpClient,
messageService: MessageService
)
- log(message: string): void
+ placeOrder(order: BaseOrder): Observable<Order>
+ getProductsOrderedByUserId(userId: number): Observable<number[ ]>

Various components use these services to interact with the backend APIs. We developed these components in different modules to organize the structure and the code to make it easier to maintain. The following diagram shows the components that we built for each of the feature modules that we created.

These are the components built under the admin feature module. It contains components to render the admin dashboard, catalog and to create and edit inventory items.

Admin Feature



```
AdminDashboardComponent
---
+ products: Product[ ]
+ router: Router
- inventoryService: InventoryService
- updateService: UpdateService
---
AdminDashboardComponent(
  router: Router,
  inventoryService: InventoryService,
  updateService: UpdateService
)

+ getProducts(): void
```

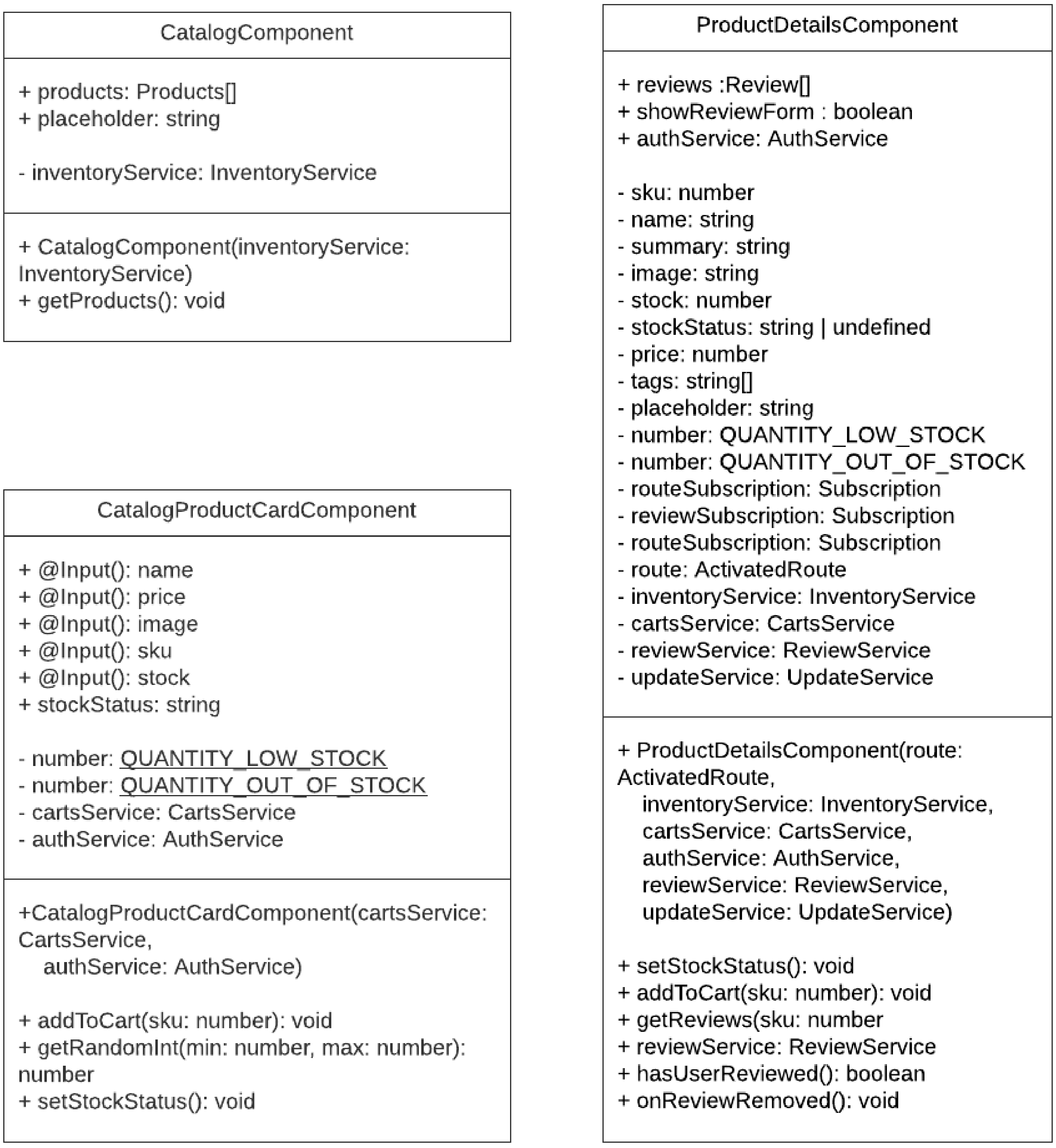
```
CreateProductComponent
---
+ createProductForm: FormGroup
+ isLoading: boolean = false
+ isSuccess: boolean = false
- fb: FormBuilder,
- inventoryService: InventoryService,
- snackBar: MatSnackBar
---
CreateProductComponent(
  fb: FormBuilder,
  inventoryService: InventoryService,
  snackBar: MatSnackBar
)

+ create(): Promise<void>
+ showSubmitFeedback(): Promise<void>
```

```
AdminProductCardComponent
---
+ @Input() name
+ @Input() price
+ @Input() description
+ @Input() image
+ @Input() sku
+ @Input() stock

- inventoryService: InventoryService,
- dialog: MatDialog,
- snackBar: MatSnackBar
---
AdminProductCardComponent(
  inventoryService: InventoryService,
  dialog: MatDialog,
  snackBar: MatSnackBar
)

+ openDialog(deleteProductConfirmation: TemplateRef<Any>)
+ onDeleteProduct()
```

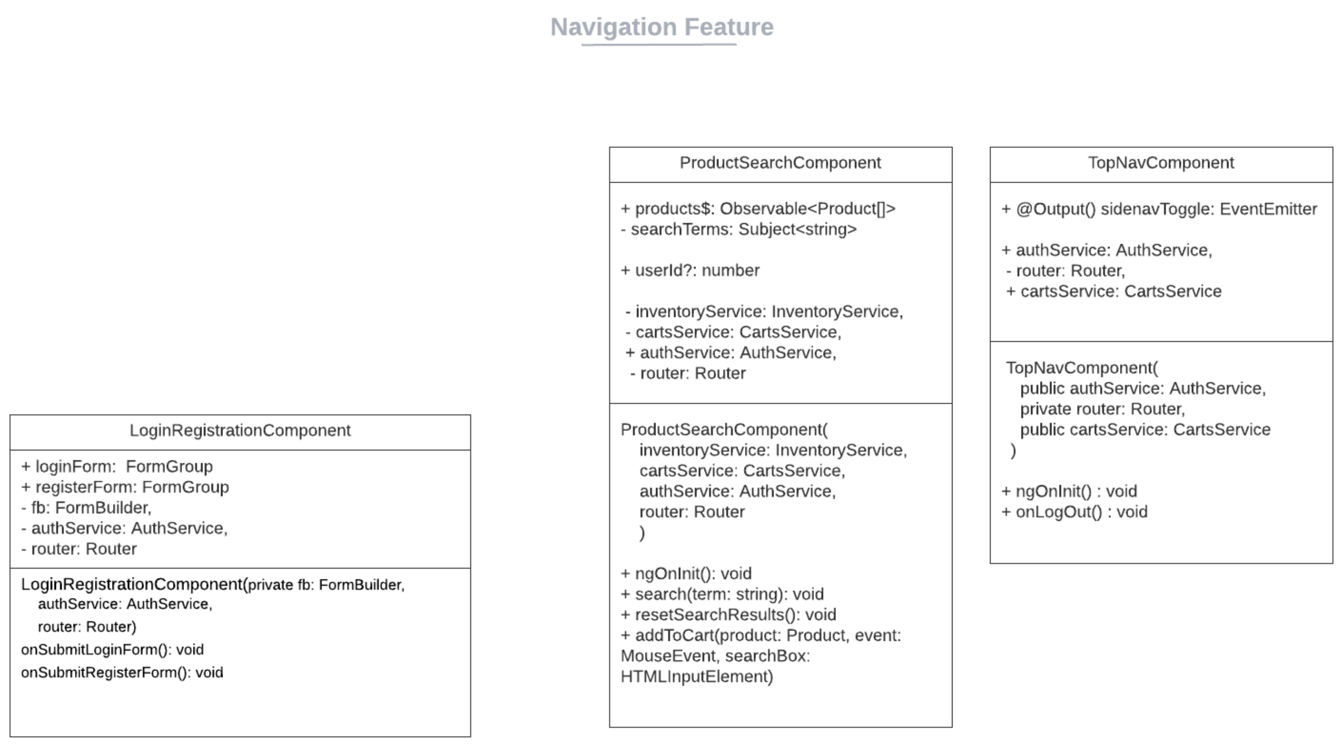
```
EditProductComponent
---
+ editProductForm: FormGroup
+ isLoading: boolean = false
+ isSuccess: boolean = false

- route: ActivatedRoute
- fb: FormBuilder
- inventoryService: InventoryService
- snackBar: MatSnackBar
---
EditProductComponent(
  route: ActivatedRoute,
  fb: FormBuilder,
  inventoryService: InventoryService,
  snackBar: MatSnackBar
)

+ getProduct(): void
+ onSubmitEditProductForm(): Promise<void>
+ showSubmitFeedback(success: boolean): Promise<void>
```

These are the components built under the browse catalog feature module. It contains components to render the catalog and the product details to the customer.

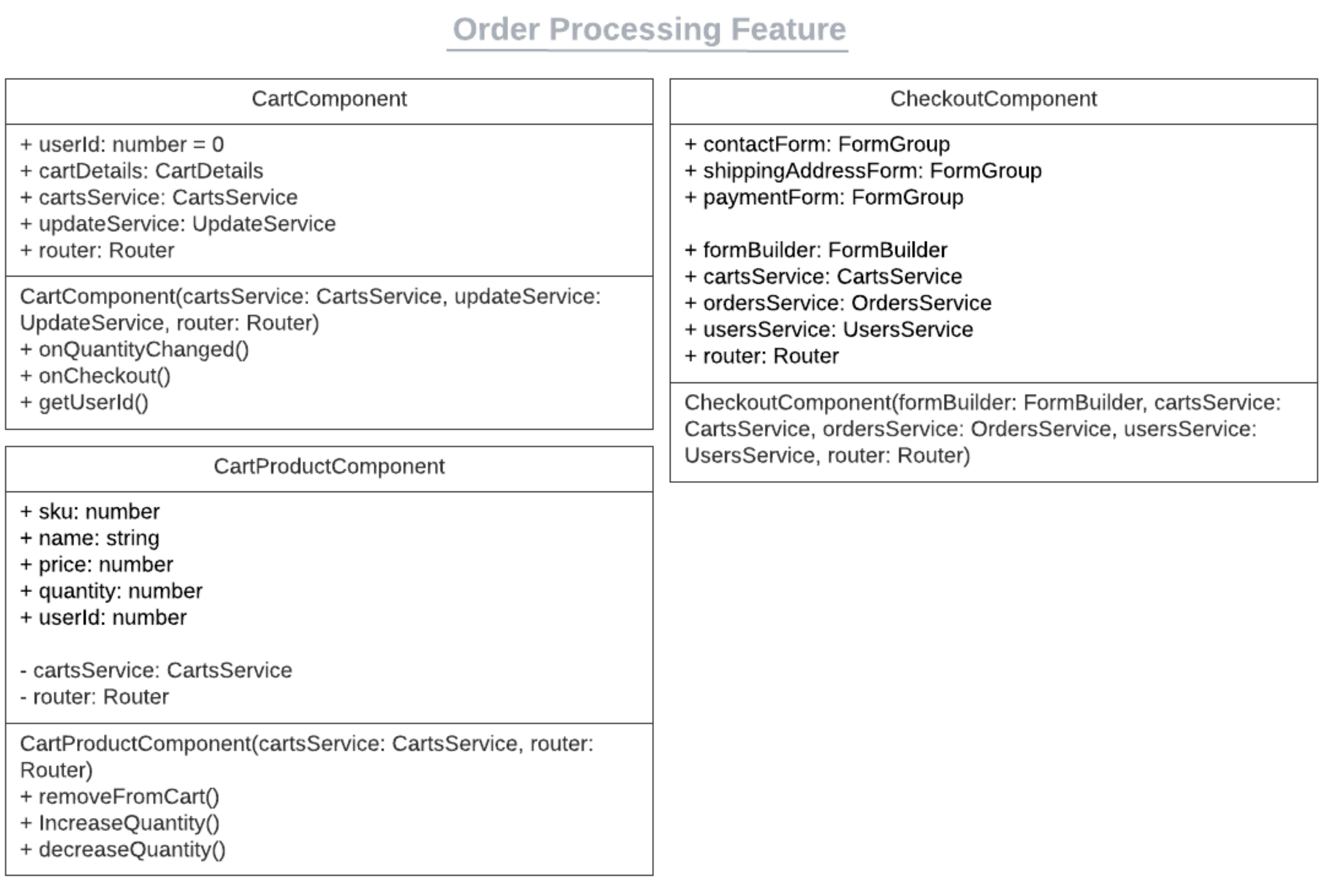
Browse Catalog Feature


CatalogComponent
+ products: Products[]
+ placeholder: string
- inventoryService: InventoryService
+ CatalogComponent(inventoryService: InventoryService)
+ getProducts(): void
CatalogProductCardComponent
+ @Input(): name
+ @Input(): price
+ @Input(): image
+ @Input(): sku
+ @Input(): stock
+ stockStatus: string
- number: QUANTITY_LOW_STOCK
- number: QUANTITY_OUT_OF_STOCK
- cartsService: CartsService
- authService: AuthService
+CatalogProductCardComponent(cartsService: CartsService, authService: AuthService)
+ addToCart(sku: number): void
+ getRandomInt(min: number, max: number): number
+ setStockStatus(): void
ProductDetailsComponent
+ reviews :Review[]
+ showReviewForm : boolean
+ authService: AuthService
- sku: number
- name: string
- summary: string
- image: string
- stock: number
- stockStatus: string | undefined
- price: number
- tags: string[]
- placeholder: string
- number: QUANTITY_LOW_STOCK
- number: QUANTITY_OUT_OF_STOCK
- routeSubscription: Subscription
- reviewSubscription: Subscription
- routeSubscription: Subscription
- route: ActivatedRoute
- inventoryService: InventoryService
- cartsService: CartsService
- reviewService: ReviewService
- updateService: UpdateService
+ ProductDetailsComponent(route: ActivatedRoute, inventoryService: InventoryService, cartsService: CartsService, authService: AuthService, reviewService: ReviewService, updateService: UpdateService)
+ setStockStatus(): void
+ addToCart(sku: number): void
+ getReviews(sku: number
+ reviewService: ReviewService
+ hasUserReviewed(): boolean
+ onReviewRemoved(): void

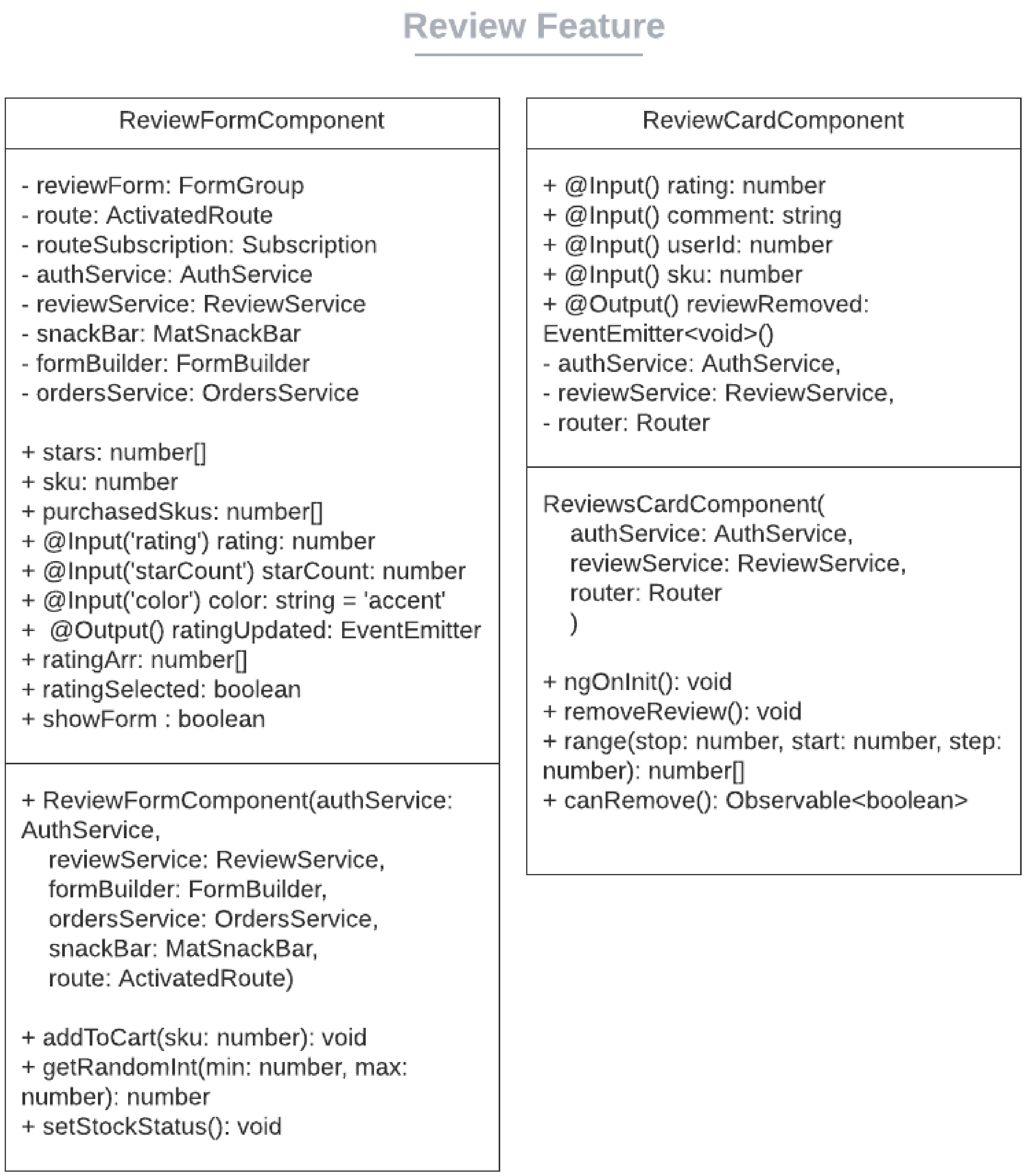
A user is able to navigate through the different pages of the website using the navigation bar. It contains links to the home page, the catalog, the cart and the login page.


Navigation Feature
ProductSearchComponent
+ products$: Observable<Product[]>
- searchTerms: Subject<string>
+ userId?: number
- inventoryService: InventoryService,
- cartsService: CartsService,
+ authService: AuthService,
- router: Router
ProductSearchComponent(
inventoryService: InventoryService,
cartsService: CartsService,
authService: AuthService,
router: Router
)
+ ngOnInit(): void
+ search(term: string): void
+ resetSearchResults(): void
+ addToCart(product: Product, event: MouseEvent, searchBox: HTMLInputElement)
TopNavComponent
+ @Output() sidenavToggle: EventEmitter
+ authService: AuthService,
- router: Router,
+ cartsService: CartsService
TopNavComponent(
public authService: AuthService,
private router: Router,
public cartsService: CartsService
)
+ ngOnInit() : void
+ onLogOut() : void
LoginRegistrationComponent
+ loginForm: FormGroup
+ registerForm: FormGroup
- fb: FormBuilder,
- authService: AuthService,
- router: Router
LoginRegistrationComponent(private fb: FormBuilder,
authService: AuthService,
router: Router)
onSubmitLoginForm(): void
onSubmitRegisterForm(): void

One of the features of the website is that the customer can add products to their cart and also proceed to checkout until the order is placed. The following diagram shows the components that are used to render the cart and the checkout page.


Order Processing Feature
CartComponent
+ userId: number = 0
+ cartDetails: CartDetails
+ cartsService: CartsService
+ updateService: UpdateService
+ router: Router
CartComponent(cartsService: CartsService, updateService: UpdateService, router: Router)
+ onQuantityChanged()
+ onCheckout()
+ getUserId()
CheckoutComponent
+ contactForm: FormGroup
+ shippingAddressForm: FormGroup
+ paymentForm: FormGroup
+ formBuilder: FormBuilder
+ cartsService: CartsService
+ ordersService: OrdersService
+ usersService: UsersService
+ router: Router
CheckoutComponent(formBuilder: FormBuilder, cartsService: CartsService, ordersService: OrdersService, usersService: UsersService, router: Router)
CartProductComponent
+ sku: number
+ name: string
+ price: number
+ quantity: number
+ userId: number
- cartsService: CartsService
- router: Router
CartProductComponent(cartsService: CartsService, router: Router)
+ removeFromCart()
+ IncreaseQuantity()
+ decreaseQuantity()

The customers can also leave a review on a product, which was our feature enhancement. The following diagram shows the components that are used to render the review form and the reviews.



```
ReviewFormComponent
--------------------------------------------------
- reviewForm: FormGroup
- route: ActivatedRoute
- routeSubscription: Subscription
- authService: AuthService
- reviewService: ReviewService
- snackBar: MatSnackBar
- formBuilder: FormBuilder
- ordersService: OrdersService

+ stars: number[]
+ sku: number
+ purchasedSkus: number[]
+ @Input('rating') rating: number
+ @Input('starCount') starCount: number
+ @Input('color') color: string = 'accent'
+  @Output() ratingUpdated: EventEmitter
+ ratingArr: number[]
+ ratingSelected: boolean
+ showForm : boolean
--------------------------------------------------
+ ReviewFormComponent(authService:
AuthService,
   reviewService: ReviewService,
   formBuilder: FormBuilder,
   ordersService: OrdersService,
   snackBar: MatSnackBar,
   route: ActivatedRoute)

+ addToCart(sku: number): void
+ getRandomInt(min: number, max:
number): number
+ setStockStatus(): void
```

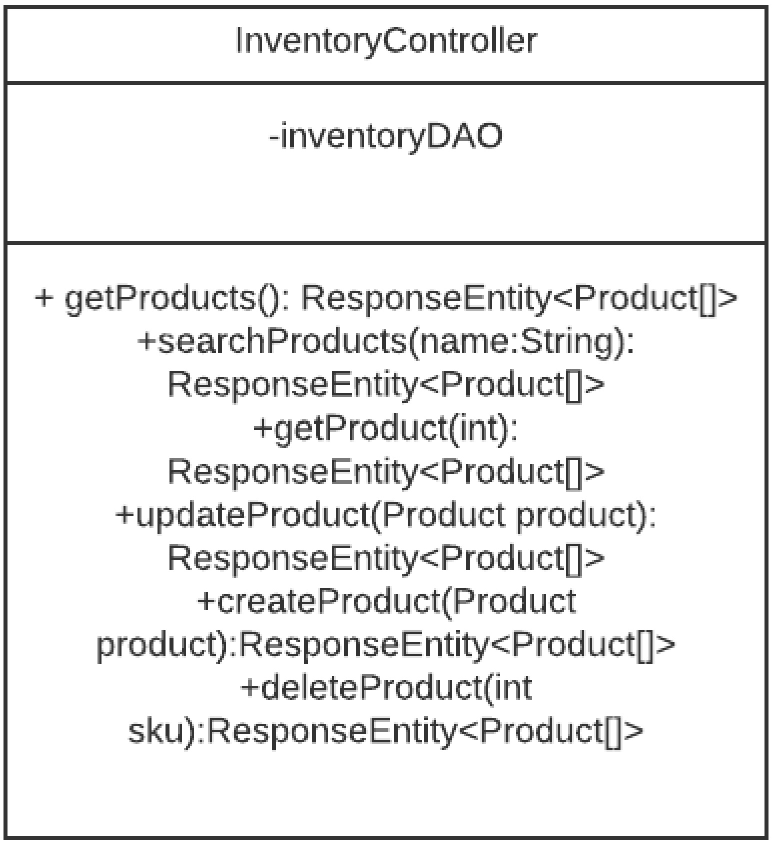
```
ReviewCardComponent
--------------------------------------------------
+ @Input() rating: number
+ @Input() comment: string
+ @Input() userId: number
+ @Input() sku: number
+ @Output() reviewRemoved:
EventEmitter<void>()
- authService: AuthService,
- reviewService: ReviewService,
- router: Router
--------------------------------------------------
ReviewsCardComponent(
   authService: AuthService,
   reviewService: ReviewService,
   router: Router
   )

+ ngOnInit(): void
+ removeReview(): void
+ range(stop: number, start: number, step:
number): number[]
+ canRemove(): Observable<boolean>
```

ViewModel Tier

Our ViewModel tier (the VM in M-V-VM) is comprised of a set of controller classes that handle REST API requests and responses. These controllers are responsible for managing the data model and ensuring that the data is properly formatted and validated before being sent to the client. The controllers are also responsible for handling any errors that may occur during the request/response cycle. The following diagram shows the classes that we built for the ViewModel tier.

InventoryController

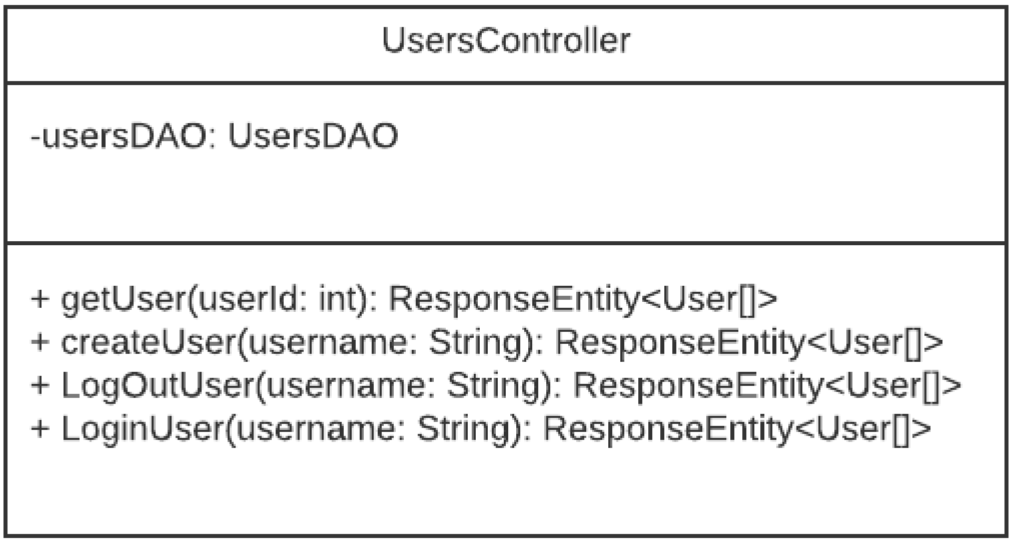
The InventoryController class is responsible for managing the Product resource by handling REST API requests. It interacts with the InventoryDAO for data access and manipulation, ensuring a clear separation of concerns within the application. The controller supports various operations, such as retrieving a specific product by SKU, fetching all products, searching for products by name, creating new products, updating existing products, and deleting products. By providing a comprehensive set of methods, the InventoryController class enables seamless interaction between the client and the underlying data model for


InventoryController
-inventoryDAO
+ getProducts(): ResponseEntity<Product[]>
+searchProducts(name:String): ResponseEntity<Product[]>
+getProduct(int): ResponseEntity<Product[]>
+updateProduct(Product product): ResponseEntity<Product[]>
+createProduct(Product product):ResponseEntity<Product[]>
+deleteProduct(int sku):ResponseEntity<Product[]>

the Product resource.

UsersController

The UsersController class manages User resources and handles their REST API requests. It communicates with the UsersDAO for data access and manipulation, maintaining a clear separation of concerns in the application. The controller supports a variety of operations such as retrieving a user by userId, creating new users, and logging users in and out. By providing a comprehensive set of methods, the UsersController class facilitates seamless interaction between the client and the underlying data model for User resources, ensuring a smooth user experience throughout the application.

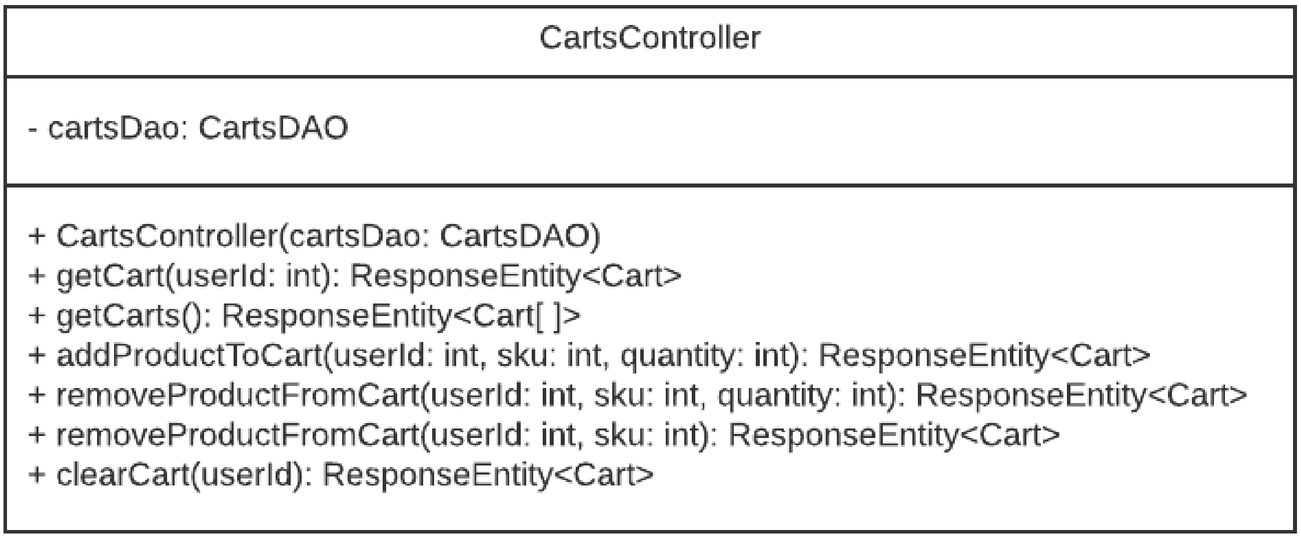

UsersController

-usersDAO: UsersDAO

+ getUser(userId: int): ResponseEntity<User[]>
+ createUser(username: String): ResponseEntity<User[]>
+ LogOutUser(username: String): ResponseEntity<User[]>
+ LoginUser(username: String): ResponseEntity<User[]>

CartsController

Our Carts controller highlights one of the main features of our application, the cart of the customer. It communicates with the CartsDAO for data access and manipulation, maintaining a clear separation of concerns in the application. The controller supports a variety of operations such as retrieving a cart,retrieving all the carts, adding product to cart, removing prodcut from cart, reducing the quantity of the product from cart and clearing the cart. By providing a comprehensive set of methods, the CartsController class facilitates seamless interaction between the client and the underlying data model for Cart resources, ensuring a smooth user experience throughout the application.

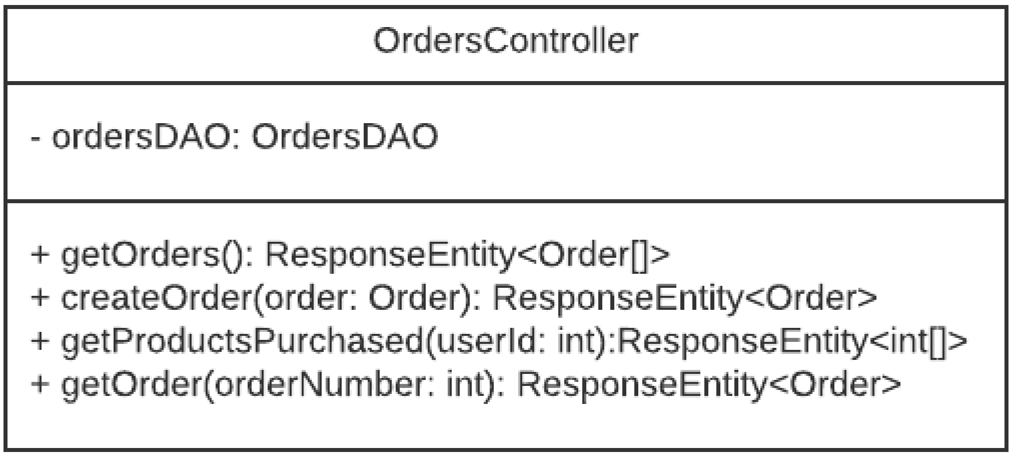

CartsController

- cartsDao: CartsDAO

+ CartsController(cartsDao: CartsDAO)
+ getCart(userId: int): ResponseEntity<Cart>
+ getCarts(): ResponseEntity<Cart[ ]>
+ addProductToCart(userId: int, sku: int, quantity: int): ResponseEntity<Cart>
+ removeProductFromCart(userId: int, sku: int, quantity: int): ResponseEntity<Cart>
+ removeProductFromCart(userId: int, sku: int): ResponseEntity<Cart>
+ clearCart(userId): ResponseEntity<Cart>

OrdersController

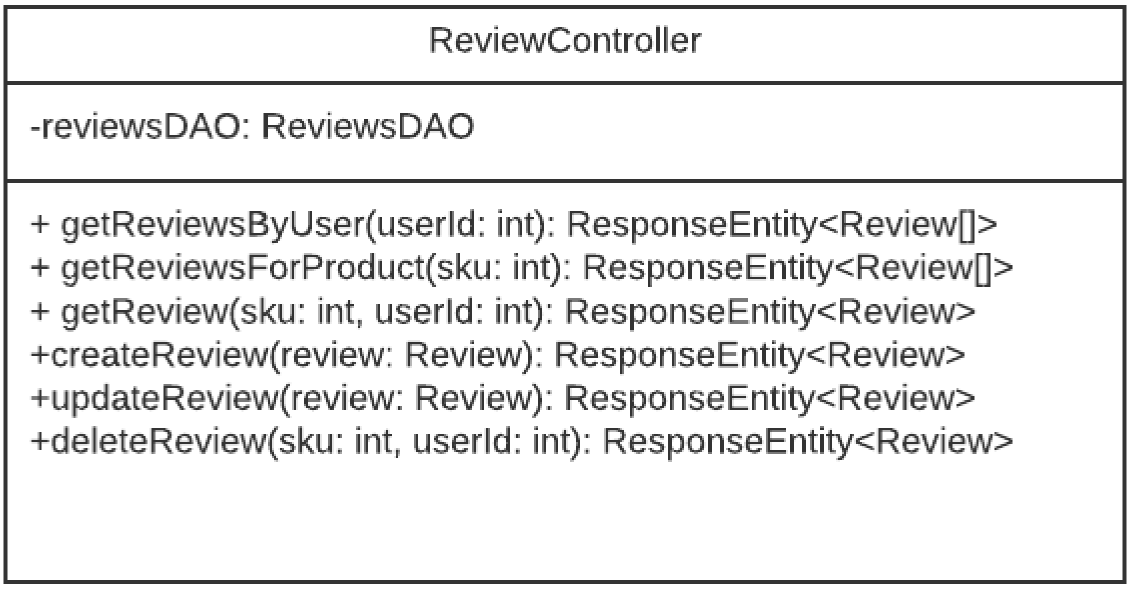
Our Orders controller manages the Orders resource and handles the REST API requests. It communicates with the OrdersDAO for data access and manipulation, adhering to the separation of concerns principle. The controller supports various operations, such as getting an order, getting list of orders, creating an order and getting the products purchased by the user.By providing a comprehensive set of methods, the OrdersController class facilitates seamless interaction between the client and the underlying data model for orders resources, ensuring a smooth user experience throughout the application.


OrdersController
- ordersDAO: OrdersDAO
+ getOrders(): ResponseEntity<Order[]>
+ createOrder(order: Order): ResponseEntity<Order>
+ getProductsPurchased(userId: int):ResponseEntity<int[]>
+ getOrder(orderNumber: int): ResponseEntity<Order>

ReviewController

The ReviewController class is responsible for managing Review resources in the e-commerce application and handling their REST API requests. It communicates with the ReviewsDAO for data access and manipulation, adhering to the separation of concerns principle. The controller supports various operations, such as retrieving reviews by product SKU, retrieving reviews by user ID, creating, updating, and deleting reviews.

The ReviewController class provides a comprehensive set of methods that allow for smooth interaction between the client and the underlying data model for Review resources. By offering endpoints for fetching reviews for a specific product or user, the class ensures a better user experience, allowing users to view and manage their reviews efficiently. Additionally, it facilitates interaction with product reviews for all users, which can be essential for understanding user feedback and improving products in the e-commerce application.


ReviewController
-reviewsDAO: ReviewsDAO
+ getReviewsByUser(userId: int): ResponseEntity<Review[]>
+ getReviewsForProduct(sku: int): ResponseEntity<Review[]>
+ getReview(sku: int, userId: int): ResponseEntity<Review>
+createReview(review: Review): ResponseEntity<Review>
+updateReview(review: Review): ResponseEntity<Review>
+deleteReview(sku: int, userId: int): ResponseEntity<Review>

Model Tier

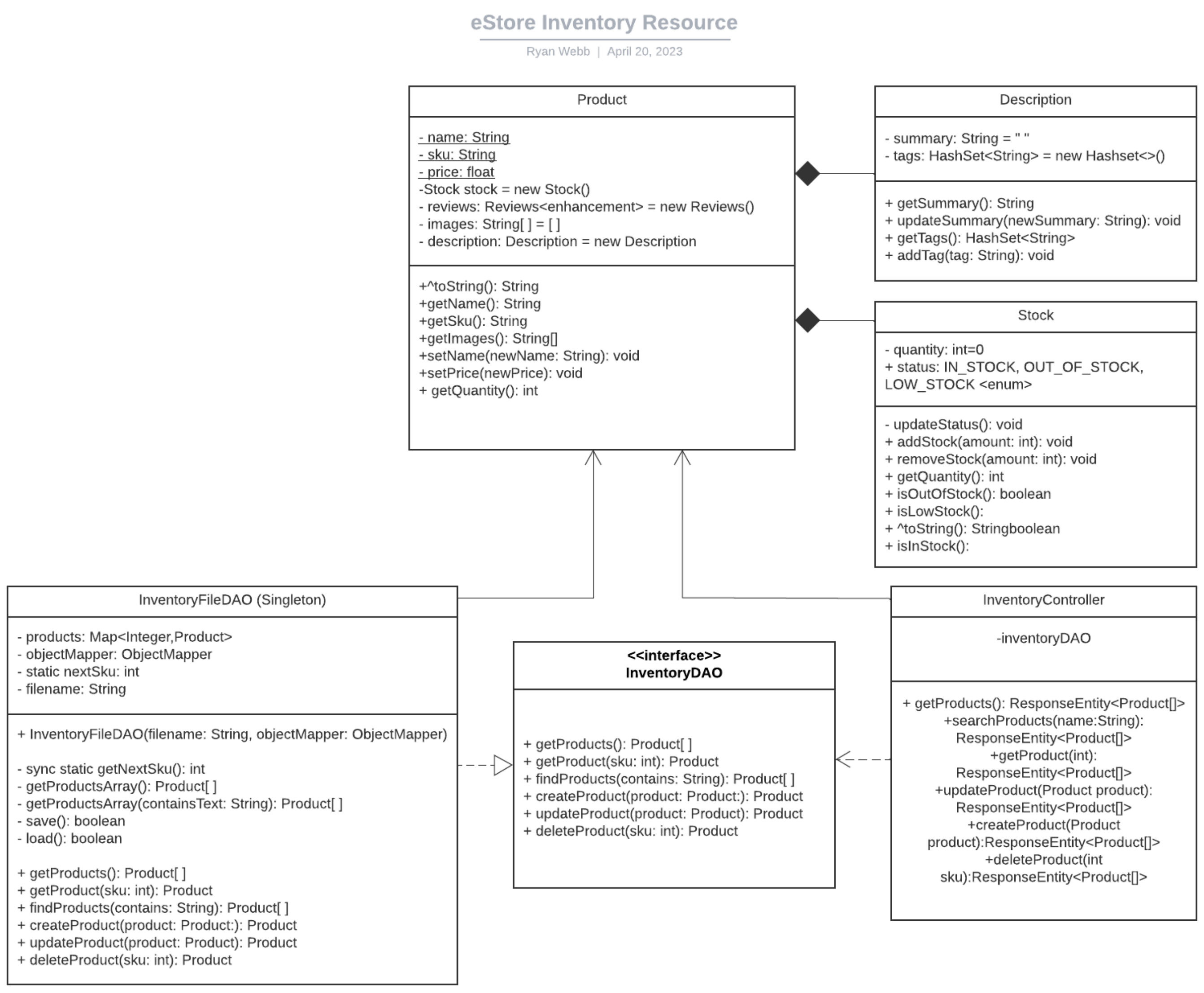
Our model tier (first M in M-V-VM) is built using Java and Spring Framework. The model tier is responsible for storing the application data objects and providing persistence. The model tier is divided into five resources: Inventory, User, Carts, Orders, and Reviews.

Inventory Resource

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 achieves this via serialization and deserialization of Product objects to and from a JSON file.

Inventory Resource Class Diagram

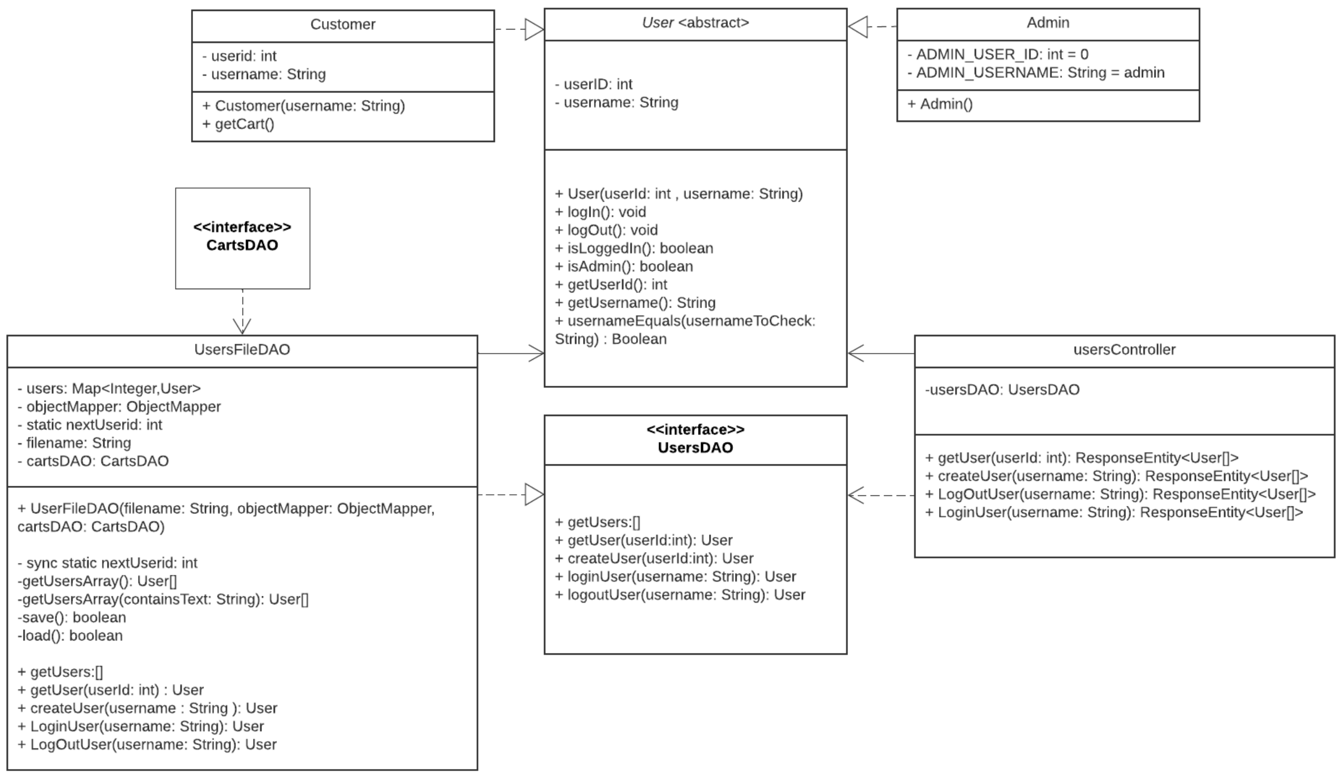

eStore Inventory Resource
Ryan Webb | April 20, 2023
Product
- name: String
- sku: String
- price: float
-Stock stock = new Stock()
- reviews: Reviews<enhancement> = new Reviews()
- images: String[ ] = [ ]
- description: Description = new Description
+^toString(): String
+getName(): String
+getSku(): String
+getImages(): String[]
+setName(newName: String): void
+setPrice(newPrice): void
+ getQuantity(): int
Description
- summary: String = " "
- tags: HashSet<String> = new Hashset<>()
+ getSummary(): String
+ updateSummary(newSummary: String): void
+ getTags(): HashSet<String>
+ addTag(tag: String): void
Stock
- quantity: int=0
+ status: IN_STOCK, OUT_OF_STOCK, LOW_STOCK <enum>
- updateStatus(): void
+ addStock(amount: int): void
+ removeStock(amount: int): void
+ getQuantity(): int
+ isOutOfStock(): boolean
+ isLowStock():
+ ^toString(): Stringboolean
+ isInStock():
InventoryFileDAO (Singleton)
- products: Map<Integer,Product>
- objectMapper: ObjectMapper
- static nextSku: int
- filename: String
+ InventoryFileDAO(filename: String, objectMapper: ObjectMapper)
- sync static getNextSku(): int
- getProductsArray(): Product[ ]
- getProductsArray(containsText: String): Product[ ]
- save(): boolean
- load(): boolean
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product
<<interface>>
InventoryDAO
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product
InventoryController
-inventoryDAO
+ getProducts(): ResponseEntity<Product[]>
+searchProducts(name:String): ResponseEntity<Product[]>
+getProduct(int): ResponseEntity<Product[]>
+updateProduct(Product product): ResponseEntity<Product[]>
+createProduct(Product product):ResponseEntity<Product[]>
+deleteProduct(int sku):ResponseEntity<Product[]>

User Resource

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 achieves this via serialization and deserialization of User objects to and from a JSON file.

User Resource Class Diagram

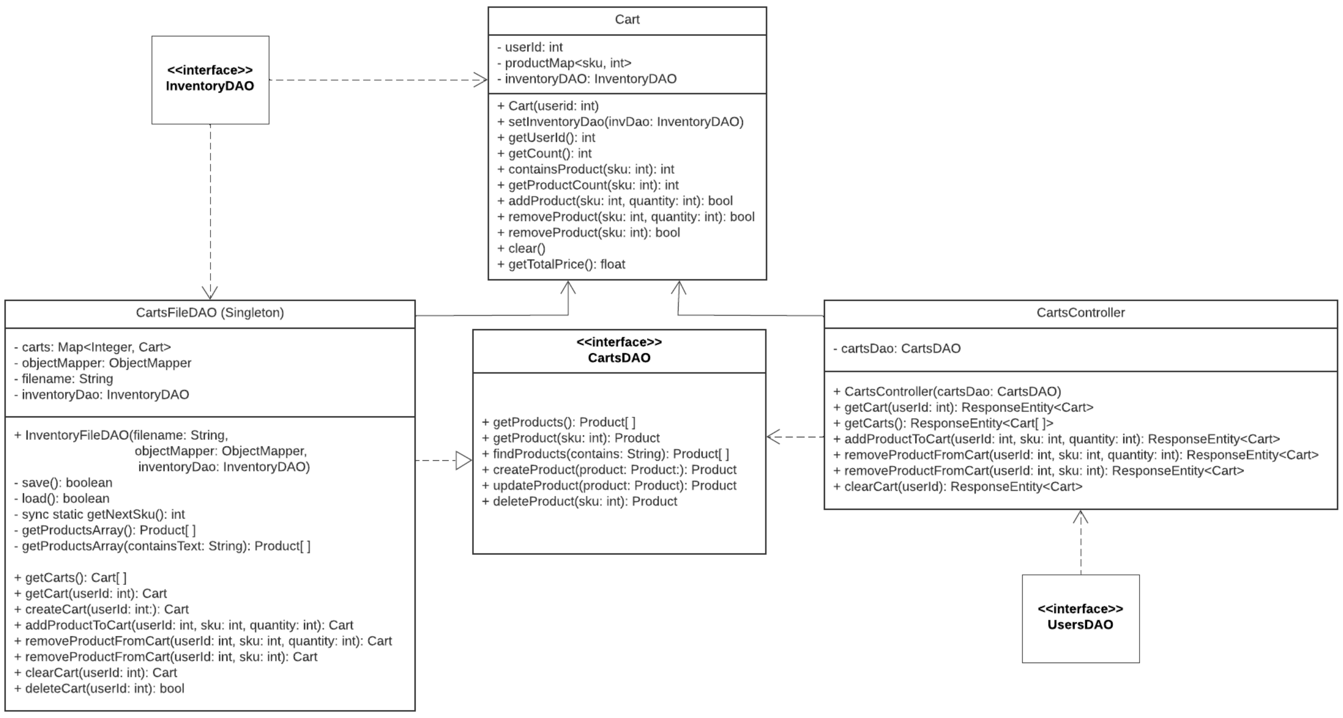

Customer
- userid: int
- username: String
+ Customer(username: String)
+ getCart()
User <abstract>
- userID: int
- username: String
+ User(userId: int , username: String)
+ logIn(): void
+ logOut(): void
+ isLoggedIn(): boolean
+ isAdmin(): boolean
+ getUserId(): int
+ getUsername(): String
+ usernameEquals(usernameToCheck: String) : Boolean
Admin
- ADMIN_USER_ID: int = 0
- ADMIN_USERNAME: String = admin
+ Admin()
<<interface>> CartsDAO
UsersFileDAO
- users: Map<Integer,User>
- objectMapper: ObjectMapper
- static nextUserid: int
- filename: String
- cartsDAO: CartsDAO
+ UserFileDAO(filename: String, objectMapper: ObjectMapper, cartsDAO: CartsDAO)
- sync static nextUserid: int
-getUsersArray(): User[]
-getUsersArray(containsText: String): User[]
-save(): boolean
-load(): boolean
+ getUsers:[]
+ getUser(userId: int) : User
+ createUser(username : String): User
+ LoginUser(username: String): User
+ LogOutUser(username: String): User
<<interface>> UsersDAO
+ getUsers:[]
+ getUser(userId:int): User
+ createUser(userId:int): User
+ loginUser(username: String): User
+ logoutUser(username: String): User
usersController
-usersDAO: UsersDAO
+ getUser(userId: int): ResponseEntity<User[]>
+ createUser(username: String): ResponseEntity<User[]>
+ LogOutUser(username: String): ResponseEntity<User[]>
+ LoginUser(username: String): ResponseEntity<User[]>

Carts Resource

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 achieves this via serialization and deserialization of Cart objects to and from a JSON file.

Carts Resource Class Diagram

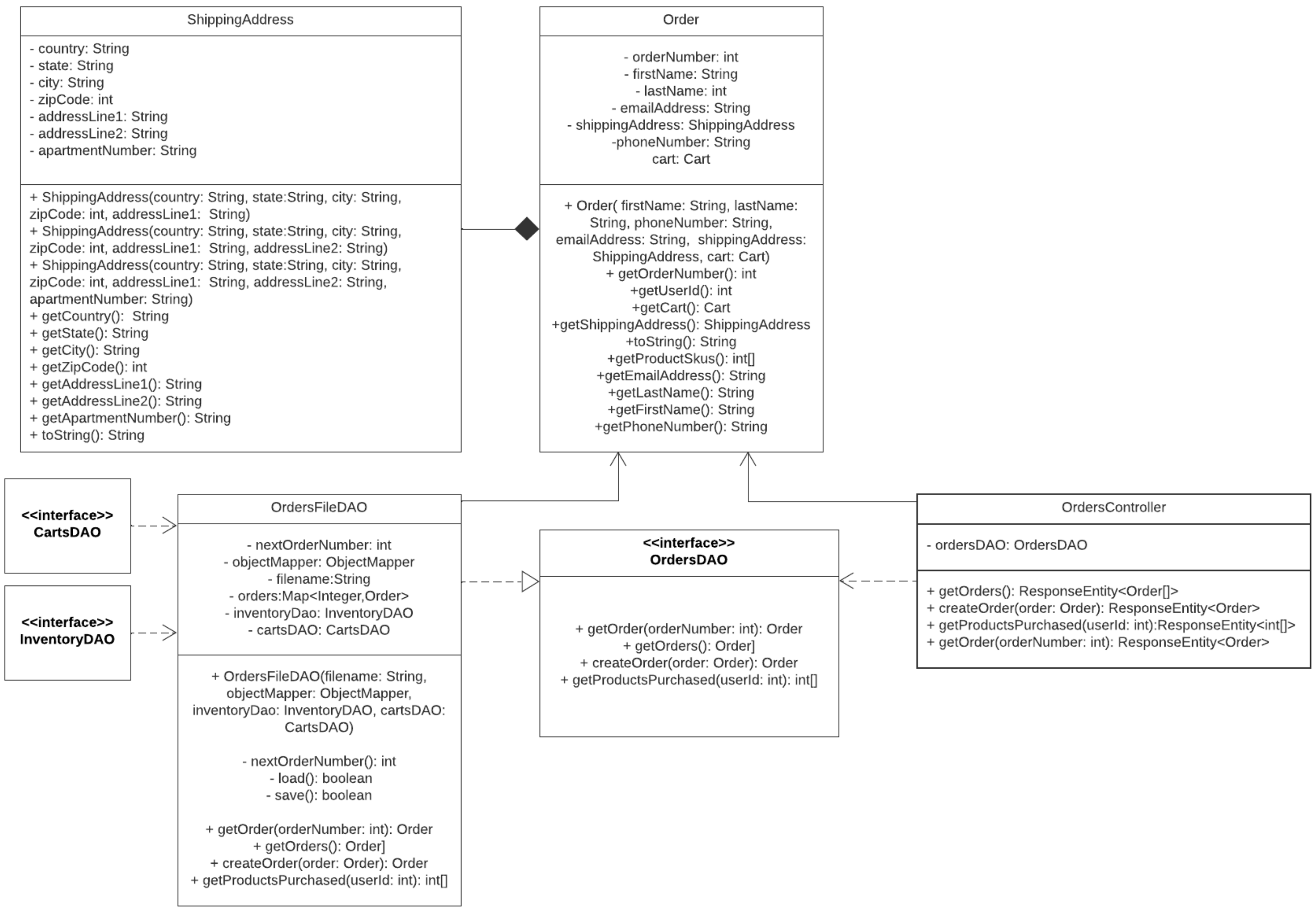

Cart
- userId: int
- productMap<sku, int>
- inventoryDAO: InventoryDAO
+ Cart(userid: int)
+ setInventoryDao(invDao: InventoryDAO)
+ getUserId(): int
+ getCount(): int
+ containsProduct(sku: int): int
+ getProductCount(sku: int): int
+ addProduct(sku: int, quantity: int): bool
+ removeProduct(sku: int, quantity: int): bool
+ removeProduct(sku: int): bool
+ clear()
+ getTotalPrice(): float
<<interface>> InventoryDAO
CartsFileDAO (Singleton)
- carts: Map<Integer, Cart>
- objectMapper: ObjectMapper
- filename: String
- inventoryDao: InventoryDAO
+ InventoryFileDAO(filename: String, objectMapper: ObjectMapper, inventoryDao: InventoryDAO)
- save(): boolean
- load(): boolean
- sync static getNextSku(): int
- getProductsArray(): Product[ ]
- getProductsArray(containsText: String): Product[ ]
+ getCarts(): Cart[ ]
+ getCart(userId: int): Cart
+ createCart(userId: int:): Cart
+ addProductToCart(userId: int, sku: int, quantity: int): Cart
+ removeProductFromCart(userId: int, sku: int, quantity: int): Cart
+ removeProductFromCart(userId: int, sku: int): Cart
+ clearCart(userId: int): Cart
+ deleteCart(userId: int): bool
<<interface>> CartsDAO
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product
CartsController
- cartsDao: CartsDAO
+ CartsController(cartsDao: CartsDAO)
+ getCart(userId: int): ResponseEntity<Cart>
+ getCarts(): ResponseEntity<Cart[ ]>
+ addProductToCart(userId: int, sku: int, quantity: int): ResponseEntity<Cart>
+ removeProductFromCart(userId: int, sku: int, quantity: int): ResponseEntity<Cart>
+ removeProductFromCart(userId: int, sku: int): ResponseEntity<Cart>
+ clearCart(userId): ResponseEntity<Cart>
<<interface>> UsersDAO

Orders Resource

The Orders resource is responsible for handling the orders of the customers. Each order contains a order number, first name and last name, phone number, email address and shipping address of the user. It also stores an the cart of the user. The shipping address is it's own class which contains attributes like country, state, address line, etc. The Orders resource also defines the OrdersDAO interface for Order object persistence.

Persistence is provided by the OrdersDAO interface, and its concrete implementation OrdersFileDAO. The OrdersFileDAO class achieves this via serialization and deserialization of Order objects to and from a JSON file.

Orders Resource Class Diagram

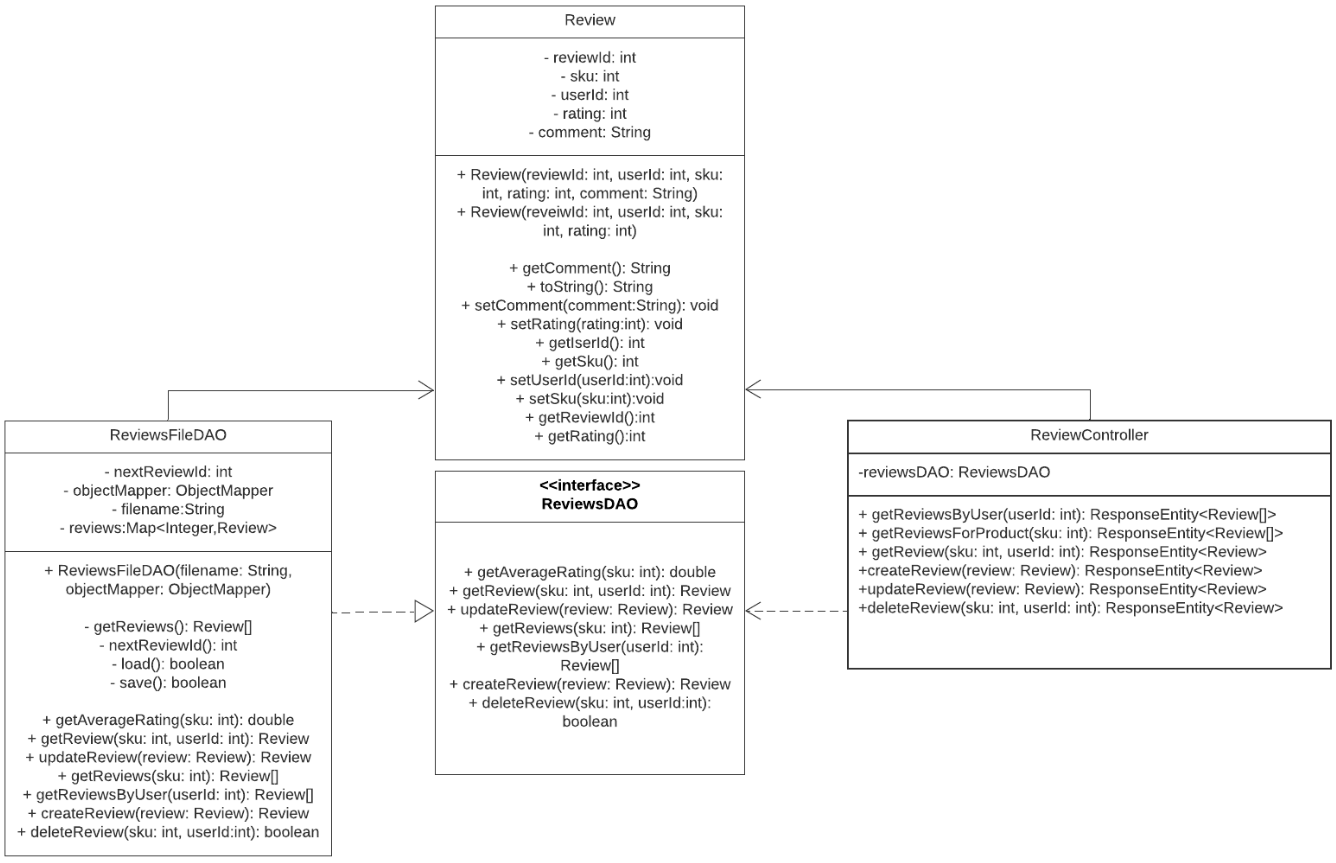

ShippingAddress
- country: String
- state: String
- city: String
- zipCode: int
- addressLine1: String
- addressLine2: String
- apartmentNumber: String
+ ShippingAddress(country: String, state:String, city: String, zipCode: int, addressLine1: String)
+ ShippingAddress(country: String, state:String, city: String, zipCode: int, addressLine1: String, addressLine2: String)
+ ShippingAddress(country: String, state:String, city: String, zipCode: int, addressLine1: String, addressLine2: String, apartmentNumber: String)
+ getCountry(): String
+ getState(): String
+ getCity(): String
+ getZipCode(): int
+ getAddressLine1(): String
+ getAddressLine2(): String
+ getApartmentNumber(): String
+ toString(): String
Order
- orderNumber: int
- firstName: String
- lastName: int
- emailAddress: String
- shippingAddress: ShippingAddress
-phoneNumber: String
cart: Cart
+ Order(firstName: String, lastName: String, phoneNumber: String, emailAddress: String, shippingAddress: ShippingAddress, cart: Cart)
+ getOrderNumber(): int
+getUserId(): int
+getCart(): Cart
+getShippingAddress(): ShippingAddress
+toString(): String
+getProductSkus(): int[]
+getEmailAddress(): String
+getLastName(): String
+getFirstName(): String
+getPhoneNumber(): String
<<interface>> CartsDAO
<<interface>> InventoryDAO
OrdersFileDAO
- nextOrderNumber: int
- objectMapper: ObjectMapper
- filename:String
- orders:Map<Integer,Order>
- inventoryDao: InventoryDAO
- cartsDAO: CartsDAO
+ OrdersFileDAO(filename: String, objectMapper: ObjectMapper, inventoryDao: InventoryDAO, cartsDAO: CartsDAO)
- nextOrderNumber(): int
- load(): boolean
- save(): boolean
+ getOrder(orderNumber: int): Order
+ getOrders(): Order]
+ createOrder(order: Order): Order
+ getProductsPurchased(userId: int): int[]
<<interface>> OrdersDAO
+ getOrder(orderNumber: int): Order
+ getOrders(): Order]
+ createOrder(order: Order): Order
+ getProductsPurchased(userId: int): int[]
OrdersController
- ordersDAO: OrdersDAO
+ getOrders(): ResponseEntity<Order[]>
+ createOrder(order: Order): ResponseEntity<Order>
+ getProductsPurchased(userId: int):ResponseEntity<int[]>
+ getOrder(orderNumber: int): ResponseEntity<Order>

Reviews Resource

The Reviews resource is responsible for storing and managing customer reviews of products sold in the e-store. The Review class is used to represent a review and contains the review ID, user ID, product SKU, rating, and a comment. The Reviews resource also defines the ReviewsDAO interface for Review object persistence.

Persistence is provided by the ReviewsDAO interface, and its concrete implementation ReviewsFileDAO. The ReviewsFileDAO class achieves this via serialization and deserialization of Review objects to and from a JSON file.

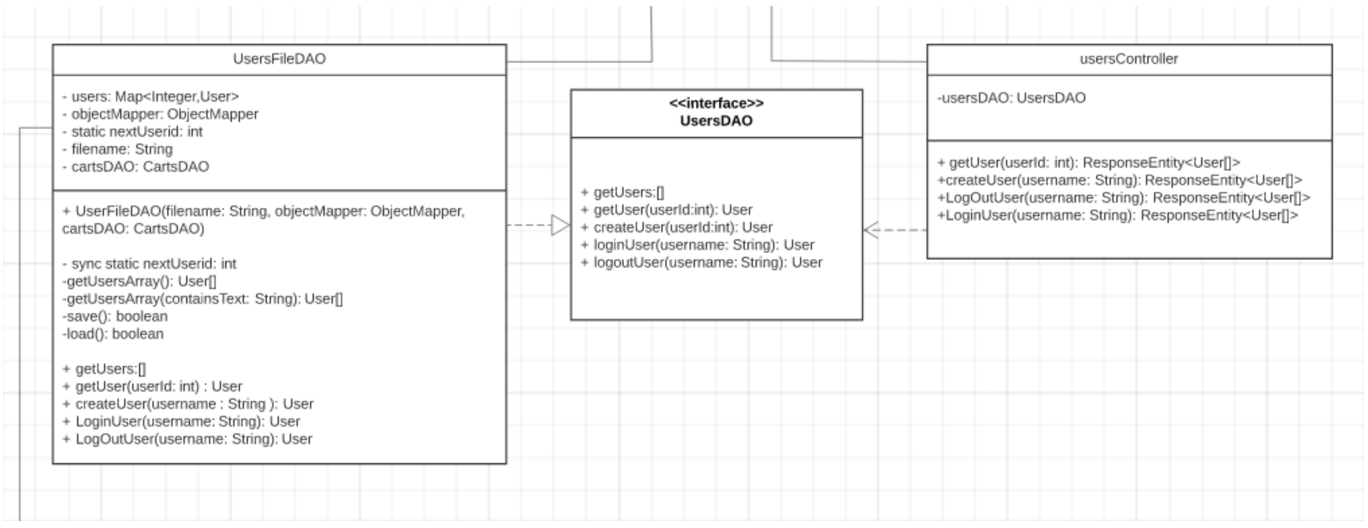
Reviews Resource Class Diagram


Review
- reviewId: int
- sku: int
- userId: int
- rating: int
- comment: String
+ Review(reviewId: int, userId: int, sku: int, rating: int, comment: String)
+ Review(reveiwId: int, userId: int, sku: int, rating: int)
+ getComment(): String
+ toString(): String
+ setComment(comment:String): void
+ setRating(rating:int): void
+ getIserId(): int
+ getSku(): int
+ setUserId(userId:int):void
+ setSku(sku:int):void
+ getReviewId():int
+ getRating():int
ReviewsFileDAO
- nextReviewId: int
- objectMapper: ObjectMapper
- filename:String
- reviews:Map<Integer,Review>
+ ReviewsFileDAO(filename: String, objectMapper: ObjectMapper)
- getReviews(): Review[]
- nextReviewId(): int
- load(): boolean
- save(): boolean
+ getAverageRating(sku: int): double
+ getReview(sku: int, userId: int): Review
+ updateReview(review: Review): Review
+ getReviews(sku: int): Review[]
+ getReviewsByUser(userId: int): Review[]
+ createReview(review: Review): Review
+ deleteReview(sku: int, userId:int): boolean
<<interface>>
ReviewsDAO
+ getAverageRating(sku: int): double
+ getReview(sku: int, userId: int): Review
+ updateReview(review: Review): Review
+ getReviews(sku: int): Review[]
+ getReviewsByUser(userId: int): Review[]
+ createReview(review: Review): Review
+ deleteReview(sku: int, userId:int): boolean
ReviewController
-reviewsDAO: ReviewsDAO
+ getReviewsByUser(userId: int): ResponseEntity<Review[]>
+ getReviewsForProduct(sku: int): ResponseEntity<Review[]>
+ getReview(sku: int, userId: int): ResponseEntity<Review>
+createReview(review: Review): ResponseEntity<Review>
+updateReview(review: Review): ResponseEntity<Review>
+deleteReview(sku: int, userId: int): ResponseEntity<Review>

OO Design Principles

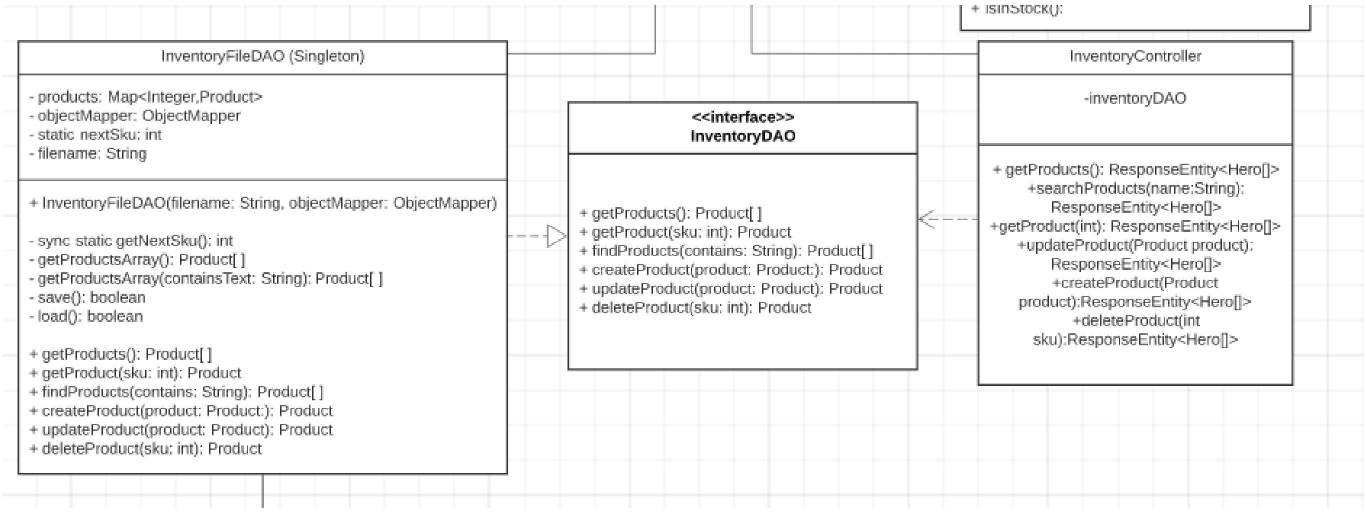
Dependency Inversion Principle (DIP)

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.


UsersFileDAO
- users: Map<Integer,User>
- objectMapper: ObjectMapper
- static nextUserid: int
- filename: String
- cartsDAO: CartsDAO
+ UserFileDAO(filename: String, objectMapper: ObjectMapper, cartsDAO: CartsDAO)
- sync static nextUserid: int
-getUsersArray(): User[]
-getUsersArray(containsText: String): User[]
-save(): boolean
-load(): boolean
+ getUsers:[]
+ getUser(userId: int) : User
+ createUser(username : String): User
+ LoginUser(username: String): User
+ LogOutUser(username: String): User
<<interface>>
UsersDAO
+ getUsers:[]
+ getUser(userId:int): User
+ createUser(userId:int): User
+ loginUser(username: String): User
+ logoutUser(username: String): User
usersController
-usersDAO: UsersDAO
+ getUser(userId: int): ResponseEntity<User[]>
+createUser(username: String): ResponseEntity<User[]>
+LogOutUser(username: String): ResponseEntity<User[]>
+LoginUser(username: String): ResponseEntity<User[]>

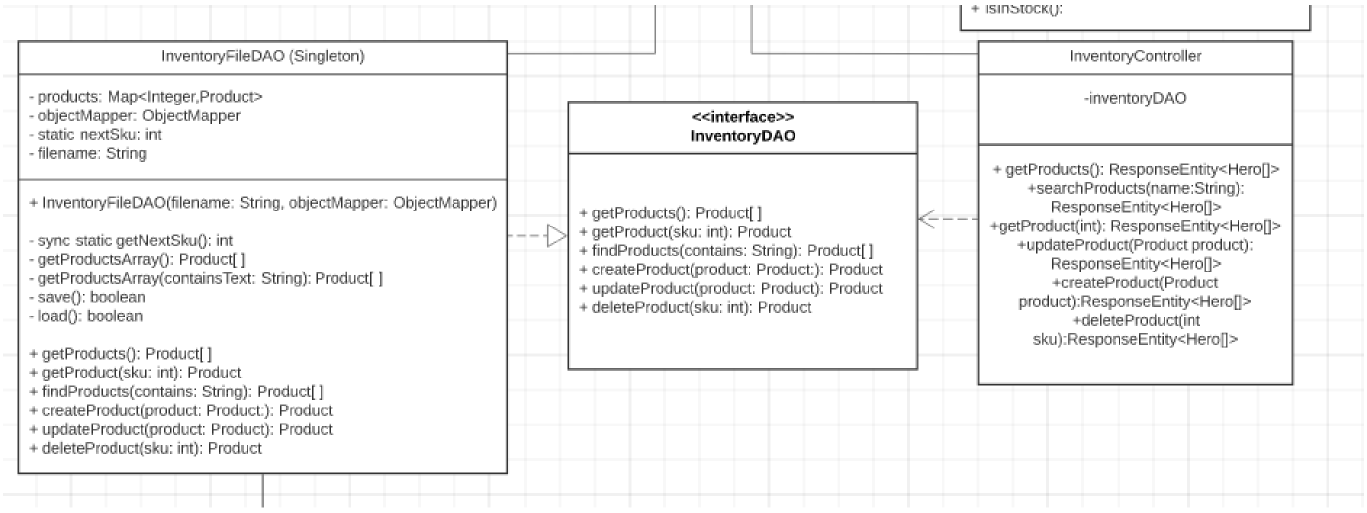
Dependency Injection

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.


+ isInStock():
InventoryFileDAO (Singleton)
- products: Map<Integer,Product>
- objectMapper: ObjectMapper
- static nextSku: int
- filename: String
+ InventoryFileDAO(filename: String, objectMapper: ObjectMapper)
- sync static getNextSku(): int
- getProductsArray(): Product[ ]
- getProductsArray(containsText: String): Product[ ]
- save(): boolean
- load(): boolean
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product
<<interface>>
InventoryDAO
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product
InventoryController
-inventoryDAO
+ getProducts(): ResponseEntity<Hero[]>
+searchProducts(name:String): ResponseEntity<Hero[]>
+getProduct(int): ResponseEntity<Hero[]>
+updateProduct(Product product): ResponseEntity<Hero[]>
+createProduct(Product product):ResponseEntity<Hero[]>
+deleteProduct(int sku):ResponseEntity<Hero[]>

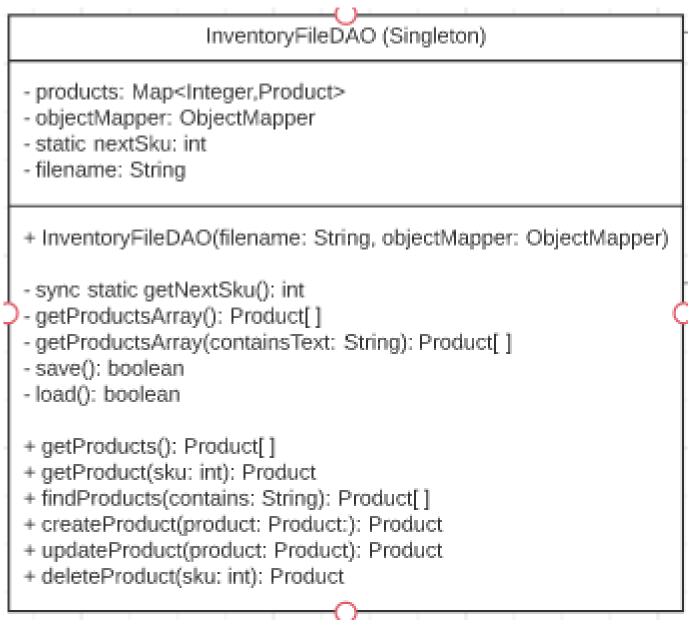
Pure Fabrication

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 hypothetical 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 hypothetical wishlist.


+ isInStock():
InventoryFileDAO (Singleton)
- products: Map<Integer,Product>
- objectMapper: ObjectMapper
- static nextSku: int
- filename: String
+ InventoryFileDAO(filename: String, objectMapper: ObjectMapper)
- sync static getNextSku(): int
- getProductsArray(): Product[ ]
- getProductsArray(containsText: String): Product[ ]
- save(): boolean
- load(): boolean
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product
<<interface>>
InventoryDAO
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product
InventoryController
-inventoryDAO
+ getProducts(): ResponseEntity<Hero[]>
+searchProducts(name:String): ResponseEntity<Hero[]>
+getProduct(int): ResponseEntity<Hero[]>
+updateProduct(Product product): ResponseEntity<Hero[]>
+createProduct(Product product):ResponseEntity<Hero[]>
+deleteProduct(int sku):ResponseEntity<Hero[]>

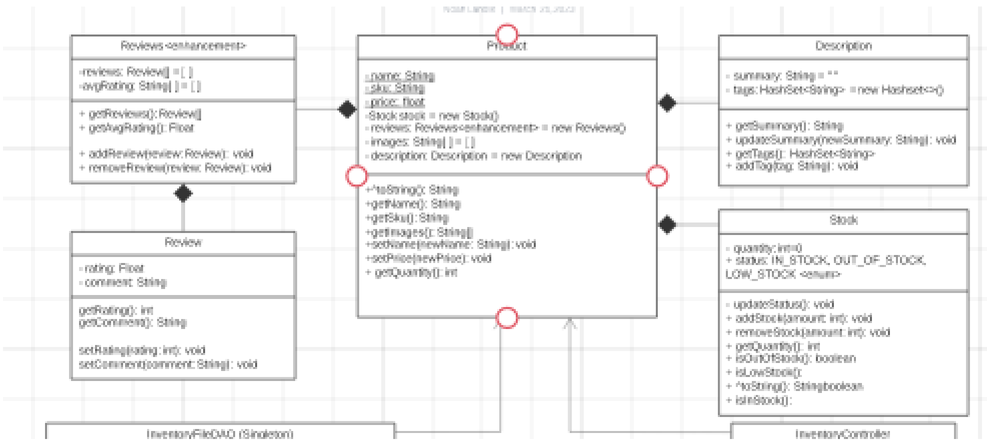
Single Responsibility

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 that 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 that 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 search 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.


InventoryFileDAO (Singleton)
- products: Map<Integer,Product>
- objectMapper: ObjectMapper
- static nextSku: int
- filename: String
+ InventoryFileDAO(filename: String, objectMapper: ObjectMapper)
- sync static getNextSku(): int
- getProductsArray(): Product[ ]
- getProductsArray(containsText: String): Product[ ]
- save(): boolean
- load(): boolean
+ getProducts(): Product[ ]
+ getProduct(sku: int): Product
+ findProducts(contains: String): Product[ ]
+ createProduct(product: Product:): Product
+ updateProduct(product: Product): Product
+ deleteProduct(sku: int): Product

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.


Reviews<enhancement>
-reviews: Review[] = [ ]
-avgRating: String[ ] = [ ]
+ getReviews(): Review[]
+ getAvgRating(): Float
+ addReview(review: Review): void
+ removeReview(review: Review): void
Review
-rating: Float
-comment: String
getRating(): int
getComment(): String
setRating(rating: int): void
setComment(comment: String): void
Product
-name: String
-sku: String
-price: float
-Stock stock = new Stock()
-reviews: Reviews<enhancement> = new Reviews()
-images: String[ ] = [ ]
-description: Description = new Description
+^toString(): String
+getName(): String
+getSku(): String
+getImages(): String[]
+setName(newName: String): void
+setPrice(newPrice): void
+ getQuantity(): int
Description
- summary: String = ""
- tags: HashSet<String> = new Hashset<>()
+ getSummary(): String
+ updateSummary(newSummary: String): void
+ getTags(): HashSet<String>
+ addTag(tag: String): void
Stock
- quantity: int=0
+ status: IN_STOCK, OUT_OF_STOCK, LOW_STOCK <enum>
- updateStatus(): void
+ addStock(amount: int): void
+ removeStock(amount: int): void
+ getQuantity(): int
+ isOutOfStock(): boolean
+ isLowStock():
+ ^toString(): Stringboolean
+ isInStock():
InventoryFileDAO (Singleton)
InventoryController

Controller

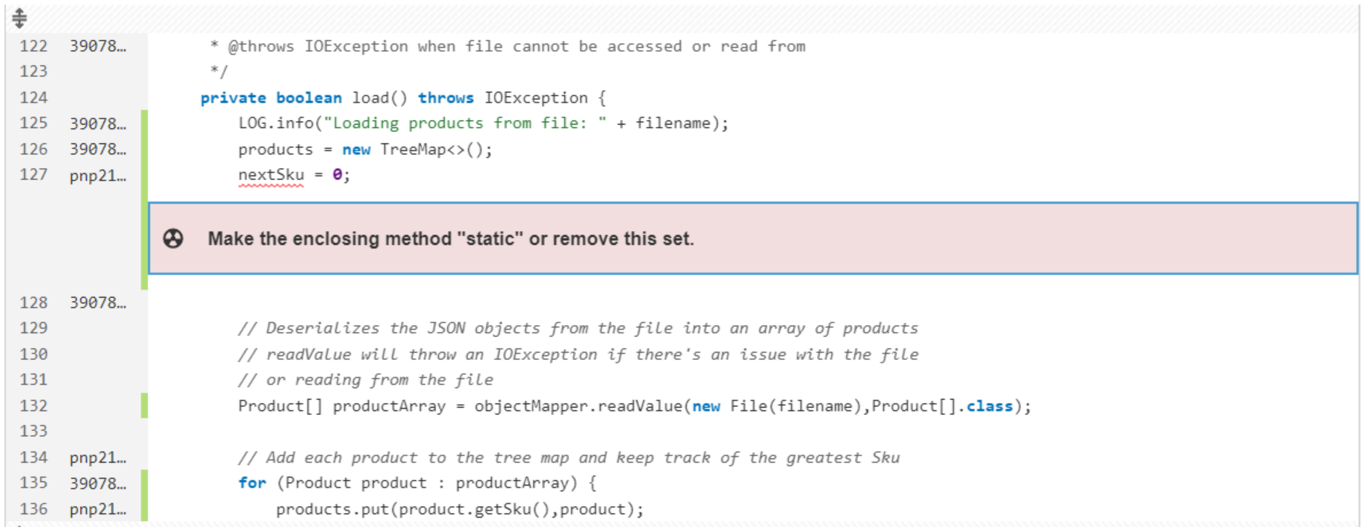
In our backend, we have a controller tier which handles API requests from the Angular frontend (view). The controller tier is responsible for handling the requests and sending the appropriate responses back to the frontend, and represents a separation of responsibility between our View tier (the Angular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

Static Code Analysis/Future Design Improvements

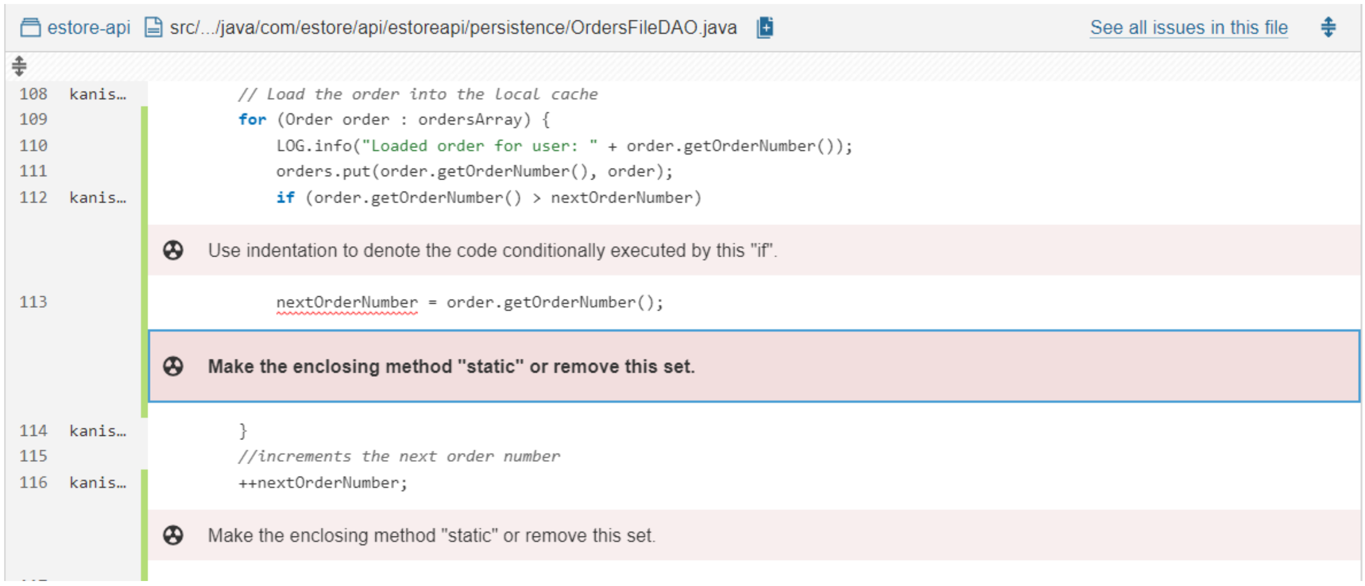
One of the places where SonarQube flagged our code was in our fileDAO classes for each of our APIs. We were not defining a variable static which is used to keep track of the ID of the object for which the API is built. Since it is not static, it is not shared across all instances of the class. This means that if we were to create a new instance of the class, the ID would be reset to 0. It currently is not an issue because there is only one instance of the fileDAO ever made in our application. However, if we were to create a new instance of the class, it would cause issues. To fix this, we would make the variable static.



```
     * @throws IOException when file cannot be accessed or read from
     */
    private boolean load() throws IOException {
        LOG.info("Loading products from file: " + filename);
        products = new TreeMap<>();
        nextSku = 0;
```

Make the enclosing method "static" or remove this set.

```

        // Deserializes the JSON objects from the file into an array of products
        // readValue will throw an IOException if there's an issue with the file
        // or reading from the file
        Product[] productArray = objectMapper.readValue(new File(filename),Product[].class);

        // Add each product to the tree map and keep track of the greatest Sku
        for (Product product : productArray) {
            products.put(product.getSku(),product);
```



estore-api src/.../java/com/estore/api/estoreapi/persistence/OrdersFileDAO.java See all issues in this file

```
        // Load the order into the local cache
        for (Order order : ordersArray) {
            LOG.info("Loaded order for user: " + order.getOrderNumber());
            orders.put(order.getOrderNumber(), order);
            if (order.getOrderNumber() > nextOrderNumber)
```

Use indentation to denote the code conditionally executed by this "if".

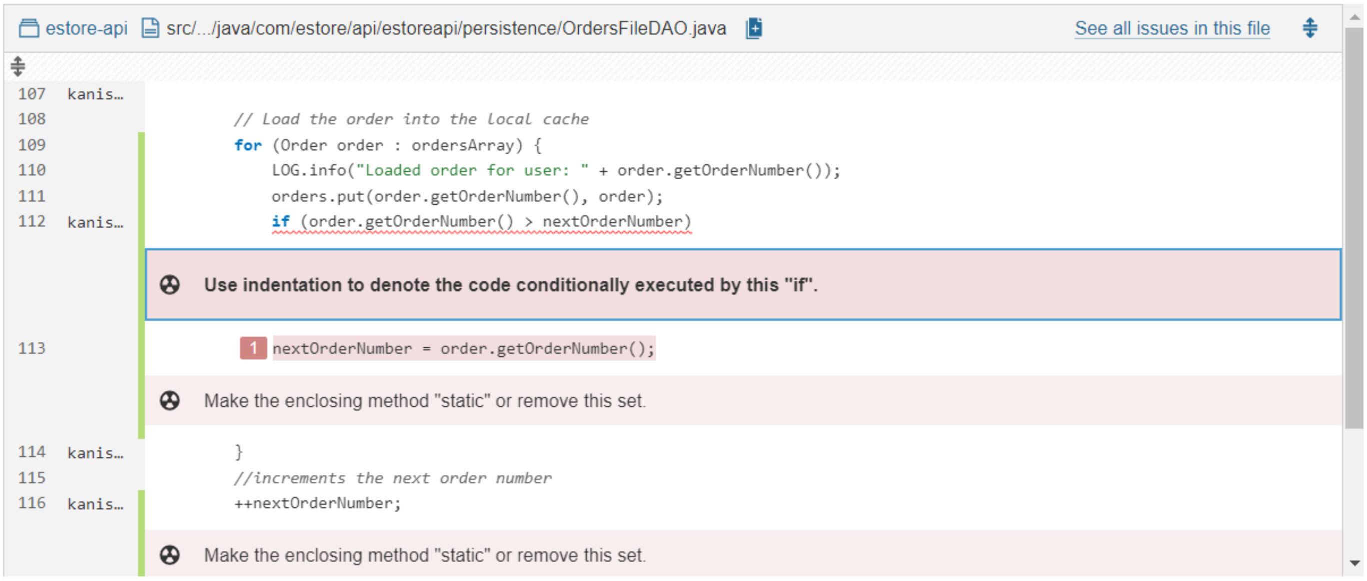
```
            nextOrderNumber = order.getOrderNumber();
```

Make the enclosing method "static" or remove this set.

```
        }
        //increments the next order number
        ++nextOrderNumber;
```

Make the enclosing method "static" or remove this set.

Another place where SonarQube flagged out code was a conditional being out of place. This is an indentation issue. The conditional is not indented properly. This is a minor issue and can be fixed by simply indenting the conditional properly.


estore-api src/.../java/com/estore/api/estoreapi/persistence/OrdersFileDAO.java See all issues in this file

```
107 kanis…
108            // Load the order into the local cache
109            for (Order order : ordersArray) {
110                LOG.info("Loaded order for user: " + order.getOrderNumber());
111                orders.put(order.getOrderNumber(), order);
112 kanis…         if (order.getOrderNumber() > nextOrderNumber)
```

Use indentation to denote the code conditionally executed by this "if".

```
113              1 nextOrderNumber = order.getOrderNumber();
```

Make the enclosing method "static" or remove this set.

```
114 kanis…     }
115            //increments the next order number
116 kanis…     ++nextOrderNumber;
```

Make the enclosing method "static" or remove this set.

If our team had additional time, we would explore more areas where we could utilize abstraction to avoid DRY violations. For instance, our Orders, Carts, Inventory, and Review services all currently have handleError<T> (operation = 'operation', result?: T), with no difference in implementation between the services. Going forward, we would make this a public function that could be called by the services that require it. In keeping with the goal of adhering to DRY, we would remove two of the 'admin-flag' attributes from our Users' resource. Currently, a User has three admin-flags: 'isAdmin', 'type', and 'admin'.

In addition to adhering to DRY principles, we would change how we display the stock to follow Information Expert. For example, our Stock model currently has a 'Status' enum which is used to represent a brief description of the stock quantity, to be displayed to customers (i.e, 'Low Stock' would be displayed when there are less than 10 products in the inventory). Instead of displaying the model's status, we calculate the status on the front end in our catalog-product-card.component.ts file via setStockStatus() method.

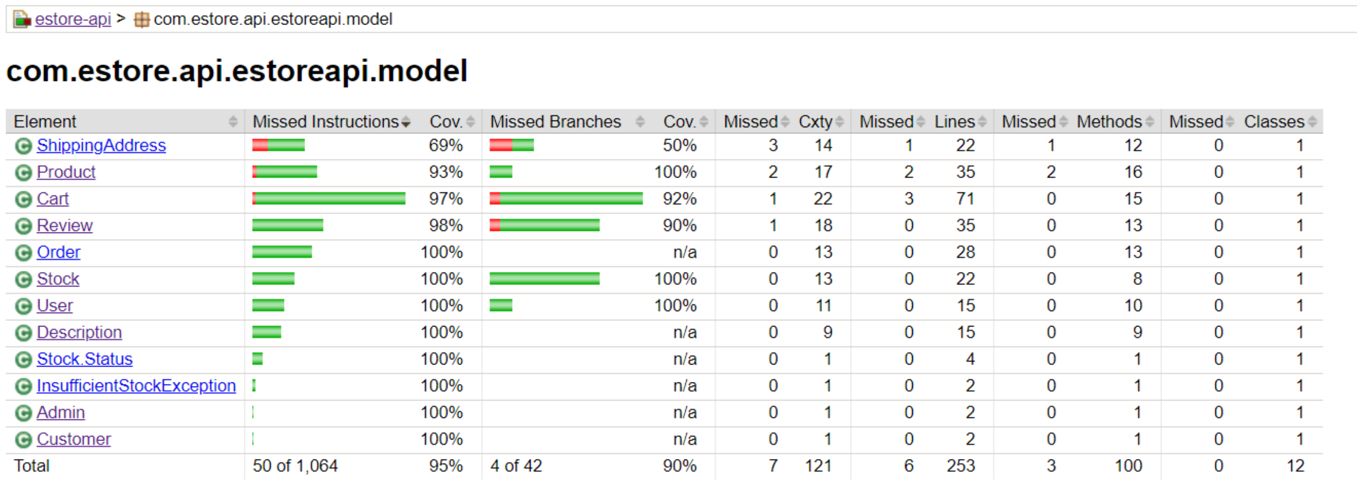
Testing

We tested our program using unit tests (guided by Jacoco), Postman, and visually in our UI. Our overall test coverage sits at 94%, with our Model Tier at 95%, our persistence tier at 95%, and our controller tier at 91%.

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 sprint 2 like adding images to the product. One functional requirement that we did fail is getting the button to remove a product from the inventory to work. This was a good practice to explore all the test cases that could break the code.

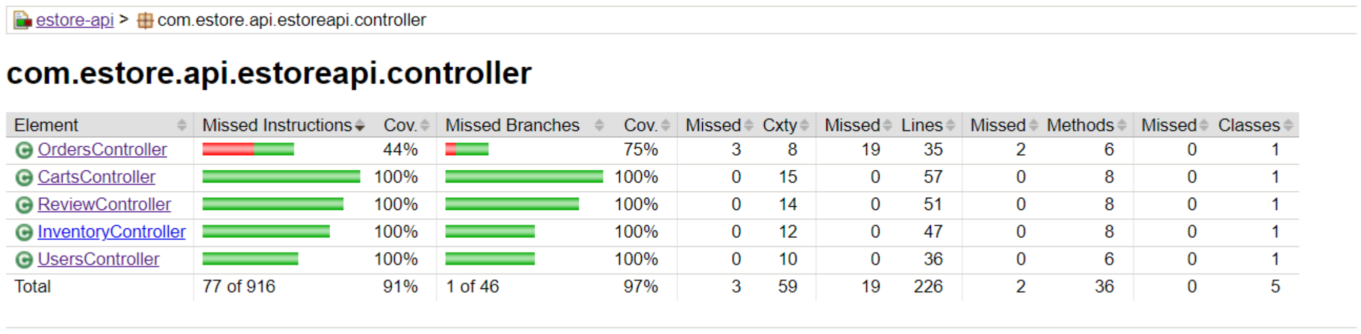
Unit Testing and Code Coverage



estore-api > com.estore.api.estoreapi.model

com.estore.api.estoreapi.model

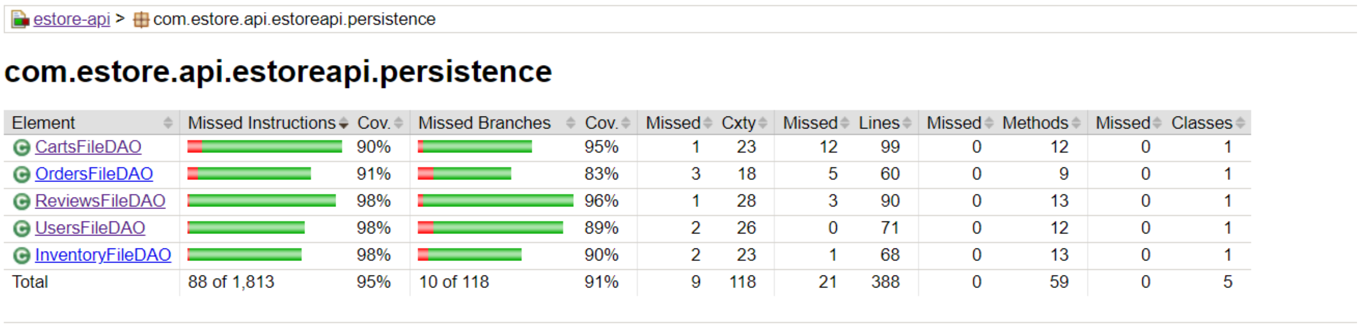
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ShippingAddress		69%		50%	3	14	1	22	1	12	0	1
Product		93%		100%	2	17	2	35	2	16	0	1
Cart		97%		92%	1	22	3	71	0	15	0	1
Review		98%		90%	1	18	0	35	0	13	0	1
Order		100%		n/a	0	13	0	28	0	13	0	1
Stock		100%		100%	0	13	0	22	0	8	0	1
User		100%		100%	0	11	0	15	0	10	0	1
Description		100%		n/a	0	9	0	15	0	9	0	1
Stock.Status		100%		n/a	0	1	0	4	0	1	0	1
InsufficientStockException		100%		n/a	0	1	0	2	0	1	0	1
Admin		100%		n/a	0	1	0	2	0	1	0	1
Customer		100%		n/a	0	1	0	2	0	1	0	1
Total	50 of 1,064	95%	4 of 42	90%	7	121	6	253	3	100	0	12



estore-api > com.estore.api.estoreapi.controller

com.estore.api.estoreapi.controller

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
OrdersController		44%		75%	3	8	19	35	2	6	0	1
CartsController		100%		100%	0	15	0	57	0	8	0	1
ReviewController		100%		100%	0	14	0	51	0	8	0	1
InventoryController		100%		100%	0	12	0	47	0	8	0	1
UsersController		100%		100%	0	10	0	36	0	6	0	1
Total	77 of 916	91%	1 of 46	97%	3	59	19	226	2	36	0	5



estore-api > com.estore.api.estoreapi.persistence

com.estore.api.estoreapi.persistence

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CartsFileDAO		90%		95%	1	23	12	99	0	12	0	1
OrdersFileDAO		91%		83%	3	18	5	60	0	9	0	1
ReviewsFileDAO		98%		96%	1	28	3	90	0	13	0	1
UsersFileDAO		98%		89%	2	26	0	71	0	12	0	1
InventoryFileDAO		98%		90%	2	23	1	68	0	13	0	1
Total	88 of 1,813	95%	10 of 118	91%	9	118	21	388	0	59	0	5

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. One of the reasons we have such high code coverage is that we ensured to make the Unit tests part of the story definition of done as is done in the industry, so for every story we had, we would make sure to have Unit tests for whatever new Java code we wrote. We initially targeted 100% code coverage, but it became difficult to follow as some of the private methods were difficult to make tests for and the tests weren't reaching all the cases that we put in some methods. Moreover, at the end of the phase, our focus shifted more to the functionality of the application than the unit test targets as we were under the pressure of the deadline. Nevertheless, overall we achieved 94% code coverage, which we feel is still great.