

PROJECT Design Documentation

Team Information

- Team name: BLACKHAWKS
- Team members
 - HAYDEN CABRAL
 - ETHAN ABBATE
 - ANGELA NGO
 - VINCENT SCHWARTZ

Executive Summary

The project is an NHL jersey store for the Chicago Blackhawks. In addition to the store itself, the project will contain the tools necessary for an administrator to control the inventory of products.

Purpose

This project is a jersey store for the NHL franchise Chicago Blackhawks. The most important user group for this project is the customer, as a vast majority of design decisions rely on making a great customer experience while shopping on the e-store.

Glossary and Acronyms

Term	Definition
SPA	Single Page
MVP	Minimum Viable Product
UI	User Interface
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
RESTful	Representational state transfer
MVVM	Model–View–ViewModel

Requirements

This section describes the features of the application.

Definition of MVP

There will be **two types of users** that will be able to interact with our website, a **Customer** and an **Admin**. There is only ONE admin and this admin can edit the inventory. Several customers can be created and can make changes to their carts before proceeding to checkout. At checkout, their carts should be empty after purchasing. Or, if they didn't proceed to checkout with jerseys in their cart, it will still be in their cart when they log back in. These changes to the backend should be reflected in the frontend user interface (UI) as well.

Epics For Main Features

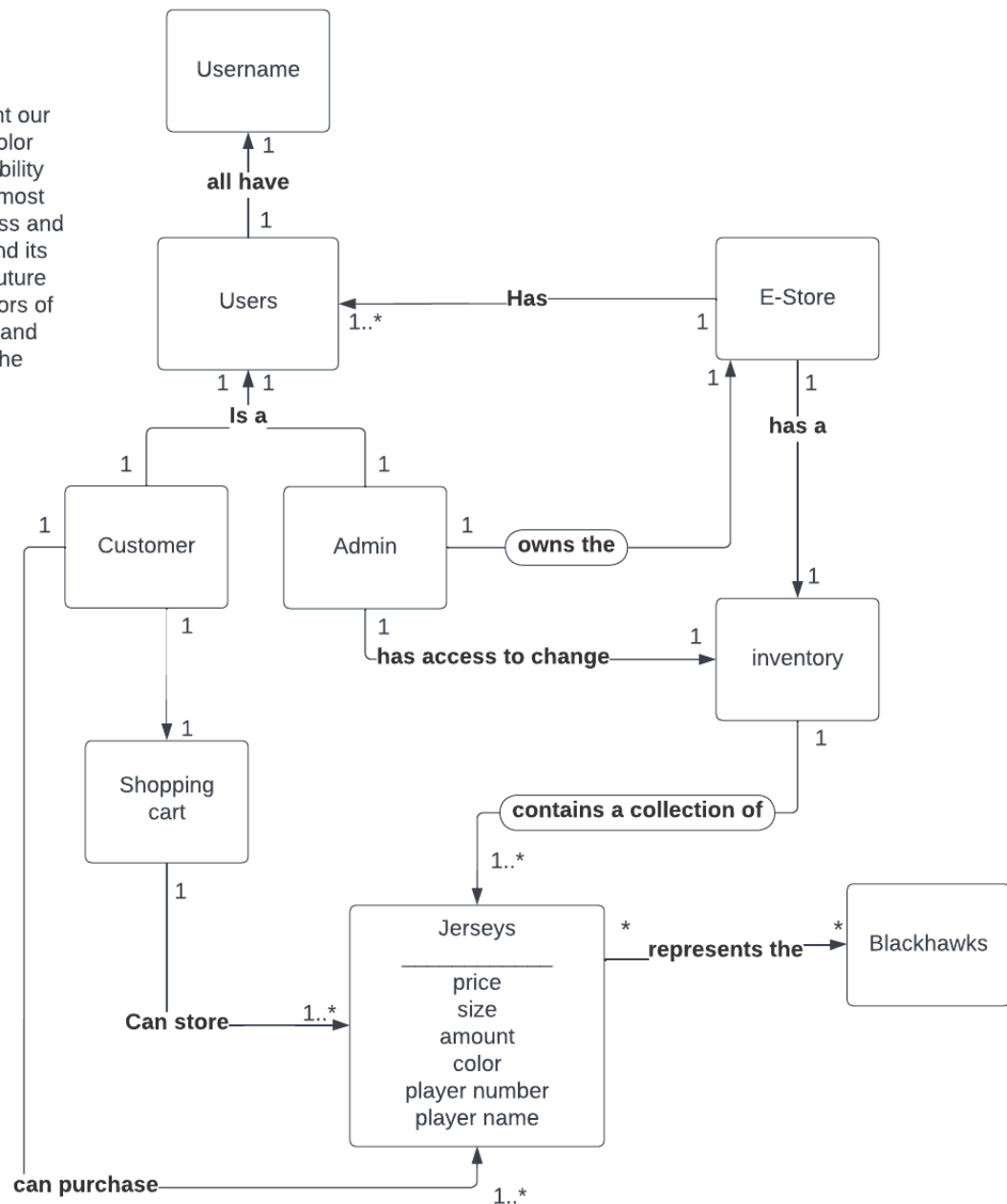
- As an Admin, I want to maintain inventory so that I can update what items I'm selling.
- An Admin must be able to maintain inventory, including possible fulfillment in accordance with their permissions.
- As a Customer I want to browse the store so that I can add or remove items to my shopping cart, check out, and see my spending.

MVP Functions

Admin	Customer	10% Feature (Color Blindness Accessibility)
can login / logout	can login / logout	have drop down to change color of buttons and text
add and remove jersey from inventory	add and remove jersey from cart	
modify a jersey	empty entire cart purchased	

Application Domain

10% feature:
as a group we want our feature to be a color blindness accessibility setting. This way most users can have access and see the website and its products. The feature will change the colors of the text, images, and background of the website.



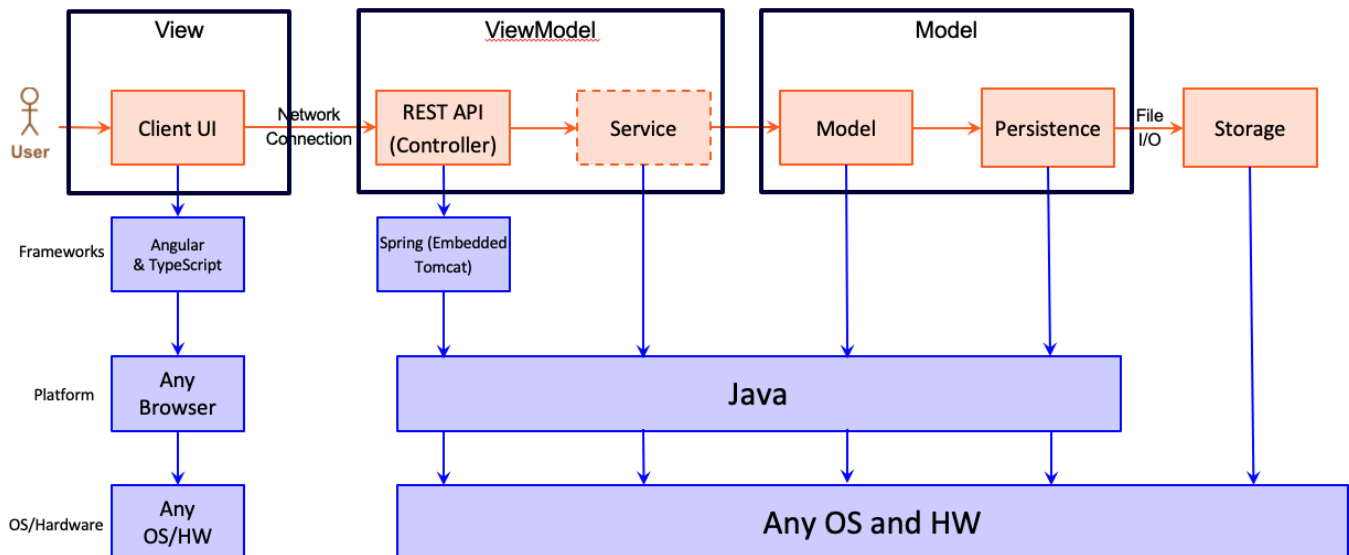
Description of Significant Objects

- **Customer** : is a user who has a shopping cart which contains jerseys
- **Admin** : is a user who has access to change the e-store's inventory which contains jerseys
- **Jerseys** : an object that contains the attributes of price, size amount etc.

Architecture and Design

Summary

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



The e-store web application, is built using the Model–View–ViewModel (MVVM) architecture pattern.

The Model stores the application data objects including any functionality to provide persistence.

The View is the client-side SPA built with Angular utilizing HTML, CSS and TypeScript. The ViewModel provides RESTful APIs to the client (View) as well as any logic required to manipulate the data objects from the Model.

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

Design Principle Analysis

Single Responsibility Principle

Where was it used?

Was used in model tier, where **Jersey** and **Customer** each have a single responsibility. A Jerseys responsibility was to create and update its data. A Customer's responsibility was to be created and edit their cart. Was also used in the Controller tier for Jersey and Customer. Their responsibility was to to handle their respective API requests and provide responses using HTTP protocols. This was also used in the front end where each component was responsible for a single thing. For example, **Admin Detail component vs Jersey Detail** component. These two are do almost the same function which is showing the details of the jersey. The only difference was that Admin Detail allowed for the information in the jerseys (i.e. price or name) to be updated by the Admin and Jersey Detail only shows the information. Since two different uses needed to do a similar functionality, we made it less complicated by making 2 different components.

Why was it used?

The Single Responsibility Principle makes your software easier to implement and prevents unexpected side-effects of future changes. If this principle is not implemented, it would make it incredibly difficult to modify and make unit tests.

Future uses?

With the way that we have our data structured, it is fine the way it is with the single responsibility principle. If we choose to further simplify our code, e.g. make a Cart class > from the Customer, I think we could implement it there. Another way we could implement it is in some of our components in the UI, but I think we did a good job with implementing that principle there.

Dependency Inversion Principle

Where was it used?

This is used in the Persistence and Controller tier of our API. The **definition** of this principle is that a high level module should not depend on a low level one. This means that both should depend on abstraction. In our Persistence an example of where this is used is the **CustomerDAO** this is an interface to **CustomerFileDAO** which actually implements the methods. However, in the Controller tier the **CustomerController** takes in the CustomerDAO instead of CustomerFileDAO as a form of *dependency injection*. Dependency injection allows for the low level module to be made outside and to be injected into the higher module through the constructor. In the front end, this principle is used when different components use services which interact with the Controllers in the API. A class receives its resources without having to create or know about them. Injectors receive instruction and instantiate a service depending on which one was requested.

Why is it used?

This is used to make testing easier. With Dependency injection, you are enabled to create mock databases and test the application without affecting the actual database.

Future uses?

This could be further implemented throughout our front end as some of the methods we used in the components were brute force methods. For example, when we had to try and calculate the total cost of a Customers cart there was not a lot of use of this principle where it should be.

Information Expert Principle

Where is it used?

This principle states that the behaviors follow the data. This is shown in our back-end in the model tier. An example would be the **Jersey** class; In this is class the Jersey is created here, therefore this means that the data (e.g. name, price, etc..) will also be here. Since that is true, the methods that have to do with the Jersey class such as accessors and mutators are also in this class. In the front-end this principle is shown in the different components we use. An example would be the **inventory component**, in this component this only has to do with Jerseys. So therefore, this class has Jersey data and can also modify them.

Why is it used?

It was used in the backend mostly because it makes sense to. It helps to determine where to delegate responsibilities such as methods and computed fields which is why it was seen frequently in our

backend. For the frontend, it is harder to say that this follows the principles thoroughly, but it is essentially the same reason.

Future uses?

This principle could be further implemented in the frontend but a lot of our components do interact with each other. Therefore, it might not be possible to implement.

Overview of User Interface

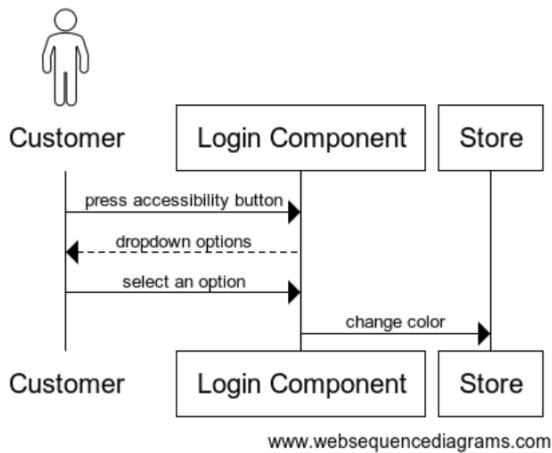
This section describes the web interface flow; this is how the user views and interacts with the e-store application.

The user interface will begin with the default page of the storefront. Once there, the user will have the option to login as a customer or admin. If an admin, they will be taken to an inventory management page. In addition, after the customer browses the store, they will navigate to a checkout page where they will be able to pay for their items.

View Tier

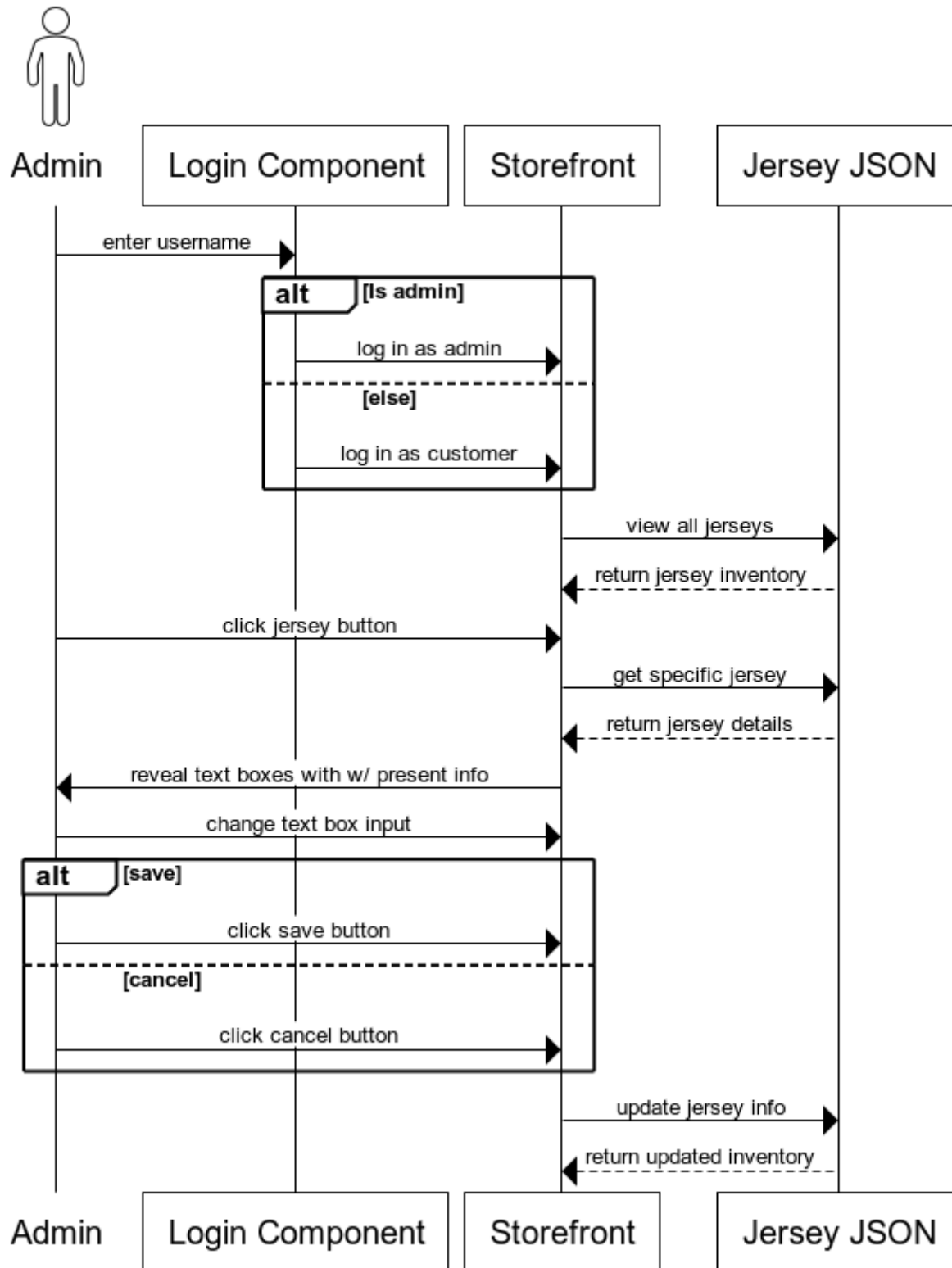
The View Tier UI of the Estore should be a cohesive, connected experience in which all components of the UI are somehow interconnected. From the very beginning of the experience, the login page registers and confirms information about the user, whether they are an admin or not, and their username. This page also holds the accessibility settings relating to the UI, so that different users of varying colorblindness can use the store. If they are an admin, denoted by the 'admin' username, they are taken to a separate page from the rest of the estore, in which they are able to add to, modify, and delete the existing inventory. Otherwise, if they are a regular user, they are taken to the main page of the estore. In this page, the user is able to browse the selection of jerseys and search for any specific one they want. Once they have decided on their selection(s), they are able to add those jerseys to their cart, which will persist if not emptied or checked out. After they view the cart, they are able to navigate to a checkout form in which the cart will empty and they will purchase their products.

Customer Changes Accessibility Settings



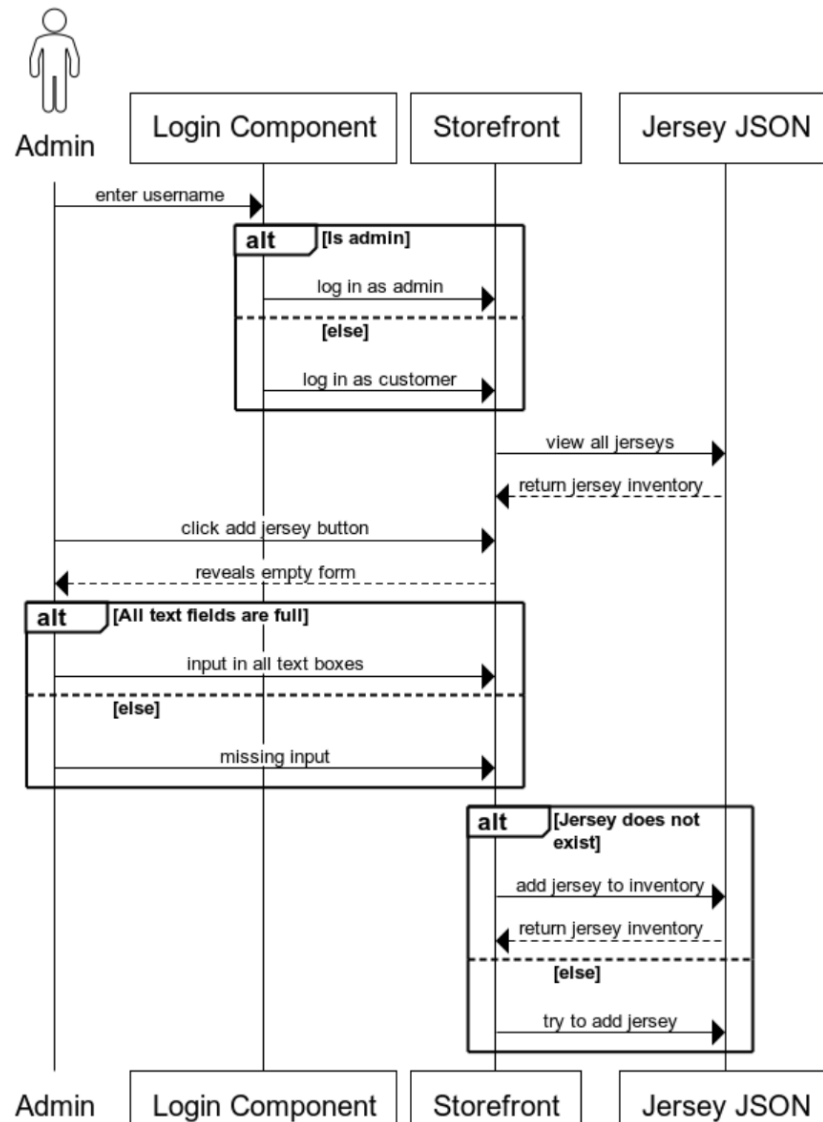
For the accessibility settings the user can press the accessibility dropdown. Then they receive the options and can select based on their colorblindness type or a high contrast mode (colorblind modes: Deuteranopia, Protanopia). Then the Jersey store will change colors based off the chosen colorblind mode.

Admin Updates Jersey

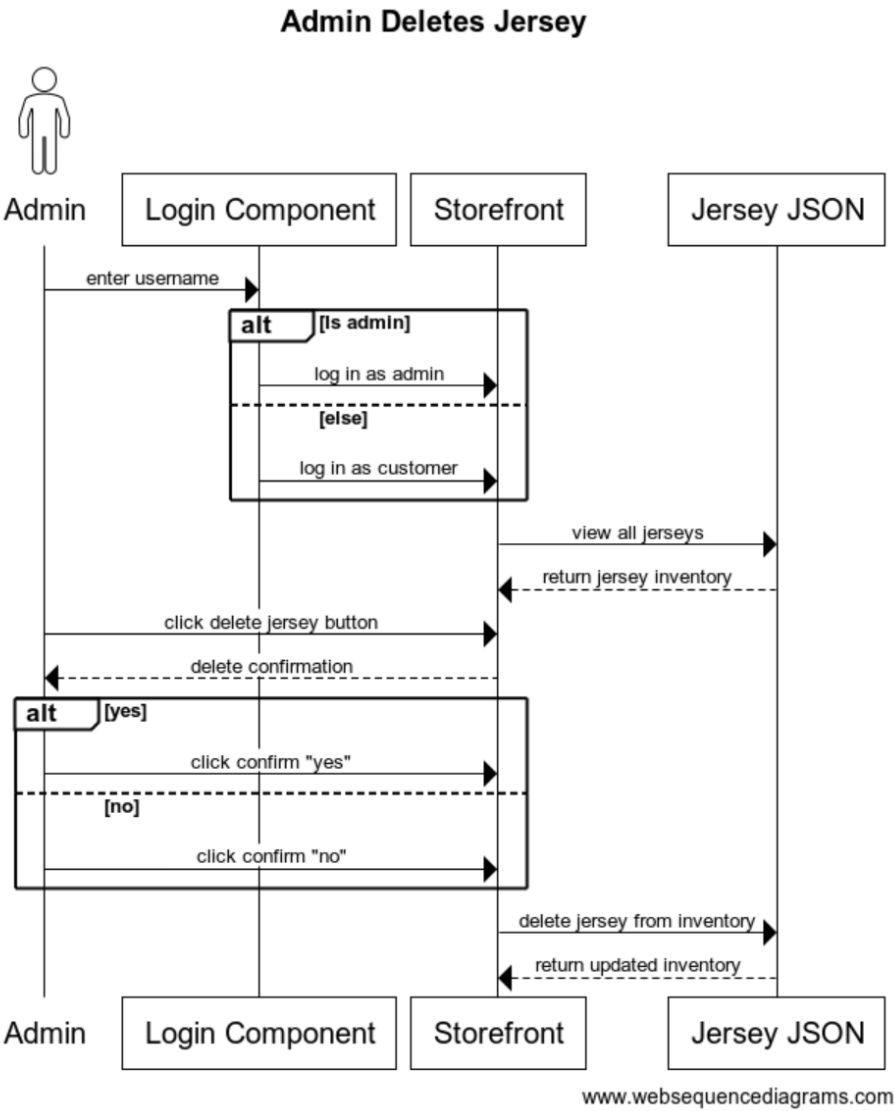


www.websequencediagrams.com

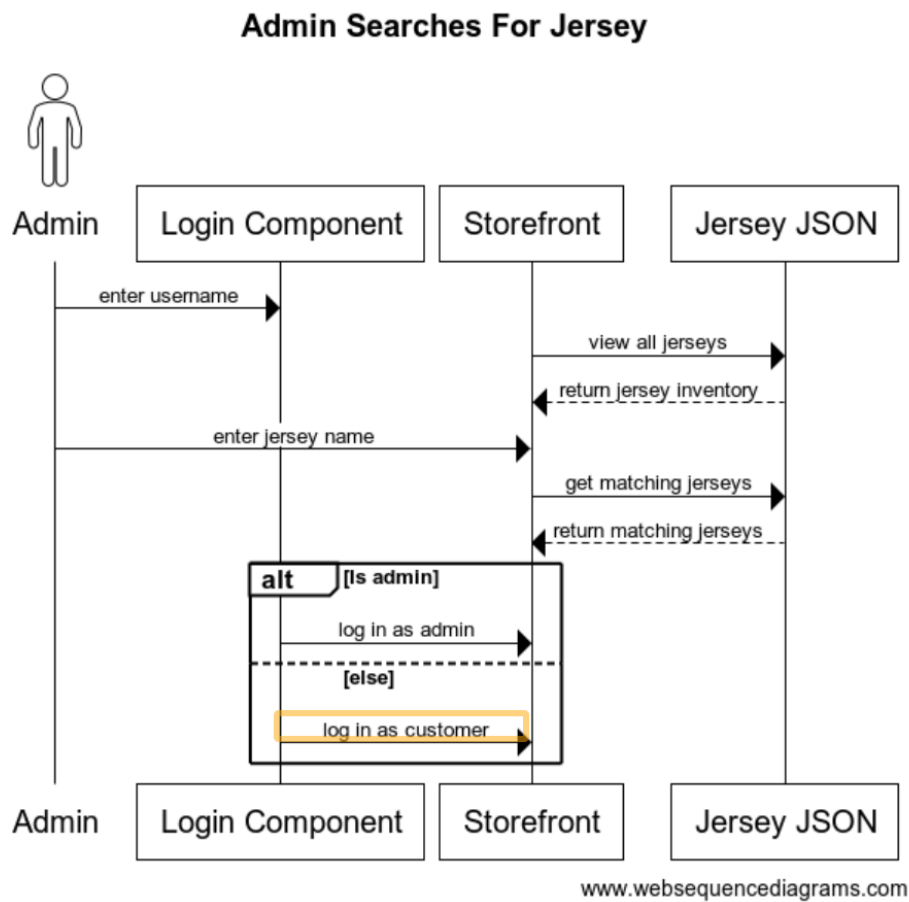
For the admin features such as updating the jersey first the admin must log in. Once they log in as an admin they will be viewing all the current jerseys that are in the inventory in the storefront. They can then click the specific jersey and then the page will switch to showing the jersey details. There will be text boxes that show the current data within the jersey and then the admin change change the details accordingly. Once the admin clicks the "save" button it will then update the JSON file containing the jerseys and then the storefront will then display the updated jersey.



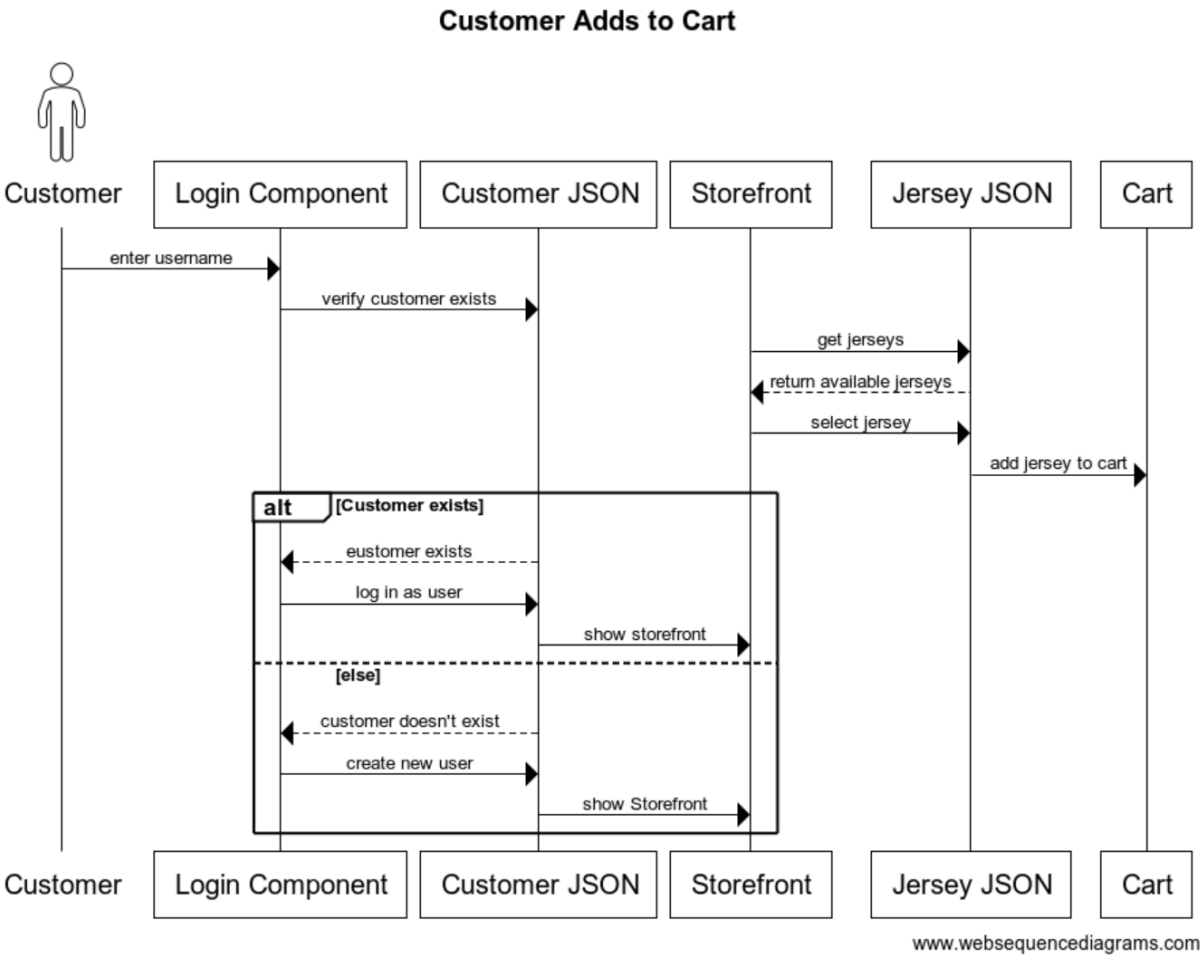
An admin will log in or again to access the inventory storefront. Then when the click the add button the website will display an empty text boxes, the admin is required to input all fields with valid data or else the jersey will not be added. Once the new jersey is going to be added to the JSON there will be a check if that exact jersey already exists, if so then it will fail to be added. Otherwise if it is a unique jersey it will be added to the inventory and displayed to the storefront.



For delete, an admin will login and view all jerseys. Then they can click a jersey and it will display the jersey details, then can click the delete button. This will delete the jersey from the JSON file and show it is no longer in the storefront.

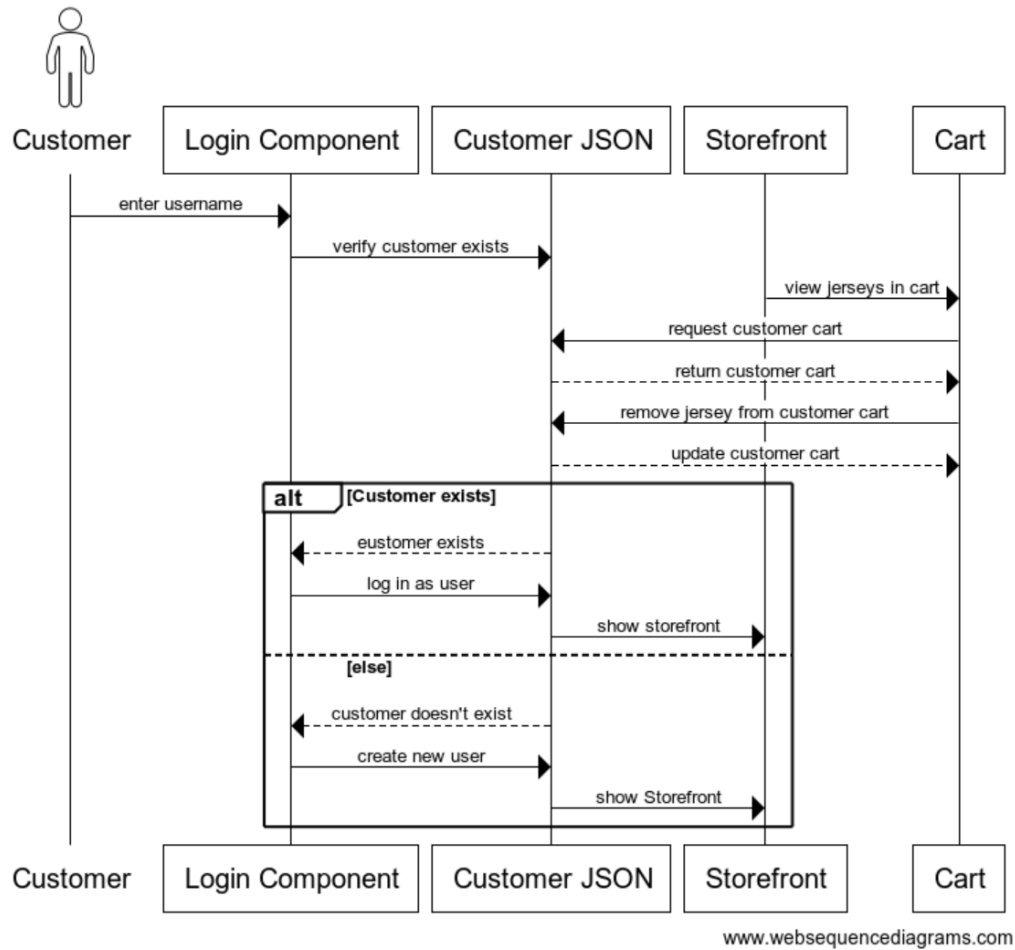


For searching for a jersey an admin will log into the store. They can then put in search terms to find a specific jersey they want to look for to either update or delete that jersey. From the search terms the jerseys that match are displayed on the storefront to the admin.



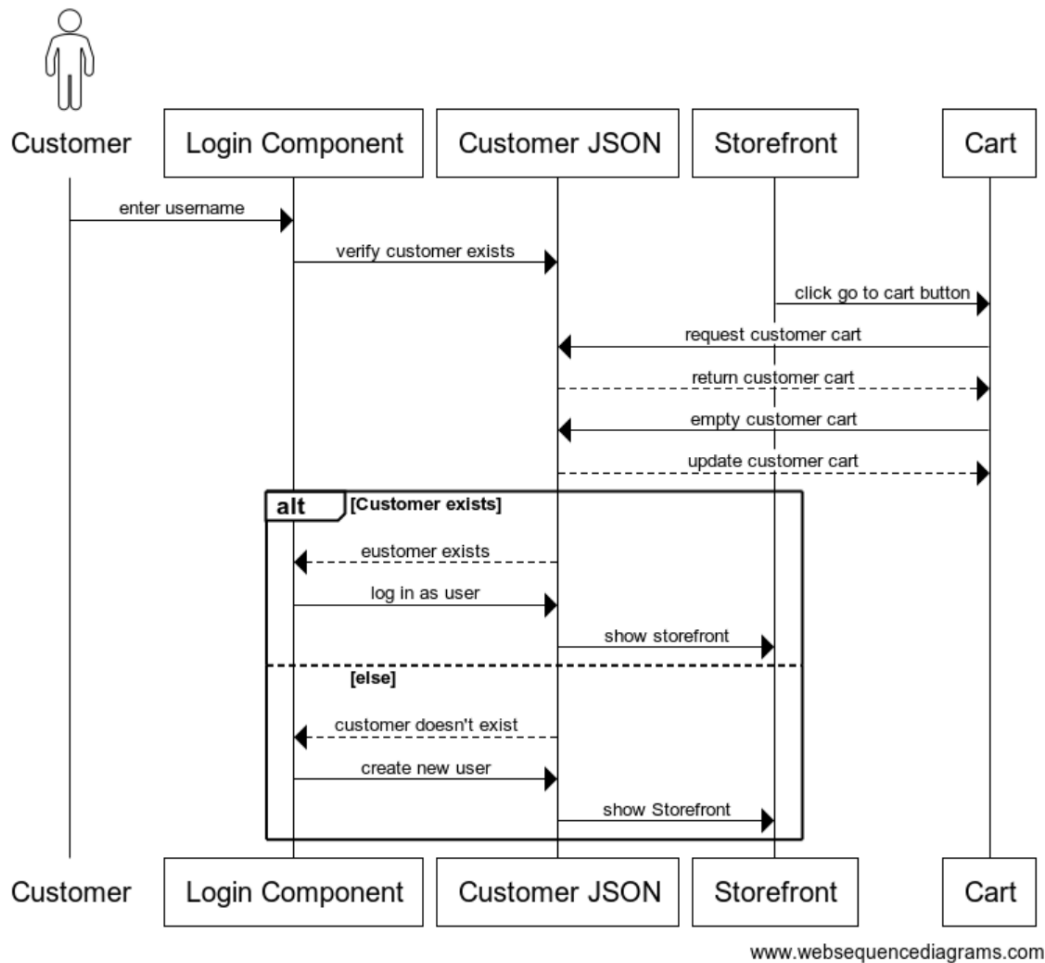
Customer will log in if they have an account, and will create a new one otherwise. After verifying the existence of the customer and logging in by typing their username, the customer is taken to the storefront. The storefront will return all available jerseys. The customer clicks on the jersey they will purchase, bringing them to the jersey's detail page. After selecting the jersey, the will input the number they want to add to the cart, and the jersey will be added to the customer's cart.

Customer Removes Jersey From Cart

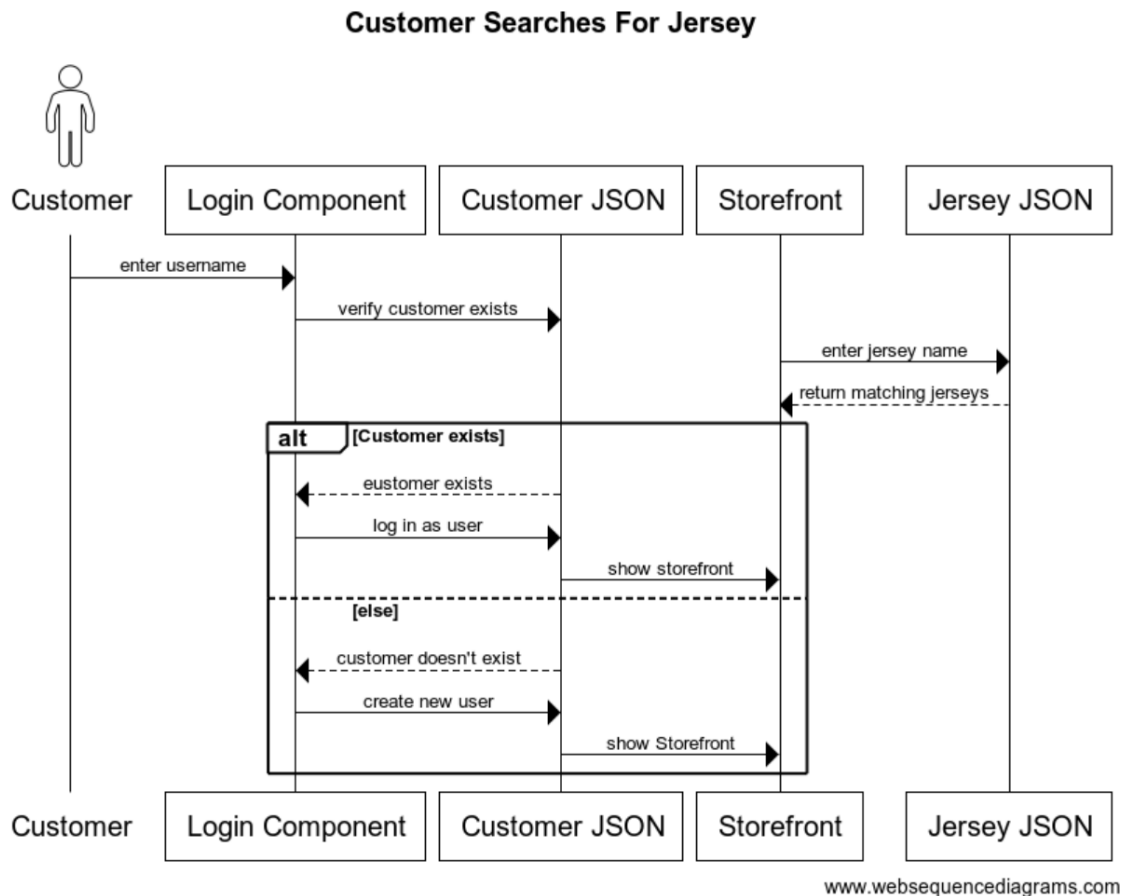


Customer will log in if they have an account, and will create a new one otherwise. After verifying the existence of the customer and logging in by typing their username, the customer is taken to the storefront. The customer then clicks the go to cart button, and will be taken to their cart with all jerseys currently in the cart displayed. After returning the requested cart, the customer removes specific jerseys from their cart by clicking the button associated with that jersey. The cart will then update and return the updated cart without the jersey that the customer deleted.

Customer Removes All Jerseys From Cart

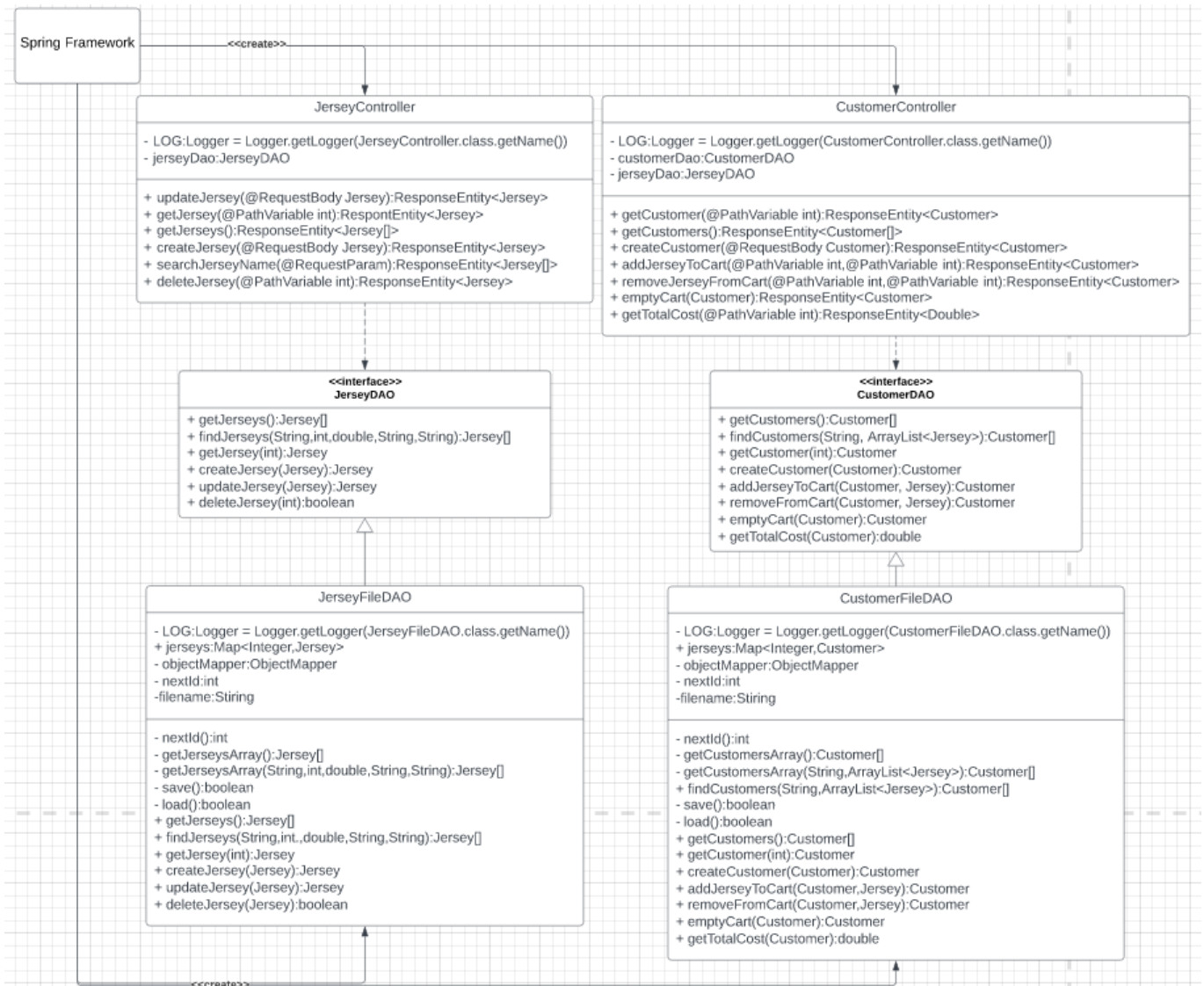


Customer will log in if they have an account, and will create a new one otherwise. After verifying the existence of the customer and logging in by typing their username, the customer is taken to the storefront. The customer then clicks the go to cart button, and will be taken to their cart with all jerseys currently in the cart displayed. After returning the requested cart, the cart will then return and update the emptied cart after the customer chooses to empty all items from it.



A Customer will log in if they have an account, and will create a new one otherwise. After verifying the existence of the customer and logging in by typing their username, the customer is taken to the storefront. The customer then enters the name of the jersey they are searching for in the search bar. The storefront will then return all matches relating to the search from the jersey JSON, and each of those results will take the customer to the details of the jersey in question.

ViewModel Tier



The main purpose of these classes is to provide persistence of the information in our json files. A shared method between these classes is *save()* and *load()*.

- **save()** - is called when making changes to the json file
- **load()** - is called when wanting to reload the json file to get its most recent changes

CustomerFileDAO

This class **inherits** from CustomerDAO and uses this in the CustomerController as a form of dependency injection

Attribute	Type	Purpose
LOG	Logger	to log messages
customers	Map<Integer, Customer>	provides a local cache of the customer objects
objectMapper	ObjectMapper	provides conversion between customer objects and JSON text
nextId	int	the next id to assign to a new customer

Attribute	Type	Purpose
filename	String	filename to read from and write to

This class is for persisting the information of each of the customers carts and other information. In this class you can find a customer, get all the customers, create a customer, add a jersey to the customer's cart, remove an item from the cart, get the total cost of the cart, and empty the entire cart. The error handling is mostly in the methods that deal with interacting with the Customer's information. So essentially, if the Customer doesn't exist in the json file, it will return null.

CustomerController

Attribute	Type	Purpose
LOG	Logger	to log messages
customerDAO	CustomerDAO	allows access to CRUD methods that change the json
jerseyDAO	JerseyDAO	allows access to methods to fetch needed jerseys

This class is for handling the REST API requests for the customer resource. In this class you can get a single customer via their id, get all existing customers, create a new customer, add a jersey to a customer's cart, remove a jersey from a customer's cart, empty a customer's cart, and get the total cost of all of the jerseys in the customer's cart. The error handling is dealt with by the DAOs and it will react by sending appropriate HTTP status codes such as 200.OK, 201.CREATED, 409.CONFLICT, 404.NOT_FOUND, and 500.INTERNAL_SERVER_ERROR

JerseyFileDAO

This class **inherits** from JerseyDAO and uses this in the JerseyController as a form of dependency injection

Attribute	Type	Purpose
LOG	Logger	to log messages
customers	Map<Integer, Jersey>	provides a local cache of the jersey objects
objectMapper	ObjectMapper	provides conversion between jersey objects and JSON text
nextId	int	the next id to assign to a new jersey
filename	String	filename to read from and write to

This class is for persisting the information of each of the jerseys. In this class you can find a jersey, get all the jerseys, create a jersey, get a single jersey, delete a jersey, and update a jersey. The error handling is mostly in the > methods that deal with interacting with the jersey's information. So essentially, if the jersey doesn't exist in the json > file, it will return null.

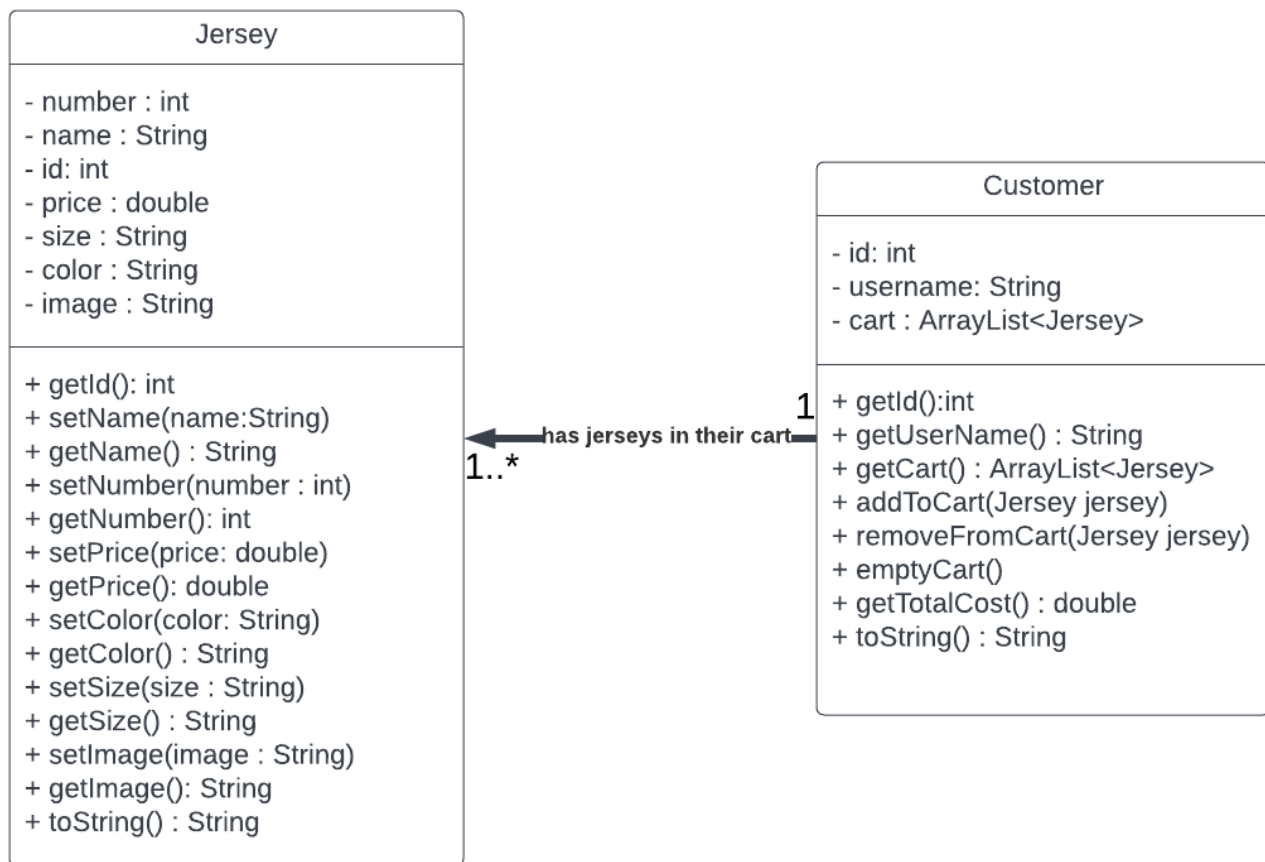
JerseyController

Attribute	Type	Purpose
-----------	------	---------

Attribute	Type	Purpose
LOG	Logger	to log messages
jerseyDAO	JerseyDAO	allows access to CRUD methods that change the json

This class is for handling the REST API requests for the jersey resource. In this class you can get a single jersey via its id, get all existing jerseys, create a new jersey, update an existing jersey, and delete an existing jersey. The error handling is dealt with by the JerseyDAO and it will react by sending appropriate HTTP status codes such as 200.OK, 201.CREATED, 409.CONFLICT, 404.NOT_FOUND, and 500.INTERNAL_SERVER_ERROR

Model Tier



There are two classes that make up our object, that will be the Jersey and Customer class.

Jersey Class

The jersey class consists of the following attributes:

1. Id of jersey
2. Name
3. Size
4. Color
5. Number (on the jersey)
6. Price

7. Image These attributes can be accessed through accessors and can be changed through mutators.

Customer Class

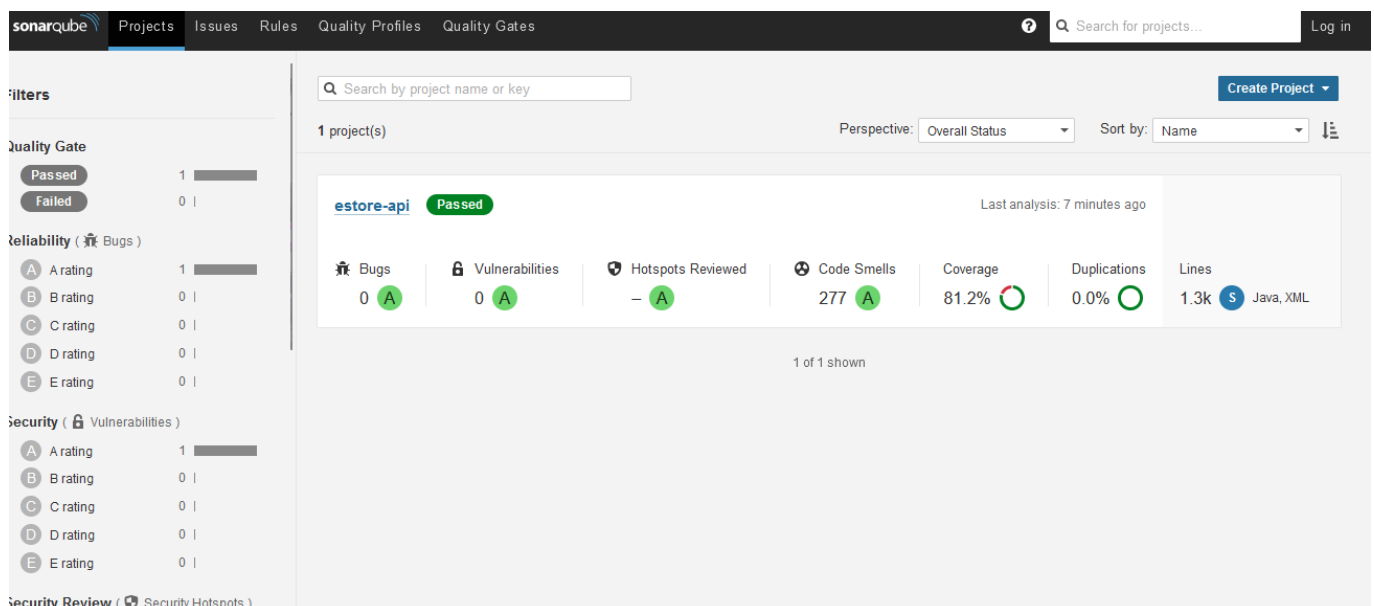
In the customer class, there are 3 attributes: id, username and cart. These attributes have various methods in them to access and mutators in them. The only methods that are different from average mutators are the ones for the cart attribute. Since cart is an ArrayList of Jerseys, the user can choose to remove or add a jersey to the cart or remove all instances of the jersey from the cart.

special methods

- totalCost() method
 - that gets the total cost from iterating through the cart arraylist and adding up the prices.
- EmptyCart() method
 - a special method that empties out all the instances of the Jerseys in the cart ArrayList

The way that the two classes interact is that the Customer class has a cart that can contain Jerseys, since the Customer can buy different Jerseys.

Static Code Analysis/Design Improvements



Overall our API had good coverage, no duplications, bugs or vulnerabilities. One thing that can be improved on our API is that there is many code smells. Most of the major issues were issues with large blocks of code that were unused were commented out instead of deleted. In order to keep our code clean we should have deleted most of these unused blocks. Some of the critical issues were about static methods or hardcoded strings so it would be a good idea to make sure we are using java principles like static right with some of the methods. It would be a good idea to try and get more code coverage as well.

Critical issues

src/.../api/estoreapi/persistence/CustomerFileDAO.java

<input type="checkbox"/>	Make the enclosing method "static" or remove this set.	25 days ago ▾ L131 🔗 ⚙️ ▾
	Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 20min effort Comment	multi-threading ▾

src/.../api/estoreapi/persistence/JerseyFileDAO.java

<input type="checkbox"/>	Make the enclosing method "static" or remove this set.	2 months ago ▾ L121 🔗 ⚙️ ▾
	Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 20min effort Comment	multi-threading ▾
<input type="checkbox"/>	Make the enclosing method "static" or remove this set.	2 months ago ▾ L132 🔗 ⚙️ ▾
	Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 20min effort Comment	multi-threading ▾
<input type="checkbox"/>	Make the enclosing method "static" or remove this set.	2 months ago ▾ L135 🔗 ⚙️ ▾
	Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 20min effort Comment	multi-threading ▾

Where is the issue?

Why is this an issue?

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

143 ngo.a...
144 ngo.a...
145

146
147 ngo.a...
148
149 /**
150 ngo.a...
151 ngo.a...

```
}  
  
// Make the next id one greater than the maximum from the file  
++nextId;  
  
  
return true;  
}
```

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

Here, a lot of our issues came from not declaring some variables to be static. Particularly, our nextId variable was marked. it was marked because it thought that by not making it static it could lead to errors with creating / referencing the id for the Customer or jersey.

Major issues

src/.../api/estoreapi/controller/CustomerController.java

<input type="checkbox"/>	Use the built-in formatting to construct this argument.	13 hours ago ▾ L57 🔗 🗑️
	Code Smell ▾ Major ▾ Open ▾ Not assigned ▾ 5min effort Comment	performance ▾
<input type="checkbox"/>	Use the built-in formatting to construct this argument.	13 hours ago ▾ L105 🔗 🗑️
	Code Smell ▾ Major ▾ Open ▾ Not assigned ▾ 5min effort Comment	performance ▾
<input type="checkbox"/>	Use the built-in formatting to construct this argument.	13 hours ago ▾ L130 🔗 🗑️
	Code Smell ▾ Major ▾ Open ▾ Not assigned ▾ 5min effort Comment	performance ▾
<input type="checkbox"/>	Use the built-in formatting to construct this argument.	13 hours ago ▾ L162 🔗 🗑️
	Code Smell ▾ Major ▾ Open ▾ Not assigned ▾ 5min effort Comment	performance ▾
<input type="checkbox"/>	This block of commented-out lines of code should be removed.	13 hours ago ▾ L167 🔗 🗑️
	Code Smell ▾ Major ▾ Open ▾ Not assigned ▾ 5min effort Comment	unused ▾
<input type="checkbox"/>	Use the built-in formatting to construct this argument.	13 hours ago ▾ L192 🔗 🗑️
	Code Smell ▾ Major ▾ Open ▾ Not assigned ▾ 5min effort Comment	performance ▾

estore-api src/.../java/com/estore/api/estoreapi/controller/JerseyController.java

[See all issues in this file](#)

```

223 vas86... @GetMapping("/searchByPrice/")
224 ngo. a... // @RequestMapping(value="/jerseys/?color={color}", method = RequestMethod.GET)
225 vas86... public ResponseEntity<Jersey[]> searchJerseyPrice(@RequestParam double price) {
226         LOG.info("GET /jerseys/searchByPrice/?price="+price);

```

Use the built-in formatting to construct this argument.

```

227 ngo. a...
228         try {
229 vas86...             Jersey [] jerseys = jerseyDao.findJerseys(null, 0, price, null, null);
230 ngo. a...             return new ResponseEntity<Jersey[]>(jerseys, HttpStatus.OK);
231         }
232         catch(IOException e) {

```

Here, a lot of our issues came from not using the built in formatting for to construct the string that is going to be used in the Logger function for our controller. It is a matter of preference I believe of this choice.

Minor Issues

src/.../api/estoreapi/controller/CustomerController.java

Replace the type specification in this constructor call with the diamond operator ("<>").

13 hours ago

L61

Code Smell

Minor

Open

Not assigned

1min effort

Comment

clumsy

Replace the type specification in this constructor call with the diamond operator ("<>").

13 hours ago

L84

Code Smell

Minor

Open

Not assigned

1min effort

Comment

clumsy

Replace the type specification in this constructor call with the diamond operator ("<>").

13 hours ago

L109

Code Smell

Minor

Open

Not assigned

1min effort

Comment

clumsy

Replace the type specification in this constructor call with the diamond operator ("<>").

13 hours ago

L137

Code Smell

Minor

Open

Not assigned

1min effort

Comment

clumsy

Replace the type specification in this constructor call with the diamond operator ("<>").

20 days ago

L168

Code Smell

Minor

Open

Not assigned

1min effort

Comment

clumsy

Replace the type specification in this constructor call with the diamond operator ("<>").

13 hours ago

L198

Code Smell

Minor

Open

Not assigned

1min effort

Comment

clumsy

estore-api

src/.../java/com/estore/api/estoreapi/controller/CustomerController.java

See all issues in this file

81 ngo.a... try {

82 ngo.a... Customer[] customers = customerDao.getCustomers();

83 if(customers != null) {

84 return new ResponseEntity<Customer[]>(customers, HttpStatus.OK);

85 } else {

86 return new ResponseEntity<>(customers, HttpStatus.NOT_FOUND);

87 } catch(IOException e) {

88 LOG.log(Level.SEVERE,e.getMessage());

89 }

Replace the type specification in this constructor call with the diamond operator ("<>").

Here a lot of the problems came from having to replace the type specification in this constructor call with the diamond operator ("<>"). however, their explanation doesnt make that much sense in regards to why you should just have it be empty instead of filled. While its understandable they thought it was a problem because java automatically does this for you, its not that much of a problem in reality.

UI Static Code Analysis

estore-ui-nov21

master

November 21, 2022 at 9:30 AM

Version not provided

Overview

Issues

Security Hotspots

Measures

Code

Activity

Project Settings

Project Information

Passed

All conditions passed.

New Code

Overall Code

17 Bugs

Reliability C

0 Vulnerabilities

Security A

0 Security Hotspots

Reviewed

Security Review A

3h 56min Debt

75 Code Smells

Maintainability A

0.0% Coverage on 348 Lines to cover

- Unit Tests

0.0% Duplications on 2.1k Lines

0 Duplicated Blocks

Overall our UI did have a good number of bugs but everything else was good. The bugs boil down to adding header tags or description to some of the tables that we used within our UI. Some other issues were some

22 / 28

commented out code and deprecated attributes in the css. In the future it would be good to be more descriptive and use good standards when it comes to certain HTML elements such as tables. We also need to utilize the power of Angular better and understand it more in order to be able to work more effectively.

Critical issues

src/app/login/login.component.ts

Unexpected var, use let or const instead.

15 days ago L67

Code Smell Critical Open Not assigned 5min effort [Comment](#)

bad-practice, es2015

Unexpected var, use let or const instead.

15 days ago L68

Code Smell Critical Open Not assigned 5min effort [Comment](#)

bad-practice, es2015

Unexpected var, use let or const instead.

6 days ago L90

Code Smell Critical Open Not assigned 5min effort [Comment](#)

bad-practice, es2015

estore-ui-nov21 src/app/login/login.component.ts

[See all issues in this file](#)

64 * @returns true if admin, false is customer

65 */

66 isAdmin(): boolean {

67 var name = this.onSubmit().toLowerCase();

Unexpected var, use let or const instead.

68 var result = name == "admin";

Unexpected var, use let or const instead.

69 if (name == "") {

70 alert("Enter a valid username")

Here, a lot of our issues came declaring our variables as VAR instead of using the LET key word. This is critical because it may become a source of scope issues. What that means is that let and var have different scopes, so if "var" was used to declare a variable that needs to be used often, it may run into issues.

Major issues

src/app/inventory/inventory.component.html

☐ Add "<th>" headers to this "<table>".

4 days ago ▾ L11 🔗 🔽
🐞 Bug ▾ 🚩 Major ▾ 🔵 Open ▾ Not assigned ▾ 2min effort Comment
🔗 accessibility, wcag2-a ▾

☐ Remove this commented out code.

15 days ago ▾ L36 🔗 🔽
🔗 Code Smell ▾ 🚩 Major ▾ 🔵 Open ▾ Not assigned ▾ 5min effort Comment
🔗 unused ▾

☐ Remove this commented out code.

15 days ago ▾ L55 🔗 🔽
🔗 Code Smell ▾ 🚩 Major ▾ 🔵 Open ▾ Not assigned ▾ 5min effort Comment
🔗 unused ▾

estore-ui-nov21 src/app/inventory/inventory.component.html See all issues in this file 🔗

8 <h3>Current Inventory:</h3>

9

10 <div>

11 vas86... <table>

🐞 Add "<th>" headers to this "<table>".

12 <ng-container *ngFor="let jersey of jerseys; let i=index;">

13 <tr *ngIf="i % 3 == 0">

14 <ng-container *ngFor="let pos of [0, 1, 2]">

15 <td *ngIf="i+pos < jerseys.length">

16 89423... <div class = "jersey">

17

Here, a lot of our issues came from not deleting commented out code. This causes crowding and limits readability. The more in depth picture talks about adding headers to the table. This is not that big of a deal because this also just promotes readability. it provides some context when > users navigates a table. Without it the user gets rapidly lost in the flow of data.

Minor Issues

src/app/admin-detail/admin-detail.component.html

☐ Add a description to this table.

15 days ago ▾ L11 🔗 🔽
🐞 Bug ▾ 🟡 Minor ▾ 🔵 Open ▾ Not assigned ▾ 5min effort Comment
🔗 accessibility, wcag2-a ▾

src/app/admin-detail/admin-detail.component.ts

☐ Remove this unused import of 'Input'.

15 days ago ▾ L1 🔗 🔽
🔗 Code Smell ▾ 🟡 Minor ▾ 🔵 Open ▾ Not assigned ▾ 2min effort Comment
🔗 es2015, unused ▾

estore-ui-nov21 src/app/admin-detail/admin-detail.component.html See all issues in this file 🔗

8 <h3>Modify Jersey</h3>

9

10 <div *ngIf="jersey" align="center">

11 <table align="center" border="1" cellpadding="1" cellspacing="1" style="width:500px">

🐞 Add a description to this table.

12 <tbody>

13 <tr>

14 <td class="image" rowspan="4" style="width:234px">

15

16 </td>

17 <td class="name" style="width:253px">{{jersey.name}} #{{jersey.number}}</td>

Here a lot of the problems came from not taking out unused import statements in the components. This can be a problem, but we didn't have time to edit our files in lieu of testing. We also had some minor issues about the table again. It's mostly about syntax which is why this code smell was picked up.

It states that it prefers a description so that visually impaired people will be able to understand what the table is trying to communicate which i think it helpful.

Roadmap of Enhancements

Since the majority of our issues are for purely reason to make the code more readable, the next step after we finish fixing the rest of our semantic bugs is to edit our code to take out commented out code and follow standard formatting practices. These formatting practices are helpful because it helps with limited errors as following one guide line does that. Another thing that would be practical to implement based off our prior analysis is improving upon the accessibility of the code. This is mostly in the UI as one of the code smells we had was about adding a header for our table to provide a description of what it is to a visually impaired user. This would benefit us because our 10% feature has to do with accessibility, so therefore we could stand to edit our code to make it that way. Other changes that we propose is to add everal aesthetic and functional improvements such as REACT, storing our jerseys in a database using POSTGRGSQL, and others to make a more streamlined and professional experience.

Testing

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

Acceptance Testing

How many user stories have...	Number
passed	25
some acceptance criteria failing	6
that havent been tested yet	0

Issues During Acceptance testing

- There is currently some bug with adding more than one jersey to the cart (in quantity section) for whatever reason its not consistent and we weren;t sure how to fix that bug
- There is an issue with getting the total cost to update as you remove from your cart. It still works but sometimes it can be inconsistent and not update right after. You would have to go back a page and then go back to the cart to see the change
- There are sometimes inconsistancies with how the cart is being rendered. We have tried a lot of different ways to see if it would change by calling `getCart()` (which reupdates the cart in the `cartComponent`) in different spots to try and see if it fixes it but it fails to.

Unit Testing and Code Coverage

Unit Testing Strategy

The goal we want to achieve is 90%-100% code coverage. In order to get to that goal, we have decided to do thorough testing in order to get the highest coverage possible. This is > our strategy thus far.











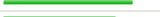
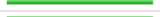






1. Work on one method at a time
2. Make sure to write tests for the happy path and then the "unhappy path"
3. Make sure that you write tests that contain errors to see if the method is able to catch and deal with them
4. If there is a problem with debugging what went wrong in unit test, consult other group members or professor for help

Code Coverage As of 11/15/22

Controller Tier


















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

CustomerController

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
emptyCart(int)		48%		50%	1	2	5	10	0	1
getTotalCost(int)		53%		50%	1	2	4	9	0	1
addJerseyToCart(int, int)		60%		50%	1	2	4	9	0	1
removeJerseyFromCart(int, int)		60%		50%	1	2	4	9	0	1
createCustomer(Customer)		100%		100%	0	2	0	9	0	1
getCustomer(int)		100%		100%	0	2	0	8	0	1
getCart(int)		100%		100%	0	2	0	8	0	1
getCustomers()		100%		100%	0	2	0	8	0	1
CustomerController(CustomerDAO, JerseyDAO)		100%	n/a	n/a	0	1	0	4	0	1
static {...}		100%	n/a	n/a	0	1	0	1	0	1
Total	74 of 318	76%	4 of 16	75%	4	18	17	75	0	10

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































JerseyController

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
updateJersey(Jersey)		100%		100%	0	2	0	9	0	1
deleteJersey(int)		100%		100%	0	2	0	7	0	1
getJersey(int)		100%		100%	0	2	0	8	0	1
createJersey(Jersey)		100%		100%	0	2	0	8	0	1
getJerseys()		100%		100%	0	2	0	8	0	1
searchJerseyName(String)		100%	n/a	n/a	0	1	0	6	0	1
searchJerseyNumber(int)		100%	n/a	n/a	0	1	0	6	0	1
searchJerseyPrice(double)		100%	n/a	n/a	0	1	0	6	0	1
searchJerseyColor(String)		100%	n/a	n/a	0	1	0	6	0	1
searchJerseySize(String)		100%	n/a	n/a	0	1	0	6	0	1
JerseyController(JerseyDAO)		100%	n/a	n/a	0	1	0	3	0	1
static {...}		100%	n/a	n/a	0	1	0	1	0	1
Total	0 of 331	100%	0 of 10	100%	0	17	0	74	0	12

Persistence Tier



























estore-api > com.estimate.estimateapi.persistence > CustomerFileDAO

CustomerFileDAO

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
createCustomer(Customer)		100%		100%	0	7	0	10	0	1
getCustomersArray(String, ArrayList)		100%		100%	0	2	0	8	0	1
load()		100%		100%	0	2	0	8	0	1
addJerseyToCart(Customer, Jersey)		100%		100%	0	2	0	8	0	1
removeFromCart(Customer, Jersey)		100%		100%	0	2	0	8	0	1
emptyCart(Customer)		100%		100%	0	2	0	8	0	1
getCustomer(int)		100%		100%	0	2	0	4	0	1
getTotalCost(Customer)		100%		100%	0	1	0	4	0	1
save()		100%		100%	0	1	0	3	0	1
CustomerFileDAO(String, ObjectMapper)		100%		100%	0	1	0	5	0	1
findCustomers(String, ArrayList)		100%		100%	0	1	0	2	0	1
getCustomers()		100%		100%	0	1	0	2	0	1
nextId()		100%		100%	0	1	0	3	0	1
getCustomersArray()		100%		100%	0	1	0	1	0	1
static {...}		100%		100%	0	1	0	1	0	1
Total	4 of 386	98%	2 of 28	92%	2	29	1	77	0	15

estore-api > com.estimate.estimateapi.persistence > JerseyFileDAO



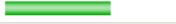








JerseyFileDAO

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
createJersey(Jersey)		70%		25%	5	7	5	14	0	1
getJerseysArray(String, int, double, String, String)		100%		100%	0	12	0	12	0	1
load()		100%		75%	1	3	0	9	0	1
updateJersey(Jersey)		100%		100%	0	2	0	7	0	1
deleteJersey(int)		100%		100%	0	2	0	6	0	1
getJersey(int)		100%		100%	0	2	0	4	0	1
findJerseys(String, int, double, String, String)		100%		100%	0	1	0	2	0	1
save()		100%		100%	0	1	0	3	0	1
JerseyFileDAO(String, ObjectMapper)		100%		100%	0	1	0	5	0	1
getJerseys()		100%		100%	0	1	0	2	0	1
nextId()		100%		100%	0	1	0	3	0	1
getJerseysArray()		100%		100%	0	1	0	1	0	1
static {...}		100%		100%	0	1	0	1	0	1
Total	27 of 371	92%	10 of 44	77%	6	35	5	69	0	13

Model Tier














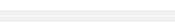


estore-api > com.estore.api.estoreapi.model > Customer

Customer

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• getTotalCost()		100%		100%	0	2	0	4	0	1
• Customer(int, String)		100%	n/a	n/a	0	1	0	5	0	1
• addToCart(Jersey)		100%	n/a	n/a	0	1	0	1	0	1
• removeFromCart(Jersey)		100%	n/a	n/a	0	1	0	1	0	1
• static {...}		100%	n/a	n/a	0	1	0	1	0	1
• emptyCart()		100%	n/a	n/a	0	1	0	1	0	1
• getId()		100%	n/a	n/a	0	1	0	1	0	1
• getUserName()		100%	n/a	n/a	0	1	0	1	0	1
• getCart()		100%	n/a	n/a	0	1	0	1	0	1
• toString()		100%	n/a	n/a	0	1	0	1	0	1
Total	0 of 69	100%	0 of 2	100%	0	11	0	17	0	10

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

Jersey

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• Jersey(int, String, int, double, String, String, String)		100%	n/a	n/a	0	1	0	9	0	1
• toString()		100%	n/a	n/a	0	1	0	1	0	1
• static {...}		100%	n/a	n/a	0	1	0	1	0	1
• setName(String)		100%	n/a	n/a	0	1	0	1	0	1
• setNumber(int)		100%	n/a	n/a	0	1	0	1	0	1
• setPrice(double)		100%	n/a	n/a	0	1	0	1	0	1
• setColor(String)		100%	n/a	n/a	0	1	0	1	0	1
• setSize(String)		100%	n/a	n/a	0	1	0	1	0	1
• setImage(String)		100%	n/a	n/a	0	1	0	1	0	1
• getId()		100%	n/a	n/a	0	1	0	1	0	1
• getName()		100%	n/a	n/a	0	1	0	1	0	1
• getNumber()		100%	n/a	n/a	0	1	0	1	0	1
• getPrice()		100%	n/a	n/a	0	1	0	1	0	1
• getColor()		100%	n/a	n/a	0	1	0	1	0	1
• getSize()		100%	n/a	n/a	0	1	0	1	0	1
• getImage()		100%	n/a	n/a	0	1	0	1	0	1
Total	0 of 90	100%	0 of 0	n/a	0	16	0	24	0	16

Statement

As of right now, our code coverage has significantly met our expectations. The controller tier has been tested and the Customer Controller was the only main class that lacked code coverage. Everything else was almost perfect. Checking over these classes and refactoring our code has led to us solving our previous problems we had with Customer and how to manage their cart.