

PROJECT Design Documentation

Team Information

- Team: 3-C
- Team name: 5Guys
- Team members
 - Jack Yakubison jcy4561
 - Adrian Burgos awb8593
 - Eligh Ros edr5068
 - Trent Wesley taw8452
 - Jack Hester jrh339

Executive Summary

This project's purpose is to act as an E-Store used for renting monkeys. It is a website on which users can search, sort, and rent monkeys for specified dates. It can also be used by an admin account to create new monkey listings, edit monkey listings, or delete monkey listings. The E-Store offers a review feature in which users who have rented a monkey can write a review to leave on the monkey's page, which other users will be able to see.

Purpose

The main goal of this project was to provide both a front-end for an e-store focused on renting monkeys for parties and an API to handle inventory management and data persistence. Another goal of this project is to handle authentication for both customers and owners, as well as a working shopping cart for customers to store desired products in until they are ready to proceed with renting.

Glossary and Acronyms

Term	Definition
SPA	Single Page Application
MVP	Minimum Viable Product
DAO	Data Access Object
API	Application Programming Interface
REST	Representational State Transfer
MVVM	Model–View–ViewModel
UI	User Interface

Requirements

Our application was required to provide users with the ability to rent various monkeys and the owners with the ability to actively manage the site through admin control. We were required to develop controls for the

owners to utilize to effectively manage the e-store, such as creating, deleting, and updating the monkeys within the store. We were also required to effectively build a website from the front-end, including product pages that show the details of each monkey, a way to search through the list of monkeys by means of a search bar and filters, and a shopping cart to checkout. In addition, we had to create a system for customers and owners to log in, as well as a way for customers to post reviews of monkeys they previously rented.

Definition of MVP

The minimum viable product for this project is a running website in which users have access to a variety of monkeys, which they can add to a shopping cart and request to rent. It should also have an admin account that can change the inventory of monkeys available on the website. Lastly, users should be able to write reviews for monkeys that they rented that can be accessed by other users.

MVP Features

- Login Backend
- Login Page
- Owner Features
- Product Page
- Search for product
- Get a product
- Update a product
- Create a new product
- Delete a product
- Search bar
- Enable Filters
- Shopping Cart
- Rental Backend
- Post Review
- Read Reviews

10% Enhancement

Our biggest enhancement is the ability to rent a monkey for your event rather than simply purchasing one of many identical monkeys. This requires each monkey to be saved independently of others, and for users to be able to return monkeys upon renting them.

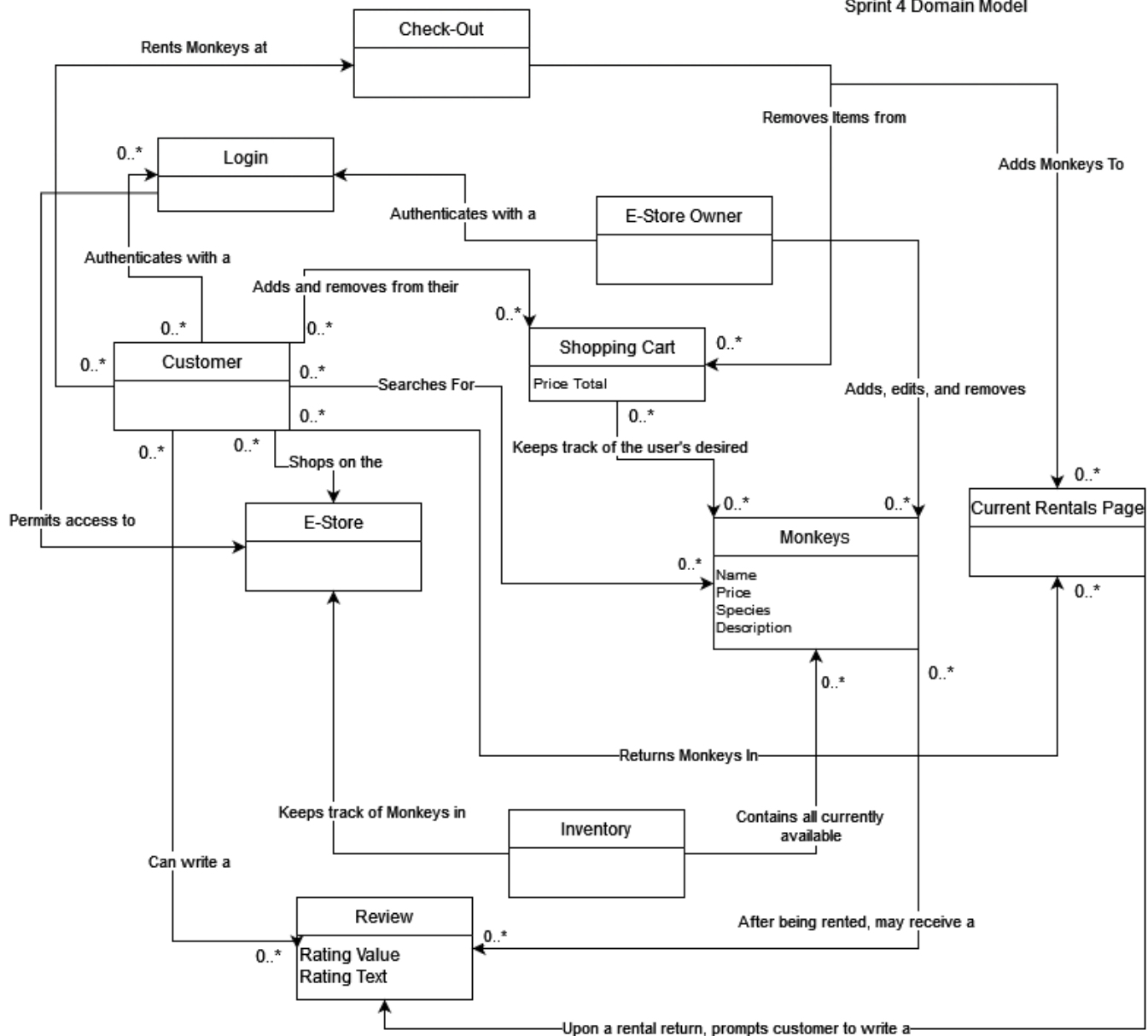
We have also implemented the ability to write reviews after you return a monkey that you have rented. Each monkey's product page displays its average rating, as well as a list of written reviews from users who have rented them.

Application Domain

This section describes the application domain.

Team 3C - 5Guys

Sprint 4 Domain Model



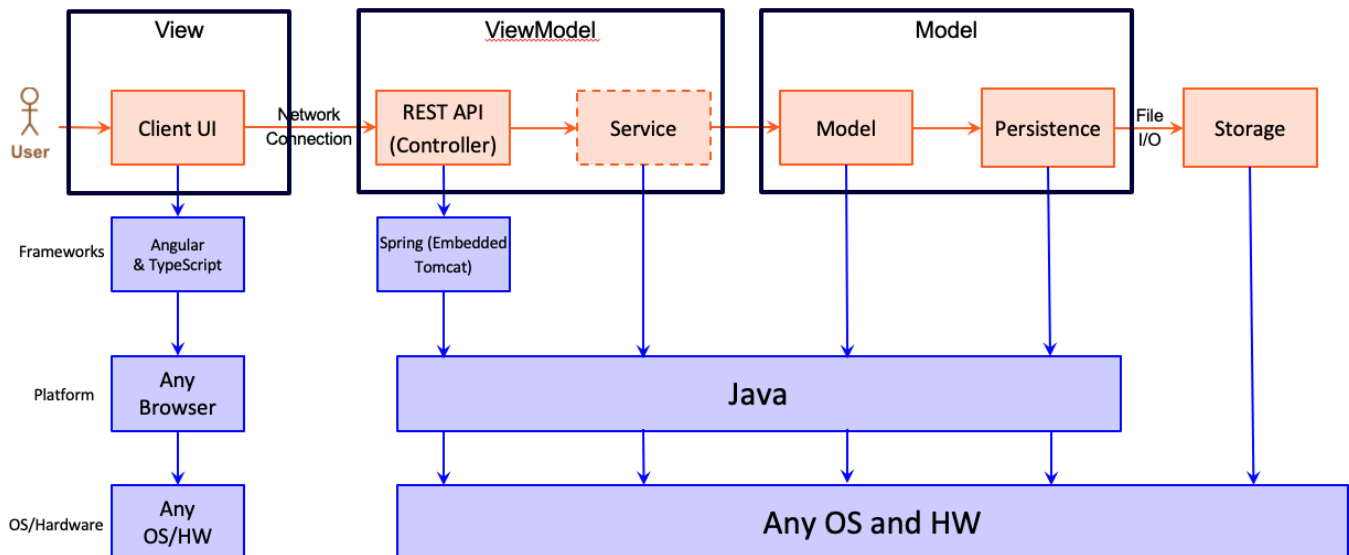
There are two ways to log in to the e-store. Using a username, the system authenticates each user as either a customer or an admin. Customers on the e-store can view a list of products and add or remove them from their shopping cart, from which they can later checkout. Customers can then return the monkeys that they are currently renting and are prompted to leave a review for other users to see. Unlike customers, the owner, authenticated as an admin, can manage the e-store's inventory, which contains all the products in the e-store. The owner can manage the e-store by adding, removing, or updating the specific details of monkeys in the inventory.

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.

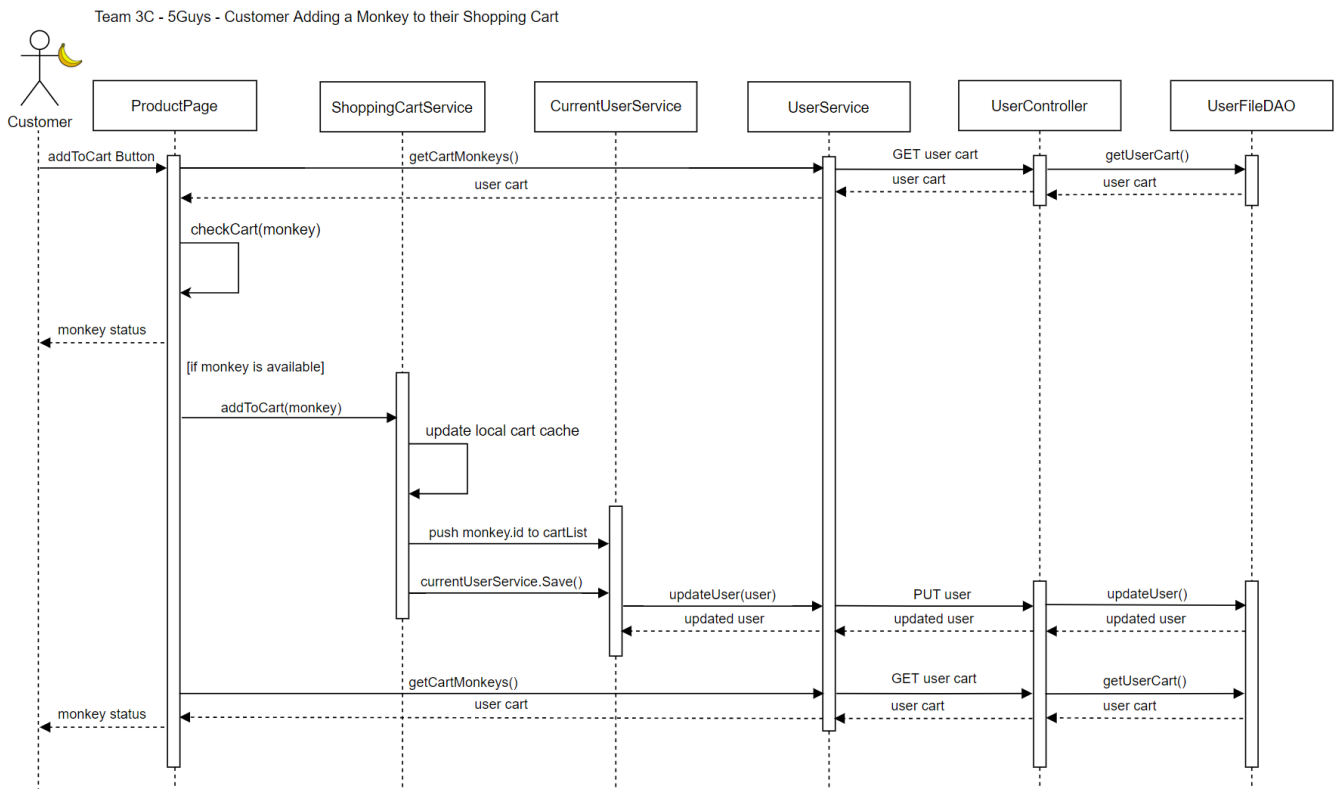
The Model stores the application data objects including any functionality to provide persistence.

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.

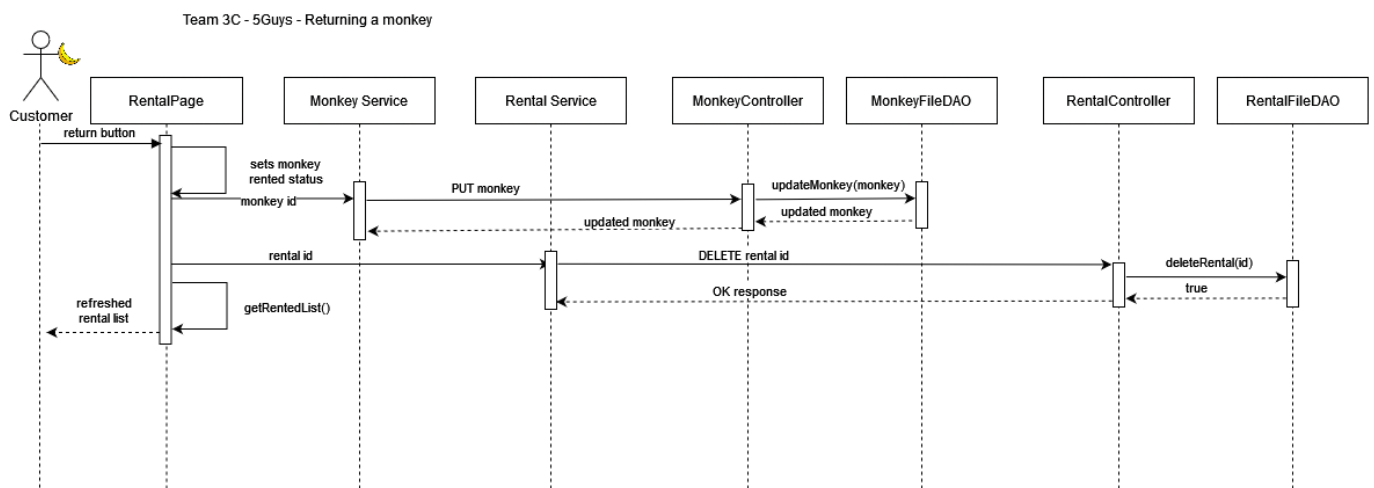
Both the ViewModel and Model are built using Java and Spring Framework. Details of the components within these tiers are supplied below.

Overview of User Interface

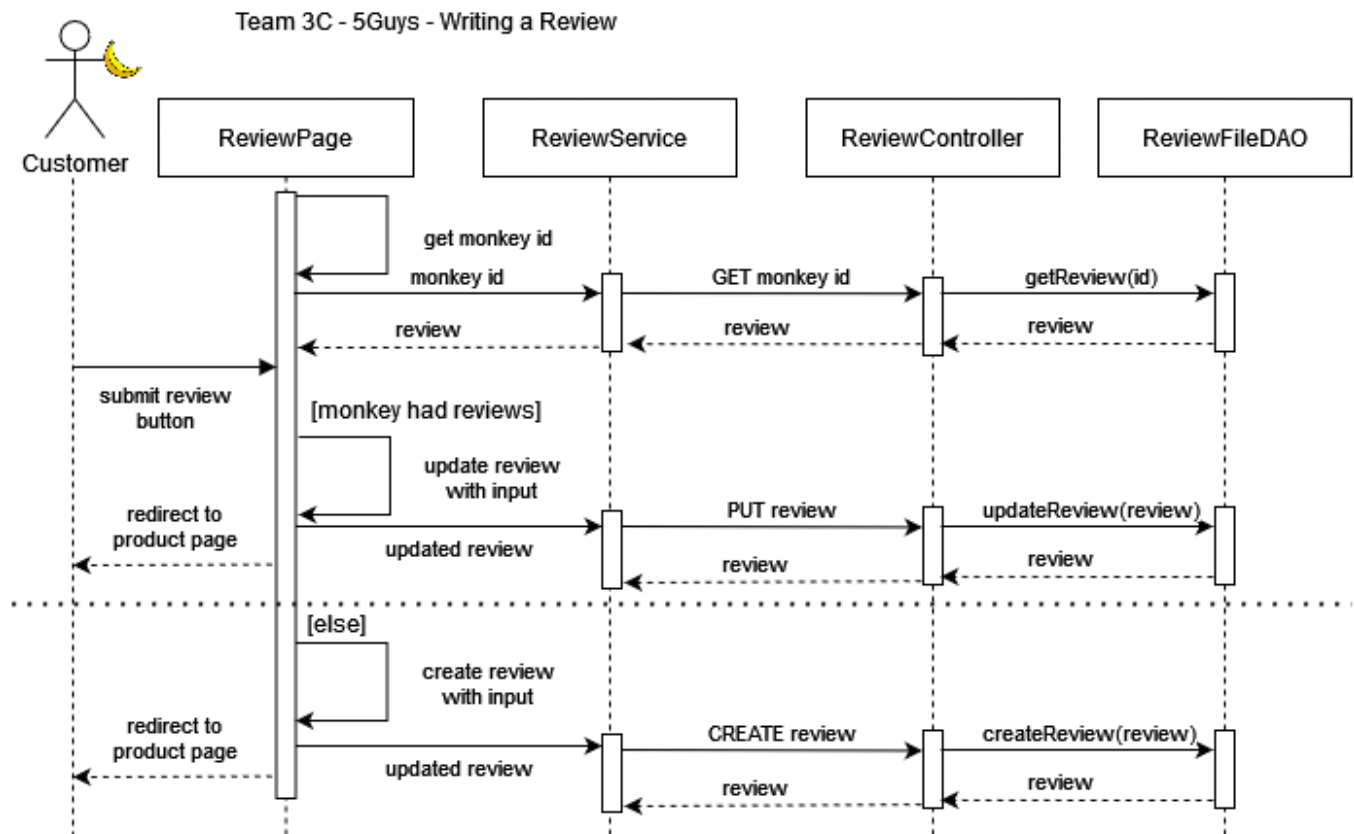
When launching the e-store, the user will first be greeted by a login page where the user will be prompted to log in with their username or create a new account. Once the user has either logged in or created an account, they will be redirected to the product list page, where a list of monkeys is displayed, as well as a search box to find specific monkeys. If the user is an admin, they can also edit and add monkeys while on this page via text boxes and buttons. When clicking on a monkey, the monkey's product page will be displayed, showing information regarding the species, name, id, description, cost, and availability of the monkey, as well as an option to add it to the user's cart. This page also displays the monkey's average rating and a list of user-written reviews. When on the product list page, the user can be redirected to their cart by clicking on the cart button, and from there they will be taken to a page where they can remove monkeys from their cart as well as checkout using the checkout button, which updates the status of each rented monkey's availability and adds each of them to the user's current rentals list. On the user's current rentals page, they can view the monkeys that they are currently renting and return them when ready, at which time they are also prompted to leave that monkey a review.



This sequence diagram shows the process of adding a monkey to your cart as a customer. A customer user will begin the process by selecting a monkey's add to cart button on their product page. This will call the `getCartMonkeys` function, which gets the user's cart through the `UserService` by sending a GET request to the `UserController`, which then uses the `getUserCart` method. This method calls `UserFileDAO` which gets information from the json file of users. After the user's cart is retrieved, the `checkCart` method is used to check if a monkey is in the user's cart. If the monkey selected is capable of being added to the user's cart (it is neither currently rented nor in the user's cart), the product page will call the `addToCart` method within the shopping cart service. The `ShoppingCart Service` will update its cart cache, then the `ShoppingCart Service` will send the updated cart to the `CurrentUser Service`'s local user cache. The `ShoppingCart Service` then calls the `save` method within the `CurrentUser Service`. The `CurrentUser Service` then will send a PUT request with the updated user to the `UserController` which will call the `updateUser` method in the `UserFileDAO` class to update the user. After the user gets updated, the `Product Page` will update its view to show whether the customer can add the monkey to their cart.



This sequence diagram shows the process that follows returning a rented monkey. A user begins this process by selecting the return button on the Rental page. The local cached monkey's rental status is updated and it is sent to the Monkey Service. This service then sends a PUT request to the MonkeyController which then will call the MonkeyFileDAO's updateMonkey method to update the server's monkey. The rental page then sends the rental id to the Rental Service. This service sends a DELETE request to the RentalController which calls the RentalFileDAO's deleteRental method to delete the server's rental. The updated local rental list is used to refresh the Rental Page and the user is redirected to the Review page.



This sequence diagram shows the process that follows creating a review for a monkey. As the review page is loaded it retrieves the id of the previously returned monkey. This id is then passed to the Review Service which sends a GET request to the ReviewController. This in turn calls the getReview method from the ReviewFileDAO and returns the review that corresponds to the monkey's id to the Review page. The User then continues the process by submitting an entered review text and rating for the monkey. If the local cache in the ReviewPage was updated with an existing review then the user's input will be added to the review object's lists of text and ratings. The page will then call the Review Service's updateReview method which will send out a PUT request to the ReviewController. The controller will call the ReviewFileDAO's updateReview method to update the server's review. The user will then be redirected to the product page. If a review object did not exist for the monkey that was returned the Review page will instead create a new review object from the input and send it to the ReviewService. This service sends a CREATE request to the ReviewController class which calls the createReview method in the ReviewFileDAO. This updates the server's review object. The user is then redirected to the product page.

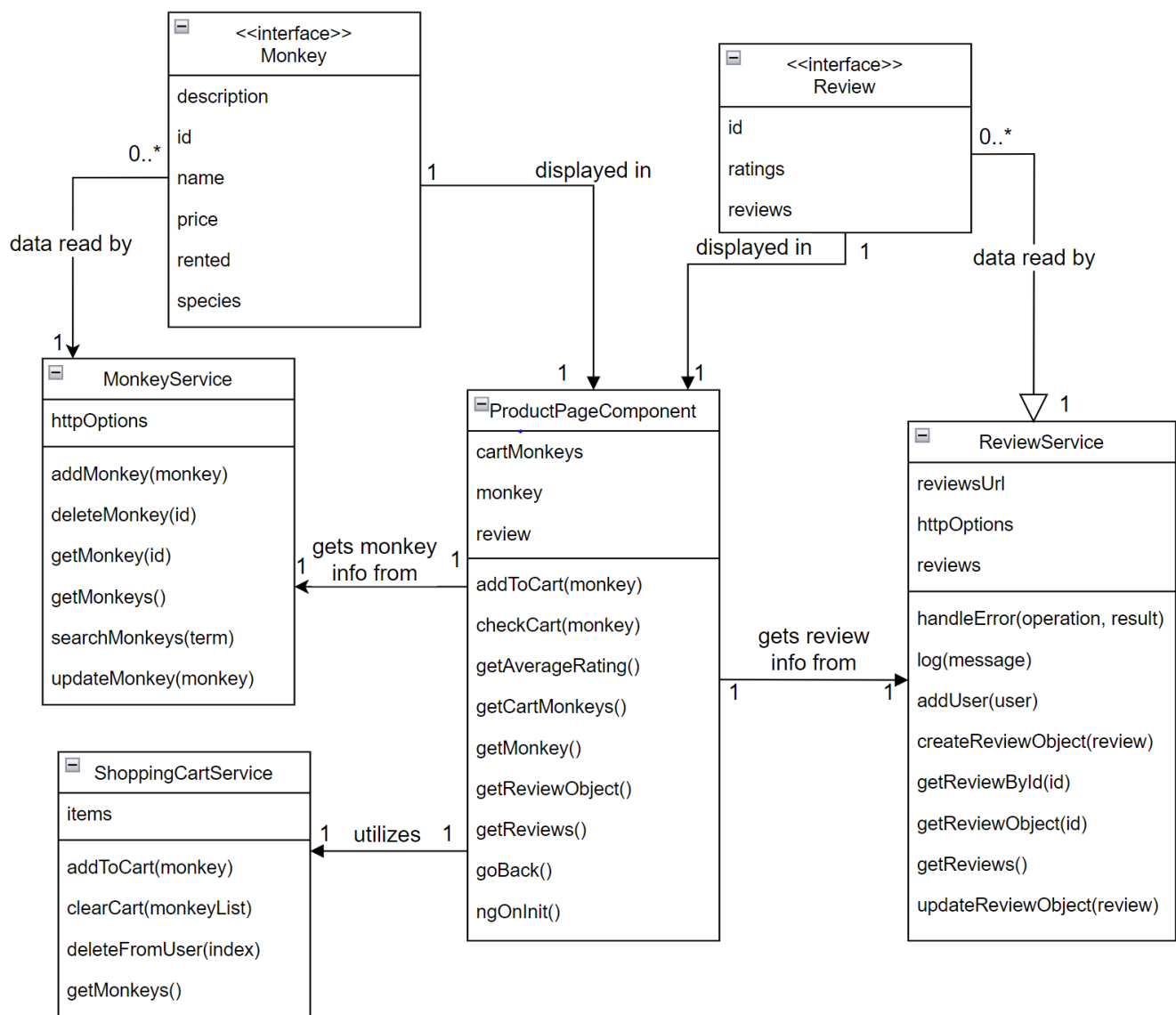
View Tier

When opening the website UI, the first thing a user will see is the Login Page Component which will give an option to enter your username or create a new one. If the username "admin" is typed, the Owner Features

Page will be opened. This page contains a user interface that allows the admin to update monkey information, create a new monkey for the e-store, or delete a monkey from the e-store.

If the user logs in with a customer account or creates a new account (which will automatically be a customer account) at the Login Page, the Buyer Product List Component will be opened. The Buyer Product List contains a list of all the monkeys in the e-store and another list of monkeys which is generated based on the search using the search bar.

After clicking a button for a specific monkey from the Buyer Product List, the Product Page Component will be opened. This will contain information about the selected monkey and an option to add to cart. This information includes their price, species, description, average rating, and list of user-written reviews.



Above is a class diagram centered around the Product Page. The Product Page as shown gets monkey info from the Monkey Service, and it gets review info from the Review Service. It takes this information and displays it in an organized fashion. The Product Page features an add to cart button which will call the Shopping Cart Service to add the monkey to the cart when clicked.

The Shopping Cart Page displays the monkeys currently in the user's shopping cart as well as a button to checkout and a button for each monkey to remove from the shopping cart.

There is also a Rental Page that displays the user's rented monkeys. After clicking on a monkey, there is an option to return the monkey. After returning the monkey, the user will be brought to a page that allows them to leave a review.

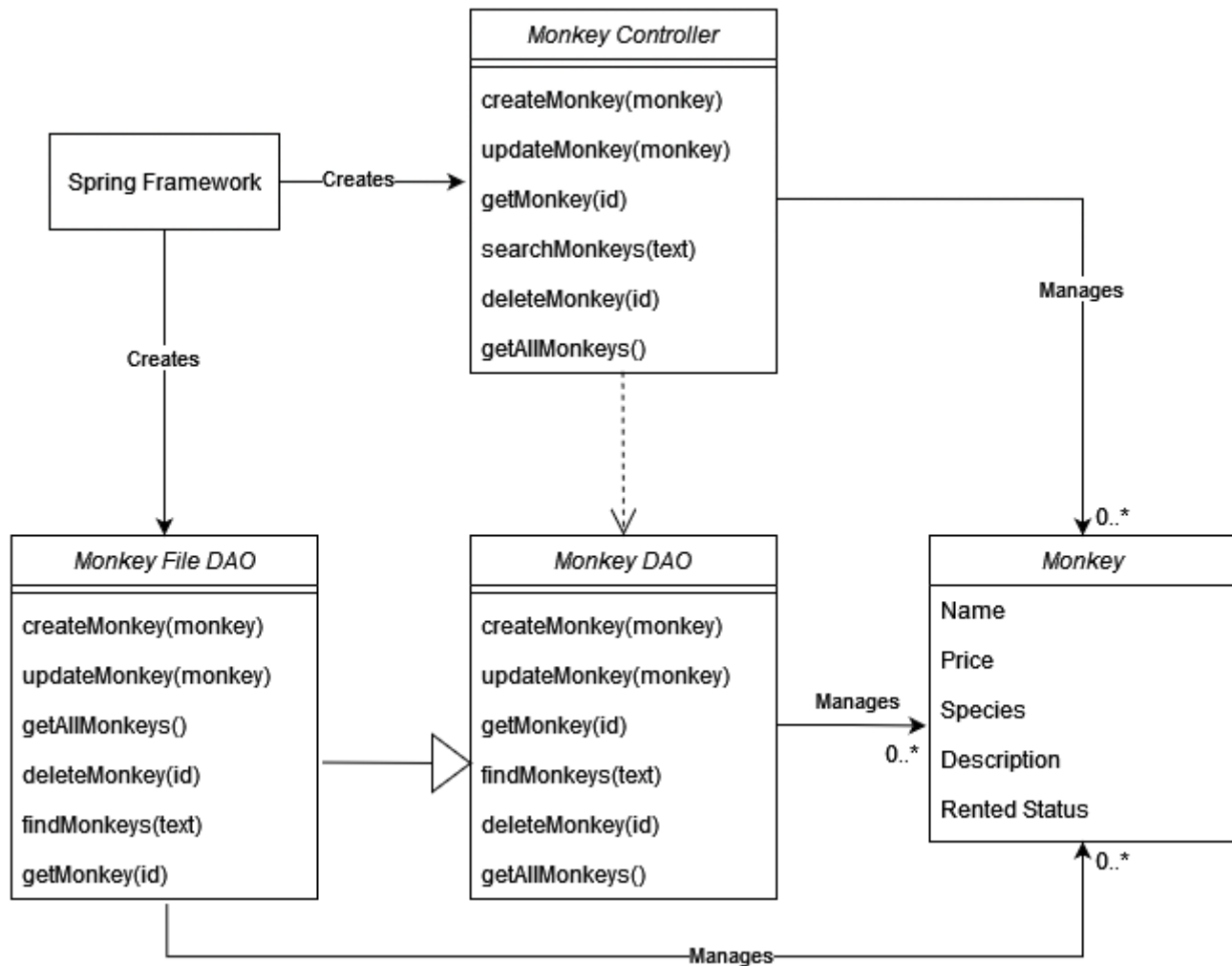
ViewModel Tier

The ViewModel Tier consists of the MonkeyController Class, the UserController Class, the RentalController Class, the ReviewController Class, the Monkey Service Component, the User Service Component, the Rental Service Component, and the Review Service component. The MonkeyController class handles REST API requests involving monkeys in the e-store. The UserController Class handles REST API requests involving user accounts.

For each Controller - Service Component pairing listed above the Service Component sends REST API requests to the Controller to obtain information from the json file where information for that object of that type is stored.

Model Tier

The Model Tier of our project communicates with the ViewModel Tier by receiving updates regarding user actions and requests, and in return sending it notifications relating to the back-end functions of the e-store and its inventory. It also houses our business logic and handles the system's data persistence. The logic is specifically concerned with managing the e-store's monkeys and users. The component concerned with product management gives the ability to create and remove monkeys from the inventory, as well as to update a monkey's specific details, such as one's species, description, and price. The user component provides the necessary abilities to create or manage specific users, manage the items in a user's shopping cart, and manage a user's list of currently rented monkeys.



Our project's backend also uses the Spring Framework. The Spring Framework creates the Monkey Controller, which depends on the Monkey DAO, which defines the interface for Monkey objects in the system. The MonkeyFileDAO inherits the Monkey DAO and implements the functionalities defined in the DAO. Upon instantiation of the MonkeyController object, the system will inject the MonkeyFileDAO into the MonkeyController object so that the MonkeyController can handle REST API requests relating to Monkey objects. Similarly, the Spring Framework creates the Review Controller and Rental Controller, which depend on their respective DAO's. The ReviewDAO defines the interface for Review objects, which each include an ID that matches the ID of the monkey whose list of reviews is contained in the object. Rental objects are similarly defined in their DAO. They also have their own ReviewFileDAO and RentalFileDAO to implement the DAO's functionalities.

Object-Oriented Design

One of the object-oriented design principles we incorporated throughout our project was single responsibility. A prime place where our use of this principle can be observed is in our controller classes, such as MonkeyController. MonkeyController has one primary purpose, which is to handle REST API requests regarding the Monkey object. Rather than tampering with the details or fields of any of the Monkey objects, it responds with HTTP status messages and gives the task of managing the Monkey objects to another class, MonkeyDAO. Our use of single responsibility made components and services much simpler and smaller, which also helped make testing and debugging much easier, as we could more quickly identify the source of problems.

Another object-oriented design principle that we used in our project was dependency inversion. The specific way we implemented this principle is through dependency injection, in which we have high-level modules injecting dependent elements. Our use of dependency injection can be seen between our controllers and their respective DAOs, such as `MonkeyController` and `MonkeyFileDAO`. Upon instantiation of the `MonkeyController` object, the system will inject the `MonkeyFileDAO` into the `MonkeyController` object. The system injects `MonkeyFileDAO` because it has lower-level purposes than `MonkeyController`. Our use of dependency inversion was helpful as it allowed us to inject objects as mock objects and independently test our DAOs and Controllers.

Our project also makes use of the Open/Closed object-oriented design principle. This principle is most clearly seen in the API side of the project where we include a layer of abstraction, in the form of the DAO class. This DAO class is inherited by the File DAO class meaning that the DAO is closed for modification, but can be extended into a new DAO class such as a Database DAO if necessary. For an actual example of this principle in action, you can look to the `Monkey DAO`, `Monkey Controller`, and `Monkey File DAO` classes in the e-store API. There the `Monkey Controller` class has the `Monkey DAO` abstraction directly injected into its constructor. This keeps the `Monkey File DAO` class which implements the DAO separate from the `Monkey Controller` while still keeping the DAO open to an extension if necessary.

Static Code Analysis/Design Improvements

Some improvements we would make if the project were to continue involve design decisions causing consistent code smells throughout the code. Many of our code smells involve an early design choice to use 'map' instead of 'foreach' in several places, whose uses of map are now going unused. If the project were to continue, we would most likely remove these instances and either leave them as removed or replace them with a 'foreach' statement to increase reliability.

One of our biggest current issues is our code's readability. This is due to several old, outdated comments that either housed code that was not working or TODO tasks that were never deleted, which has hurt our code's readability. In the future, we would remove these to improve our readability and code style consistency. We also had consistent code smells relating to minor unnecessary add-ons, such as including type specifications where they are not needed. We would replace these type specifications with the diamond operator "<>" in the future because they are unnecessary in the version of Java we are using and are making the code longer and more verbose than it needs to be.

```
<!--a href="" style="position:fixed; top: 40px; right: 50px">
```

Remove this commented out code. Why is this an issue?

15 days ago ▾ L56 🔗

🔗 Code Smell 🚩 Major 🔵 Open Not assigned 5min effort

🔒 unused

```

```

```
try {
    User m = userDao.updateUser(user);
    if (m != null) {
        if (!m.equals(user)) {
            return new ResponseEntity<User>(m, HttpStatus.NOT_ACCEPTABLE);
        }
    }
    else {
        return new ResponseEntity<User>(user, HttpStatus.OK);
    }
}
```

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

Why is this an issue?

1 month ago ▾ L100 🔗

🔗 Code Smell 🟢 Minor 🔵 Open Not assigned 1min effort

🔒 clumsy

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

Why is this an issue?

1 month ago ▾ L103 🔗

🔗 Code Smell 🟢 Minor 🔵 Open Not assigned 1min effort

🔒 clumsy

Something else that was flagged during our static code analysis was a consistent issue in our unit tests. We frequently made the mistake of swapping the expected and actual values in some of our test cases. Because our code always produced the expected output, the expected and actual values were still always equal and did not present any issues during development. However, if a test were to fail, these backward assertions would be problematic as it would make it difficult to figure out why the expected output was not being produced.

Another improvement to make if we were to continue development would be to refactor our current Review storage system. As it currently exists reviews are created and accessed through their own objects that store the list of reviews and the list of ratings for each monkey. Moving these two lists into the Monkey objects themselves would cut down on code complexity and would make both the API and UI of the project simpler.

Testing

To ensure that the project is meeting our requirements, we conducted two types of tests. Acceptance testing allowed us to ensure that requirements related to the front-end of the e-store were working as expected, while Unit Testing allowed us to test the system itself and make sure it is handling everything correctly on the back-end.

Acceptance Testing

- 38 of stories that we finished pass all of their testing criteria
- 1 of the stories partially pass acceptance criteria
- 0 of the stories have yet to be tested

The project now meets nearly all the acceptance criteria we have defined. We accomplished this by making sure that each user story passed its acceptance criteria before considering it "done." We also considered the acceptance criteria that were previously partially passing as bugs and worked on fixing them alongside working on the user stories. This approach has led to all of these previous bugs now being fixed and passing

their acceptance tests. Finally, the acceptance criteria for our review feature that were previously failing due to reviews not being implemented are now all passing, as reviews are now fully implemented.

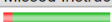
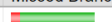




The one criterion that is currently failed is that a message should display upon an attempt to checkout with no monkeys in the user's cart. This feature was left undone as it didn't cause any issues with no message being displayed, and because the UI was found to be more usable without the message as not being able to checkout with an empty cart is generally intuitive.

Unit Testing and Code Coverage

The way that we have handled unit testing is by creating a doc of all tests that need to be written and allowing team members to evenly split the work among themselves. We have 100% coverage in the model tier and 95% coverage total. Our goal for overall coverage is 90% minimum but we like to strive for higher if we have time. We chose this number because it results in a balance of having enough testing without wasting time working on tests when more important parts of the project need to be done. Our lowest current element is e-storeapi with a current coverage of 88%. The coverage is this low because main is not fully covered with tests.

 estore-api

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.controller		91%		89%	5	50	16	191	1	31	0	4
com.estore.api.estoreapi.persistence		96%		88%	8	82	7	214	1	48	0	4
com.estore.api.estoreapi		88%		n/a	1	4	2	7	1	4	0	2
com.estore.api.estoreapi.model		100%		n/a	0	46	0	69	0	46	0	4
Total	109 of 2,257	95%	12 of 106	88%	14	182	25	481	3	129	0	14

Our strategy in creating unit tests is to attempt to cover every method by checking its possible success and failure conditions ensuring that they match what we expect them to. For example, when writing unit tests for the creation of a monkey in the MonkeyController class, we tested the output from the MonkeyController Creation method when the monkey is created correctly, when the monkey creation is rejected, and when the MonkeyDAO throws an error.