

SBESCI Design Documentation

Team Information

- Team name: Software Business Enterprise Solutions Corp. Incorporated (SBESCI)
- Team members
 - Eli Wertzberger
 - Aazeem Vaidya Shaikh
 - Charles DiGiovanni
 - Conrad Cox
 - Willem Dalton

Executive Summary

A web application that allows teachers to request supplies and allows users to browse and donate school supplies to a school in need, as well as view supply reviews and a dynamically updated sponsor leaderboard.

Purpose

The project is a simple one page website where members of the community can donate items to their local school board. Teachers of the school can make requests, and see the status of their requested items. Users can donate to fulfill requests, and admins can manage which items teachers can request.

Glossary and Acronyms

Term	Definition
SPA	Single Page Application
Auth	Authentication
Need	An item the teacher requests
Session	Identifies a User
Request	A group of items a teacher asks for
Type	The user's role in the project
BasicInfo	A user's login information
Supply	Another term for a need.
SBESCI	Software Business Enterprise Solutions Corp. Inc.
Helper	Synonymous with user or donator
Manager	Synonymous with admin or U-Fund Manager

Requirements

This section describes the features of the application.

Our application is designed to support underfunded schools by providing a platform where users' can donate school supplies directly to classrooms in need. Teachers can submit personalized requests for specific supplies and quantities, which are shown on the site for donors to view. Users can search for items, or use categories to browse by supply type, and add items to a cart. We will have login system to authenticate users and distinguish between users, teachers, and a U-Fund Manager.

Definition of MVP

Our Minimum Viable Product for this single page application contains:

- login/logout authentication utilizing cookies to store existing sessions.
- A Search bar for users to locate needs.
- Control for U-Fund manager to add, remove and edit needs from the cupboard.
- Users are able to add items to their funding basket, then head to checkout to purchase those needs.

MVP Features

General MVP Features:

- A login and logout feature.
- An About Us page.
- A search bar to find information.
- User authentication to check if the person logging in is a manager or a regular user.
- Specific functionality per type of account:

U-Fund Manger MVP:

- Buttons for the fund manager to add, edit, and delete needs.

Helper MVP:

- A cart page where users can add items and view their cart.
- EPIC: U-Fund Manager Functionality
 - U-Fund Manager Add Needs to Cupboard
 - U-Fund Manager Edit Needs in Cupboard
 - U-Fund Manager Remove Needs From Cupboard
 - U-fund Manager Can't Have Access to Funding Basket
- EPIC: Helper/User Functionality
 - Helper Add Item to Basket
 - Helper Remove Item from Basket
 - Helper Search
 - Helper Log In Authentication
 - Helper Log Out Authentication
 - Create Account
 - Checkout

Enhancements

Supply Requests

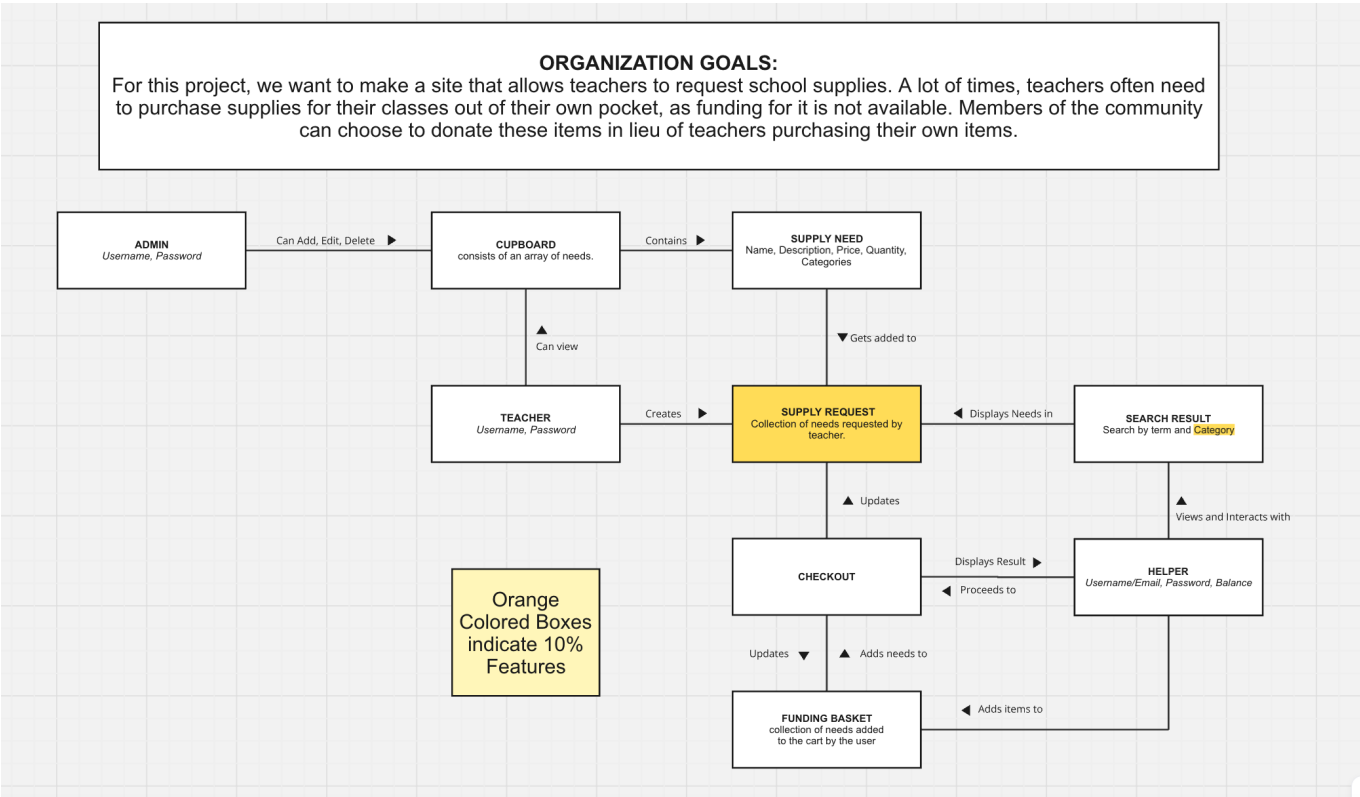
This enhancement allows teachers to create detailed and specific requests for the school supplies they need in their classrooms. Instead of relying on general donations from the cupboard, teachers can now specify exactly which items they want and how many of each item they want. For example, a teacher might request 10 notebooks, 15 pencils, and 5 glue sticks. These requests are then visible to users, who can choose to fulfill them item by item or the user can fill part of a request, like donating 5 pencils and 3 gluesticks then another user can contribute to the same request. This feature makes the donation process more influential as specific needs can be met quickly, allowing students to easily obtain the supplies they need to succeed. One major benefit of the requests system is that multiple teachers can create different requests, and all those requests are displayed to the user. This makes it possible for a whole school to utilize the website, with each teacher having their own account information to make their own requests.

Filtered Searching

This enhancement updated the search functionality to include filtering by item categories. In the MVP, users could only search using the exact name of an item, which made it hard for users' to find specific needs if they didn't know the exact name of the supply. Filtered searching allows users to type in a category like "writing" or "art" and see all related items within that category. This makes it easier for donors to explore and find items they might want to contribute, even if they don't know the exact item name. Filtered searching improves the overall browsing experience and helps users search for a wider range of needs.

Application Domain

This section describes the application domain.



Our domain model centers around supply requests. Teachers can create and view their own supply requests, while helpers/users can "search" which shows all the needs contained inside requests that need to be fulfilled. For example, if Teacher John Doe made a request for 10 pencils, then when the user searches "pencil" they see

a request for 10 pencils. This is not a need from the cupboard, rather a need that's in a supply request. There can be multiply requests for pencils at once!

Some other complicate parts of our domain model is a the cupboard. The cupboard shows all *needs* which is different from a request. Admins can create new needs that teachers can pick from (for example, I am the administrator of the school, and I want teachers to be able to ask for pencils. I can add that need to the cupboard, teachers can now see the pencil.)

Our two 10% enhancements are as follows:

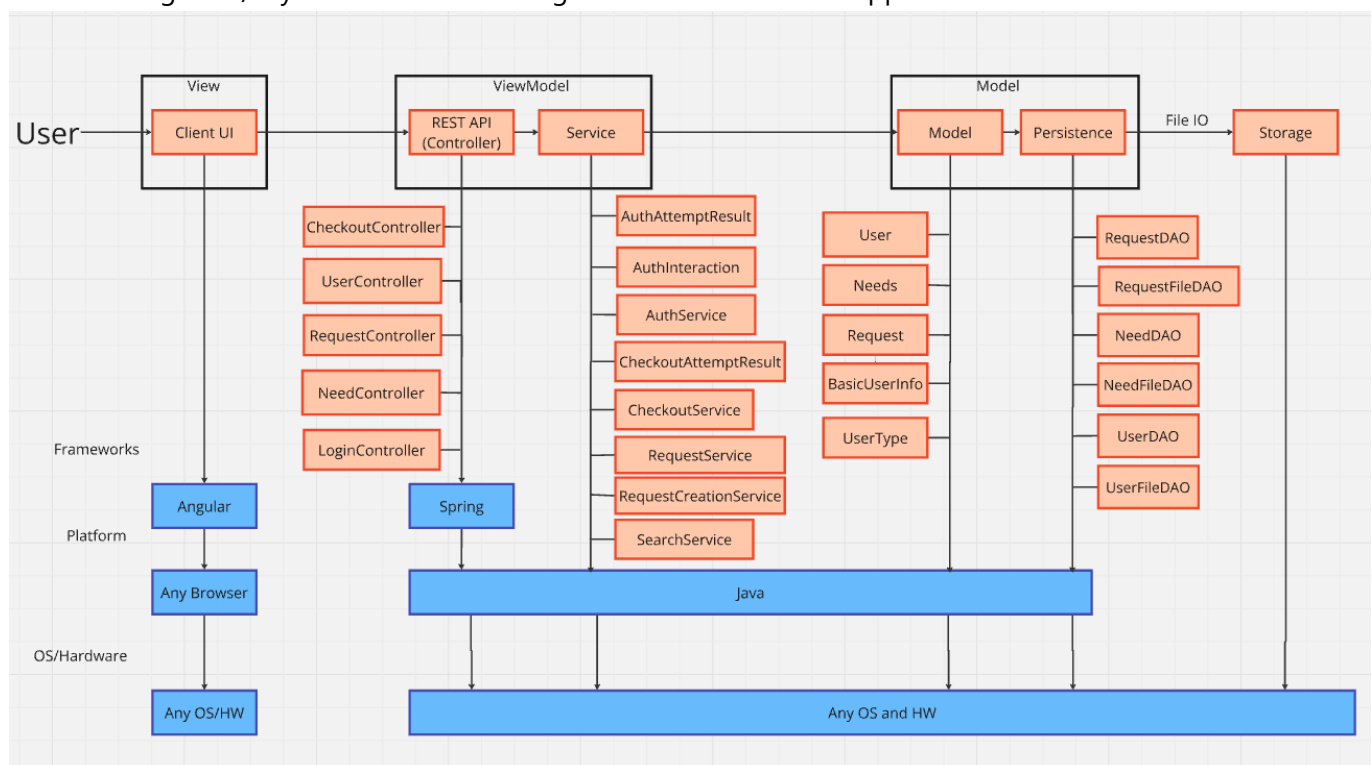
Users can filter their search, i.e, search by category. This gives our users more control over what they want to donate to. Teachers/Supply requests. This one is a very big enhancement, as It essentially creates a third type of user. Teachers are the ones who are directly affected by our donation website, so we wanted them to request items directly. Teachers can create "supply requests" where they request needs that users can donate to. This differs from the original project's schema, where you could only have one copy of a need.

Architecture and Design

This section describes the application architecture.

Summary

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



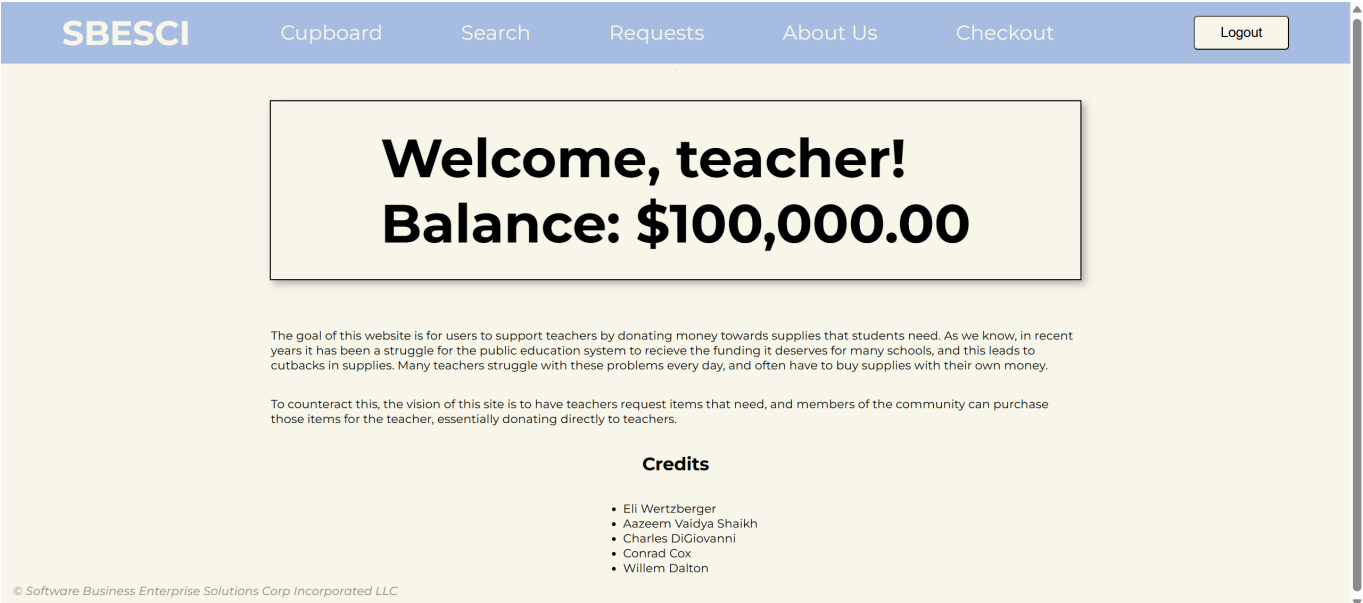
The web application is built using the Model–View–ViewModel (MVVM) architecture pattern. The Model stores the application data objects including any functionality to provide persistence. The View is the client-side SPA built with Angular utilizing HTML, CSS and TypeScript. The ViewModel provides RESTful 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 web application.

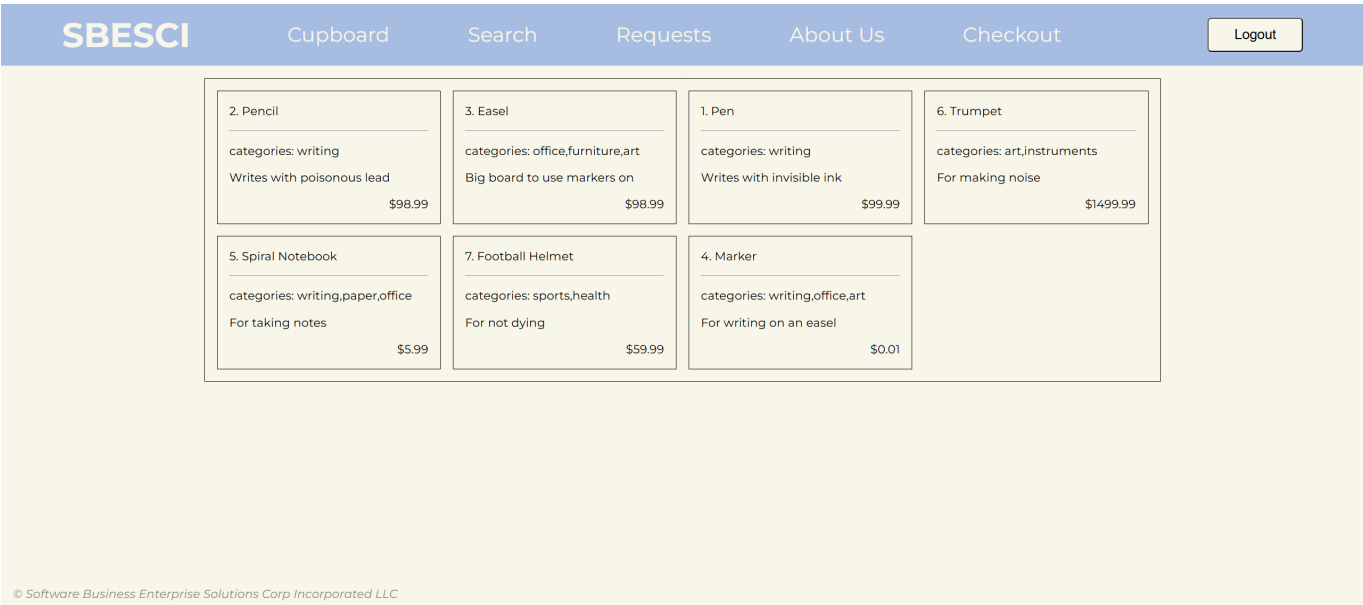
HOME PAGE

All of our users start out on our home page. If they are logged in, they are greeted and we show them their current balance (how much money they have input to be donated). We also have a navbar that spans the entire top of the project, Featuring a number of other pages. There is: Cupboard, Search, Requests, About Us, and Checkout. There is also a login/logout button at the end.



CUPBOARD

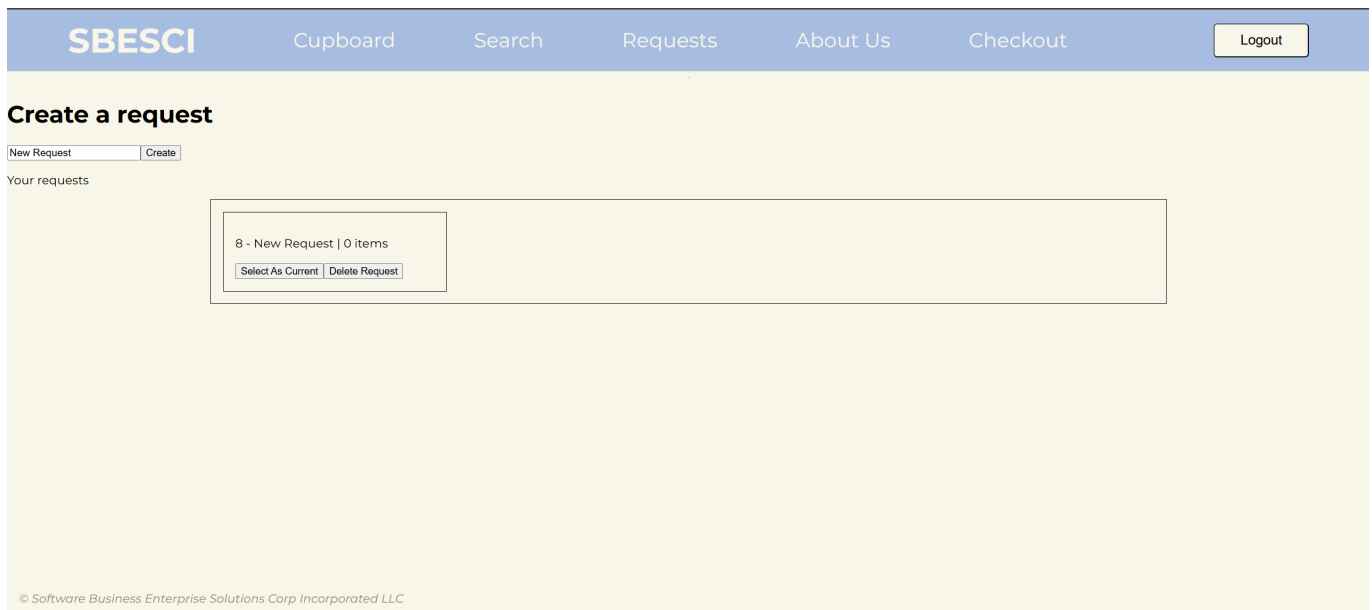
Let's start with the cupboard. The cupboard is a special feature that only teachers and admin can see, which displays a list of all needs that currently exist. Notice that *requests* is different from *needs*. We are showing all needs. Admin can add new needs, edit needs or remove needs from this page, while teachers can add needs to a request. Users can **not** see this page.



REQUESTS

Next up is our requests page. Requests is a teacher specific page, where teachers can create new requests and view existing ones. We display all of a teachers requests in the center. Requests are specific to teacher, and are not shared by multiple teachers. Requests have a title, and are made up of multiple needs. Teachers can select a request as their current request, then add/remove needs to it.

The best way to think of requests is like a to do list. Teachers can make a "to do list" then add items that need to be done to that list. Users can "check off" items in that list and when the list is complete, the supply is complete. Teachers all have their own lists, and can have multiple to do lists at once, each with a different amount of progress done (some are almost done, some haven't even been started).



SEARCH

Our search feature is pretty self explanatory, So we felt like a picture was not warranted. Users and teachers can access the search feature. We have two ways to search: search by name, and search by category. Every need has a category (*writing, sports, instrument, art, paper, etc*) and so we can search by just the stuff we want (maybe i just want to donate paper). One thing I need to state is: The search feature does not search **needs**, but **needs that are in active requests**.

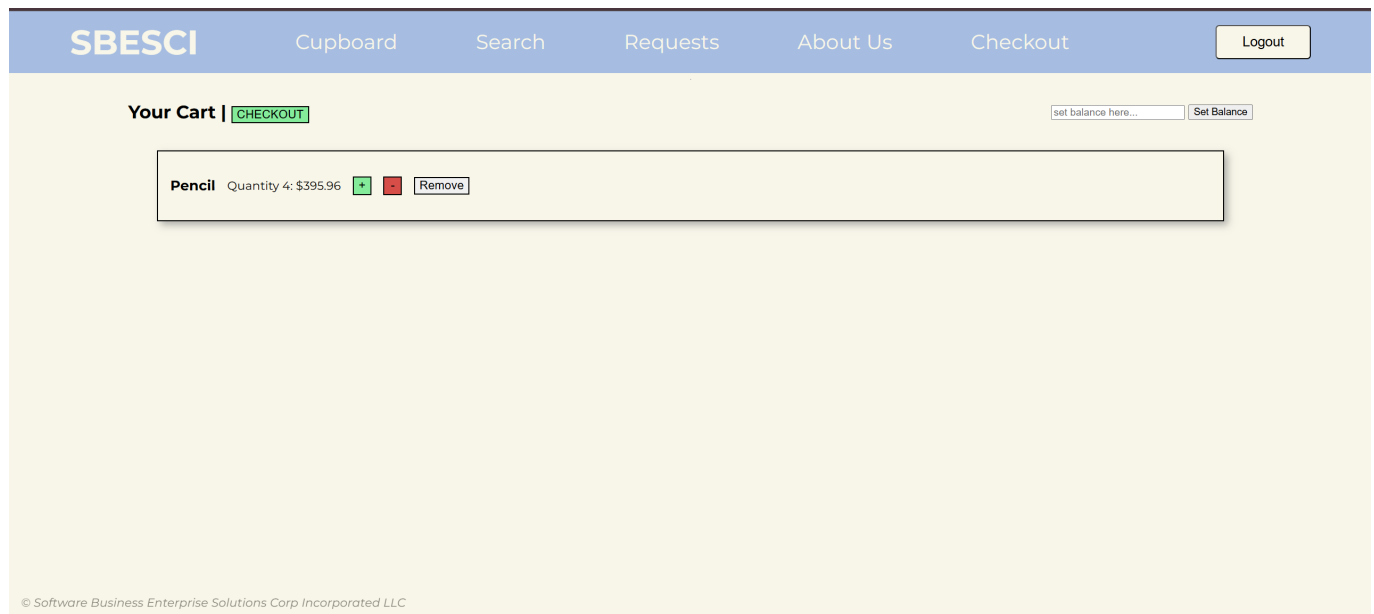
Users can Either search for products by name, where the list of products will be filtered in the center beneath the search bar, or they can click one of the sidebar categories, in which case the middle section will be populated with items in that category. The list of items will show the title, image, and cost, but if you click that item, you'll be able to see teacher reviews of the item, a description, and an add to cart button. There will be a cart button at the top right of the screen at all times.

About us is our landing page, containing some basic info about our site.

CHECKOUT

Checkout is where we can finally purchase our items and give them to those who need it. Any items we add to our cart from the search are displayed here, in our cart. We can increment and decrement the count of items in our cart, or remove that item altogether.

When we are satisfied with our cart, we can checkout, and if we have enough money, the item no longer appears in our cart and the request is fulfilled. The item will no longer appear in the search too, (if we bought it all.)



Future Recommendations

there are some glaring issues that should be addressed if this project were to go further. I believe one of the biggest issues is accessibility. Our website is not designed with accessibility in mind, and because of that it doesn't run well on low end websites, have good screen reader capabilities, and is not very usable by those with disabilities. These should be changed as soon as possible in the project. The internet was designed to be accessible by all, not most.

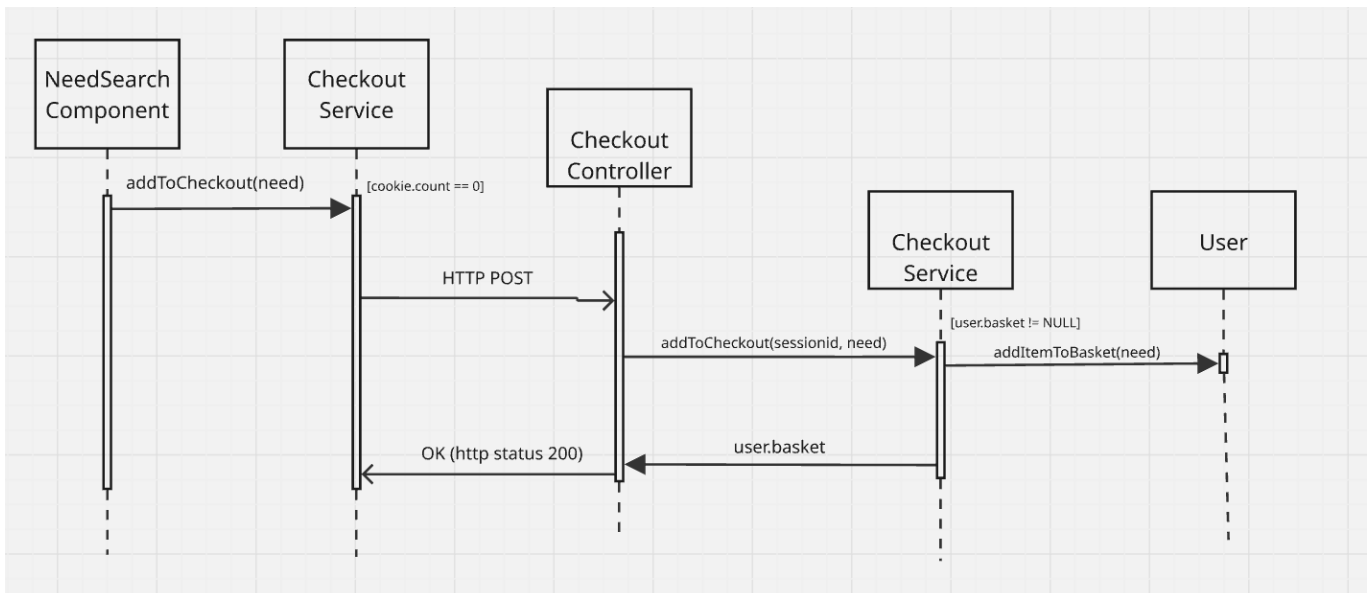
Another problem with our user interface does not scale friendly with mobile devices. Nearly 60% of all internet traffic comes from mobile devices, so by not making our website usable on a lower end device, we are effectively limiting potential donors by over half, which would greatly affect the surrounding community. Mobile development should be a top priority for future sprints.

The final recommendation we have is to include more non-text content on our site. The saying "a picture is worth a thousand words" goes a long way in web development. We should plan to display as much content as possible through other mediums so that our users do not get bored (though technically, text is the most user friendly).

View Tier

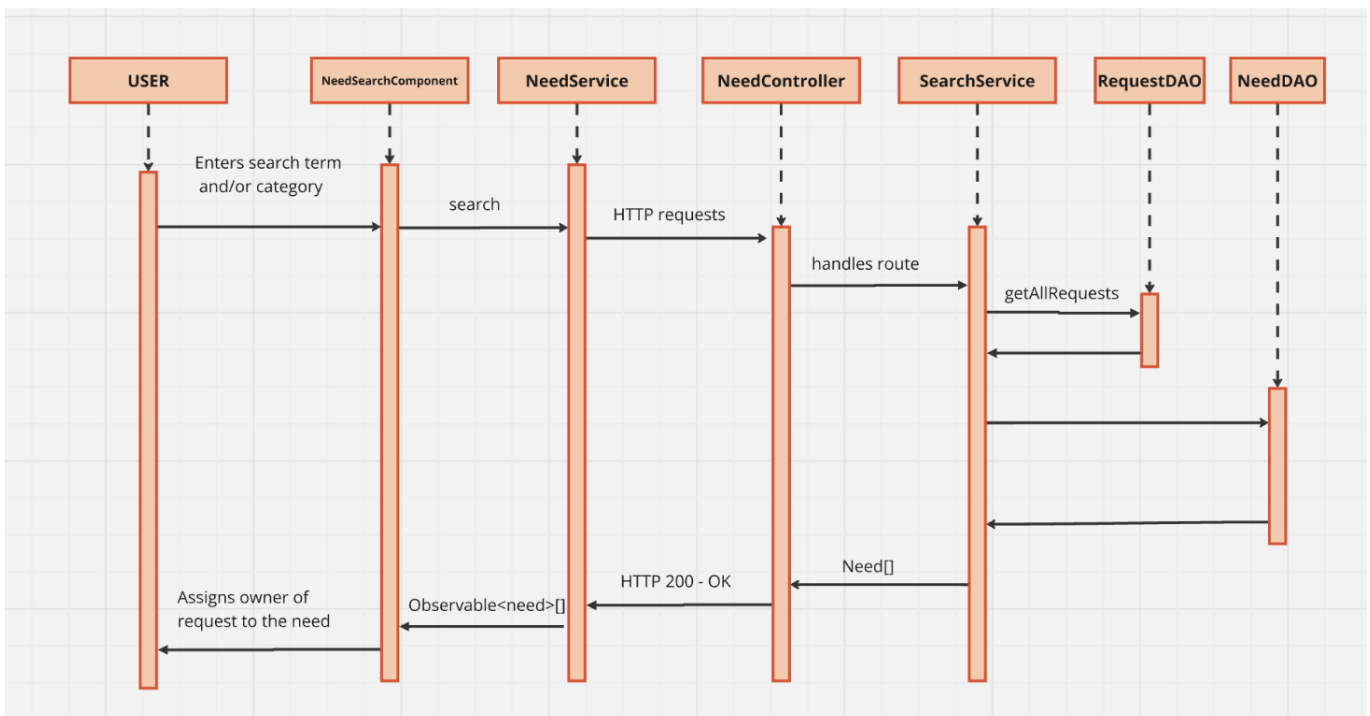
Our view tier of our architecture mainly consists of Angular's component architecture which we used often to achieve our vision for the website. We also utilized model typescript objects to represent our need, requests and users. Each "page" is made of 1 or more components that house the related code. We would sometimes need to make service classes for the components to interact with, especially if two components needed to interact with one shared service class.

Here's one example sequence diagram for our "add to cart" feature, which users do often so that they can checkout. It's a simple example and a good representative of how our system moves through every tier.



We start with a user, who, when pressing the "add to cart" button on a need fires the NeedSearch Component. This action is not shown on the sequence diagram, but it starts the lifetime of the action. NeedSearch Component then calls on our CheckoutService typescript component, which checks if the add to card is valid. One particular action it does is checks if any existing cookie exists (i.e. do we have a session running?) If there is no session, we don't add to the cart and don't run the rest of the sequence. Next, we do an HTTP post request to add the item to the cart. Checkout controller catches this request, and calls the appropriate function in Checkout Service (different from the other one, this one is on the backend!). Checkout service calls upon the user class to update the basket property (what items are in their funding basket). We then return an HTTP 200, OK.

Here's another example of something more complicated, our 10% enhancement of search by category.



The user inputs their search term into the search bar, and this calls on our NeedSearchComponent. NeedSearchComponent is pretty complicated, but the gist of it is that it runs calls on our Need Service to actually get all the needs with those categories. We need to jump few a few more hoops than the original

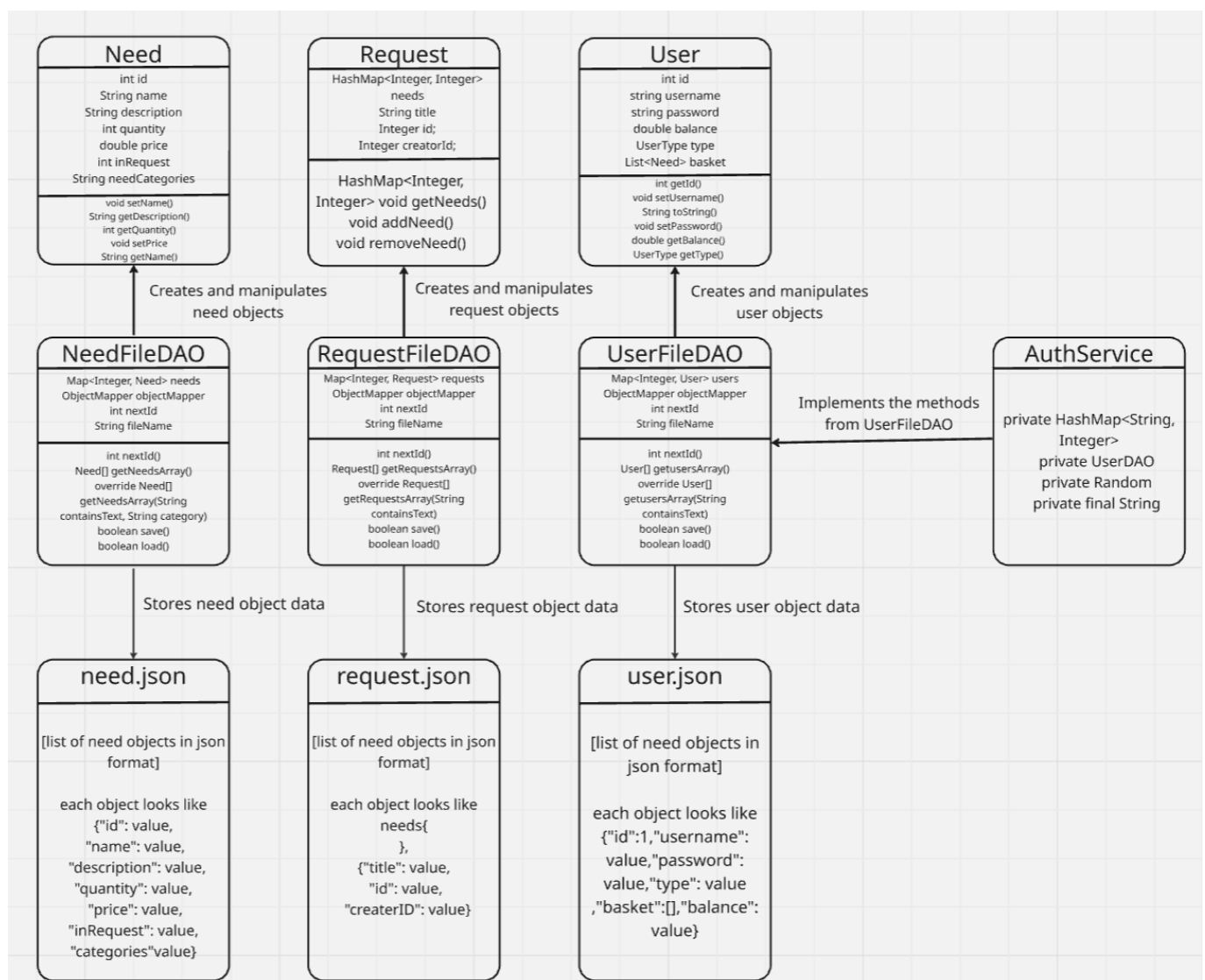
project because we actually need to get all the requests first and get the needs in those that follow our parameters. Need Service runs the http requests, which Need Controller catches. Need Controller handles the routing for the requests, and calls on Search Service. search service calls on both requestDAO and needDAO which, finally, returns the list of needs following our category. This content is sent back to the user in an http request which NeedSearchComponent and NeedService handle and display as a list of needs.

ViewModel Tier

Our view model tier is what sets up our file daos, and acts as an mediator between model and controller classes. The view model tier is one of the most important sections of our project, as it is responsible for keeping our backend running as expected.

NeedDAO.java: Defines an interface for managing Need objects in storage, including retrieval, search, creation, updating, and deletion. UserDAO.java: Defines an interface for managing User objects in storage, including retrieval, search, creation, updating, and deletion. RequestDAO.java: Defines an interface for managing Request objects in storage, including retrieval, search, creation, updating, and deletion.

NeedFileDAO.java: Manages Need objects in json storage, including retrieval, search, creation, updating, and deletion. UserFileDAO.java: Manages User objects in json storage, including retrieval, search, creation, updating, and deletion. RequestFileDAO.java: Manages Request objects in json storage, including retrieval, search, creation, updating, and deletion.



Model Tier

Our model tier holds all of our most important classes, which are accessed and manipulated by both the Persistence and Controller tiers. They are essentially the building blocks that make up our project. They are as follows:

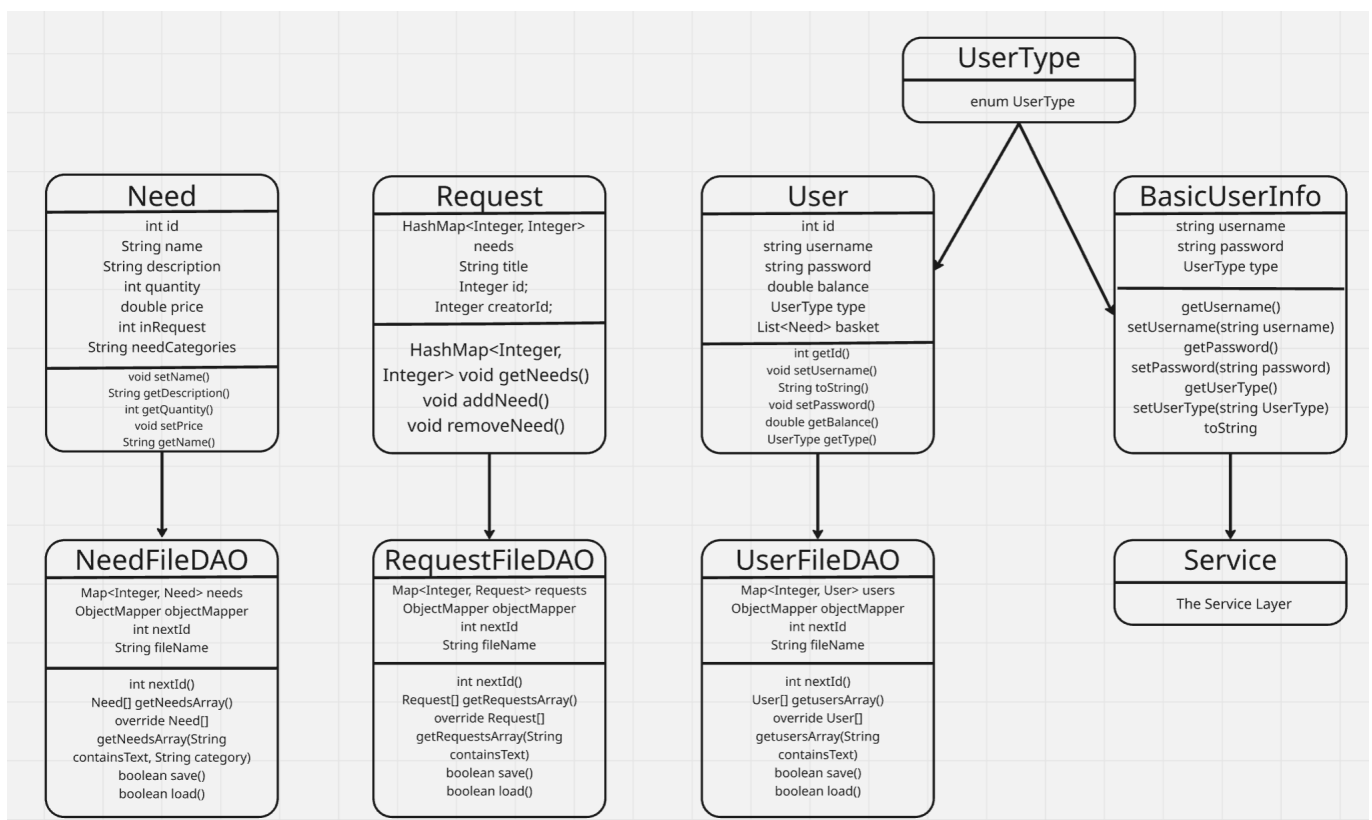
Need.java: Defines a model class representing a Need with attributes like id, name, description, quantity, and price, along with getters and setters for data management.

Request.java: Defines a model class that manages a collection of Need objects using a HashMap, allowing retrieval, addition, and removal of needs by their ID.

User.java: Defines a model class representing a user with attributes like id, username, password, balance, and type, providing getters and setters for managing user data.

BasicUserInfo.java: Holds the basic data necessary for creating and verifying login info, such as username, password, and usertype (admin, teacher, helper, etc).

UserType.java: exports a UserType enum that improves readability and allows for clearly organized user functions that differ based on the type of user that is logged in during a given session.

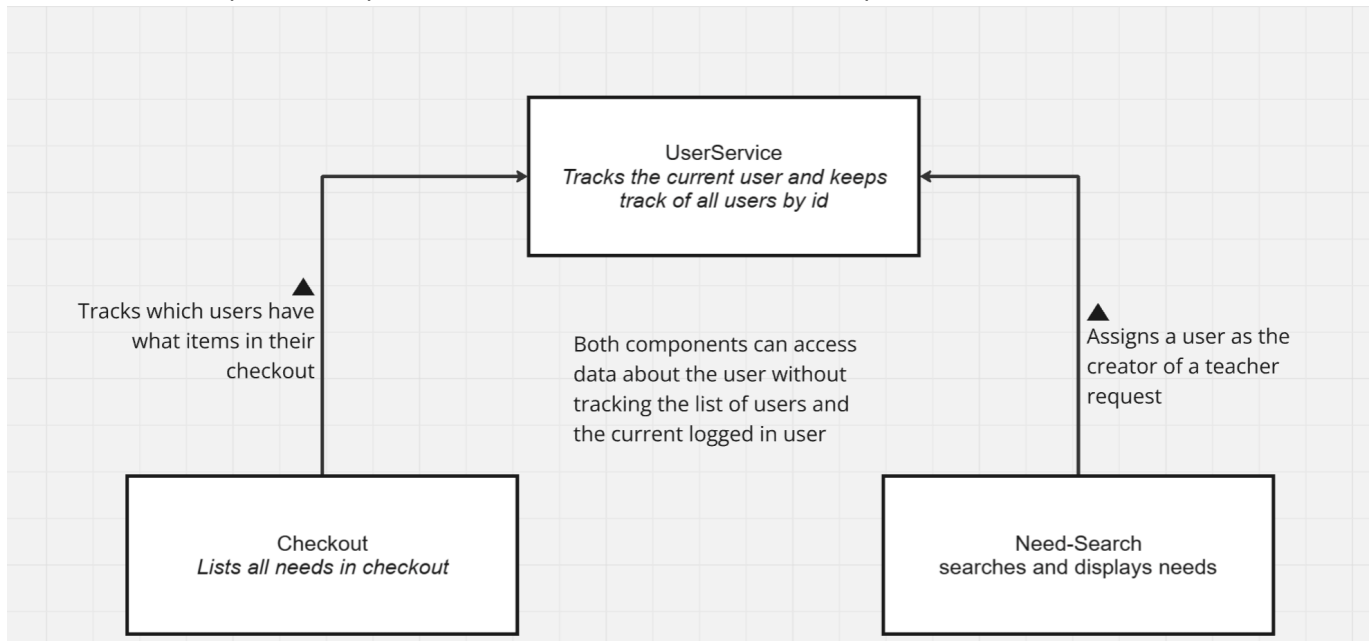


OO Design Principles

Low Coupling:

We implemented low coupling by using services instead of direct connections between components to keep the system flexible and easy to manage. In the functions outlined by our model diagram, we use services to streamline the function flexibility. For example, both the NeedSearchComponent and the CheckoutComponent need to work with user information, like getting a user by their ID or updating the current user's balance. Instead of writing that code directly in the components, we used the UserService to

handle it. This keeps the components cleaner and makes it easier to update or reuse the user-related code.



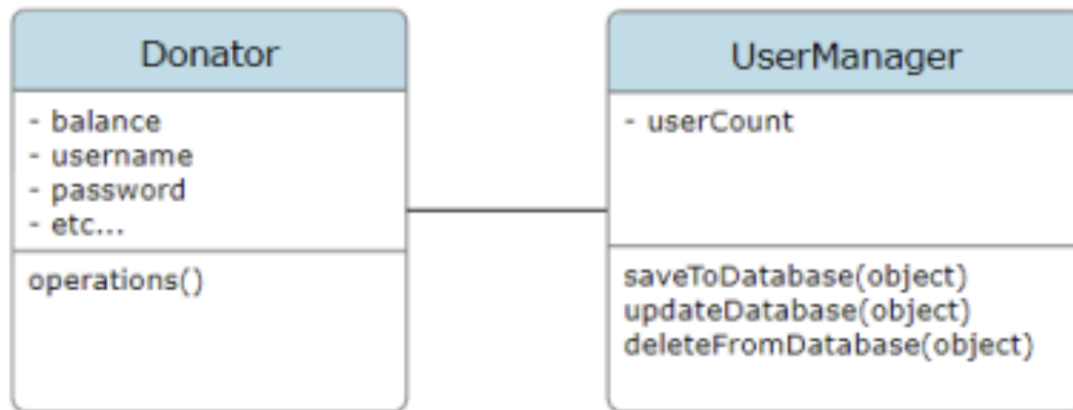
Pure Fabrication:

Pure Fabrication is a fundamental design principle that encourages the creation of classes that don't map directly to domain concepts but instead serve specific technical purposes. These fabricated classes are deliberately introduced to maintain clean architectural boundaries and support essential technical operations without polluting domain objects. When domain objects cannot handle certain responsibilities without violating principles like Single Responsibility or High Cohesion, Pure Fabrication provides an elegant solution. The principle particularly shines in scenarios where technical concerns need to be separated from business logic. For example, persistence operations don't naturally belong in domain entities, even though these entities contain the data being persisted. Instead of forcing database operations into domain classes like `Supplies` or `Donator`, we create Pure Fabrication classes like `PersistentStorage` or `DAOs` to handle these responsibilities. This approach maintains high cohesion within domain classes while providing a clean separation of technical concerns.

How we Applied It:

In our implementation, we use Spring. Pure Fabrication manifests through various technical components such as `Controllers`, `Repositories`, and `Service` layers. These components don't represent real-world concepts but rather serve as essential technical abstractions that keep the domain model clean and focused. The `DAO` pattern is a classic example of Pure Fabrication, where data access logic is encapsulated in specialized classes that shield domain objects from persistence concerns while maintaining high cohesion and low coupling. This separation allows domain objects to focus solely on business rules and behavior, while technical concerns are

handled by these fabricated classes.



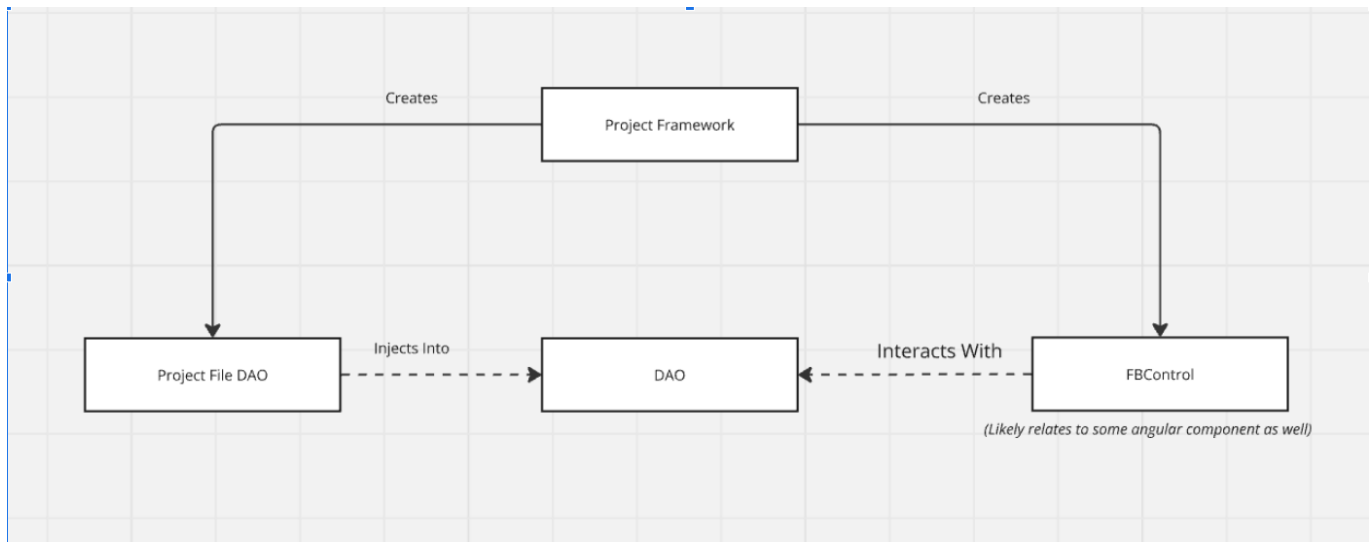
Dependency Injection:

Dependency Injection, at its simplest, is the principle that higher level modules should not depend upon lower level modules in order to function properly. When high level operations co-exist with lower level operations (say, for example, we are manipulating our database component directly in our funding basket) unit testing can be extremely difficult. We need to look for better solutions to our problems.

One common solution to this principle is dependency injections. Dependency injections operate by injecting the high level element into the low level module. By doing so, the low-level element depends only on the high-level element, so there is no dependency for the higher level module, which better adheres to Object Oriented design principles. There's no required time to do dependency injections—they can be done in the constructor, getter/setters, or as a parameter.

How We Applied It:

Dependency injections are implemented thoroughly in our service and controller classes, nearly in every class. Dependency injections are especially viable in our controllers, where we need access to Data Access Objects but do not want them as dependencies throughout our unit testing or as coupling. Our service classes also use dependency injections often as they allow us to interjoin two persistence classes without coupling together. One example is our `SearchService.java`. We wanted access to both `requestDAO` and `needDAO`, so we created a new service class that injects both via the Spring Framework. We found Dependency injection to be a very, very powerful tool in our arsenal of OO concepts. Often, we didn't even think about applying it to our project, it was second nature.

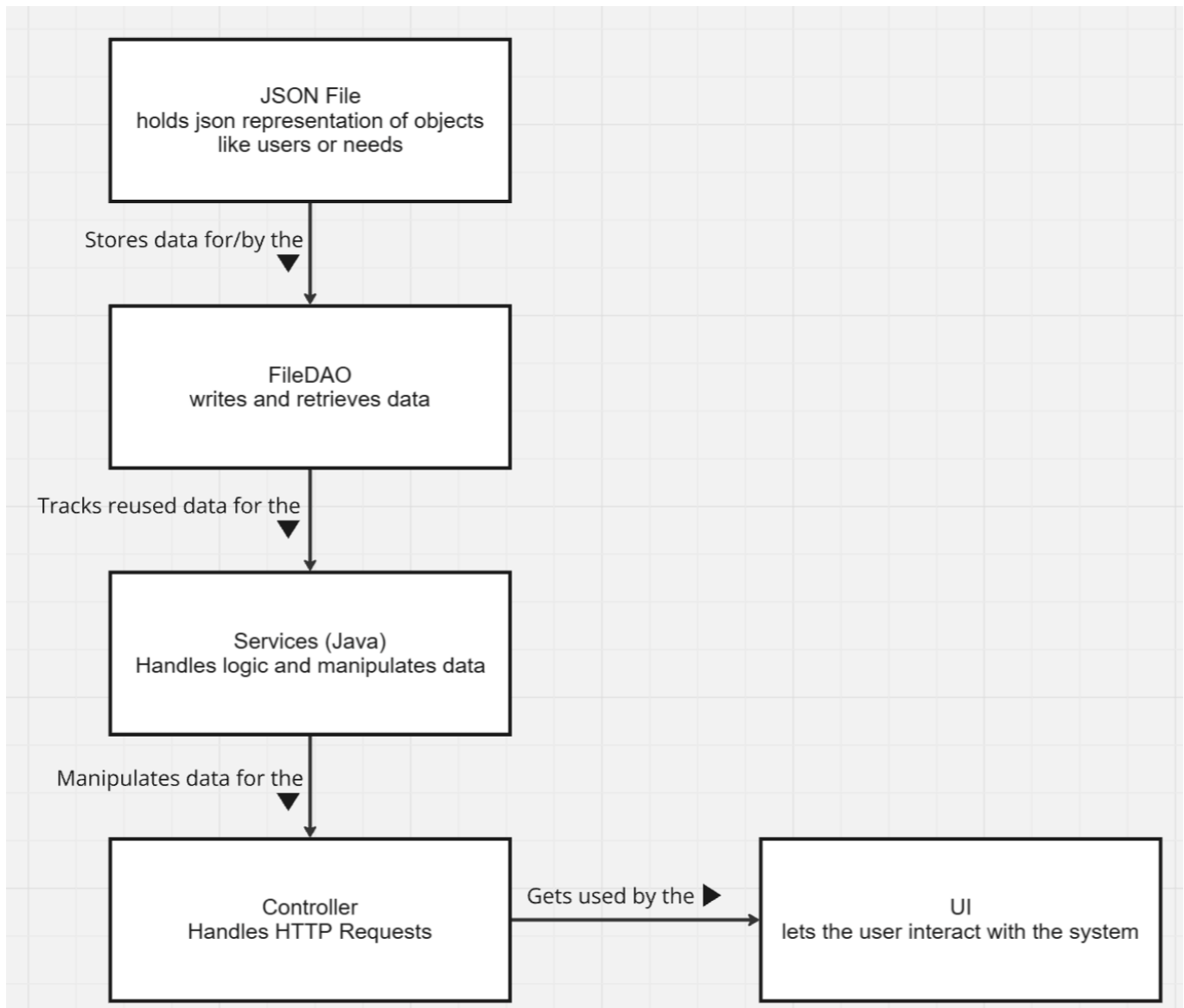


Single Responsibility :

Single Responsibility follows the concept that each class should only have one responsibility, and that it should not do more than it is required to. When adding new content to a class, one should ask themselves "does this belong in the class, or should I create a new class for this responsibility?" Single responsibility also clearly states that the class should have a clear, well defined task, so that two classes do not complete one task, or vice versa. By implementing single responsibility, we force ourselves to make smarter, more informed decisions on our Software Architecture, Design choices, and consider alternatives. Single Responsibility also has the benefit of not forming a "Kludge" which is known as an ill-assorted collection of parts made to fulfill a purpose. Through careful planning, decision making, and communication, Single Responsibility has created a modular, cleaner project that can be added to in the future.

How We Applied It:

Single responsibility can be seen in nearly all aspects of our project, specifically in our service classes. Service classes have one particular purpose: to fulfill a well defined task. For example, `SearchService`'s single solitary purpose is to handle searching requests by a search term. There is no other functionality to it, and it was not implemented in request service as that would be unrelated to its task, and would only create more clutter.



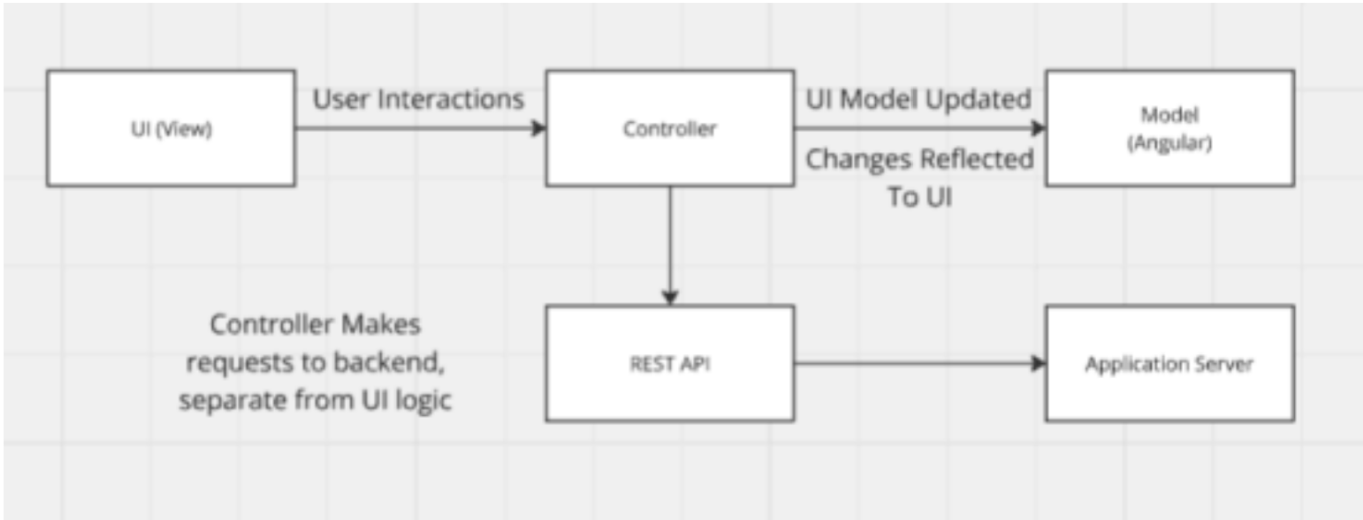
Controller :

The controller serves as a crucial intermediary component that provides a well-defined interface for the View to communicate user interactions and actions. By intercepting and processing these user-initiated events, the controller ensures proper coordination between the user interface and the underlying system operations. This architectural decision creates a clear channel of communication while maintaining a separation of concerns between different parts of the application. This separation establishes a distinct boundary between the presentation layer and the business logic, preventing direct coupling between these two critical aspects of the application. The View remains focused on displaying information and capturing user input, while the Model concentrates on data management and business rules. The controller acts as a mediator, ensuring that these components can operate independently while still working together effectively to achieve the application's goals.

How We Applied It:

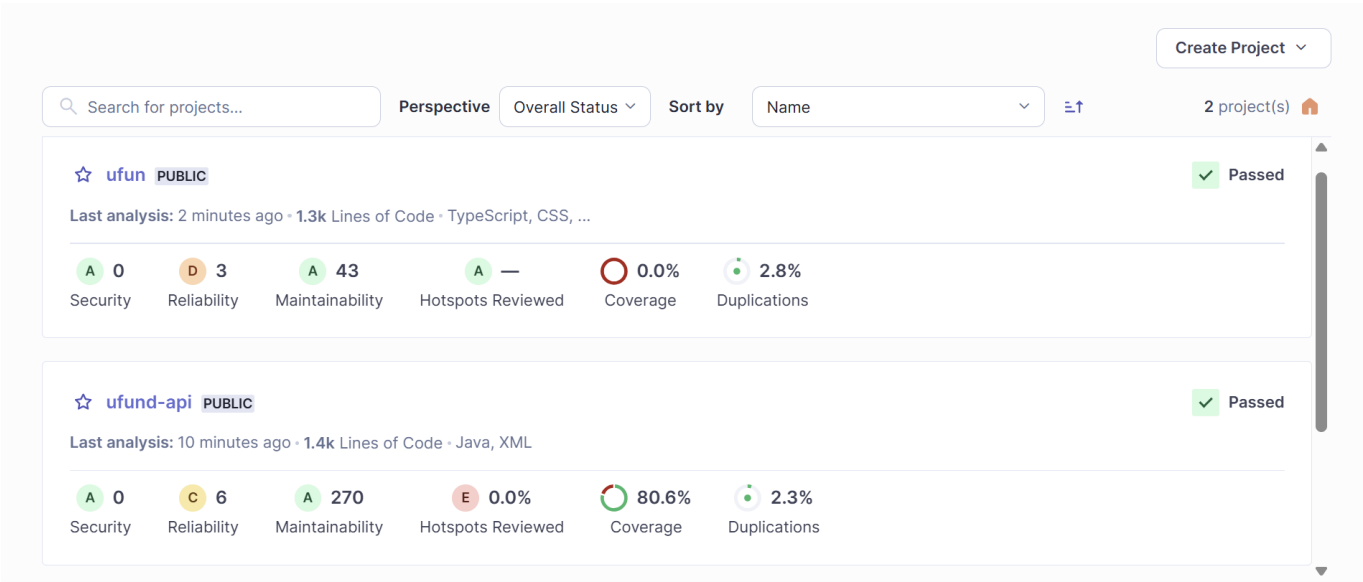
In our specific implementation, we adhered to the Controller and Single Responsibility Principle by designing a dedicated controller class that takes sole ownership of managing UI state changes and their corresponding effects on the system. This controller class is specifically tasked with intercepting user interface events and translating them into appropriate REST API calls to the Spring backend. By maintaining this single, well-defined responsibility, the controller promotes code maintainability, reduces complexity, and facilitates easier

testing and future modifications to the system.

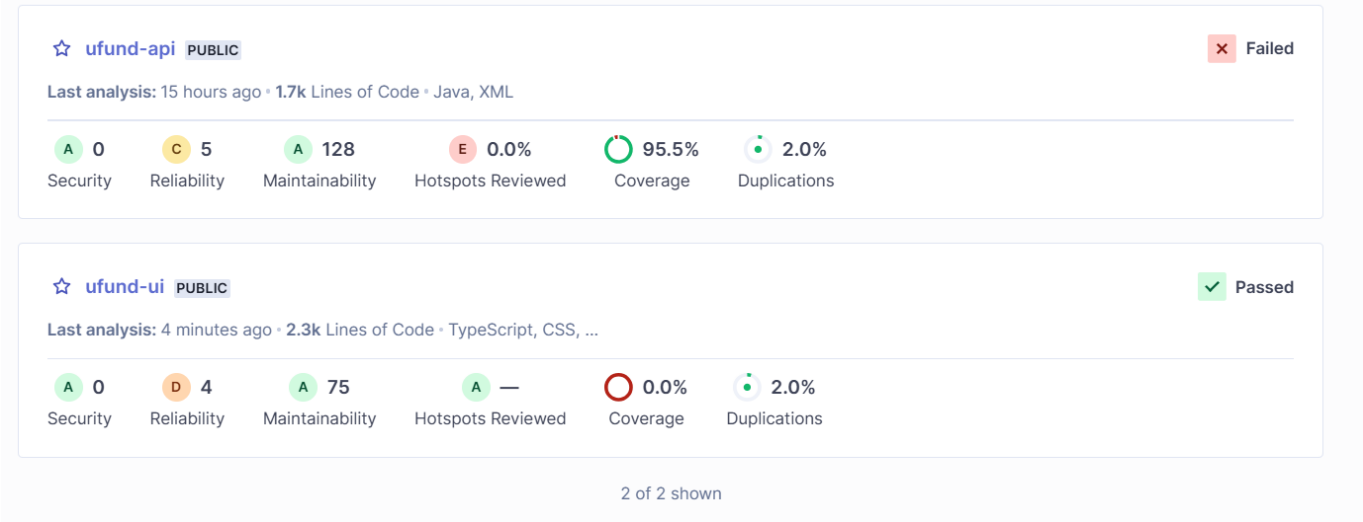


Static Code Analysis/Future Design Improvements

BEFORE:



AFTER:



Here are the results from our static code analysis. As you can see, both our projects have an A in maintainability. However, for some reason, SonarQube is reading our project as "failed" because our API has extremely minor issues (there are no major warnings). We are unsure of why this is, all of the issues SonarQube recommends are extremely minor. In fact, our project went down from 100+ issues to 23 on the API side. Our UI also went down from the hundreds to 75, which could be improved.

We did see an increase in our coverage for the API, for what was previously 80% is now around 95%. Maintainability decreased from 270 Open issues to only 128 in our API, meanwhile our UI went from 43 to 75. This is due to the fact we did not focus on lowering our static code analysis much during Sprint 4, we spent majority of our time on our backend, as we found it to be more important to the project.

Static Code Flags

```
@Test
void testLoginWithNullUsername() throws IOException
```

Replace these 4 tests with a single Parameterized one.

```
{
    // setup
    BasicUserInfo userInfo = new BasicUserInfo(
        null, "password123", "Teacher");
```

This is our first flag we found, where SonarQube was recommending us to replace our unit tests with parameterized ones. This is a pretty good idea, but it isn't 100% required. We should refactor it and fix it, though it doesn't cause much damage right now. Parameterized tests are a pretty good idea, but we also find them harder to read than 2-3 hard coded tests. For a larger product, we should definitely keep this in mind.

```
@GetMapping("/fromuid/{uid}")
public ResponseEntity<List<Request>> getRequestsById(@PathVariable Integer uid)
{
    return handleRoute(() -> {
```

Remove useless curly braces around statement and then remove useless return keyword

```
        return requestService.getRequestsById(uid);
    });
}
```

This is the second flag we found, where SonarQube was recommending we remove the useless curly brackets and get rid of the return, i.e. rewrite as `return handleRoute(() -> requestService.getRequestsById(uid))`. We don't really understand why this was such a problem, but it can certainly be fixed if we were given more time to develop the project.

```
input {
    margin-left: 0.5rem;
    width: 10rem;
    margin: 0 auto;
```

Unexpected shorthand "margin" after "margin-left"

This is the third flag we found, where SonarQube was recommending we remove this margin shorthand or put it before margin-left. This is a good idea because this margin actually overrides the CSS on margin-left.


```
<div class="form-control">
  <label>User Type</label> <br />
```

A form label must be associated with a control.

This is the fourth flag we found, where SonarQube was letting us know a label outside of a control is considered invalid HTML. This is important because some engines may not be able to render out page with invalid HTML, and it is also less accessible to some users who may utilize screen readers. We should fix this.

Testing

Acceptance Testing

SPRINT 2

9 User Stories have been completed. They have passed their acceptance criteria. The only story that could be tweaked is some of the u-fund manager admin stories. We are going to change it in the following sprint that the u-fund manager doesn't even have the option to click the checkout button, it will be disabled/grayed out. There weren't any serious issues, and we all performed well as a team.

SPRINT 3

12 User Stories have been completed, passing all acceptance criteria. Remaining time is being spent on cleaning up general bugs with out enhancements and code cleanup. There were no major issues found through acceptance testing, and we found it to be very helpful. Lots of bugs were fixed in this sprint, and our website as a whole came together. Enhancement features and CSS were both added this sprint which gave our site a modern looks, and features that seperated our site from competitors.

SPRINT 4

All user stories are complete with acceptance criteria.







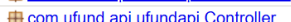

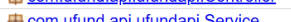

Unit Testing and Code Coverage

ufund-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi.Model	<div></div>	58%	<div></div>	0%	8	45	28	99	7	44	1	6
com.ufund.api.ufundapi.Service	<div></div>	56%	<div></div>	20%	13	22	30	66	8	17	1	5
com.ufund.api.ufundapi	<div></div>	0%	<div></div>	n/a	4	4	10	10	4	4	2	2
com.ufund.api.ufundapi.Controller	<div></div>	99%	<div></div>	95%	1	68	1	160	0	58	0	5
com.ufund.api.ufundapi.Persistence	<div></div>	99%	<div></div>	93%	3	63	1	162	0	39	0	3
Total	374 of 2,252	83%	14 of 80	82%	29	202	70	497	19	162	4	21

There are no major anomalies, though we are not at perfect coverage, especially for our service model. We need to finish authentication unit testing, but that can be done in sprint 3 as it is not required. One thing we are very happy about is 99% coverage in controller and persistence.

ufund-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi		0%		n/a	4	4	10	10	4	4	2	2
com.ufund.api.ufundapi.Model		95%		76%	9	70	2	132	1	53	0	5
com.ufund.api.ufundapi.Persistence		99%		89%	5	64	1	177	0	36	0	3
com.ufund.api.ufundapi.Controller		99%		96%	1	85	1	208	0	69	0	5
com.ufund.api.ufundapi.Service		99%		92%	3	53	0	166	0	26	0	7
Total	95 of 3,101	96%	19 of 176	89%	22	276	14	693	5	188	2	22

There are no major anomalies, in fact, it looks exactly as it should. We are at 99% coverage for Persistence, Controller, and Service. We are also at a fantastic 95% coverage for model spanning a total of 141 unit tests! We are satisfied with our unit testing for sprint 3.

Sprint 4 is the same as sprint 3, as we made no changes to the code.

Unit Testing Rationale

Some of the rationale we used throughout our testing was to utilize our resources (mockito, and jupiter) as much as possible. If you look into any of our controller or service classes, you will notice that we utilize mock objects to keep our unit tests isolated. We found mockito to be an exemplary tool throughout the unit testing process. We also communicated as a team that it was the priority of the developer who created the new class to write unit tests for it, which we followed often. If a developer had forgotten to do so, that was fine and it would be tested at a later date. This kept our unit testing coverage high for the majority of the sprint. Our major goals for coverage were to get over 90% coverage for each tier, because we felt it was a sufficient threshold that would not be difficult to reach, and make sure all of our features work as intended. We met all of our targets.

Ongoing Rationale

- (2025/ 02/ 17) Sprint 1: Finalized and agreed on a UI Style/structure.
- (2025/ 02/ 17) Sprint 1: Switched our naming conventions from snakecase to camel case
- (2025/ 03/ 02) Sprint 2: Conversation on Indentation
- (2025/ 03/ 06) Sprint 2: User Auth Conversation (Decided on Cookies)
- (2025/ 03/ 06) Sprint 2: Split up user stories, requests concept.
- (2025/ 03/ 12) Sprint 2: Decided to implement a grid to display needs to users.
- (2025/ 03/ 18) Sprint 2: Design Documentation Commentating with Aazeem. Updated Diagrams
- (2025/ 03/ 27) Sprint 3: Discussed color scheme for the application
- (2025/ 03/ 29) Sprint 3: Finalized and started implementing color scheme
- (2025/ 03/ 30) Sprint 3: Finalized functionality for user login persistence. Decided to store auth tokens in cookies instead of local storage after team discussion on security best practices.
- (2025/ 03/ 30) Sprint 3: Held a mid sprint meeting to delegate remaining tasks.
- (2025/ 04/ 04) Sprint 3: Major Bug Fixes, talk about sprint 3 finish
- (2025/ 04/ 06) Sprint 3: Updated/constructed diagrams for Design Documentation
- (2025/ 04/ 07) Sprint 3: Sprint 3 Demo/Release talk
- (2025/ 04/ 08) Sprint 4: Sprint 4 Planning, splitting up remaining tasks.
- (2025/ 04/ 14) Sprint 4: Major conversation about slideshow presentation, and what content should be shown.
- (2025/ 04/ 16) Sprint 4: Finished!!!