PROJECT Design Documentation

> *The following template provides the headings for your Design Documentation. As you edit each section make sure you remove these commentary 'blockquotes'; the lines that start with a > character and appear in the generated PDF in italics but do so only* ***after*** *all team members agree that the requirements for that section and current Sprint have been met.* ***Do not*** *delete future Sprint expectations.*

Team Information

- Team name: 8H
- Team members
  - Kelli Lung
  - Mina Baba
  - Jet Li
  - Jennifer Chen

Executive Summary

We are a food bank that allows users to buy individual items(needs) or supply cash to supply those in need with said items.

Purpose

> ***[Sprint 2 & 4]*** *Provide a very brief statement about the project and the most important user group and user goals.*

- A food bank that has multiple functions for users, such as checkout, and two enhancements, which are a donation bank, and a dashboard to display top donators and their donation amount.
- admin: The admin will use this application to edit the entire cupboard of needs, including deleting, updating, and adding needs
- user: The user will use this application as a donator, who will be able to add money that will be used to buy needs or directly buy needs themselves to support the cause.

Glossary and Acronyms

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

Term	Definition
SPA	Single Page
need	One individual item
basket	User Cart that holds the needs to be purchased
user	app user that can purchase needs
admin	app manager that modifies needs
DAO	Data Access Object that handles reading and writing to files

Term	Definition
Controller	Handles the logic behind need Objects
Login	A system that separates access between the admin and users
Model	Stores data objects and provides persistence
Cupboard	the inventory of needs
Persistence	Saving data even after the application stops running
contributions	monetary value of the user's donation

Requirements

This section describes the features of the application.

> *In this section you do not need to be exhaustive and list every story. Focus on top-level features from the Vision document and maybe Epics and critical Stories.*

Definition of MVP

> ***[Sprint 2 & 4]** Provide a simple description of the Minimum Viable Product.*

- The controller connects to the persistence classes so that it can return http statuses based on the input to the functions.
- The model contains the Need class which is the object class for the Needs.
- The persistence classes are responsible for saving the information of all of the created needs and for defining the logic behind need creation, deletion. etc.

MVP Features

> ***[Sprint 4]** Provide a list of top-level Epics and/or Stories of the MVP.*

- Cupboard actions (add, delete, update)
- Login (helper & admin)
- Basket actions (add, delete, decrement)
- Donation Bank
- Top Contributors Leaderboard
- Basket Checkout

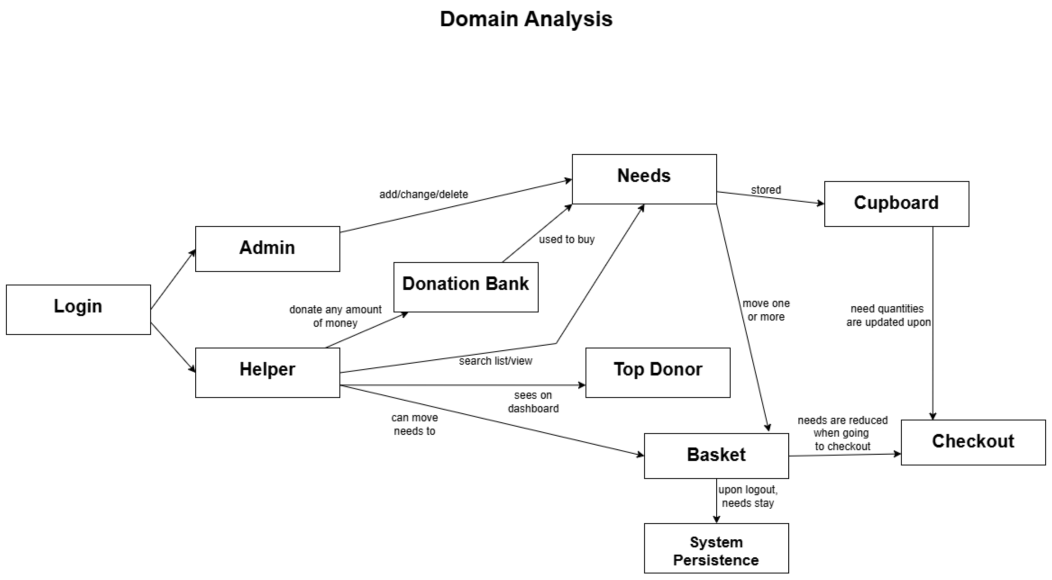
Enhancements

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

- donation bank: users are allowed to donate an x amount of money to the donation bank
- top contributors dashboard: displays the top 3 contributors by who donated the most

Application Domain

This section describes the application domain.



> ***[Sprint 2 & 4]*** *Provide a high-level overview of the domain for this application. You can discuss the more important domain entities and their relationship to each other.*

- Firstly, there are 2 different roles: the admin and user, which are separated through the login page. The admin creates, edits, and deletes needs in the cupboard which holds all of this information. For the user, they see the top donors on their dashboard and can donate any sum of money to the donation bank. They are allowed to add needs to their basket, and then check out, thereby purchasing them.

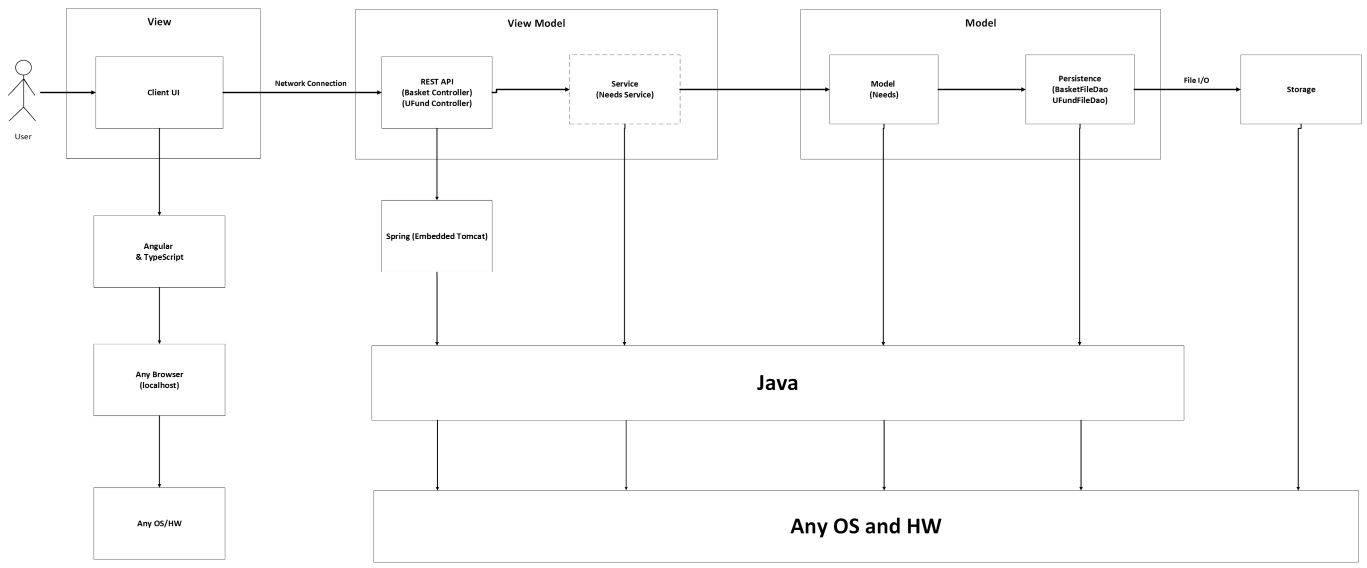
Architecture and Design

This section describes the application architecture.

Summary

The following Tiers/Layers model shows a high-level view of the webapp's architecture. **NOTE**: detailed diagrams are required in later sections of this document.

> ***[Sprint 1]*** *(Augment this diagram with your* ***own*** *rendition and representations of sample system classes, placing them into the appropriate M/V/VM (orange rectangle) tier section. Focus on what is currently required to support* ***Sprint 1 - Demo requirements****. Make sure to describe your design choices in the corresponding* ***Tier Section*** *and also in the* ***OO Design Principles*** *section below.)*



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 persistance.

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

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

Overview of User Interface

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

> *Provide a summary of the application's user interface. Describe, from the user's perspective, the flow of the pages/navigation in the web application. (Add low-fidelity mockups prior to initiating your **[Sprint 2]** work so you have a good idea of the user interactions.) Eventually replace with representative screen shots of your high-fidelity results as these become available and finally include future recommendations improvement recommendations for your **[Sprint 4]**)*

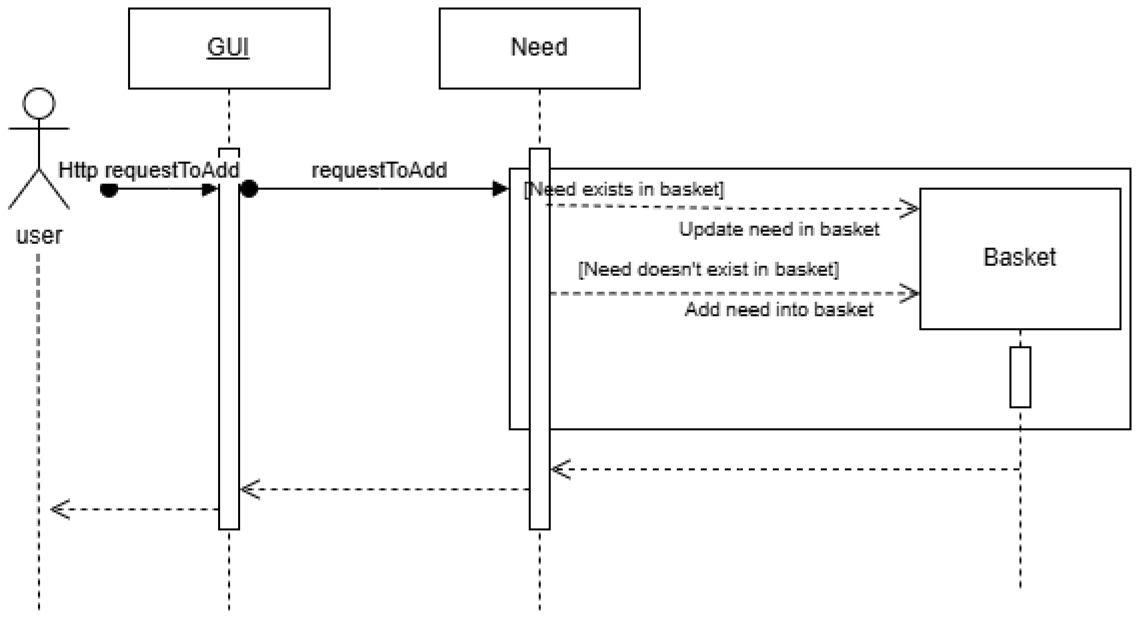
View Tier

> ***[Sprint 4]** Provide a summary of the View Tier UI of your architecture. Describe the types of components in the tier and describe their responsibilities. This should be a narrative description, i.e. it has a flow or "story line" that the reader can follow.*

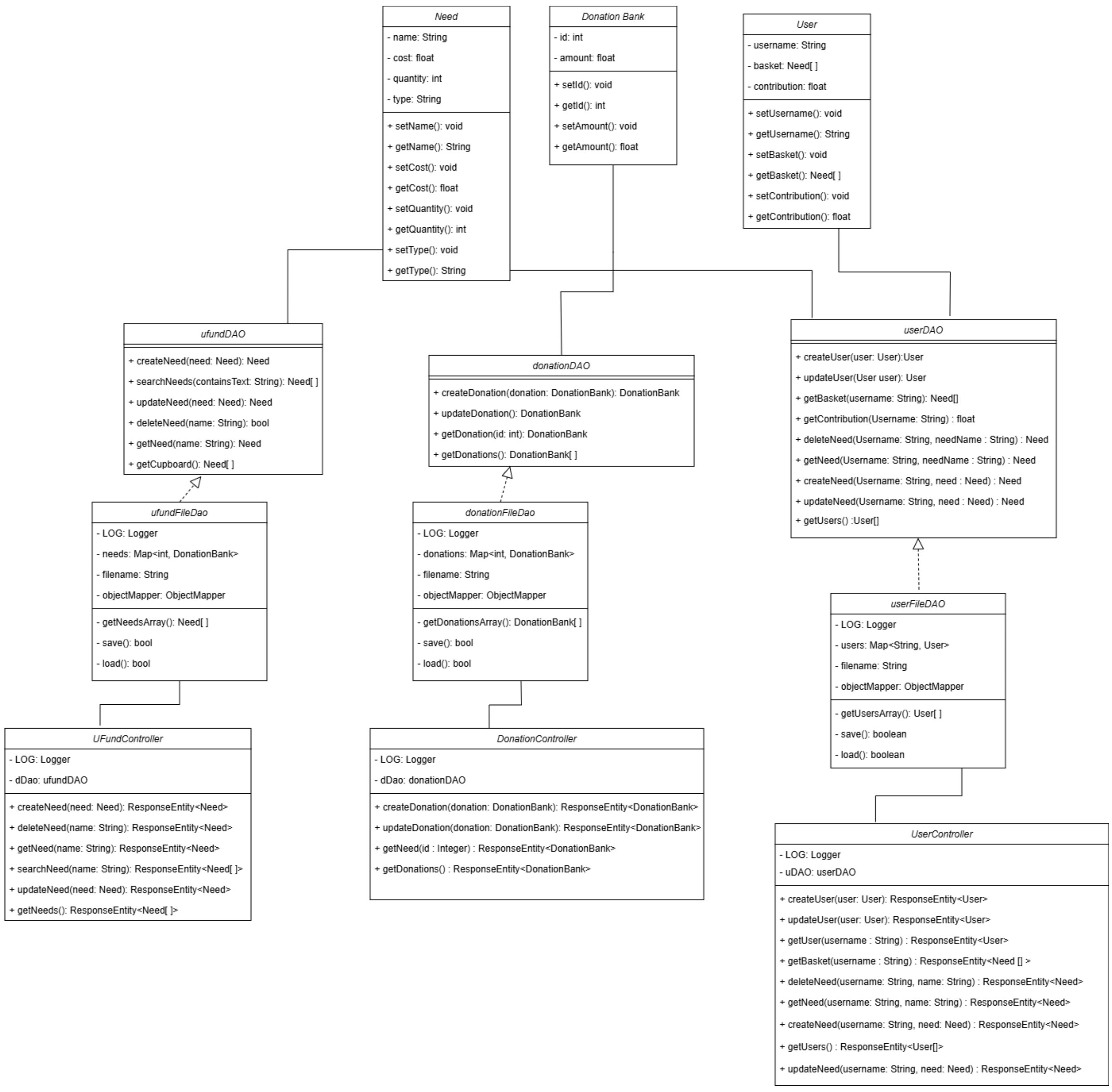
- Our UI starts off with a login page, where users and admin can put in the login information. Then, when admin logins, there is an admin page where the admin can update and create needs. Then when a user logins, there is a user page, where a user can do many actions, such as add, delete a need to their basket, checkout, donate and a dashboard for top 3 donators. Then, there is a logout button for whenever a user/admin wants to logout.

> ***[Sprint 4]** You must provide at least **2 sequence diagrams** as is relevant to a particular aspects of the design that you are describing. (**For example**, in a shopping experience application you might create a sequence diagram of a customer searching for an item and adding to their cart.) As these can span*

> *multiple tiers, be sure to include an relevant HTTP requests from the client-side to the server-side to help illustrate the end-to-end flow.*



> ***[Sprint 4]*** *To adequately show your system, you will need to present the* ***class diagrams*** *where relevant in your design. Some additional tips:*
>
> - *Class diagrams only apply to the* ***ViewModel*** *and* ***Model*** *Tier*
> - *A single class diagram of the entire system will not be effective. You may start with one, but will be need to break it down into smaller sections to account for requirements of each of the Tier static models below.*
> - *Correct labeling of relationships with proper notation for the relationship type, multiplicities, and navigation information will be important.*
> - *Include other details such as attributes and method signatures that you think are needed to support the level of detail in your discussion.*


Need
- name: String
- cost: float
- quantity: int
- type: String
+ setName(): void
+ getName(): String
+ setCost(): void
+ getCost(): float
+ setQuantity(): void
+ getQuantity(): int
+ setType(): void
+ getType(): String
Donation Bank
- id: int
- amount: float
+ setId(): void
+ getId(): int
+ setAmount(): void
+ getAmount(): float
User
- username: String
- basket: Need[ ]
- contribution: float
+ setUsername(): void
+ getUsername(): String
+ setBasket(): void
+ getBasket(): Need[ ]
+ setContribution(): void
+ getContribution(): float
ufundDAO
+ createNeed(need: Need): Need
+ searchNeeds(containsText: String): Need[ ]
+ updateNeed(need: Need): Need
+ deleteNeed(name: String): bool
+ getNeed(name: String): Need
+ getCupboard(): Need[ ]
donationDAO
+ createDonation(donation: DonationBank): DonationBank
+ updateDonation(): DonationBank
+ getDonation(id: int): DonationBank
+ getDonations(): DonationBank[ ]
userDAO
+ createUser(user: User):User
+ updateUser(User user): User
+ getBasket(username: String): Need[]
+ getContribution(Username: String) : float
+ deleteNeed(Username: String, needName : String) : Need
+ getNeed(Username: String, needName : String) : Need
+ createNeed(Username: String, need : Need) : Need
+ updateNeed(Username: String, need : Need) : Need
+ getUsers() :User[]
ufundFileDao
- LOG: Logger
- needs: Map<int, DonationBank>
- filename: String
- objectMapper: ObjectMapper
- getNeedsArray(): Need[ ]
- save(): bool
- load(): bool
donationFileDao
- LOG: Logger
- donations: Map<int, DonationBank>
- filename: String
- objectMapper: ObjectMapper
- getDonationsArray(): DonationBank[ ]
- save(): bool
- load(): bool
userFileDAO
- LOG: Logger
- users: Map<String, User>
- filename: String
- objectMapper: ObjectMapper
- getUsersArray(): User[ ]
- save(): boolean
- load(): boolean
UFundController
- LOG: Logger
- dDao: ufundDAO
+ createNeed(need: Need): ResponseEntity<Need>
+ deleteNeed(name: String): ResponseEntity<Need>
+ getNeed(name: String): ResponseEntity<Need>
+ searchNeed(name: String): ResponseEntity<Need[ ]>
+ updateNeed(need: Need): ResponseEntity<Need>
+ getNeeds(): ResponseEntity<Need[ ]>
DonationController
- LOG: Logger
- dDao: donationDAO
+ createDonation(donation: DonationBank): ResponseEntity<DonationBank>
+ updateDonation(donation: DonationBank): ResponseEntity<DonationBank>
+ getNeed(id : Integer) : ResponseEntity<DonationBank>
+ getDonations() : ResponseEntity<DonationBank>
UserController
- LOG: Logger
- uDAO: userDAO
+ createUser(user: User): ResponseEntity<User>
+ updateUser(user: User): ResponseEntity<User>
+ getUser(username : String) : ResponseEntity<User>
+ getBasket(username : String) : ResponseEntity<Need [] >
+ deleteNeed(username: String, name: String) : ResponseEntity<Need>
+ getNeed(username: String, name: String) : ResponseEntity<Need>
+ createNeed(username: String, need: Need) : ResponseEntity<Need>
+ getUsers() : ResponseEntity<User[]>
+ updateNeed(username: String, need: Need) : ResponseEntity<Need>

ViewModel Tier

> ***[Sprint 1]** List the classes supporting this tier and provide a description of there purpose.*

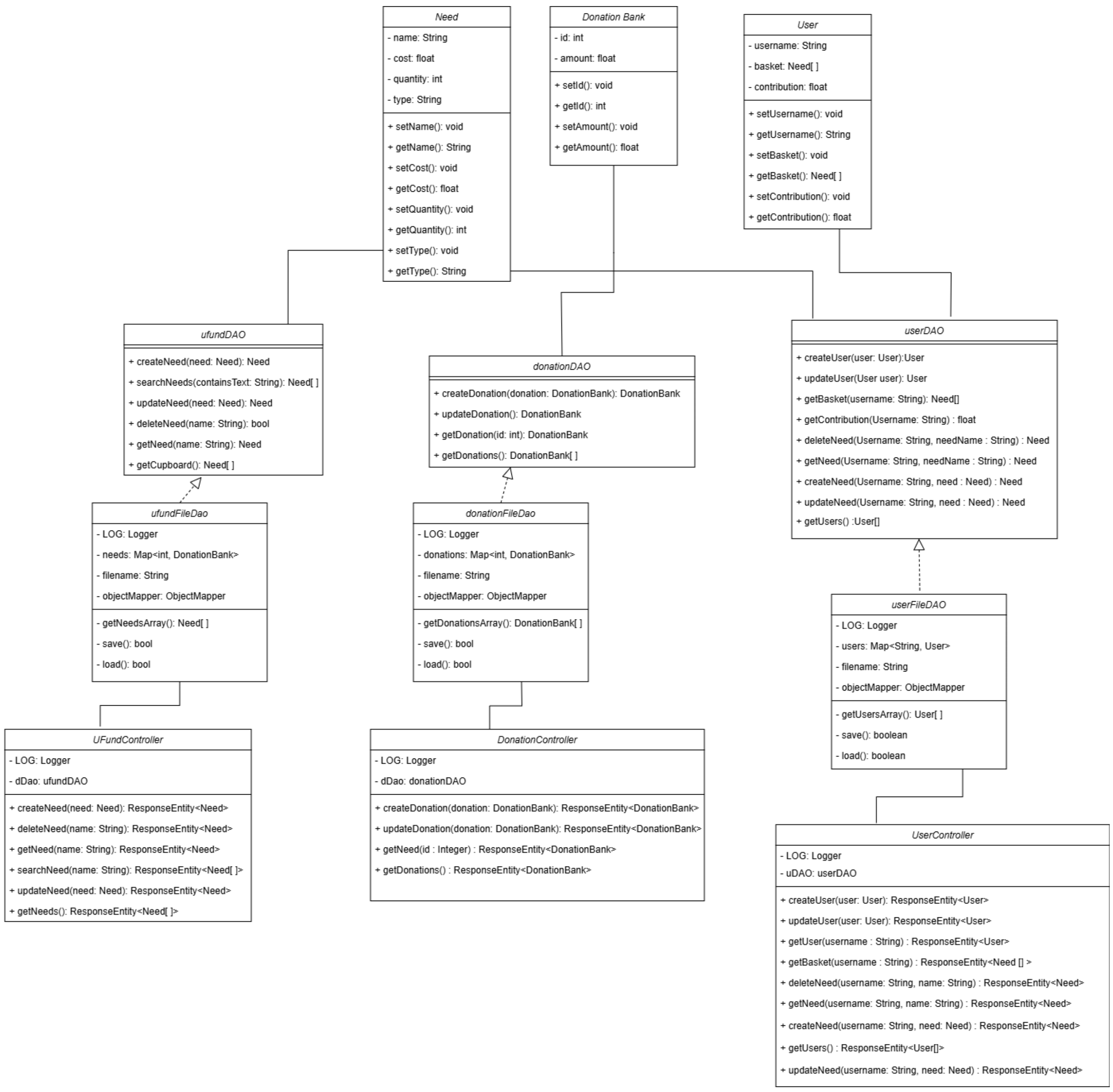
- UfundController.java: The controller class connects to the persistance class, UfundFileDao.java and handles user input. The class proceses it based on how each function is defined and returns a response.

> _**[Sprint 4]** Provide a summary of this tier of your architecture.

- DonationController.java
- UserController.java
- UFundController.java
  - These classes provide the response entity functionality of our application, which connects the user input with to the backend logic of our application.

> *At appropriate places as part of this narrative provide **one** or more updated and **properly labeled** static models (UML class diagrams) with some details such as associations (connections) between classes, and*

> *critical attributes and methods. (**Be sure** to revisit the Static **UML Review Sheet** to ensure your class diagrams are using correct format and syntax.)*


Need
- name: String
- cost: float
- quantity: int
- type: String
+ setName(): void
+ getName(): String
+ setCost(): void
+ getCost(): float
+ setQuantity(): void
+ getQuantity(): int
+ setType(): void
+ getType(): String
Donation Bank
- id: int
- amount: float
+ setId(): void
+ getId(): int
+ setAmount(): void
+ getAmount(): float
User
- username: String
- basket: Need[ ]
- contribution: float
+ setUsername(): void
+ getUsername(): String
+ setBasket(): void
+ getBasket(): Need[ ]
+ setContribution(): void
+ getContribution(): float
ufundDAO
+ createNeed(need: Need): Need
+ searchNeeds(containsText: String): Need[ ]
+ updateNeed(need: Need): Need
+ deleteNeed(name: String): bool
+ getNeed(name: String): Need
+ getCupboard(): Need[ ]
donationDAO
+ createDonation(donation: DonationBank): DonationBank
+ updateDonation(): DonationBank
+ getDonation(id: int): DonationBank
+ getDonations(): DonationBank[ ]
userDAO
+ createUser(user: User):User
+ updateUser(User user): User
+ getBasket(username: String): Need[]
+ getContribution(Username: String) : float
+ deleteNeed(Username: String, needName : String) : Need
+ getNeed(Username: String, needName : String) : Need
+ createNeed(Username: String, need : Need) : Need
+ updateNeed(Username: String, need : Need) : Need
+ getUsers() :User[]
ufundFileDao
- LOG: Logger
- needs: Map<int, DonationBank>
- filename: String
- objectMapper: ObjectMapper
- getNeedsArray(): Need[ ]
- save(): bool
- load(): bool
donationFileDao
- LOG: Logger
- donations: Map<int, DonationBank>
- filename: String
- objectMapper: ObjectMapper
- getDonationsArray(): DonationBank[ ]
- save(): bool
- load(): bool
userFileDAO
- LOG: Logger
- users: Map<String, User>
- filename: String
- objectMapper: ObjectMapper
- getUsersArray(): User[ ]
- save(): boolean
- load(): boolean
UFundController
- LOG: Logger
- dDao: ufundDAO
+ createNeed(need: Need): ResponseEntity<Need>
+ deleteNeed(name: String): ResponseEntity<Need>
+ getNeed(name: String): ResponseEntity<Need>
+ searchNeed(name: String): ResponseEntity<Need[ ]>
+ updateNeed(need: Need): ResponseEntity<Need>
+ getNeeds(): ResponseEntity<Need[ ]>
DonationController
- LOG: Logger
- dDao: donationDAO
+ createDonation(donation: DonationBank): ResponseEntity<DonationBank>
+ updateDonation(donation: DonationBank): ResponseEntity<DonationBank>
+ getNeed(id : Integer) : ResponseEntity<DonationBank>
+ getDonations() : ResponseEntity<DonationBank>
UserController
- LOG: Logger
- uDAO: userDAO
+ createUser(user: User): ResponseEntity<User>
+ updateUser(user: User): ResponseEntity<User>
+ getUser(username : String) : ResponseEntity<User>
+ getBasket(username : String) : ResponseEntity<Need [] >
+ deleteNeed(username: String, name: String) : ResponseEntity<Need>
+ getNeed(username: String, name: String) : ResponseEntity<Need>
+ createNeed(username: String, need: Need) : ResponseEntity<Need>
+ getUsers() : ResponseEntity<User[]>
+ updateNeed(username: String, need: Need) : ResponseEntity<Need>

Model Tier

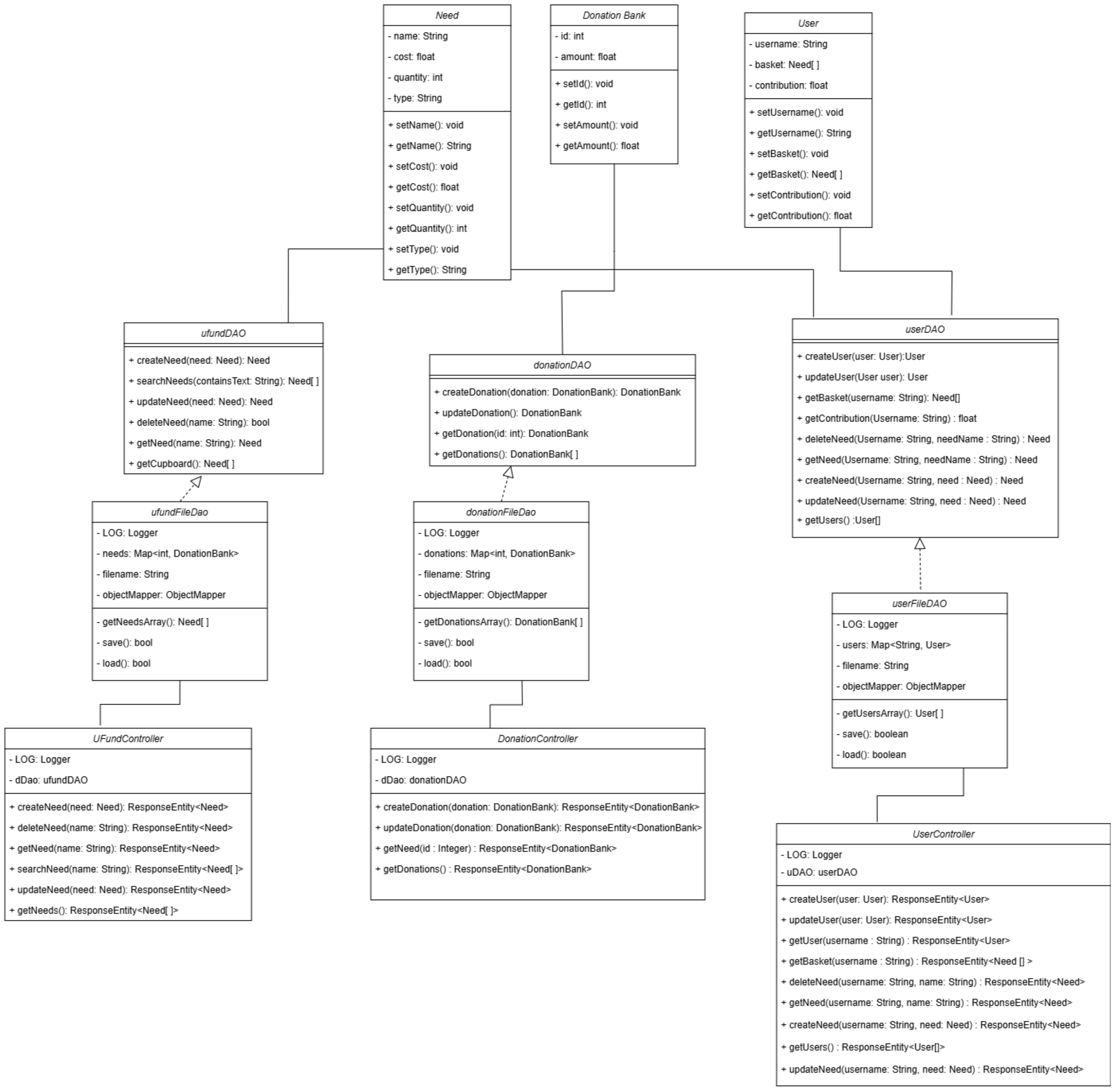
> ***[Sprint 1]** List the classes supporting this tier and provide a description of there purpose.*

- Need.java: The model class represents the structure of each Need object and each function is correlated with the Need object.

> ***[Sprint 2, 3 & 4]** Provide a summary of this tier of your architecture. This section will follow the same instructions that are given for the View Tier above.*

- The Model is responsible for handling the logic behind the whole project by defining the general properties of each Need function and defining the object class. In addition, the model tier handles the User class, which manages user-related information and roles. The Donation Bank class manages the properties of a donation, such as the amount.

> *At appropriate places as part of this narrative provide **one** or more updated and **properly labeled** static models (UML class diagrams) with some details such as associations (connections) between classes, and critical attributes and methods. (**Be sure** to revisit the Static **UML Review Sheet** to ensure your class diagrams are using correct format and syntax.)*


Need
- name: String
- cost: float
- quantity: int
- type: String
+ setName(): void
+ getName(): String
+ setCost(): void
+ getCost(): float
+ setQuantity(): void
+ getQuantity(): int
+ setType(): void
+ getType(): String
Donation Bank
- id: int
- amount: float
+ setId(): void
+ getId(): int
+ setAmount(): void
+ getAmount(): float
User
- username: String
- basket: Need[ ]
- contribution: float
+ setUsername(): void
+ getUsername(): String
+ setBasket(): void
+ getBasket(): Need[ ]
+ setContribution(): void
+ getContribution(): float
ufundDAO
+ createNeed(need: Need): Need
+ searchNeeds(containsText: String): Need[ ]
+ updateNeed(need: Need): Need
+ deleteNeed(name: String): bool
+ getNeed(name: String): Need
+ getCupboard(): Need[ ]
donationDAO
+ createDonation(donation: DonationBank): DonationBank
+ updateDonation(): DonationBank
+ getDonation(id: int): DonationBank
+ getDonations(): DonationBank[ ]
userDAO
+ createUser(user: User):User
+ updateUser(User user): User
+ getBasket(username: String): Need[]
+ getContribution(Username: String) : float
+ deleteNeed(Username: String, needName : String) : Need
+ getNeed(Username: String, needName : String) : Need
+ createNeed(Username: String, need : Need) : Need
+ updateNeed(Username: String, need : Need) : Need
+ getUsers() :User[]
ufundFileDao
- LOG: Logger
- needs: Map<int, DonationBank>
- filename: String
- objectMapper: ObjectMapper
- getNeedsArray(): Need[ ]
- save(): bool
- load(): bool
donationFileDao
- LOG: Logger
- donations: Map<int, DonationBank>
- filename: String
- objectMapper: ObjectMapper
- getDonationsArray(): DonationBank[ ]
- save(): bool
- load(): bool
userFileDAO
- LOG: Logger
- users: Map<String, User>
- filename: String
- objectMapper: ObjectMapper
- getUsersArray(): User[ ]
- save(): boolean
- load(): boolean
UFundController
- LOG: Logger
- dDao: ufundDAO
+ createNeed(need: Need): ResponseEntity<Need>
+ deleteNeed(name: String): ResponseEntity<Need>
+ getNeed(name: String): ResponseEntity<Need>
+ searchNeed(name: String): ResponseEntity<Need[ ]>
+ updateNeed(need: Need): ResponseEntity<Need>
+ getNeeds(): ResponseEntity<Need[ ]>
DonationController
- LOG: Logger
- dDao: donationDAO
+ createDonation(donation: DonationBank): ResponseEntity<DonationBank>
+ updateDonation(donation: DonationBank): ResponseEntity<DonationBank>
+ getNeed(id : Integer) : ResponseEntity<DonationBank>
+ getDonations() : ResponseEntity<DonationBank>
UserController
- LOG: Logger
- uDAO: userDAO
+ createUser(user: User): ResponseEntity<User>
+ updateUser(user: User): ResponseEntity<User>
+ getUser(username : String) : ResponseEntity<User>
+ getBasket(username : String) : ResponseEntity<Need [] >
+ deleteNeed(username: String, name: String) : ResponseEntity<Need>
+ getNeed(username: String, name: String) : ResponseEntity<Need>
+ createNeed(username: String, need: Need) : ResponseEntity<Need>
+ getUsers() : ResponseEntity<User[]>
+ updateNeed(username: String, need: Need) : ResponseEntity<Need>

OO Design Principles

> ***[Sprint 1]** Name and describe the initial OO Principles that your team has considered in support of your design (and implementation) for this first Sprint.*

- Single Responsibility: Each class is responsible for only one responsiblity
- Model: We want to use the Model-View-Controller strategy when designing our project

> ***[Sprint 2, 3 & 4]** Will eventually address upto **4 key OO Principles** in your final design. Follow guidance in augmenting those completed in previous Sprints as indicated to you by instructor. Be sure to include any diagrams (or clearly refer to ones elsewhere in your Tier sections above) to support your claims.*

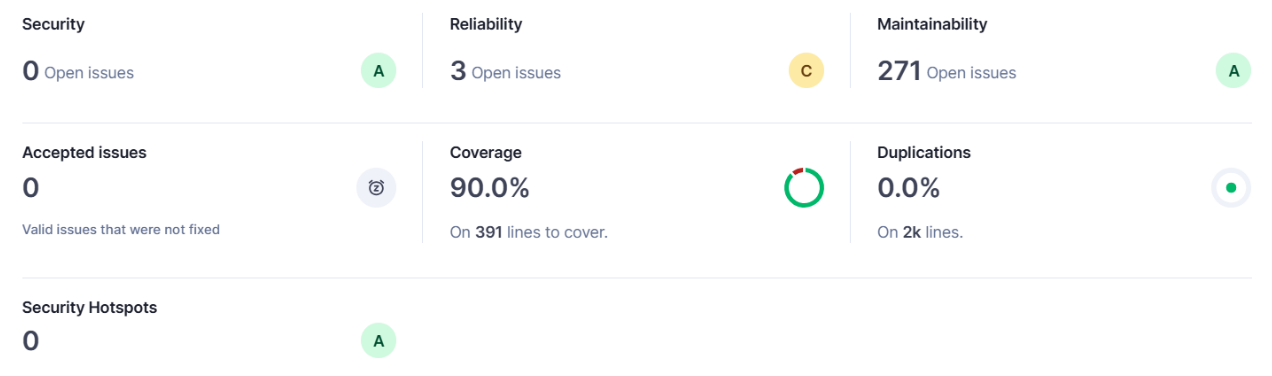
- We adopted the Model-View-Controller design by having a UI, a controller folder, and a model folder, and each of these handle only their own requirements.
- We adopted single responsibility by separating the concerns of each class. For example, the basketController is only responsible for managing requests related to the basket, as in adding to the basket and deleting from the basket, and so on.
- We adopted Law of Demeter by having the angular components rely on services to fetch data rather than accessing APIs or models directly.
- In addition, the project uses dependency injection principle because in angular, the services are injected in components through constructors, keeping the logic separate from the UI.

> ***[Sprint 3 & 4]*** *OO Design Principles should span across* ***all tiers.***

Static Code Analysis/Future Design Improvements

> ***[Sprint 4]*** *With the results from the Static Code Analysis exercise,* ***Identify 3-4*** *areas within your code that have been flagged by the Static Code Analysis Tool (SonarQube) and provide your analysis and recommendations.*
> *Include any relevant screenshot(s) with each area.*

- An area within our code that was partially flagged was reliability and maintainability.



Security 0 Open issues — A	Reliability 3 Open issues — C	Maintainability 271 Open issues — A
Accepted issues 0 Valid issues that were not fixed	Coverage 90.0% On 391 lines to cover.	Duplications 0.0% On 2k lines.
Security Hotspots 0 — A		

> ***[Sprint 4]*** *Discuss* ***future*** *refactoring and other design improvements your team would explore if the team had additional time.*

- If the team had additional time, other design improvements we would like to explore are extra security measures, such as a password system and password encryption and limited accessibility to certain roles.

Testing

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

Acceptance Testing

> ***[Sprint 2 & 4]*** *Report on the number of user stories that have passed all their acceptance criteria tests, the number that have some acceptance criteria tests failing, and the number of user stories that have not had any testing yet. Highlight the issues found during acceptance testing and if there are any concerns.*

- 10 stories have all acceptance criteria tests passed.
- 1 story has 1 acceptance criteria failed

- 3 stories have no testing
- the story that has 1 acceptance criteria failed is that the search by partial gets some needs that don't have the letter
- all the stories has passed all the acceptance criteria

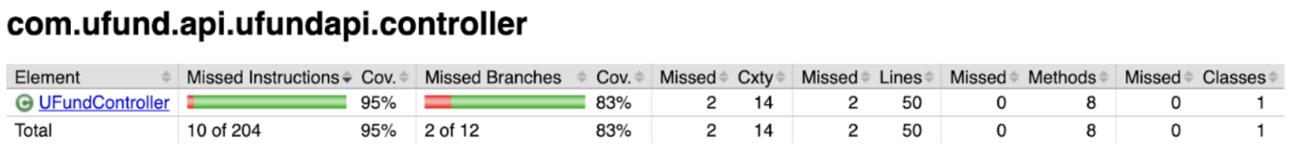
Unit Testing and Code Coverage

> ***[Sprint 4]*** *Discuss your unit testing strategy. Report on the code coverage achieved from unit testing of the code base. Discuss the team's coverage targets, why you selected those values, and how well your code coverage met your targets.*

- We made our own test files for the controller, model and persistence classes and tested each method and if it is not found. To find our code coverage, we ran sonarqube and docker desktop to find the total code coverage from our test files.

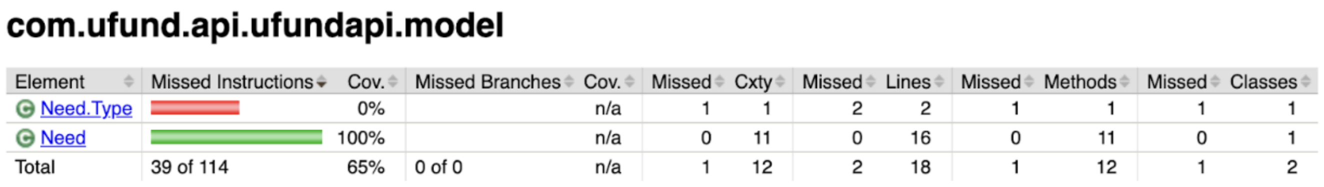
> ***[Sprint 2, 3 & 4] Include images of your code coverage report.*** *If there are any anomalies, discuss those.*

Sprint 2:



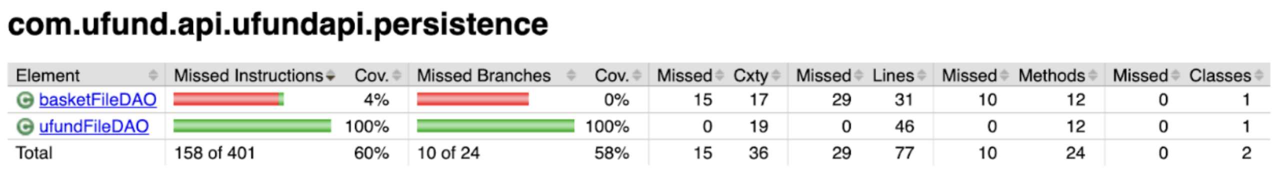
com.ufund.api.ufundapi.controller

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
UFundController		95%		83%	2	14	2	50	0	8	0	1
Total	10 of 204	95%	2 of 12	83%	2	14	2	50	0	8	0	1



com.ufund.api.ufundapi.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Need.Type		0%		n/a	1	1	2	2	1	1	1	1
Need		100%		n/a	0	11	0	16	0	11	0	1
Total	39 of 114	65%	0 of 0	n/a	1	12	2	18	1	12	1	2

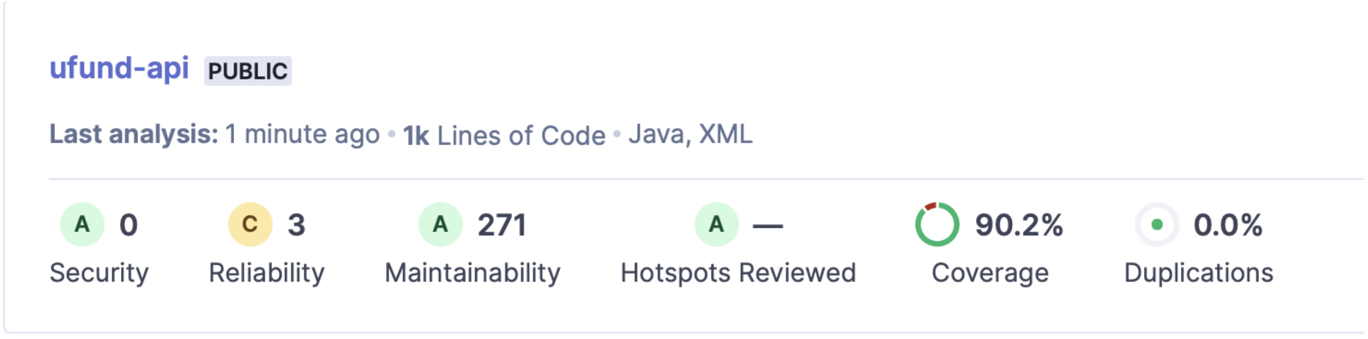


com.ufund.api.ufundapi.persistence

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
basketFileDAO		4%		0%	15	17	29	31	10	12	0	1
ufundFileDAO		100%		100%	0	19	0	46	0	12	0	1
Total	158 of 401	60%	10 of 24	58%	15	36	29	77	10	24	0	2

Summary: Overall the code coverage percentages were high, except for need.type because we decided not to use the need type enum so there weren't any tests for it. Also basketFileDao had not yet been written by the time we did that code coverage so there weren't tests for it either.

Sprint 3:

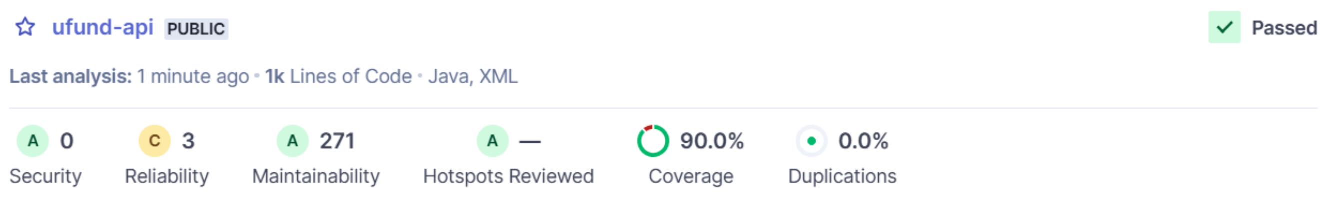


ufund-api PUBLIC

Last analysis: 1 minute ago • 1k Lines of Code • Java, XML

A 0	C 3	A 271	A —	90.2%	0.0%
Security	Reliability	Maintainability	Hotspots Reviewed	Coverage	Duplications

Sprint 4:


ufund-api PUBLIC Passed

Last analysis: 1 minute ago • 1k Lines of Code • Java, XML

A 0	C 3	A 271	A —	90.0%	0.0%
Security	Reliability	Maintainability	Hotspots Reviewed	Coverage	Duplications

Ongoing Rationale

> ***[Sprint 1, 2, 3 & 4]*** *Throughout the project, provide a time stamp* ***(yyyy/mm/dd): Sprint # and description*** *of any* ***mayor*** *team decisions or design milestones/changes and corresponding justification.*

Sprint 1: 2/20/2025 Using String instead of Enum for need type

- We decided to use a String for the type of each need instead of Enum to increase simplicity and because it made more sense to us if the type can be anything.

Sprint 1: 2/22/2025 Not using ID

- We decided to just use name instead of ID to increase simplicity and because we felt we could do it with just the name alone.

Sprint 2: 3/6/2025 Adding Basket Files

- We decided to add a file for BasketController and BasketFileDAO to separate the logic behind the basket and the cupboard so that it will be more organized.

Sprint 2: 3/16/2025 Moving Basket

- We decided to display the basket at the bottom of the user page instead of making it its own page to increase simplicity.

Sprint 3: 4/6/25 New Component For Enhancement

- For the Top Contributors enhancement, we added an angular component for the top donators.

Sprint 3: 3/30/25 Implemented Donation Bank Classes

- We decided that for our first enhancement, Donation Bank wpuld require model, controller and persistence classes. In addition, we made Donation Bank a singular object that we continously added an amount to.

Sprint 4: 4/18/2025 Implemented images for Needs

- For the user page, we decided to implement images for each need, so as you press on the description, it turns into an image of the need and then press it again, it will go back to its description.