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: 6f
- Team members
  - Eric Zheng Wu
  - Evan Lin
  - Alexandra Mantagas
  - Sophia Le

Executive Summary

This project revolves around making a website to help communities affected by wildfires by connecting them with donors to provide them with any needs they may require. Relief organizations sometimes struggle to manage and track these needs effectively, so we're creating a simple solution that is user-friendly to make the process easier.

Purpose

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

The U-Fund project is a web-based platform designed to help community with financial assistance. It allows users to create and contribute to funding requests for essential needs. The primary user groups include individuals seeking financial support and donors willing to contribute.

Glossary and Acronyms

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

Term	Definition
SPA	Single Page
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
CRUD	Create, Read, Update, Delete
REST	Representational State Transfer
DAO	Data Access Object

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.*

> *Sprint 2*

> *User Authentication: Users should be able to log in and log out to the system and basic security mechanisms like password validation*

> *Data Management (DAO Layer - CupboardDAO.java, CupboardFileDAO.java): Provides basic Create, Read, Update, Delete (CRUD) operations and stores data in a simple file.*

> *Basic Logic (Controller Layer - CupboardController.java, InventoryController.java): Handles requests for managing cupboard items.*

MVP Features

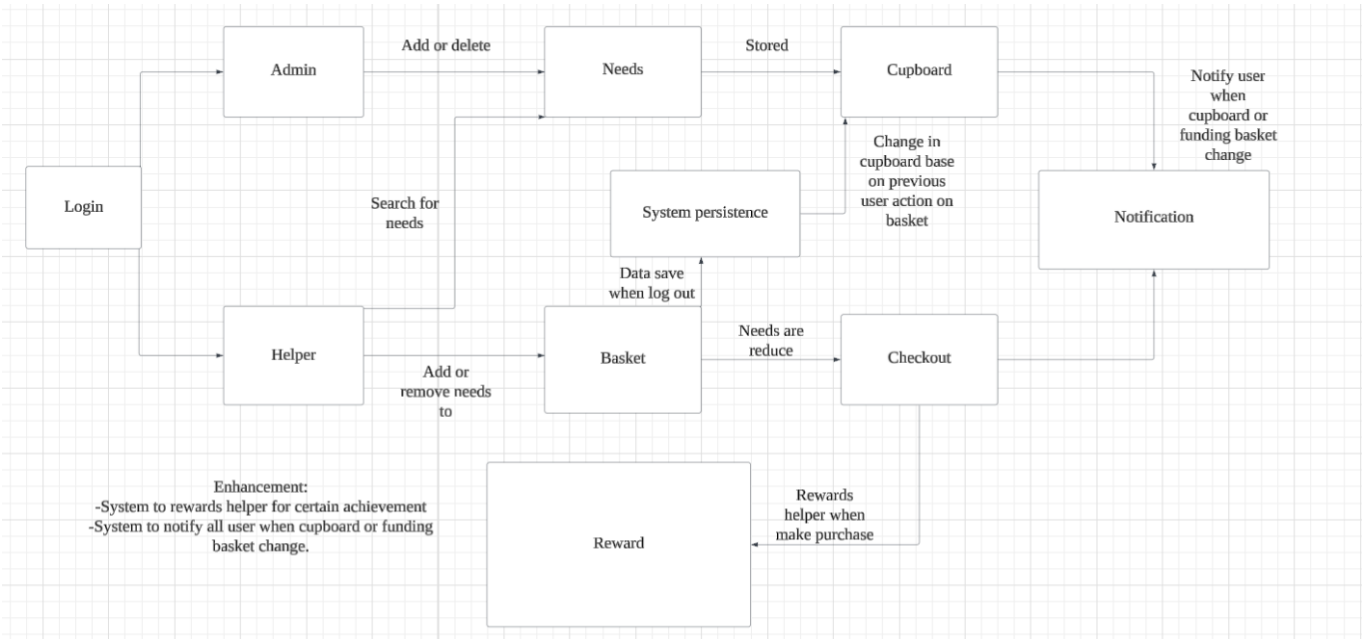
> ***[Sprint 4]*** *Provide a list of top-level Epics and/or Stories of the MVP.*

Enhancements

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

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.*
>
> User: A person using the system, either a Helper or a U-Fund manager. Can interact with Needs and Funding Basket.
>
> Need: A request for resources. Stored in Cupboard, managed by U-Fund managers, funded by Helpers.
>
> Cupboard: Holds all active Needs. Accessed by U-Fund managers.
>
> Funding Basket: A temporary selection of Needs a Helper wants to fund. Belongs to a specific Helper.
>
> Authentication System: Manages user login and role-based access. Determines if user is a Helper or a U-Fund Manager.

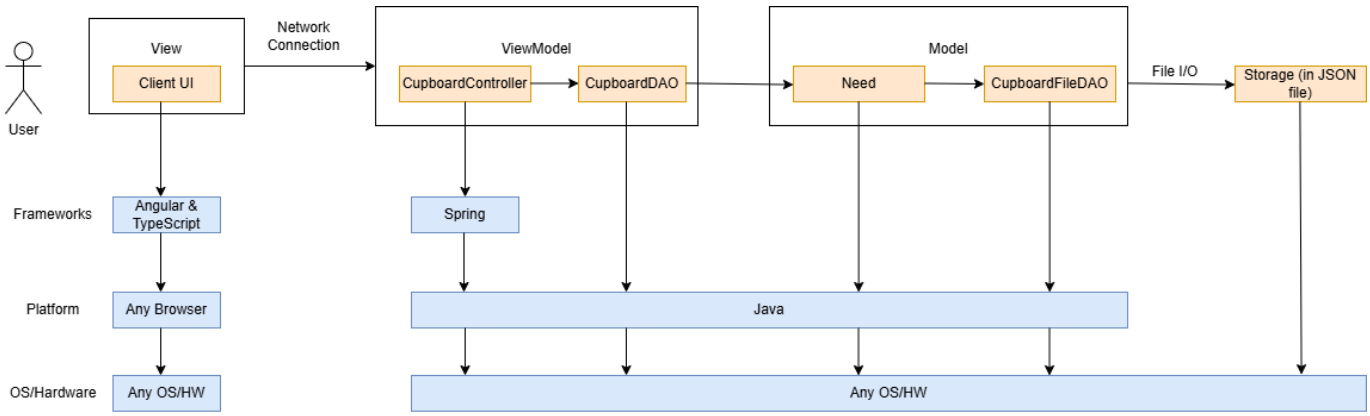
Architecture and Design

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.*

> ***[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.*

The View Tier is responsible for the user interface and interactions. It provides a seamless and intuitive experience for both wildfire relief managers and supporters. This tier consists of:

Landing Page: Introduces the platform and provides access to login/signup.

Helper Dashboard: Allows supporters to browse needs, add them to their funding basket, and proceed with the donation process.

Manager Dashboard: Enables relief managers to create, update, and remove wildfire relief needs.

Checkout Page: Facilitates the funding process with a simple and secure transaction mechanism.

ViewModel Tier

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

> *CupboardController: Implements the REST API endpoints for CRUD operations for wildfire relief needs. It also interacts with the CupboardDAO file for data persistence.*

> *CupboardDAO (interface): Defines the methods for need management, such as retrieving, adding, and updating a need.*

> ***[Sprint 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.*

> *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.)*

 Replace with your ViewModel Tier class diagram 1, etc.

The ViewModel Tier bridges the View Tier and Model Tier by handling business logic and API interactions. This tier consists of:

CupboardController: Implements REST API endpoints for CRUD operations on wildfire relief needs and interacts with the CupboardDAO for data persistence.

Model Tier

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

> *Model:* _Need: Represents a single need. Contains the attributes name, cost, quantity, and type. _ *CupboardFileDAO: Reads and writes needs to a JSON file.*

> ***[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.*

> *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.)*

 Replace with your Model Tier class diagram 1, etc.

The Model Tier is responsible for data representation and persistence. It ensures that wildfire relief needs and funding transactions are accurately managed. This tier consists of:

Need: Represents a single wildfire relief need with attributes such as name, cost, quantity, and urgency level.

CupboardFileDAO: Handles reading and writing of relief needs to a JSON file, ensuring data persistence.

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 Principle (SRP): Need is responsible only for storing data about a need.*

> *Open/Closed Principle (OCP): CupboardDAO is an interface which allows for future modifications without changing the existing logic.*

> *Controller: Component that manages user input and interacts with the model and view to process requests and update the user interface.*

> *Pure Fabrication: Designed to improve the overall system by encapsulating complex logic or providing a clean interface between > other classes. Dependency Inversion/Injection: important parts of the code should not rely on details. They should rely on general ideas or rules. We follow this by using interfaces.*

> *Law of Demeter: The object should know as little as possible about the others, basically everything should be private. Other objects should not have direct access to the data. For example, the getters we use.*

> *Low Coupling: One person can edit the interface while another team member can work on the backend development at the same time.*

> *Low Coupling: The Need class is the information expert for all need-related data.*

> ***[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.*

OO PRINCIPLE #1: Single responsibility. This principle states that a class should have only one reason to change (it should only have one job). We applied this in our project by separating concerns across multiple different classes, and making sure each class has a focused purpose. For example, with the Need management system, everything was broken down into different parts. The Need Class (Model Layer) represents an individual need and only contains data and basic validation. Next, the CupboardDAO Interface defines operations for adding, retrieving, updating, and deleting needs from the cupboard. After that, the CupboardFileDAO (Persistence Layer) implements CupboardDAO and handles the file-based data storage. It ensures that anything to do with persistence is separate from the business login. Finally, CupboardController handles HTTP requests/responses for needs operations. This allows for easier maintenance and better readability.

OO PRINCIPLE #2: Open/closed principle. This principle states that a class should be open for extension but closed for modification (new functionality can be added without changing existing code). We applied this by making key components extensible, instead of having modifications be made to existing code. For example, the LoginService class determines user roles based on their username (admin = U-Fund Manager, anything else = Helper). In the future, if another role needs to be added, we can extend the system instead of modifying the existing logic. In addition, instead of hardcoding different need categories, we could use polymorphism to add these things easily and without modifying existing code. This allows for easier expansion of our code in the future.

> ***[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.*

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

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.*

Sprint 2: Login System user story passed all acceptance criteria tests.

Sprint 2: Search for a need user story passed all acceptance criteria tests.

Sprint 2: Add/remove a helper user story passed all acceptance criteria tests.

Sprint 2: Add/remove/edit needs user story passed all acceptance criteria tests.

Sprint 2: Create admin account, funding basket obscurity not tested.

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.*

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

ufund-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi.controller		95%		80%	11	45	3	91	0	17	0	3
com.ufund.api.ufundapi.dao		97%		88%	2	18	1	32	0	9	0	1
com.ufund		0%		n/a	1	1	1	1	1	1	1	1
com.ufund.api.ufundapi.log		97%		75%	1	14	1	24	0	12	0	2
com.ufund.api.ufundapi.model		100%		n/a	0	9	0	18	0	9	0	1
com.ufund.api.ufundapi		100%		n/a	0	4	0	7	0	4	0	2
Total	26 of 733	96%	14 of 78	82%	15	91	6	173	1	52	1	10

We did not run into any anomalies while testing.

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.*

2025/03/03: Sprint 2, implemented basic login system where user can log in as either a U-Fund Manager or a Helper.

2025/03/18: Sprint 2, implemented manager dashboard and ability for manager to add/remove/edit needs.

2025/03/19: Sprint 2, implemented helper dashboard and ability for helper to add/remove needs to their basket and search for needs.