PROJECT Design Documentation

Team Information

- Team name: Team 1
- Team members
  - Rafid Khan
  - Ryan Ackerley
  - Patrick Wynne
  - Victoria Rigoglioso

Executive Summary

This web application allows users to contribute to a homeless charity. The application is managed by the administrator who can manage the current needs. The user can contribute to the needs they'd like to.

Purpose

This project is an application which simulates a fundraiser. There is a collection of several 'needs' (organized by the administrator) that require funds to advance by users. There are two user groups, the helpers, which select the needs that they'd like to contribute to, and do so and the administrators, who manage the fundraiser, and the specfic needs in the agenda.

Glossary and Acronyms

Term	Definition
SPA	Single Page
Needs	Elements of the fundraiser
Helper	Users who are looking to contribute to the fundraiser
Admin	Manager of the fundraiser
Data Persistance	System has data integrity, changes are saved
Minimal Authentication	Security of user login and privileges

Requirements

In a simplified authentication system for a Helper/U-fund Manager application, users log in with just a username, where logging in as "admin" identifies a user as the U-fund Manager. Helpers can view, search, add, or remove needs from their funding basket and checkout to fund chosen needs. U-fund Managers have the authority to add, remove, and edit needs in the system but cannot view Helpers' funding baskets. All data is persisted to files, ensuring that changes are reflected for the next user session. Users will also have the ability to submit suggestion messages to the U-fund Manager.

Definition of MVP

The MVP for the U-Fund project includes minimal authentication for user login, helper functionality enabling viewing, searching, and funding of needs, alongside needs management capabilities for U-Fund managers, all ensuring data persistence across sessions.

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.

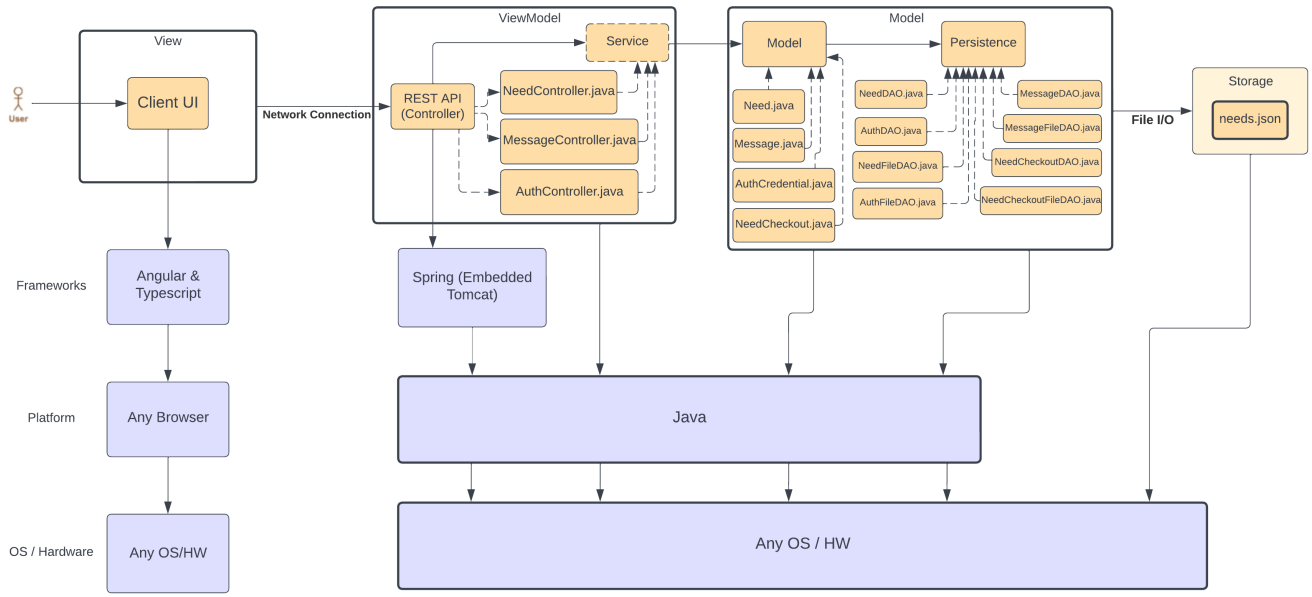
In this domain model, the relationships between the User / Helper and the Needs and Funding Basket are clearly defined. Similarly, the relationships with the U-Fund administrator are defined as well. It is clear that the administrator is the one in charge of of managing the cupboard, which contains the needs that the User / Helper can add to their funding basket, wishlist, or checkout.

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 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.

Our application's user interface consists of a website where users are first prompted with a log-in screen. Following this, they are presented a window which contains the Needs Cupboard, containing all of the available Needs. They can select a Need from this window which moves the Need to the Helper's funding basket, where they can then proceed to checkout. For a U-Fund Manager, the site appears the same, however there are buttons / options for the Manager to be able to manage the Cupboard (i.e, add, remove, edit Needs).

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.*

ViewModel Tier

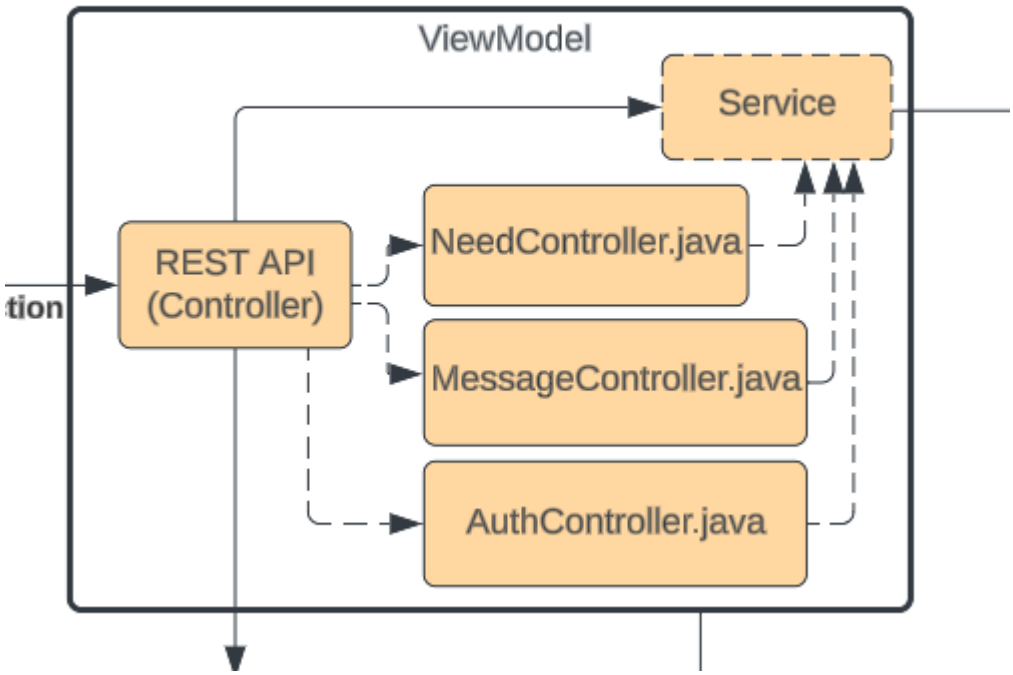
NeedController.java Manages REST API requests for the Need resource, handling operations related to needs within the application. It interfaces with NeedService and AuthService to perform actions like retrieving, creating, updating, and deleting needs based on user permissions. This controller ensures that only authorized users can perform sensitive operations, aligning with the system's security and business logic. By managing how needs are interacted with, NeedController directly supports the Needs Management and Helper functionality, providing endpoints for Helpers to add needs to their funding basket, and for U-fund Managers to manage needs in the cupboard.

MessageController.java Handles communication within the system by managing REST API requests for messaging functionalities. It uses MessageService for operations on messages, such as retrieving all messages, fetching messages from a specific user, creating, and deleting messages. The controller ensures that these operations are performed according to the user's authorization level, facilitated by AuthService. This setup enables a dynamic interaction platform within the U-Fund application, allowing users to communicate needs, updates, and information effectively, enriching the user experience and engagement.

AuthController.java Responsible for authentication-related actions within the application, AuthController interfaces with AuthService to manage authorization levels and credentials verification. It provides endpoints for checking a user's permission level, enhancing the system's security by ensuring that operations are executed by users with the appropriate access rights. This controller is crucial for maintaining the integrity and security of the U-Fund application, directly supporting the Minimal Authentication feature by managing user login and logout functionalities.

> ***[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 critical attributes and methods.*



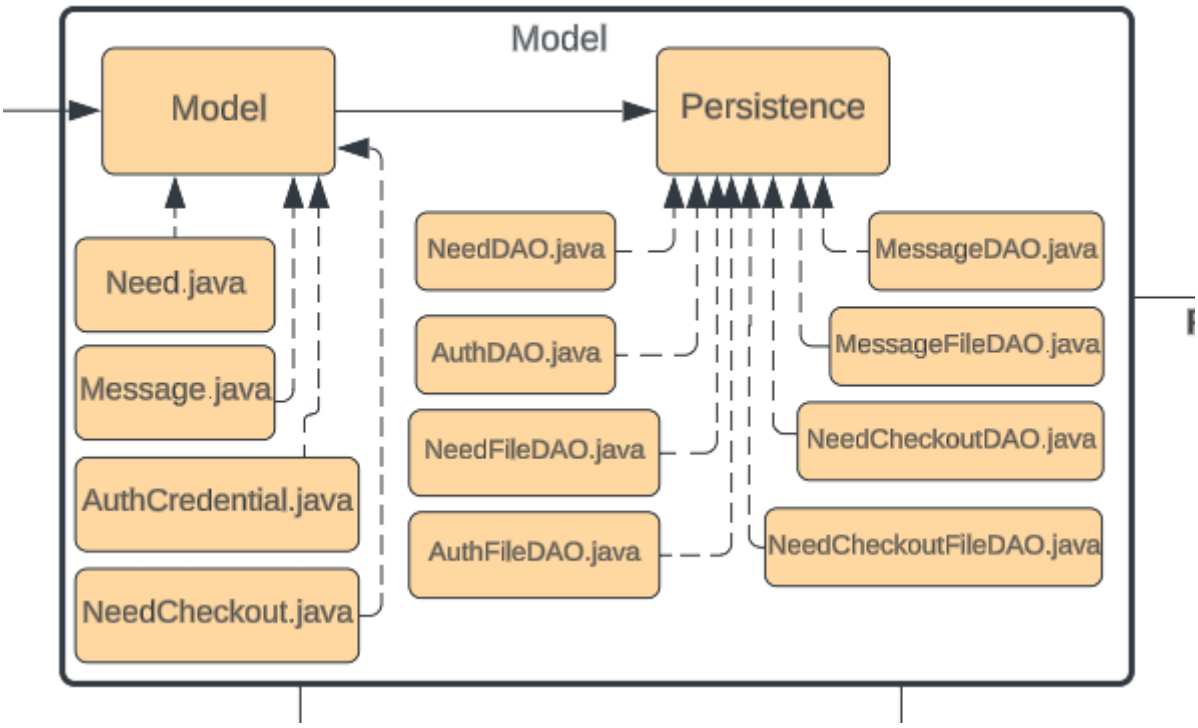
Model Tier

Need.java Represents a Need entity within the system. This class is responsible for encapsulating all the properties and behaviors of a need, including its identification, description, status (fulfilled or unfulfilled), and other attributes. The Need class plays a crucial role in the Needs Management feature, allowing U-fund Managers to add, remove, and edit the details of needs stored in their Needs Cupboard.

AuthCredentials.java Represents Authorization Credentials used in the system for authentication purposes. This class holds information necessary for authenticating users, such as usernames and passwords. It is integral to the system's security, ensuring that operations like login and logout are handled securely. The AuthCredentials class underpins the Minimal Authentication feature, enabling a basic but essential level of security for Helper and U-fund Manager interactions with the platform.

NeedCheckout.java Represents the Helper's Checkout Basket, a key component of the Helper functionality within the application. This class manages the collection of needs that Helpers wish to support, allowing them to add or remove needs from their basket. The NeedCheckout class is vital for facilitating the checkout process, where Helpers can review their selected needs before proceeding to fund them, aligning with the project's goal to connect non-profit organizations with those who can help fulfill their needs.

Message.java Represents a Message object, crucial for enabling communication within the system. This class encapsulates messages sent by users, containing attributes such as message ID, username (sender), timestamp, and the text of the message itself. The Message class is designed to support features such as messaging between Helpers and U-fund Managers or between users and the system for notifications, queries, and updates. By facilitating communication, the Message class enhances user engagement and the overall functionality of the U-fund application, making interactions more dynamic and responsive to user needs.



OO Design Principles

Controller (GRASP): A design pattern that assigns the responsibility of dealing with system events to a non-UI class that makes decisions and controls the flow of the application. It acts as an intermediary between the UI and the business logic, handling user input and invoking responses in the system.

Single Responsibility Principle (SOLID): A principle that states a class should have only one reason to change, meaning it should have only one job or responsibility. This promotes a cleaner, more modular design by separating concerns within a software system.

Information Expert (SOLID): In the U-Fund project, the Information Expert principle is applied by assigning tasks to classes that hold the relevant data. For instance, the NeedsCupboard manages needs, and a UserManager handles user authentication. To further enhance adherence to this principle we can centralize all data-related operations within the owning class. For example, a SearchService in the NeedsCupboard should manage search operations. Use a FundingBasket class to manage all actions on the funding basket, ensuring changes are made through its methods for better data encapsulation and responsibility localization.

Law of Demeter (SOLID): the Law of Demeter is applied to limit direct interactions between system components. For example, a Helper should use methods on the NeedsCupboard to interact with needs instead of directly accessing them. To further enhance adherence to this principle we can encourage using class methods for operations, like a Helper using FundingBasket methods to add needs, keeping the system's parts decoupled. Implement facade or service layers, such as a NeedManagementService, to mediate interactions between the UI and data models, reducing direct component coupling

Open/Closed Principle (SOLID): This principle is applied as a means so that the system is open for extension, but closed for modification. This approach is crucial to implement additional features such as our 10% additional feature enhancnements, so that it coexists with our existing functionalities seamlessly. An example of this would be our existing Needs functionality. This was implemented through our Java Spring backend, (which defines contracts for adding, removing, and editing needs in places throughout our

application) thereby allowing for new implementations to extend these behaviors without modifying existing logic.

Dependency Inversion/Injection (SOLID): Dependency inversion is key in minimizing coupling between components, especially between the backend server logic written in Java Spring and the frontend in Angular. By depending on abstractions rather than concrete classes, we can make our backend services easily interchangeable and testable, enhancing the project's maintainability. Dependency injection, facilitated by Spring's IoC (Inversion of Control) container, will be used to inject these dependencies at runtime, allowing for more flexible and decoupled code. For instance, a AuthService interface can be defined and implemented by different classes that manage authentication, and the specific implementation can be injected into controllers that require it, depending on the project's current configuration.

Low Coupling (GRASP): Low coupling is critical throughout our application to ensure modularity and ease of modification. This principle is essential in the design of our system's components, in the sense that they require the least possible knowledge of each other to function. For instance, in the implementation of the Helper functionality and the Needs Management, we designed our Angular services to interact with our API in a way that minimizes their dependencies of each other. Furthermore, this can be faciliated by creating focused services that perform specific tasks such as our AuthService, rather than a monolithic service that handles multiple responsibilites.

Pure Fabrication (GRASP): Pure fabrication is a principle that suggests creating classes do not necessarily represent a concept in the problem domain, especially to achieve low coupling and high cohesion between functionalities. In our application, pure fabrication is demonstrated through our utility classes / services that handle application-wide tasks such as our authentication services and data persistance classes, where instead of coupling authentication logic with user logic, our AuthService handles security concerns can be implemented, making the system more modular and easier to maintain.

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

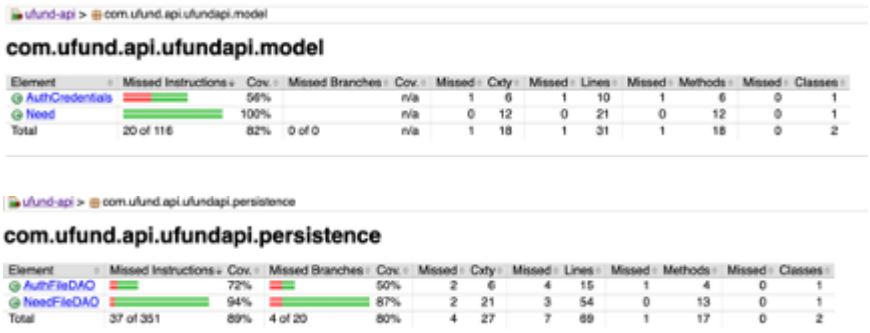
Acceptance Testing

All of our user stories have passed 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.*

These are some of our code coverage test results. As shown, our code is generally well-tested. These test results gave us insights into how to improve our test coverage.



ufund-api > com.ufund.api.ufundapi.model

com.ufund.api.ufundapi.model

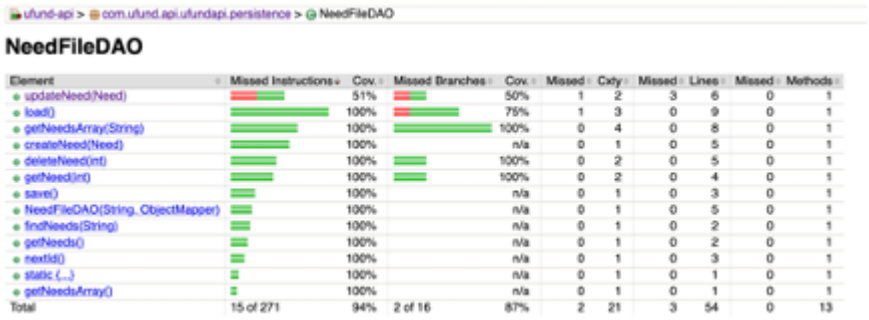
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AuthCredentials		56%		n/a	1	6	1	10	1	6	0	1
Need		100%		n/a	0	12	0	21	0	12	0	1
Total	20 of 116	82%	0 of 0	n/a	1	18	1	31	1	18	0	2

ufund-api > com.ufund.api.ufundapi.persistence

com.ufund.api.ufundapi.persistence

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AuthFileDAO		72%		50%	2	6	4	15	1	4	0	1
NeedFileDAO		94%		87%	2	21	3	54	0	13	0	1
Total	37 of 351	89%	4 of 20	80%	4	27	7	69	1	17	0	2

Both the AuthFileDAO and the NeedFileDAO have missed instructions and missed branches. In the AuthFileDAO, the getAuthCredentials method has two branches that have not been tested: if the authCredentials contains the given username key and if it does not. We must add two tests: one that covers if the authCredentials contains the key and one that covers the case where it does not.



ufund-api > com.ufund.api.ufundapi.persistence > NeedFileDAO

NeedFileDAO

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
updateNeed(Need)		51%		50%	1	2	3	6	0	1
load()		100%		75%	1	3	0	9	0	1
getNeedsArray(String)		100%		100%	0	4	0	8	0	1
createNeed(Need)		100%		n/a	0	1	0	5	0	1
deleteNeed(int)		100%		100%	0	2	0	5	0	1
getNeed(int)		100%		100%	0	2	0	4	0	1
save()		100%		n/a	0	1	0	3	0	1
NeedFileDAO(String, ObjectMapper)		100%		n/a	0	1	0	5	0	1
findNeeds(String)		100%		n/a	0	1	0	2	0	1
getNeeds()		100%		n/a	0	1	0	2	0	1
nextId()		100%		n/a	0	1	0	3	0	1
static {...}		100%		n/a	0	1	0	1	0	1
getNeedsArray()		100%		n/a	0	1	0	1	0	1
Total	15 of 271	94%	2 of 16	87%	2	21	3	54	0	13

The NeedFileDAO's code coverage needs to be improved by implementing a test for the successful updating of a Need. So far, it only covers handling the exception in which the user tries to update a Need that is not found. There must be a test covering the successful updating of a Need.

Ongoing Rationale

(2024/02/21): Sprint #1 -- Created the necessary utilities needed to perform cURL operations on the Need entity. Includes functionalities such as creating / deleting need, retrieving specific need, searching need by partial name, updating needs, and listing all needs.

(2024/03/22): Sprint #2 -- Implemented all functionalities for minimal authentication so users are limited to performing actions according to their privileges. Introduced testing, and code coverage testing to ensure that our code is robust and functional. Also implemented functionalities related to the 'Checkout' of our application, such that users can fund a specific need, and ensure consistency with the cupboard.

(2024/04/08): Sprint #3 -- Totally implemented all minimal viable product features as outlined in the vision document. The 10% additional features, messaging / suggestion service, and sorting features were completely implemented. Additional changes were made to our UI to represent the fully-fledged product, which includes images, color styling, and other visually appealing features.