PROJECT Design Documentation

Team Information

- Team name: Platy Possibilities
- Team members
  - Daniel Baek
  - Javier Amaro
  - Kinkade Knox-Silva
  - Kaila Grant
  - Leo Mai

Executive Summary

Platy Possibilities is a web application focused around funding wilderness preservation. On the website, users are able to search through and choose various needs to give funding to, from ecosystem monitoring to restoration efforts. These needs can be created, changed, and removed by the admins so that any changes in those projects can be quickly reflected on the website and new projects can be brought to the attention of the users. Users are also able to message the admin to give them suggestions to improve the application.

Purpose

The project focuses on developing an Angular web application that manages the needs of a wilderness preservation organization. The most important user group of the project are the environmentalists, conservationists, and volunteers dedicated to preserving wilderness areas, while the most important user goals are to monitor ecosystems, track endangered species, and promote conservation efforts to safeguard natural habitats for future generations.

Glossary and Acronyms

Term	Definition
SPA	Single Page
User	Anyone loged in and using the app.
Helper	The users who contribute to the needs.
U-Fund Manager/Admin	The user who creates, modifies, and deletes the needs.
Cupboard	The area of the app where helpers can search through and find needs they want to fund.
Need	Goals for the Helpers to fund.
Funding Basket	The menu containing the needs that a Helper has chosen to fund. Each helper has their own Funding Basket.

Requirements

This section describes the features of the application.

Definition of MVP

The Minimum Viable Product involves a basic login system, in which a user only needs to put in a username to log in. If the username inputted is "admin," the user will log in as the U-Fund Manager. Otherwise, they will be a helper. They can also log out of the application. A helper can search for a need in the cupboard, and is able to add and remove needs from their funding basket. Helpers can checkout their basket at any time, funding the needs and emptying the basket. A U-Fund Manager can add, remove, and edit needs in the cupboard, but they do not have any access to a funding basket; this is something exclusive to helpers.

MVP Features

- Login
    - Can login as the Admin, allowing for the managment of needs
    - Can login as a Helper, allowing for the funding of needs
- Helper & Admin
    - Can view the needs in the Cupboard
- Helper
    - As a Helper, can add needs to the funding basket
    - As a Helper, can remove needs from the funding basket
    - As a Helper, can checkout the funding basket
    - Can search the Cupboard for needs based on title
- Admin
    - As the Admin, can create a new need in the cupboard
    - As the Admin, can remove the needs from the cupboard
    - As the Admin, can edit the needs in the cupboard

Enhancements

Message Board

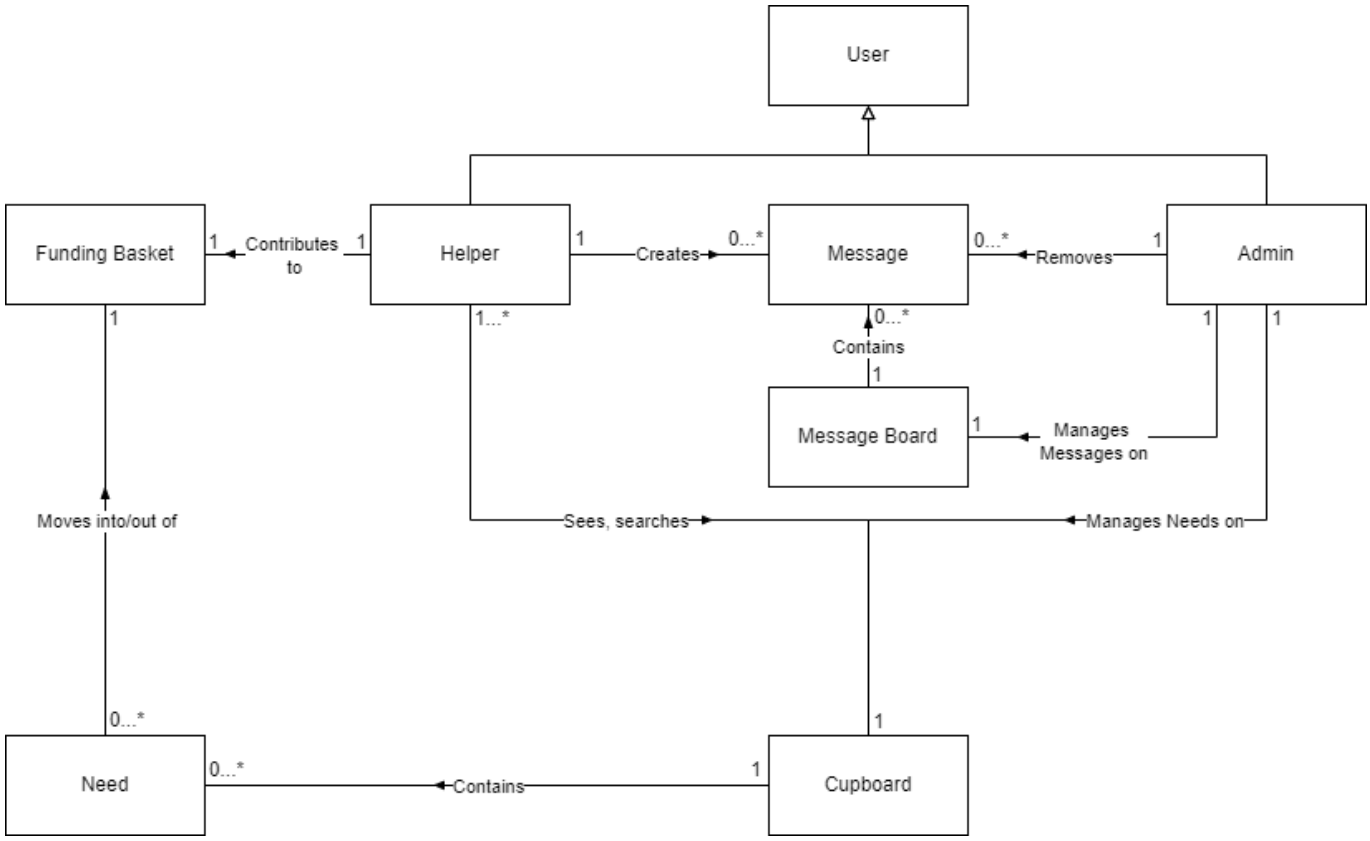
There is a message board feature that allows any helper to send a message that can be read only by the admin. The admin is able to see any messages sent and can clear them from the message board.

Multi Select

When selecting needs, whether it be the helper choosing needs to add or remove from their basket or the admin choosing needs to delete, there is a feature to allow the selecting of multiple needs so the selected needs can be added, removed, or deleted with one button press.

Application Domain

This section describes the application domain.



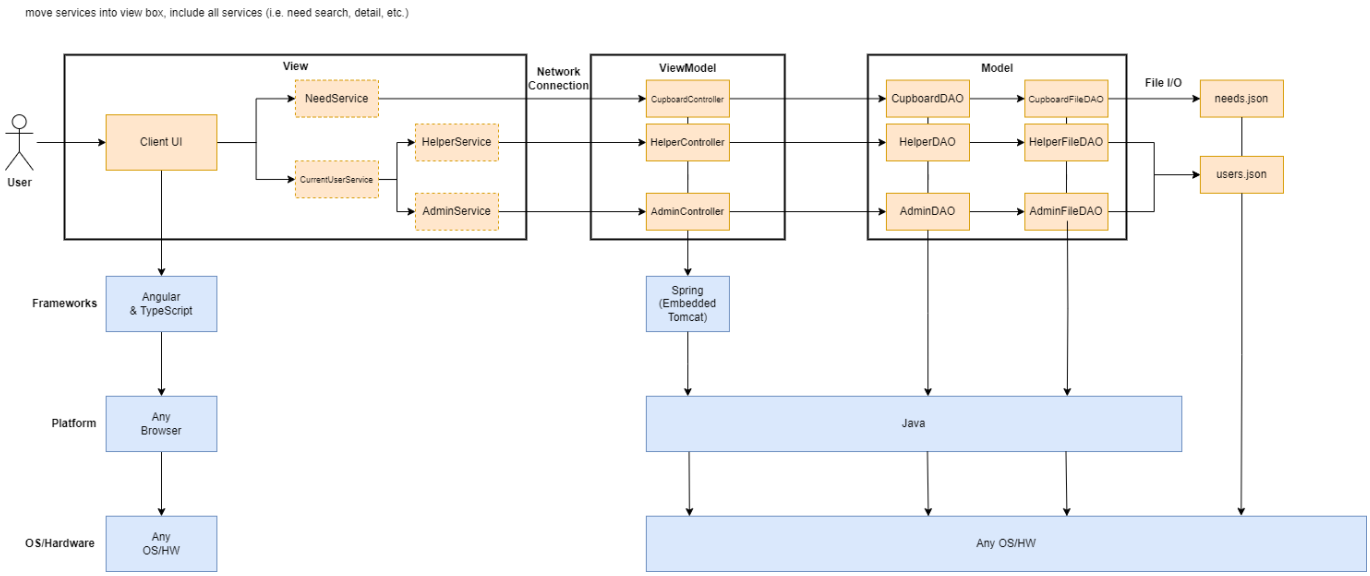
Everyone who signs into the application is a User. Most Users are Helpers who each have a Funding Basket where they can place the Needs that they want to fund and remove the ones that they do not. The Needs are stored in the Cupboard, where the Helpers can search for Needs they want and select Needs to move to their Basket. The other kind of User is the U-Fund Manager, or Admin, who can create, modify, and remove Needs from the Cupboard. The Admin does not have a Funding Basket.

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.

The user starts off on the login screen and, depending on which username they input, they will be routed to either the "helper" or "admin" page. Generally speaking, if the username is anything other than "admin," the user will be routed to the helper page; however, this will not work if the username is not already in the users.json, which only houses the usernames: johndoe, janedoe, and admin.

If they input either johndoe or janedoe, they will be taken to the helper page, which shows their username at the top of the page, a router link to the basket page, a "need search" portion so the user can look for a specific Need, a list of the Needs, and an "add message" portion so that the helper can send a message to the admin about a certain issue that needs to be addressed. The user can click on the Needs to see overall details such as id, description, cost, quantity, and quantity funded. They can also either add Needs one by one into their funding baskets or select multiple to be added via the checkboxes next to each. They then would click the "add to basket" button to add the multiple selected Needs. Lastly, there is a logout button to be brought back to the login screen.

The basket page will show the individual Needs in the user's funding basket. If there are none, there will be a message stating this. If there are, it will list the Needs similarly to the helper page, in which the helper can either remove or select Needs to be removed from the basket or checked out respectively.

The admin page will show the admin username up top, an input box to add a new Need to the cupboard, a list of all the Needs in the cupboard, and a message board of the messages the admin receives from the helpers. Here, the admin can choose to remove individual Needs via the remove button next to the Needs, or they can remove multiple via the checkboxes next to each as well as the "remove from cupboard" button. This works similarly with the message board, as the admin can remove messages that they've read. Lastly, a logout button is also available for the admin to be brought back to the login page.

View Tier

The View Tier consists of admin, basket, helper, login, message board, need, need detail, and need search components.

The admin component focuses solely on the admin page and houses their responsibilities, which is mainly getting the message board and deleting messages from it. They also have full access to the Need cupboard, but the admin takes from the Need component for the functionalities.

The basket component focuses on the basket page of the UI, in which it manages the Needs in a helper's basket. On the page, Needs can either be removed from the basket or checked out, meaning they will be fully

funded. Multiple Needs can also be removed and checked out with the "select all" button present on the page.

The helper component houses responsibilities soley arranged for the helper role, whether it be adding Needs to the basket or sending a message to the message board. Like the basket component, the helper can select multiple Needs and add them to their respective basket.

The login component only serves as a means of routing the user to the correct pages, so if a helper username that is in the database is given, they will be routed to that helper specific page. If "admin" is inputted, they will be routed to the admin page. If the user is not found, a message will be displayed saying this on the login page.

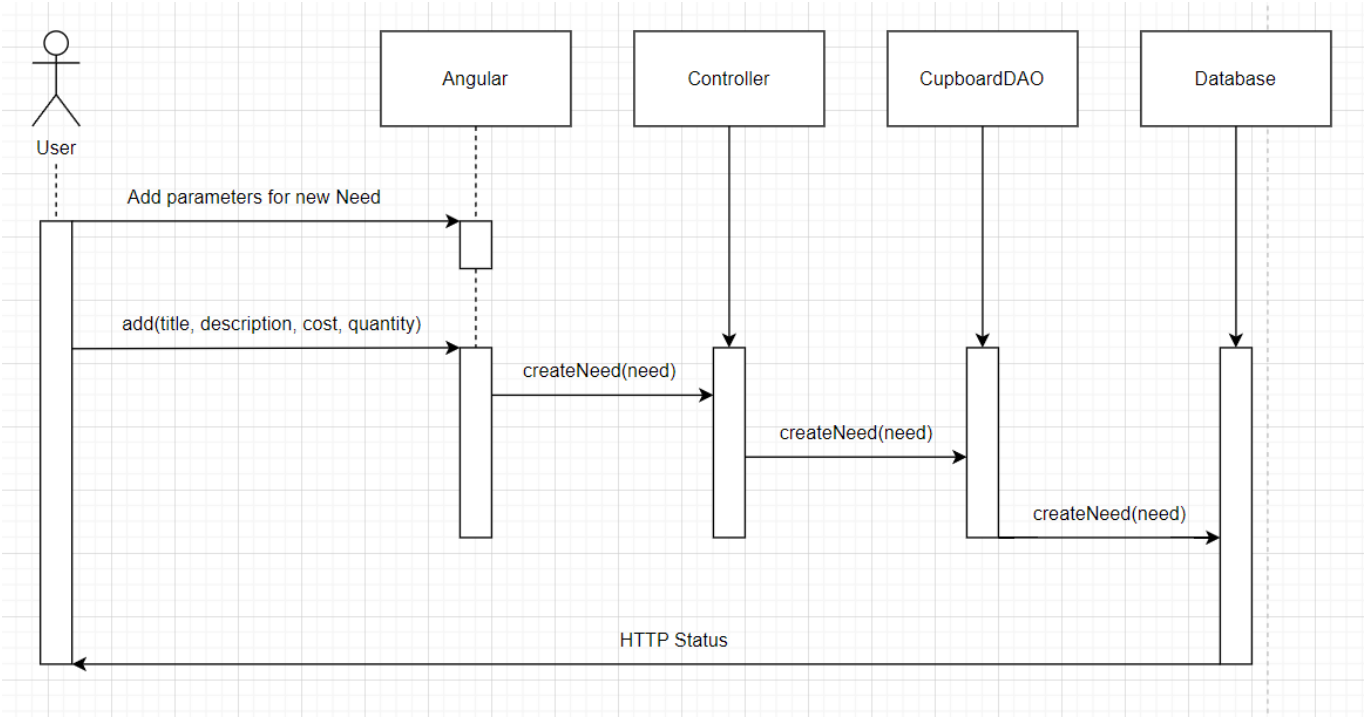
The message board component correlates with the admin's message board, in which helpers can send messages/suggestions for new Needs the admin can add to the cupboard. While the methods to add and remove messages are in the message board component, the helpers and admin respectively have the responsibility of them.

The Need component is where the main Need functionality lies, like adding a new Need and deleting a Need. The admin, through this component, also has access to the multiple selection of Needs, as this component is the one that implements it.

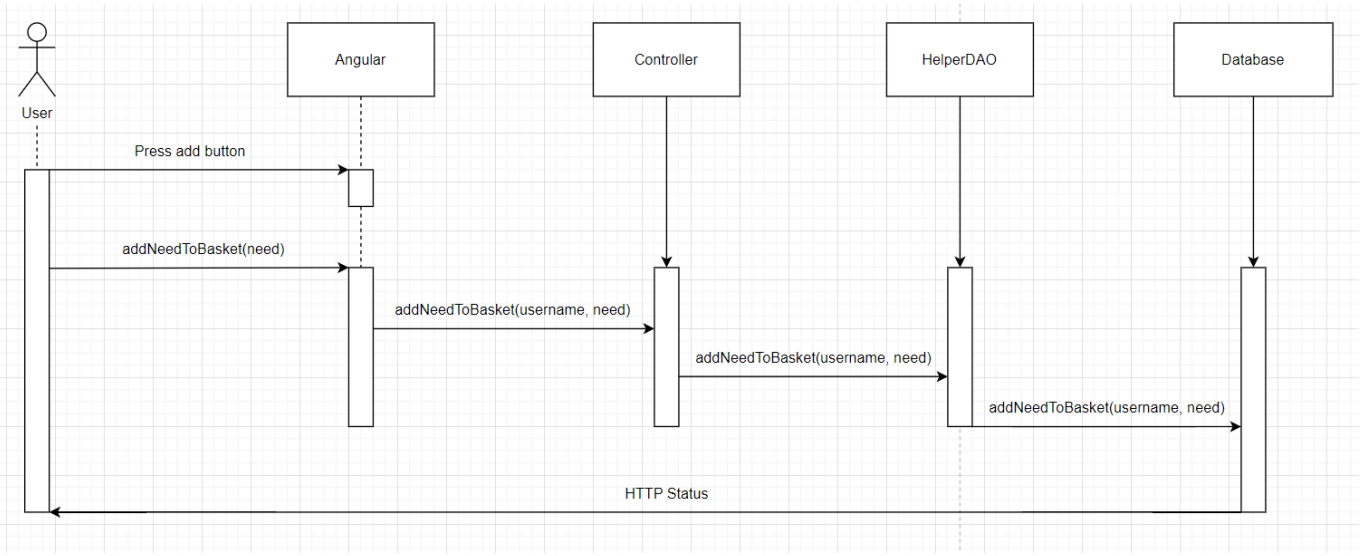
The Need detail component essentially acts as a view of a specific method, displaying the ID, title, description, cost, quantity, and quantity funded of the Need. The only notable method in this component is the save method, as this is where you can change and update the information of a specific Need, like changing the title, description, etc.

The Need search component is a simple search engine in which the helper, who has access to this component, can input specific characters to find a specific Need. For example, if there is a Need titled "Fox Release Training" and the character "F" is input into the search bar, the Need will be displayed.

Sequence Diagram: Creating a new need

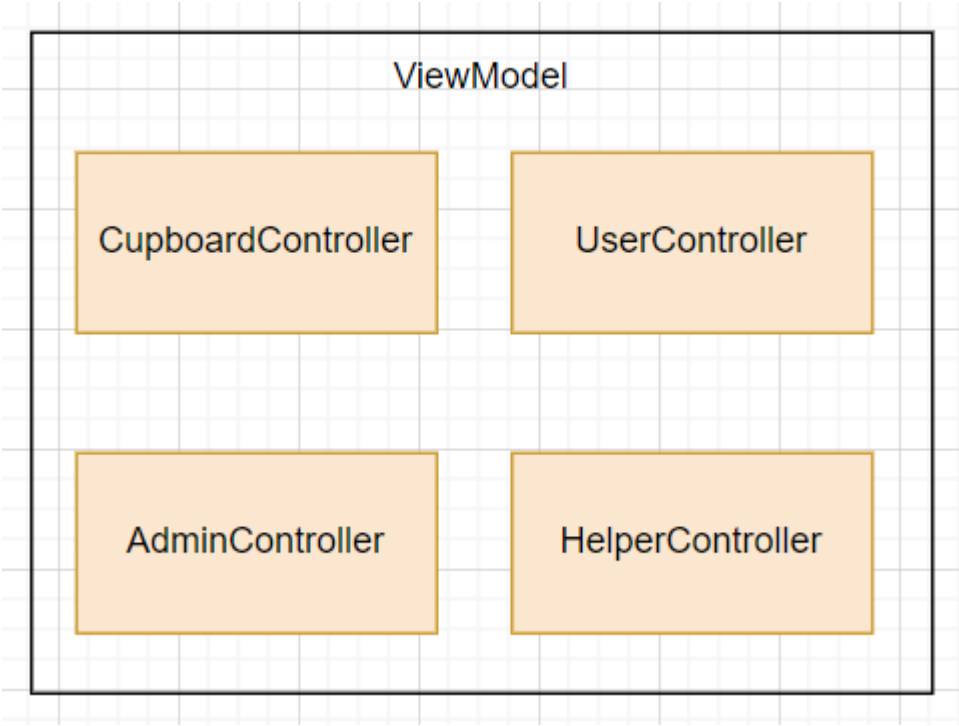


Sequence Diagram: Adding a need to the basket



ViewModel Tier

The classes supporting the ViewModel tier of the model would be the CupboardController public class, which houses the functions pertaining the needs, whether it be creating a need, deleting a need, get a need or needs, etc. The AdminController only has methods for getting the messages on the message board and removing messages from it. The HelperController is where the basket methods and adding message method is located. The UserController is where the searchUsers method lies, in which it searches for all the users that have the inputted username.



Model Tier

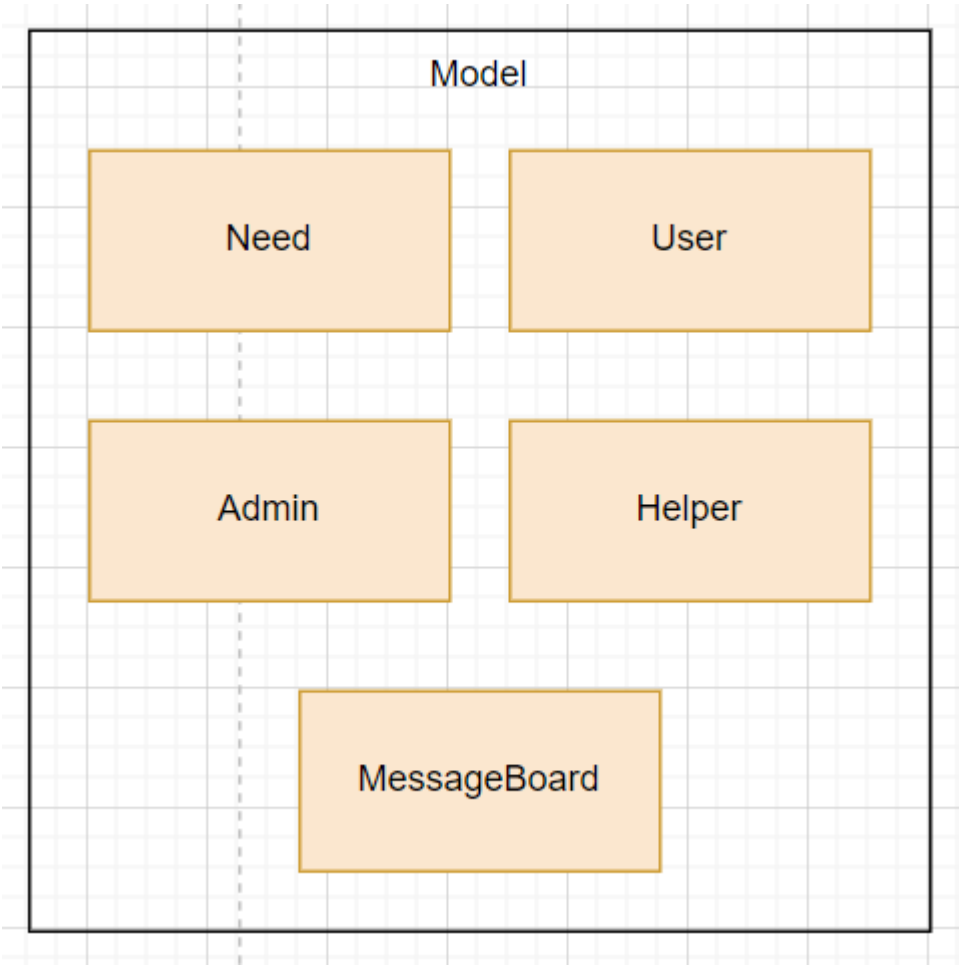
The class supporting the Model tier would be the Need, Message Board, and User public classes. The Need class houses the ID, title, description, cost, quantity, and quantity-funded of the specific need, while defining the functions connected to that class. The User class houses the username and type, which splits into either a

Helper or an Admin. The Message Board class is interesting. While this has the functionality for adding and removing messages on the board, it uses a private adminDAO in these methods, as the admin is ultimately who has the message board. This was made so that the helper can access it.

The Need class details every responsibility directly correlated with a Need, whether it be getting the properties (id, title, description, cost, quantity, quantity_funded) from the object or changing the title of a specific Need to another one by updating it.

The User class acts as a generalized backbone for users, housing a method that gets the username of the User. This class can then be split into either a Helper or an Admin depending on the given username; if the given username is "admin," they will be classified as such. Otherwise, they will be a Helper.

The Message Board class is where the messages get managed, mainly by the Admin. Generally, this class will be used by the Admin, such as seeing and deleting messages from the board; however, they cannot add messages to the board. Instead, the Helpers are the ones who send messages to them. This is because it acts as a sort of suggestion post, so that the Admin can create new Needs that fit these suggestions.



OO Design Principles

The initial OO Principles that the team has considered for this first Sprint are the Single Responsibility and Controller principles. For the Single Responsibility, we have each class assigned to a specific portion of the project, whether it be having a class for the Cupboard or for the Need itself. For the Controller principle, we are implementing a CupboardController that allows for the manipulation of the needs, whether it be deleting a need, getting a need, or updating a need.

Encapsulation, Inheritance, Abstraction, Polymorphism

Encapsulation: The MessageBoard class encapsulates the overall board and exposes the methods so that the Helper and Admin classes can use its methods.

Inheritance: The two different users, the helpers and admin, inherit from a more generalized user class.

Abstraction: The overall User class is abstract, as it serves as a base class for the Helper and Admin classes.

Polymorphism: The MessageBoard class, in a sense, showcases polymorphism, as it treats the Helper and Admin classes as User objects.

Static Code Analysis/Future Design Improvements

Static Analysis

```
@Injectable({
  providedIn: 'root'
})
export class HelperService {
  private helperURL = 'http://localhost:8080/Helper';

  httpOptions = {
    headers: new HttpHeaders({'Content-Type': 'application/json'})
  };

  constructor(private http: HttpClient) {

  }
```

Member 'helperURL' is never reassigned; mark it as `readonly`.

Member 'http: HttpClient' is never reassigned; mark it as `readonly`.

Often flagged pieces of code were variables that never changed in runtime, but were mutable. The solution provided was to make all these variables readonly, or essentially a constant. My recommendation follows this solution. While no issues arised related to these variables in our time working on this project, to prevent future errors it is a great idea to make these variables immutable.

```
import { Component, OnInit } from '@angular/core';
import { Need } from '../need';
import { FormsModule } from '@angular/forms';
import { NgIf, UpperCasePipe } from '@angular/common';

import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

import { NeedService } from '../need.service';
```

'C:\Users\Phoen\Courses\SWEN-261\team-project-2241-swen-261-07-e-platypossibilities\ufund-ui\angular-u-fund\node_modules\@angular\common\fesm2022\common.mjs' imported multiple times.

'C:\Users\Phoen\Courses\SWEN-261\team-project-2241-swen-261-07-e-platypossibilities\ufund-ui\angular-u-fund\node_modules\@angular\common\fesm2022\common.mjs' imported multiple times.

This example is a small but crucial practice which is important for the readability and maintainability of the codebase. The general idea is to keep imports clean and concise by not having duplicate imports and unused imports in files. My recommendation is to follow this idea and keep all imports up-to-date and trimmed as the work on the project continues.

```
     * @param objectmapper - The object mapper.
     */
    public AdminFileDAO(@Value("ufund-api/data/users.json") String filename, ObjectMapper objectmapper) throws IOException {
        super(filename, objectmapper);
        List<Admin> admins = users.values().stream()
                .filter(user -> user instanceof Admin)
```

Replace this lambda with method reference 'Admin.class::isInstance'.

```
                .map(user -> (Admin) user)
```

This issue is simple and related to the efficiency and readability of the codebase. The general idea is to simplfy code as much as possible to make it easier to read, debug and expand upon. My recommendation here is to use method references where possible, and simplfy overall logic (i.e returning a condition instead of using if -> return true/false)

```
        Need[] needList = objectMapper.readValue(new File(filename),Need[].class);
        for (Need currNeed : needList){ // for each need, load hashmap of needs with ID and need structure.
            needs.put(currNeed.getId(), currNeed);

            if(currNeed.getId() > nextID){
                nextID = currNeed.getId();
```

Make the enclosing method "static" or remove this set.

```
            }
        }
        ++nextID;
```

Make the enclosing method "static" or remove this set.

This issue is irrelevant for our project, as we only have one user logged in at a time, and thus don't have to worry about syncronization problems. With that being said, if this project were to be expanded this fault would be extremely critical to fix and prevent in the future. The general theme is thread safety and proper syncronization. Here we have a shared resource used to dynamically set the id for the next created need. To prevent issues with two users attempting to access this resource at the same time, I would recommend making the methods using this resource to be syncronized with a key to ensure only one user is changing or accessing the resource at a time.

Future Design Improvements

Company Managers: Wildlife conservation organizations are able to make their own manager accounts and house many needs in their own needs list.

Search specific organizations: Helpers could filter by or get the needs list of an individual organization.

Need detail expansion: Needs have images, contact info, and generally more information about how the money will be used to help wildlife conservation.

General Donation: For helpers unsure of what needs to fund, a general donation will be routed to various open needs. General donation could go specifically to an organization.

Testing

Acceptance Testing

Sprint 4

Out of a total of 19 user stories, all 19 have passed their acceptance criteria tests. There were initially a few bugs, such as a bug preventing the admin from changing certain fields of a need, but they have all been fixed.

Unit Testing and Code Coverage

Sprint 2

ufund-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi.persistence		89%		71%	9	36	10	98	2	20	0	2
com.ufund.api.ufundapi.model		82%		37%	7	28	8	49	3	24	1	4
com.ufund.api.ufundapi.controller		68%		75%	4	26	30	90	1	16	0	2

One anomaly for this code coverage is that the CupboardController tests do not test for when the DAO may throw an IOException. This makes it appear that it is missing more coverage than it actually is.

Sprint 3

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi.model		82%		44%	12	41	11	69	3	32	0	5
com.ufund.api.ufundapi.controller		76%		83%	5	39	34	136	1	24	0	4
com.ufund.api.ufundapi.persistence		71%		56%	21	55	43	158	7	31	1	4

Many tests do not test for the situation where the DAO would throw an IOException, but considering they should behave in the same way in that scenario, it should be fine.

Sprint 4

ufund-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi.model		99%		81%	3	40	1	68	0	32	0	5
com.ufund.api.ufundapi.controller		97%		93%	2	37	4	134	0	22	0	4
com.ufund.api.ufundapi.persistence		96%		84%	7	52	5	144	0	29	0	4
com.ufund.api.ufundapi		0%		n/a	4	4	7	7	4	4	2	2
Total	81 of 1,472	94%	12 of 92	86%	16	133	17	353	4	87	2	15

There are no real anomolies with these tests, most situations are covered.

The strategy for unit testing was to look at each significant function and determine the different paths that the code could take, creating a test for each different output. By the end of the project we wanted each element to have around 80%-90% code coverage. That would mean that we would be covering the important functions while not necessitating unecessary tests such as tests for toString functions. In this regard we met our goal quite well.

Ongoing Rationale

2024/09/29 Sprint 1: The team worked on and implemented the functionality of the CupboardController, while also implementing the routine formatting for the code.

2024/10/09 Sprint 2: The team decided that, for the helpers and admin, there would be a more generalized user class that the other two classes inherit from. We did this because both the helpers and admin both shared many aspects and it would allow us to store them in a singular file for authentication.

2024/11/06 Sprint 3: The team made a new class in the model, MessageBoard, so that the helper can access message board and add new messages.

2024/11/17 Sprint 4: The team decided that checkout wouldn't remove a need from the cupboard, but instead fully fund it by updating the quantity funded field. We did this because we felt that it would be confusing from a user standpoint if the checked out needs simply disappeared from the cupboard as you wouldn't know which need disappeared.