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.*

Team Information

Team name: Teamftw, team07

- Team members
- Gonzalo Estrella
- Abel Girma
- Ansley Orell
- Aniruddha Roy

Executive Summary

This is a summary of the project. FundMorocco is a relief fund website focusing on aiding Morocco after a recent earthquake. It not only serves as a donation platform but also allows users to create and log into accounts to view their donation cart and make purchases. Additionally, it features administrative capabilities for managing inventory and posting on a Announcement board, making it a comprehensive tool for both relief efforts and product management.

Purpose

Our purpose is to enable users (helpers) to donate goods to Morocco. They can do this by purchasing goods from our site, which will then be directly shipped to Morocco.

Glossary and Acronyms

Term	Definition
SPA	Single Page
Helper	A person who logs in with the intention to donating
Admin	A person who logs in with the admin username and upkeeps the website
Donation cart	A collection of goods chosen by a user for them to pay at the checkout
Need	The goods which are sold in the store
10% feature	Single Page
DAO	Data Access Object, within the persistence tier
HTTP	HyperText Transfer Protocol, a network protocol for specifications on how data should be transferred

Term	Definition
CSS	Cascading Style Spreadsheets that describe how HTML elements are to be displayed on screen
UI	View section of the project to be shown to the user
API	Application Programming Interface, connection interface between computers or computer programs
MVP	Minimum Viable Product

Requirements

FundMorocco allows user creation through signup and will display an error message if the provided username is already in use. Users can log in using a username, with "admin" reserved for the site owner. This controls access to two main pages: the helper browse page and the admin browse page.

User Features: Users can view a list of goods in the inventory. This inventory can be searched by the name of the good, updating the display to showing only goods that contain the user-defined search term. Users can click on any product to access its detail page. Users can view, add to, and remove items from their donation cart. The donation cart allows users to see their selected goods and proceed to checkout to finalize their donation. Users can navigate between the Login, Browse, and Donation Cart pages at any time.

Admin Features: The admin can view the inventory, search for specific goods, and add new goods and delete listings on their browse page. Clicking on a specific product listing redirects the admin to the good's detail page, where they can update its cost and quantity.

Definition of MVP

The minimum viable product includes:

- Login and logout functionality for Administrators and Customers.
- Functionalities such as searching for goods, adding goods to the donation cart, and checking out.
- Inventory management, which allows the site owner to add, remove, and update goods in the inventory.
- Data persistence, ensuring that the inventory, users, and user donation carts are saved.

MVP Features

Top-Level Epics:

- Accounts
  - User should be able to create an account.
  - User cannout create an account with a pre-existing username.
- Login
  - Users and Admins should be able to login into their own accounts.
- Logout
  - Users and Admins should be able to logout of their own accounts.
  - Users and Admins cannot logout if they are not logged in.
- Donation Cart

  - Users should be able to add products to a cart.
  - Users should be able to remove products from a cart.
  - Users should be able to checkout a cart.
  - Users should be able to see the total price of their cart.
- Products
  - Users should be able to see a list of products.
  - Users should be able to search for a product.
  - Users should be able to view a product's details.
  - Admins should be able to add a product.
  - Admins should be able to remove a product.
  - Admins should be able to edit a product

Enhancements

In our MVP, we have introduced a feature that allows the admin to post Announcement on a Announcement board. These Announcements are displayed on the Donation Cupboard page as soon as a user logs in.

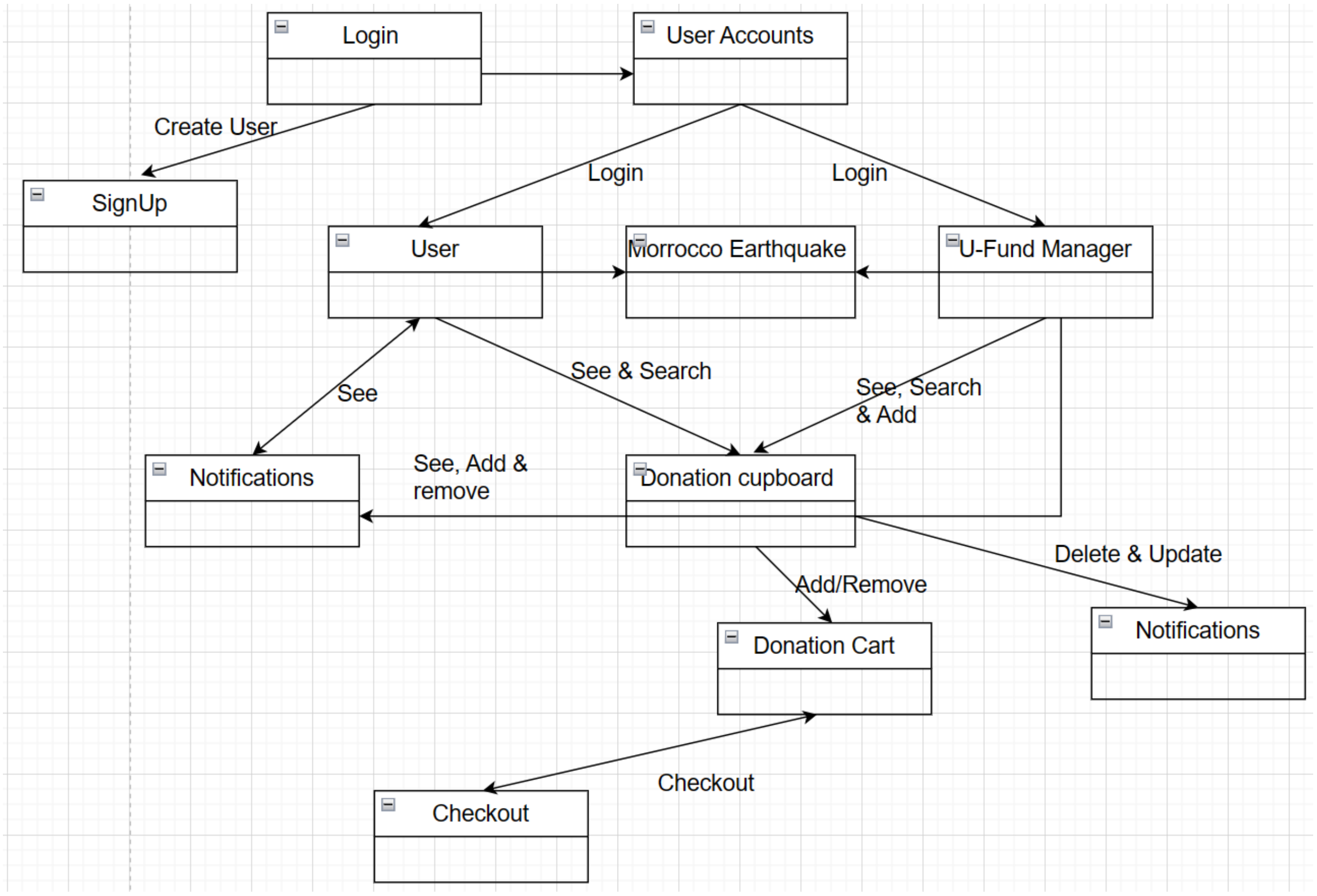
To implement this enhancement, an 'Announcement Management' button has been added to the admin page, enabling the admin to effortlessly post and delete Announcements.

Application Domain

This section describes the application domain.

FundMorocco's primary domain entity revolves around its users, which are categorized into regular users and an admin, termed as the "U-Fund Manager". Regular users access the platform to view the list of goods available for donation to aid the Morocco Earthquake relief efforts. They can add these goods to their "Donation Cart" and eventually finalize their donation through a checkout process.

The U-Fund Manager, on the other hand, oversees the "Donation Cupboard" which houses all items that can be donated. The manager can add, remove, or modify the list of goods in the inventory, ensuring the platform stays updated with available relief items. Communication is integral, with a Announcement Management button the admin can post Announcements to facilitate updates between users and the admin.



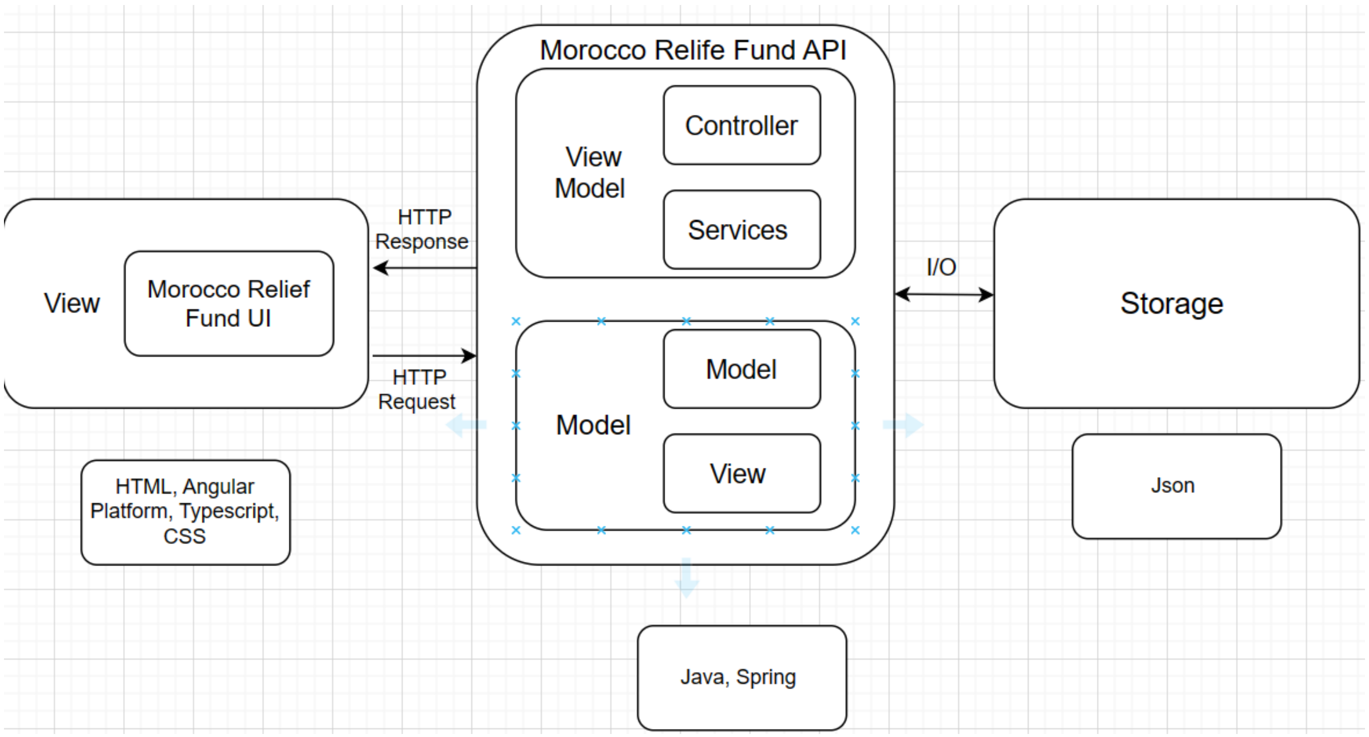
Architecture and Design

This section describes the application architecture.

Summary

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

Our application is comprised of three primary components: the view, the uFund API, and the storage. The uFund UI, part of the view component, is developed with HTML, CSS, and TypeScript using the Angular framework. The view component is responsible for sending HTTP requests to the ufund API, which then returns HTTP responses. Within the ufund API component, we have the view model and the model. The controller and services, both written in Java, are located in the view model. The controller additionally utilizes the Spring framework. The model comprises our persistence layer and our application model, also written in Java. The ufund-api interfaces with the storage through I/O operations. The storage itself consists of JSON files.



In Storage:

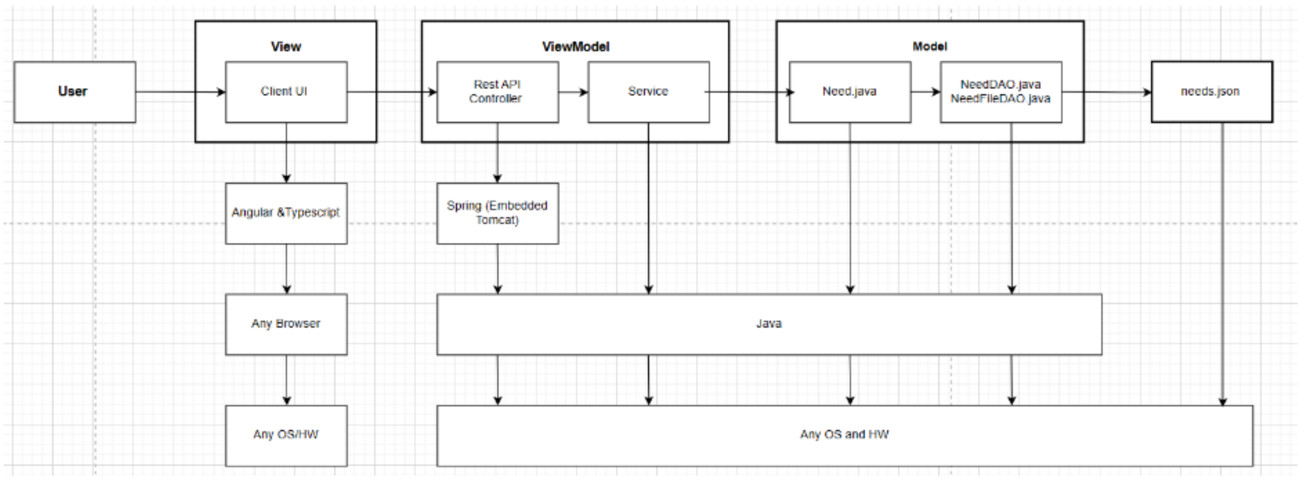
- In the needs.json file, every need we create, update, and delete will be saved.

In Model:

- Need.java defines the structure of data, encapsulates logic related to that data, and includes functionality for data serialization and string formatting. The model layer represents the core of the application's data and logic.

- NeedDAO interface defines the contract and methods for interacting with the model layer, specifically Need objects, The implementation of these methods interacts with the data stage. The model and persistence layers work together to manage and manipulate data entities like 'Need'.

- NeedFileDAO in the persistence layer is responsible for managing "Need" objects using JSON file storage. It provides methods for data retrieval, storage, and modification.



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.

When a user first opens the application, they are presented with the login page. Here, they have the choice to either log in or create an account. If they opt for the sign-up option, they are redirected to a page that prompts them for a username. They can either proceed to create their account or cancel. Choosing "Cancel" redirects the user back to the login page. If they choose "Create Account", the system will establish a new account with the specified username, log them in, and navigate them to the browse page. However, if an account with that username already exists, the user will be notified of the duplication.

When a user inputs an existing username (other than "admin") on the login page and selects the login option, they are granted access to the application and taken to the Donation Cupboard page. Here, they view the inventory of goods, displayed below a menu bar with options like "Login", "Browse", "Donation Cart", and Announcement Tab . There's also a search bar to find goods by name. Selecting a specific item redirects the user to a detailed page about that good, where they have the option to add it to their cart. If they select the "Donation Cart" from the menu, they can view the items currently in their cart.

However, if the user enters the username "admin" on the login page, they are directed to the admin-browse page. Here, they can view, add, or remove items from the inventory and post or remove Announcements. Selecting a specific item leads them to the admin-good-detail page, allowing them to modify the item's information and availability.

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 donation 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

> ***[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.*

Replace with your ViewModel Tier class diagram 1, etc.

Model Tier

> ***[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.*

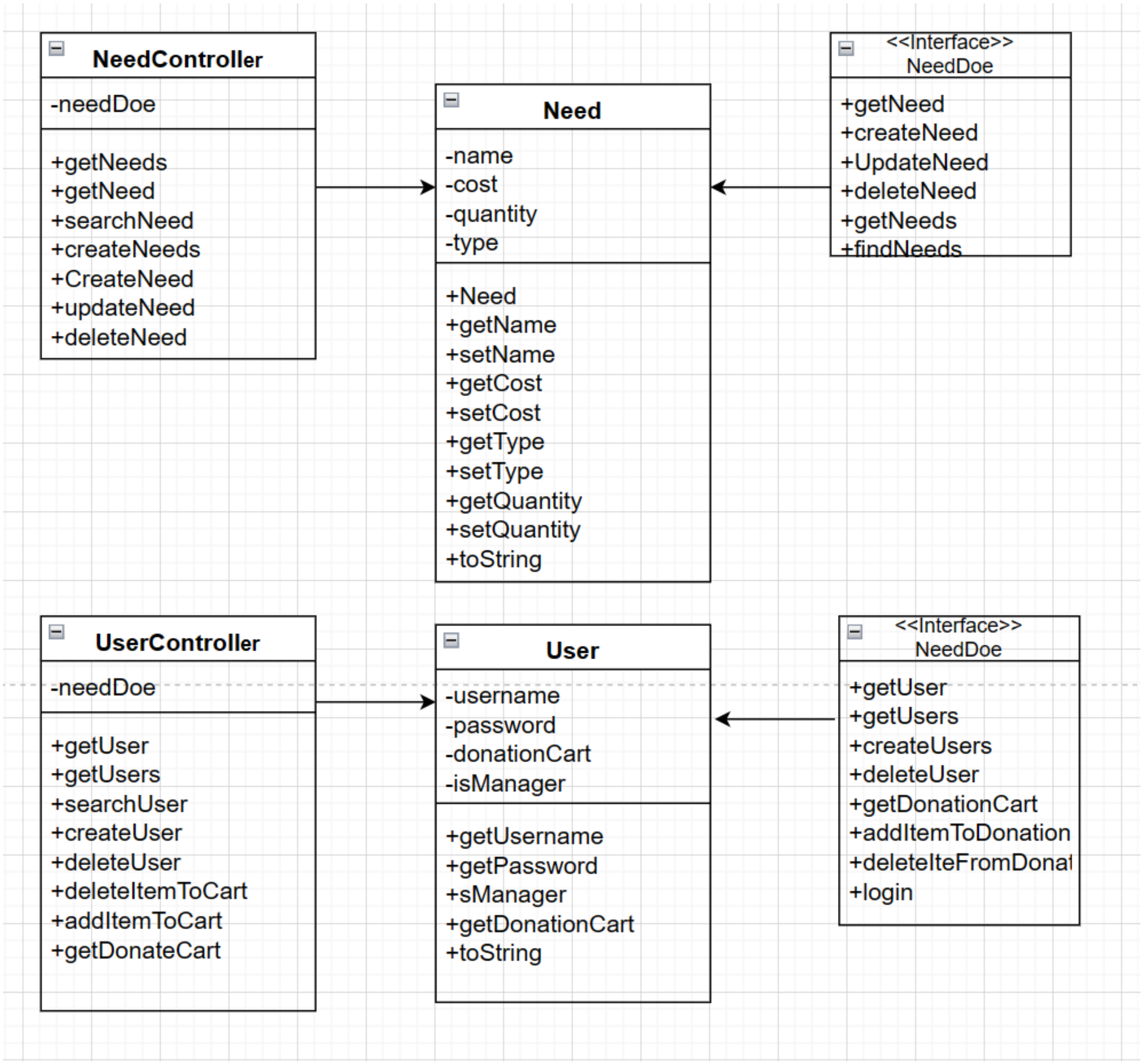
In Model Tier, we have 4 models.

Announcement Class: This class is designed to represent announcements, with each instance holding an id and detail. The id is automatically generated and incremented for each new instance. It includes standard getters, a setter for detail, and a custom toString method for string representation.

DonationCart Class: Represents a donation cart-like structure for managing donations. It uses an ArrayList to store Need objects. The class includes methods to add, increment, decrement, and remove items, as well as a method to retrieve all items in the cart as an array.

Need Class: Defines the details of a specific need, such as the item's name, cost, current quantity, required quantity, and type. It includes getters and setters for each field and a custom toString method.

Users Class: Represents user accounts, with fields for username, password, and a flag indicating whether the user is a manager. The class includes getters for these fields and a custom toString method. The donationCart field and associated methods are commented out, suggesting a possible design change or feature deprecation.



> *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.*

Replace with your Model Tier class diagram 1, etc.

OO Design Principles

> ***[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.*

Single responsibility. principle. A class should have only one reason to change, meaning it should have only one job or responsibility. In our code, each controller class adheres to this principle by focusing on a specific area of functionality. For example, AnnouncementController manages announcements, DonationCartController handles operations related to the donation cart, etc. This separation of concerns ensures that each class has only one reason to change.

AnnouncementController Responsibility: Manages all operations related to announcements. Examples in Code: The getAnnouncements() method retrieves all announcements. The createAnnouncement(@RequestBody String announcement) method creates a new announcement. These methods show that AnnouncementController is solely responsible for handling HTTP requests that are related to announcements. It's not concerned with other entities like users, needs, or donation carts. This singular focus on announcements is a clear indication of SRP compliance.

DonationCartController Responsibility: Handles the functionality related to donation carts. Examples in Code: The getDonationCart(@PathVariable("username") String usernameString) method fetches a user's donation cart. The addItemToCart(...) and deleteItemToCart(...) methods manage adding and removing items from the donation cart. The checkout(@PathVariable("username") String usernameString) method processes the cart's checkout.

Open/closed principle Software entities (like classes, modules, functions, etc.) should be open for extension but closed for modification. This means that the behavior of a module can be extended without modifying its source code. Let's analyze how our code complies with the OCP: Example in Code DAO Interfaces and Their Implementations: Interfaces Defined: UserDAO, NeedDAO, and CartDAO are interfaces defining the contract for operations like getUser, createUser, getNeeds, addItemToDonationCart, etc. Implementations: UserFileDAO, NeedFileDAO, and CartFileDAO are concrete implementations of these interfaces. OCP Compliance: The use of interfaces allows for extending the behavior of these DAOs without modifying the interface or other parts of the application that use these DAOs. For example, if we decide to switch from a file-based persistence mechanism to a database, we can create new classes like UserDatabaseDAO, NeedDatabaseDAO, and CartDatabaseDAO that implement the respective interfaces. The rest of our application won't need to change because it depends on the interfaces, not the concrete implementations.

Dependency Injection in DAO Constructors The Dependency Inversion Principle (DIP), one of the SOLID design principles, states that high-level modules should not depend on low-level modules, but both should depend on abstractions. Additionally, abstractions should not depend upon details, but details should depend upon abstractions. Example: In our NeedFileDAO and UserFileDAO, we inject dependencies like ObjectMapper and file paths (@Value("${needs.file}") String filename) through their constructors. DIP Compliance: These DAOs depend on abstractions (like ObjectMapper) rather than concrete details of JSON processing or file handling. This means we can easily switch out or modify these dependencies without needing to alter the DAOs' internals. For example, if we decide to change the way we handle JSON processing, we can inject a different implementation of a JSON processor without changing the DAO code.

Configuration through External Properties: Example: Using @Value annotations to inject properties like file paths (@Value("${needs.file}") String filename). DIP Compliance: This is another form of dependency injection where our classes depend on values that are externalized, not hardcoded within the classes themselves. It's an adherence to DIP because our DAOs are not directly dependent on the details of these configurations; they rely on abstractions provided by the framework (Spring in this case) to receive these values.

> ***[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

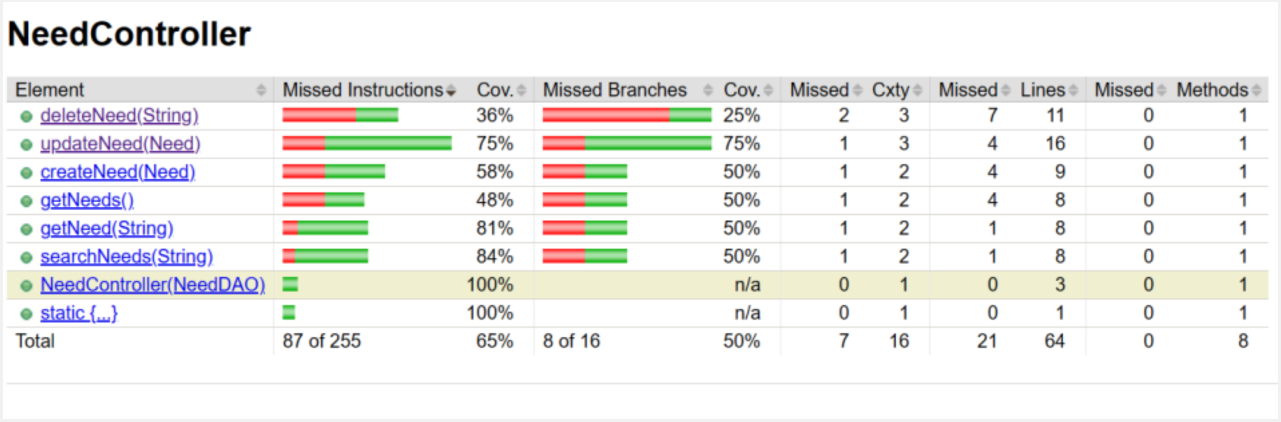
> acceptance testing and if there are any concerns.

15/15 user stories passed all their acceptance criteria tests. 25/31 acceptance criteria tests have passed - at a high code coverage - at least 90% overall.

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.*

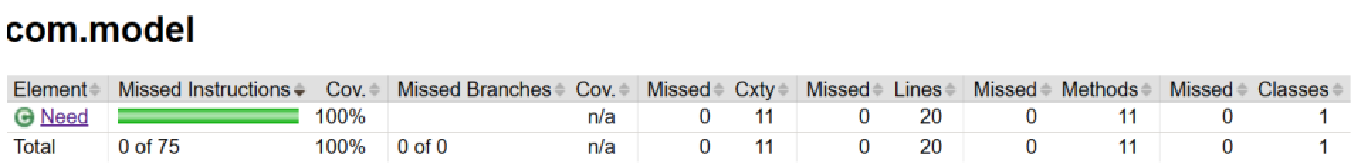
> ***[Sprint 2 & 4] Include images of your code coverage report.** If there are any anomalies, discuss those.*



NeedController

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
deleteNeed(String)		36%		25%	2	3	7	11	0	1
updateNeed(Need)		75%		75%	1	3	4	16	0	1
createNeed(Need)		58%		50%	1	2	4	9	0	1
getNeeds()		48%		50%	1	2	4	8	0	1
getNeed(String)		81%		50%	1	2	1	8	0	1
searchNeeds(String)		84%		50%	1	2	1	8	0	1
NeedController(NeedDAO)		100%		n/a	0	1	0	3	0	1
static {...}		100%		n/a	0	1	0	1	0	1
Total	87 of 255	65%	8 of 16	50%	7	16	21	64	0	8

> We can see from the unit test above we have not tested all limits of the code, because of this the team will continue working on the tests to be able to check and test all aspects of the different function such as the update need which is missing 25%. In the Need Controller class, we must implement tests that check exceptions.



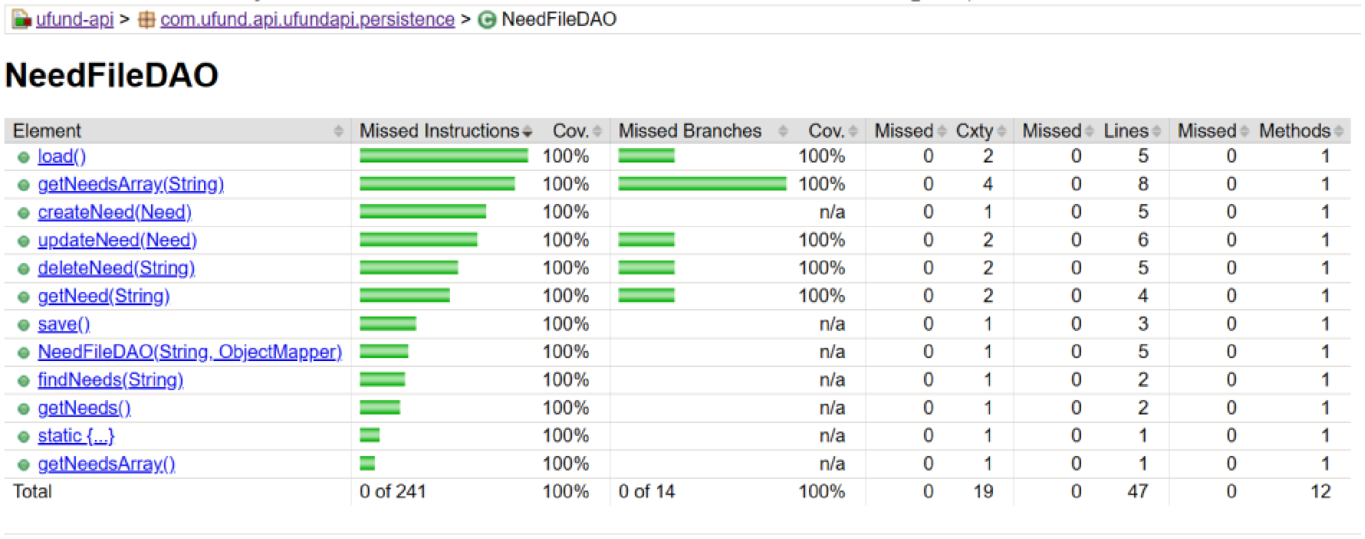
com.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Need		100%		n/a	0	11	0	20	0	11	0	1
Total	0 of 75	100%	0 of 0	n/a	0	11	0	20	0	11	0	1

The unit tests for this component are good because the unit tests we implemented for this class cover a wide variety of tests. The unit tests created for this class evaluate if the Need class correctly formats into a string and if the setters and getters for each attribute works.

Persistence Tier

This is our analysis at the Persistence Tier code for the project.

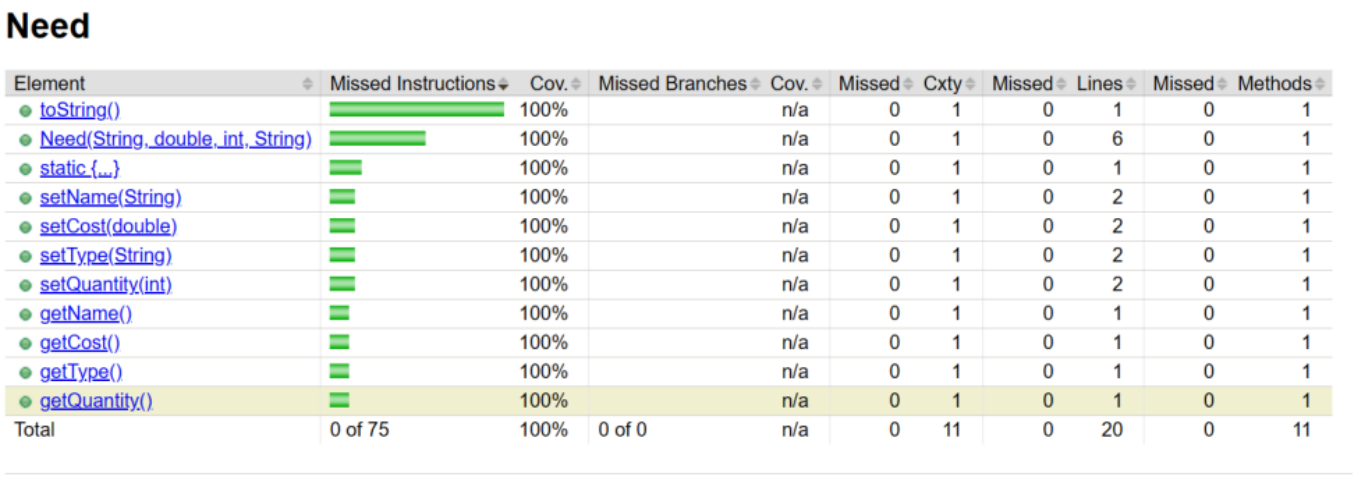


ufund-api > com.ufund.api.ufundapi.persistence > NeedFileDAO

NeedFileDAO

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
load()	🟩	100%	🟩	100%	0	2	0	5	0	1
getNeedsArray(String)	🟩	100%	🟩	100%	0	4	0	8	0	1
createNeed(Need)	🟩	100%		n/a	0	1	0	5	0	1
updateNeed(Need)	🟩	100%	🟩	100%	0	2	0	6	0	1
deleteNeed(String)	🟩	100%	🟩	100%	0	2	0	5	0	1
getNeed(String)	🟩	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
static {...}	🟩	100%		n/a	0	1	0	1	0	1
getNeedsArray()	🟩	100%		n/a	0	1	0	1	0	1
Total	0 of 241	100%	0 of 14	100%	0	19	0	47	0	12

The code coverage for this component is good because the unit tests we implemented for this class cover a wide variety of tests. The unit tests created for this NeedFileDAO class evaluate if the persistence tier correctly formats into a string. Also, it checks if the setters, getters, updateNeed, createNeed and deleteNeed function correctly.



Need

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
toString()	🟩	100%		n/a	0	1	0	1	0	1
Need(String, double, int, String)	🟩	100%		n/a	0	1	0	6	0	1
static {...}	🟩	100%		n/a	0	1	0	1	0	1
setName(String)	🟩	100%		n/a	0	1	0	2	0	1
setCost(double)	🟩	100%		n/a	0	1	0	2	0	1
setType(String)	🟩	100%		n/a	0	1	0	2	0	1
setQuantity(int)	🟩	100%		n/a	0	1	0	2	0	1
getName()	🟩	100%		n/a	0	1	0	1	0	1
getCost()	🟩	100%		n/a	0	1	0	1	0	1
getType()	🟩	100%		n/a	0	1	0	1	0	1
getQuantity()	🟩	100%		n/a	0	1	0	1	0	1
Total	0 of 75	100%	0 of 0	n/a	0	11	0	20	0	11

The unit tests for this component are good because the unit tests we implemented for this class cover a wide variety of tests. The unit tests created for this class evaluate if the Need class correctly formats into a string and if the setters and getters for each attribute works.