

U-Fund Design Documentation

LAWNN

Modification Log

Date	Version	Description	Author(s)
2/20/24	1.0	Initial Draft	
3/21/24	2.0	Second draft	Ada Grande
4/23/24	3.0	Sprint 4 revision, updated images	Nathan Klein

Team Information

Team Name: **LAWNN**

Team Members: **Nathan Klein**
Nathan McAndrew
Ada Grande
Wendy Carrillo Monarca
Leif Gunhus

Executive Summary:

Bucks for Baddies is a website only the most sinister villains can access. Villains in search of donations for their evil master plans can access our site to petition for donations by logging in as an 'admin' and creating needs. And those wanting to 'help' villains in need can donate to their fellow villains by adding evil needs to their carts.

Purpose:

Bucks for Baddies aims to be a user-friendly platform that is easy for the management of needs and resources for an evil organization.

User group and User goals:

- **Managing needs:** Add new needs, update existing one and marking the as complete
- **Searching and viewing needs:** Easy to see all needs, filter them based on criteria and find specific needs fast.
- **Managing shopping cart:** Add new items to cart.

Glossary and Acronyms:

Term	Definition
UI	User Interface
API	Application Programming Interface
Need	A monetary request, paid for by other users
MVP	Minimum Viable Product
UML	Unified Modeling Language

Requirements:

User Sign-In/Sign-Out:

- A user must sign in with an alias that is not currently being used.
- Once signed-in, a user can access an inventory of needs

Helper abilities

- Users that log in as helpers will be able to view needs and add them to a shopping cart.
- Users can search for specific needs.
- Users can also delete needs from their shopping cart.

Admin abilities

- A user can log in as an admin and create a need, delete a need and change the information of a need.

Definition of MVP:

A MVP or Minimum Viable Product is a design technique in which, when creating a program, you first create the bare minimum of the project. This means that you code only what is required of the base program, and no extras. In our ***U-Fund*** project, this means that we are created the base game of checkers before creating extra features like 'Help' or 'AI Player'

MVP Features:

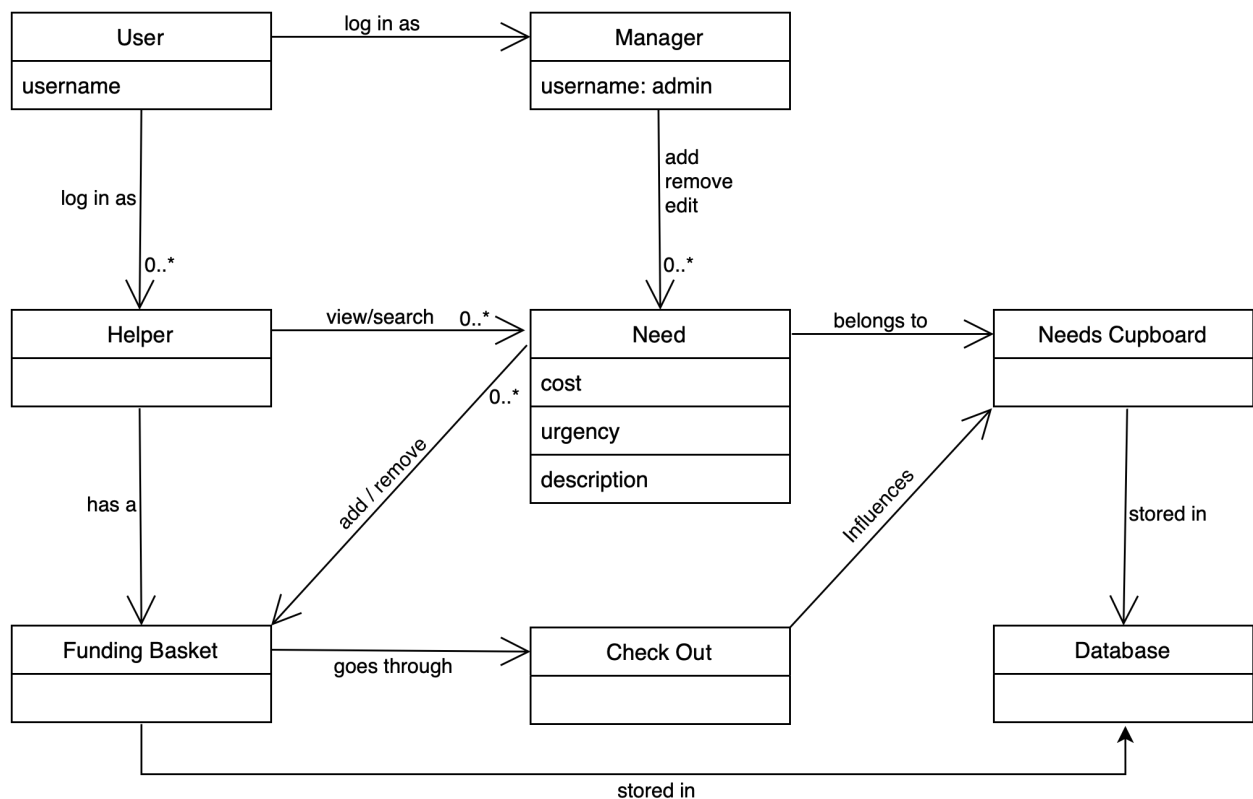
- **Minimal Authentication for Helper/U-fund Manager login & logout**
 - The server will (admittedly insecurely) trust the browser of who the user is. A simple username to login is all that is minimally required. Assume a user logging in as admin is the U-fund Manager.
 - You are not expected to do full credential and session management, although the system will look different depending on who is logged in. Obviously this isn't how things are done in real life, but let's sidestep that complexity for this class.
- **Helper functionality**
 - Helper can see list of needs
 - Helper can search for a need
 - Helper can add/remove an need to their funding basket
 - Helper can proceed to check-out and fund all needs they are supporting
- **Needs Management**
 - U-fund Manager(s) can add, remove and edit the data of all their needs stored in their needs cupboard
 - A U-fund Manager cannot see contents of funding basket(s)
- **Data Persistence**
 - Your system must save everything to files such that the next user will see a change in the needs cupboard based on the previous user's actions. So if a Helper had something in their funding basket and logged out, they should see the same needs in their cupboard when they log back in.
 - Ordinarily, we would want to use a database for this - but this semester our system will not reach a complexity that requires a database. You learned basic file I/O in your programming courses, so utilize what you know from there.
- **10% additional feature enchantment(s)**
 - Team will propose this to your instructor by end of Sprint 1, and then build it into your solution as the semester progresses. See below for more details.

Application Domain:

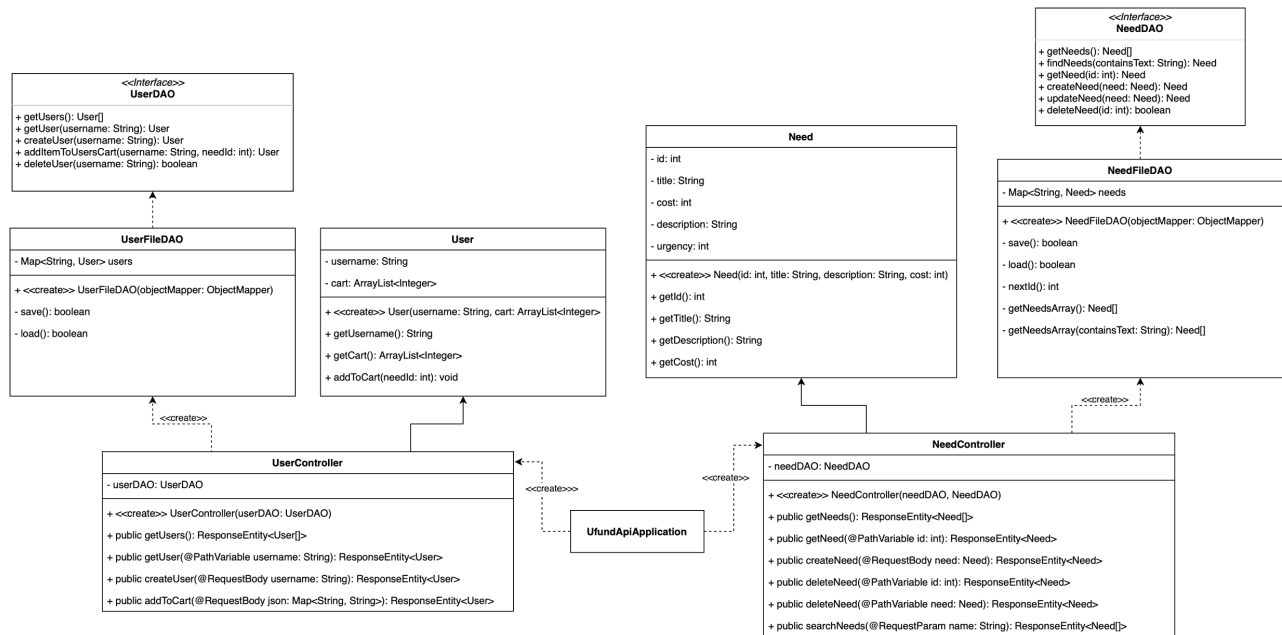
Overview:

A user can either log in as a Manager or a Helper. A manager can add, remove and edit a need. Each need has a cost, urgency, and description. A need belongs to a needs cupboard and can be placed or removed from a funding basket. A helper has a unique funding basket and this basket goes through a checkout process. This checkout directly influences the needs cupboard.

Domain Area Detail

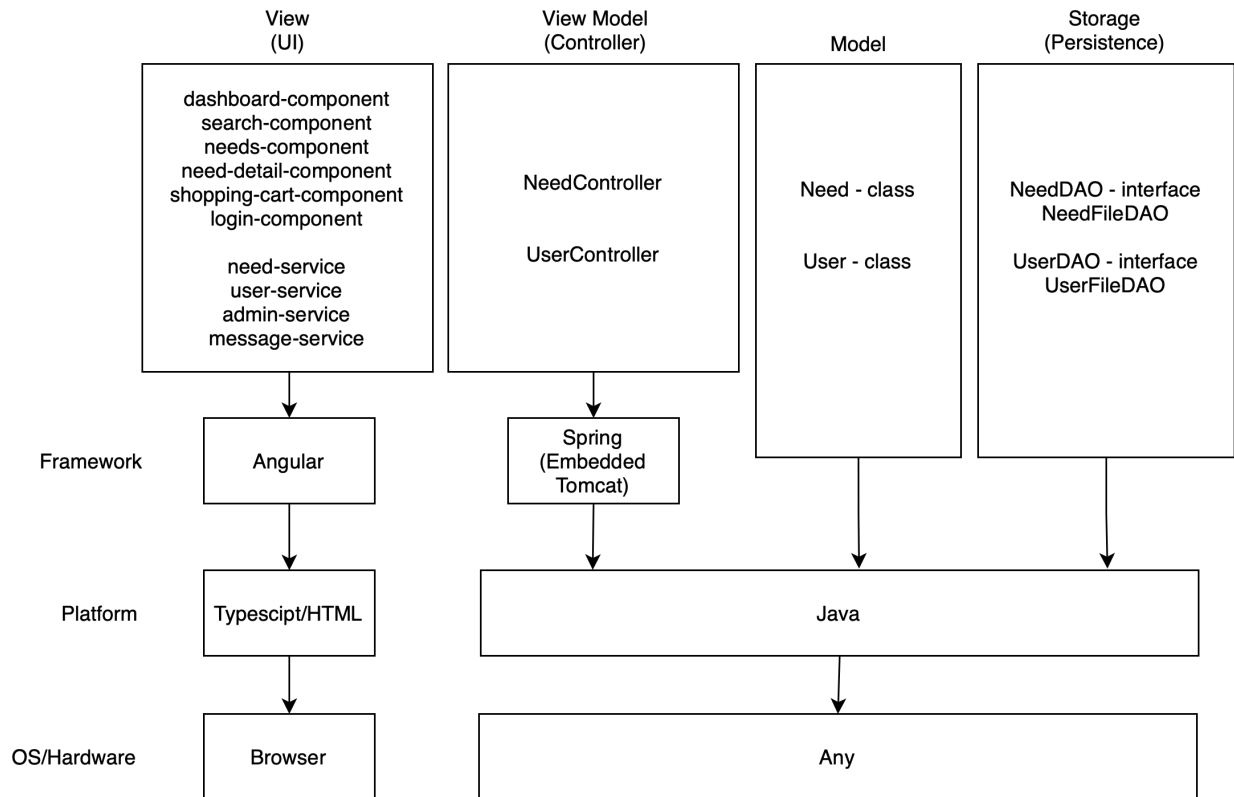


UML Class Diagram:



Application Architecture:

Summary:



OO Design Principles:

Law of Demeter:

The law of demeter suggests that objects should have limited knowledge about other objects and should interact only with their friends and not with strangers or unfamiliar objects. This design principle is seen with how the helper and the manager interact directly with the need. To improve to have better adherence to the law of demeter principle, we could consider encapsulating interactions between objects more strictly. Classes can be introduced to help with the interactions between the helper and other objects.

Pure Fabrication:

Pure fabrication is used to balance other design principles. This design principle is seen with our funding basket since it is used to facilitate the management of needs by the helper. To improve to have a better adherence to the pure fabrication principle we can work on ensuring that the responsibilities of the needs cupboard and funding basket are well defined.

Single Responsibility:

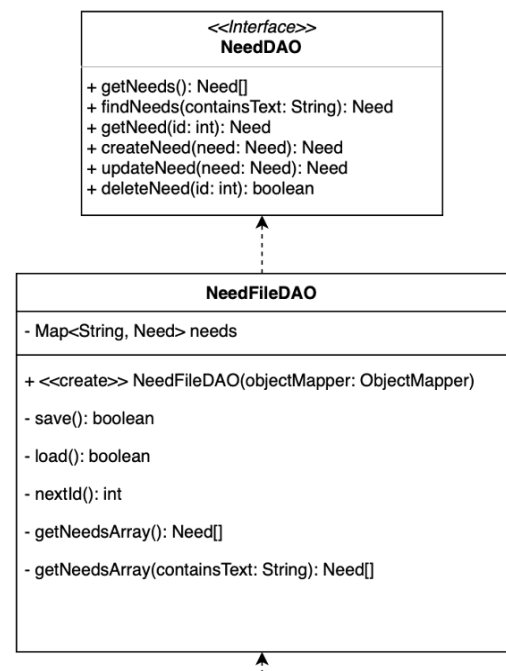
To comply with Single Responsibility, each of our classes should have only one responsibility. For example, our NeedFileDAO class is only responsible for interfacing with the Need storage file. This is an example of Single Responsibility because the NeedFileDao is only responsible for handling requests for Needs. It will not handle requests for retrieving a saved funding basket, or any other stored information.

Open/Closed:

To comply with the Open/Closed principle, our classes should only modify the functionality of existing classes by adding new code to extend them, not by changing the existing code. An example of this is our NeedDAO interface, and the NeedFileDAO that implements it. The NeedDAO interface provides all the necessary declarations for storing and retrieving Needs, while the NeedFileDAO class extends this by implementing these methods specifically with respect to storage in Files. The NeedDAO interface could be extended to implement the methods with respect to a Database storage system in place of the File storage system.

Low Coupling:

Low Coupling attempts to minimize the impact of changes in the system and assign responsibility so that unnecessary coupling remains low. While this may require two objects to be heavily connected, our design will have adding and deleting items in one class. This will be done so error checking will be simple if a need is incorrectly added or removed. Updating/editing a need will take place in a separate class as it is a separate idea/process.



Testing:

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

Acceptance Testing:

All stories passed acceptance criteria testing.

Here are the following stories with the completed acceptance criteria.

- **Helper Authentication**
 - When entering a username and the username exists already then I will be logged in that user
 - When entering a username and the username does not exist already then a new user is created, and I get logged in as them.
- **Manager Authentication**
 - When a username is entered as admin then the user will be logged in as a manager.
- **Admin Service**
 - was able to subscribe to observable ****isAdmin**** boolean
 - was able to subscribe to observable ****isLoggedIn**** boolean
 - was able to subscribe to observable ****username**** string
- **Need Service**
 - users were able to send need requests
- **Creating a new need**
 - Given the need does not exist yet, when I create a new need, I expect to see it appear in the cupboard
 - Given the need already exists, when I try to create one with the same name or id, I expect to be notified of some kind of error.
- **Updating a need**
 - created and tested `updateNeed()` function. And the method is connected to a button and is styled.
- **Getting a single need**
 - Given I click on a need, I see a detail page with more information about the need
- **Getting the entire cupboard**
 - Given the home screen as a helper or manager, I see the entire needs cupboard when I log onto the home screen
- **Searching for needs**

- Given a search bar, when I type in a need's title, I see it appear in the results
- Given a search bar, when I type in a title that no need has, I see no results
- Deleting a single need
 - When there is an existing need and it is deleted it no longer appears in the needs cupboard
- CartDAO
 - given a username...
 - when the username is associated with a user, the method returns that user object and HTTP OK
 - when the username is not associated with a user, the method returns HTTP NOT FOUND
 - when the username is associated with a user, the method returns that users cart in Need[] format and with a HTTP OK
 - When the method is called, a list of all users returned with a HTTP OK
- User service
 - user was able to send a request to create a user
 - user was able to send a request to update a users cart
 - user was able to send a request to get a specific user
 - user was able to send a request to get all users

Unit Testing and Code Coverage

Controller Tier

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

Element	Missed Instructions	Cov.	Missed Branches	Cov.
getNeed(int)		67%		100%
createNeed(Need)		67%		100%
updateNeed(Need)		66%		100%
deleteNeed(int)		63%		100%
searchNeeds(String)		57%		n/a
getNeeds()		54%		n/a
NeedController(NeedDAO)		100%		n/a
static {...}		100%		n/a
Total	66 of 192	65%	0 of 8	100%

Analysis

The only lines of code that are not reached by our tests for this class are the catch branch of our try/catch statements for catching IO exceptions in the DAO. We weren't sure how to generate an IO exception for testing purposes, so this was left out of our testing. Adding a test to test this would be a good improvement to make.

Model Tier

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

Need

Element	Missed Instructions	Cov.
setTitle(String)		0%
setDescription(String)		0%
setCost(int)		0%
toString()		100%
Need(int, String, String, int)		100%
static {...}		100%
getId()		100%
getTitle()		100%
getDescription()		100%
getCost()		100%
Total	12 of 71	83%

(branch statistics omitted because our model class has no branches!)




















Analysis

Our tests reached all the code except for the attribute setter methods. Since we implemented full functionality for creating/updating/deleting needs, perhaps this code will never be used and we can delete it. More team discussion on this is needed to determine if we should keep and test the code or just delete it.

Persistence Tier

This is our analysis [at](#) the Persistence Tier code for the project.

NeedFileDAO

Element	Missed Instructions	Cov.	Missed Branches	Cov.
createNeed(Need)		100%		100%
load()		100%		75%
getNeedsArray(String)		100%		100%
updateNeed(Need)		100%		100%
deleteNeed(int)		100%		100%
getNeed(int)		100%		100%
save()		100%		n/a
findNeeds(String)		100%		n/a
getNeeds()		100%		n/a
NeedFileDAO(ObjectMapper)		100%		n/a
nextId()		100%		n/a
static {...}		100%		n/a
getNeedsArray()		100%		n/a
Total	0 of 293	100%	1 of 20	95%

















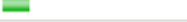


Analysis

Every line of code in our persistence class was reached. In terms of logic reached, we missed half a branch of an if statement that checks for the max id encountered in order the set `nextId` correctly. Upon brief analysis however, we think that this if statement might be redundant since with the way we save needs, the scenario for which the if statement tests will never be encountered.

Well-tested Component

This is our analysis of a well-tested component.

NeedFileDAO





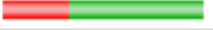
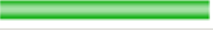






Element	Missed Instructions	Cov.	Missed Branches	Cov.
createNeed(Need)		100%		100%
load()		100%		75%
getNeedsArray(String)		100%		100%
updateNeed(Need)		100%		100%
deleteNeed(int)		100%		100%
getNeed(int)		100%		100%
save()		100%		n/a
findNeeds(String)		100%		n/a
getNeeds()		100%		n/a
NeedFileDAO(ObjectMapper)		100%		n/a
nextId()		100%		n/a
static {...}		100%		n/a
getNeedsArray()		100%		n/a
Total	0 of 293	100%	1 of 20	95%

Analysis

We hit every single line of code in the class, which is very good. Missing half a branch of if statement logic means it's not perfect, but we think that if statement can be removed entirely, which would yield 100% coverage for the entire class!

Poorly-tested Component

This is our analysis of a poorly-tested component.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
● getNeed(int)		67%		100%
● createNeed(Need)		67%		100%
● updateNeed(Need)		66%		100%
● deleteNeed(int)		63%		100%
● searchNeeds(String)		57%		n/a
● getNeeds()		54%		n/a
● NeedController(NeedDAO)		100%		n/a
● static {...}		100%		n/a
Total	66 of 192	65%	0 of 8	100%

Analysis

Our controller class was lacking in the fact that we only hit 65% of the lines of code. This was because we did not test with IO exceptions. Adding this to our test suite would yield 100% coverage. On the flip side, we did test all branches of logic which is good.

Design Quality and Possible Revisions

We believe the design of our application is both functional and practical. Implementation went smoothly and all required features of the MVP were met, indicating that our backend and frontend designs were sound and meshed together well. Effort was put into making our design as simple as possible for easier maintenance and understanding of the application. This is displayed by our Domain Model, which is simple, orderly, and easy to interpret. Adding extra congestion and nuances to our design would have yielded the same set of functional MVP features while complicating the implementation process immensely, which was our reasoning for designing the application the way that we did.

While we are happy with the end result of our application, there are some areas for potential improvement. The first of which would be redesigning the UI to look more modern, perhaps with the use of more images. This would have no impact on the functionality of our application or on the underlying design and architecture, rather it would serve to make the application look nicer and be more enjoyable to use. Secondly, we would like to change the checkout process so that users are able to purchase portions of needs. This would require the code for the shopping cart Angular component to be modified, but it would not be a major change.