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: placeholder
- Team members
  - Justin Wu
  - Daniel Arcega
  - Ricky Yang
  - John Li

Executive Summary

This is a summary of the project. This is a website designed for ArtsRoc, an organization which provides music and lego programs for young kids. It takes in donations from others via items which denote where the money spent will go, shows everybody what events the organization will hold in the future, and provides donation rewards for people who give enough. It also allows for members of the organization, given some permissions, to directly modify the events, the list of stuff people can buy from, and the rewards for donating some amount of money.

Purpose

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

This website is for ArtsRoc to help fund its endeavors through outside donations. Users should be able to pick and choose what programs they want to fund and receive rewards for their actions.

Glossary and Acronyms

> ***[Sprint 2 & 4]*** *Provide a table of terms and acronyms.* | Term | Definition | |------|------------| | SPA | Single Page |

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.*

The ArtsRoc website that allows people to donate to ArtsRoc's various programs (or/and aspects of said programs). It also allows the people responsible for managing the funds to change what the donors can buy, given the organization wants more funding towards a different set of things than what's there already.

MVP Features

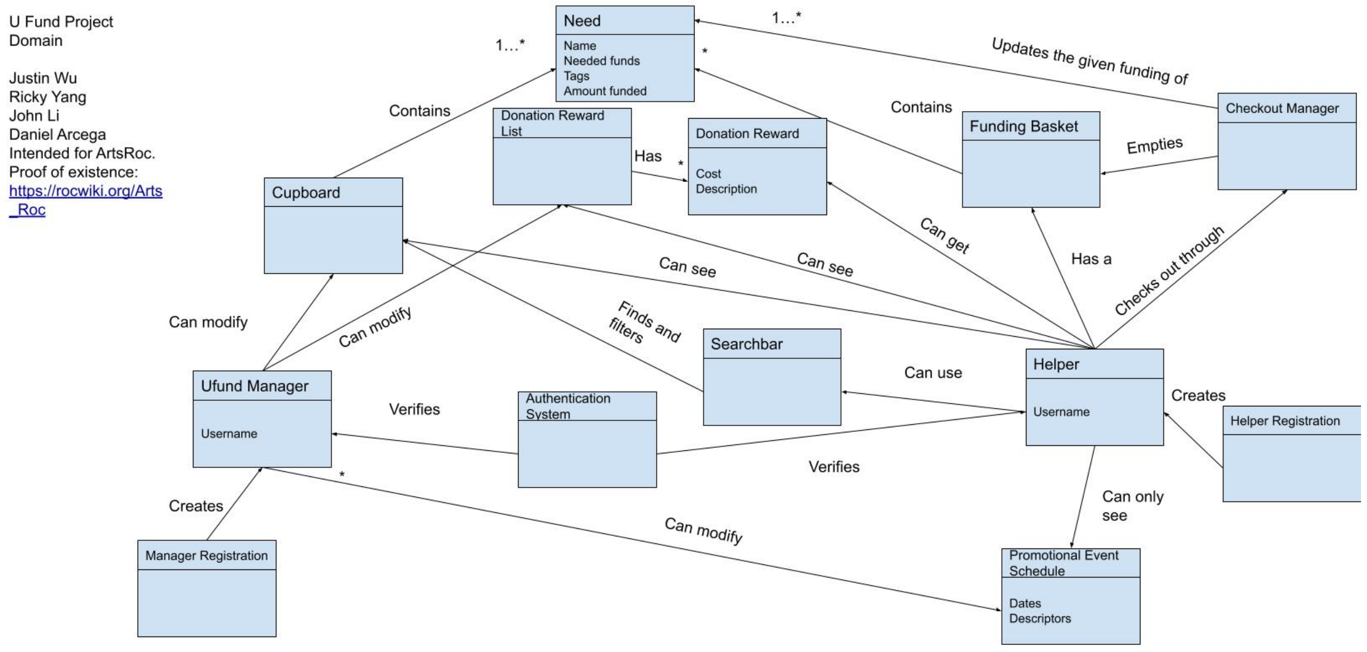
> ***[Sprint 4]*** *Provide a list of top-level Epics and/or Stories of the MVP.*

Enhancements

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

Application Domain

This section describes the application domain.



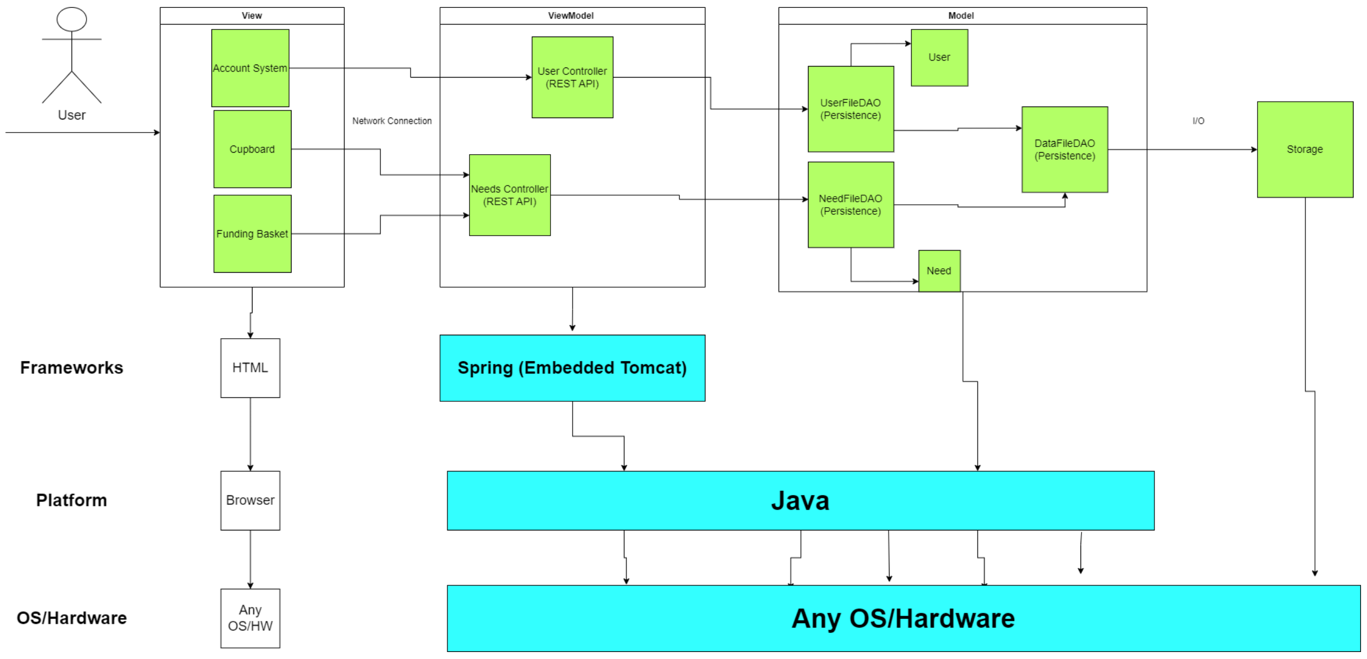
As of now, the helper is a donor who has a set of items they can purchase in order to help ArtsRoc fund its programs (or aspects of them), and in turn, could get a reward given they've donated enough. They can also see specific events scheduled by the organizations. The Ufund Manager directly modifies what the helper and search through and buy. They also will have full control over the donation rewards and the event schedule. The checkout manager helps the user manage the set of orders they want to purchase alongside providing a way for them to do such.

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. (*When requested, replace this diagram with your* ***own*** *rendition and representations of sample classes of your system*.)



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 in the web application.*

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.*

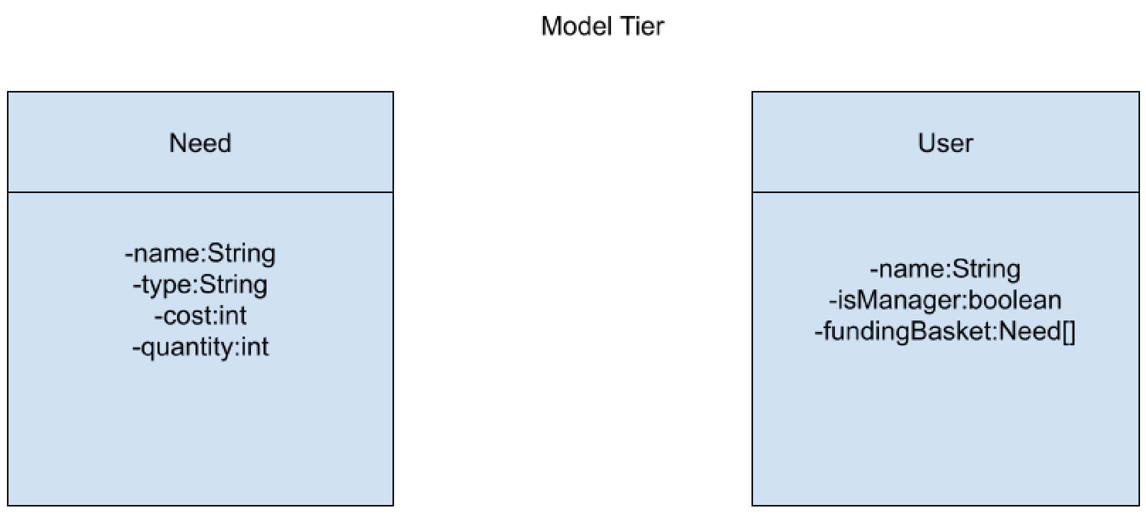
ViewModel Tier

> ***[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. 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.*



The model tier is an intermediary between the objects that can be modified by the frontend Angular section and the JSON database. There are two main classes in this tier: User.java and Need.java. User.java represents the individual data of a user, while Need.java represents a need that could be bought by a user. Need.java contains a price, a name which acts as the unique id to prevent duplicates, a tag which can be used to filter out details, and finally a quantity. User.java contains the username, the current items that the user will buy, and if it is an admin or not.

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.*

Open/closed

The way the DAO and Controller objects are designed are under the assumption that there will be similarities in how the data is handled. Specifically, for the DAOs, if all objects serialized and deserialized will have a string as a unique identifier and are organized the same way, then there could be a common interface that is used. The controllers which each handle different types of data(User, etc.), on the other hand, all handle the transmission of data via http the same. Therefore each one inherits a common abstract class with all https methods implemented: the only difference is that each class uses a different path for accessing data and themselves for the logger (in order to show which controllers are called).

Single Responsibility

Each class has a single responsibility. For instance, the User.java class handles data pertaining to individuals who use the website, while the DAO classes each handle deserialization/serialization of a given object. The controller objects each handle a different DAO and link the app to whatever uses the database.

> ***[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.*

User Stories and Unit Tests: The majority of the user stories were successfully implemented without significant issues. All the unit tests that were executed passed, ensuring the functionality adhered to the predefined criteria. Ufund Manager Story Testing: Testing the ufund manager story demonstrated that all the functionalities met the expected criteria. During live testing, users were able to interact with the cupboard and funding basket, performing operations such as adding items to the funding basket and canceling or reverting actions by clicking outside the designated buttons. Helper Stories Testing: The add and subtract functionalities for the funding basket were tested thoroughly, and both met the acceptance criteria. Live testing confirmed that users could add multiple needs to the funding basket and remove them accordingly. Both the subtract and add operations supported quantity increments and decrements as expected. User Login and Logout Testing: The user login, sign-up, and logout functionalities were found to be satisfactory, considering that the input was not entirely composed of whitespace. The sign-up process ignored usernames that already existed, while the login process verified the credentials against existing records. Successful login granted users access to the funding basket and associated functionalities, and the logout process effectively restricted access to the account until re-login. Search Functionality Testing: The search functionality successfully filtered results from the original list of fundable items, as expected. The fundable items were displayed in a list grid format, consistent with the predefined acceptance criteria. Checkout Functionality Issue: While the checkout functionality seemed to meet the criteria during unit testing, there was an issue with updating or refreshing the funding basket after checkout in the live testing phase. This issue requires further investigation to ensure the funding basket is appropriately updated and refreshed post-checkout.

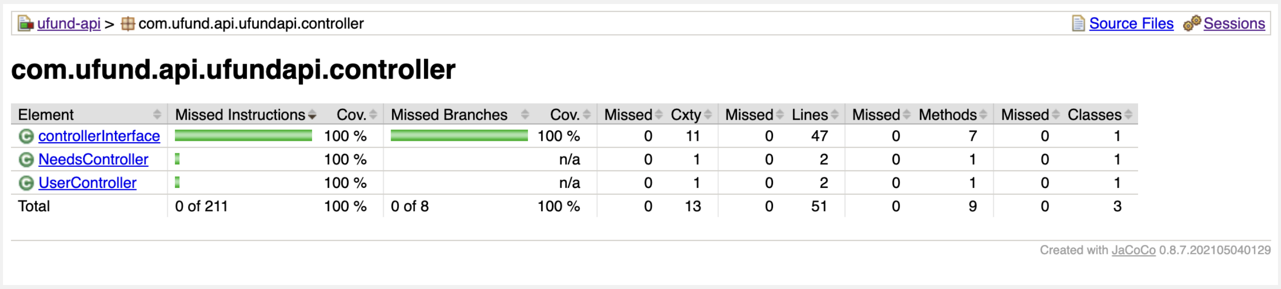
Acceptance Testing

> ***[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.*

The majority of the user stories passed without much issues and all the unit tests that we tested passed. For the ufund manager story, all of them performed to our expectations. During our live testing of the website the, user/helper was able to interact with the need in the cupboard to add it to the funding basket as well as any functionality such as cancelling/reverting the operation of moving the needs by clicking anywhere but the button responsible for moving it to the funding basket. The helper stories, add and subtract to/from funding basket also met our acceptance criteria and in live testing on the webpage, add and subtract performed to our expectations, the add was able to add multiple needs to the funding basket and the subtract was able to take them out. Both the subtract and the add worked with quantity increment and decrements. The user log in, sign up, and logout stories also performed sufficiently in the case that the input was not entirely compose of whitespace. In the event that a username already existed the sign up would ignore it. The log in would match the credentials of with existing ones in the system otherwise it would ignore them. Once logged in the user had access to the funding basket to add needs, remove needs as well as any functionality given to users. The live testing confirmed that the log out worked and made sure that the user had lost access to their account as well as any functionality tied to having an account until they logged back in. Our search story also performed to our expectations since it would filter out results from our original list of fundable things. The fundable things were also displayed in a list grid format as consistent with our acceptance criteria. However, while the checkout functionality seemed to meet the criteria for the checkout story in unit testing there was an issue with updating/refreshing the funding basket after checkout (where checkout consisted of clearing the funding basket) during live testing

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.*

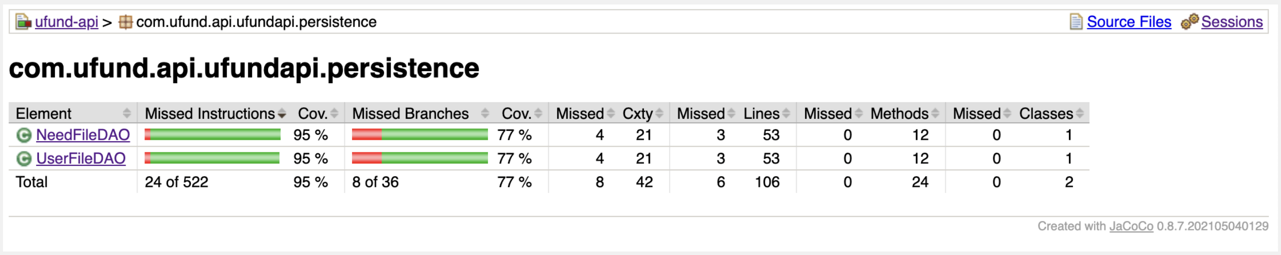


ufund-api > com.ufund.api.ufundapi.controller Source Files Sessions

com.ufund.api.ufundapi.controller

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
controllerInterface		100 %		100 %	0	11	0	47	0	7	0	1
NeedsController		100 %		n/a	0	1	0	2	0	1	0	1
UserController		100 %		n/a	0	1	0	2	0	1	0	1
Total	0 of 211	100 %	0 of 8	100 %	0	13	0	51	0	9	0	3

Created with JaCoCo 0.8.7.202105040129

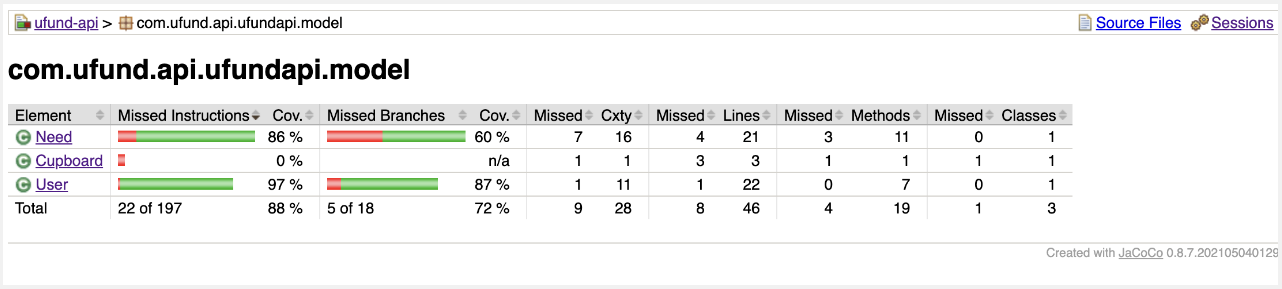


ufund-api > com.ufund.api.ufundapi.persistence Source Files Sessions

com.ufund.api.ufundapi.persistence

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
NeedFileDAO		95 %		77 %	4	21	3	53	0	12	0	1
UserFileDAO		95 %		77 %	4	21	3	53	0	12	0	1
Total	24 of 522	95 %	8 of 36	77 %	8	42	6	106	0	24	0	2

Created with JaCoCo 0.8.7.202105040129



ufund-api > com.ufund.api.ufundapi.model Source Files Sessions

com.ufund.api.ufundapi.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Need		86 %		60 %	7	16	4	21	3	11	0	1
Cupboard		0 %		n/a	1	1	3	3	1	1	1	1
User		97 %		87 %	1	11	1	22	0	7	0	1
Total	22 of 197	88 %	5 of 18	72 %	9	28	8	46	4	19	1	3

Created with JaCoCo 0.8.7.202105040129

> Most of the uncovered code are the getters and setters in the Need.java object. Our group thought that if one getter worked, then the others would, given all the untested ones handled primitive values. As of writing, the cupboard.java object has been removed. A branch involving two objects not being equal in the equals function of User.java isn't covered either, mainly because they don't have anythign in particular outside of a return value.