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: Music Magic
- Team members
  - Teddy Davies
  - Aaliyah Dalhouse
  - Daniel Tsouri
  - Omar Morales-Saez
  - Sean Gaines

Executive Summary

The New Orleans Music Fund website allows organizers to maintain a list of needs requested by various public school music departments. These needs can then be viewed by potential helpers, who can choose to donate in order to buy these needs for the respective schools.

Purpose

> ***[Sprint 2 & 4]*** *Provide a very brief statement about the project and the most important user group and user goals.* Allows administrators to add needs to a cupboard of needs from which helpers can add needs to their basket and fund them.

Glossary and Acronyms

> ***[Sprint 2 & 4]*** *Provide a table of terms and acronyms.*

Term	Definition
SPA	Single Page
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
IDE	Integrated Development Environment
CRUD	Create, read, update, and delete
JSON	JavaScript Object Notation
OOP	Object Oriented Proggraming

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.* For a minimum viable produce, a user must be able to:

- login as either a helper or an admin
- If they are a helper, see a list of needs and add or remove needs to/from their funding basket.
- If they are an admin, see a list of needs and add or remove needs from the list.
- Both should be able to search the list of needs.

MVP Features

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

EPIC: Helper Checkout

EPIC: 10% Feature

EPIC: Visual Aspect of the Website

EPIC: Desing Document

Enhancements

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

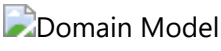
Our two enhancements were an interactive slideshow embedded in our login page, and a music player embedded in our login page.

The slideshow displays an image with a caption underneath it with a description of the image. There are two buttons that allow the user to navigate to the previous or next image. There are also buttons underneath the image that allow the user to jump directly to the image. The number of buttons automatically adjusts when adding new images to the slideshow. However, adding a new image to the slideshow can only be done by editing the html.

The music player is very simple, it is just an audio player element that plays a recording from one of the school music departments that we support.

Application Domain

This section describes the application domain.


Domain Model

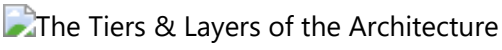
> ***[Sprint 2 & 4]*** *Provide a high-level overview of the domain for this application. You can discuss the more important domain entities and their relationship to each other.*

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 Tiers & Layers of the 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.

> *Provide a summary of the application's user interface. Describe, from the user's perspective, the flow of the pages in the web application.*

Any new users is first directed to the login page where they can sign in as either a helper or an admin.

If they login as an admin they are directed to the administrator view, where they are able to see a list of needs, a button next to each need which allows them to remove a need, a form which lets them add a need, and a search bar which lets them search the needs.

If they login as a helper they ared directed to the helper view, where they are able to see a list of and a button next to each need which allows them to add a need to the funding basket. They can also see a funding basket, which they can remove needs from, and a search bar which lets them search the list of 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.*

Upon entering our site, users are greeted by the landing page, seamlessly integrated with the login component. The site offers the option to enter a username and password for login, accompanied by a captivating image slider component showcasing local events. To initiate the login process, users may choose between two credentials: helper or admin.

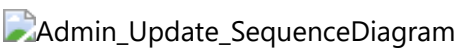
For admin access, entering 'admin' as the username and 'password' as the password is required. Once authenticated, users are directed to the admin component, granting them the authority to add, delete, and

update the needs of local school music departments in our database. Within the admin component, the add-need and needs components empower administrators to view current needs and add new requirements. The need component, with functions such as getNeeds, delete needs, and move needs to the basket, ensures a functional admin view. Meanwhile, the add-need component simplifies the process of adding new needs.

To log in as a user/helper, input 'helper' for both username and password. The helper component provides access to the needs component, displaying the current cupboard of needs, the personal funding basket component for tracking selected needs, a need-search component facilitating cupboard exploration, and logout functionality. Progressing to checkout through the checkout component requires adding at least one item from the current cupboard to the funding basket. After this step, users can click the "proceed to checkout" button, review the summary of purchased items, and place an order for swift delivery.

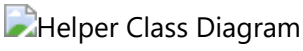
In addition to these user-facing components, various behind-the-scenes components contribute to the functionality. A routing component facilitates smooth transitions between different components, a message component displays error/success messages, and modules inject data into constructors within specific components. These collectively ensure a seamless and efficient user experience within our architecture.

> ***[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.*


Admin_Update_SequenceDiagram


Helper_Add_SequenceDiagram

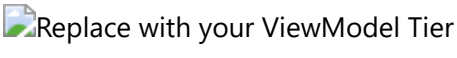
> ***[Sprint 4]*** *To adequately show your system, you will need to present the **class diagrams** where relevant in your design. Some additional tips:*
>
> - *Class diagrams only apply to the **ViewModel** and **Model** Tier*
> - *A single class diagram of the entire system will not be effective. You 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.*


Admin Class Diagram 
Helper Class Diagram

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.*

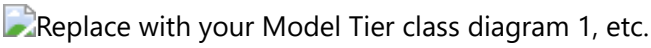

Replace with your ViewModel Tier class diagram 1, etc.

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.*

Our api backend functions by maintaining two json files of needs, one representing the cupboard and the other representing the funding basket. The json files are able to be modified by the NeedFileDAO and BasketFileDAO, which are both controlled by the interfaces NeedDAO and BasketDAO respectively. NeedController and BasketController allow editing of the json files using http requests, and the Need class represents one Need itself.

> *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.*

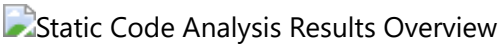
Sprint 2

- We apply **Single Responsibility** in our project by having classes that are limited to one responsibility. For example we have multiple classes for the funding basket because we have one class that is responsible for the deserialization of the basket.json file and handling of objects, and another class that is responsible for responding to HTTP requests and returning the correct objects for the funding basket.
- We apply **Open/Closed** in our project by extending interfaces such as NeedDAO and BasketDAO into the new classes NeedFileDAO and BasketFileDAO respectively. This allows us to make modifications to the child classes while keeping the parent class untouched.
- We apply Pure Fabrication in our project by having classes like NeedFileDAO, which is not something that has a real-life representation. Pure Fabrication just means that some objects in our code do not exist in the real world.
- We apply Law of Demeter in our project by restricting the scope of a class's knowledge of other classes to only it's direct neighbors. For example, the NeedController physically cannot reach into the need.json file, it needs to rely on other classes, NeedDAO and NeedFileDAO, in order to get what it needs.

> ***[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.*

 Static Code Analysis Results Overview

Our static code analysis found 128 potential maintainability issues, with 7 marked as high severity.

Issue 1


Static Code Issue 1

The first high-severity problem is our repeated use of the identical string literal "Put /needs" in several parts of our code. If we wanted to edit the output of our log messages, we would need to change every instance of "Put /needs" in our code, which is not only unnecessarily laborious, but could lead to future issues with the logger if one of the instances is missed. By replacing the literal with a constant variable, defined elsewhere, we could make our code more maintainable by allowing every instance of "Put /needs" in the code to be changed by only changing one value.

Issue 2


Static Code Issue 2-1

The next high-severity issue was flagged three times by sonar-qube, but is really the same problem repeated several times.


Static Code Issue 2-2

Whenever the static variable nextId is modified in NeedFileDAO, sonar-qube flagged the method saying it should also be static. The reason for this is that if a static value is modified from a non-static and non-synchronized method, it could cause problems if there are multiple instances of the class at once trying to access or modify the static variable, or multiple threads.

Despite sonar-qubes recommendation that the method be made static, NeedFileDAO is an injectable depedency that should only be instantiated once, so the method being static or not static shouldn't have any effect. However, the methods should be changed to synchronized, as there could be situations where multiple threads are trying to access NeedFileDAO, and therefore we need to avoid a race condintion or similar.

Issue 3

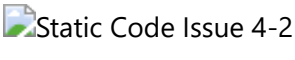

Static Code Issue 3-1

The next 3 issues marked are also fundimentally the same issue appearing in several places. In some of our unit testing there is a test class and two tests which have either no assertions or no tests. Investigating these tests, it seems that these were sections of code that were either made irrelevant or discarded at some point during development, but were not removed. If left in, these segments could confuse future developers, who are trying to understand what this extra code with seemingly no purpose is doing. Therefore, in order to improve the maintainability of our code, we should remove these segments to prevent future confusion.

Issue 4


Static Code Issue 4-1

This issue was flagged every time information was logged using the Logger in our code.


Static Code Issue 4-2

Whenever we pass a string concatination in as an argument to a Logger function, we potential take an unncessary performance hit. The logging level may not be high enough to justify displaying the information in

the logger, but code will still process the concatination operation, despite it having no purpose. As a result our code may sometimes perform unnecessary operations during runtime. To fix this, we can simply use the Logger functions' built in formatting to avoid this issue, such as seen in the example provided by sonar-qube.


Static Code Issue 4-3

> ***[Sprint 4]*** *Discuss* ***future*** *refactoring and other design improvements your team would explore if the team had additional time.*

Future Improvements

In addition to what was mentioned above, several further improvements could be made to the project's design and architecture.

- As it stands there is no way to change the admin or user password without directly modifying the json file storing them. This especially problematic since the passwords are naturally stored as a hash, meaning in order to change a password someone with access to the backend server has to create a hash of the password, with the correct parameters outline, and put it into the json file storing user login information. This is a lot of work just to change someone's password, even for a developer. A future improvement would be to add a way on the website to change a user's password directly.
- While the backend was modified so that the funding basket only stores a set of ids, each referring to a need in the cupboard, the frontend still has the funding basket store it's own full array of needs. While whether or not that same change to the frontend should be made is not as clear cut as it was for the backend. However, it is worth investigating to see if it would be better for the frontend server to store a set of ids which refer to needs in the cupboard, rather than it's own array of needs.
- While our unit testing coverage is at an acceptable 82.5%, it could still be higher. Adding more unit tests or modifying our current ones in order to further increase our code coverage would be a worthwhile improvement our team could investigate in the future.

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.*

- SPRINT 2: So far, the team has been able to keep up with Acceptance Testing. The entirety of stories in both Sprint 1 and Sprint 2 have passed all of their acceptance criteria tests. The stories in Sprint 3 do not have any tests yet. They wil be assigned tests during the Sprint 3 planning. There has not been found any issues with the acceptance tests.
- SPRINT 4: All of the user stories have had testing and have passed them all excpet for "Add Landing Page". This story was not able to be implemented and therefore did not pass any of the acceptance criteria. Besides this story, there has been no problem with the story acceptance tests.

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.*

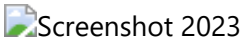
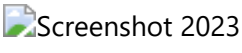
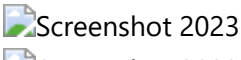
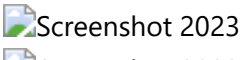
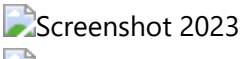
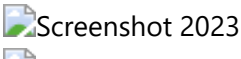
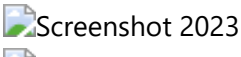
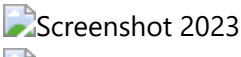
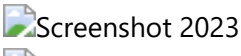
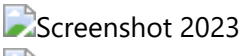
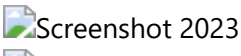
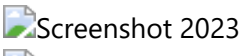
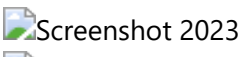
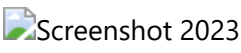
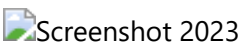
> ***[Sprint 2 & 4] Include images of your code coverage report.*** *If there are any anomalies, discuss those.*

Sprint 2

- There is one large chunk of code that we are having trouble writing unit tests for, which includes updateNeedCost(), updateNeedquantity(), and updateNeedName. It is hard for us to write unit tests for this code because there was nothing like it in the Heroes api for us to base our code off of.
- Apart from that chunk, we just have a few holes spotted around the place which should be no problem to patch up in Sprint 3.

Sprint 4

- We did not have a set strategy for the Unit Tests. For the most part, after the JUnit tests assignment, all team members that added a new function also made the test at the same time. This worked for the most part. However, as seen in the pictures below, this strategy is not perfect. The persistence tier is lower than the rest because the UserFileDAO had no tests. Our team aimed for 90 percent coverage because this was the amount that allowed for confidence in the functions ability to perform their respective duties. We met this goal with all tiers except for the persistence for the reason mentioned above.

*SPRINT 2  Screenshot 2023-11-01 at 10 02 41 PM  Screenshot 2023-11-01 at 10 05 10 PM
 Screenshot 2023-11-01 at 10 05 07 PM  Screenshot 2023-11-01 at 10 04 59 PM
 Screenshot 2023-11-01 at 10 04 55 PM  Screenshot 2023-11-01 at 10 04 52 PM
 Screenshot 2023-11-01 at 10 04 47 PM  Screenshot 2023-11-01 at 10 04 42 PM
 Screenshot 2023-11-01 at 10 04 37 PM  Screenshot 2023-11-01 at 10 03 32 PM
 Screenshot 2023-11-01 at 10 03 18 PM  Screenshot 2023-11-01 at 10 03 06 PM
 Screenshot 2023-11-01 at 10 02 59 PM  Screenshot 2023-11-01 at 10 02 54 PM
 Screenshot 2023-11-01 at 10 02 49 PM

*SPRINT 4

*NOTE: All code coverage screenshots not shown in Sprint 4 were the same/have not changed from Sprint 2