Food Bank Design Documentation

Team Information

- Team name: ChaosControl
- Team members
  - Owen
  - Jay
  - Ariel
  - Lila
  - Vlad

Executive Summary

The project is a web-based application designed to support a Food Pantry that provides food to people in need.

This platform serves two user types: Managers (Admins) and Helpers. The Manager accounts, representing the organization (in our case, a ficticious company) have the ability to add, remove, and update needs. Helpers, representing volunteers within the organization, have the ability to search for and sign up for needs.

This workflow is optimized through a "basket" system, enabling Helpers to add multiple needs, review them, and commit to their schedule via a checkout process.

Key features include:

- A Cupboard accessible to helpers and managers, editable by managers and usable by users.
- A basket to hold needs for helper access that allows them to view the total cost of all the needs within the basket.
- User Interface to ensure smoother usability for both helpers and managers.
- Authorization for different user types.
- The ability to search for, browse, or otherwise parse needs as a helper, and the ability to add, edit, or remove needs as a manager.

Purpose

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

The purpose of this project is to create a web portal for a food bank that needs certain objectives fulfilled. The most important user group is the helpers who purchase needs for the organization. The goals of the helpers are to add needs to their funding baskets and buy the needs they add to support the food bank.

Glossary and Acronyms

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

Term	Definition

Term	Definition
SPA	Single Page
MVP	Minimum Viable Product
DAO	Data Access Object
MVVM	Model-View-ViewModel
CRUD	Create, Read, Update, Delete

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 MVP will be an online portal for a food bank to request certain items to be purchased by helpers to restock the food bank. It will allow helpers to add needs to a funding basket and checkout with those needs to purchase them. It will also allow managers to manage a cupboard of needs by adding, removing, and modifying needs in the cupboard.

MVP Features

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

Epic: Populate and Manage the Cupboard (admin):

- As an admin, I want to manage the cupboard of needs so that helpers can access accurate and up to date information.
- Admins are able to see list of needs, add, remove, and edit a need, and authorize with the system

Epic: Basket (helper):

- As a helper, I want to manage my funding basket so that I can select and fund multiple needs efficiently.
- Helpers are able to browse and search for needs, add and remove needs to basket, view total cost of their basket, and checkout / fund selected needs.

Epic: Search Filter:

- I want the ability to sort through my cupboard using a search filter to ensure I can quickly find the needs I would like.
- Helpers and admins are able to filter by price (increasing or decreasing).

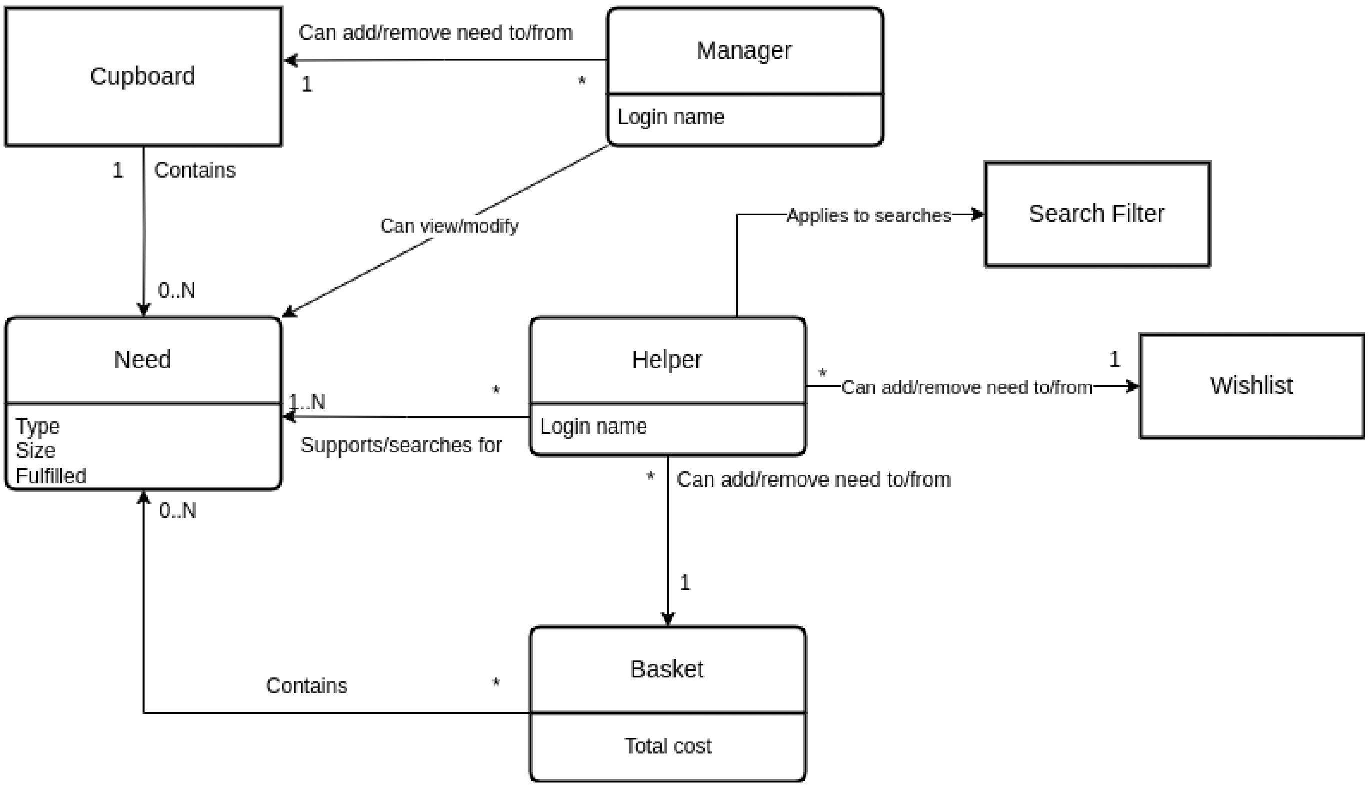
Enhancements

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

- Adding password verification to ensure better safety practices and prevent from others logging into accounts that aren't theirs.
- Adding a 'filter by price' to the cupboard to ensure better sorting for users.
- UI updates and runability to ensure a smoother user experience.
- Leaderboard

Application Domain

This section describes the application domain.



> ***[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.*

The domain for this project mainly consists of managers and helpers and the operations they can perform on baskets and cupboards that contain needs.

The cupboard is the central storage for the needs of the organization. In this project, the needs are food items that a food bank is in need of. Managers oversee the cupboard and add, remove, and update needs when necessary.

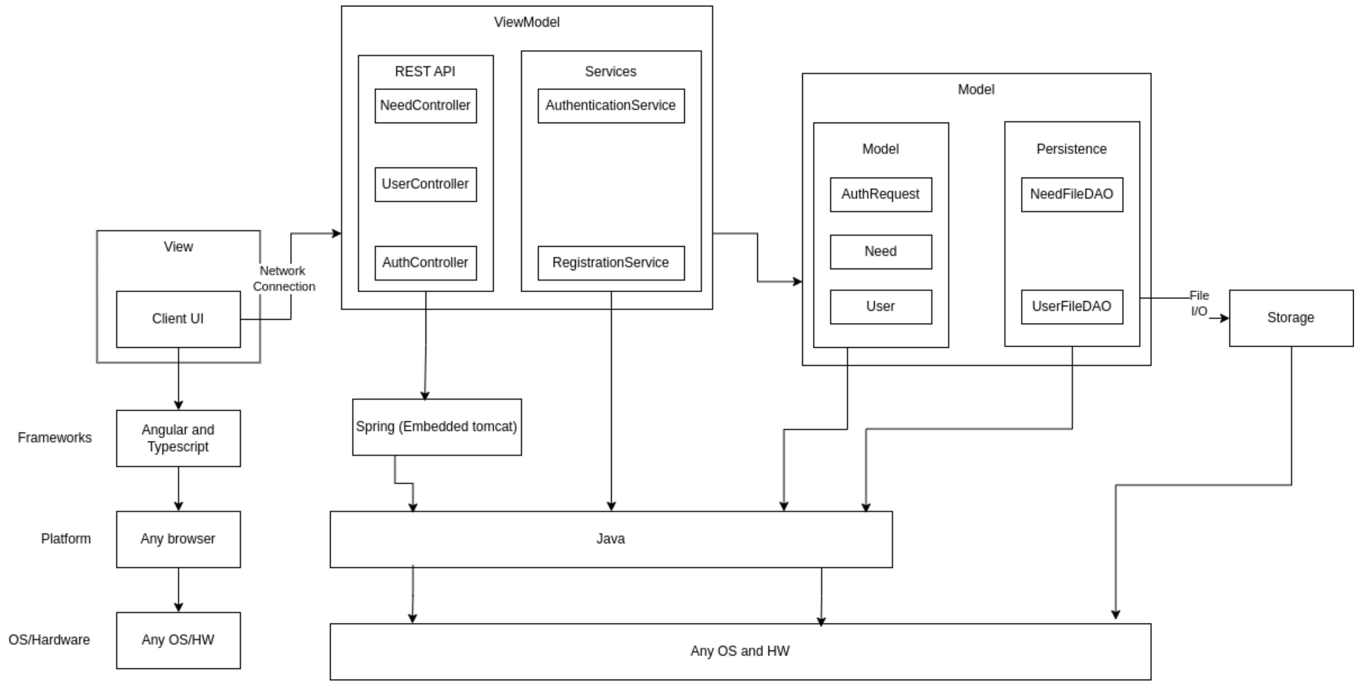
A basket belongs to a helper to keep track of the food items they plan to buy for the food bank. A user has access to the cupboard so they can add the food items they wish to buy to their basket.

Architecture and Design

This application uses a REST API to keep track of all user information and needs information. The API uses an Angular user interface, providing both a helper and manager view.

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.



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/navigation in the web application. (Add low-fidelity mockups prior to initiating your **[Sprint 2]** work so you have a good idea of the user interactions.) Eventually replace with representative screen shots of your high-fidelity results as these become available and finally include future recommendations improvement recommendations for your **[Sprint 4]**)*

When a user navigates to the site, they are greeted with a login prompt. They can either choose to log in with a username and password, or register as a new user. Once logged in, they will see the need cupboard for the organization where they can click on needs to view their details such as name, price, and the food group they belong to. When viewing the need as a helper, there will be options to return to the cupboard, or add the need to their funding basket. As a manager, they will have the options to return to the cupboard, or remove the need from the cupboard. When a helper views their basket, a table of all the needs in the basket are shown as well as the total cost of the basket.

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.*

The View tier is primarily concerned with presentation and user interaction. Components within this tier all function in their own way, some with forms, events, and even subscriptions. Components that need access to our data always delegate their fetching and business logic to the ViewModel tier, where our services sit.

1. Navigation (AppComponent)

- Responsible for authentication state and rendering either Auth pages or the User pages. User pages like the BasketComponent are dependent on who is authenticated and will not be the same for any two distinct users.

2. Page-Level Views (LoginComponent, RegisterComponent, CupboardComponent, BasketComponent, CreateComponent)

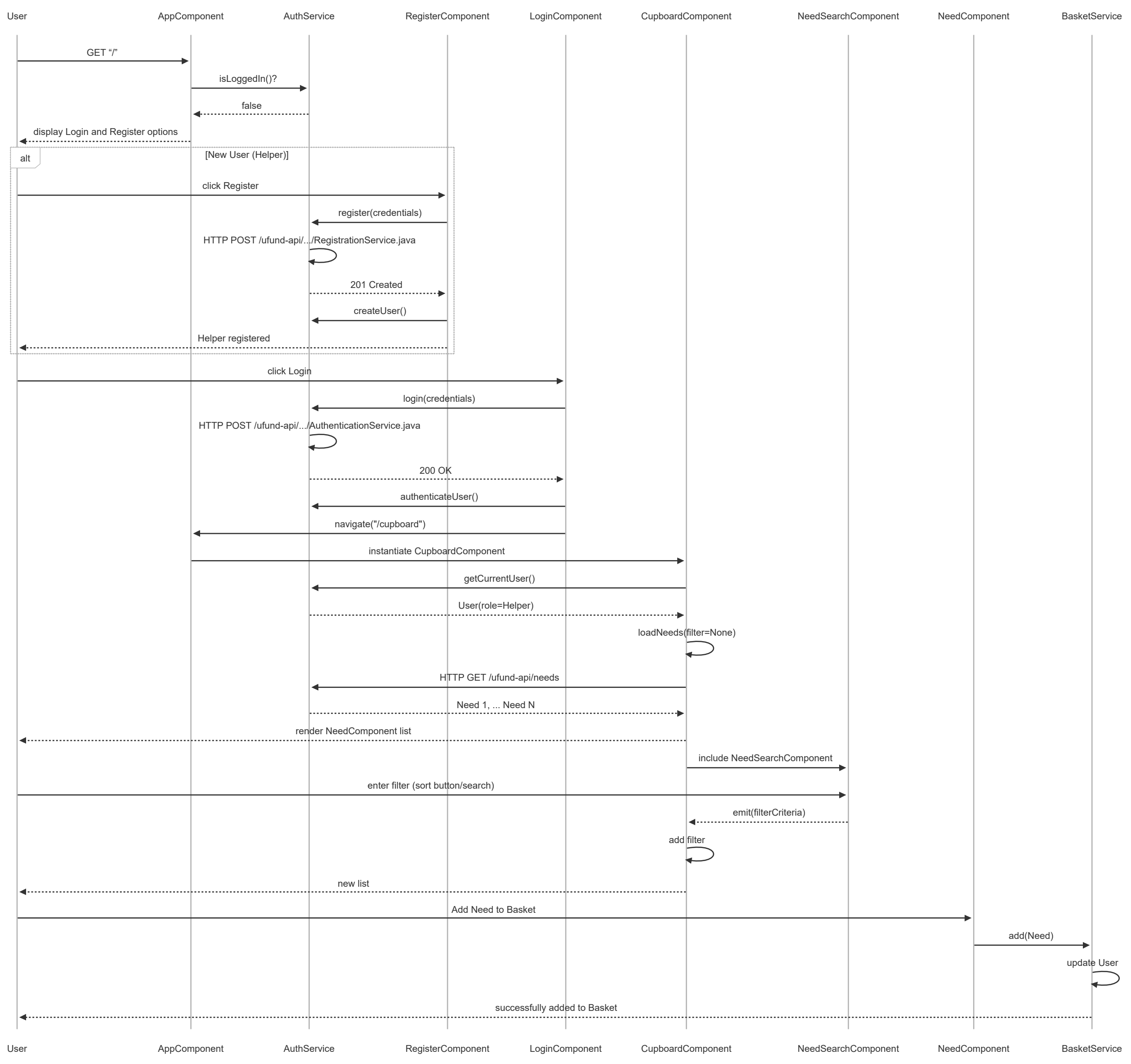
- Responsible for making service calls and returning accurate Need, Basket, and Cupboard information. This includes but is not limited to: status of current Needs, current User Basket, current Cupboard, current filter on the Needs.

3. Need Views (NeedComponent, NeedSearchComponent)

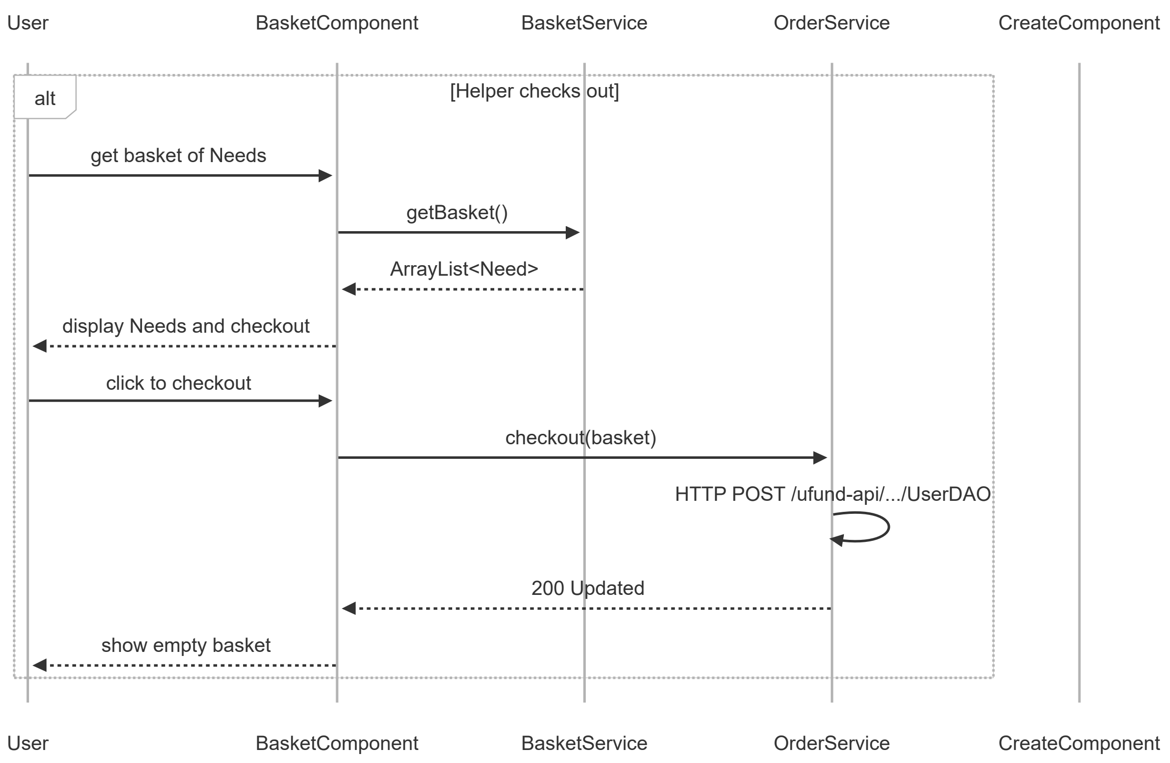
- Responsible for loading in our current Needs and allowing the proper querying of Needs by their names and prices.

> ***[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.*

In Chaos Control, an unauthenticated user loads the App component and sees a Login and Register option. They register appropriately. The Helper logs in to the Cupboard component. They search and filter with the NeedSearch, find an item they are interested in, and add it to their Cupboard.



At any time, if the Helper clicks to check out, the Basket component loads with an order summary and an option to checkout. If a Manager/Admin logs in, they can use the Create component to add a new Need. It will then be visible to Helpers.



> ***[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.*

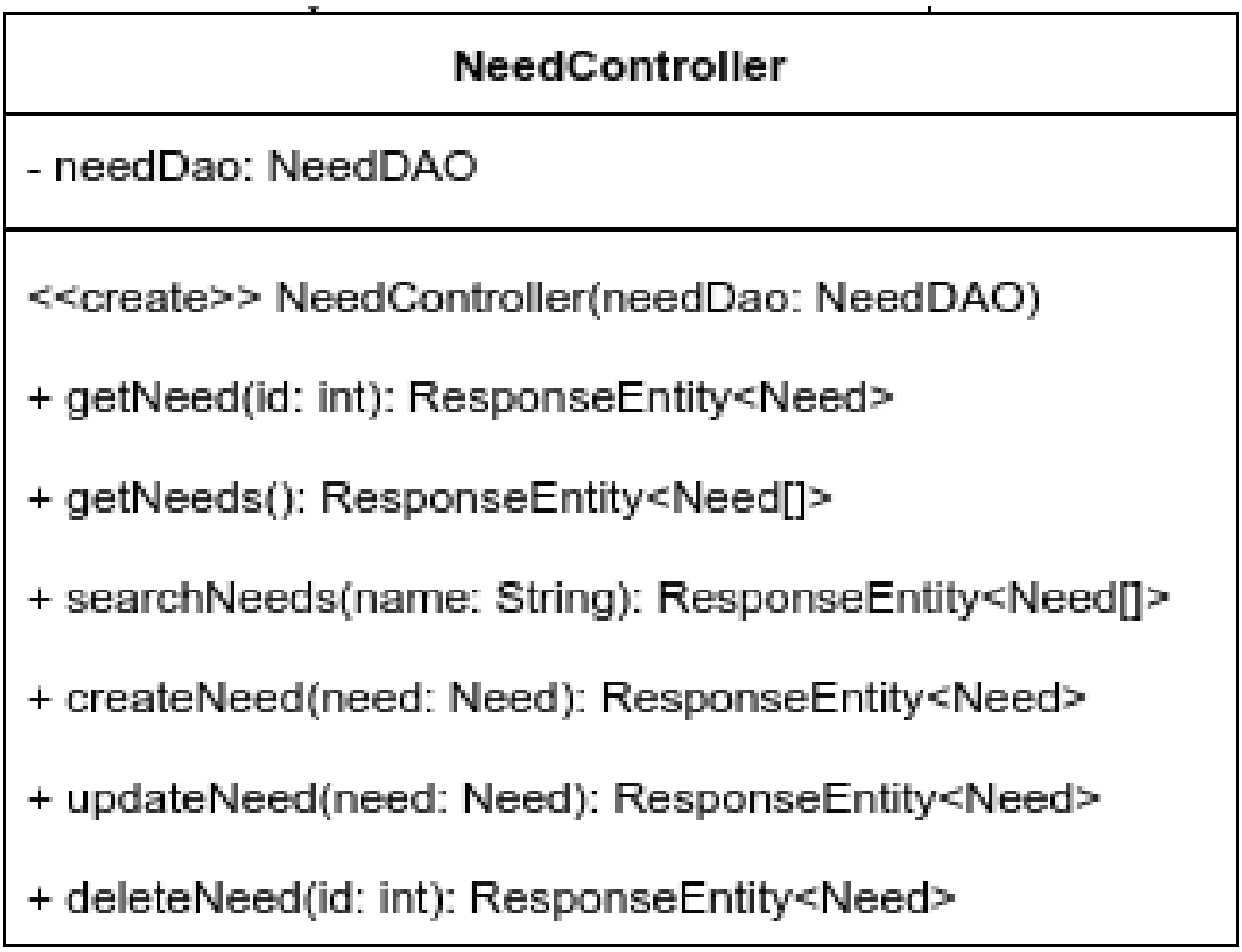
ViewModel Tier

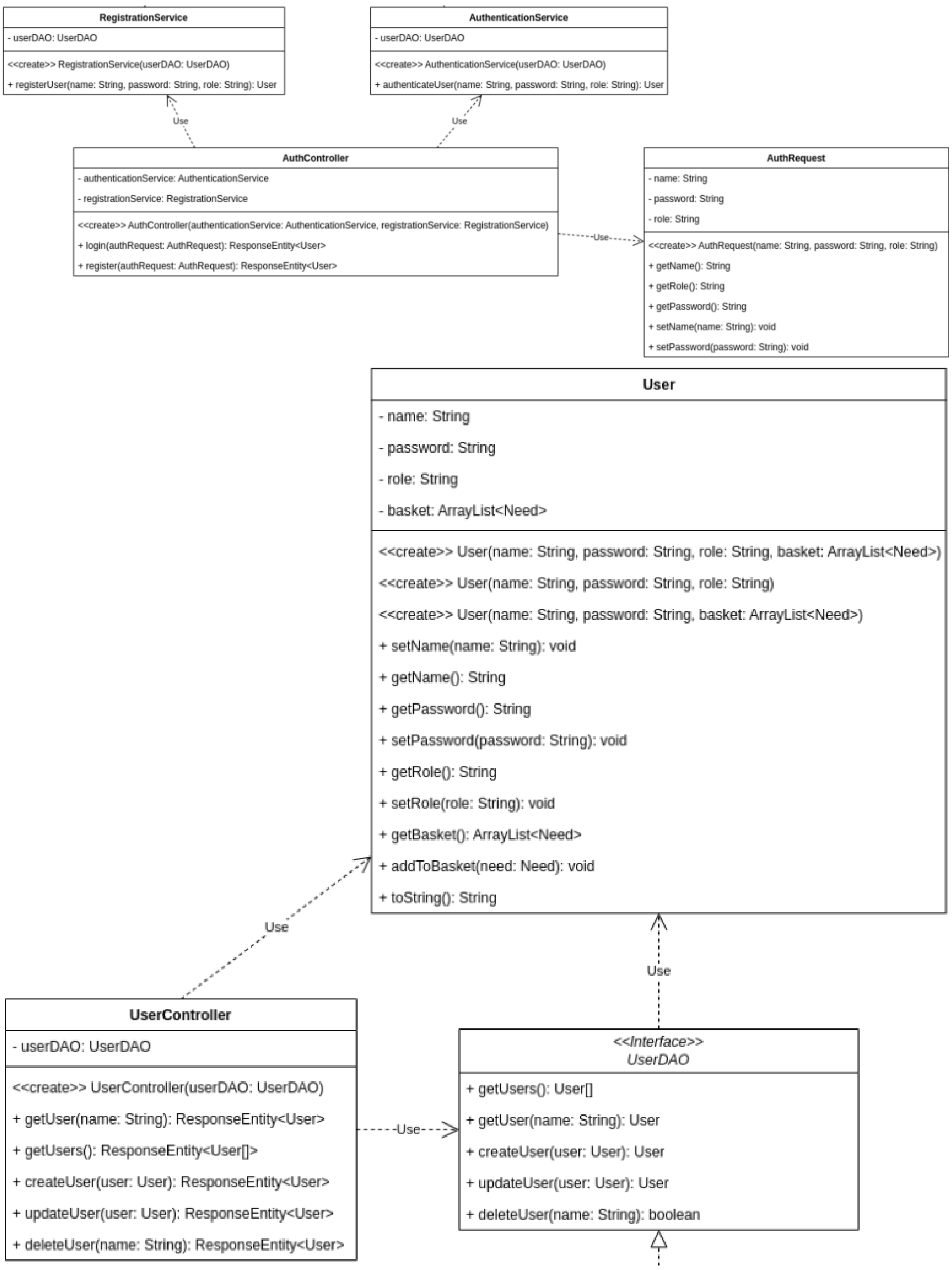
The ViewModel tier provides the application’s interface for client interaction and business logic, bridging the model (data) with the controller (API endpoints):

```
1.  Controllers:
• NeedController handles HTTP requests related to Need entities.
  It uses REST endpoints (GET, POST, PUT, DELETE) for CRUD operations and return
appropriate ResponseEntity objects, managing HTTP status codes based on the
success or failure of each operation.
• UserController handles HTTP requests related to users.
  It allows you to perform CRUD operations to manage the users of the system.
• AuthController allows users to either login to the system, or register as a new
user.
```

> ***[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 associations (connections) between classes, and critical attributes and methods. (****Be sure*** *to revisit the Static* ***UML Review Sheet*** *to ensure your class diagrams are using correct format and syntax.)*


NeedController
- needDao: NeedDAO
<<create>> NeedController(needDao: NeedDAO)
+ getNeed(id: int): ResponseEntity<Need>
+ getNeeds(): ResponseEntity<Need[]>
+ searchNeeds(name: String): ResponseEntity<Need[]>
+ createNeed(need: Need): ResponseEntity<Need>
+ updateNeed(need: Need): ResponseEntity<Need>
+ deleteNeed(id: int): ResponseEntity<Need>

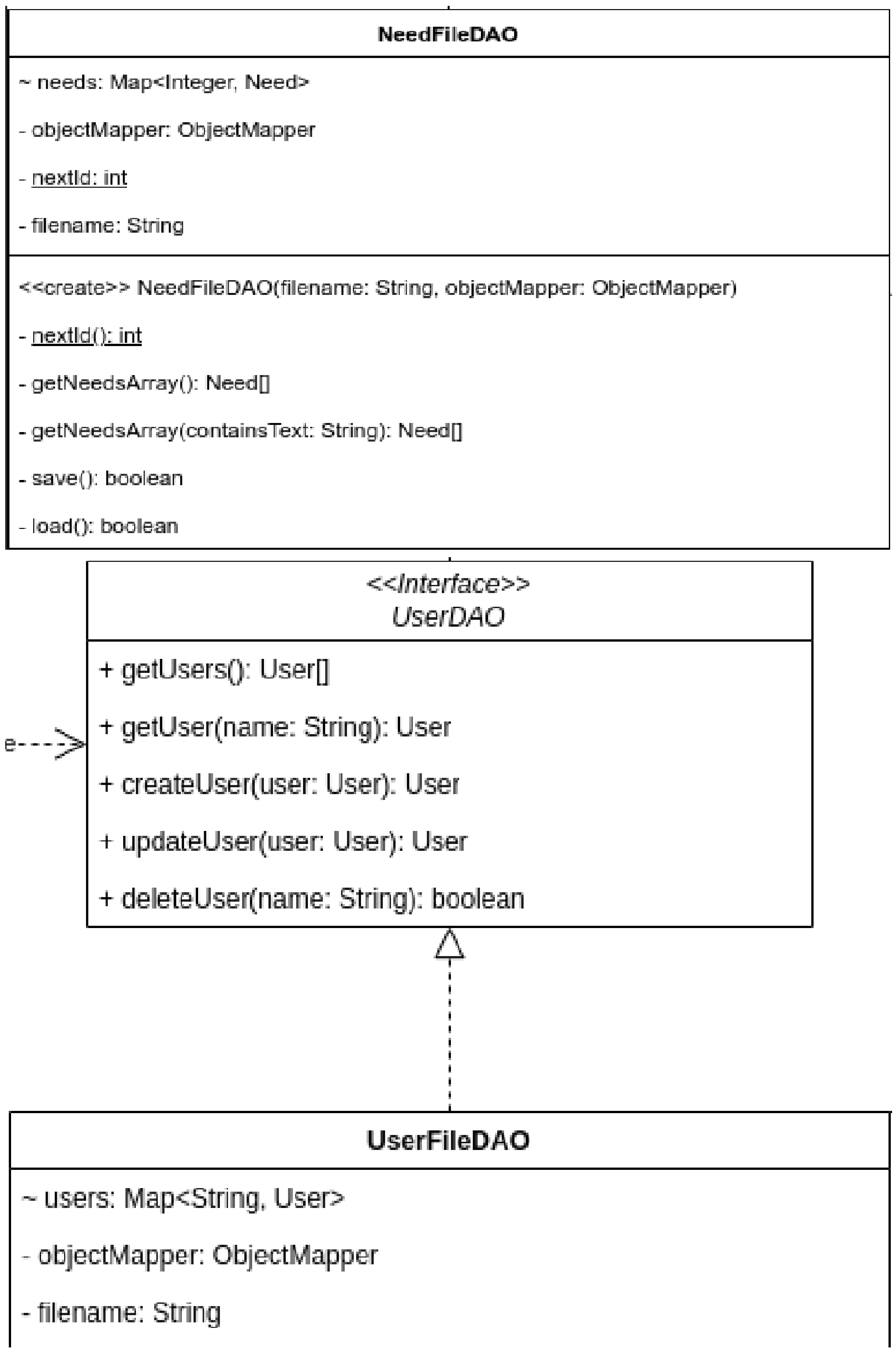

RegistrationService
- userDAO: UserDAO
<<create>> RegistrationService(userDAO: UserDAO)
+ registerUser(name: String, password: String, role: String): User
AuthenticationService
- userDAO: UserDAO
<<create>> AuthenticationService(userDAO: UserDAO)
+ authenticateUser(name: String, password: String, role: String): User
Use
Use
AuthController
- authenticationService: AuthenticationService
- registrationService: RegistrationService
<<create>> AuthController(authenticationService: AuthenticationService, registrationService: RegistrationService)
+ login(authRequest: AuthRequest): ResponseEntity<User>
+ register(authRequest: AuthRequest): ResponseEntity<User>
Use
AuthRequest
- name: String
- password: String
- role: String
<<create>> AuthRequest(name: String, password: String, role: String)
+ getName(): String
+ getRole(): String
+ getPassword(): String
+ setName(name: String): void
+ setPassword(password: String): void
User
- name: String
- password: String
- role: String
- basket: ArrayList<Need>
<<create>> User(name: String, password: String, role: String, basket: ArrayList<Need>)
<<create>> User(name: String, password: String, role: String)
<<create>> User(name: String, password: String, basket: ArrayList<Need>)
+ setName(name: String): void
+ getName(): String
+ getPassword(): String
+ setPassword(password: String): void
+ getRole(): String
+ setRole(role: String): void
+ getBasket(): ArrayList<Need>
+ addToBasket(need: Need): void
+ toString(): String
Use
Use
UserController
- userDAO: UserDAO
<<create>> UserController(userDAO: UserDAO)
+ getUser(name: String): ResponseEntity<User>
+ getUsers(): ResponseEntity<User[]>
+ createUser(user: User): ResponseEntity<User>
+ updateUser(user: User): ResponseEntity<User>
+ deleteUser(name: String): ResponseEntity<User>
Use
<<Interface>>
UserDAO
+ getUsers(): User[]
+ getUser(name: String): User
+ createUser(user: User): User
+ updateUser(user: User): User
+ deleteUser(name: String): boolean

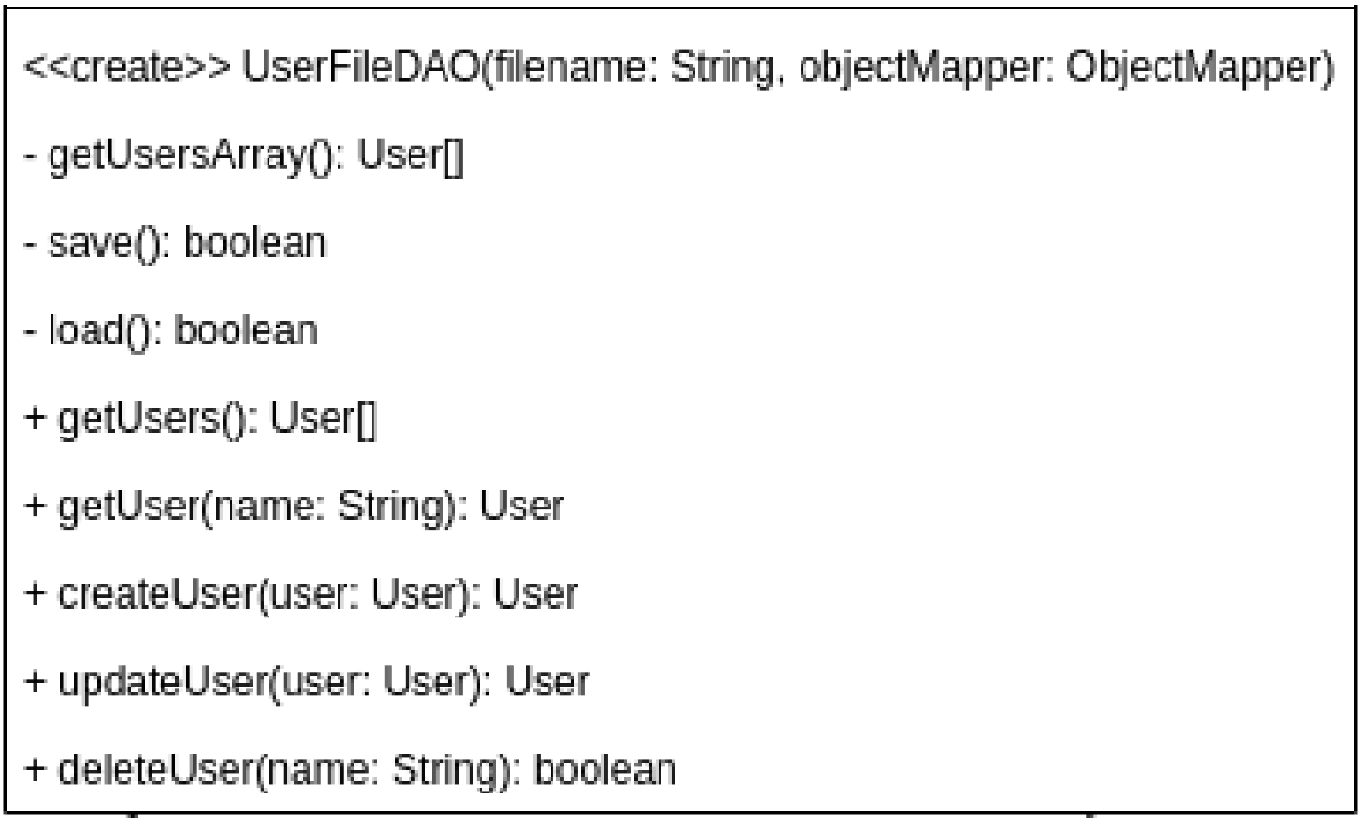
Model Tier

The model tier provides the interface for business logic and persistence. This bridges the gap between storage and the model.

```
1.  Data Access Objects:
• NeedFileDAO provides the methods for interacting with the need storage.
There are methods to support creating, reading, updating, and deleting needs from
storage.
• UserFileDAO provides the methods for managing the storage of user accounts.
  It supports retrieving all or single users, creating users, updating users, and
deleting users.
```

> ***[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 associations (connections) between classes, and critical attributes and methods. (**Be sure** to revisit the Static* ***UML Review Sheet*** *to ensure your class diagrams are using correct format and syntax.)*


NeedFileDAO
~ needs: Map<Integer, Need>
- objectMapper: ObjectMapper
- nextId: int
- filename: String
<<create>> NeedFileDAO(filename: String, objectMapper: ObjectMapper)
- nextId(): int
- getNeedsArray(): Need[]
- getNeedsArray(containsText: String): Need[]
- save(): boolean
- load(): boolean
<<Interface>>
UserDAO
+ getUsers(): User[]
+ getUser(name: String): User
+ createUser(user: User): User
+ updateUser(user: User): User
+ deleteUser(name: String): boolean
e
UserFileDAO
~ users: Map<String, User>
- objectMapper: ObjectMapper
- filename: String


<<create>> UserFileDAO(filename: String, objectMapper: ObjectMapper)

- getUsersArray(): User[]

- save(): boolean

- load(): boolean

+ getUsers(): User[]

+ getUser(name: String): User

+ createUser(user: User): User

+ updateUser(user: User): User

+ deleteUser(name: String): boolean

OO Design Principles

Abstraction

We have applied abstraction in our design by creating an interface for our data access object that our controller uses for file I/O operations.

Low Coupling

Low coupling was applied in our design by utilizing dependency injection in our NeedController class. Instead of instantiating a NeedDAO in NeedController, we pass one in through it's constructor.

Encapsulation

We applied encapsulation in our design by grouping related methods and data into classes. The Need class has methods and data needed to represent the state and behavior of a need, for example.

Dependency Injection

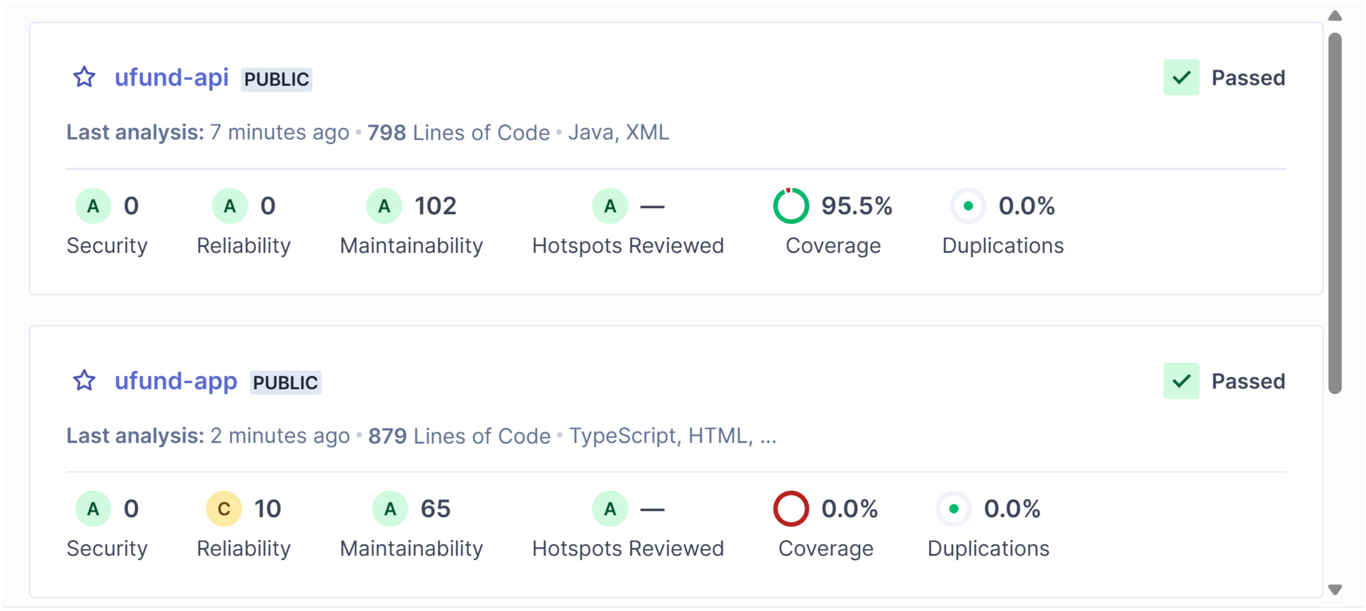
Dependency injection is applied in our design with the Angular services we created. With the authentication service, we've injected it into the root of the app so other components will use the same instance of the service instead of creating new instances.

> ***[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 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.*



ufund-api PUBLIC — Passed

Last analysis: 7 minutes ago • 798 Lines of Code • Java, XML

Security	Reliability	Maintainability	Hotspots Reviewed	Coverage	Duplications
A 0	A 0	A 102	A —	95.5%	0.0%

ufund-app PUBLIC — Passed

Last analysis: 2 minutes ago • 879 Lines of Code • TypeScript, HTML, ...

Security	Reliability	Maintainability	Hotspots Reviewed	Coverage	Duplications
A 0	C 10	A 65	A —	0.0%	0.0%

Our html / typescript had some reliability issues under static analysis, which are outlined below:

- Some of the files contained declared imports that were never used. While this is not harmful, it does add unnecessary clutter that could lead to confusion in the future. For this, we removed the redundant imports.

- We had a couple strings hardcoded directly into HTML templates. This reduces flexibility in the future. For this, we changed these strings into constants.

- Some functions were flagged for being too complex, with nested conditionals and loops. For this, we refractored larger functions into smaller, more modular helper functions.

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

If given more time, our team would focus on the following refactoring and design improvements:

- Improved UI, the user interface we have works very well, but could always be updated to be even more geared for users
- Receipts for User Checkout - right now we have that when a user checks out, a message pops up saying checkout successful and all items from their basket are removed. A receipt could be useful if the user wants to keep track of what exactly they have checked out in the past.
- Filter searches - we have a filter for items by increasing and decreasing price. We could also add filters by food group or viewed status, etc.

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.*

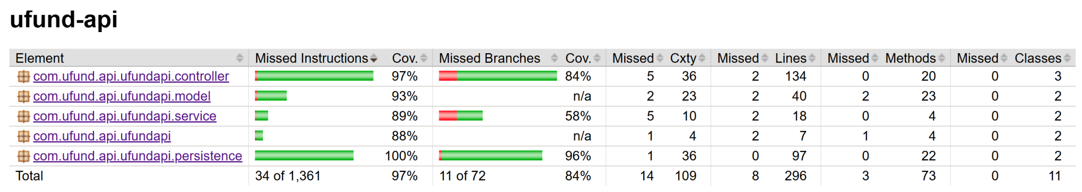
All of our user stories for sprint 4 have been tested and their functionality has been verified to match the acceptance criteria. In total, we tested 8 different user stories each with at least 2 acceptance criterion.

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.*

The goal was to aim for around 95% code coverage. Overall, we managed to reach 97%. Our coverage targets for Persistence, Controller, and Model tiers were 95%. These values were selected because a 95% code coverage means that most branches and test cases have been exhausted with maybe a few outliers that were unable to properly be tested for. With 95%, our code and functionality proves to be working as expected. For persistence, we reached 100%, controller - 97%, and model - 93%. Our testing for the model tier could have been a bit better, but it still managed to get really close to our target, as well as achieving our overall target.

> ***[Sprint 2, 3 & 4] Include images of your code coverage report.*** *If there are any anomalies, discuss those.*



ufund-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi.controller		97%		84%	5	36	2	134	0	20	0	3
com.ufund.api.ufundapi.model		93%		n/a	2	23	2	40	2	23	0	2
com.ufund.api.ufundapi.service		89%		58%	5	10	2	18	0	4	0	2
com.ufund.api.ufundapi		88%		n/a	1	4	2	7	1	4	0	2
com.ufund.api.ufundapi.persistence		100%		96%	1	36	0	97	0	22	0	2
Total	34 of 1,361	97%	11 of 72	84%	14	109	8	296	3	73	0	11

Ongoing Rationale

> ***[Sprint 1, 2, 3 & 4]*** *Throughout the project, provide a time stamp **(yyyy/mm/dd): Sprint # and description** of any **mayor** team decisions or design milestones/changes and corresponding justification.*

Sprint 1

- **2025-02-02**: - Implementation of design from Heroes API base.
- **2025-02-23**: Changed base meeting dates to Tuesdays [5:00pm-6:00pm] and Fridays [3:00pm-4:30pm]

Sprint 2

- **2025/03/05**: Getting the UI to interface with the backend.
- **2025/03/15**: Finished acceptance testing.

Sprint 3

- **2025/04/03**: Perform user acceptance testing with other group.
- **2025/04/07**: Hash out sprint 3 demo plan.

Sprint 4

- **2025/04/10**: Short meeting to discuss Sprint 4 plans.
- **2025/04/11**: Work on DesignDoc updates, ideation of Epics and Enhancements.