Software Engineering 14:332:452

Group #9

Full Report #2: ChefPal

Submission Date: 03/23/2021

GitHub

Team Members:

Shahir Ghani
Malena Bashar
Dymytriy Zyunkin
Daniel Samojlik
Amanda Phan

Michael Fong
Malak Khalifa
Aswathy Aji
Azim Khan
Nirav Patel

Table of Contents

Individual Breakdowns 3

1) Analysis and Domain Modeling 4

- **a. Conceptual Model** 4
  - **i. Concept Definition** 4
  - **ii. Association Definitions** 5
  - **iii. Attribute Definitions** 6
  - **iv. Traceability Matrix** 7
- **b. System Operation Contracts** 8
- **c. Data Model and Persistent Data Storage** 12
- **d. Mathematical Model** 12

2) Interaction Diagrams 13

3) Class Diagram and Interface Specification 21

- **a. Class Diagram** 21
- **b. Data Types and Operation Signatures** 21
- **c. Traceability Matrix** 24

4) Algorithms and Data Structures 27

- **a. Algorithms** 27
- **b. Data Structures** 27
- **c. Concurrency** 27

5) User Interface Design and Implementation 27

6) Design of Test 28

7) Project Management 29

- **a. Merging the Contributions from Individual Team Members** 29
- **b. Project Coordination and Progress Report** 29
- **c. Plan of Work** 30
- **d. Breakdown of Responsibilities** 31

8) References 32

Individual Breakdowns

	Amanda	Aswathy	Azim	Daniel	Dymytriy	Malak	Malena	Michael	Nirav	Shahir
Concept definitions	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Association definitions	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Attribute definitions	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Traceability matrix	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
System Operation Contracts	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Data Model and Persistent Data Storage	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Mathematical Model	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Interaction Diagrams	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Class Diagram	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Data Types and Operation Signatures	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Traceability Matrix	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Project Management	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Algorithms	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Data Structures	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Concurrency	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
UI Design and Implementation	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Design of Tests	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Project Management and Plan of Work	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
References	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

1) Analysis and Domain Modeling

a. Conceptual Model

i. Concept Definition

Responsibility	**Type (D- Doing, K- Knowing, N- Neither)**	**Concept**
R1: Display user registration or log in information	D	Interface
R2: Verify user's username and password entered	D	Authenticator
R3: Display distance and dietary preference selection list	D	Interface
R4: Display a list of all possible ingredients user can select	D	Interface
R5: Cross-reference database for available recipes	D	Controller
R6: Receive selected ingredients	C	Communicator
R7: Send a list of possible recipes using selected ingredients	C	Communicator
R8: Get keyword inputted in search bar to find recipes	D	Controller
R9: Display all possible recipes by keyword search	D	Interface
R10: Display filters for user to select	D	Interface
R11: Receive selected filters	C	Communicator
R12: Update list from selected filters	D	Controller
R13: Display updated list	D	Interface
R14: Store saved recipes for future use	K	Database
R15: Get user's current location	D	Controller
R16: Get nearby grocery stores with missing ingredients	D	Controller
R17: Display grocery stores and their location and hours	D	Interface
R18: Store user's login credentials	K	Database
R19: Store user's password hash	K	Database
R20: Store existing ingredients	K	Database

ii. Association Definitions

Concept Pair	Association Description	Association Name
Interface → Authenticator	Interface sends input data to Checker for verification	Check information
Interface → Controller	Interface sends user data, Controller reads input data	Read input
Controller → Interface	Controller gets requests from database and sends it to interface	Send result
Authenticator → Controller	Checker sends the validated information back to controller	Send validation message
Communicator → Interface	Communicator sends messages and interface displays it to user	Send message
Interface → Communicator	User sends in a message and communicator sends it to the system	Get/receive message
Controller ↔ Communicator	Controller gets requests from communicator and responds, communicator sends it back to system	Get a request and send an answer
Communicator ↔ Database	Communicator sends a request to database for stored info and sends response back	Send request, receive a response and send response

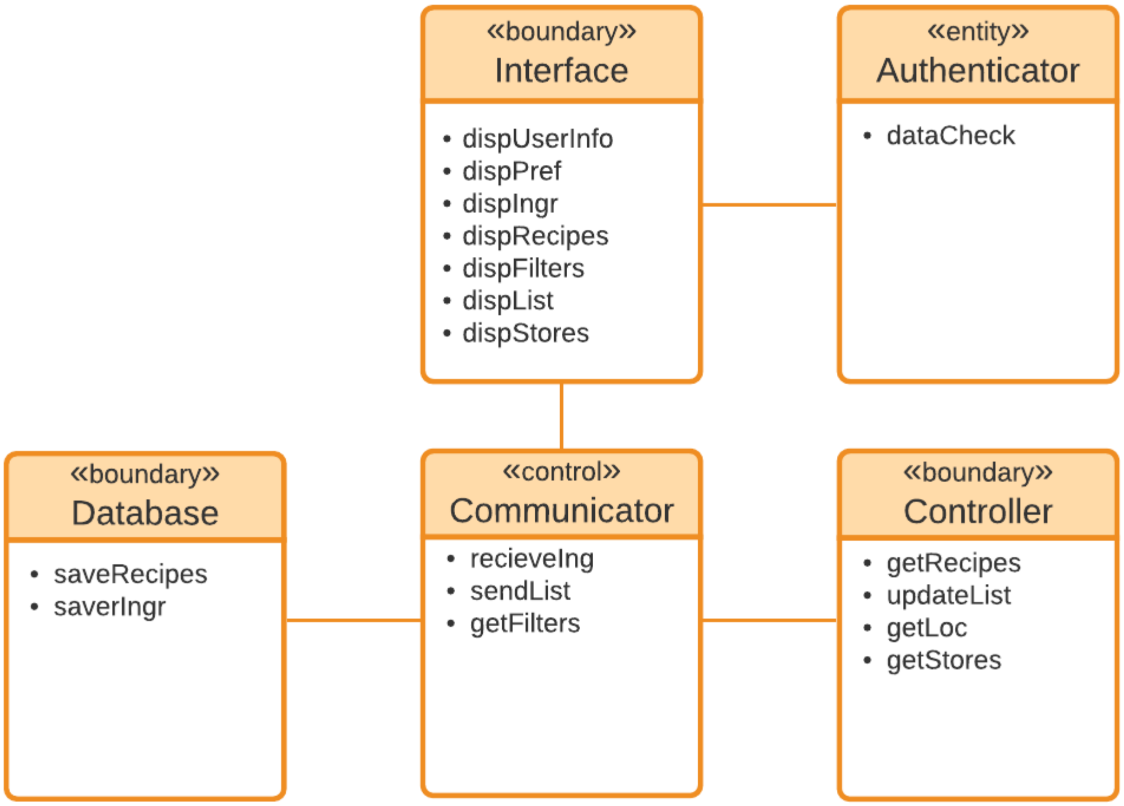
iii. Attribute Definitions

Responsibility	**Attribute**	**Concept**
R1: Display user registration or log in information	dispUserInfo	Interface
R2: Verify user’s username and password entered	dataCheck	Authenticator
R3: Display distance and dietary preference selection list	dispPref	Interface
R4: Display a list of all possible ingredients user can select	dispIngr	Interface
R5: Cross-reference database for available recipes	getRecipes	Controller
R6: Receive selected ingredients	receiveIng	Communicator
R7: Send a list of possible recipes using selected ingredients to the user	sendList	Communicator
R8: Get keyword inputted in search bar to find recipes	getKeyword	Controller
R9: Display all possible recipes by keyword search	dispRecipes	Interface
R10: Display filters for user to select	dispFilters	Interface
R11: Receive selected filters	getFilters	Communicator
R12: Update list from selected filters	updateList	Controller
R13: Display updated list	dispList	Interface
R14: Store saved recipes for future use	saveRecipes	Database
R15: Get user’s current location	getLoc	Controller
R16: Get nearby grocery stores with missing ingredients	getStores	Controller
R17: Display grocery stores and their location and hours	dispStores	Interface
R18: Store user’s login credentials	saveUserCred	Database
R19: Store user’s password hash	saveUserPass	Database
R20: Store user’s existing ingredients	saveIngr	Database

iv. Traceability Matrix

The table below depicts how the use cases map to the domain concepts

Concept	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8	UC-9	UC-10	UC-11
Interface	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Authenticator	✔		✔								
Controller					✔	✔	✔			✔	
Communicator		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Database		✔	✔	✔				✔	✔		



b. <u>System Operation Contracts</u>

Operation:	Registration
Use Case:	UC-1
Responsibilities:	● Use the database system to register a new user's username and password
Expectations:	● The new user is stored into the database system
Preconditions:	● The user has to be a first time user ● The user doesn't have an account
Postconditions:	● The user can select their dietary preference ● Username and password will be saved to database

Operation:	Primary Preference Selection
Use Case:	UC-2
Responsibilities:	● Use the database system to register a new user's dietary information, restrictions, and distance preference to their profile
Expectations:	● The user preferences are stored into the database system and to the user's profile
Preconditions:	● User is registered and can now operate the mobile app
Postconditions:	● User can now receive recipe suggestions based on preferences and ingredients

Operation:	Login
Use Case:	UC-3
Responsibilities:	● Firebase authenticates the user’s login information
Expectations:	● The user logs into the system and is able to interact with the app
Preconditions:	● User can login with credentials used to sign up for mobile app
Postconditions:	● User has gained access back to their account ● User can begin searching recipes and make changes to their account

Operation:	Ingredient Selection
Use Case:	UC-4
Responsibilities:	● The database system will store the user’s selected ingredients so that the data can be used to view applicable recipes
Expectations:	● User will be able to view available recipes based off of any of the selected ingredients
Preconditions:	● User has full access and can operate their own account
Postconditions:	● User can enter any preference regarding meal they desire (e.g. meal type, cuisine of meal)

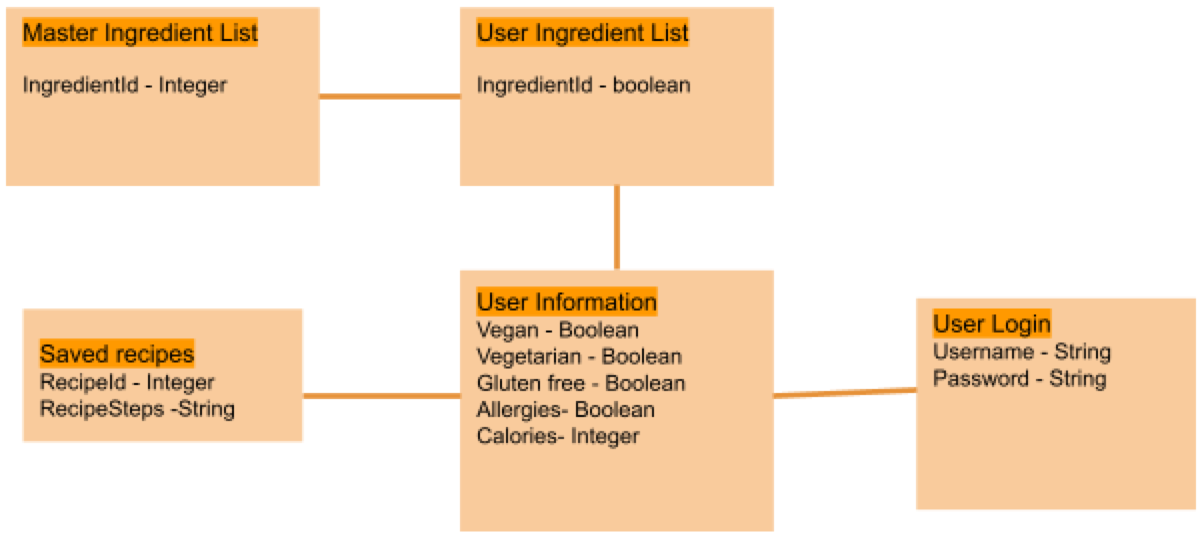
Operation:	Generated Recipes
Use Case:	UC-5
Responsibilities:	● The data from the database are transferred via the Spoonacular API which consolidates applicable recipes for the user to view.
Expectations:	● Users will be able to view a list of generated recipes based on their dietary preferences and selected ingredients
Preconditions:	● User has inputted preferences and dietary needs in order to generate recipes based on current ingredients
Postconditions:	● Recipes will be pulled that detail user's selected preferences and ingredients on hand

Operation:	Filtering System for Generated Recipes
Use Case:	UC-6
Responsibilities:	● Include more user preferences to better their user experience ● To provide the user with more specific and detailed recipes
Expectations:	● User will receive a more specific set of recipes based on their filters
Preconditions:	● User has a generated list of recipes corresponding to their selections
Postconditions:	● User can narrow down results received even further by choosing type of meal, cuisine of meal, calorie count of meal, etc.

Operation:	Recipe Search
Use Case:	UC-7
Responsibilities:	● To find all recipes that are both within the user's dietary preferences and require only (if not, most of) the user's available ingredients
Expectations:	● User will be able to view recipes based on their search
Preconditions:	● User has selected dietary restrictions and preferences ● User has inputted available ingredients
Postconditions:	● User is able to view all available recipes based off of their search

Operation:	Grocery Store Locator
Use Case:	UC-10
Responsibilities:	● To locate grocery stores closest to the user that have the missing ingredients available using mapping API.
Expectations:	● The app will display nearby grocery stores with items that the user will need for their chosen recipe
Preconditions:	● User has indicated current location ● User has shared distance willing to travel to go grocery shopping ● User has selected all current ingredients in possession
Postconditions:	● User has received name and address of grocery store nearest to their current location for missing ingredients of the recipe from the mobile app

c. Data Model and Persistent Data Storage



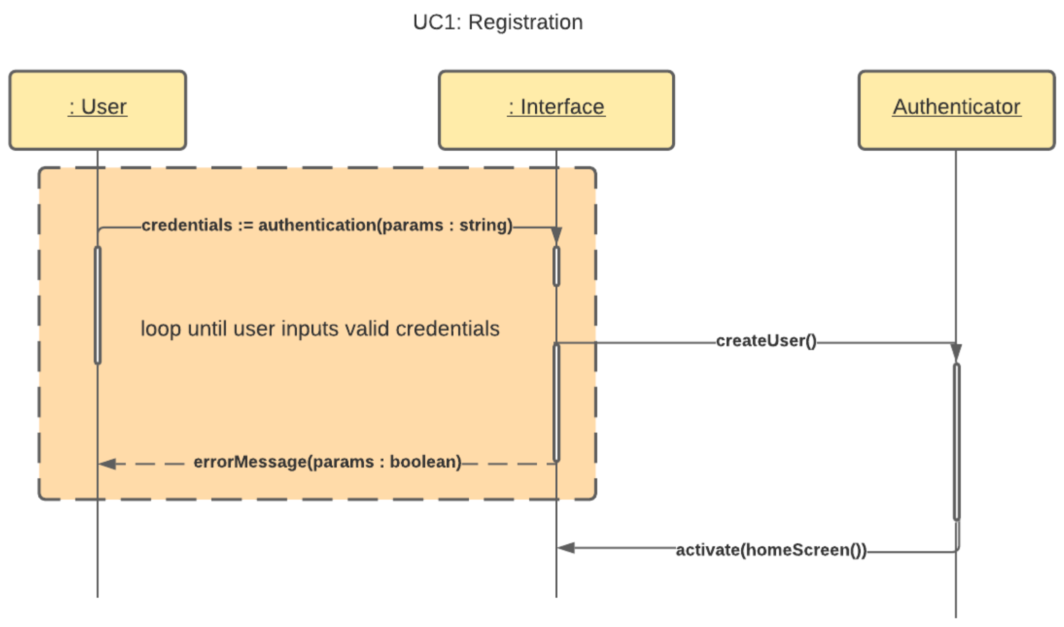
Our mobile application does need to have specific data stored within the database. It stores user information in the database, like dietary restrictions (vegan, vegetarian, gluten free, allergies, etc). It also stores the user's corresponding username and password which secure how their information is saved. In addition, it stores preferences the user might want to sort the recipes by, such as type of cuisine, meal, preparation time and serving size. The ingredients are stored in a master ingredient list as an integer, and if the user has an ingredient in the master ingredient list, it is stored in the user ingredient list as a boolean. This optimizes the storage space used by using the smallest possible data types. The resulting recipes are saved in order of most matched to the user's preferences and optimization of ingredients. Recipes are saved by "liking" them and these can be accessed at a later time via a separate drop-down menu. Users can save current ingredients in their household that include attributes such as name. In addition, the database would have to store the user's liked/saved recipes that they would like to return to the future. All of this information will be stored in a database running on our Firebase server.

d. Mathematical Model

Our mobile application does not use any mathematical model in order for it to operate.

2) Interaction Diagrams

UC1:

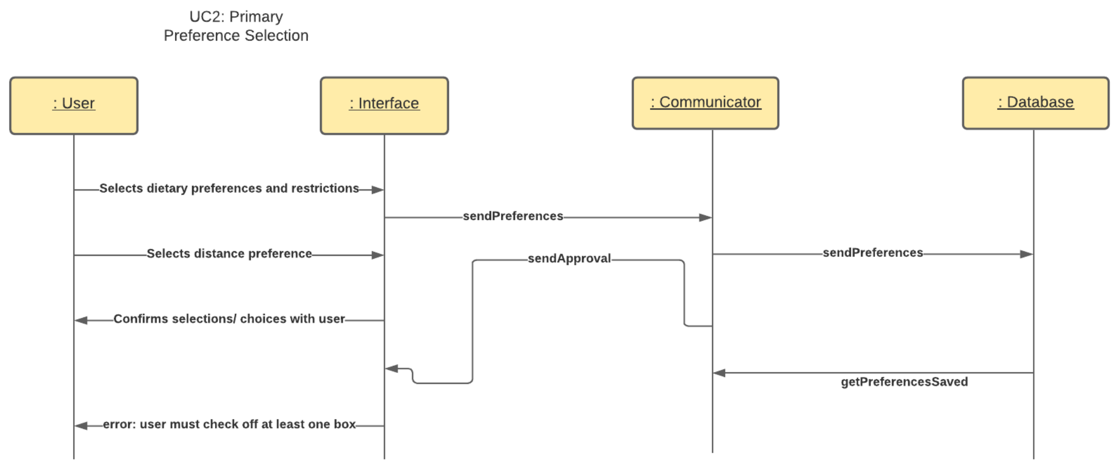

UC1: Registration
: User
: Interface
Authenticator
credentials := authentication(params : string)
loop until user inputs valid credentials
createUser()
errorMessage(params : boolean)
activate(homeScreen())

The above interaction diagram is for UC1: Registration. The user enters their desired username and password through the Interface. The Interface will then send the information to the Authenticator, which will check to see if the entered username is already existent. If it is, then it sends an error message back to the user, prompting them to enter in another username and password combination. If the Authenticator declares that the username and password combination are valid, then the home screen for the application is displayed and the user is able to use the other functionalities of the application.

Design Pattern Used: Published - Subscriber Pattern

The reason we chose to use the Published - Subscriber Pattern is because this use case is event-driven. This pattern separates the publisher and subscriber, where the Authenticator is the publisher. The authenticator notifies the subscriber (in this case the interface and the user) whether the registration has been completed or not. The Authenticator will simply send back the event, and the publisher will decide what to do depending on what the event shows. This way, the publisher and the subscriber are separated, which is known as loose coupling.

UC2:

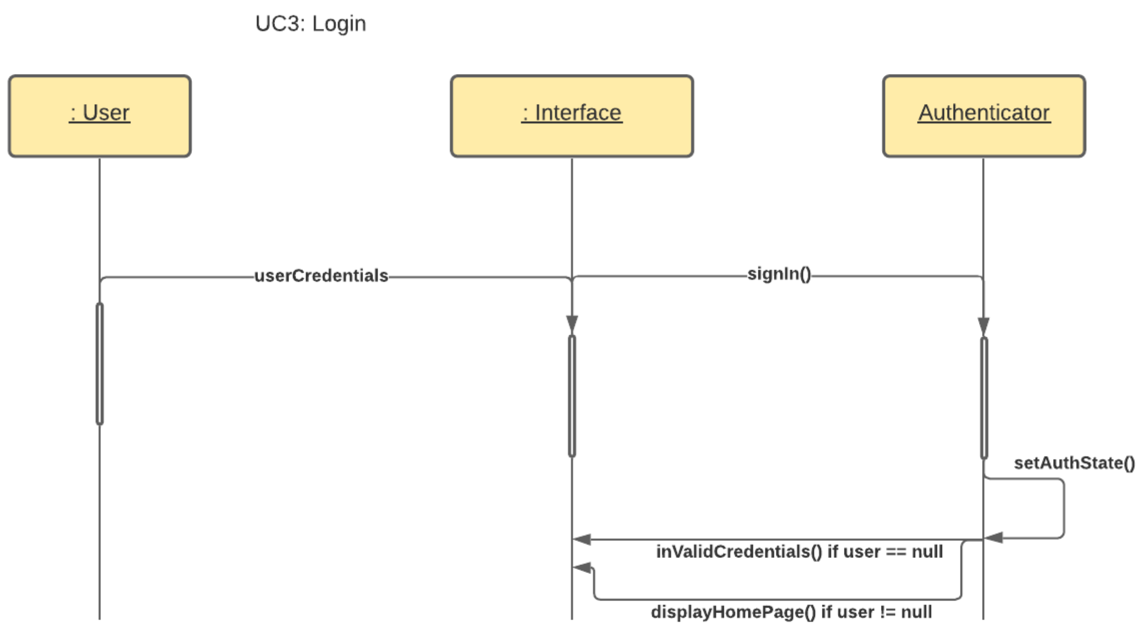

UC2: Primary Preference Selection
: User
: Interface
: Communicator
: Database
Selects dietary preferences and restrictions
sendPreferences
Selects distance preference
sendApproval
sendPreferences
Confirms selections/ choices with user
getPreferencesSaved
error: user must check off at least one box

The above interaction diagram is for UC2: Primary Preference Selector. The user will select dietary preferences and restrictions, such as Vegetarian, Vegan, or additional allergies that they might have. The user will also select their distance preference for grocery stores. These will all be done via the Interface, which will then confirm the user's selections with them. The user's preferences/restrictions will be sent to the Communicator by the Interface, and will be stored there in the Database. The Communicator can also get saved preferences from the Database, and the Communicator can send the approval to the interface. If the user hasn't selected any boxes then an error message will be sent by the Interface to the User.

Design Pattern Used: Indirection Design Pattern

The Indirection Design Pattern is good for this use case because we have a Communicator. This pattern is best used when there is an intermediate object between other objects. This fits the Communicator role description, where the Communicator is between the Database and the Interface. This design also supports the low coupling design pattern, where each component doesn't completely affect another component, and can have more reuse between components.

UC3:

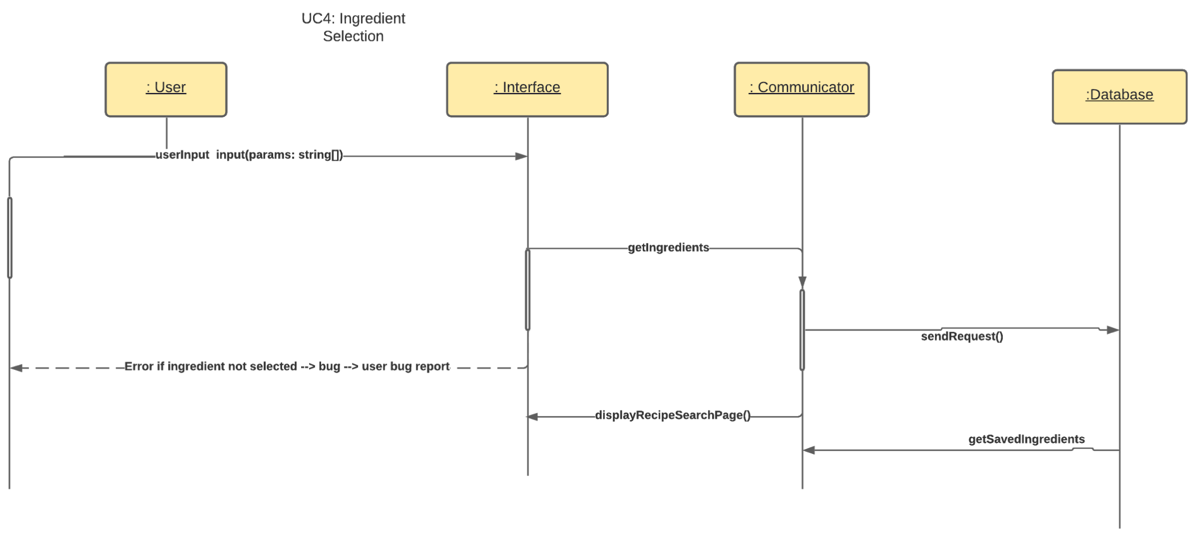


The above interaction diagram is for UC3: Log-in. The user will enter their username and password that was validated back when they first registered through UC1: Registration. The user will input in their credentials via the Interface. The Interface will send the credentials to the Authenticator when the User selects “Sign - In”, and will then set the authentication state depending on whether or not the credentials match those stored or not. If the login was successful then the home screen will display. If the login was not successful, then the user will be told to input their user information again.

Design Pattern Used: Publisher - Subscriber Pattern

The reason we chose this pattern is because it separates the publisher and subscriber. Much like UC1: Registration, the publisher (in this case the Authenticator), will set the state for the event that will be sent to the subscriber (Interface). This will separate the publisher and separator, where the publisher will send the events and the subscriber will determine what to do with that information.

UC4:

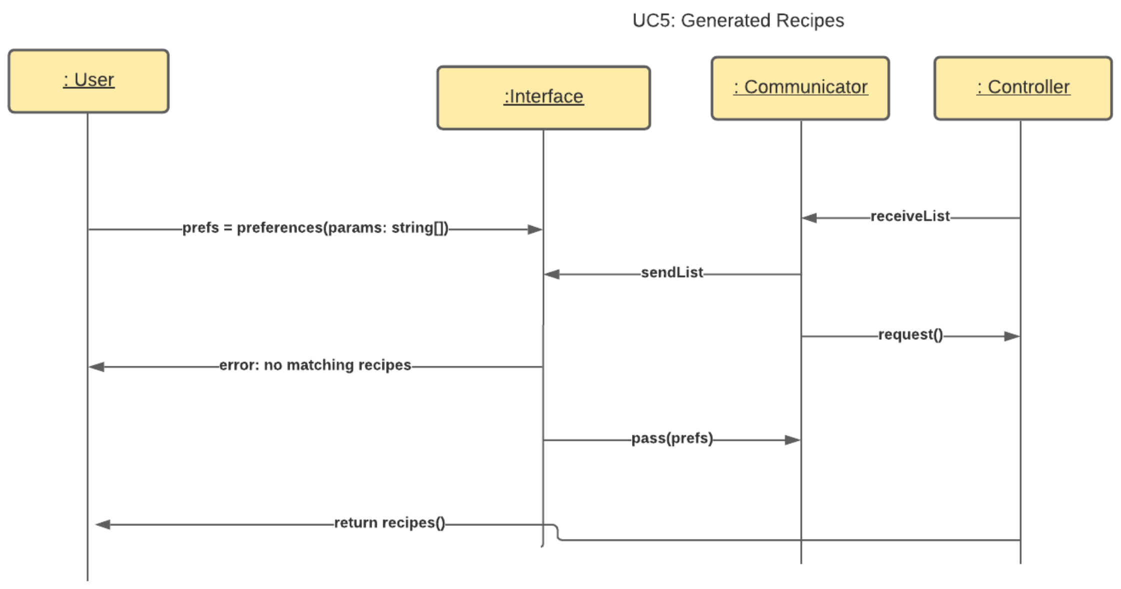

UC4: Ingredient Selection
: User
: Interface
: Communicator
:Database
userInput input(params: string[])
getIngredients
sendRequest()
Error if ingredient not selected --> bug --> user bug report
displayRecipeSearchPage()
getSavedIngredients

The above interaction diagram is for UC4: Ingredient Selection. The user will input their available ingredients which will output a list of recipes based on the imputed ingredients. The Communicator will send the request to the Database, who will send back the saved ingredients to the Communicator. The Communicator will display the recipe search page to the Interface. If no ingredients can be selected/are selected, then an error report is sent.

Design Pattern Used: Indirection Design Pattern

The Indirection Design Pattern is good for this use case because we have a Communicator. This pattern is best used when there is an intermediate object between other objects. This fits the Communicator role description, where the Communicator is between the Database and the Interface. This design also supports the low coupling design pattern, where each component doesn't completely affect another component, and can have more reuse between components.

UC5:

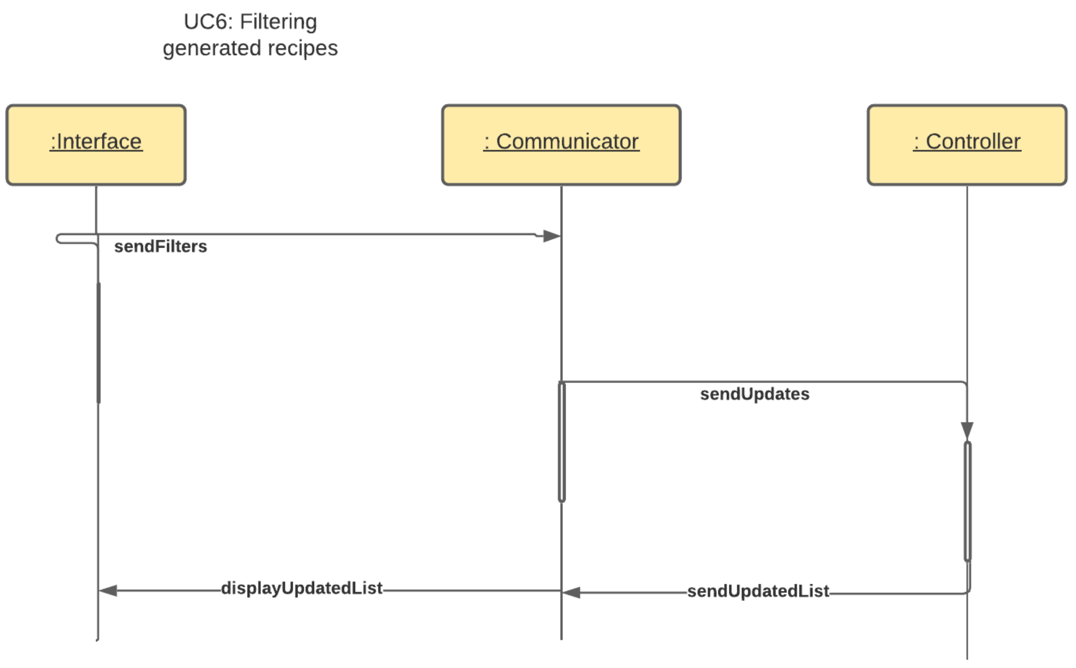


The above interaction diagram is for UC5: Generated Recipes. The user will be able to select and filter out recipes based on their dietary restrictions and current ingredients that they have already inputted into the application. The Communicator will send a request to the Controller, and the Controller will send back the recipes to the Communicator, which will send it to the Interface. If there are no matching recipes, an error message will be shown from the Interface to the User. The Controller will send the recipes to the Interface to be displayed.

Design Pattern Used: Command-based Design Pattern

The Command-based Design Pattern is useful for this design because the Communicator is sending requests to the Controller to access the ingredients so recipes can be generated. The Command-Based design pattern is the pattern where the request is sent as an object, and in this case, it is a JSON object. As such, the information is able to be modified and sent to the interface as it needs to.

UC6:



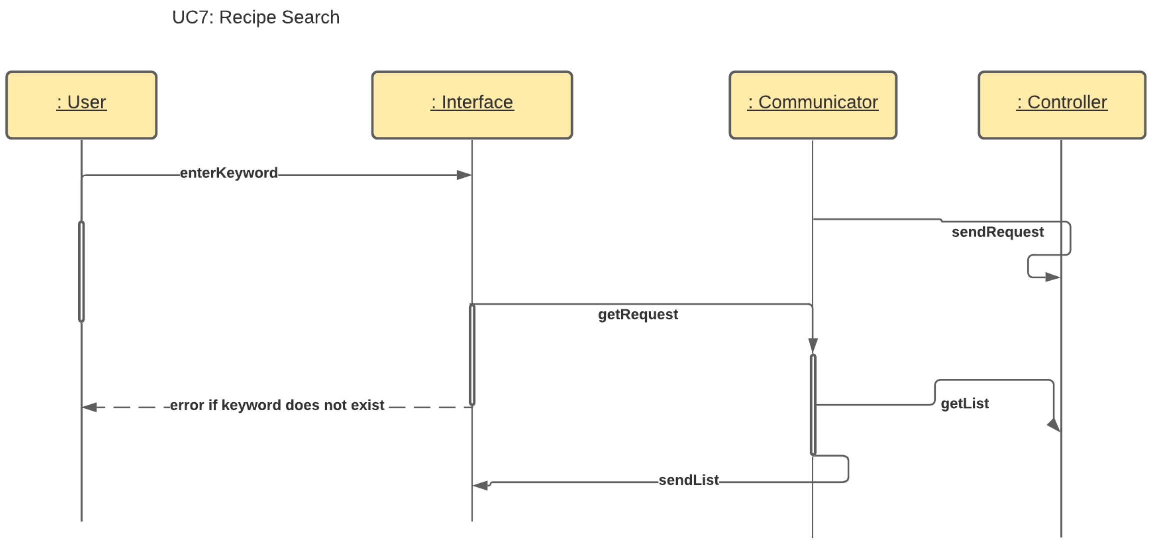
The above interaction diagram is for UC6: Filtering Generated Recipes. After a generated list is displayed via the Interface, the user can filter it out to make it narrower and more focused on what they want. The Communicator will send those updates to the Controller. The Controller will then update the list and send it back to the Communicator so it can send it to the Interface again and the new updated list can be displayed.

Design Pattern Used: Indirection and High Cohesion Design Pattern, Polymorphism

The reason we chose this design is because the Indirection Design pattern is meant to have an intermediate component which communicates with other components. For this use case, we have a communicator in between the interface and the Controller. The Controller is the middle component, which will handle all the requests and method passing from the Interface and the Controller.

The reason we chose polymorphism is because it is used to handle alternatives based on type. Polymorphism assigns responsibility for alternatives based on how each one varies by type (class). This will help us distinguish between filters for each recipe.

UC7:

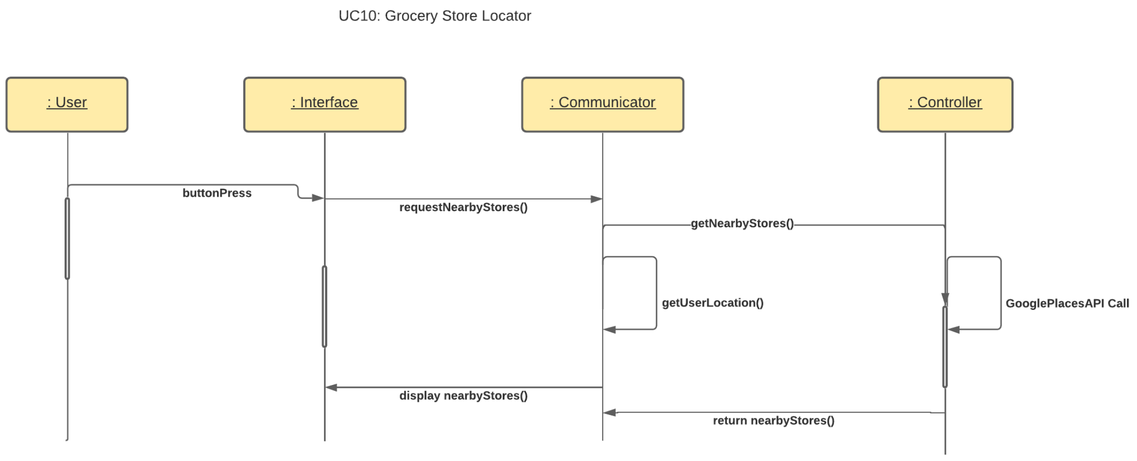


The above interaction diagram is for UC7: Recipe Search. The user will be able to input a keyword via the Interface to search for what specific recipe they want. The Communicator will receive those requests, and will then send back a list of recipes that match the keyword inputted. The Communicator will send the request to the Controller. The Interface will now display that list for the user to view and browse. If there are no recipes that match to the keyword entered, then an error message will be displayed.

Design Pattern Used: Command-based Design Pattern

The Command-based Design Pattern is useful for this design because the Communicator is sending requests to the Controller to access the ingredients so recipes can be generated. The Command-Based design pattern is the pattern where the request is sent as an object, and in this case, it is a JSON object. As such, the information is able to be modified and sent to the interface as it needs to.

UC10:



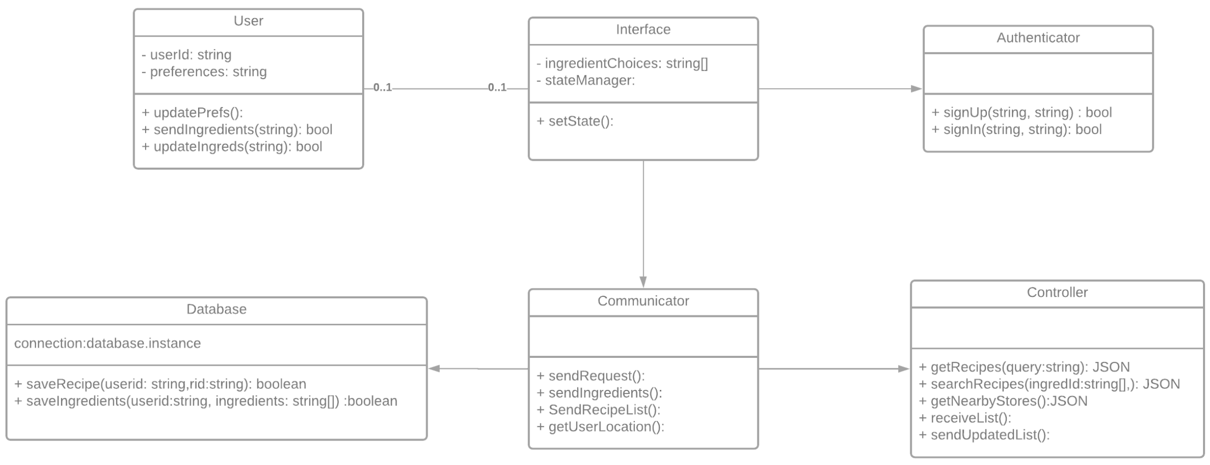
The above interaction is for UC10: Grocery Store Locator. Once the user is informed on what ingredients are missing from their desired recipe, they will interact with the Interface and seek a grocery store nearby with the missing ingredients by pressing a button. The Interface will then request nearby grocery stores to the Communicator, and the Communicator will then tell the Controller to get the nearby stores. The Communicator will get the user's current location through the location feature on their device, and the Controller will place an API call to the Google API. The Controller will return the nearby grocery stores to the Communicator, which will then send the information to the Interface to be displayed.

Design Pattern Used: High Cohesion Design Pattern

For this use case, we decided to use the High Cohesion Pattern. The High Cohesion design pattern is when an object doesn't take on too many responsibilities. We see that each object that we have created here is not handling all of the responsibility for the program. The Communicator is handling different responsibilities from the Controller, which is different from the Interface. In this way, the objects handle various different responsibilities so as not to make one thing do too much. This is the case for UC 10, and also for the other use cases, where the responsibility has been split up between the classes.

3) Class Diagram and Interface Specification

a. Class Diagram


User
- userId: string
- preferences: string
+ updatePrefs():
+ sendIngredients(string): bool
+ updateIngreds(string): bool
0..1
0..1
Interface
- ingredientChoices: string[]
- stateManager:
+ setState():
Authenticator
+ signUp(string, string) : bool
+ signIn(string, string): bool
Database
connection:database.instance
+ saveRecipe(userid: string,rid:string): boolean
+ saveIngredients(userid:string, ingredients: string[]) :boolean
Communicator
+ sendRequest():
+ sendIngredients():
+ SendRecipeList():
+ getUserLocation():
Controller
+ getRecipes(query:string): JSON
+ searchRecipes(ingredId:string[],): JSON
+ getNearbyStores():JSON
+ receiveList():
+ sendUpdatedList():

b. Data Types and Operation Signatures

i. User:

The concept behind the User class is to allow an individual to enter and search for information regarding their account and recipes. The various attributes and functions that the User will perform are:

1. Attributes:
   a. **userId: string**
   This attribute will allow for the user to have an identity within the mobile app's system. The user will use this ID to sign in and operate the app. The userId will be the user's preferred email address.
   b. **preferences: string**
   This attribute will allow for the user's dietary preferences and restrictions to be selected, saved and passed to other parts of the system when searching for recipes.
2. Functions:
   a. **updatePrefs():**
   This function will allow for the user to change their dietary preferences and restrictions anytime they desire. If any preferences are changed, they will be saved and updated to the database.
   b. **sendIngredients(string): bool**
   This function will allow for current ingredients that are selected by the user to be saved into the database. This information will then be used when searching for recipes for the user, with consideration of the user's preferences.
   c. **updateIngreds(string): bool**

This function will be used for when the user has to update their current ingredients within their possession. Allowing for the user's ingredients list to be updated will allow an easier convenience for them, as they don't have to keep entering in prior information with every use of the app.

ii. Interface:

The concept behind the Interface class is the software product that will allow the user to interact with the mobile application product. The Interface should be user-friendly and simple to operate in order to make the user experience enjoyable and useful. The operations that will be performed by the Interface are:

1. Attributes:
    a. **ingredientChoices: string[]**
    This attribute will allow for a list of ingredients that the user has selected to be generated and stored under the user's account. These ingredients will be used to help look for recipes for the user's meal.
    b. **stateManager:**
    Each part of the interface will have a state manager will be updated with the call of the setState function
2. Functions:
    a. **setState(): void**
    This method will be to update the state of all the elements of the app which are interactable.

iii. Controller:

The concept behind the Controller class is to manage the flow of data coming in from other classes within the system. The various functions that are performed by the Controller are:

1. Functions:
    a. **getRecipes(query:string): JSON**
    This function allows for a list of found recipes to be generated that are based around the user's information that was given when searching for a recipe to make.
    b. **searchRecipes(ingredid:string[]): JSON**
    This function allows for recipes to be searched for based on the user's current ingredients and preferences that are stored in the Database. The API then takes this information and looks for recipes with these specifics from the user.
    c. **getNearbyStores(): JSON**
    This function allows the mobile app to locate nearby grocery stores that have the user's missing ingredients available, as well as accounting for the distance the user is willing to travel.
    d. **receiveList():**
    This function allows for a generated list of recipes to be sent back to the user. Based on the user's current ingredients and dietary preferences/ restrictions, an organized list of recipes with specific instructions will be presented to the user.
    e. **sendUpdatedList():**

This function allows the the generated list of recipes that have been narrowed down even further to be sent back to the user. If the user decides to filter more specific requirements to their meal, those filters would then be applied and a newly generated recipe list will be presented to the user.

iv. Communicator:

The concept behind the Communicator class is that it allows the application to talk between different classes within the software. The Communicator will be able to receive and send data across the system, especially from the Interface and the Controller. The various functions that will be performed by the Communicator are:

1. Functions:
    a. **sendRequest():**
    This function allows for the system to communicate with other classes within the mobile app. Requests will usually be sent from the Communicator to either the Controller or Database in order to incite the system to perform a certain action.
    b. **SendRecipeList():**
    This function is used to allow the recipes that fit the user's criteria in a meal to be displayed on the Interface. It will allow the user to see all the options that the Database and API have found for them.
    c. **getUserLocation():**
    This function allows for the mobile app to get the current location of the user. The current location of the user will be requested by the system and stored on the device, as their location could be everchanging. The user's location will allow them to find stores that have their missing ingredients for their recipe chosen.

v. Authenticator:

The concept behind the Authenticator class is to ensure the safety and security of a user's account. The Authenticator will make sure that the account the user is logging in with is a valid account within the Database. The various functions that are performed by the Authenticator are:

1. Functions:
    a. **signUp(string, string): bool**
    This function is used to create a new user within the system. After signing up for the mobile app, the user's information is sent to the database to create a new user, as well as checks that the username isn't already taken.
    b. **signIn(string, string): bool**
    This function is used to authenticate the user, which will allow the user to operate the application. The system checks whether the user is present within the database or not.

vi. Database:

The concept behind the Database class is to allow for the storage of all data that will be used to run and operate this mobile application. The Database allows for information to be kept in one secured location and be accessible anytime when utilizing the mobile application. The data that will be stored will be generated by the user and API that are used in our application. The operations that are performed by the Database are:

1. Attributes:
    a. **connection.database.instance**
    This attribute allows for the database to connect to the software behind the mobile application and lets for information to enter the database and be returned back to the interface.
2. Functions:
    a. **saveRecipe(userid; string,rid:string): boolean**
    This function will save any recipe that the user likes/favorites to their profile and allow the user to refer back to the recipe at a later time.
    b. **saveIngredients(userid:string, ingredients: string[]): boolean**
    This function will save any ingredients that the user has inputted into the system, allowing the user to add and remove their current ingredients with ease.

c. Traceability Matrix

The table below depicts how the classes discussed above have evolved from previous domain concepts and how they relate to the use cases.

	User	**Interface**	**Communicator**	**Controller**	**Authenticator**	**Database**
UC-1	Enters desired username and password through the Interface	Relays the user's information to the Authenticator			Verifies whether the entered username is already existent	
UC-2	Selects dietary preferences and restrictions and distance preferences for grocery stores through the Interface	Confirms user's selections and sends them to Communicator; receives approval from Communicator of saved preferences	Sends preferences to the Database; receives saved preferences from the Database and sends approval to the interface			Stores user's preferences
UC-3	Enters valid username and password through the Interface	Sends credentials to the Authenticator; shows state of login to user			Sets the authentication state; relays state back to Interface	
UC-4	Inputs available ingredients for recipe generation through the	Sends user's inputs to the Communicator	Sends request to the Database; displays the recipe search page to the Interface			Stores ingredients and sends back saved ingredients to the

	Interface	and displays recipes to user				Communicator
UC-5	Applies selection for recipes based dietary restrictions and current ingredients through the Interface	Sends filters to the Communicator and displays filtered recipes to user	Sends a request to the Controller; sends recipes back to the Interface	Receives request and returns list of recipes back to Communicator		
UC-6	Applies more specific filters to generated recipes through the Interface	Relays the user's filters to the Communicator and displays filtered list to user	Sends updated preferences to the Controller; receives and sends updated list back to the Interface	Updates the list of recipes and sends it back to the Communicator		
UC-7	Inputs a keyword via to search for a specific recipe through the Interface	Relays inputs to the Communicator and displays list of recipes to user	Receives requests from the Interface and sends them to Controller; receives a list of recipes that match the keyword inputted and sends it to the Interface	Handles request and sends back list to the Communicator		
UC-10	Presses a button to seek a grocery store through the Interface	Requests nearby grocery stores to the Communicator; displays nearby grocery stores to the user	Uses location of user's device and sends a request for a nearby grocery store to the Controller; receives request and sends it to the Interface	Uses an API to return location of nearby grocery stores to Communicator		

4) Algorithms and Data Structures

a. <u>Algorithms</u>

Our mobile application does not use any algorithms in order for it to operate.

b. <u>Data Structures</u>

Many of the data structures involved in this application are predefined classes within flutter. They are described below:

1) <u>Trees</u>

Flutter operates off of widgets, and these widgets are organized in a tree data structure. Each widget has depths, where interacting with a certain widget can cause you to go deeper within the tree. The tree is meant to organize the widgets in a way that the user can interact with them in an organized manner.

2) Lists for navigation pages

The navigation pages manage the user experience and their redirection to a number of other pages. As such, they contain an array of references to other page links. Navigation pages offer an easy to implement solution for redirecting users between screens with minimal computation overhead. Navigation pages are a predefined class.

3) Lists, dictionaries for JSON

The JSON information that we are retrieving through API requests comes in the form of lists and dictionaries. We have to parse through the JSON data in order to get the information we need. Lists and dictionaries make the information easy to navigate, and is a crucial part to our application.

c. Concurrency

Our application is being programmed with Dart which is only single threaded. Dart, which is the language used in Flutter, operates using a widget tree, where the code is processed in order from top to bottom. No threads are created when using Flutter, therefore our program is single threaded.

5) User Interface Design and Implementation

For the UI design, we are using Figma to help illustrate the application and the steps that a typical user would go through to access and find recipes. This allows us to get a general idea of how we want our mobile app to look and feel. In addition, it allows every member to see and give feedback on their likes and dislikes of the app design. With this user interface design, we are able to offer a preview of what our users should be expecting to see and be using from this app. The way that the user interface was initially designed, it was meant to be user friendly and allow the user to use minimum effort. There have been no significant changes that have been made for the User Interface since the mock-up drawings from Report 1.

After having the UI design done and a general guideline of what we expect the mobile application to look like, implementation is the next step. Implementing the initial mock-up drawing for the User Interface will be done with the use of Flutter for frontend development. Within Flutter, we will be using Dart to implement our design, which is the coding language of Flutter. The use of Flutter was decided upon as it is easy to use, which will allow us to make the User Interface user friendly. With such simple mock-up drawings and concept design that is easy to follow, having to implement the drawings into an interface will not be as complex as we had originally expected.

6) Design of Test

To ensure that our application works with no errors on the user interface and backend, we will be using Flutter to test the functionalities of the application. Below are the areas where the test cases will be used to ensure thorough functionality.

1. User Registration
    a. Goal: The user will be able to create a new account with a unique username and password.
    b. Test case: This test will be used to make sure that users can register themselves for a ChefPal account without any problems, given that the username is unique and that the password is sufficient.
        i. Success: The user creates a new account.
        ii. Failure: The user is unable to create a new account.
2. User Login
    a. Goal: The user will be able to login to his or her ChefPal account.
    b. Test case: This test will be to make sure that users can login into their ChefPal accounts without any problems.
        i. Success: The user logs into the ChefPal account with the correct username and password.
        ii. Failure: The user cannot login to the account, even with the correct username and password.
3. Ingredient Selection
    a. Goal: The user will be able to enter the ingredients to be used for recipe generation.
    b. Test case: This test will be to make sure that users can select ingredients into the application.
        i. Success: The user selects ingredients and will be able to view available recipes based off of the selected ingredients.
        ii. Failure: The user is unable to select ingredients, and the application will not register the ingredients as selected.
4. Recipe Search with Ingredients
    a. Goal: The user will be able to look up recipes based on his or her selection of available ingredients the user has that they selected from the drop down menu.
    b. Test case: This test will be to make sure that when users search for recipes using ingredients, the search results are valid.
        i. Success: The user views the available recipes based on his or her available ingredient list.
        ii. Failure: The user cannot view any recipes that include the selected ingredients available, or the application does not output any search matches, specifically when matches exist.
5. Recipe Search with Keyword
    a. Goal: For users to search for specific recipes using a keyword that they can type in.
    b. Test case: This test will assure that relevant recipes related to the keyword user types will be shown.
        i. Success: Recipes match the keyword user provided and outputs relevant results.

ii. Failure: Recipes do not match to the keyword user provided and or yields no results to the keyword given.

7) Project Management

a. Merging the Contributions from Individual Team Members

While completing the final copy of Report 2, we ensured that all requirements were completed and updated if suggestions were given to us to improve upon. In addition, our group ensured that the report was organized which included keeping the format uniformed and consistent. This detailed that all the wording was the same font and similar sizing in regards to the report's format. Also, we had made sure that all charts, graphs, and images were neatly presented and easy to read. Some of the issues that we encountered was figuring out what was expected of us pertaining to the Interaction Diagrams. As a group we didn't know how to approach creating the diagrams at first. To resolve our issue, we looked at prior projects for inspiration and to gain a basic understanding of what is expected. Another issue that we faced as a group was during the Conceptual Model part of the report. We found difficulty identifying which responsibility of our mobile application fell under what concept. After some deeper research under what aspect is supposed to perform what function, we were able to decipher which function would perform which responsibility. In addition, as we continued the report, we found that we had to update which concept was used for each use case. At first there was some confusion on what each case would use, but after a group discussion we had solved our own problem and allowed everyone to be on the same page.

b. Project Coordination and Progress Report

While continuing to finish up Report 2, we used Discord in order to communicate and discuss the future of ChefPal. We were able to still hold weekly meetings every Monday and Thursday and remain in contact with instant messages and voice calls. In addition, we would at times hold smaller meetings to finish up and finalize the reports. In order for everyone to contribute towards writing and formatting the report, we used Google Docs in order to share and collaborate on the completion of the report as a group. We are also using Github in order to collaborate on the coding portion of the project in preparation of the first demo being due very shortly.

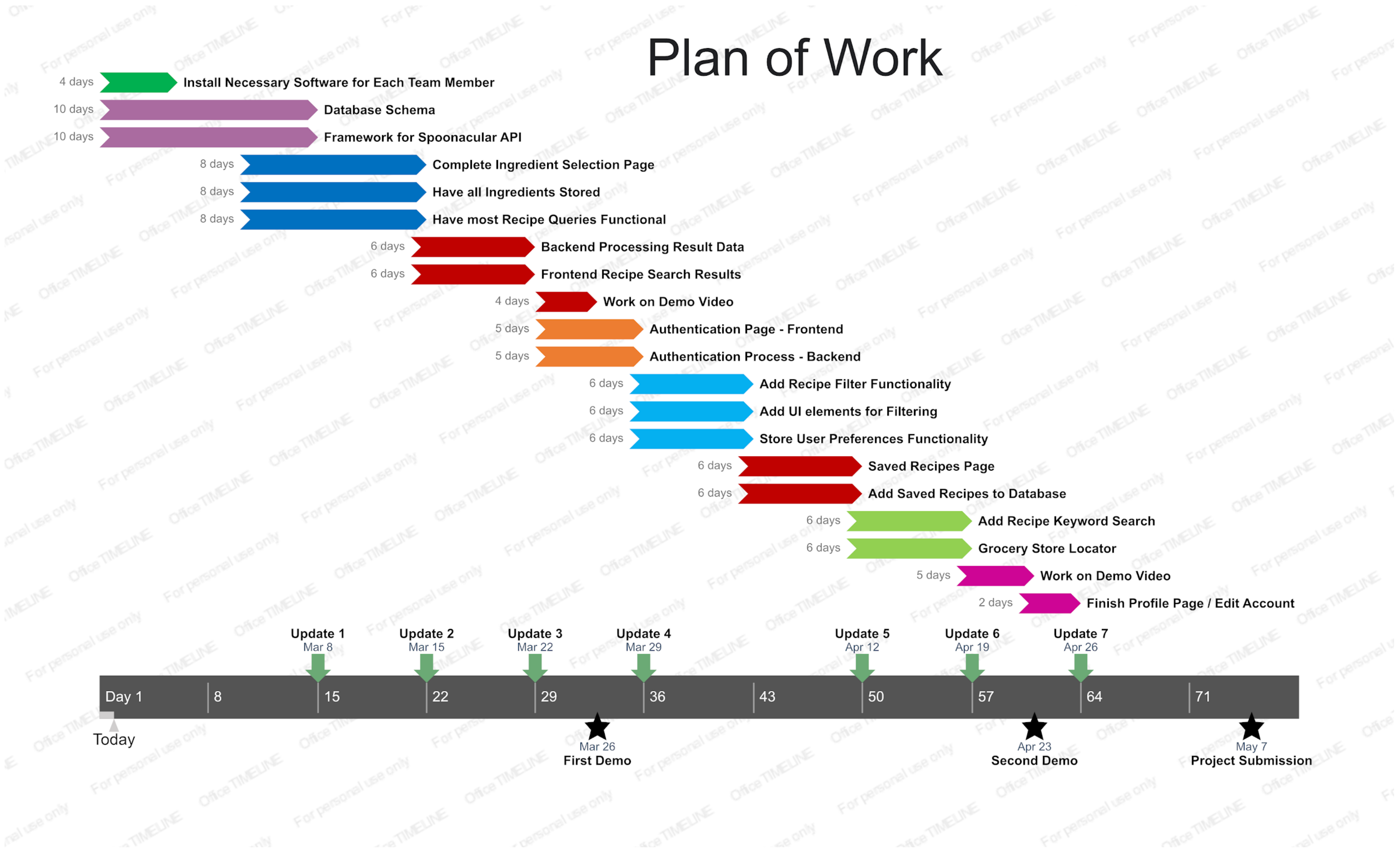
For the coding portion of ChefPal, we will be using Flutter, JavaScript and Dart for the mobile app development. Flutter is being used to write and develop our mobile app, while JavaScript will be used for the backend portion and Dart will be used for the frontend portion of this mobile app. In addition, in order to store the user's data and other data needed for this project, we will be using Firebase and SQL web services.

So far, at the time of the report being submitted, we have been able to code and implement Use Case 3, which is the log-in portion of the mobile app. Setting up the log-in portion of the mobile-app allows for us to have a starting point in the mobile app development. The log-in page is functional and is able to handle invalid logins, which is good for security purposes. In addition, authentication state management is set up, which is part of helping the log-in process figure out which user is valid or not.

Currently, we are working on creating the framework for the navigation of the mobile app once the user has successfully logged-in. The framework will allow for the user to access different aspects of the app itself. This will help implement Use Case 4, 7, 8 and 9. In addition, we are also working on developing the drop down boxes that will be used for the user's ingredient selection. This will help implement Use Case 4. Furthermore, we will start tackling the data management portion of the project by starting to develop the database of the mobile app and allowing for information to be saved.

c. <u>Plan of Work</u>

To keep track of all of our deadlines as well as to make sure that the application development is running smoothly, we have also generated models to help us keep track of our progress and what we need to accomplish. The models that we have created can be seen below.



d. <u>Breakdown of Responsibilities</u>

We have divided the project into four major parts so far and have established who will be responsible for what part.

Group 1 - Malena and Azim:

- Allowing users to check off the ingredients they have available to them.
- Allow users to check off what specifications/ restrictions they want, whether it is any dietary restrictions, cuisine or types of meals.

Group 2 - Nirav, Shahir, and Dymytriy:

- Offering a variety of recipes based on user preferences and inputted ingredients
    - Recipes shown are based on the filter selected by the user
    - Overarching goal is to create an analytics engine capable of basing recipe suggestions off past choices.

Group 3 - Michael, Malak, and Daniel:

- Storing both a user's favorite recipes in a recipe book and their previously checked off ingredients and preferences for future use.

Group 4 - Aswathy, Amanda, and Azim:

- Offering a variety of grocery stores based on user preferences, such as convenience, based on the missing ingredients in the recipe

In regards to integration, we are all sharing our own code and work through our Github repository. This will allow all group members to collaborate and view each other's code to make the creation process of the mobile app easier on all aspects. Sharing the code allows for the frontend, backend, and data management to be consistent with one another and make sure that all of the code can operate with one another. The integration will take part once testing is completed and each group is satisfied with their end product. The integration will be coordinated with at least one of the members from each individual group, totaling 4 people working on integration. This will allow for a integration to go smoothly and not be reliant on just one person who may not understand the other group's code.

Each individual group will perform and integrate testing based off of the responsibilities that are assigned to them. Each group's testing will be different and have their own approach on how to test their code. Testing should be done by the members in those individual groups that had developed the original code. All members should be able perform the testing of the code to make sure that it runs correctly, as well as to get a better understanding of how everything else is connected.

8) References

Amazon Web Services, Inc. 2021. *Build a Flutter Application on AWS*. [online] Available at: <https://aws.amazon.com/getting-started/hands-on/build-flutter-app-amplify/> [Accessed 24 February 2021].

Amplify Framework Documentation. 2021. *Amplify Framework Documentation*. [online] Available at: <https://docs.amplify.aws/lib/datastore/sync/q/platform/flutter> [Accessed 24 February 2021].

Docs.aws.amazon.com. 2021. *Getting started in React Native - AWS SDK for JavaScript*. [online] Available at: <https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/getting-started-react-native.html> [Accessed 6 March 2021].

Eceweb1.rutgers.edu. 2021. [online] Available at: <http://eceweb1.rutgers.edu/~marsic/books/SE/projects/HealthMonitor/2019-g2-report3.pdf> [Accessed 3 March 2021].

Eceweb1.rutgers.edu. 2021. [online] Available at: <http://eceweb1.rutgers.edu/~marsic/books/SE/projects/Restaurant/2019-g13-report3.pdf> [Accessed 16 February 2021].

Ers.usda.gov. 2021. *USDA ERS - Food Prices and Spending*. [online] Available at: <https://www.ers.usda.gov/data-products/ag-and-food-statistics-charting-the-essentials/food-prices-and-spending/#:~:text=In%202019%2C%20households%20in%20the,representing%208.0%20percent%20of%20income> [Accessed 16 February 2021].

Firebase.flutter.dev. 2021. *FlutterFire Overview | FlutterFire*. [online] Available at: <https://firebase.flutter.dev/docs/overview/> [Accessed 6 March 2021].

Firebase. 2021. *Firebase Realtime Database*. [online] Available at: <https://firebase.google.com/docs/database> [Accessed 22 March 2021].

Flutter.dev. 2021. *Firebase*. [online] Available at: <https://flutter.dev/docs/development/data-and-backend/firebase> [Accessed 24 February 2021].

Flutter.dev. 2021. *Write your first Flutter app, part 1*. [online] Available at: <https://flutter.dev/docs/get-started/codelab> [Accessed 24 February 2021].

GitHub. 2021. *GitHub: Where the world builds software*. [online] Available at: <https://github.com> [Accessed 3 March 2021].

Google Developers. 2021. *Overview | Places API | Google Developers*. [online] Available at: <https://developers.google.com/places/web-service/overview> [Accessed 16 February 2021].

Lucidchart. 2021. *Online Diagram Software & Visual Solution | Lucidchart*. [online] Available at: <https://www.lucidchart.com/pages/> [Accessed 12 March 2021].

Marsic, I., 2021. *Software Engineering Project Report - Requirements*. [online] Ece.rutgers.edu. Available at: <https://www.ece.rutgers.edu/~marsic/Teaching/SE/report1.html> [Accessed 16 February 2021].

Medium. 2021. *Size matters: Reducing Flutter App size best practices*. [online] Available at: <https://suryadevsingh24032000.medium.com/size-matters-reducing-flutter-app-size-best-practices-ca992207782> [Accessed 24 February 2021].

Medium. 2021. *Must try: Use Firebase to host your Flutter app on the web*. [online] Available at: <https://medium.com/flutter/must-try-use-firebase-to-host-your-flutter-app-on-the-web-852ee533a469> [Accessed 6 March 2021].

Office Timeline Online. 2021. *Build native PowerPoint timelines online - Office Timeline Online*. [online] Available at: <https://online.officetimeline.com/app/> [Accessed 12 March 2021].

Rnfirebase.io. 2021. *React Native Firebase | React Native Firebase*. [online] Available at: <https://rnfirebase.io> [Accessed 6 March 2021].

Spoonacular.com. 2021. *spoonacular recipe and food API*. [online] Available at: <https://spoonacular.com/food-api> [Accessed 16 February 2021].