Find Your Recipe Final Report

Adam Ryan ([ryan.1781@osu.edu](mailto:ryan.1781@osu.edu)) and Danny Kan ([kan.74@osu.edu](mailto:kan.74@osu.edu))

# I. Introduction

With the recent increase in the prevalence and popularity of food-related delivery services such as Grubhub, DoorDash, and Uber Eats, many users find themselves willing to pay a higher price tag in exchange for spending less time planning and preparing meals [**1**]. While these services are convenient for the average customer, they do not consider recommendations, possible dietary restrictions, or delivering health-conscious options tailored to support individual needs and preferences.

As a result, we propose a multi-functional recipe mobile application using Kotlin to make it easier for everyday users to prioritize their general well-being by providing them with cost-effective and health-conscious alternative recipes.

There is a strong emphasis on the user experience. With just a few taps on the screen, users sign in with our platform, discover popular and trending recipes among community members, have the option to favorite them, receive personalized recommendations based on interaction history, store and display custom recipes, and search for new ones using the integrated search functionality. The planning stage is time-consuming. With that in mind, we designed and developed a mobile application to deliver strong insights into meal planning and macronutrient tracking to support user goals.

# II. Design Process and Intended Goals

Once the team members identified the intended audience and the problem at hand, the initial stage of the design process consisted of identifying candidate nouns from the narrative to develop a unified modeling language (UML) class diagram for the proposed application's domain model. The diagram served as a basis for implementing classes using object-oriented principles, and the verbs served to describe relationships between them.

The development of the schema was relatively straightforward. The mobile application focuses on home cooking and can share recipes, so the design requires the Recipe, Ingredient, Instruction, and User classes. Ingredients and Instructions map to Recipes many-to-one, and Recipes map to Users many-to-one. Most consideration was given to the specifics of these classes. The team opted for UUIDs for each recipe and author since users can create a recipe offline and asynchronously. The ID of any recipe and user must be universally unique, hence the choice of UUIDs. The team categorizes an ingredient into an amount, unit, name, and RawText. This allows most ingredients to be further categorized while allowing RawText to represent the Ingredient if the other three cannot sufficiently represent an Ingredient. Separating amounts from the rest of the ingredients enables the team to categorize and export them to shopping lists, although this feature did not make it in time for the course.

Instructions were straightforward, and plain text was simply stored. Current work focuses on the standardization of instructions, and future work may allow a more specific representation of instructions.

The team decided that storing its difficulty, time to prepare, title and description were sufficient for each recipe. These metrics are specific to home cooks; some have different skill levels when in the kitchen. Additionally, the target audience is those who resort to food delivery applications, so being able to pick recipes that take less time to prepare is likely vital to them.

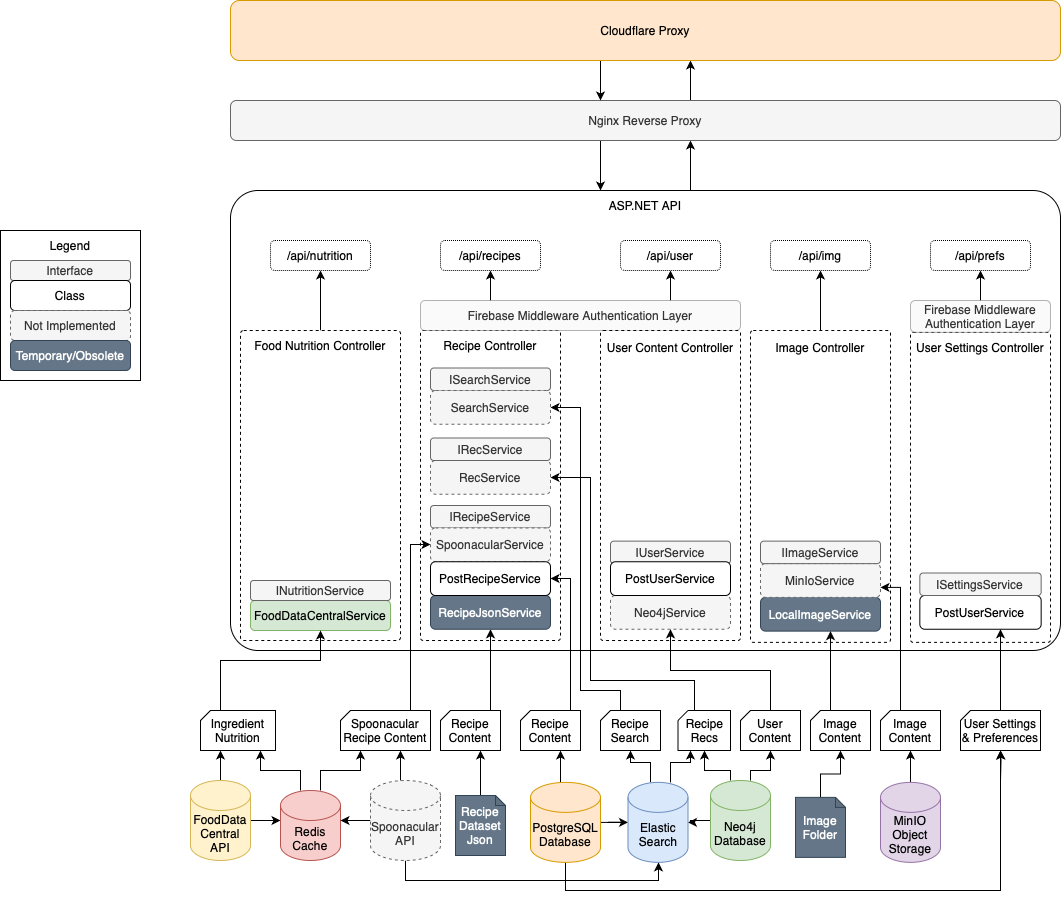
# III. Translating Design to Implementation

The design translated very cleanly to implementation, as the team had previous experience working on a similar problem beforehand. The design already contained the schema needed for this process and got a head start developing the backend API. The main challenge for this course was to create the Android app component.

However, one thing not fully considered was how the database schema may differ between the actual app and the backend. As a first step, the team isolated what type of data must be stored to achieve the design goals. Because the focus was a health-focused recipe app with network connectivity, users, recipes, instructions, ingredients, and nutrition would all be some of the objects that would be stored. Its functionality without access to the Internet was more of an afterthought, not the primary focus because the average user will be cooking in their home, with access to the Internet.

Afterward, the group separated the data between what would be stored locally and in the cloud. While a strict schema had to be enforced online, a simplified approach was taken in the local database, where less emphasis was placed on recording the author of the recipe and nutrition information (due to time).

Online, the team makes an API using ASP.NET [**2**] to build the backend, which serves recipe/search content, recommendations, and user information as a minimum viable product for this project. To provide maximum user security, the team takes a security-first approach by implementing Firebase OAuth [**3**] before implementing any API data logic. Once each endpoint is abstracted and implemented from the design process, the team also decides whether or not to protect this endpoint behind authorization to prevent improper access and maintain consistency with the project's goals. Much thought has gone into this API, but most were developed outside this class's scope, so the rest of the implementation details will be skipped. The final scope of the API for this course is left as an MVP that will allow the user to request recipes (whether via search, recommendation, or plain request) and request respective images for said recipes. For a high-level design of the API, see Figure 1 below. Overall, the development of the API was smooth, with no blockages, aside from the expected difficulties of working with a new framework. The extensive planning beforehand and previous experience made this process much smoother. One member built another version of this API using a Python backend framework, making the process faster and quicker.

**Figure 1** - *High Level API Architecture*

The team's next step was to break use cases into individual tasks to provide the user with an intuitive and concise experience. First, the interface was broken into three significant abstractions:

* **Discovery Recipe Content**: This is a place for the user to view recommended recipes, popular recipes, and search for specific recipes. The discovery recipe content encompasses all MVP online recipe-related content.
* **Localized Recipe Content**: This is a place for the user to create, edit, delete, and view locally saved and/or self-made recipes. The localized recipe content encompasses all MVP offline features.
* **User Account Content**: This is a place for the user to manage their account with our service. Any account management features, such as sign-up/login controls, settings, and log-out controls will be managed here.

Now that the feature goals have been broken into three groups, the team creates a fragment for each and implements a bottom navigation [**4**] on the Main Activity to switch between the three.

Next, the group splits to work on individual fragments. The Recipe Fragment is further broken into three different pieces, one for each online recipe feature:

* **Popular Recipe Content**: Recipes that many other users have recently saved.
* **Tailored Recommendation Content**: Recipes that our service thinks the user may like based on a proprietary algorithm.
* **Search Recipe Content**: Recipes that match a search query the user provides.

The three fragments are switched between using a ViewPager2 [**5**]. The design takes inspiration from the popular mobile app TikTok [**6**]. Taking a similar approach to this app, the main form of content is shown right at the app's start and is swappable between different forms (search, trending, and recommended). The other secondary content (saved, create, and account features) is provided at the bottom and allows the user to tap to switch to these. By using a similar UI design to tried-and-true applications in production, the team reduces the time needed for the user to learn a new UI. This division of features proved successful and formed an intuitive design, drawing on the design of other successes.

Because the three forms of content delivery all result in one or more recipes, the team abstracts the work of displaying recipes into a reusable recipe list component utilizing a recycler view [**7**]. This component provides numerous features, including:

* **Skeleton Loading View**: When set to loading mode, placeholder skeleton rows are displayed, which play an animation that increases responsiveness. Each item matches the shape of a loaded recipe and is constantly moving, helping the user feel that something is continually happening, even when waiting for the API to return information.
* **Automatic Pagination**:When the user scrolls near the bottom of the recipes (i.e., within five recipes of the bottom), a callback option is provided that allows the fragment to load more recipes automatically. While it is up to the developer to handle pagination, this simplifies the process and will enable features such as infinite scroll, even though the API paginates recipe content by default.
* **Automatic Metric Display**: Additional data (e.g., prep time, difficulty, and short numbers) are displayed under the title and description as tags. When more than a few are displayed, they are automatically collapsed, and a “+n more” tag is shown.
* **Expandable Recipes**: An onclick handler that opens the stored recipe in further detail in a new Recipe Detail Activity.
* **Recipe Detail Activity**: A new activity that allows the user to view the recipe in full detail, display important information, ingredients, and instructions. It also provides a back button and a “Save Recipe” button, which writes to the ROOM database [**8**].
* **Glide-based Image Loading**: The Android library Glide [**9**] was utilized for efficient and fast image loading.

By abstracting the recipe list task into a reusable component, all three pages can be implemented much faster while providing in-depth functionality, increasing responsiveness, and an easy-to-understand UI. Overall, this strategy was extremely time-saving, as the changes required for each sub-fragment contained much overlap.

The Trending fragment is implemented first and hosts the recipe list component. On load, the view model asks an API client class to request trending recipes. The View tells the recipe list component to display the “loading screen” until the list of recipes is populated with data from the API.

The Discover fragment is implemented similarly but passes in the Firebase authorization token in the header. It is handled similarly, except the API only returns one recommendation. Because sensors are a requirement in this project, the team implemented a shake-to-refresh feature to get a new recommendation.

The Search fragment is also implemented similarly to the trending fragment, but a search bar is added to the top. Editing the text in the search bar and pressing the “right submit arrow” button performs a similar task, asking the API client class to send a query to the API. However, we now pass in a query parameter that the API can use to filter recipes.

Implementing the saved fragment was relatively straightforward. Inspired by the previously implemented recycler view class, the main change was removing the skeleton holders and allowing the user to perform CRUD operations.

The team implemented Firebase OAuth [**3**] to manage user logins and tokens in a Firebase Utils class. This class is responsible for logging in and registering users using a username and password and also allows classes to request the current user’s authorization token for use in API requests.

With all basic functionality implemented, the team focused on improving the user experience. The first step taken was to ensure that all features intended to be specific to the user would work offline. First, the team added the feature allowing users to save recipes to the ROOM database for offline use and modified Glide to offline cache [**10**] the images. Additionally, rotation logic was decided to force portrait mode on the list view (to fit better to the UI design) and allow landscape rotation when a Recipe Detail Activity is open. Enabling the save to ROOM feature was initially a struggle due to the discrepancy between how recipes were returned from the API and the structure of the ROOM database. Some migrations were needed to match the ROOM schema well and the need for custom loaders using Serializable [**1**].

# IV. Suggested Changes and Improvements

One central point of struggle resides in mapping API data to recipe data. The API returns a list of Recipes with accompanying data nested inside. Still, the Recipe schema in the app does not correctly map all Ingredients and Instructions in a way that ROOM can understand [**8**]. The team’s solution was to wrap the Recipe, Ingredients list, Instruction list, and Author object in a new ApiRecipe class and manually write each to ROOM when saving. We will plan this with a more robust solution if we redo the project.

Another point of frustration stems from the API’s ID system, where the schema utilized UUIDs for User and Recipe primary keys. This was a point of struggle in Android, which requires the Java UUID library [**11**] to work. Switching this over required a Room database migration and some hacks to differentiate the User from API recipes. Implementing an Author-centered approach in the future will be majorly beneficial.

More fine-grained control over image loading serves as an avenue for performance improvement. While the team has offline image caching enabled with Glide [**9**], the selection of automatic caching poses a risk of an image un-caching itself when it should not (before a user un-saves a recipe) or remaining in cache forever (even after a user removes a recipe). Manually caching and managing all images should fix this issue.

Finally, a rewrite of the recipe fragment with more vigorous testing is beneficial. The team could not alleviate a significant memory leak in this component, where switching between the Recipe fragment and other fragments causes megabytes of memory to not be unloaded. The team reduced it to a small fraction of that, but there is still a leak somewhere. A different approach to this would be beneficial.

# V. References

[1] <https://doi.org/10.1186/s12889-022-13721-9>

[2] Microsoft ASP.NET, <https://dotnet.microsoft.com/en-us/apps/aspnet>

[3] Firebase Authentication, <https://firebase.google.com/docs/auth>

[4] Bottom Navigation, <https://developer.android.com/reference/com/google/android/material/bottomnavigation/BottomNavigationView>

[5] ViewPager2, <https://developer.android.com/develop/ui/views/animations/screen-slide-2>

[6] TikTok, <https://www.tiktok.com/?lang=en>

[7] Reusable RecyclerView Adapter, <https://medium.com/@vitorthemyth/building-a-flexible-and-reusable-custom-adapter-for-recyclerview-in-android-23c62945406b>

[7] <https://medium.com/@dugguRK/deep-dive-into-parcelable-and-serializable-1d32d38f59d0>

[8] Room <https://developer.android.com/training/data-storage/room>

[9] Glide <https://github.com/bumptech/glide>

[10] DAO <https://developer.android.com/training/data-storage/room/accessing-data>

[11] Java UUID Library <https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html>