My Recipe App: Development Journey Summary

This summary chronicles the development process of the "My Recipe App," an Android application designed to provide users access to meal recipes, drawing from the milestones documented throughout the Platform-Based Development CSC475 course at Colorado State University Global.

Initial Concept and Planning (Module 2)

The journey began with defining the app's core purpose: allowing users to view, store on-device, search, add, modify, and favorite meal recipes. The initial scope included using store-on-device meal recipe data from TheMealDB, an open, crowd-sourced database. Right from the start, scalability was a key consideration, with future plans for integrating TheMealDB API for broader recipe access and sharing, and potentially an LLM chatbot for recipe suggestions and cooking help.

To support this future vision and ensure maintainability, the Model-View-ViewModel (MVVM) architecture was chosen. This approach separates the UI (View) from the application logic (Model) using an intermediary ViewModel, promoting modularity. The app was planned to be developed using Kotlin and Jetpack Compose.

The target audience was identified as broad, encompassing busy parents, cooking enthusiasts, novice cooks, and health-conscious individuals looking for specific dietary recipes. Key initial functionalities included searching recipes (by name, ingredient, category), manually adding recipes, modifying existing ones, and managing a favorites list. Early design artifacts included sequence diagrams illustrating user flows (like searching), initial class diagrams for the data logic, and a Kotlin `Meal` data class example wrapping TheMealDB's JSON structure. UI/UX requirements focused on an intuitive interface built with Jetpack Compose, visualized through use case and activity diagrams.

Detailed Design and Architecture (Module 4)

The second phase delved deeper into the technical design, elaborating on the MVVM architecture. The `Model` layer's responsibilities were detailed: managing local data persistence using the Room Persistence Library (an abstraction over SQLite) and handling API calls to TheMealDB using the Retrofit library. The Moshi library was selected for parsing JSON responses from the API into Kotlin objects suitable for Room storage. The `ViewModel` layer's role was clarified as mediating between the `Model` and `View`, exposing data via LiveData for asynchronous updates and a responsive UI.

TheMealDB API methods were listed, including the use of the test API key "1" for development, with the intention to acquire a premium key for expanded features like multi-ingredient filtering and recipe submission . Code snippets illustrated the implementation of the Data Layer (Room entities, DAOs, Database setup; Retrofit service interfaces, Moshi integration), the Repository Layer (acting as a bridge between data sources and ViewModel), the ViewModel Layer (using

Android ViewModel & LiveData), and the UI Layer (Jetpack Compose examples for list screens and individual item cards).

A significant portion of this phase involved creating detailed UI wireframes using diagrams to illustrate the layout, components, navigation flow, and user interactions across various screens like the Home Screen, Recipe Lists (My Recipes, Favorites, TheMealDB), Search Dialogs, Add/Edit Screens, and Recipe Detail Views.

Implementation and Refinement (Module 6)

This stage focused on the actual coding and refinement, listing specific library versions used (Jetpack Compose, Kotlin, Room, Retrofit, Moshi, Coil for image loading, etc.). Acquiring a lifetime TheMealDB API key ($10) enabled access to premium API features. Secure handling of the API key was implemented by storing it in `local.properties` (excluded from version control via `.gitignore`) and loading it via the Gradle build script, adhering to Android best practices.

A detailed file structure breakdown was provided, mapping directories and key files (like `MainActivity.kt`, `Recipe.kt`, `RecipeDao.kt`, `RecipeRepository.kt`, `RecipeViewModel.kt`, `HomeScreen.kt`, `Navigation.kt`) to their respective MVVM layers (Application, Data/Model, Repository, ViewModel, UI/View) and functionalities [cite: 172-181, 183-200].

Reflecting on the development process, it was noted as challenging, requiring significant effort. The prior documentation proved helpful. Issues encountered were managed by implementing the code in phases aligned with the MVVM layers and adopting a modular approach. Specific challenges included handling TheMealDB's particular formatting requirements for ingredient and area searches (requiring normalization logic in the `OnlineRecipeViewModel`) and resolving UI layout issues (like misaligned icons) through iterative adjustments [cite: 211-219, 220]. Implementing the Navigation component also presented a learning curve. Despite challenges, the MVVM architecture provided a solid, maintainable foundation. The functionality was showcased via screenshots of various screens in light/dark modes and a video demonstration [cite: 230-234, 236, 237, 238, 239, 242, 243, 244, 245].

Testing, Deployment, and Future Plans (Module 8)

The final phase concentrated on testing the application thoroughly and navigating the deployment process. Testing followed a functional approach, categorized into unit, integration, UI, and end-to-end (E2E) testing. Tools like JUnit, Hamcrest, Robolectric, UIAutomator, Espresso, MockWebServer, Android Studio virtual devices, and Firebase Test Lab were utilized. A dedicated 'test' branch was used in Git. The test file structure was organized, separating unit tests (`/src/test`) from instrumentation tests (`/src/androidTest`).

- **Unit Tests:** Covered ViewModel logic (requiring making a repository class `open` to fix inheritance issues), Repository CRUD operations, and utility functions (like Room converters and string formatting).

- **Integration Tests:** Focused on Room database operations and TheMealDB API interactions (using MockWebServer), run using Robolectric. An issue with Android Studio's cache causing `ConnectException` errors was resolved by invalidating caches. Combined repository tests verified interactions between local and remote data sources.
- **UI Tests:** Used Jetpack Compose test rules and UIAutomator/Espresso to test individual composables (`RecipeFormScreen`, `RecipeListScreen`) and navigation flows. Testing revealed UI optimization needs for tablet screens, particularly image scaling. Firebase Test Lab provided broader device compatibility testing.
- **E2E Testing:** Initial automated exploration was done using Firebase Robo test, followed by manual verification of complete user workflows.

Deployment was targeted for the Google Play Store. This involved creating a developer account ($25 fee, ID verification), setting up app content details (privacy policy, content rating, data safety declarations), configuring the store listing (name, description, graphics), and building a signed Android App Bundle (AAB). A challenge arose with the default `com.example` package name being restricted, requiring a change to the `applicationId` in `build.gradle`. A significant hurdle was the mandatory closed testing requirement for new developers (post-Nov 2023), demanding at least 12 testers for 14 days before production release, which proved challenging to fulfill.

Post-publication plans include ongoing maintenance adhering to Google Play policies (like updating target API levels), bug fixing based on user feedback and crash reports, marketing efforts (App Store Optimization, social media), and future enhancements. Planned enhancements include implementing user authentication (potentially with Firebase Authentication), enabling recipe sharing (possibly requiring a custom backend like Cloud Firestore due to TheMealDB API limitations), and integrating an LLM for personalized recipe generation based on ingredients and dietary needs.

Overall, the project spanned the entire app development lifecycle, proving to be a challenging yet rewarding experience and a valuable portfolio addition.