Technical Design Document for Fetch Rewards Recipe App

Overview

This project is a recipe app that displays recipes from a provided API endpoint. The app should show each recipe's name, photo, and cuisine type, and the user can refresh the list at any time. Image caching is implemented for optimized network usage, while recipe data is only fetched on launch or manual refresh. The app should gracefully handle malformed and empty data cases. Unit tests are required to demonstrate effective testing practices.

Project Architecture

Clean Architecture Overview

The clean architecture pattern enables a clear separation of concerns, with each layer handling a specific responsibility. This structure supports scalability, testability, and maintainability, aligning with Fetch's preference for a production-quality codebase.

- **Presentation Layer**: Manages UI logic and displays data to the user.
- **Domain Layer**: Contains the business logic in the form of use cases.
- **Data Layer**: Manages data sources, including network calls, and abstracts data retrieval.

Layers and Components

1. Presentation Layer (MVVM Pattern)

RecipeListViewModel

- **Responsibilities**:
    - Interacts with `FetchRecipesUseCase` to load recipes.
    - Manages UI state (loading, error, empty states).
    - Publishes recipe data for the view to observe.
- **View**: Displays a list of recipes, error messages, and an empty state if no recipes are available. Includes pull-to-refresh functionality.

    - Implemented in SwiftUI for rapid development and built in functionality for pull to refresh.

2. Domain Layer

FetchRecipesUseCase

- **Responsibilities**:
    - Encapsulates the business logic for fetching recipes.
    - Handles empty data gracefully.
- **Implementation**:
    - Calls `RecipeRepository` to retrieve data.
    - Performs any data validation needed (e.g. checking for empty recipes list).
    - Returns a list of valid recipes or an error (e.g. malformed data) for the ViewModel to handle.

This use case simplifies testing and can be expanded for additional business logic without affecting other layers.

3. Data Layer

RecipeRepository

- **Responsibilities**:
    - Serves as the main data access layer, abstracting the source of data.
    - Exposes a `fetchRecipes` method that returns recipe data from an API client.
- **API Client (RecipeAPIClient)**:
    - Handles network calls to retrieve recipe data.
    - Uses Swift's `async`/`await` for concurrency, ensuring non-blocking UI updates.
    - Throws error for malformed data cases.

By isolating network calls within the repository, the app can swap data sources easily (e.g., for local caching in the future) with minimal changes outside the Data Layer.

Image Loading and Caching

Image Caching: To optimize bandwidth usage, a 3rd party library named `Kingfisher` will be used to handle disk caching for images.

- Kingfisher provides a SwiftUI view, `KFImage`, that will be straightforward to include in our views.

- KFImage automatically performs lazy loading when used in a `ListView` or `LazyVStack`, with the option to specify a placeholder view.

UI/UX Design

Recipe List Screen

- **Recipe Cell**: Displays each recipe's name, photo, and cuisine type in a clean, scrollable list.
- **Pull-to-Refresh Control**: Allows users to manually refresh the list on demand.
- **Empty State**: A message indicating "No recipes available" when there are no recipes to display.
- **Error State**: Shows an error message when the data is malformed.

Performance Optimization

- **Lazy Loading**: Loads images as they appear on screen, reducing initial load time and saving bandwidth.
- **Concurrency**: Utilizes `async/await` for non-blocking, concurrent network calls.

Testing Strategy

Unit Tests

- **ViewModel Tests**: Tests data flow through the ViewModel, focusing on handling success, empty, and malformed data scenarios.
- **Use Case Tests**: Tests business logic in `FetchRecipesUseCase` for conditions like malformed data and empty responses.
- **Repository Tests**: Uses mocks to simulate API responses, testing for cases like successful, empty, and malformed data.