Flutter Meal Tracker App - Documentation

Main.dart

This Flutter application serves as a Meal Tracker, allowing users to input and store meal data. The app uses Hive for local storage and Riverpod for state management.

Main Components:

1. Hive Initialization:
   - The app initializes Hive, a lightweight and fast NoSQL database used for local storage.
   - Two Hive models (MealModel and MealItem) are registered, allowing the app to save meal-related data.
   - A mealsDB Hive box is opened to store the data persistently.

2. Riverpod State Management:
   - The ProviderScope from Riverpod wraps the app, ensuring that all state management in the app can be efficiently handled and shared across different parts of the application.

3. MainApp Entry Point:
   - The app starts by running the MyApp widget, which sets up a MaterialApp with basic configurations such as:
   - Title: "Flutter Meal Tracker App"
   - Home Screen: The initial screen is InputMealData, where users can input meal details.
   - Debug Banner: Hidden for a cleaner UI.

Key Packages:

- flutter_riverpod: For efficient state management.
- hive and hive_flutter: For local storage of meals using Hive database.

Important Classes:

- MyApp: The main widget that sets up the app with the necessary configurations and initial screen (InputMealData).
- InputMealData: The screen responsible for taking user input related to meals (imported but not defined in this snippet).

Usage:

1. Ensure Hive is properly initialized with MealModelAdapter and MealItemAdapter registered.
2. Use ProviderScope for Riverpod to manage state across the application.
3. Start the app by running MyApp.

InputMealData

This class is a stateful widget that allows users to input and track meal data. It extends StatefulHookConsumerWidget from Riverpod, integrating both hooks and Riverpod state management.

Class: _InputMealDataState
This class manages the state of the InputMealData widget. It handles data initialization, meal selections, and user interactions like saving meals.

Properties:
- allMeals: A list to store the fetched meals.
- completed: A flag to track if all meal selections are made.
- id: Stores the ID of the first meal in the list for tracking purposes.

Method: initState()
- Initializes the widget and calls _initializeMeals() to load meal data from the Riverpod provider (getAllMealsProvider).

Method: _initializeMeals()
- Asynchronously fetches all meals and sets the id and initializes the meal selection using the first meal (if available).

Method: build(BuildContext context)
- Builds the UI of the widget, which includes:
  - App Bar: A custom app bar with a title and alert dialog.
  - Body:
    - Displays a list of meal types (e.g., breakfast, lunch, etc.) using a ListView.builder.
    - Renders meal selections or provides options to add or edit meals.
    - Two buttons: "Save" and "Cancel".
  - The UI updates the completed state when four meals are selected.

Method: _buildMealSelectionTile()
- Builds a list tile for each meal type, allowing the user to add or edit the meal data. If no selection is made, an "Add" button appears; otherwise, "Add" and "Edit" buttons are shown.

Method: _buildMealDetails(MealItem mealSelection)
- Displays nutritional details (fat and sugar) of a selected meal using small info badges.

Method: _buildInfoBadge(String value)

- Builds a small, styled badge to display meal information like fat or sugar values.

Method: _showMealBottomSheet()
- Displays a bottom sheet where users can input or edit meal details for a selected meal type.

Method: _saveData()
- Saves meal data to Hive:
  - If all four meal types are selected, creates a new MealModel and saves it to the Hive database.
  - If meals already exist, it updates the first entry.
  - Shows a confirmation message upon success or an error message if not all data is entered.

Key Components:
- Meal Selection Providers:
  - Uses Riverpod state management to fetch, store, and manage meal data (getAllMealsProvider and mealSelectionProvider).
- UI Elements:
  - Custom app bar, list of meal types with add/edit options, and buttons to save or cancel.
- Data Management:
  - Reads and writes to the Hive database for storing meals persistently.

This code focuses on meal selection and storing the data through Riverpod and Hive integration, providing a user-friendly interface for meal tracking.

Riverpod Providers for Meal Tracking

1. providerHive
- Type: Provider<DatabaseHelper>
- Purpose: This provider exposes an instance of the DatabaseHelper class, which is likely responsible for interfacing with Hive (or another database) for managing meal data.
- Usage: Other parts of the app can access DatabaseHelper via this provider for database operations like adding, updating, or retrieving meal data.

2. hiveData
- Type: StateNotifierProvider<MealController, List<MealModel>?>
- Purpose: This provider exposes a MealController that manages the state of the list of MealModel objects (meals). The state is either a list of meals or null if no data is available.
- MealController: A state notifier that manages the business logic for manipulating the meal data.
- Usage: It can be used to add, update, or delete meals, and automatically update the UI when the meal data changes.

3. getAllMealsProvider
- Type: Provider<List<MealModel>>
- Purpose: This provider watches the state from hiveData and returns a list of meals. If no data is available (i.e., hiveData is null), it returns an empty list.
- Usage: It simplifies accessing all meals in a consistent way by always providing a list (even if empty), ensuring the consuming widgets don't have to handle null values.

Meal Selection State Management

This code defines a state management system using Riverpod's StateNotifier to manage meal selections in a Flutter application. It handles different meal types (like breakfast, lunch, dinner, and snacks) and allows users to select and modify the sugar and fat content for each meal.

Class: MealSelectionNotifier

This class extends StateNotifier and manages a Map<String, MealItem>, where:
- Key: Represents the meal type (breakfast, lunch, etc.).
- Value: An instance of MealItem, which contains sugar and fat values.

Key Methods:
1. setSugarSize(String mealType, String sugarSize):
   - Updates the sugar content for the specified meal type (mealType).
   - Retains the existing fat value if already set, or defaults to "mittle" (likely meaning "medium").

2. setFatSize(String mealType, String fatSize):
   - Updates the fat content for the specified meal type (mealType).
   - Retains the existing sugar value or defaults to "mittle" if not set.

3. initializeMealSelection(MealModel meal):

- Initializes the meal selection with a full meal object (MealModel), which contains all meal types (breakfast, lunch, etc.) and their respective sugar and fat values.
- Used to load pre-existing data into the state when starting or editing a meal.

4. removeMeal(String mealType):
   - Removes a meal from the current selection by deleting its entry from the state map.

5. clearAllMeals():
   - Clears all meal selections, resetting the state to an empty map.

Provider: mealSelectionProvider

- Type: StateNotifierProvider<MealSelectionNotifier, Map<String, MealItem>>
- Purpose: Provides the MealSelectionNotifier to the app and exposes the current state (a map of meal types and their selections) to be consumed by widgets.
- Usage: Widgets can watch this provider to reactively display or modify the current meal selections.

DatabaseHelper

The DatabaseHelper class is a utility for managing meal data using Hive, a lightweight NoSQL database for Flutter. It provides methods to interact with the Hive database for CRUD (Create, Read, Update, Delete) operations on meal data (MealModel).

Properties:
- _hive: A Box<MealModel> that interacts with the Hive database storing MealModel objects.
- _box: A list holding all the MealModel objects from the Hive box.

Constructor:
- The class constructor initializes the helper but does not directly instantiate the Hive box, which is done lazily when a method is called.

Methods:

1. getMeals():
   - Retrieves all meals from the Hive database.
   - Converts the values from the Hive box into a list of MealModel and returns them.
   - Usage: Used to fetch all the meals from the database.

2. addMeal(MealModel meal):
   - Adds a new MealModel to the Hive database.
   - After adding, it returns the updated list of all meals.
   - Usage: Used to insert a new meal into the database.

3. removeMeal(String id):
   - Deletes a meal from the Hive box by searching for its id.
   - It removes the meal by finding its index and then deleting it using deleteAt.
   - Returns the updated list of meals after deletion.
   - Usage: Used to remove a specific meal from the database based on its id.

4. updateMeal(int index, MealModel meal):
   - Updates an existing meal at a specific index in the Hive box.
   - It replaces the meal at the given index with the new MealModel.
   - Returns the updated list of meals after the update.
   - Usage: Used to modify a meal's data in the database.

5. deleteAll():
   - Clears the _box list (in-memory cache), effectively removing all meals from the application memory.
   - Note: This method clears only the in-memory list (_box), not the actual Hive database. You might want to use _hive.clear() if you intend to remove all meals from the Hive database.
   - Usage: Used to clear the meal list in memory (could be extended to clear Hive data as well).

MealController

MealController is a state management class in Flutter using Riverpod's StateNotifier. It is responsible for managing the state of meals (List<MealModel>) and interacting with the local storage (Hive database) via a DatabaseHelper repository. This class provides CRUD operations for managing meals and synchronizes these changes with the UI.

Properties:
1. repo:
   - Type: DatabaseHelper?
   - Purpose: Acts as the repository for interacting with the Hive database. It's initialized by reading the providerHive to access the database.

2. ref:
   - Type: StateNotifierProviderRef

- Purpose: A reference to the provider container, allowing access to other providers and maintaining dependency injection.

Constructor:
- The constructor initializes the repo from the providerHive (which provides DatabaseHelper) and calls fetchMeals() to load existing meals into the state when the controller is created.

Methods:

1. fetchMeals():
- Purpose: Fetches all meals from the local storage (Hive) and updates the state with the list of MealModel objects.
- Usage: This method is called during the initialization of the controller to populate the state with stored meal data.

2. addMeal(MealModel meal):
- Purpose: Adds a new meal (MealModel) to the Hive database and updates the state with the new list of meals.
- Usage: Used to add a meal to the local storage and automatically reflect the change in the UI.

3. removeMeal(String id):
- Purpose: Removes a meal from the Hive database by its id and updates the state with the new list of meals.
- Usage: Called when a specific meal needs to be removed from the local storage and updates the UI accordingly.

4. updateMeal(int index, MealModel meal):
- Purpose: Updates an existing meal at a given index in the Hive database and updates the state with the new list of meals.
- Usage: Used to modify an existing meal in the local storage and ensure the UI reflects the changes.

Dependencies:

hooks_riverpod:

- A state management solution that enhances Riverpod by combining it with Hooks. It simplifies building reactive applications by improving state management.

flutter_hooks:
- Provides a set of utility hooks for Flutter. Hooks allow you to reuse stateful logic without needing to write classes. Works well with hooks_riverpod for managing state in functional components.

hive:
- A lightweight and fast NoSQL database for Flutter. Used for local data storage. Ideal for persisting meal data, user settings, or other lightweight data in mobile applications.

hive_flutter:
- Flutter-specific extensions for Hive, allowing seamless integration with the Flutter lifecycle and simplifying initialization, especially with Flutter widgets.

uuid:
- A package that generates unique identifiers (UUIDs). Typically used to create unique id values for objects like meals in your app.

intl:
- The Internationalization (i18n) library. Used for formatting dates, numbers, and currencies, as well as handling localization in Flutter applications. It helps you support multiple languages in your app.

Dev Dependencies:

build_runner:
- A command-line tool for generating code using Dart's build system. This is used to generate boilerplate code for packages like Hive and others that rely on code generation.

hive_generator:
- A code generator for Hive. It automatically generates type adapters, making it easier to work with custom data types in the Hive database.

flutter_lints:
- A package providing linting rules for Flutter projects. Helps maintain code quality by enforcing coding standards and best practices. ^4.0.0 is the version of the linting rules used.