Table of Contents

---

1. Overview

The Solar Monitoring App is a Flutter-based mobile application designed to provide users with real-time insights into their solar energy systems. The app allows users to visualize their power generation and consumption data through interactive graphs. It connects to a Dockerized API to fetch and display solar generation, household consumption, and battery consumption data.

2. Core Functionalities

- **Graph and Data Visualization**
    - **Solar Generation**: Displays a graph showing solar generation over time.
    - **Household Consumption**: Visualizes household energy consumption.
    - **Battery Consumption**: Shows battery energy usage over time.
        - Utilizes a line chart with the x-axis representing datetime and the y-axis showing energy values in watts or kilowatts.
        - Preloads data for seamless tab switching.
        - Allows unit switching between watts and kilowatts.
        - Allows date filtering.
- **Caching**
    - Fetched data is cached to prevent redundant network requests.
    - Users can clear cached data via an option in the app settings.
- **Error Handling**
    - Displays user-friendly messages for connectivity issues or server errors.
- **Additional Features**
    - **Pull-to-Refresh**: Allows users to manually refresh data on demand.
    - **Dark/Light Mode Support**: Dynamically adapts the theme based on system preferences.
    - **Data Polling**: Automatically refreshes data every 5 seconds for real-time updates.

3. Architectural Overview

The application follows a feature-first architecture with a clean separation of concerns:

- **Data Layer**:
    - Repository Pattern: Centralized logic for data fetching and transformation.
    - Dio for network requests to the backend server.
- **Business Logic Layer**:
    - BLoC Pattern: Handles state management for each data type (solar, house, battery).
- **Presentation Layer**:
    - Widgets and UI components are organized by feature for modularity.

4. State Management

- **BLoC (Business Logic Component)**:
    - **MonitoringBloc** handles events and states related to data visualization.
    - Separate **SolarBloc**, **HouseBloc**, and **BatteryBloc**, implements **MonitoringBloc**.
    - Singleton pattern is used for all instances to maintain a shared state across the app.
    - **ThemeCubit** handles changes the theme between a dark and light theme
    - **UnitPreferenceCubit** handles changing between watts and kilowatts
- **Events**:
    - **FetchMonitoringDataEvent**: Fetches data for a specific date and type.
    - **MonitoringDataPollEvent**: Periodically updates data.
- **States**:
    - Initial, MonitoringDataIsLoading, MonitoringData, MonitoringDataFailed.

5. Caching Strategy

- **Persistent Caching**: Drift-based SQLite database stores fetched data locally.

6. Error Handling

- **API Errors**: Network errors and HTTP errors are captured, and user-friendly error messages are displayed.
- **Empty Data**: If no data is available for a selected date, a placeholder message is shown.

7. Testing Approach

- **Unit Tests**: Validated core business logic in the repositories and BLoC classes. Tested transformations for unit conversions (watts ↔ kilowatts).
- **Widget Tests**: Ensured proper rendering of graph widgets and interactions. Tested DateFilter for accurate date selection.

8. How to Run

- **API Setup**
    - Download the provided resources from the link:

Build and run the Dockerized API:
Copy code
```
cd solar-monitor-api/
docker build -t solar-monitor-api .
docker run -p 3000:3000 solar-monitor-api
```

  - Verify API is running at http://localhost:3000/api-docs.
- **App Setup**

Clone the repository:
bash
Copy code
```
git clone https://github.com/chardoo/enpal-demo
cd solar-monitor-app
```
Install dependencies:
arduino
Copy code
```
flutter pub get
```
Run the app:
Copy code
```
Dart run build_runner build
flutter run
```

Note

The cache implementation has been commented out but it works as expected. However, to ensure the polling is functioning correctly, that section of the code has been committed. This is necessary to enable live updates, which would not occur if the same endpoint were accessed repeatedly. If I returned the cached response each time, the same data would be received without any updates