WEATHER APPLICATION

DEPI Project

- **Mahmoud ELDemerdash**
- **Nada Wahba**
- **Toka Abdelfatah**
- **Mohamed Haddad**
- **Mahmoud Mohamed**

OCTOBER 24, 2024

DEPI

Weather Application: Service Layer

1. Introduction

As part of the development team for the Weather Application, my primary responsibility was implementing the service layer of the app. This layer involves handling location services, managing API requests for weather data, and ensuring robust error handling mechanisms. The service layer is essential for ensuring that the app can accurately retrieve real-time weather data based on the user's location and display it in an intuitive manner.

In this report, I will outline my contributions to the project, including the core functionality I implemented, the technology used, and how the system handles errors and exceptions to provide a seamless user experience.

2. Location Handling

The first part of the service layer involves accurately fetching the user's location using GPS. The app requires the current city name to display localized weather data, which is retrieved using coordinates from the device.

Code Implementation:

```
class MyLocation {
  Future<Either<LocationFailure, String?>> getCurrentCity() async {
    try {
      Position position = await _determinePosition();

      List<Placemark> locationDetails =
        await placemarkFromCoordinates(position.latitude, position.longitude);

      String? city = locationDetails[0].subAdministrativeArea;
      return right(city);
    } on CurrentLocationException catch (e) {
      return left(LocationFailure(e.cause));
    } catch (e) {
      return left(LocationFailure("Error!!"));
    }
  }
}
```

Explanation:

- Position Retrieval: The function _determinePosition() obtains the user's current geographic coordinates.
- Reverse Geocoding: These coordinates are passed to placemarkFromCoordinates() to convert the latitude and longitude into a human-readable city name.
- Error Handling: Custom errors, such as CurrentLocationException, are thrown in cases where the location cannot be retrieved.

3. API Calling and Parsing

Another key responsibility was implementing the API integration, where the app communicates with the WeatherAPI to fetch weather data for a given city. I handled both the API requests and the parsing of the received data into structured models for further use in the app.

Code Implementation:

```
class WeatherService {
  final Dio dio = Dio();

  String apiKey = "your_api_key";
  String baseUrl = "http://api.weatherapi.com/v1";

  Future<Either<Failure, WeatherModel>> getWeatherForCity(String cityName) async {
    try {
      Response response = await dio.get(

"$baseUrl/forecast.json?key=$apiKey&q=$cityName&days=3&aqi=yes&alerts=yes");

      WeatherModel weatherModel = WeatherModel.fromJson(response.data);
      return right(weatherModel);
    } on DioException catch (dioException) {
      return left(ServerFailure.fromDioException(dioException));
    } catch (e) {
      return left(ServerFailure("Unexpected error!!"));
    }
  }
}
```

Explanation:

- API Call: The app sends a GET request to the WeatherAPI using the city name. The API key is authenticated, and a 3-day weather forecast is retrieved.
- Data Parsing: The JSON response from the API is parsed into a WeatherModel object, which contains the necessary weather information.
- Error Handling: If the API call fails, a DioException is caught, and a user-friendly error message is returned.

4.Notification Pushing

The implementatin of the notification system involves two primary components:
- `work_manager_services.dart`: Sets up background task scheduling using the `workmanager` package. This file is responsible for invoking the notification at the scheduled time.
- `notification_service.dart`: Handles the notification logic using `flutter_local_notifications`. This file is responsible for initializing the notification system, fetching weather data, and scheduling the notification to display at the correct time.

Code Walkthrough

1. 1. **Work Manager Setup**:

```dart
Workmanager().initialize(callbackDispatcher, isInDebugMode: true);
```

This code initializes the `WorkManager`, which allows background tasks to be scheduled and executed.

2. 2. **Callback Dispatcher Function**:

```dart
void callbackDispatcher() {
  Workmanager().executeTask((taskName, inputData) async {
    await LocalNotificationService.showDailyScheduledNotification();
    return Future.value(true);
  });
}
```

This function is responsible for scheduling the daily notifications using the `LocalNotificationService`.

3. 3. **Notification Scheduling**:

```dart
var scheduleTime = tz.TZDateTime(
  tz.local,
  currentTime.year,
  currentTime.month,
  currentTime.day,
  7,
);
```

The notification is scheduled for 7 AM local time. If the current time is after 7 AM, it schedules the notification for the next day.

5. Error Handling

Robust error handling is a critical part of any application, especially when dealing with external services like location data and APIs. I implemented a structured approach to handle various types of errors gracefully, ensuring that users receive clear feedback when something goes wrong.

Code Implementation:

```
abstract class Failure {
  final String errMsg;
  const Failure(this.errMsg);
}

class LocationFailure extends Failure {
  LocationFailure(super.errMsg);
}

class ServerFailure extends Failure {
  ServerFailure(super.errMsg);

  factory ServerFailure.fromDioException(DioException dioException) {
    switch (dioException.type) {
      case DioExceptionType.connectionTimeout:
        return ServerFailure('Connection timeout with server');
      case DioExceptionType.unknown:
        return ServerFailure('An unknown error occurred.');
      default:
        return ServerFailure('Something went wrong!');
    }
  }
}
```

Explanation:

- Custom Exceptions: I created custom classes for handling location-related failures (LocationFailure) and API server errors (ServerFailure).
- Dio Error Handling: The ServerFailure.fromDioException() method captures specific API-related errors (e.g., timeouts, bad responses) and generates appropriate error messages based on the type of error.
- User-Friendly Messages: Whether the issue is a network failure or an invalid API request, the app provides feedback in the form of clear, human-readable error messages.

5. Conclusion

In summary, my contributions to the Weather Application centered around developing a robust service layer, which includes:
- Accurately fetching the user's current location.
- Integrating with the WeatherAPI to retrieve weather data for specific cities.
- Implementing comprehensive error handling to manage both location and API-related issues.

These components ensure that the app provides real-time, accurate weather data and handles failures gracefully, resulting in a reliable and user-friendly application.

6. Technologies Used

- Dio for HTTP requests.
- WeatherAPI for fetching weather data.
- GeoLocation and GeoCoding for handling location services.
- flutter_local_notifications
- workmanager

Using Provider in Flutter for State Management

1. Introduction

In this report, I will explain how the Provider package is utilized for state management in my Flutter project. This project is focused on displaying weather information and allows users to select cities, view live weather data, and manage a list of cities for which they want to track weather updates.

2. The Role of Provider

Provider is a simple yet powerful state management tool in Flutter that helps efficiently manage and share data between different parts of the app. It allows for separating logic from UI components, ensuring cleaner code and better maintainability. In this application, Provider is used to manage the list of selected cities and fetch weather data for both user-selected cities and the user's current location.

3. CityProvider Class

The CityProvider class plays a central role in managing the data related to cities. It handles:

- Storing the list of cities selected by the user.
- Fetching weather data for those cities.
- Managing the user's current location and corresponding weather updates.

Below is an example of how the CityProvider is structured:

```
class CityProvider extends ChangeNotifier {
 List<String> selectedCities = [];
 String? userCity;

 void addCity(String city) {
  selectedCities.add(city);
  notifyListeners();
 }

 void removeCity(int index, BuildContext context) {
  selectedCities.removeAt(index);
  notifyListeners();
 }

 Future<void> fetchUserCity(BuildContext context) async {
  // Logic to get user location and fetch city data
  userCity = 'Current Location City';
  notifyListeners();
 }
}
```

4. Conclusion

Using Provider for state management in this application makes it easier to handle asynchronous data and maintain clean code. It helps ensure that the state is properly managed and updated, allowing for smooth user interaction.

UI Design and Implementation Report

Introduction

This report provides an overview of the UI design and its implementation in the Flutter application. It focuses on creating a user-friendly weather display interface, showcasing weather details like wind speed, hourly forecasts, and weather conditions using custom widgets and graphical elements. The design aims for simplicity, readability, and real-time weather updates.

1. Implementation of DismissibleCard

The DismissibleCard widget is responsible for displaying city information in a card format and allowing the user to swipe to remove a city from the list. Here's how the Provider is used in the DismissibleCard widget:

```
class DismissibleCard extends StatelessWidget {
 const DismissibleCard({super.key, required this.index});
 final int index;

 @override
 Widget build(BuildContext context) {
  CityProvider cityProvider = Provider.of<CityProvider>(context);
  String cityName = cityProvider.selectedCities[index];
  return Dismissible(
   key: UniqueKey(),
   direction: DismissDirection.endToStart,
   onDismissed: (direction) {
    cityProvider.removeCity(index, context);
   },
   child: CityCard(cityName: cityName),
  );
 }
}
```

2. Implementation of LiveWeatherCard

The LiveWeatherCard widget shows the weather information for the user's current location. It fetches the user's location and updates the weather data accordingly:

```
class LiveWeatherCard extends StatefulWidget {
 const LiveWeatherCard({super.key, required this.index});

 @override
 State<LiveWeatherCard> createState() => _LiveWeatherCardState();
}
```

```
class _LiveWeatherCardState extends State<LiveWeatherCard> {
 late CityProvider cityProvider;
 bool loading = true;

 @override
 void initState() {
  super.initState();
  SchedulerBinding.instance.addPostFrameCallback(
   (_) => _getUserLocation(),
  );
 }

 _getUserLocation() async {
  if (cityProvider.userCity == null) {
   await cityProvider.fetchUserCity(context);
   setState(() {
    loading = false;
   });
  }
 }

 @override
 Widget build(BuildContext context) {
  cityProvider = Provider.of<CityProvider>(context);
  return GestureDetector(
   onTap: () {
    _getUserLocation();
   },
   child: CityCard(
    cityName: cityProvider.userCity ?? '',
   ),
  );
 }
}
```

Main Features of the UI

1. Wind Speed Indicator with Circular Slider

A SleekCircularSlider widget is used to display the current wind speed. The slider gives a visual indication of wind speed with animated circular progress, creating an intuitive interface for users.

Customization:

The slider has minimum and maximum values of 0 and 100 respectively, with an initial value fetched dynamically from the weather model (weatherModel.current?.windKph).

The appearance of the slider is customized to include:
- Main Label showing the wind speed value in km/h.
- Bottom Label indicating that the data represents 'Wind Speed'.
- Progress Bar Colors transitioning smoothly from blue, cyan, to green, reflecting varying wind speeds.

2. Hourly Weather Forecast Section

A dynamic weather forecast is displayed in the form of hourly weather cards.

Widget: _buildWeatherHour creates a column-based layout showing:
- Hour: A textual representation of the time (e.g., '2 PM').
- Weather Icon: Small graphical icons are used to visually represent the weather condition at that hour.
- Temperature: Displayed in degrees, dynamically updated based on the weather data.

3. Daily Forecast Section

The UI also contains a detailed daily weather forecast.

Widget: _buildForecastDay builds a row layout showing:
- Day Name: Such as 'Monday' or 'Tuesday.'
- Weather Icon: Represents the weather condition for that day.
- High and Low Temperatures: Shows the maximum and minimum temperatures for the day, ensuring users have complete information about the day's forecast.

4. Dynamic Weather Backgrounds

The method _getWeatherImage() dynamically selects appropriate background images based on the weather condition (e.g., sunny, cloudy, rain, fog). This helps create a more immersive experience, as users see visuals that match the current weather.

5. Weather Icons for Different Conditions

The UI utilizes _getWeatherIcon() to select the correct weather icon based on specific weather conditions (e.g., 'Rain,' 'Cloud,' 'Sunny'). Each condition triggers the display of a different icon to make it clear to users what type of weather to expect.

Visual and Interactive Elements

Colors and Fonts

The design uses consistent colors and font styles, especially white text on a dark background to ensure clarity and readability.

Text styles are customized to give clear visual hierarchy:
- Main Labels: Font size of 22 for key information like wind speed.
- Secondary Labels: Smaller font size of 14 for supplementary information (e.g., labels under the wind speed slider).

Responsiveness and Adaptability

The UI elements are flexible and adjust smoothly to screen sizes due to the use of widgets like Flexible, ensuring the app looks good on a wide range of devices.

Graphical Assets

The icons and images (like weather icons and background images) are stored locally in the app (assets/images/ and assets/icons/) and loaded dynamically based on the weather conditions, which reduces network load and ensures quick display updates.

Code Integration and Best Practices

Reusable Widgets

Widgets such as _buildWeatherHour and _buildForecastDay are created to ensure code modularity and reusability. This approach helps keep the codebase clean and makes future updates easier.

Custom Animations

The circular slider widget includes smooth animations (animationEnabled: true), enhancing the user experience by making data transitions more fluid.

Conclusion

The UI section of this Flutter application is designed to be visually appealing and user-friendly, presenting critical weather information in a simple, intuitive layout. By leveraging Flutter's flexibility, we created a responsive interface with clear, dynamic visuals that adapt to real-time weather data.

Future Improvements

Consider adding more weather data visualizations like precipitation, humidity, and atmospheric pressure for a more comprehensive weather overview.
Explore the possibility of including light and dark themes to enhance user experience in different lighting conditions.