jai

- **App Architecture**:
  - The app is built using **Kotlin** and **Jetpack Compose** for the UI.
  - We followed an **MVVM (Model-View-ViewModel)** architecture for maintaining a clean separation of concerns:
    - **Model**: Handles data and logic.
    - **View**: The UI components created using **Jetpack Compose**.
    - **ViewModel**: Manages UI-related data lifecycle and integrates with the Model for fetching data.
- **Login Page Animation**:
  - The login page uses **Jetpack Compose animations** to make transitions smoother. For example, `animateFloatAsState` was used to animate the logo and buttons fading in and scaling up.
  - The transition happens when the user launches the app, making it feel interactive and responsive.
  - Used `LaunchedEffect` and `AnimatedVisibility` to control the start and end of the animation when the screen is first shown.
- **Weather Data Fetching**:
  - Integrated **OpenWeatherMap API** to fetch real-time weather data. Here's how:
    - **Retrofit** is used for making HTTP requests. The data is fetched asynchronously using **Kotlin Coroutines** to avoid blocking the UI thread.
    - Weather data like temperature, humidity, and conditions are fetched as JSON objects.
    - The data is then parsed into Kotlin **data classes** (e.g., `WeatherResponse` and `Main` classes).
    - The **ViewModel** calls the API and updates the UI through **LiveData** or **StateFlow** to ensure the UI stays in sync with the data.

  Example Code:

```
kotlin
CopyEdit
// Retrofit API call
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.openweathermap.org/data/2.5/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

val service = retrofit.create(WeatherService::class.java)

// Fetch weather data asynchronously
val weatherResponse = service.getWeather("Sathyamangalam", "API_KEY")
```

  - **StateFlow** is used to **collect data** and update the UI whenever the weather data is fetched successfully or fails.

- **Real-Time Weather Display**:
  - The **UI** is updated in real-time using **StateFlow** or **LiveData**:
    - Whenever new weather data is received, **StateFlow** emits the updated data to the **ViewModel**.
    - The **ViewModel** updates the **Compose UI** (e.g., temperature, humidity, and weather conditions) dynamically.

  Example Code for UI:

```
kotlin
CopyEdit
// Observe weather data in Composables
val weatherData by viewModel.weatherData.collectAsState()

Text(text = "Temperature: ${weatherData?.main?.temp} °C")
Text(text = "Humidity: ${weatherData?.main?.humidity}%")
```

- **Dynamic UI Updates**:

- The weather section of the app is designed to update automatically. Every time the **weather data** is fetched from the API, the UI is **recomposed** to show the latest values. This ensures the **user always sees the latest weather conditions**.