Project Overview

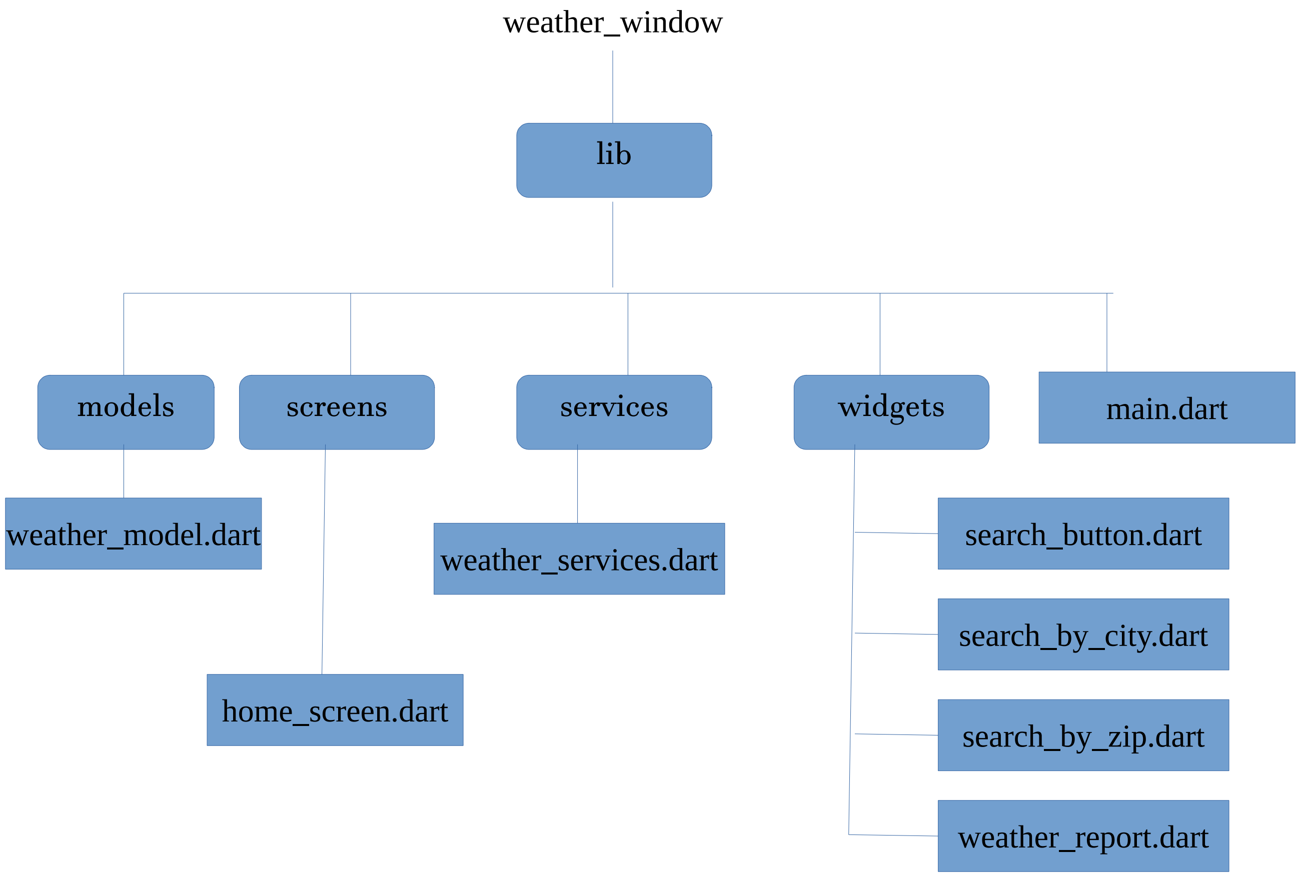
Project Name: Weather Window

- **Description:** It a mobile app which is created using Flutter framework using dart. This app works as a weather report app which gets current weather report using OpenWeather API. It displays current weather information using live API data.
- **Goal:** My primary goal of this project is learning. I wanted to learn about API, how flutter works, and state changes using flutter. I have learned how to handle API, how to get information using API and store it as a model and then display it with a good UI.
- **Tech Stack:** Flutter(Framework), Dart (language), API- https://openweathermap.org (free weather API).

Features

- Search weather by city name.
- Search weather by zip code and country code.
- Clean file structure
- Display weather data like temperature, feels like, cloudiness, min/max temperature, wind speed, Humidity etc.
- Weather Icon changes.
- Error handling (eg., empty text fields can not submitted).

Project Structure



API

- In this project I have used - https://openweathermap.org free current weather API
- In this API I have used two methods →
  - Search by city name
  - Search by zip code and country code

- For city name we have used: `https://api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}` &units=metric
  - Here units= metric means Units of measurement. `standard`, `metric` and `imperial` units are available. If you do not use the `units` parameter, `standard` units will be applied by default.
  - q = city name, state code and country code divided by comma,You can specify the parameter not only in English. In this case, the API response should be returned in the same language as the language of requested location name if the location is in our predefined list of more than 200,000 locations.

- For Zip code and country code:
  - `https://api.openweathermap.org/data/2.5/weather?zip={zip code},{country code}&appid={API key}` &units=metric
  - Here zip code will be the zip code you want to search (like 741221).
  - And country code will be country code, like India = IN

<u>Json data I got:</u>

`coord`

  - `coord.lon` Longitude of the location
  - `coord.lat` Latitude of the location
- `weather` (more info Weather condition codes)
  - `weather.id` Weather condition id
  - `weather.main` Group of weather parameters (Rain, Snow, Clouds etc.)
  - `weather.description` Weather condition within the group. Please find more here. You can get the output in your language. Learn more
  - `weather.icon` Weather icon id
- `base` Internal parameter
- `main`
  - `main.temp` Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit
  - `main.feels_like` Temperature. This temperature parameter accounts for the human perception of weather. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit
  - `main.pressure` Atmospheric pressure on the sea level, hPa

  - `main.humidity` Humidity, %
  - `main.temp_min` Minimum temperature at the moment. This is minimal currently observed temperature (within large megalopolises and urban areas). Please find more info here. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit
  - `main.temp_max` Maximum temperature at the moment. This is maximal currently observed temperature (within large megalopolises and urban areas). Please find more info here. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit
  - `main.sea_level` Atmospheric pressure on the sea level, hPa
  - `main.grnd_level` Atmospheric pressure on the ground level, hPa
- `visibility` Visibility, meter. The maximum value of the visibility is 10 km
- `wind`
  - `wind.speed` Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour
  - `wind.deg` Wind direction, degrees (meteorological)
  - `wind.gust` Wind gust. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour
- `clouds`
  - `clouds.all` Cloudiness, %
- `rain`
  - `1h`(where available)Precipitation, mm/h. Please note that only mm/h as units of measurement are available for this parameter
- `snow`
  - `1h`(where available) Precipitation, mm/h. Please note that only mm/h as units of measurement are available for this parameter
- `dt` Time of data calculation, unix, UTC
- `sys`
  - `sys.type` Internal parameter
  - `sys.id` Internal parameter
  - `sys.message` Internal parameter
  - `sys.country` Country code (GB, JP etc.)
  - `sys.sunrise` Sunrise time, unix, UTC
  - `sys.sunset` Sunset time, unix, UTC
- `timezone` Shift in seconds from UTC
- `id` City ID. Please note that built-in geocoder functionality has been deprecated. Learn more here
- `name` City name. Please note that built-in geocoder functionality has been deprecated. Learn more here
- `cod` Internal parameter

Code Explanation

main.dart

- This file is the **entry point** of the Weather Report App. It defines how the app starts and sets up the main widget tree.
- The `main()` function is the **starting point** of every Flutter app.
- `runApp()` launches the app and takes a widget (here, `MyApp`) as its root.
- `MyApp` is a **StatelessWidget**, meaning it doesn't store or update any state by itself.
- The `MaterialApp` widget provides app-wide features like theming, navigation, and routes.
- `home: HomeScreen()` sets the **first screen** to be displayed when the app starts.
- This file initializes the Flutter app and defines the root widget (`MyApp`), which loads the main `HomeScreen` where all weather features are displayed.

home_screen.dart

- Home Screen is a screen that is the main screen where all the things displays.

```
bool zipMode = false; - this is a bool flag to toggle the search by city or
search by zip.

 String cityName = ""; // City name when searching using city
 String zipCode = ""; // zip code when searching using zip mode
 String countryCode = ""; // // country code when searching using zip mode

 final WeatherService _weatherService = WeatherService(); - Instance of
weatherService which will be responsible for calling the API
 WeatherModel?  _weatherData; the model which we will be getting after calling
the API
  bool _isLoading =  false; - to inform if the API is in loading state or not

void updateCity(String newCity) {} - this function updates the cityName with
newCity which it will get from SearchByCity widget

void updateZip(String newZipCode, String newCountryCode) {} - this function
updates the zip and countryname which it will get from SearchByCity widget.

  Future<void> searchWeather() async {} this function will actually call the API
service.
   Here Future<void> means :
      In Dart, a Future represents a value that will be available later, after
      an asynchronous operation finishes (like fetching data from the
      internet).
      The <void> means the function doesn't return any value once it completes
      — it just performs an action (e.g., updating the UI or variables).
```

Example analogy: It’s like saying “I’ll go get the data and let you know when I’m done — but I won’t bring anything back directly.”

Here async means :

The async keyword tells Dart that this function will contain **asynchronous operations**, such as network requests or waiting for data.
It allows you to use the await keyword inside the function — so you can **pause execution** until a task (like the API call) is completed.
This prevents blocking the main UI thread and keeps your app **smooth and responsive.**

```
      if (!zipMode && cityName.isEmpty) {
    // if Zip Mode is not true means Searching by city and then City is empty show an
warning message and return without calling the API
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text("Please enter city name")),
    ); //showing the SnackBar after getting empty cityName
    return; //return the function don't have to call the API
  } else if (zipMode && (zipCode.isEmpty || countryCode.isEmpty)) {
    //in Zip Mode if the zip and country is empty we don't need to call the API
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text("Please enter ZIP code and Country code")),
    ); //showing warning about the required fields
    return; // return without calling the API
  }
```

- ○ This condition prevents API calls when required inputs are missing and alerts the user using SnackBars.

- Now if the all the condition above is false then we can cal the API so now we will isLoading = true;

  ```
  setState(() {    _isLoading = true;   });
  ```

  - `setState()` is a Flutter method used to **update the UI** when the state of a widget changes.
  - Here, it sets `_isLoading` to `true` and **rebuilds the widget** so the app can show a loading indicator or spinner on the screen.

- After that we will call the fetching Service API service according to the zipmode like if Zip Mode = true call fetch by zip if not search by city and store the response as Weather Model and display to as UI.

- In appbar we have created a IconButton which will help to toggle the zip mode like either we have to switch between Search by city or search by zip and also according to that in ListView we have to swirch Search by city or Search by Zip

- and then if we have the data we can call WeatherReport widget class which is responsible for weather report UI , where we have to pass the Weather Model object.

weather_service.dart

- Weather service class is responsible for calling the api handling the api returning the Weather Model after successfully fetching the API.
- Here we have two function:

```
 Future<WeatherModel?> fetchWeatherByCityName(String city) async {}
 Future<WeatherModel?> fetchWeatherByZipCode(String zipCode,String 
countryCode) async {}
```

```
First Function :   Future<WeatherModel?> fetchWeatherByCityName(String city) 
async {} --- this is calling search by city name with a parameter of city (name)
```

```
final url = Uri.parse('$baseUrl?q=$city&appid=$apiKey&units=metric'); --- 
this is the url which will be called by this function.
```

```
 final response = await http.get(url); ---- http.get returns a 
Future<response> so we have to put await.
```

- `    if (response.statusCode == 200) {} --- `This line checks if the API request was **successful** — a status code of **200** means the server responded **OK** with valid data.

```
 final data = json.decode(response.body); ---- This line converts the API's 
JSON response (response.body) into a Dart map object (data) so the app can 
easily access and use the weather information.
```

```
 return WeatherModel.fromJson(data); ---- This line creates a WeatherModel 
object from the parsed JSON data using the fromJson factory method, making 
it easy to access weather details through the model's properties.
```

- Like this the same way the second method works for search by zip code.

weather_model.dart

- This class defines the **structure** of the weather data used in the app.
  It helps organize all the values received from the weather API into a single, easy-to-use object.
- Each property (like `cityName`, `temperature`, `humidity`, etc.) stores a **specific piece of weather information**.
- The `WeatherModel` **constructor** initializes these properties.
- The **factory constructor** `WeatherModel.fromJson()` takes the **JSON response** from the API and **converts it into a Dart object**.
- It safely extracts values (like `main.temp`, `wind.speed`, `sys.country`) — using `??` and `?.` to prevent errors if some fields are missing.

search_by_city.dart

This widget allows the user to **search for weather information by entering a city name.** It's a **stateful widget** that manages its own text input and communicates changes back to the parent widget using a **callback function.**

- class SearchByCity extends StatefulWidget {final Function(String) onCityChanged;
  - SearchByCity is a **stateful widget** since the text field state (input) can change.
  - onCityChanged is a **callback function** — it sends the typed city name back to **the parent widget** when the user submits or taps outside the field.
- class _SearchByCityState extends State<SearchByCity> { final TextEditingController _cityController = TextEditingController(); String cityName = "";
  - _cityController manages and listens to the text typed into the field.
  - `cityName` temporarily stores the typed city name locally.
- @override void dispose() { _cityController.dispose();   super.dispose(); }
  - This method **frees up memory** by disposing of the text controller when the widget is removed from the screen.
- void _notifyParent() { final cityName = _cityController.text.trim(); widget.onCityChanged(cityName); FocusScope.of(context).unfocus(); }
  - Gets the entered city name.
  - Sends it to the **parent widget** using the callback (`onCityChanged`).
  - Unfocuses the text field to hide the keyboard after submission.
- TextField(controller: _cityController, onSubmitted: (value) => _notifyParent(), onTapOutside: (event) => _notifyParent(),
  - `controller` links the text field to `_cityController`.
  - `onSubmitted` triggers when the user presses "Enter".
  - `onTapOutside` triggers when the user taps outside the text field.
  - Both call `_notifyParent()` to send the city name to the parent.
- decoration: InputDecoration(
  hintText: 'Enter city name...',
  prefixIcon: const Icon(Icons.search, color: Colors.blueAccent),
  filled: true,
  fillColor: const Color.fromARGB(255, 236, 241, 238),

    - Provides a clean, modern design with rounded borders and a blue search icon.
    - The field has a light background and smooth focus effects for better user experience.

search_by_zip.dart

The `SearchByZip` widget allows the user to search weather information using a ZIP code and a country code.
It is a stateful widget that manages two text fields — one for ZIP and one for country — and uses a callback function to send these values to the parent widget.

- class SearchByZip extends StatefulWidget { final Function(String zip, String country) onZipChanged;
    - The widget takes a **callback function** `onZipChanged` as a parameter.
    - This function is called whenever the user submits or exits the input fields, sending the ZIP and country codes back to the parent widget.
- final TextEditingController _zipController = TextEditingController();
- final TextEditingController _countryController = TextEditingController();
    - These controllers handle the text entered into each field (ZIP and Country).
    - They make it easy to read or modify the user's input when needed.
- void _notifyParent() { widget.onZipChanged(_zipController.text.trim(), _countryController.text.trim(),); FocusScope.of(context).unfocus();}
    - This function **collects the user's input** from both text fields.
    - It **calls the parent widget's callback** (`onZipChanged`) to send the ZIP and country data.
    - Finally, it **unfocuses** the text fields to hide the keyboard once input is done.
- `Column( children: [ TextField(...), // ZIP code input SizedBox(height: 12), TextField(...), // Country code input ], )`
    - The UI contains **two text fields** arranged vertically:
    - **ZIP Code Field** — lets the user type a postal code.
    - **Country Code Field** — accepts a two-letter country code (like `US`, `IN`, `GB`).
- onSubmitted: (_) => _notifyParent(),
- onTapOutside: (event) => _notifyParent(),
    - When the user **presses Enter** or **taps outside** the text fields, the app automatically triggers `_notifyParent()` to update the parent widget with the latest inputs.
- **Design and Styling**
    - Both text fields use:

    - A light gray background (`fillColor`) for a clean look.
    - Rounded corners (`borderRadius: BorderRadius.circular(20)`).
    - Blue accent icons (`Icons.pin_drop` and `Icons.flag_outlined`) for a polished UI.
    - Highlighted borders when focused to indicate active input.

search_button.dart

The `SearchButton` is a stateless widget that creates a styled button for triggering the weather search API call when tapped.

- `final VoidCallback onSearchPressed;`
    - `onSearchPressed` is a **callback function** passed from the parent widget.
    - It defines what happens when the user clicks the button (in this case — calls the weather API).
- ElevatedButton.icon(onPressed: onSearchPressed, icon: const Icon(Icons.location_on_outlined, color: Colors.white), label: const Text("Search Weather", ...),)
    - Uses **`ElevatedButton.icon`** to create a button with both an icon and text.
    - The **location icon** visually represents searching weather by location.
    - The `onPressed` property executes the callback function when the button is tapped.
- `style: ElevatedButton.styleFrom( backgroundColor: Colors.blueAccent, padding: EdgeInsets.symmetric(vertical: 16), shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(20)), )`
    - Styled with a **blue accent background** and **rounded corners** for a modern look.
    - Adds **elevation and shadow** to make the button stand out visually.
    - The button text and icon are both white for clear contrast.

`SearchButton` is a reusable stateless widget that displays a stylish "Search Weather" button and triggers the provided callback (`onSearchPressed`) to perform the weather API request when clicked.

weather_report.dart

The `WeatherReport` widget is a stateless widget that displays the complete weather details fetched from the API in a visually appealing and organized layout.
It receives a `WeatherModel` object as input, extracts its data, and renders the city name, temperature, weather icon, and various atmospheric details.

- `final WeatherModel weather;`
    - The widget accepts a `WeatherModel` instance containing all weather details such as temperature, humidity, wind speed, and more.
    - Being **stateless**, it only displays the passed data and doesn't manage any internal state.
- Container(
  decoration: BoxDecoration(
  gradient: LinearGradient(
  colors: [Colors.blue.shade400, Colors.blue.shade800],
  begin: Alignment.topCenter,
    end: Alignment.bottomCenter,
              ),
              borderRadius: BorderRadius.circular(24),
               boxShadow: [BoxShadow(...)]
              ),
)
    - Uses a **blue gradient background** to represent sky/weather visually.
    - Adds **rounded corners** and **shadow effects** for a polished UI.
    - Provides consistent **padding and spacing** for readability
- Text('${weather.cityName}, ${weather.country}')
- Image.network('https://openweathermap.org/img/wn/${weather.iconId}@4x.png')
- Text('${weather.temperature.toStringAsFixed(1)}°C')
- Text(weather.description.toUpperCase())
- Text('Feels like ${weather.feelsLike.toStringAsFixed(1)}°C')
-
    - Displays **city name and country** at the top.
    - Loads a **weather icon** dynamically from OpenWeatherMap.
    - Shows **temperature**, **description**, and **feels-like temperature** in large, bold text.
    - Data formatting (e.g., `toStringAsFixed(1)`) ensures consistent numeric precision.
- Container(

color: Color.fromARGB(48, 255, 255, 255),

child: Column(children: [

  _buildWeatherStatRow(...), // Min/Max Temp

  _buildWeatherStatRow(...), // Humidity

  _buildWeatherStatRow(...), // Wind Speed

  _buildWeatherStatRow(...), // Cloudiness

```
    _buildWeatherStatRow(...), // Pressure
  ]),
)
```

  - Displays detailed metrics (min/max temperature, humidity, wind speed, etc.).
  - Each stat is separated by a **Divider** for clarity.
  - A semi-transparent background improves contrast and visual balance.
- `Widget _buildWeatherStatRow({required IconData icon, required String label, required String value})`
  - A reusable function that builds a **single row** of weather data with:
  - An **icon** (e.g., thermometer, wind, cloud)
  - A **label** (like "Humidity")
  - The **value** (e.g., `75%`)
  - Keeps the code clean and modular, avoiding repetition.

`WeatherReport` is a stateless, reusable UI component that takes weather data from `WeatherModel` and presents it in a modern, responsive, and easy-to-read card layout with temperature, weather icons, and detailed statistics.