Weather Application Documentation

Architecture Overview

This weather application is structured using **Clean Architecture** principles, which separates the app into three main layers: **Data Layer**, **Domain Layer**, and **Presentation Layer**. The goal of this architecture is to create a clear separation of concerns, making the codebase more maintainable and scalable.

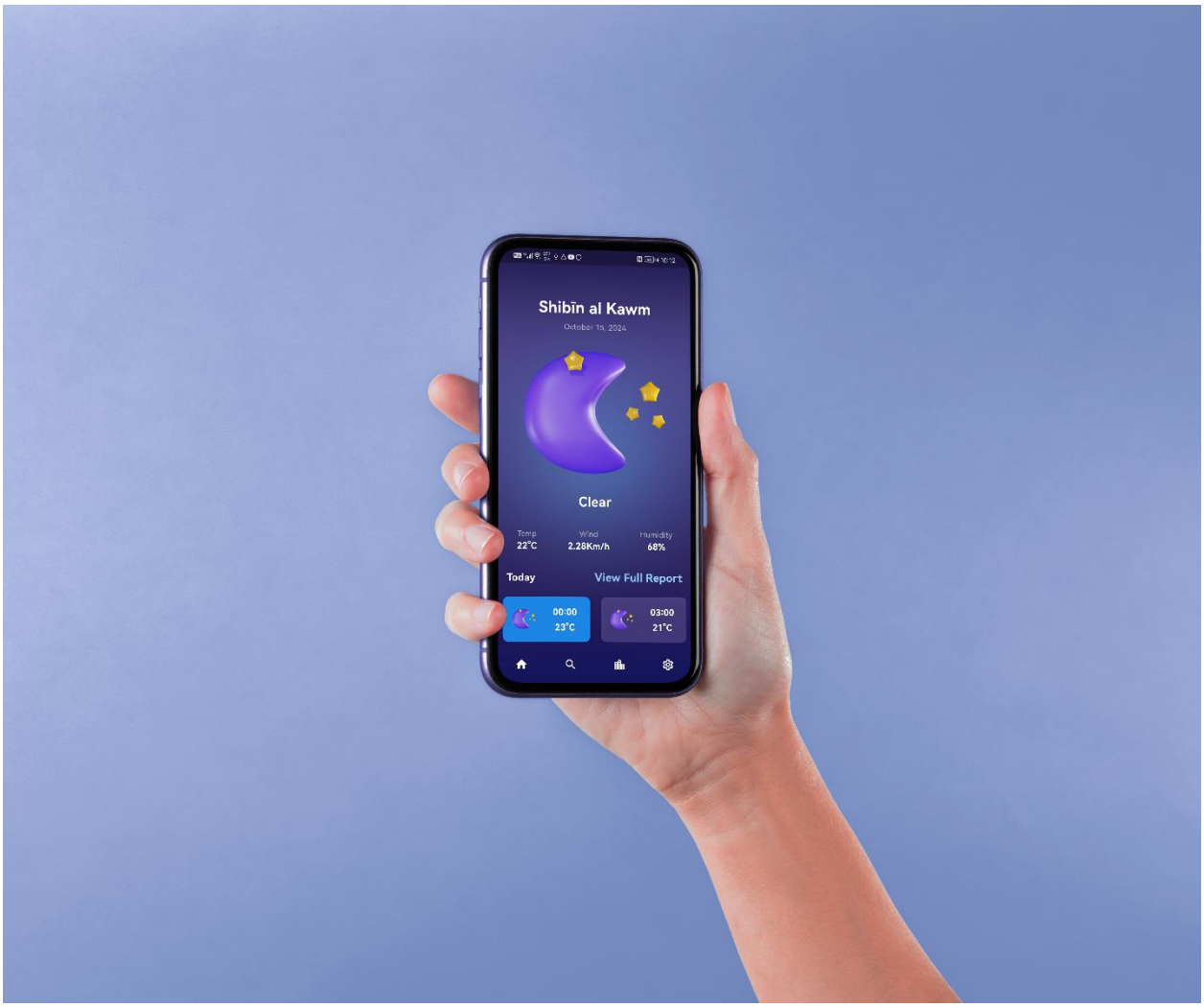
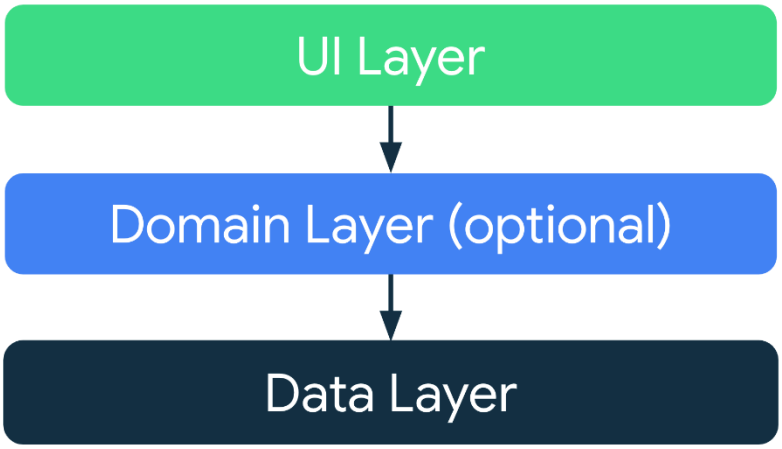


Table of Contents



Data Layer

The **Data Layer** is responsible for interacting with external APIs, local storage, and handling other data sources. It includes API services, model classes, and repository implementations.

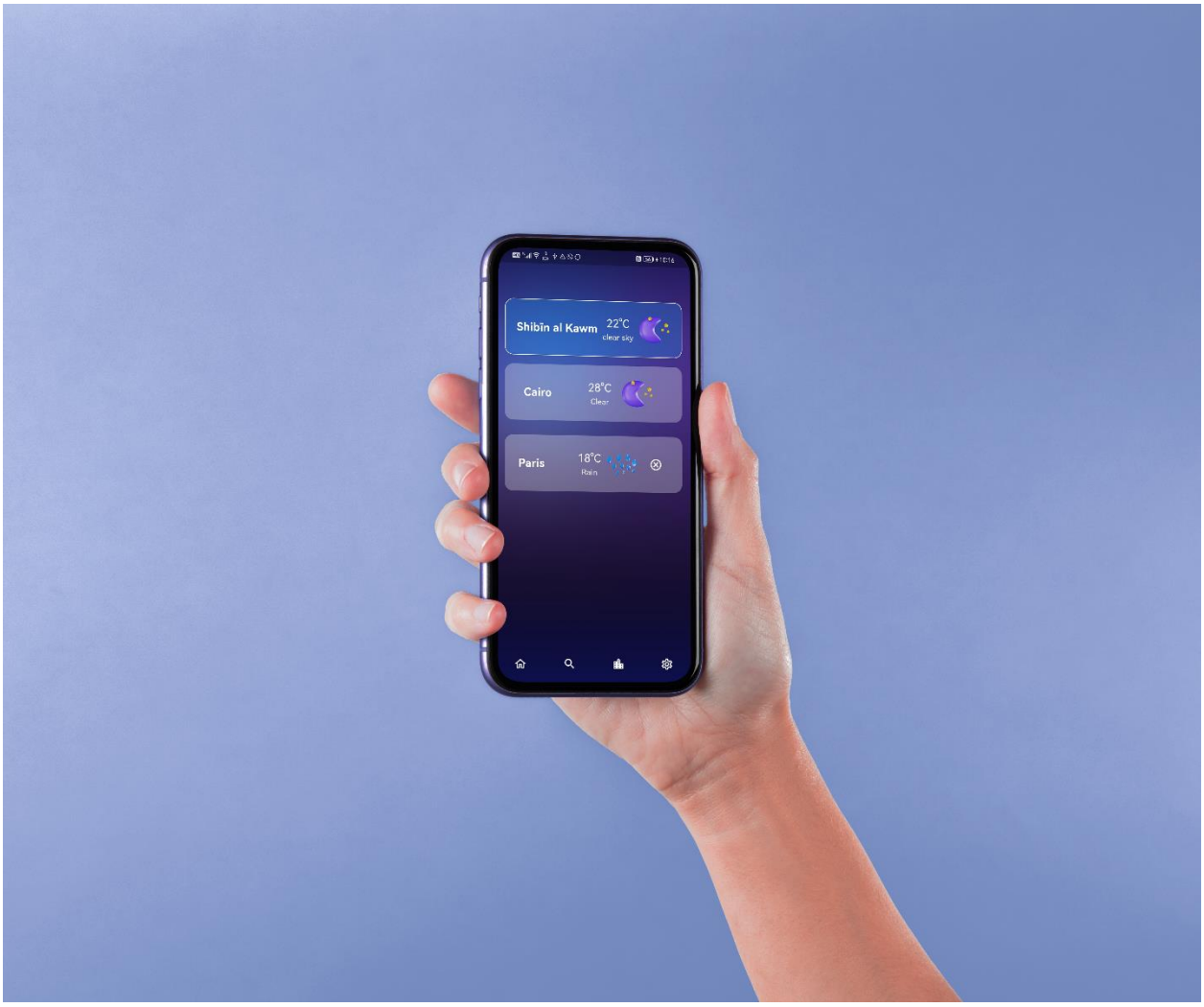
Geolocator Repository Implementation

The GeolocatorRepositoryImplementation handles the retrieval of the user's current location using the Geolocator package. This data is then used by other services to fetch weather information based on the user's location.

Saved Cities Repository Implementation

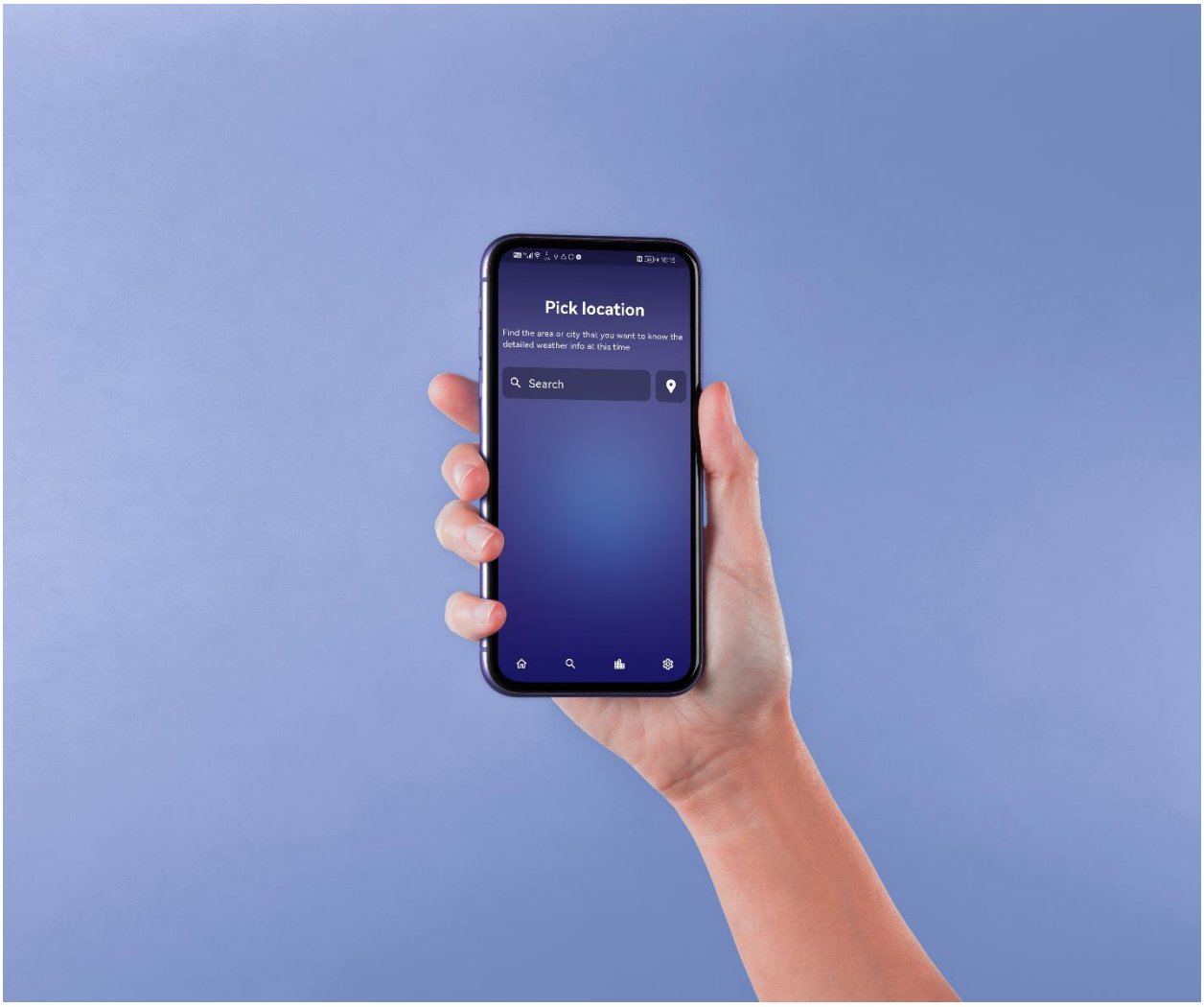
The SavedCitiesRepoImplementation is responsible for updating the weather information of saved cities every 3 hours. If the last update was less than 3 hours ago, the cached data is used.

```
class SavedCitiesRepoImplementation extends SavedCitiesRepo {
  @override
  Future<Either> updateCity(String cityName) async {
    // Check and update the weather for saved cities
  }
}
```



Suggested Cities Repository Implementation

The SuggestedCitiesRepoImpl retrieves a list of suggested cities based on a user's search query and the city name based on their geographical location.

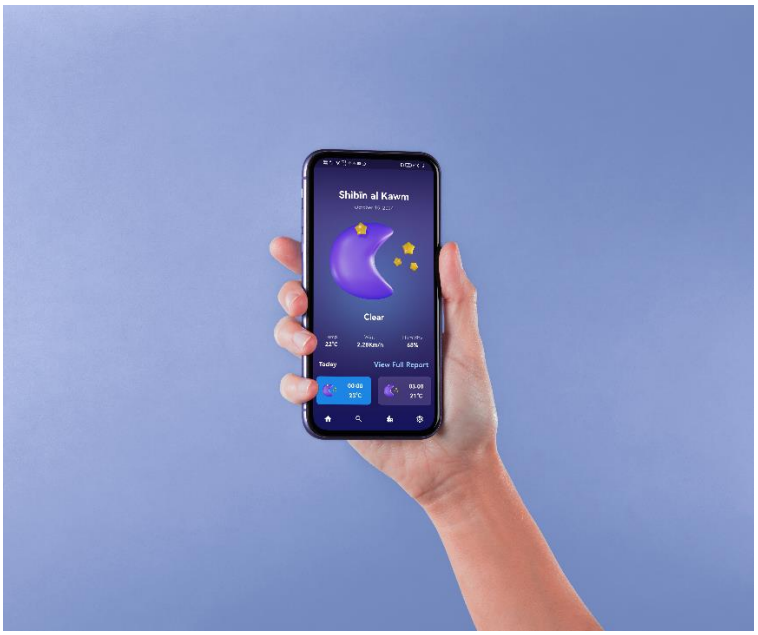


```
class SuggestedCitiesRepoImpl extends SuggestedCitiesRepo {
  @override
  Future<Either<List<SuggestedCity>, String>> getSuggestedCities(String query) async {
    // Fetch suggested cities based on user input.
  }

  @override
  Future<Either<List<SuggestedCity>, String>> getCityNameByPosition() async {
    // Retrieve city name by geographical position.
  }
}
```

Weather Repository Implementation

The WeatherRepoImplementation handles the fetching of both current weather data and weather forecasts based on either a city name or a geographical location. It also includes methods to notify users about weather updates.

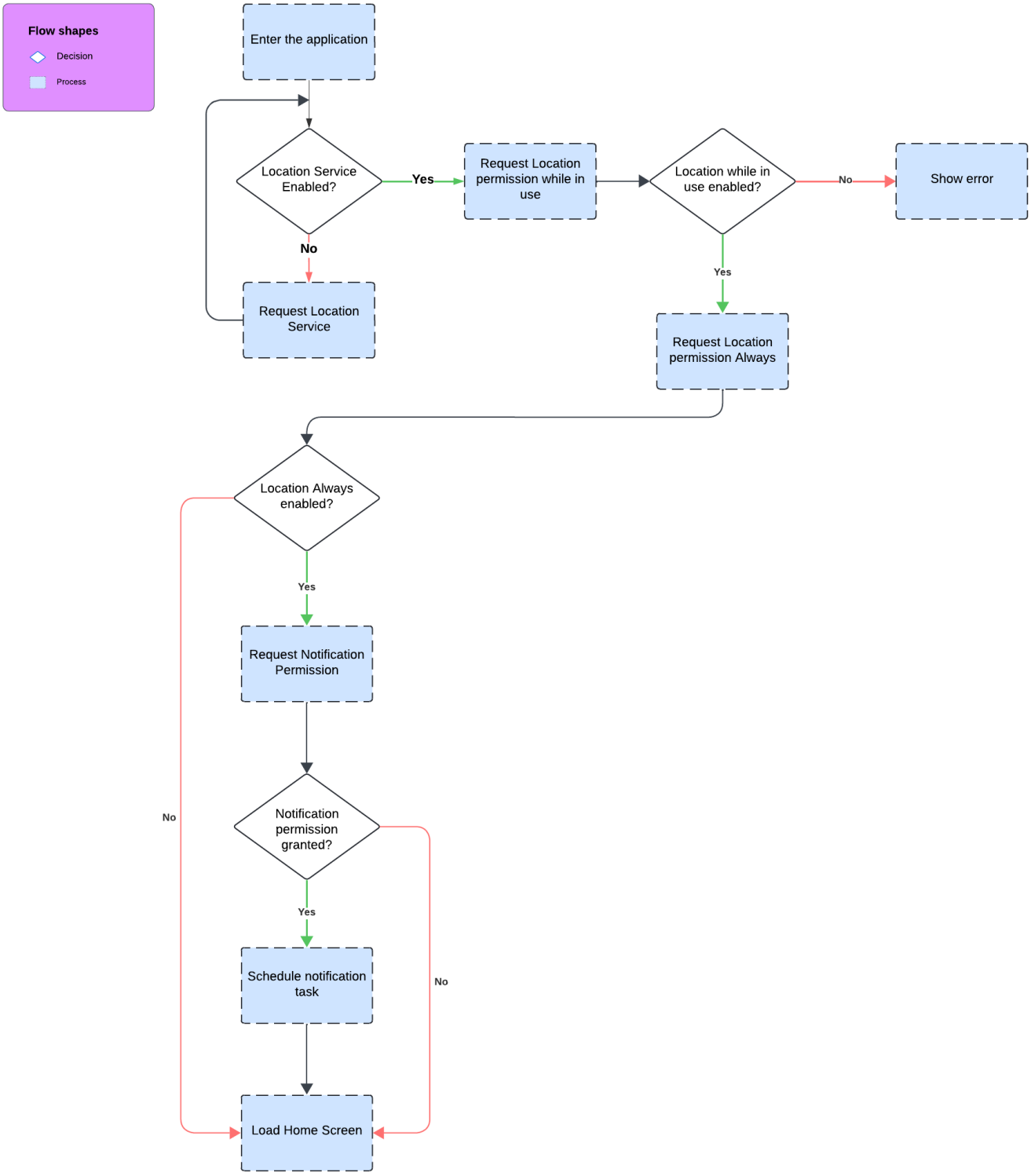
```
class WeatherRepoImplementation implements WeatherRepo {
  @override
  Future<Either<CurrentWeather, String>> getCurrentWeatherByCity(String city) async {
    // Fetch current weather by city name.
  }

  @override
  Future<Either<ForecastWeather, String>> getForecastByCity(String city) async {
    // Fetch weather forecast by city.
  }
  @override
  Future<Either<CurrentWeather, String>> getCurrentWeatherByPosition(Position position) async {
    // Fetch current weather by geographical location.
  }
}
```



Permissions Repository Implementation

The PermissionsRepoImplementation manages user permissions for location and notification services, ensuring that the application has the necessary access to provide weather data and notifications.

```
class PermissionsRepoImplementation extends PermissionRepo {
  @override
  Future<bool> checkLocationWhenInUse() async {
    // Check 'Location When In Use' permission.
  }
  @override
  Future<void> requestLocationPermission() async {
    // Request location permission from the user.
     }
}
```


Flow shapes
Decision
Process
Enter the application
Location Service Enabled?
Yes
No
Request Location permission while in use
Request Location Service
Location while in use enabled?
No
Show error
Yes
Request Location permission Always
Location Always enabled?
Yes
No
Request Notification Permission
Notification permission granted?
Yes
No
Schedule notification task
Load Home Screen

Domain Layer

The **Domain Layer** contains the business logic of the application, such as use cases. These are application-specific and responsible for executing the core functionality by interacting with repositories.

Use Cases

1. **Get City Name by Position**
    - Retrieves the name of the city based on the user's current location.

Get Current Weather by City

- Fetches the current weather for a given city name.

Work Manager Use Case

- Schedules background tasks like weather notifications using the WorkManager.

Presentation Layer

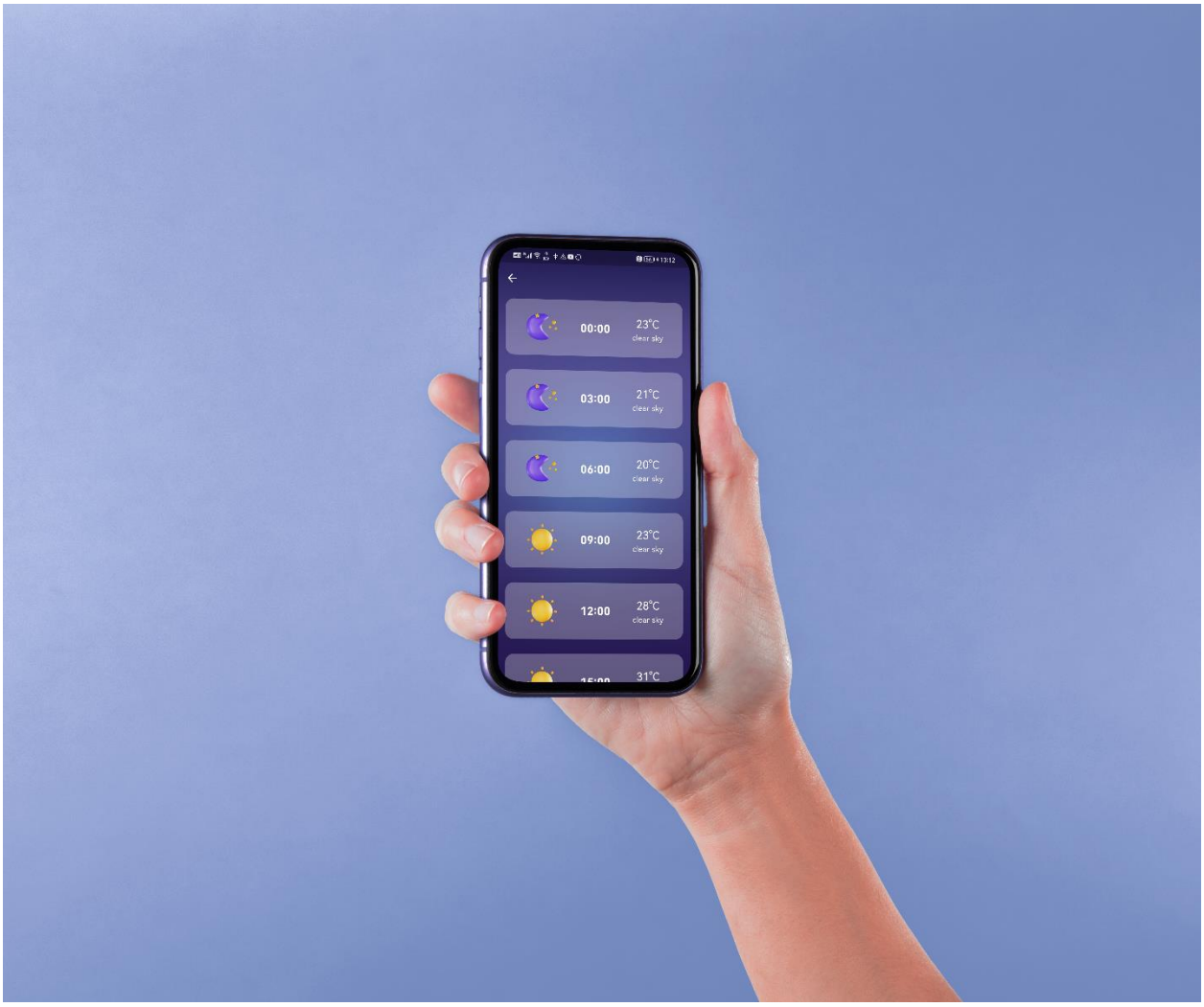
The **Presentation Layer** manages the UI and user interaction, utilizing the **Cubit** state management approach.

Cubit State Management

ForecastWeatherCubit

The ForecastWeatherCubit fetches and manages weather forecasts, emitting the appropriate states (loading, success, or failure) depending on the outcome of the request.

```
class ForecastWeatherCubit extends Cubit<ForecastWeatherState> {
  ForecastWeatherCubit() : super(ForecastWeatherInitial());
  Future<void> getForecastWeatherByCity(String city) {
    _fetchWeather(
      () => getIt<WeatherRepo>().getForecastByCity(city),
      onSuccess: (data) => emit(ForecastWeatherSuccess(forecastWeather: data)),
      onError: (error) => emit(ForecastWeatherFailed(error: error)),
    );
  }
}
```



GetCityNameCubit

The GetCityNameCubit retrieves and manages the city name based on the user's position, using the corresponding use case from the **Domain Layer**.

```
class GetCityNameCubit extends Cubit<GetCityNameState> {
  GetCityNameCubit() : super(GetCityNameInitial());
  Future<void> getCityNameByPosition() async {
    final response = await GetCityNameByPositionUsecase().call();
    response.fold((data) {
      emit(GetCityNameSuccess(suggestedCities: data));
    }, (error) {
      emit(GetCityNameFailed(errorMessage: error));
    });
  }}
```

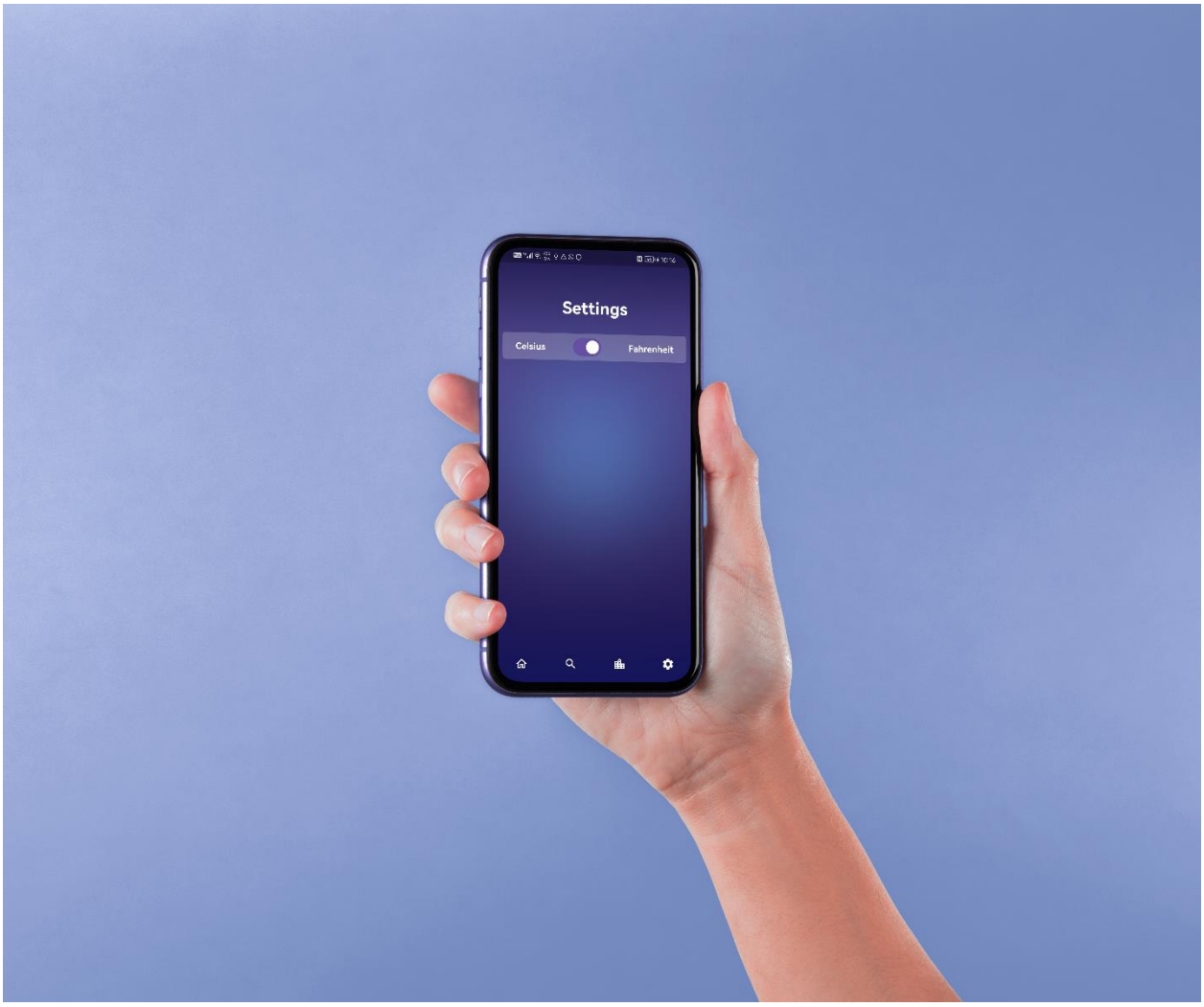
PermissionsCubit

The PermissionsCubit checks and requests the necessary location and notification permissions from the user. It also listens to location service status changes.

```
class PermissionsCubit extends Cubit<PermissionsState> {
  final PermissionsRepoImplementation permissionsRepoImplementation;
  PermissionsCubit(this.permissionsRepoImplementation) : super(PermissionsInitial());

  Future<void> checkPermissions() async {
    // Logic to check various permissions
  }

  Future<void> requestLocationPermission() async {
    // Logic to request location and notification permissions
  }
}
```

TemperatureUnitCubit

The TemperatureUnitCubit switches between Celsius and Fahrenheit temperature units.



Service Locator (getIt)

The **Service Locator** pattern is implemented using the getIt package to handle dependency injection across the application. It provides a centralized way of managing instances and ensures loose coupling between components.

Conclusion

This **Clean Architecture** approach ensures that the application remains modular and scalable, with a clear separation between data handling, business logic, and presentation. The **Cubit** state management provides a structured way of handling states across the UI, while the **Service Locator** simplifies dependency injection.