Clean architecture

La clean architecture permet de développer une application maintenable avec du code de qualité, c'est une architecture qui permet de développer chaque fonctionnalité sous 3 couches dont la couche **data, domain, présentation.**

- **La couche présentation :** elle est composée notamment des dossiers bloc pour le gestionnaire d'état dans le cas de notre application, pages et widgets pour la construction des pages et des interfaces de l'application .

- **La couche domain** : c'est la couche interne qui n'est pas sensible au changement de sources de données ou du portage de notre application. Elle contiendra la logique de la couche métier de base (uses cases) et les objets métier (les entity). Elle est indépendante de toute autre couche.

- **La couche de données** : se compose d'une implémentation du référentiel (le contrat provient de la couche de domaine) et de source de données. L'une sert généralement à obtenir des données distantes (API) et l'autre à mettre ces données en cache. Le référentiel est l'endroit où l'on décide d'envoyer des données fraîches ou mises en cache, quand les mettre en cache, etc.
  Les sources de données ne renvoient pas des entités mais plutôt des modèles . La raison derrière cela est que la transformation des données brutes (par exemple du JSON) en dart nécessite du code de conversion JSON .
  WeatherRemoteDatasource effectue des requêtes HTTP, WeatherLocalDatasource met simplement des données en cache via le package flutter_hive qui est une base de données NoSql qui permet de stocker les données localement.

Explication des composants

- **route** : pour la navigation j'ai utilisé le package Gorouter car avec ce package c'est plus facile de faire les deep links

- **Gestion des erreurs** : j'ai utilisé result qui est équivalent au package Dartz, il retourne un failure en cas d'échec et un success en cas de succès , les erreurs sont interceptées depuis le repositoryImpl afin d'éviter de propager les exceptions dans toutes les couches.

- **Network** : internet_connection_checker est un package qui permet de savoir si l'on est connecté ou pas

- **Styles** : Ce dossier permet de gérer la gestion des couleurs, style de textes et les thèmes

- **Injection de dépendance** : pour l'injection de dépendance , j'ai utilisé le pattern Singleton et le package Get_it, l'injection de dépendance avec le singleton permet de créer une seul instance pour tout le cycle de vie de l'application pour les classes

- **Internationalisation** :`flutter_localizations` ce package permet de créer des fichiers arb pour les différentes langues que nous aimerions utiliser dans l'application

- **Freezed** : est un générateur de code qui fournit beaucoup de méthode telle que copyWith, toString et d'autres méthodes pour comparer des classes, il est équivalent à equatable mais freezed possède des méthodes supplémentaires.

- **Gestion des données locales** : j'ai utilisé le package flutter_hive car ce package à la différence de sharedpreference permet de stocker des données avec des structures plus complexes tel que les objets.

- **Gestion des états** : j'ai utilisé flutter bloc

- **Test unitaire** : pour les test unitaire j'ai utilisé mocktail car avec ce package on ne génère pas de fichier