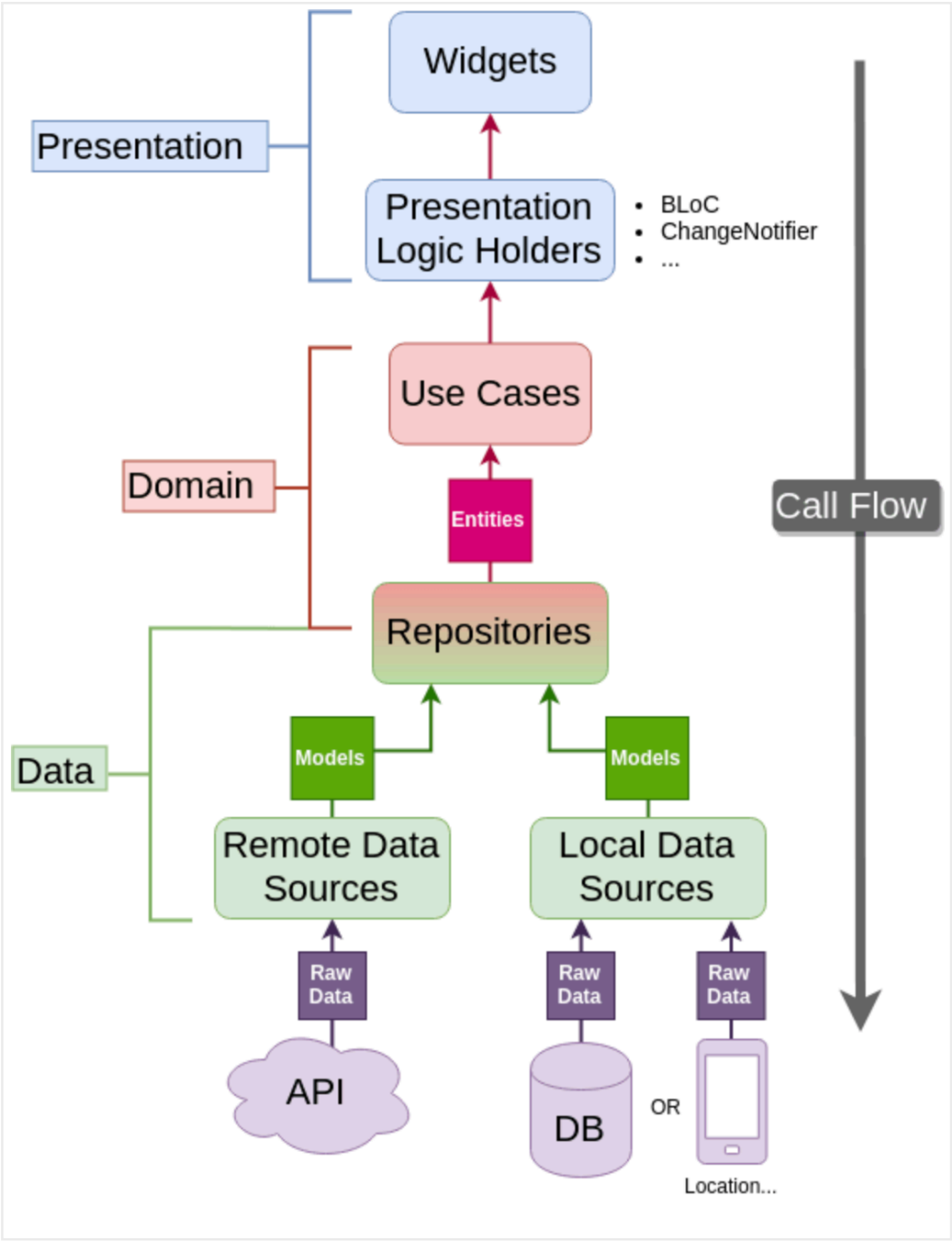
Clean Architecture

Garder votre code propre et testé sont les deux pratiques de développement les plus importantes. Dans Flutter, c'est encore plus vrai qu'avec d'autres frameworks. D'une part, c'est agréable de pirater une application rapide ensemble , d'autre part, des projets plus importants commencent à s'effondrer lorsque vous mélangez la logique métier partout. Même les modèles de gestion d'état comme BLoC ne sont pas suffisants en eux-mêmes pour permettre une base de code facilement extensible.

Clean Architecture & Flutter:



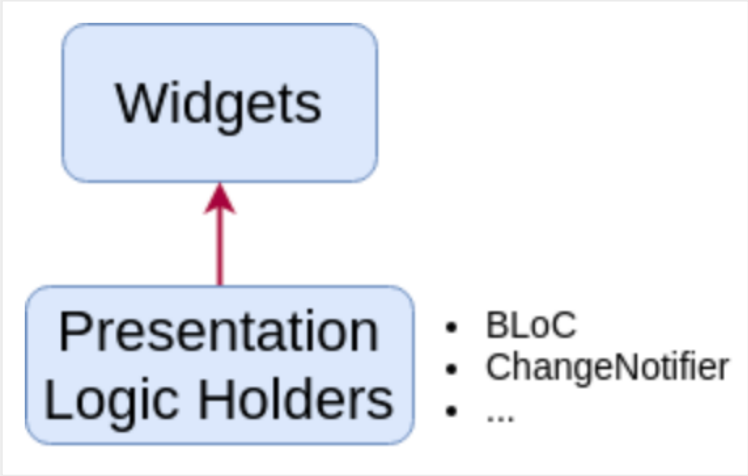
Explication & Organisation :

Chaque « features » de l'application sera divisée en 3 couches - **presentation** , **domain** et **data.**

Pour faciliter la création de dossiers de features, vous pouvez utiliser l’extension Flutter Feature Scaffolding de code VS

Presentation :

C'est ce à quoi vous êtes habitué de l'architecture Flutter «unclean». Vous avez évidemment besoin de widgets pour afficher quelque chose à l'écran. Ces widgets envoient ensuite des événements au Bloc et écoutent les états (ou un équivalent si vous n'utilisez pas Bloc pour la gestion des états).



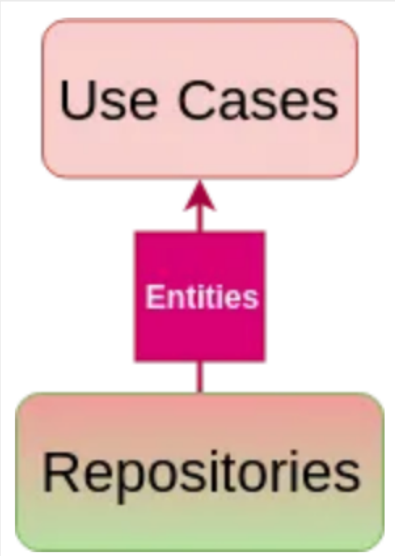
Notez que le "Presentation Logic Holder" (par exemple Bloc) ne fait pas grand-chose par lui-même. Il délègue tout son travail à des cas d'utilisation. Tout au plus, la couche de présentation gère la conversion et la validation de base des entrées.

Domain :

Le domaine est la couche interne qui ne devrait pas être sensible aux caprices de la modification des sources de données. Il ne contiendra que la logique métier de base (cas d'utilisation) et les objets métier (entités). Il doit être totalement indépendant de toutes les autres couches.

Les usecases sont des classes qui encapsulent toute la logique métier d'un cas d'utilisation particulier de l'application.

Comment la couche de domaine est- elle complètement indépendante lorsqu'elle obtient des données d'un Repository, qui provient de la couche de données ? Voyez-vous ce dégradé coloré fantaisiste pour le Repository ? Cela signifie qu'il appartient aux deux couches à la fois. Nous pouvons accomplir cela avec l'inversion de dépendance.



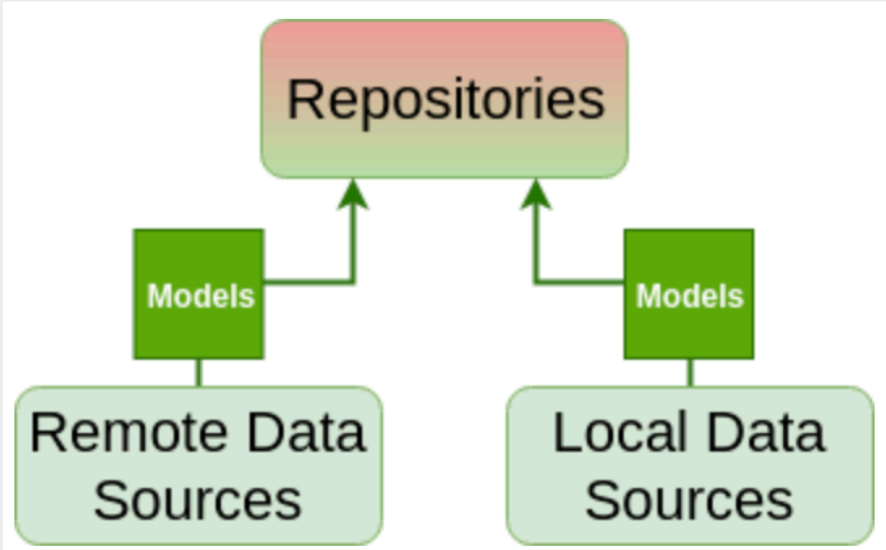
Repositories are on the edge between data and domain

C'est juste une façon fantaisiste de dire que nous créons une classe de référentiel abstraite définissant un contrat de ce que le référentiel doit faire. Nous dépendons alors du "contrat" du Repository défini dans le domaine, sachant que l'implémentation effective du Repository dans la couche de data remplira ce contrat.

Data :

La couche de données se compose d'une implémentation de référentiel (le contrat provient de la couche de domaine) et de sources de données. L'une sert généralement à obtenir des données distantes (API) et l'autre à mettre en cache ces données. Le référentiel est l'endroit où vous décidez si vous renvoyez des données fraîches ou mises en cache, quand les mettre en cache, etc.

Vous remarquerez peut-être que les datasources ne renvoient pas Entities mais plutôt Models. La raison derrière cela est que la transformation de données brutes (par exemple JSON) en objets Dart nécessite un code de conversion JSON. Nous ne voulons pas de ce code spécifique à JSON dans le domaine Entities.



The data layer works with Models, not Entities

Par conséquent, nous créons des classes Model qui étendent les entités et ajoutent des fonctionnalités spécifiques (toJson , fromJson) ou des champs supplémentaires, comme l'ID de la base de données, par exemple.

Source :

ResoCoder : https://resocoder.com/2019/08/27/flutter-tdd-clean-architecture-course-1-explanation-project-structure/#t-1670409302482