Ne Plus S'embrouiller avec les Relations Dans L'Architecture Clean Code Comme Un Pro

Par ESSOME MENANG FRED ARTHUR

Qui Suis Je?

Je m'appelle ESSOME MENANG FRED ARTHUR. Je suis étudiant en génie logiciel à l'Université Adventiste de Cosendai au Cameroun. Passionné de technologie, j'aime démystifier les secrets des technologies et des professionnels. J'ai moi-même rencontré des difficultés par le passé, mais j'ai réussi à les surmonter. C'est pourquoi je souhaite faciliter la vie d'autres développeurs. Je préfère éviter le jargon technique excessif et m'exprimer clairement et simplement. Ce n'est pas un ouvrage complet, mais plutôt un éclairage sur le sujet.

Ici je ne veux pas apprendre l'architecture clean code mais eclairer sur les notions et les rapports entre ces differents notion. tout comme on peut apprendre sans

Remerciements

Tout D'abord je ne peux rien faire sans remercier mon Dieu qui a donne son fils Jesus qui est mort pour vous et moi , qui me donne la vie et la santer et qui par jesus me donne l'assurance de la vie apres la mort au paradis quand il reviendra dans son regene. je remercie aussi mes parents qui m'ont toujours soustenu mes freres et mes Soeur et Mon Grand Frere Loic Fokam qui m'a fait decouvrir le code et cette architecture en particulier . je remercie mes amis et mon universiter.

Pré-requis

- Connaitre un language de programmation , ici nous utiliserons le language Dart de Google
- Etre a l'aise avec les notions de Poo(Programmation Oriente Object)
- avoir deja au moins pratiquer le codage meme sur un Project
- connaitre le Principes SOLID que vous pouvez retrouvez ici **https://medium.com/@abderrahmane.roumane.ext/tout-comprendre-des-principes-solid-en-10-minutes-votre-guide-rapide-pour-un-code-plus-efficace-bc625c3634f5**
- avoir deja connaissance de l'architeture clean code : **https://dev.to/marwamejri/flutter-clean-architecture-1-an-overview-project-structure-4bhf**

Introduction

C'est quoi La logique métier dans une application

Selon l'article du **https://www.superforge.io/articles/logique-metier-application**
ecrite par **Stephanie Menezes** et **Dimitri Nicolas**

La logique métier est un ensemble de règles et de processus qui **régissent les opérations** de l'application. Elle est utilisée pour décrire les étapes nécessaires pour accomplir une tâche particulière, ainsi que ses **règles et conditions** pour que cette tâche soit considérée comme terminée avec succès. Cette logique est appliquée à des nombreux domaines différents, de la sécurité à la gestion des utilisateurs en passant par les transactions financières.

Prenons un exemple pour mieux illustrer la logique métier dans la construction d'une application. Supposons que tu développes une application e-commerce ou une marketplace et que tu as besoin de mettre en place un processus de paiement. La logique métier pour ce processus pourrait inclure des étapes telles que :

- Afficher les produits à acheter.
- Ajouter les produits au panier.
- Vérifier les informations de paiement.
- Facturer le client.
- Valider le paiement.
- Envoyer une confirmation de paiement.

La logique métier pour ce processus pourrait aussi inclure des règles et de conditions, telles que :

- Si les informations de paiement sont incorrectes, renvoyer un message d'erreur à l'utilisateur.
- Si le paiement est refusé, avertir l'utilisateur et proposer un autre moyen de paiement.
- Si la transaction est annulée, annuler aussi la facturation.

Ces règles et processus sont des éléments clés dans la logique métier pour le processus de paiement.

Exemple pour un débutant

Imaginons que tu développes une application de gestion de bibliothèque. La logique métier dans cette application répondrait aux questions comme :

- **Comment emprunter un livre ?**
- **Combien de livres un utilisateur peut-il emprunter à la fois ?**
- **Comment savoir si un livre est disponible ou non ?**

La logique métier n'a rien à voir avec la présentation des données (ce que l'utilisateur voit à l'écran) ni avec l'endroit où les données sont stockées (base de données, serveur, etc.). Elle se concentre uniquement sur **les règles** et **les processus** qui répondent aux besoins de l'utilisateur.

Rôle dans une application

La logique métier va traiter des actions et des décisions comme :

- **Calculer** les frais de retard pour un livre rendu en retard.
- **Vérifier** qu'un utilisateur est éligible pour emprunter un livre (par exemple, qu'il n'a pas encore atteint sa limite de prêt).
- **Gérer** l'inventaire des livres, c'est-à-dire marquer un livre comme disponible ou emprunté.

En relation avec les autres parties de l'application

- **Interface utilisateur (UI)** : La logique métier ne décide pas de la manière dont les informations sont affichées à l'utilisateur, mais elle **envoie les bonnes données** à la couche de présentation pour qu'elles soient affichées correctement.
- **Données** : La logique métier ne gère pas directement où et comment les données sont stockées, mais elle sait **quelle action** doit être effectuée (comme enregistrer un prêt de livre) et demande à la couche de gestion des données (via un **Repository**, par exemple) de faire le travail.

Un exemple en Flutter/Dart

Supposons que tu veux coder une application qui permet de calculer des frais de retard pour la bibliothèque :

Logique métier (simplifiée) :

```
class LibraryService {
  static const int finePerDay = 1; // 1 unité de monnaie par jour de retard

  int calculateFine(DateTime dueDate, DateTime returnDate) {
    if (returnDate.isBefore(dueDate)) {
      return 0; // Pas de frais si le livre est rendu avant la date limite
    }
    final int daysLate = returnDate.difference(dueDate).inDays;
    return daysLate * finePerDay;
  }
}
```

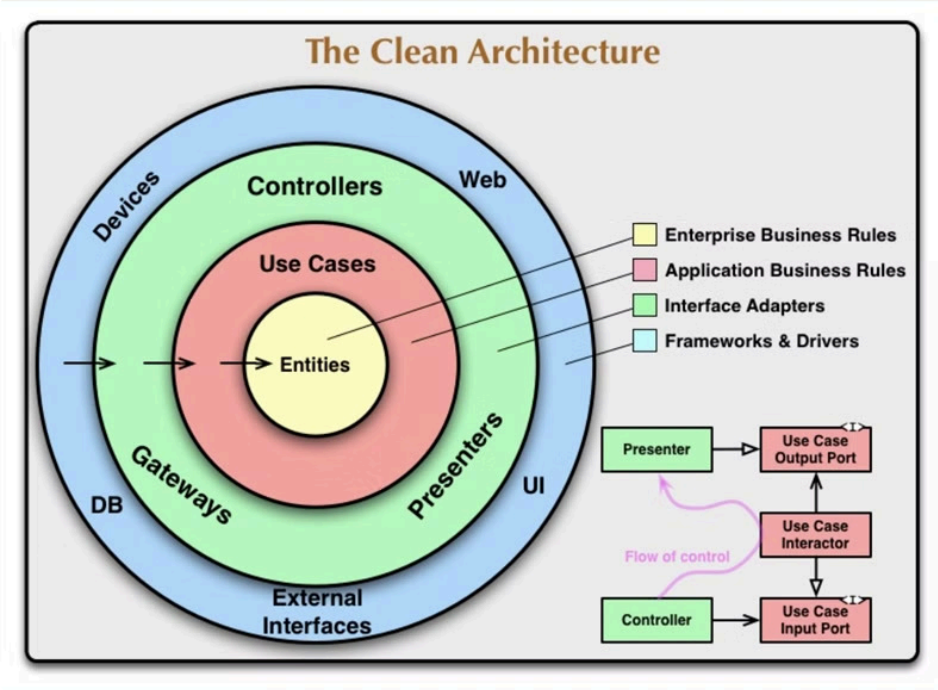
- **calculateFine** : C'est une règle métier qui calcule les frais en fonction du nombre de jours de retard.

La Clean Architecture C'est quoi?

La **Clean Architecture** est une architecture logicielle proposée par Robert C. Martin, également connu sous le nom d'Oncle Bob. Son objectif principal est de rendre les applications **modulaires**, **faciles à maintenir**, **flexibles**, et **testables**. Elle repose sur l'idée de **séparation des préoccupations**, c'est-à-dire que chaque partie du code doit avoir une responsabilité bien définie et ne pas se mélanger avec les autres.

Principes de base de la Clean Architecture

1. **Indépendance des frameworks** : L'application ne doit pas dépendre de frameworks externes. Ces derniers sont des outils, pas des éléments essentiels de la logique métier.
2. **Indépendance de l'interface utilisateur** : L'interface utilisateur peut changer sans avoir d'impact sur la logique métier. Ainsi, il est facile de changer ou d'adapter l'UI (par exemple passer d'une application mobile à une application web).
3. **Testabilité** : Les différentes couches de l'application peuvent être testées de manière isolée, facilitant la création de tests unitaires.
4. **Indépendance des bases de données** : La logique métier ne dépend pas de la façon dont les données sont stockées. Cela permet de changer facilement de base de données ou de méthode de stockage sans toucher au cœur de l'application.
5. **Indépendance des détails externes** : Les frameworks, bibliothèques externes, et systèmes externes ne doivent pas influencer la logique métier.



Elle est souvent représentée de la sorte, mais ne nous mentons pas, on ne comprend rien. C'est pour les pros là-bas. Nous, on aime quand c'est clair et simple. Mais là, pour un junior, c'est un peu difficile à comprendre. En gros, ça dit que: **Rien dans un cercle intérieur ne peut rien savoir de quelque chose dans un cercle extérieur.**

Autrement dit, aucune variable, classe ou fonction déclarée dans un cercle extérieur ne peut être mentionnée dans un cercle intérieur.

Organisation des Couches

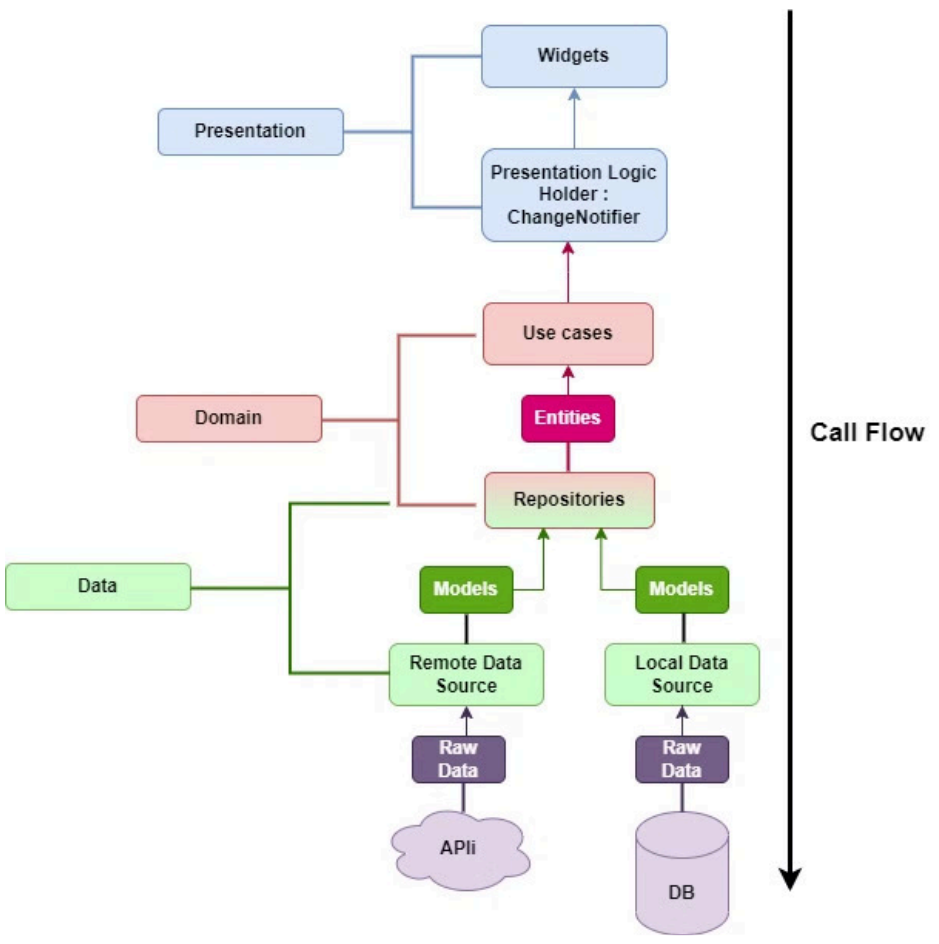
La Clean Architecture est souvent représentée sous forme de cercles concentriques, où chaque cercle représente une couche de l'application, avec les couches extérieures dépendant des couches intérieures.

Voici les principales couches :

1. **Entities** : Ce sont les objets métier avec leurs règles de validation et de logique propre. Par exemple, un objet **Utilisateur** avec des attributs comme un nom et une méthode pour valider son email.
2. **Use Cases** : Les Use Cases définissent les actions possibles dans le système. Ils sont spécifiques à l'application et dictent les interactions de la logique métier. Par exemple, un Use Case pourrait être **Créer un utilisateur** ou **Effectuer une transaction**.

1. **Interface Adapters (Repositories, Présentation)** : Cette couche contient les adaptateurs qui permettent à la logique métier d'interagir avec les systèmes externes (comme la base de données ou l'interface utilisateur). Par exemple, les **Repositories** sont des classes qui servent d'intermédiaires entre la logique métier et la source de données.
2. **Frameworks et Drivers (Couche Externe)** : Il s'agit de la couche la plus externe, qui contient les frameworks, bibliothèques, outils, bases de données, APIs, ou l'interface utilisateur (UI) elle-même. Cette couche peut être facilement modifiée sans impacter le cœur de l'application.

Le Flux de Dépendances



Ce schéma représente l'architecture **Clean Code** et montre comment les différentes couches d'une application communiquent entre elles. Voici une explication détaillée de chaque partie du schéma et de la façon dont elles interagissent :

1. Couche Presentation

- **Widgets** : Ils représentent l'interface utilisateur (UI), où l'utilisateur interagit avec l'application (boutons, champs de texte, etc.).
- **Presentation Logic Holder (ChangeNotifier)** : Cette partie gère la **logique de présentation**, comme l'état de l'interface utilisateur (chargement, affichage d'erreurs, etc.). Le **ChangeNotifier** est un moyen d'informer les widgets de changements dans les données ou l'état.
-

2. Couche Domaine (Domain)

- **Use Cases** : Ce sont des éléments de la logique métier qui dictent les actions que l'utilisateur peut effectuer. Ils contiennent les règles qui contrôlent comment les données sont manipulées. Par exemple, un Use Case pourrait définir comment créer un utilisateur ou valider une commande.
- **Entities** : Les **Entities** représentent les concepts fondamentaux du domaine métier (comme un **Utilisateur**, un **Produit**, ou une **Commande**). Ils définissent les **objets** que les Use Cases manipulent et incluent des règles spécifiques.

3. Couche Repository

- Les **Repositories** servent d'intermédiaires entre la logique métier (Use Cases et Entities) et les sources de données (API ou base de données). Ils permettent aux Use Cases d'accéder aux données sans se soucier de l'endroit d'où elles proviennent.
- Les Repositories s'occupent de **récupérer les données** depuis des sources distantes (**Remote Data Source**) ou locales (**Local Data Source**).

4. Couche Data

- **Remote Data Source** : Il s'agit d'une source de données externe, telle qu'une **API** ou un service web. Les données récupérées sont souvent en format brut, comme du JSON, et doivent être transformées en **Models** pour être utilisées dans l'application.
- **Local Data Source** : Il s'agit d'une source de données locale, comme une base de données locale (par exemple, SQLite). Les données y sont stockées pour être accessibles rapidement et hors ligne.

5. Models

- Les **Models** se trouvent dans la couche Data et servent à **représenter les données** provenant des sources de données distantes (API) ou locales (base de données).
- Ces Models servent d'interface entre les **données brutes** et le reste de l'application. Ils sont convertis en **Entities** pour que la couche Domaine puisse les utiliser.

6. Raw Data

- Le **Raw Data** correspond aux **données brutes** telles qu'elles sont reçues depuis une API ou une base de données, avant d'être transformées en **Models**. Par exemple, les données reçues d'une API REST en format JSON sont du Raw Data avant d'être traitées.

7. Flow d'Appel

- Le **Call Flow** (flux d'appel) décrit la manière dont les données et les actions traversent les couches de l'application :
  - **Widgets (UI)** : L'utilisateur interagit avec l'interface, par exemple en appuyant sur un bouton.
  - **Presentation Logic Holder** : Ce composant traite l'action de l'utilisateur, comme déclencher une requête pour obtenir des données.
  - **Use Cases** : Le Use Case est appelé pour exécuter une action spécifique (par exemple, récupérer un utilisateur à partir d'une API).
  - **Entities** : Les Use Cases manipulent les **Entities** pour appliquer les règles métiers.
  - **Repositories** : Le Use Case interagit avec le Repository pour accéder aux données.
  - **Data Sources (Remote/Local)** : Le Repository va chercher les données depuis une source distante (API) ou locale (base de données).
  - **Raw Data** : Les données brutes sont récupérées, transformées en **Models**, puis éventuellement en **Entities** pour être utilisées dans le domaine métier.
  - **Retour** : Les données sont renvoyées à travers les couches (Repository → Use Case → UI) pour être affichées dans l'interface.

En résumé :

- **Widgets** : Gèrent l'interface utilisateur et les interactions avec l'utilisateur.
- **Presentation Logic Holder (ChangeNotifier)** : Gère l'état de l'interface et notifie les widgets des changements.
- **Use Cases** : Gèrent la logique métier et définissent les actions possibles.
- **Entities** : Représentent les objets métiers avec leurs règles.
- **Repositories** : Intermédiaires entre la couche métier et les sources de données.
- **Remote et Local Data Source** : Fournissent les données depuis des APIs ou des bases de données locales.
- **Models** : Représentent les données structurées après traitement du Raw Data.

Exemple de Flux d'Exécution

Prenons l'exemple d'une application bancaire simple. Si un utilisateur veut vérifier son solde :

1. **L'utilisateur clique sur un bouton** dans l'interface utilisateur (UI) pour voir son solde.
2. **L'interface envoie cette action** à un **Use Case** qui sait comment obtenir le solde de l'utilisateur.
3. Le **Use Case** contacte un **Repository** pour récupérer les données depuis une base de données ou un service externe.
4. Le **Repository** retourne les données au **Use Case**, qui applique les règles métier.
5. Enfin, le **Use Case** retourne l'information à l'interface utilisateur pour l'afficher à l'utilisateur.

Pourquoi utiliser la Clean Architecture ?

1. **Facilité de maintenance** : Avec une séparation claire des responsabilités, il est plus facile de maintenir et de faire évoluer l'application sans casser d'autres parties du code.
2. **Réutilisation** : La logique métier est indépendante des détails techniques (base de données, frameworks UI), ce qui permet de réutiliser le même code sur plusieurs plateformes.
3. **Facilité de test** : Chaque partie de l'application peut être testée de manière isolée, ce qui rend le processus de test plus simple et plus fiable.
4. **Flexibilité** : Il devient plus facile de modifier une partie du système (comme passer d'une base de données locale à une base de données distante) sans affecter le reste de l'application.

ILLUSTRATION DE L'ARCHITECTURE CLEAN CODE:

Ici nous prenons exemple d'une application flutter de Gestion de D'utilisateur.

```
lib/
├── features/
│   ├── user_management/
│   │   ├── data/
│   │   │   ├── models/
│   │   │   │   ├── user_model.dart
│   │   │   ├── repositories/
│   │   │   │   ├── user_repository_impl.dart
│   │   │   ├── data_sources/
│   │   │   │   ├── local/
│   │   │   │   │   ├── user_local_data_source.dart
│   │   │   │   ├── remote/
│   │   │   │   │   ├── user_remote_data_source.dart
│   │   ├── domain/
│   │   │   ├── entities/
│   │   │   │   ├── user.dart
│   │   │   ├── repositories/
│   │   │   │   ├── user_repository.dart
│   │   │   ├── use_cases/
│   │   │   │   ├── get_user.dart
│   │   │   │   ├── create_user.dart
│   │   │   │   ├── update_user.dart
│   │   │   │   ├── delete_user.dart
│   │   ├── presentation/
│   │   │   ├── pages/
│   │   │   │   ├── user_list_page.dart
│   │   │   │   ├── user_detail_page.dart
│   │   │   ├── state_management/
│   │   │   │   ├── user_state.dart
│   │   │   │   ├── user_state_notifier.dart
│   │   │   ├── widgets/
│   │   │   │   ├── user_card.dart
│   │   │   │   ├── user_form.dart
│   │   ├── utils/
│   │   │   ├── validators.dart
├── core/
│   ├── error/
│   │   ├── failure.dart
│   ├── network/
│   │   ├── network_info.dart
│   ├── utils/
│   │   ├── constants.dart
│   ├── use_cases/
│   │   ├── use_case.dart
│   ├── database/
│   │   ├── database_helper.dart
├── main.dart
```

Explication des dossiers et fichiers

1. features/user_management/

C'est ici que chaque feature de l'application est organisée. Chaque feature a ses propres sous-dossiers pour gérer les différents aspects de l'architecture (data, domain, presentation).

a. **data/**

Cette partie contient tout ce qui concerne la **couche Data**, c'est-à-dire les **Models**, les **Repositories Implémentés**, et les **Data Sources** (locales et distantes).

- **models/** : Contient les modèles qui représentent les données telles qu'elles sont reçues depuis une API ou une base de données. Par exemple, `user_model.dart` pour le modèle utilisateur.
- **repositories/** : Contient l'implémentation des repositories. Par exemple, `user_repository_impl.dart` implémente l'interface repository pour gérer les sources de données.
- **data_sources/** : Séparée en deux sous-dossiers :
  - **local/** : Gère l'accès aux sources locales comme les bases de données internes (`user_local_data_source.dart`).
  - **remote/** : Gère l'accès aux APIs externes (`user_remote_data_source.dart`).

b. **domain/**

Cette partie concerne la logique métier avec les **Entities**, les **Use Cases** et les interfaces des **Repositories**

- **entities/** : Contient les entités qui représentent les objets métiers utilisés dans les Use Cases, comme `user.dart`.
- **repositories/** : Définit les interfaces des repositories qui seront utilisées dans les Use Cases (`user_repository.dart`).
- **use_cases/** : Contient les cas d'utilisation, comme la création, la récupération, la mise à jour ou la suppression d'un utilisateur (`create_user.dart`, `get_user.dart`, etc.).

c. **presentation/**

La couche **Presentation** gère tout ce qui est visible pour l'utilisateur, comme les widgets, les pages et les view models (ou gestionnaires d'état).

- **pages/** : Contient les différentes pages de l'interface utilisateur, comme la liste des utilisateurs (`user_list_page.dart`) ou les détails d'un utilisateur (`user_detail_page.dart`).
- **State Management/** : Gère l'état et la logique de la présentation. Par exemple, `user_state.dart` communique avec les Use Cases pour fournir les données aux widgets.
- **widgets/** : Contient des widgets réutilisables, comme un formulaire pour ajouter un utilisateur (`user_form.dart`) ou une carte pour afficher un utilisateur (`user_card.dart`).

d. **utils/**

Contient des fonctions utilitaires propres à cette feature, comme des validateurs de formulaires (`validators.dart`).

2. core/

Le dossier **core** contient des éléments communs à toute l'application, réutilisables dans plusieurs features.

a. **error/**

Gestion des erreurs globales, avec des classes comme `failure.dart` pour définir les erreurs qui pourraient survenir dans toute l'application.

b. **network/**

Gère la détection de l'état du réseau avec des classes comme `network_info.dart`.

c. **utils/**

Contient des constantes et des utilitaires communs à toute l'application (`constants.dart`).

d. **use_cases/**

Contient la définition générique d'un **Use Case** qui peut être étendue pour chaque feature (`use_case.dart`).

e. **database/**

Contient des helpers pour l'accès à la base de données, comme `database_helper.dart`.

3. main.dart

Le fichier principal de l'application Flutter où la configuration initiale (comme les routes) est définie

Second Partie. Ici Nous Attquerons les concepts les plus complexe de la La clean Archictecture

Les "Use Cases"

Qu'est-ce qu'un "use case" ?

Un **use case** représente un scénario précis dans lequel l'utilisateur ou un acteur externe interagit avec ton application pour accomplir un objectif. En termes simples, c'est ce que ton application **doit faire** pour répondre aux besoins de l'utilisateur. Chaque use case correspond à une fonctionnalité métier spécifique.

Le rôle des use cases dans l'architecture Clean

Dans l'architecture **Clean**, la logique métier de ton application est organisée autour des **use cases**. Ces use cases représentent l'ensemble des règles et actions métier spécifiques à ce que l'application doit accomplir. Ils forment le cœur de l'application (appelée "couche application" ou "business rules"). Ils sont **indépendants de l'interface utilisateur**, de la base de données, ou de toute autre infrastructure.

Comment se présentent les use cases ?

Dans la Clean Architecture, chaque **use case** est généralement représenté sous la forme d'une classe ou d'une fonction qui encapsule une tâche spécifique. Ces classes ou fonctions sont responsables de la coordination des différentes actions nécessaires pour accomplir un objectif utilisateur, sans se soucier des détails d'implémentation (comme l'UI ou la persistance des données).

Exemple simplifié de présentation

Imaginons que tu développes une application de gestion de stock. Un **use case** pourrait être : "Ajouter un produit au stock".

1. **Entrées :**
   - Les informations sur le produit (nom, quantité à ajouter).
2. **Logique métier :**
   - Vérifier si le produit existe déjà.
   - Ajouter la quantité spécifiée au stock.
   - Enregistrer le changement dans la base de données.
3. **Sorties :**
   - Le stock mis à jour, ou un message d'erreur si quelque chose s'est mal passé.

Dans une classe Dart, ça pourrait ressembler à ceci :

```
class AddProductToStockUseCase {
  final StockRepository stockRepository;

  AddProductToStockUseCase(this.stockRepository);

  void execute(Product product, int quantity) {
    if (stockRepository.productExists(product)) {
      stockRepository.addQuantity(product, quantity);
    } else {
      print("Erreur : Produit non trouvé.");
    }
  }
}
```

Ici, la classe `AddProductToStockUseCase` encapsule le processus de vérification et d'ajout de la quantité à un produit, indépendamment de la base de données (qui est gérée par un `StockRepository`).

Les Use Cases reçoivent des **données d'entrée** (fournies par l'UI ou une autre source), appliquent les règles métiers, et renvoient un **résultat**.

Exemple 1 : Gestion des utilisateurs

Imaginons un système où l'utilisateur peut s'inscrire. Le Use Case pourrait être intitulé `RegisterUser`.

```
class RegisterUser {
  final UserRepository userRepository;

  RegisterUser(this.userRepository);

  Future<void> execute(String username, String password) async {
    if (username.isEmpty || password.isEmpty) {
      throw Exception("Le nom d'utilisateur et le mot de passe ne peuvent pas être vides.");
    }

    // Ici, on vérifie les règles métiers, par exemple la longueur minimale du mot de passe
    if (password.length < 8) {
      throw Exception("Le mot de passe doit contenir au moins 8 caractères.");
    }

    // On enregistre l'utilisateur via le repository
    await userRepository.saveUser(username, password);
  }
}
```

Dans cet exemple :

- Le **Use Case** `RegisterUser` gère la logique métier de l'inscription d'un utilisateur.
- Il reçoit un **repository** qui permet d'accéder aux données (userRepository), mais il ne sait pas comment ces données sont stockées (cela peut être une base de données locale, une API distante, etc.).
- Le Use Case applique des règles comme vérifier si le mot de passe est assez long avant d'enregistrer l'utilisateur.

Exemple 2 : Calcul de réduction sur une commande

Prenons un autre exemple : appliquer une réduction sur une commande dans un système e-commerce. Le Use Case pourrait s'appeler `ApplyDiscount`.

```
class ApplyDiscount {
  final OrderRepository orderRepository;

  ApplyDiscount(this.orderRepository);

  Future<double> execute(String orderId, double discountPercentage) async {
    if (discountPercentage < 0 || discountPercentage > 100) {
      throw Exception("Le pourcentage de réduction doit être compris entre 0 et 100.");
    }

    // Récupération de la commande à partir du repository
    final order = await orderRepository.getOrder(orderId);

    // Application de la réduction
    final discountedAmount = order.totalAmount * (1 - discountPercentage / 100);

    // Mise à jour de la commande avec le montant réduit
    await orderRepository.updateOrderAmount(orderId, discountedAmount);

    return discountedAmount; // Retourner le montant après réduction
  }
}
```

Dans cet exemple :

- Le **Use Case** `ApplyDiscount` est responsable d'appliquer une réduction sur une commande spécifique.
- Il effectue la validation des données (par exemple, vérifier que la réduction est entre 0 et 100 %).
- Il interagit avec un **repository** pour récupérer et mettre à jour la commande.

À quoi renvoient les Use Cases ?

Les **Use Cases** renvoient généralement un **résultat** ou un **effet**, comme une action terminée (par exemple, utilisateur enregistré) ou une donnée mise à jour (par exemple, montant de la commande après réduction). Ces résultats sont ensuite traités par une autre couche, comme le **contrôleur** ou l'**interface utilisateur**, qui va les afficher à l'utilisateur ou les envoyer à un autre service

À quoi servent les use cases ?

Les **use cases** servent à **organiser** et **structurer** la logique métier. Voici leurs principales fonctions :

1. **Séparer la logique métier de l'interface utilisateur** : Ils permettent de garder la logique métier indépendante des détails techniques (comme l'interface utilisateur, la base de données, etc.).
2. **Rendre le code plus maintenable** : En séparant chaque cas d'utilisation dans sa propre classe ou fonction, le code devient plus facile à tester, à comprendre et à modifier.
3. **Faciliter les tests unitaires** : Comme les use cases sont isolés de l'infrastructure, ils sont plus simples à tester individuellement. Par exemple, tu peux tester le `AddProductToStockUseCase` sans avoir besoin de réellement interagir avec une base de données ou une interface utilisateur.
4. **Faciliter les évolutions** : Si un jour tu dois changer ta base de données ou ton interface utilisateur, la logique métier reste intacte, car elle est séparée et gérée par les use cases.

En résumé

- Les **Use Cases** définissent **ce que** l'application doit faire en termes de règles métiers.
- Ils sont **isolés** des détails techniques comme la base de données ou l'interface utilisateur.
- Cela rend l'application plus **facile à maintenir**, **testable**, et **modulaire**.

Ces exemples montrent comment les **Use Cases** aident à organiser la logique métier et à séparer les responsabilités dans une architecture propre.]

Comment ça se connecte aux autres couches ?

Dans la Clean Architecture, les **use cases** sont au centre, entourés par différentes couches :

- **Interface utilisateur (UI)** : C'est l'extérieur, où les interactions utilisateur se produisent (par exemple, un écran Flutter).
- **Accès aux données (Infrastructure)** : Ce sont les mécanismes pour stocker et récupérer les données (base de données, API).
- **Entités** : Les objets métiers qui contiennent les règles les plus générales (par exemple, un objet `Product` qui représente un produit).

Les use cases **reçoivent des entrées de l'interface utilisateur**, traitent la logique métier, et ensuite **appellent des services ou des repositories** pour persister ou récupérer des données.

Entities

Dans l'architecture **Clean Code**, les **Entities** (entités) sont des objets qui représentent les concepts fondamentaux et les règles d'une application métier. Elles sont au cœur de la logique métier, car elles encapsulent les données et les comportements essentiels, indépendamment de toute infrastructure ou technologie spécifique (comme la base de données ou l'interface utilisateur).

À quoi servent les Entities ?

Les **Entities** définissent les objets qui composent le domaine métier de ton application et s'assurent que les règles et comportements de ces objets sont respectés. Ce sont les éléments les plus stables dans ton code, car ils représentent le cœur du système. Une **Entity** pourrait être un utilisateur, une commande, un produit, etc. Elles ne dépendent ni de la manière dont les données sont stockées ni de la manière dont elles sont présentées à l'utilisateur.

Comment se présentent les Entities ?

Une **Entity** est souvent une classe qui contient des attributs (les propriétés de l'entité) et des méthodes (les comportements ou actions que l'entité peut réaliser). Elle peut également définir des règles métiers spécifiques à l'entité.

Exemple en Dart :

Imaginons que nous avons une application de gestion de commandes, et qu'une **Entity** représente une **Commande** :

```
class Order {
  final String id;
  double totalAmount;
  bool isPaid;

  Order(this.id, this.totalAmount, this.isPaid);

  void payOrder() {
    if (isPaid) {
      throw Exception("La commande est déjà payée.");
    }
    isPaid = true;
  }

  void applyDiscount(double percentage) {
    if (percentage < 0 || percentage > 100) {
      throw Exception("Le pourcentage de réduction doit être entre 0 et 100.");
    }
    totalAmount = totalAmount * (1 - percentage / 100);
  }
}
```

Dans cet exemple :

- L'**Entity** `Order` représente une commande dans le système.
- Elle possède des propriétés (`id`, `totalAmount`, `isPaid`) et des méthodes pour manipuler ces propriétés (`payOrder`, `applyDiscount`).
- Les règles métiers, comme s'assurer qu'une commande ne peut pas être payée deux fois ou que la réduction est entre 0 et 100%, sont encapsulées directement dans l'entité.

Rapport entre Use Cases et Entities

Les **Use Cases** définissent **ce que** l'application doit faire (les actions métier), tandis que les **Entities** sont les objets sur lesquels ces actions s'appliquent. Les **Use Cases** manipulent généralement les **Entities** pour accomplir une action métier.

Les **Entities** sont comme les acteurs principaux du domaine métier, et les **Use Cases** sont les scénarios dans lesquels ces acteurs jouent un rôle.

Comment connecter les Use Cases et les Entities ?

Les **Use Cases** utilisent les **Entities** pour appliquer les règles métiers. Ils reçoivent des données, effectuent des actions via les **Entities**, et renvoient un résultat.

Exemple de connexion entre Use Case et Entity

Prenons le **Use Case** d'application d'une réduction sur une commande, et la **Entity** `Order` que nous avons définie plus haut. Voici comment nous pouvons les connecter :

```
class ApplyDiscountUseCase {
  final OrderRepository orderRepository;

  ApplyDiscountUseCase(this.orderRepository);

  Future<void> execute(String orderId, double discountPercentage) async {
    // Récupérer la commande via le repository
    final order = await orderRepository.getOrder(orderId);

    // Appliquer la réduction via l'entité Order
    order.applyDiscount(discountPercentage);

    // Sauvegarder la commande mise à jour
    await orderRepository.updateOrder(order);
  }
}
```

Dans cet exemple :

1. Le **Use Case** `ApplyDiscountUseCase` récupère une instance de l'**Entity** `Order` via le `OrderRepository`.
2. Il applique une action métier (`applyDiscount`) en utilisant une méthode de l'**Entity**.
3. Il renvoie ensuite les résultats (ici, en sauvegardant la commande mise à jour via le repository).

Récapitulatif

- **Entities** : Elles représentent les objets du domaine métier avec leurs attributs et leurs comportements. Elles encapsulent les règles et les logiques propres à ces objets.
- **Use Cases** : Ils représentent des actions ou des processus métiers spécifiques qui manipulent les **Entities** pour accomplir des tâches.
- **Connexion** : Les **Use Cases** récupèrent et modifient les **Entities** pour accomplir leurs tâches métiers. C'est ainsi que la logique métier est réalisée de manière modulaire et testable.

Cela permet une grande **séparation des préoccupations** et facilite l'évolution de l'application sans casser d'autres parties du code.

Repository

Dans l'architecture **Clean Code**, un **Repository** (ou dépôt en français) est un concept qui sert de pont entre la **logique métier** (les règles et les actions spécifiques à ton application) et la **source des données** (comme une base de données, une API, un fichier, etc.). Il permet à la logique métier de ne pas se soucier de **comment** les données sont stockées ou récupérées, mais juste de **quoi** elle a besoin.

À quoi sert un repository ?

Un **repository** est là pour s'occuper des **détails de stockage** des données. Il aide à **organiser** et **gérer** les informations sans que le reste de ton application ne se préoccupe de **comment** ou **où** ces informations sont stockées. **IL** est comme un intermédiaire qui se charge de récupérer et de stocker les données pour la logique métier. Au lieu que le **Use Case** (la partie qui contient les actions et les règles de ton application) accède directement à la base de données ou à l'API, il passe par le **Repository**, qui s’occupe des détails.

Cela permet de **séparer les responsabilités** :

- Le **Use Case** peut se concentrer uniquement sur les règles métier.
- Le **Repository** gère tout ce qui concerne l'accès aux données.

Si tu as une application qui gère des utilisateurs, ton repository va se charger de **stocker** ces utilisateurs quelque part (comme dans une base de données, sur un serveur distant ou même juste dans la mémoire de ton ordinateur).
Quand tu voudras **retrouver** ces utilisateurs, c'est aussi le repository qui va s'en occuper.

Pourquoi utiliser un repository ?

Le but du repository est de **séparer** la logique métier de la manière dont les données sont manipulées. Cela rend ton code **plus propre** et **facile à modifier**. Par exemple :

- Si un jour tu veux changer **l'endroit** où tu stockes les utilisateurs (passer d'une base de données locale à un serveur distant), tu pourras le faire en modifiant seulement le repository, sans toucher à la logique de ton application.

Comment se présente un repository ?

Un repository est souvent une **classe** qui a des **méthodes** pour ajouter, récupérer, mettre à jour ou supprimer des informations.. Par exemple, pour un utilisateur, un **UserRepository** pourrait contenir des méthodes comme `saveUser` (pour enregistrer un utilisateur) et `getUserById` (pour récupérer un utilisateur via son ID).

Exemple simple d'un repository en Dart :

```
// Interface du repository pour les utilisateurs
abstract class UserRepository {
  Future<void> saveUser(User user);
  Future<User?> getUserById(String id);
}
```

Dans cet exemple :

- `saveUser` : Cette méthode permet de **stocker** un utilisateur.
- `getUserById` : Cette méthode permet de **récupérer** un utilisateur à partir de son identifiant.

Ici, le **UserRepository** définit deux actions possibles : **enregistrer un utilisateur** et **récupérer un utilisateur** par son identifiant. Cependant, il ne dit pas **comment** ces actions seront faites. Il s'agit simplement de la promesse que ces actions peuvent être réalisées.

Différence entre Repository de la couche Domaine et Data

- **Repository de la couche Domaine** : C’est une abstraction (interface) qui ne sait pas comment les données seront obtenues ou enregistrées. Elle décrit seulement les actions possibles (par exemple, "enregistrer un utilisateur", "récupérer un utilisateur"). C'est comme dire : "Je veux faire ces actions, mais je ne me soucie pas de comment."
- **Repository de la couche Data** : C’est là où on décide **comment** les actions vont être faites. Par exemple, si tu utilises une base de données, une API, ou un fichier, c'est le **Repository de la couche Data** qui contient la logique pour lire et écrire dans ces sources de données.

Exemple simple d'un repository dans la couche Domaine ：

```
// Interface du repository pour les utilisateurs
abstract class UserRepository {
  Future<void> saveUser(User user);
  Future<User?> getUserById(String id);
}
```

Exemple du Repository dans la couche Data (implémentation concrète) :

```
// Repository dans la couche data (implémentation en mémoire)
class InMemoryUserRepository implements UserRepository {
  final Map<String, User> users = {};

  @override
  Future<void> saveUser(User user) async {
    users[user.id] = user;
    print("Utilisateur enregistré en mémoire : ${user.username}");
  }

  @override
  Future<User?> getUserById(String id) async {
    return users[id];
  }
}
```

Fonctionnement de chaque couche

1. **Côté Domaine (Domain Layer)** :
    - On utilise le **UserRepository** pour dire quelles actions sont possibles (enregistrer, récupérer un utilisateur).
    - Le **Use Case** va utiliser ce repository sans savoir comment il fonctionne réellement. Il sait juste que l'utilisateur sera enregistré et récupéré.
2. **Côté Data (Data Layer)** :
    - C'est ici qu'on décide **comment** les données sont gérées. Par exemple, dans le code ci-dessus, les utilisateurs sont stockés dans un simple tableau en mémoire (Map), mais cela pourrait être une base de données, une API distante, ou autre.

En resumer le Repository de la couche domaine nous promet que les donner seront traiter et la le repository de la couche data accompli cette promesse(traite cette promesse)

Quel rapport avec le reste du code ?

Le **repository** est utilisé par les **Use Cases** et les **Entities**. Par exemple :

- Les **Use Cases** ont besoin du repository pour **enregistrer** ou **récupérer** des données.
- Les **Entities** ne s'occupent pas de savoir **comment** les données sont stockées, elles laissent cela au repository.

Comment ça fonctionne ensemble ?

1. **Use Case** : Il dit "Je veux enregistrer cet utilisateur."
2. **Repository** : Il se charge de trouver où et comment enregistrer cet utilisateur.
3. **Entity** : L'utilisateur (qui est une entity) est passé au repository pour être sauvegardé.

Récapitulatif :

- Un **repository** est un endroit où tu stockes et récupères des informations.
- Il te permet de ne pas te soucier de **comment** ou **où** les informations sont stockées.
- Il travaille avec les **Use Cases** pour accomplir les actions métier, et il manipule les **Entities** pour interagir avec les données.

En résumé, le repository est comme un **assistant** : il gère le stockage des données pour que tu puisses te concentrer sur ce que ton application doit faire.

Exemple D'application(Use Cases, Entities, Repositories)

Voici un exemple complet avec un **Use Case**, une **Entity**, un **Repository** de la couche **Domaine**, et un **Repository** de la couche **Data**. À la fin, je vais créer un code où ces quatre parties communiquent ensemble.

1. Code de l’Entity : Représentation de l'utilisateur

L'**Entity** est la définition de ce qu'est un utilisateur dans le domaine de ton application.

```
// Entity : User
class User {
  final String id;
  final String username;
  String password;

  User(this.id, this.username, this.password);

  // Méthode pour mettre à jour le mot de passe
  void updatePassword(String newPassword) {
    if (newPassword.length < 8) {
      throw Exception("Le mot de passe doit comporter au moins 8 caractères.");
    }
    password = newPassword;
  }
}
```

2. Code du Use Case : Enregistrer un utilisateur

Le **Use Case** contient la logique métier. Ici, on va créer un Use Case pour enregistrer un nouvel utilisateur.

```
// Use Case : RegisterUser
class RegisterUser {
  final UserRepository userRepository;

  RegisterUser(this.userRepository);

  Future<void> execute(String username, String password) async {
    if (username.isEmpty || password.isEmpty) {
      throw Exception("Le nom d'utilisateur et le mot de passe ne peuvent pas être vides.");
    }

    if (password.length < 8) {
      throw Exception("Le mot de passe doit contenir au moins 8 caractères.");
    }

    final newUser = User(DateTime.now().toString(), username, password);
    await userRepository.saveUser(newUser);
  }
}
```

3. Code du Repository de la couche Domaine

Dans cette couche, le **Repository** ne fait que décrire ce que l'on peut faire sans dire comment cela sera fait.

```
// Repository dans la couche Domaine
abstract class UserRepository {
  Future<void> saveUser(User user);
  Future<User?> getUserById(String id);
}
```

4. Code du Repository de la couche Data

Ici, on implémente le **Repository** et on décide **comment** les utilisateurs seront enregistrés. Pour cet exemple, on utilise un simple stockage en mémoire.

```
// Repository dans la couche Data (implémentation en mémoire)
class InMemoryUserRepository implements UserRepository {
  final Map<String, User> users = {};

  @override
  Future<void> saveUser(User user) async {
    users[user.id] = user;
    print("Utilisateur enregistré : ${user.username}");
  }

  @override
  Future<User?> getUserById(String id) async {
    return users[id];
  }
}
```

5. Code global : Les quatre communiquent ensemble

Voici un exemple qui montre comment faire interagir l'**Entity**, le **Use Case**, et les **Repositories**.

```
void main() async {
  // Créer une instance du repository de la couche Data
  final userRepository = InMemoryUserRepository();

  // Créer une instance du Use Case
  final registerUserUseCase = RegisterUser(userRepository);

  try {
    // Exécuter le Use Case pour enregistrer un utilisateur
    await registerUserUseCase.execute("john_doe", "securePassword123");

    // Récupérer l'utilisateur enregistré et afficher ses informations
    final registeredUser = await userRepository.getUserById("1");
    if (registeredUser != null) {
      print("Utilisateur enregistré avec succès : ${registeredUser.username}");
    }
  } catch (e) {
    print("Erreur : $e");
  }
}
```

Résumé du fonctionnement :

1. **Entity (User)** : Représente les utilisateurs de ton application.
2. **Use Case (RegisterUser)** : Encapsule la logique pour enregistrer un nouvel utilisateur.
3. **Repository (Domaine)** : Définit ce qu’on peut faire avec les utilisateurs (enregistrer, récupérer).
4. **Repository (Data)** : Gère les détails d’enregistrement des utilisateurs en mémoire.

Ce code montre comment les différentes couches de l'architecture **Clean Code** fonctionnent ensemble pour garder la logique métier séparée des détails techniques, comme la manière dont les données sont stockées.

Models

Dans l'architecture **Clean Code**, les **models** de la couche **Data** sont des objets utilisés pour manipuler les **données brutes** qui viennent de l'extérieur (API, base de données, fichiers, etc.). Leur but est de représenter les données **dans leur format de stockage ou d'échange** (par exemple, le format JSON d'une API ou la structure d'une table dans une base de données). Ces **models** ne contiennent pas de logique métier complexe, mais sont utilisés pour stocker, récupérer ou échanger des données.

À quoi sert un model dans la couche Data ?

1. **Conversion des données externes** : Les **models** dans la couche Data permettent de **convertir** les données qui viennent de l'extérieur dans un format compréhensible par les autres parties de l'application. Par exemple, si tu reçois des données JSON d'une API, tu vas d'abord les convertir en un **model** Dart avant de les utiliser.
2. **Mappe les données au format de stockage** : Ils permettent de **mapper** les données telles qu'elles sont stockées (dans une base de données, fichier, ou JSON) vers des **Entities** (couche métier). Parfois, les données en base ou dans les API ont des noms ou structures différents de celles utilisées dans le domaine de l'application. Le **model** gère cette conversion.

À quoi renvoie un model dans la couche Data ?

Les **models** dans cette couche renvoient généralement à des objets structurés pour stocker des données ou les échanger. Ils agissent comme des "containers" pour des informations qui viennent de sources externes et doivent être **adaptées** avant d'être utilisées dans la logique métier de l'application.

Comment se présente un model ?

Voici comment un **model** dans la couche **Data** pourrait se présenter dans une application en Dart. Prenons l'exemple d'un utilisateur dans une API qui retourne des données au format JSON.

Exemple de model dans la couche Data :

Disons que tu reçois les données suivantes d'une API :

```
{
  "id": "1",
  "user_name": "john_doe",
  "password_hash": "abc123hash"
}
```

Tu vas créer un **model** pour représenter ces données dans ton application :

```
// Model dans la couche Data : Représentation de l'utilisateur venant d'une API
class UserModel {
  final String id;
  final String username;
  final String passwordHash;

  UserModel({
    required this.id,
    required this.username,
    required this.passwordHash,
  });

  // Factory method pour créer un UserModel à partir d'un JSON
  factory UserModel.fromJson(Map<String, dynamic> json) {
    return UserModel(
      id: json['id'],
      username: json['user_name'],
      passwordHash: json['password_hash'],
    );
  }

  // Méthode pour convertir le UserModel en JSON (pour renvoyer à l'API)
  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'user_name': username,
      'password_hash': passwordHash,
    };
  }
}
```

Explication du code :

1. `UserModel` : Ce modèle est une représentation des données de l'utilisateur venant d'une source externe (ici, une API).
2. `fromJson` : C'est une méthode qui convertit les données JSON reçues en un objet `UserModel` pour qu'elles puissent être utilisées dans ton application.
3. `toJson` : Cette méthode prend un objet `UserModel` et le convertit à nouveau en JSON, si tu as besoin de renvoyer ces données vers l'API ou de les sauvegarder.

Différence entre Models et Entities dans l'architecture Clean Code

- **Entity (couche Domaine)** : L'Entity représente l'objet "réel" que ton application utilise dans sa **logique métier**. Elle contient des règles spécifiques à l'application. Par exemple, un utilisateur dans la couche **Domaine** pourrait contenir des méthodes pour valider son mot de passe, etc.
- **Model (couche Data)** : Le **Model** est une **représentation des données externes** telles qu'elles sont stockées ou échangées. Il n’a pas de logique métier, mais il est utilisé pour transférer les données entre l'extérieur (API, base de données) et l'application.

Comment un Model communique avec une Entity ?

Souvent, un **model** doit être **converti** en **Entity** pour être utilisé dans la logique métier. Par exemple, si tu reçois un utilisateur d'une API au format JSON, tu utilises un **model** pour lire ces données, mais ensuite tu les **transformes** en une **Entity** avant de les utiliser dans le Use Case.

Exemple de conversion entre un Model et une Entity :

Disons que nous avons une **Entity** `User` dans la couche **Domaine** :

```
// Entity dans la couche Domaine
class User {
  final String id;
  final String username;
  final String password;

  User(this.id, this.username, this.password);
}
```

Pour convertir un **model** de la couche **Data** en une **Entity**, tu peux créer une méthode comme celle-ci :

```
// Méthode pour convertir un UserModel en Entity User
User toEntity(UserModel userModel) {
  return User(
    userModel.id,
    userModel.username,
    userModel.passwordHash, // ou tu peux aussi vouloir transformer cela selon ta logique métier
  );
}
```

Exemple complet : Du Model à l'Entity

1. L'API renvoie un JSON qui est converti en un **Model** (via `fromJson`).
2. Ce **Model** est ensuite converti en une **Entity** pour être utilisé dans la logique métier (par exemple dans un Use Case).

```
void main() {
  // JSON simulé reçu d'une API
  Map<String, dynamic> apiResponse = {
    "id": "1",
    "user_name": "john_doe",
    "password_hash": "abc123hash"
  };

  // 1. Créer un model depuis le JSON
  UserModel userModel = UserModel.fromJson(apiResponse);
  print("Model : ${userModel.username}");

  // 2. Convertir le model en Entity pour l'utiliser dans la logique métier
  User userEntity = toEntity(userModel);
  print("Entity : ${userEntity.username}");
}
```

Conclusion :

- Les **Models** dans la couche **Data** servent à manipuler les données brutes venant de l'extérieur, comme les API ou les bases de données.
- Ils sont souvent utilisés pour **convertir** ces données dans un format que l'application peut comprendre (via les **Entities**).
- Les **Models** et les **Entities** sont différents. Les **Models** représentent les données externes, tandis que les **Entities** sont utilisées dans la logique métier de l'application.

Cela te permet d'avoir une architecture où chaque couche est indépendante et concentrée sur un seul rôle, ce qui rend ton code plus facile à maintenir et à faire évoluer.

Différence entre Models et Entities dans l'architecture Clean Code

Voici une explication simple de la différence entre **Models** et **Entities** dans l'architecture **Clean Code**, adaptée à un débutant.

1. Models :

- **Qu'est-ce que c'est ?**
  Les **Models** sont des objets qui représentent les données **telles qu'elles sont** dans des systèmes externes, comme des API, des bases de données ou des fichiers. Par exemple, si tu as une API qui retourne des informations sur les utilisateurs, le **Model** va représenter la structure de ces données.
- **À quoi servent-ils ?**
  Les **Models** permettent de **convertir** les données d'un format brut (comme JSON) en un format que ton application peut utiliser. Ils sont souvent utilisés pour **recevoir** des données d'une API ou **préparer** des données à envoyer.
- **Exemple :**

Si tu reçois ce JSON d'une API :

```
{
  "id": "1",
  "user_name": "john_doe",
  "password_hash": "abc123hash"
}
```

Tu peux créer un **Model** qui ressemble à cela :

```
class UserModel {
  final String id;
  final String username;
  final String passwordHash;

  UserModel(this.id, this.username, this.passwordHash);

  factory UserModel.fromJson(Map<String, dynamic> json) {
    return UserModel(
      json['id'],
      json['user_name'],
      json['password_hash'],
    );
  }
}
```

2. Entities :

- **Qu'est-ce que c'est ?**
  Les **Entities** représentent des objets **dans la logique métier** de ton application. Elles contiennent les **règles et comportements** associés à ces objets. Par exemple, dans un système de gestion d'utilisateurs, une **Entity** peut inclure des méthodes pour valider un mot de passe ou mettre à jour les informations d'un utilisateur.
- **À quoi servent-elles ?**
  Les **Entities** sont utilisées dans la **logique métier** de ton application. Elles sont responsables des règles et des opérations que tu peux effectuer sur les données. Elles interagissent avec les **Use Cases** et sont souvent créées à partir des **Models**.
- **Exemple :**

Pour l'utilisateur, tu pourrais avoir une **Entity** comme ceci :

```
class User {
  final String id;
  final String username;
  String password;

  User(this.id, this.username, this.password);

  void updatePassword(String newPassword) {
    // Logique pour mettre à jour le mot de passe
    if (newPassword.length < 8) {
      throw Exception("Le mot de passe doit contenir au moins 8 caractères.");
    }
    password = newPassword;
  }
}
```

Ici, `User` contient des méthodes pour interagir avec l'utilisateur et appliquer des règles de validation.

En résumé :

- **Models** :
  - Représentent des données venant de sources externes.
  - Ne contiennent pas de logique métier, juste des propriétés pour stocker des données.
  - Utilisés pour **recevoir** ou **envoyer** des données.
- **Entities** :
  - Représentent des objets dans la logique métier de l'application.
  - Contiennent des règles et des comportements (méthodes).
  - Utilisées pour manipuler et interagir avec les données de manière logique.

Visualisation :

- **Models** = Ce que tu reçois d'une API (comme un formulaire à remplir avec des données).
- **Entities** = Ce que tu fais avec ces données (comme les règles que tu appliques une fois que le formulaire est rempli).

Exemple pratique :

1. Tu reçois des données utilisateur d'une API, que tu stockes dans un **Model**.
2. Ensuite, tu crées une **Entity** à partir de ce **Model** pour effectuer des opérations (comme changer le mot de passe).

Cette séparation aide à garder ton code bien organisé, ce qui facilite la maintenance et l'évolution de l'application.

Résumé des différences entre Entities et Models :

Aspect	Entities (Domaine)	Models (Data)
Rôle	Représentent les objets métiers et la logique de l'application.	Représentent les données brutes échangées avec des sources externes.
Couche	Domaine (logique métier).	Data (stockage et échange de données).
Logique métier	Contiennent des règles métier importantes (validation, calculs).	Ne contiennent pas de logique métier, juste des données brutes.
Indépendance des technologies	Indépendantes des systèmes de stockage (API, BDD, etc.).	Dépendent des formats externes (API, BDD, JSON, etc.).
Modifications	Changent lorsque la logique métier évolue.	Changent lorsque le stockage ou le format de données change.
Exemple d'utilisation	Un utilisateur dans le système avec des règles comme validation du mot de passe.	Un utilisateur représenté en JSON reçu d'une API.

C'est Quoi Les Remote Data Source ,Le Local Data Source , Les Raw Data Ca veut dire quoi

Dans l'architecture **Clean Code**, les termes **Remote Data Source**, **Local Data Source**, et **Raw Data** sont utilisés pour décrire différentes façons de gérer et d'accéder aux données dans ton application. Voici une explication simple de chacun de ces concepts :

1. Remote Data Source :

- **Qu'est-ce que c'est ?**
  Un **Remote Data Source** est une source de données qui est située **à distance**, généralement sur un serveur, comme une API web. Lorsque ton application a besoin de données qui ne sont pas stockées localement, elle les récupère à partir de cette source.
- **À quoi ça sert ?**
  Cela te permet d'accéder à des données qui peuvent être partagées par plusieurs utilisateurs ou qui sont mises à jour fréquemment. Par exemple, tu pourrais récupérer des données d'utilisateurs, des articles, ou toute autre information à partir d'un serveur distant.
- **Exemple :**
  Lorsque ton application fait une requête à une API pour obtenir les informations d'un utilisateur :

```
class UserRemoteDataSource {
 final http.Client client;

 UserRemoteDataSource(this.client);

 Future<UserModel> fetchUser(String userId) async {
  final response = await client.get('https://api.example.com/users/$userId');
  if (response.statusCode == 200) {
   return UserModel.fromJson(json.decode(response.body));
  } else {
   throw Exception('Failed to load user');
  }
 }
}
```

2. Local Data Source :

- **Qu'est-ce que c'est ?**
  Un **Local Data Source** est une source de données qui est **stockée localement** sur l'appareil de l'utilisateur. Cela peut être dans une base de données locale (comme SQLite) ou dans un stockage de fichiers.
- **À quoi ça sert ?**
  Les **Local Data Sources** permettent d'accéder rapidement aux données sans nécessiter de connexion Internet. Elles sont particulièrement utiles pour stocker des données qui sont souvent utilisées, comme les préférences de l'utilisateur ou les données qui ne changent pas fréquemment.
- **Exemple :**
  Voici comment tu pourrais avoir un **Local Data Source** avec SQLite

```
class UserLocalDataSource {
 final Database database;

 UserLocalDataSource(this.database);

 Future<void> saveUser(UserModel user) async {
  await database.insert('users', user.toJson());
 }

 Future<UserModel?> getUser(String userId) async {
  final List<Map<String, dynamic>> maps = await database.query(
   'users',
   where: 'id = ?',
   whereArgs: [userId],
  );
  if (maps.isNotEmpty) {
   return UserModel.fromJson(maps.first);
  }
  return null;
 }
}
```

3. Raw Data :

- **Qu'est-ce que c'est ?**
  Le **Raw Data** (données brutes) désigne les données **dans leur format d'origine**, avant qu'elles ne soient traitées ou transformées par l'application. Cela peut inclure des données JSON reçues d'une API, des données lues à partir d'un fichier, ou toute autre forme de données qui n'a pas encore été structurée ou interprétée.
- **À quoi ça sert ?**
  Comprendre les données brutes est crucial car cela permet de savoir comment elles doivent être traitées pour être utilisées dans l'application. Cela aide également à définir les **Models** qui vont représenter ces données dans le code.
- **Exemple :**
  Si tu reçois des données d'une API sous forme de JSON : `{ "id": "1", "user_name": "john_doe", "password_hash": "abc123hash" }` Ces données brutes sont ce que tu reçois avant de les transformer en **Model** ou en **Entity**.

En résumé :

- **Remote Data Source** : Données venant d'un serveur ou d'une API à distance. Utile pour des données partagées et mises à jour fréquemment.
- **Local Data Source** : Données stockées localement sur l'appareil, permettant un accès rapide sans connexion Internet.
- **Raw Data** : Données dans leur format d'origine avant traitement, servant de base pour créer des **Models** ou **Entities**.

Ces concepts aident à structurer l'architecture de ton application, en définissant clairement comment les données sont gérées et utilisées à différents niveaux de l'application.

Projet Final

Présentation de l'exemple et de la logique métier

Pour cet exemple, on va gérer des **utilisateurs** dans une application (User Management). Voici ce qu'on va faire :

1. **Entities** : Définir l'entité qui représente un **utilisateur** dans la couche **Domain**.
2. **Use Cases** : Créer les cas d'utilisation (Use Cases) pour manipuler les utilisateurs (ex: obtenir la liste des utilisateurs, ajouter un utilisateur).
3. **Repository (Domain)** : Définir un contrat (interface) qui décrit comment on peut interagir avec les données utilisateurs depuis la couche **Domain**.
4. **Repository (Data)** : Implémenter ce contrat avec les détails d'implémentation pour récupérer les données (depuis une API ou une base de données).
5. **Models** : Définir le modèle (Model) qui représente l'utilisateur tel qu'il est stocké dans les sources de données (API ou base de données).
6. **Widgets (Presentation)** : Créer un **widget** qui utilise ces données pour afficher une liste d'utilisateurs.

Arborescence des fichiers pour cet exemple

```
lib/
├── features/
│   ├── user_management/
│   │   ├── data/
│   │   │   ├── models/
│   │   │   │   ├── user_model.dart
│   │   │   ├── repositories/
│   │   │   │   ├── user_repository_impl.dart
│   │   │   ├── data_sources/
│   │   │   │   ├── user_remote_data_source.dart
│   │   │   │   ├── user_local_data_source.dart
│   │   ├── domain/
│   │   │   ├── entities/
│   │   │   │   ├── user.dart
│   │   │   ├── repositories/
│   │   │   │   ├── user_repository.dart
│   │   │   ├── use_cases/
│   │   │   │   ├── get_users.dart
│   │   ├── presentation/
│   │   │   ├── widgets/
│   │   │   │   ├── user_list_widget.dart
```

1. Entity : `user.dart`

Les **entities** sont les objets métiers utilisés dans la logique métier. Elles ne contiennent pas de détails sur comment ou où les données sont stockées.

```
// domain/entities/user.dart

class User {
  final String id;
  final String name;
  final String email;

  User({
    required this.id,
    required this.name,
    required this.email,
  });
}
```

- Ici, `User` est une **entity** qui représente un utilisateur avec un identifiant, un nom, et une adresse e-mail.

2. Use Case : `get_users.dart`

Un **use case** est une action spécifique qui se produit dans ton application. Par exemple, récupérer tous les utilisateurs.

```
// domain/use_cases/get_users.dart

import '../entities/user.dart';
import '../repositories/user_repository.dart';

class GetUsers {
  final UserRepository repository;

  GetUsers(this.repository);

  Future<List<User>> execute() async {
    return await repository.getUsers();
  }
}
```

- Ce cas d'utilisation (`GetUsers`) appelle le **repository** pour obtenir la liste des utilisateurs.

3. Repository (Domain) : `user_repository.dart`

Le **repository** dans la couche **Domain** est un contrat qui définit les méthodes disponibles pour interagir avec les données des utilisateurs. Il ne connaît pas les détails d'où ces données proviennent (API ou base de données).

```
// domain/repositories/user_repository.dart

import '../entities/user.dart';

abstract class UserRepository {
  Future<List<User>> getUsers();
  Future<void> addUser(User user);
}
```

- `UserRepository` est une interface qui déclare deux méthodes : `getUsers()` pour obtenir les utilisateurs et `addUser()` pour ajouter un utilisateur.

4. Repository (Data) : `user_repository_impl.dart`

L'implémentation du **repository** dans la couche **Data** contient les détails de comment et où les données sont récupérées (depuis une API ou une base de données locale).

```
// data/repositories/user_repository_impl.dart

import '../../domain/entities/user.dart';
import '../../domain/repositories/user_repository.dart';
import '../data_sources/user_remote_data_source.dart';

class UserRepositoryImpl implements UserRepository {
  final UserRemoteDataSource remoteDataSource;

  UserRepositoryImpl(this.remoteDataSource);

  @override
  Future<List<User>> getUsers() async {
    final userModels = await remoteDataSource.getUsersFromApi();
    return userModels.map((userModel) => User(
      id: userModel.id,
      name: userModel.name,
      email: userModel.email,
    )).toList();
  }

  @override
  Future<void> addUser(User user) async {
    await remoteDataSource.addUserToApi(user);
  }
}
```

- `UserRepositoryImpl` utilise un **data source** (une API par exemple) pour récupérer ou ajouter les utilisateurs.

5. Model : `user_model.dart`

Le **model** représente les données telles qu'elles sont reçues ou envoyées à une source de données (comme une API).

```
// data/models/user_model.dart

class UserModel {
  final String id;
  final String name;
  final String email;

  UserModel({
    required this.id,
    required this.name,
    required this.email,
  });

  factory UserModel.fromJson(Map<String, dynamic> json) {
    return UserModel(
      id: json['id'],
      name: json['name'],
      email: json['email'],
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'name': name,
      'email': email,
    };
  }
}
```

- `UserModel` est utilisé pour la conversion des données JSON reçues depuis l'API en objet métier.

6. Data Sources : `user_remote_data_source.dart`

Le **data source** est responsable de la récupération des données depuis une source externe (API, base de données locale).

```
// data/data_sources/user_remote_data_source.dart

import '../models/user_model.dart';
import 'dart:convert';
import 'package:http/http.dart' as http;

class UserRemoteDataSource {
  final http.Client client;

  UserRemoteDataSource(this.client);

  Future<List<UserModel>> getUsersFromApi() async {
    final response = await client.get(Uri.parse('https://api.example.com/users'));

    if (response.statusCode == 200) {
      final List<dynamic> data = json.decode(response.body);
      return data.map((userJson) => UserModel.fromJson(userJson)).toList();
    } else {
      throw Exception('Failed to load users');
    }
  }

  Future<void> addUserToApi(User user) async {
    final response = await client.post(
      Uri.parse('https://api.example.com/users'),
      body: json.encode({
        'name': user.name,
        'email': user.email,
      }),
    );

    if (response.statusCode != 201) {
      throw Exception('Failed to add user');
    }
  }
}
```

7. Widget : `user_list_widget.dart`

Enfin, dans la couche **Presentation**, on crée un widget qui affiche la liste des utilisateurs. Ce widget interagit avec le **state management** et les **use cases** pour obtenir les données.

```
// presentation/widgets/user_list_widget.dart

import 'package:flutter/material.dart';
import '../../domain/use_cases/get_users.dart';
import '../../domain/entities/user.dart';

class UserListWidget extends StatelessWidget {
  final GetUsers getUsersUseCase;

  UserListWidget({required this.getUsersUseCase});

  @override
  Widget build(BuildContext context) {
    return FutureBuilder<List<User>>(
      future: getUsersUseCase.execute(),
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
          return CircularProgressIndicator();
        } else if (snapshot.hasError) {
          return Text('Error: ${snapshot.error}');
        } else if (!snapshot.hasData || snapshot.data!.isEmpty) {
          return Text('No users available');
        }

        final users = snapshot.data!;
        return ListView.builder(
          itemCount: users.length,
          itemBuilder: (context, index) {
            final user = users[index];
            return ListTile(
              title: Text(user.name),
              subtitle: Text(user.email),
            );
          },
        );
      },
    );
  }
}
```

C'est La Fin

C'est la fin. Merci d'avoir lu. J'espère que cela t'a vraiment aidé et que tu as au moins compris une notion de l'architecture Clean. Je suis Essome Menang Fred Arthur, développeur mobile et web à mes heures perdues, et étudiant en sciences de l'intelligence artificielle en autodidacte.

retouve moi sur Twitter(X): **https://x.com/essomefre**🌏
Linkedin:**www.linkedin.com/in/essome-fred-arthur-a6a449318**
github:**https://github.com/EFredArthur**