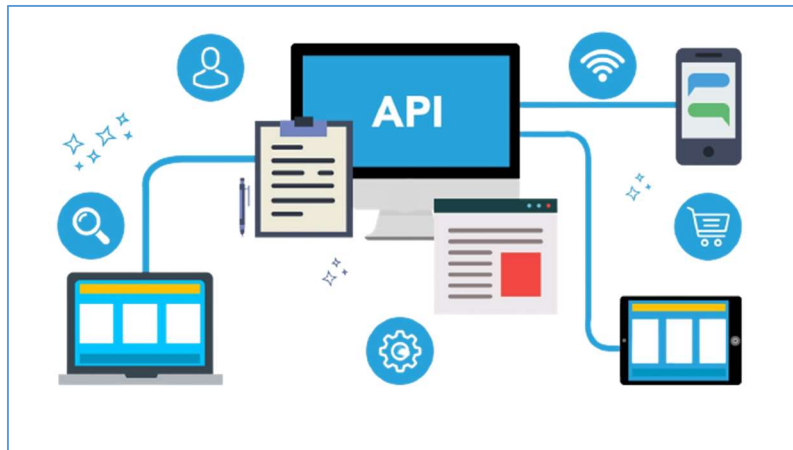


---

# Atelier 7 Flutter

## Utilisation de GetX pour la gestion d'état et de Dio pour les appels API

---



### I. GetX

Cet atelier vous montre comment utiliser GetX (également appelé Get) pour gérer les états dans Flutter. Nous allons créer une application simple qui affiche une liste de produits donnés. Un utilisateur peut ajouter ou supprimer des produits à partir d'un endroit appelé liste de souhaits.

GetX est une bibliothèque de gestion d'état rapide, stable et légère en flutter. Il y a tellement de bibliothèques de gestion d'état dans flutter comme MobX, BLoC, Redux, Provider, etc. GetX est également un micro Framework puissant et en l'utilisant, nous pouvons gérer les états, faire du routage et effectuer une injection de dépendance.

Il existe trois principes de GetX :

- **Performance** : Par rapport aux autres bibliothèques de gestion d'état, GetX est la meilleure car elle consomme un minimum de ressources et offre de meilleures performances.
- **Productivité** : La syntaxe de GetX est simple donc productive. Cela fait gagner beaucoup de temps aux développeurs et augmente la vitesse de l'application car elle n'utilise pas de ressources supplémentaires. Il utilise uniquement les ressources qui sont actuellement nécessaires et une fois son travail terminé, les ressources seront automatiquement libérées. Si toutes les ressources sont chargées dans la mémoire, ce ne sera pas aussi productif. Il vaut donc mieux utiliser GetX pour cela.
- **Organisation** : le code GetX est organisé en vue, logique, navigation et injection de dépendances. Nous n'avons donc plus besoin de contexte pour naviguer vers un autre écran. Nous pouvons naviguer vers l'écran sans utiliser le contexte afin de ne pas dépendre de l'arborescence des widgets.

GetX est un framework de gestion d'état pour Flutter, souvent utilisé pour simplifier le développement d'applications grâce à des fonctionnalités telles que :

- **Gestion d'état** : GetX permet de gérer l'état de votre application de manière réactive. Vous pouvez créer des variables réactives qui se mettent à jour automatiquement lorsqu'elles sont modifiées, sans avoir besoin de widgets setState. il existe deux types de gestion d'état :
  - Gestionnaire d'état simple : il utilise GetBuilder .
- **Navigation** : GetX offre une gestion simplifiée de la navigation, sans avoir besoin d'un Navigator explicite. Cela permet de définir et de gérer les routes avec plus de flexibilité. Si nous voulons créer des widgets tels que Snackbar, Bottomsheets, boîtes de dialogue, etc., nous pouvons utiliser GetX pour cela, car GetX peut créer ces widgets sans utiliser de contexte.
- **Gestion des dépendances** : GetX facilite l'injection de dépendances, en vous permettant de partager des contrôleurs et des services dans toute l'application sans avoir besoin de bibliothèques comme Provider. Si nous voulons récupérer des données d'une autre classe, à l'aide de GetX, nous pouvons le faire en une seule ligne de code. Ex : Get.put()

Nous savons tous que Flutter est rapide, mais il y a des choses qui doivent être évitées comme lorsque vous naviguez vers un autre écran, nous utilisons Navigator.of(context).push(context, builder(.....)) et le faisons tellement de fois n'est pas une bonne approche pour un développeur. Au lieu d'écrire ceci, nous pouvons simplement écrire Get.to(HomePage()) pour accéder à HomePage.

Lorsque nous voulons mettre à jour un widget, nous utilisons souvent StatefulWidget pour cela. Au lieu de créer des StatefulWidgets, nous pouvons faire la même tâche en utilisant Stateless Widget également en utilisant GetX. En ajoutant « .obs » à la variable qui doit être mise à jour, et en plaçant le Widget dans Obx, nous pouvons mettre à jour l'écran lorsqu'une variable change la valeur sans rafraîchir toute la page.

Lorsque nous naviguons entre les écrans, créons des widgets comme un snack-bar, des feuilles de fond, etc., nous n'avons plus besoin d'utiliser le contexte. Ainsi, grâce à GetX, les performances augmentent.

## II. Dio

Dans Flutter, **Dio** est une bibliothèque très populaire utilisée pour effectuer des requêtes HTTP. Elle permet de faire des appels API (GET, POST, PUT, DELETE, etc.) de manière asynchrone et offre plusieurs fonctionnalités avancées telles que :

1. **Requêtes HTTP** : Dio prend en charge toutes les méthodes HTTP standard comme GET, POST, PUT, DELETE, etc.
2. **Gestion des en-têtes (Headers)** : Vous pouvez facilement ajouter des en-têtes personnalisés à vos requêtes.
3. **Intercepteurs (Interceptors)** : Dio permet d'ajouter des intercepteurs pour intercepter les requêtes ou les réponses avant qu'elles ne soient traitées, ce qui est utile pour gérer les tokens d'authentification comme JWT ou la gestion des erreurs globales.
4. **Annulation des requêtes** : Vous pouvez annuler une requête en cours à l'aide d'un CancelToken.
5. **Téléchargement et upload de fichiers** : Dio permet facilement d'envoyer des fichiers ou de télécharger des fichiers avec des capacités de suivi de la progression.
6. **Time-out** : Vous pouvez définir des limites de temps d'attente pour les requêtes afin d'éviter qu'elles ne se bloquent.

7. **Transformation des données** : Dio permet de transformer les données des requêtes ou des réponses avant de les envoyer ou de les traiter.

### III. Installation et appel

- **getx**

Référence : <https://pub.dev/packages/get>

On peut installer getx via ligne de commande

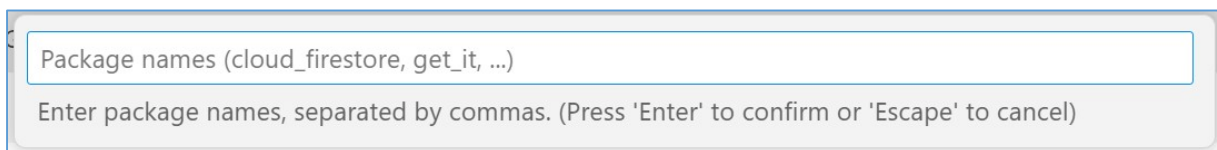
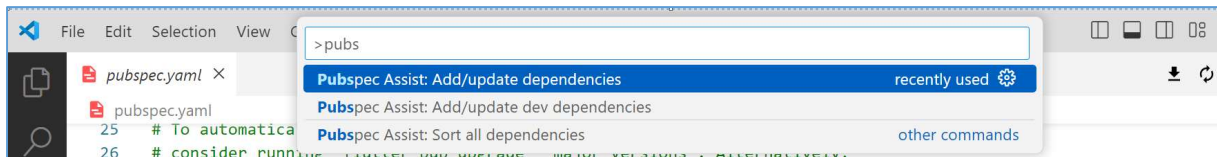
```
flutter pub add get
```

Ou bien à partir de l'extension pubspec :

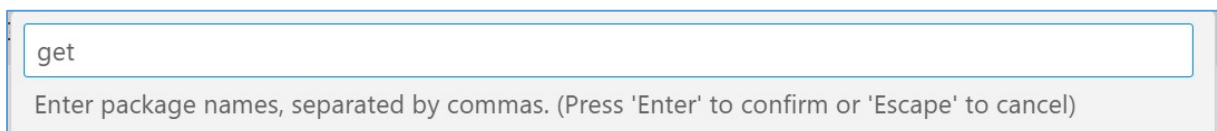
Accéder au fichier pubspec.yaml

Ctrl + shift + p

Puis choisir Pubspec Assist Add/update dependencies

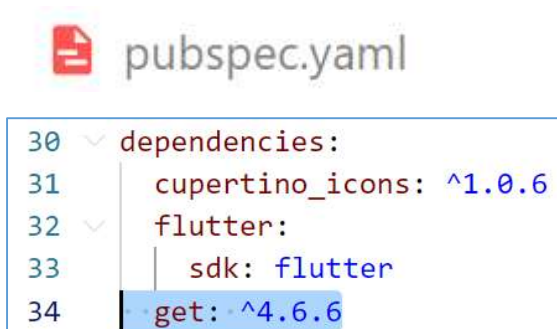


Saisir get



Valider par Entrée

On aura la dépendance ajoutée :



Lors de son utilisation, importer get dans le fichier :

```
import 'package:get/get.dart';
```

- **dio**

Référence : <https://pub.dev/packages/dio>

On peut installer getx via ligne de commande

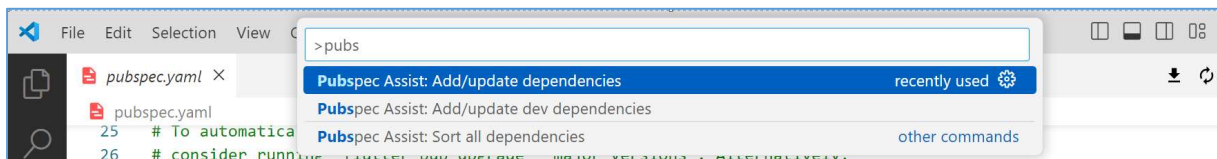
```
flutter pub add dio
```

Ou bien à partir de l'extension pubspec :

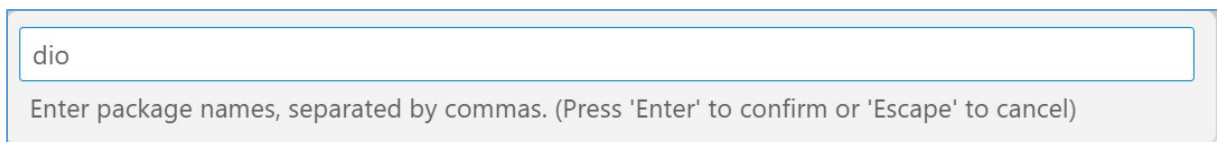
Accéder au fichier pubspec.yaml

Ctrl + shift + p

Puis choisir Pubspec Assist Add/update dependencies



Puis saisir dio et valider par Entrée.



 pubspec.yaml

```
30 dependencies:
31   cupertino_icons: ^1.0.6
32   dio: ^5.7.0
33 flutter:
34   sdk: flutter
35   get: ^4.6.6
```

Lors de son utilisation, importer dio dans le fichier :

```
import 'package:dio/dio.dart';
```

## IV. Clean Architecture

A mesure que les applications deviennent de plus en plus complexes, la gestion du code devient un défi de taille. La clean architecture est un modèle de conception de logiciel qui répond à ce défi en favorisant la séparation du code en couches distinctes, facilitant ainsi la maintenabilité et l'évolutivité. Nous allons nous pencher sur la mise en œuvre pratique de l'architecture propre dans Flutter à l'aide de la bibliothèque GetX.

Une architecture propre repose sur la séparation du code en couches, chacune ayant une responsabilité distincte. Cette approche architecturale, popularisée par Robert C. Martin, vise à créer

des applications logicielles maintenables, testables et évolutives en séparant clairement les différentes responsabilités du code. Les **couches principales** comprennent :

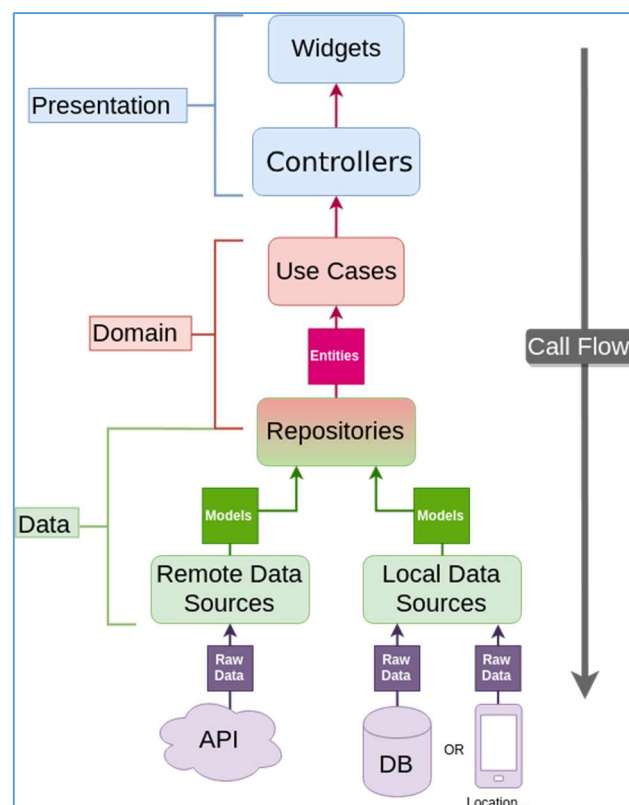
**Couche de présentation** (Presentation Layer) : cette couche concerne l'interface utilisateur et les interactions. Elle comprend des widgets, des vues et des contrôleurs responsables de la saisie utilisateur et de l'affichage des données.

**Couche d'application** (Application Layer) : optionnelle elle permet l'orchestration, gestion des services, flux de travail de haut niveau. La couche d'application contient la logique métier principale de l'application. Elle sert d'intermédiaire entre les couches de présentation et de domaine, coordonnant le flux de données et appliquant les règles métier. Utilisée seulement dans le cas de très grandes applications.

**Couche de domaine** (Domain Layer) : la couche de domaine définit la logique métier fondamentale et les modèles de domaine de l'application. Elle reste indépendante des cadres d'interface utilisateur ou des bases de données spécifiques, garantissant une réutilisabilité et une testabilité élevées.

*La logique métier* (business logic) désigne l'ensemble des règles, des processus et des décisions spécifiques à une application qui gèrent la manière dont les données doivent être manipulées en fonction des exigences d'un domaine d'activité. Exemple dans une application de commerce électronique : Calcul du total d'une commande en fonction du prix des produits, des réductions, des taxes, et des frais de livraison. Ou aussi la vérification que les produits sont en stock avant de permettre au client de valider une commande. Aussi, la gestion des retours et remboursements en fonction des règles établies (par exemple, un produit peut être retourné dans les 30 jours suivant l'achat).

**Couche de données** (Data Layer) : cette couche gère le stockage, la récupération et les interactions des données avec les API externes. Elle englobe les référentiels, les sources de données et les clients API.



### ***Les composants :***

**Contrôleurs** (Controllers) : Ils agissent comme des intermédiaires entre la présentation et le domaine. Ils reçoivent les entrées de l'utilisateur, les transmettent aux cas d'utilisation et mettent à jour l'interface en fonction des résultats.

**Cas d'utilisation** (Use Cases) : Ils encapsulent les règles métier et les actions que l'utilisateur peut effectuer. Ils reçoivent les requêtes des contrôleurs, interagissent avec les entités et les référentiels (repositories), puis retournent les résultats. Un use case peut interagir avec plusieurs repositories ou services, orchestrant les actions nécessaires pour atteindre un objectif métier. Par exemple, un use case pourrait appeler plusieurs repositories, combiner des données, et appliquer des règles de gestion avant de renvoyer un résultat.

**Entités** (Entities) : Ce sont les objets métier qui représentent les concepts de votre domaine. Elles représentent les concepts métier de votre application, avec une identité propre et des règles métier associées. Elles sont souvent persistées dans une base de données.

**Référentiels** (Repositories) : Ils sont responsables de la récupération et de la persistance des données. Ils peuvent interagir avec des sources de données distantes (API) ou locales (bases de données).

**Modèles** (Models) : Ce sont des représentations des données, souvent utilisées pour mapper les données brutes provenant des sources de données vers les entités du domaine. Ils peuvent être utilisés pour représenter des vues spécifiques d'une entité, par exemple pour l'affichage dans l'interface utilisateur. Ils sont souvent plus légers que les entités et peuvent contenir des données supplémentaires ou moins.

### ***Exemple :***

Imaginons un schéma représentant un "Produit".

Si ce "Produit" a une identité unique, des attributs comme le nom, le prix et une description, et s'il est utilisé pour représenter un produit réel dans votre application, il s'agit d'une entité.

Si ce "Produit" est une structure de données plus simple, utilisée uniquement pour afficher des informations sur un produit dans une liste, il s'agit d'un modèle.

Entité : C'est le concept métier lui-même (ex : un produit).

Modèle : C'est une représentation de ce concept, adaptée à un contexte spécifique (ex : un modèle de produit pour l'affichage dans une liste).

### ***Fonctionnement***

L'utilisateur interagit avec les widgets de l'interface utilisateur (couche Présentation).

Les contrôleurs reçoivent les événements et appellent les cas d'utilisation appropriés.

Les cas d'utilisation exécutent la logique métier, en interagissant avec les entités et les référentiels.

Les référentiels accèdent aux sources de données (API, base de données) pour récupérer ou enregistrer des données.

Les données récupérées sont mappées vers des modèles puis vers des entités.

Les résultats sont retournés à la couche Présentation pour mettre à jour l'interface utilisateur.

## V. Structure globale des dossiers



## VI. Créer le fichier utilitaire des constantes

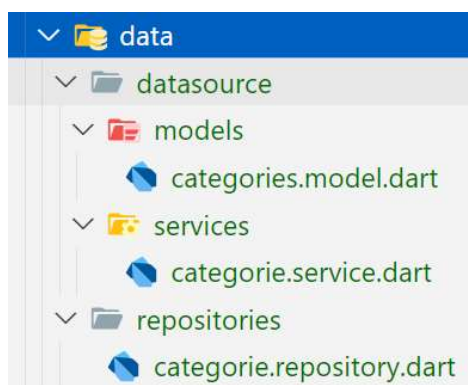
lib > utils >  constants.dart

```
const baseUrl="http://192.168.48.1:3001/api";
```

**Attention ! Mettre l'adresse ip adéquate de votre serveur.**

## VII. Couche Data

Par la suite nous allons construire cette arborescence à l'intérieur de la couche Data :



La couche data est essentielle car elle est responsable de la gestion des données de l'application. Cette couche se situe au niveau le plus bas de l'architecture et a pour rôle d'interagir avec les sources de données externes (API, base de données, services locaux ou distants) et de les adapter au reste de l'application.

La couche data interagit avec une API distante pour récupérer la liste des catégories.

Elle utilise un service pour envoyer une requête HTTP via un client comme Dio.

La réponse de l'API est en format JSON, et la couche data transforme ce JSON en un objet `CategorieModel`.

Le repository dans la couche data va ensuite prendre cet objet et le rendre disponible pour la couche domain en appliquant éventuellement des transformations supplémentaires).

La couche domain n'a pas besoin de savoir comment les données sont récupérées, seulement qu'elles sont disponibles via des méthodes du repository.

## 1. Créer le Model

/lib/data/datasource/models/categories.model.dart

lib > data > datasource > models >  categories.model.dart

```
class Categorie {
  String? id;
  String? nomcategorie;
  String? imagecategorie;

  Categorie({this.id, this.nomcategorie, this.imagecategorie});

  Categorie.fromJson(Map<String, dynamic> json) {
    id = json['_id'];
    nomcategorie = json['nomcategorie'];
    imagecategorie = json['imagecategorie'];
  }

  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = <String, dynamic>{};
    data['_id'] = id;
    data['nomcategorie'] = nomcategorie;
    data['imagecategorie'] = imagecategorie;
    return data;
  }
}
```

### Explication

Les variables `String?` peuvent avoir des valeurs nulles (null).

```
Categorie.fromJson(Map<String, dynamic> json) {
  id = json['_id'];
  nomcategorie = json['nomcategorie'];
  imagecategorie = json['imagecategorie'];
}
```



Ce constructeur fromJson permet de créer un objet Categorie à partir d'une structure de données JSON (qui est souvent renvoyée par une API).

Map<String, dynamic> représente un dictionnaire JSON où la clé est une chaîne de caractères (String), et la valeur peut être de n'importe quel type (dynamic).

Le constructeur extrait les valeurs des champs du JSON, tels que \_id, nomcategorie, et imagecategorie, et les affecte aux attributs de la classe.

```
Map<String, dynamic> toJson() {  
  final Map<String, dynamic> data = <String, dynamic>{};  
  data['_id'] = id;  
  data['nomcategorie'] = nomcategorie;  
  data['imagecategorie'] = imagecategorie;  
  return data;  
}
```

Cette méthode toJson est utilisée pour convertir un objet Categorie en une structure JSON (sous forme de Map<String, dynamic>) afin de l'envoyer à une API.

Elle crée un Map vide, puis ajoute les valeurs de l'objet actuel dans ce dictionnaire, avec les clés correspondantes, avant de le retourner.

## Remarque

Il est possible d'obtenir le code de cette classe directement en convertissant le code json en dart.

Exemple de site :

[https://javiercbk.github.io/json\\_to\\_dart/](https://javiercbk.github.io/json_to_dart/)

javiercbk.github.io/json\_to\_dart/

## JSON to Dart

Paste your JSON in the textarea below, click convert and get your Dart classes for free.

JSON

```

1 {
2   "_id": 1,
3   "nomcategorie": "unecateg",
4   "imagecategorie": "image.jpg"
5 }

```

Your dart class name goes here

**Generate Dart** ☐ Use private fields

Copy Dart code to clipboard

```

class Autogenerated {
  int? id;
  String? nomcategorie;
  String? imagecategorie;

  Autogenerated({this.id, this.nomcategorie, this.imagecategorie});

  Autogenerated.fromJson(Map<String, dynamic> json) {
    id = json['_id'];
    nomcategorie = json['nomcategorie'];
    imagecategorie = json['imagecategorie'];
  }

  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = new Map<String, dynamic>();
    data['_id'] = this.id;
    data['nomcategorie'] = this.nomcategorie;
    data['imagecategorie'] = this.imagecategorie;
    return data;
  }
}

```

## 2. Créer le service

Ce service permet de récupérer des catégories depuis une API en utilisant Dio pour gérer les requêtes HTTP, avec des délais et une gestion des erreurs simplifiée.

/lib/data/datasource/services/categories.service.dart

lib > data > datasource > services >  categorie.service.dart

```

import 'package:dio/dio.dart';
import 'package:myflutterapplication/utils/constants.dart';

class CategorieService{
  late Dio dio;

  CategorieService(){
    BaseOptions options=BaseOptions(
      baseUrl: baseUrl,
      receiveDataWhenStatusError: true,
      connectTimeout: const Duration(seconds: 5),
      receiveTimeout: const Duration(seconds: 3),
    );
    dio=Dio(options);
  }
}

```

```

}
Future<List<dynamic>> getCategories() async{
  try{
    Response response=await dio.get('/categories');
    print(response.data.toString());
    return response.data;
  }
  catch(e){
    print(e.toString());
    return [];
  }
}
}

```

## Explication

Ce code utilise le package Dio, un client HTTP pour Dart/Flutter, afin de gérer les requêtes HTTP.

dio.dart : Ce package permet d'effectuer des requêtes HTTP de manière simple et efficace.

constants.dart : Ce fichier contient des constantes, plus précisément l'URL de base utilisée pour l'API.

CategorieService : Cette classe encapsule la logique pour interagir avec l'API et gérer les catégories.

late Dio dio : Le mot-clé late indique que la variable dio sera initialisée plus tard, ici dans le constructeur.

Le constructeur crée une instance de BaseOptions, qui contient des configurations pour les requêtes HTTP.

baseUrl: Récupéré de constants.dart, c'est l'URL de base pour les requêtes (par exemple, une URL d'API).

receiveDataWhenStatusError: Permet de recevoir des données même lorsque le statut HTTP indique une erreur.

connectTimeout et receiveTimeout: Spécifient les délais avant qu'une requête ne soit abandonnée (5 secondes pour la connexion et 3 secondes pour recevoir une réponse).

Cette méthode est asynchrone et retourne une Future contenant une liste de catégories.

Elle effectue une requête GET vers l'endpoint /categories.

En cas de succès, elle affiche et retourne les données reçues dans la réponse.

En cas d'erreur (ex : une exception lors de la requête), elle affiche l'erreur et retourne une liste vide.

Dans Flutter, lorsqu'on utilise la bibliothèque Dio pour faire des requêtes HTTP, la classe Response est utilisée pour représenter la réponse d'une requête. Le type Response de Dio contient des informations sur la réponse, comme le corps de la réponse, le code de statut HTTP, les en-têtes de réponse, et plus encore.

### 3. Créer le repository

Dans ce code on va définir un repository dans une architecture typique pour une application Flutter. Un repository sert d'intermédiaire entre la couche de services (qui interagit avec l'API) et la couche métier de l'application, et encapsule la logique de gestion des données.

Le `CategorieRepository` fait office de couche intermédiaire pour récupérer les catégories depuis le service (`CategorieService`), les transformer en objets `Categorie`, et les fournir à la partie logique ou UI de l'application.

`/lib/data/repositories/categorie.repository.dart`

lib > data > repositories >  categorie.repository.dart

```
import
'package:myflutterapplication/data/datasource/models/categories.model.dart';
import
'package:myflutterapplication/data/datasource/services/categorie.service.dart'
;

class CategorieRepository{
final CategorieService catserv ;

CategorieRepository({required this.catserv});
Future<List<Categorie>> getCategories() async{
final categories=await catserv.getCategories();
return categories.map((c) => Categorie.fromJson(c)).toList();
}
}
```

#### Explication

`CategorieRepository` : Cette classe encapsule la logique d'accès aux données des catégories en utilisant un service.

`final CategorieService catserv` : Ce champ est une instance du service `CategorieService`, injectée via le constructeur. Cela permet au repository d'utiliser ce service pour effectuer des requêtes HTTP.

Le constructeur prend une instance de `CategorieService` comme paramètre, qui est marquée comme `required` (obligatoire).

La méthode `getCategories` est asynchrone et retourne une liste de `Categorie`.

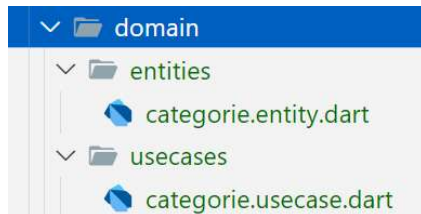
Étape 1 : Elle utilise le service `CategorieService` pour appeler la méthode `getCategories()`, qui renvoie une liste de catégories sous forme de données brutes (généralement des objets JSON).

Étape 2 : Les données JSON récupérées sont transformées en objets `Categorie` grâce à la méthode `fromJson()` présente dans le modèle `Categorie`.

Étape 3 : `map()` applique la fonction de transformation (`Categorie.fromJson`) à chaque élément de la liste brute, et `toList()` convertit l'itérateur renvoyé par `map` en une liste. Le repository ne renvoie pas directement les données brutes de l'API. Il les transforme en objets métier (`Categorie`) via `fromJson()`, garantissant ainsi que l'application ne manipule que des objets typés et structurés.

## VIII. Couche Domain

La couche Domain aura comme contenu



Rôle de la couche "domain"

- **Logique métier pure** : La couche "domain" contient les règles de gestion, les calculs et la logique qui sont spécifiques à l'application. Ces règles ne changent pas en fonction de la technologie utilisée (par exemple, que vous utilisiez une API REST, une base de données SQL ou NoSQL).
- **Indépendance technologique** : La couche "domain" ne devrait pas dépendre de détails d'implémentation, comme la manière dont les données sont récupérées (par exemple, une API ou une base de données). Cela permet de rendre la logique métier indépendante des choix techniques et facilite les modifications futures (comme un changement de base de données).
- **Testabilité** : En séparant la logique métier du reste de l'application, il devient beaucoup plus simple d'écrire des tests unitaires.
- **Représentation des concepts métier** : La couche "domain" permet de définir clairement les entités métier (comme `CategorieEntity`), qui sont les représentations des objets métier manipulés dans l'application. Cela aide à organiser et structurer les données de manière cohérente avec le domaine d'activité de l'application.

### 1. Créer Entity

Créer `/lib/domain/entities/categorie.entity.dart`

lib > domain > entities >  categorie.entity.dart

```
class CategorieEntity {  
  final String id;  
  final String nomcategorie;  
  final String imagecategorie;  
  
  CategorieEntity({required this.id, required this.nomcategorie, required  
this.imagecategorie});  
}
```

## Explication

La classe `CategorieEntity` représente une entité métier. Une entité est un objet qui a une identité unique et qui est important pour l'application d'un point de vue métier.

`CategorieEntity` est une représentation du concept de catégorie dans le domaine métier de l'application, sans se soucier des détails techniques (comme la manière dont les données sont récupérées ou stockées). Cette classe ne contient pas de méthodes spécifiques à la base de données ou à l'API, elle se concentre uniquement sur la représentation des données importantes pour le métier.

Cela sépare la représentation des données de la logique d'accès aux données, qui peut être gérée par des modèles ou des services dans d'autres couches (comme la couche "data").

Une `CategorieEntity` dans la couche domain est différente d'un modèle dans la couche data :

***La classe modèle (`Categorie` dans `categories.model.dart`) est souvent liée à des détails d'implémentation, comme la manière dont les données sont stockées ou récupérées (ex : JSON, API, base de données).***

***La classe entité représente la logique métier et est utilisée par la couche domain. Elle est souvent plus abstraite et indépendante des choix technologiques.***

## 2. Créer UseCase

Créer `/lib/domain/usecases/categorie.usecase.dart`

lib > domain > usecases >  categorie.usecase.dart

```
import
'package:myflutterapplication/data/repositories/categorie.repository.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';

class CategorieUseCase {
  final CategorieRepository _respository;

  CategorieUseCase({required CategorieRepository respository})
    : _respository = respository;

  Future<List<CategorieEntity?>>> fetchCategories() async {
    final result = await _respository.getCategories();

    final data = result.map((element) {
      return CategorieEntity(
        id : element.id ?? "",
        nomcategorie: element.nomcategorie ?? "",
        imagecategorie: element.imagecategorie ?? "",
      );
    }).toList();
    return data;
  }
}
```

```
}
```

## Explication

`CategorieUseCase({required CategorieRepository repository})` : C'est un constructeur pour la classe `CategorieUseCase`. Il prend un paramètre nommé obligatoire (required) de type `CategorieRepository` appelé `repository`.

`CategorieRepository repository` : C'est le paramètre qui sera passé lors de la création de l'objet. Ce paramètre représente une instance du repository pour la gestion des catégories.

`: _repository = repository;` : Cette partie est appelée la liste d'initialisation. Elle permet d'initialiser les champs de la classe avant que le corps du constructeur ne soit exécuté. Cela est particulièrement utile lorsque les champs doivent être initialisés avant toute autre logique.

`_repository = repository;` : Cela signifie que le champ privé `_repository` de la classe est initialisé avec la valeur du paramètre `repository` passé au constructeur.

Ce use case prend le résultat du repository (une liste d'objets `Categorie`), les transforme en entités métier (`CategorieEntity`), et applique des règles supplémentaires (comme le remplacement des valeurs null par des valeurs par défaut "").

Dans une architecture propre (Clean Architecture), les use cases jouent un rôle central dans la gestion de la logique métier spécifique et l'orchestration des actions.

Un Use Case représente une action ou un scénario métier spécifique. Chaque use case encapsule une seule opération métier bien définie et s'assure que la logique métier est respectée, indépendante des interfaces utilisateur ou des détails techniques.

Un use case concentre une logique métier précise, par exemple, "récupérer des catégories". Cela permet de séparer la logique métier des aspects techniques ou des détails d'implémentation (comme les appels API).

Les use cases peuvent être appelés par différentes parties de l'application (interface utilisateur, services, etc.), favorisant la réutilisation et le découplage. Chaque use case peut être testé et développé indépendamment des autres couches.

***Un repository est responsable de la gestion de l'accès aux données (récupération, création, mise à jour, suppression). Il sert de passerelle entre les données brutes (API, base de données, fichiers) et l'application.***

***Un use case gère la logique métier spécifique à un scénario ou une action donnée. Il ne se préoccupe pas de la manière dont les données sont récupérées ou stockées. Il coordonne les repositories pour accomplir une opération métier.***

## Remarque

La séparation entre les entités (entity) et les modèles (model) dans une architecture comme Clean Architecture peut sembler redondante au premier abord, mais elle a des raisons importantes. Voici pourquoi on utilise des entités dans les use cases, plutôt que des modèles, et pourquoi il est important de créer des entités séparées.

## 1. Séparation des préoccupations (Separation of Concerns)

Entité : représente un concept du domaine métier pur, sans dépendance aux détails de l'infrastructure (comme la base de données, les formats JSON, les API externes). Les entités sont généralement indépendantes des sources de données et sont directement liées à la logique métier.

Modèle : souvent lié à une source de données spécifique, comme un modèle de base de données ou un modèle JSON utilisé pour les appels d'API. Le modèle peut contenir des détails d'implémentation comme des ID ou des formats qui ne sont pas pertinents pour la logique métier.

La couche métier (domain) devrait rester indépendante des détails techniques de l'infrastructure (comme l'API, la base de données, etc.), d'où l'utilisation des entités qui reflètent uniquement le domaine métier.

## 2. Isolation de la logique métier

Les entités dans la couche métier sont des représentations pures des concepts de l'application. En évitant de mélanger ces entités avec les modèles (qui peuvent contenir des informations liées à la structure des données), on s'assure que la logique métier reste isolée et indépendante des aspects techniques (comme le stockage des données, la sérialisation ou la désérialisation).

Exemple :

Modèle : Un modèle de catégorie utilisé dans une base de données pourrait avoir des champs supplémentaires comme `createdAt`, `updatedAt`, ou d'autres métadonnées qui ne concernent pas directement la logique métier.

Entité : L'entité métier pourrait ne contenir que les champs nécessaires pour définir une catégorie (`id`, `nomcategorie`, `imagecategorie`), qui sont les seuls éléments nécessaires à la prise de décision dans le domaine métier.

## 3. Éviter la dépendance à l'infrastructure

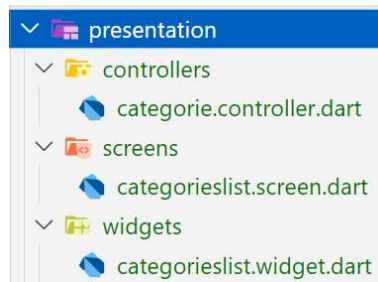
Si les use cases utilisaient des modèles (qui sont souvent liés à la base de données ou aux formats d'API), cela créerait une dépendance directe entre la logique métier et l'infrastructure. Si un jour la source de données change (par exemple, on passe d'une base de données NoSQL à une base de données NoSQL relationnelle, ou d'une API REST à GraphQL), il faudrait adapter toute la logique métier.

Les entités abstraient cette complexité en maintenant la logique métier décorrélée de ces détails techniques. La couche data est responsable de la conversion entre les modèles et les entités.

# IX. Couche Presentation

La couche Presentation aura comme contenu





La couche Presentation interagit directement avec l'utilisateur final. Elle est responsable de l'affichage des informations, de la capture des interactions de l'utilisateur, et de l'envoi des actions vers les couches sous-jacentes (comme la couche domain). Cette couche contient l'interface utilisateur (UI) et gère la logique de présentation, souvent en utilisant des contrôleurs ou des gestionnaires d'état pour coordonner les interactions entre l'interface utilisateur et la logique métier.

### 3. Créer Controller

Créer `/lib/presentation/controllers/categorie.controller.dart`

lib > presentation > controllers >  categorie.controller.dart

```
import 'package:get/get.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import 'package:myflutterapplication/domain/usecases/categorie.usecase.dart';

class CategorieController extends GetxController {
  final CategorieUseCase _useCase;

  var categoriesList = <CategorieEntity>[].obs;
  var isLoading = true.obs;

  CategorieController({required CategorieUseCase useCase})
    : _useCase = useCase;

  fetchAllCategories() {
    _useCase.fetchCategories().then((data) {
      isLoading.value = false;
      if (data != null) {
        final result = data
          .map(
            (element) => CategorieEntity(
              id: element?.id ?? "",
              nomcategorie: element?.nomcategorie ?? "",
              imagecategorie: element?.imagecategorie ?? "",
            ),
          )
          .toList();
      }
    });
  }
}
```

```
        categoriesList.value = result;
    }
  }).catchError((error) {});
}
}
```

## Explication

Le controller agit comme une interface entre la logique métier (gérée par le use case) et la vue (l'interface utilisateur). Son rôle est de recevoir les données du use case, les préparer et les exposer à la vue pour qu'elle puisse les afficher.

La méthode `fetchAllCategories()` appelle le use case (`_useCase.fetchCategories()`), récupère les données, puis les stocke dans `categoriesList` pour que la vue puisse les afficher.

La couche controller, dans ce cas, représentée par la classe `CategorieController` utilisant GetX, joue un rôle essentiel dans l'architecture d'une application Flutter.

Dans une architecture avec GetX, la gestion de l'état se fait de manière réactive. Le controller est responsable de cette gestion de l'état et de la mise à jour de l'interface utilisateur en fonction des changements d'état.

`categoriesList` et `isLoading` sont des observables (via **.obs**), ce qui signifie que la vue qui les consomme sera automatiquement mise à jour lorsque ces valeurs changent. Grâce à cette gestion d'état, la vue peut réagir dynamiquement aux changements de données sans nécessiter de gestion complexe des événements.

`isLoading` permet de gérer l'état de chargement et de mettre à jour la vue pour afficher, par exemple, un indicateur de chargement pendant que les catégories sont en train d'être récupérées.

Le controller gère les appels asynchrones aux use cases (qui eux-mêmes appellent des repositories et des services). Cela permet de garder la vue simple et découplée de la logique complexe de récupération des données.

`fetchAllCategories()` fait appel à la méthode `fetchCategories()` du use case de manière asynchrone avec `.then()` et gère les erreurs potentielles via `.catchError()`.

Le controller expose des données propres à la vue (comme `categoriesList`) pour qu'elle puisse les utiliser sans avoir à s'occuper des détails des appels réseau ou des erreurs.

Le controller est souvent responsable de la transformation des données provenant du use case pour les adapter au format attendu par la vue. Cela permet de garder la logique métier et la logique de présentation séparées.

Dans cet exemple, après avoir reçu les données sous forme d'Entity, le controller les transforme en une liste d'entités utilisables par la vue (`CategorieEntity`), en prenant soin de gérer les valeurs null pour éviter des erreurs d'affichage.

Par la suite la vue est représentée par un screen qui appelle à son tour un widget pour faire un l'affichage des données.

## 4. Créer le widget

Créer /lib/presentation/widgets/categorieslist.widget.dart

lib > presentation > widgets >  categorieslist.widget.dart

```
import 'package:flutter/material.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';

class Categorieslistwidget extends StatelessWidget {
  final CategorieEntity categories;
  const Categorieslistwidget({super.key, required this.categories});

  @override
  Widget build(BuildContext context) {

    return SingleChildScrollView(
      child: Card(
        child: ListTile(
          leading: Image.network(
            categories.imagecategorie,
            width: 68,
            height: 68,
          ),
          title: Text(categories.nomcategorie),
        ),
      ),
    );
  }
}
```

## Explication

Pour scroller une seule colonne contenant plusieurs widgets, dont des ListTile, on l'a encapsulé dans un SingleChildScrollView pour permettre le défilement manuel.

ListTile est un widget Flutter très utilisé pour afficher du contenu sous forme de ligne simple et structurée dans une liste. C'est un composant de base qui permet d'organiser des informations sous forme de tuiles, souvent utilisées dans des listes.

Un ListTile est structuré de manière à présenter un titre, un sous-titre, une icône à gauche (leading) et/ou à droite (trailing).

## 5. Créer le screen

Créer /lib/presentation/screens/categorieslist.screen.dart

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import
'package:myflutterapplication/presentation/controllers/categorie.controller.da
rt';
import
'package:myflutterapplication/presentation/widgets/categorieslist.widget.dart'
;

class Categorieslist extends StatelessWidget {
  const Categorieslist({super.key});

  @override
  Widget build(BuildContext context) {
    var contoller = Get.find<CategorieController>();
    contoller.fetchAllCategories();
    return Scaffold(
      appBar: AppBar(
        title: const Text("Categories"),
      ),
      body: Obx(
        () => contoller.isLoading.value == true
          ? const Center(
              child: CircularProgressIndicator(),
            )
          : ListView.builder(
              itemCount: contoller.categoriesList.length,
              itemBuilder: (context, index) {
                final categories = contoller.categoriesList[index];
                return Categorieslistwidget(
                  categories: categories,
                );
              },
            ),
      ),
    );
  }
}
```

## Explication

**Obx** est un widget fourni par GetX pour rendre les widgets réactifs. Cela signifie que chaque fois qu'une variable observable (dans ce cas, `isLoading` ou `categoriesList`) change de valeur, le widget qui est enveloppé dans Obx sera automatiquement reconstruit pour refléter ces changements.

Dans ce code, le corps du Scaffold (la propriété body) est un widget Obx. Cela signifie que **chaque fois que la variable observable isLoading ou categoriesList dans le controller change, la partie du code enveloppée par Obx est reconstruite.**


La variable isLoading est définie dans le CategoryController comme une variable observable (.obs), ce qui signifie qu'elle peut être surveillée pour les changements de valeur.

Lors du premier chargement, isLoading.value est initialement true, ce qui entraîne l'affichage du cercle de chargement (CircularProgressIndicator) au centre de l'écran.

Une fois que les données des catégories sont chargées et que la variable isLoading passe à false, le widget est reconstruit automatiquement et affiche une ListView.builder qui affiche la liste des catégories récupérées.

## X. Injection des dépendances

Dans main.dart ajouter l'injection de dépendances :

```
lib >  main.dart

import 'package:flutter/material.dart';
import 'package:myflutterapplication/approuter.dart';

import 'package:get/instance_manager.dart';
import
'package:myflutterapplication/data/datasource/services/categorie.service.dart'
;
import
'package:myflutterapplication/data/repositories/categorie.repository.dart';
import 'package:myflutterapplication/domain/usecases/categorie.usecase.dart';
import
'package:myflutterapplication/presentation/controllers/categorie.controller.dart';

// Fonction principale qui lance l'application Flutter
void main() {
// Injection de dépendances

  Get.put(CategorieService());
  Get.put(CategorieRepository(catserv: Get.find()));
  Get.put(CategorieUseCase(respository: Get.find()));
  Get.put(CategorieController(useCase: Get.find()));

  // Lance l'application en exécutant MyApp
  runApp(const MyApp());
}

// Définition de la classe stateless MyApp
```

```

class MyApp extends StatelessWidget {
  // Constructeur constant avec une clé facultative
  const MyApp({super.key});

  // Ce widget est la racine de l'application
  @override
  Widget build(BuildContext context) {
    // Retourne un MaterialApp configuré
    return MaterialApp(
      // Titre de l'application
      title: 'Flutter Application',
      // Thème de l'application avec une palette de couleurs personnalisée
      theme: ThemeData(
        // Utilisation d'un ColorScheme basé sur une couleur de départ
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
        // Activation de Material Design 3
        useMaterial3: true,
      ),
      // Désactivation du bandeau "DEBUG"
      debugShowCheckedModeBanner: false,
      initialRoute: '/',
      routes: appRoutes(), // Utilisation des routes depuis le fichier
séparé
    );
  }
}

```

## Explication

Les lignes de code suivantes sont utilisées pour l'injection de dépendances avec le package GetX dans Flutter. L'injection de dépendances est une technique qui permet de gérer et d'injecter les dépendances (comme les services, les repositories, et les contrôleurs) dans les différentes parties de l'application de manière centralisée. Cela facilite la gestion des dépendances et améliore la modularité du code.

Chaque ligne utilise **Get.put()** pour enregistrer une instance d'une classe dans le conteneur de dépendances de GetX.

Lors de la création des instances suivantes, **Get.find()** est utilisé pour récupérer les instances précédemment enregistrées. Cela permet d'assurer que toutes les dépendances sont correctement injectées dans chaque composant.

Chaque composant (service, repository, use case, et controller) est créé en utilisant les dépendances dont il a besoin, qui sont fournies via le système d'injection de GetX.

## XI. Appel du screen dans la route

```
'/Categories': (context) => const Categorieslist(), // Route pour l'écran  
Categories
```

## Le code complet

```
// Importation des packages nécessaires  
import 'package:flutter/material.dart';  
import 'package:myflutterapplication/models/Product.class.dart';  
import  
'package:myflutterapplication/presentation/screens/categorieslist.screen.dart'  
;  
import 'package:myflutterapplication/screens/details.dart';  
import 'package:myflutterapplication/screens/documents.dart';  
import 'package:myflutterapplication/screens/exitscreen.dart';  
import 'package:myflutterapplication/screens/menu.dart';  
import 'package:myflutterapplication/screens/myproducts.dart';  
import 'package:myflutterapplication/screens/products.dart';  
import 'package:myflutterapplication/screens/subscribe.dart';  
import 'package:myflutterapplication/widgets/myappbar.dart';  
import 'package:myflutterapplication/widgets/mybottomnavbar.dart';  
import 'package:myflutterapplication/widgets/mydrawer.dart';  
  
// Définition d'une fonction qui retourne les routes  
Map<String, WidgetBuilder> appRoutes() {  
  
  return {  
    '/': (context) => const Scaffold(  
      appBar: Myappbar(),  
      body: Menu(),  
      drawer: MyDrawer(),  
      bottomNavigationBar: MyBottomNavigation(),  
    ),  
    '/Items': (context) => Scaffold(  
      appBar: AppBar(  
        title: const Text('My Products'),  
      ),  
      body: const Myproducts(),  
      drawer: const MyDrawer(),  
      bottomNavigationBar: const MyBottomNavigation(),  
    ),  
    '/Exit': (context) => const ExitScreen(), // Route associée à l'action de  
fermeture  
    '/Documents': (context) => const Documents(), // Route pour l'écran  
Documents  
    '/Products': (context) => const Products(), // Route pour l'écran Products
```

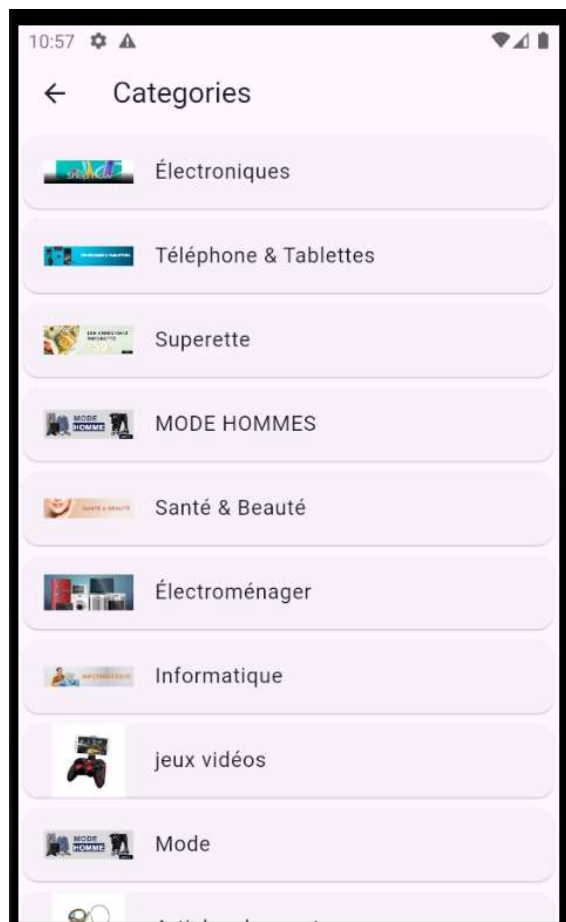
```

'/details': (context) {final product =
ModalRoute.of(context)!.settings.arguments as Product;
    return Details(myListElement: product);
}, // Route pour l'écran Details
'/Subscribe': (context) => const Subscribe(), // Route pour l'écran
Subscribe
'/Categories': (context) => const Categorieslist(), // Route pour l'écran
Categories
};
}

```

## Résultat

Pour avoir un résultat il ne faut pas oublier de **démarrer le serveur du back end**.



## XII. L'ajout

lib > data > datasource > services >  categorie.service.dart



```

import 'dart:convert';
import 'dart:io';

import 'package:dio/dio.dart';
import 'package:myflutterapplication/utils/constants.dart';

class CategorieService{
  late Dio dio;

  CategorieService(){
    BaseOptions options=BaseOptions(
      baseUrl: baseUrl,
      receiveDataWhenStatusError: true,
      connectTimeout: const Duration(seconds: 5),
      receiveTimeout: const Duration(seconds: 3),
    );
    dio=Dio(options);
  }
  Future<List<dynamic>> getCategories() async{
    try{
      Response response=await dio.get('/categories');
      print(response.data.toString());
      return response.data;
    }
    catch(e){
      print(e.toString());
      return [];
    }
  }

  Future<Map<String, dynamic>> postCategorie(String nom, dynamic image) async{

    var params = {
      "nomcategorie": nom,
      "imagecategorie": image,
    };

    Response response = await dio.post('/categories',
      options: Options(headers: {
        HttpHeaders.contentTypeHeader: "application/json",
      }),
      data: jsonEncode(params),
    );
    return response.data;
  }
}

```

lib > data > repositories >  categorie.repository.dart

```
import
'package:myflutterapplication/data/datasource/models/categories.model.dart';
import
'package:myflutterapplication/data/datasource/services/categorie.service.dart'
;

class CategorieRepository{
final CategorieService catserv ;

CategorieRepository({required this.catserv});

Future<List<Categorie>> getCategories() async{
final categories=await catserv.getCategories();
return categories.map((c) => Categorie.fromJson(c)).toList();
}

Future<Map> postCategorie(String nom, dynamic image) async {
    final categorie = await catserv.postCategorie(nom, image);
    return categorie;
}

}
```

lib > domain > usecases >  categorie.usecase.dart

```
import
'package:myflutterapplication/data/repositories/categorie.repository.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';

class CategorieUseCase {
    final CategorieRepository _respository;

    CategorieUseCase({required CategorieRepository respository})
        : _respository = respository;

    Future<List<CategorieEntity?>>> fetchCategories() async {
        final result = await _respository.getCategories();

        final data = result.map((element) {
            return CategorieEntity(
                id : element.id ?? "",
                nomcategorie: element.nomcategorie ?? "",
                imagecategorie: element.imagecategorie ?? "",
            );
        });
    }
}
```

```

    );
  }).toList();
  return data;
}

Future<CategorieEntity?> addCategorie(String nom, dynamic image) async {
  final result = await _respository.postCategorie(nom, image);

  if (result.isNotEmpty) {
    return CategorieEntity(
      id: result['id'] ?? "",
      nomcategorie: result['nomcategorie'] ?? "",
      imagecategorie: result['imagecategorie'] ?? "",
    );
  }
  return null;
}
}

```

## Explication :

addCategorie : Cette méthode appelle le postCategorie du repository et récupère le résultat.

Transformation des données : Le résultat du repository est ensuite mappé à un objet CategorieEntity qui est retourné.

Gestion d'erreur : Si le résultat est vide ou qu'il y a un problème, la méthode retourne null.

lib > presentation > controllers >  categorie.controller.dart

```

import 'package:get/get.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import 'package:myflutterapplication/domain/usecases/categorie.usecase.dart';

class CategorieController extends GetxController {
  final CategorieUseCase _useCase;

  var categoriesList = <CategorieEntity>[].obs;
  var isLoading = true.obs;
  var isPosting = false.obs;

  CategorieController({required CategorieUseCase useCase})
    : _useCase = useCase;

  fetchAllCategories() {
    _useCase.fetchCategories().then((data) {
      isLoading.value = false;
    });
  }
}

```

```

        if (data != null) {
            final result = data
                .map(
                    (element) => CategorieEntity(
                        id: element?.id ?? "",
                        nomcategorie: element?.nomcategorie ?? "",
                        imagecategorie: element?.imagecategorie ?? "",
                    ),
                )
                .toList();

            categoriesList.value = result;
        }
    }).catchError((error) {}));
}

// Post a new category
Future<void> postCategorie(String nom, dynamic image) async {
    try {
        isPosting.value = true;
        final newCategory = await _useCase.addCategorie(nom, image);
        if (newCategory != null) {
            categoriesList.add(newCategory);
        }
    } catch (error) {
        // Handle error (e.g., show a message)
        print("Error posting category: $error");
    } finally {
        isPosting.value = false;
    }
}
}

```

## Explication

Propriété isPosting : Cette propriété observable vous permet de gérer l'état de chargement pendant l'ajout d'une nouvelle catégorie.

Méthode postCategorie : Cette méthode appelle la méthode addCategorie du use case pour créer une nouvelle catégorie.

Si la catégorie est créée avec succès, elle est ajoutée à la liste des catégories (categoriesList.add(newCategory)).

En cas d'échec, un message d'erreur pour informer l'utilisateur.

Ensuite, créer le fichier qui contiendra le formulaire :

lib > presentation > widgets >  addcategorieform.widget.dart

On va utiliser :

Get.put() : Crée et injecte une instance dans le gestionnaire de GetX.

Get.find() : Récupère une instance existante de la hiérarchie de dépendances de GetX.

Cela permet une gestion automatique et centralisée des dépendances, ce qui simplifie le code et facilite la réutilisation et les tests.

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import
'package:myflutterapplication/presentation/controllers/categorie.controller.da
rt';

class Addcategorieform extends StatefulWidget {
  const Addcategorieform({super.key});

  @override
  State<Addcategorieform> createState() => _AddcategorieformState();
}

class _AddcategorieformState extends State<Addcategorieform> {

  // Initialisation du contrôleur
  final CategorieController _controller = Get.put(CategorieController(
    useCase: Get.find(),
  ));

  final _formKey = GlobalKey<FormState>();

  late TextEditingController _nomcategorieController;
  late TextEditingController _imagecategorieController;

  @override
  void initState() {
    super.initState();
    _nomcategorieController = TextEditingController();
    _imagecategorieController = TextEditingController();
  }

  @override
  Widget build(BuildContext context) {
```

```

return Form(
  key: _formKey,
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: <Widget>[
      TextFormField(
        controller: _nomcategorieController,
        decoration: const InputDecoration(
          hintText: "Category name", labelText: "Name"),
        validator: (value) {
          if (value!.isEmpty) {
            return 'Please enter the name';
          }
          return null;
        },
      ),
      TextFormField(
        controller: _imagecategorieController,
        decoration: const InputDecoration(
          hintText: "Category image", labelText: "Image"),
      ),
      Center(
        child: ElevatedButton(
          onPressed: () {
            // Retourne true si le formulaire est valide, sinon false
            if (_formKey.currentState!.validate()) {
              print(_nomcategorieController.text);
              print(_imagecategorieController.text);

              // Appelle la méthode du contrôleur GetX avec les valeurs
des champs
              _controller.postCategorie(
                _nomcategorieController.text,
                _imagecategorieController.text,
              );

              // Affiche le Snackbar si le formulaire est valide
              ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(content: Text('Processing Data')),
              );
            }
          },
          style: ButtonStyle(
            backgroundColor: WidgetStateProperty.all<Color>(
              Colors.blueAccent,
            ),
          ),
          child: const Text("submit"),
        ),
      ),
    ],
  ),

```

```

    ),
  ],
),
);
}
}

```

## Explication :

1. **CategorieController \_controller :**
  - Vous déclarez une variable `_controller` de type `CategorieController` qui va contenir l'instance de votre contrôleur `GetX`.
2. **Get.put() :**
  - `Get.put()` est une méthode fournie par `GetX` pour injecter et initialiser une instance d'une classe dans le gestionnaire d'état de `GetX`.
  - Cela permet à `GetX` de gérer automatiquement l'instance (son cycle de vie, etc.), et de la rendre disponible pour d'autres parties de votre application.
3. **CategorieController(useCase: Get.find()) :**
  - Vous créez une instance de `CategorieController` en appelant son constructeur.
  - **useCase :** Le constructeur du contrôleur attend un argument nommé `useCase`, qui est une instance de `CategorieUseCase`.
  - **Get.find() :** `Get.find()` permet de récupérer une instance préalablement injectée d'une classe (dans ce cas, `CategorieUseCase`). Il cherche cette instance dans la hiérarchie de dépendances gérée par `GetX`. Si l'instance n'a pas été encore injectée, une erreur sera levée.

Ensuite, créer le fichier `addcategorie.screen.dart` qui fera appel au widget.

lib > presentation > screens >  addcategorie.screen.dart

```

import 'package:flutter/material.dart';
import
'package:myflutterapplication/presentation/widgets/addcategorieform.widget.dar
t';

class Addcategorie extends StatelessWidget {
  const Addcategorie({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        elevation: 15,
        backgroundColor: Colors.blueAccent,
        title: const Text("Add Categories"),
        leading: IconButton(

```

```

        onPressed: () {},
        icon: const Icon(Icons.category_sharp),
      ),
    ),
    body: const Addcategorieform(),
  );
}
}

```

lib >  approuter.dart

Ajouter la route :

```

'/addcategories': (context) => const Addcategorie(), // Route pour l'écran
Addcategorie

```

lib > presentation > screens >  categorieslist.screen.dart

Ajouter le bouton qui appelle le screen addcategorie.screen

```

import 'package:flutter/material.dart';
import 'package:get/get.dart';
import
'package:myflutterapplication/presentation/controllers/categorie.controller.dart';
import
'package:myflutterapplication/presentation/widgets/categorieslist.widget.dart'
;

class Categorieslist extends StatelessWidget {
  const Categorieslist({super.key});

  @override
  Widget build(BuildContext context) {
    var controller = Get.find<CategorieController>();
    controller.fetchAllCategories();
    return Scaffold(
      appBar: AppBar(
        title: const Text("Categories"),
      ),
      body: Obx(
        () => controller.isLoading.value == true
          ? const Center(
              child: CircularProgressIndicator(),
            )
          : ListView.builder(
              itemCount: controller.categoriesList.length,

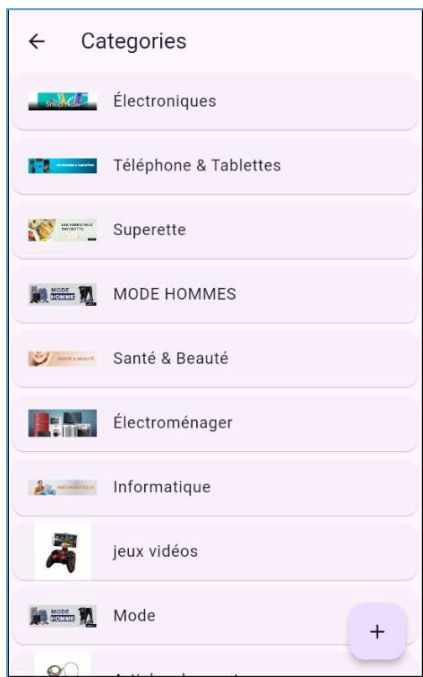
```



```

        itemBuilder: (context, index) {
          final categories = controller.categoriesList[index];
          return Categorieslistwidget(
            categories: categories,
          );
        },
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: () {
        Navigator.of(context).pushNamed('/addcategories');
      },
      child: const Icon(Icons.add),
    ),
  );
}
}

```



On va améliorer le code :

Si l'ajout d'une catégorie réussit, un message de succès apparaît, et l'utilisateur est redirigé vers la liste des catégories.

En cas d'erreur, un message d'erreur est affiché dans un `SnackBar` pour informer l'utilisateur.

lib > presentation > widgets >  addcategorieform.widget.dart

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import
'package:myflutterapplication/presentation/controllers/categorie.controller.dart';

class Addcategorieform extends StatefulWidget {
```

```

const Addcategorieform({super.key});

@override
State<Addcategorieform> createState() => _AddcategorieformState();
}

class _AddcategorieformState extends State<Addcategorieform> {

  // Initialisation du contrôleur
  final CategorieController _controller = Get.put(CategorieController(
    useCase: Get.find(),
  ));

  final _formKey = GlobalKey<FormState>();

  late TextEditingController _nomcategorieController;
  late TextEditingController _imagecategorieController;

  @override
  void initState() {
    super.initState();
    _nomcategorieController = TextEditingController();
    _imagecategorieController = TextEditingController();
  }

  @override
  Widget build(BuildContext context) {
    return Form(
      key: _formKey,
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          TextFormField(
            controller: _nomcategorieController,
            decoration: const InputDecoration(
              hintText: "Category name", labelText: "Name"),
            validator: (value) {
              if (value!.isEmpty) {
                return 'Please enter the name';
              }
              return null;
            },
          ),
          TextFormField(
            controller: _imagecategorieController,
            decoration: const InputDecoration(
              hintText: "Category image", labelText: "Image"),
          ),
          Center(

```

```

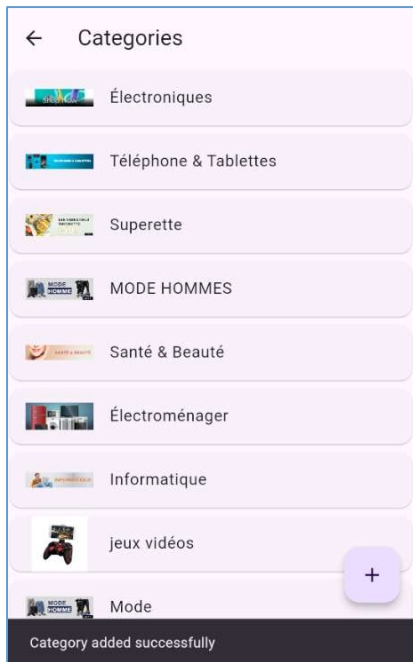
        child: ElevatedButton(
          onPressed: () {
            // Retourne true si le formulaire est valide, sinon false
            if (_formKey.currentState!.validate()) {
              print(_nomcategorieController.text);
              print(_imagecategorieController.text);

              try {
                // Appelle la méthode du controller GetX avec les valeurs
des champs
                _controller.postCategorie(
                  _nomcategorieController.text,
                  _imagecategorieController.text,
                );

                // Si l'ajout est réussi, afficher message
                ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(content: Text('Category added
successfully'))),
                );

                // Redirection vers la page des catégories
                Navigator.of(context).pushNamed('/Categories');
              } catch (error) {
                // Si une erreur survient, afficher un message d'erreur
                ScaffoldMessenger.of(context).showSnackBar(
                  SnackBar(content: Text('Error adding category:
$error'))),
                );
              }
            }
          },
          style: ButtonStyle(
            backgroundColor: WidgetStateProperty.all<Color>(
              Colors.blueAccent,
            ),
          ),
          child: const Text("submit"),
        ),
      ],
    ),
  );
}

```



## XIII. Upload image

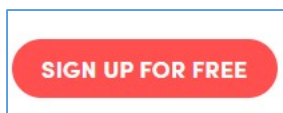
Pour le champ image on va choisir la valeur à partir de la galerie ou bien de la caméra. Cette valeur sera uploadée dans le cloud cloudinary.

Tout d'abord, il faut créer un compte dans le site Cloudinary.

### 1. Inscription

Site :

<https://cloudinary.com/>



cloudinary.com/users/register/free

Cloudinary PRODUCTS SOLUTIONS RESOURCES COMPANY PRICING Support Documentation Login SIGN UP FOR FREE

## Sign Up to Cloudinary

Your Name:  
S.H.

E-mail:  
sandrahmamtilili@gmail.com


Password: (At least 8 characters, must contain at least one lower-case letter, one upper-case letter, one digit and a special character)  
\*\*\*\*\*

Country:  
Tunisia

Company or Site Name (Optional)  
ISET

Select a product:  
Programmable Media for image and video API

Assigned Cloud Name: **ISET** [Edit](#)

 [Terms of Service](#) and [Privacy Policy](#)

**CREATE ACCOUNT**

**Programmable Media**  
API-based video and image management with dynamic transformations—for resizing, cropping, overlaying—automated optimization and accelerated delivery of content via CDN.

**Media Optimizer**  
Images and videos automatically delivered in the format and quality suited for each end-user device, browser and connection speed. All with no code or manual work.


**Digital Asset Management**  
Asset management to meet the unique needs of today, focusing on flexibility, intelligent automation and scale.

cloudinary.com/users/email\_verification\_required

Cloudinary PRODUCTS SOLUTIONS RESOURCES COMPANY PRICING Support Documentation Login SIGN UP FOR FREE

## Email Verification

Thank you for signing up for a Cloudinary account.



### Verify your email address

Verification email has been sent to:  
sandrahmamtilili@gmail.com

Click on the link in the email to activate your account.

Didn't receive the email? [Resend email](#)

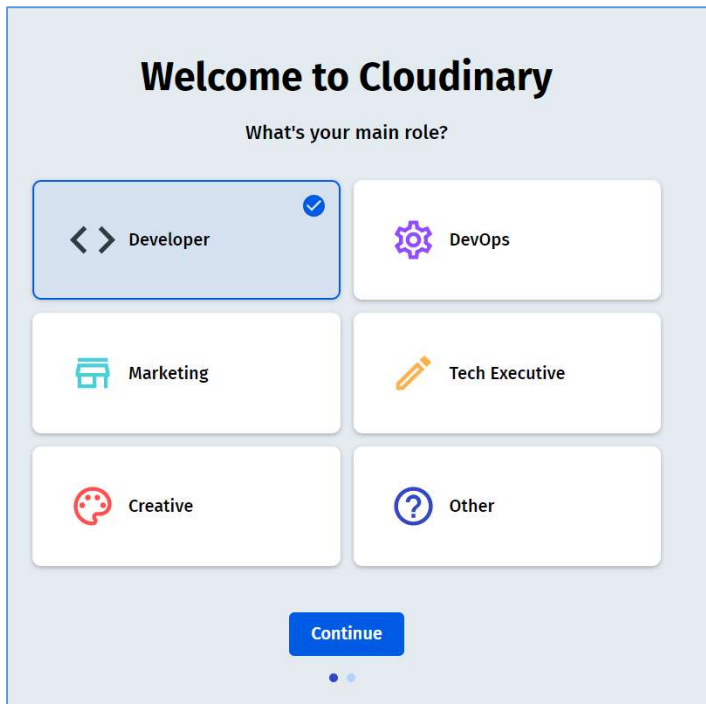
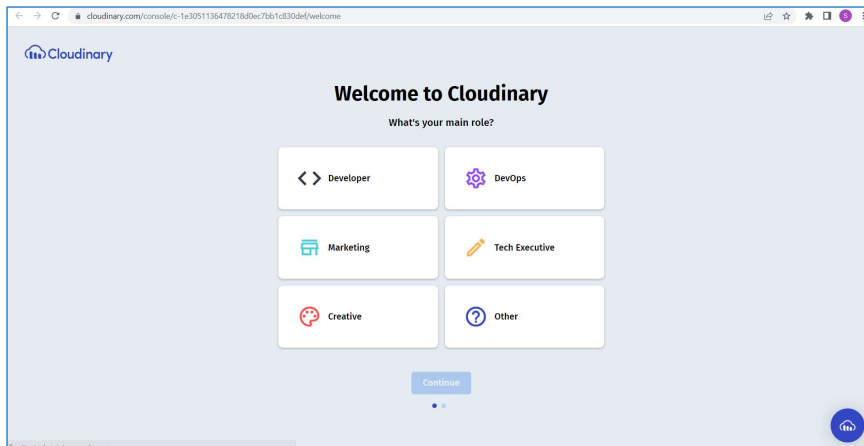
S.H.,

Congratulations! You're almost set to start using Cloudinary. Just click the button below to validate your email address.

**Sign in to Validate your Email Address**


If the button does not work for any reason, you can also paste the following into your browser:  
[https://cloudinary.com/users/verify\\_email?token=0ol7bYCjG9MPWwYv8o5k\\_BdFLBOuHdmt\\_vjzeczply&utm\\_source=welcome%20email&utm\\_medium=email&utm\\_campaign=welcome%20verification%20email&utm\\_content=dam](https://cloudinary.com/users/verify_email?token=0ol7bYCjG9MPWwYv8o5k_BdFLBOuHdmt_vjzeczply&utm_source=welcome%20email&utm_medium=email&utm_campaign=welcome%20verification%20email&utm_content=dam)

Note: You must perform this validation within the next 24 hours to keep your new account enabled.




## Welcome to Cloudinary


What's your first Cloudinary project likely to be?




Build a new website,  
app or product




Optimize my site for  
fast delivery




Adapt my media to  
any device and  
design



Enhance my site  
visual appearance



I still don't know



Other

Done



cloudinary.com/console/c-1e3051136478218ddec7bb1c830def

DashboardMedia LibraryTransformationsReportsAdd-ons

Dashboard

## Welcome to your Cloudinary Dashboard


Find all the information you need about your plan, usage, and how to get the most out of, or even impact, Cloudinary features.

Hide banner

Transformations


Image	Width	Height
100x100	100	100
100x100	100	100
100x100	100	100

Account Details




Cloud Name

iset-sfax



API Key

7346274149555



API Secret

\*\*\*\*\*

<>

API Environment variable

CLOUDINARY\_URL=cloudinary://\*\*\*\*\*@iset-sfax

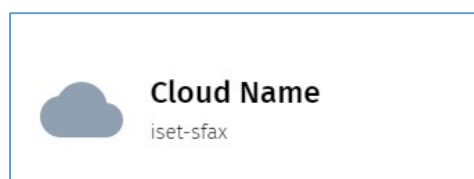
Plan Current Usage

Credit Usage For Last 30 Days

0 of 25

Free

Upgrade Plan



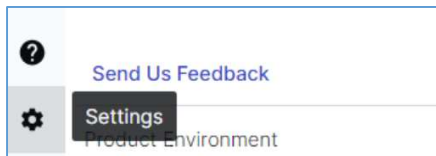
Enregistrer l'information de ce Cloud Name.



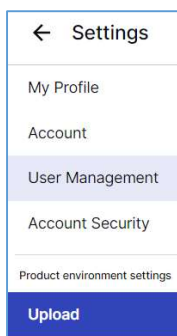
Nous avons 2 données précises à noter qu'on va en avoir besoin par la suite dans le code.

D'abord la valeur du cloudName.

Par la suite, accédez à Settings (en bas à gauche)



Puis Upload (Menu à gauche)



Puis en bas de la page, on retrouvera, « Upload presets », il faut aussi copier le nom. Exemple :



Maintenant installer **cloudinary\_public**

Site : [https://pub.dev/packages/cloudinary\\_public](https://pub.dev/packages/cloudinary_public)

CTRL SHIFT P




Ensuite, on va installer image\_picker

Site : [https://pub.dev/packages/image\\_picker](https://pub.dev/packages/image_picker)

CTRL SHIFT P

>|

Pubspec Assist: Add/update dependencies

recently used 

image\_picker

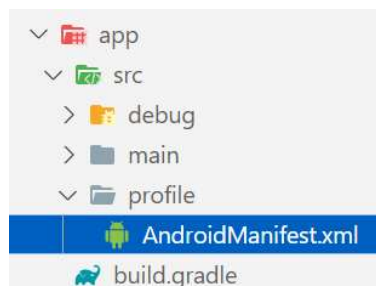
Enter package names, separated by commas. (Press 'Enter' to confirm or 'Escape' to cancel)

On verra que les dépendances sont ajoutées.


 pubspec.yaml

```
30 dependencies:
31   cloudinary_public: ^0.23.1
32   cupertino_icons: ^1.0.6
33   dio: ^5.7.0
34   flutter:
35     sdk: flutter
36   get: ^4.6.6
37   image_picker: ^1.1.2
38   sliver_tools: ^0.2.12
```

Puis ajouter les permissions dans :



```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.FLASHLIGHT" />
```

```
android > app > src > profile >  AndroidManifest.xml
You, 1 second ago | 1 author (You)
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android">
2   <!-- The INTERNET permission is required for development. Specifically,
3   the Flutter tool needs it to communicate with the running application
4   to allow setting breakpoints, to provide hot reload, etc.
5   -->
6   <uses-permission android:name="android.permission.INTERNET"/>
7   <uses-permission android:name="android.permission.CAMERA" />
8   <uses-permission android:name="android.permission.FLASHLIGHT" />
9   You, 2 seconds ago • Uncommitted changes
10 </manifest>
11
```

Modifier le code dans :

lib > presentation > widgets >  addcategorieform.widget.dart

En supprimant `TextFormField(...` relatif à image et le remplacer par le nouveau code. Nous n'avons plus besoin de `_imagecategorieController`

```
final clouinary =  
    CloudinaryPublic('██████████', '██████████', cache: false);
```

La première valeur est le cloud name et la deuxième valeur est upload presets

`_pickcamera` : Capture une image à l'aide de la caméra et télécharge l'image sur Cloudinary. Une fois l'image téléchargée, l'URL de l'image est stockée dans `imagecloud`.

`_pickgallery` : Sélectionne une image depuis la galerie, télécharge l'image sur Cloudinary, et stocke l'URL de l'image dans `imagecloud`.

Construction de l'interface utilisateur :

`ElevatedButton` : Boutons pour sélectionner une image depuis la galerie ou capturer une image avec la caméra.

`Image.network` : Affiche l'image téléchargée depuis Cloudinary.

```
import 'package:flutter/material.dart';  
import 'package:get/get.dart';  
import  
'package:myflutterapplication/presentation/controllers/categorie.controller.dart';  
import 'package:cloudinary_public/cloudinary_public.dart';  
import 'package:image_picker/image_picker.dart';  
import 'dart:io';  
  
class Addcategorieform extends StatefulWidget {  
    const Addcategorieform({super.key});  
  
    @override  
    State<Addcategorieform> createState() => _AddcategorieformState();  
}  
  
class _AddcategorieformState extends State<Addcategorieform> {  
  
    // Initialisation du contrôleur  
    final CategorieController _controller = Get.put(CategorieController(  
        useCase: Get.find(),  
    ));  
  
    final _formKey = GlobalKey<FormState>();  
  
    late TextEditingController _nomcategorieController;
```

```
@override
void initState() {
  super.initState();
  _nomcategorieController = TextEditingController();
}
```

```
var path;
var imagecloud = "";

final cloundinary =
  CloudinaryPublic(' ', ' ', cache: false);
```

```
ImagePicker picker = ImagePicker();
```

```
void _pickcamera() async {
  XFile? image = await picker.pickImage(source: ImageSource.camera);
  path = File(image!.path);

  try {
    CloudinaryResponse response = await cloundinary.uploadFile(
      CloudinaryFile.fromFile(image.path,
        resourceType: CloudinaryResourceType.Image),
    );

    print(response.secureUrl);
    setState(() {
      imagecloud = response.secureUrl;
    });
  } on CloudinaryException catch (e) {
    print(e.message);
    print(e.request);
  }
}
```

```
void _pickgallery() async {
  XFile? image = await picker.pickImage(source: ImageSource.gallery);
  path = File(image!.path);

  try {
    CloudinaryResponse response = await cloundinary.uploadFile(
      CloudinaryFile.fromFile(image.path,
        resourceType: CloudinaryResourceType.Image),
    );

    print(response.secureUrl);
    setState(() {
      imagecloud = response.secureUrl;
    });
  } on CloudinaryException catch (e) {
```

```

        print(e.message);
        print(e.request);
    }
}

```

```

@override
Widget build(BuildContext context) {
  return Form(
    key: _formKey,
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget>[
        TextFormField(
          controller: _nomcategorieController,
          decoration: const InputDecoration(
            hintText: "Category name", labelText: "Name"),
          validator: (value) {
            if (value!.isEmpty) {
              return 'Please enter the name';
            }
            return null;
          },
        ),

```

```

        Container(
          height: 40.0,
        ),
        Column(
          children: [
            ElevatedButton.icon(
              icon:
                const Icon(Icons.photo_album_rounded, color:
Colors.pink),
              style: ElevatedButton.styleFrom(
                backgroundColor: Colors.greenAccent,
              ),
              onPressed: () {
                _pickgallery();
              },
              label: const Text(
                "PICK FROM GALLERY",
                style: TextStyle(
                  color: Colors.blueGrey, fontWeight: FontWeight.bold),
              ),
            ),
            Container(
              height: 40.0,
            ),
            ElevatedButton.icon(
              icon:

```

```

        const Icon(Icons.camera_alt_sharp, color:
Colors.purple),
        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.lightGreenAccent,
        ),
        onPressed: () {
          _pickcamera();
        },
        label: const Text(
          "PICK FROM CAMERA",
          style: TextStyle(
            color: Colors.blueGrey, fontWeight: FontWeight.bold),
        ),
      ),
    Container(
      height: 40.0,
    ),
    SizedBox(
      height: 250,
      width: 250,
      child: imagecloud != ""
        ? Image.network(imagecloud)
        : Container(
            decoration: BoxDecoration(color: Colors.red[200]),
            child: Icon(
              Icons.camera_alt,
              color: Colors.grey[800],
            ),
          ),
    ),
    Container(
      height: 20.0,
    ),

```

```

Center(
  child: ElevatedButton(
    onPressed: () {
      // Retourne true si le formulaire est valide, sinon false
      if (_formKey.currentState!.validate()) {
        print(_nomcategorieController.text);
        print(imagecloud);

        try {
          // Appelle la méthode du controller GetX avec les valeurs
des champs
          _controller.postCategorie(
            _nomcategorieController.text,
            imagecloud,
          );

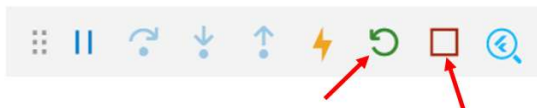
```


```

        // Si l'ajout est réussi, afficher message
        ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(content: Text('Category added
successfully'))),
        );


        // Redirection vers la page des catégories
        Navigator.of(context).pushNamed('/Categories');
    } catch (error) {
        // Si une erreur survient, afficher un message d'erreur
        ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(content: Text('Error adding category:
$error'))),
        );
    }
},
style: ButtonStyle(
    backgroundColor: WidgetStateProperty.all<Color>(
        Colors.blueAccent,
    ),
),
child: const Text("submit"),
),
),
],
),
);
}
}


```




 Add Categories

Name

 PICK FROM GALLERY

 PICK FROM CAMERA


submit





# Add Categories

Name

Intérieurs

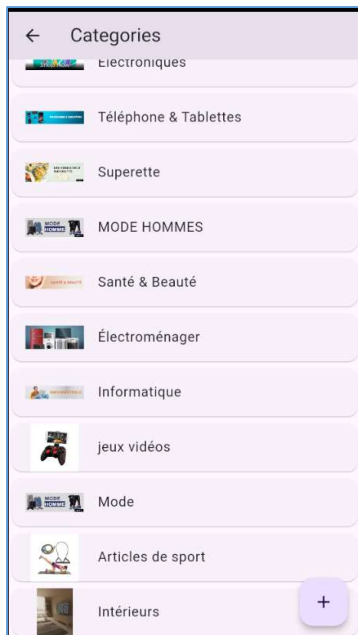
 PICK FROM GALLERY

 PICK FROM CAMERA



submit





## XIV. La suppression

On va ajouter le traitement relatif à la suppression des catégories.

lib > data > datasource > services >  categorie.service.dart :

```
import 'dart:convert';
import 'dart:io';

import 'package:dio/dio.dart';
import 'package:myflutterapplication/utils/constants.dart';

class CatégorieService {
  late Dio dio;

  CatégorieService() {
    BaseOptions options = BaseOptions(
      baseUrl: baseUrl,
      receiveDataWhenStatusError: true,
      connectTimeout: const Duration(seconds: 5),
      receiveTimeout: const Duration(seconds: 5),
    );
    dio = Dio(options);
  }

  //Affichage
  Future<List<dynamic>> getCategories() async {
```

```

    try {
        Response response = await dio.get('/categories');
        print(response.data.toString());
        return response.data;
    } catch (e) {
        print(e.toString());
        return [];
    }
}

//Ajout
Future<Map<String, dynamic>> postCategorie(String nom, dynamic image) async
{
    var params = {
        "nomcategorie": nom,
        "imagecategorie": image,
    };

    Response response = await dio.post(
        '/categories',
        options: Options(headers: {
            HttpHeaders.contentTypeHeader: "application/json",
        }),
        data: jsonEncode(params),
    );
    return response.data;
}

//suppression
Future<String> deleteCategorie(String id) async {
    try {
        final response = await dio.delete('/categories/$id');

        if (response.statusCode != 200) {
            throw Exception('Failed to delete data');
        }

        // Supposons que la réponse contient un Map avec un champ "message"
        //le backend retourne {message: categorie deleted successfully.}
        if (response.data is Map<String, dynamic>) {
            return response.data['message'] ?? 'Unknown error occurred';
        }

        // Gestion des cas où response.data n'est pas un Map
        return response.data.toString();
    } catch (e) {
        print('Erreur lors de la suppression: $e');
        rethrow;
    }
}

```

```
}  
}
```

lib > data > repositories >  categorie.repository.dart

```
import  
'package:myflutterapplication/data/datasource/models/categories.model.dart';  
import  
'package:myflutterapplication/data/datasource/services/categorie.service.dart'  
;  
  
class CategorieRepository{  
  final CategorieService catserv ;  
  
  CategorieRepository({required this.catserv});  
  
  //Affichage  
  Future<List<Categorie>> getCategories() async{  
    final categories=await catserv.getCategories();  
    return categories.map((c) => Categorie.fromJson(c)).toList();  
  }  
  
  //Ajout  
  Future<Map> postCategorie(String nom, dynamic image) async {  
    final categorie = await catserv.postCategorie(nom, image);  
    return categorie;  
  }  
  
  //Suppression  
  Future<String> deleteCategorie(String id) async {  
    try {  
      final response = await catserv.deleteCategorie(id);  
      return response;  
    } catch (e) {  
      rethrow;  
    }  
  }  
}
```

lib > domain > usecases >  categorie.usecase.dart

```
import  
'package:myflutterapplication/data/repositories/categorie.repository.dart';  
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
```

```

class CategorieUseCase {
    final CategorieRepository _respository;

    CategorieUseCase({required CategorieRepository respository})
        : _respository = respository;

    Future<List<CategorieEntity?>> fetchCategories() async {
        final result = await _respository.getCategories();

//Affichage
        final data = result.map((element) {
            return CategorieEntity(
                id : element.id ?? "",
                nomcategorie: element.nomcategorie ?? "",
                imagecategorie: element.imagecategorie ?? "",
            );
        }).toList();
        return data;
    }

//Ajout
    Future<CategorieEntity?> addCategorie(String nom, dynamic image) async {
        final result = await _respository.postCategorie(nom, image);

        if (result.isNotEmpty) {
            return CategorieEntity(
                id: result['id'] ?? "",
                nomcategorie: result['nomcategorie'] ?? "",
                imagecategorie: result['imagecategorie'] ?? "",
            );
        }
        return null;
    }

//Suppression
    Future<void> deleteCategorie(String id) async {
        try {
            await _respository.deleteCategorie(id);
        } catch (e) {
            // Gestion des erreurs, affichage de message d'erreur
            print("Erreur lors de la suppression de la catégorie: $e");
        }
    }
}

```

**Remarque :**

Ce cas de usecase est simple. Cette méthode prend en paramètre l'ID de la catégorie à supprimer. Elle appelle la méthode `deleteCategorie` du repository, pour effectuer la suppression. Mais on peut ajouter des contrôles supplémentaires, pour renforcer la logique métier de suppression de catégories pour s'assurer que les catégories ne sont pas supprimées par erreur ou sans vérifications appropriées. Voici un récapitulatif des contrôles possibles :

- Vérification de l'existence de la catégorie
- Vérification des droits d'utilisateur (autorisation)
- Vérification des dépendances (scategories, ou autres entités)
- Vérification des transactions passées
- Soft delete (désactivation plutôt que suppression)
- Enregistrement des actions de suppression (historisation)

---

lib > presentation > controllers >  categorie.controller.dart

Pour implémenter la fonctionnalité de suppression d'une catégorie dans le `CategorieController`, on va ajouter une méthode qui utilise le `CategorieUseCase` pour supprimer une catégorie en vérifiant d'abord s'il n'y a pas de dépendances.

```
import 'package:get/get.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import 'package:myflutterapplication/domain/usecases/categorie.usecase.dart';

class CategorieController extends GetxController {
  final CategorieUseCase _useCase;

  var categoriesList = <CategorieEntity>[].obs;
  var isLoading = true.obs;
  var isPosting = false.obs;

  CategorieController({required CategorieUseCase useCase})
    : _useCase = useCase;

  //Affichage
  fetchAllCategories() {
    _useCase.fetchCategories().then((data) {
      isLoading.value = false;
      if (data != null) {
        final result = data
          .map(
            (element) => CategorieEntity(
              id: element?.id ?? "",
              nomcategorie: element?.nomcategorie ?? "",
              imagecategorie: element?.imagecategorie ?? "",
            ),
          )
          .toList();

        categoriesList.value = result;
      }
    });
  }
}
```

```

    }
  }).catchError((error) {}));
}

// Ajout
Future<void> postCategorie(String nom, dynamic image) async {
  try {
    isPosting.value = true;
    final newCategory = await _useCase.addCategorie(nom, image);
    if (newCategory != null) {
      categoriesList.add(newCategory);
    }
  } catch (error) {
    // Handle error (e.g., show a message)
    print("Error posting category: $error");
  } finally {
    isPosting.value = false;
  }
}

// Suppression d'une catégorie
Future<void> deleteCategorie(String id) async {
  try {
    isPosting.value = true;

    // Appeler la méthode du usecase pour supprimer la catégorie
    await _useCase.deleteCategorie(id);

    // Mettre à jour la liste des catégories en supprimant celle qui a été
    supprimée
    categoriesList.removeWhere((categorie) => categorie.id == id);
  } catch (error) {
    // Gérer les erreurs, afficher un message d'erreur
    print("Erreur lors de la suppression de la catégorie: $error");
  } finally {
    isPosting.value = false;
  }
}
}

```

---

lib > presentation > widgets >  categorieslist.widget.dart

```

import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';

```

```

import
'package:myflutterapplication/presentation/controllers/categorie.controller.dart';

class Categorieslistwidget extends StatelessWidget {
  final CategorieEntity categories;
  Categorieslistwidget({super.key, required this.categories});
  //on a enlevé const pour Categorieslistwidget
  // car : Can't define the 'const' constructor because the field 'controller'
  is initialized with a non-constant value.

  final controller = Get.find<CategorieController>();

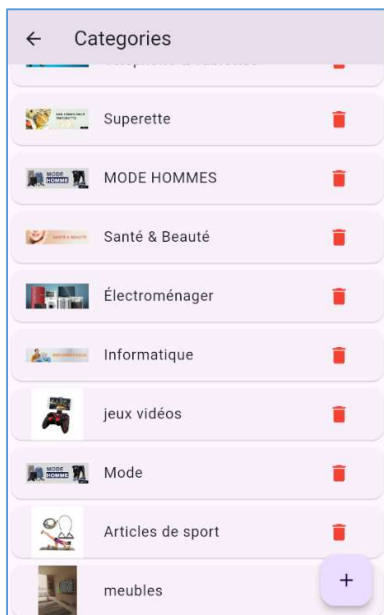
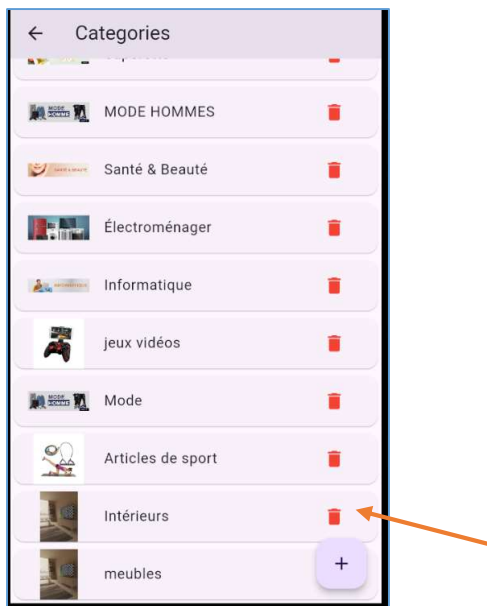
  @override
  Widget build(BuildContext context) {
    return SingleChildScrollView(
      child: Card(
        child: ListTile(
          leading: Image.network(
            categories.imagecategorie,
            width: 68,
            height: 68,
          ),
          title: Text(categories.nomcategorie),
          trailing: IconButton(
            icon: const Icon(Icons.delete, color: Colors.red,),
            onPressed: () => controller.deleteCategorie(categories.id),
          ),
        ),
      ),
    );
  }
}

```

### Remarque :

Dans ce cas particulier, il n'est pas nécessaire d'utiliser une classe StatefulWidget puisqu'on n'a pas besoin de gérer l'état interne du widget lui-même. Le StatelessWidget est suffisant, car on utilise le contrôleur GetX (CategorieController) pour gérer l'état de l'application (comme la suppression des catégories), et le contrôleur est déjà réactif grâce à GetX.

Si le widget n'a pas besoin de modifier son propre état interne et que tout est géré par le contrôleur via GetX, on peut rester avec un StatelessWidget. L'utilisation d'un StatefulWidget n'est nécessaire que si on a besoin de suivre ou de modifier l'état au sein du widget lui-même.



## XV. La modification

On va réaliser la modification des catégories. Le backend retourne le nouvel objet catégorie modifié.

lib > data > datasource > services >  categorie.service.dart

```
import 'dart:convert';
import 'dart:io';

import 'package:dio/dio.dart';
import 'package:myflutterapplication/utils/constants.dart';
```



```

class CategorieService {
    late Dio dio;

    CategorieService() {
        BaseOptions options = BaseOptions(
            baseUrl: baseUrl,
            receiveDataWhenStatusError: true,
            connectTimeout: const Duration(seconds: 5),
            receiveTimeout: const Duration(seconds: 5),
        );
        dio = Dio(options);
    }

    //Affichage
    Future<List<dynamic>> getCategories() async {
        try {
            Response response = await dio.get('/categories');
            print(response.data.toString());
            return response.data;
        } catch (e) {
            print(e.toString());
            return [];
        }
    }

    //Ajout
    Future<Map<String, dynamic>> postCategorie(String nom, dynamic image) async
    {
        var params = {
            "nomcategorie": nom,
            "imagecategorie": image,
        };

        Response response = await dio.post(
            '/categories',
            options: Options(headers: {
                HttpHeaders.contentTypeHeader: "application/json",
            }),
            data: jsonEncode(params),
        );
        return response.data;
    }

    //suppression
    Future<String> deleteCategorie(String id) async {
        try {
            Response response = await dio.delete('/categories/$id');

            if (response.statusCode != 200) {

```

```

        throw Exception('Failed to delete data');
    }

    // Supposons que la réponse contient un Map avec un champ "message"
    //le backend retourne {message: categorie deleted successfully.}
    if (response.data is Map<String, dynamic>) {
        return response.data['message'] ?? 'Unknown error occurred';
    }

    // Gestion des cas où response.data n'est pas un Map
    return response.data;
} catch (e) {
    print('Erreur lors de la suppression: $e');
    rethrow;
}
}

//update
Future<Map<String, dynamic>> updateCategorie(String id, String nom, dynamic
image) async {
    var params = {
        "nomcategorie": nom,
        "imagecategorie": image,
    };

    Response response = await dio.put(
        '/categories/$id',
        options: Options(headers: {
            HttpHeaders.contentTypeHeader: "application/json",
        }),
        data: jsonEncode(params),
    );
    return response.data;
}

}

```

lib > data > repositories >  categorie.repository.dart

```

import
'package:myflutterapplication/data/datasource/models/categories.model.dart';
import
'package:myflutterapplication/data/datasource/services/categorie.service.dart'
;

class CategorieRepository{
    final CategorieService catserv ;
}

```

```

CategorieRepository({required this.catserv});

//Affichage
Future<List<Categorie>> getCategories() async{
final categories=await catserv.getCategories();
return categories.map((c) => Categorie.fromJson(c)).toList();
}

//Ajout
Future<Map> postCategorie(String nom, dynamic image) async {
    final categorie = await catserv.postCategorie(nom, image);
    return categorie;
}

//Suppression
Future<String> deleteCategorie(String id) async {
    try {
        final response = await catserv.deleteCategorie(id);
        print("Response $response");
        return response;
    } catch (e) {
        rethrow;
    }
}

//modification
Future<Map> updateCategorie(String id,String nom, dynamic image) async {
    final categorie = await catserv.updateCategorie(id,nom, image);
    return categorie;
}
}

```

lib > domain > usecases >  categorie.usecase.dart

```

import
'package:myflutterapplication/data/repositories/categorie.repository.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';

class CategorieUseCase {
    final CategorieRepository _respository;

    CategorieUseCase({required CategorieRepository repository})
        : _respository = repository;
}

```

```

Future<List<CategorieEntity?>>> fetchCategories() async {
    final result = await _respository.getCategories();

//Affichage
    final data = result.map((element) {
        return CategorieEntity(
            id : element.id ?? "",
            nomcategorie: element.nomcategorie ?? "",
            imagecategorie: element.imagecategorie ?? "",
        );
    }).toList();
    return data;
}

//Ajout
Future<CategorieEntity?> addCategorie(String nom, dynamic image) async {
    final result = await _respository.postCategorie(nom, image);

    if (result.isNotEmpty) {
        return CategorieEntity(
            id: result['id'] ?? "",
            nomcategorie: result['nomcategorie'] ?? "",
            imagecategorie: result['imagecategorie'] ?? "",
        );
    }
    return null;
}

//Suppression
Future<void> deleteCategorie(String id) async {
    try {
        final res =await _respository.deleteCategorie(id);
        print ("UC : $res");
    } catch (e) {
        // Gestion des erreurs, affichage de message d'erreur
        print("Erreur lors de la suppression de la catégorie: $e");
    }
}

//modification
Future<CategorieEntity?> updateCategorie(String id,String nom, dynamic image)
async {
    final result = await _respository.updateCategorie(id,nom, image);

    if (result.isNotEmpty) {
        return CategorieEntity(
            id: result['id'] ?? "",
            nomcategorie: result['nomcategorie'] ?? "",
            imagecategorie: result['imagecategorie'] ?? "",

```

```

    );
  }
  return null;
}

}

```

lib > presentation > controllers >  categorie.controller.dart

```

import 'package:get/get.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import 'package:myflutterapplication/domain/usecases/categorie.usecase.dart';

class CategorieController extends GetxController {
  final CategorieUseCase _useCase;

  var categoriesList = <CategorieEntity>[].obs;
  var isLoading = true.obs;
  var isPosting = false.obs;

  CategorieController({required CategorieUseCase useCase})
    : _useCase = useCase;

  //Affichage
  fetchAllCategories() {
    _useCase.fetchCategories().then((data) {
      isLoading.value = false;
      if (data != null) {
        final result = data
          .map(
            (element) => CategorieEntity(
              id: element?.id ?? "",
              nomcategorie: element?.nomcategorie ?? "",
              imagecategorie: element?.imagecategorie ?? "",
            ),
          )
          .toList();

        categoriesList.value = result;
      }
    }).catchError((error) {});
  }

  // Ajout
  Future<void> postCategorie(String nom, dynamic image) async {
    try {

```

```

        isPosting.value = true;
        final newCategory = await _useCase.addCategorie(nom, image);
        if (newCategory != null) {
            categoriesList.add(newCategory);
        }
    } catch (error) {
        // Handle error (e.g., show a message)
        print("Error posting category: $error");
    } finally {
        isPosting.value = false;
    }
}

// Suppression d'une catégorie
Future<void> deleteCategorie(String id) async {
    try {
        isPosting.value = true;

        // Appeler la méthode du usecase pour supprimer la catégorie
        await _useCase.deleteCategorie(id);

        // Mettre à jour la liste des catégories en supprimant celle qui a été
        supprimée
        categoriesList.removeWhere((categorie) => categorie.id == id);
    } catch (error) {
        // Gérer les erreurs, afficher un message d'erreur
        print("Erreur lors de la suppression de la catégorie: $error");
    } finally {
        isPosting.value = false;
    }
}

// Mise à jour d'une catégorie
Future<void> updateCategorie(String id, String nom, dynamic image) async {
    try {
        isPosting.value = true;

        // Appeler la méthode du usecase pour mettre à jour la catégorie
        final updatedCategory = await _useCase.updateCategorie(id, nom, image);

        // Si la mise à jour est réussie, mettre à jour la liste des catégories
        if (updatedCategory != null) {
            final index = categoriesList.indexWhere((categorie) => categorie.id ==
id);
            if (index != -1) {
                categoriesList[index] = updatedCategory;
                categoriesList.refresh(); // Actualiser la liste observable
            }
        }
    }
}

```

```

    }
  } catch (error) {
    // Gérer les erreurs, afficher un message d'erreur
    print("Erreur lors de la mise à jour de la catégorie: $error");
  } finally {
    isPosting.value = false;
  }
}
}

```

Ensuite créer le fichier :

lib > presentation > widgets >  editcategorieform.widget.dart

```

import 'dart:io';

import 'package:cloudinary_public/cloudinary_public.dart';
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:image_picker/image_picker.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import
'package:myflutterapplication/presentation/controllers/categorie.controller.da
rt';

class EditCategorieForm extends StatefulWidget {
  final CategorieEntity
    categorie; // Paramètre pour passer la catégorie à modifier

  const EditCategorieForm({super.key, required this.categorie});

  @override
  State<EditCategorieForm> createState() => _EditCategorieFormState();
}

class _EditCategorieFormState extends State<EditCategorieForm> {
  final CategorieController _controller = Get.put(CategorieController(
    useCase: Get.find(),
  ));

  final _formKey = GlobalKey<FormState>();

  late TextEditingController _nomcategorieController;

  @override
  void initState() {
    super.initState();
  }
}

```

```

_nomcategorieController = TextEditingController(
    text: widget.categorie.nomcategorie); // Remplir avec le nom actuel
imagecloud = widget
    .categorie.imagecategorie; // Remplir avec l'URL de l'image actuelle
}

var path;
var imagecloud = "";

final cloundinary =
    CloundinaryPublic('iset-sfax', 'Ecommerce_cloundinary', cache: false);

ImagePicker picker = ImagePicker();

void _pickcamera() async {
    XFile? image = await picker.pickImage(source: ImageSource.camera);
    path = File(image!.path);

    try {
        CloundinaryResponse response = await cloundinary.uploadFile(
            CloundinaryFile.fromFile(image.path,
                resourceType: CloundinaryResourceType.Image),
        );

        print(response.secureUrl);
        setState(() {
            imagecloud = response.secureUrl;
        });
    } on CloundinaryException catch (e) {
        print(e.message);
        print(e.request);
    }
}

void _pickgallery() async {
    XFile? image = await picker.pickImage(source: ImageSource.gallery);
    path = File(image!.path);

    try {
        CloundinaryResponse response = await cloundinary.uploadFile(
            CloundinaryFile.fromFile(image.path,
                resourceType: CloundinaryResourceType.Image),
        );

        print(response.secureUrl);
        setState(() {
            imagecloud = response.secureUrl;
        });
    } on CloundinaryException catch (e) {

```



```

        print(e.message);
        print(e.request);
    }
}

@override
Widget build(BuildContext context) {
    return Form(
        key: _formKey,
        child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: <Widget>[
                TextFormField(
                    controller: _nomcategorieController,
                    decoration: const InputDecoration(
                        hintText: "Category name", labelText: "Name"),
                    validator: (value) {
                        if (value!.isEmpty) {
                            return 'Please enter the name';
                        }
                        return null;
                    },
                ),
                Container(
                    height: 40.0,
                ),
                Column(
                    children: [
                        ElevatedButton.icon(
                            icon: const Icon(Icons.photo_album_rounded, color:
Colors.pink),
                            style: ElevatedButton.styleFrom(
                                backgroundColor: Colors.greenAccent,
                            ),
                            onPressed: () {
                                _pickgallery();
                            },
                            label: const Text(
                                "PICK FROM GALLERY",
                                style: TextStyle(
                                    color: Colors.blueGrey, fontWeight: FontWeight.bold),
                            ),
                        ),
                        Container(
                            height: 40.0,
                        ),
                        ElevatedButton.icon(
                            icon: const Icon(Icons.camera_alt_sharp, color:
Colors.purple),

```

```

        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.lightGreenAccent,
        ),
        onPressed: () {
          _pickcamera();
        },
        label: const Text(
          "PICK FROM CAMERA",
          style: TextStyle(
            color: Colors.blueGrey, fontWeight: FontWeight.bold),
        ),
      ),
    ),
    Container(
      height: 40.0,
    ),
    SizedBox(
      height: 250,
      width: 250,
      child: imagecloud != ""
        ? Image.network(imagecloud)
        : Container(
            decoration: BoxDecoration(color: Colors.red[200]),
            child: Icon(
              Icons.camera_alt,
              color: Colors.grey[800],
            ),
          ),
    ),
    ),
    Container(
      height: 20.0,
    ),
    Center(
      child: ElevatedButton(
        onPressed: () {
          if (_formKey.currentState!.validate()) {
            try {
              // Mise à jour de la catégorie
              _controller.updateCategorie(
                widget.categorie.id, // ID de la catégorie à
modifier
                _nomcategorieController.text,
                imagecloud,
              );

              ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(
                  content: Text('Category updated successfully'),
                ),
              );
              Navigator.of(context).pushNamed('/Categories');
            } catch (e) {}
          }
        },
      ),
    ),
  ],
);

```

```

        } catch (error) {
          ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(
              content: Text('Error updating category:
$error')),
            );
        }
      },
      child: const Text("Update"),
    ),
  ],
),
],
),
);
}
}

```

Principales différences entre le formulaire add et edit :

- Passer un objet `Categorie` à l'écran du formulaire, ce qui permettra de pré-remplir les champs.
- Dans la méthode `initState()`, pré-remplir les champs du formulaire avec les valeurs de la catégorie existante.
- Dans la partie soumission, remplacer la méthode d'ajout par une méthode de mise à jour. Adapter le contrôleur `GetX` pour appeler une méthode de mise à jour.

Ensuite créer le fichier :

lib > presentation > screens >  editcategorie.screen.dart

```

import 'package:flutter/material.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import
'package:myflutterapplication/presentation/widgets/editcategorieform.widget.da
rt';

class Editcategorie extends StatelessWidget {
  final CategorieEntity categorie;
  const Editcategorie({super.key, required this.categorie});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(

```

```

        elevation: 15,
        backgroundColor: Colors.greenAccent,
        title: const Text("Edit Categories"),
        leading: IconButton(
          onPressed: () {},
          icon: const Icon(Icons.category_rounded),
        ),
      ),
      body: EditCategorieForm(categorie: categorie),
    );
  }
}

```

Ajouter la route correspondante

lib >  approuter.dart

```

// Importation des packages nécessaires
import 'package:flutter/material.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import 'package:myflutterapplication/models/Product.class.dart';
import
'package:myflutterapplication/presentation/screens/addcategorie.screen.dart';
import
'package:myflutterapplication/presentation/screens/categorieslist.screen.dart'
;
import
'package:myflutterapplication/presentation/screens/editcategorie.screen.dart';
import 'package:myflutterapplication/screens/details.dart';
import 'package:myflutterapplication/screens/documents.dart';
import 'package:myflutterapplication/screens/exitscreen.dart';
import 'package:myflutterapplication/screens/menu.dart';
import 'package:myflutterapplication/screens/myproducts.dart';
import 'package:myflutterapplication/screens/products.dart';
import 'package:myflutterapplication/screens/subscribe.dart';
import 'package:myflutterapplication/widgets/myappbar.dart';
import 'package:myflutterapplication/widgets/mybottomnavbar.dart';
import 'package:myflutterapplication/widgets/mydrawer.dart';

// Définition d'une fonction qui retourne les routes
Map<String, WidgetBuilder> appRoutes() {

  return {
    '/': (context) => const Scaffold(
      appBar: Myappbar(),
      body: Menu(),
      drawer: MyDrawer(),

```

```

        bottomNavigationBar: MyBottomNavigation(),
      ),
      '/Items': (context) => Scaffold(
        appBar: AppBar(
          title: const Text('My Products'),
        ),
        body: const Myproducts(),
        drawer: const MyDrawer(),
        bottomNavigationBar: const MyBottomNavigation(),
      ),
      '/Exit': (context) => const ExitScreen(), // Route associée à l'action de
fermeture
      '/Documents': (context) => const Documents(), // Route pour l'écran
Documents
      '/Products': (context) => const Products(), // Route pour l'écran Products
      '/details': (context) {final product =
ModalRoute.of(context)!.settings.arguments as Product;
        return Details(myListElement: product);
      }, // Route pour l'écran Details
      '/Subscribe': (context) => const Subscribe(), // Route pour l'écran
Subscribe
      '/Categories': (context) => const Categorieslist(), // Route pour l'écran
Categories
      '/addcategories': (context) => const Addcategorie(), // Route pour l'écran
Addcategorie
      '/editcategories': (context) {final categorie =
ModalRoute.of(context)!.settings.arguments as CategorieEntity;
        return Editcategorie(categorie: categorie);
      }, // Route pour l'écran Editcategorie
    };
  }

```

### Remarque :

Getx dispose de son propre système de routage qu'on aurait pu utiliser également.

Il offre des fonctionnalités avancées comme la navigation sans contexte, le passage d'arguments et la gestion de middlewares.

Mais il faut définir vos routes dans l'application en utilisant **GetMaterialApp** au lieu de **MaterialApp**.

Par exemple :

```

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(

```

```

title: 'My Flutter Application',

initialRoute: '/',

getPages: [

  GetPage(name: '/', page: () => HomePage()),

  GetPage(name: '/edit', page: () => Editcategorie()),

], ); }}

```

Puis utiliser Get pour naviguer

```
Get.toNamed('/edit', arguments: categorie);
```

Pour plus de détail, consulter :

<https://medium.com/@onalojoseph96/getx-routing-management-in-flutter-7c44f785e592>

Enfin, pour faire appel à editcategorie.screen, ajouter ce code à :

lib > presentation > widgets >  categorieslist.widget.dart

```

import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:myflutterapplication/domain/entities/categorie.entity.dart';
import
'package:myflutterapplication/presentation/controllers/categorie.controller.dart';

class Categorieslistwidget extends StatelessWidget {
  final CategorieEntity categories;
  Categorieslistwidget({super.key, required this.categories});
  //on a enlevé const pour Categorieslistwidget
  // car : Can't define the 'const' constructor because the field 'controller'
  is initialized with a non-constant value.

  final controller = Get.find<CategorieController>();

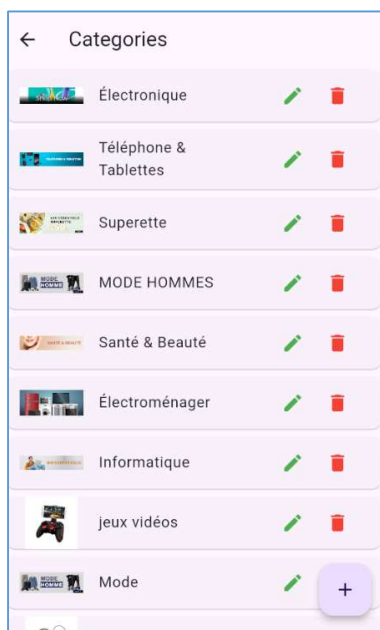
  @override
  Widget build(BuildContext context) {
    return SingleChildScrollView(
      child: Card(
        child: ListTile(
          leading: Image.network(
            categories.imagecategorie,
            width: 68,
            height: 68,
          ),


```

```

        title: Text(categories.nomcategorie),
        trailing: Wrap(
          children: <Widget>[
            IconButton(
              icon: const Icon(
                Icons.edit,
                color: Colors.green,
              ),
              onPressed: () => {
                Navigator.of(context)
                  .pushNamed("/editcategories", arguments:
categories)
              },
            ),
            IconButton(
              icon: const Icon(
                Icons.delete,
                color: Colors.red,
              ),
              onPressed: () => controller.deleteCategorie(categories.id),
            ),
          ],
        ),
      ),
    ),
  );
}
}

```








Edit Categories

Name


Mode

 PICK FROM GALLERY

 PICK FROM CAMERA




Update





Edit Categories

Name

Modes et décoration

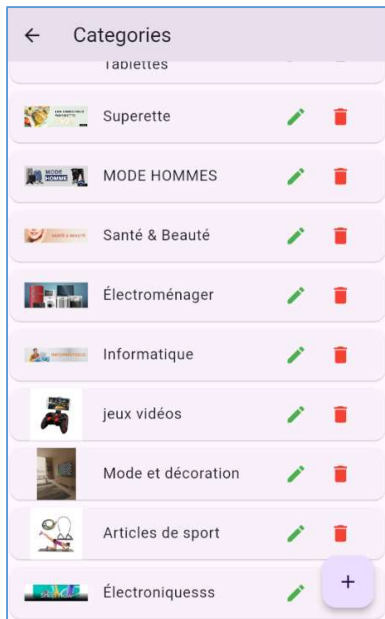
 PICK FROM GALLERY

 PICK FROM CAMERA



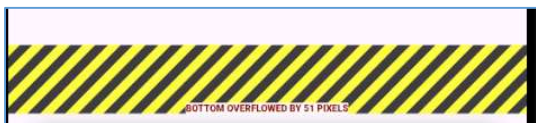
Update



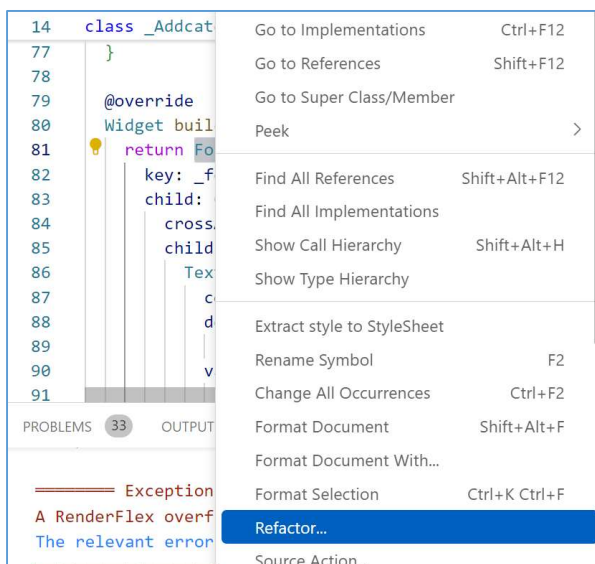


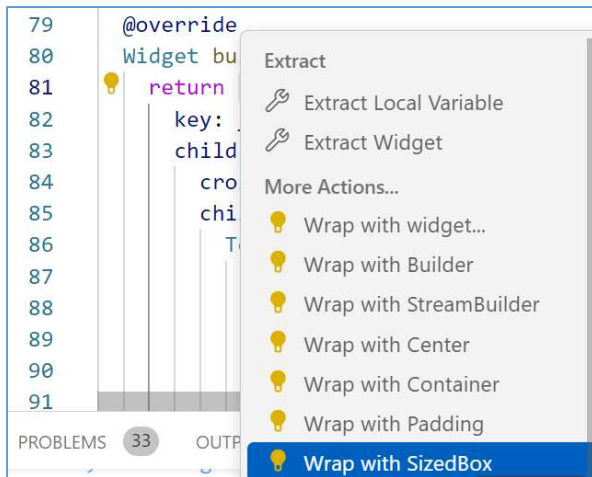
### Remarque :

L'exception `A RenderFlex overflowed by 22 pixels on the bottom` se produit généralement lorsque le contenu d'un widget dépasse l'espace disponible à l'intérieur d'une colonne, ligne ou autre layout de type Flex.



Pour résoudre ce problème :





```
@override
Widget build(BuildContext context) {
  return SizedBox(
    height: MediaQuery.of(context).size.height * 0.8, // 80% de la hauteur de l'écran
    child: Form(
```

```
height: MediaQuery.of(context).size.height * 0.8, // 80% de la hauteur de
l'écran
```

Une autre solution consiste à ajouter le Widget `SingleChildScrollView`

```
@override
Widget build(BuildContext context) {
  return SingleChildScrollView(
    child: Form(
```