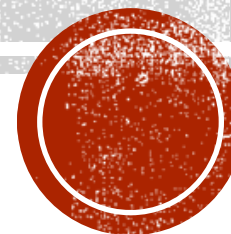


PROJET





```
void main() {  
  runApp(MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'My E-Commerce',  
      theme: ThemeData(  
        primarySwatch: Colors.deepOrange,  
        scaffoldBackgroundColor: Colors.white,  
        // Ajout d'un style pour les icônes du BottomNavigationBar  
        iconTheme: IconThemeData(color: Colors.white),  
      ),  
      home: MainScreen(),  
    );  
  }  
}
```

lib
screens
main_screen.dart
main.dart

- **AppBar :**
- `backgroundColor: Colors.deepOrange` pour un rendu vif.
- `centerTitle: true` pour centrer le titre et donner une apparence plus équilibrée.
- Légère `elevation: 4.0` pour un effet d'ombre discret.
- **BottomNavigationBar :**
- `backgroundColor: Colors.deepOrange` pour faire écho à l'AppBar.
- `selectedItemColor: Colors.white` et `unselectedItemColor: Colors.white70` pour un bon contraste, facilitant la lecture et l'identification de l'onglet sélectionné.



```
class MainScreen extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      // AppBar avec des couleurs vives, titre centré et icône pour le rendu plus moderne  
      appBar: AppBar(  
        title: Text('My E-Commerce', style: TextStyle(color: Colors.white)),  
        centerTitle: true,  
        backgroundColor: Colors.deepOrange,  
        elevation: 4.0,  
      ),  
  
      // Body inchangé : texte au centre  
      body: Center(  
        child: Text(  
          'Home Screen',  
          style: TextStyle(fontSize: 20),  
        ),  
      ),  
  
      // BottomNavigationBar avec un fond coloré et des icônes contrastées  
      bottomNavigationBar: bottomNav(),  
    );  
  }  
}  
  
BottomNavigationBar bottomNav() {  
  return BottomNavigationBar(  
    currentIndex: 0,  
    backgroundColor: Colors.deepOrange, // Couleur vive de fond  
    selectedItemColor: Colors.white, // Couleur de l'item sélectionné  
    unselectedItemColor: Colors  
      .white70, // Couleur légèrement atténuée pour les items non sélectionnés  
    items: const [  
      BottomNavigationBarItem(  
        icon: Icon(Icons.home),  
        label: 'Home',  
      ),  
      BottomNavigationBarItem(  
        icon: Icon(Icons.shopping_cart),  
        label: 'Cart',  
      ),  
      BottomNavigationBarItem(  
        icon: Icon(Icons.person),  
        label: 'Profile',  
      ),  
    ],  
  );  
}
```

My E-Commerce

Design

Home Screen

Home

Cart

Profile



PRODUCT CARD

•Constructeur dans ProductCard :

Le constructeur de la classe ProductCard permet d'initialiser ses propriétés (name, price, imageUrl, onTap) lorsque vous créez une instance de ce widget. Il garantit que le widget dispose des informations nécessaires pour s'afficher (nom, prix, image) et éventuellement réagir à une action (onTap).

•shape dans Card :

L'attribut shape permet de définir la forme de la carte, par exemple des coins arrondis avec RoundedRectangleBorder. Sans cela, la carte a une forme rectangulaire standard.

•clipBehavior dans Card :

clipBehavior spécifie comment le contenu est rogné aux limites de la carte. Avec Clip.antiAlias ou Clip.hardEdge, par exemple, tout ce qui dépasse la forme définie par shape sera coupé, assurant un rendu propre (par exemple, l'image suit la forme arrondie de la carte).

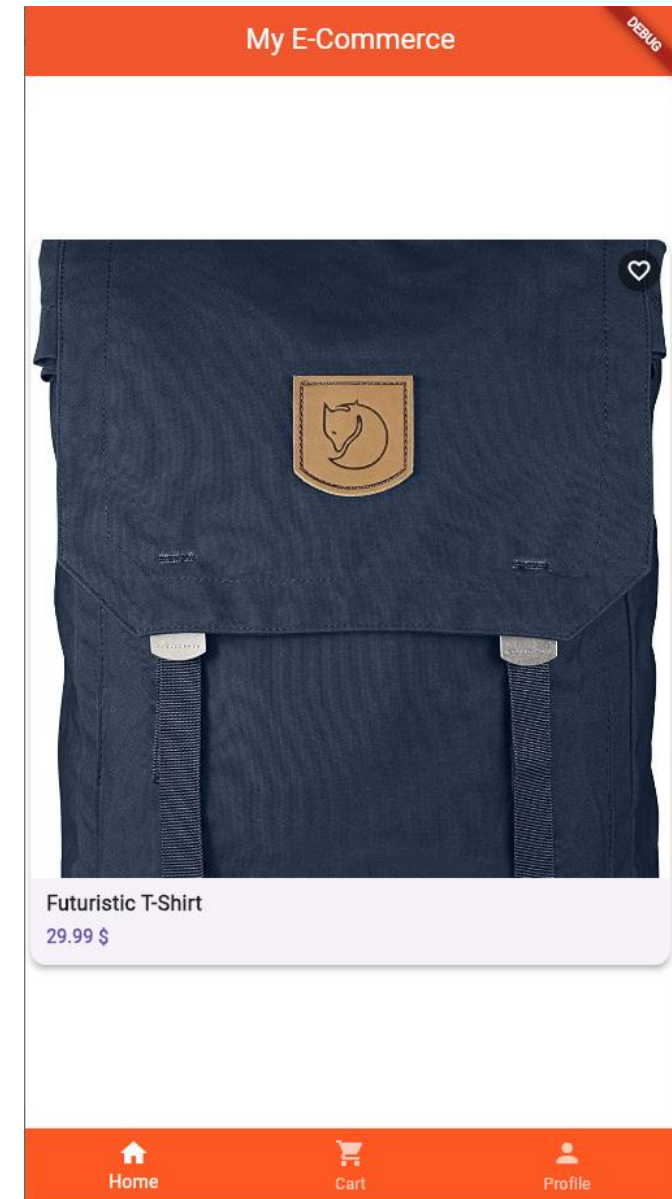
•VoidCallback :

C'est un type de fonction (une signature) qui ne prend aucun paramètre et ne retourne rien. On l'utilise pour onTap (lors d'un clic sur la carte), afin de définir une action à effectuer sans avoir besoin d'arguments ou de valeur de retour. C'est simplement un "bouton poussoir" logiciel.

```
class ProductCard extends StatelessWidget {
  final String name;
  final String price;
  final String imageUrl;
  final VoidCallback? onTap;

  const ProductCard({
    required this.name,
    required this.price,
    required this.imageUrl,
    this.onTap,
  });

  @override
  Widget build(BuildContext context) {
    return InkWell(
      onTap: onTap,
      borderRadius: BorderRadius.circular(12),
      child: Card(
        elevation: 4.0,
        shape:
          RoundedRectangleBorder(borderRadius: BorderRadius.circular(12.0)),
        clipBehavior: Clip.antiAlias,
```



PRODUCT CARD

Stack

- Permet de superposer plusieurs widgets comme des calques.
- Ici, l'image est au fond et l'icône favoris est par-dessus, dans le coin supérieur droit.

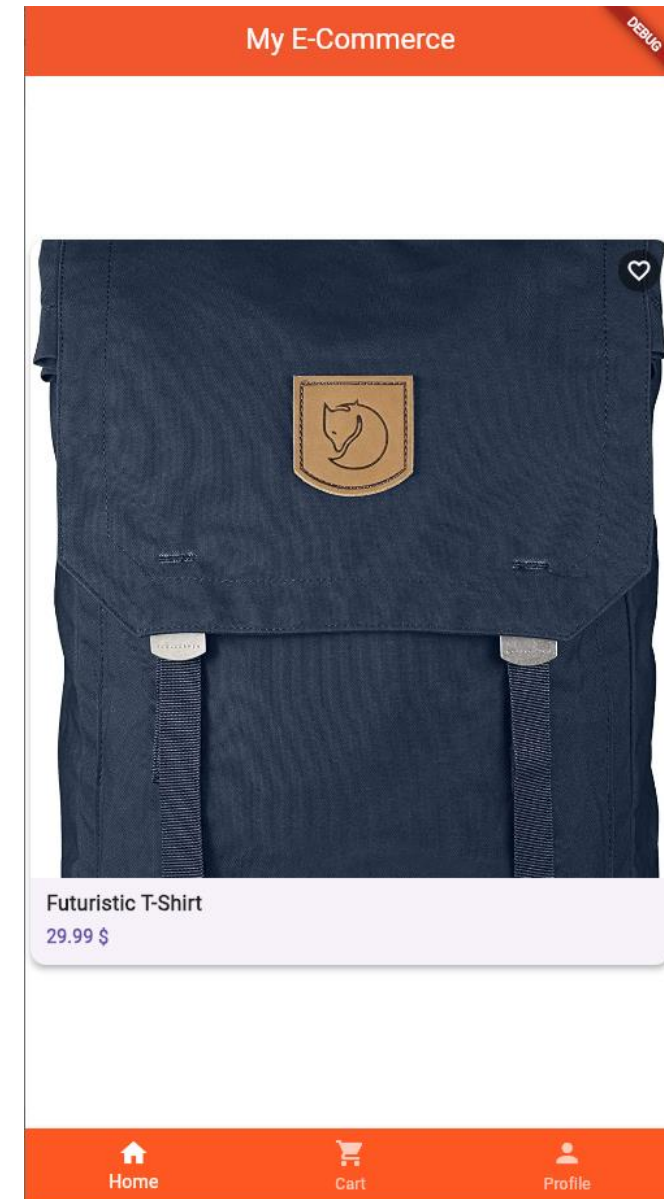
Positioned

- S'utilise à l'intérieur d'un Stack.
- Positionne un widget précisément (ex: top: 8, right: 8).
- Ici, on place l'icône favoris exactement où on le souhaite sur l'image.

AspectRatio

- Contraint un widget (ex: une image) à une certaine proportion largeur/hauteur.
- aspectRatio: 1 rend l'image carrée, ce qui est plus harmonieux.

```
children: [
  Stack(
    children: [
      AspectRatio(
        aspectRatio: 1,
        child: Image.network(
          imageUrl,
          fit: BoxFit.cover,
        ),
      ),
      Positioned(
        top: 8,
        right: 8,
        child: Container(
          decoration: BoxDecoration(
            color: Colors.black45,
            borderRadius: BorderRadius.circular(20),
          ),
          padding: const EdgeInsets.all(6.0),
          child: Icon(
            Icons.favorite_border,
            color: Colors.white,
            size: 20,
          ),
        ),
      ),
    ],
  ),
],
```



LISTS

En Flutter, il existe plusieurs widgets pour afficher des collections d'éléments (texte, images, cartes, etc.). Voici les plus courants :

1. **ListView** :

- Affiche une liste d'éléments disposés verticalement les uns en dessous des autres.
- Idéal pour un défilement vertical simple, comme une liste de produits ou de messages.
- Peut utiliser un constructeur builder (`ListView.builder`) pour créer les éléments à la volée, ce qui est plus performant lorsque la liste est longue.

2. **GridView** :

- Affiche les éléments en grille, c'est-à-dire en plusieurs colonnes et plusieurs lignes.
- Utile pour afficher des galeries d'images, des produits sous forme de cartes, ou tout autre contenu qui s'adapte mieux en mosaïque.
- Peut également utiliser un builder (`GridView.builder`) et un délégué (`SliverGridDelegate`) pour un contrôle plus fin du nombre de colonnes et de l'espacement.

3. **CustomScrollView + Slivers** :

- Permet de composer des mises en page plus complexes en utilisant des "slivers" (des portions défilables).
- On peut mélanger des listes, des grilles, des headers, etc., dans un même défilement.
- Offre plus de flexibilité et de personnalisation, au prix d'une complexité un peu plus élevée.

4. **SingleChildScrollView** :

- Permet de rendre défilable un contenu unique et potentiellement long.
- Utile pour un contenu simple (un long article de texte, une grande image), mais pas optimisé pour les longues listes d'éléments multiples.
- Moins performant si vous avez beaucoup d'éléments, car tout est rendu en même temps.

En résumé :

- *ListView* : pour des listes simples et longues, une colonne défilante.
- *GridView* : pour des mises en page en grille.
- *CustomScrollView/Slivers* : pour des mises en page complexes et mixtes (combinaison de listes, grilles, headers...).
- *SingleChildScrollView* : pour un contenu unique qui déborde de l'écran, mais pas pour une collection importante d'éléments.



LIST PRODUCT

- LayoutBuilder :**

LayoutBuilder est un widget qui donne accès à la taille disponible pour son enfant. Grâce à son builder, on obtient les constraints qui indiquent la largeur (maxWidth) et la hauteur disponibles. Cela permet de calculer dynamiquement le nombre de colonnes en fonction de la place réelle sur l'écran.

- Calcul du crossAxisCount :**

On choisit d'abord une largeur approximative pour une carte, par exemple `cardWidth = 180.0` pixels.

On divise ensuite la largeur disponible `constraints.maxWidth` par `cardWidth`.

Par exemple, si `maxWidth` est de 500 pixels, $500 / 180 \approx 2.77$.

En utilisant `.floor()`, on obtient 2, ce qui signifie qu'on peut afficher 2 cartes côte à côte sans débordement.

Si la largeur était de 800 pixels, $800 / 180 \approx 4.44$, donc `.floor()` donne 4 cartes par ligne.

- gridDelegate**

Le `gridDelegate` est un paramètre essentiel du `GridView` qui détermine la manière dont la grille va organiser ses éléments. Il indique au `GridView` comment répartir les widgets (ici, les cartes produits) en lignes et en colonnes, et définit l'espacement entre eux.

- SliverGridDelegateWithFixedCrossAxisCount :**

Avec la valeur de `crossAxisCount` calculée dynamiquement, on la transmet à `SliverGridDelegateWithFixedCrossAxisCount`.

Ce délégué gère la mise en page de la grille.

- `crossAxisCount`: Nombre de colonnes calculé.

- `mainAxisSpacing`: Espace vertical entre les lignes.

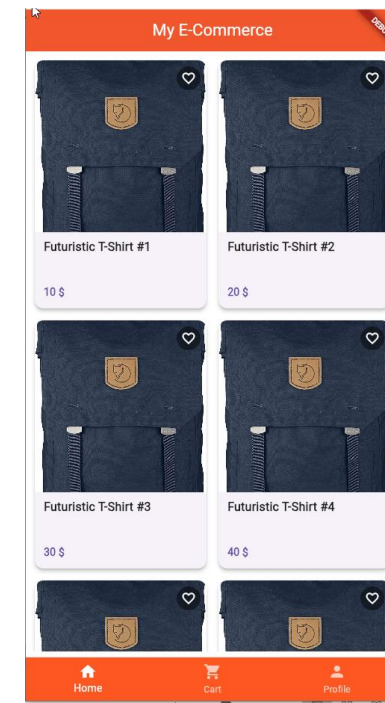
- `crossAxisSpacing`: Espace horizontal entre les colonnes.

- `childAspectRatio`: Ratio largeur/hauteur pour les cartes afin d'obtenir un rendu plus harmonieux.

- GridView.builder :**

Une fois le nombre de colonnes déterminé, `GridView.builder` construit la grille de produits.

Chaque produit est affiché dans une carte (`ProductCard`).



```
LayoutBuilder(  
  builder: (context, constraints) {  
    // On choisit une largeur de carte approximative (par ex. 180 pixels).  
    // En divisant la largeur disponible par 180, on obtient un nombre "idéal" de colonnes.  
    double cardWidth = 180.0;  
    int crossAxisCount = (constraints.maxWidth / cardWidth).floor();  
  
    // On s'assure qu'il y ait au moins 2 colonnes.  
    if (crossAxisCount < 2) {  
      crossAxisCount = 2;  
    }  
  
    return Padding(  
      padding: const EdgeInsets.all(8.0),  
      child: GridView.builder(  
        itemCount: products.length,  
        gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(  
          crossAxisCount: crossAxisCount,  
          mainAxisSpacing: 8.0,  
          crossAxisSpacing: 8.0,  
          childAspectRatio: 0.7,  
        ),  
      ),  
    );  
  },  
)
```

DETAIL SCREEN

Navigator.pushNamed :

- **context** : Contexte actuel pour la navigation.
- **DetailScreen.routeName** : Route nommée vers l'écran de détail.
- **arguments** : Passe les informations du produit (name, price, imageUrl) à l'écran de détail.

```
return InkWell(  
  onTap: () {  
    // Navigation vers l'écran de détail avec les i  
    nformations du produit  
    Navigator.pushNamed(  
      context,  
      DetailScreen.routeName,  
      arguments: {  
        'name': name,  
        'price': price,  
        'imageUrl': imageUrl,  
      },  
    );  
  },  
);
```

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'My E-Commerce',  
      theme: ThemeData(  
        primarySwatch: Colors.deepOrange,  
        scaffoldBackgroundColor: Colors.white,  
      ),  
      home: MainScreen(),  
      routes: {  
        // Définition de la route nommée pour l'écran de détail  
        DetailScreen.routeName: (context) => DetailScreen(),  
      },  
    );  
  }  
}
```



DETAIL SCREEN

SingleChildScrollView

- Permet de rendre le contenu défilable si la hauteur totale dépasse celle de l'écran.

Image.network

- **imageUrl** : Affiche l'image du produit en plein écran avec une hauteur fixe de 300 pixels.

- **fit: BoxFit.cover** : Assure que l'image couvre toute la zone disponible sans déformation.

List.generate

- Qu'est-ce que c'est ?

- List.generate est une méthode qui crée une liste en générant chaque élément à l'aide d'une fonction de génération.

- Syntaxe :

dart

List.generate(int length, E generator(int index))

- **length** : Le nombre d'éléments à générer.
- **generator** : Une fonction qui prend un index et retourne un élément de la liste.

- Dans notre cas :

- List.generate(5, (index) { ... }) crée une liste de 5 éléments.
- Chaque élément est une icône d'étoile, soit remplie (Icons.star) soit vide (Icons.star_border), en fonction de la note du produit.

```
body: SingleChildScrollView(
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      // Image du produit avec une hauteur fixe
      Image.network(
        product['imageUrl'],
        width: double.infinity,
        height: 300,
        fit: BoxFit.cover,
      ),
      // Nom du produit
      Padding(
        padding: const EdgeInsets.all(16.0),
        child: Text(
          product['name'],
          style: TextStyle(
            fontSize: 28,
            fontWeight: FontWeight.bold,
            color: Colors.black87,
          ),
        ),
      ),
      // Prix du produit et Rating
      Padding(
        padding: const EdgeInsets.symmetric(horizontal: 16.0),
        child: Row(
          children: [
            Text(
              product['price'],
              style: TextStyle(
                fontSize: 22,
                color: Theme.of(context).primaryColor,
                fontWeight: FontWeight.w600,
              ),
            ),
            SizedBox(width: 16),
            // Rating stars
            Row(
              children: List.generate(5, (index) {
                return Icon(
                  index < (product['rating'] ?? 0)
                    ? Icons.star
                    : Icons.star_border,
                  color: Colors.amber,
                  size: 24,
                );
              }),
            ),
          ],
        ),
      ),
    ],
  ),
)
```



DETAIL SCREEN

```
final Map<String, dynamic> product = ModalRoute.of(context)!.settings.arguments as Map<String, dynamic>;
```

Objectif de cette Ligne

Cette ligne permet de **recupérer les données du produit** qui ont été passées lors de la navigation vers l'écran de détail (DetailScreen). En d'autres termes, elle extrait les informations spécifiques du produit sélectionné (comme le nom, le prix, l'image, et le rating) pour les afficher sur l'écran de détail.

Décomposition et Explication

1.ModalRoute.of(context)

•Qu'est-ce que c'est ?

- ModalRoute.of(context) est une méthode qui permet d'accéder à la route actuellement affichée dans le contexte donné.
- Route** : En Flutter, une "route" représente une page ou un écran de l'application. Lorsque vous naviguez d'un écran à un autre, vous passez d'une route à une autre.

•Retourne :

- Un objet de type ModalRoute<T>?, où T est le type des arguments passés à la route.
- Le point d'interrogation (?) indique que la méthode peut retourner null si aucune route n'est trouvée dans le contexte.

2.!.settings.arguments

•Exclamation (!) :

- L'opérateur ! est l'opérateur de **non-null assertion** en Dart. Il indique au compilateur que vous êtes certain que la valeur n'est pas null.
- Attention** : Utilisez-le uniquement lorsque vous êtes certain que la valeur ne sera jamais null, sinon cela peut entraîner des erreurs d'exécution.

•settings.arguments :

- settings est une propriété de la route (ModalRoute) qui contient des informations sur la route, y compris les **arguments** passés lors de la navigation.
- arguments est un champ générique qui peut contenir n'importe quel type de données (souvent un objet ou une carte de données).

3.as Map<String, dynamic>

•Qu'est-ce que c'est ?

- as est l'opérateur de **cast** (conversion de type) en Dart. Il permet de forcer la conversion d'un objet d'un type à un autre.

•Map<String, dynamic> :

- Un Map en Dart est une collection de paires clé-valeur.
- <String, dynamic> indique que les clés sont des chaînes de caractères (String) et que les valeurs peuvent être de n'importe quel type (dynamic).
- Dans ce contexte, le Map<String, dynamic> contient les informations du produit, comme le nom, le prix, l'image, et le rating.



API

Définition :

API signifie **Application Programming Interface** (Interface de Programmation d'Applications). C'est un ensemble de règles et de protocoles qui permet à différentes applications de communiquer entre elles.

Fonctionnement :

- **Client-Serveur** : Une application (client) envoie une requête à une autre application ou serveur, qui répond avec les données demandées.
- **Endpoints** : Les points d'accès spécifiques dans une API où les requêtes sont envoyées (par exemple, <https://fakestoreapi.com/products>).
- **Méthodes HTTP** : Les actions que vous pouvez effectuer sur les données, telles que :
 - **GET** : Récupérer des données.
 - **POST** : Envoyer des données pour créer une nouvelle ressource.
 - **PUT/PATCH** : Mettre à jour une ressource existante.
 - **DELETE** : Supprimer une ressource.

Exemple d'utilisation :

Imaginons que vous souhaitez afficher une liste de produits dans votre application Flutter. Vous pouvez utiliser une API comme FakeStore API pour récupérer les données des produits en envoyant une requête GET.



PACKAGES

Packages Flutter :

- **Définition** : Les packages sont des collections de fonctionnalités réutilisables (widgets, bibliothèques, etc.) qui facilitent le développement d'applications Flutter.

- **Types de Packages** :

- **Packages officiels** : Fournis par l'équipe Flutter (ex. flutter/material.dart).


- **Packages tiers** : Développés par la communauté et disponibles sur pub.dev.

Fichier pubspec.yaml :

- **Rôle** : C'est le fichier de configuration principal de votre projet Flutter. Il spécifie les dépendances (packages) dont votre projet a besoin, ainsi que d'autres configurations comme les assets (images, fichiers, etc.).

Ajouter un Package :

1. Ouvrez pubspec.yaml.
2. Ajoutez le package sous dependencies
3. Sauvegardez le fichier.
4. Flutter exécutera automatiquement `flutter pub get` pour télécharger les packages.



```
dependencies:  
  flutter:  
    sdk: flutter  
  http: ^0.13.4
```



LE PACKAGE HTTP DANS FLUTTER

Qu'est-ce que c'est ?

Le package **HTTP** est une bibliothèque Flutter/Dart qui facilite l'envoi de requêtes HTTP et la gestion des réponses. Il est couramment utilisé pour interagir avec des APIs RESTful.

Fonctionnalités Principales :

- Envoyer des requêtes **GET, POST, PUT, DELETE**.
- Gérer les en-têtes (headers) et les paramètres.
- Parser les réponses **JSON**.
- Gérer les erreurs et les statuts de réponse.

Installation :

Ajoutez le package HTTP dans votre pubspec.yaml



•Importations :

- dart:convert : Utilisé pour convertir les données JSON en objets Dart.
- package:http/http.dart : Fournit les fonctionnalités pour envoyer des requêtes HTTP.

•Fonction fetchPosts :

•Définition de l'URL :

- Convertit la chaîne de caractères en un objet Uri.

•Envoyer une Requête GET :

- Utilise la méthode get pour envoyer une requête HTTP GET à l'URL définie.
- await attend que la requête soit terminée avant de passer à l'étape suivante.

•Vérifier le Statut de la Réponse :

- statusCode 200 signifie que la requête a réussi.

•Décoder et Utiliser les Données JSON :

- Convertit la chaîne JSON en une liste d'objets Dart
- Parcourt chaque post et affiche son ID et son titre.

•Gestion des Erreurs :

Capture et affiche toute erreur survenue lors de l'envoi de la requête ou du traitement des données.

```
import 'dart:convert'; // Pour décoder le JSON
import 'package:http/http.dart' as http;

Future<void> fetchPosts() async {
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');

  try {
    final response = await http.get(url);

    if (response.statusCode == 200) {
      // Décoder la réponse JSON
      final List<dynamic> posts = json.decode(response.body);

      // Afficher les titres des posts
      for (var post in posts) {
        print('Post ${post['id']}: ${post['title']}');
      }
    } else {
      print('Erreur: Statut ${response.statusCode}');
    }
  } catch (e) {
    print('Erreur: $e');
  }
}

void main() async {
  await fetchPosts();
}
```

Qu'est-ce qu'un Future ?

Définition :

Un **Future** en Dart représente une opération asynchrone qui peut produire une valeur ou une erreur à un moment futur. C'est un moyen de gérer des tâches qui prennent du temps sans bloquer l'exécution de votre programme.

Pourquoi utiliser Future ?

- Opérations Asynchrones** : Parfait pour des tâches comme les requêtes réseau, la lecture de fichiers, ou toute autre opération qui prend du temps.
- Non-Bloquant** : Permet à votre application de rester réactive pendant que l'opération est en cours.

Comment Fonctionne un Future ?

Étapes Clés :

- 1.**Création d'un Future** : Déclencher une opération asynchrone.
- 2.**Attente d'un Future** : Attendre que l'opération soit terminée pour obtenir le résultat.
- 3.**Gestion des Résultats** : Utiliser le résultat ou gérer les erreurs une fois le Future complété.

Méthodes Principales :

- then** : Exécute une fonction lorsque le Future est complété avec succès.
- catchError** : Gère les erreurs si le Future échoue.
- async et await** : Syntaxe plus lisible pour attendre les Future et gérer les résultats.



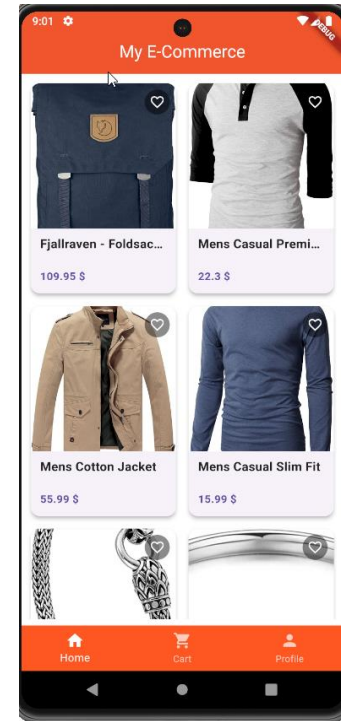
INTEGRATION API

Le Type de Retour : `Future<List<Map<String, dynamic>>>`

- **Future<...>** :
Un Future en Dart représente une opération asynchrone qui se terminera plus tard, soit avec une valeur, soit avec une erreur. Par exemple, récupérer des données d'une API prend du temps, et on ne veut pas bloquer l'application pendant ce temps. En utilisant un Future, on peut continuer à faire d'autres choses pendant que les données sont en cours de chargement.
- **List<Map<String, dynamic>>>** :
Ici, le type contenu dans le Future est `List<Map<String, dynamic>>>`. Cela signifie que lorsque le Future sera complété (l'opération asynchrone terminée), il fournira une **liste** (List) d'éléments, et chaque élément est une Map dont les clés sont des chaînes de caractères (String) et les valeurs peuvent être de n'importe quel type (dynamic).

En d'autres termes, ce Future promet qu'à la fin de l'opération, on aura accès à une liste de produits, où chaque produit est représenté par un objet clé-valeur

```
Future<List<Map<String, dynamic>>> fetchProductsFromApi() async {  
  final response =  
    await http.get(Uri.parse('https://fakestoreapi.com/products'));  
  
  if (response.statusCode == 200) {  
    final List data = json.decode(response.body);  
    return data.map((product) {  
      return {  
        "name": product['title'],  
        "price": "${product['price']} $",  
        "imageUrl": product['image'],  
        "rating": (product['rating']['rate'] as num).round(),  
      };  
    }).toList();  
  } else {  
    throw Exception('Failed to load products');  
  }  
}
```



- `data.map(...)` applique une fonction à chaque élément de la liste `data`.
- Chaque product est un Map contenant des clés comme `title`, `price`, `image`, `rating`.
- On crée une nouvelle `Map<String, dynamic>` pour chaque produit, avec les clés `"name"`, `"price"`, `"imageUrl"` et `"rating"`.
- On formate le prix en ajoutant le symbole \$.
- On convertit le rating en nombre entier avec `.round()`.
- `.toList()` reconstruit le résultat du map en une `List<Map<String, dynamic>>`.



INTEGRATION API

Ce code utilise un FutureBuilder pour afficher le contenu de votre application en fonction de l'état d'un appel asynchrone (récupérer des produits via fetchProductsFromApi()).

1.future: fetchProductsFromApi() – C'est la fonction qui récupère les données des produits depuis l'API.

2.builder:

- **ConnectionState.waiting** : Affiche un indicateur de chargement pendant que les données sont en cours de récupération.
- **snapshot.hasError** : Affiche un message d'erreur si la récupération échoue.
- **!snapshot.hasData || snapshot.data!.isEmpty** : Affiche un message si aucun produit n'est disponible.
- **Sinon (données disponibles)** : Affiche une grille de produits.

3.Grille de produits :

- LayoutBuilder calcule le nombre de colonnes en fonction de la largeur disponible, assurant une présentation responsive.
- GridView.builder affiche chaque produit à l'aide de ProductCard.

```
FutureBuilder<List<Map<String, dynamic>>>({
  future:
    fetchProductsFromApi(), // Appel de la fonction depuis api_service.dart
  builder: (context, snapshot) {
    // Gestion des états : chargement, erreur, données
    if (snapshot.connectionState == ConnectionState.waiting) {
      return Center(child: CircularProgressIndicator());
    } else if (snapshot.hasError) {
      return Center(child: Text('Erreur: ${snapshot.error}'));
    } else if (!snapshot.hasData || snapshot.data!.isEmpty) {
      return Center(child: Text('Aucun produit disponible.'));
    } else {
      final products = snapshot.data!;
      return LayoutBuilder(
        builder: (context, constraints) {
          double cardWidth = 180.0;
          int crossAxisCount = (constraints.maxWidth / cardWidth).floor();
          if (crossAxisCount < 2) {
            crossAxisCount = 2;
          }
          return Padding(
            padding: const EdgeInsets.all(8.0),
            child: GridView.builder(
              itemCount: products.length,
              gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: crossAxisCount,
                mainAxisSpacing: 8.0,
                crossAxisSpacing: 8.0,
                childAspectRatio: 0.7,
              ),
              itemBuilder: (context, index) {
                final product = products[index];
                return ProductCard(
                  name: product["name"],
                  price: product["price"],
                  imageUrl: product["imageUrl"],
                  rating: product["rating"],
                  onTap: () {
                    // Navigation déjà gérée dans ProductCard si nécessaire
                  },
                );
              },
            ),
          );
        },
      );
    }
  },
);
```

