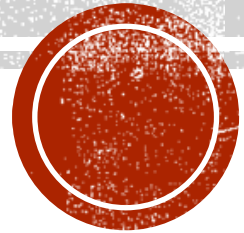
BLOC

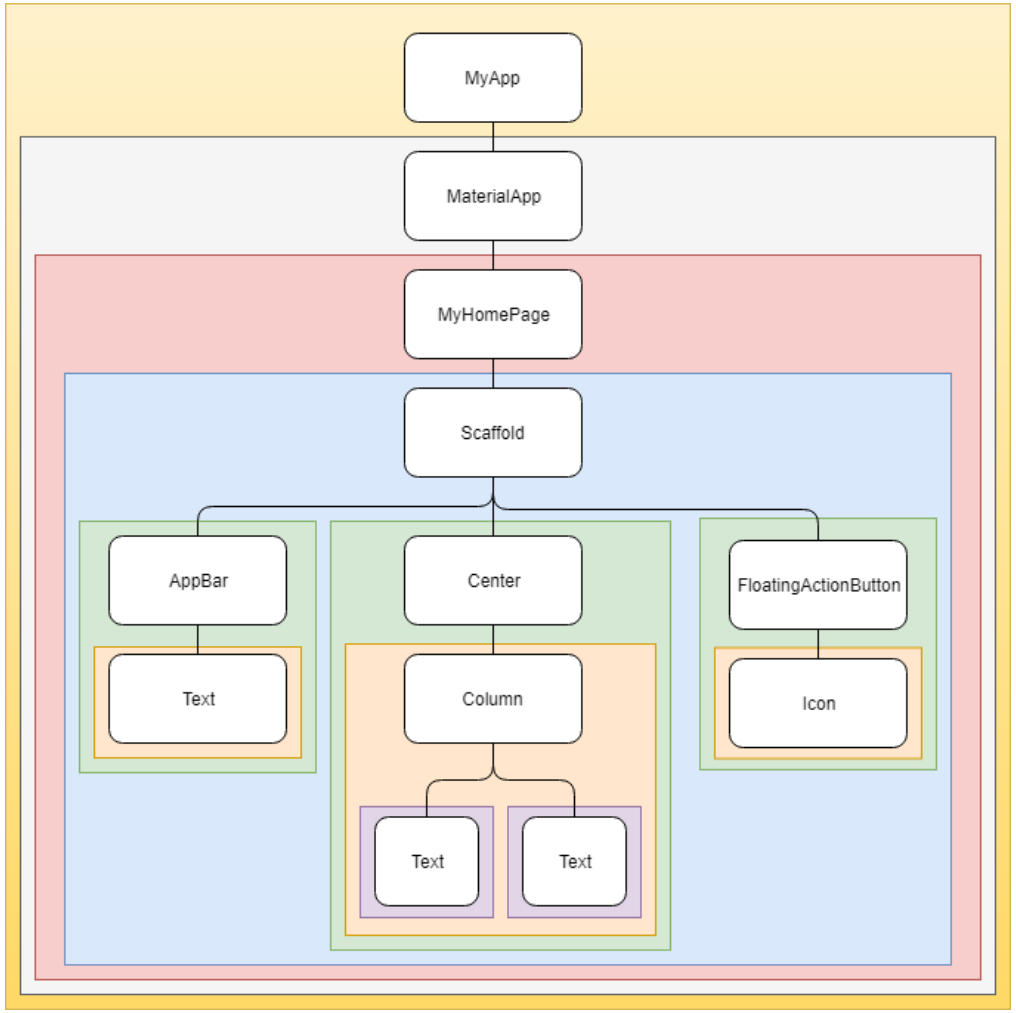
Formation flutter Avancée

Union Of Education



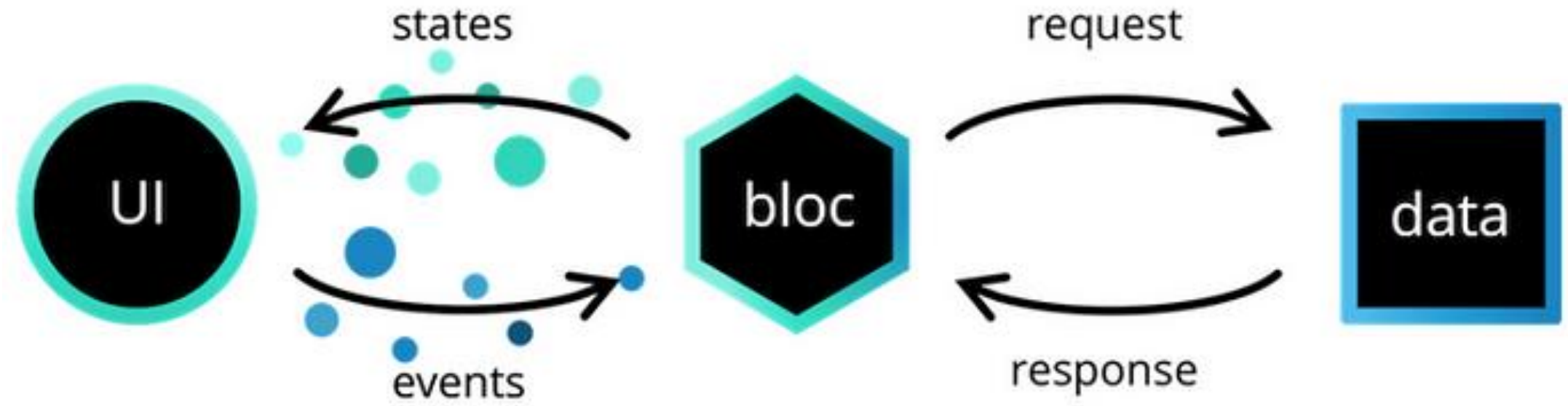
Formateur :Kamel.bahmed.info@gmail.com

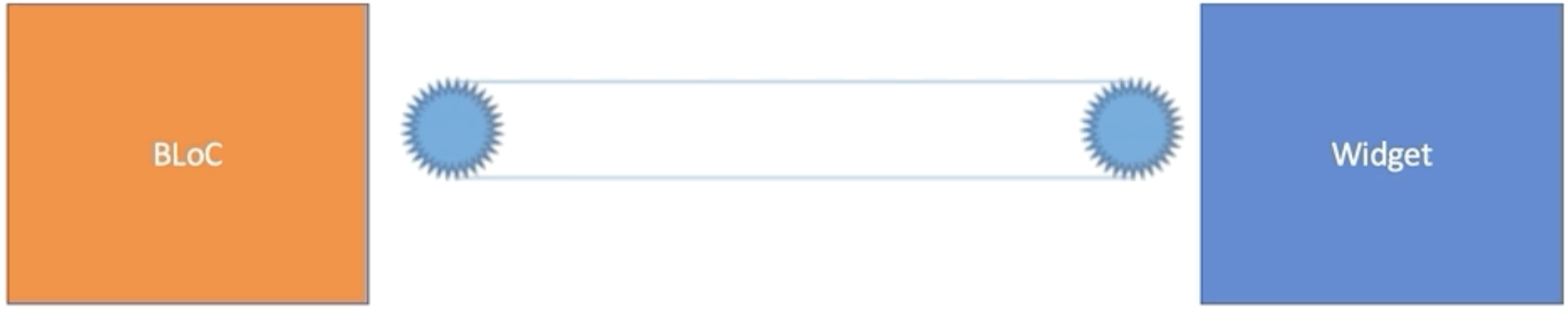
INTRODUCTION



- Une gestion d'état efficace permet de maintenir la cohérence des données et de garantir que l'interface utilisateur reflète toujours l'état actuel de l'application. Cependant, si la gestion d'état est négligée ou mal implémentée, cela peut entraîner des problèmes tels que des interfaces utilisateur non synchronisées, des erreurs d'affichage et une expérience utilisateur générale insatisfaisante.
- Les développeurs d'applications mobiles séparent le projet sur plusieurs couches pour mieux dispatcher les responsabilités entre les modules. Le modèle et la vue sont séparés, le contrôleur (ou ViewModel) envoyant des évènements entre eux. Ces approches, bien connues des développeurs natifs et Xamarin, sont : **Model-View-Controller (MVC)** ou **Model-View-ViewModel (MVVM)**
- BLoC signifie Business Logic Components. L'essentiel de BLoC est que **tout, dans l'application, doit être représenté comme un flux d'événements**. Le widget reproduit des évènements : interaction avec l'interface graphique, évènement système, changement d'orientation, etc. Le BLoC se trouve derrière le widget et il a la charge d'écouter et de réagir aux évènements produits par le widget. L'objectif de BLoC est en quelques sorte de gérer le cycle de vie d'un widget. Le langage de programmation Dart est même livré avec une syntaxe pour travailler avec des flux intégrés dans le langage. On peut donc considérer le BLoC comme étant une architecture clé en utilisant Flutter dès lors que l'on souhaite piloter un widget contenant de la logique métier.

FLUTTER BLOC




BLoC
Widget

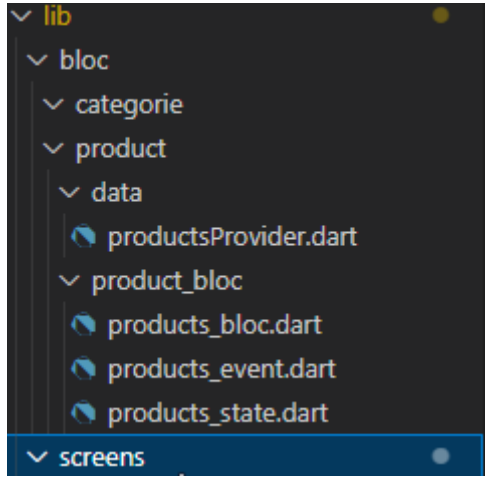
INSTALLATION

- On ajoute la ligne suivant dans le pubspec

```
cupertino_icons: ^1.0.2
curved_navigation_bar: ^1.0.3 #latest version
http: ^1.1.0
flutter_bloc: ^8.1.3
```

UTILISATION

- Bloc ce base sur les event(‘action’) et les states tout en assurant une comunication avec le backend (ou autre) de ce fait pour le bon deroulement on créer l’architecture suivante pour chaque bloc, (on prend comme exemple product bloc)



```dart
abstract class ProductsEvent {}

class GetProductsEvent extends ProductsEvent {}

class AddProductEvent extends ProductsEvent {}
```

En utilisant ces classes d'événements, vous pouvez définir et déclencher des actions spécifiques dans votre application. Dans le contexte d'un Bloc, ces événements seront traités par la logique métier du Bloc, permettant ainsi à votre application de réagir de manière appropriée aux différentes actions de l'utilisateur ou aux changements d'état. Par exemple, chaque événement pourrait être associé à une requête réseau, à une mise à jour de l'état de l'application, ou à d'autres opérations nécessaires pour maintenir la cohérence de votre application.

- Dans le fichier event on s'assure de mettre tout les evenement qu'on souhaite déclencher
- abstract class ProductsEvent {} :
  - C'est une classe abstraite nommée ProductsEvent.
  - Les classes abstraites sont souvent utilisées pour définir un ensemble de fonctionnalités communes sans les implémenter directement. Ici, elle sert de classe de base pour les événements liés aux produits.
- class GetProductsEvent extends ProductsEvent {} :
  - Cette classe représente un événement spécifique appelé "GetProducts".
  - Elle étend la classe abstraite ProductsEvent, ce qui signifie qu'elle hérite de toutes les fonctionnalités de la classe de base.
  - Cet événement pourrait être déclenché lorsque l'application doit récupérer la liste des produits.
- class AddProductEvent extends ProductsEvent {} :
  - Cette classe représente un autre événement appelé "AddProduct".
  - De même, elle étend également la classe abstraite ProductsEvent.
  - Cet événement pourrait être déclenché lorsque l'utilisateur souhaite ajouter un nouveau produit.

```dart
import 'package:first/bloc/product/data/productsProvider.dart';

abstract class ProductsState {}

class InitialState extends ProductsState {}

class SuccessProductsList extends ProductsState {
  List<Product> productsFromState;
  SuccessProductsList(this.productsFromState);
}

class LoadingState extends ProductsState {}
```

- Ici on liste les diferent state de notre applicatio, il est preferable de toujour créer un state « InitialState »
- class InitialState extends ProductsState {} :
  - Cette classe représente l'état initial de votre application.
  - Elle étend la classe abstraite ProductsState, ce qui signifie qu'elle hérite de toutes les fonctionnalités de la classe de base.
  - Cet état pourrait être utilisé pour indiquer que l'application est dans son état initial, avant tout chargement de données.
- class SuccessProductsList extends ProductsState { ... } :
  - Cette classe représente un état de succès où une liste de produits a été récupérée avec succès.
  - Elle contient une propriété productsFromState qui est une liste d'objets de type Product (peut-être définie dans productsProvider.dart).
  - Cet état pourrait être utilisé pour mettre à jour l'interface utilisateur avec la liste de produits récupérée.

lass LoadingState extends ProductsState {} :

- Cette classe représente un état de chargement.
- Elle indique que l'application est en train de récupérer ou de traiter des données.
- Cet état pourrait être utilisé pour afficher un indicateur de chargement à l'utilisateur pendant que les données sont en cours de récupération.

L'utilisation de classes pour représenter les états permet de créer une structure claire et modulaire dans votre code. Ces états seront généralement émis par votre Bloc en réponse à des événements spécifiques, et l'interface utilisateur réagira en conséquence à ces états. Par exemple, lorsqu'un événement "GetProductsEvent" est déclenché, le Bloc pourrait émettre un état "LoadingState" pendant que les données sont récupérées, puis passer à un état "SuccessProductsList" une fois que les données ont été récupérées avec succès.

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:http/http.dart';

class ProductProvider {
  Future<List<Product>> getProducts() async {
    print("getting products List.....");
    List<Product> products = [];
    var url = Uri.https("bahmedkamel.com", 'wp-json/wc/v3/products');
    Response response = await http.get(
      url,
      headers: {
        "Content-Type": "application/json",
        'Authorization':
            'Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE3MD
IzOTEwNDEsImV4cCI6MTcwMjM5NDY0MSwiZW1haWwiOiJhZG1pbkBiYWhtZWRrYW1lbC5jb
20iLCJpZCI6IjEiLCJzaXRlIjoiaHR0cHM6XC9cL3d3dy5iYWhtZWRrYW1lbC5jb20iLCJ1
c2VybmFtZSI6ImFkbWluIiwiaXNzIjoiaHR0cHM6XC9cL3d3dy5iYWhtZWRrYW1lbC5jb20
ifQ.FobKbISlzUxn8EqGzkeBnpXlCNCuhXq5mejB23MOU5o',
      },
      // encoding: Encoding.getByName("utf-8"),
    );
    if (response.statusCode == 200) {
      final body = json.decode(response.body) as List;
      body.forEach((json) {
        products.add(Product.fromJson(json));
      });
      return products;

      //print(json.decode(response.body));
      // Do the rest of job here
    } else {
      print(response.statusCode);

      return [];
    }
  }
}

class Product {
  late int id;
  late String name;
  late List images;
  late String description;
  late String price;

  Product.fromJson(Map<String, dynamic> json) {
    id = json['id'];
    name = json['name'];
    images = json['images'];
    description = json['description'];
    price = json['price'];
  }
}
```

- Ce fichier définit une classe ProductProvider qui gère la récupération des produits à partir d'une API, ainsi qu'une classe Product utilisée pour représenter les données des produits
- class ProductProvider :
  - Cette classe est responsable de la récupération des produits depuis une API.
  - La méthode getProducts envoie une requête HTTP GET à l'URL spécifiée avec les en-têtes nécessaires, puis traite la réponse.
  - En cas de succès (code de statut 200), elle convertit le corps de la réponse en une liste de produits en utilisant la classe Product.
- class Product :
  - Cette classe représente les données d'un produit.
  - Elle possède des propriétés telles que id, name, images, description, et price.
  - Le constructeur fromJson prend un objet Map (généralement issu d'un JSON) et initialise les propriétés de la classe.

```dart
return BlocBuilder<ProductsBloc, ProductsState>(
    builder: (context, state) {
      if (state is LoadingState) {
        return CircularProgressIndicator();
      }
      if (state is SuccessProductsList) {
        List<Product> hamouda = state.productsFromState;
        print(hamouda);
        if (hamouda.isEmpty) {
          print("awja");
          print(hamouda);
          print("awkhlass");
          WidgetsBinding.instance.addPostFrameCallback((_) {
            Navigator.pushNamed(context, "login");
          });
        }

        return GridView.builder(
          gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(
            maxCrossAxisExtent: 200,
            childAspectRatio: 0.6,
            crossAxisSpacing: 10,
            mainAxisSpacing: 10,
          ),
          physics: const NeverScrollableScrollPhysics(),
          itemCount: hamouda.length,
          shrinkWrap: true,
          itemBuilder: (BuildContext ctx, index) {
            return item(hamouda, index);
          },
        );
      }
      return InkWell(
          onTap: () {
            context.read<ProductsBloc>().add(GetProductsEvent());
          },
          child: Text("get Data"));
    },
  );
}
```

- Le widjet Products utilise BlocBuilder pour réagir aux changements d'état émis par le bloc ProductsBloc
- BlocBuilder<ProductsBloc, ProductsState> :
  - Ce widget BlocBuilder observe le bloc ProductsBloc et reconstruit son contenu chaque fois que l'état du bloc change. On l'utilise sur chaque widget qui interagi avec le bloc ProductsBloc
- Conditions basées sur l'état :
- Le builder vérifie l'état actuel émis par le bloc.
  - Si l'état est LoadingState, un indicateur de chargement (CircularProgressIndicator) est renvoyé.
  - Si l'état est SuccessProductsList, la liste de produits est extraite de l'état, et en fonction de sa disponibilité, soit un GridView avec les produits est renvoyé, soit l'utilisateur est redirigé vers la page de connexion si la liste est vide.
- Bouton "get Data" :
  - Si l'état n'est ni LoadingState ni SuccessProductsList, un bouton "get Data" est affiché. Lorsque l'utilisateur appuie sur ce bouton, un événement GetProductsEvent est ajouté au bloc, déclenchant ainsi le processus de récupération des données.