



# Trading Portfolio

**Maksim Dudarenka**  
**Jérémy Goffin**  
**B2 Cyber**

## Table

Table	2
Présentation de l'application :	2
Objectifs de l'application :	2
Notre Trello pour la suivis de notre projet :	2
Architecture des classes :	4
Différentes vues de l'application :	9

## Présentation de l'application :

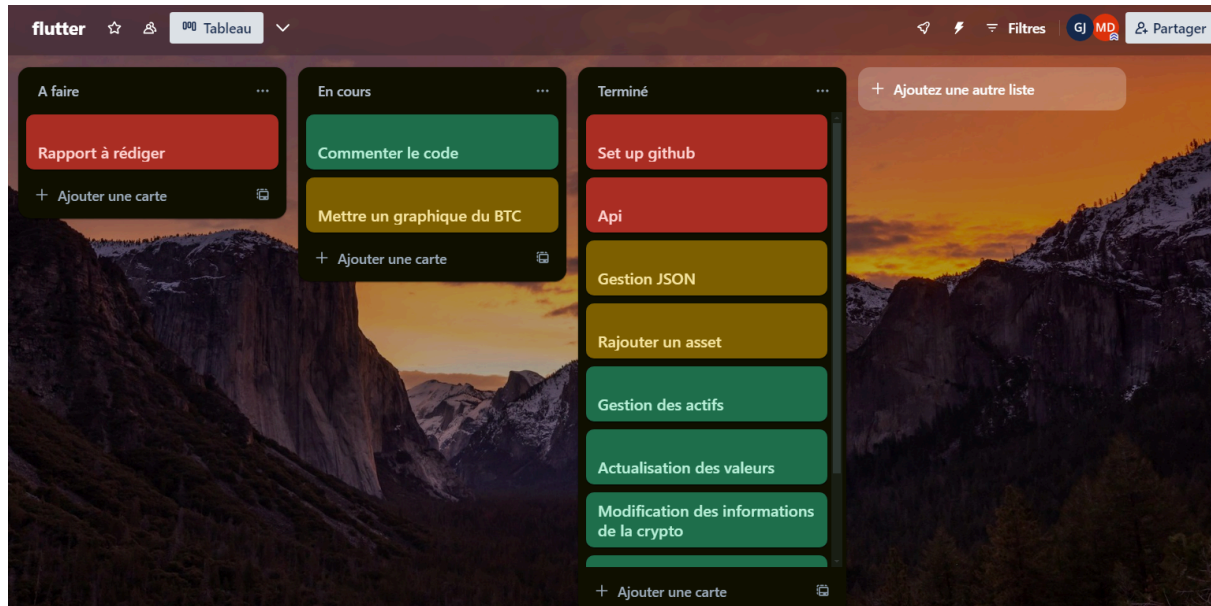
L'objectif de l'application Trading Portfolio est de permettre la création de plusieurs portefeuilles pour suivre les prix du marché des cryptomonnaies via l'API de CoinGecko. Les utilisateurs peuvent créer plusieurs portefeuilles pour diversifier leurs investissements et suivre leurs valeurs. Il leur suffit d'ajouter le nom, le symbole et la quantité de la crypto-monnaie détenue, la valeur affichée est donc la quantité \* la valeur récupérée via l'api.

## Objectifs de l'application :

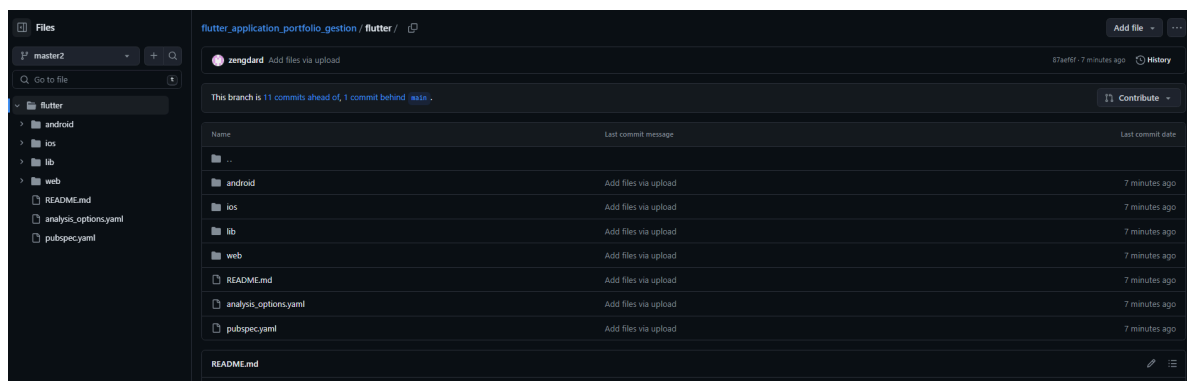
L'application a pour but de fournir les fonctionnalités suivantes :

- Permettre la création d'un ou plusieurs portefeuilles
- Ajouter des actifs (assets) à un portefeuille spécifique
- Organiser les actifs en groupes au sein d'un portefeuille
- Ajouter des cryptomonnaies aux actifs avec leurs noms, symboles et quantité détenus
- Mettre à jour automatiquement la valeur des actifs en rafraichissant
- Offrir la possibilité de modifier ou supprimer les informations d'un actif spécifique

## Notre Trello pour la suivis de notre projet :



En utilisant Trello pour la gestion des tâches, notre équipe a pu créer des tableaux pour chaque projet et diviser ces projets en tâches spécifiques sous forme de cartes. Chaque carte contient le nom d'une tâche avec une description détaillée de la tâche à réaliser. Nous avons assigné des couleurs aux tâches pour indiquer leur degré d'importance : le rouge pour les tâches importantes, l'orange pour les tâches intermédiaires et le vert pour les tâches simples qui n'ont pas d'impact sur l'avancée des autres tâches.



De plus, nous avons utilisé GitHub pour suivre le projet de notre côté, grâce aux fonctionnalités de commit et de push. Le commit permet d'enregistrer des modifications dans un dépôt local, tandis que le push permet d'envoyer les modifications vers la plateforme GitHub.

Grâce à cette approche, notre équipe a pu collaborer efficacement à distance, en sachant toujours qui travaillait sur quoi et où en étaient les différentes tâches. Trello nous a aidés à rester organisés et à suivre les progrès de nos projets, tandis que

GitHub nous a fourni les outils nécessaires pour développer, réviser et déployer notre code de manière transparente et décentralisée. Dans l'ensemble, l'utilisation conjointe de Trello et GitHub a grandement amélioré notre efficacité et notre productivité en tant qu'équipe de développement à distance.

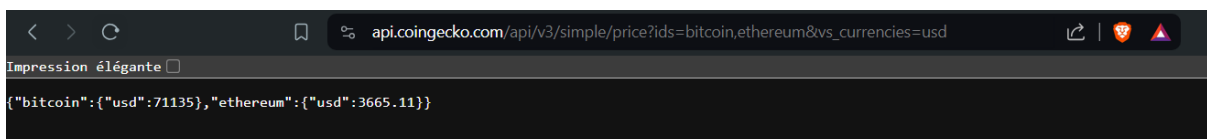
## Architecture des classes :

Ci-joint la présentation succincte des différentes classe :

1. HomePage et \_HomePageState : la page d'accueil et son état. Cette page contient une liste de portefeuilles et permet d'ajouter, de supprimer et de modifier des portefeuilles.
2. \_updateAssetPrices : une méthode pour mettre à jour les prix des actifs d'un portefeuille en récupérant les données de prix à partir d'une API.
3. \_loadPortfolios et \_savePortfolios : des méthodes pour charger et sauvegarder les données des portefeuilles à l'aide de la classe PortfolioStorage.
4. \_readPortfoliosJson : une méthode pour charger les données des portefeuilles à partir d'un fichier JSON.
5. TotalPortfolioWidget : un widget affichant la valeur totale de tous les portefeuilles.
6. AssetsPage : une page pour afficher et modifier les actifs d'un portefeuille spécifique (non fourni dans le code).
7. fetchPrices(List<String> ids) : une fonction asynchrone qui récupère les prix des actifs spécifiés dans la liste 'ids' en utilisant l'API Coingecko. Elle renvoie un objet Map<String, dynamic> contenant les prix en USD.
8. \_buildApiUrl(String apiVersion, String endpoint, Map<String, String> queryParameters) : une fonction qui construit une URL d'API à partir des paramètres donnés.
9. Portfolio : une classe représentant un portefeuille de crypto-monnaies. Elle contient un nom, une variation en 24h (non utilisée), une liste d'actifs et une valeur totale. La classe inclut des méthodes pour ajouter et supprimer des actifs, mettre à jour la liste des actifs, calculer la valeur totale et sérialiser/désérialiser le portefeuille au format JSON.
10. Asset : une classe représentant un actif (crypto-monnaie) dans le portefeuille. Elle contient un nom, un symbole, une valeur en unités et une valeur en dollars. La classe inclut des méthodes pour sérialiser/désérialiser l'actif au format JSON.

```
Future<Map<String, dynamic>> fetchPrices(List<String> ids) async {
  final String joinedIds = ids.join(',');
  final String url =
    'https://api.coingecko.com/api/v3/simple/price?ids=$joinedIds&vs_currencies=usd';
  print(url);
  try {
    final response = await http.get(Uri.parse(url));
    print('Response status: ${response.statusCode}');
    if (response.statusCode == 200) {
      print(response.body);
      return json.decode(response.body);
    } else {
      throw Exception(
        'Failed to load prices with status code: ${response.statusCode}');
    }
  } catch (e) {
    throw Exception('Failed to fetch prices: $e');
  }
}
```

La Data de Coingecko est récupérée par la fonction `fetchPrices`, c'est une méthode asynchrone qui prend une liste d'IDs de crypto-monnaies en entrée et renvoie un objet `Map<String, dynamic>` contenant leurs prix respectifs en USD. Pour ce faire, elle utilise l'API Coingecko en construisant une URL avec les IDs joints et en envoyant une requête GET. Si la réponse a le code de statut 200, le corps de la réponse est décodé en format JSON et renvoyé. En cas d'échec de la requête ou de réponse avec un code de statut différent de 200, une exception est levée avec un message d'erreur approprié. Cette fonction est utile pour récupérer les prix actuels des crypto-monnaies spécifiées dans la liste d'IDs et facilite la gestion des portefeuilles de crypto-monnaies. Notons que c'est une api publique sans nécessité de disposer d'une clé API.



The screenshot shows a web browser window with the address bar displaying the URL: `api.coingecko.com/api/v3/simple/price?ids=bitcoin,ethereum&vs_currencies=usd`. The page content shows the JSON response: `{"bitcoin":{"usd":71135},"ethereum":{"usd":3665.11}}`.

Voilà au format Json la réponse d'un GET.

```
37
38 class Portfolio {
39   final String name;
40   final double change24h;
41   List<Asset> assets;
42   final double value;
43
44   Portfolio(
45     {required this.name,
46     required this.change24h,
47     required this.assets,
48     required this.value});
49
50   void addAsset(Asset asset) {
51     assets.add(asset);
52   }
53
54   void removeAsset(Asset asset) {
55     assets.remove(asset);
56   }
57 }
```

La classe `Portfolio` représente un portefeuille de crypto-monnaies dans une application. Elle a quatre propriétés : un nom de type `String`, une variation de prix en 24h de type `double`, une liste d'actifs de type `List<Asset>`, et une valeur totale du

portefeuille de type double. Le nom est le titre du portefeuille, la variation de prix en 24h représente la change en pourcentage de la valeur du portefeuille sur les dernières 24 heures, la liste d'actifs contient les différentes crypto-monnaies détenues dans le portefeuille avec leurs quantités respectives, et la valeur totale représente la valeur combinée de tous les actifs dans le portefeuille. Le constructeur de la classe Portfolio prend ces quatre propriétés en tant qu'arguments requis.

```
06 class Asset {
07     final String name;
08     final String symbol;
09     double _dollars;
10     final double value;
11
12     Asset(
13         required this.name,
14         required this.symbol,
15         required this.value,
16         required double dollars,
17     ) : _dollars = dollars;
18
19     double get dollars => _dollars;
20
21     set dollars(double newValue) {
22         _dollars = newValue;
23     }
24 }
```

La classe Asset représente un actif, tel qu'une crypto-monnaie, dans une application de portefeuille. Elle a quatre propriétés : un nom de type String, un symbole de type String, une valeur en dollars de type double et une valeur en unités de l'actif de type double. Le nom et le symbole identifient l'actif, tandis que la valeur en dollars représente la valeur monétaire de l'actif en USD. La valeur en unités de l'actif représente la quantité détenue dans le portefeuille.

Le constructeur Asset prend quatre arguments requis : name, symbol, value et dollars. La valeur en dollars est stockée dans une variable privée \_dollars, qui est accessible via un getter et un setter. Le getter dollars renvoie la valeur de \_dollars, tandis que le setter permet de mettre à jour cette valeur.

```

class PortfolioStorage {
    static const _portfoliosKey = 'portfolios';

    static Future<void> savePortfolios(List<Portfolio> portfolios) async {
        final prefs = await SharedPreferences.getInstance();
        final jsonString = json.encode(portfolios.map((p) => p.toJson()).toList());
        print(
            "Saving portfolios: $jsonString");
        await prefs.setString(_portfoliosKey, jsonString);
    }

    static Future<List<Portfolio>> loadPortfolios() async {
        final prefs = await SharedPreferences.getInstance();
        final jsonString = prefs.getString(_portfoliosKey);
        print(
            "Loading portfolios: $jsonString");
        if (jsonString == null) {
            return [];
        }
        final jsonList = json.decode(jsonString) as List<dynamic>;
        return jsonList.map((json) => Portfolio.fromJson(json)).toList();
    }
}

```

La classe PortfolioStorage est responsable de la sauvegarde et du chargement des portefeuilles de crypto-monnaies dans le stockage local de l'application à l'aide du plugin SharedPreferences. Elle contient deux méthodes statiques : savePortfolios et loadPortfolios.

La méthode savePortfolios prend une liste de portefeuilles et les convertit en une chaîne JSON, puis stocke cette chaîne dans le stockage local en utilisant une clé spécifique (\_portfoliosKey). La méthode loadPortfolios récupère la chaîne JSON à partir du stockage local en utilisant la même clé, la décode en une liste d'objets Portfolio et renvoie cette liste. Les logs ont été ajoutés pour la sauvegarde et le chargement des portefeuilles afin de faciliter le débogage.

```

children: [
  btcPriceSpots.isNotEmpty
    ? Container(
      height: 200,
      child: LineChart(
        LineChartData(
          gridData: FLGridData(show: false),
          titlesData: FLTitlesData(show: false),
          borderData: FLBorderData(show: false),
          lineBarsData: [
            LineChartBarData(
              spots: btcPriceSpots,
              isCurved: true,
              barWidth: 5,
              isStrokeCapRound: true,
              dotData: FLDotData(show: false),
              belowBarData: BarAreaData(show: false),
              color: Colors.red,
            ), // LineChartBarData
          ],
        ),
      ),
    ],

```

La classe LineChartData est responsable de l'affichage du graphique pour les données de crypto-monnaies dans l'application. Elle utilise le plugin flutter\_chart pour implémenter des graphiques en fonction d'un tableau.

La classe contient plusieurs méthodes pour configurer et afficher les graphiques, notamment la méthode setData qui prend une liste de données de crypto-monnaies et met à jour le graphique en conséquence. La méthode botPriceSpot est utilisée pour récupérer les données sur Coingecko et les transformer pour la bonne concordance entre LineChartData et la réponse GET API .

```

class Portfolio {
  final String name;
  final double change24h;
  List<Asset> assets;
  final double value;

  Portfolio(
    {required this.name,
    required this.change24h,
    required this.assets,
    required this.value});
}

class Asset {
  final String name;
  final String symbol;
  double _dollars;
  final double value;

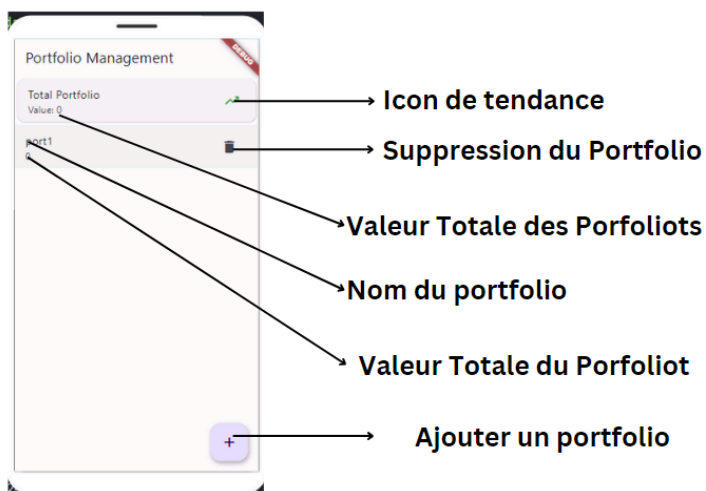
  Asset({
    required this.name,
    required this.symbol,
    required this.value,
    required double dollars,
  }) : _dollars = dollars;
}

```

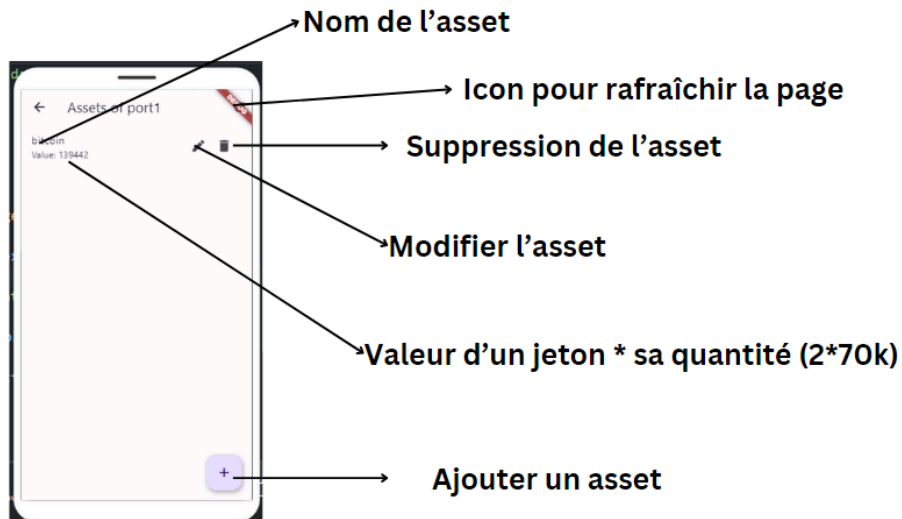


## Différentes vues de l'application :

### Vue 'Portfolio Management'



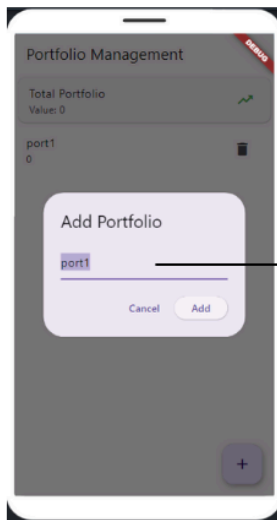
### Vue 'Asset of portfolio'



### Vue 'Add Asset'



### Vue 'Add Portfolio'



→ Champ de saisi