TP - Tintin

Flutter

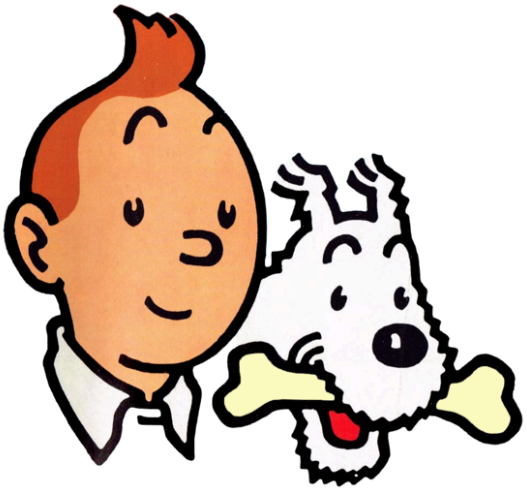


Table des matières

Alexandre Leroux (alex@shrp.dev) - 2024

Objectifs

- Modélisation de données avec une classe Dart,
- Mapping JSON / Dart (méthode factory),
- Future / async / await,
- FutureBuilder,
- ListView.builder,
- Interaction,
- Master / Details,
- Navigation basique,
- UI.

Consignes

- Consultez le ***Cookbook*** de Fluttter pour réaliser ce TP https://docs.flutter.dev/cookbook/,
- Pour améliorer la qualité de votre apprentissage il est recommandé de désactiver de votre *IDE* les outils d'IA générative tels que *Github Copilot, Gemini, ChatGPT*…
- Réalisez les étapes dans l'ordre.
- Consultez entièrement chaque étape de TP dans son intégralité avant de la réaliser.
- Il ne s'agit pas d'un exercice, prenez le temps nécessaire (documentation, expérimentation, correction, refactoring…). Contactez-moi en cas de difficulté majeure.
- A noter : les consignes sont données avec un objectif d'apprentissage progressif. Dans un cadre professionnel, cette application pourrait être réalisée de façon plus rapide et directe.

Etape 01 : Modèle

Créez un nouveau projet flutter vierge avec la commande

```
$ flutter create -e tintin
```

- Créez un dossier ./lib/models et un fichier ./lib/models/album.dart
- Programmez une classe Dart ***Album*** (./lib/models/album.dart) disposant des attributs suivants :
    - ***title*** (chaîne de caractères) - valeur requise,
    - ***numero*** (entier) - valeur requise,
    - ***year*** (entier) - valeur requise,
    - ***yearInColor*** (entier ou nul) - valeur optionnelle,
    - ***image*** (chaîne de caractères) - valeur requise,
    - ***resume*** (chaîne de caractères) - valeur requise.
- Programmez le constructeur de la classe Album.
- Programmez une méthode ***toString()*** retournant une chaîne de caractères permettant de consulter les valeurs d'un objet de type ***Album*** dans la console au moyen de la commande ***print()***.
- Programmez une méthode ***toJson()*** retournant une chaîne de caractères au format *JSON* (attention à la syntaxe).

Etape 02 : Assets

- Récupérez les images fournies (crédits : https://www.tintin.com/fr),
- Intégrez les images aux sources du projet et faites en sorte qu'elles soient intégrées au build de l'application (cf. fichier pubspec.yaml). Attention, les modifications dans le fichier pubspec.yaml ne prennent effet qu'après redémarrage de l'application (le *hot reloading / hot restarting* ne suffit pas).

Etape 03 : Service

- Créez une classe Dart ***AlbumService*** (./lib/services/album_service.dart),
- Au sein de cette classe, programmez une méthode statique et asynchrone nommée ***generateAlbums*** retournant une liste d'objets de type ***Album*** dont les valeurs seront générées de façon aléatoire. Pour ce faire, employez la méthode ***generate*** de l'objet ***List*** de Dart (cf. https://docs.flutter.dev/cookbook/lists/long-lists#1-create-a-data-source),
- La méthode ***generateAlbums*** simule le résultat fourni par une opération asynchrone de type I/O (ex: appel d'API, interaction avec le système de fichiers…).
- Saisissez manuellement une dizaine d'objets de type ***Album*** avec des valeurs cohérentes.

Etape 04 : Master

- Créez une classe de widget sans état (./lib/screens/albums_master.dart),
- Cette classe fera office d'écran, ajoutez la structure de widgets de base nécessaire,
- Ajoutez le titre ***"Albums"***,
- En employant un widget de type ***FutureBuilder***, faites appel à la méthode ***generateAlbums*** de la classe ***AlbumService*** et générez dynamiquement une liste verticale scrollable à l'aide de ***ListView.builder*** (cf. https://docs.flutter.dev/cookbook/lists/long-lists). La liste affichera le titre de chaque objet de type ***Album*** retourné.

Etape 05 : Preview

- Créez un widget sans état nommé ***AlbumPreview*** (./lib/widgets/album_preview.dart) prenant en paramètre de son constructeur une valeur (requise) ***album*** de type ***Album***.
- Dans ***AlbumPreview***, faites en sorte que la méthode build retourne un widget de type ***ListTile*** qui affichera le titre de l'album concerné.
- Affichez l'image de l'album concerné sous forme de miniature. Si l'image de l'album n'est pas disponible, affichez une icône de votre choix au moyen d'un widget ***Icon***.

Etape 06 : Details

- Créez une classe de widget sans état ***AlbumDetails*** (./lib/screens/album_details.dart).
- Cette classe fera office d'écran, ajoutez la structure de widgets de base nécessaire,
- Affichez dynamiquement le titre, l'image (si disponible), la date de parution et le numéro de l'album concerné.
- Faites en sorte qu'au clic sur un widget AlbumPreview, l'utilisateur soit redirigé vers l'écran ***AlbumDetails***, généré dynamiquement (cf. https://docs.flutter.dev/cookbook/navigation/navigation-basics).

Etape 07 : JSON

- Récupérez le fichier JSON fourni et ajoutez-le dans ***./data/albums.json***,
- Intégrez le fichier ***albums.json*** au build de l'application (cf. étape précédente),
- Programmez une méthode de type factory ***fromJson*** permettant de générer un objet de type ***Album*** à partir de données au format JSON (cf. https://docs.flutter.dev/cookbook/networking/background-parsing).
- Depuis la classe ***AlbumService***, ajoutez une méthode ***fetchAlbums*** afin d'importer dynamiquement le fichier JSON (cf. https://www.geeksforgeeks.org/flutter-load-json-assets/),
- Employez les données JSON importées afin de générer autant d'objets de type ***Album*** qu'il y a d'objets dans le tableau de données JSON et construisez dynamiquement la liste présente dans le widget .
- Adaptez l'écran pour que la liste soit à présent alimentée par la méthode ***fetchAlbums***.

Etape 08 : Liste de lecture

- A partir de l'écran ***AlbumDetails***, permettez à l'utilisateur de cliquer sur un élément UI de votre choix (ex: ***FloatingActionButton***), afin d'ajouter l'album courant à une liste de lecture (une liste d'objets de type ***Album***).

- La liste de lecture n'a pas à être persistée. Elle est réinitialisée à chaque lancement de l'application (liste vide par défaut).

- Si l'album courant est présent dans la liste de lecture de l'utilisateur, ce dernier doit pouvoir le retirer en cliquant à nouveau sur l'élément UI préalablement sélectionné (système de type *toggle*).

- Faites en sorte qu'au retour sur l'écran ***AlbumMaster***, les albums ajoutés à la liste de lecture soient mis en évidence (ex: couleur d'arrière-plan, icône, typographie… du widget ***AlbumPreview***) :

    - ***AlbumsMaster*** étant l'écran principal, la liste de lecture doit être gérée et centralisée dans ce widget. Convertissez ***AlbumsMaster*** en widget *Stateful*, créez une variable d'état nommée ***_readingList*** ne pouvant contenir que des objets de type ***Album***.

    - Un changement d'état dans ***AlbumsMaster*** permettra de rafraîchir le widget ***ListView***.

    - L'information d'ajout ou de suppression d'un album à la liste de lecture doit transiter du widget ***AlbumDetails*** vers le widget , en passant par le widget ***AlbumPreview*** qui a le rôle de médiateur entre les 2 widgets.

      La communication *"ascendante"* entre widgets *Enfant* vers *Parent* implique que le widget *Parent* communique une référence vers une de ses méthodes, en tant qu'argument du constructeur de son widget enfant.

      En disposant d'une référence vers une méthode de son parent, le widget enfant peut l'exécuter et lui passer des valeurs en paramètres.

      Aussi, le widget ***AlbumDetails*** doit communiquer avec le widget **.** Le widget ***AlbumPreview*** ayant un rôle intermédiaire, la communication entre ***AlbumDetails*** et sera indirecte.

Etape 09 : Provider

- Cette étape consiste en un refactoring de l'étape précédente basée sur ***Provider***.
- Afin de faciliter la communication des données entre widgets, mettez en place un store de données avec le package ***Provider***.

  Provider permet de découpler les widgets ***AlbumsMaster, AlbumPreview et AlbumDetails***.

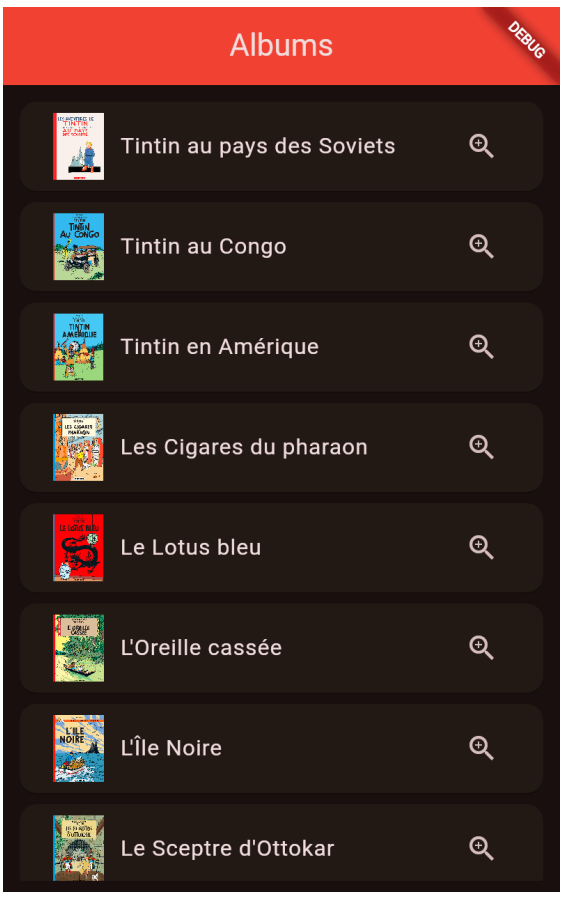
  Provider fonctionne sur le principe ***Fournisseur*** / ***Consommateur*** et s'inspire du ***Design Pattern Observer***. Les données sont centralisées par le ***Provider*** (*fournisseur*) et directement accessibles depuis l'ensemble des widgets (*consommateurs*) contenus dans l'application.
- Consultez l'exemple suivant : https://docs.flutter.dev/development/data-and-backend/state-mgmt/simple
- Créez une classe *Dart* ***ReadingListProvider (lib/providers/reading_list_provider.dart)*** héritant de ***ChangeNotifier***.
- Procédez à l'adaptation de la méthode ***main*** du fichier ***./lib/main.dart*** afin d'intégrer un widget ***ChangeNotifierProvider*** en tant que *"wrappeur"* de l'application ***MainApp*** (https://docs.flutter.dev/data-and-backend/state-mgmt/simple#changenotifierprovider).
- Dans la classe ***ReadingListProvider***, mettez en place les méthodes permettant d'accéder en lecture et en écriture aux données partagées par le ***Provider*** (***addAlbum, removeAlbum, getAlbumByNumero***… ainsi qu'un *getter* ***albums*** de la liste ***_albums***).

  Pour chaque méthode effectuant une modification des données partagées (ajout, suppression) exécutez la méthode ***notifyListeners*** héritée de ***ChangeNotifier*** afin d'informer les widgets *"consommateurs" (wrappés* par un widget ***Consumer***).
  Ces widgets se mettront automatiquement à jour à la réception des notifications émises par le ***Provider***.
- A noter : avec l'intégration de Provider, le widget ***AlbumsMaster*** n'a plus la nécessité d'être de type ***Stateful***. Les références entre widgets peuvent également être supprimées.

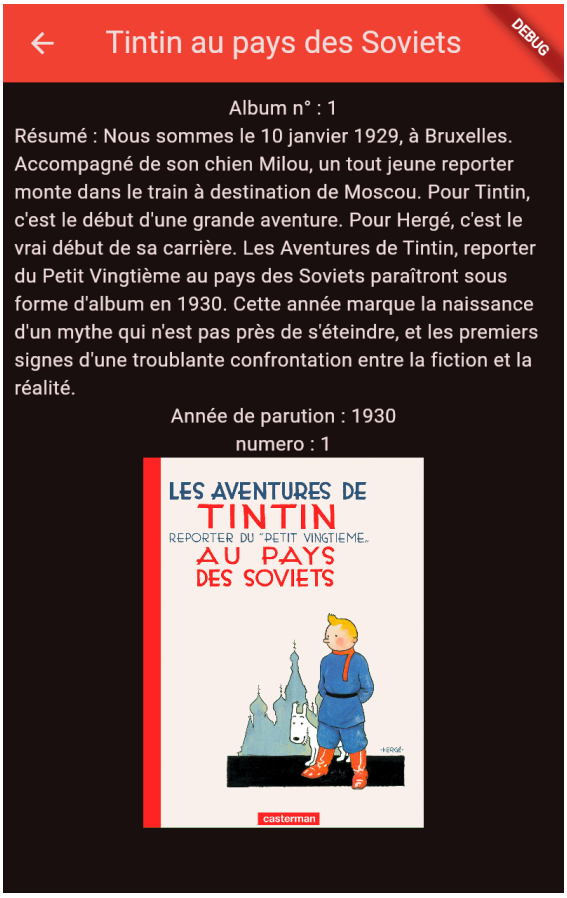
Etape 10 : UI

- Affichez l'application en mode sombre,
- Appliquez un thème personnalisé : couleurs, typographies… (https://docs.flutter.dev/cookbook/design/themes).

Aperçu



Master



Details