Flutter

Fiche 5 : API & JSON

Table des matières

1 Objectifs de la fiche

ID	Objectifs
F08	Consommation d'une RESTful API
F09	Parsing de JSON

2 Concepts

2.1 Introduction

Pour commencer le tutoriel, créez un nouveau projet (New Flutter Project) nommé *tuto5* dans votre repository de cours.

Nous allons développer une application Flutter qui utilisera l'API REST suivante : https://sebstreb.github.io/flutter-fiche-5/films-api.

Pour ce tutoriel, nous avons besoin de deux dépendances à rajouter dans votre projet. La dépendance **http** permet de faire des requêtes à des API. Vous pouvez l'ajouter à votre projet avec la commande **flutter pub add http**. La dépendance **url_launcher** permet d'ouvrir un lien dans un navigateur. Vous pouvez l'ajouter à votre projet avec la commande **flutter pub add url_launcher**. L'installation de ce package peut afficher une erreur sur windows, mais elle ne devrait pas poser de problèmes.

Sur certaines plateformes, l'accès à internet est bloqué par défaut pour les applications. Il est alors nécessaire de modifier les permissions de l'application pour indiquer qu'elle a besoin de l'accès à la connexion internet de l'appareil de l'utilisateur. C'est par exemple le cas pour la plateforme Android. Si vous souhaitez lancer l'application sur un téléphone ou l'émulateur android, vous aurez besoin de rajouter les permissions suivantes dans le fichier *AndroidManifest.xml* présent dans le dossier *android/app/src/main*, avant le tag application

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

Vous pouvez trouver plus d'informations à ce sujet et pour les autres plateformes à ce lien : https://docs.flutter.dev/development/data-and-backend/networking.

A travers les différentes étapes de ce tutoriel, nous suivrons l'architecture MVVM vue durant la fiche précédente. Nous n'utiliserons cependant pas de couche *View Model*. En effet, l'état de l'application créée est toujours confiné à un seul widget. En essayant de suivre les bonnes pratiques pour choisir à quel niveau placer le *ChangeNotifier* et le *Consumer*, ils se retrouveraient à la même place puisqu'on on accède à l'état de l'application à un seul endroit. Cela montre bien que dans ce cas, un *View Model* n'est pas nécessaire et que l'on peut se contenter d'utiliser une variable d'état dans un stateful widget.

2.2 Step 1 : Requête HTTP

L'objectif de cette première étape est de faire une première requête à notre API, et d'afficher son résultat. Créez un fichier *home_screen.dart* dans un dossier *views* et copiez-y le code suivant :

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

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

  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  var message = "Click on the button to launch the request.";

  Future<void> _initFilm() async {
    const url = "https://sebstreb.github.io/flutter-fiche-5/films-api/1";
    try {
      setState(() => message = "Loading, please wait…"); // Uncompleted
      var response = await http.get(Uri.parse(url));
      setState(() => message = response.body); // Completed with a value
    } catch (error) {
      setState(() => message = error.toString()); //Completed with an error
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: const Text("Tutoriel 5"),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            Expanded(child: Center(child: Text(message))),
            ElevatedButton(
              onPressed: _initFilm,
              child: const Text("Fetch movie n°1"),
            ),
          ],
        ),
      ),
    );
  }
}
```

Ce widget affiche un texte et un bouton. Le texte affiche le contenu d'une variable d'état **result**. Lorsque l'on appuie sur le bouton, l'application lance une requête vers le serveur. Cette requête, *GET <server>/1*, récupère le film ayant l'ID n°1. L'application modifie ensuite l'état du widget pour l'afficher. La requête est lancée avec la méthode **http.get** du plugin http. Il existe également des méthodes http.post, http.delete, … qui ne seront pas vues dans ce tutoriel.

Cette méthode est asynchrone. Comme en javascript, on peut lancer une méthode asynchrone et attendre qu'elle se termine avec le mot clé **await**, à condition de se trouver dans une fonction **async**. Ces fonctions retournent des **Future** pour encapsuler leur résultat, de la même façon que javascript utilise des *Promise.*

Une Future représente l'état d'une méthode asynchrone. Avec la syntaxe async/await, on peut manipuler les Futures comme s'il s'agissait de code synchrone. Tant que une méthode asynchrone appelée avec *await* ne s'est pas terminée, la Future qu'elle renvoie est dans l'état ***Uncompleted*** et la fonction *async* où l'on y fait appel est interrompue. Quand elle se termine correctement, la Future est dans l'état ***Completed with a value*** et on peut récupérer sa valeur de retour avec une simple assignation. Si une erreur se produit, la Future est dans l'état **Completed with an error** et on peut intercepter cette erreur avec un try/catch.

Vous pouvez obtenir plus d'informations sur le développement asynchrone en dart à ce lien : https://dart.dev/codelabs/async-await.

Faites appel au *HomeScreen* dans le fichier *main.dart* et lancez l'application pour la tester.

[Commit "T05.1 Requête HTTP"]

2.3 Step 2 : Parsing de JSON

Nous allons maintenant transformer le JSON reçu depuis le serveur en un objet utilisable dans l'application. Créez un fichier *film.dart* dans un dossier *models*, et copiez-y le code suivant :

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

class Film {
  static const baseUrl = "https://sebstreb.github.io/flutter-fiche-5/films-
api";

  final int id;
  final String title;
  final String director;
  final int duration;
  final String link;

  const Film(this.id, this.title, this.director, this.duration, this.link);

  Film.fromJson(Map<String, dynamic> jsonObj)
      : this(
          jsonObj["id"],
          jsonObj["title"],
          jsonObj["director"],
          jsonObj["duration"],
          jsonObj["link"],
        );

  @override
  String toString() =>
      'Film: $title, directed by $director, $duration min, $link';

  static Future<Film> fetchFilm(int id) async {
    var response = await http.get(Uri.parse("$baseUrl/$id"));

    if (response.statusCode != 200) {
```

```
      throw Exception("Error ${response.statusCode} fetching movie");
    }

    return Film.fromJson(jsonDecode(response.body));
  }
}
```

Cette classe représente un objet *Film*. Elle possède une méthode statique et asynchrone *fetchFilm*. Cette méthode fait appel à l'API comme vu précédemment. Elle utilise la fonction **jsonDecode** du package *dart:convert* pour parser la réponse HTTP et en récupérer un *Map*.

Le film est ensuite créé sur base de ce *Map* grâce au constructeur *Film.fromJson*, et est retourné par la méthode *fetchFilm*. Comme cette fonction est asynchrone, son type de retour est **Future<Film>**.

Créez un widget *FilmRow* dans un fichier *film_row.dart* du dossier *views* et copiez-y le code suivant :

```
import 'package:flutter/material.dart';
import 'package:url_launcher/url_launcher.dart';

import '../models/film.dart';

class FilmRow extends StatelessWidget {
  final Film film;

  const FilmRow({super.key, required this.film});

  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: Text(film.title),
      subtitle: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          Text("Directed by ${film.director} - ${film.duration} minutes"),
          InkWell(
            onTap: () => launchUrl(Uri.parse(film.link)),
            child: Text(film.link),
          )
        ],
      ),
    );
  }
}
```

Ce widget affiche les informations d'un film au sein d'un *ListTile*. Il utilise un widget **InkWell** pour créer un lien cliquable. Lorsque l'on clique sur son enfant, un *Text* ici, on fait appel à la méthode **launchUrl** du package *url_launcher* pour ouvrir le lien.

Remplacez ensuite la classe *_HomeScreenState* dans le fichier *home_screen.dart* :

```
class _HomeScreenState extends State<HomeScreen> {
  var message = " Loading, please wait…";
  Film? film;

  Future<void> _initFilm() async {
    try {
      var response = await Film.fetchFilm(2);
      setState(() => film = response);
    } catch (error) {
      setState(() => message = error.toString());
    }
  }

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: const Text("Tutoriel 5"),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: film == null
            ? Column(children: [Center(child: Text(message))])
            : FilmRow(film: film!),
      ),
    );
  }
}
```

Le layout du widget a changé. Il n'y a plus de bouton permettant de lancer la requête http. Au lancement de l'application quand la future est dans l'état *Uncompleted*, l'interface affiche un texte de chargement. Ensuite si une erreur arrive et la future est dans l'état *Completed with an error*, le texte est modifié pour afficher cette erreur. Autrement quand la future arrive à l'état *Completed with a value*, l'interface affiche à la place le widget *FilmRow* avec le film qui a été récupéré.

Pour effectuer la connexion à l'API dès le chargement du widget, nous faisons appel à *_initFilm* dans la méthode **initState**. Cette méthode est appelée automatiquement par le framework à la création de la classe *State* d'un *StatefulWidget*. Elle n'est donc appelée qu'une seule fois à travers le cycle de vie du widget, et n'est pas rappelée lorsque le widget est rebuild.

[Commit "T05.2 Parsing JSON"]

2.4 Step 3 : Isolate

Pour terminer ce tutoriel, nous allons récupérer la liste de tous les films au lieu de juste un film. Cela correspond cette fois-ci à la requête *GET <server>/*. Rajoutez la méthode suivante dans la classe *Film* du fichier *models/film.dart*.

```
static Future<List<Film>> fetchFilms() async {
  var response = await http.get(Uri.parse("$baseUrl/"));

  if (response.statusCode != 200) {
    throw Exception("Error ${response.statusCode} fetching movies");
  }

  return compute((input) {
    final jsonList = jsonDecode(input);
    return jsonList.map<Film>((jsonObj) =>
Film.fromJson(jsonObj)).toList();
  }, response.body);
}
```

Comme la précédente, cette méthode statique asynchrone du modèle effectue la requête vers l'API. Cependant, cette méthode utilise cette fois la fonction **compute** pour traiter la réponse HTTP. Cette fonction prend comme arguments une callback et les arguments à envoyer cette callback. Elle exécute la callback dans un nouvel **Isolate**.

Un *Isolate* en dart est un thread d'exécution qui s'exécute de manière isolée des autres threads (contrairement au C par exemple ou les threads partagent une partie de la mémoire). Par défaut, les opérations en flutter s'exécutent sur le *main Isolate*. Cela implique que si une opération prend trop de temps, elle risque de coincer la gestion de l'interface qui s'exécute sur le *main Isolate* également. Et du coup l'application semblera ne plus répondre correctement aux inputs de l'utilisateur.

Dans ce cas-ci, la callback décode le corps JSON de la réponse HTTP reçue grâce à *jsonDecode*. Puis elle transforme chaque élément de cette liste JSON en *Film* en utilisant la méthode *map*. Sans Isolate, cette fonction risque de prendre beaucoup de temps s'il y a beaucoup de films à traiter. C'est pourquoi on utilise la fonction *compute* pour exécuter cette callback dans un nouvel *Isolate* et garder le *main Isolate* disponible pour la gestion de l'interface.

Vous pouvez obtenir plus d'informations sur les Isolate dans la vidéo suivante : https://www.youtube.com/watch?v=vl_AaCgudcY

Remplacez cette fois-ci la classe *_HomeScreenState* par le code suivant :

```
class _HomeScreenState extends State<HomeScreen> {
  var message = "Loading…";
  final films = <Film>[];

  Future<void> _initFilms() async {
    try {
      var response = await Film.fetchFilms();
      setState(() {
        if (response.isEmpty) message = "No films found";
        films.addAll(response);
      });
    } catch (error) {
      setState(() => message = error.toString());
    }
  }
```

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: const Text("Tutoriel 5"),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: films.isEmpty
            ? Column(children: [Center(child: Text(message))])
            : ListView.separated(
                itemCount: films.length,
                itemBuilder: (context, index) => FilmRow(film:
films[index]),
                separatorBuilder: (context, index) => const Divider(),
              ),
      ),
    );
  }
}
```

Par rapport au step précédent, l'état est devenu une liste de films initialement vide. À l'initialisation de la classe State, le widget fait appel à la méthode que l'on vient de créer et rajoute à la liste tous les films récupérés via la requête.

[Commit "T05.3 Isolate"]

3 Exercice

3.1 Introduction

Qui ne connait pas le légendaire Studio Ghibli ? Rares sont celles et ceux n'ayant jamais entendu parler d'Anime créés par l'illustre Miyazaki. Nous allons créer une application présentant les animes de Studio Ghibli, sur base de la Studio Ghibli API.

Veuillez créer un nouveau projet (New Flutter Project) nommé *ex5* dans votre repository de cours.

3.2 Consommation d'une RESTful API

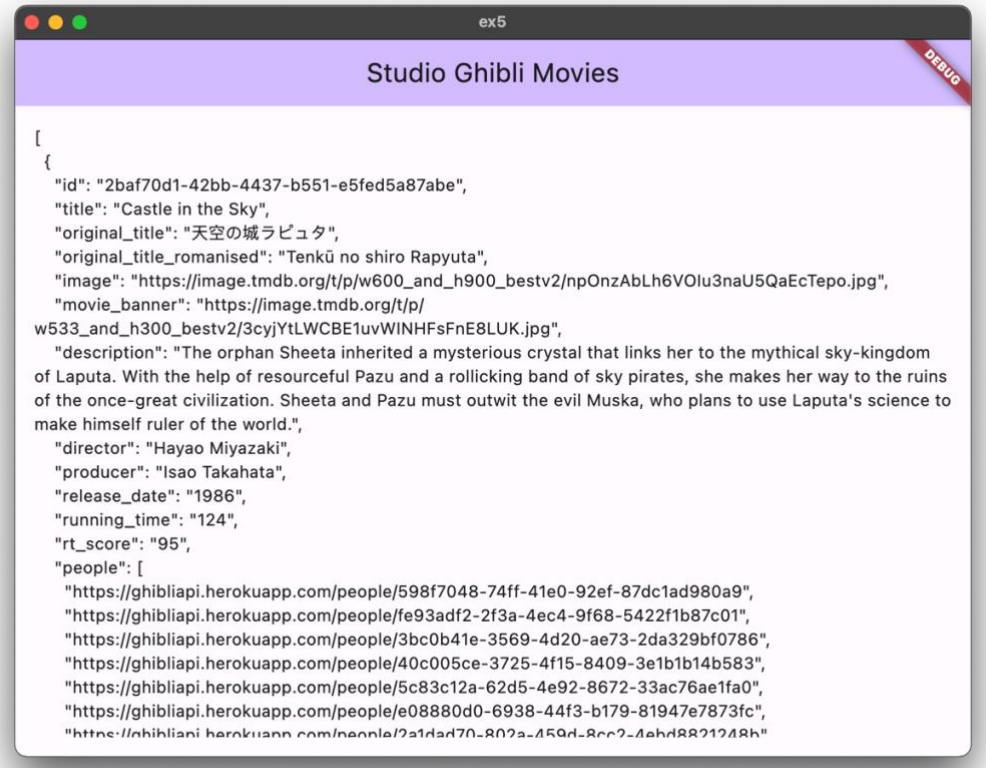
Veuillez afficher le contenu de la réponse à la requête vers l'API. L'URI pour obtenir les informations sur les films du Studio Ghibli est la suivante : https://sebstreb.github.io/flutter-fiche-5/ghibli-films

Lancez cette requête à l'initialisation de votre application. Affichez l'état de la requête au centre de l'écran, avec un texte décrivant si la requête n'est pas encore terminée, si elle a eu une erreur, ou en affichant le contenu de la réponse si elle est terminée.

Pour avoir l'occasion de visualiser ce temps de chargement, même si vous avez une bonne connexion, utilisez Future.delayed pour attendre 3 secondes avant de lancer la requête.

Si vous avez des difficultés à afficher l'entièreté de la réponse de l'API à cause de problèmes d'overflow, vous pouvez utiliser un *SingleChildScrollView* qui permet de faire scroller un widget unique.

A cette étape, votre application pourra ressembler à :



⚡ **[commit avec message : F08.1 Consommation d'API]** ⚡

3.3 Parsing de JSON

Créez maintenant un modèle pour représenter les données d'un film. Parmi les attributs des objets JSON reçus, vous ne devez récupérer et traiter que les suivants : id, title, image, description, release_date, director, running_time et rt_score. Créez également une méthode toString qui représente ces attributs de manière lisible.

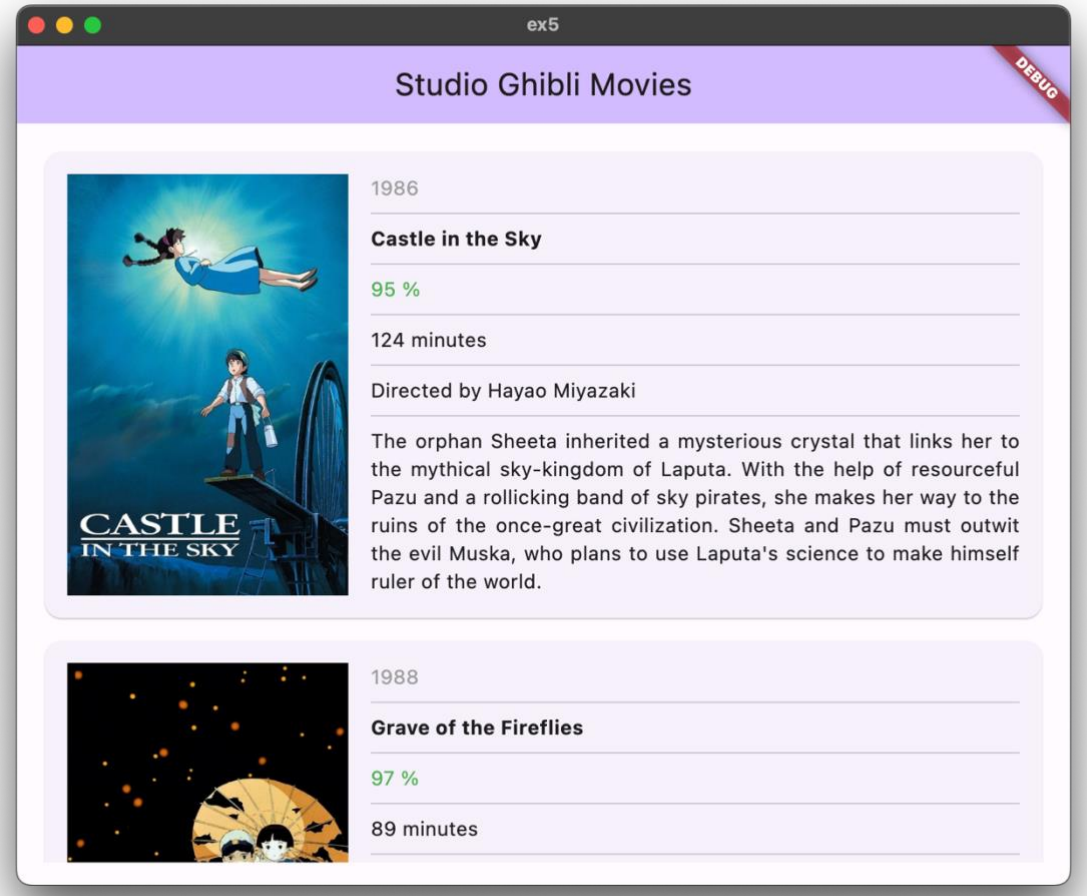
Ajoutez à ce modèle une méthode pour récupérer la liste des films à partir de l'API. Cette méthode devra utiliser un Isolate pour que l'application reste fluide si le nombre de films augmente.

Faites appel à cette méthode au sein de votre application. Celle-ci doit exécuter la quête une seule fois à la création du widget. Affichez une liste de widget pour chaque film. À cette étape, vous ne devez afficher qu'un simple texte présentant le film en tant que string pour chaque élément de la liste.

⚡ **[commit avec message : F09.1 Parsing de JSON]** ⚡

3.4 Affichage des films

Créez maintenant un widget pour afficher un film avec un layout soigné. Vous devez afficher toutes les informations que vous avez enregistrées d'un film, sauf l'id. Soyez imaginatifs pour créer l'affichage qui vous satisfait. Cela pourra donner par exemple :



Pour l'image, vous pouvez utiliser un widget *Image.network* qui affiche un image à une URI en argument. Attention sur la plateforme web, les images ne chargent pas correctement en utilisant le moteur de rendu par défaut. Pour changer de moteur de rendu et régler ce problème, il faut lancer l'application avec la commande suivante :

flutter run -d chrome --web-renderer html

Modifiez votre application pour qu'elle fasse appel à ce nouveau widget lors de l'affichage des films récupérés lors de l'étape précédente.

⚡ **[commit avec message : F09.2 Affichage des films]** ⚡

3.5 Challenge optionnel

Travailler l'aspect visuel vous intéresse ? Découvrez la documentation de Flutter afin de rendre votre application tant responsive que adaptive : Creating responsive and adaptive apps.

Retravaillez l'affichage de votre app pour qu'elle paraisse bien en fonction de la taille de l'écran et de son orientation. Veuillez afficher :

- Toutes les informations si la largeur est plus grande que 960px
- Toutes les informations sauf l'image si la largeur est entre 480px et 960px
- Seulement le titre, l'année et le score si la largeur est inférieure à 480px

⚡ **[commit avec message : FX.1 Challenge]** ⚡