Flutter

Fiche 4 : Navigation & Provider

Table des matières

1 Objectifs de la fiche

ID	Objectifs
F06	Navigation entre écrans
F07	Gestion d'état avec Provider

2 Concepts

2.1 Introduction

Pour commencer le tutoriel, créez un nouveau projet (New Flutter Project) nommé *tuto4* dans votre repository de cours.

2.2 Navigation entre écrans

2.2.1 Installation du package & mise en place des écrans

Il existe plusieurs façons de gérer un router et la navigation avec Flutter. Nous vous proposons ici la façon la plus moderne, qui offre un maximum de possibilités, en utilisant une navigation déclarative plutôt qu'impérative.
Flutter offre le package **go_router** pour naviguer au sein d'une application.

Veuillez installer la dépendance : **flutter pub add go_router**

Vous allez ensuite créer deux nouveaux écrans.
Veuillez créer le fichier **first_screen.dart** :

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

class FirstScreen extends StatelessWidget {
  final int nbClicks;

  const FirstScreen({super.key, this.nbClicks = 0});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("First screen"),
        backgroundColor: Theme.of(context).colorScheme.primaryContainer,
      ),
      body: Center(
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            const Text("Hello from first screen."),
            const SizedBox(height: 16),
            Text("There were $nbClicks clicks in the second page."),
            const SizedBox(height: 16),
            ElevatedButton(
              onPressed: () {}, // Here we will go to the second screen
              child: const Text("Go to second screen"),
            ),
```

```
          ],
        ),
      ),
    );
  }
}
```

Veuillez créer le fichier **second_screen.dart** :

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

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

  @override
  State<SecondScreen> createState() => _FirstScreenState();
}

class _FirstScreenState extends State<SecondScreen> {
  var nbClicks = 0;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Second screen"),
        backgroundColor: Theme.of(context).colorScheme.primaryContainer,
        /* if we don't use a child route, we would have to create our own
back button
        leading: IconButton(
            icon: const Icon(Icons.arrow_back),
            // Here we would come back to the first screen passing data
            onPressed: () {}),*/
      ),
      body: Center(
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            const Text("Hello from second screen."),
            const SizedBox(height: 16),
            Text("There were $nbClicks clicks."),
            const SizedBox(height: 16),
            ElevatedButton(
              onPressed: () => setState(() => nbClicks++),
              child: const Text("click me"),
            ),
          ],
        ),
      ),
    );
  }
}
```

2.2.2 Configuration des routes

Nous allons ensuite faire en sorte de configurer deux routes dans **main.dart** à l'aide de **GoRouter** et une route initiale qui affiche **FirstScreen** :

```dart
import 'package:flutter/material.dart';
import 'package:go_router/go_router.dart';
import 'second_screen.dart';
import 'first_screen.dart';

final GoRouter _router = GoRouter(
  initialLocation: '/',
  routes: [
    GoRoute(
      path: '/',
      builder: (context, state) => const FirstScreen(),
      routes: [
        GoRoute(
          path: 'secondscreen',
          builder: (context, state) => const SecondScreen(),
        ),
      ],
    ),
  ],
);

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp.router(
      routerConfig: _router,
      title: 'Flutter Demo',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
        useMaterial3: true,
      ),
    );
  }
}
```

On passe la configuration du router à **MaterialApp.router** via l'argument **routerConfig**.

Ici, nous avons configuré la route "**secondscreen**" comme enfant de la route "**/**". Une route « enfant » permet d'afficher un écran au-dessus du précédent écran et offre directement un bouton de retour en arrière lorsque la route est utilisée !

NB : Si nous avions défini deux routes au même niveau, sans route « enfant », nous aurions plus tard des soucis quand nous passerons d'un écran à un autre : nous n'aurions pas les bonnes animations lors du retour d'un écran, nous devrions nous même créer notre bouton de retour en arrière et nous pourrions difficilement utiliser la fonction intégrée de retour en arrière d'iOS et d'Android (lors d'un slide de la gauche vers la droite). Ainsi, ce type de configuration est à éviter si l'on revient toujours à l'écran d'accueil de l'application :

```dart
final GoRouter _router = GoRouter(
  initialLocation: '/firstscreen',
```

```
  routes: [
    GoRoute(
      path: '/firstscreen',
      builder: (context, state) => const FirstScreen(),
    ),
    GoRoute(
      path: '/secondscreen',
      builder: (context, state) => const SecondScreen(),
    ),
  ],
);
```

2.2.3 Navigation simple entre deux écrans

Nous allons maintenant voir comment naviguer vers le deuxième écran lorsqu'on clique sur le **ElevatedButton** « Go to second screen » à l'aide de la méthode **context.go().**
Veuillez mettre à jour le **ElevatedButton** de **first_screen.dart** :

```
ElevatedButton(
  onPressed: () => context.go("/secondscreen"),
  child: const Text("Go to second screen"),
),
```

Veuillez exécuter l'application pour voir que la navigation fonctionne, qu'il est possible de voyager vers le deuxième écran, ainsi que de revenir vers le 1er écran à l'aide du bouton « back » automatiquement créé pour vous.
[commit avec message : T04.1 Navigation simple]

2.2.4 Passage d'information lors de la navigation avec extra

Maintenant, nous allons voir comment naviguer du deuxième écran vers le 1er en passant une valeur, le compteur ici.

Jusqu'ici, le retour vers le 1er écran se faisait de manière automatique. Mais pour renvoyer des données, nous ne pouvons plus laisser le framework gérer un « pop automatique » de l'écran. Nous allons utiliser un nouveau widget pour gérer le pop de l'écran lors d'un « system back gesture ». Veuillez emballer le **Scaffold** (**Wrap with widget…**) dans un **PopScope** et ajoutez ce code-ci au **PopScope** :

```
return PopScope(
  canPop: false,
  onPopInvoked: (didPop) {
    if (!didPop) {
      context.go("/", extra: nbClicks);
    }
  },
  child: Scaffold( // suite de votre code
```

La propriété **extra** permet de passer des données qui seront considérées de type **Object**.
Le PopScope est un peu compliqué à gérer :

- On configure **canPop** à **false** pour qu'un « system back gesture » ne « pop » pas la route.
- On gère le cas que **onPopInvoked** est appelé deux fois :

- quand il y a un « system back gesture » (**didPop** est à **false**)
- quand la navigation est terminée suite à l'appel de la méthode go (**didPop** est à **true**)

Pour récupérer ces données pour le 1er écran, nous allons utiliser le **GoRouterState** :
Dans **main.dart**, veuillez mettre à jour la route vers le **FirstScreen** :

```
GoRoute(
  path: '/',
  builder: (context, state) {
    final int nbClicks = (state.extra ?? 0) as int;
    return FirstScreen(nbClicks: nbClicks);
  },
```

Ici, nous retrouvons dans l'état du router la variable **extra**. Si nous n'avons pas navigué vers le 2ème écran, elle sera « Null », et donc nous considérons la valeur 0. Comme la variable **extra** est de type **Object**, nous devons la « caster » vers un **int**.

NB : il serait possible directement dans **first_screen.dart** d'appeler la variable extra dans la méthode **build** . Voici comment vous auriez fait :

```
final int nbClicks = (GoRouterState.of(context).extra  ?? 0) as int;
```

Néanmoins, nous trouvons plus propre, comme le composant **FirstScreen** est stateless, de passer le nombre de clicks en paramètre du widget.

[commit avec message : T04.2 Navigation avec extra]

2.2.5 Passage d'information avec des paramètres de chemin

Dans un premier temps, nous allons mettre à jour le 1er écran afin d'afficher une liste de usernames.
Lorsqu'on clique sur un de ces usernames, nous aimerions que cela appelle l'écran UserScreen pour afficher toutes les informations associées à l'utilisateur sélectionné.

Dans **first_screen.dart**, nous ajoutons le composant **UserListView**, qui lorsqu'on clique sur un des utilisateurs, fera appel à la navigation via **context.go('/users/$username')** :

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

class FirstScreen extends StatelessWidget {
  final int nbClicks;

  const FirstScreen({super.key, this.nbClicks = 0});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("First screen"),
        backgroundColor: Theme.of(context).colorScheme.primaryContainer,
      ),
      body: Center(
        child: Column(
```

```
          mainAxisSize: MainAxisSize.min,
          children: [
            const Text("Hello from first screen."),
            const SizedBox(height: 16),
            Text("There were $nbClicks clicks in the second page."),
            const SizedBox(height: 16),
            ElevatedButton(
              onPressed: () => context.go("/secondscreen"),
              child: const Text("Go to second screen"),
            ),
            Expanded(child: UserListView()),
          ],
        ),
      ),
    );
  }
}

class UserListView extends StatelessWidget {
  final List<String> usernames = ['mcCain123', 'greg123', 'sarah123'];

  UserListView({super.key});

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: usernames.length,
      itemBuilder: (context, index) {
        final username = usernames[index];
        return ListTile(
          title: Text(username),
          onTap: () => context.go('/users/$username'),
        );
      },
    );
  }
}
```

Nous allons maintenant créer l'écran **UserScreen** qui permettra d'afficher les données d'un utilisateur. Veuillez créer le script **user_screen.dart** :

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

class UserScreen extends StatelessWidget {
  final String username;

  const UserScreen({super.key, required this.username});

  @override
  Widget build(BuildContext context) {
    final Map<String, String> userData = getUserData(username);

    return Scaffold(
      appBar: AppBar(
        title: const Text('User Details'),
        backgroundColor: Theme.of(context).colorScheme.primaryContainer,
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
```

```
          children: [
            Text('Username: ${userData['username']}'),
            Text('First Name: ${userData['firstname']}'),
            Text('Last Name: ${userData['lastname']}'),
            Text('Email: ${userData['email']}'),
          ],
        ),
      ),
    );
  }
}

Map<String, String> getUserData(String username) {
  final List<Map<String, String>> userList = [
    {
      'username': 'mcCain123',
      'firstname': 'John',
      'lastname': 'McCain',
      'email': 'john.mccain@example.com',
    },
    {
      'username': 'greg123',
      'firstname': 'Greg',
      'lastname': 'Doe',
      'email': 'greg.doe@example.com',
    },
    {
      'username': 'sarah123',
      'firstname': 'Sarah',
      'lastname': 'Johnson',
      'email': 'sarah.johnson@example.com',
    },
  ];

  return userList.firstWhere((user) => user['username'] == username);
}
```

Nous devons maintenant rajouter une route au sein de **main.dart** afin d'appeler le composant **UserScreen** en récupérant la variable de chemin qui est passée lors de l'appel de **context.go('/users/$username')** (par exemple "macCain123" si l'on clique sur l'utilisateur macCain123). Au sein de **GoRouter** dans **main.dart**, veuillez ajouter une route « enfant » de la route "/" nommée "users/:username" (ça nous permettra d'avoir un bouton « back » automatiquement). Voici le code du router après cet ajout :

```
final GoRouter _router = GoRouter(
  initialLocation: '/',
  routes: [
    GoRoute(
      path: '/',
      builder: (context, state) {
        final int nbClicks = (state.extra ?? 0) as int;
        return FirstScreen(nbClicks: nbClicks);
      },
      routes: [
        GoRoute(
          path: 'secondscreen',
          builder: (context, state) => const SecondScreen(),
        ),
```

```
        GoRoute(
          path: 'users/:username',
          builder: (context, state) =>
              UserScreen(username: state.pathParameters['username'] ?? ''),
        ),
      ],
    ),
  ],
);
```

Ainsi, nous avons une route qui s'occupe d'afficher n'importe quel utilisateur. Pour cela, nous avons utilisé un « path parameter ». Ce paramètre est récupéré via le state du routeur. Notons qu'il est aussi possible de récupérer des « query parameters ».

[commit avec message : T04.3 Navigation avec path parameter]

2.2.6 En savoir plus

Si vous souhaitez en savoir plus sur Go Router, n'hésitez pas à consulter sa documentation officielle : https://pub.dev/documentation/go_router/latest/

Si vous souhaitez explorer d'autre façon de gérer un router et la navigation en Flutter : https://docs.flutter.dev/ui/navigation

2.3 Architecture MVVM

Avec des applications toujours plus complexes, il devient vite nécessaire de rajouter une structure au sein du code pour s'y retrouver entre les différents fichiers et bien séparer les responsabilités. Vous avez déjà vu différents modèles d'architectures dans différents cours tels que MVC, fat-model ou l'architecture trois-tiers. Dans le monde de l'informatique mobile et de Android, un modèle d'architecture très fréquemment utilisé est le modèle **MVVM**, pour **Model – View – View Model**.

- La couche **Model** correspond comme dans beaucoup de modèles architecturaux à la définition des données. Comme dans l'architecture fat-model, nous pourrons également retrouver dans cette couche des responsabilités supplémentaires comme la sérialisation ou l'accès aux données.
- La couche **View** correspond aux différents widgets qui vont composer nos écrans et leurs composants. Elle définit la structure et l'affichage de l'application. Pour bien séparer les responsabilités, nous allons déplacer la gestion de l'état de l'application vers la couche suivante.
- La couche **View Model** correspond à la gestion de l'état partagé de l'application. C'est maintenant dans cette couche que nous allons enregistrer les variables d'état ainsi que les méthodes permettant leur modification. Les widgets n'ont alors plus qu'à faire s'inscrire aux view models et afficher l'application en fonction de leur état.

Dans le projet de ce tutoriel, nous n'avons pas de couche **Model** comme les données utilisées sont juste un simple entier. Nous n'avons pas encore de couche **View Model**, cela viendra dans la section suivante. Nous pouvons par contre regrouper les fichiers de notre couche **View**. Créez un dossier *views* et déplacez-y les fichiers contenant les écrans.

2.4 Package Provider

Dans le but de créer un view model en flutter, nous allons utiliser le package **Provider**. Afin de rajouter ce package à un projet flutter, il est nécessaire d'exécuter les commandes suivantes dans un terminal.

- **flutter pub add provider**
- **flutter pub get**

Veuillez lire l'article suivant sur la gestion de l'état avec ce package : [Simple app state management].

OBSERVATIONS

- On définit un view model en créant une classe qui étend la classe **ChangeNotifier**. Il contient les variables d'état et les méthodes permettant de les modifier.
- Une bonne pratique est de garder les variables d'état privées, et de définir des getter permettant d'accéder à leur valeur mais de ne pas les modifier. Pour les variables d'état contenant des listes, la méthode getter renvoie un **UnmodifiableListView** avec la liste en argument. Cette classe permet de visualiser l'état de la liste encapsulée en implémentant l'interface Iterable elle aussi, mais elle ne permet pas de modifier. De cette façon, les widgets pourront récupérer l'état de la liste sans risquer de la modifier. Assurez-vous de bien comprendre la syntaxe pour créer des getter.
- Au sein des méthodes de modification des variables d'état, on utilise la méthode **notifyListener** pour indiquer aux widgets qui utilisent le view model que l'état a changé et qu'ils doivent mettre à jour leur affichage.
- Le widget **ChangeNotifierProvider** permet de rendre disponible un view model à tous les widgets enfants de celui-ci. On utilise le paramètre **create** pour indiquer comment créer le view model. Le widget se chargera de créer l'instance du view model et de la partager avec tous ses enfants qui la requièrent.
- Pour accéder au view model dans un enfant d'un **ChangeNotifierProvider**, on utilise un widget **Consumer<MyViewModel>**. Il prend un argument **builder** avec une fonction dont le deuxième argument est l'instance du view model et qui renvoie le widget à afficher. Il est alors possible d'utiliser le view model pour accéder aux variables d'état et de faire appel à des méthodes de modification. Lorsque l'état du view model change, cette méthode builder est re-appelée pour mettre à jour l'affichage en fonction du nouvel état.
- Lorsqu'on souhaite récupérer la valeur d'une variable d'état, ou faire appel à une méthode de modification, sans vouloir faire de rebuild en fonction de la modification de l'état, on peut utiliser la fonction **Provider.of<MyViewModel>(context, listen : false)** qui renvoie une instance du view model au sein d'un enfant d'un **ChangeNotifierProvider**.

2.5 Création d'un view model

Créez un dossier **view_models** avec un fichier **click_view_model.dart** contenant le code suivant :

```
import 'package:flutter/cupertino.dart';

class ClickViewModel extends ChangeNotifier {
  var _clicks = 0;
```

```
  int get clicks => _clicks;

  void increment() {
    _clicks++;
    notifyListeners();
  }
}
```

Ce view model contient une variable d'état entière **_clicks**, avec un getter clicks permettant d'y accéder et une méthode **increment** qui permet de l'incrémenter et de réafficher les widgets qui l'utilisent grâce à l'appel à la méthode **notifyListeners**.

2.6 Utilisation du view model

Pour rendre disponible le view model au sein de l'application, il faut d'abord décider à quel niveau faire appel au **ChangeNotifierProvider**. Comme nous aurons besoin du view model au sein des deux premiers écrans, il faut rajouter le provider au-dessus du widget qui y fait appel – le MaterialApp. Nous pouvons le faire directement au sein de *runapp* : modifiez le fichier *main.dart* de la façon suivante :

```
void main() {
  runApp(ChangeNotifierProvider<ClickViewModel>(
    create: (context) => ClickViewModel(),
    child: const MyApp(),
  ));
}
```

Grâce à l'utilisation du view model, nous pouvons récupérer le nombre de clics au sein du router. Nous n'utiliserons donc plus la valeur *extra* du router pour initialiser *nbClicks.*
Modifiez le fichier *main.dart* de la façon suivante pour initialiser *nbClicks*:

```
final int nbClicks =
    Provider.of<ClickViewModel>(context, listen: false).clicks;
```

Ici, chaque navigation vers le 1er écran implique déjà un rebuild associé au router. Il n'est donc pas utile d'appeler un Consumer.

Au sein du second écran, celui-ci peut devenir stateless grâce à l'utilisation d'un ViewModel ! Nous allons faire appel à un Consumer pour assurer un rebuild de l'application à chaque fois que le nombre de clic est incrémenté.

Modifiez le fichier *second_screen.dart* de la façon suivante :

```
class SecondScreen extends StatefulWidget {
  const SecondScreen({Key? key}) : super(key: key);

  @override
  State<SecondScreen> createState() => _FirstScreenState();
}

class _FirstScreenState extends State<SecondScreen> {
  var nbClicks = 0;

  @override
  Widget build(BuildContext context) {
```

```
    return PopScope(
      canPop: false,
      onPopInvoked: (didPop) {
        if (!didPop) {
          context.go("/", extra: nbClicks);
        }
      },
      child: Scaffold(
        appBar: AppBar(
          title: const Text("Second screen"),
          backgroundColor: Theme.of(context).colorScheme.primaryContainer,
        ),
        body: Center(
          child: Consumer<ClickViewModel>(builder: (context, viewModel,
child) {
            return Column(
              mainAxisSize: MainAxisSize.min,
              children: [
                const Text("Hello from second screen."),
                const SizedBox(height: 16),
                Text("There were ${viewModel.clicks} clicks."),
                const SizedBox(height: 16),
                ElevatedButton(
                  onPressed: () => viewModel.increment(),
                  child: const Text("click me"),
                ),
              ],
            );
          }),
        ),
      ),
    );
  }
}
```

Lancez l'application et vérifiez qu'elle fonctionne bien. Chose assez exceptionnelle, vous devriez maintenant pouvoir passer d'un écran à un autre tout en conservant le nombre de clics !

[commit avec message : T04.4 View model provider]

3 Exercice

3.1 Introduction

Veuillez créer un nouveau projet (New Flutter Project) nommé *ex4* dans votre repository de cours.

L'objectif de cet exercice est de créer une application de lecture et de création d'articles. Vous trouverez sur moodle 4 fichiers. Vous y trouverez une classe article avec quelques articles de test, ainsi que trois écrans.

La classe Article définit les données de notre application. Elle s'insère dans la couche *models* de l'architecture MVVM.

L'écran d'accueil affiche une liste d'articles non lus. Un bouton dans la barre de titre permet d'afficher (ou de cacher) les articles lus dans la liste. Pour chaque article, un bouton permet de le marquer comme lu et un autre de le supprimer de la liste. En cliquant sur un article, l'application navigue vers l'écran d'affichage. Un floating action button permet de naviguer vers l'écran de création.

L'écran de création affiche un formulaire permettant d'entrer le titre, auteur et contenu d'un nouvel article. Un bouton dans la barre de titre permet d'annuler la création et de revenir à la page d'accueil. Le bouton de soumission du formulaire permet de créer l'article et faire un reset du formulaire.

L'écran d'affichage d'un article présente le titre, l'auteur et le contenu d'un article. Un bouton dans la barre de titre permet de revenir à la page d'accueil. Un floating action button permet de marquer l'article comme lu/non lu.

Tel que vous les avez reçus, les écrans ne font que l'affichage et aucun bouton ne fonctionne. Pour cet exercice, vous devrez suivre les TODO marqués pour leur rendre leurs fonctionnalités. Copiez ces fichiers au sein de votre projet en respectant l'architecture MVVM.

3.2 Navigation

Faites appel aux différents écrans au sein de votre application en implémentant une navigation, et suivez les TODO F06 pour ajouter les fonctionnalités de changement d'écrans.
Lancez l'application et vérifiez que vous pouvez passer d'un écran à l'autre.
L'écran à afficher initialement est le *ListScreen*. De là, vous pourrez passer vers le *FormScreen* ou le *ArticleScreen*.

⚡ **[commit avec message : F06.1 Navigation]** ⚡

3.3 View model & état partagé avec Provider

Créez un view model qui garde comme variable d'état une liste d'articles et un booléen indiquant s'il faut afficher les articles lus. Il doit être possible de récupérer un article par son id (à utiliser dans la route permettant d'afficher un article sur base du paramètre de chemin **id** au lieu d'utiliser le paramètre **extra**), ajouter un article ou supprimer un article à la liste, marquer un article de la liste comme lu ou non lu et de modifier la valeur du booléen pour indiquer qu'il faut afficher les articles lus ou non.
N'oubliez pas de respecter l'architecture MVVM et d'ajouter le package provider à votre projet.

Utilisez le view model que vous avez créé au sein de vos écrans, et suivez les TODO F07 pour ajouter les fonctionnalités liées à l'état de l'application.
Lancez l'application et vérifiez que vous pouvez modifier l'état de l'application.
⚡ **[commit avec message : F07.1 Etat partagé avec Provider]** ⚡