|  |  |
| --- | --- |
| Flutter |  |
| **Fiche 4 : Navigation & Provider** | |

Table des matières

[1 Objectifs à valider 2](#_Toc129792423)

[2 Concepts 2](#_Toc129792424)

[2.1 Introduction 2](#_Toc129792425)

[2.2 Navigation entre écrans 2](#_Toc129792426)

[2.3 Deux écrans simple 3](#_Toc129792427)

[2.4 Architecture MVVM 5](#_Toc129792428)

[2.5 Package Provider 5](#_Toc129792429)

[2.6 Création d’un view model 6](#_Toc129792430)

[2.7 Utilisation du view model 6](#_Toc129792431)

[3 Exercice 8](#_Toc129792432)

[3.1 Introduction 8](#_Toc129792433)

[3.2 Navigation 8](#_Toc129792434)

[3.3 View model 9](#_Toc129792435)

[3.4 Etat partagé avec Provider 9](#_Toc129792436)

# Objectifs à valider

|  |  |  |
| --- | --- | --- |
| **ID** | **Objectifs** | **Indicateurs** |
| F06 | Navigation entre écrans | * Les messages de commits F06.X sont visibles. * Vous définissez plusieurs écrans dans votre application. * Vous montrez et expliquez comment vous passez d’un écran à l’autre. * Vous expliquez comment passer un argument à une route et comment le récupérer. |
| F07 | Gestion d’état avec Provider | * Les messages de commits F07.X sont visibles. * Vous utilisez une architecture MVVM dans votre projet. * Vous montrez et expliquez comment vous définissez un view model qui définit un état partagé dans votre application. * Vous expliquez comment récupérer et faire des actions sur votre état partagé au sein de vos écrans. |

# Concepts

## Introduction

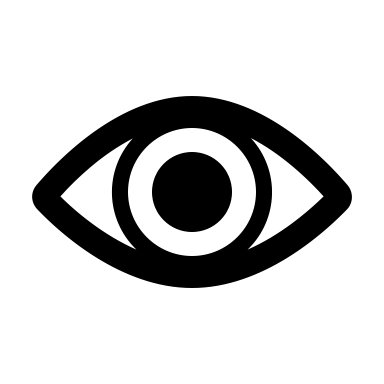
**Pour commencer le tutoriel, créez un nouveau projet (New Flutter Project) nommé *ftuto4* dans votre repository de cours.**

## Navigation entre écrans

Nous allons découvrir dans cette fiche comment naviguer entre plusieurs écrans. Une fois de plus, les équipes de flutter ont écrit de nombreuses recettes de cuisine : [Navigation](https://docs.flutter.dev/cookbook#navigation).

Parmi les différentes manières de faire de la navigation en flutter, nous allons utiliser les routes nommées. Veuillez lire les articles suivants :

* [Navigate with named routes](https://docs.flutter.dev/cookbook/navigation/named-routes) : différents principes de navigation avec flutter.
* [Pass arguments to a named route](https://docs.flutter.dev/cookbook/navigation/navigate-with-arguments) : passage d’arguments à un écran. Ne passez pas trop de temps sur cet article qui rend les choses plus compliquées que nécessaires. Il est possible de passer juste une valeur ou un objet directement comme argument à une route sans avoir besoin de créer de classe spécifique pour les arguments.

****Observations

* Au lieu d’afficher un seul écran avec le paramètre **home**, le widget **MaterialApp** permet également de définir la navigation, en définissant le paramètre **routes** pour paramétrer le widget d’écran à afficher en fonction de la route, et le paramètre **initialRoute** pour indiquer la route à afficher par défaut.
* On utilise la fonction **Navigator.pushNamed(context, '/route-name')** pour naviguer vers un autre écran.
* Il est également possible de passer un argument à cet écran avec le paramètre nommé **arguments**. L’appel à la fonction devient alors **Navigator.pushNamed(context, '/route-name', arguments: argumentValue)**. Pour récupérer cet argument, il faut alors faire appel à la fonction **ModalRoute.of(context)!.settings.arguments** lors de la création de l’écran suivant.
* On peut revenir à l’écran précédent avec la fonction **Navigator.pop(context)**.

## Deux écrans simple

Nous allons maintenant créer deux écrans simples pour tester la navigation. Commencez par créer un fichier *first\_screen.dart* et copiez-y le code suivant :

*class* FirstScreen *extends* StatefulWidget {  
 *const* FirstScreen({Key? key}) : *super*(key: key);  
  
 *@override* State<FirstScreen> createState() => \_FirstScreenState();  
}  
  
*class* \_FirstScreenState *extends* State<FirstScreen> {  
 *var* nbClicks = 0;  
  
 *@override* Widget build(BuildContext context) {  
 *return* Scaffold(  
 appBar: AppBar(title: *const* Text("First screen")),  
 body: Center(  
 child: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: [  
 *const* Text("Hello from first screen."),  
 *const* SizedBox(height: 16),  
 ElevatedButton(  
 onPressed: () => setState(() => nbClicks++),  
 child: *const* Text("click me"),  
 ),  
 *const* SizedBox(height: 16),  
 ElevatedButton(  
 onPressed: () => Navigator.pushNamed(  
 context,  
 "/second",  
 arguments: nbClicks,  
 ),  
 child: *const* Text("Go to second screen"),  
 ),  
 ],  
 ),  
 ),  
 );  
 }  
}

Cet écran affiche deux boutons. Le premier ne semble à première vue ne rien faire, mais incrémente en réalité un compteur du nombre de clicks. Lorsque l’utilisateur clique sur le deuxième bouton, il est redirigé avec **Navigator.pushNamed** vers une route nommée « /second » avec en argument ce compteur.

**[commit avec message : FT04.1 Premier écran]**

Créez également un fichier *second\_screen.dart* avec ce code :

*class* SecondScreen *extends* StatelessWidget {  
 *const* SecondScreen({Key? key}) : *super*(key: key);  
  
 *@override* Widget build(BuildContext context) {  
 *final* nbClicks = ModalRoute.of(context)!.settings.arguments *as* int;  
  
 *return* Scaffold(  
 appBar: AppBar(  
 title: *const* Text("Second screen"),  
 leading: IconButton(  
 icon: *const* Icon(Icons.arrow\_back),  
 onPressed: () => Navigator.pop(context),  
 ),  
 ),  
 body: Center(  
 child: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: [  
 *const* Text("Hello from second screen."),  
 *const* SizedBox(height: 16),  
 Text("There were $nbClicks clicks in the first page."),  
 ],  
 ),  
 ),  
 );  
 }  
}

Ce deuxième écran récupère d’abord comme argument un entier avec **ModalRoute.of(context)!.settings.arguments**. Il affiche ensuite ce compteur. Un bouton dans la barre de titre permet de revenir au premier écran avec **Navigator.pop**.

**[commit avec message : FT04.2 Deuxième écran]**

Pour terminer, modifiez le fichier *main.dart* pour définir la navigation entre vos deux écrans.

*@override*Widget build(BuildContext context) {  
 *return* MaterialApp(  
 title: 'Flutter Demo',  
 theme: ThemeData(primarySwatch: Colors.blue),  
 initialRoute: "/first",  
 routes: {  
 "/first": (context) => *const* FirstScreen(),  
 "/second": (context) => *const* SecondScreen(),  
 },  
 );  
}

Cette navigation définit deux routes avec les deux écrans que nous avons créé, le premier avec la route « /first » et le deuxième avec « /second ». La route à afficher par défaut est « /first ».

Lancez l’application et vérifiez qu’elle fonctionne comme on l’attend.

**[commit avec message : FT04.3 Navigation]**

## 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.

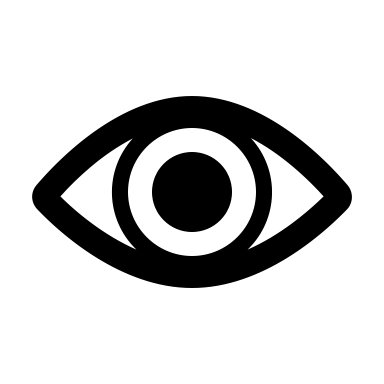
**[commit avec message : FT04.4 Couche views]**

## 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](https://docs.flutter.dev/development/data-and-backend/state-mgmt/simple).

****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**.

## Création d’un view model

Créez un dossier *view\_model* avec un fichier *click\_view\_model.dart* contenant le code suivant :

*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*.

**[commit avec message : FT04.5 Couche view model]**

## 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 écrans, il est nécessaire de rajouter le provider juste au-dessus du widget qui y fait appel – le MaterialApp. Modifiez le fichier *main.dart* de la façon suivante :

*@override*Widget build(BuildContext context) {  
 *return* ChangeNotifierProvider<ClickViewModel>(  
 create: (context) => ClickViewModel(),  
 child: MaterialApp(  
 title: 'Flutter Demo',  
 theme: ThemeData(primarySwatch: Colors.blue),  
 initialRoute: "/first",  
 routes: {  
 "/first": (context) => *const* FirstScreen(),  
 "/second": (context) => *const* SecondScreen(),  
 },  
 ),  
 );  
}

**[commit avec message : FT04.5 ChangeNotifierProvider]**

Grâce à l’utilisation du view model, nous pouvons retirer la variable d’état *nbClicks* du premier écran, et du coup le transformer en stateless widget. Nous pouvons ensuite faire appel à la méthode *increment* du view model grâce à *Provider.of<ClickViewModel>*. Modifiez le fichier *first\_screen.dart* de la façon suivante :

*class* FirstScreen *extends* StatelessWidget {  
 *const* FirstScreen({Key? key}) : *super*(key: key);  
  
 *@override* Widget build(BuildContext context) {  
 *return* Scaffold(  
 appBar: AppBar(title: *const* Text("First screen")),  
 body: Center(  
 child: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: [  
 *const* Text("Hello from first screen."),  
 *const* SizedBox(height: 16),  
 ElevatedButton(  
 onPressed: () {  
 *final* viewModel = Provider.of<ClickViewModel>(  
 context,  
 listen: *false*,  
 );  
 viewModel.increment();  
 },  
 child: *const* Text("click me"),  
 ),  
 *const* SizedBox(height: 16),  
 ElevatedButton(  
 onPressed: () => Navigator.pushNamed(context, "/second"),  
 child: *const* Text("Go to second screen"),  
 ),  
 ],  
 ),  
 ),  
 );  
 }  
}

**[commit avec message : FT04.6 Provider.of]**

Au sein du second écran, nous pouvons retirer l’argument lié au nombre de clicks. Nous pouvons entourer le widget *Text* qui a besoin du nombre de click par un widget *Consumer<ClickViewModel>* pour récupérer l’instance du view model et accéder à la valeur de la variable d’état. Modifiez le fichier *second\_screen.dart* de la façon suivante :

*@override*Widget build(BuildContext context) {  
 *return* Scaffold(  
 appBar: AppBar(  
 title: *const* Text("Second screen"),  
 leading: IconButton(  
 icon: *const* Icon(Icons.arrow\_back),  
 onPressed: () => Navigator.pop(context),  
 ),  
 ),  
 body: Center(  
 child: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: [  
 *const* Text("Hello from second screen."),  
 *const* SizedBox(height: 16),  
 Consumer<ClickViewModel>(  
 builder: (context, viewModel, child) =>  
 Text("There were ${viewModel.clicks} clicks in the first page."),  
 ),  
 ],  
 ),  
 ),  
 );  
}

Lancez l’application et vérifiez qu’elle fonctionne toujours de la même façon.

**[commit avec message : FT04.7 Consumer]**

# Exercice

## 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 model 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 de revenir à la page d’accueil.

L’écran d’affichage affiche un article avec son titre, son auteur et son contenu. 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 reçu, 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.

**⚡ [commit** **avec message : F06.1 Récupération du boilerplate]** ⚡

## 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.

**⚡ [commit avec message : F06.2 Navigation]** ⚡

## View model

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, 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.

**⚡ [commit avec message : F07.1 View model]** ⚡

## Etat partagé avec Provider

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 : F07.2 Etat partagé avec Provider]** ⚡