Flutter - Navigation et Popup

Splash Screen

Un splash screen (écran de démarrages) offre une expérience initiale simple pendant le chargement de votre application mobile. Il prépare le terrain pour votre application, tout en laissant le temps au moteur d'application de se charger et à votre application de s'initialiser.
Voici un exemple

```
class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});
  final String title;
  @override
  SplashScreenState createState() => SplashScreenState();
}

class SplashScreenState extends State<MyHomePage> {
  @override
  void initState() {
   super.initState();
   Timer(Duration(seconds: 3),
            ()=>Navigator.pushReplacement(context,
            MaterialPageRoute(builder: (context) => HomeScreen() )
        )
   );
 }
  @override
  Widget build(BuildContext context) {
   return Container(
       color: Colors.amber,
       child:FlutterLogo(size:MediaQuery.of(context).size.height)
   );
 }
}
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title:Text("Exemple")),
     body: const Center(
         child:Text("Welcome to Home Page",
             style: TextStyle( color: Colors.black, fontSize: 30)
         )
     ),
```

```
    );
  }
}
```

Routes

Il est maintenant temps de s'intéresser à un point crucial, les routes. L'objectif est de voir comment la navigation d'écran en écran peut être mise en place.
Le widget chargé de ce rôle est la classe **Navigator**. Il fonctionne en plaçant les écrans les uns devant les autres sous la forme d'une pile. Cet effet de superposition permet en outre de revenir en arrière puisque cette pile reste mémorisée.
Pour être vraiment efficaces, les routes peuvent être définies à l'avance sous une forme proche d'une URL.

Push et Pop

https://docs.flutter.dev/development/ui/navigation

La classe **Navigator** possède deux méthodes pour naviguer d'un écran à l'autre.
Push, qui permet d'ouvrir une route, d'un écran vers un autre, et **Pop** qui permet de revenir à la route précédente.
https://docs.flutter.dev/cookbook/navigation/navigation-basics
Autre lecture : https://medium.com/flutter-community/flutter-push-pop-push-1bb718b13c31

Pour faciliter la maintenance de l'application, il est possible de référencer les différentes routes qui serviront tout au long du projet. Dans la classe qui sert au lancement (**runApp**), il existe un paramètre du constructeur **MaterialApp** qui se nomme ***routes***. Il va accueillir une map possédant une clé de type chaîne de caractères et une valeur de type **WidgetBuilder.**

Le code qui permet le référencement des routes est le suivant :

```
class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: const FirstPage(title: 'Flutter Navigation'),
     routes: <String, WidgetBuilder>{
       '/route1': (BuildContext context) => FirstPage(title: 'Page 1'),
       '/route2': (BuildContext context) => SecondPage(title: 'Page 2'),
       '/route3': (BuildContext context) => ThirdPage(title: 'Page 3'),
     },
   );
```

```
 }
}
```

Il fait référence à 3 classes prédéfinies dans le projet (First, Second et ThirdPage).
Au niveau des clés, des chaînes de caractères faisant penser à des URL sont placées. En guise de valeurs, des fonctions qui renvoient vers les constructeurs des différentes classes, correspondants aux trois pages, sont affectées.

Une fois référencées de la sorte, les pages peuvent être appelées grâce à la méthode **Navigator.pushNamed** qui prend deux paramètres : le contexte et une des clés définies dans routes. Par exemple, en ce qui concerne l'appel de la seconde page :

```
Navigator.pushNamed(context, '/route2');
```

Une fois en place, il est intéressant de constater l'effet de pile grâce à **l'AppBar**. Passé le premier écran, **l'AppBar** dispose d'une flèche de retour en arrière qui permet de remonter la pile comme le ferait la méthode **pop** de **Navigator**.

Passage d'argument

Il est possible de passer des arguments, via l'attribut **argument** de **pushNamed()**

```
Navigator.pushNamed(context, '/route2', arguments: 'Mon argument' );
```

Ainsi, vous pouvez récupérer des objets, ou listes d'objets, dans la page correspondante a la transmission d'information (ici la classe associé à la /route2)

```
@override
Widget build(BuildContext context) {
  var arguments = ModalRoute.of(context)?.settings.arguments as String;
```

ScaffoldMessenger

Il est possible de faire apparaître des barres d'informations dans nos interfaces. Pour cela, on utilise le widget **ScaffoldMessenger**, auquel on doit associer un widget **SnackBar** ou **MaterialBanner.**

Une **Snackbar** est une barre qui s'affiche en bas du Scaffold, bien souvent pour donner une information (comme une validation d'envoie de données, ou de connexion).
Son utilisation est très simple, est ce fait comme ceux-ci :

```
ScaffoldMessenger.of(context).showSnackBar(const SnackBar(
   content: Text("Message de SnackBar !"),
));
```

A l'inverse, la **MaterialBanner** s'affiche en haut du Scaffold, et est associé à au moins une "action" (correspondant bien souvent à un **TextButton)** pour valider la prise de connaissance de l'information transmise

```
ScaffoldMessenger.of(context).showMaterialBanner(const MaterialBanner(
   content: Text("Message de MaterialBanner"),
   actions: <Widget>[
      TextButton(
         child: Text('Ok!'),
         onPressed: () {
             ScaffoldMessenger.of(context).hideCurrentMaterialBanner();
         }),
      ),
   ],
));
```

/!\ Attention, avec les **MaterialBanner** et la navigation. Si une **MaterielBanner** n'est pas fermé au moment d'une navigation (pop ou push par exemple), il risque d'y avoir un problème avec la fermeture de cette dernière au le nouveau contexte, car cette dernière ne sera plus associée au **context** avec lequel la **MaterialBanner** était associé.
Pour cela, je vous invite à ajouter le widget **PopScope**, qui prendra comme **child** le **Scaffold**, comme dans l'exemple ci-après.

```
...
return PopScope(
     canPop: true,
     onPopInvokedWithResult: (didPop, dyn) {
       ScaffoldMessenger.of(context).removeCurrentMaterialBanner();
     },

     child: Scaffold(
...
```

AlertDialog

Il est possible de faire apparaître des barres d'informations dans nos interfaces. Pour cela, on utilise le widget **ScaffoldMessenger**, auquel on doit associer un widget **SnackBar** ou **MaterialBanner.**

A l'inverse, la **MaterialBanner** s'affiche en haut du Scaffold, et est associé à au moins une "action" (correspondant bien souvent à un **TextButton)** pour valider la prise de connaissance de l'information transmise:

```
...
onPressed: () => showDialog<String>(
   context: context,
   builder: (BuildContext context) => AlertDialog(
      title: const Text('Titre de l'AlertDialog'),
      content: const Text('Description de l'AlertDialog'),
      actions: <Widget>[
         TextButton(
            onPressed: () => Navigator.pop(context, 'Quitter'),
            child: const Text('Quitter'),
         ),
         TextButton(
            onPressed: () => Navigator.pop(context, 'Ok'),
            child: const Text('Ok'),
         ),
      ],
   ),
),
...
```