

Le développement multi-plateforme avec Flutter

Contexte et introduction à Flutter

Qu'est ce que Flutter ? - 1

- [Flutter](#) est un framework pour développer des applications avec une interface utilisateur qui cible présentement :
 - Les mobiles avec un système IOS ou Android
 - Les navigateurs
 - Les desktop Linux, Windows et MacOS
- Avantages :
 - Un code compilé natif/javascript avec de bonnes performances pour les plates-formes ciblées
 - Une vision plus récente du développement pour Android
 - Le développement des interfaces utilisateurs s'appuie sur des technos récentes (Interfaces Utilisateurs déclaratives comme [Compose](#) ou [SwiftUI](#)) et sur les composants [Material](#) de Google.
 - A la différence de Kotlin (ou Java) qui doit prendre en compte toutes les nombreuses briques d'Android existantes (anciennes ou récentes), le développement avec Flutter nécessite moins d'expérience.

Qu'est ce que Flutter ? - 2

- Inconvénients :
 - Il n'est pas adapté pour reprendre une application existante et la maintenir
 - Une application existante a probablement été écrite en Java et plus récemment en Kotlin
 - Le développement s'effectue avec un langage qui s'appelle [Dart](#) qui doit être appris
 - NB : le passage d'un langage type Java à un autre comme C#, Kotlin (Dans Android Studio on peut convertir des sources Java en Kotlin), Dart, ... est relativement aisé
 - L'interface utilisateur est codée
- Les outils :
 - On peut utiliser le mode ligne de commandes et un plug-in dans Visual Studio Code
 - On peut aussi utiliser Android Studio (Normalement dédié aux projets natifs Android en Java ou Kotlin) mais qui gère les projets Flutter :
 - Nécessite les plugins Flutter et Dart

Nos objectifs avec Flutter

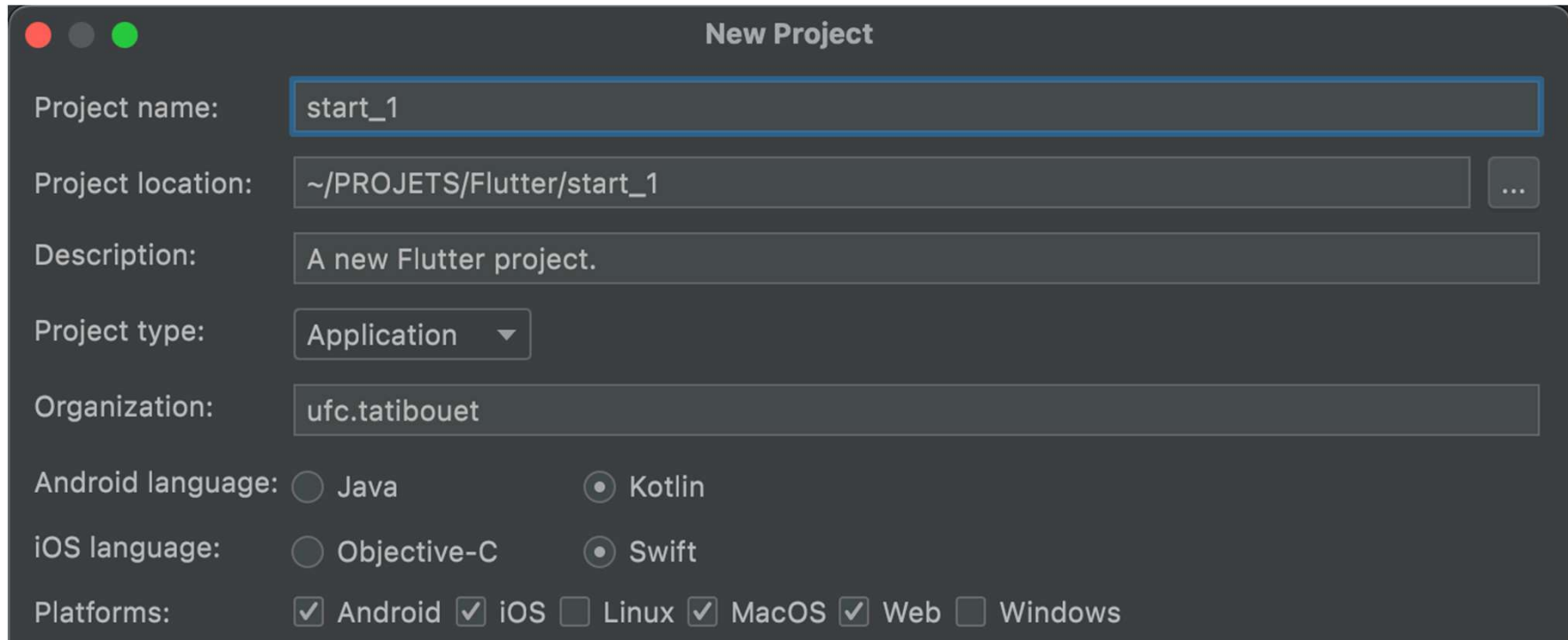
- On vise avant tout le développement mobile et plus particulièrement le développement pour Android.
- 1° solution :
 - En utilisant un mobile (ou tablette) branché sur le port USB
 - Le mode développeur doit avoir été activé sur le mobile
 - Inconvénient : on ne dispose que d'une cible
 - Avantage : on peut tester plus facilement tous les dispositifs matériels :
 - Photo, GPS, ...
- 2° solution :
 - En créant des mobiles virtuels exécutés par l'émulateur
 - Inconvénients :
 - Très consommateur d'espace disque et de ressources à l'exécution, moins simple de tester les dispositifs matériels du mobile
 - Heureusement les émulateurs sont devenus beaucoup plus performants au fil des années

Les ressources bibliographiques

- Livres :
 - Flutter Développez vos applications mobiles multiplateformes avec Dart – Julien Trillard – Editions ENI – ISBN : 978-2-409-02527-3
 - Flutter Complete Reference – Alberto Miola – ISBN : 9798691939952
- WEB :
 - Flutter : <https://flutter.dev/docs>
 - Dart : <https://dart.dev/>
 - Material Design : <https://material.io/design>
 - Les conférences Google IO : <https://io.google/2022/intl/fr/> qui donnent l'orientation ou la montée en puissance de certaines technologies.

L'application créée avec Android Studio - 1

Dans Android Studio, faire [File](#) → [New](#) → [New Flutter Project](#)

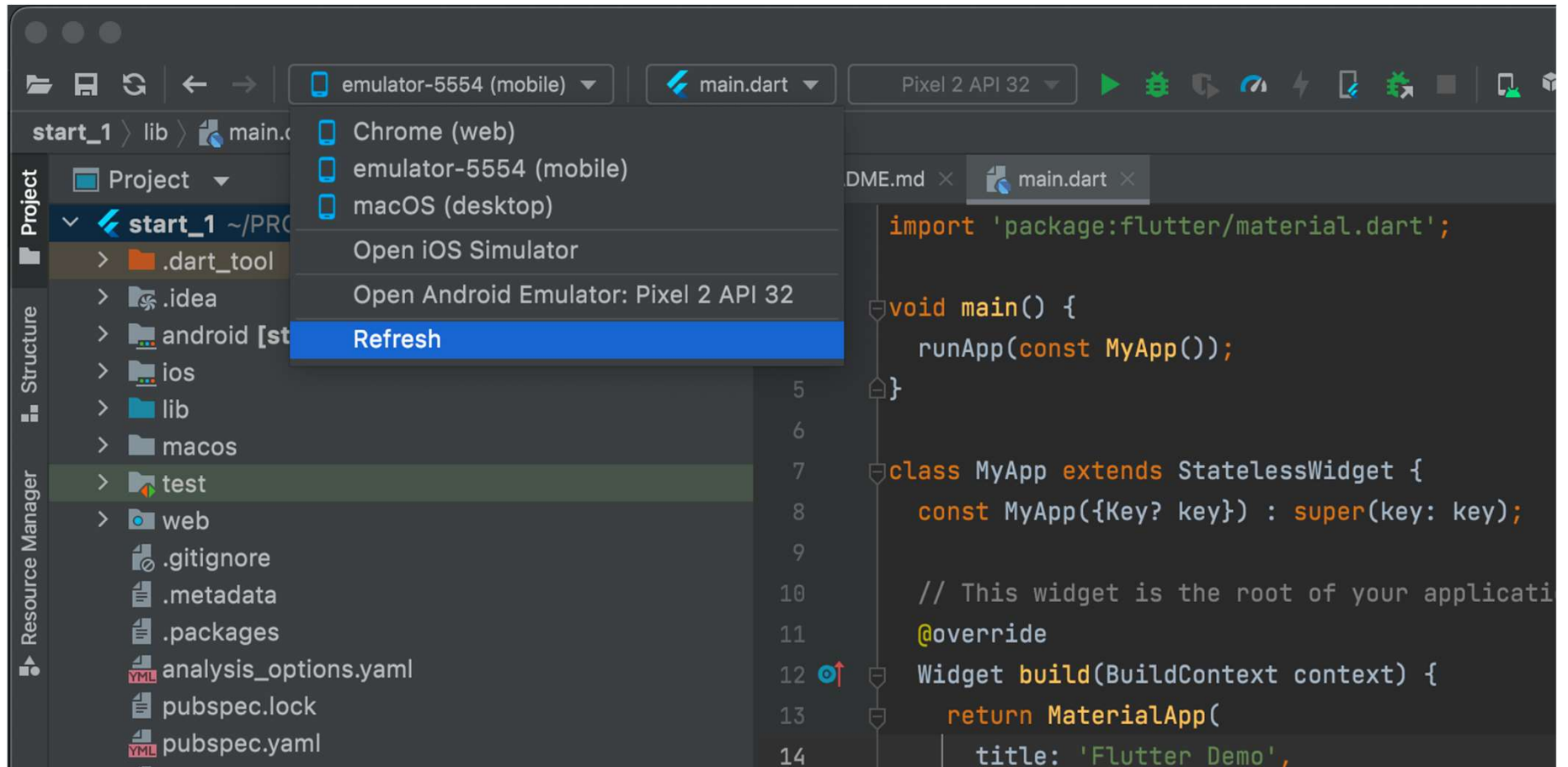


The screenshot shows the 'New Project' dialog in Android Studio. The dialog has a dark theme and contains the following fields and options:

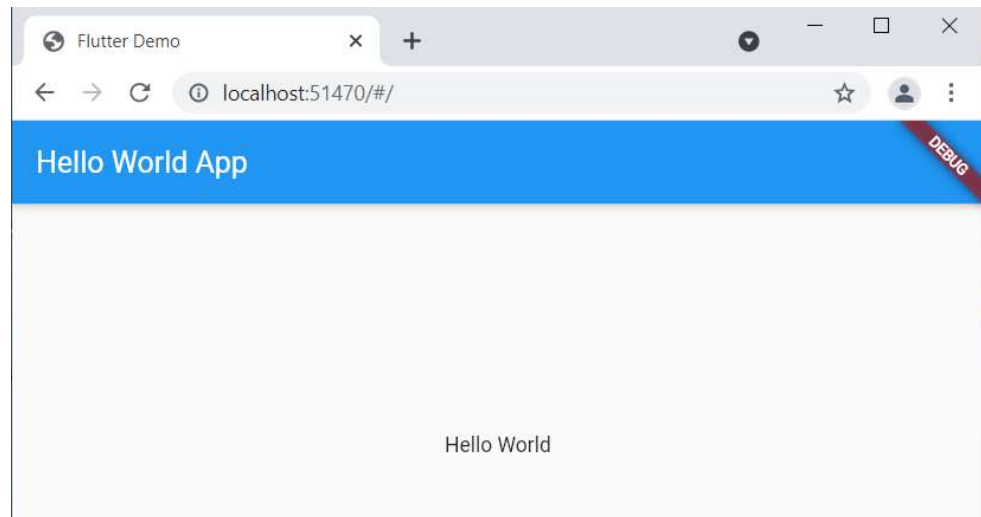
- Project name:** start_1
- Project location:** ~/PROJETS/Flutter/start_1
- Description:** A new Flutter project.
- Project type:** Application
- Organization:** ufc.tatibouet
- Android language:** ☐ Java ☒ Kotlin
- iOS language:** ☐ Objective-C ☒ Swift
- Platforms:** ☒ Android ☒ iOS ☐ Linux ☒ MacOS ☒ Web ☐ Windows

Les plates formes visées réellement dépendent de votre plate forme de développement
[Ici sur MacOS, on peut viser cette plate-forme mais aussi IOS, Android et le WEB.](#)

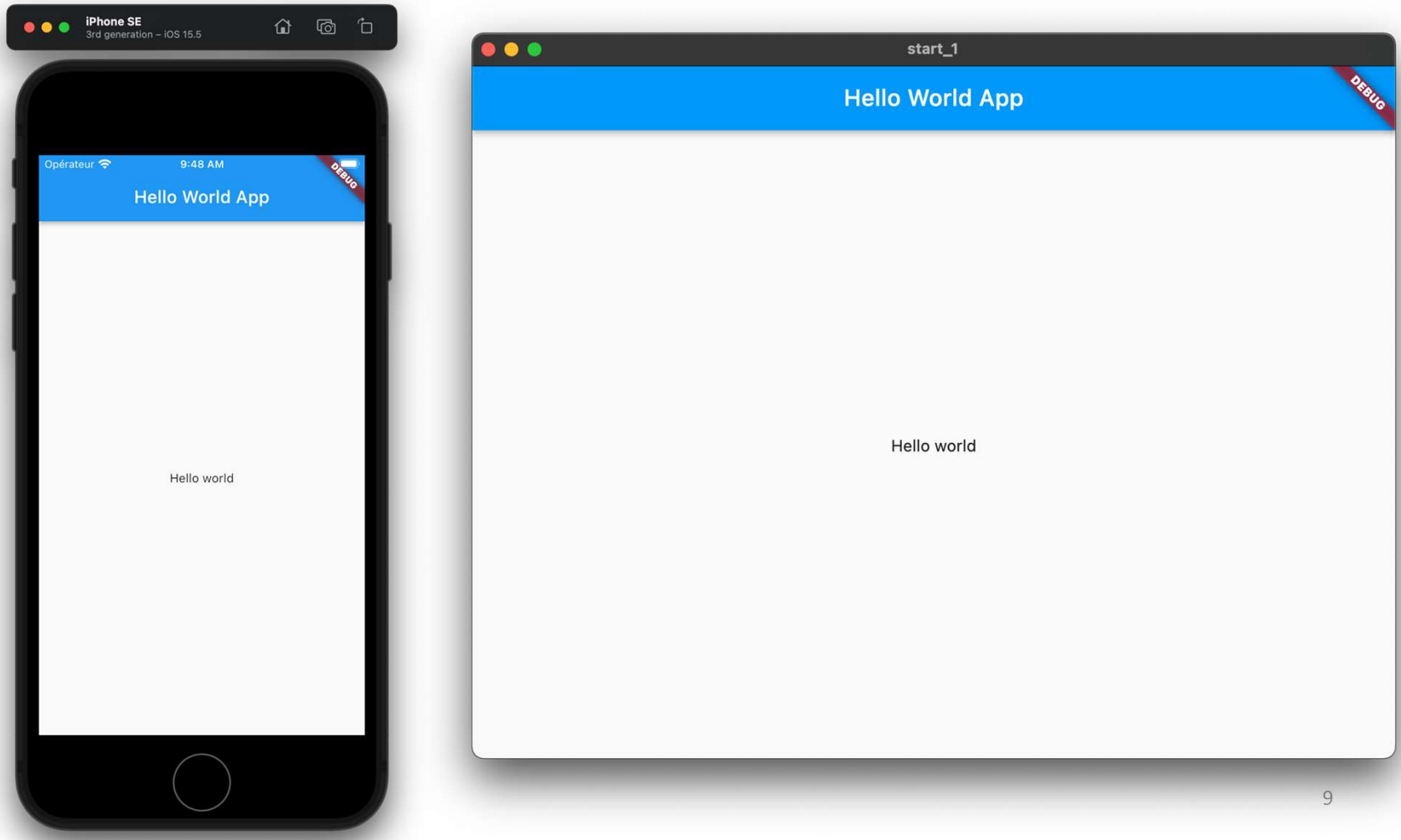
L'application créée avec Android Studio - 2



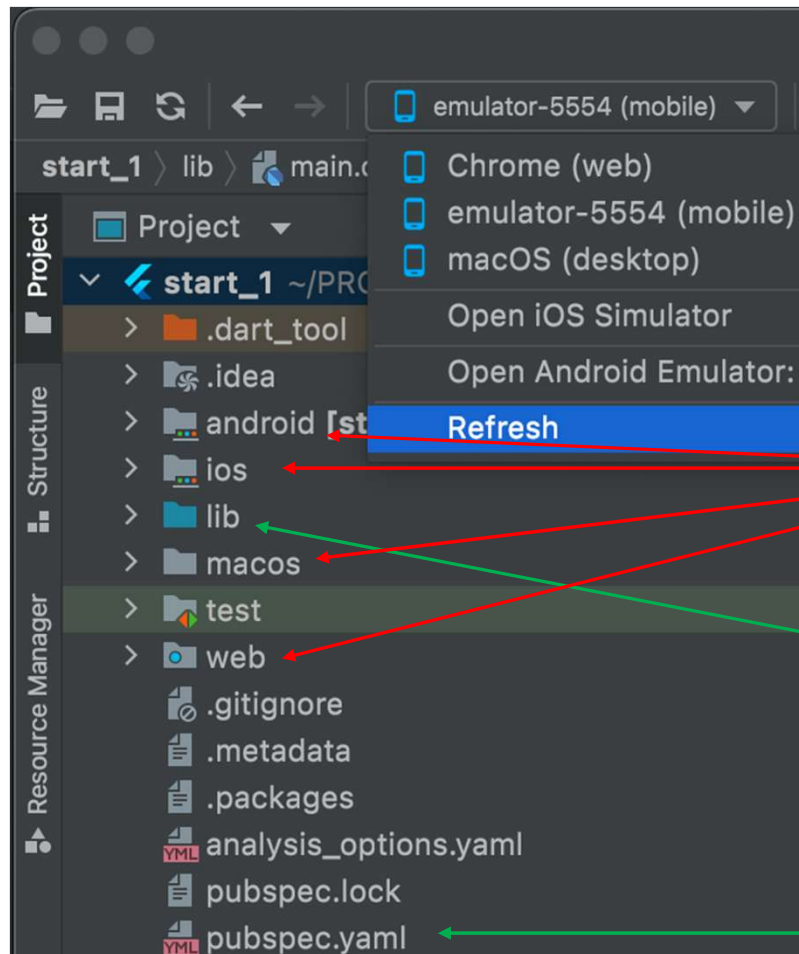
Exécutée sur Android (émulateur) puis sur Chrome



Exécutée sur un émulateur iPhone SE puis sur MacOS



L'application créée avec Android Studio - 2



Projets générés pour les différentes plate-formes ciblées : Normalement on n'y intervient pas mais cela peut arriver au niveau de la mise à jour des fichiers de configuration d'un projet ancien

Le dossier dans lequel est placé votre code Dart/Flutter multi-plateforme

Permet d'indiquer les dépendances, les ressources et le versionnage

Le fichier pubspec.yaml du HelloWorld

```
name: intro
description: A new Flutter project.

publish_to: 'none' # Remove this line if you wish to publish to pub.dev

version: 1.0.0+1

environment:
  sdk: ">=2.12.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  # The following adds the Cupertino Icons font to your application.
  cupertino_icons: ^1.0.2

dev_dependencies:
  flutter_test:
    sdk: flutter

flutter:
  # The following line ensures that the Material Icons font is
  # included with your application
  uses-material-design: true
```

→ Les packages flutter sont publiés ici :
<https://pub.dev/>

- Le fichier est documenté pour indiquer ce que l'on peut y référencer
- Ouvert dans [Android Studio](#) on dispose des commandes [flutter](#) pour mettre à jour ou récupérer des packages que l'on y a ajouté.

Rajouter une dépendance dans pubspec.yaml

- Il suffit de chercher le package approprié ici : <https://pub.dev/>
- Supposons que l'on veuille utiliser une base de données embarquée `sql`
 - Parmi ceux proposés on a le package `sqflite`
 - 3 éléments qui justifient le choix : 100% de popularité, Null safety (conforme aux évolutions de Dart) et Flutter Favorite

sqflite

Flutter plugin for SQLite, a self-contained, high-reliability, embedded, SQL database engine.

v 2.0.3 (12 hours ago)  BSD-2-Clause  

SDK | FLUTTER | PLATFORM | ANDROID | IOS | MACOS

3042 130 100%
LIKES PUB POINTS POPULARITY

- Modifier pubspec.yaml pour indiquer cette nouvelle dépendance

`dependencies:`

`sqflite: ^2.0.3`

- Dans `Android Studio` on dispose des commandes `flutter` pour mettre à jour ou récupérer des packages : `pub get`
 - Sinon en ligne de commandes : `flutter pub add sqflite`
-
- Dans les fichiers sources où le package est utilisé :
`import 'package:sqflite/sqflite.dart' ;`

L'application HelloWorld avec Flutter - 1

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

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

```
class MyApp extends StatelessWidget {  
  // This widget is the root of your application.
```

```
  @override
```

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

```
    return MaterialApp(  
      title: 'Flutter Demo',  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: MyHomePage(),  
    );
```

```
  }  
}
```

La classe est de type `StatelessWidget` :
l'interface utilisateur ne dépend pas
de données variables

La fonction `build` retourne un `Widget`
qui est la racine des widgets de l'application

`build` crée le widget à partir d'autres composants

Le widget retourné est un
composant `MaterialApp` avec :

- Un titre
- Un thème
- Un autre widget qui va être notre écran d'accueil (home page)

L'application HelloWorld avec Flutter - 2

```
class MyHomePage extends StatelessWidget {
```

Notre écran d'accueil est un widget sans état

```
  @override
```

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

```
    return Scaffold(
```

```
      appBar: AppBar(
```

```
        title: Text('Hello World App'),
```

```
      ),
```

```
      body: const Center(
```

```
        child: Text('Hello World'),
```

```
      ),
```

```
    );
```

```
  }
```

```
}
```

Le widget retourné est un Scaffold (widget qui va occuper tout l'espace) avec des propriétés :

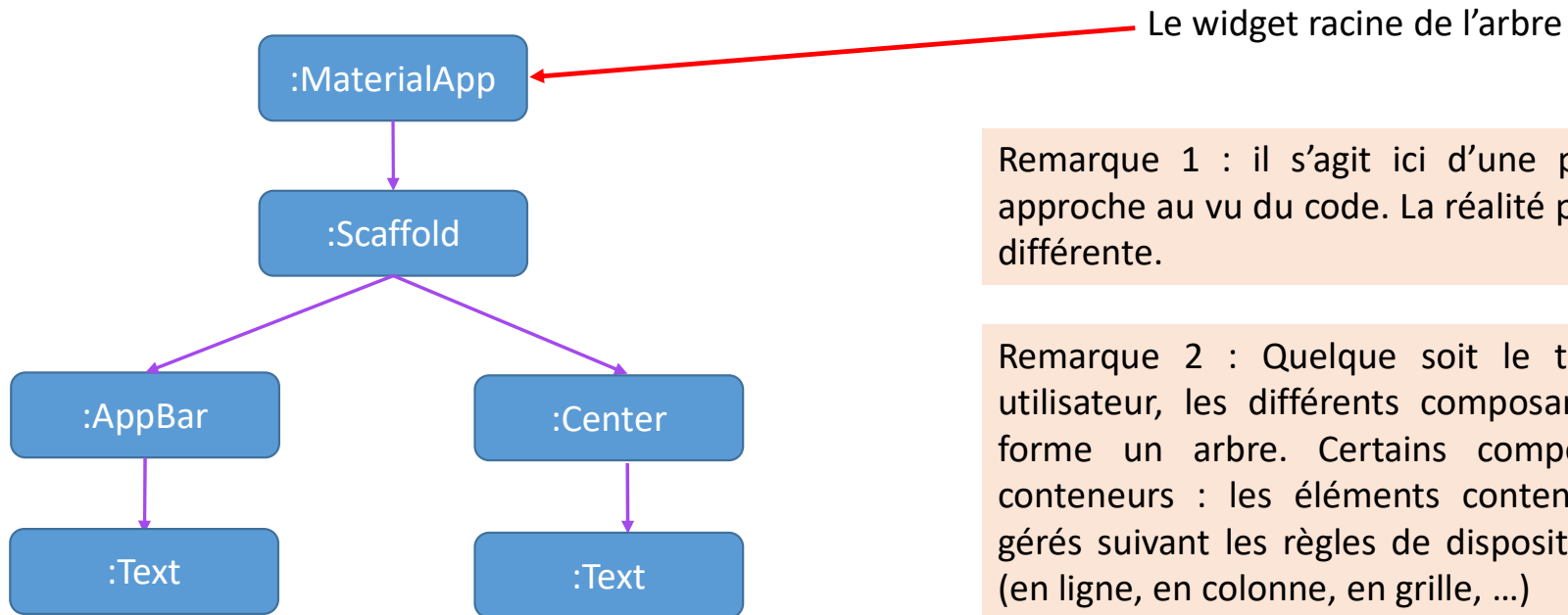
- `appBar` qui possède un titre
- `body` qui est ici un conteneur avec un fils qui est un widget de type `Text`
- `floatingActionButton` absent ici
- ...

StatelessWidget : sans état ce qui convient à des parties d'interfaces utilisateurs qui ne sont pas mises à jour

StatefulWidget : avec état pour les interfaces utilisateurs dont l'affichage dépend de données variables

(un compteur comme dans l'exemple suivant puisque sa valeur affichée va évoluer en fonction des actions de l'utilisateur)

L'arbre des widgets dans l'application HelloWorld



Remarque 1 : il s'agit ici d'une première approche au vu du code. La réalité peut être différente.

Remarque 2 : Quelque soit le toolkit d'interface utilisateur, les différents composants de l'interface forme un arbre. Certains composants sont des conteneurs : les éléments contenus (ou fils) sont gérés suivant les règles de disposition du conteneur (en ligne, en colonne, en grille, ...)

Remarque 3 : Cette organisation arborescente se prête bien à une représentation textuelle arborescente autre que du code. Par exemple de type XML puisque un document XML est basiquement un arbre. XAML de Microsoft en est un exemple avec MAUI

Les paramètres nommés en Dart - 1

```
class Personne {  
  String? _nom ;  
  String? _prenom ;  
  int _age = 0 ;  
  Personne ({required String nom, String? prenom, int age = 0}) {  
    this._nom = nom ;  
    this._prenom = prenom ;  
    this._age = age ;  
  }  
  String toString() {  
    StringBuffer sPersonne = StringBuffer() ;  
    sPersonne.write (_nom);  
    if (_prenom != null) sPersonne.write (" $_prenom") ;  
    if (_age > 0) sPersonne.write(" $_age") ;  
    return sPersonne.toString() ;  
  }  
}
```

Les paramètres sont nommés (la position n'est plus importante, certains ne sont pas utilisés, d'autres sont requis)

```
class PersonneBuilder {  
  Personne build () {  
    return Personne(  
      age: 600,  
      nom: 'Yoda'  
    ) ;  
  }  
}
```

Une instance de personne est créée dans la fonction build et est retournée.
L'âge est fourni en premier et le nom obligatoire en deuxième

Les paramètres nommés en Dart - 2

La création de la personne

```
Personne p = PersonneBuilder().build();
```

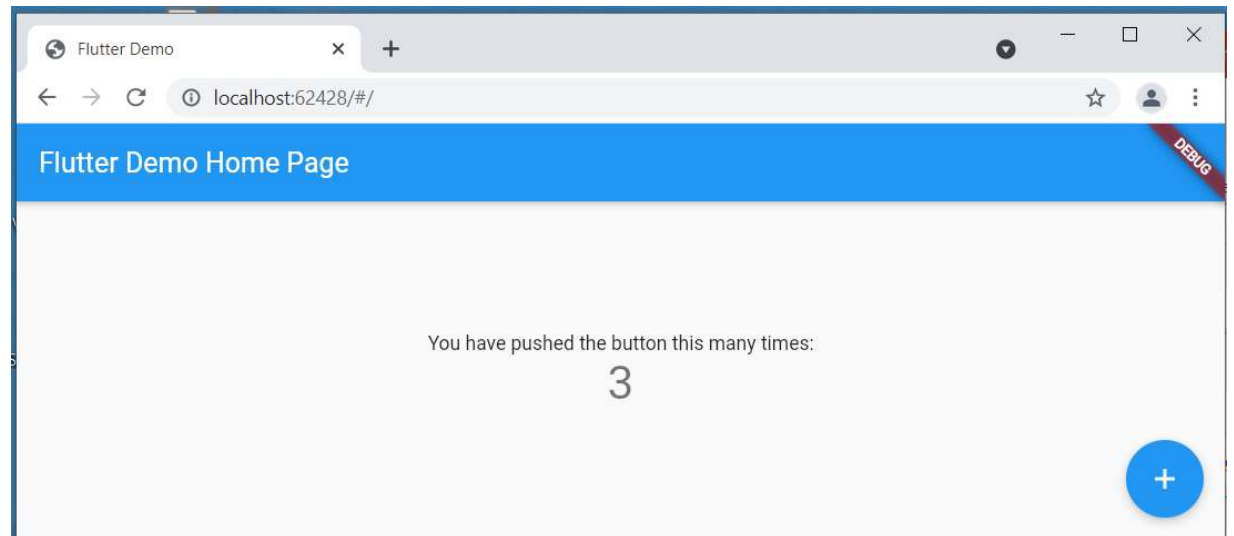
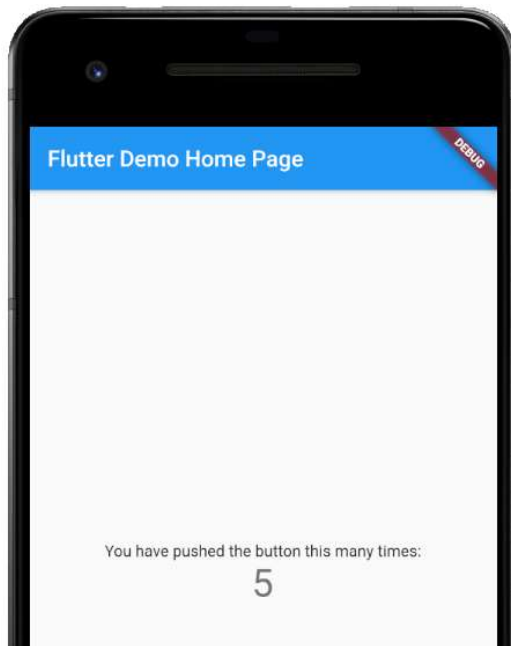
```
children: <Widget>[  
  Text(  
    'Personne : $p',  
    style: Theme.of(context).textTheme.headline4,  
  ),  
],
```

L'affichage de la personne en flutter

ExParamètres nommés en Dart

Personne : Yoda 600

Le Statefull Widget illustré par le compteur



L'application Compteur avec Flutter - 1

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

Pas de changement. La classe est de type `StatelessWidget` : l'interface utilisateur ne dépend pas de données variables

La fonction `build` retourne un Widget de type `MaterialApp` qui est la racine des widgets de l'application

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key? key, required this.title}) : super(key: key);

  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
  // State<StatefulWidget> createState() {return _MyHomePageState(); }
}
```

L'interface utilisateur va être mise à jour en fonction du compteur : la classe est donc de type `StatefulWidget`

L'état est créé ici

Autre façon d'écrire la création de l'état

L'application Compteur avec Flutter - 2

```
class _MyHomePageState extends State<MyHomePage> {  
  int _counter = 0;  
  
  void _incrementCounter() { setState(() {_counter++; }); }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.title),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            Text(  
              'You have pushed the button this many times:',  
            ),  
            Text(  
              '$_counter',  
              style: Theme.of(context).textTheme.headline4,  
            ),  
          ],  
        ),  
      ),  
    ),  
  ),  
}
```

Bruno Tatibouët

Le compteur (privé grâce au `_`)

L'appui sur le bouton flottant va appeler la fonction qui va utiliser `setState` ce qui va permettre à Flutter de réexécuter le `build`.

Le widget retourné est un Scaffold (widget qui va occuper tout l'espace) avec des propriétés :

- `appBar` qui possède un titre
- `body` qui est ici un conteneur avec un fils qui est une colonne contenant des fils : deux de type `Text` dont le compteur
- `floatingActionButton` qui est le bouton flottant avec la propriété `onPressed`

```
floatingActionButton: FloatingActionButton(  
  onPressed: _incrementCounter,  
  tooltip: 'Increment',  
  child: Icon(Icons.add),  
),  
);  
}
```

Les icônes **Material Design** sont disponibles

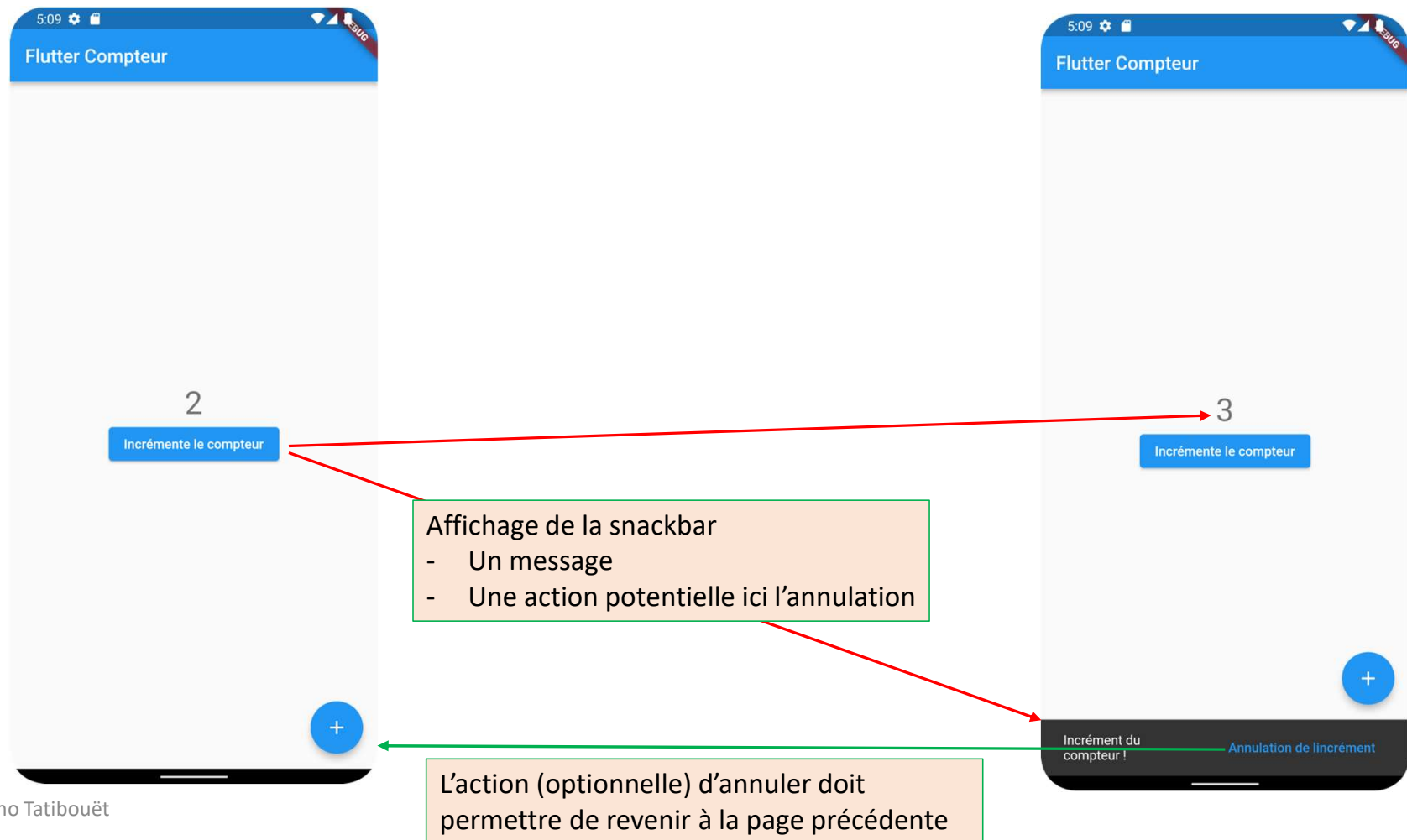
Points importants dans Flutter

- Flutter implémente lui-même les composants graphiques et ne s'appuie pas sur le système hôte
- Les éléments graphiques dans Flutter ne sont rendus/reconstruits que lorsque cela est nécessaire
- Flutter dispose d'un système de rechargement à chaud (Hot Reload) qui permet d'accélérer le développement
- L'interface utilisateur de votre application est codée directement en Dart.
- Les composants visuels différents entre les systèmes (IOS : `cupertino.dart` et Android : `material.dart`) nécessitent de tester le système pour faire du développement spécifique
- L'accès au matériel (Photo, GPS, ...) nécessite :
 - l'installation de packages spécifiques qui ne sont pas intégrés dans Flutter.
 - Le développement spécifique en s'interfaçant avec du code natif

Les widgets de flutter

- Widget : contraction de Windowing Gadgets
- Quels sont les composants graphiques disponibles dans flutter ?
 - On dispose d'une galerie permettant de les parcourir
 - Elle est disponible ici : <https://gallery.flutter.dev/>
 - Cette galerie est disponible comme une application sur le store pour IOS et Android

La Snackbar : L'application compteur revisitée - 0



La Snackbar : L'application compteur revisitée - 1

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

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

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter Demo',  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: MyHomePage(title: 'Flutter Compteur'),  
    );  
  }  
}
```

Pas de changements ici !!

Pas de changement. La classe est de type `StatelessWidget` : l'interface utilisateur ne dépend pas de données variables

La fonction `build` retourne un Widget de type `MaterialApp` qui est la racine des widgets de l'application

```
class MyHomePage extends StatefulWidget {  
  MyHomePage({Key? key, required this.title}) : super(key: key);
```

L'interface utilisateur va être mise à jour en fonction du compteur : la classe est donc de type `StatefulWidget`

```
  final String title;
```

```
  @override
```

```
  _MyHomePageState createState() => _MyHomePageState();
```

L'état est créé ici

```
  // State<StatefulWidget> createState() {return _MyHomePageState(); }
```

Autre façon d'écrire la création de l'état

```
}
```


La Snackbar : L'application compteur revisitée - 2

```
class _MyHomePageState extends State<MyHomePage> {  
  int _counter = 0;  
  
  void _incrementCounter() { setState(() {_counter++; }); }  
  
  void _undoIncrementCounter() { setState(() {_counter--; }); }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.title),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            Text(  
              '$_counter',  
              style: Theme.of(context).textTheme.headline4,  
            ),  
            ElevatedButton(  
              // Le texte et l'action  
            ),  
          ],  
        ),  
      ),  
    ),  
  ),  
}
```

Le compteur (privé grâce au _)

L'appui sur le bouton flottant va appeler la fonction qui va utiliser `setState` ce qui va permettre à Flutter de réexécuter le `build`.

L'annulation de l'incrément

Le widget retourné est un Scaffold :

- `appBar` qui possède un titre
- `body` qui est ici un conteneur avec un fils qui est une colonne contenant des fils : un de type `Text` et un de type `ElevatedButton`
- `floatingActionButton` qui est le bouton flottant avec la propriété `onPressed`

```
floatingActionButton: FloatingActionButton(  
  onPressed: _incrementCounter,  
  tooltip: 'Increment',  
  child: Icon(Icons.add),  
),  
);  
}
```

Les icônes `Material Design` sont disponibles

La Snackbar : L'application compteur revisitée - 3

```
ElevatedButton(  
  onPressed: () {  
    _incrementCounter() ;  
    final SnackBar _snack = SnackBar(  
      content: Text('Incrément du compteur !'),  
      // duration: const Duration(milliseconds: 1500),  
      action: SnackBarAction(  
        label: 'Annulation de l\'incrément',  
        textColor: Colors.blue,  
        onPressed: _undoIncrementCounter),  
    );  
    ScaffoldMessenger.of(context)  
      ..removeCurrentSnackBar()  
      ..showSnackBar(_snack);  
  },  
  child: Text('Incrémente le compteur')  
)
```

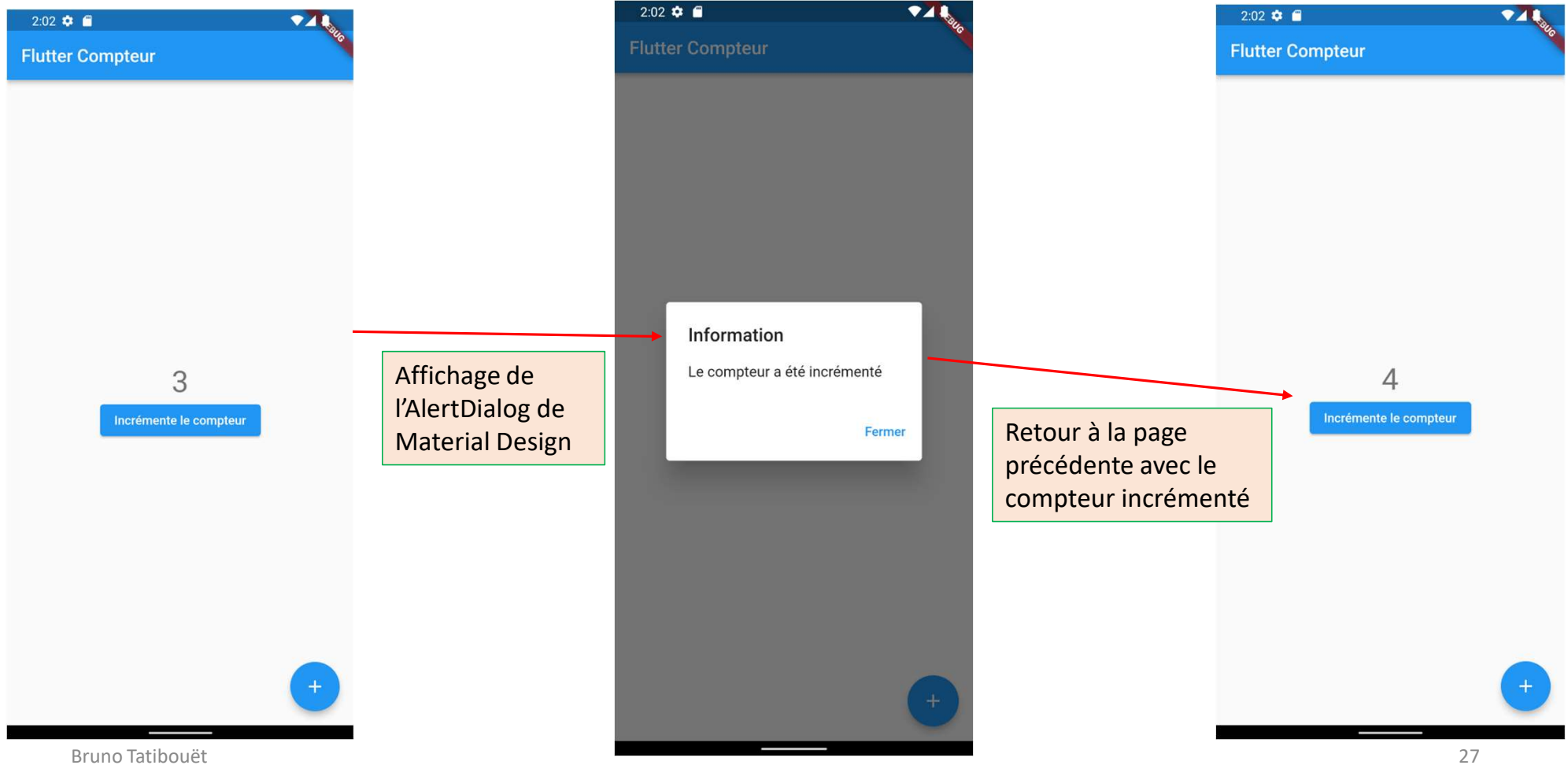
Lorsque le bouton est pressé :

- Il est incrémenté
- Une `SnackBar` est créée et visualisée
 - Elle affiche un message d'incrément du compteur
 - Elle dispose d'une action pour annuler l'incrément

Il est possible d'écrire plus simplement ceci s'il n'y a pas d'action associée à la `SnackBar` :

```
ScaffoldMessenger.of(context)  
  .showSnackBar(SnackBar(content : Text('Incrément du compteur !')));
```

AlertDialog : La nécessité d'une confirmation - 0



AlertDialog : La nécessité d'une confirmation - 1

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

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

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter Demo',  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: MyHomePage(title: 'Flutter Compteur'),  
    );  
  }  
}
```

Pas de changements ici !!

Pas de changement. La classe est de type `StatelessWidget` : l'interface utilisateur ne dépend pas de données variables

La fonction `build` retourne un Widget de type `MaterialApp` qui est la racine des widgets de l'application

```
class MyHomePage extends StatefulWidget {  
  MyHomePage({Key? key, required this.title}) : super(key: key);
```

L'interface utilisateur va être mise à jour en fonction du compteur : la classe est donc de type `StatefulWidget`

```
  final String title;
```

```
  @override
```

```
  _MyHomePageState createState() => _MyHomePageState();
```

L'état est créé ici

```
  // State<StatefulWidget> createState() {return _MyHomePageState(); }
```

Autre façon d'écrire la création de l'état

```
}
```

AlertDialog : La nécessité d'une confirmation - 2

```
class _MyHomePageState extends State<MyHomePage> {  
  int _counter = 0;  
  
  void _incrementCounter() { setState(() { _counter++; }); }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.title),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            Text(  
              '$_counter',  
              style: Theme.of(context).textTheme.headline4,  
            ),  
            ElevatedButton(  
              //    
            ),  
          ],  
        ),  
      ),  
    );  
  }  
}
```

Bruno Tatibouët

Le compteur (privé grâce au _)

L'appui sur le bouton flottant va appeler la fonction qui va utiliser `setState` ce qui va permettre à Flutter de réexécuter le `build`.

Le widget retourné est un Scaffold :

- `appBar` qui possède un titre
- `body` qui est ici un conteneur avec un fils qui est une colonne contenant des fils : un de type `Text` et un de type `ElevatedButton`
- `floatingActionButton` qui est le bouton flottant avec la propriété `onPressed`

```
floatingActionButton: FloatingActionButton(  
  onPressed: _incrementCounter,  
  tooltip: 'Increment',  
  child: Icon(Icons.add),  
),  
);  
}
```

Les icônes `Material Design` sont disponibles

Voir diapo suivante

AlertDialog : La nécessité d'une confirmation - 3

```
ElevatedButton(
  onPressed: () {
    _incrementCounter();
    showDialog(
      context: context,
      barrierDismissible: false,
      builder: (BuildContext context) {
        return AlertDialog(
          title: Text('Information'),
          content: Text("Le compteur a été incrémenté"),
          actions: <Widget>[
            TextButton(
              onPressed: () {
                Navigator.of(context).pop();
              },
              child: const Text('Fermer')
            ),
          ],
        );
      },
    );
  },
  child: Text('Incrémente le compteur')
),
```

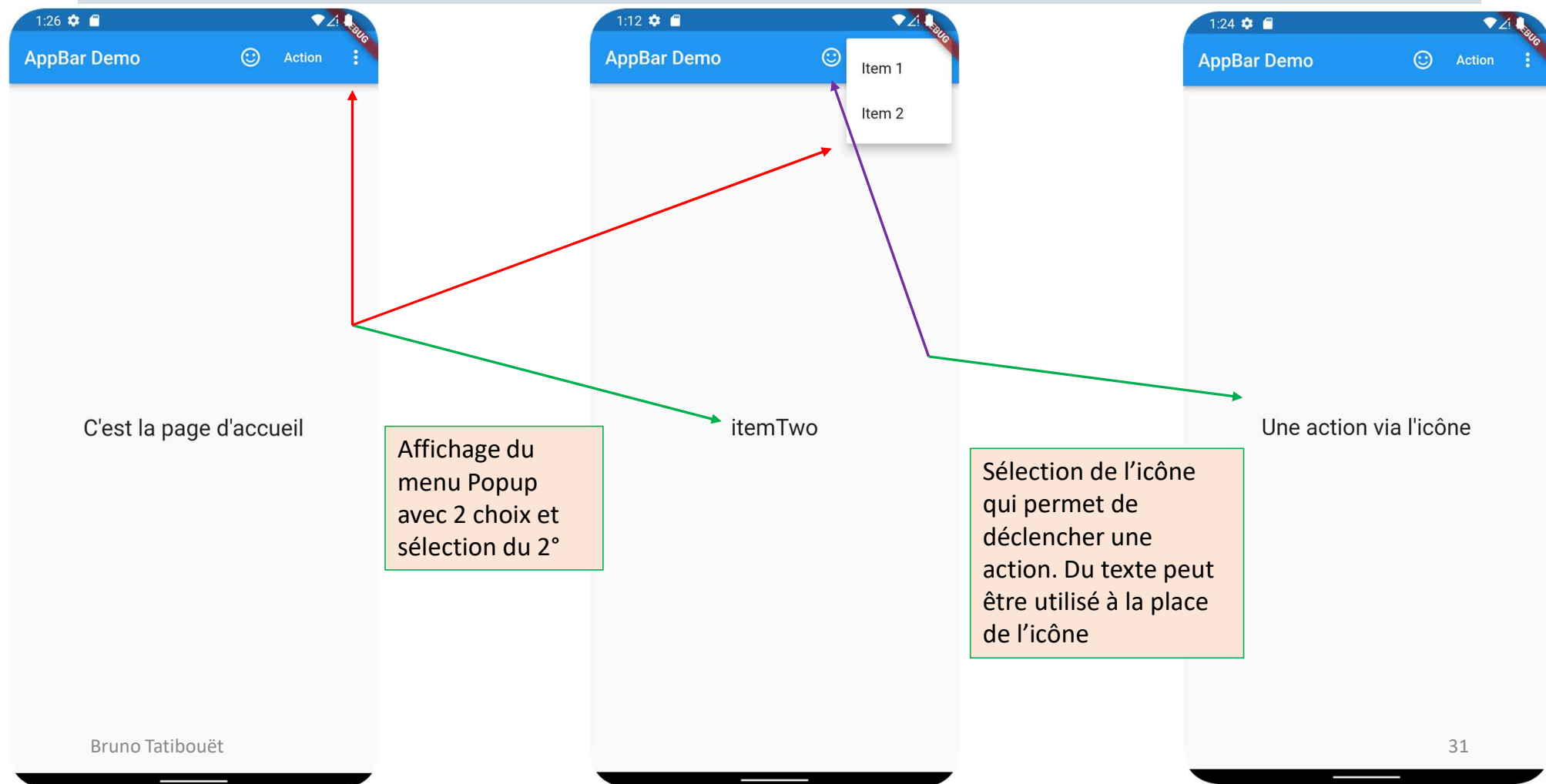
Lorsque le bouton est pressé :

- Il est incrémenté
- Une `AlertDialog` est visualisée qui attend une confirmation via le `TextButton`
- On navigue dans la pile des pages pour retourner à la page précédente mise à jour.

La boîte de dialog est spécifique à Material Design.

Pour avoir une boîte plus spécifique à iOS, il faut une `CupertinoAlertDialog` et tester le système sur lequel s'exécute l'application.

AppBar : Utiliser la barre d'application - 0



AppBar : Utiliser la barre d'application - 1

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'AppBar Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Compteur'),
    );
  }
}
```

Pas de changements notables ici !!

Pas de changement. La classe est de type `StatelessWidget` : l'interface utilisateur ne dépend pas de données variables

La fonction `build` retourne un Widget de type `MaterialApp` qui est la racine des widgets de l'application

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key? key, required this.title}) : super(key: key);

  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
  // State<StatefulWidget> createState() {return _MyHomePageState(); }
}
```

L'interface utilisateur va être mise à jour en fonction du compteur : la classe est donc de type `StatefulWidget`

L'état est créé ici

Autre façon d'écrire la création de l'état


AppBar : Utiliser la barre d'application - 2

```
enum Menu { itemOne, itemTwo}
```

```
class _MyStatefulWidgetState extends State<MyStatefulWidget> {  
  String _textePage = 'C'est la page d'accueil';  
  void modifierMessage(String message) {  
    setState(() {  
      _textePage = message;  
    });  
  }  
}
```

Le message (privé grâce au _)

La mise à jour du message fonction va utiliser `setState` ce qui va permettre à Flutter de réexécuter le `build`.

```
@override  
Widget build(BuildContext context) {  
  final ButtonStyle style =  
    TextButton.styleFrom(primary: Theme.of(context).colorScheme.onPrimary);  
  return Scaffold(  
    appBar: AppBar(  
        
    ),  
    body: Center(  
      child: Text(  
        _textePage,  
        style: TextStyle(fontSize: 24),  
      ),  
    ),  
  );  
}
```

Du au fond (ici bleu) de l'AppBar. Il faut prendre la couleur de fond du thème

Le widget retourné est un Scaffold :

- `appBar` qui possède un titre, les actions symbolisées par du texte ou des icônes, le menu et des éléments de navigation que l'on verra après
- `body` qui est ici un conteneur avec un fils de type Text permettant d'afficher le message

voir diapo suivante

AppBar : Utiliser la barre d'application - 3

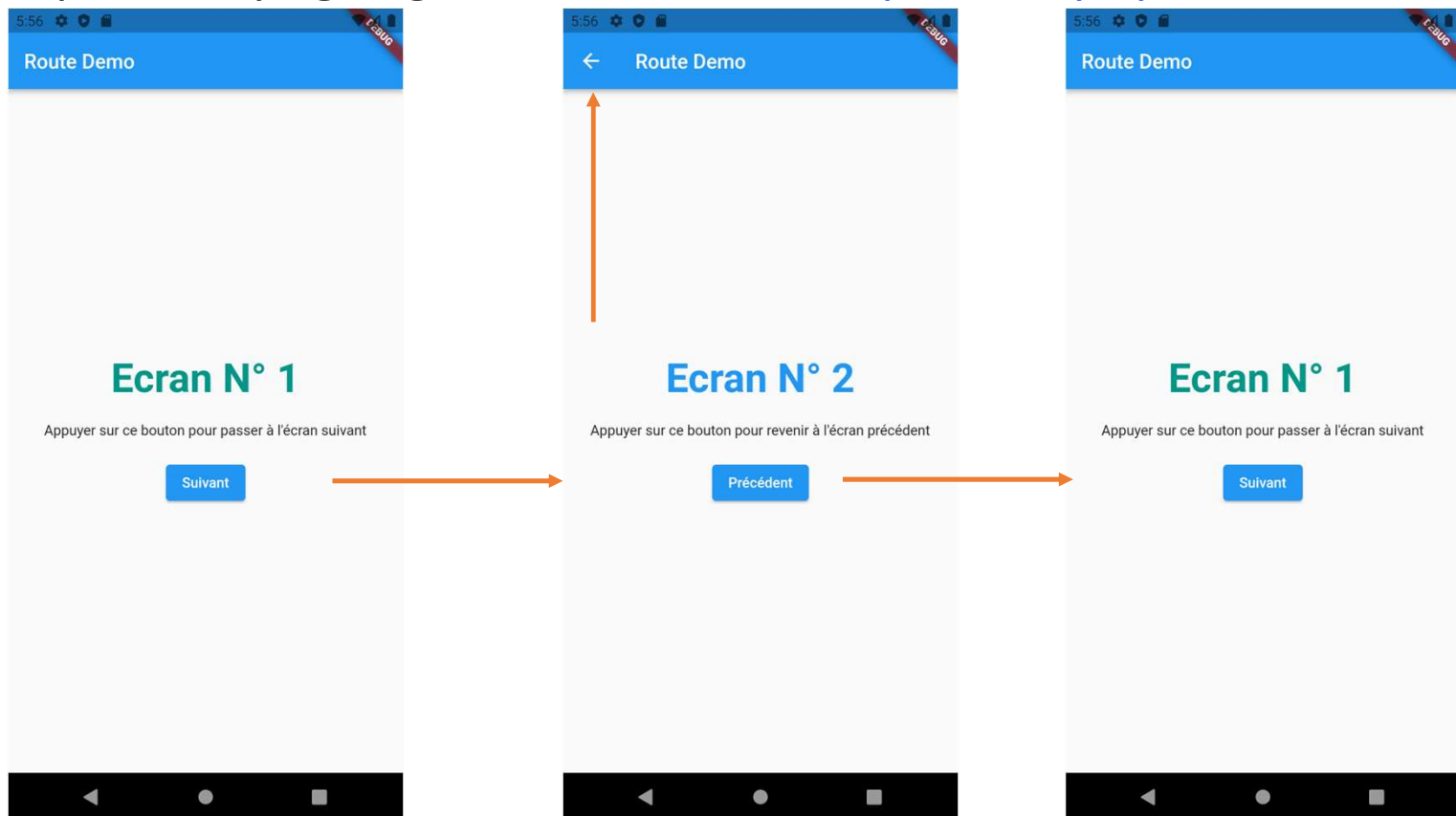
```
title: Text('AppBar Demo'),  
actions: <Widget>[  
  IconButton(  
    icon: const Icon(Icons.sentiment_satisfied_sharp),  
    onPressed: () {  
      modifierMessage('Une action via l\'icône');  
    },  
  ),  
  TextButton( ... ),  
  PopupMenuButton<Menu>(  
    onSelected: (Menu item) {  
      modifierMessage(item.name);  
    },  
    itemBuilder: (BuildContext context) => <PopupMenuEntry<Menu>>[  
      const PopupMenuItem<Menu>(  
        value: Menu.itemOne,  
        child: Text('Item 1'),  
      ),  
      const PopupMenuItem<Menu>(  
        value: Menu.itemTwo,  
        child: Text('Item 2'),  
      ),  
    ],  
  ),  
,  
],
```

Les éléments de l'AppBar:

- Le titre
- Les actions
 - IconButton
 - TextButton
 - PopupMenuNutton

Changer d'interface utilisateur – 6.1

- On peut passer d'une page à une autre avec le widget **Navigator** qui gère la pile des pages grâce aux fonctions **push** et **pop**

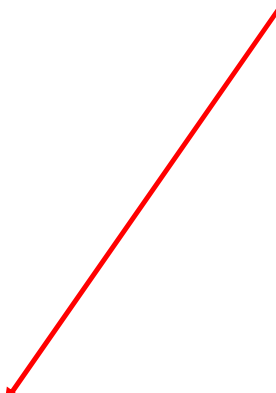


Changer d'interface utilisateur – 6.2.1

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

```
class MyApp extends StatelessWidget {  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: MyHomePage(title: 'Route Demo'),  
    );  
  }  
}
```

Notre écran d'accueil est un widget sans état

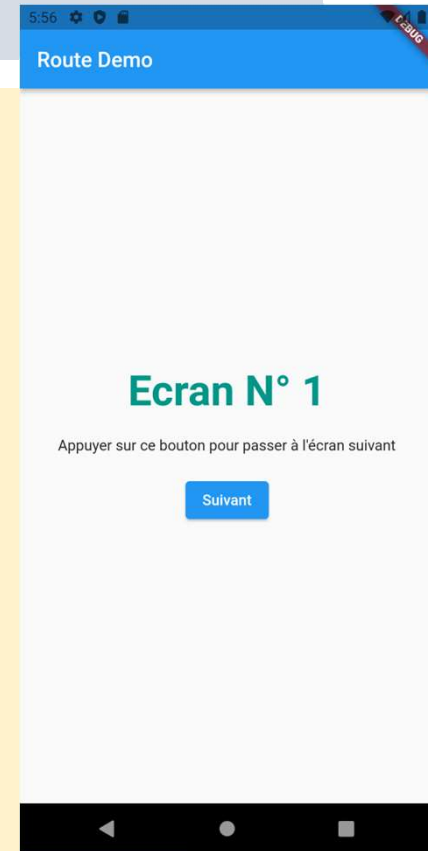


```
class MyHomePage extends StatelessWidget {  
  MyHomePage({Key? key, required this.title}) : super(key: key);  
  
  final String title;  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(title),  
      ),  
    );  
  }  
}
```

Changer d'interface utilisateur – 6.2.2

```
body: Center(  
  child: Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: <Widget>[  
      Text(  
        "Ecran N° 1",  
        style: TextStyle(fontWeight: FontWeight.bold,  
          fontSize: 40,  
          color: Colors.teal),  
      ),  
      Padding(padding: EdgeInsets.only(bottom: 20)),  
      Text("Appuyer sur ce bouton pour passer à l'écran suivant"),  
      Padding(padding: EdgeInsets.only(bottom: 20)),  
      ElevatedButton(  
        child: Text('Suivant'),  
        onPressed: () {  
          Navigator.push(  
            context,  
            MaterialPageRoute(  
              builder: (BuildContext context) => MySecondPage (title : title)));  
        }  
      ),  
    ],  
  ),  
);
```

Le widget Navigator permet de créer une nouvelle page qui sera mise au sommet de la pile.



Changer d'interface utilisateur – 6.2.3

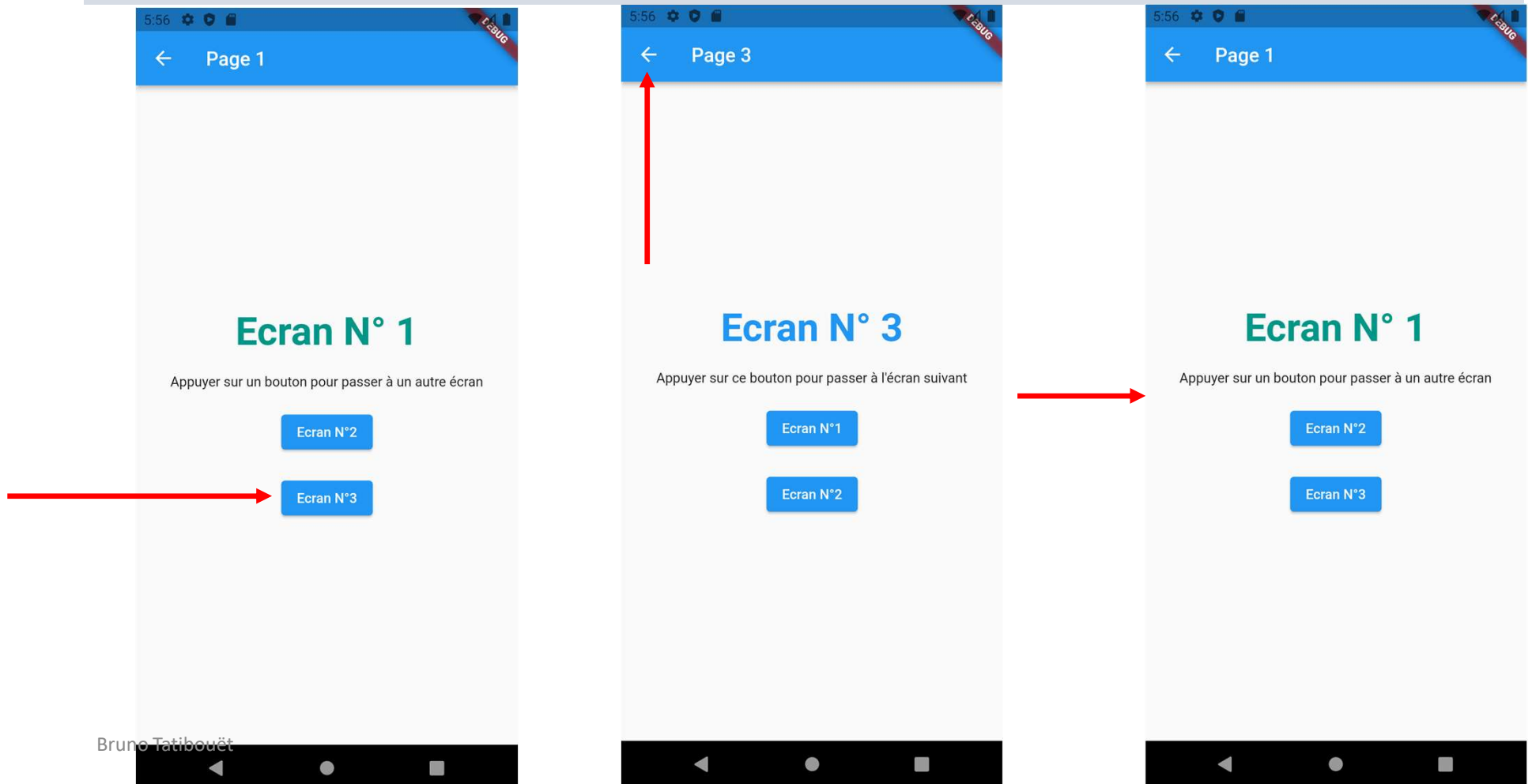
```
class MySecondPage extends StatelessWidget {  
  MySecondPage ({required this.title});  
  
  final String title ;  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('$title'),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            ...  
            Text("Appuyer sur ce bouton pour revenir à l'écran précédent"),  
            Padding(padding: EdgeInsets.only(bottom: 20)),  
            ElevatedButton(  
              child: Text('Précédent'),  
              onPressed: () {  
                Navigator.pop(context) ;  
              },  
            ),  
          ],  
        ),  
      ),  
    );  
  }  
}
```

Bruno Tatibouët

Le widget Navigator permet de dépiler pour revenir à notre page



La navigation avec les routes – 7.1



La navigation avec les routes – 7.2

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

```
class MyApp extends StatelessWidget {  
  // This widget is the root of your application.
```

```
  @override
```

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

```
    return MaterialApp(  
      home: MyHomePage(title: 'Routes nommées'),
```

```
      initialRoute : '/route1',
```

La première page est définie dans `initialRoute`

```
      routes: {
```

```
        '/route1': (BuildContext context) => MyHomePage(title: 'Page 1'),
```

```
        '/route2': (BuildContext context) => MySecondPage(title: 'Page 2'),
```

```
        '/route3': (BuildContext context) => MyThirdPage(title: 'Page 3'),
```

```
      },
```

```
    );
```

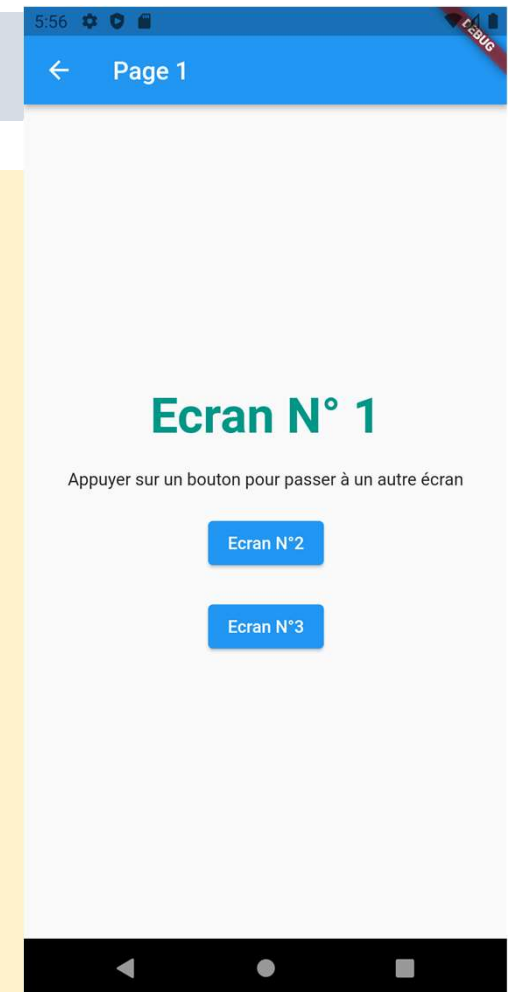
```
  }
```

```
}
```

Les différentes routes sont définies au préalable sous la forme d'URL et utilisables via le widget `Navigator` par la fonction `pushNamed`

La navigation avec les routes – 7.3.1

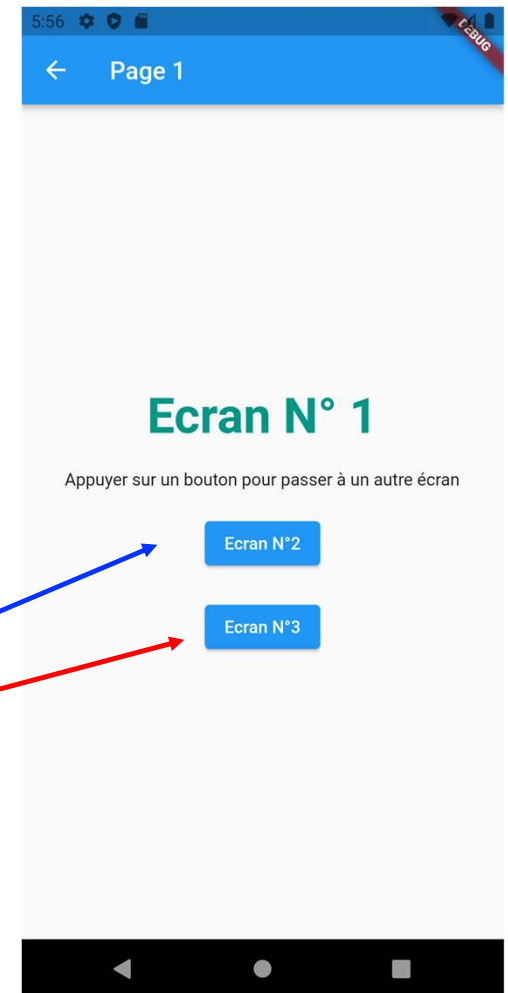
```
class MyHomePage extends StatelessWidget {  
  MyHomePage({Key? key, required this.title}) : super(key: key);  
  
  final String title;  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(title),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            Text(  
              "Ecran N° 1",  
              style: TextStyle(fontWeight: FontWeight.bold,  
                fontSize: 40,  
                color: Colors.teal),  
            ),  
          ],  
        ),  
      ),  
    );  
  }  
}
```



La navigation avec les routes – 7.3.2

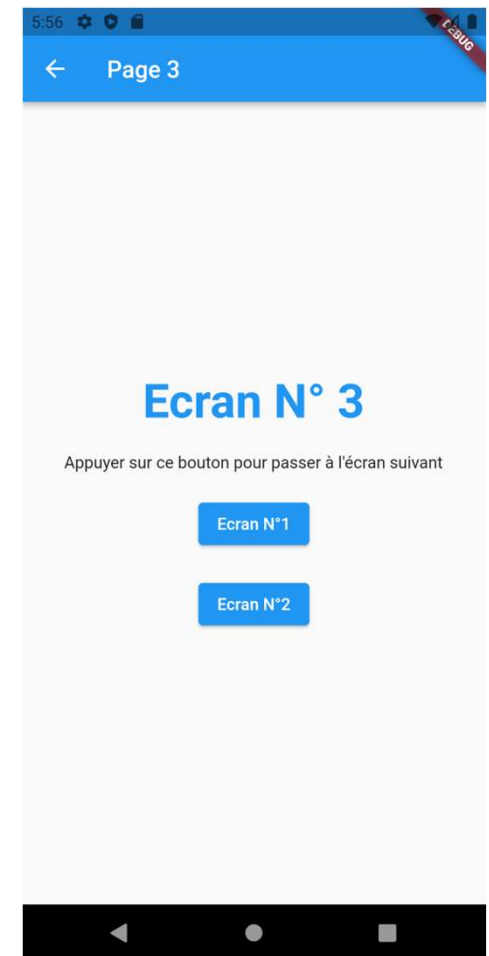
```
Padding(padding: EdgeInsets.only(bottom: 20)),
Text("Appuyer sur un bouton pour passer à un autre écran"),
Padding(padding: EdgeInsets.only(bottom: 20)),
ElevatedButton(
  child: Text('Ecran N°2'),
  onPressed: () {
    Navigator.pushNamed(context, '/route2');
  }
),
Padding(padding: EdgeInsets.only(bottom: 20)),
ElevatedButton(
  child: Text('Ecran N°3'),
  onPressed: () {
    Navigator.pushNamed(context, '/route3');
  }
),
```

Le widget Navigator permet de passer à la page indiquée par la route



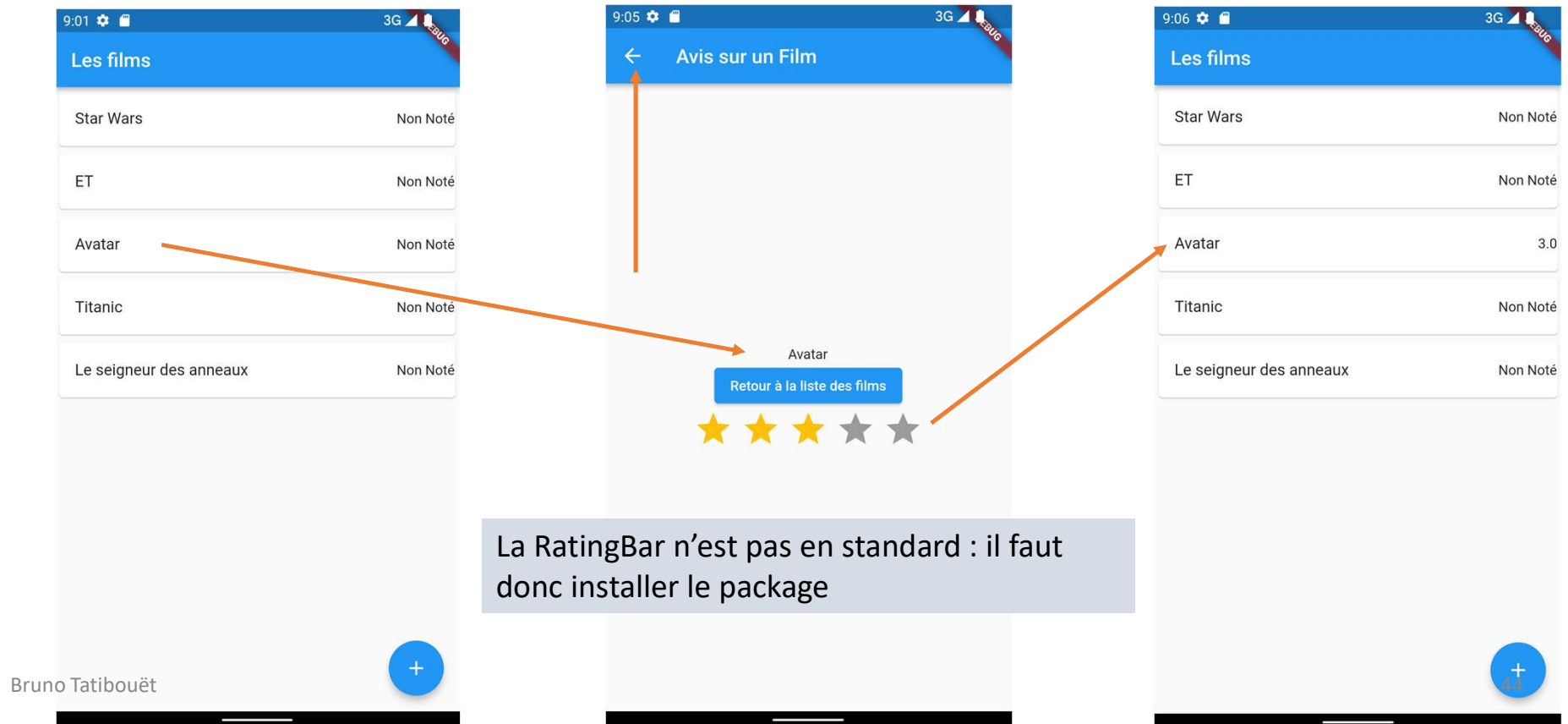
La navigation avec les routes – 7.4

- Les deux autres pages (Ecran 2 et Ecran 3) correspondant aux 2 routes ont un code qui est presque identique à celui de la première page.
 - Seules changent les routes (la 1 et la 3 pour la page2, la 1 et la 2 page 3) auxquelles on peut accéder



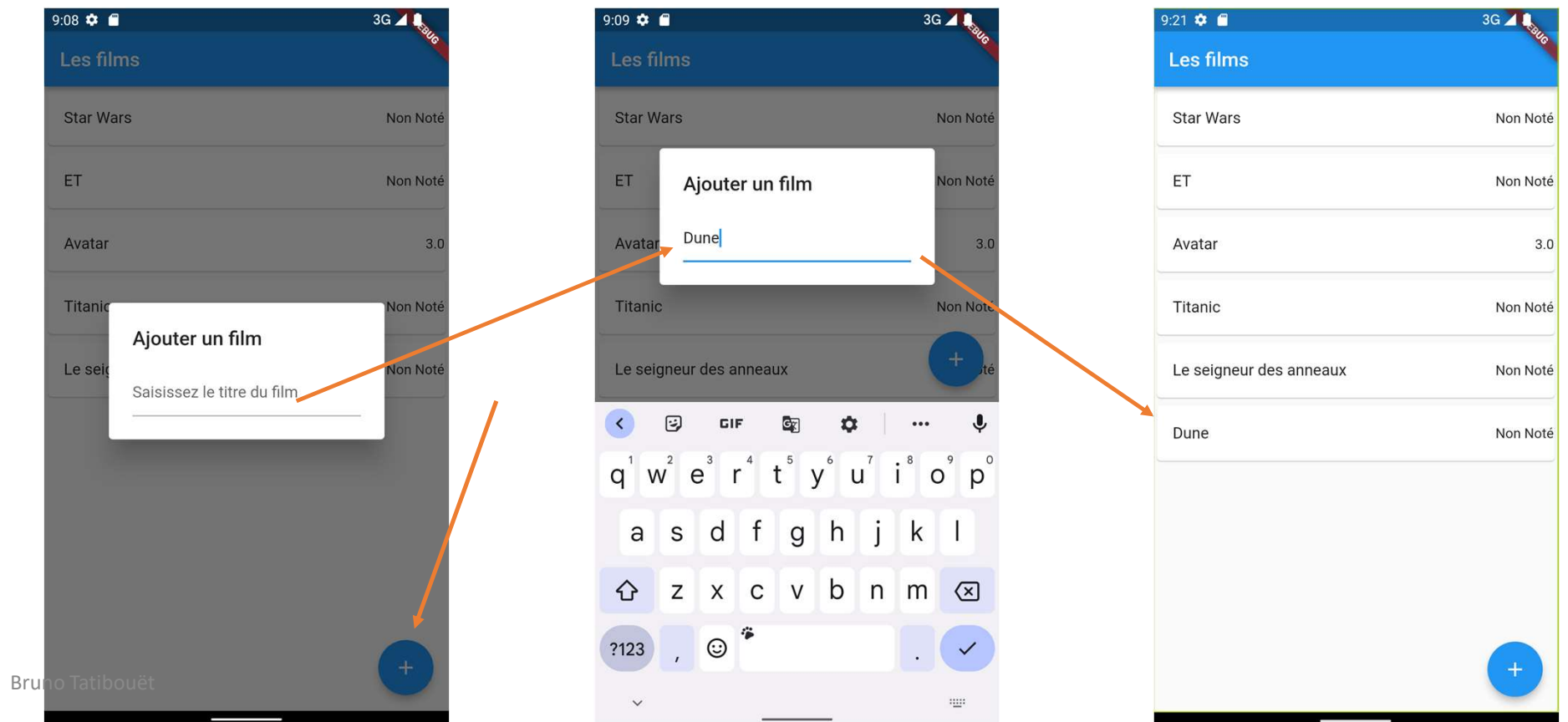
Un exemple avec deux pages – 7.1

- L'idée est d'avoir une liste de films avec pour chacun la possibilité de donner un avis sous la forme d'une note de 0 à 5.



Un exemple avec deux pages – 7.2

- On peut ajouter des films via le floating action button (FAB) qui effectue la saisie via une AlertDialog.



Les données globales manipulées – 7.3

```
String titreFilm = '' ;
```

```
class Film {  
    double _noteFilm = -1 ;  
    double get note { return _noteFilm ;}  
    set note(double valeur) {_noteFilm = valeur ;}  
}
```

- Les films ne contiennent qu'une note initialement à -1
- La représentation se fait sous la forme d'une paire clé (le titre du film) – valeur (les infos sur le film)
- Les paires sont stockées dans une Map

```
Map<String, Film> tableDesFilms = {  
    // key : value  
    'Star Wars' : Film (),  
    'ET' : Film (),  
    'Avatar' : Film (),  
    'Titanic' : Film (),  
    'Le seigneur des anneaux' : Film ()  
};
```

L'interface utilisateur – 7.4

```
void main() {  
  runApp(MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext ctxt) {  
    return MaterialApp(  
      home: const ListFilmDisplay(title: 'Les films'),  
    );  
  }  
}
```

2 pages sont utilisables :

- La liste des Films [ListFilmDisplay](#) avec la note éventuelle
- L'évaluation d'un film dans l'écran ou la page [FilmRatingPage](#)

Les données affichées sont modifiables : la note ou l'ajout d'un film

```
class ListFilmDisplay extends StatefulWidget {  
  const ListFilmDisplay({Key? key, required this.title}) :  
    super(key: key);  
  final String title;  
  @override  
  ListFilmStateView createState() {  
    return ListFilmStateView(title);  
  }  
}
```

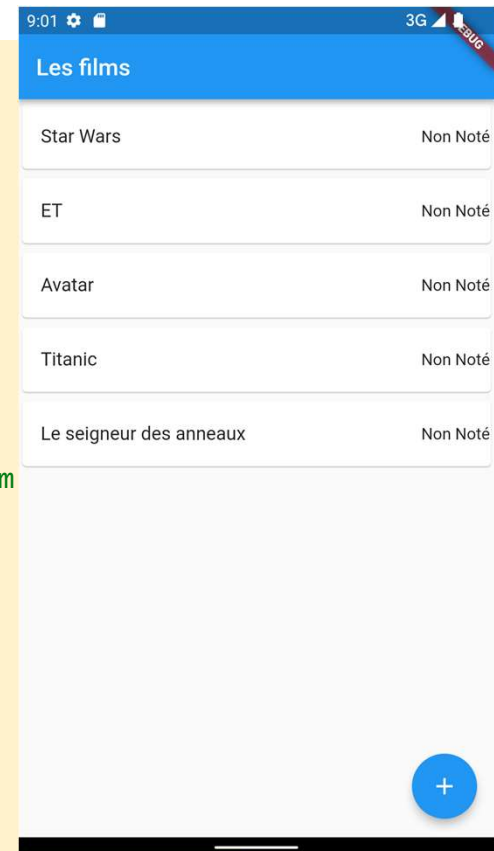
La première page – 7.5.1

```
class ListFilmStateView extends State<ListFilmDisplay> {  
  
  final String title;  
  ListFilmStateView(this.title);  
  
  List<String> listeDesFilms = tableDesFilms.keys.toList();  
  
  Future<void> _navigationToRatingPage(BuildContext context) async {  
    final result = await Navigator.push(  
      context,  
      // Create the SelectionScreen in the next step.  
      MaterialPageRoute(builder: (context) => const FilmRatingPage(title: 'Avis sur un Film'  
    ));  
    Film film = tableDesFilms[titreFilm]! ;  
    film.note = result as double ;  
    setState(() {});  
  }  
  
  void insertFilm(String titreFilm) {  
    tableDesFilms[titreFilm] = Film();  
    setState(() {  
      listeDesFilms = tableDesFilms.keys.toList() ;  
    });  
  }  
}
```

Le résultat n'est pas à coup sur immédiat

Le widget Navigator permet de créer une nouvelle page qui sera mise au somme de la pile.

Le setState va permettre la reconstruction de l'interface puisque l'on vient d'insérer une nouvelle donnée



La première page – 7.5.2

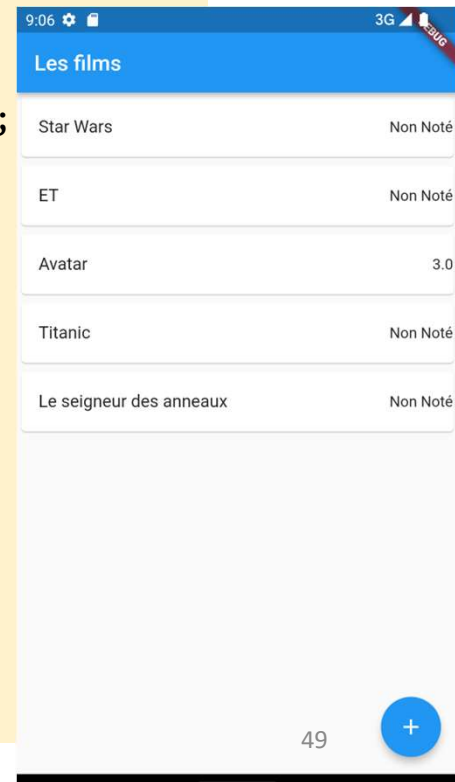
```
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(title: Text(title)),  
    body: ListView.builder(  
      itemCount: listeDesFilms.length,  
      itemBuilder: (BuildContext context, int index) {  
        String noteFilm = (tableDesFilms[listeDesFilms[index]]!.note != -1) ?  
          tableDesFilms[listeDesFilms[index]]!.note.toString() : 'Non Noté';  
        return Card(  
          child: Row(  
            children: [  
              Expanded(  
                child: ListTile(  
                  title: Text(listeDesFilms[index]),  
                  onTap: () {  
                    titreFilm = listeDesFilms[index] ;  
                    _navigationToRatingPage(context) ;  
                  }  
                ),  
              ),  
              Text(noteFilm)  
            ],  
          ),  
        ),  
      ),  
    ),  
  ),  
);
```

Le widget `ListView` permet de gérer une source de données de taille quelconque en gérant l'espace destiné à la visualisation de celles-ci

Si la note est -1 on affiche Non Noté sinon on affiche la note

Appui sur un titre de film

On affiche la page de notation du film.

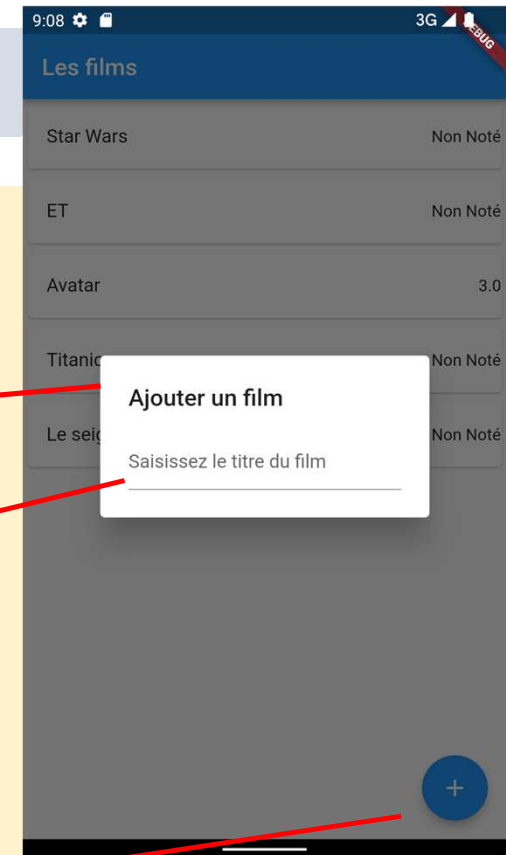


La première page – 7.5.3

```
floatingActionButton: FloatingActionButton(  
  onPressed: () {  
    showDialog(  
      context: context,  
      barrierDismissible: false,  
      builder: (BuildContext context) {  
        return AlertDialog(  
          title: const Text('Ajouter un film'),  
          content: TextField(  
            decoration: const InputDecoration(  
              hintText: "Saisissez le titre du film"),  
              onSubmitted: (value) {  
                insertFilm(value);  
                Navigator.of(context).pop();  
              },  
            );  
          });  
        },  
        tooltip: 'ajouter un film',  
        child: const Icon(Icons.add),  
      ),  
    );  
  },  
);
```

L'entrée est
validée

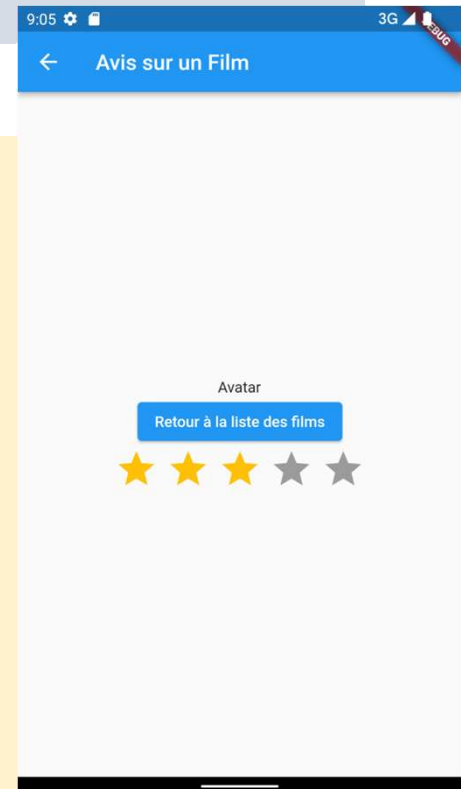
On revient à la première page



La deuxième page – 7.6.1

```
class FilmRatingPage extends StatefulWidget {  
  const FilmRatingPage({Key? key, required this.title}) : super(key: key);  
  final String title;  
  @override  
  RatingFilmStateView createState() {  
    return RatingFilmStateView(title);  
  }  
}  
  
class RatingFilmStateView extends State<FilmRatingPage> {  
  final String title;  
  double note = -1;  
  RatingFilmStateView(this.title);  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(title),  
      ),  
    ),  
  }  
}
```

Permet de mémoriser la
valeur du rating



La deuxième page – 7.6.2

```
body: Center(  
  child: Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: <Widget>[  
      Text(titreFilm),  
      ElevatedButton(  
        child: Text('Retour à la liste des films'),  
        onPressed: () {  
          Navigator.pop(context, note);  
        }  
      ),  
      RatingBar.builder(  
        initialRating: 3,  
        minRating: 1,  
        direction: Axis.horizontal,  
        allowHalfRating: true,  
        itemCount: 5,  
        itemPadding: const EdgeInsets.symmetric(horizontal: 4.0),  
        itemBuilder: (context, _) => const Icon(Icons.star, color: Colors.amber, ),  
        onRatingUpdate: (rating) {  
          note = rating ;  
        }  
      ),  
    ],  
  ),  
);
```

Le widget Navigator permet de dépiler pour revenir à la première page

La note est mise à jour avec la valeur courante de rating

