

# La navigation dans Flutter

# Table des matières

<b>I. Navigation dans son ensemble</b>	<b>3</b>
A. Navigation simple en Flutter .....	3
B. Partager des données entre les pages .....	6
C. Naviguer avec des routes nommées .....	9
<b>II. Exercice : Quiz</b>	<b>12</b>
<b>III. Animations dans la navigation</b>	<b>13</b>
A. Principe des transitions en Flutter .....	13
B. Animer un widget pendant la transition .....	16
<b>IV. Exercice : Quiz</b>	<b>18</b>
<b>V. Variantes de routes de navigation</b>	<b>19</b>
A. Pages de dialogue .....	19
B. Barre de navigation .....	21
<b>VI. Exercice : Quiz</b>	<b>24</b>
<b>VII. Essentiel</b>	<b>25</b>
<b>VIII. Auto-évaluation</b>	<b>26</b>
A. Exercice : .....	26
B. Test .....	26
<b>Solutions des exercices</b>	<b>27</b>

## I. Navigation dans son ensemble

### Contexte

La grande majorité des applications mobile actuelles ont de multiples pages pour afficher différents types d'informations. En Flutter, ces pages sont nommés **routes** et sont reliées entre elles par la navigation. C'est ce procédé qui va gérer le passage entre les pages, l'animation de transition ou même le format dans lequel la page sera affichée.

Cette fonctionnalité en Flutter est donc indispensable à n'importe quel développeur voulant créer une application digne de ce nom. Il est aussi important de savoir quels moyens laisser à l'utilisateur pour naviguer dans l'application, car l'ergonomie et le *user design* font partie des points les plus importants à étudier lorsqu'on veut développer sur le support mobile.

La navigation est un vaste sujet, et qu'importe l'application que vous allez développer, elle fera définitivement partie de celle-ci. Il est donc important de maîtriser au mieux cette fonctionnalité pour obtenir le meilleur rendu possible pour votre futur projet, professionnel ou non.

Mais pourquoi utiliser la navigation en Flutter ? Quelles sont les différents moyens de navigation disponibles ? Comment savoir lequel choisir pour optimiser l'expérience utilisateur ? Comment choisir l'animation pour la transition entre les différentes routes ?

### A. Navigation simple en Flutter

#### Définition

##### La classe Navigator

La classe `Navigator` gère un ensemble de pages enfants avec un système de pile. Les pages enfants de la classe `Navigator` se nomment *routes*. `Navigator` va donc gérer les passages entre ces mêmes pages, que ce soit pour naviguer vers une nouvelle page ou pour le retour en arrière.

##### Une route

Une route correspond au **système de passage entre les pages**. Une route est donc un *widget* qui sera instancié au moment du changement de page. Pour les plus confirmés, sur Android, une route correspond à une `Activity` et sur iOS, elle correspond à un `ViewController`.

#### Méthode Le fonctionnement de la classe Navigator avec son système de pile

`Navigator` gère donc **une pile** de routes, comme expliqué précédemment.

Qu'est-ce qu'un système de pile ? C'est une file, un tableau, où l'on stocke dans l'ordre les éléments souhaités par rapport au moment où ils entrent dans la pile.

Le haut de la pile constitue le début de celle-ci, le premier élément ajouté. Le bas est donc la fin de la pile, le dernier élément.

La première route de votre application sera en haut de la pile. Si vous allez sur une nouvelle page pour une quelconque raison depuis la première page, une route sera ajoutée dans la pile.

Pour revenir à la page précédente, il suffit donc de supprimer la dernière route de la pile, et Flutter se chargera du reste.

**Exemple** Un exemple simple d'une page menant à une autre grâce à Navigator

```

1  import 'package:flutter/material.dart';
2
   Run | Debug | Profile
3  void main() {
4      runApp(MaterialApp(
5          title: 'Exemple-1.2',
6          home: FirstPage(),
7      )); // MaterialApp
8  }
9
10 class FirstPage extends StatelessWidget {
11     @override
12     Widget build(BuildContext context) {
13         return Scaffold(
14             appBar: AppBar(
15                 title: Text('First Route'),
16             ), // AppBar
17             body: Center(
18                 child: ElevatedButton(
19                     child: Text('Open second Route'),
20                     onPressed: () {
21                         // First button pressed
22                     },
23                 ), // ElevatedButton
24             ), // Center
25         ); // Scaffold
26     }
27 }

```

```
29 class SecondPage extends StatelessWidget {
30   @override
31   Widget build(BuildContext context) {
32     return Scaffold(
33       appBar: AppBar(
34         title: Text("Second Route"),
35       ), // AppBar
36       body: Center(
37         child: ElevatedButton(
38           onPressed: () {
39             // Second button pressed
40           },
41           child: Text('Go back to first Route'),
42         ), // ElevatedButton
43       ), // Center
44     ); // Scaffold
45   }
46 }
47
```

Dans l'exemple qui suit, 2 pages sont présentes : **FirstPage** et **SecondPage**. Ces 2 pages contiennent 1 bouton chacune, et c'est sur ce bouton que l'on va assigner la fonction de changement de page. **FirstPage** est automatiquement placée en premier dans la pile de la classe **Navigator**.

Pour effectuer le passage à **SecondPage**, il faut utiliser la fonction **Navigator.push** et mettre en argument la route voulue. Dans l'exemple, nous utiliserons **MaterialPageRoute**, une route préconfigurée basique.

```
20   onPressed: () {
21     Navigator.push(
22       context,
23       MaterialPageRoute(builder: (context) => SecondPage()),
24     );
25   },
```

```
onPressed: () {
  Navigator.pop(context);
},
```

Voici le rendu de cet exemple sur un écran de téléphone :



Nous verrons plus loin dans le cours les différentes routes disponibles en Flutter et comment les personnaliser.

## B. Partager des données entre les pages

### Comment envoyer des données dans la prochaine page

Envoyer des données à la page voulue fonctionne un peu comme envoyer des données à une fonction. En fait, une page étant un *widget*, celle-ci est donc une **classe**. Et en Flutter, nous pouvons attribuer des fonctions **constructeur** aux classes pour les instancier et ajouter des arguments.

Cela fonctionne de la même façon pour les pages : il suffit en théorie de rajouter une variable à votre classe, puis de créer la fonction `constructeur` et de mettre cette même variable en argument de votre constructeur.

Nous verrons une utilisation plus approfondie dans l'exemple.

#### **Méthode** Comment retourner de la donnée depuis la page actuelle

Dans la partie précédente, nous avons vu les fonctions `Navigator.push()` et `Navigator.pop()`. Ce sont finalement ces 2 fonctions qui nous permettront d'envoyer et de retourner des données depuis une page :

- La fonction `Navigator.pop()` permet de revenir à la page précédente, elle possède aussi un argument optionnel qui est la donnée que vous voulez retourner.
- La fonction `Navigator.push()` retourne donc le résultat envoyé dans la fonction `Navigator.pop()`.

Le type de variable du résultat est de votre choix, vous pouvez donc passer en argument un tableau, ou même une classe de votre choix, pour retourner plusieurs données.

**Exemple 2 méthodes sur 2 routes simples**

Reprenons l'exemple précédent et modifions-le.

```

10 class FirstPage extends StatelessWidget {
11   @override
12   Widget build(BuildContext context) {
13     return Scaffold(
14       appBar: AppBar(
15         title: Text('First Page'),
16       ), // AppBar
17       body: Center(
18         child: Column(mainAxisAlignment: MainAxisAlignment.center, children: [
19           ElevatedButton(
20             child: Text('Open second Page with text 1'),
21             onPressed: () async {
22               final int result = await Navigator.push(
23                 context,
24                 MaterialPageRoute(
25                   builder: (context) => SecondPage(text: "Text sent is 1")), // MaterialPageRoute
26               );
27               ScaffoldMessenger.of(context)
28                 ..removeCurrentSnackBar()
29                 ..showSnackBar(SnackBar(content: Text("The result is $result")));
30             },
31           ), // ElevatedButton
32           ElevatedButton(
33             child: Text('Open second Page with text 2'),
34             onPressed: () async {
35               final int result = await Navigator.push(
36                 context,
37                 MaterialPageRoute(
38                   builder: (context) => SecondPage(text: "Text sent is 2")), // MaterialPageRoute
39               );
40               ScaffoldMessenger.of(context)
41                 ..removeCurrentSnackBar()
42                 ..showSnackBar(SnackBar(content: Text("The result is $result")));
43             },
44           ), // ElevatedButton
45         ]), // Column // Center
46       ); // Scaffold
47     }
48   }

```

Nous avons donc ici 2 boutons différents dans **FirstPage**, qui enverront en argument à **SecondPage** 2 textes différents en fonction du bouton appuyé.

Voici la modification dans **SecondPage**, ou nous avons ajouté plusieurs éléments :

```

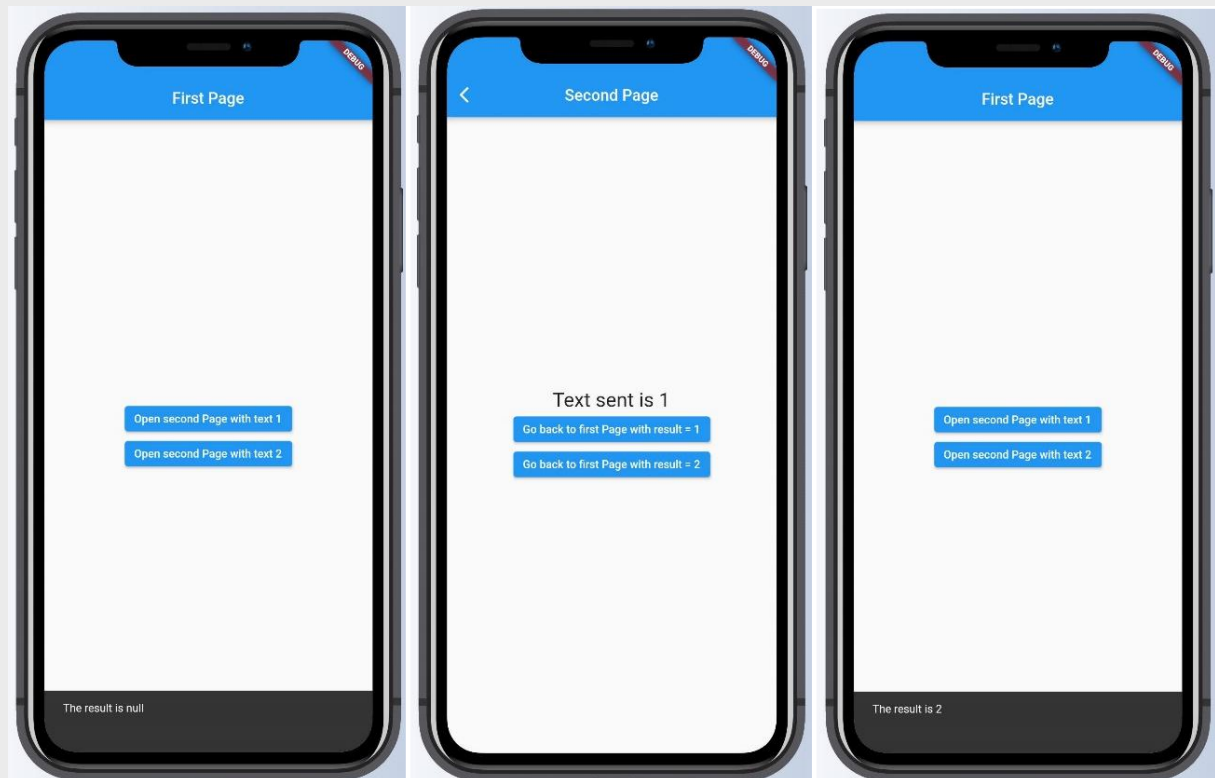
50 class SecondPage extends StatelessWidget {
51   final String text;
52
53   const SecondPage({Key key, @required this.text}) : super(key: key);
54
55   @override
56   Widget build(BuildContext context) {
57     return Scaffold(
58       appBar: AppBar(
59         title: Text("Second Page"),
60       ), // AppBar
61       body: Center(
62         child: Column(mainAxisAlignment: MainAxisAlignment.center, children: [
63           Text(
64             this.text,
65             style: TextStyle(fontSize: 25),
66           ), // Text
67           ElevatedButton(
68             onPressed: () {
69               Navigator.pop(context, 1);
70             },
71             child: Text('Go back to first Page with result = 1'),
72           ), // ElevatedButton
73           ElevatedButton(
74             onPressed: () {
75               Navigator.pop(context, 2);
76             },
77             child: Text('Go back to first Page with result = 2'),
78           ), // ElevatedButton
79         ]), // Column
80     )); // Center // Scaffold
81   }
82 }

```

- Le plus important est l'ajout du constructeur, qui nous permettra d'ajouter un argument obligatoire lors de la définition de la classe, **this.text**. L'argument a été ajouté, il faut aussi ajouter une variable propre à la classe, ici la **String text**. Cette **String** sera donc utilisable dans la globalité de la classe.
- Un **Widget Text** a été ajouté, qui contiendra **this.text** en fonction du bouton appuyé dans **FirstPage**.
- Enfin, un deuxième bouton est présent ici pour différencier les 2 façons de revenir à la route précédente. Sur le code de **FirstPage**, ce résultat sera enregistré dans la variable **result**, et sera par la suite affiché à l'aide d'une simple **SnackBar**.



Voici le rendu de cet exemple sur un écran de téléphone :



Sur le premier *screenshot*, nous pouvons voir que le résultat est *null*. Cela est dû au fait que l'utilisateur a utilisé la flèche de retour au lieu d'appuyer sur un des boutons bleus.

#### **Complément**    **Pour aller plus loin**

Cette fonctionnalité est surtout intéressante à utiliser avec les pages **StatefulWidget**, car les données affichées peuvent être dynamiquement changées entre les pages.

## **C. Naviguer avec des routes nommées**

### **Quand utiliser ce type de navigation et pourquoi ?**

Vous avez précédemment appris à naviguer vers une nouvelle page en créant une route et en l'envoyant à la classe **Navigator**.

Si, dans votre application, vous devez naviguer vers la même page à plusieurs occasions, la solution serait de définir cette même page en route nommée, et de l'utiliser pour la navigation.

Nous allons voir plus en détail comment la réaliser en code avec l'exemple suivant.

**Exemple** Route avec la navigation nommée

```

3  void main() {
4      runApp(
5          MaterialApp(
6              title: 'Named Routes Demo',
7              initialRoute: '/',
8              routes: {
9                  '/': (context) => FirstPage(),
10                 '/second': (context) => SecondPage(),
11             },
12         ), // MaterialApp
13     );
14 }
15
16 class FirstPage extends StatelessWidget {
17     @override
18     Widget build(BuildContext context) {
19         return Scaffold(
20             appBar: AppBar(
21                 title: const Text('First Screen'),
22             ), // AppBar
23             body: Center(
24                 child: ElevatedButton(
25                     onPressed: () {
26                         Navigator.pushNamed(context, '/second');
27                     },
28                     child: const Text('Launch screen'),
29                 ), // ElevatedButton
30             ), // Center
31         ); // Scaffold
32     }
33 }

```

```
35 class SecondPage extends StatelessWidget {
36   @override
37   Widget build(BuildContext context) {
38     return Scaffold(
39       appBar: AppBar(
40         title: const Text('Second Screen'),
41       ), // AppBar
42       body: Center(
43         child: ElevatedButton(
44           onPressed: () {
45             Navigator.pop(context);
46           },
47           child: const Text('Go back!'),
48         ), // ElevatedButton
49       ), // Center
50     ); // Scaffold
51   }
52 }
53
```

**Méthode**

La marche à suivre est donc la suivante :

- Il faut premièrement préciser les routes possibles et la route initiale de l'application dans l'objet **MaterialApp**.
- Les 2 fonctions qui permettront le passage entre les routes sont **Navigator.pushNamed()** et **Navigator.pop()**. La fonction **pushNamed()** prend donc en argument le nom de la route en **String**, et la fonction **Navigator.pop()** fonctionne exactement comme dans l'exemple de la sous-partie « Navigation simple en Flutter » : elle permet de revenir à la route précédente.

**Exemple**

Voici le rendu de cet exemple sur un téléphone :



## Exercice : Quiz

Question 1

Qu'est-ce qu'une route ?

- ☐ Une page de l'application
- ☐ Un *widget* permettant la transition entre 2 pages
- ☐ Un *widget* permettant d'afficher des formes géométriques à l'écran

Question 2

De quelle façon la classe `Navigator` va-t-elle stocker les routes de l'application ?

- ☐ Avec un système de pile
- ☐ Dans un tableau où les routes seront triées par nom
- ☐ Dans la variable `Navigator.routing`

Question 3

Parmi ces fonctions, laquelle permet d'ajouter dans la pile une route nommée ?

- ☐ `Navigator.pushNamed()`
- ☐ `Navigator.push()`
- ☐ `Navigator.pop()`

#### Question 4

Si je veux envoyer des données à la page suivante, où placer la donnée ?

- ☐ En argument de la fonction `Navigator.pop()`
- ☐ En argument de la fonction `Navigator.push()`
- ☐ En argument du constructeur de la prochaine page

#### Question 5

Il est nécessaire d'ajouter un constructeur à une page à laquelle on veut envoyer des données.

- ☐ Vrai
- ☐ Faux

## II. Animations dans la navigation

### A. Principe des transitions en Flutter

#### **Rappel** Pourquoi animer son application ?

Même si dans ce cours nous allons seulement évoquer les animations disponibles lors des transitions de pages, il faut savoir que lorsque l'on développe une application mobile, l'animation est une très grosse partie de la *user experience*. Attention à ne pas non plus surcharger l'application.

#### **Conseil**

Nous allons voir dans la suite de ce cours à quel point les transitions animées sont un moyen efficace pour créer un flux continu lors du passage d'une page à une autre. Ce sont ces animations qui vont rendre votre projet unique.

#### **Fondamental** Le fonctionnement d'une transition en Flutter

Une transition en Flutter se déroule lors du passage d'une page à une autre. C'est la classe **PageRouteBuilder** qui va contenir toutes les opérations en fournissant un objet **Animation**. Nous verrons plus en détail la classe **PageRouteBuilder** par la suite.

#### **Définition** La classe PageRouteBuilder

La classe **PageRouteBuilder** est à utiliser comme un *widget*. Il y a plusieurs façons d'instancier le *widget* : vous pouvez hériter de la classe sur votre propre classe, ou alors comme dans l'exemple qui suit, directement la placer dans votre arbre de *widget*.

Une **PageRouteBuilder** contient 2 fonctions *callback* :

- **pageBuilder** : il va construire la prochaine page,
- **transitionBuilder** : il va construire la transition de la route.

Le callback `pageBuilder` sera seulement appelé lorsque la page est construite, tandis que `transitionBuilder` sera appelé lors de la construction et de la destruction, car l'animation sera inversée lors de la destruction (`Navigator.pop()` entre autres).

#### Remarque

Dans cette partie, nous verrons principalement `transitionBuilder`, car c'est ce callback qui va gérer l'animation.

#### Méthode Les moyens mis en place par Flutter pour personnaliser sa transition

Une transition en Flutter peut se configurer à l'aide de plusieurs *widgets* :

- **Tween** : cet objet va définir quel sens aura l'animation utilisée. Il sera composé de 2 vecteurs 2D qui définissent la *position* de départ et de fin de l'animation.
- **AnimatedWidget** : ce *widget* sera l'animation que vous allez utiliser. Flutter possède un set entier de plusieurs animations préprogrammées (**SlideTransition**, **FadeTransition**, etc.) et vous pourrez passer en paramètres le `tween` instancié auparavant pour changer la direction que prendra l'animation.
- **CurveTween** : il permettra simplement de modifier la vitesse de l'animation en fonction du temps. Flutter possède aussi un set entier de *curves* différentes.

#### Exemple SlideTransition

L'exemple suivant montre une **SlideTransition** à partir de l'exemple de la sous-partie « Navigation simple en Flutter ».

```
14 Route createPageRoute(Widget page) {
15   return PageRouteBuilder(
16     pageBuilder: (context, animation, secondaryAnimation) => page,
17     transitionsBuilder: (context, animation, secondaryAnimation, child) {
18       const begin = Offset(0.0, 1.0);
19       const end = Offset.zero;
20       const curve = Curves.ease;
21
22       var tween =
23         Tween(begin: begin, end: end).chain(CurveTween(curve: curve));
24
25       return SlideTransition(
26         position: animation.drive(tween),
27         child: child,
28       ); // SlideTransition
29     }); // PageRouteBuilder
```

Les 3 *widgets* décrits précédemment sont utilisés :

- **Tween** : ce *widget* est instancié à l'aide d'un `begin` et d'un `end`, les 2 vecteurs 2D précédemment cités : **Offset(0, 1)** vers **Offset.zero**, qui correspond à **Offset(0, 0)**. Cet enchaînement d'*offset* fera sortir la **SlideTransition** du bas de l'écran à la prochaine page.
- **CurveTween** : il utilise la **Curve.ease** ici, qui correspond à une vitesse rapide en début d'animation et se termine en finesse.

- **AnimatedWidget** : il correspond ici à la **SlideTransition**. La **SlideTransition** est une transition de glissement.

Vous pouvez aussi remarquer que la page voulue sera passée en argument de la fonction, et retournée par le callback **pageBuilder**.

Il suffit ensuite de rajouter cette fonction dans l'appel de **Navigator.push()** avec **SecondPage()** en argument :

```
onPressed: () {  
  Navigator.push(context, createPageRoute(SecondPage()));  
},
```

Et l'animation devrait s'exécuter correctement lors du changement de page :



#### **Complément** Les différentes transitions disponibles pour la navigation

Comme cité précédemment, il existe différentes transitions mises en place par Flutter :

- **SlideTransition** : animer une glissade de l'objet,
- **FadeTransition** : animer l'opacité de l'objet,
- **RotationTransition** : animer la rotation de l'objet,
- **SizeTransition** : animer la taille attribuée à l'objet par rapport à l'écran,
- **ScaleTransition** : animer la taille de l'objet et de son contenu.

**Complément** Pour aller plus loin

Vous pouvez mélanger 2 transitions dans la **PageRouteBuilder** pour, par exemple, avoir une **SlideTransition** et une **FadeTransition** en même temps. Vous pouvez aussi avoir une transition pendant l'ouverture de la page, et une autre transition lors de la fermeture en utilisant la variable **animation.status** dans le **transitionsBuilder**.

## B. Animer un widget pendant la transition

**Fondamental** Le widget Hero et son utilisation

Vous avez probablement vu des `hero` animations plus d'une fois sans vous en rendre compte. Ce type d'animation permet d'animer un élément de l'écran lors d'un changement de page. Par exemple, sur une application de shopping, lorsque la sélection d'un article va changer, sur l'écran actuel, l'image de l'article qui vole d'un écran à un autre s'appelle en Flutter une `hero` animation. Le même mouvement peut parfois être aussi appelé `shared element transition` dans d'autres langages.

**Exemple** Widget animé sur 2 pages différentes

Pour utiliser le widget **Hero**, il suffit d'instancier 2 *widgets* avec la même valeur sur le paramètre `tag`, et Flutter se chargera de l'animation entre ces 2 *widgets*. Dans l'exemple ci-dessous, nous avons une custom classe **HeroWidget** prenant en argument le `callback` de **onTap** pour détecter le *tap* de l'utilisateur et déclencher le changement de page, et en second argument la taille du carré qui sera en mouvement pendant l'animation du changement de page.

```

14 class HeroWidget extends StatelessWidget {
15   const HeroWidget({Key key, this.onTap, this.width}) : super(key: key);
16
17   final VoidCallback onTap;
18   final double width;
19
20   Widget build(BuildContext context) {
21     return Hero(
22       tag: "Hero",
23       child: Material(
24         color: Colors.transparent,
25         child: InkWell(
26           onTap: onTap,
27           child: Container(
28             width: width,
29             height: width,
30             color: Colors.blue,
31           )), // Container // InkWell
32       ), // Material
33     ); // Hero
34   }
35 }

```

Il suffit ensuite d'instancier notre **HeroWidget** dans la **FirstPage** et la **SecondPage** de l'exemple « Navigation simple en Flutter ».



```
37 class FirstPage extends StatelessWidget {
38   @override
39   Widget build(BuildContext context) {
40     return Scaffold(
41       appBar: AppBar(
42         title: const Text('First Page'),
43       ), // AppBar
44       body: Center(
45         child: HeroWidget(
46           width: 300.0,
47           onTap: () {
48             Navigator.of(context).push(MaterialPageRoute<void>(
49               builder: (BuildContext context) => SecondPage())); // MaterialPageRoute
50           },
51         ), // HeroWidget
52       ), // Center
53     ); // Scaffold
54   }
55 }
56
57 class SecondPage extends StatelessWidget {
58   @override
59   Widget build(BuildContext context) {
60     return Scaffold(
61       appBar: AppBar(
62         title: const Text('Second Page'),
63       ), // AppBar
64       body: HeroWidget(
65         width: 100.0,
66         onTap: () {
67           Navigator.of(context).pop();
68         },
69       ), // HeroWidget
70     ); // Scaffold
71   }
72 }
```

Nous avons donc remplacé les boutons par notre **HeroWidget**. En exécutant ce code, on peut voir le mouvement du carré bleu, et aussi la diminution de sa taille. Et en revenant en arrière, le mouvement inverse se produit.

Voici le rendu sur un téléphone de l'animation :



## Exercice : Quiz

### Question 1

Quels sont les 2 *callback* de la classe `PageRouteBuilder` ?

- ☐ `routeBuilder` et `transitionBuilder`
- ☐ `pageBuilder` et `animationBuilder`
- ☐ `pageBuilder` et `transitionBuilder`

### Question 2

Lequel de ces 2 *widgets* va gérer la vitesse de l'animation dans `PageRouteBuilder` ?

- ☐ `CurveTween`
- ☐ `Tween`
- ☐ `TweenSpeed`

### Question 3

De quoi le *widget* `Tween` est-il composé ?

- ☐ De 2 vecteurs 3D
- ☐ De 2 vecteurs 2D
- ☐ D'une position (x, y)

## Question 4

Il est possible de fusionner 2 `AnimatedWidget` dans le même `PageRouteBuilder`.

- ☐ Vrai
- ☐ Faux

## Question 5

Comment utiliser le `widget Hero` correctement ?

- ☐ Instancier le `widget` seulement sur la première page
- ☐ Instancier le `widget` seulement sur la deuxième page
- ☐ Instancier 2 `widget Hero` avec le même paramètre `tag`

### III. Variantes de routes de navigation

#### A. Pages de dialogue

**Définition** Afficher une page de dialogue en Flutter

Une page de dialogue est une page *pop-up* qui va permettre à l'utilisateur d'avoir plusieurs choix (oui / non, valider / annuler, etc.). Le tout est géré par la fonction `showDialog()`. Cette fonction ajoutera dans la pile de la classe `Navigator` la *pop-up* passée en *callback*. Cette fonction prend donc en argument le `widget` de dialogue et retourne le résultat.

#### Les différentes pages de dialogue

Flutter a mis en place plusieurs pages de dialogue :

- **SimpleDialog** : cette page va lister plusieurs éléments et l'utilisateur pourra en sélectionner un.
- **AlertDialog** ou **CupertinoAlertDialog** : cette page permet de récupérer l'attention de l'utilisateur. Elle va ouvrir une *pop-up* qui offrira un choix, par exemple *oui* ou *non*.
- **AboutDialog** : cette page permet d'afficher des informations à propos de l'application, par exemple sa version, les copyrights, etc.

**Remarque**

Il existe bien d'autres dialogues fournis par Flutter, mais souvent vous devrez développer votre propre classe de dialogue pour plus de personnalisation.

**Exemple** AlertDialog et AboutDialog

Comme les dialogues sont des *pop-ups* apparaissant sur la page actuelle, nous aurons besoin d'une seule page pour cet exemple.

Reprenons la **FirstPage** de l'exemple « Navigation simple en Flutter », et ajoutons un bouton par dialogues que nous voulons tester.

```

14 class FirstPage extends StatelessWidget {
15   @override
16   Widget build(BuildContext context) {
17     return Scaffold(
18       appBar: AppBar(
19         title: Text('First Page'),
20       ), // AppBar
21       body: Center(
22         child: Column(children: [
23           ElevatedButton(
24             child: Text('Open AboutDialog'),
25             onPressed: () {
26               showDialog(
27                 context: context,
28                 builder: (BuildContext context) {
29                   return new AboutDialog(
30                     applicationName: "Exemple-3.1",
31                     applicationVersion: "1.0",
32                     applicationLegalese: "Copyright",
33                   ); // AboutDialog
34                 });
35             },
36           ), // ElevatedButton
37           ElevatedButton(
38             child: Text('Open AlertDialog'),
39             onPressed: () async {
40               final result = await showDialog(
41                 context: context,
42                 builder: (BuildContext context) {
43                   return new AlertDialog(
44                     title: new Text("Popup AlertDialog"),
45                     actions: <Widget>[
46                       new ElevatedButton(
47                         onPressed: () => Navigator.pop(context, false),
48                         child: new Text("Non"),
49                       ), // ElevatedButton
50                       new ElevatedButton(
51                         onPressed: () => Navigator.pop(context, true),
52                         child: new Text("Oui"),
53                       ), // ElevatedButton
54                     ], // <Widget>[]
55                   ); // AlertDialog
56                 });
57             },
58           ), // ElevatedButton
59         ]), // Column
60       ), // Center
61     ); // Scaffold
62   }
63 }

```

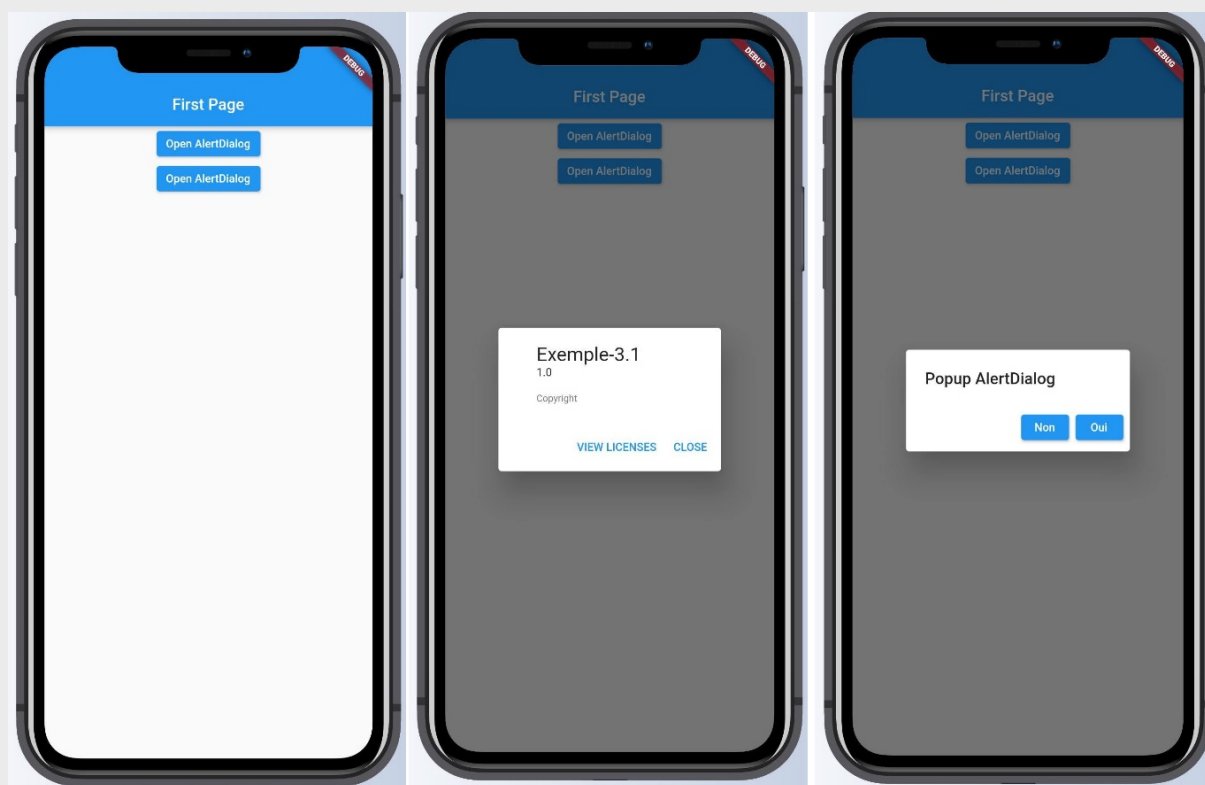
Dans l'exemple de code suivant, nous avons donc les 2 **ElevatedButtons**, qui appellent **showDialog()** lors du clic de l'utilisateur. Cette même fonction **showDialog()** va donc prendre en argument **builder** le *callback* de la page *pop-up* que nous voulons afficher.

**AboutDialog** : celle-ci prend en argument les informations de l'application qui sont assez explicites.

**AlertDialog** : cette page de dialogue est plus complexe à comprendre. Elle possède un titre qui sera affiché en haut de la page *pop-up*. Elle possède aussi un argument *action*, qui est en fait un tableau de *widgets* qui seront affichés les uns après les autres en bas de la page *pop-up*.

Les dialogues étant dans la pile de la classe **Navigator**, il suffit pour les quitter d'exécuter la fonction **Navigator.pop()**, sans oublier d'envoyer le résultat. Ce même résultat sera retourné par la fonction **showDialog()** appelée plus haut, comme cela apparaît à la ligne 40 de l'exemple.

Voici le rendu sur l'écran du téléphone :



## B. Barre de navigation

### Fondamental La classe **BottomNavigationBar**

La classe **BottomNavigationBar** est, comme son nom l'indique, une barre de navigation qui s'affiche en bas de l'écran de l'appareil, pour sélectionner plusieurs vues différentes, généralement entre 3 et 5.

### Remarque **BottomNavigatorBar** est différente de la classe **Navigator**

Cette barre de navigation consiste à faire de la navigation dans une même page. Il n'y aura alors pas de changement de page. Seul le type d'affichage des éléments change lorsque l'utilisateur rentre en contact avec la barre.

**Attention**

Pour travailler avec la classe `BottomNavigationBar`, inutile de préciser qu'il faut que la page hérite de `StatefulWidget`, car l'intérieur de la page se doit d'être dynamique.

**Exemple**

**La classe `BottomNavigationBar`**

Dans l'exemple suivant, nous aurons 3 boutons qui changeront le `body` du `widget Scaffold` de la page.

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

  @override
  State<FirstPage> createState() => _FirstPageState();
}

class _FirstPageState extends State<FirstPage> {
  int _selectedIndex = 0;
  static const List<Widget> _widgetOptions = <Widget>[
    Text(
      'Home',
      style: TextStyle(fontSize: 30, fontWeight: FontWeight.bold),
    ), // Text
    Text(
      'Computer',
      style: TextStyle(fontSize: 30, fontWeight: FontWeight.bold),
    ), // Text
    Text(
      'School',
      style: TextStyle(fontSize: 30, fontWeight: FontWeight.bold),
    ), // Text
  ]; // <Widget>[]

  void _onItemTapped(int index) {
    setState(() {
      _selectedIndex = index;
    });
  }
}
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Example-3.2'),
    ), // AppBar
    body: Center(
      child: _widgetOptions.elementAt(_selectedIndex),
    ), // Center
    bottomNavigationBar: BottomNavigationBar(
      items: const <BottomNavigationBarItem>[
        BottomNavigationBarItem(
          icon: Icon(Icons.home),
          label: 'Home',
        ), // BottomNavigationBarItem
        BottomNavigationBarItem(
          icon: Icon(Icons.computer),
          label: 'Computer',
        ), // BottomNavigationBarItem
        BottomNavigationBarItem(
          icon: Icon(Icons.school),
          label: 'School',
        ), // BottomNavigationBarItem
      ], // <BottomNavigationBarItem>[]
      currentIndex: _selectedIndex,
      onTap: _onItemTapped,
    ), // BottomNavigationBar
  ); // Scaffold
}
```

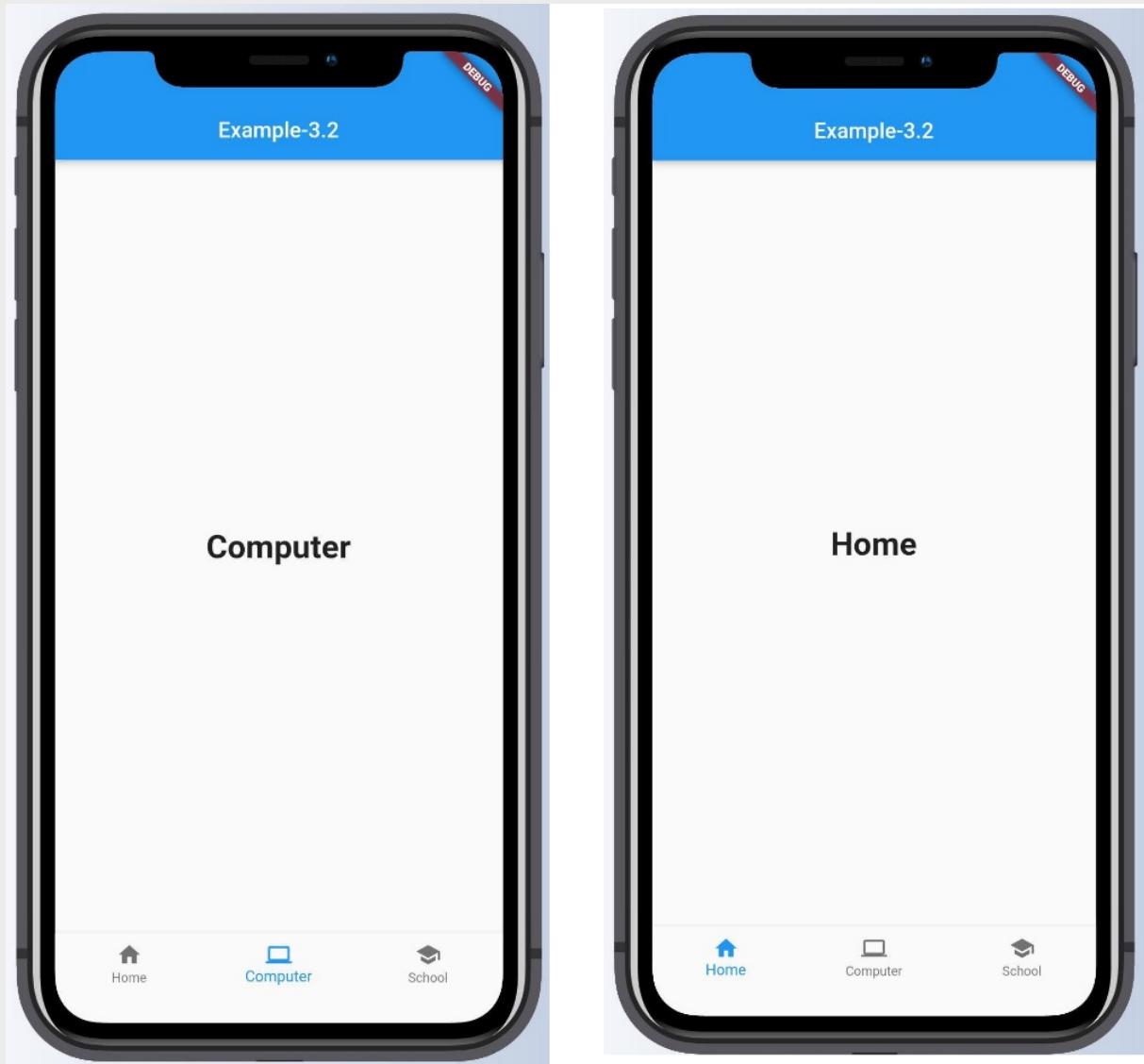
Nous avons ici 2 variables stockées dans notre page.

- **\_selectedIndex** qui indique quel bouton de la `BottomNavigationBar` est sélectionné.
- **\_widgetOptions** qui est simplement un tableau de *widgets*, et qui sera utilisé pour afficher le `body` de la page en fonction de `selectedIndex`.

Enfin, nous avons le paramètre **bottomNavigationBar**, qui permet de rajouter notre barre de navigation à la page actuelle. Cette même classe prend en arguments :

- **items** : une liste, pour chaque éléments de la liste il y aura un bouton en plus dans la **NavigationBar**. Le type de l'item ne peut être que **BottomNavigationBarItem**. Chaque item possède sa propre icône et son propre label.
- **currentIndex** : cet argument se doit d'être assigné à l'itérateur mis en place par le développeur pour la **BottomNavigationBar**. Cet argument est nécessaire au bon fonctionnement de la classe.
- **onTap** : le *callback* pour chaque *tap* de l'utilisateur sur notre **BottomNavigationBar**.

Voici le rendu à l'écran du téléphone :



## Exercice : Quiz

### Question 1

Par quelle fonction sont gérées les pages de dialogue ?



- ☐ `showPopup`
- ☐ `showDialog`
- ☐ `showAlert`

#### Question 2

Dans quelle classe mettre l'argument `bottomNavigationBar` pour ajouter une `BottomNavigationBar` à la page actuelle ?

- ☐ `Center`
- ☐ `Scaffold`
- ☐ La classe `BottomNavigationBar` est à mettre en enfant de widget

#### Question 3

Il est obligatoire d'avoir le paramètre `currentIndex` dans la classe `BottomNavigationBar`.

- ☐ Vrai
- ☐ Faux

#### Question 4

Que représente le paramètre `actions` dans le widget `AlertDialog` ?

- ☐ Liste des actions de la *pop-up*
- ☐ Activation / désactivation des actions de la *pop-up*
- ☐ Le titre de la *pop-up*

#### Question 5

Comment quitter une `AlertDialog` en cours ?

- ☐ *Return* la fonction actuelle
- ☐ Exécuter la fonction `Navigator.pop()`
- ☐ Exécuter la fonction `AlertDialog.exit()`

## IV. Essentiel

Ce cours nous a permis de maîtriser plusieurs façons de naviguer en Flutter :

- La classe `Navigator` et ses routes, les transitions et animations entre ces routes, et l'envoi / la réception de données entre chaque page.
- Les pages de dialogue liées à la classe `Navigator` et la réception de données de ces mêmes pages de dialogue.
- Le widget `BottomNavigationBar` et le widget `Hero`.

## V. Auto-évaluation

### A. Exercice :

Vous développez une application mobile : un nouveau réseau social. Vous souhaitez créer une page d'accueil qui sort des standards grâce à une animation de transition entre 2 pages, et une page de dialogue qui permettra de quitter l'application.

Dans cet exercice, nous allons créer 2 pages dont une contenant 2 boutons, un bouton ouvrant une page de dialogue demandant à l'utilisateur s'il veut quitter l'application, et l'autre bouton naviguant vers la seconde page, qui contient elle-même un bouton permettant de revenir en arrière.

#### Conseil :

Pour quitter l'application, vous pouvez utiliser `SystemNavigator.pop()` de la librairie `import package:flutter/services.dart` ou `exit(0)` de la librairie `dart:io`.

#### Question 1

Créez une page contenant simplement un bouton au centre de l'écran.

#### Question 2

Assignez la fonction d'ouverture d'une `AlertDialog` grâce à la fonction `showDialog` au bouton de l'écran.

#### Question 3

Mettez en place 2 boutons dans cette `AlertDialog`, possédant comme titre « *êtes-vous sûr de vouloir quitter l'application ?* », un bouton « *Oui* » et un bouton « *Non* » (si l'utilisateur clique sur « **Oui** », cela ferme l'application, si l'utilisateur clique sur « **Non** », cela revient sur la page principale).

#### Question 4

Mettez en place un second bouton en dessous du bouton de *pop-up*, et une seconde page pas encore joignable par l'application qui possède elle aussi un bouton au centre de l'écran.

#### Question 5

Liez ces 2 pages par une animation transition de votre choix entre `SlideTransition` et `ScaleTransition`.

### B. Test

#### Exercice : Quiz

##### Question 1

Quelle est la classe qui permet la navigation entre plusieurs pages grâce à un système de pile ?

- ☐ Navigator
- ☐ Navigation
- ☐ Navigate

##### Question 2

Parmi ces animations de transition, lesquelles sont déjà implémentées en Flutter ?

- ☐ `SlideTransition`
- ☐ `RotationTransition`
- ☐ `DeepTransition`
- ☐ `AlertTransition`
- ☐ `ScaleTransition`

## Question 3

La barre de navigation `BottomNavigationBar` utilise la classe `Navigator` et son système de pile.

- ☐ Vrai
- ☐ Faux

## Question 4

Quelle page de dialogue permet d'afficher les informations de l'application (version, copyright, etc.) ?

- ☐ `AlertDialog`
- ☐ `SimpleDialog`
- ☐ `AboutDialog`

## Question 5

Quelle est l'utilité d'ajouter un constructeur à une page existante par rapport à la fonction `Navigator.push()` ?

- ☐ Il n'y a pas d'utilité
- ☐ Le constructeur permettra d'envoyer des données à la page en question
- ☐ À rendre la page instanciable car, sans constructeur, elle ne l'est pas


**Solutions des exercices**



**Exercice p. 12 Solution n°1**


## Question 1

Qu'est-ce qu'une route ?

- ☐ Une page de l'application
- ☒ Un *widget* permettant la transition entre 2 pages
- ☐ Un *widget* permettant d'afficher des formes géométriques à l'écran
-  Une route n'est pas une page de l'application. Une route est le *widget* qui, couplé à la classe `Navigator`, permettra le passage entre plusieurs pages de l'application.


## Question 2

De quelle façon la classe `Navigator` va-t-elle stocker les routes de l'application ?

- ☒ Avec un système de pile
- ☐ Dans un tableau où les routes seront triées par nom
- ☐ Dans la variable `Navigator.routing`
-  C'est le système de pile de la classe `Navigator` qui va stocker les routes de l'application.


## Question 3

Parmi ces fonctions, laquelle permet d'ajouter dans la pile une route nommée ?

- ☒ `Navigator.pushNamed()`
- ☐ `Navigator.push()`
- ☐ `Navigator.pop()`
-  La fonction `Navigator.push()` va ajouter une route passée en argument, et la fonction `Navigator.pop()` va revenir à la route précédente. C'est la fonction `Navigator.pushNamed()` qui va ajouter dans la file une route nommée.


## Question 4

Si je veux envoyer des données à la page suivante, où placer la donnée ?

- ☐ En argument de la fonction `Navigator.pop()`
- ☐ En argument de la fonction `Navigator.push()`
- ☒ En argument du constructeur de la prochaine page
-  La fonction `Navigator.push()` va prendre la prochaine page en argument, et c'est le constructeur de la page qui prendra en argument les données à passer.

## Question 5

Il est nécessaire d'ajouter un constructeur à une page à laquelle on veut envoyer des données.

- ☒ Vrai
- ☐ Faux
-  C'est nécessaire car le constructeur par défaut n'a pas d'argument, donc les données ne peuvent pas être envoyées.

### Exercice p. 18 Solution n°2

#### Question 1

Quels sont les 2 *callback* de la classe `PageRouteBuilder` ?

- ☐ `routeBuilder` et `transitionBuilder`
- ☐ `pageBuilder` et `animationBuilder`
- ☒ `pageBuilder` et `transitionBuilder`



Les *callback* de la classe `PageRouteBuilder` sont `pageBuilder`, qui va gérer l'affichage de la page, et `transitionBuilder`, qui va gérer l'animation de transition.

#### Question 2

Lequel de ces 2 *widgets* va gérer la vitesse de l'animation dans `PageRouteBuilder` ?

- ☒ `CurveTween`
- ☐ `Tween`
- ☐ `TweenSpeed`



C'est le *widget* `CurveTween` qui va gérer la vitesse de l'animation dans le *callback* `transitionBuilder`.

#### Question 3

De quoi le *widget* `Tween` est-il composé ?

- ☐ De 2 vecteurs 3D
- ☒ De 2 vecteurs 2D
- ☐ D'une position (x, y)



Le *widget* `Tween` prend en argument `begin`, un vecteur 2D et `end`, un autre vecteur 2D. Il est donc composé de 2 vecteurs 2D.

#### Question 4

Il est possible de fusionner 2 `AnimatedWidget` dans le même `PageRouteBuilder`.

- ☒ Vrai
- ☐ Faux



Oui, il est possible de fusionner 2 `AnimatedWidget`, par exemple `ScaleTransition` et `RotationTransition`, pour coupler plusieurs animations.

#### Question 5

Comment utiliser le *widget* `Hero` correctement ?

- ☐ Instancier le *widget* seulement sur la première page
- ☐ Instancier le *widget* seulement sur la deuxième page
- ☒ Instancier 2 *widget* `Hero` avec le même paramètre `tag`




Il est nécessaire d'instancier le *widget* `Hero` 2 fois avec le même paramètre `tag` pour que Flutter comprenne que ces 2 *widgets* sont liés.

### Exercice p. 24 Solution n°3

## Question 1

Par quelle fonction sont gérées les pages de dialogue ?


- ☐ `showPopup`
- ☒ `showDialog`
- ☐ `showAlert`

 C'est la fonction `showDialog` qui va gérer les pages de dialogue et retourner leur résultat. Les fonctions `showPopup` et `showAlert` n'existent pas en Flutter.

## Question 2

Dans quelle classe mettre l'argument `bottomNavigationBar` pour ajouter une `BottomNavigationBar` à la page actuelle ?


- ☐ `Center`
- ☒ `Scaffold`
- ☐ La classe `BottomNavigationBar` est à mettre en enfant de widget

 C'est la classe `Scaffold` qui va contenir la `BottomNavigationBar` dans son paramètre `bottomNavigationBar`.

## Question 3

Il est obligatoire d'avoir le paramètre `currentIndex` dans la classe `BottomNavigationBar`.


- ☒ Vrai
- ☐ Faux

 Vrai, il est obligatoire pour le bon fonctionnement de la classe `BottomNavigationBar`.

## Question 4

Que représente le paramètre `actions` dans le widget `AlertDialog` ?


- ☒ Liste des actions de la *pop-up*
- ☐ Activation / désactivation des actions de la *pop-up*
- ☐ Le titre de la *pop-up*

 Le paramètre `actions` représente la liste des actions de la *pop-up*. Par exemple, dans une `AlertDialog`, la liste des actions pourra être une liste de widgets `ElevatedButton`, qui auront chacun une fonctionnalité différente.

## Question 5

Comment quitter une `AlertDialog` en cours ?

- ☐ *Return* la fonction actuelle
- ☒ Exécuter la fonction `Navigator.pop()`
- ☐ Exécuter la fonction `AlertDialog.exit()`

 Une `AlertDialog` étant poussée dans la pile de la classe `Navigator` grâce à la fonction `showDialog`, pour la quitter il suffit d'appeler `Navigator.pop()`. Vous pouvez aussi passer un argument à `Navigator.pop()` pour retourner de la donnée.

**Exercice p. Solution n°4**

```

1  import 'package:flutter/material.dart';
2
   Run | Debug | Profile
3  void main() {
4      runApp(MaterialApp(
5          title: 'Exercice rédactionnel',
6          home: FirstPage(),
7      )); // MaterialApp
8  }
9
10 class FirstPage extends StatelessWidget {
11     @override
12     Widget build(BuildContext context) {
13         return Scaffold(
14             appBar: AppBar(
15                 title: Text('First Page'),
16             ), // AppBar
17             body: Center(
18                 child: ElevatedButton(
19                     child: Text('Button 1'),
20                     onPressed: () {
21                         // not yet implemented
22                     },
23                 ), // ElevatedButton
24             ), // Center
25         ); // Scaffold
26     }
27 }

```

Nous avons donc ici la fonction `main()` contenant la `FirstPage()`, qui elle-même contient une `Scaffold()`, et dans son `body`, notre bouton centré à l'écran.

**Exercice p. Solution n°5**

```

21         onPressed: () {
22             showDialog(
23                 context: context,
24                 builder: (BuildContext context) {
25                     return new AlertDialog(
26                         title: new Text("Dialog"),
27                         actions: <Widget>[],
28                     ); // AlertDialog
29                 });
30         },

```



Nous avons donc ajouté la fonction `showDialog()` retournant le *widget* `AlertDialog`, avec une liste d'action vide.

### Exercice p. Solution n°6

```

21     onPressed: () async {
22       final result = await showDialog(
23         context: context,
24         builder: (BuildContext context) {
25           return new AlertDialog(
26             title: new Text(
27               "Etes-vous sur de vouloir quitter l'application ?"), // Text
28             actions: <Widget>[
29               new ElevatedButton(
30                 onPressed: () => Navigator.of(context).pop(false),
31                 child: new Text("Non"),
32               ), // ElevatedButton
33               new ElevatedButton(
34                 onPressed: () => Navigator.of(context).pop(true),
35                 child: new Text("Oui"),
36               ), // ElevatedButton
37             ], // <Widget>[]
38           ); // AlertDialog
39         });
40       if (result == true) {
41         SystemNavigator.pop();
42       }
43     },

```

Celle-ci contient 2 boutons, qui retournent *vrai* » ou « *faux* » en fonction du résultat. Ce même résultat sera stocké dans la variable `result` qui sera par la suite testée, et si celle-ci est *vrai*, quittera l'application.

Attention à ne pas oublier d'inclure la librairie pour pouvoir appeler `SystemNavigator.pop()`.

```

2   import 'package:flutter/services.dart';

```

### Exercice p. Solution n°7

```

18     body: Center(
19       child: Column(mainAxisAlignment: MainAxisAlignment.center, children: [
20         ElevatedButton(
21           child: Text('Button 1'),
22           onPressed: () async {
23             final result = await showDialog(
24               context: context,
25               builder: (BuildContext context) {
26                 return new AlertDialog(
27                   title: new Text(
28                     "Etes-vous sur de vouloir quitter l'application ?"), // Text
29                   actions: <Widget>[
30                     new ElevatedButton(
31                       onPressed: () => Navigator.of(context).pop(false),
32                       child: new Text("Non"),
33                     ), // ElevatedButton
34                     new ElevatedButton(
35                       onPressed: () => Navigator.of(context).pop(true),
36                       child: new Text("Oui"),
37                     ), // ElevatedButton
38                   ], // <Widget>[]
39                 ); // AlertDialog
40               });
41             if (result == true) {
42               SystemNavigator.pop();
43             }
44           }), // ElevatedButton
45         ElevatedButton(
46           onPressed: () {
47             //
48           },
49           child: Text("Button 2")) // ElevatedButton
50       ]), // Column
51     ), // Center

```

Nous avons donc ajouté un *widget* `Column` pour pouvoir mettre un bouton en plus en dessous du bouton de dialogue. Le bouton a par la suite été rajouté.

```

56 class SecondPage extends StatelessWidget {
57   @override
58   Widget build(BuildContext context) {
59     return Scaffold(
60       appBar: AppBar(
61         title: Text("Second Page"),
62       ), // AppBar
63       body: Center(
64         child: ElevatedButton(
65           onPressed: () {
66             //
67           },
68           child: Text('Button 3'),
69         ), // ElevatedButton
70       ), // Center
71     ); // Scaffold
72   }
73 }

```

Et nous avons ici la deuxième page de notre application, avec un bouton centré qui pour l'instant n'a pas d'utilité.

### Exercice p. Solution n°8

```

56 Route createPageRoute(Widget page) {
57   return PageRouteBuilder(
58     pageBuilder: (context, animation, secondaryAnimation) => page,
59     transitionsBuilder: (context, animation, secondaryAnimation, child) {
60       const begin = Offset(0.0, 1.0);
61       const end = Offset.zero;
62       const curve = Curves.ease;
63
64       var tween =
65         Tween(begin: begin, end: end).chain(CurveTween(curve: curve));
66
67       return SlideTransition(
68         position: animation.drive(tween),
69         child: child,
70       ); // SlideTransition
71     }); // PageRouteBuilder
72 }

```

Pour la FirstPage :

```

46   onPressed: () {
47     Navigator.push(context, createPageRoute(SecondPage()));
48   },

```

Pour la SecondPage :

```

83   onPressed: () {
84     Navigator.pop(context);
85   },

```

Pour cette dernière question, nous avons développé la fonction `createPageRoute` qui va appliquer la `SlideTransition`. Ici, nous allons de bas en haut à cause de notre `Tween`. La `CurveTween` choisie est `Curves.ease`.

Enfin, il reste à implémenter les 2 *callback* des 2 boutons restants, Button 2 et 3. Le 2 appellera donc `Navigator.push()` qui ajoutera la nouvelle route dans la pile de `Navigator`, et le Button 3 permettra de retirer cette route de la pile, ce qui fera revenir à `FirstScreen`.

### Exercice p. 26 Solution n°9

Question 1

Quelle est la classe qui permet la navigation entre plusieurs pages grâce à un système de pile ?

- ☒ Navigator
- ☐ Navigation
- ☐ Navigate

**Q** C'est la classe `Navigator` qui permet la navigation entre plusieurs pages grâce à son système de pile.

#### Question 2

Parmi ces animations de transition, lesquelles sont déjà implémentées en Flutter ?

- ☒ `SlideTransition`
- ☒ `RotationTransition`
- ☐ `DeepTransition`
- ☐ `AlertTransition`
- ☒ `ScaleTransition`

**Q** Les transitions déjà implémentées en Flutter sont `ScaleTransition`, `SlideTransition` et `RotationTransition`.

#### Question 3

La barre de navigation `BottomNavigationBar` utilise la classe `Navigator` et son système de pile.

- ☐ Vrai
- ☒ Faux

**Q** La `BottomNavigationBar` n'utilise pas le système de pile car ce sont différents *widgets* qui sont affichés et changés dynamiquement durant la même route.

#### Question 4

Quelle page de dialogue permet d'afficher les informations de l'application (version, copyright, etc.) ?

- ☐ `AlertDialog`
- ☐ `SimpleDialog`
- ☒ `AboutDialog`

**Q** C'est la page de dialogue `AboutDialog` qui permet d'afficher les informations de l'application.

#### Question 5

Quelle est l'utilité d'ajouter un constructeur à une page existante par rapport à la fonction `Navigator.push()` ?

- ☐ Il n'y a pas d'utilité
- ☒ Le constructeur permettra d'envoyer des données à la page en question
- ☐ À rendre la page instanciable car, sans constructeur, elle ne l'est pas

**Q** C'est le constructeur de la page qui va recevoir les données envoyées par la fonction `Navigator.push()` pour les stocker de son côté. En Flutter, une page est instanciable de base car elle possède un constructeur par défaut.