Flutter

Fiche 2 : Listes & Stateful Widgets

Table des matières

1 Objectifs de la fiche

ID	Objectifs
F02	Liste de widgets
F03	Stateful widget

2 Concepts

Pour commencer le tutoriel, créez un nouveau projet flutter nommé *tuto2* dans votre repository de cours. Dans ce tutoriel, nous allons créer ensemble une application permettant d'afficher une liste de contacts. Créez d'abord un fichier **contact.dart** dans le dossier **lib** et écrivez-y le code suivant. Cette classe définit un objet Contact, avec son nom, son numéro de téléphone, et un booléen indiquant s'il fait partie des contacts favoris.

```
class Contact {
  final String name;
  final String phone;
  final bool isFavorite;

  Contact({required this.name, required this.phone, this.isFavorite =
false});
}
```

[commit avec message : T02.1 Contacts]

2.1 Stateful widget basique

Avant de continuer le tutoriel, observez le code généré dans le fichier **main.dart**, en particulier les classes **MyHomePage** et **_MyHomePageState**.

La classe ***MyHomePage*** étend la classe **StatefulWidget** au lieu de **StatelessWidget** contrairement à ce que nous avons vu jusqu'ici. Elle ne propose pas de méthode **build**. À la place, on retrouve une méthode **createState** qui renvoie une instance de la classe **_MyHomePageState**. La classe ***_MyHomePageState*** possède elle bien une méthode **build** similaire à celles que nous avons pu voir dans les **StatelessWidget**. Ce widget affiche un **Scaffold** présentant une barre de titre, du texte au centre de l'écran, et un **floatingActionButton** – un style de bouton définit dans les principes Material Design de Google présentant l'action principale d'une page.

La **HomePage** possède deux variables, une définie dans la classe ***MyHomePage*** et l'autre dans la classe **_MyHomePageState**. Au sein de la classe héritant de **StatefulWidget**, nous pouvons retrouver des variables immuables (toujours marquées *final*) passées en paramètre, tel que nous avons pu le voir dans les **StatelessWidget**. Nous pouvons les utiliser dans la méthode **build** de la classe héritant de *State* avec l'objet **widget** tel que montré avec **widget.title** à la ligne 76. Au sein de cette classe State, nous pouvons également définir des variables destinées à changer durant l'exécution du programme.

Dans le widget **HomePage**, la variable **_counter** est initialisée à 0, et elle pourra être incrémentée durant l'exécution du programme. Lorsque l'utilisateur appuie sur le floating action button, le programme fait appel à la méthode **_incrementCounte***r* grâce à l'argument **onPressed**. Dans cette méthode, on retrouve un appel à une méthode **setState**. L'objectif de cette méthode est de notifier à l'application que l'état du widget est modifié. Elle prend un argument une méthode au sein de laquelle nous pouvons modifier l'état du widget. Dans ce cas-ci, le compteur est incrémenté. Et grâce à l'utilisation de **setState**, l'affichage de l'application est mis à jour pour refléter le nouvel état du widget.

Lancez l'application, et observez son fonctionnement. Modifiez ensuite la fonction **_incrementCounter** pour ne plus utiliser la méthode **setState** de cette façon :

```
void _incrementCounter() {
  _counter++;
}
```

Relancez l'application, fonctionne-elle encore comme attendu ? Pour mieux comprendre le fonctionnement des **StatefulWidget**, regardez cette vidéo : [How Stateful Widgets Are Used Best]

Annulez les changements et revenez à l'état initial pour le fichier **main.dart**.

2.2 Un élément d'une liste

Nous allons maintenant définir le widget permettant d'afficher un contact. Créez un fichier **contact_row.dart** et écrivez-y le code suivant.

```
class ContactRow extends StatelessWidget {
  final Contact contact;

  const ContactRow({
    Key? key,
    required this.contact,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: Text(contact.name),
      subtitle: Text(contact.phone),
      trailing:
          Icon(contact.isFavorite ? Icons.star : Icons.star_border),
    );
  }
}
```

N'oubliez pas d'importer la librairie "package:flutter/materials.dart". Vous pouvez la retrouver facilement en faisant un clic droit sur l'erreur affichée sur la classe **StatelessWidget** et en choisissant le bon package dans le menu « Show context actions ». Vous devrez également importer le fichier **contact.dart** en faisant la même procédure à partir de la classe **Contact**.

Ce widget affiche un contact en utilisant un widget **ListTile** qui permet d'afficher une ligne présentant un titre, un sous-titre optionnel, des icônes optionnelles et plus encore. Dans ce cas-ci, nous l'utilisons pour afficher les informations de notre contact en paramètre du widget. Avec le paramètre **trailing**, nous affichons à la fin de la ligne une icône représentant une étoile pleine si le contact fait partie des favoris et vide sinon.

Faites appel à ce widget dans le fichier **main.dart** avec un contact de test de la façon suivante.

```
children: <Widget>[
  const Text(
    'You have pushed the button this many times:',
  ),
  Text(
    '$_counter',
    style: Theme.of(context).textTheme.headlineMedium,
  ),
  ContactRow(contact: Contact(name: "Test", phone: "0123456789")),
],
```

[commit avec message : T02.2 Un élément d'une liste]

2.3 Liste de widgets

Dans la fiche 1, nous avons vu plusieurs manières d'afficher un ensemble de widgets. Nous avons vu les **Row** et les **Column** qui permettent d'afficher des widgets répartis horizontalement et verticalement respectivement. Nous allons voir ici de nouvelles manières de créer des listes de widgets à afficher. Avant cela, nous allons créer des données de test. Rajoutez le code suivant dans le fichier **contact.dart**.

```
List<Contact> _createContacts() {
  final contacts1 = {
    "Pierre": "123",
    "Paul": "456",
    "Jean": "789",
  }.entries.map((c) => Contact(name: c.key, phone: c.value));

  final contacts2 = [
    for (var i = 0; i < 1000; i++)
      Contact(name: "Contact ${i + 1}", phone: "$i", isFavorite: true)
  ];

  final contacts3 = List.generate(10, (i) => Contact(name: "C$i", phone:
"$i"));

  return [...contacts1, ...contacts2, ...contacts3];
}

final defaultContacts = _createContacts();
```

Nous pouvons voir dans ce code trois manières différentes de créer des listes de données. Nous renvoyons à la fin une liste reprenant tous les éléments des autres en utilisant le **spread operator**.

Créez ensuite un stateless widget **HomeScreen** dans un fichier **home_screen.dart**. Rajoutez le code suivant dans la fonction **build**.

```
@override
Widget build(BuildContext context) {
  final contactRows =
      defaultContacts.map((contact) => ContactRow(contact:
contact)).toList();

  return Scaffold(
    appBar: AppBar(
      backgroundColor: Theme.of(context).colorScheme.inversePrimary,
      title: const Text("Contact list"),
    ),
    body: ListView(children: contactRows),
  );
}
```

Ce widget affiche un écran avec un **Scaffold**, contenant une barre de titre et une **ListView**. Les enfants de la **ListView** proviennent d'une liste de widgets que nous avons créée à partir de la liste des contacts par défaut. La **ListView** fonctionne similairement à une **Column** ou une **Row**, dans le sens qu'il répartit ses enfants horizontalement ou verticalement (par défaut). Cependant, la **ListView** est conçu spécifiquement pour gérer des listes avec une quantité importante d'éléments. Si l'espace disponible n'est pas suffisant pour afficher tous les widgets enfants, la **ListView** proposera automatiquement une barre de défilement pour parcourir tous les éléments malgré tout.

Faites appel à ce widget dans le fichier **main.dart** à la place du widget **MyHomePage** que nous avons analysé précédemment. Vous pouvez également supprimer ce widget **MyHomePage** ainsi que sa classe state qui ne sont plus utilisés.

[commit avec message : T02.3 Liste de contacts]

Une autre manière d'afficher des widgets avec une **ListView** est d'utiliser un **builder**. Dans ce cas-là, c'est la **ListView** qui se chargera de créer les widgets nécessaires. Il ne faut plus que lui décrire comment le faire. Un autre avantage de cette méthode est que les widgets sont créés dynamiquement, au moment où ils doivent être affichés. Quand le nombre de widgets dans une **ListView** devient important, cela peut augmenter les performances de l'application.

Modifiez la méthode **build** du **HomeScreen** pour utiliser un **ListView.builder**. La fonction **log** provient de la librairie **dart:developer** que vous devrez importer.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      backgroundColor: Theme.of(context).colorScheme.inversePrimary,
      title: const Text("Contact list"),
    ),
    body: ListView.builder(
      itemCount: defaultContacts.length,
      itemBuilder: (context, index) {
        log("index : $index");
        return ContactRow(contact: defaultContacts[index]);
      },
    ),
  );
}
```

Testez l'application et vérifiez que vous voyez la liste des contacts attendue. Grâce à la fonction **log**, vous pouvez observer dans la console d'Android Studio le moment où les widgets **ContactRow** sont créés. Testez l'application pour observer son comportement.

Rajoutons maintenant quelques widgets pour améliorer l'affichage de la liste. Vous pouvez également retirer le **log** maintenant que vous avez compris le fonctionnement du **ListView.Builder**.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      backgroundColor: Theme.of(context).colorScheme.inversePrimary,
      title: const Text("Contact list"),
    ),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Center(
        child: SizedBox(
          width: 512,
          child: ListView.builder(
            itemCount: defaultContacts.length,
            itemBuilder: (context, index) =>
                ContactRow(contact: defaultContacts[index]),
          ),
```

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

[commit avec message : T02.4 Liste générée]

Dans certains cas, on souhaite afficher une séparation entre les différents éléments de la liste. Dans ce cas, on peut utiliser un widget **ListView.seperated**. Ce widget s'utilise comme un **ListView.builder**, mais avec un argument supplémentaire pour définir la séparation. Cela donne par exemple :

```
ListView.separated(
  itemCount: defaultContacts.length,
  separatorBuilder: (context, index) => const Divider(),
  itemBuilder: (context, index) =>
      ContactRow(contact: defaultContacts[index]),
),
```

2.4 Liste de widgets manipulables

Nous allons maintenant terminer l'objectif de ce tutoriel en permettant de filtrer la liste des contacts pour n'afficher que les contacts favoris. Pour pouvoir rendre l'interface du **HomeScreen** dynamique, nous devons transformer celui-ci en **StatefulWidget**. Nous devons également rajouter dans la classe d'état du **HomeScreen** une variable indiquant si l'écran doit afficher tous les contacts ou uniquement ceux en favoris.

Faites un clic droit sur le nom de la classe **HomeScreen** et sélectionnez l'option « **Convert to StatefulWidget** » dans le menu « **Show Context Actions** ». Vous pouvez également utiliser l'auto-complétion pour créer un **stateful widget** vide – comme pour les **stateless widget** comme vu dans la fiche précédente – en tapant **stful** dans Android Studio.

Rajoutez ensuite le code suivant au début de la classe **HomeScreenState**.

```
var showFavorites = false;

void _toggleFavorites() => setState(() => showFavorites = !showFavorites);
```

La méthode **_toggleFavorites** permet de modifier l'état de la variable **showFavorites**. Grâce à l'utilisation de la méthode **setState**, l'affichage du **HomeScreen** sera mis à jour pour refléter le nouvel état de cette variable.

Nous pouvons maintenant modifier la méthode **build** pour réagir à l'état du widget.

```
@override
Widget build(BuildContext context) {
  final displayedContacts = [
    for (var contact in defaultContacts)
      if (!showFavorites || contact.isFavorite) contact
  ];

  return Scaffold(
```

```
    appBar: AppBar(
      title: const Text("Contact list"),
      actions: [
        IconButton(
          icon: Icon(showFavorites ? Icons.star : Icons.star_border),
          onPressed: _toggleFavorites,
        ),
      ],
    ),
    body: Container(
      padding: const EdgeInsets.all(16.0),
      child: Center(
        child: SizedBox(
          width: 512.0,
          child: ListView.builder(
            itemCount: displayedContacts.length,
            itemBuilder: (context, index) =>
                ContactRow(contact: displayedContacts[index]),
          ),
        ),
      ),
    ),
  );
}
```

Nous avons rajouté dans l'**AppBar** une liste d'**actions**, des boutons sous forme d'icônes qui s'affichent dans la barre de titre de l'application et permettent de réaliser des actions sur l'écran actuel. Le bouton que nous utilisons ici affiche l'état de la variable **showFavorites** avec une étoile vide ou pleine, et permet de modifier cet état en faisant appel à la méthode **_toggleFavorites** quand il est appuyé.

Si vous ne voyez pas le bouton à cause du bandeau « debug » rajouté par flutter sur les applications en développement, vous pouvez rajouter l'argument suivant au widget **MaterialApp** dans le fichier **main.dart** pour le cacher :

```
debugShowCheckedModeBanner: false,
```

Au lieu d'afficher simplement tous les contacts de la liste **defaultContacts**, nous définissons une variable **displayedContacts** qui contiendra soit tous les contacts de la liste si la variable **showFavorites** est *false*, soit uniquement les contacts favoris si elle est *true*. Nous utilisons pour ça la syntaxe de « collection for » pour faciliter la création de cette liste. Nous pouvons ensuite utiliser cette variable comme base du **ListView.builder** pour afficher les contacts filtrés ou non au sein de l'application.

Lancez l'application et vérifiez qu'elle correspond à ce que vous attendez.

[commit avec message : T02.5 Filtrage de liste]

3 Exercice

3.1 Introduction

Vous souhaitez créer une vitrine pour votre entreprise d'agence immobilière. Vous proposez des propriétés à la vente ou à la location, et vous cherchez à améliorer la visibilité de votre commerce. Vous vous lancez donc dans la réalisation d'une application en flutter.

Créez un nouveau projet nommé *ex2* dans votre repository de cours.

3.2 Représentation des données

Commencez par définir les données de votre application. Une propriété est caractérisée par :

- Si la propriété est à vendre ou à louer,
- S'il s'agit d'un appartement ou d'une maison,
- Son nombre de mètres carrés,
- son nombre de chambres,
- son prix.

Créez également un ensemble de propriétés pour pouvoir tester votre application. Vous devez inclure au minimum deux propriétés à vendre et deux propriétés à louer.

⚡ **[commit avec message : F02.1 Représentation des données]** ⚡

3.3 Affichage d'un élément

Créez maintenant un widget **PropertyWidget** dans un fichier dédié. Ce widget doit afficher toutes les informations d'une propriété en argument. Nous vous conseillons d'utiliser le widget **ListTile**.

Créez également un widget **HomeScreen** définissant un écran avec un titre. Pour le moment, vous devez juste afficher comme corps de l'écran votre **PropertyWidget** avec une des propriétés que vous avez défini précédemment.

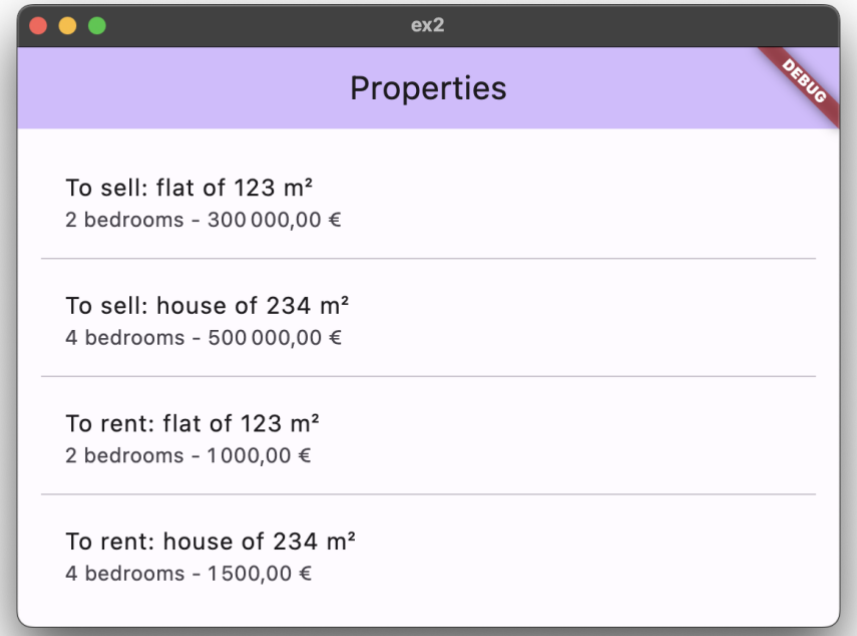
Enlevez le code non nécessaire dans le fichier **main.dart** et affichez votre écran. Voici un exemple de l'interface que votre application pourra avoir à cette étape :

⚡ **[commit avec message : F02.2 Affichage d'une propriété]** ⚡

3.4 Liste de widgets

Modifiez maintenant votre widget **HomeScreen** pour afficher toutes les propriétés existantes. Cette liste doit contenir toutes les propriétés venant des deux listes que vous avez défini précédemment. L'affichage de ce widget doit rester efficace si le nombre de propriétés augmente.

Voici un exemple de l'interface que votre application pourra avoir à cette étape :



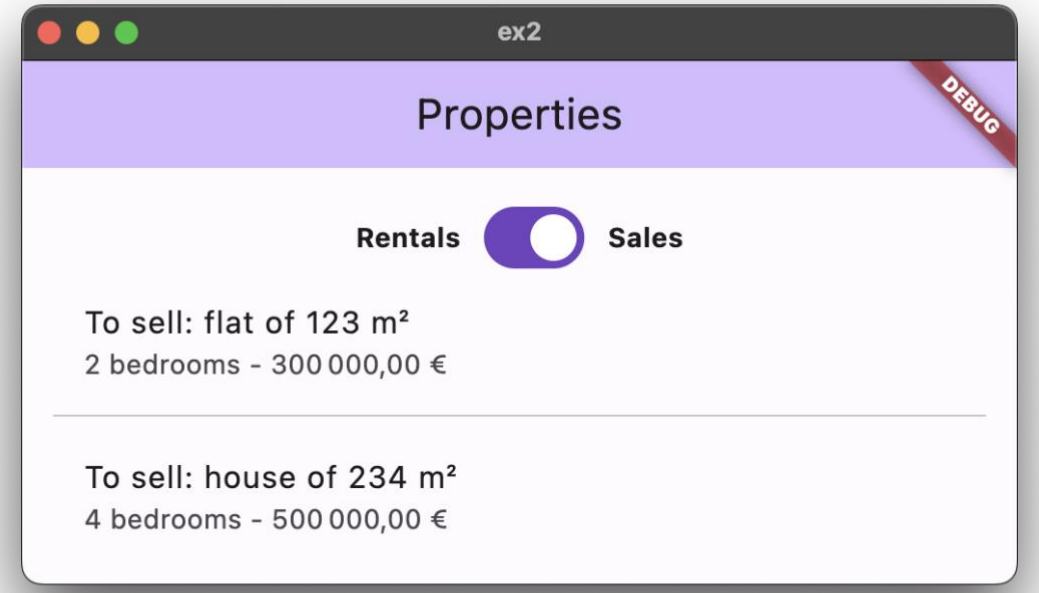
⚡ **[commit avec message : F02.3 Liste de propriétés]** ⚡

3.5 Stateful widget

Modifiez l'écran principal de votre application pour rajouter au-dessus de la liste des propriétés un interrupteur entre les ventes et les locations. Lorsque l'application est dans l'état « ventes », les seules propriétés affichées sont celles qui sont « à vendre ». Lorsque l'application est en « locations », ce sont celles « à louer » qui sont affichées.

Pour afficher cet interrupteur, vous pouvez utiliser le widget de la librairie standard de flutter ***Switch***. Ce widget représente un interrupteur qui affiche la valeur d'une variable d'état. Il propose également d'enregistrer une fonction qui sera appelée lorsque l'utilisateur tape sur l'interrupteur, prenant en argument la nouvelle valeur de l'interrupteur. Vous pouvez retrouver la documentation de ce widget à ce lien : https://api.flutter.dev/flutter/material/Switch-class.html.

Voici un exemple de l'interface que votre application pourra avoir à cette étape :



⚡ **[commit avec message : F03.1 Sélection de mode]** ⚡