Context et Inherited widget

Rappel sur le `Context`

Comme nous l'avons vu le `context`, de type `BuildContext` est un `Element` créé avec `createElement()` pour chaque `widget`.

Un `Element` garde des références au `Widget`, au `RenderObject`, et éventuellement au `State` correspondants.

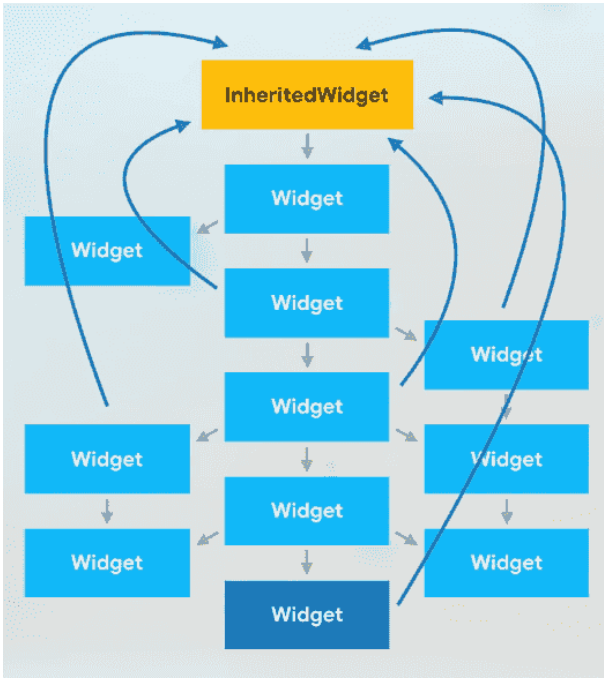
Il existe une différence entre ces références : si la référence au `Widget` et au `RenderObject` peuvent être modifiées lors de mises à jour, il n'en va pas de même pour le `State` qui reste toujours lié à son `Element / Context`.

Enfin, un `Element` a également une référence à son `Element` parent et à ses éventuels `Elements` enfants.

Les `InheritedWidgets`

Les `InheritedWidgets` permettent de partager des informations le long de l'arbre des `widgets`, de manière unidirectionnelle, du haut vers le bas.

Un `InheritedWidget` est un `Widget` spécial que nous pouvons placer comme parent d'une branche de notre arbre de `widgets`. Tous les `widgets` de cette branche auront la possibilité d'interagir avec les données exposées par l'`InheritedWidget` :



Voilà à quoi ressemble un `InheritedWidget` minimaliste :

```
void main() => runApp(const MaterialApp(home: PositionedTiles()));

class ColorInfo extends InheritedWidget {
  final List<Color> colors;

  const ColorInfo({super.key, required this.colors, required super.chil
d}) : super(child: child);

  @override
  bool updateShouldNotify(ColorInfo old) => colors != old.colors;
}
```

Première remarque, un `InheritedWidget` **ne fait que contenir des données**.

Comme tous les `widgets` il est immuable et à un `Element` qui est lui modifiable.

Tous les `widgets` en dessous du `InheritedWidget` peuvent accéder à la liste de couleurs en faisant :

```
final color = context.dependOnInheritedWidgetOfExactType<ColorInfo>();
```

Mais il y a un moyen pour pouvoir y accéder plus simplement.

Pour cela, il suffit de définir deux méthodes statiques sur notre `InheritedWidget` :

```
class ColorInfo extends InheritedWidget {
  final List<Color> colors;

  const ColorInfo({super.key, required this.colors, required super.chil
d}) : super(child: child);

  static ColorInfo? maybeOf(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<ColorInfo>();
  }

  static ColorInfo of(BuildContext context) {
    final ColorInfo? result = maybeOf(context);
    assert(result != null, 'No ColorInfo found in context');
    return result!;
  }

  @override
```

```
  bool updateShouldNotify(ColorInfo old) => colors != old.colors;
}
```

Grâce à ces méthodes, nous pouvons désormais directement accéder à `colors` sur les `widgets` plus bas dans l'arbre en faisant :

```
class SomeWidget... {
    ...

    @override
    Widget build(BuildContext context){
      final colors = ColorInfo.of(context);
…
```

Nous verrons dans les leçons suivantes que ce pattern est utilisé par `Flutter` pour plusieurs `InheritedWidget` internes : `Theme`, `MediaQuery` etc.

La méthode `updateShouldNotify()` permet à `Flutter` de notifier les `widgets` qui utilisent la propriété `colors` de notre `InheritedWidget` qu'ils doivent se mettre à jour.

Lorsque l'`InheritedWidget` est `rebuild`, par exemple parce que sa propriété `colors` change, la méthode `updateShouldNotify()` permet de décider si les `widgets`, utilisant `colors`, et situés en dessous dans l'arbre doivent également être `rebuild`.

La méthode reçoit l'ancien `InheritedWidget`, c'est-à-dire avant qu'il soit `rebuild`, et nous pouvons donc comparer si la propriété `colors` a été modifiée lors du `rebuild`.

Si la propriété est modifiée nous retournons `true` et les `widgets` seront `rebuild`, dans le cas contraire nous retournons `false`.

Utilisation dans notre application

Pour que vous puissiez parfaitement comprendre le fonctionnement du `InheritedWidget`, nous allons l'utiliser dans notre application.

Nous allons en fait ne plus utiliser directement le fichier `data.dart` mais utiliser un `InheritedWidget`.

Pour cela, nous allons commencer par créer un dossier `widgets` dans le dossier `lib`.

Nous le plaçons au plus au niveau pour signifier que ce sont les `widgets` partagés par toute l'application.

Dans ce dossier, nous allons créer un fichier `data.dart` qui va contenir notre `InheritedWidget`.

Nous allons créer un `InheritedWidget` :

```
import 'package:flutter/material.dart';
import '../data/data.dart' as data;
import '../models/activity.model.dart';

class Data extends InheritedWidget {
  final List<Activity> activities = data.activities;

  Data({super.key, required super.child});

  static Data? maybeOf(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<Data>();
  }

  static Data of(BuildContext context) {
    final Data? result = maybeOf(context);
    assert(result != null, 'No Data found in context');
    return result!;
  }

  @override
  bool updateShouldNotify(Data oldWidget) => true;
}
```

Nous notons que nous utilisons le constructeur afin de passer le `child` à notre classe `super InheritedWidget` comme vu précédemment.

Nous définissons une méthode statique `of` qui retourne une instance de notre classe et que nous utilisons pour pouvoir utiliser le raccourci syntaxique `Data.of()` sur les `widgets` souhaitant accéder aux `activities` de notre `InheritedWidget`.

Nous avons enfin à définir la méthode `updateShouldNotify()` qui permet de notifier les `widgets` utilisant `activities`. Ici nous retournons toujours `true`, ce qui signifie que dès lors que `InheritedWidget` change, alors tous les `widgets` utilisant `activities` seront `rebuild`.

Nous allons ensuite positionner l'`InheritedWidget` dans `main.dart` :

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

```dart
import 'widgets/data.dart';

void main() => runApp(const DymaTrip());

class DymaTrip extends StatelessWidget {
  const DymaTrip({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Data(
        child: City(),
      ),
    );
  }
}
```

Comme nous le plaçons à un niveau très haut, tous les `widgets` en dessous du `widget City` et le `widget CIty` peuvent maintenant accéder aux données en passant par l'`Inherited Widget`.

Nous allons modifier `city.dart` pour accéder à nos `activities` depuis notre `Inherited Widget` :

```dart
import 'package:flutter/material.dart';
import '../../widgets/data.dart';
import 'widgets/trip_activity_list.dart';
import 'widgets/activity_list.dart';
import 'widgets/trip_overview.dart';
import '../../data/data.dart' as data;
import '../../models/activity.model.dart';
import '../../models/trip.model.dart';

class City extends StatefulWidget {
  final List<Activity> activities = data.activities;

  City({super.key});
  @override
  State<City> createState() => _CityState();
}

class _CityState extends State<City> {
  late Trip mytrip;
```

```dart
  late int index;
  late List<Activity> activities;

  @override
  void initState() {
    super.initState();
    mytrip = Trip(activities: [], city: 'Paris', date: null);
    index = 0;
  }

  @override
  didChangeDependencies() {
    super.didChangeDependencies();
    activities = Data.of(context).activities;
  }

  List<Activity> get tripActivities {
    return activities
        .where((activity) => mytrip.activities.contains(activity.id))
        .toList();
  }

  void setDate() {
    showDatePicker(
      context: context,
      firstDate: DateTime.now(),
      initialDate: DateTime.now().add(const Duration(days: 1)),
      lastDate: DateTime(2025),
    ).then((newDate) {
      if (newDate != null) {
        setState(() {
          mytrip.date = newDate;
        });
      }
    });
  }

  void switchIndex(newIndex) {
    setState(() {
      index = newIndex;
    });
  }

  void toggleActivity(String id) {
    setState(() {
```

```
      mytrip.activities.contains(id)
          ? mytrip.activities.remove(id)
          : mytrip.activities.add(id);
    });
  }

  void deleteTripActivity(String id) {
    setState(() {
      mytrip.activities.remove(id);
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        leading: const Icon(Icons.chevron_left),
        title: const Text('Organisation du voyage'),
        actions: const <Widget>[
          Icon(Icons.more_vert),
        ],
      ),
      body: Column(
        children: [
          TripOverview(
            mytrip: mytrip,
            setDate: setDate,
          ),
          Expanded(
            child: index == 0
                ? ActivityList(
              activities: activities,
              selectedActivities: mytrip.activities,
              toggleActivity: toggleActivity,
            )
                : TripActivityList(
              activities: tripActivities,
              deleteTripActivity: deleteTripActivity,
            ),
          )
        ],
      ),
      bottomNavigationBar: BottomNavigationBar(
        currentIndex: index,
        items: const [
```

```
            BottomNavigationBarItem(
              icon: Icon(Icons.map),
              label: 'Découverte',
            ),
            BottomNavigationBarItem(
              icon: Icon(Icons.stars),
              label: 'Mes activités',
            )
          ],
          onTap: switchIndex,
        ),
      );
    }
}
```

Nous faisons les modifications nécessaires car nous n'accédons plus aux données depuis le fichier mais depuis notre `InheritedWidget`.

La méthode `didChangeDependencies()` est appelée lors de chaque `rebuild`, elle est également appelée par `updateShouldNotify()` de notre `InheritedWidget`.

C'est pour cette raison que nous pouvons placer `activities = Data.of(context).activities;` soit dans ce `hook` soit dans la méthode `build`.

Nous préférons vous montrer ce `hook` car nous le retrouverons parfois dans `Flutter`.

Nous avons terminé notre exemple d'utilisation. Nous allons voir des cas d'utilisation du `InheritedWidget` dans les prochaines leçons.