



Séance 2 – Le mental model Compose

1. Objectifs de la séance

Dans la séance précédente, nous avons vu que Jetpack Compose ne consiste pas seulement à remplacer les fichiers XML par du Kotlin.

Compose introduit une autre manière de penser l'interface Android : au lieu de manipuler des vues déjà créées, on décrit l'interface à afficher à partir de données.

Dans cette séance, nous allons clarifier ce modèle mental.

À la fin de la séance, vous devez être capables de :

- expliquer ce qu'est une fonction composable ;
- comprendre qu'une interface Compose est organisée sous forme d'arbre ;
- expliquer ce qu'est la recomposition ;
- comprendre pourquoi les paramètres sont essentiels en Compose ;
- distinguer données affichées et événements utilisateur ;
- comprendre le principe du flux de données unidirectionnel.

Le but n'est pas encore de construire une architecture complète. Nous préparons les bases nécessaires avant d'aborder **l'état**, les écrans plus riches, puis plus tard **ViewModel**, **StateFlow** et **MVVM**.

2. Rappel : Compose décrit l'interface

Dans une approche **Android classique**, vous avez souvent manipulé directement des vues :

```
title.text = "Produit ajouté"
button.isEnabled = false
progressBar.visibility = View.GONE
```

Dans **Compose**, on préfère décrire l'interface attendue :

```
@Composable
fun ProductButton(isAdded: Boolean) {
    Button(onClick = { }) {
        Text(
            text = if (isAdded) "Produit ajouté" else "Ajouter au panier"
        )
    }
}
```

La différence est importante.

- Dans le premier cas, on **donne des ordres** à des composants graphiques.
- Dans le second cas, on **décrit** ce que l'interface doit afficher selon une donnée.

La formule à retenir est :

État de l'application → Interface affichée



Si l'état change, Compose peut mettre à jour l'interface.

3. Les fonctions composables

Une **fonction composable** est une fonction Kotlin annotée avec **@Composable**. Cette fonction ne retourne pas une vue. Elle décrit une **portion d'interface**.

Une fonction composable peut appeler d'autres fonctions composables :

```
@Composable
fun ProductCard() {
    Surface {
        Text(text = "Produit")
    }
}
```

Ici, **ProductCard()** utilise **Surface**, pour créer une zone visuelle et **Text**, pour afficher du texte.

Une interface **Compose** est donc construite en assemblant plusieurs composables.

Un bon composable doit être :

- simple ;
- lisible ;
- réutilisable ;
- rapide à exécuter ;
- concentré sur l'affichage.

Un composable ne doit pas devenir l'endroit où l'on mélange toute la logique métier, les appels réseau, les accès base de données ou les traitements lourds.

4. L'arbre de composition

Une interface **Compose** est organisée sous forme d'arbre. Par exemple :

```
@Composable
fun ProductScreen() {
    Surface {
        ProductCard(
            name = "Sac à dos",
            price = 49.99
        )
    }
}

@Composable
fun ProductCard(
    name: String,
    price: Double
) {
    Text(text = "$name - $price €")
}
```

On peut représenter cela ainsi :

```
ProductScreen
├── Surface
│   └── ProductCard
│       └── Text
```



Chaque composable peut appeler d'autres composables. Cela crée une hiérarchie.

Avec XML, vous aviez aussi une hiérarchie de vues. Mais avec **Compose**, cette hiérarchie est écrite en Kotlin et peut dépendre directement des données.

Exemple :

```
@Composable
fun ProductAvailability(isAvailable: Boolean) {
    if (isAvailable) {
        Text(text = "Produit disponible")
    } else {
        Text(text = "Produit indisponible")
    }
}
```

L'interface affichée dépend ici de la valeur de **isAvailable**.

5. La recomposition

La recomposition est le mécanisme par lequel **Compose** réexécute certaines fonctions composables lorsque les données dont elles dépendent changent.

Ici par exemple, si **isFavorite** change, le texte du bouton peut changer :

```
@Composable
fun FavoriteButton(isFavorite: Boolean) {
    Button(onClick = { }) {
        Text(
            text = if (isFavorite) "★ Favori" else "☆ Ajouter"
        )
    }
}
```

Vous n'avez pas besoin de **récupérer le bouton** pour modifier son texte. Vous donnez une nouvelle valeur, et **Compose** met à jour l'interface concernée.

Il faut retenir deux points importants :

1. Un composable peut être appelé **plusieurs** fois.
2. **Compose** essaie de **recomposer** seulement ce qui est nécessaire.

C'est pour cela qu'il faut éviter de faire directement dans un composable :

- un appel réseau ;
- une écriture en base de données ;
- une modification de variable globale ;
- un calcul lourd répété ;
- une action qui ne devrait être exécutée qu'une seule fois.

Pour l'instant, on retient une règle simple : **Un composable décrit l'interface. Il ne doit pas porter toute la logique applicative.**



6. Les paramètres

Les paramètres sont essentiels en **Compose**. Un composable doit recevoir les données dont il a besoin pour afficher l'interface.

Exemple **peu réutilisable**, ce composable affiche toujours le même texte :

```
@Composable
fun ProductTitle() {
    Text(text = "Sac à dos")
}
```

Version plus intéressante, le composable peut afficher **n'importe quel produit** :

```
@Composable
fun ProductTitle(name: String) {
    Text(text = name)
}
```

Autre exemple :

```
@Composable
fun ProductPrice(price: Double) {
    Text(text = "$price €")
}
```

Grâce aux paramètres, on peut créer des composants réutilisables.

7. Données et événements

Un composable peut recevoir deux types importants de paramètres.

1. D'abord, des **données** à afficher : **isFavorite: Boolean** ;
2. Ensuite, des **fonctions** « callback » à appeler quand l'utilisateur fait une action : **onFavoriteClick: () -> Unit**.

Exemple :

```
@Composable
fun FavoriteButton(
    isFavorite: Boolean,
    onFavoriteClick: () -> Unit
) {
    Button(onClick = onFavoriteClick) {
        Text(
            text = if (isFavorite) "★ Favori" else "☆ Ajouter"
        )
    }
}
```

Ici, le bouton ne décide pas lui-même comment gérer les favoris. Il reçoit l'état actuel avec **isFavorite**. Il signale le clic avec **onFavoriteClick**.

Cette séparation est très importante.

8. Flux de données unidirectionnel

Le **flux de données unidirectionnel**, ou **UDF**, peut se résumer ainsi :



- Les **données descendent**.
- Les **événements remontent**.

Cela signifie :

- un composable parent fournit les données à ses enfants ;
- les enfants affichent ces données ;
- quand l'utilisateur interagit, les enfants déclenchent des événements ;
- le parent ou une couche supérieure décide quoi faire ;
- l'état change ;
- la nouvelle interface est affichée.

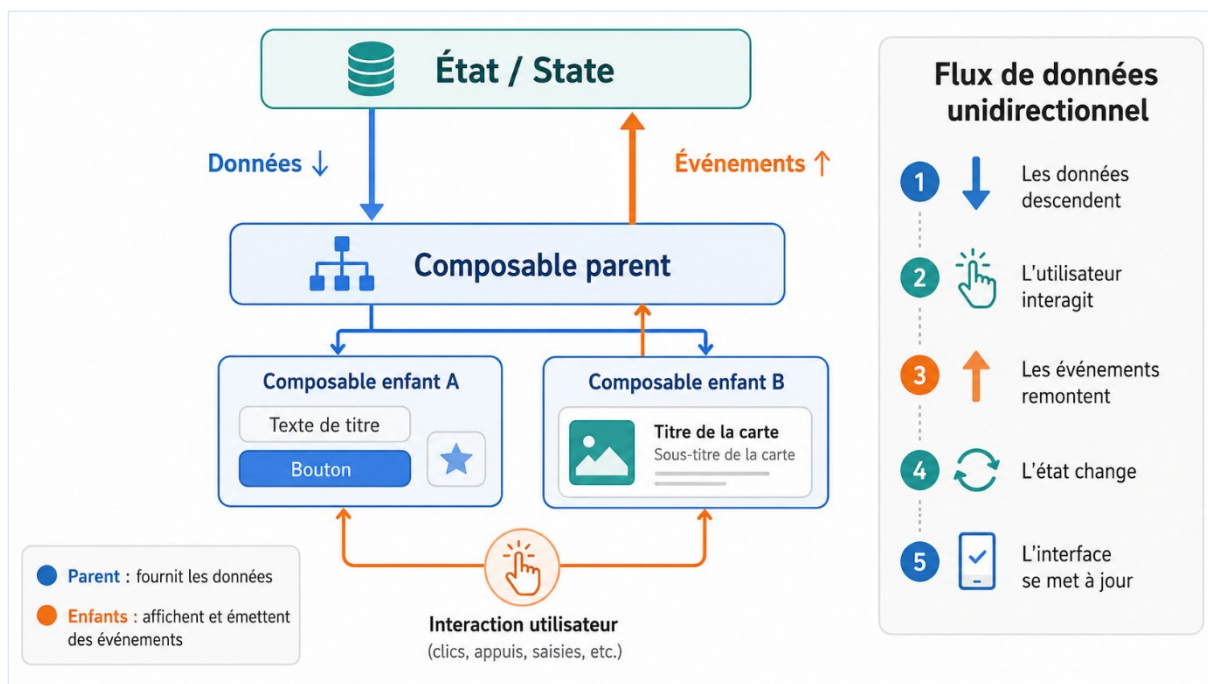
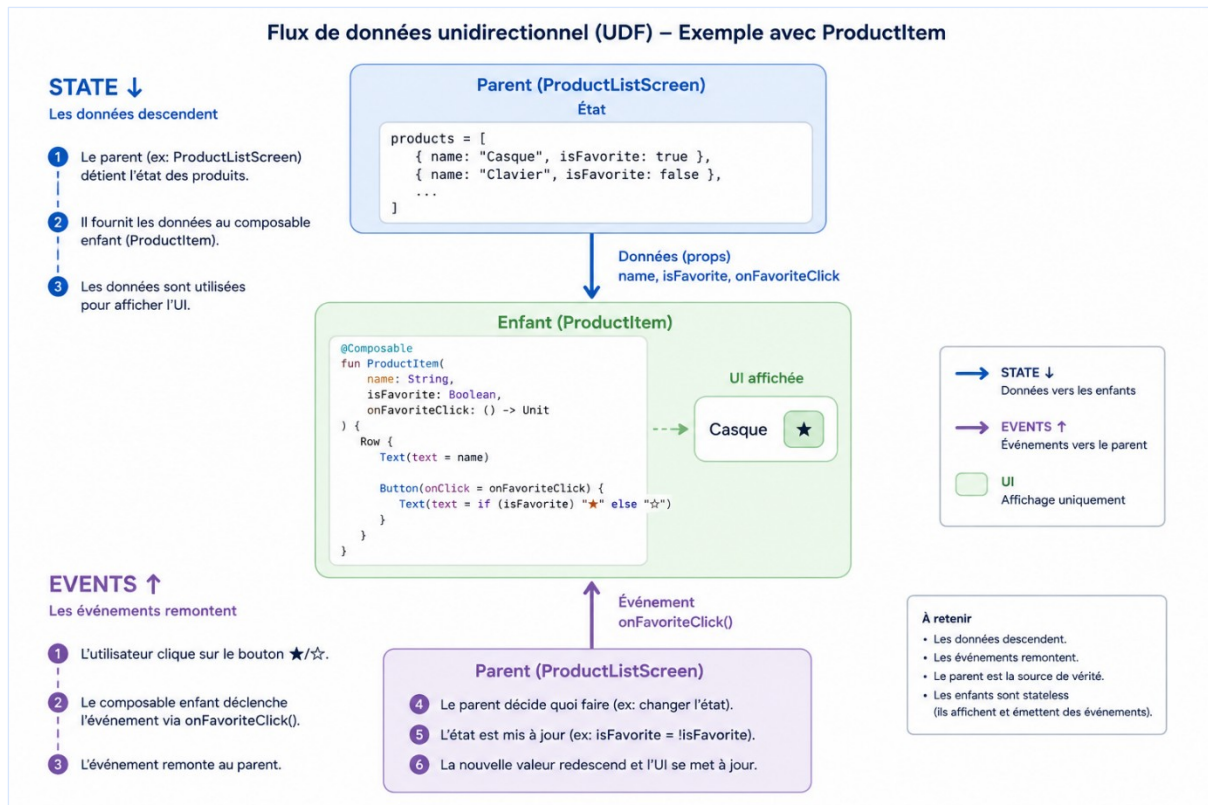


Schéma : State ↓ vers UI, Events ↑ vers parent

Exemple :

```
@Composable
fun ProductItem(
    name: String,
    isFavorite: Boolean,
    onFavoriteClick: () -> Unit
) {
    Row {
        Text(text = name)

        Button(onClick = onFavoriteClick) {
            Text(
                text = if (isFavorite) "★" else "☆"
            )
        }
    }
}
```



Ce composable ne sait pas d'où viennent les données. Il ne sait pas si le produit vient d'une API, d'une base locale ou d'une liste temporaire.

Son rôle est **d'afficher une interface** et de **signaler les actions** utilisateur.

C'est exactement le raisonnement que nous utiliserons plus tard dans nos TP et nos projets.

9. Ce qu'il faut retenir

Avant de passer au TP, reprenez ces idées :

- Une fonction **composable décrit** une portion d'interface.
- Une **interface Compose** est construite comme un **arbre** de composables.
- La **recomposition** permet de **mettre à jour** l'interface quand les données changent.
- Un composable peut être **rappelé plusieurs fois**.
- Les **paramètres** permettent de rendre les composables **réutilisables**.
- Les **données descendent** vers les composables.
- Les **événements remontent** vers le parent.
- Un composable doit rester **centré sur l'affichage**.

Dans le TP, nous allons appliquer ces idées progressivement avec un petit écran dynamique.