



Android Jetpack Compose

Bloc 1 – Thinking in Compose

Séance 1 – Android moderne et Jetpack Compose

1. Objectifs de la séance

Dans cette première séance, nous allons poser les bases du développement Android moderne avec **Jetpack Compose**.

Vous avez déjà travaillé sur Android avec Java et XML. Vous avez également déjà étudié Kotlin. L'objectif de cette séance n'est donc pas de reprendre Android depuis zéro, mais de comprendre le changement de logique introduit par **Jetpack Compose**.

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

- expliquer ce qu'est Jetpack Compose ;
- comprendre la différence entre une interface **XML** et une interface **Compose** ;
- distinguer une approche **impérative** d'une approche **déclarative** ;
- créer un projet Android utilisant **Jetpack Compose** ;
- identifier les premières fonctions **composables** d'un projet ;
- modifier un premier écran **Compose** ;
- comprendre le rôle de **setContent**, **@Composable**, **Text**, **Surface** et **Modifier**.

2. Pourquoi parle-t-on d'Android moderne ?

Le développement Android a beaucoup évolué.

Pendant longtemps, une application Android classique reposait principalement sur :

- des fichiers XML pour décrire les interfaces ;
- des classes Java ou Kotlin pour gérer la logique ;
- des **Activity** et parfois des **Fragment** pour organiser les écrans ;
- des appels comme `findViewById()` pour récupérer des composants graphiques ;
- des **listeners** pour réagir aux actions de l'utilisateur.

Exemple simplifié d'une approche classique :

```
val title = findViewById<TextView>(R.id.title)
title.text = "Shop Explorer"
```

Dans cette approche, le développeur manipule directement les éléments de l'interface. Il récupère une vue, puis il change son texte, sa couleur, sa visibilité ou son comportement.

Cela fonctionne, mais plus une interface devient riche, plus cette approche peut devenir lourde :



- beaucoup de fichiers différents ;
- une séparation parfois confuse entre XML et code ;
- des mises à jour manuelles de l'interface ;
- des risques d'incohérence entre l'état réel de l'application et ce qui est affiché à l'écran.

Jetpack Compose propose une autre manière de penser.

3. Qu'est-ce que Jetpack Compose ?

Jetpack Compose est le *toolkit moderne de Google* pour construire des interfaces Android natives.

Avec **Compose**, l'interface n'est plus principalement écrite en **XML**. Elle est écrite directement en Kotlin, à l'aide de fonctions particulières appelées **fonctions composables**.

Exemple :

```
@Composable
fun WelcomeMessage() {
    Text(text = "Bienvenue dans Shop Explorer")
}
```

Ici, la fonction `WelcomeMessage()` décrit une partie de l'interface. Elle ne retourne pas une chaîne, un objet graphique ou une vue XML. Elle décrit ce qui doit être affiché à l'écran.

Le mot-clé important est l'**annotation** : `@Composable`. Cette annotation indique au compilateur que la fonction participe à la construction de l'interface utilisateur.

4. XML vs Jetpack Compose

4.1 Ancienne approche : XML + Kotlin/Java

Dans une application Android classique, l'interface est souvent décrite dans un fichier **XML** :

```
<TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Shop Explorer" />
```

Puis le code Kotlin ou Java vient manipuler cette interface :

```
val title = findViewById<TextView>(R.id.title)
title.text = "Nouveau titre"
```

On a donc deux mondes :

- le XML pour décrire l'interface ;
- Kotlin ou Java pour la logique.

4.2 Nouvelle approche : Compose

Avec **Compose**, l'interface est écrite directement en Kotlin :



```
@Composable
fun ShopTitle() {
    Text(text = "Shop Explorer")
}
```

Le composant **Text** remplace ici le **TextView**.

L'intérêt n'est pas seulement de supprimer le XML. Le vrai changement est plus profond : **Compose change la manière de penser l'interface**.

5. UI impérative vs UI déclarative

5.1 UI impérative

Dans une interface **impérative**, on explique à l'application comment modifier l'interface.

Exemple :

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

Le développeur donne des instructions précises : *change ce texte ; désactive ce bouton ; cache cette barre de progression*.

Cette approche est **directe**, mais elle peut devenir difficile à maintenir quand l'écran dépend de nombreux états.

5.2 UI déclarative

Dans une interface **déclarative**, on décrit l'interface à afficher en fonction d'un état.

Exemple :

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

Ici, on ne dit pas : « Va chercher le bouton, puis change son texte. »

On dit plutôt : « Si le produit est ajouté, le bouton affiche "**Produit ajouté**". Sinon, il affiche "**Ajouter au panier**". »

C'est une différence fondamentale ! L'interface devient une **conséquence de l'état** de l'application.

6. Le principe central de Compose

On peut résumer Compose avec cette idée : **État de l'application → Interface affichée**.

Autrement dit, l'interface est calculée à partir des données disponibles à un instant donné.

Si les données changent, **Compose** peut **réexécuter les fonctions composables** concernées et mettre à jour l'interface. Ce mécanisme s'appelle la **recomposition**.



Nous l'étudierons plus en détail dans la séance suivante. Pour aujourd'hui, reprenez simplement ceci :

- une **fonction composable décrit** une portion d'interface ;
- cette fonction peut recevoir des **paramètres** ;
- si les **paramètres changent, l'interface peut changer** ;
- **Compose** se charge de **mettre à jour** ce qui doit l'être.

7. Exemple simple : une fiche-produit

Imaginons une application de boutique en ligne. Nous voulons afficher une fiche-produit.

Avec Compose, nous pouvons écrire :

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

Cette fonction reçoit deux données : **name** - le nom du produit et **price** - le prix du produit et elle produit une interface affichant ces informations.

On peut appeler cette fonction ainsi :

```
ProductCard(
    name = "Sac à dos",
    price = 49.99
)
```

La fonction composable ne va pas chercher elle-même les données. Elle les reçoit. C'est une bonne pratique importante : *les composants d'interface doivent être aussi réutilisables que possible.*

8. Les premiers éléments d'un projet Compose

Quand vous créez un projet Android avec **Jetpack Compose**, vous obtenez généralement une classe **MainActivity**.

Elle ressemble à ceci :

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyApplicationTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    WelcomeScreen(
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

Regardons les éléments importants.



Ce code contient plusieurs éléments importants :

- MainActivity
- onCreate
- enableEdgeToEdge
- setContent
- le thème de l'application
- Scaffold
- innerPadding
- le premier composable affiché à l'écran

Nous allons les analyser progressivement.

8.1 MainActivity

```
class MainActivity : ComponentActivity()
```

MainActivity est l'activité principale de l'application.

Une **Activity** est un **composant Android** qui représente généralement un écran ou un point d'entrée visuel de l'application.

Même si **Jetpack Compose** change la manière de construire les interfaces, nous restons dans une application **Android classique**. Les concepts Android comme les **activités**, le **cycle de vie**, le **manifeste**, les **ressources** ou les **permissions** existent toujours.

ComponentActivity est une classe de base adaptée aux applications modernes, notamment aux applications utilisant **Compose**.

8.2 onCreate

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
}
```

La méthode **onCreate()** est appelée au moment de la création de l'activité. C'est ici que l'on initialise l'écran principal de l'application.

Dans une application Android plus ancienne, on aurait souvent trouvé une instruction comme : **setContentView(R.layout.activity_main)**. Cette instruction permettait d'associer un fichier XML à l'activité.

Avec **Jetpack Compose**, on ne charge plus un fichier XML. On définit directement l'interface avec du code Kotlin à l'intérieur de **setContent**.

Même avec **Compose**, on reste dans une application Android classique. **Compose** ne supprime pas les concepts Android : il modernise principalement la création de l'interface.

8.3 enableEdgeToEdge()

```
enableEdgeToEdge()
```

Cette fonction active un affichage dit **edge-to-edge**. Cela signifie que l'application peut dessiner son interface **derrière les barres système Android**, par exemple : la barre de statut en haut, la barre de navigation en bas, certaines zones liées aux découpes d'écran ou aux gestes système.



L'objectif est d'utiliser davantage l'espace disponible à l'écran et d'obtenir une interface plus moderne.

Attention : activer l'affichage **edge-to-edge** ne signifie pas que l'on peut ignorer les barres système. Il faut ensuite s'assurer que les contenus importants ne sont pas cachés derrière elles.

C'est précisément pour cela que le **template Compose** utilise aussi **Scaffold** et **innerPadding**.

8.4 setContent

```
setContent {  
    // Interface Compose  
}
```

La fonction **setContent** indique que le contenu graphique de l'activité sera construit avec **Jetpack Compose**. Tout ce qui se trouve à l'intérieur du bloc **setContent** correspond à l'interface **Compose** de l'écran.

Avec XML, on associait une activité à un fichier de **layout**. Avec **Compose**, on associe une activité à un **arbre de fonctions composables**.

8.5 Le thème

```
MyApplicationTheme {  
    // Contenu de l'application  
}
```

Le thème permet de définir l'apparence générale de l'application :

- couleurs ;
- typographies ;
- formes ;
- thème clair ou sombre ;
- règles Material Design.

Dans un projet généré automatiquement, Android Studio crée généralement un thème dans le package **ui.theme**.

Pour l'instant, nous utiliserons le thème généré par défaut. Plus tard, nous le modifierons pour construire une vraie identité visuelle avec Material Design 3.

8.6 Le Scaffold

```
Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->  
    WelcomeScreen(  
        modifier = Modifier.padding(innerPadding)  
    )  
}
```

Scaffold (échafaudage) est un composant **Material Design** qui sert de structure de base pour un écran. Il permet d'organiser les grandes zones d'une interface : *barre supérieure*, *barre inférieure*, *bouton flottant*, *menu de navigation*, *contenu principal*.

Google décrit **Scaffold** comme une structure fondamentale permettant d'assembler les différentes parties d'une interface complexe, par exemple les **app bars** et les **floating action buttons**.



Le paramètre `innerPadding` représente les **marges internes** calculées par **Scaffold**. **Scaffold** transmet des **PaddingValues** au contenu, et que ces valeurs doivent être appliquées au **composable racine** du contenu pour contraindre correctement sa taille. C'est pour cette raison que le *template* contient : `modifier = Modifier.padding(innerPadding)`. Sans cette ligne, le contenu pourrait se retrouver **trop près des bords**, ou **être masqué** par certaines zones système ou par des composants du **Scaffold**.

Même si notre premier écran est très simple, Android Studio l'utilise déjà car c'est une bonne base pour construire des écrans modernes. Exemple plus avancé, que nous verrons plus tard :



Dans ce cas, **Scaffold** devient le squelette de l'écran :

1. `topBar` affiche la barre supérieure ;
2. `bottomBar` affiche la barre inférieure ;
3. `floatingActionButton` affiche un bouton flottant ;
4. `navigationIcon` dans `topBar` affiche l'Icon de navigation supérieur et `NavigationBar` dans `bottomBar` affiche un menu de navigation avec 3 entrées ;
5. le bloc final affiche le contenu principal.

8.7 Modifier.fillMaxSize()

```
Scaffold(modifier = Modifier.fillMaxSize())
```

`Modifier.fillMaxSize()` indique que le **Scaffold** doit occuper tout l'espace disponible. Sans cela, certains composants peuvent ne prendre que l'espace nécessaire à leur contenu.

Ici, on veut que la structure principale de l'écran occupe toute la surface de l'activité.

9. Les fonctions composables

Une fonction **composable** est une fonction Kotlin annotée avec `@Composable`.



Par exemple `WelcomeScreen` est une fonction composable qui représente le premier écran de notre application :

```
@Composable
fun WelcomeScreen() {
    Scaffold(...) { innerPadding ->
        Text(
            text = "Contenu de la page",
            modifier = Modifier.padding(innerPadding)
        )
    }
}
```

Une fonction **composable** respecte plusieurs principes :

- elle décrit une partie de l'interface ;
- elle peut appeler d'autres fonctions composables ;
- elle ne retourne généralement rien ;
- son nom commence souvent par une majuscule ;
- elle doit rester rapide à exécuter ;
- elle doit éviter les effets de bord non contrôlés.

Il est important de remarquer que le `modifier` est passé en **paramètre**. C'est une bonne pratique en **Compose** : un **composable** réutilisable doit souvent accepter un `Modifier` optionnel. Ainsi, le parent peut décider de la **position** ou de l'**espace** occupé par le composant, sans que le composant impose tout lui-même.

Un autre exemple avec paramètres :

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

L'intérêt est de pouvoir construire une interface par assemblage de petits composants.

10. Le "Modifier"

Dans **Compose**, `Modifier` sert à **modifier l'apparence**, la **disposition** ou le **comportement** d'un composant.

Par exemple ici, on ajoute une **marge intérieure** autour du texte :

```
Text(
    text = "Shop Explorer",
    modifier = Modifier.padding(16.dp)
)
```

Les `modifier` peuvent être chaînés :

```
Text(
    text = "Shop Explorer",
    modifier = Modifier
        .padding(16.dp)
        .fillMaxWidth()
)
```



L'ordre des **modifieurs** peut avoir de l'importance. Par exemple, appliquer un **padding** avant ou après une **couleur de fond** ne produit pas toujours le même résultat. Nous approfondirons ce point dans les séances à venir.

11. Surface

Surface est un conteneur proposé par **Material Design 3**. Il permet de représenter une zone visuelle cohérente dans l'interface :

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

On peut lui donner une **couleur**, une **forme**, une **élévation** ou un **modifier**.

Par exemple :

```
@Composable
fun ProductCard() {
    Surface(
        modifier = Modifier.padding(16.dp)
    ) {
        Text(text = "Produit")
    }
}
```

Dans les prochains cours, nous utiliserons plutôt des composants comme **Card**, **Scaffold**, **LazyColumn**, **TopAppBar** et les composants **Material 3**. Mais **Surface** est un bon premier **conteneur** pour comprendre la logique **Compose**.

12. Preview Compose

Compose propose un système très pratique : **les previews**. Une **preview** permet d'afficher un composant directement dans Android Studio, sans lancer complètement l'application sur un émulateur ou un téléphone.

Exemple :

```
@Preview(showBackground = true)
@Composable
fun WelcomeScreenPreview() {
    WelcomeScreen()
}
```

La **Preview** est un outil important pour travailler rapidement sur les interfaces.

Elle permet de tester un composant isolé, de vérifier rapidement le rendu, de comparer plusieurs états visuels et surtout – **d'éviter de relancer toute l'application à chaque modification**.

Dans un projet professionnel, les **previews** peuvent devenir un outil très efficace pour développer et documenter des composants UI.