DAA - Développement d'applications Android

Jetpack Compose

02 January 2026

Table des matières

1 Jetpack Compose

- Jetpack Compose est une API déclarative permettant de définir l'UI
- Uniquement disponible en Kotlin
- Basé sur une approche « Qu'est-ce que je veux faire » au lieu de « Comment est-ce que je vais le faire »
- Possibilité de créer des composants UI, réutilisables et facilement testables

Jetpack Compose est un ensemble de librairies, ce qui peut-être difficile de gérer la compatibilité entre les versions. C'est pourquoi Gradle propose l'utilisation d'un Bill of Materials (BOM) pour gérer les versions des dépendances Compose.

1.1 Exemple simple

Une activité doit hériter de `ComponentActivity` et utiliser la fonction `setContent` pour définir l'UI avec Compose.

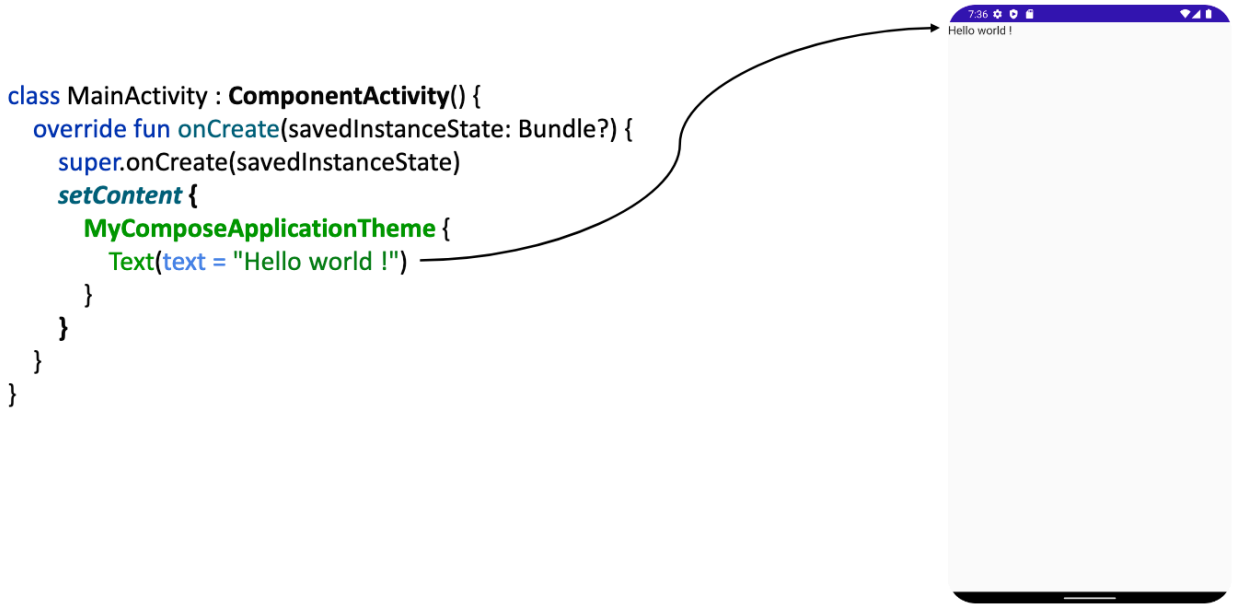

Fig. 1. – Capture des slides du cours – Exemple d'une activité avec Jetpack Compose

1.2 Fonction @Composable

Les fonctions annotées avec `@Composable` permettent de définir des composants UI. Elles peuvent être imbriquées pour créer des interfaces complexes.

```
@Composable
fun Hello() {
  Text(text = "Hello world !")
}
// On peut remplacer le contenu de setContent par notre fonction composable
setContent {
  MyComposeApplicationTheme {
    Hello()
  }
}
```

1.2.1 Fonctions avec paramètres

Les fonctions composables peuvent accepter des paramètres pour rendre les composants plus dynamiques.

```
@Composable
fun Hello(name: String) {
  Text(text = "Hello $name !")
}

setContent {
  MyComposeApplicationTheme {
    Hello(name = "Android")
  }
}
```

> **i Info**
>
> Une fonction composable peut être exécutée très fréquemment. C'est le cas par exemple pour un composant effectuant une animation (60 fps <=> 16.6 ms)

Jetpack Compose permet d'optimiser le processus de recomposition:

- Eviter de recomposer un composant qui ne change pas
- Recompositions en parallèle (multi-threading)

Cela implique qu'une fonction composable doit:

- Être rapide à s'exécuter
- Eviter les effets de bords, en particulier:
  - Ne pas modifier de variables externes
  - Ne pas réaliser d'opérations I/O
- Être idempotente (même entrée => même sortie)

1.2.2 Prévisualisation dans Android Studio

Il est possible de prévisualiser une fonction composable directement dans Android Studio en utilisant l'annotation `@Preview`.

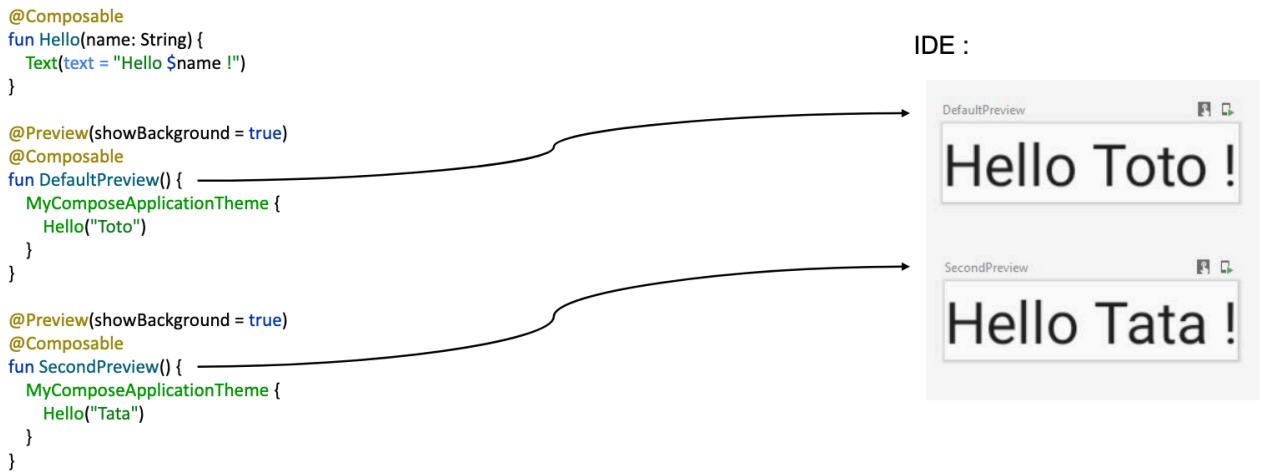

Fig. 2. – Capture des slides du cours – Exemple d'une prévisualisation avec Jetpack Compose

1.3 Layout

Les layouts permettent d'organiser les composants UI à l'écran. Jetpack Compose propose plusieurs layouts de base, tels que `Column`, `Row` et `Box`.

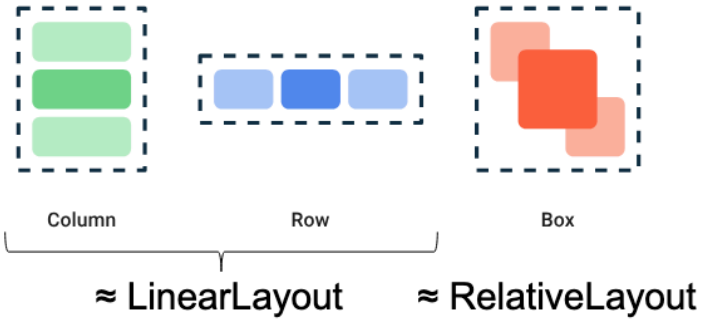

Fig. 3. – Capture des slides du cours – Exemple d'utilisation des layouts Column et Row

1.3.1 Column

Le layout `Column` organise les composants verticalement.

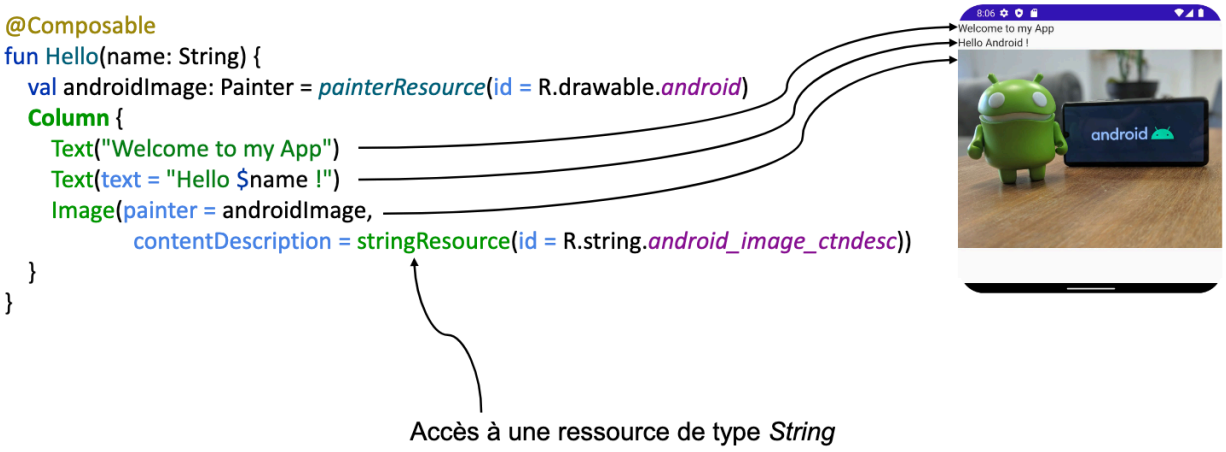

Fig. 4. – Capture des slides du cours – Exemple d'un layout Column

La fonction `Column` accepte plusieurs paramètres optionnels pour personnaliser l'agencement des éléments:

- `modifier` : Permet de modifier l'apparence ou le comportement du layout (ex: taille, marges, etc.)
- `verticalArrangement` : Définit l'espacement vertical entre les éléments
- `horizontalAlignment` : Définit l'alignement horizontal des éléments
- `content` : Contient les éléments enfants à afficher dans la colonne

```
Column(content = {
  Text(text = "Premier élément")
  Text(text = "Deuxième élément")
  Text(text = "Troisième élément")
})
```

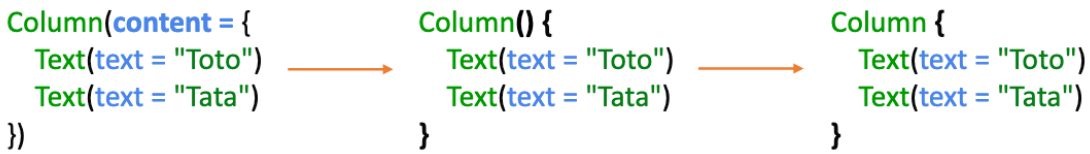

Fig. 5. – Capture des slides du cours – Exemple d'un layout Row

1.3.2 Row

Le layout `Row` organise les composants horizontalement.

```
@Composable
fun Hello() {
  Row {
    Button(onClick = {}) {
      Text(text = "Un")
    }
    Button(onClick = {}) {
      Text(text = "Deux")
    }
    Button(onClick = {}) {
      Text(text = "Trois")
    }
  }
}
```

Fig. 6. – Capture des slides du cours – Exemple d'un layout Row

La fonction `Row` accepte plusieurs paramètres optionnels similaires à ceux de `Column`. Elle propose également:

- `horizontalArrangement` : Définit l'espacement horizontal entre les éléments avec par exemple `Arrangement.SpaceBetween` pour espacer les éléments de manière égale.

1.4 Eléments de base

Jetpack Compose propose plusieurs composants UI de base, tels que `Text`, `Button` et `Image`.

1.4.1 Scaffold

Le composant `Scaffold` fournit une structure de base pour une application, incluant des éléments comme la barre d'application, le tiroir de navigation et le bouton d'action flottant.

```
setContent {
  MyComposeApplicationTheme {
    Scaffold(
      modifier = Modifier.fillMaxSize(),
      topBar = {
        TopAppBar(
          title = {Text(text = "Accueil")},
          navigationIcon = {
            IconButton(onClick = { /* ... */ }) { Icon(painterResource(R.drawable.home), contentDescription = "Home") }
          }
        )
      },
      bottomBar = {
        BottomAppBar(
          actions = {
            IconButton(onClick = { /* ... */ }) { Icon(painterResource(...), contentDescription = "Check")}
            IconButton(onClick = { /* ... */ }) { Icon(painterResource(...), contentDescription = "Edit")}
            IconButton(onClick = { /* ... */ }) { Icon(painterResource(...), contentDescription = "Build")}
            IconButton(onClick = { /* ... */ }) { Icon(painterResource(...), contentDescription = "Info")}
          },
          floatingActionButton = {
            FloatingActionButton(onClick = { /* ... */ }) {Icon(painterResource(...), contentDescription = null) }
          },
        )
      }
    )

    { /* contenu */ }
  }
}
```

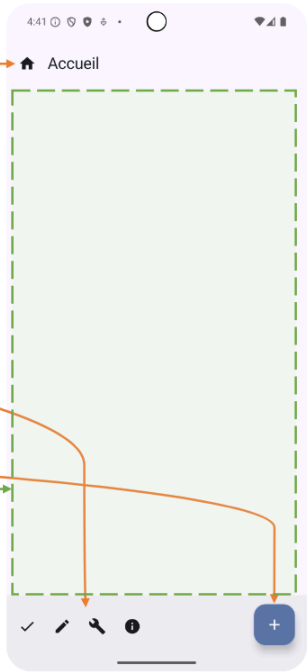

Fig. 7. – Capture des slides du cours – Exemple d'un Scaffold avec une TopAppBar et un FloatingActionButton

Le Scaffold accepte plusieurs paramètres pour personnaliser son apparence et son comportement:

- `topBar` : Permet de définir une barre d'application en haut de l'écran
- `bottomBar` : Permet de définir une barre en bas de l'écran

- `floatingActionButton` : Permet d'ajouter un bouton d'action flottant
- `content` : Contient le contenu principal de l'écran

1.4.2 Fonctions composables paresseuses

À la place des ListView ou RecyclerView, il existe les layouts paresseux:

- `LazyColumn` : Pour afficher une liste verticale
- `LazyRow` : Pour afficher une liste horizontale
- `LazyVerticalGrid` : Pour afficher une grille verticale

Ces vues sont scrollables et n'affichent que les éléments visibles à l'écran, ce qui améliore les performances. Cependant, les vues ne sont pas recyclées, elles sont systématiquement recomposées.

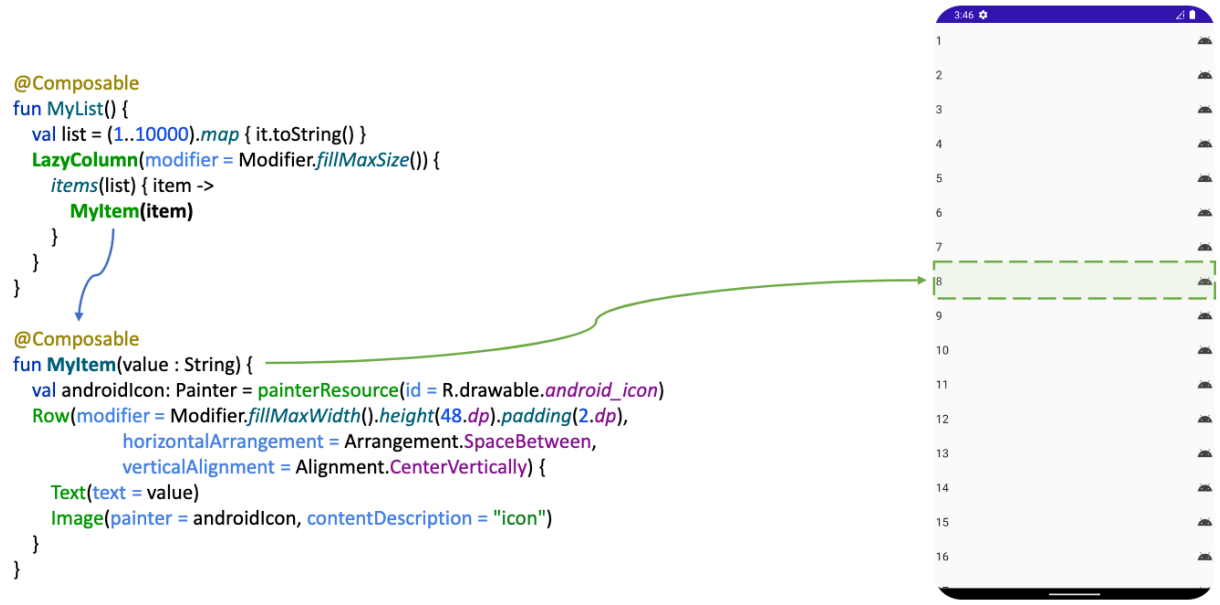

Fig. 8. – Capture des slides du cours – Exemple d'une LazyColumn

1.4.2.1 Gestion d'événements

Il est possible de gérer des événements sur des actions utilisateurs en utilisant des paramètres de type lambda dans les fonctions composables.

Pour ajouter l'affichage d'un `toast` lors du clic sur une ligne, nous pouvons redéfinir `MyItem` :

```
@Composable
fun MyItem(value: String) {
  val context = LocalContext.current
  val androidIcon: Painter = painterResource(id = R.drawable.android_icon)
  Row(modifier = Modifier.fillMaxWidth()
      .height(48.dp)
      .padding(2.dp)
      .clickable {
        Toast.makeText(context, "Vous avez cliqué sur $value", Toast.LENGTH_SHORT).show()
      },
      horizontalArrangement = Arrangement.SpaceBetween,
      verticalAlignment = Alignment.CenterVertically {
        Text(text = value)
        Icon(painter = androidIcon, contentDescription = "Android Icon")
      }
    )
}
```

2 Gestion des états