DAA - Développement d'applications Android

Jetpack Compose

03 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.

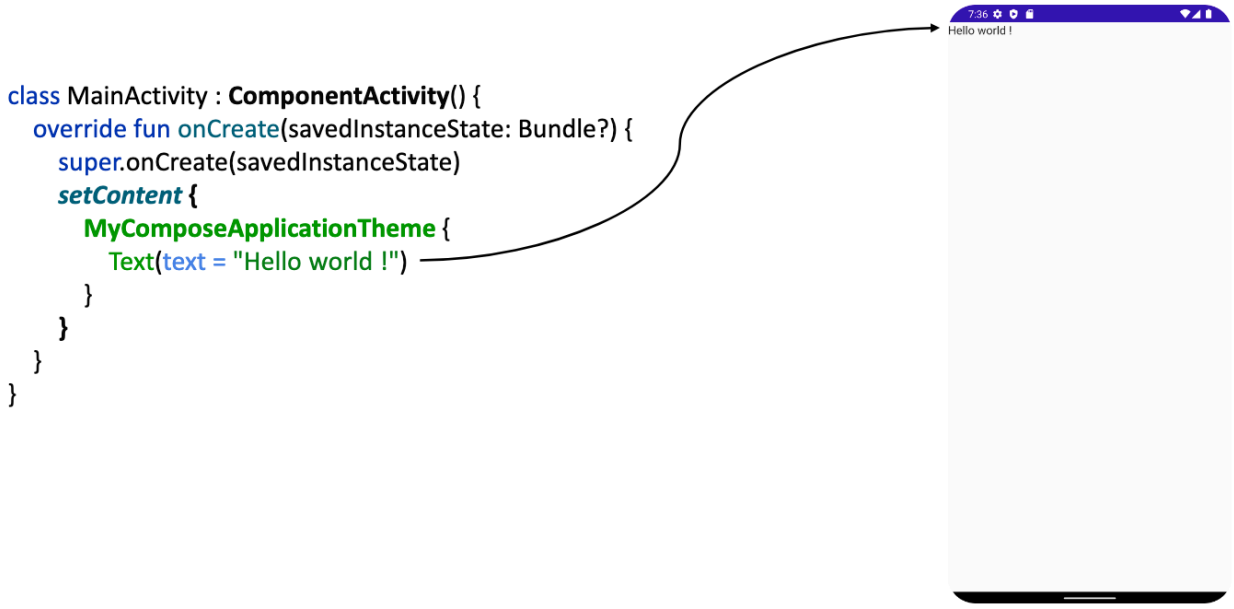

Fig. 1. – Capture des slides du cours – Exemple d'une activité avec Jetpack Compose

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`.

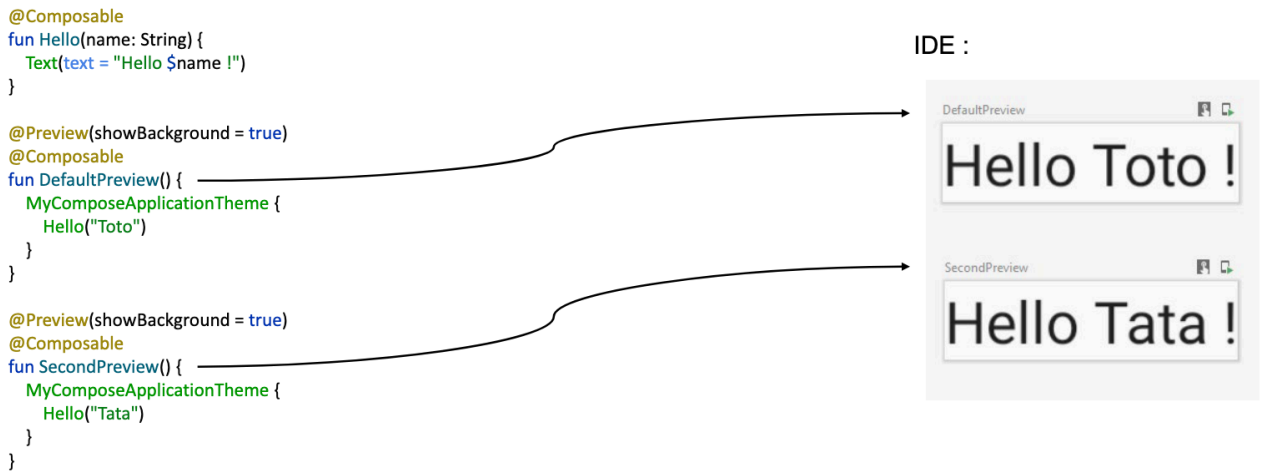

Fig. 2. – Capture des slides du cours – Exemple d'une prévisualisation avec Jetpack Compose

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`.

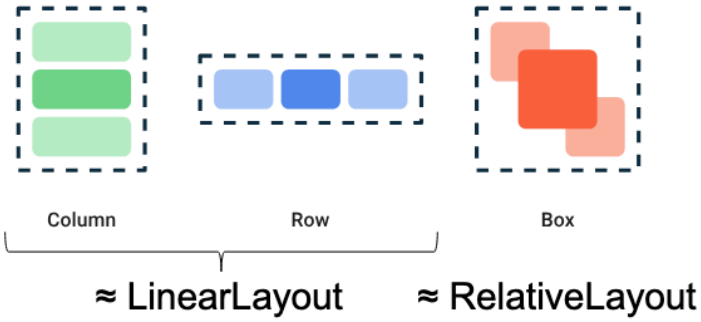

Fig. 3. – Capture des slides du cours – Exemple d'utilisation des layouts Column et Row

1.3.1 Column

Le layout `Column` organise les composants verticalement.

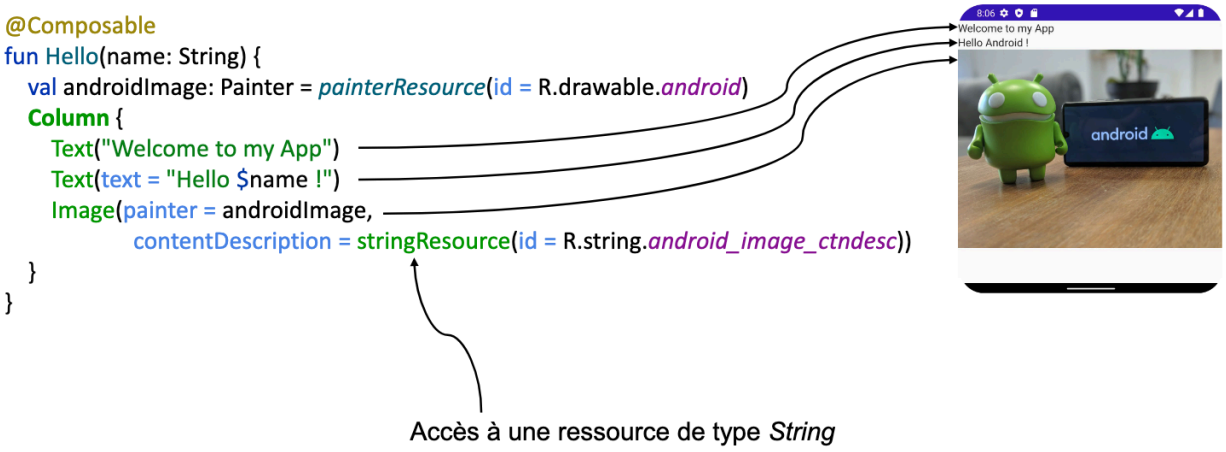

Fig. 4. – Capture des slides du cours – Exemple d'un layout Column

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")
})
```

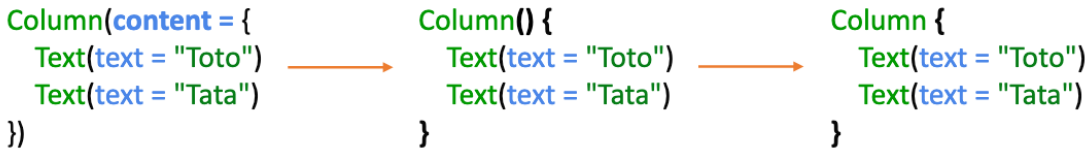

Fig. 5. – Capture des slides du cours – Exemple d'un layout Row

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 */ }
  }
}
```

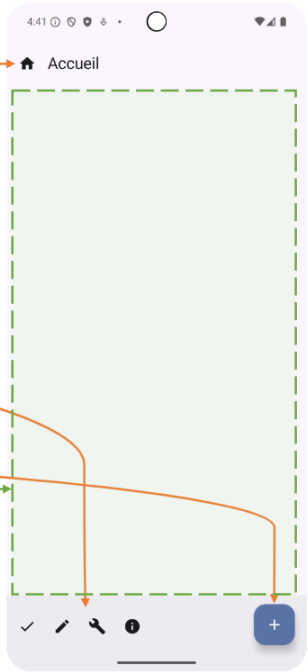

Fig. 7. – Capture des slides du cours – Exemple d'un Scaffold avec une TopAppBar et un FloatingActionButton

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.

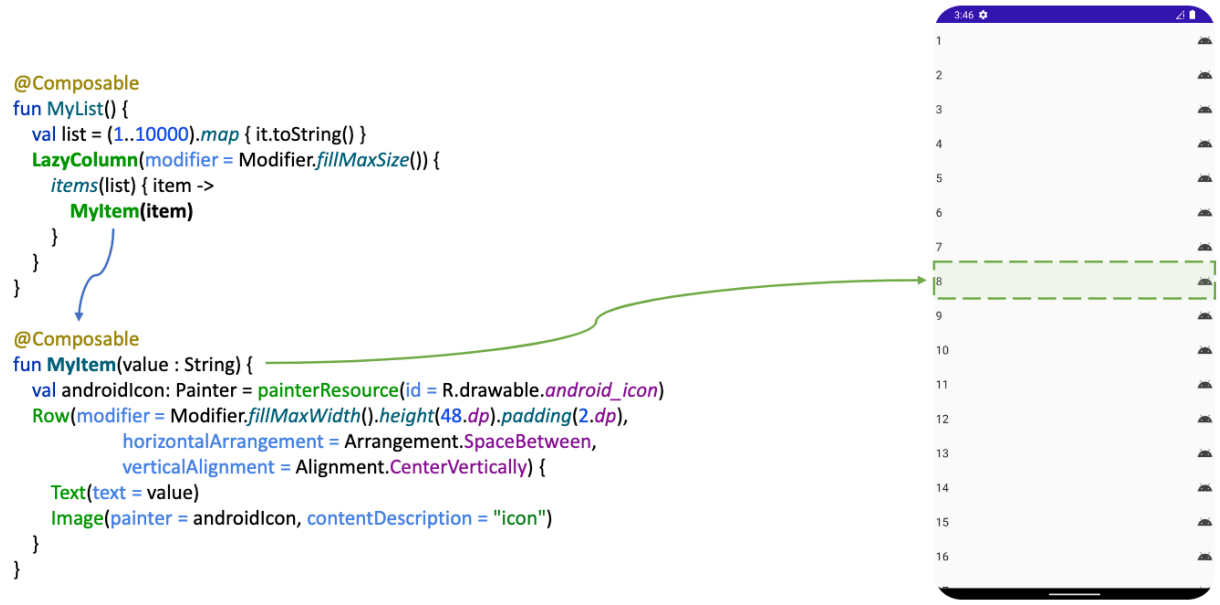

Fig. 8. – Capture des slides du cours – Exemple d'une LazyColumn

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

Pour pouvoir ajouter de l'interactivité (par exemple, un champ textuel de saisie), il est nécessaire de gérer des états dans les fonctions composables. Une interface déclarative nécessite d'appeler à novueau les fonctions composables avec de nouvelles valeurs pour mettre à jour l'UI. L'état d'une fonction composable doit être explicitement mis à jour.

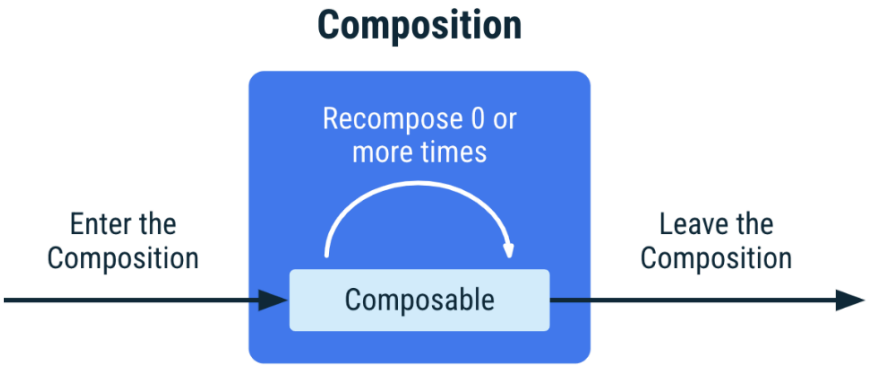

Fig. 9. – Capture des slides du cours – Gestion des états avec Jetpack Compose

2.1 State / MutableState

Jetpack Compose propose des classes pour gérer les états:

- `State<T>` : Représente une valeur immuable de type T
- `MutableState<T>` : Représente une valeur mutable de type T, qui peut être modifiée

Le mot clé `remember` permet de conserver la valeur d'un état entre les recompositions d'une fonction composable.

Toute modification d'un `MutableState` déclenche automatiquement la recomposition des fonctions composables qui l'utilisent, ce qui permet de mettre à jour l'UI en fonction des changements d'état. Si la composition n'est plus visible, les valeurs mémorisées sont libérées pour économiser de la mémoire.

> ⚠ **Warning**
>
> - `remember` permet uniquement de garder l'état entre les recompositions successives de la fonction. L'état ne survit pas à la recréation de l'Activité.
> - `rememberSaveable` permet de sauvegarder l'état même lors de la recréation de l'Activité (ex: rotation de l'écran).

2.1.1 Exemple compteur

```
@Composable
fun Counter() {
  var counter by remember { mutableStateOf(0) }

  Row(modifier = Modifier.fillMaxWidth(),
      horizontalArrangement = Arrangement.SpaceBetween,
      verticalAlignment = Alignment.CenterVertically) {

    Text("$counter")
    Button(onClick = { ++counter }) {
      Text(text = "+")
    }
  }
}
```

2.1.2 Exemple TextField

```
@Composable
fun Editor() {
  var name by remember { mutableStateOf("") }

  Column(modifier = Modifier.fillMaxWidth(),
      horizontalAlignment = Alignment.CenterHorizontally) {

    Text("Bienvenue $name !")
    TextField(value = name, onValueChange = {name = it} )
  }
}
```

2.2 ViewModel et LiveData

L'état d'une fonction composable peut être une LiveData contenue dans un ViewModel, cela permet de réaliser une architecture MVVM. Cette approche permet de séparer la logique métier de l'interface utilisateur et de gérer plus facilement le cycle de vie des données.

2.2.1 Examples

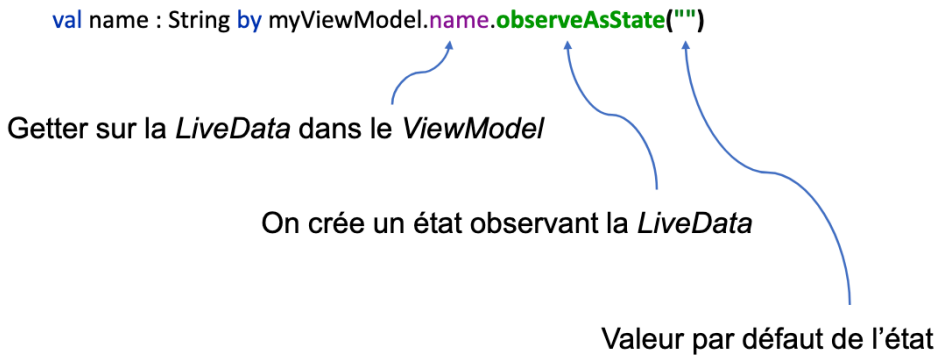

Fig. 10. – Capture des slides du cours – Référence d'un ViewModel dans une fonction composable

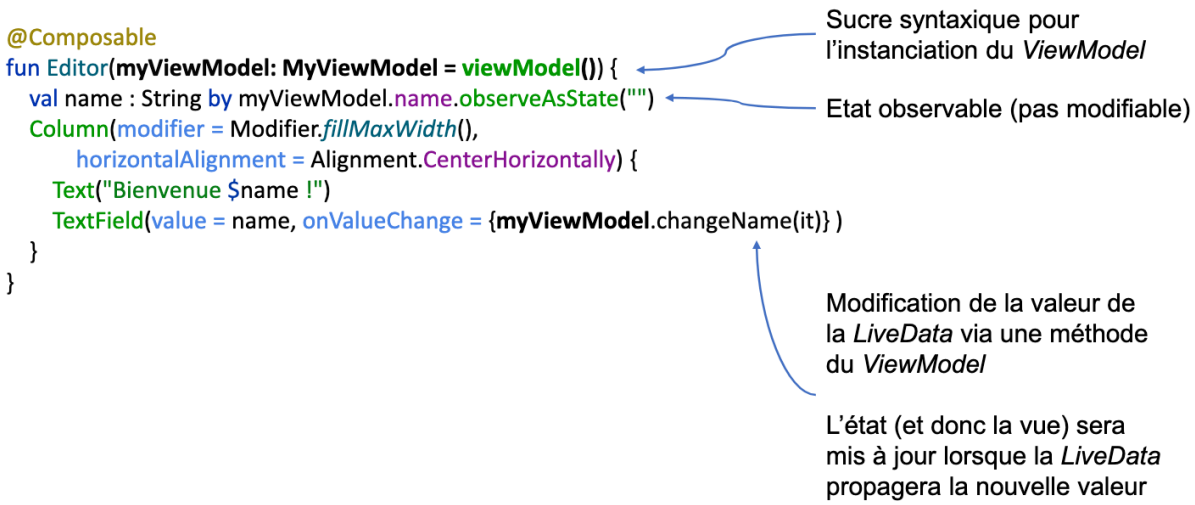

Fig. 11. – Capture des slides du cours – Exemple d'un ViewModel avec LiveData

2.2.2 Example complet

Le *ViewModel* et sa *Factory* :

```
class MyViewModel(initialName: String) : ViewModel() {
  private val _name = MutableLiveData(initialName)

  val name : LiveData<String> get() = _name

  fun changeName(newName : String) {
    _name.value = newName
  }
}

class MyViewModelFactory(private val initialName: String) : ViewModelProvider.Factory {
  override fun <T : ViewModel> create(modelClass: Class<T>): T {
    if(modelClass.isAssignableFrom(MyViewModel::class.java))
      return MyViewModel(initialName) as T
    throw IllegalArgumentException("Unknown ViewModel class")
  }
}
```

La fonction composable utilisant la *Factory* :

```
@Composable
fun Editor(myViewModel: MyViewModel = viewModel(factory = MyViewModelFactory("Toto"))) {
  val name : String by myViewModel.name.observeAsState("")
  Column(modifier = Modifier.fillMaxWidth(),
      horizontalAlignment = Alignment.CenterHorizontally) {
    Text("Bienvenue $name !")
    TextField(value = name, onValueChange = {myViewModel.changeName(it)} )
  }
}
```

Fig. 12. – Capture des slides du cours – Utilisation d'un ViewModel dans une fonction composable

Le flux d'ecécution est le suivant:

1. L'utilisateur saisit du texte dans le TextField
2. La fonction `changeName()` du ViewModel est appelée pour mettre à jour la LiveData
3. La valeur `_name.value` est modifiée, ce qui déclenche la notification des observateurs
4. `observeeAsState()` détecte le changement et met à jour la variable `name` dans la fonction composable
5. La fonction composable est recomposée avec la nouvelle valeur de `name`, mettant à jour l'UI

2.3 State hoisting

Le State hoisting est un principe de conception dans Jetpack Compose qui consiste à déplacer la gestion de l'état d'un composant vers un composant parent. Cela permet de rendre les composants plus réutilisables et testables en les rendant « stateless » (sans état).

```
@Composable //stateful
fun EditorScreen() {
  var name by remember { mutableStateOf("") }
  EditorContent(name = name, onNameChange = { name = it })
}

@Composable //stateless
fun EditorContent(name: String, onNameChange: (String) -> Unit) {
  Column(modifier = Modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally) {
    Text("Bienvenue $name !")
    TextField(value = name, onValueChange = onNameChange )
  }
}
```

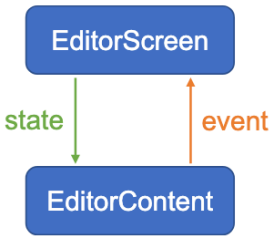

Fig. 13. – Capture des slides du cours – Exemple de State hoisting

Les éléments dont l'état a été levé possèdent certaines propriétés intéressantes:

- Single Source of Truth: L'état est géré à un seul endroit, ce qui facilite la maintenance et la compréhension du code.
- Encapsulation: Le composant enfant ne gère pas son propre état, ce qui le rend plus simple et plus réutilisable.
- Interceptable: Les événements peuvent être ignorés

- Partage: L'état peut être partagé entre plusieurs composants enfants.
- Découplage: Le composant enfant est découplé de la logique de gestion de l'état, ce qui facilite les tests unitaires.

2.3.1 Problèmes

Pour que le LiveData et Compose détecte les changements d'état, il est nécessaire de créer une nouvelle instance de l'objet d'état (ex: data class) à chaque modification car les objets sont comparés par référence.

On peut alors faire appel à la fonction `copy()` d'une data class pour créer une nouvelle instance avec les modifications souhaitées.

```
fun changePerson(name : String? = null, firstName : String? = null) {
    val p = _person.value!!.copy()  // On COPIE pour créer une NOUVELLE instance
    if(name != null)
        p.name = name
    if(firstName != null)
        p.firstName = firstName
    _person.postValue(p)  // On poste une NOUVELLE référence
}
```

Pour éviter l'asynchronisme provoquant la désynchronisation des TextFields on va devoir mettre à jour la LiveData directement avec son `value` (setter) au lieu de `postValue()`.

```
...
    _person.value = p
...
```

2.4 StateFlow

Avec Jetpack Compose, il est recommandé d'utiliser `StateFlow` à la place de `LiveData` pour gérer les états dans les ViewModels. `StateFlow` est une API de flux réactif qui offre plusieurs avantages par rapport à `LiveData`, notamment une meilleure intégration avec les coroutines Kotlin, une gestion plus fine des états et une meilleure performance.

Définir une classe avec StateFlow dans un ViewModel:

```
class PersonViewModel : ViewModel() {
  private val _person = MutableStateFlow(Person("", ""))

  val person : StateFlow<Person> = _person.asStateFlow()

  fun changePerson(name: String? = null,
                   firstName: String? = null) {

    _person.update { currentPerson ->
      currentPerson.copy(
        name = name ?: currentPerson.name,
        firstName = firstName ?: currentPerson.firstName
      )
    }
  }
}
```

2.4.1 StateFlow et Android Room

`StateFlow` peut être utilisé avec Android Room pour observer les changements de données dans une base de données locale. Room prend en charge les flux Kotlin, ce qui permet de récupérer des `Flow` directement à partir des requêtes DAO.

- On peut préciser dans le DAO vouloir le résultat sous forme de `Flow`

```
@Query("SELECT * FROM Contact")
fun getAllContacts(): Flow<List<Contact>>
```

- Le Repository remonte le Flow

```
val allContacts: Flow<List<Contact>> = contactsDao.getAllContacts()
```

- Le ViewModel devra transformer le Fow en StateFlow et le lier à son cycle de vie

```
val allContacts: StateFlow<List<Contact>> = repository.allContacts
    .stateIn(
      scope = viewModelScope,
      started = SharingStarted.WhileSubscribed(5000L),
      initialValue = emptyList<Contact>()
  )
```

- Il pourra ensuite être utilisé comme état dans Compose

```
val contacts: List<Contact> by contactViewModel.allContacts.collectAsStateWithLifecycle()
```

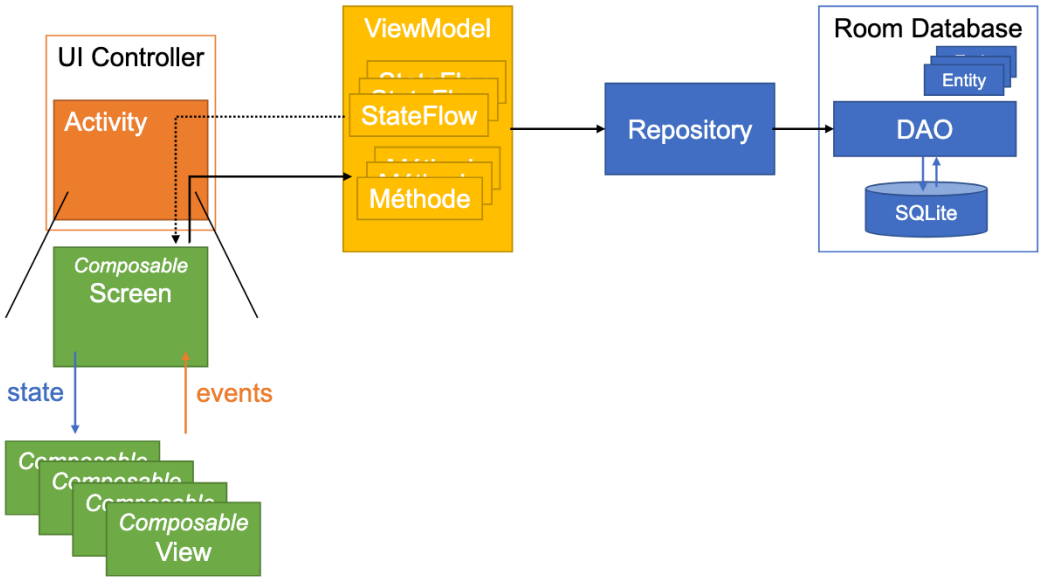

Fig. 14. – Capture des slides du cours – Utilisation de StateFlow avec Android Room dans un ViewModel

3 Layout adaptatif

Jetpack Compose propose des outils pour créer des interfaces utilisateur adaptatives qui s'ajustent automatiquement en fonction de la taille et de l'orientation de l'écran.

Il est important que notre application soit utilisable sur différents types d'appareils (téléphones, tablettes, ordinateurs de bureau) et dans différentes orientations (portrait, paysage). Pour cela, nous devons adapter la disposition de notre interface utilisateur en fonction de l'espace disponible.

3.1 Elements racines

Les fonctions racines devront s'adapter à la taille de l'écran.

Fig. 15. – Capture des slides du cours – Exemple d'une interface adaptative avec Jetpack Compose

3.1.1 BoxWithConstraints

Le composant `BoxWithConstraints` permet d'obtenir les contraintes de taille disponibles pour son contenu. Il fournit des propriétés telles que `maxWidth`, `maxHeight`, `minWidth` et `minHeight` pour adapter dynamiquement la disposition des éléments enfants en fonction de l'espace disponible.

Cela permet d'écrire du code conditionnel pour ajuster l'interface utilisateur en fonction de la taille de l'écran.

```
@Composable
fun Card(/* ... */) {
  BoxWithConstraints {
    if (maxWidth < 400.dp) {
      Column {
        Image(/* ... */)
        Title(/* ... */)
      }
    } else {
      Row {
        Column {
          Title(/* ... */)
          Description(/* ... */)
        }
        Image(/* ... */)
      }
    }
  }
}
```

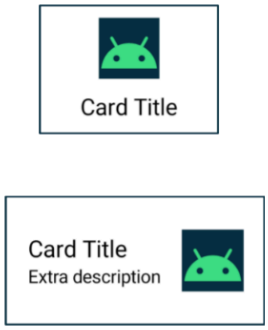

Fig. 16. – Capture des slides du cours – Exemple d'utilisation de BoxWithConstraints

4 Divers

4.1 UI Hybride

De manière générale il est pas du tout recommandé de mélanger les vues traditionnelles Android (XML) avec Jetpack Compose dans une même application. Cependant, il est possible d'intégrer des composants Compose dans une application existante en utilisant des `ComposeView` ou d'inclure des vues Android dans une interface Compose avec `AndroidView`. Souvent l'objectif est de migrer progressivement une application existante vers Compose.

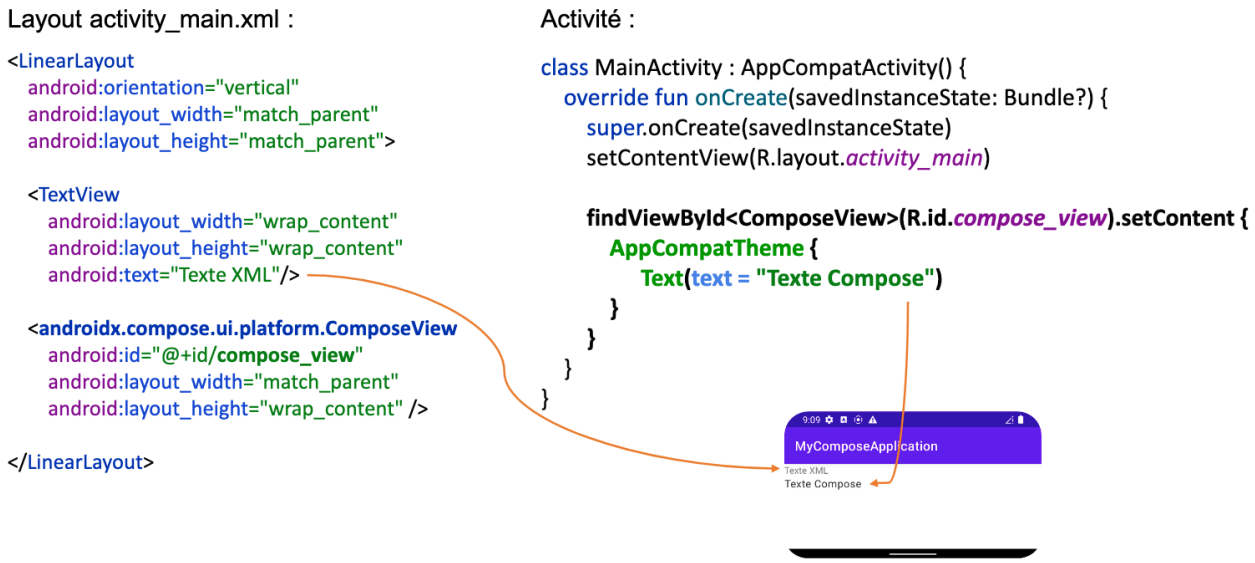

Fig. 17. – Capture des slides du cours – Intégration de Jetpack Compose dans une application Android existante