Android - Jetpack Compose

Pierre Nerzic

février-mars 2025

Cette semaine nous allons découvrir l'environnement JetPack Compose.

Il met en œuvre plusieurs concepts autour du patron MVVM = « Modèle-Vue-Vue Modèle »

- Modèle
- Vue = fonctions *composables*
- Contrôleur → *View Model*

Pierre Nerzic

Introduction

Pierre Nerzic

JetPack Compose ?

C'est la nouvelle manière (2021) de programmer des activités dans une application Android.

On retrouve des concepts de « programmation réactive » de Vue.js et React.js, voir R6.A.05 l'an prochain. Le principe est que l'interface est automatiquement mise à jour quand les données changent.

Les applications sont plus simples à mettre au point. La manière de gérer les données réduit énormément les bugs de mise à jour. On ne peut pas avoir des données incohérentes à cause, par exemple, de délais de réponse de serveurs.

Comme toujours, Google fait de nombreux changements à chaque API et il faut se former constamment.

Pierre Nerzic

Patron M-V-VM

Vous connaissez le patron MVC pour concevoir une application avec interface. Son but est de séparer les problématiques. Ce qui concerne les informations (données) est placé dans le modèle. Ce qui concerne l'affichage est placé dans la vue. Entre les deux, on a un contrôleur qui fait afficher le modèle dans la vue et qui modifie le modèle selon les actions utilisateur.

Dans le patron MVVM, le contrôleur est remplacé par un « View Model ». C'est un mécanisme de liaison individuelle entre les éléments de la vue et les données concernées, dans le modèle. Les éléments de la vue s'enregistrent en tant qu'observateurs des données, afin que leur affichage se fasse automatiquement.

Pierre Nerzic

Par quoi commencer ?

Jetpack Compose est très différent de Android Views.

- La définition d'une interface (Vue) est totalement différente.
- La définition des données (Modèle) est totalement différente.
- La programmation du contrôleur (Vue-Modèle) est différente.

Tout est nouveau, à part la notion d'activité.

On va commencer par là.

Pierre Nerzic

Interfaces avec Jetpack Compose

Pierre Nerzic

Activités

Une activité garde le même cycle de vie : appel à `onCreate` lorsqu'elle apparaît sur l'écran pour mettre en place l'interface.

L'interface est définie par `setContent` :

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            //... création de l'interface ...
        }
    }
}
```

La méthode `setContent` ne doit pas être confondue avec `setContentView`.

Pierre Nerzic

Rappel sur `setContentView`

La méthode `setContentView` d'Android Views demande un identifiant de *layout* ou une vue racine d'un *View Binding*.

- Il doit y avoir un fichier `res/layout/nom_layout.xml` définissant l'interface par des balises représentant les vues `TextView`, `Button`, `EditText`...
- `onCreate` appelle `setContentView(R.layout.nom_layout)` ou mieux, initialise un `NomLayoutBinding ui` puis appelle `setContentView(ui.root)`

Ça n'est pas du tout comme ça avec Compose.

setContent vs setContentView

La méthode `setContent` n'utilise aucun layout XML. On lui fournit un bloc d'instructions qui créent l'interface.

```
setContent {
    // ajout d'un titre
    // ajout d'une zone de saisie
    // ajout d'un bouton
    ...
}
```

La syntaxe Kotlin `methode { bloc }` est un raccourci pour `methode({ bloc })` dans le cas d'une méthode dont tous les paramètres ont des valeurs par défaut et que le dernier est une *lambda* sans paramètre.

Lambda fonctions en Kotlin

En Kotlin, une *lambda* s'écrit :

```
{ paramètres ->
    instructions ou expressions, la dernière est le résultat }
```

Exemple :

```
{ a: Int, b: Int -> a+b }
```

Quand il n'y a pas de paramètre, on écrit seulement `{ instructions }`.

Exemple :

```
{ println("bonjour") }
```

Lambda fonctions en Kotlin, suite

Une *lambda* est aussi un objet qu'on peut passer en paramètre à une autre fonction, parce que les lambda possèdent un type qu'on écrit (types des paramètres) -> type du résultat.

```
// définition
fun calcul(operation: (Int, Int) -> Int) {
    println("je suis dans calcul, je lance operation")
    val res = operation(4, 5)
    println("ça donne ${res}")
}

// appels de calcul avec des lambda
calcul({ a: Int, b: Int -> a+b })
calcul({ a: Int, b: Int -> Math.max(a,b) })
```


Pierre Nerzic

Lambda fonctions en Kotlin, suite

Pour le type d'une lambda sans paramètre, on met des parenthèses vides, et si elle n'a pas de résultat (`void` en Java), on met `Unit`. Lors de l'appel, on passe un simple bloc `{...}`.

```
// définition
fun sequence(oper1: () -> Unit, oper2: () -> Unit) {
    oper1()     // parenthèses obligatoires
    oper2()
}

// appel
sequence( {println("super")}, {println("bien")} )

// appel avec des paramètres nommés
sequence(oper2={println("bien")}, oper1={println("super")})
```


Pierre Nerzic

Lambda fonctions en Kotlin, fin

Quand le dernier paramètre d'une fonction est une lambda, on peut la sortir des parenthèses lors de l'appel.

```
// définition
fun calcul2(v1: Int, v2: Int, operation: (Int,Int)->Int) {
    println("je lance operation sur ${v1} et ${v2}")
    val res = operation(v1, v2)
    println("ça donne ${res}")
}

// appel normal
calcul2(6, 2, { a: Int, b: Int -> a+b })

// appel simplifié
calcul2(6, 2) { a: Int, b: Int -> a+b }
```


Pierre Nerzic

Retour à `setContent`

La méthode `setContent` appelée dans `onCreate` est définie comme ceci, en simplifiant au maximum :

```
public fun setContent(
    parent: CompositionContext? = null,
    content: @Composable () -> Unit
) {
    ...
}
```

Son premier paramètre, `parent` est optionnel. Il sera alors `null`.

Son dernier paramètre, `content` est un bloc d'instructions.

Il est annoté par `@Composable`. Cela signifie que son contenu doit être des appels à des fonctions annotées elles-aussi avec `@Composable`. On les appelle **fonctions composables**.

Annotation `@Composable` (javadoc)

Les fonctions composables sont les éléments fondamentaux d'une application créée avec Compose.

`@Composable` peut être appliqué à une fonction ou lambda pour indiquer qu'elle peut être utilisé dans le cadre d'une composition (création d'une interface) pour décrire une transformation des données d'application en une arborescence ou une hiérarchie.

Les fonctions composables ne peuvent être appelées qu'à partir d'une autre fonction composable.

Les fonctions composables transforment des données en éléments d'interface : texte, curseurs, etc.

Pierre Nerzic

Retour à `setContent`, suite

Ainsi, on doit définir un bloc contenant des appels à des fonctions composables et le fournir à `setContent`.

```
setContent {
    // appel à la fonction qui ajoute un titre
    // appel à la fonction qui ajoute une zone de saisie
    // appel à la fonction qui ajoute un bouton
    ...
}
```

On ne crée pas des instances de `TextView`, `Button`, etc comme avec Android Views. Ces objets permettaient de changer ensuite l'interface, grâce à leur *setters*, ex: `tvTitre.text = "nouveau"`

On ne peut pas faire cela avec une fonction composable, parce qu'elles n'ont pas de résultat qui serait un objet manipulable.

Fonctions composables et modèle de données

Les fonctions composables ne retournent rien (type du résultat `Unit`), mais elles construisent une structure en mémoire, la vue. Cette vue est **réactive**, c'est à dire qu'elle se met à jour automatiquement. Voici comment :

- Lors de leur appel, les fonctions composables sont associées à des parties du modèle de données. Par exemple, quand on appelle une fonction pour afficher une valeur présente dans les données, on passe cette donnée en paramètre.
- Quand cette donnée change, la fonction est rappelée automatiquement et modifie l'interface, parce que l'interface est abonnée par un patron *observateur*. On appelle ça la **recomposition** (réaffichage automatique).

C'est un concept qu'on retrouve dans Vue.js, voir R6.A.05.

Réactivité

Petit exemple :

```
setContent {
    Text(text = "Bonjour ${state.utilisateur.prenom}",
        style = MaterialTheme.typography.titleLarge
    )
}
```

La fonction composable `Text` est comme la fonction `printf`. Elle affiche un texte sur l'écran. La différence avec `printf`, c'est que si la valeur de `state.utilisateur.prenom` change, alors l'affichage est automatiquement refait.

On n'a pas à rappeler de *setter*. La programmation est bien plus simple. On associe les vues aux données et c'est tout.

Hiérarchie de composables

Certaines fonctions composables sont chargées d'organiser d'autres composables, comme `Column` et `Row` qui jouent le rôle des `LinearLayout`.

```
setContent {
    Column() {
        Row() {
            Text(text = state.utilisateur.prenom)
            Text(text = state.utilisateur.nom)
        }
        Button() {
            Text(text = "Ok")
        }
    }
}
```

Petit inventaire des fonctions composables

Il y a un grand nombre de fonctions disponibles.

- Il y a les composables venant de Material 3, voir ces explications et cette liste
- Il y a aussi des fonctions tierce partie, voir cette liste .

On les étudiera en TP.

Pierre Nerzic

Paramétrage des composables

La mise en page se fait par les paramètres de la fonction composable. Par exemple, la fonction `Text` possède de très nombreux paramètres : `color`, `fontSize`, `fontStyle`, `fontWeight`, `fontFamily`, `style`, etc. voir doc .

L'un des paramètres s'appelle `modifier` et est toujours présent en tant que premier paramètre optionnel dans les composables. Les paramètres optionnels sont ceux avec des valeurs par défaut. Ceux qui n'ont pas de valeur par défaut sont obligatoires.

Le paramètre `modifier` permet de changer la géométrie : taille, espacement, etc.

Paramètre `modifier`

Ce paramètre est généralement fourni par le composable parent.

```
setContent {
    Column() {
        Button(modifier = Modifier.fillMaxWidth()) {
            Text(text = "Ok")
        }
    }
}
```

`Modifier` est une interface qui possède de très nombreuses surcharges. C'est assez difficile à comprendre.

Par exemple, il y a l'import `androidx.compose.foundation.layout.fillMaxWidth` qui se trouve dans un fichier appelé `Size.kt` et qui définit l'extension `Modifier.fillMaxWidth`.

Pierre Nerzic

Paramètre `modifier`, suite

La mise en page est définie via des méthodes de `Modifier`. On peut les enchaîner :

```
setContent {
    Column() {
        Button(modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
                .background(Color.Green)) {
            Text(text = "Ok")
        }
    }
}
```

L'ordre a de l'importance, on verra cela en TP.

Actions de l'utilisateur

Les clics et autres actions utilisateur sont gérées comme avec Android Views, à l'aide d'écouteurs.

```
setContent {
    Button(
        onClick = {
            // ce qu'on veut ici
        }) {
        Text(text = "Ok")
    }
}
```

`onClick` est un *paramètre* de `Button` qui demande un bloc d'instructions. Généralement, il appelle une méthode dans le contrôleur (*ViewModel*) pour agir sur les données.

Pierre Nerzic

Modèle de données et *ViewModel*

Pierre Nerzic

Modèle de données immuable

En général, le modèle de données est défini sous la forme d'une classe qui regroupe toutes les informations. Ensuite ce modèle est instancié pour obtenir les données effectivement manipulées dans l'application, affichées dans la vue.

Les *getters* du modèle permettent d'extraire les informations voulues pour chaque vue sur l'écran.

Dans Android Views, il y a des *setters* qui sont appelés par le contrôleur pour changer les informations.

Dans Jetpack Compose, il n'y a aucun *setter* sur les données. Aucune information du modèle de données n'est modifiable, même pas par le modèle lui-même. On dit que le modèle est immuable (*immutable*). On l'appelle aussi **état** (*state*).

Modification par copie

Comment faire, alors, pour que l'application puisse évoluer en fonction des actions de l'utilisateur ?

Le principe est de **recopier toutes les données** dans une autre instance en modifiant seulement ce qu'il faut lors de la copie.

Quand la copie est finie, on remplace l'ancienne instance du modèle par la nouvelle, en une seule opération (indivisible).

La vue est abonnée aux changements d'instance du modèle. Cela déclenche une recomposition.

Les recompositions sont efficaces, car chaque composable mémorise son état et le compare au nouvel état à afficher. Rien ne se passe s'ils sont identiques.

Propriétés ACID

Ce mode de fonctionnement se situe ainsi :

- Atomicité : le changement d'état est instantané à partir du moment où le modèle a été copié/modifié.
- Cohérence : le modèle ne peut pas être bancal, constitué d'informations anciennes pas à jour mélangées avec des nouvelles. Les changements sont effectués avec la totalité des données. Les délais de transfert n'entraînent pas d'incohérence car tous les changements respectent la logique métier.
- Isolation : à tout instant, il n'y a qu'un seul état. Les modifications sont soit faites, soit en attente.
- Durabilité : l'état peut être enregistré localement ou sur un serveur.

Classe de données

Si on devait programmer un modèle de données immuable en Java, on déclarerait toutes les variables membres comme étant `final`, c'est à dire constantes. On n'aurait que des constructeurs pour affecter ces variables, tous les *getters* qu'on veut, mais aucun *setter*.

En Kotlin, il y a une syntaxe pour cela, les `data class` :

```
data class Personne(
    val prenom: String,
    val nom: String)
```

Cette syntaxe définit une classe, ainsi que son constructeur et des *getters* sur chaque propriété, mais pas de *setters*.

On peut instancier cette classe comme on veut, mais plus rien modifier aux instances une fois qu'elles sont créées

Pierre Nerzic

Classe de données, suite

Il y a une méthode supplémentaire, un constructeur par copie.

La classe définit implicitement une méthode `copy` à laquelle on passe des nouvelles valeurs, généralement nommées :

```
val p1 = User("Alfred", "Einstein")
val p1mod = p1.copy(prenom = "Albert")
```

`p1mod` est une copie de `p1` dans laquelle seule la propriété `prenom` a été changée.

Noter que les propriétés copiées sont partagées entre les deux instances. On appelle ça une copie superficielle (*shallow copy*), par opposition à une copie intégrale (*deep copy*).

Pierre Nerzic

Modèle de données, fin

Généralement, le modèle de données définit des méthodes de copie avec modification correspondant à la logique métier. C'est le modèle qui vérifie que les changements demandés sont conformes.

```
data class PersonneState(
        val prenom: String,
        val nom: String,
        val age: Int) {
    fun copyChangeAge(nouvelage: Int) {
        if (nouvelage > age) {
            return copy(age = nouvelage)
        } else {
            return this // ou exception
        }
    }
}
```


Pierre Nerzic

« Vue-Modèle »

Présentation

Dans ce patron de conception, on ne parle pas de contrôleur, parce que ce dernier a de multiples rôles que n'a pas un Vue-Modèle.

Un Vue-Modèle fait essentiellement ceci :

1. Il abonne la vue aux données, simplement en « emballant » l'état dans un patron observateur. Il y a plusieurs manières de faire cela et nous verrons la plus simple en TP.
2. Il offre des méthodes de modification de l'état par copie, celles du modèle de données ou d'autres ; ces méthodes sont appelées par les écouteurs de l'interface

C'est tout.

Pierre Nerzic

Exemple

```
class PersonneViewModel : ViewModel() {

    private val _state = mutableStateOf(PersonneState())

    var state
        get() = _state.value
        private set(newstate) {
            _state.value = newstate
        }

    fun incrementerAge() {
        state = state.copyChangeAge(state.age + 1)
    }
}
```


Pierre Nerzic

Commentaires sur l'exemple

- `_state` est une variable interne de type observateur sur une instance de l'état, ici un `PersonneState`
- `state` est un *getter* public destiné à la vue, pour afficher les informations de l'état, par exemple avec un `Text(text=viewmodel.state.nom)`, `viewmodel` étant l'instance du Vue-Modèle de l'application.
- `state` est également modifiable via le *setter*, mais celui-ci est privé. Rien d'autre que le Vue-Modèle ne peut changer l'état. Le changement se fait en une seule affectation indivisible.
- `incrementerAge` est une méthode publique qui peut être appelée par la vue, par exemple avec un `Button(onClick={viewmodel.incrementerAge()})`.

Pierre Nerzic

Et voilà

C'est fini pour cette semaine, rendez-vous en TP pour la mise en pratique.

Me faire remonter toutes les remarques : pas clair, fautes et autres...

Pierre Nerzic