DAA - Développement d'applications Android

Live Data et MVVM

16 November 2025

Table des matières

1 LiveData

LiveData est une classe observable qui permet de stocker des données de manière réactive. Elle est conçue pour être utilisée avec le cycle de vie des composants Android, ce qui signifie qu'elle respecte automatiquement le cycle de vie des activités et des fragments.

1.1 Description

Avantages

- L'interface graphique se met à jour automatiquement lorsque les données changent.
- Évite les fuites de mémoire en respectant le cycle de vie des composants.
- L'observateur est appelé dans le UI-thread → on peut modifier l'interface graphique directement.

Généralement, les LiveData sont créées dans un ViewModel et permettent de:

- Remplacer la sauvegarde de l'état lors de la re-création d'une activité.
- Partager des données et propager des événements entre plusieurs composants (fragments, activités).

> ⚠ **Warning**
>
> Les `LiveData` ne sont pas modifiables directement. Pour cela, il faut utiliser `MutableLiveData`.

1.2 Exemple d'utilisation

Création d'une LiveData

```
val data = MutableLiveData(0)
```

Accès à une valeur

```
data.value
```

Mise à jour d'une valeur

```
data.value = 42     // Synchrone, doit être appelé dans le UI-thread
// ou
data.postValue(42)  // Asynchrone, peut être appelé depuis un thread en arrière-plan
```

1.3 Observation des données

Pour observer les changements de données dans une `LiveData`, on utilise la méthode `observe`, qui prend un `LifecycleOwner` (comme une activité ou un fragment) et un observateur (une fonction lambda).

Depuis une activité

```
data.observe(this) { value ->
    textview.text = "$value" // Mettre à jour l'interface utilisateur avec la nouvelle valeur
}
```

Depuis un fragment

```
data.observe(viewLifecycleOwner) { value ->
    textview.text = "$value" // Mettre à jour l'interface utilisateur avec la nouvelle valeur
}
```

1.4 Particularités

- Le type générique peut être inféré à partir de la valeur initiale. `val data = MutableLiveData("Toto") → String`
- En cas d'une initialisation sans valeur, le type doit être spécifié. `val data = MutableLiveData<Int>()`

- Initialisation paresseuse possible avec `by lazy`.

```
val data: MutableLiveData<String> by lazy { MutableLiveData("Toto") }
```

- Les `LiveData` peuvent encapsuler des types complexes, comme des listes ou des objets personnalisés.

```
val data = MutableLiveData<List<String>>(listOf())
```

> ⚠ **Warning**
>
> Les LiveData sont implémentées en Java, l'appel au getter `value` peut retourner `null`. Utilisez l'opérateur `!!` avec précaution.

1.5 Transformations

Les LiveData peuvent être transformées automatiquement avec `map` pour créer une LiveData dérivée synchronisée :

```
data class User(val firstname: String, val name: String)

val users = MutableLiveData(listOf(
    User("Philibert", "Guillaume"),
    User("Berthe", "Crozier")
))

private val userNames: LiveData<List<String>> = users.map { usersList ->
    usersList.map { "${it.firstname} ${it.name}" }
}
```

1.6 MediatorLiveData

Permet de fusionner plusieurs LiveData et de réagir aux changements de l'une ou l'autre source :

```
private val ld1 = MutableLiveData<Int>(1)
private val ld2 = MutableLiveData<String>("0")

val ld = MediatorLiveData<Int>().apply {
    addSource(ld1) { v -> value = v }
    addSource(ld2) { v -> value = v.toInt() }
}
```

Utile pour combiner plusieurs sources d'information en une seule LiveData observable.

2 MVVM

2.1 Rappel MVC

Le modèle MVC (Model-View-Controller) divise une application en trois composants principaux:

- **Modèle (Model)** : Gère les données et la logique métier.
- **Vue (View)** : Affiche les données à l'utilisateur et gère l'interface utilisateur.
- **Contrôleur (Controller)** : Interagit avec le modèle et la vue pour traiter les entrées utilisateur et mettre à jour l'interface.

Avec Android, il n'est pas possible d'appliquer strictement le modèle MVC car les widgets ne peuvent pas directement interagir avec le modèle. Les activités et fragments se retrouvent avec trop de responsabilités :

- Gestion des vues (instanciation, mise à jour)
- Réaction aux actions utilisateur
- Gestion des API système (capteurs, Bluetooth, permissions)
- Chargement et traitement des données

Problèmes du MVC sur Android :

- Activités monolithiques difficiles à maintenir et tester
- Cycle de vie géré par le système → risque de perte de données
- Appels asynchrones complexes à gérer lors des re-créations

2.2 Architecture MVVM

MVVM (Model-View-ViewModel) sépare les responsabilités en trois couches :

- **View** : Interface utilisateur (Activités, Fragments, Views)
- **ViewModel** : Logique de présentation et gestion de l'état UI
- **Model** : Logique métier et accès aux données (Repository, Room Database)

Les composants nécessaires sont disponibles dans Android Jetpack : LiveData, ViewModel, Room.

2.3 ViewModel

Le ViewModel est une classe qui survit aux re-créations d'activité (rotation, changement de configuration) et stocke les données UI.

2.3.1 Création basique (sans paramètres)

```
class MyViewModel : ViewModel() {
    private val _counter = MutableLiveData(0)
    val counter: LiveData<Int> get() = _counter

    fun increment() {
        _counter.postValue(_counter.value!! + 1)
    }
}
```

Dans l'activité :

```
private val viewModel: MyViewModel by viewModels()

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Setup UI...

    button.setOnClickListener {
        viewModel.increment()
    }

    viewModel.counter.observe(this) { value ->
        textview.text = "$value"
    }
}
```

> ⚠ **Warning**
>
> L'appel `by viewModels()` est paresseux et échouera si l'activité est inactive. Il nécessite également un constructeur sans argument.

2.3.2 Avec Factory (paramètres requis)

Pour un ViewModel avec paramètres de constructeur :

```
class MyViewModel(defaultValue: Int) : ViewModel() {
    private val _counter = MutableLiveData(defaultValue)
    val counter: LiveData<Int> get() = _counter

    fun increment() {
        _counter.postValue(_counter.value!! + 1)
    }
}

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

Utilisation :

```
private val viewModel: MyViewModel by viewModels { MyViewModelFactory(10) }
```

2.3.3 Partage entre Fragments

Un même ViewModel peut être partagé entre une activité et ses fragments via `activityViewModels()` :

```
// Dans le Fragment
private val sharedViewModel: MyViewModel by activityViewModels()

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    sharedViewModel.counter.observe(viewLifecycleOwner) { value ->
        textview.text = "$value"
    }
}
```

> ⚠ **Warning**
>
> Dans les fragments, toujours utiliser `viewLifecycleOwner` pour observer les LiveData.

Avec Factory :

```
private val sharedViewModel: MyViewModel by activityViewModels { MyViewModelFactory(10) }
```

> ⚠ **Warning**
>
> Si plusieurs composants utilisent des Factory avec des paramètres différents, on ne peut pas garantir laquelle sera utilisée. Une seule instance du ViewModel sera créée.

2.3.4 AndroidViewModel

Pour accéder au contexte Application (utile pour SharedPreferences, resources) :

```kotlin
class MyViewModel(application: Application) : AndroidViewModel(application) {
    private val prefs = application.getSharedPreferences("myPrefs", Context.MODE_PRIVATE)

    // Utilisation de getApplication() pour accéder au contexte
}
```

> ⚠ **Warning**
>
> `AndroidViewModel` donne accès au contexte **Application** uniquement, pas au contexte d'activité ou de fragment.

Si le seul paramètre est `Application`, pas besoin de Factory :

```kotlin
private val viewModel: MyViewModel by viewModels()
```

2.4 Bonnes pratiques

> ⚠ **Warning**
>
> **À ne JAMAIS faire** :
> - Référencer une View, Activity, Fragment ou Context d'activité dans un ViewModel
> - Exposer directement des `MutableLiveData` publiques

Recommandations :

- Exposer uniquement des `LiveData` (lecture seule), garder `MutableLiveData` privées :

```kotlin
private val _counter = MutableLiveData(0)
val counter: LiveData<Int> get() = _counter
```

- Utiliser plusieurs ViewModels si nécessaire pour séparer les responsabilités
- Les ViewModels ne sont **pas** pour la persistance à long terme → utiliser Room Database
- Le ViewModel est détruit uniquement quand l'activité est terminée définitivement (pas lors des re-créations)

Cycle de vie : Un ViewModel survit aux re-créations d'activité dues aux changements de configuration (rotation, etc.) mais est détruit via `onCleared()` quand l'activité est définitivement terminée.

2.5 Dépendances

```
implementation("androidx.lifecycle:lifecycle-livedata-ktx:x.y.z")
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:x.y.z")
implementation("androidx.activity:activity-ktx:x.y.z")
implementation("androidx.fragment:fragment-ktx:x.y.z")
```