Développement d'application mobile natif

Chapitre5: Architecture Android Moderne: Fragments, Navigation & MVVM

1

Hend Ben Ayed Kharrat & Marwa Chaabani

AU: 2025-2026

L3DSI

Ordre du Jour

- Architecture MVVM
- Notions de Fragments
- Navigation & Menu Drawer
- Le View Model

Architecture MVVM

ÉVOLUTION DES ARCHITECTURES ANDROID

- Historiquement, les premières applications Android reposaient sur une architecture simple où les **Activity** géraient toutes les responsabilités :
- affichage,
- logique métier,
- et gestion des données.

Cette approche « tout dans l'Activity » posait plusieurs problèmes :

- forte dépendance entre logique et interface,
- faible testabilité,
- maintenance complexe,
- erreurs lors de la rotation ou de la navigation.

MVC – MVP - MVVM

👉 Pour améliorer cela, Android a évolué vers des architectures plus claires: **MVC** → **MVP** → **MVVM**, favorisant la **séparation des responsabilités**.

- **MVC** est le modèle le plus simple, mais les `Activity/Fragment` (Contrôleurs) seront trop volumineux, rendant le code difficile à tester.
- **MVP** et **MVVM** résolvent cela en introduisant une couche intermédiaire isolée (Présentateur ou ViewModel) :
  - MVP utilise la **commande active** (le Présentateur *dit* à la Vue quoi faire),
  - **MVVM** utilise l'**observation réactive** (la Vue *écoute* le ViewModel), ce qui en fait le modèle **recommandé** par Google.

ARCHITECTURE ANDROID MODERNE

- **La Solution MVVM :** Séparer les responsabilités en trois couches claires :
  - **Modèle (Model) :** Les données brutes. Source de données (DataClass Room, API, fichiers, etc.)
  - **Vue (View) :** Responsables uniquement de l'affichage de l'UI.
  - **Modèle de Vue (ViewModel) :** Responsable de la logique métier et de la gestion de l'état (données).
- Avantages:

Séparation des préoccupations

Facilite la gestion de la complexité et l'évolution du code Android en distinguant logique UI, métier et données.

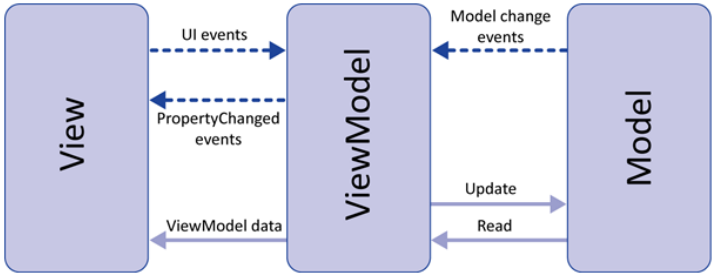
Testabilité accrue

Le ViewModel peut être testé indépendamment de la Vue, simplifiant les tests unitaires et la validation de la logique.

Meilleure collaboration

Les concepteurs UI/UX travaillent sur la Vue (Fragments/Activities) pendant que les développeurs se concentrent sur le ViewModel et le Modèle.

LES TROIS COMPOSANTS DU MVVM



1. Model (Modèle):

Contenu : Il inclut les classes de données pures, les classes de base de données (Room), les appels réseau (Retrofit), et les Repositories (qui coordonnent les sources de données).
Flux : Il ne communique qu'avec le ViewModel, l'informant lorsque les données ont changé.

2. View(s) (Vue(s)):

Contenu : Elle contient le code qui gère l'affichage, les animations et la collecte des entrées utilisateur. La View est **"stupide"** ; elle ne contient aucune logique métier.
Flux : Elle envoie les **"User actions"** (clics, saisies) au ViewModel. En retour, elle **"Observe UI states"** (observe l'état de l'interface) exposé par le ViewModel pour mettre à jour l'affichage.

3. ViewModel (Modèle-Vue):

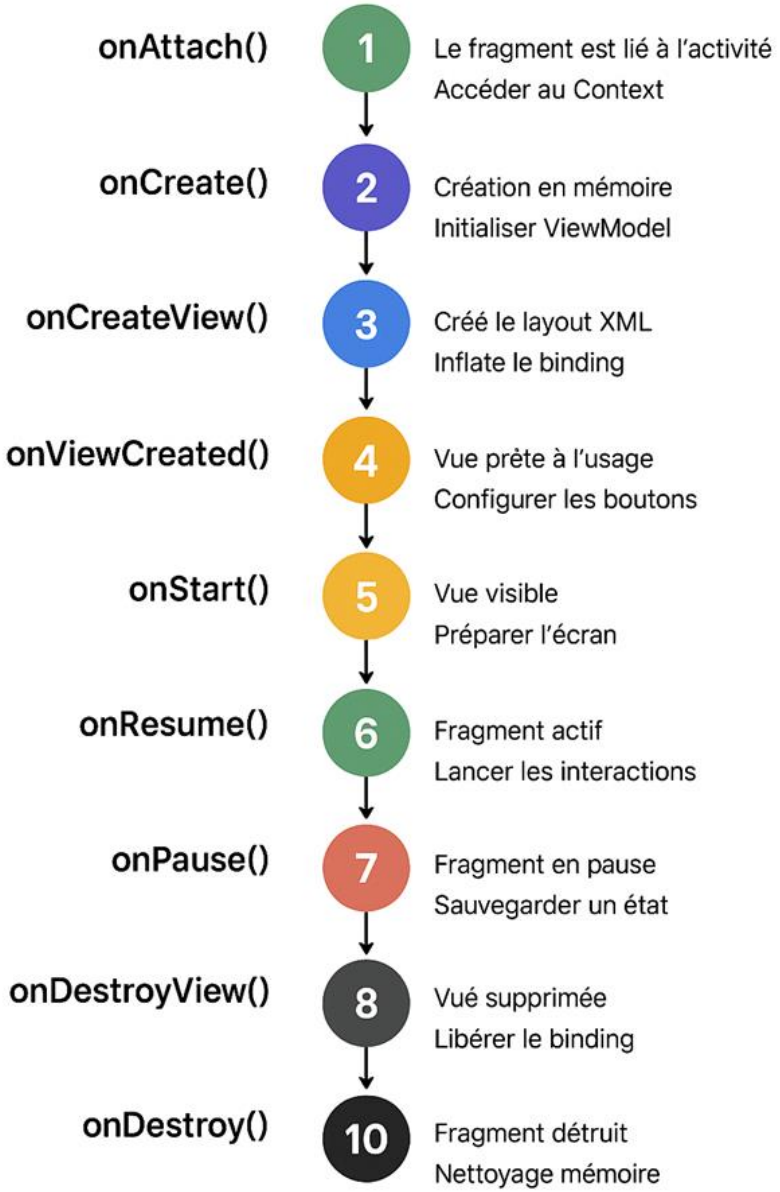
Contenu : Il détient les données nécessaires à l'interface utilisateur . Il contient la logique de présentation (comment les données doivent être formatées pour l'UI).
Flux : Il reçoit les actions de l'utilisateur de la View. Il demande ensuite au Model de mettre à jour ou de fournir des données ("Update Model"). Il expose ensuite ces données de manière réactive à la View.

Notion de fragments

LE FRAGMENT : LE PILIER DE LA VUE (VIEW)

- **Le Fragment : Un Module d'Interface Utilisateur**
- **Définition :** Un Fragment est un composant modulaire de l'interface utilisateur. Il représente une portion du comportement ou de l'UI au sein d'une activité.
- **Rôle :**
  - **Afficher l'UI :** Il gonfle (inflate) son propre layout XML.
  - **Gérer les Interactions :** Il capture les clics et autres événements.
  - **Déléguer :** Il ne fait **AUCUNE** logique métier ou de gestion de données ; il délègue ces tâches au **ViewModel**.
- **Cycle de Vie :** Il possède son propre cycle de vie (onCreateView, onViewCreated, etc.) qui est géré par l'Activité hôte.
- **ViewBinding :** Utilisation de _binding et binding!! pour accéder aux vues en toute sécurité.

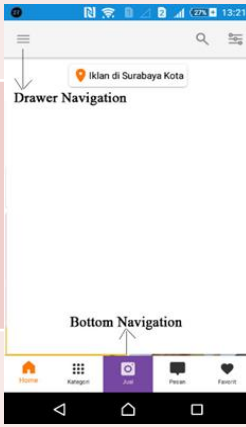
Cycle de vie d'un Fragment

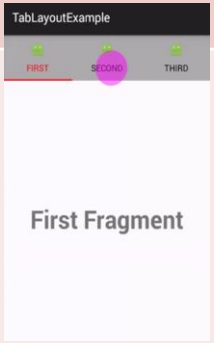


Notion de Navigation

MÉTHODES DE NAVIGATION ANDROID

Composant Android	Usage Principal	Avantages
DrawerLayout et NavigationView	Navigation entre les destinations principales ou les paramètres.	Idéal pour de nombreuses destinations (6+) sans surcharger l'écran principal.
BottomNavigation View	Navigation rapide entre les 3 à 5 destinations les plus fréquentes.	Visibilité et accessibilité immédiate
TabLayout et ViewPager	Organisation des sous-sections (vues sœurs) d'un même écran principal.	Permet de glisser horizontalement entre des contenus connexes sans quitter l'écran.





Toutes ces méthodes sont conçues pour fonctionner ensemble et sont gérées par le **Composant Navigation** qui assure la logique des transitions.

LE MENU DRAWER ET LA TOOLBAR: (DRAWERLAYOUT)

- Le Menu Latéral (`DrawerLayout`) et la Toolbar
- **`DrawerLayout` :** Le conteneur principal de `MainActivity.xml` qui permet de faire glisser le menu.
- **`NavigationView` :** Le contenu réel du menu latéral (Liste des liens : Accueil, Ajout, Liste).
- **Liaison (dans `MainActivity.kt`) :**
  - **`AppBarConfiguration` :** Définit quels Fragments sont des destinations de "premier niveau" (qui affichent le bouton **hamburger** au lieu de la flèche de retour).
  - **Liaison de la Toolbar :**

```
setupActionBarWithNavController(navController, appBarConfiguration)
```

  - **Liaison du Menu :** `navView.setupWithNavController(navController)` (Le clic sur un item du menu latéral navigue automatiquement vers le Fragment correspondant).

DRAWERLAYOUT: FICHIERS CLÉS

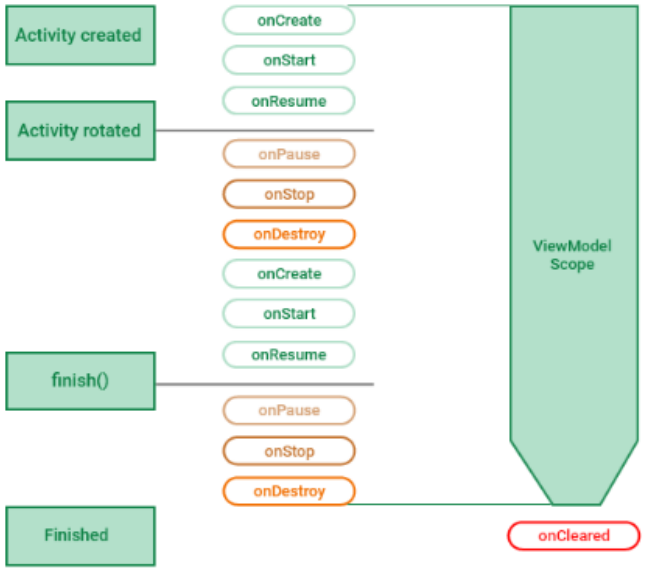
- **Fichiers Clés :**
  - Le layout principal (activity_main.xml) contient le DrawerLayout.
  - NavigationView pour le contenu du **menu** latéral.

- **Élément Clé :** Le NavHostFragment dans MainActivity est l'espace réservé où les différents Fragments s'affichent.

- **Navigation Graph (mobile_navigation.xml) :** Gérer les transitions entre les Fragments de manière simple et sécurisée:
  - Les **Destinations** (vos Fragments).
  - Les **Actions** (les flèches de transition entre les destinations).
  - L'identifiant des éléments de menu doit correspondre à l'ID des Destinations dans le **Navigation Graph** pour la navigation automatique.

ViewModel partagée et LiveData

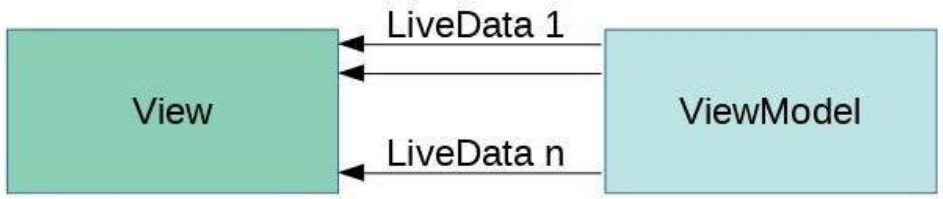
LE VIEWMODEL : LE CERVEAU QUI SURVIENT AUX ÉVÉNEMENTS



- **Rôle Principal :** Conserver et gérer les données liées à l'UI d'une manière insensible aux changements de configuration (comme la rotation de l'écran ou le passage en arrière-plan). Il est la "**mémoire de la maison**".
- **Gestion de l'État :** Quand l'écran tourne, le Fragment est détruit puis recréé, mais le **ViewModel survit** et conserve l'état (e.g., la liste d'étudiants n'est pas rechargée).
- **Partage (activityViewModels) :** En initialisant le ViewModel avec activityViewModels() dans les Fragments (Ajout et Liste, ou autre), ceux-ci accèdent à la **même instance** de données, permettant la communication centralisée. Il devient la "**source unique de vérité**".

LIVEDATA : CANAL DE COMMUNICATION OBSERVER PATTERN

- **LiveData : Le Conteneur de Données Intelligent** qui est **observable** et **conscient du cycle de vie**.
- Pour chaque état de la vue, nous avons un objet LiveData dans le ViewModel pour le propager et le mettre en cache
- **Encapsulation (Sécurité) :**
  - **Interne :** Utilisez **private val _data = MutableLiveData<T>()** pour permettre la modification *uniquement* dans le ViewModel. Et indiquant que la donnée est observable
  - **Externe :** Exposez la donnée en lecture seule : **val data: LiveData<T>= _data.**



EXEMPLE

```
class EtudiantViewModel : ViewModel() {
 // Utilisation de MutableLiveData pour que la liste soit observable et modifiable
 //_etudiants (privé) est modifiable uniquement dans le ViewModel.
3 Usages
private val _etudiants = MutableLiveData<MutableList<Etudiant>>()
// etudiants (public) est en lecture seule pour les Fragments.
2 Usages
val etudiants: LiveData<MutableList<Etudiant>> = _etudiants
2 Usages
private val _etudiantSelectionne = MutableLiveData<Etudiant?>()
1 Usage
val etudiantSelectionne: LiveData<Etudiant?> = _etudiantSelectionne
```

Cela empêche un Fragment de faire _etudiants.value?.add(...) directement --> ce qui protégerait la “source unique de vérité”.

Les Fragments observent les données, mais ne les modifient jamais directement.
Seul le ViewModel a le droit de changer l’état via ses méthodes (addEtudiant(), removeEtudiant()...).

MISE À JOUR DES DONNÉES VIA LE **VIEWMODEL**

- Le processus de mise à jour des données via le **ViewModel** à **toute action utilisateur qui modifie la liste** (ajout, suppression, modification), il faut standardiser le cycle de vie de la donnée.

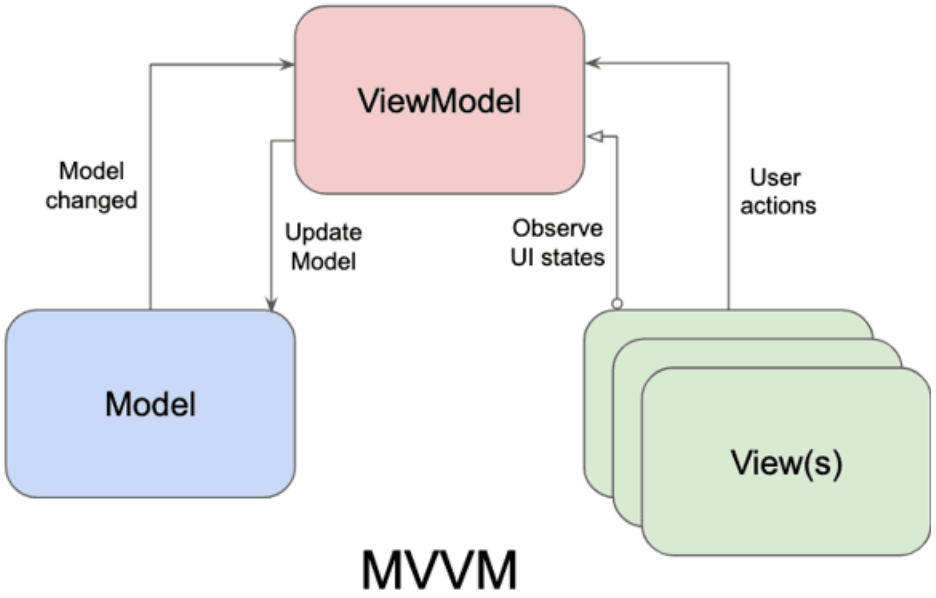
Ce cycle se déroule toujours en trois étapes :

- **Capturer et Transmettre (Fragment d'Action).**
- **Modifier et Notifier (ViewModel).**
- **Observer et Afficher (Fragment de Liste, Detail).**

Flux d'interaction dans architecture MVVM

Cycle de vie de la donnée

FLUX D'INTERACTION DANS L'ARCHITECTURE MVVM

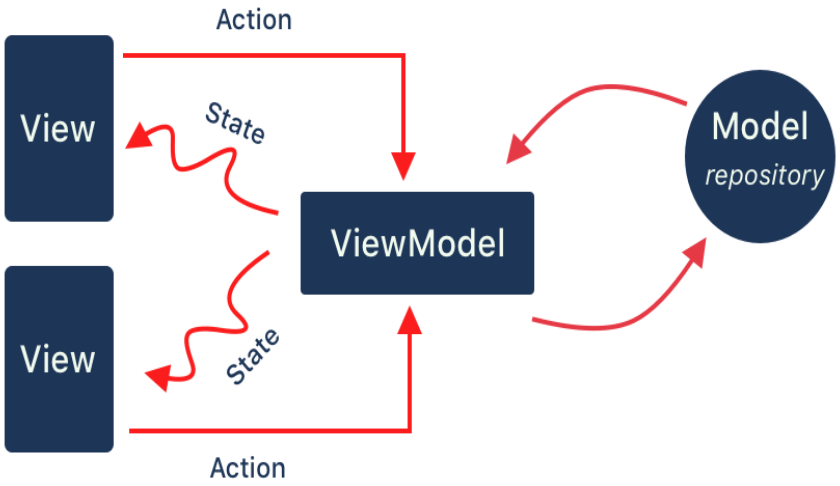


L'interaction en MVVM se divise en deux flux cruciaux:

- le flux de MAJ de données (du Model à la Vue)
- le flux d'actions/commandes (de la Vue au Model, via le ViewModel).

- **L'utilisateur agit** (clic sur "VALIDER..."). La **View** envoie cette action au **ViewModel**.
- Le **ViewModel** reçoit l'action et demande au **Model** de valider la réponse ("Update Model").
- Le **Model** effectue la validation et notifie le **ViewModel** du résultat ("Model changed").
- Le **ViewModel** met à jour un objet **LiveData** (par exemple, ToastMessage ou NavigationEvent).
- La **View** qui observe ce LiveData détecte le changement ("Observe UI states") et exécute l'action appropriée (affiche le Toast ou lance l'Intent).

COMPRENDRE LES DEUX FLUX D'INTERACTION



1 Flux de Mise à Jour (Données)

Quand les données du **Model** changent (ex: base de données, API), le Model notifie le **ViewModel**.

Le ViewModel met à jour ses **LiveData**, que la **Vue** (Fragment/Activity) observe pour se rafraîchir automatiquement et instantanément, assurant une cohérence visuelle.

2 Flux d'Action (Commandes)

Une action utilisateur dans la **Vue** (ex: clic, saisie) est capturée par le **ViewModel**.

Le ViewModel traite l'interaction, exécute la logique et peut transmettre une demande au **Model**.

MISE EN ŒUVRE PRATIQUE : SYNTHÈSE POUR EXEMPLE D'APPLICATION

Étape	Fichier / Composant	Rôle dans l'Architecture
1. Modèle	data class Etudiant	Structure de la donnée.
2. Logique/État	EtudiantViewModel	Contient etudiants: LiveData<MutableList> et etudiantSelectionne: LiveData<Etudiant?>. Logique ajouterEtudiant() et selectionnerEtudiant(), ...
3. Vue A (Liste)	ListeEtudiantFragment	Observe la liste complète (etudiants). Utilise activityViewModels(). Déclenche la sélection et la navigation.
4. Vue B (Détail)	DetailEtudiantFragment	Observe l'étudiant sélectionné (etudiantSelectionne). Utilise activityViewModels(). Met à jour l'UI (et le titre) en fonction.
5. Vue C (Ajout)	AjoutEtudiantFragment	Capturer les évenement click. Utilise activityViewModels().
5. Navigation	mobile_navigation.xml	Définit le chemin **action_liste_to_detail, action_ajout_to_liste** et gère le passage d'un écran à l'autre. findNavController().navigate(R.id.action_liste_to_detail)

LE CYCLE RÉACTIF : LE POINT CENTRAL DU MVVM

Problème : Comment le ListeEtudiantFragment et le DetailEtudiantFragment peuvent-ils accéder au même objet étudiant ?

Solution : Les deux fragments demandent la même instance de EtudiantViewModel, mais l'attachent au cycle de vie de l'**Activité Hôte** (et non à leur propre cycle de vie).

Le cycle ci-dessous explique comment le **ViewModel** gère l'état pendant que le **Fragment** se met à jour, et pourquoi les données ne sont jamais perdues.

Etape 1.Démarrage : Instanciation (Via le Délégué)

Étape	Action	Rôle
Fragment	Déclaration de l'accès au ViewModel : val vm by activityViewModels().	Demande une instance du ViewModel liée à l'Activity.
ViewModel	L'instance du ViewModel est créée une seule fois et conserve l'état interne (_etudiants).	

2. Abonnement (Observation des Données)

Action	Localisation	Explication
Abonnement	ListeEtudiantFragment (onViewCreated)	Le Fragment s'abonne à la source de vérité : viewModel.etudiants.observe(**viewLifecycleOwner) { … }.**
Rôle de l'Observer	viewLifecycleOwner	Garantit que le Fragment ne reçoit des mises à jour que lorsqu'il est actif. L'observation est automatiquement stoppée lorsque le Fragment est mis en pause ou détruit.
1er Affichage	LiveData	Immédiatement après l'abonnement, le LiveData envoie la valeur actuelle (même si elle est nulle ou vide) pour initialiser l'UI

3. Interaction (Modification de l'État)

Action	Localisation	Explication
Entrée Utilisateur	AjoutEtudiantFragment (au clic sur Ajouter)	Le Fragment capture les données du formulaire et appelle la méthode du ViewModel : viewModel.addEtudiant(nouveauEtudiant).
Logique Métier	EtudiantViewModel	La méthode addEtudiant met à jour la donnée interne : _etudiants**.value** = nouvelleListe.
Rôle du **.value**	MutableLiveData	L'affectation de cette propriété à LiveData.value est le signal qui déclenche la notification à tous les Fragments abonnés "actif" (started ou resumed).

Règle : Le Fragment ne modifie **jamais** la LiveData directement ; il délègue l'action à une fonction dédiée dans le ViewModel. (addEtudiant, removeEtudiant, selectionnerEtudiant...)

4. MISE À JOUR RÉACTIVE (COMMUNICATION)

LE FRAGMENT D'AFFICHAGE (OBSERVE ET MET À JOUR L'UI)

Le Fragment de Liste reste **passif** et réagit uniquement aux signaux du ViewModel. Son code ne change pas, quelle que soit la modification effectuée.

Rôle	LiveData Observée	Mécanisme	Description
Abonnement	etudiants	etudiantViewModel.etudiants.observe(**viewLifecycleOwner**) { nouvelleListe -> ... }	Cette ligne capture le signal envoyé par le ViewModel après toute action utilisateur (ajout, suppression, modif).
Mise à Jour UI	etudiants	etudiantAdapter.updateList(nouvelleListe)	Le Fragment passe la nouvelle liste complète à l'Adapter. L'Adapter se charge de mettre à jour la RecyclerView via notifyDataSetChanged().
Affichage Détail	etudiantSelectionne	etudiantViewModel.etudiantSelectionne.observe(...) { etudiant -> /* Afficher les détails de l'étudiant dans le Fragment de Détail */ }	Le Fragment de Détail n'a pas besoin de connaître le Fragment de Liste ni l'événement de clic. Il se contente de demander au ViewModel : "Donne-moi l'état de sélection actuel, et notifie-moi s'il change."

Les **LiveData** sont observés via **viewLifecycleOwner**, garantissant un cycle de vie sûr.

CONCLUSION: MAÎTRISER L'ARCHITECTURE MVVM

- **Ce que vous avez appris :** La séparation claire des responsabilités entre **Fragment (Vue)**, **ViewModel (État)** et **Navigation (Flux)**.
- **Principes Clés à Retenir :**
  - Ne jamais mettre de logique métier directement dans un Fragment.
  - Utiliser LiveData pour les données réactives et sûres.
  - Utiliser activityViewModels() pour le partage de données.
  - La navigation doit être gérée par le NavController.
- Dans une architecture MVVM, les fragments ne se parlent jamais directement: c'est le viewModel via le LiveData qui fait circuler les donnees de manière propre, reactive decouplée.

REFERENCES

Lesson6: App Navigation

Lesson 7: Activity and Fragment Lifecycles

Lesson 8: App architecture (UI layer)