Développement Android avec Kotlin

Cours - 08 - Architecture d'une application Android

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com

Expériences utilisateur dans les applications Android

- Une application Android contient plusieurs **composants** :

  Activités, **Fragments**, **Services**, **Fournisseurs de contenu**, et **Broadcast Receivers**.

  Ces composants sont déclarés dans le fichier **manifeste** et intégrés à l'expérience utilisateur par l'OS Android.

- **Environnement mobile** :

  Les applications doivent s'adapter à des workflows variés et à des tâches souvent interrompues.

  Les ressources sont limitées, et le système peut arrêter des processus pour en libérer d'autres.

- **Règles clés :**

  Les composants peuvent être lancés individuellement ou dans le désordre.

  Ils peuvent être détruits à tout moment par le système ou l'utilisateur.

=> On ne stock pas d'état ou de données directement dans ses composants de l'application

Principes architecturaux courants

- **Problématique** :

  Si les **composants d'application** ne peuvent pas stocker les données ou l'état de l'application, comment concevoir une application robuste et scalable ?

- **Pourquoi une architecture ?**

  Les applications Android deviennent de plus en plus complexes.

  - **Une bonne architecture permet :**
    - **Évolutivité** : Ajouter de nouvelles fonctionnalités facilement.
    - **Robustesse** : Réduire les bugs et gérer les interruptions du système.
    - **Testabilité** : Faciliter les tests unitaires et fonctionnels.

Une architecture d'application définit les limites entre les parties de l'application et les responsabilités de chaque partie.

Principes architecturaux courants

Séparation des préoccupations (Separation of concerns)

- **Principe clé**
  - Évitez de regrouper tout votre code dans une **Activity** ou un **Fragment**.
- **Pourquoi ?**
  - Ces classes doivent uniquement gérer les interactions entre l'UI et l'OS
  - Leur cycle de vie est contrôlé par l'OS, qui peut les détruire à tout moment.
- **Bonnes pratiques ?**
  - **Allégez** ces classes pour éviter les problèmes liés au cycle de vie.
  - Traitez les **Activity** et **Fragment** comme de la "colle" entre l'OS et votre application.
- **Avantages** :
  - **Maintenance facilitée** et code plus robuste.
  - **Testabilité améliorée** grâce à un découplage clair.

Principes architecturaux courants

Contrôle de l'UI à partir de modèles de données

- **Principe :**
  - L'UI doit être contrôlée par des **modèles de données**, idéalement **persistants**.

- **Qu'est-ce qu'un modèle de données ?**
  - Représente les données de l'application.
  - **Indépendant** de l'UI et des autres composants.
  - Non affecté par le cycle de vie des composants de l'application.

- **Pourquoi des modèles persistants ?**
  - **Protection des données** : Pas de perte si l'OS détruit l'application pour libérer des ressources.
  - **Résilience** : Fonctionne même avec une connexion réseau irrégulière ou inexistante.

- **Avantages** :
  - Améliore la **testabilité** et la **robustesse** de l'application.

Principes architecturaux courants

Single Source of Truth (Référence unique)

- **Principe :**
  - Chaque type de données doit avoir une **source unique de vérité (SSOT)** qui :
    - Est **propriétaire** des données.
    - Expose les données sous une forme **immutable**.
    - Permet les modifications via des fonctions ou événements contrôlés.

- **Avantages** :
  - **Centralisation** des modifications pour un type de données.
  - **Protection** des données contre les accès non contrôlés.
  - Facilite le **suivi des modifications** et la détection des bugs.

- **Exemples de SSOT :**
  - **Base de données**
  - **ViewModel** ou **l'interface utilisateur** dans des cas simples
  - Le **Repository** (gère les données locales et distantes)

Principes architecturaux courants

Flux de données unidirectionnel (UDF)

- **Principe :**
  - L'état ou les données circulent dans **une seule direction**, tandis que les événements de modification des données vont dans la direction opposée.

- **Flux typique sur Android** :
  - **Données** : Transmises des **SSOT** vers l'UI.
  - **Événements utilisateur** : Exemples, clics sur un bouton, transmis depuis l'UI vers la **SSOT** pour modifier les données.

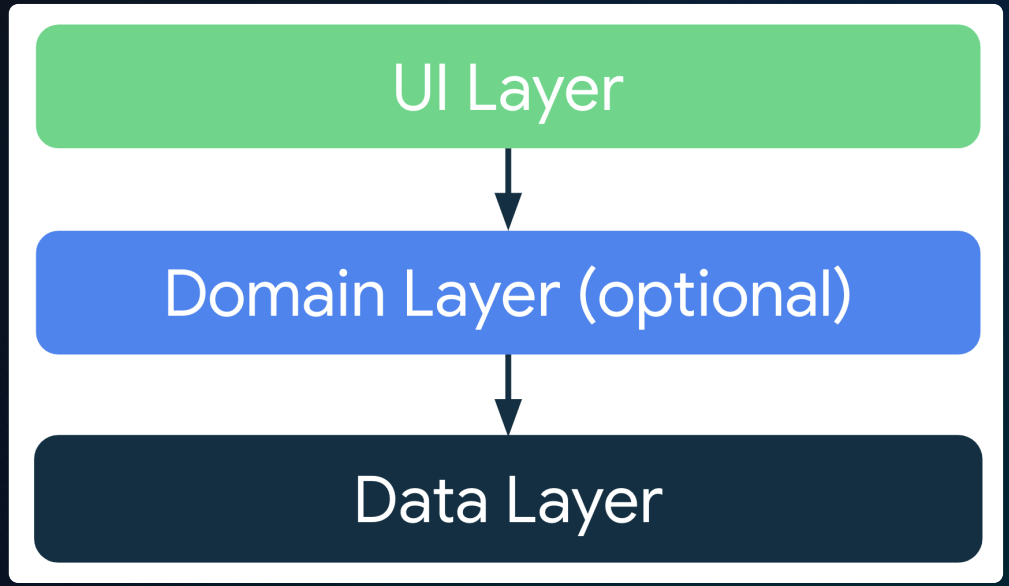
- **Avantages** :
  - Garantit une **cohérence des données**.
  - Moins de **risques d'erreurs**.
  - Plus facile à **déboguer**.

Architecture d'application recommandée

Chaque application doit avoir au moins **deux couches** :

1. **Couche d'interface utilisateur (UI)** : Affiche les données à l'écran.
2. **Couche de données** : Contient la logique métier et expose les données.

Une **couche de domaine** peut être ajoutée pour simplifier et réutiliser les interactions entre l'UI et les couches de données

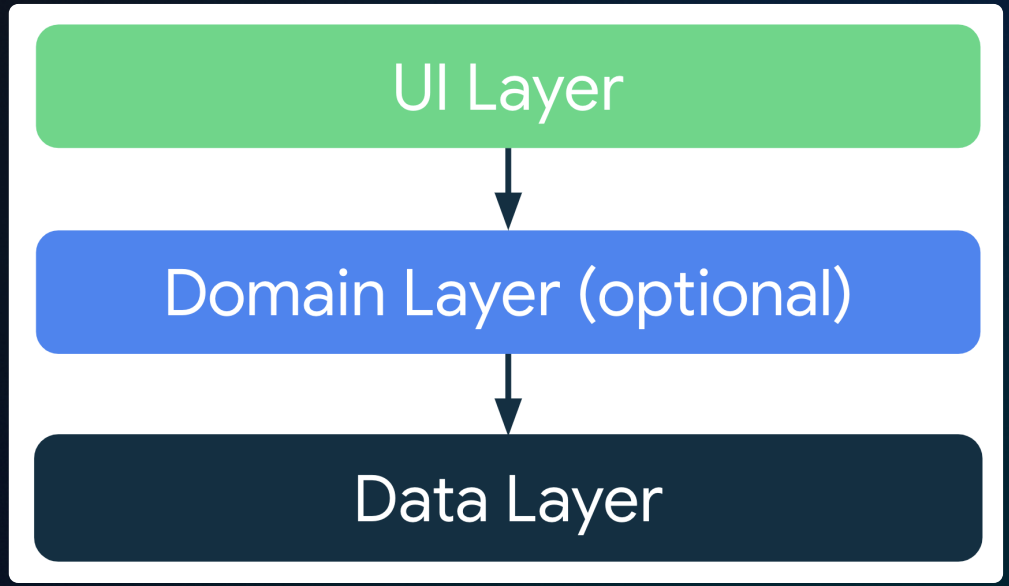


⚠️ **Remarque** : Ces recommandations sont des **bonnes pratiques**, mais elles ne sont pas des consignes strictes. Elles doivent être **adaptées** en fonction des besoins spécifiques de l'application.

Architecture d'application recommandée

Cette architecture encourage l'utilisation des techniques suivantes :

- **Architecture réactive et multicouche**
- **Flux de données unidirectionnel** dans toutes les couches de l'application
- **Couche d'UI** avec des **conteneurs d'état** pour gérer la complexité de l'UI
- **Coroutines et flux** pour la gestion des tâches asynchrones
- **Bonnes pratiques** pour l'**injection de dépendances**

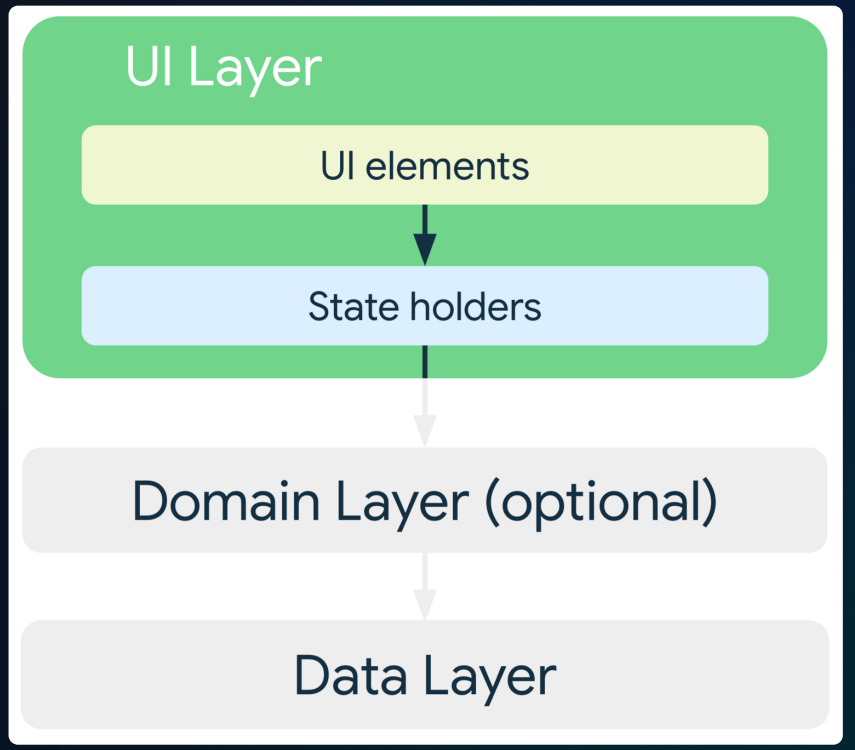


Couche de l'interface utilisateur

Son rôle est d'afficher les données de l'application et de servir de point principal d'interaction utilisateur.

- L'UI reflète les changements d'état de l'application, qu'ils proviennent :
  - **D'interactions utilisateur** (ex. : appuyer sur un bouton).
  - **D'entrées externes** (ex. : réponse réseau).

- Les données obtenues de la couche de données sont souvent converties pour répondre aux besoins de l'interface utilisateur. Cela inclut :
  - La sélection de **certaines données uniquement.**
  - **La fusion de sources multiples** pour afficher des informartions pertinentes.



En résumé : L'UI agit comme un pipeline entre les **données** et leur **représentation visuelle.**

Architecture de la couche d'interface utilisateur

L'UI inclut les éléments d'interface (activités, fragments, vues ou Jetpack Compose) qui affichent les données.

- **Étapes principales** :

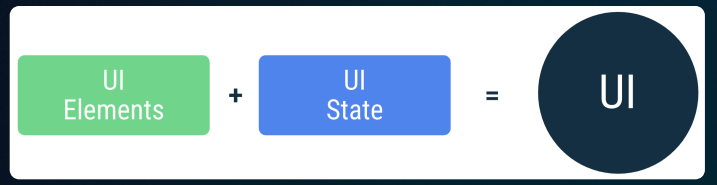
1. Transformer les **données d'application** pour les rendre affichables.
2. Convertir ces données en **éléments d'interface utilisateur** visibles.
3. Réagir aux **interactions utilisateur** pour mettre à jour les données d'interface utilisateur si nécessaire.
4. Répéter ce processus selon les besoins.

Architecture de la couche d'interface utilisateur

L'UI inclut les éléments d'interface (activités, fragments, vues ou Jetpack Compose) qui affichent les données.

- **Concepts clés :**

1. Définir **l'état de l'UI.**
2. Utiliser le **Flux de données unidirectionnel (UDF)** pour gérer l'état.
3. Exposer l'état avec des **types de données observables**.
4. Construire une UI qui s'appuie sur cet état observable.



En résumé, le cœur de cette architecture repose sur une gestion claire et réactive de l'état de l'interface utilisateur.

Architecture de la couche d'interface utilisateur

```kotlin
data class NewsUiState(
    val isSignedIn: Boolean = false,
    val isPremium: Boolean = false,
    val newsItems: List<NewsItemUiState> = listOf(),
    val userMessages: List<Message> = listOf()
)

data class NewsItemUiState(
    val title: String,
    val body: String,
    val bookmarked: Boolean = false,
    ...
)
```

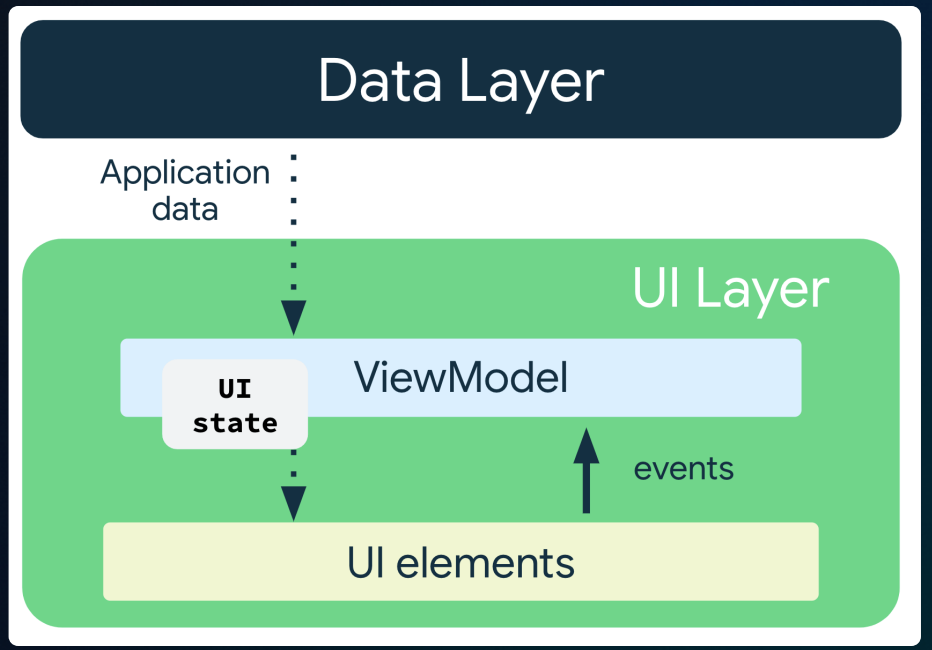
Immuabilité : un point clé

- **Les objets immuables garantissent la cohérence des données.**
- Ne modifiez jamais l'état de l'UI, sauf si l'UI est **la seule source de ses données.**
- Les classes immuables empêchent les **incohérences** entre l'UI et la couche de données.

Architecture de la couche d'interface utilisateur

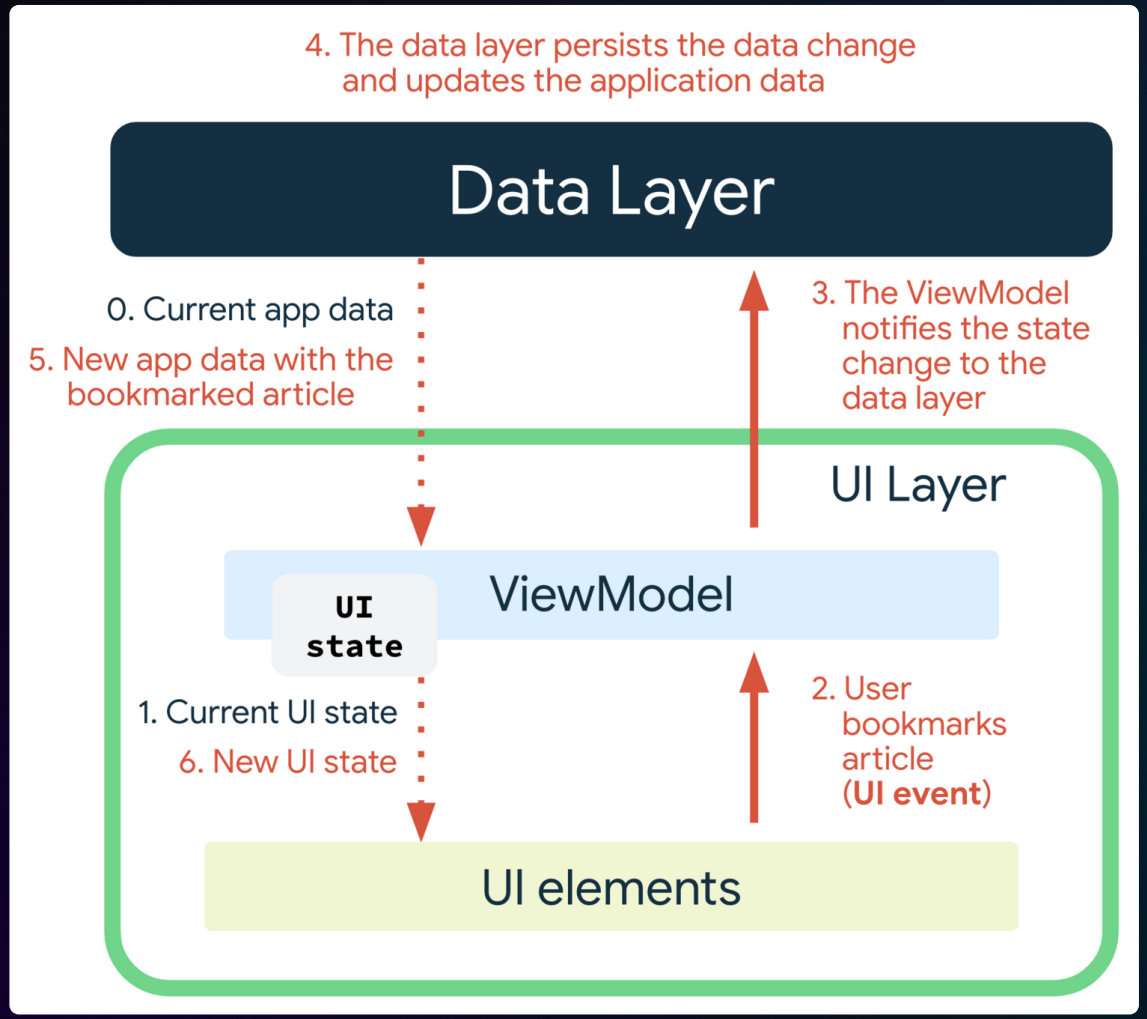
Conteneurs d'état (state holders)

- **Rôle :** Produire l'état de l'UI et contenir la logique associée.
- **Exemples :** Un widget ou une destination de navigation (souvent un ViewModel).
- **Recommandation :** Utilisez un `ViewModel` pour gérer l'état de l'UI au niveau de l'écran et accéder à la couche de données. Il survit aux modifications de configuration.
- **Fonctionnement :** Gère les événements entrants et produit un état mis à jour (voir schéma).



📝 **Principe de UDF (Unidirectional Data Flow)** : Les données circulent dans une seule direction, des conteneurs d'état vers l'UI, et les événements dans l'autre sens.

Architecture de la couche d'interface utilisateur



Architecture de la couche d'interface utilisateur

Exposer l’état de l’interface utilisateur

- **Objectif :** Permettre à l’UI de réagir automatiquement aux modifications de l’état produit sans extraire manuellement les données.
- **Comment ?** En exposant l’état de l’interface utilisateur via un conteneur de données observable, tel que :
  - `LiveData`
  - `StateFlow`
- **Avantages des conteneurs observables :**
  - L’UI suit automatiquement les changements d’état.
  - Mise en cache de la dernière version de l’état (utile pour les restaurations après des modifications de configuration).
  - Produit des états successifs sous forme de flux.

```
1 class NewsViewModel(...) : ViewModel() {
2
3     val uiState: StateFlow<NewsUiState> = …
4 }
```

Architecture de la couche d'interface utilisateur

```
class NewsViewModel(
    private val repository: NewsRepository,
    ...
) : ViewModel() {

    private val _uiState = MutableStateFlow(NewsUiState())
    val uiState: StateFlow<NewsUiState> = _uiState.asStateFlow()

    private var fetchJob: Job? = null

    fun fetchArticles(category: String) {
        fetchJob?.cancel()
        fetchJob = viewModelScope.launch {
            try {
                val newsItems = repository.newsItemsForCategory(category)
                _uiState.update {
                    it.copy(newsItems = newsItems)
                }
            } catch (ioe: IOException) {
                // Handle the error and notify the UI when appropriate.
                _uiState.update {
                    val messages = getMessagesFromThrowable(ioe)
                    it.copy(userMessages = messages)
                 }
            }
        }
    }
}
```

Architecture de la couche d'interface utilisateur

```
@Composable
fun LatestNewsScreen(
    modifier: Modifier = Modifier,
    viewModel: NewsViewModel = viewModel()
) {
    val uiState by viewModel.uiState.collectAsState()

    Box(modifier.fillMaxSize()) {

        if (viewModel.uiState.isFetchingArticles) {
            CircularProgressIndicator(Modifier.align(Alignment.Center))
        }

        // Add other UI elements. For example, the list.
    }
}
```

Architecture de la couche d'interface utilisateur

```
class NewsActivity : AppCompatActivity() {

    private val viewModel: NewsViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...

        lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.uiState.collect {
                    // Update UI elements
                }
            }
        }
    }
}
```

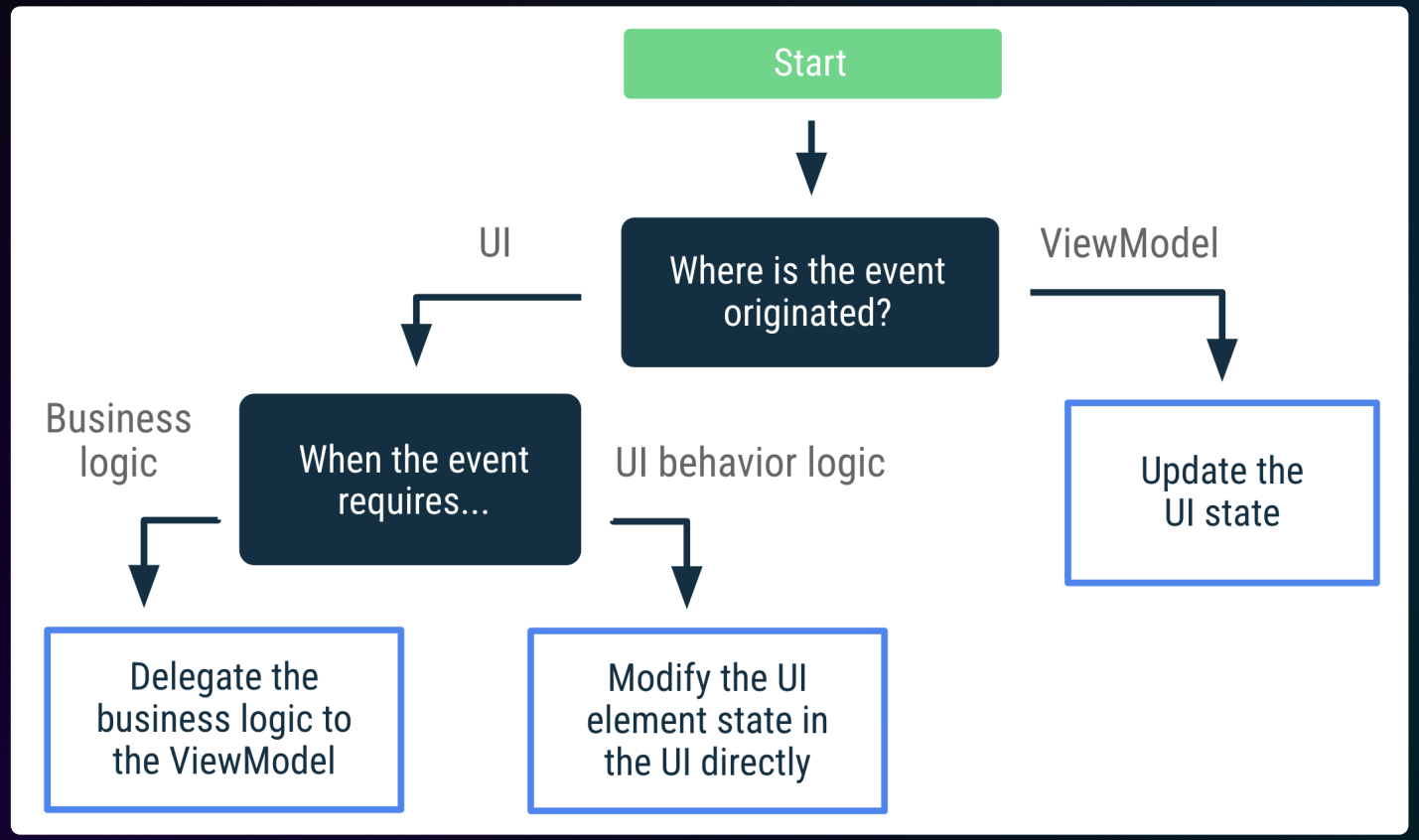
Architecture de la couche d'interface utilisateur

Gérer les évènements utilisateur

```
@Composable
fun LatestNewsScreen(viewModel: LatestNewsViewModel = viewModel()) {

    // State of whether more details should be shown
    var expanded by remember { mutableStateOf(false) }

    Column {
        Text("Some text")
        if (expanded) {
            Text("More details")
        }

        Button(
          // The expand details event is processed by the UI that
          // modifies this composable's internal state.
          onClick = { expanded = !expanded }
        ) {
          val expandText = if (expanded) "Collapse" else "Expand"
          Text("$expandText details")
        }

        // The refresh event is processed by the ViewModel that is in charge
        // of the UI's business logic.
        Button(onClick = { viewModel.refreshNews() }) {
            Text("Refresh data")
        }
    }
}
```

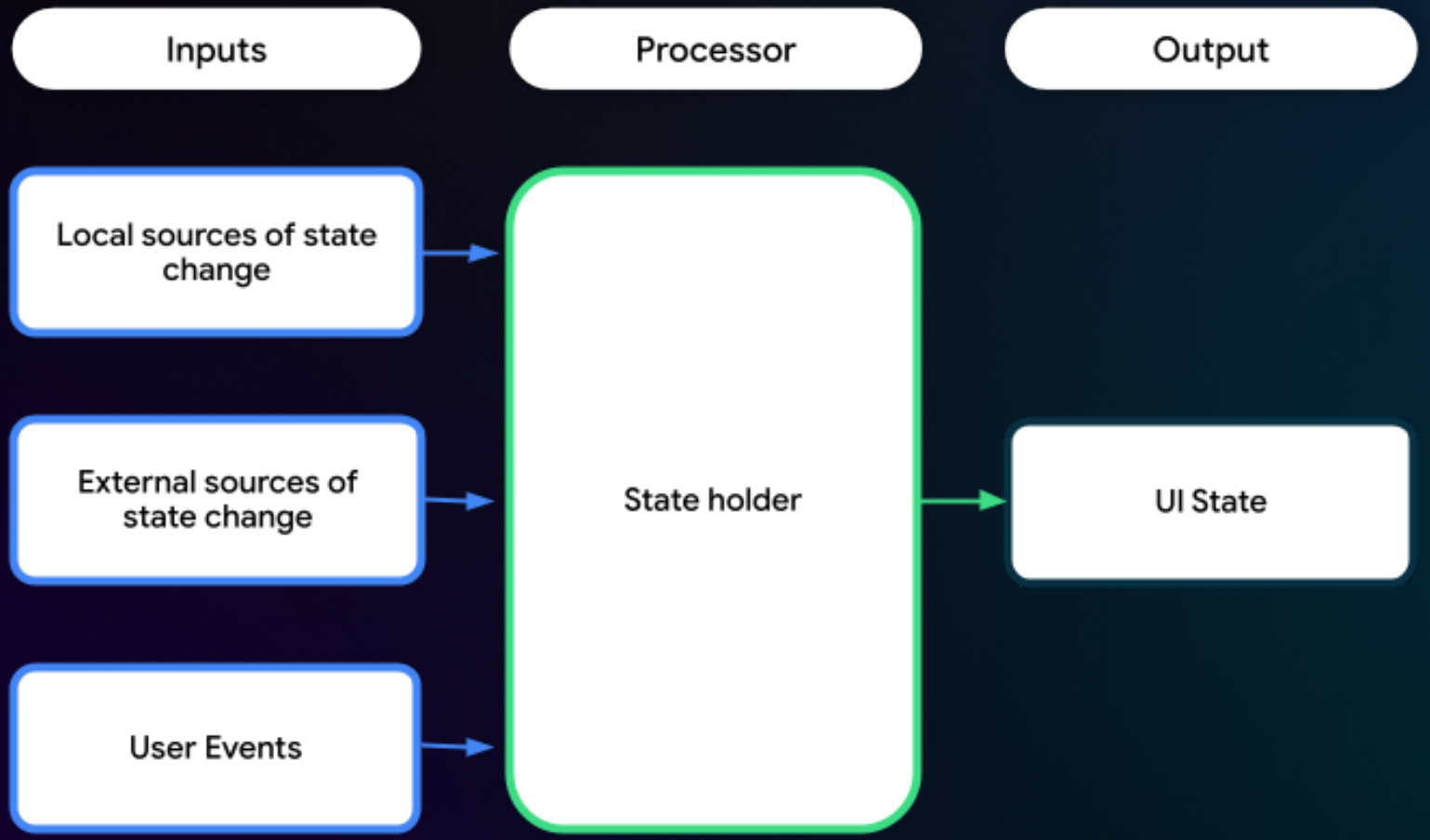
Architecture de la couche d'interface utilisateur

Identifier la meilleure méthode pour gérer un évènement



Architecture de la couche d'interface utilisateur

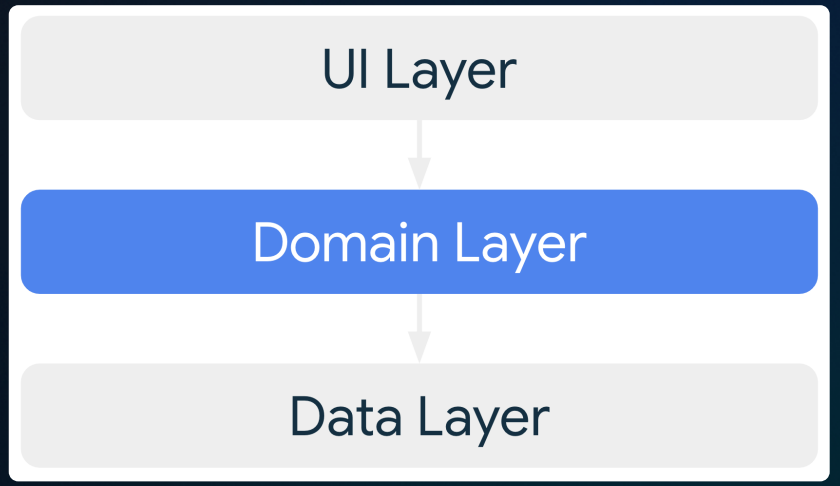
En résumé



Couche de domaine

Exposer l’état de l’interface utilisateur

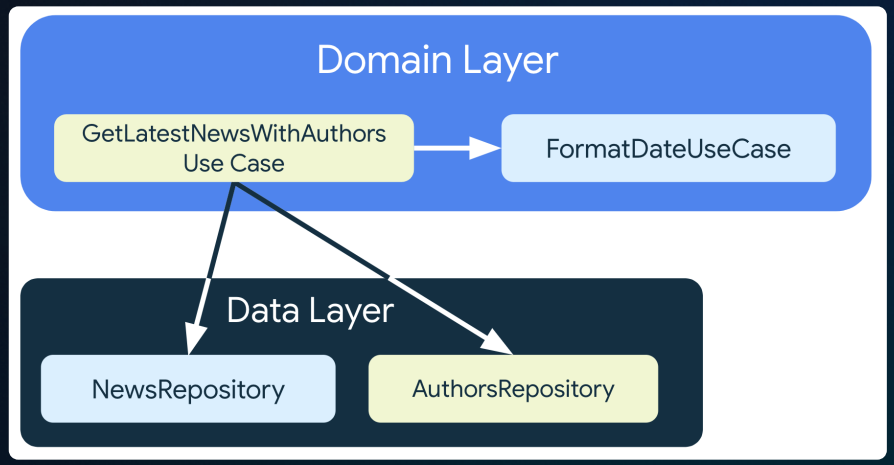
- **Rôle** : Encapsuler la logique métier complexe ou partagée entre plusieurs ViewModels.
- **Avantages** :
  - ✅ Évite la duplication de code.
  - ✅ Simplifie les classes qui l'utilisent.
  - ✅ Améliore la testabilité.
  - ✅ Allège les classes en partageant les responsabilités.
- **Dépendances**
  - Ces classes se situent entre les ViewModels (UI) et les dépôts de la couche de données.
  - Ils dépendent des dépôts et communiquent avec l'UI via des **callback** (java) ou **coroutines** (kotlin)



Couche de domaine

UseCase (Cas d'utilisation)

- Les **UseCase** encapsulent une fonctionnalité métier spécifique.
- Les UseCase peuvent également **dépendre les uns des autres**, permettant de composer des fonctionnalités complexes.
- **Exemple illustré dans le schéma :**
  - **GetLatestNewsWithAuthorsUseCase** utilise **FormatDateUseCase** pour formater les dates des actualités avant de les transmettre à la couche UI.




```
class GetLatestNewsWithAuthorsUseCase(
  private val newsRepository: NewsRepository,
  private val authorsRepository: AuthorsRepository,
  private val formatDateUseCase: FormatDateUseCase
) { /* ... */ }
```

Couche de domaine

En Kotlin, les instances de UseCase peuvent être appelées comme des fonctions en utilisant `operator fun invoke()`.

```kotlin
class FormatDateUseCase(userRepository: UserRepository) {

    private val formatter = SimpleDateFormat(
        userRepository.getPreferredDateFormat(),
        userRepository.getPreferredLocale()
    )

    operator fun invoke(date: Date): String {
        return formatter.format(date)
    }
}
```

Points clés :

- La méthode `invoke()` accepte un nombre quelconque de paramètres et peut renvoyer n'importe quel type.
- Elle peut être **surchargée** avec différentes signatures pour répondre à divers besoins.

Couche de domaine

En Kotlin, les instances de UseCase peuvent être appelées comme des fonctions en utilisant `operator fun invoke()`.

```kotlin
// Utilisation d'une classe UseCase
class MyViewModel(formatDateUseCase: FormatDateUseCase) : ViewModel() {
    init {
        val today = Calendar.getInstance()
        val todaysDate = formatDateUseCase(today)
        /* ... */
    }
}
```

Points importants :

- Les UseCases peuvent être appelés depuis :
  - Les classes de la couche **UI** (ViewModel, Activity).
  - Les **services** ou la classe **Application**.

- Les **données des UseCases** doivent être **immuables**, garantissant qu'ils peuvent être réutilisés sans effet de bord.

Couche de domaine

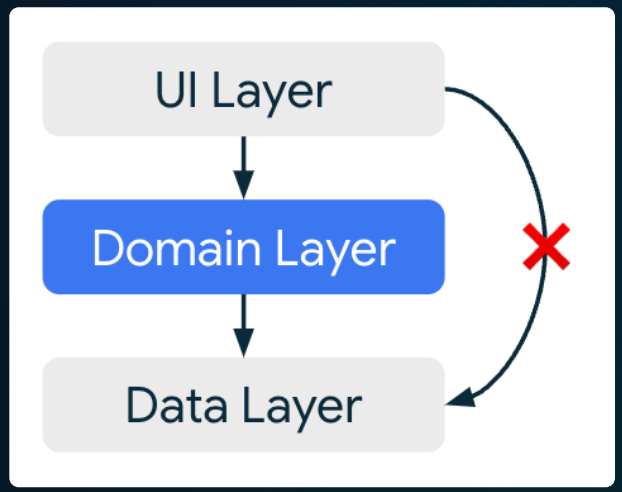
Restriction d'accès à la couche de données : Faut-il autoriser l'accès direct à la couche de données depuis la couche UI ou toujours passer par la couche de domaine ?

✅ Avantages de la restriction :

- Empêche l'UI de contourner la logique métier définie dans la couche de domaine.
- Simplifie la journalisation ou les actions globales sur les appels à la couche de données (ex. : analytique).

❌ Inconvénients potentiels :

- Complexité accrue : nécessite des **cas d'utilisation** même pour des appels simples.
- Peut rendre le code plus lourd sans valeur ajoutée significative.

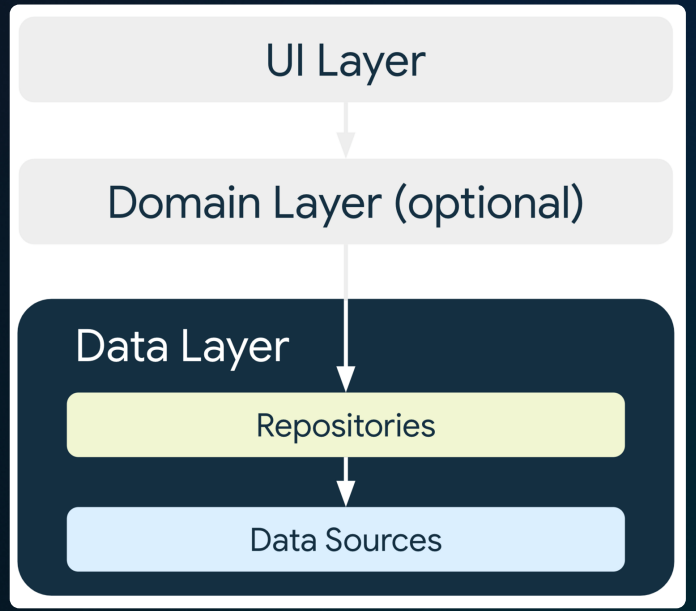


Cela va dépendre de la **taille** et la **complexité** du codebase ainsi que de votre **préférence** entre règles strictes ou flexibilité.

Couche de données

- **Rôle** :
  - Contient les **données de l'application** et la **logique métier**.
  - La **logique métier** définit les règles concernant la création, le stockage et la modification des données.

- **Pourquoi séparer la couche de données ?**
  - **Réutilisation :** Utilisable sur plusieurs écrans.
  - **Partage :** Centralise les informations partagées entre différentes parties de l'application.
  - **Testabilité :** Permet de tester la logique métier en dehors de l'interface utilisateur.

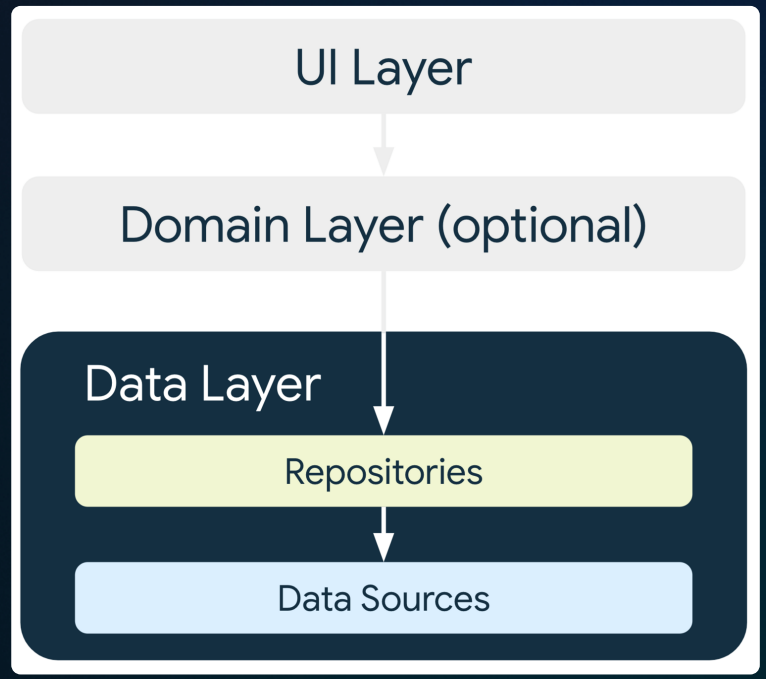


Architecture de la couche de données

Rôle des classes de dépôt (repositories) :

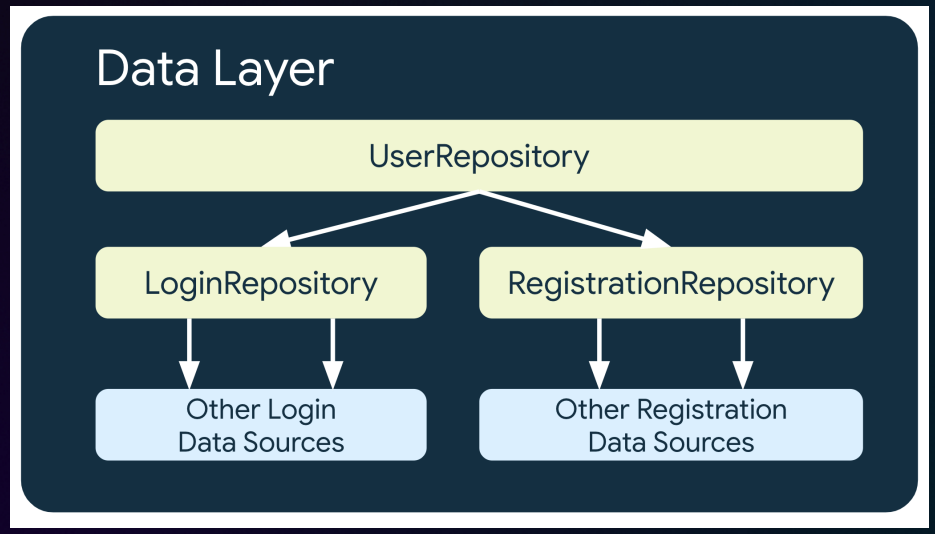
Les dépôts agissent comme un point central pour la gestion des données et peuvent inclure plusieurs sources de données (locales, distantes, etc.).

- **Responsabilités des classes de dépôt :**
  - **Présenter les données** au reste de l'application.
  - **Centraliser les modifications** apportées aux données.
  - **Résoudre les conflits** entre différentes sources de données.
  - **Extraire les sources de données** du reste de l'application.
  - **Contenir la logique métier** liée aux données.



Architecture de la couche de données

```
class ExampleRepository(
    private val exampleRemoteDataSource: ExampleRemoteDataSource, // network
    private val exampleLocalDataSource: ExampleLocalDataSource // database
) { /* ... */ }
```



Architecture de la couche de données

Opérations et Flux dans la couche de données

1. Opérations ponctuelles

La couche de données doit permettre l'exécution d'opérations CRUD ponctuelles. En Kotlin, cela se fait par des fonctions de suspension (`suspend`) pour effectuer des appels asynchrones de manière simple et efficace.

```kotlin
suspend fun insertMovie(movie: Movie)
suspend fun getMovies(): List<Movie>
```

2. Flux de données en temps réel

Pour être informé des changements de données au fil du temps, la couche de données utilise des **flux** en Kotlin avec `Flow` ou `StateFlow` pour permettre à l'application de réagir à toute modification des données de manière réactive.

```kotlin
fun getMoviesStream(): Flow<List<Movie>>
```

Application orientée hors connexion

Une application hors connexion peut exécuter tout ou partie de sa logique métier **sans accès à Internet**.

- **Problèmes fréquents liés au réseau :**
  - **Bande passante limitée**
  - **Interruptions transitoires** (ascenseur, tunnel, etc.)
  - **Accès occasionnel au réseau** (par exemple, tablettes Wi-Fi uniquement)

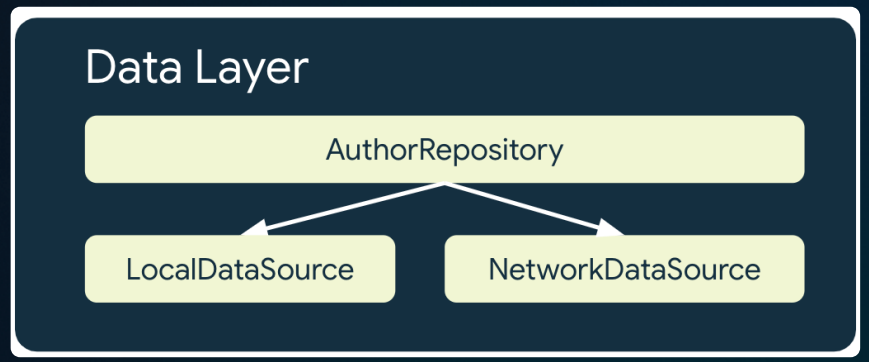
- **Objectifs pour une application hors connexion :**
  - **Rester fonctionnelle sans connexion réseau fiable.**
  - **Afficher immédiatement les données locales** (pas d'attente pour les appels réseau).
  - **Optimiser la récupération des données** (conditions optimales recharge ou Wi-Fi)

Modélisation des données : Applications hors connexion

Deux sources de données par dépôt :

1. Source de données locale

- Source **fiable** et **exclusive** pour les couches supérieures.
- Garantit la **cohérence des données** entre états connectés et hors connexion.
- **Stockage local :**
  - Bases de données relationnelles (**Room**).
  - Formats non structurés (**Datastore**, fichiers...)



2. Source de données réseau

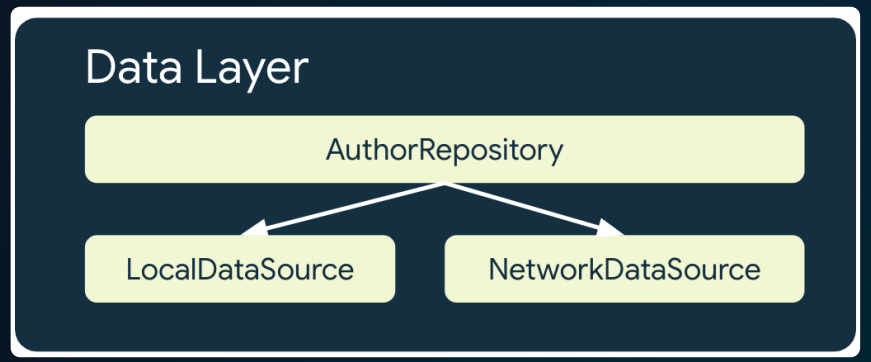
- Représente l'**état réel** des données.
- Synchronisée avec la source locale lorsque possible.
- Peut être :
  - **En avance** (lorsque les données locales ne sont pas actualisées).
  - **En retard** (jusqu'à la restauration de la connectivité).

Modélisation des données : Applications hors connexion

Rôle du dépôt (Repository)

- **Seul responsable** de :
  - La communication avec la source réseau.
  - La mise à jour de la source locale.

- Les couches de domaine et UI n'accèdent jamais directement au réseau.



Modélisation des données : Applications hors connexion

```kotlin
/**
 * Network representation of [Author]
 */
@Serializable
data class NetworkAuthor(
    val id: String,
    val name: String,
    val imageUrl: String,
    val twitter: String,
    val mediumPage: String,
    val bio: String,
)

/**
 * Defines an author for local database.
 */
@Entity(tableName = "authors")
data class AuthorEntity(
    @PrimaryKey
    val id: String,
    val name: String,
    @ColumnInfo(name = "image_url")
    val imageUrl: String,
    @ColumnInfo(defaultValue = "")
    val twitter: String,
    @ColumnInfo(name = "medium_page", defaultValue = "")
    val mediumPage: String,
    @ColumnInfo(defaultValue = "")
    val bio: String,
)
```

```kotlin
/**
 * External data layer representation of Author
 */
data class Author(
    val id: String,
    val name: String,
    val imageUrl: String,
    val twitter: String,
    val mediumPage: String,
    val bio: String,
)
```

Modélisation des données : Applications hors connexion

Les défis des applications hors connexion

Principaux défis : Synchronisation des données

- Trouver la bonne stratégie de synchronisation entre la source locale et la source réseau.
- Gérer les cas où :
  - Le réseau est lent ou intermittent.
  - Les données locales ou réseau sont obsolètes.
  - Des conflits surviennent entre les mises à jour locales et réseau.

Conclusion

Les principes fondamentaux :

- **Séparation des responsabilités** : Chaque couche a un rôle clair (UI, domaine, données), ce qui améliore la lisibilité, la maintenabilité et la testabilité.
- **Single Source of Truth (SSOT)** : Garantir une gestion centralisée et cohérente de l'état pour éviter les conflits de données.
- **Modularité et réutilisabilité** : Créer des composants facilement testables et réutilisables, comme les UseCases, les ViewModels et les Repositories.
- **Réactivité** : Exploiter des flux de données comme **StateFlow** ou **LiveData** pour réagir efficacement aux changements d'état.

Conclusion

Une architecture évolutive :

- **Pas de solution universelle** : L'architecture doit être adaptée au contexte de l'application (simple ou complexe, connectée ou hors ligne, etc.).
- **Ajouter de la complexité uniquement lorsque nécessaire** : Par exemple, la couche de domaine ou les UseCases ne sont indispensables que pour gérer des scénarios complexes ou favoriser la réutilisation.
- **Prise en compte des contraintes modernes** : Optimisation pour des environnements instables (mode hors connexion), gestion efficace des ressources (batterie, réseau) et prise en charge de plusieurs plateformes (tablettes, téléphones, etc.).

Conclusion

Clé de la réussite :

- Penser à l'**expérience utilisateur** en priorité. Une architecture bien conçue permet :
  - Une application réactive et fluide.
  - Une gestion transparente des données, même dans des conditions réseau difficiles.
  - Une capacité à évoluer avec les besoins de l'application et des utilisateurs.

Et ensuite ?

- **Pratique, pratique, pratique** :
  Mettre en œuvre les concepts vus en cours à travers des projets réels.

- **Suivre les bonnes pratiques** :
  Rester à jour avec les recommandations officielles et les outils modernes d'Android.