Recommandations pour l'architecture Android

Cette page présente plusieurs bonnes pratiques et recommandations concernant l'architecture (https://developer.android.com/topic/architecture?hl=fr). Adoptez-les pour améliorer la qualité, la robustesse et l'évolutivité de votre application. Elles facilitent également la gestion et le test de votre application.

Remarque : Vous devez considérer les recommandations du document comme tel et non comme des exigences strictes. Adaptez-les à votre application si nécessaire.

Les bonnes pratiques ci-dessous sont regroupées par thème. Chacune d'elles a une priorité indiquant à quel point l'équipe la recommande. Voici la liste des priorités :

- **Fortement recommandé**:vous devriez appliquer cette pratique, sauf si elle entre fondamentalement en conflit avec votre approche.
- **Recommandé** : il est probable que cette pratique améliore votre application.
- **Facultatif** : cette pratique peut améliorer votre application dans certains cas.

Remarque : Pour comprendre ces recommandations, vous devez connaître les conseils sur l'architecture (https://developer.android.com/topic/architecture?hl=fr).

Architecture multicouche

Notre architecture multicouche (https://developer.android.com/topic/architecture?gclid=CjwKCAjw6raYBhB7EiwABge5Klm_5PN8nJF0Jrb_ymrPP0JAEsbmemmGv_nsn0nBQKQtQMCBuvjehRoC7qcQAvD_BwE&%3Bgclsrc=aw.ds&hl=fr#recommended-app-arch) recommandée favorise la séparation des tâches. Elle pilote l'interface utilisateur à partir de

modèles de données, est conforme au principe de référence unique et suit les principes du flux de données unidirectionnel (https://developer.android.com/topic/architecture?gclid=CjwKCAjw6raYBhB7EiwABge5Klm_5PN8nJF0Jrb_ymrPP0JAEsbmemmGv_nsn0nBQKQtQMCBuvjehRoC7qcQAvD_BwE&%3Bgclsrc=aw.ds&hl=fr#unidirectional-data-flow). Voici quelques bonnes pratiques pour l'architecture multicouche :

Recommandation	Description
Utilisez une couche de données (https://developer.android.com/jetpack/guide/data-layer?hl=fr) clairement définie. **Fortement recommandé**	La couche de données (https://developer.android.com/jetpack/guide/data-layer?hl=fr) expose les données de l'application au reste de l'application et contient la grande majorité de la logique métier de votre application. • Créez des dépôts (https://developer.android.com/topic/architecture/data-layer?hl=fr#architecture) même s'ils ne contiennent qu'une seule source de données. • Dans les petites applications, vous pouvez choisir de placer les types de couches de données dans un module ou un package `data`.
Utilisez une couche d'interface utilisateur (https://developer.android.com/jetpack/guide/ui-layer?hl=fr) bien définie. **Fortement recommandé**	La couche d'interface utilisateur (https://developer.android.com/jetpack/guide/ui-layer?hl=fr) affiche les données de l'application à l'écran et sert de point principal d'interaction utilisateur. • Dans les petites applications, vous pouvez choisir de placer les types de couches de données dans un module ou un package `ui`. En savoir plus sur les bonnes pratiques pour la couche d'interface utilisateur (#ui-layer).
La couche de données (https://developer.android.com/jetpack/guide/data-layer?hl=fr) doit exposer les données d'application à l'aide d'un dépôt. **Fortement recommandé**	Les composants de la couche d'interface utilisateur tels que les composables, les activités ou les ViewModels ne doivent pas interagir directement avec une source de données. Exemples de sources de données : • API Database, DataStore, SharedPreferences et Firebase. • Fournisseurs de géolocalisation GPS. • Fournisseurs de données Bluetooth. • Fournisseur d'état de connectivité réseau.

Recommandation	Description
Utilisez des coroutines et des flux (https://developer.android.com/kotlin/coroutines?gclid=CjwKCAjwhNWZBhB_EiwAPzlhNtReVIBfrUFBUt6SqZz3YLezP9YEiGuBube4YSTrOF-0ovxzpNGNaRoCiYsQAvD_BwE&gclsrc=aw.ds&hl=fr). Fortement recommandé	Utilisez des coroutines et des flux (https://developer.android.com/kotlin/coroutines?gclid=CjwKCAjwhNWZBhB_EiwAPzlhNtReVIBfrUFBUt6SqZz3YLezP9YEiGuBube4YSTrOF-0ovxzpNGNaRoCiYsQAvD_BwE&gclsrc=aw.ds&hl=fr) pour communiquer entre les couches. Autres bonnes pratiques pour les coroutines (https://developer.android.com/kotlin/coroutines/coroutines-best-practices?hl=fr)
Utilisez une couche de domaine (https://developer.android.com/jetpack/guide/domain-layer?hl=fr). Recommandé pour les applications volumineuses	Utilisez une couche de domaine (https://developer.android.com/jetpack/guide/domain-layer?hl=fr) si vous devez réutiliser la logique métier qui interagit avec la couche de données sur plusieurs ViewModels, ou si vous souhaitez simplifier la logique métier d'un ViewModel particulier.

Couche d'interface utilisateur

Le rôle de la couche d'interface utilisateur (https://developer.android.com/topic/architecture/ui-layer?hl=fr) est d'afficher les données de l'application à l'écran et de servir de point principal d'interaction utilisateur. Voici quelques bonnes pratiques pour la couche d'interface utilisateur :

Recommandation	Description
Suivez les principes du flux de données unidirectionnel (UDF) (https://developer.android.com/jetpack/compose/architecture?hl=fr#udf). Fortement recommandé	Suivez les principes du flux de données unidirectionnel (UDF) (https://developer.android.com/jetpack/compose/architecture?hl=fr#udf), selon lesquels les ViewModels exposent l'état de l'UI à l'aide du modèle d'observateur et reçoivent les actions de l'UI via des appels de méthode.

Recommandation	Description
Utilisez des ViewModels AAC (https://developer.android.com/topic/libraries/architecture/viewmodel?hl=fr) si cela peut être utile pour votre application. Fortement recommandé	Utilisez des ViewModels AAC (https://developer.android.com/topic/libraries/architecture/viewmodel?hl=fr) pour gérer la logique métier (https://developer.android.com/jetpack/guide/ui-layer?hl=fr#logic-types) et récupérer les données de l'application pour exposer l'état de l'UI à l'UI (Compose ou Android Views). En savoir plus sur les bonnes pratiques des ViewModels (#viewmodel) En savoir plus sur les avantages des ViewModels (https://developer.android.com/topic/architecture/ui-layer/stateholders?hl=fr#viewmodel-as)
Collectez l'état de l'UI en tenant compte du cycle de vie. Fortement recommandé	Collectez l'état de l'interface utilisateur depuis celle-ci à l'aide du compilateur de coroutines qui tient compte du cycle de vie : `repeatOnLifecycle` (https://developer.android.com/reference/kotlin/androidx/lifecycle/package-summary?hl=fr#(androidx.lifecycle.Lifecycle).repeatOnLifecycle(androidx.lifecycle.Lifecycle.State,kotlin.coroutines.SuspendFunction1)) dans le système View et `collectAsStateWithLifecycle` (https://developer.android.com/reference/kotlin/androidx/lifecycle/compose/package-summary?hl=fr#(kotlinx.coroutines.flow.StateFlow).collectAsStateWithLifecycle(androidx.lifecycle.LifecycleOwner,androidx.lifecycle.Lifecycle.State,kotlin.coroutines.CoroutineContext)) dans Jetpack Compose. En savoir plus sur `repeatOnLifecycle` (https://medium.com/androiddevelopers/a-safer-way-to-collect-flows-from-android-uis-23080b1f8bda) En savoir plus sur `collectAsStateWithLifecycle` (https://medium.com/androiddevelopers/consuming-flows-safely-in-jetpack-compose-cde014d0d5a3)
N'envoyez pas d'événements du ViewModel à l'UI. Fortement recommandé	Traitez l'événement immédiatement dans ViewModel et mettez à jour l'état avec le résultat de la gestion de l'événement. En savoir plus sur les événements d'interface utilisateur (https://developer.android.com/topic/architecture/ui-layer/events?hl=fr#handle-viewmodel-events)
Utilisez une application à activité unique.	Utilisez Navigation Fragments (https://developer.android.com/guide/navigation?hl=fr) ou

Recommandation	Description
Recommandé	Navigation Compose (https://developer.android.com/jetpack/compose/navigation?hl=fr) pour naviguer entre les écrans, et utilisez des liens profonds vers votre application si elle comporte plusieurs écrans.
Utilisez Jetpack Compose (https://developer.android.com/jetpack/compose?hl=fr) . Recommandé	Utilisez Jetpack Compose (https://developer.android.com/jetpack/compose?hl=fr) pour créer des applications pour téléphones, tablettes, appareils pliables et Wear OS.

L'extrait de code suivant montre comment collecter l'état de l'interface utilisateur en tenant compte du cycle de vie :

VuesCompose (#compose)
(#vues)

```
class MyFragment : Fragment() {

    private val viewModel: MyViewModel by viewModel()

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

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.uiState.collect {
                    // Process item
                }
            }
        }
    }
}
```

ViewModel

Les ViewModels (https://developer.android.com/topic/architecture/ui-layer/stateholders?hl=fr#business-logic) fournissent

l'état de l'interface utilisateur et l'accès à la couche de données. Voici quelques bonnes pratiques pour les ViewModels :

Recommandation	Description
Les ViewModels doivent être indépendants du cycle de vie d'Android. Fortement recommandé	Les ViewModels ne doivent faire référence à aucun type lié au cycle de vie. Ne transmettez pas d'`Activity`, `Fragment`, `Context` ni de `Resources` en tant que dépendance. Si un élément a besoin d'un `Context` dans le ViewModel, vous devez sérieusement vous demander s'il se trouve dans la bonne couche.
Utilisez des coroutines et des flux (https://developer.android.com/kotlin/coroutines?gclid=CjwKCAjwhNWZBhB_EiwAPzlhNtReVIBfrUFBUt6SqZz3YLezP9YEiGuBube4YSTrOF-0ovxzpNGNaRoCiYsQAvD_BwE&gclsrc=aw.ds&hl=fr). Fortement recommandé	Le ViewModel interagit avec les couches de données ou de domaine à l'aide des éléments suivants : • Les flux Kotlin, pour recevoir des données d'application • Les fonctions `suspend`, pour effectuer des actions avec `viewModelScope` (https://developer.android.com/topic/libraries/architecture/coroutines?hl=fr#viewmodelscope)
Utilisez les ViewModels au niveau de l'écran. Fortement recommandé	N'utilisez pas de ViewModels dans les éléments réutilisables de l'UI. Vous devez utiliser les ViewModels dans les éléments suivants : • Composables au niveau de l'écran • Activités/Fragments dans Views • Destinations ou graphiques lorsque vous utilisez Jetpack Navigation (https://developer.android.com/guide/navigation?hl=fr)
Utilisez des classes de conteneurs d'état simples (https://developer.android.com/topic/architecture/ui-layer/stateholders?hl=fr#ui-logic) dans les composants d'UI réutilisables. Fortement recommandé	Utilisez des classes de conteneurs d'état simples (https://developer.android.com/topic/architecture/ui-layer/stateholders?hl=fr#ui-logic) pour gérer la complexité des composants d'UI réutilisables. Cela permet de hisser et de contrôler l'état en externe.

Recommandation	Description
N'utilisez pas `AndroidViewModel` (https://developer.android.com/reference/androidx/lifecycle/AndroidViewModel?hl=fr) . Recommandé	Utilisez la classe `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?hl=fr) , pas `AndroidViewModel` (https://developer.android.com/reference/androidx/lifecycle/AndroidViewModel?hl=fr) . La classe `Application` ne doit pas être utilisée dans le ViewModel. Déplacez plutôt la dépendance vers l'UI ou la couche de données.
Exposez un état de l'interface utilisateur. Recommandé	Les ViewModels doivent exposer les données à l'UI via une seule propriété appelée `uiState`. Si l'interface utilisateur affiche plusieurs données sans liens entre elles, le ViewModel peut exposer plusieurs propriétés d'état de l'interface utilisateur (https://developer.android.com/jetpack/guide/ui-layer?hl=fr#additional-considerations) . • Vous devez définir `uiState` comme `StateFlow`. • Vous devez créer l'`uiState` à l'aide de l'opérateur `stateIn` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/state-in.html) avec la règle `WhileSubscribed(5000)` (https://medium.com/androiddevelopers/migrating-from-livedata-to-kotlins-flow-379292f419fb) (exemple) (https://github.com/android/compose-samples/blob/main/JetNews/app/src/main/java/com/example/jetnews/ui/interests/InterestsViewModel.kt#L56) si les données arrivent sous la forme d'un flux de données provenant d'autres couches de la hiérarchie. • Dans les cas plus simples où aucun flux de données ne provient de la couche de données, il est possible d'utiliser un `MutableStateFlow` exposé en tant que `StateFlow` immuable (exemple) (https://github.com/android/compose-samples/blob/main/Jetcaster/app/src/main/java/com/example/jetcaster/ui/home/category/PodcastCategoryViewModel.kt#L37) . • Vous pouvez choisir d'utiliser `${Screen}UiState` comme classe de données pouvant contenir des données, des erreurs et des signaux de chargement. Cette classe peut également être scellée si les différents états sont exclusifs.

L'extrait de code suivant montre comment exposer l'état de l'interface utilisateur à partir d'un ViewModel :

```
@HiltViewModel
class BookmarksViewModel @Inject constructor(
    newsRepository: NewsRepository
) : ViewModel() {

    val feedState: StateFlow<NewsFeedUiState> =
        newsRepository
            .getNewsResourcesStream()
            .mapToFeedState(savedNewsResourcesState)
            .stateIn(
                scope = viewModelScope,
                started = SharingStarted.WhileSubscribed(5_000),
                initialValue = NewsFeedUiState.Loading
            )

    // ...
}
```

Cycle de vie

Voici quelques bonnes pratiques pour le cycle de vie Android (https://developer.android.com/guide/components/activities/activity-lifecycle?hl=fr) :

Recommandation	Description
N'ignorez pas les méthodes du cycle de vie dans les activités ou les fragments. Fortement recommandé	N'ignorez pas les méthodes du cycle de vie comme `onResume` dans les activités ou les fragments. Utilisez `LifecycleObserver` (https://developer.android.com/reference/androidx/lifecycle/LifecycleObserver?hl=fr) à la place. Si l'application doit effectuer des tâches lorsque le cycle de vie atteint un certain `Lifecycle.State`, utilisez l'API `repeatOnLifecycle` (https://developer.android.com/reference/kotlin/androidx/lifecycle/package-summary?hl=fr#(androidx.lifecycle.Lifecycle).repeatOnLifecycle(androidx.lifecycle.Lifecycle.State,kotlin.coroutines.SuspendFunction1)) .

L'extrait de code suivant montre comment effectuer des opérations en fonction d'un certain état du cycle de vie :

VuesCompose (#compose)
(#vues)

```
class MyFragment: Fragment() {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycle.addObserver(object : DefaultLifecycleObs
            override fun onResume(owner: LifecycleOwner) {
                // ...
            }
            override fun onPause(owner: LifecycleOwner) {
                // ...
            }
        }
    }
}
```

Gérer les dépendances

Il existe plusieurs bonnes pratiques à suivre lorsque vous gérez des dépendances entre des composants :

Recommandation	Description
Utilisez l'injection de dépendances (https://developer.android.com/training/dependency-injection?hl=fr) . Fortement recommandé	Dans la mesure du possible, suivez les bonnes pratiques concernant l'injection de dépendances (https://developer.android.com/training/dependency-injection?hl=fr) , en particulier l'injection de constructeur (https://developer.android.com/training/dependency-injection?hl=fr#what-is-di) .

Recommandation	Description
Si nécessaire, limitez la portée à un composant. **Fortement recommandé**	Limitez la portée à un conteneur de dépendances (https://developer.android.com/training/dependency-injection/manual?hl=fr#dependencies-container) lorsque le type contient des données modifiables devant être partagées, ou que son initialisation est coûteuse et qu'il est largement utilisé dans l'application.
Utilisez Hilt (https://developer.android.com/training/dependency-injection/hilt-android?hl=fr) . **Recommandé**	Utilisez Hilt (https://developer.android.com/training/dependency-injection/hilt-android?hl=fr) ou une injection de dépendances manuelle (https://developer.android.com/training/dependency-injection/manual?hl=fr) dans les applications simples. Utilisez Hilt (https://developer.android.com/training/dependency-injection/hilt-android?hl=fr) si votre projet est suffisamment complexe. Par exemple, dans les cas suivants : • Utilisation de plusieurs écrans avec ViewModels : intégration • Utilisation de WorkManager : intégration • Utilisation avancée de la navigation (ViewModels limités au graphique de navigation, par exemple) : intégration

Tests

Voici quelques bonnes pratiques pour les tests (https://developer.android.com/training/testing/fundamentals?hl=fr) :

Recommandation	Description
Identifiez les éléments à tester. (https://developer.android.com/training/testing/fundamentals/what-to-test?hl=fr) **Fortement recommandé**	À moins que le projet ne soit à peu près aussi simple qu'une application Hello World, vous devez le tester, au minimum avec : • ViewModels de test unitaire, y compris Flow. • Entités de couche de données de test unitaire. Autrement dit, les dépôts et les sources de données. • Tests de navigation dans l'interface utilisateur utiles pour effectuer des tests de régression dans CI.

Recommandation	Description
Préférez les faux aux simulations. Fortement recommandé	Pour en savoir plus, consultez Utiliser les doubles de test dans la documentation Android (https://developer.android.com/training/testing/fundamentals/test-doubles?hl=fr) .
Testez les StateFlows. Fortement recommandé	Lorsque vous testez `StateFlow` : • Revendiquez la propriété `value` (https://developer.android.com/kotlin/flow/test?hl=fr#stateflows) dans la mesure du possible. • Vous devez créer un `collectJob` (https://developer.android.com/kotlin/flow/test?hl=fr#statein) si vous utilisez `WhileSubscribed`.

Pour en savoir plus, consultez le guide Éléments à tester dans Android DAC (https://developer.android.com/training/testing/fundamentals/what-to-test?hl=fr).

Modèles

Appliquez ces bonnes pratiques lorsque vous développez des modèles dans vos applications :

Recommandation	Description
Créez un modèle par couche dans les applications complexes. Recommandé	Dans les applications complexes, créez des modèles dans différentes couches ou différents composants lorsque cela est pertinent. Exemples : • Une source de données distante peut mapper le modèle qu'elle reçoit via le réseau sur une classe plus simple avec uniquement les données dont l'application a besoin. • Les dépôts peuvent mapper des modèles DAO à des classes de données plus simples avec seulement les informations dont la couche d'interface utilisateur a besoin. • ViewModel peut inclure des modèles de couches de données dans les classes `UiState`.

Conventions d'attribution de noms

Lorsque vous nommez votre codebase, vous devez tenir compte des bonnes pratiques suivantes :

Recommandation	Description
Nom des méthodes. **Facultatif**	Le nom des méthodes doit être un groupe verbal. Par exemple, `makePayment()`.
Nom des propriétés. **Facultatif**	Le nom des propriétés doit être un groupe nominal. Par exemple, `inProgressTopicSelection`.
Nom des flux de données. **Facultatif**	Lorsqu'une classe expose un flux Flow, LiveData ou autre, la convention d'attribution de noms est `get{model}Stream()`. Par exemple, `getAuthorStream(): Flow<Author>`. Si la fonction renvoie une liste de modèles, le nom du modèle doit être au pluriel : `getAuthorsStream(): Flow<List<Author>>`
Nom des implémentations d'interface. **Facultatif**	Les noms des implémentations d'interface doivent être explicites. Utilisez `Default` comme préfixe si vous ne trouvez aucun nom plus adapté. Par exemple, pour une interface `NewsRepository`, vous pouvez utiliser `OfflineFirstNewsRepository` ou `InMemoryNewsRepository`. Si vous ne trouvez pas de nom approprié, utilisez `DefaultNewsRepository`. Les fausses implémentations doivent être précédées de `Fake`, par exemple `FakeAuthorsRepository`.