

Programmation Mobile

Partie II : Android Moderne

L3 MIAGE

2024-2025

Leo Donati

Programmation Mobile

- Partie II : Android Moderne
 - 1. Gradle
 - 2. Activité
 - 3. UI avec Jetpack Compose
 - 4. Couches
 - 5. Navigation
 - 6. Repository

Gradle

- Gradle est le système d'automatisation des builds et de gestion des dépendances officiel de Android
- Basé sur Kotlin (ou Groovy) plutôt que le XML de Maven
 - Plus lisible
 - Plus concis
 - Plus flexible et personnalisable
 - Prend en charge le build incrémental et/ou parallèle

Fichiers de configuration

- **build.gradle.kts**
 - Fichier de configuration principal du projet
 - Écrit en Kotlin DSL (Domain-Specific language)
 - Définit les dépendances, les plugins, les configurations de compilation et les tâches de build
- **settings.gradle.kts**
- **gradle.properties**
- **gradlew**



```
1  plugins {  
2      alias(libs.plugins.android.application)  
3      alias(libs.plugins.kotlin.android)  
4      alias(libs.plugins.kotlin.compose)  
5  }  
6  android {  
7      namespace = "fr.unica.miage.donati.pizzapp"  
8      compileSdk = 34  
9  
10     defaultConfig {  
11         applicationId = "fr.unica.miage.donati.pizzapp"  
12         minSdk = 29  
13         targetSdk = 34  
14         versionCode = 1  
15         versionName = "1.0"  
16     }  
17     buildTypes {  
18         release {...}  
19     }  
20     compileOptions {...}  
21     kotlinOptions { jvmTarget = "11" }  
22     buildFeatures { compose = true }  
23 }  
24 dependencies {  
25     implementation(libs.appcompat.v7)  
26     ...  
27     testImplementation(libs.junit)  
28     ...  
29     androidTestImplementation(libs.junit)  
30     ...  
31     debugImplementation(libs.ui.tooling)  
32     ...  
33 }
```

- On conseille d'utiliser le gradle wrapper plutôt que directement les commandes gradle



```
1 ./gradlew build // Linux ou OSX
2
3 ./gradlew build // Windows
4
5 ./gradlew tasks
```

Commandes gradle

Welcome to Gradle 8.9!

Here are the highlights of this release:

- Enhanced Error and Warning Messages
- IDE Integration Improvements
- Daemon JVM Information

For more details see <https://docs.gradle.org/8.9/release-notes.html>

Starting a Gradle Daemon (subsequent builds will be faster)

> Task :tasks

Tasks runnable from root project 'PizzApp'

Android tasks

Version Catalog

- Gradle a rajouté la notion de catalogue de version dans le fichier `libs.versions.tom`
- Avantages :
 - un seul lieu où définir les dépendances au lieu de les répéter pour chaque module : plus de cohérence (on risquait d'avoir des versions différentes dans des modules différents)
 - Facile à mettre à jour : seulement le n° de version à changer
 - Vue centralisée
 - Typés comme String
 - Refactorisation

Comment fonctionne le catalogue

C'est un fichier TOML qui définit les dépendances d'une façon structurée

- [versions] : on définit des alias pour les n° de version
- [libraries] : on définit une librairie à partir d'un module (groupe et nom) et d'un alias de version
- [plugins] : on définit les plugin gradle qu'on utilise (avec ID et version)

Ensuite dans le build.gradle.kts on fait référence juste aux noms des dépendances et des plugins

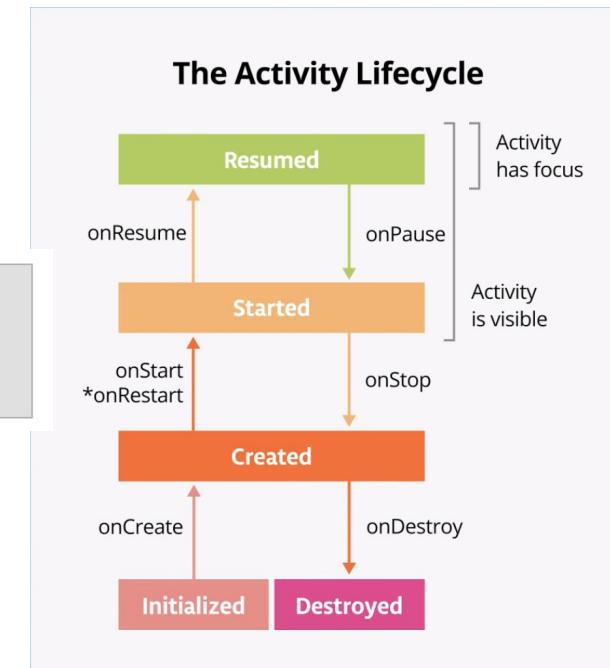
Programmation Mobile

- Partie II : Android Moderne
 - 1. Gradle
 - 2. Activité
 - 3. UI avec Jetpack Compose
 - 4. Couches
 - 5. Navigation
 - 6. Repository

Cycle de vie d'une activité

- `onCreate()`
 - Appelle `setContent` et lui passe le composable parent de la hiérarchie

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

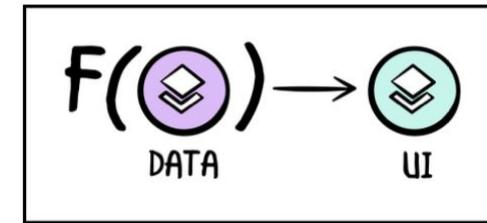


Programmation Mobile

- Partie II : Android Moderne
 - 1. Gradle
 - 2. Activités
 - 3. UI avec Jetpack Compose
 - 4. Couches
 - 5. Navigation
 - 6. Repository

Composable

- Une composable est une fonction Kotlin
 - marquée par le décorateur @Composable
 - avec autant d'arguments que l'on veut
 - Qui ne renvoie rien
- Sa responsabilité ?
 - => Émettre un élément d'interface utilisateur (UI)



UI as a Function of Data

Fonctions composables

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Bonjour $name!",
        modifier = modifier
            .background(color = androidx.compose.ui.graphics.Color.Yellow),
        style = MaterialTheme.typography.headlineSmall
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    PizzAppTheme {
        Greeting(name: "Leo")
    }
}
```

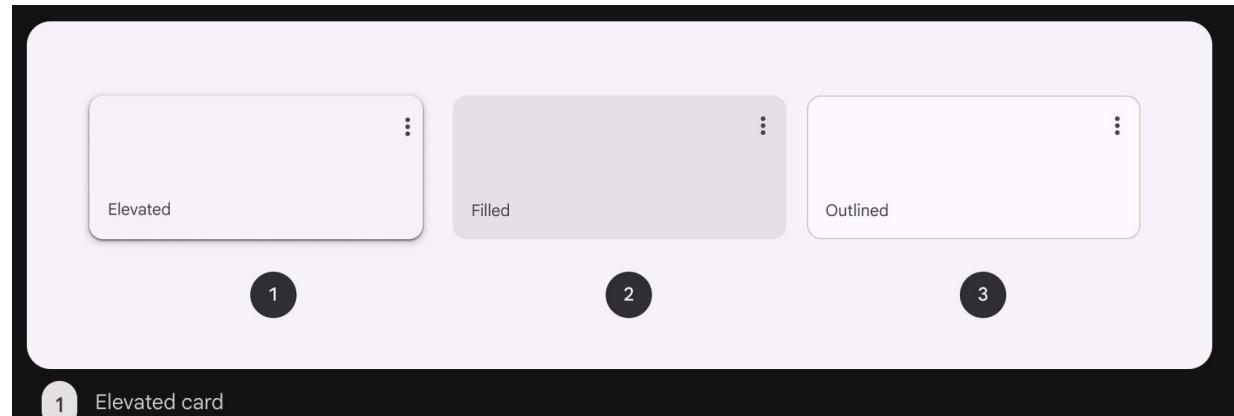
GreetingPreview

Bonjour Leo!

Composable conteneurs

- Pour placer à l'intérieur plusieurs composables

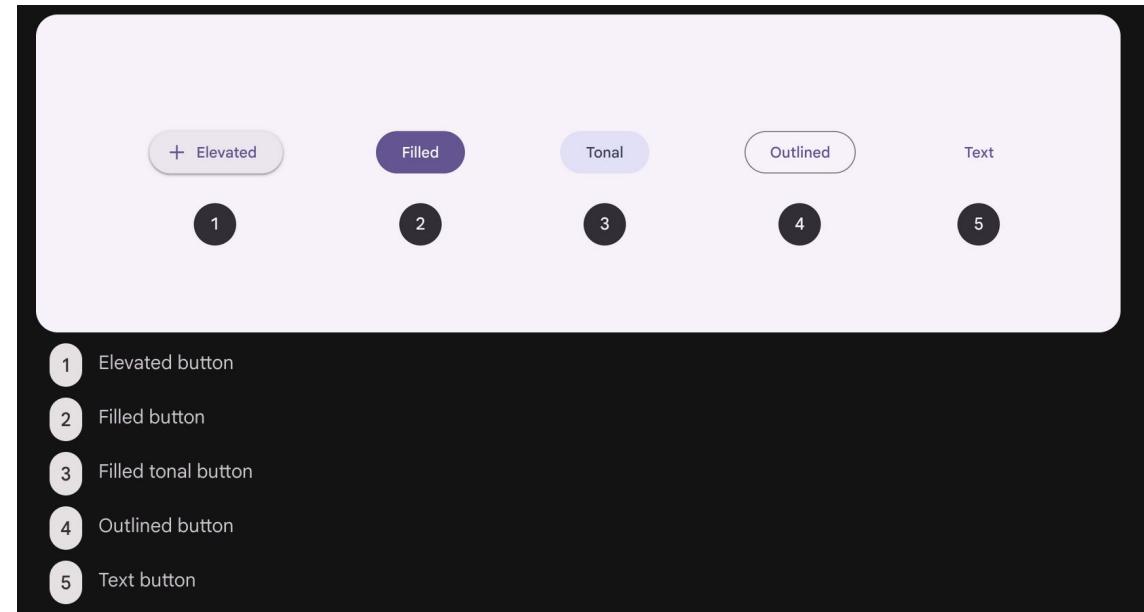
- Column
- Row
- Surface
- Scaffold
- Card



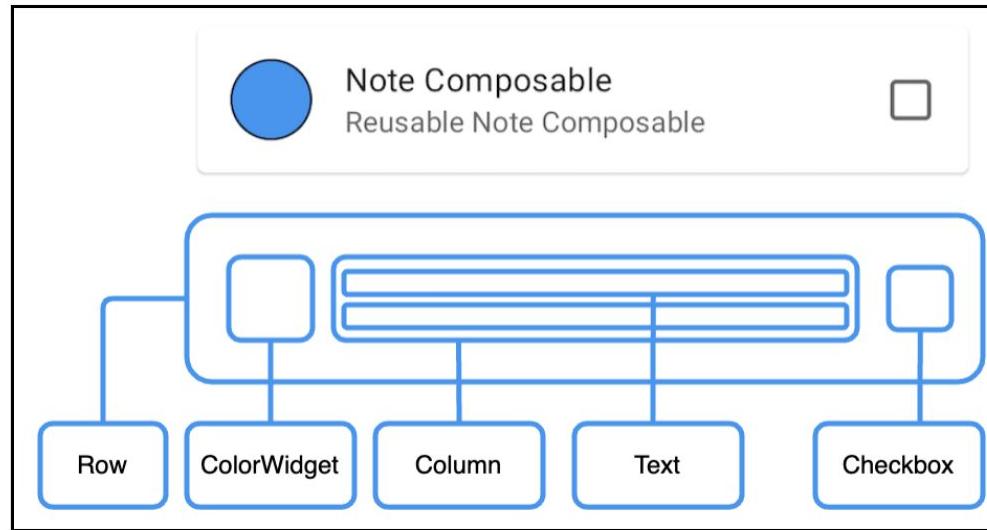
```
@Composable
inline fun Column(
    modifier: Modifier = Modifier,
    verticalArrangement: Arrangement.Vertical = Arrangement.Top,
    horizontalAlignment: Alignment.Horizontal = Alignment.Start,
    content: @Composable ColumnScope.() -> Unit
)
```

Exemples de composables

- Text
- Box
- Button
- Icon
- Image
- TextField
- ProgressBar
- CheckBox
- FAB



Hiérarchie de composables



Recomposition

- Les composables sont par défaut sans état (stateless)
 - Ils reçoivent les données dont ils ont besoin et les méthodes pour réagir aux actions en tant que arguments
- Chaque fois que les données dont elles dépendent change il y a une phase de recomposition => les composables impliqués sont recalculés et une nouvelle version de l'UI est produite

Argument d'un composable

```
@Composable
fun MyText() {
    Text(text = stringResource(id = R.string.jetpack_compose),
        fontStyle = FontStyle.Italic, // 1
        color = colorResource(id = R.color.colorPrimary), // 2
        fontSize = 30.sp, // 3
        fontWeight = FontWeight.Bold // 4
    )
}
```

Etat de l'UI



Etat d'un composable

- Si on définit une variable locale dans un composable pour stocker l'état interne de ce composable alors lors de la recomposition la valeur stockée est remise à la valeur de départ
 - => on perd la mémoire
- Solution : il faut explicitement définir la variable comme
 - Représentant un état avec stateOf ou mutableStateOf
 - Être persistante : avec remember

Cycle de vie d'un composable

- Les fonctions modulables ont leur propre cycle de vie, qui est indépendant de celui de l'activité. Son cycle de vie est constitué des événements suivants : accéder à la composition, exécuter zéro ou plusieurs recompositions, puis quitter la composition.
- Pour que Compose puisse suivre et déclencher une recomposition, il doit savoir à quel moment l'état a changé. Pour indiquer à Compose qu'il doit suivre l'état d'un objet, celui-ci doit être de type State ou MutableState. Le type State est immuable et peut seulement être lu. Un type MutableState est modifiable, et autorise les lectures et les écritures.

```
var revenue by remember { mutableStateOf(0) }
```

Ajouter un état

```
@Composable
fun MyTextField() {
    val textValue = remember { mutableStateOf("") }

    TextField(
        value = textValue.value,
        onValueChange = {
            textValue.value = it
        },
        label = {}
    )
}
```

```
@Composable
fun TextField(
    value: TextFieldValue,
    onValueChange: (String) -> Unit,
    label: @Composable () -> Unit,
    keyboardOptions: KeyboardOptions = KeyboardOptions.Default,
    keyboardActions: KeyboardActions = KeyboardActions(),
    ...
)
```

State hoisting

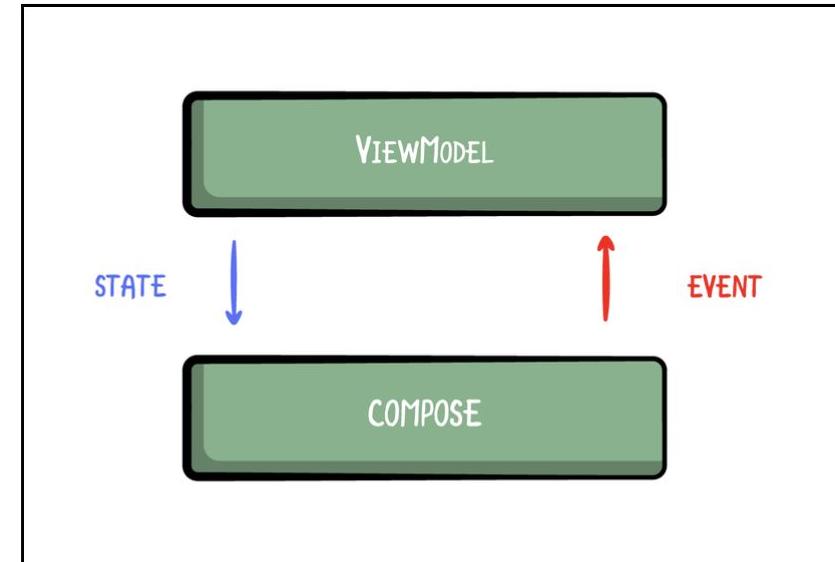
- **state hoisting** = “remonter l’état”.
 - On choisit de définir l’état d’un composant dans un autre composant “parent” plutôt que de le garder localement dans le composant lui-même.
 - Ensuite, on passe cet état et une fonction pour le mettre à jour en tant que paramètres aux composants enfants.
- Favorise une meilleure séparation des préoccupations (SOC) et permet à plusieurs composants de partager et manipuler le même état.

Programmation Mobile

- Partie II : Android Moderne
 - 1. Gradle
 - 2. Activité
 - 3. UI avec Jetpack Compose
 - 4. **Couches**
 - 5. Navigation
 - 6. Repository

UDF : Unique Direction of Flow

- Principe de séparation entre UI et Données
- Il n'y a qu'une seule direction :
 - Pour les changements d'état
 - Pour les mises à jour de l'UI
- Les UI observent les états

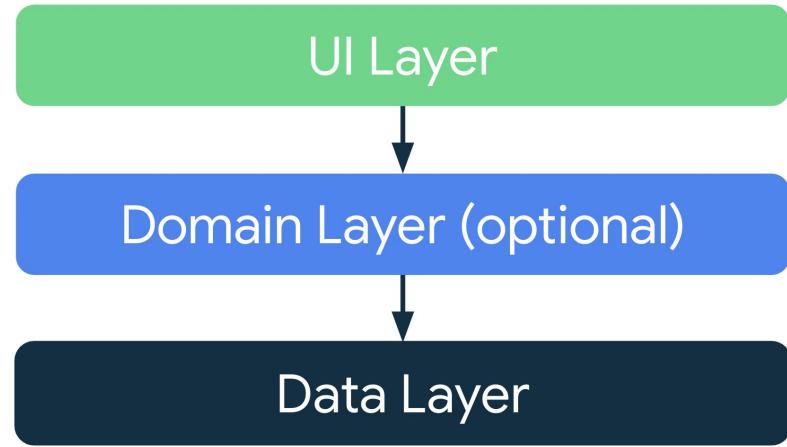


SSOT

- Single Source of Truth
- La SSOT est le *propriétaire* de ces données, et elle seule peut les modifier ou les muter. Pour ce faire, la SSOT expose les données avec un type immuable et, pour les modifier, elle expose des fonctions ou reçoit des événements que d'autres types peuvent appeler.

Architecture d'Application Recommandée

- Chaque application doit comporter au moins deux couches
 - La couche de *l'interface utilisateur* qui affiche les données d'application à l'écran
 - La *couche de données* qui contient la logique métier de votre application et expose les données de l'application
- On peut ajouter une couche appelée couche du domaine pour simplifier les interactions



Architecture moderne

- Cette *architecture d'application moderne* 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
 - Bonnes pratiques pour l'injection de dépendances

Injection de dépendance

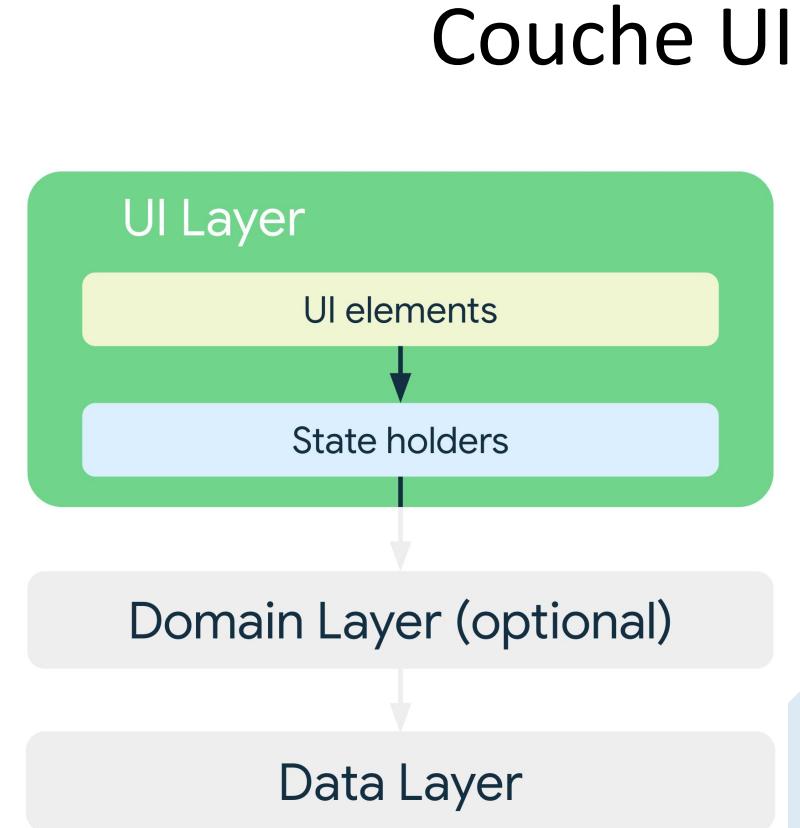
Sans injection de dépendance

```
● ● ●  
1 class Voiture {  
2     private val moteur = Moteur()  
3     fun démarre() {  
4         moteur.démarre()  
5     }  
6 }  
7  
8 fun main(args: Array) {  
9     val voiture = Voiture()  
10    voiture.start()  
11 }  
12 }
```

Avec injection de dépendance

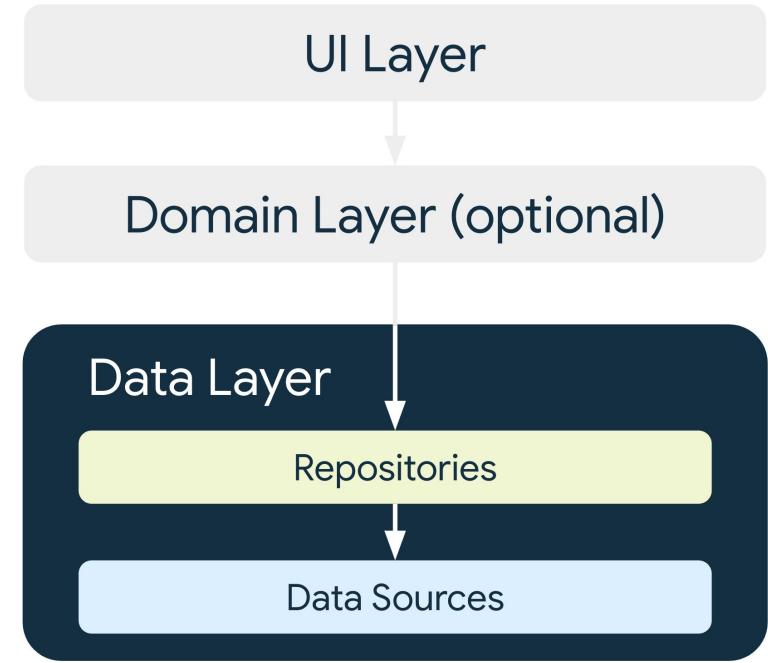
```
● ● ●  
1 class Voiture(private val moteur : Moteur) {  
2     fun démarre() {  
3         moteur.démarre()  
4     }  
5 }  
6  
7 fun main(args: Array) {  
8     val moteur = Moteur()  
9     val voiture = Voiture(moteur)  
10    voiture.start()  
11 }
```

- contient les deux éléments suivants :
 - Éléments d'interface utilisateur qui affichent les données à l'écran à l'aide fonctions composables
 - Conteneurs d'état à l'aide de classes ViewModel :
 - Contiennent les données
 - Les exposent à l'interface utilisateur
 - Gèrent la logique



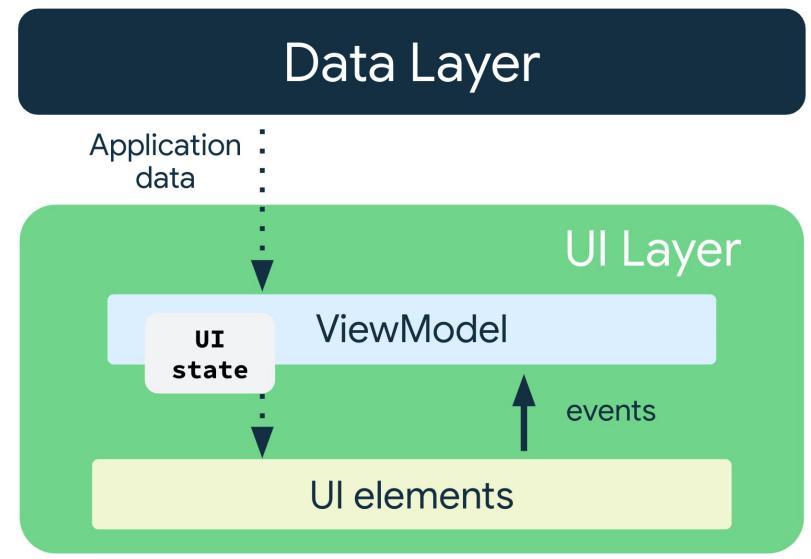
- contient la *logique métier* :
 - ce qui donne de la valeur à votre application
 - repose sur des règles qui déterminent la manière dont votre application crée, stocke et modifie les données.
- Constituée de **dépôts** (repositories) pouvant contenir une ou plusieurs **sources de données**

Couche des Données



- classes responsables de la production de l'état de l'interface utilisateur et contenant la logique nécessaire à cette tâche
- implémentation type est une instance d'un objet [ViewModel](#)

Conteneurs d'état

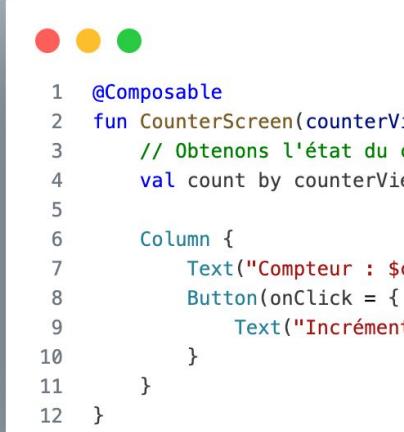


ViewModel

- Le ViewModel fait partie de l'architecture **MVVM** (Model-View-ViewModel) dans Android.
- Il est conçu pour conserver les données d'une interface utilisateur à travers les changements de configuration, comme les rotations d'écran, sans que les données soient perdues.
- Rôles :
 - Gérer et encapsuler l'état
 - Centraliser la logique métier hors de l'UI
 - Exposer l'état sous forme de State ou Flow avec LiveData

Exemple

```
1 class CounterViewModel : ViewModel() {  
2     // Pas besoin de "var" ici car mutable  
3     private var count = 0  
4     val count by mutableStateOf(count)  
5  
6     fun increment() {  
7         count++  
8     }  
9 }
```



```
1 class MainActivity : ComponentActivity() {  
2     override fun onCreate(savedInstanceState: Bundle?) {  
3         super.onCreate(savedInstanceState)  
4         enableEdgeToEdge()  
5         setContent {  
6             SimpleViewModelTheme {  
7                 Surface(modifier = Modifier.fillMaxSize()) {  
8                     val counterVM = CounterViewModel()  
9                     CounterScreen(counterVM)  
10                }  
11            }  
12        }  
13    }  
14 }
```

Délégation

- En Kotlin le mot clé **by** permet de **déléguer** certaines fonctionnalités à d'autres objets :
 - Délégation de propriété
 - Avec initialisation différée avec lazy (seulement quand elle est observée)
 - Avec un observable : on surveille et on réagit aux modifications de la propriété
 - Pour déléguer à un ViewModel
 - Délégation de classe
 - On dit qu'une classe implémente une interface en déléguant son implémentation à un autre objet

Délégation de propriété



```
1 val pi: Double by lazy {  
2     println("Calcul de pi...")  
3     3.14159  
4 }  
5  
6 fun main() {  
7     println(pi) // La première fois que pi est appelée, "Calcul de pi..." sera imprimé.  
8     println(pi) // Cette fois, pi est déjà initialisée, donc pas de recalcul.  
9 }
```



```
1 import kotlin.properties.Delegates  
2  
3 var name: String by Delegates.observable("No Name") { _, old, new ->  
4     println("Nom changé de $old à $new")  
5 }  
6  
7 fun main() {  
8     name = "Alice" // Affiche: Nom changé de No Name à Alice  
9     name = "Bob" // Affiche: Nom changé de Alice à Bob  
10 }
```

Délégation de classe

```
● ● ●  
1 interface Greeter {  
2     fun greet()  
3 }  
4  
5 class GreeterImpl : Greeter {  
6     override fun greet() {  
7         println("Hello from Greeter!")  
8     }  
9 }
```

```
● ● ●  
1 // Classe qui délègue Greeter à GreeterImpl  
2 class FriendlyPerson(greeter: Greeter) : Greeter by greeter  
3  
4 fun main() {  
5     val greeter = GreeterImpl()  
6     val person = FriendlyPerson(greeter)  
7     person.greet() // Affiche "Hello from Greeter!"  
8 }
```

Programmation Mobile

- Partie II : Android Moderne
 - 1. Gradle
 - 2. Activité
 - 3. UI avec Jetpack Compose
 - 4. Couches
 - 5. **Navigation**
 - 6. Repository

- Elément central de la navigation
 - Contient le graphe de Navigation
 - Propose des méthodes pour que l'app se déplace entre les destinations
 - Garde la trace des destinations visitées

NavController



```
1 //Création du composant de navigation
2 val navController = rememberNavController()
```

Créer le graphe des destinations

- Il y a 3 types de destinations :
 - Hosted : la destination recouvre totalement le host – pour la navigation entre écrans
 - Dialog : la destination se superpose à la précédente – pour les alertes et les formulaires
 - Activity : point de sortie du graphe de destination vers une nouvelle activité
- Création du Graphe
 - En créant le NavHost
 - Par programme en utilisant la méthode createGraph de NavController

- C'est un composable qui se place là où l'on veut naviguer entre destinations (typiquement un Scaffold)
- Arguments :
 - navController
 - startDestination
 - modifier

- Lambda :
 - Des appels répétés de la méthode composable() de NavGraphBuilder qui définissent les routes.
- Route
 - Une string correspondant à une destination ou n'importe quel type serializable
- navGraphBuilder.composable() :
 - Appelle le Composable destination
 - passe à chaque destination la méthode qui provoque la navigation vers une nouvelle destination
 - Méthode navigate de navController
 - A accès à la route destination

NavHost

- Historiquement les routes étaient des String sous la forme de URI
 - Il fallait récupérer dans le URI les informations sur les arguments
 - Source d'erreurs
 - Textes en dur

• State Safe Route

- Les routes sont représentées par des objets ou des classes serialisables
- Les attributs des objets sont les paramètres qui sont passés à la destination
- La destination récupère ces données avec la méthode toRoute du backStack

Exemple

```
1  enum class NavigationScreen {
2      Start,
3      Detail,
4      Summary
5  }
6
7  @Composable
8  fun Destination1() {}
9
10 @Composable
11 fun Destination2() {}
12
13 @Composable
14 fun Destination3() {}
```



```
1  @Composable
2  fun ExempleApp(
3      navController: NavHostController = rememberNavController()
4  ) {
5
6      Scaffold( ) {
7          innerPadding ->
8          NavHost(
9              navController = navController,
10             startDestination = NavigationScreen.Start.name,
11             modifier = Modifier
12                 .padding(innerPadding)
13         ) {
14             composable(route = NavigationScreen.Start.name) {
15                 Destination1(
16                     ...
17                 )
18             }
19             composable(route = NavigationScreen.Detail.name) {
20                 Destination2(
21                     ...
22                 )
23             }
24             composable(route = NavigationScreen.Summary.name) {
25                 Destination3(
26                     ...
27                 )
28             }
29         }
30     }
31 }
```

Programmation Mobile

- Partie II : Android Moderne
 1. Gradle
 2. Jetpack Compose
 3. UDF
 4. SSOT
 5. Navigation
 6. Repository

- Couche d'architecture utilisée pour gérer l'accès aux données.
 - crucial dans les architectures **MVVM** (Model-View-ViewModel) pour séparer la logique métier de la logique d'interface utilisateur.
- Sert de source unique de vérité (SSOT) pour les données provenant
 - D'une API distante
 - D'une BD locale
 - Du cache

Repository

- Responsabilités :
 - Abstraire la source des données
 - Présenter les données au reste de l'application
 - Centraliser les modifications apportées aux données
 - Résoudre les conflits entre plusieurs sources de données
 - Séparer des sources de données du reste de l'application
 - Contenir la logique métier

- Le ViewModel est responsable de l'état de l'UI
- Il utilise le repository pour obtenir les données et expose ces données à l'UI via des objets State ou Flow

Repository et ViewModel

```
● ● ●  
1 import androidx.lifecycle.ViewModel  
2 import androidx.lifecycle.viewModelScope  
3 import kotlinx.coroutines.flow.MutableStateFlow  
4 import kotlinx.coroutines.flow.asStateFlow  
5 import kotlinx.coroutines.launch  
6  
7 class UserViewModel(private val repository: UserRepository) : ViewModel() {  
8  
9     private val _users = MutableStateFlow<List<User>>(emptyList())  
10    val users = _users.asStateFlow()  
11  
12    init {  
13        fetchUsers()  
14    }  
15  
16    private fun fetchUsers() {  
17        viewModelScope.launch {  
18            _users.value = repository.getUsers()  
19        }  
20    }  
21 }
```

Coroutines

- Les Coroutines sont un Design Pattern de concurrence pour simplifier l'écriture de code qui s'exécute de façon asynchrone
 - alternatives aux threads
 - ajoutés à Kotlin dans la version 1.3
- Utilité dans Android
 - gérer les tâches qui prennent du temps (accès à un BD, chargement d'images) sans bloquer le UI
- Solution recommandée pour la programmation asynchrone dans Android
- <https://developer.android.com/kotlin/coroutines?hl=fr#kts>

Fonctionnalités des coroutines

- **Légèreté** : vous pouvez exécuter de nombreuses coroutines sur un seul thread
- **Moins de fuites de mémoire**:
- **Résiliation intégrée**
- **Intégration Jetpack** : de nombreuses bibliothèques Jetpack incluent des extensions compatibles avec toutes les coroutines.
- La classe `ViewModel` inclut un ensemble d'extensions qui fonctionnent directement avec les coroutines

```
dependencies {  
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")  
}
```

Comment utiliser les coroutines

- On a besoin
 - d'un builder de coroutines : des fonctions comme *async* ou *launch*
 - d'une portée de coroutine (*coroutine scope*) dans lequel la coroutine va travailler et qui détermine sa durée de vie
 - de fonctions dont l'exécution peut être suspendue (*suspend function*) qui sont les uniques fonctions que l'on peut appeler dans le corps de la coroutine
 - d'un *dispatcher* qui contrôle le type de thread dans lequel la coroutine va être exécutée (par exemple Dispatcher.IO pour les opérations d'entrée/sortie)

Exemple de coroutine

```
Main.kt x
1 package fr.unice.miage.donati
2 import kotlinx.coroutines.*
3
4 suspend fun downloadData() : String {
5     delay( timeMillis: 2000L )
6     return "Data downloaded in Kotlin Coroutine"
7 }
8
9 fun main() {
10     runBlocking {
11         launch {
12             println("download started")
13             val data = downloadData()
14             println("data: $data")
15             println("download finished")
16         }
17         println("Doing other works during the download")
18     }
19 }
```

```
Doing other works during the download
download started
data: Data downloaded in Kotlin Coroutine
download finished

Process finished with exit code 0
```

Coroutine dans un ViewModel



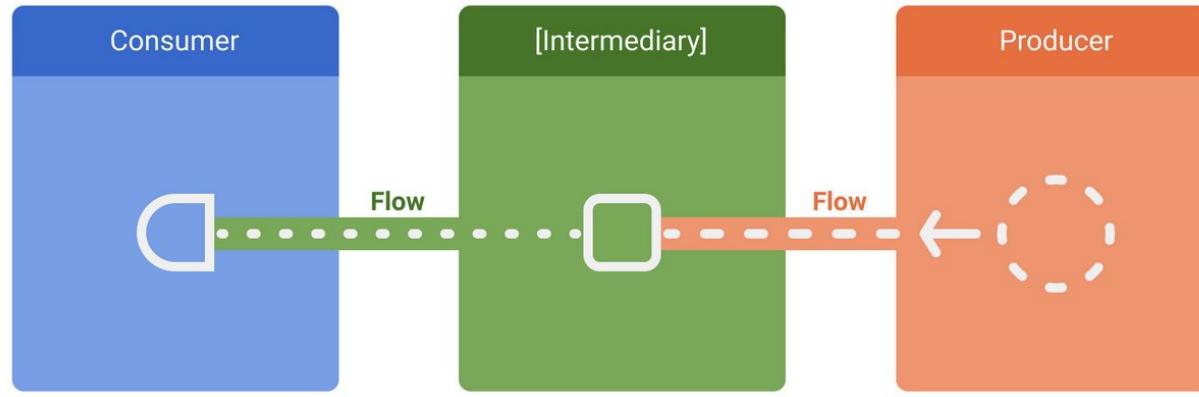
```
1 class LoginViewModel(  
2     private val loginRepository: LoginRepository  
3 ): ViewModel() {  
4  
5     fun login(username: String, token: String) {  
6         // Create a new coroutine to move the execution off the UI thread  
7         viewModelScope.launch(Dispatchers.IO) {  
8             val jsonBody = "{ username: \"$username\", token: \"$token\"}"  
9             loginRepository.makeLoginRequest(jsonBody)  
10        }  
11    }  
12 }
```

Flux Kotlin

- Dans les coroutines, un *flux* est un type qui peut émettre plusieurs valeurs de manière séquentielle, par opposition aux *fonctions de suspension* qui ne renvoient qu'une seule valeur.
- Par exemple, vous pouvez utiliser un flux pour recevoir les mises à jour en direct d'une base de données.
- Par exemple, `Flow<Int>` est un flux qui émet des valeurs entières.
- <https://developer.android.com/kotlin/flow?hl=fr>

Concepts dans le flux

- Un flux est très semblable à un **Iterator** qui produit une séquence de valeurs, mais qui utilise des fonctions de suspension pour produire et consommer des valeurs de manière asynchrone.
- Trois entités sont impliquées :
 - un producteur qui produit des données qui sont ajoutées au flux
 - un intermédiaire (facultatif) qui peut modifier (ou filtrer) en cours de route les valeurs émises par le producteur
 - un consommateur qui utilise les données



- On utilise la fonction [flow](#) de kotlin pour créer le flux
- Le constructeur flow est automatiquement exécutée dans une coroutine
- A l'intérieur du flux la fonction emit() émet des données dans le flux

Créer un flux



```
1  class NewsRemoteDataSource(
2      private val newsApi: NewsApi,
3      private val refreshIntervalMs: Long = 5000
4  ) {
5      val latestNews: Flow<List<ArticleHeadline>> = flow {
6          while(true) {
7              val latestNews = newsApi.fetchLatestNews()
8              emit(latestNews) // Emits the result of the request to the flow
9              delay(refreshIntervalMs) // Suspends the coroutine for some time
10         }
11     }
12 }
13
14 // Interface that provides a way to make network requests with suspend functions
15 interface NewsApi {
16     suspend fun fetchLatestNews(): List<ArticleHeadline>
17 }
```

- les intermédiaires peuvent utiliser des opérations sur les flux pour modifier le flux de données sans le consommer
- Ici on va transformer le dataSource en Repository en filtrant en fonction des préférences de l'utilisateur

Modifier le flux



```
1 class NewsRepository(
2     private val newsRemoteDataSource: NewsRemoteDataSource,
3     private val userData: UserData
4 ) {
5     val favoriteLatestNews: Flow<List<ArticleHeadline>> =
6         newsRemoteDataSource.latestNews
7             // Intermediate operation to filter the list of favorite topics
8             .map { news -> news.filter { userData.isFavoriteTopic(it) } }
9             // Intermediate operation to save the latest news in the cache
10            .onEach { news -> saveInCache(news) }
11    }
12 }
```

- Pour obtenir toutes les valeurs du flux à mesure qu'elles sont émises, utilisez collect
- C'est une fonction suspendable donc on a intérêt à la lire dans une coroutine

Lire un flux



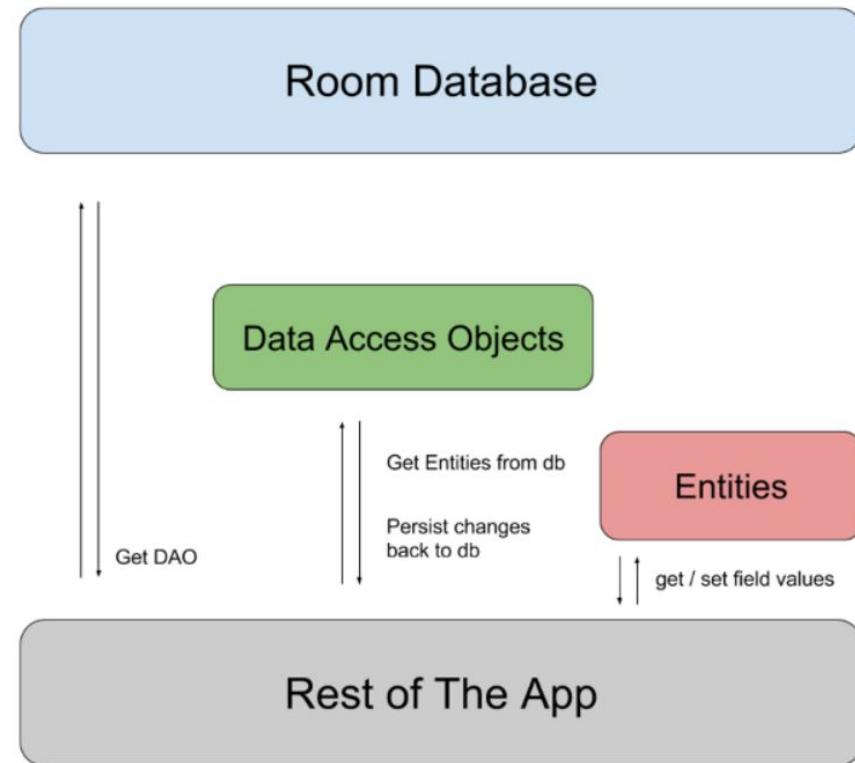
```
1  class LatestNewsViewModel(
2      private val newsRepository: NewsRepository
3  ) : ViewModel() {
4
5      init {
6          viewModelScope.launch {
7              // Trigger the flow and consume its elements using collect
8              newsRepository.favoriteLatestNews.collect { favoriteNews ->
9                  // Update View with the latest favorite news
10             }
11         }
12     }
13 }
```

- Persistence : sauvegarde locale des données
- Room est une librairie Jetpack qui offre une couche d'abstraction au dessus de SQLite

```
dependencies {  
    val room_version = "2.6.1"  
  
    implementation("androidx.room:room-runtime:$room_version")  
  
    // If this project uses any Kotlin source, use Kotlin Symbol Processing (KSP)  
    // See Add the KSP plugin to your project  
    ksp("androidx.room:room-compiler:$room_version")  
  
    // If this project only uses Java source, use the Java annotationProcessor  
    // No additional plugins are necessary  
    annotationProcessor("androidx.room:room-compiler:$room_version")  
  
    // optional - Kotlin Extensions and Coroutines support for Room  
    implementation("androidx.room:room-ktx:$room_version")  
  
    // optional - RxJava2 support for Room  
    implementation("androidx.room:room-rxjava2:$room_version")  
  
    // optional - RxJava3 support for Room  
    implementation("androidx.room:room-rxjava3:$room_version")  
  
    // optional - Guava support for Room, including Optional and ListenableFuture  
    implementation("androidx.room:room-guava:$room_version")
```

Composants de Room

- Trois principaux composants
 - la classe database qui représente la BD
 - les Data Entities qui représentent les tables dans la BD
 - le DAO (Data Access Object) qui offre les méthodes que la app peut appeler pour accéder aux données : query, update, insert, delete



Entités

- L'entité doit contenir des champs pour chaque colonne de la table
 - Doit avoir au moins une clé primaire
- L'annotation `@ColumnInfo` permet de changer le nom de la colonne par rapport au nom de l'attribut
- <https://developer.android.com/training/data-storage/room/defining-data>

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

DAO

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

BD

- L'annotation `@Database` doit citer toutes les entités de la base
- la classe doit être abstraite et hériter de `RoomDatabase`
- Pour toute classe de DAO associée à la BD il faut une méthode abstraite sans arguments qui renvoie une instance de la classe de DAO

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Utilisation

- Après avoir défini les entités, les DAO et la BD, le code suivant crée une instance de la BD

```
val db = Room.databaseBuilder(  
    applicationContext,  
    AppDatabase::class.java, "database-name"  
).build()
```

- A partir de la bd on récupère la DAO via la méthode abstraite et ensuite on fait les requêtes

```
val userDao = db.userDao()  
val users: List<User> = userDao.getAll()
```