# Développement Android

Christophe Nasica
nasica@goodbarber.com

# Objectifs du cours

Comprendre les architectures modernes

Maîtriser les outils Android modernes

Adopter les bonnes pratiques

Optimiser les performances de l’application

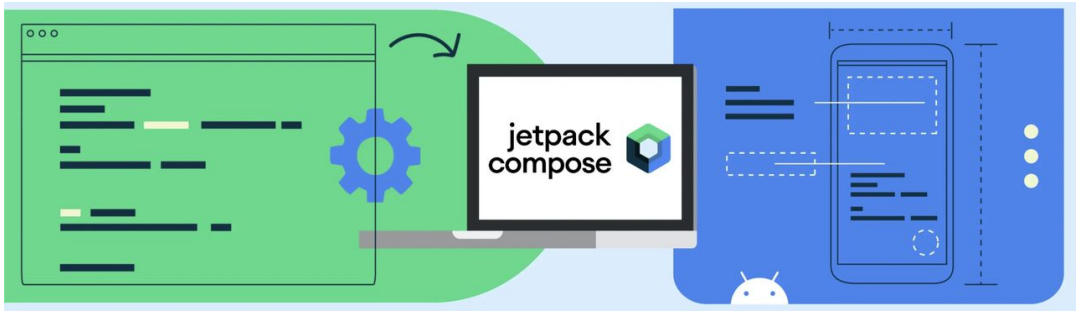
Développer une application prête pour la production

# Évolution du langage : Java vers Kotlin

- Écriture plus concise (- de code + de productivité)
- Performances similaires (JVM)
- Null safety revue entièrement
- Multithreading simplifié

# Évolution des outils : Views vs Compose

- Impératif (Views) :
  - Définir une suite d'étapes pour manipuler l'UI => suite d'appels à des méthodes sur le composants.
- Déclaratif (Compose) :
  - Décrire à quoi ressemble l'UI pour un état donné. Compose s'occupe de mettre à jour l'UI lorsque l'état change.



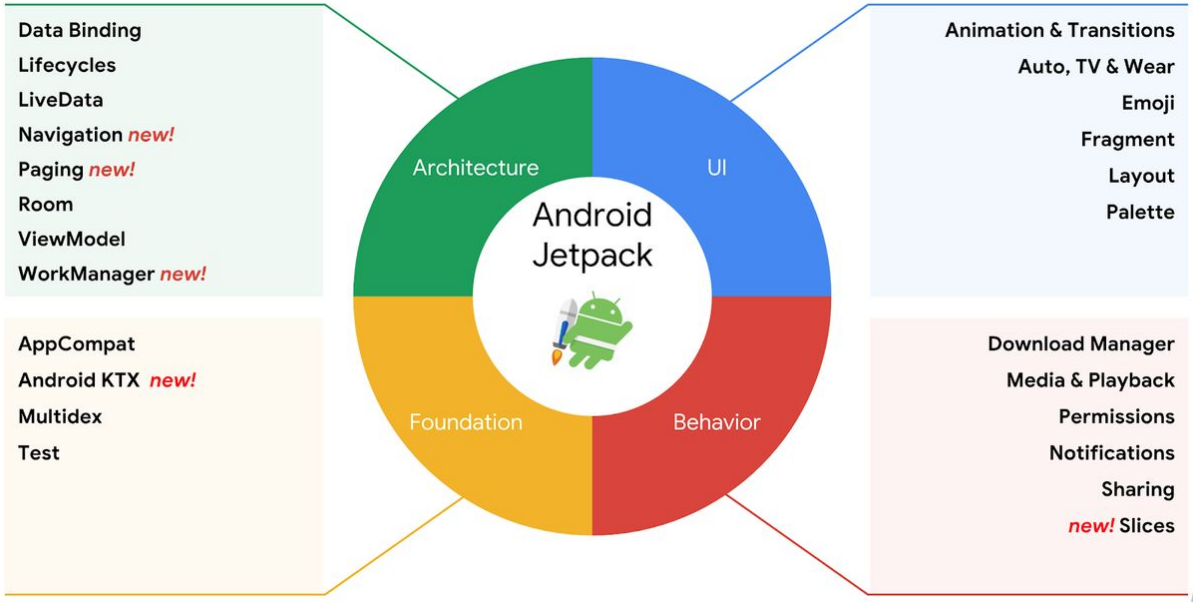
# Évolution des outils : Compose

- Efficience : Limite le nombre de redraw (recompose)
- Fortement réutilisable : composables
- Langage unique pour la logique métier et l'UI (Kotlin)
- Popularité grandissante (guides, tutoriels, documentations...)

# Jetpack et l’écosystème Android moderne

Des bibliothèques clés :

- ViewModel
- LiveData
- Flow
- Room
- Paging
- Navigation

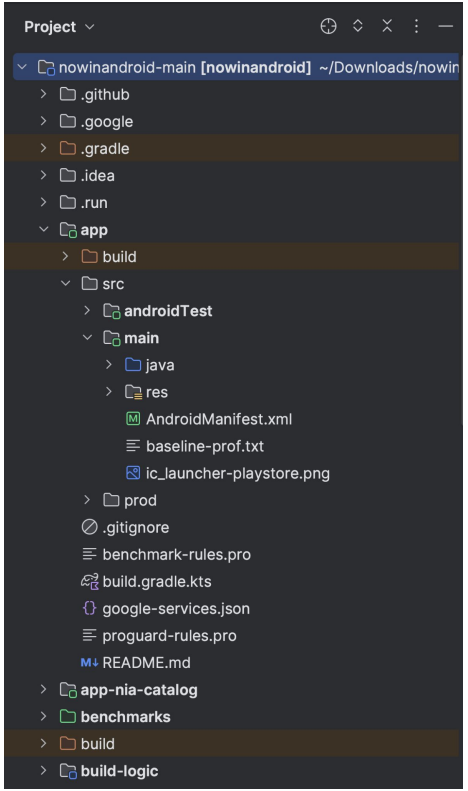


# Rappels : Interface et outils Android Studio

Profitez en pour télécharger ou mettre à jour votre Android Studio ;)

# Rappels : Les composantes d’un projet Android

- AndroidManifest
- build.gradle
- /res
- proguard
- gradle.properties



# Rappels Kotlin

- Les bases : val, var, fun, if, when, for, String
- class vs data class
- collections : listOf, mutableListOf
- null safety
- lambdas
- fonctions d’extension

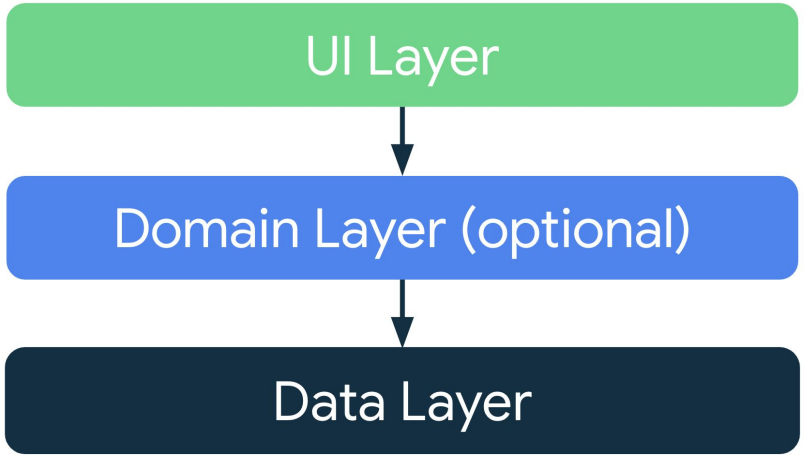
# Architecture moderne

“Séparation des pouvoirs”

Piloter l’UI via des data models

Source unique de vérité

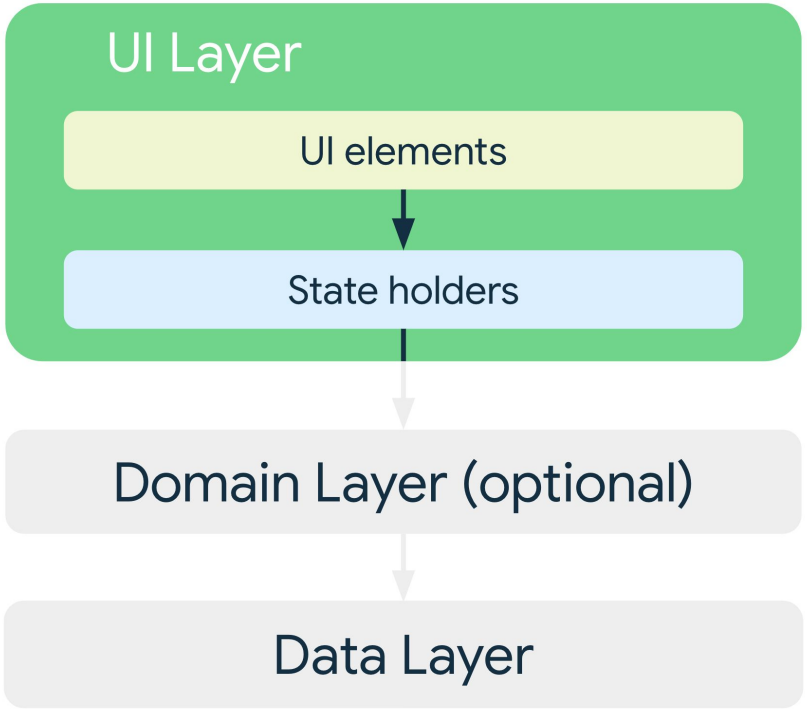
“Unidirectional Data Flow”



# Architecture : Couche UI

UI : Views ou Compose

State holder : ViewModel



# Architecture : Couche UI - State holder

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

State

```kotlin
class NewsViewModel(...) : ViewModel() {

    val uiState: NewsUiState = …
}
```

State holder

UI

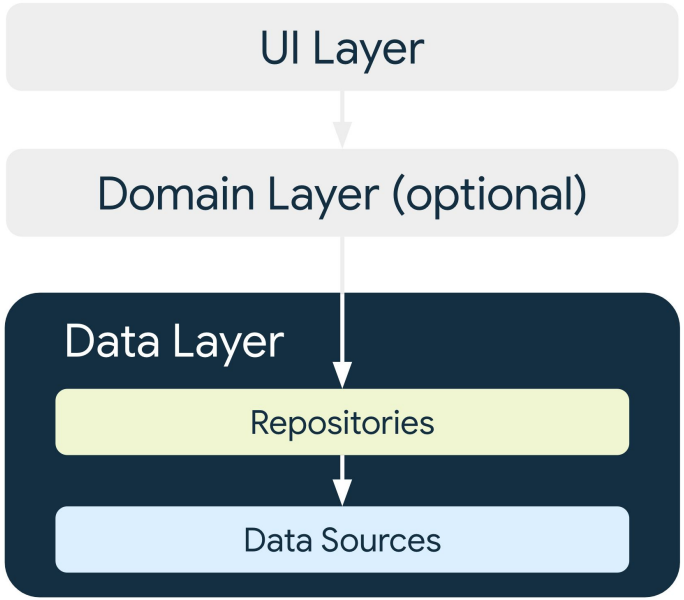
```kotlin
@Composable
fun LatestNewsScreen(
    modifier: Modifier = Modifier,
    viewModel: NewsViewModel = viewModel()
) {
    Box(modifier.fillMaxSize()) {

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

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

# Architecture : Couche Data

Il s’agit de la logique métier.

Rôles :

- Exposer la donnée au reste de l’application
- Centraliser les changements de donnée
- Abstraction des sources de donnée



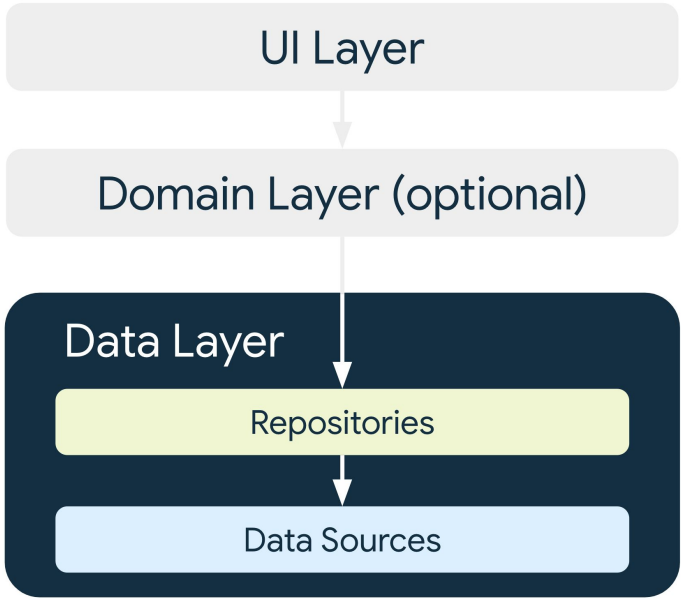
# Architecture : Couche Data

Les données exposées doivent être immuables

Couche data = source de vérité

Exemples d’opérations :

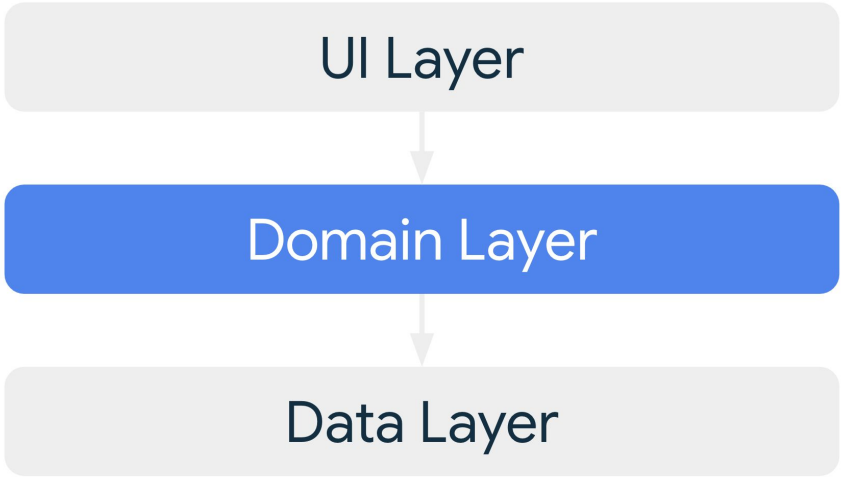
- Requêtes réseau
- Cache



# Architecture : Couche Domaine (optionnelle)

Intérêt :

- Simplifier une logique métier complexe
- Réutilisation de code



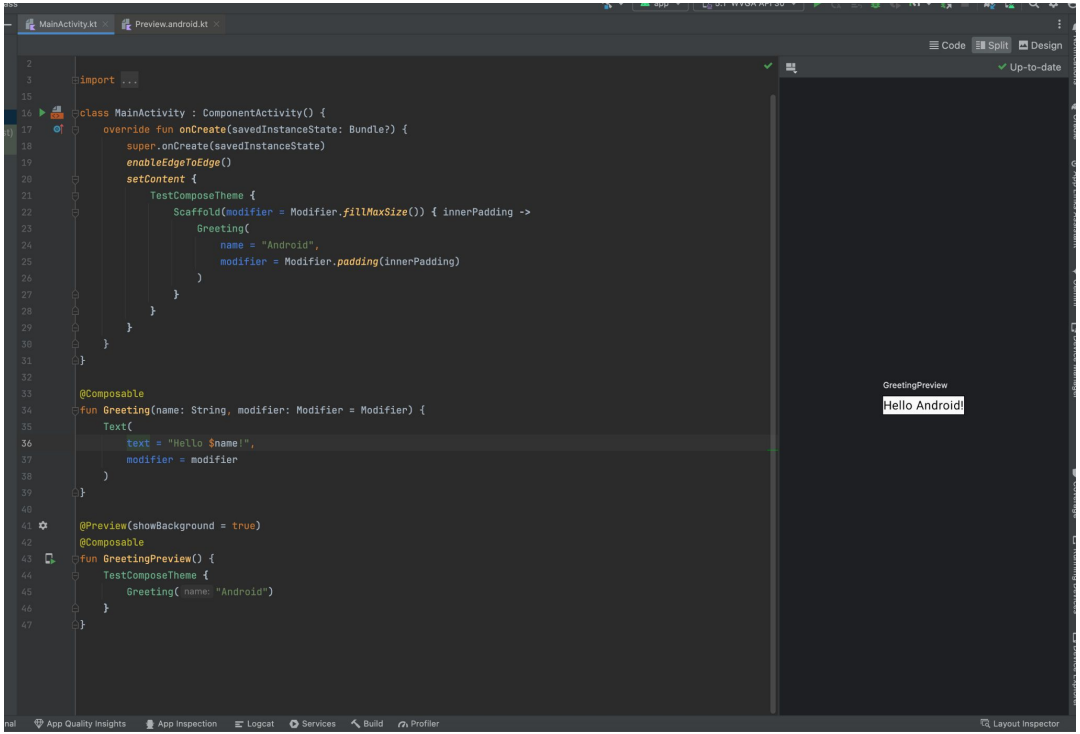
# Compose : @Composable

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

# Compose : @Composable

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

# Compose : @Preview



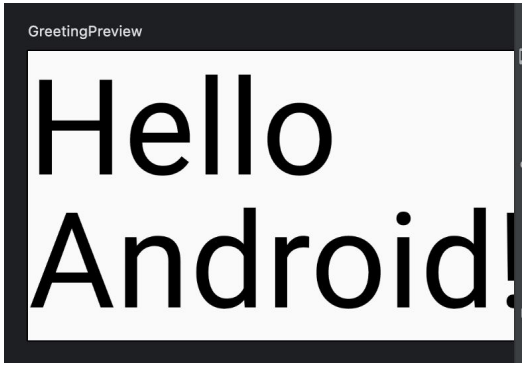
# Compose : Convention

```
// Do: This function is a descriptive PascalCased noun as a visual UI element
@Composable
fun FancyButton(text: String) {}


// Do: This function is a descriptive PascalCased noun as a non-visual element
// with presence in the composition
@Composable
fun BackButtonHandler() {}


// Don't: This function is a noun but is not PascalCased!
@Composable
fun fancyButton(text: String) {}


// Don't: This function is PascalCased but is not a noun!
@Composable
fun RenderFancyButton(text: String) {}


// Don't: This function is neither PascalCased nor a noun!
@Composable
fun drawProfileImage(image: ImageAsset) {}
```

# Compose : Text et preview

```
@Composable
fun Greeting(name: String, modifier:
Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        fontSize = 100.sp,
        modifier = modifier
    )
}
```

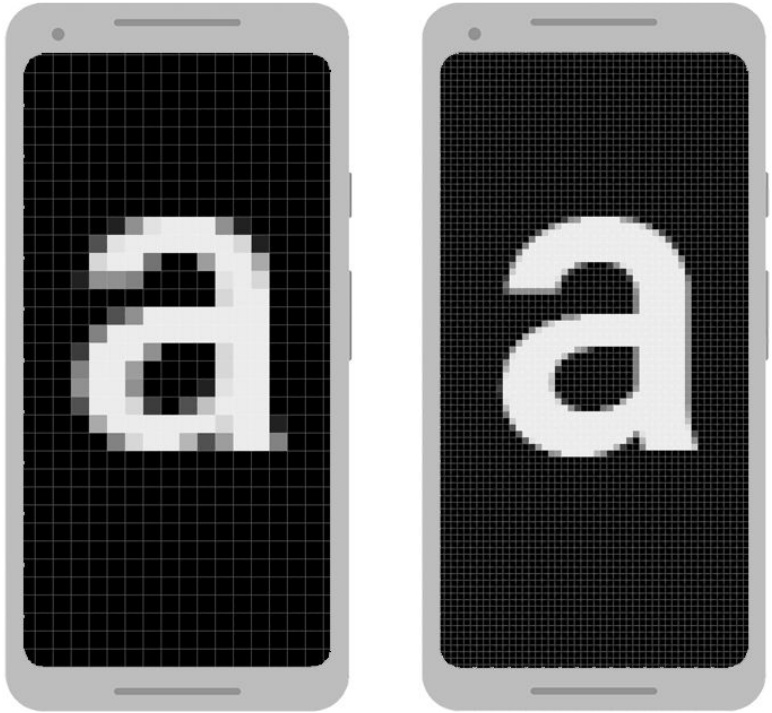

```
@Composable
fun Greeting(name: String, modifier:
Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        fontSize = 100.sp,
        lineHeight = 100.sp,
        modifier = modifier
    )
}
```



# Rappels : Densité d’écran

Deux écrans de même taille ne veut pas dire même nombre de px :

- Nécessité d’utiliser une unité intermédiaire au pixel

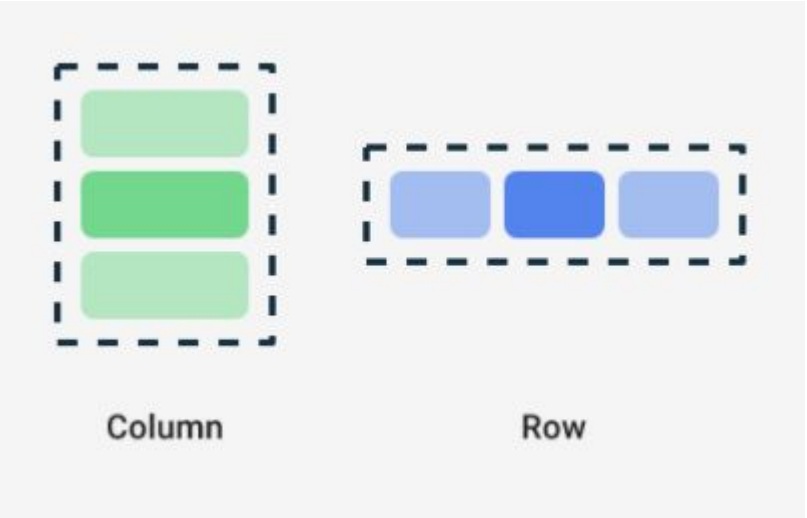


# Compose : Les conteneurs de base

Trois conteneurs de base :

- Box
- Column
- Row

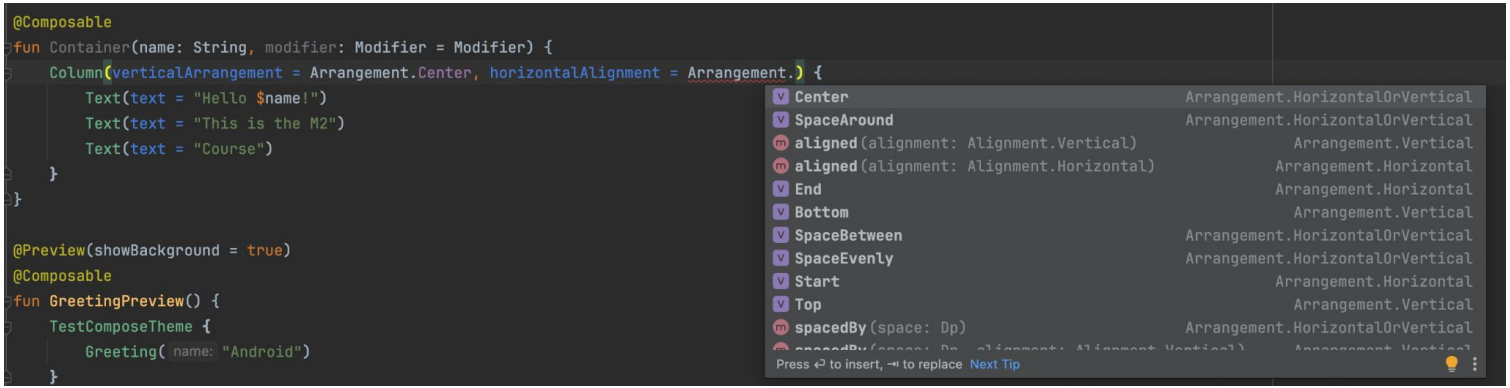
```
Column {
    Text(text = "Hello $name!")
    Text(text = "This is the M2")
    Text(text = "Course")
}
```



# Compose : Arrangement

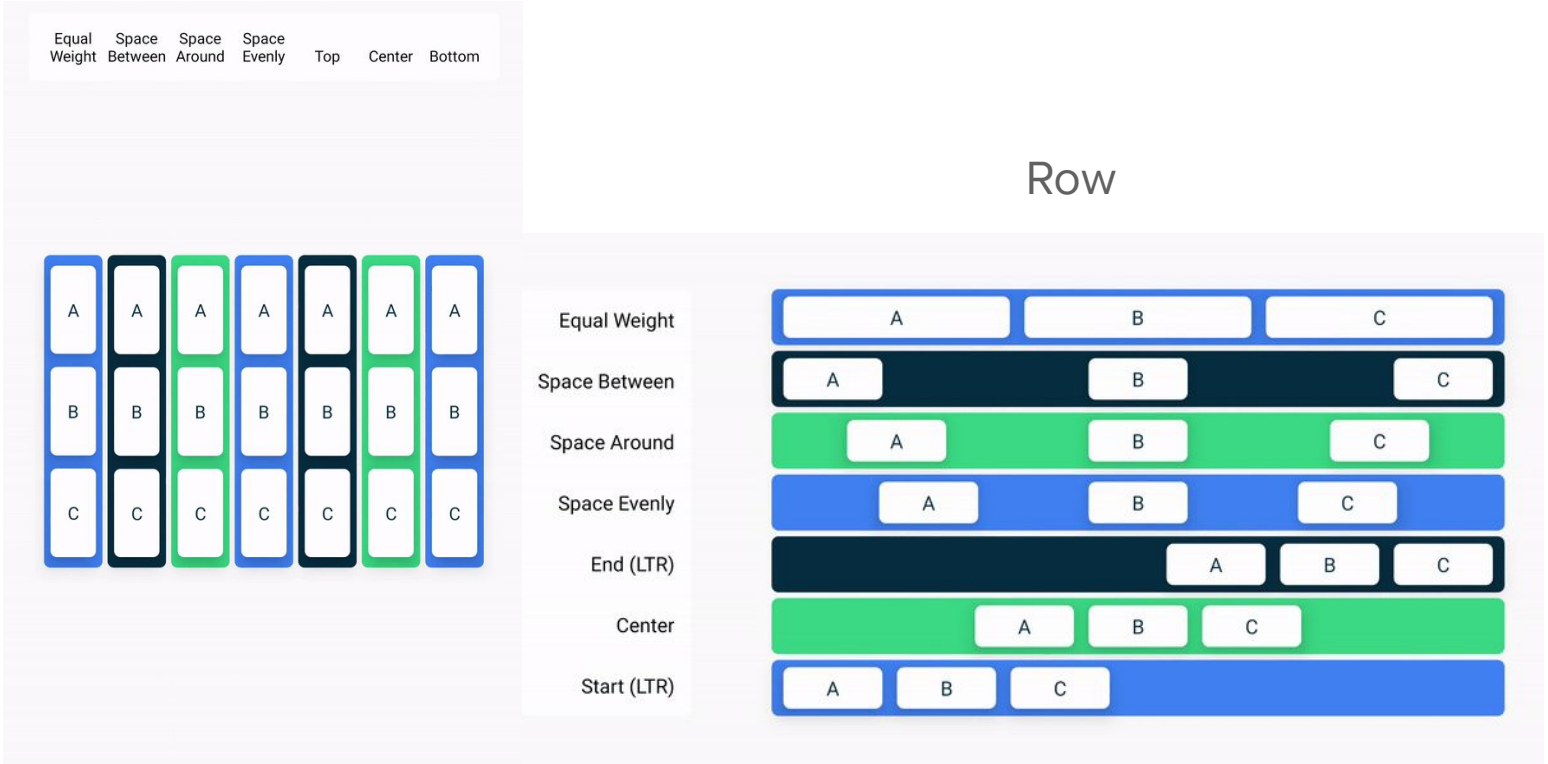
**verticalArrangement** et
**horizontalArrangement** utilisables sur :

- Column
- Row



# Compose : Arrangement

Column

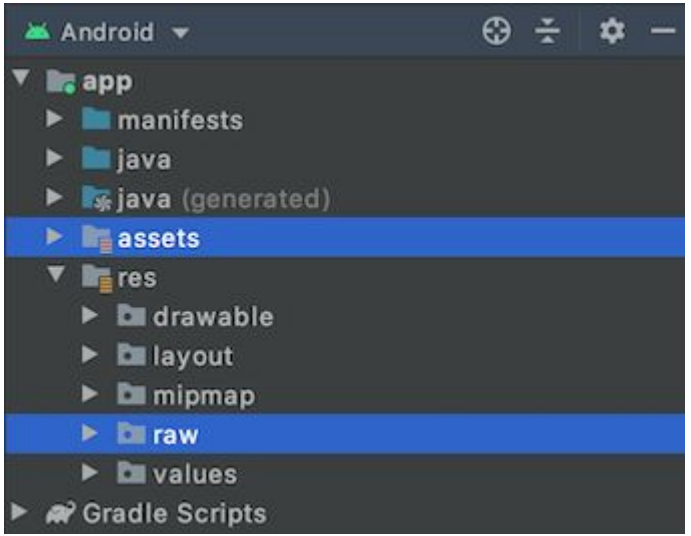



Row

# Quelques bonnes pratiques

Utiliser les fichiers ressources comme :

- string.xml
- dimen.xml



# Compose : Listes

Problématique pour les longues listes :

Column = problème de performance sur liste longue

```
@Composable
fun LongList(list: List<Item>) {
    Column {
        list.forEach { item ->
            Item(item = item)
        }
    }
}
```

LazyColumn = performant sur liste longue car recyclage

```
@Composable
fun LazyList(items: List<Item>)
{
    LazyColumn {
        items(items) { item ->
            Item(item = item)
        }
    }
}
```

# Compose : Listes

Chaque composant permettant d’afficher une liste existe également en version “Lazy” (sauf Box) :

- LazyRow
- LazyColumn
- Lazy[Vertical | Horizontal]Grid
- Lazy[Vertical | Horizontal]StaggeredGrid

# Compose : Listes

En reprenant l'exemple suivant :

```
@Composable
fun LazyList(items: List<Item>)
{
    LazyColumn {
        items(items) { item ->
            Item(item = item)
        }
    }
}
```

Par défaut gestion des listes basée sur la position
=> **potentiel problème**

# Compose : Listes

Fournir une “key” basée sur une valeur unique aide à résoudre le problème :

```
@Composable
fun LazyList(items: List<Item>) {
   LazyColumn {
       items(items = items, key = { item -> item.id }) {
item ->
           Item(item = item)
       }
   }
}
```

Note : Le paramètre key doit être d’un type supporté par la classe **Bundle** (primitives, enums, Parcelable, etc)

# Compose : Gestion d’état

Rappel : Compose = déclaratif

Notion de recomposition :

- mot clé **remember** : persistance de la donné pendant les recompositions
- Syntaxe : `var value by remember { mutableStateOf(default) }`
- Attention aux objets mutables (mutableList) qui ne sont pas observables :

```
@Composable
fun TestMutable(items: MutableList<Item>) {
   val its by remember {
mutableStateOf(items) }
}
```

```
@Composable
fun TestNonMutable(items:
MutableList<Item>) {
   val its by remember {
mutableStateOf(items.toList()) }
}
```

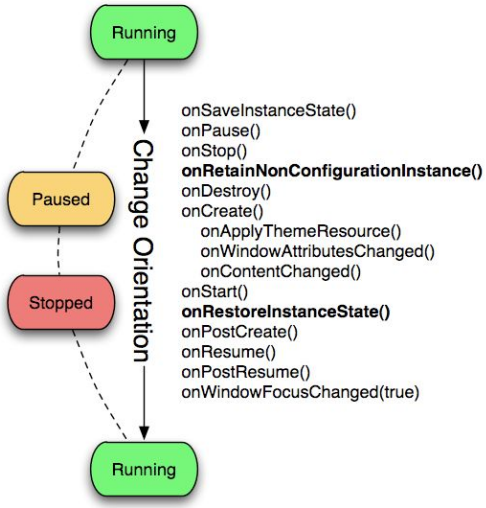
# Compose : Restauration d’état

Cas des changements de configuration = rotation de l’écran

Variante à remember : rememberSaveable

- Permet la “survie” de l’état lors des rotations d’écran
- Cas d’utilisation : Formulaire utilisateur, préférences utilisateur…

Rappels : Cycle de vie + rotation



# Coroutines : Bases

- Permet le traitement asynchrone de tâches
- Plusieurs coroutines par thread
- Possibilité de les annuler
- Doivent être exécutées dans un **scope**

Appel à doLogin = bloquant

```
class LoginViewModel(): ViewModel() {
    fun makeLoginApiRequest(jsonBody: String) {
        // *Server request*
    }

    fun doLogin(username: String, password: String) {
        val jsonBody = "{ username: \"$username\",
password: \"$password\"}"
        makeLoginApiRequest(jsonBody)
    }
}
```

Appel à doLogin = non bloquant

```
class LoginViewModel(): ViewModel() {
    fun makeLoginApiRequest(jsonBody: String) {
        viewModelScope.launch(Dispatchers.IO) {
            // *Server request*
        }
    }

    fun doLogin(username: String, password: String) {
        val jsonBody = "{ username: \"$username\", password:
\"$password\"}"
        makeLoginApiRequest(jsonBody)
    }
}
```

# Coroutines : withContext et suspend fun

- Notion “main-safe” = non bloquant pour l’UI
- Autre manière (recommandée) de définir notre méthode makeLoginApiRequest :

**suspend fun**

**withContext**

**scope (main thread)**

```
class LoginViewModel(): ViewModel() {
    suspend fun makeLoginApiRequest(jsonBody: String) {
        withContext(Dispatchers.IO) {
            // *Server request*
        }
    }

    fun doLogin(username: String, password: String) {
        val jsonBody = "{ username: \"$username\", password: \"$password\"}"
        viewModelScope.launch {
            makeLoginApiRequest(jsonBody)
        }
    }
}
```

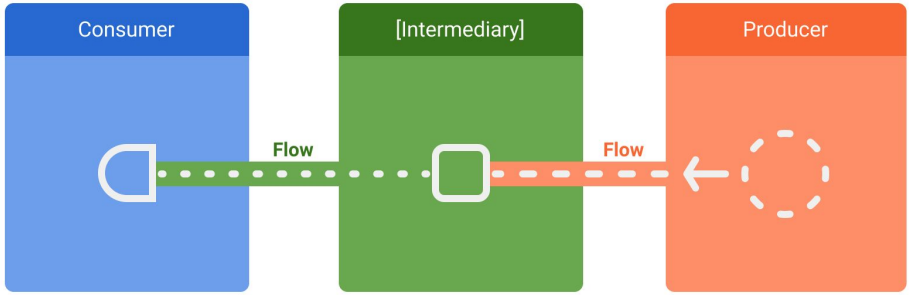
# Coroutines : CoroutineScope

- Un scope garde une trace de toutes les coroutines créées
- Certaines libs fournissent un scope (ViewModel = viewModelScope)
- Une coroutine peut être “cancel” à partir du scope
- Possibilité de créer un scope :

```
val myCoroutineScope = CoroutineScope(Dispatchers.IO)

myCoroutineScope.launch {
    // Some code
}
myCoroutineScope.cancel()
```

# Rappels : Kotlin Flows

Flow = type permettant d'émettre plusieurs valeurs de manière séquentielle




- Notion de cold vs hot pour les flows
  - Cold : commence à émettre lorsqu'il est collecté
  - Hot : émet même lorsqu'il n'y a aucun collecteur

# Rappels : Kotlin Flows - Flow

Caractéristiques :

- Cold
- Collecteur unique
- Émission de donnée en “one-shot”
- Cas d’utilisation : requête réseau, DB...

```
val simpleFlow: Flow<APIResponse> =  flow
{
   emit(APIResponse( 404, "Not found"))
}
```

# Rappels : Kotlin Flows - StateFlow

Caractéristiques :

- Hot
- Multiples collecteurs possibles
- Émission de la dernière donnée disponible (source of truth)
- Cas d’utilisation : State management (équivalent LiveData)

```
val stateFlow: MutableStateFlow<APIResponse> =  MutableStateFlow(APIResponse())
stateFlow.emit(APIResponse(200, “Success”))
```

Note : Bien que similaire au LiveData, le StateFlow diffère car a besoin d’une valeur par défaut

# Rappels : Kotlin Flows - SharedFlow

Caractéristiques :

- Hot
- Multiples collecteurs possibles
- Permet le replay d'événements passés
- Cas d’utilisation : Broadcast d’évènements (notifications, évènements de navigation, suivi parcours utilisateur...)

```
val sharedFlow: MutableSharedFlow<APIResponse> =  MutableSharedFlow()
sharedFlow.emit(APIResponse(200, “Success”)
```

# Rappels : Kotlin Flows - Collect

Pour enregistrer un collecteur, rien de plus simple :

```
simpleFlow.collect { apiResponse ->
    // Do something with the apiResponse
}
```

À noter que collect doit être appelé dans une coroutine. Exemple :

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {

    init {
        viewModelScope.launch {
            // Trigger the flow and consume its elements using collect
            newsRepository.favoriteLatestNews.collect { favoriteNews ->
                // Update View with the latest favorite news
            }
        }
    }
}
```

# Rappels : Kotlin Flows - Collect + Composable

```
class UserViewModel: ViewModel() {
    private val  userState = MutableStateFlow(UserState())
    val userState: StateFlow<UserState> =  _userState
}

@Composable
fun UserScreen(userViewModel: UserViewModel) {
    val userState: UserState  by
userViewModel.userState.collectAsStateWithLifecycle ()
    Text(text = userState.username)
}
```

Moyen recommandé de collecter la valeur d'un flow depuis un composable

- Prise en compte du cycle de vie

# Architecture : Couche UI - State holder (Cas concret)

```
data class UserState(val username: String = "")
```

```
class UserViewModel: ViewModel() {
   private val _userState = MutableStateFlow(UserState())
   val userState: StateFlow<UserState> = _userState

   fun login(userName: String) {
       viewModelScope.launch(Dispatchers.IO) {
           //makeLoginApiRequest(userName)
           _userState.emit(_userState.value.copy(username = userName))
       }
   }
}
```

```
@Composable
fun UserScreen(userViewModel: UserViewModel) {
   val userState: UserState by
userViewModel.userState.collectAsStateWithLifecycle()
   Text(text = userState.username)
}
```

# Mini-project : Affichage d’une liste

Votre application doit récupérer une liste de boissons à partir d'une d'une liste fournie et les afficher sous forme de liste verticale. Chaque élément de la liste doit comporter :

- l'image de la boisson
- le nom de la boisson

Le design est libre, mais vous devez veiller à rendre l'interface claire et intuitive.

# Mini-project : Affichage d’une liste depuis le réseau

Même consigne sauf que la source de donnée devient le réseau :

[GET] : https://www.thecocktaildb.com/api/json/v1/1/filter.php?a=Alcoholic

Nouvelles notions : appel réseau, parsing json


```
{
   "drinks": [{
      "strDrink": "110 in the shade",
      "strDrinkThumb": "https:\/\/www.thecocktaildb.com\/images\/media\/drink\/xxyywq1454511117.jpg",
      "idDrink": "15423"
   }, {
      "strDrink": "151 Florida Bushwacker",
      "strDrinkThumb": "https:\/\/www.thecocktaildb.com\/images\/media\/drink\/rvwrvv1468877323.jpg",
      "idDrink": "14588"
   }
}
```

# Mini-project : Fonctionnalités avancées

Quelques idées:

- Barre de recherche et résultats (ex : www.thecocktaildb.com/api/json/v1/1/search.php?s=margarita)
- Affichage d'une image en grand après un clic sur un item
- Mode hors-ligne (gestion cache simple)
- Ajout en favoris + affichage des favoris
- Ajout d'animations
- etc

# Framework Navigation : Concept

Notions :

- NavGraph : Liste des routes / destinations de votre app
- NavController : élément central qui trace le parcours de l'utilisateur et permet de changer d'écran (de destination)
- NavHost : Composable "hôte" qui détient un NavController et un NavGraph

On récupère un NavController ainsi :

```
val navController = rememberNavController()
```

Note : Le NavController doit être initialisé au niveau le plus haut de votre app.

# Framework Navigation : Mise en place

```
interface NavigationDestination {
    /**
     * Unique path for a screen
     */
    val route: String
}
```

```
object HomeDestination: NavigationDestination {
    override val route = "home"
}

object MyAccountDestination: NavigationDestination {
    override val route = "myaccount"
}
```

```
@Composable
fun NavigationGraph(navController: NavHostController, modifier: Modifier = Modifier) {
    NavHost(
        navController = navController,
        startDestination = HomeDestination.route,
        modifier = modifier
    ) {
        composable(route = HomeDestination.route) {
            HomeScreen(...)
        }
        composable(route = MyAccountDestination.route) {
            MyAccountScreen(...)
        }
    }
}
```

# Framework Navigation : Comment naviguer (1)

- On navigue entre les routes via :

```
navController.navigate( route = MyAccountDestination. route)
```

- Bonne pratique : ne jamais passer le NavController à vos composables :

```
@Composable
fun NavigationGraph (navController: NavHostController , modifier:
Modifier = Modifier) {
    NavHost (
        navController = navController ,
        startDestination = HomeDestination. route ,
        modifier = modifier
    ) {
        composable(route = HomeDestination. route) {
            HomeScreen (navController)
        }
        composable(route = MyAccountDestination. route) {
            MyAccountScreen (navController )
        }
    }
}
```

```
@Composable
fun NavigationGraph(navController: NavHostController, modifier: Modifier = Modifier)
{
    NavHost(
        navController = navController,
        startDestination = HomeDestination.route,
        modifier = modifier
    ) {
        composable(route = HomeDestination.route) {
            HomeScreen(onGoToMyAccount = { navController.navigate(route =
MyAccountDestination.route) })
        }
        composable(route = MyAccountDestination.route) {
            MyAccountScreen(...)
        }
    }
}
```

# Framework Navigation : Comment naviguer (2)

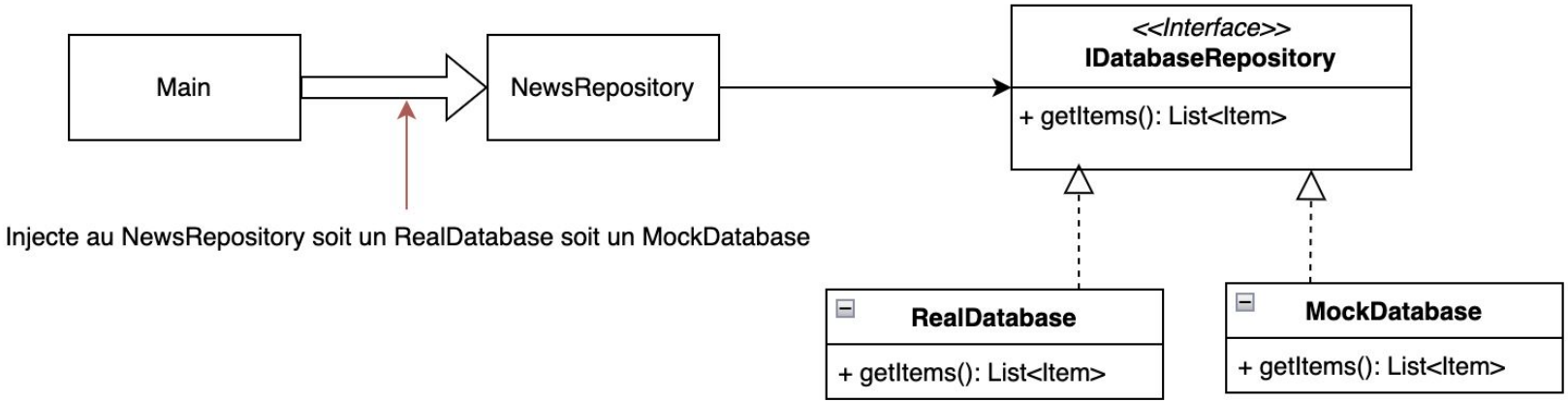
Navigation en spécifiant un path, en utilisant le **NavDeepLinkRequest** :

```
val request = NavDeepLinkRequest.Builder
    .fromUri("android-app://androidx.navigation.app/[ROUTE_PATH]" .toUri())
    .build()
navController.navigate(request)
```

Note : Il est possible de naviguer vers n'importe quelle destination de cette manière. Notamment vers une destination appartenant à un autre graphe.

# Injection de dépendance : Concept

- Permet de fournir les dépendances depuis l'extérieur



# Injection de dépendance : Exemple

```
interface IDatabaseRepository {
   fun getItems(): List<Item>
}
```

```
class RealDatabase: IDatabaseRepository {
   val database: Database = Database()

   override fun getItems(): List<Item> {
       return database.fetchAll()
   }
}
```

```
class MockDatabase: IDatabaseRepository {
   override fun getItems(): List<Item> {
       return listOf(
           Item(id = "1", s = "a"),
           Item(id = "2", s = "b"),
           Item(id = "3", s = "c")
       )
   }
}
```

```
fun main() {
   val newsRepository = NewsRepository(RealDatabase())
// OU MockDatabase()
   val news = newsRepository.fetchNews()
}
```

```
class NewsRepository(val databaseRepository: IDatabaseRepository)
{
   fun fetchNews(): List<Item> {
       return databaseRepository.getItems()
   }
}
```

# Injection de dépendance : Avantages et types d’injection

- **Testabilité** : Possibilité de mock ou remplacer les dépendances pour les tests unitaires
- **Modularité et Réutilisabilité** : Chaque composant est indépendant et peut donc être réutilisé facilement
- **Maintenabilité**

Types d’injections :

- Constructeur
- Setter
- Interface (voir bibliothèque Hilt)

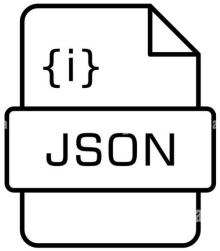
# Couche Data : Data source (optionnel)

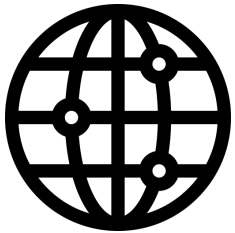
Data source = pont entre l'app et le système pour des opérations sur les données

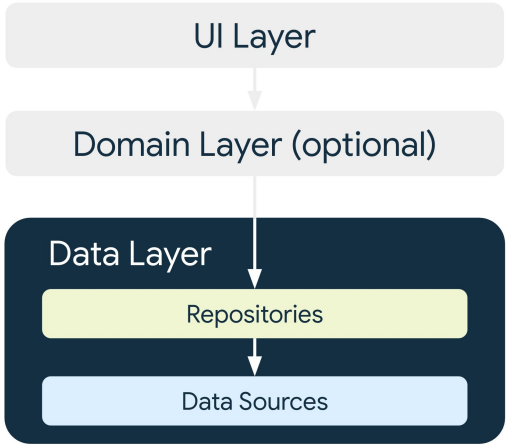
1 data source = 1 source de donnée -> Fichier, réseau, BDD locale

Classe non indispensable si logique peu complexe :

- Possibilité de fusionner la logique (avec le Repository)

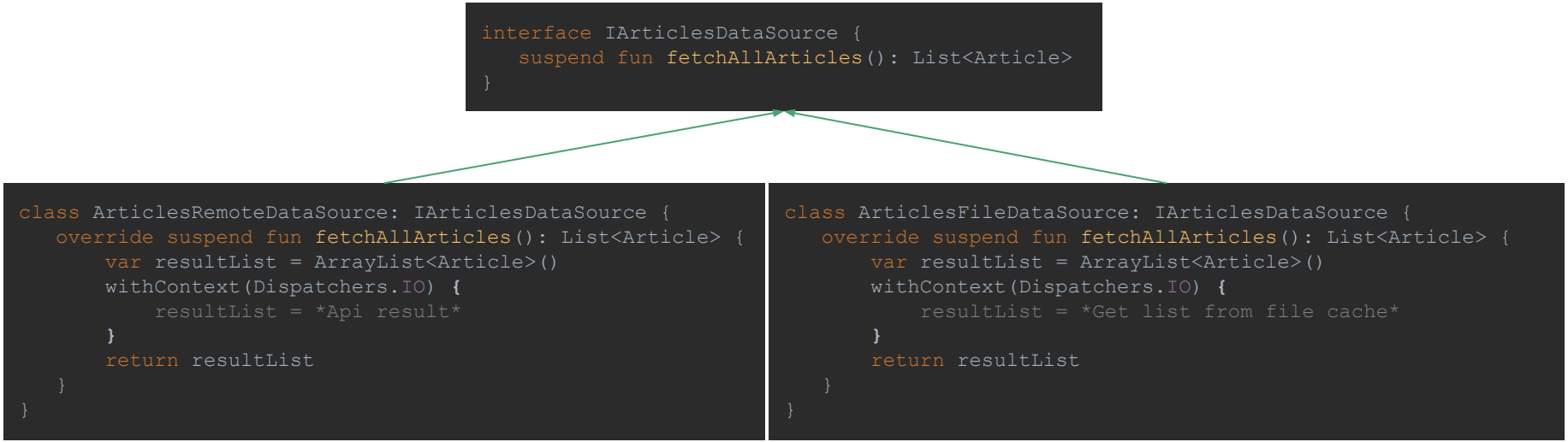


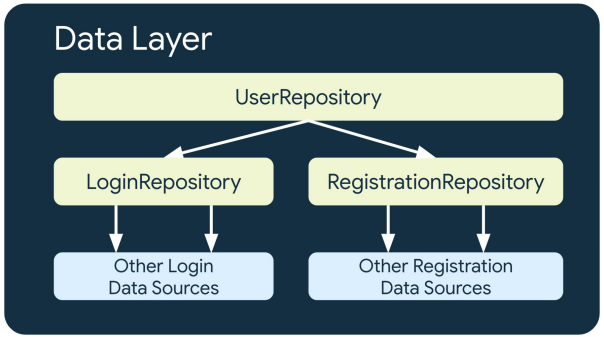
# Couche Data : Data source - Exemple

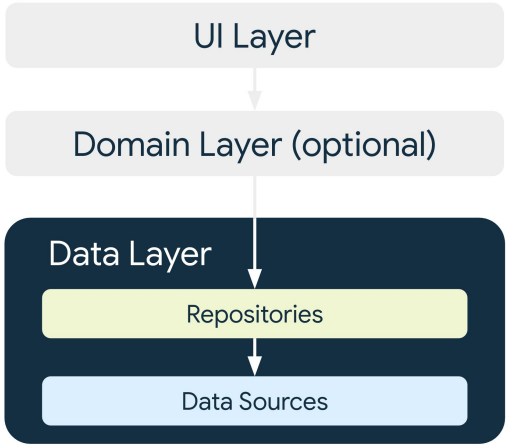
Convention d’écriture : [DataType] + [Source] + DataSource.



# Couche Data - Repository

- Chaque type de donnée doit avoir son repository associé
- Repository = méthodes *suspend* + flows
- Convention pour le nom d'un Repository :
  - [DataType] + "Repository
  - Ex : PaymentsRepository, NewsRepository, DrinksRepository
- Cas du multiple level repositories :



# Couche Data : Repository - Exemple 1 (Avec DataSource)

```
interface IArticlesRepository {
   suspend fun fetchAllArticles(): Flow<List<Article>>
}
```



```
class ArticlesRepository(private val dataSource: IArticlesDataSource): IArticlesRepository {
   override suspend fun fetchAllArticles(): Flow<List<Article>> = flow {
       emit(dataSource.fetchAllArticles())
   }
}
```

# Couche Data : Repository - Exemple 2 (Sans DataSource)

```
interface IArticlesRepository {
    suspend fun fetchAllArticles(): Flow<List<Article>>
}
```



```
class ArticlesRepository: IArticlesRepository {
    override suspend fun fetchAllArticles(): Flow<List<Article>> = flow {
        var resultList = ArrayList<Article>()
        withContext(Dispatchers.IO) {
            resultList = *Api result*
        }
        emit(resultList)
    }
}
```

On a fusionné la logique du DataSource dans le repository

# Couche Data : Repository vs Data source

En résumé :

- Data source = est le moyen par lequel on récupère de la donnée (fichier, réseau, BDD locale).
- Repository = moyen d’exposition de la donnée au reste de l’app*
- Sur des applications avec une complexité moindre on pourra fusionner ces deux “composants” pour ne garder que le Repository (qui contiendra la logique du Data source)

*Reste de l’app = toute l’app en dehors du layer Data

# Couche Data : Représentation de la donnée

Différenciation des modèles de donnée :

Accessible uniquement
dans la couche Data

```
data class ArticleApi(
    val id: String,
    val title: String,
    val content: String,
    val publicationDate: Date,
    val author: String,
    val lastModification: Date,
    val authorId: String
)
```

Accessible partout dans l'app

```
data class Article(
    val id: String,
    val title: String,
    val content: String,
    val publicationDate: Date,
    val author: String
)
```

# Appels réseau et gestion d’erreurs (1)

Intégrer simplement la gestion d’erreur dans l’app :

- Utilisation de la classe Result<T>

```
suspend fun getFromApi(): Result<List<Article>> {
    val apiResult = { doRequest() }
    return if (apiResult.is4XX()) {
        Result.failure(Exception(apiResult.errorMessage))
    }
    else {
        Result.success(apiResult.getList())
    }
}
```

UI Layer

```
rememberCoroutineScope().launch {
    val getResult = getFromApi()
    if (getResult.isSuccess) {
        // update display
    }
    else {
        // display message to user
    }
}
```

Note : Ici la méthode getFromApi() est appelée directement par l’UI. Ce qui n’est clairement pas conseillé. C’est seulement à titre d’exemple afin de simplifier la logique.

# Appels réseau et gestion d’erreurs (2) - Méthode “custom”

```
sealed class MyApiResult<out T> {
    data class Success<out R>(val value: R): MyApiResult<R>()
    data class Error(val errorMessage: String): MyApiResult<Nothing>()
}
```

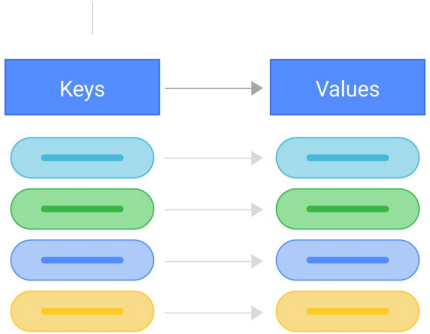
```
suspend fun getFromApi():
MyApiResult<List<Article>> {
    val apiResult = { doRequest() }
    return if (apiResult.is4XX()) {
        MyApiResult.Error(apiResult.errorMessage)
    }
    else {
        MyApiResult.Success(apiResult.getList())
    }
}
```

```
rememberCoroutineScope().launch {
    val getResult = getFromApi()
    when (getResult) {
        is MyApiResult.Error -> {
            // display message to user
        }
        is MyApiResult.Success -> {
            // update display
        }
    }
}
```

# Gestion du cache

Différentes manières possibles (liste non exhaustive) :

- Gestion manuelle par sauvegarde en fichiers
- Sauvegarde clé-valeur : SharedPreferences et DataStore
- Base de donnée


# Gestion du cache : DataStore

- Stockage en mode dictionnaire : clé - valeur
- Cas d'utilisation (exemples) : Préférences utilisateurs, informations de session…

**Déclaration du DataStore et des clés d'accès**

```
// At the top level of your kotlin file:
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "user_prefs")
// Preferences keys
val USER_PREF_DARKMODE = booleanPreferencesKey("user_pref_darkmode")
```

**SET la donnée**

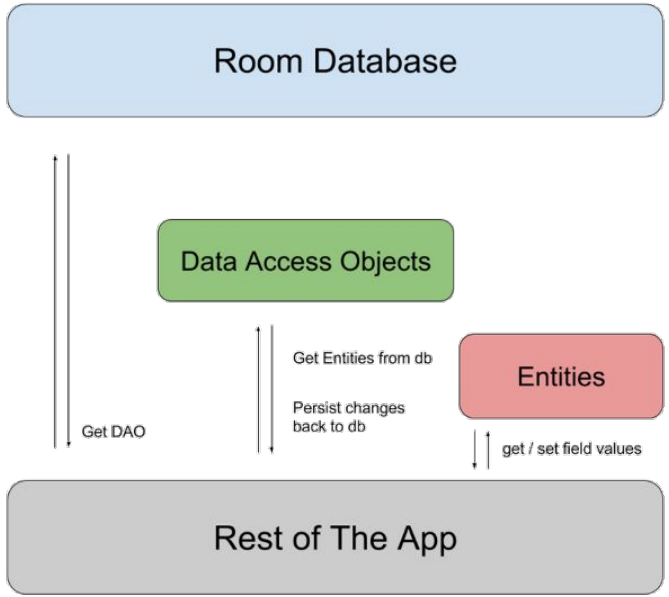
```
suspend fun setIsDarkMode(isDarkMode: Boolean) {
    baseContext.dataStore.edit { preferences ->
        preferences[USER_PREF_DARKMODE] = isDarkMode
    }
}
```

**GET la donnée**

```
fun getUserPrefDarkMode(): Flow<Boolean> {
    return baseContext.dataStore.data.map { preferences ->
        preferences[USER_PREF_DARKMODE] ?: false
    }
}
```

# Gestion du cache : Room

- Room = BDD locale
- But : la BDD = seule source de vérité
- On retiendra 3 composants principaux :
  - La classe Database
  - Les Data Entities (=tables)
  - Data Access Objects (DAO) : fournissent des méthodes pour récupérer, mettre à jour, insérer (etc) des données dans la BDD.



# Gestion du cache : Room - Liens utiles

Guide Room : https://developer.android.com/training/data-storage/room

Projet Git implémentant Room : https://github.com/android/architecture-samples/tree/main/app/src/main/java/com/example/android/architecture/blueprints/todoapp

# Ressources utiles

Guides officiels : https://developer.android.com/guide

Cours et codelabs : https://developer.android.com/courses

Code avec divers exemples :
https://github.com/android/snippets/tree/5e2384bb0b6a6df62ab5bd2e13fe13852a769e6b/compose/snippets/src/main/java/com/example/compose/snippets

Exemple d'application simple
https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app

Exemple de mise en oeuvre du framework navigation :
https://github.com/android/compose-samples/blob/main/JetNews/app/src/main/java/com/example/jetnews/ui/JetnewsNavGraph.kt