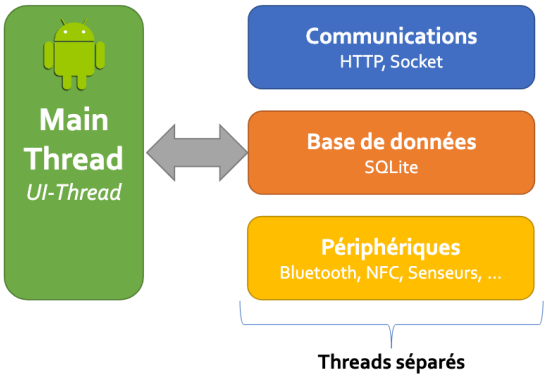
Threads & Coroutines

Threads

UI-Thread et opérations bloquantes
- Thread principal responsable de l'UI, ne doit jamais être bloqué
- Opérations réseau → exception si sur UI-Thread
- Opérations longues → thread séparé obligatoire



Handler
- Communication entre threads
- Post tâches sur thread principal depuis background

```
val handler = Handler(Looper.getMainLooper())
handler.post { myImage.setImageBitmap(bmp) }
```

runOnUiThread dans Activity

```
runOnUiThread { myImage.setImageBitmap(bmp) }
```

Limites des threads
- Non conscients du cycle de vie Android
- Risque de fuites mémoire (références Activity)
- Gestion concurrence complexe
- Solution : WeakReference pour éviter memory leaks

Coroutines

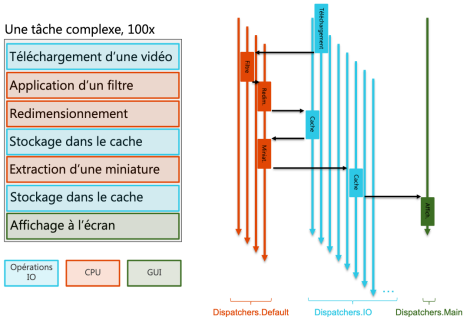
Caractéristiques
- Unité légère d'exécution asynchrone
- Suspendent/reprennent sans bloquer thread
- Code asynchrone séquentiel (lisible)
- Mot-clé `suspend`
- Coroutine = partage de threads

Suspending vs Bloquantes : Fonctions **suspendues** libèrent le thread pendant l'attente (améliore réactivité), fonctions **bloquantes** empêchent le thread de continuer. L'approche `suspended` permet d'écrire du **code asynchrone** de manière **séquentielle**, facilitant la lecture et la maintenance.

Exemple :

```
suspend fun downloadImage(url: String): ByteArray {
  return withContext(Dispatchers.IO) {
    URL(url).readBytes() // bloquant mais hors UI-Thread
  }
}
```

Dispatchers Associe une tâche à un thread ou pool de threads spécifique.
- **`Dispatchers.Main`** : UI thread
- **`Dispatchers.IO`** : I/O (réseau, fichiers, DB) - max 64 threads
- **`Dispatchers.Default`** : calculs CPU - nbr threads = nbr coeurs




```
suspend fun downloadImage(url : URL) : ByteArray? = withContext(Dispatchers.IO) {
  try {
    url.readBytes()
  } catch (e: IOException) {
    Log.w(TAG, "Exception while downloading image", e)
    null
  }
}
```
Le téléchargement de l'image dans le dispatcher *IO*

```
suspend fun decodeImage(bytes : ByteArray?) : Bitmap? = withContext(Dispatchers.Default) {
  try {
    BitmapFactory.decodeByteArray(bytes, 0, bytes?.size ?: 0)
  } catch (e: IOException) {
    Log.w(TAG, "Exception while decoding image", e)
    null
  }
}
```
Le décodage de l'image dans le dispatcher "*CPU*"

```
suspend fun displayImage(bmp : Bitmap?) = withContext(Dispatchers.Main) {
  if(bmp != null)
    myImage.setImageBitmap(bmp)
  else
    myImage.setImageResource(R.drawable.error_placeholder)
}
```
L'affichage de l'image dans l'*UI-Thread*

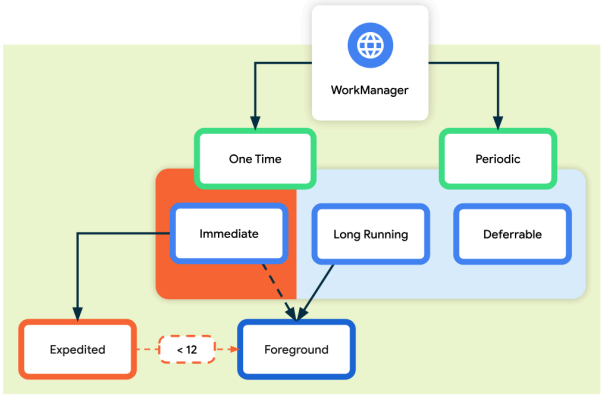
Scopes Les scopes permettent de limiter la durée de vie des coroutines à un contexte spécifique.
- **`GlobalScope`** : scope application (éviter, memory leaks)
- **`lifecycleScope`** : lié Activity/Fragment, auto-stop
- **`viewModelScope`** : lié ViewModel

```
lifecycleScope.launch {
  val bytes = downloadImage(url)
  val bmp = decodeImage(bytes)
  displayImage(bmp)
}
```

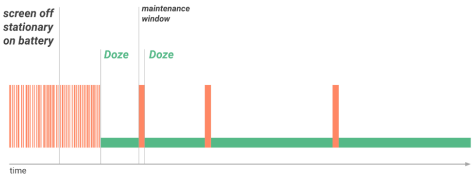
delay vs Thread.sleep
- `delay()` : suspend coroutine, libère thread
- `Thread.sleep()` : bloque thread

WorkManager

Usage
- Tâches longues/périodiques garanties
- Survit fermeture app et redémarrage
- Contraintes : réseau, batterie, stockage



Mode Doze
- Économie batterie : tâches différées
- Fenêtres activité contrôlées par système
- Classification apps : Active, Working set, Frequent, Rare, Restricted



Implémentation

```
class MyWork(appContext: Context, workerParams: WorkerParameters) :
Worker(appContext, workerParams) {
  override fun doWork(): Result {
    // tâche
    return Result.success()
  }
}

val workManager = WorkManager.getInstance(applicationContext)
val myWorkRequest = OneTimeWorkRequestBuilder<MyWork>().build()
workManager.enqueue(myWorkRequest)
```

```
val constraints = Constraints.Builder()
  .setRequiresCharging(false)
  .setRequiresBatteryNotLow(true)
  .setRequiredNetworkType(NetworkType.UNMETERED)
  .setRequiresDeviceIdle(true)
  .build()
```
Il est possible de spécifier des contraintes

```
val myPeriodicWorkRequest = PeriodicWorkRequestBuilder<MyWork>(15, TimeUnit.MINUTES)
  .setConstraints(constraints)
  .setBackoffCriteria(BackoffPolicy.EXPONENTIAL,
                      PeriodicWorkRequest.MIN_BACKOFF_MILLIS, TimeUnit.MILLISECONDS)
  .build()
workManager.enqueue(myPeriodicWorkRequest)
```
Il n'est pas possible de définir un intervalle de moins de 15 minutes

Si la tâche échoue, le système va tenter de la relancer plusieurs fois. Ici après 10s., 20s., 40s, 80s., etc.

Communication Web

Connectivité

Technologies

- Wi-Fi : 2.4 GHz (portée) vs 5 GHz (débit)
- Réseaux mobiles : 2G, 3G, 4G, 5G

Norme	Nom	Date	Fréquences	Débit maximum
802.11	N/A	1997	2.4 GHz	2 Mbps
802.11b	Wi-Fi 1	1999	2.4 GHz	11 Mbps
802.11a	Wi-Fi 2	1999	5 GHz	54 Mbps
802.11g	Wi-Fi 3	2003	2.4 GHz	54 Mbps
802.11n	Wi-Fi 4	2009	2.4 GHz ou 5 GHz	72 Mbps 450 Mbps
802.11ac	Wi-Fi 5	2014	5 GHz	1'000 Mbps
802.11ax	Wi-Fi 6	2019	2.4 et 5 GHz	2'400 Mbps
802.11ax	Wi-Fi 6E	2021	2.4, 5 et 6 GHz	4'800 Mbps
802.11be	Wi-Fi 7	2024	2.4, 5 et 6 GHz	30'000 Mbps
802.11bn	Wi-Fi 8	2028 ?	2.4, 5 et 6 GHz	100'000 Mbps

Génération	Année	Données (pointe)	Latence	Remarques
1G	1983	ø	ø	Système analogique
2G	1992	Kbit / s	< 1000 ms	Premier réseau numérique, initialement uniquement voix
3G	2003	Mbit / s	< 500 ms	Intégration des données dès la conception
4G	2010	Gbit / s	< 100 ms	Réseau données uniquement, intégration ultérieure de la voix
5G	2019+	Gbit / s	< 5 ms	Ouverture à l'Internet des Objets (IoT)

Permissions

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Services REST

Appel GET avec java.net.URL

```
val url = URL("https://api.example.com/data")
thread {
  val json = url.readText(Charsets.UTF_8)
}
```

Désérialisation JSON avec Gson

```
val type = object : TypeToken<List<FruitDTO>>() {}.type
val fruits = Gson().fromJson<List<FruitDTO>>(json, type)
```

Appel POST/PUT

```
val connection = url.openConnection() as HttpURLConnection
connection.requestMethod = "PUT"
connection.doOutput = true
connection.setRequestProperty("Content-Type", "application/json")
connection.outputStream.bufferedWriter(Charsets.UTF_8).use {
  it.append(Gson().toJson(newFruit))
}
val responseCode = connection.responseCode
```

> ⚠ **Warning**
> - Méthodes synchrones/bloquantes
> - Connexion lors de `outputStream`, `inputStream` ou `responseCode`
> - Réutilisation automatique connexion

Alternatives : OkHttp, Volley, Retrofit (planification, annulation, cache, retry, sérialisation)

Comparaison java.net.URL vs Volley

java.net.URL :

```
val url = URL("https://www.heig-vd.ch")
thread {
  val html = url.readText(Charsets.UTF_8)
  runOnUiThread { txtView.text = html }
}
```

- Gestion «manuelle» des threads
- On utilise *readText* car on s'attend à recevoir du texte
- Pas de gestion des erreurs

Volley :

```
val url = "https://www.heig-vd.ch"
val queue = Volley.newRequestQueue(this)
val textRequest = StringRequest(Request.Method.GET, url,
  { response -> txtView.text = response },
  { txtView.text = "error" })
queue.add(textRequest)
```

- Pas de gestion des threads, on donne deux callbacks (succès et erreur) que la librairie appellera dans l'*UI-Thread*
- On précise vouloir traiter du texte en retour en utilisant une *StringRequest*
- Utilisation d'une queue qui va gérer l'exécution des requêtes

java.net.URL :

```
val url = URL("https://www.heig-vd.ch/logo.png")
thread {
  val bytes = url.readBytes()
  val bmp = BitmapFactory
        .decodeByteArray(bytes,0, bytes.size)
  runOnUiThread { imageView.setImageBitmap(bmp) }
}
```

- On utilise *readBytes* pour obtenir le payload brut
- L'image est décodée puis affichée

Volley :

```
val url = " https://www.heig-vd.ch/logo.png "
val queue = Volley.newRequestQueue(this)
val imgRequest = ImageRequest(url,
  { bmp -> imageView.setImageBitmap(bmp) },
  1024, 1024,
  ScaleType.CENTER_INSIDE,
  Bitmap.Config.ARGB_8888,
  { error -> Log.d("MainActivity", "${error.message}") })
queue.add(imgRequest)
```

- Utilisation d'une *ImageRequest* pour traiter un payload contenant une image, la librairie va convertir l'image chargée (changement de l'encodage, redimensionnement, crop)

java.net.URL :

```
val url = URL("https://www.fruityvice.com/api/fruit/all")
thread {
  val json = url.readText(Charsets.UTF_8)
  val type = object : TypeToken<List<FruitDTO>>() {}.type
  val fruits = Gson().fromJson<List<FruitDTO>>(json, type)
  runOnUiThread { txtView.text = fruits.toString() }
}
```

- On utilise *readText* pour obtenir le *json* brut, puis *Gson* pour le désérialiser

Volley :

```
val url = "https://www.fruityvice.com/api/fruit/all"
val queue = Volley.newRequestQueue(this)
val jsonRequest = JsonArrayRequest(Request.Method.GET, url, null,
  { json -> txtView.text = json.toString() },
  { error -> Log.d("MainActivity", "${error.message}");
        error.printStackTrace() }
)
queue.add(jsonRequest)
```

- L'endpoint retournant un tableau, on doit utiliser une *JsonArrayRequest*
- L'objet *json* qui est passé au callback est du type ***org.json**.JSONArray* qui n'est pas directement exploitable

java.net.URL avec coroutines

```
suspend fun downloadHTML(urlParam: String): String =
withContext(Dispatchers.IO) {
  URL(urlParam).readText(Charsets.UTF_8)
}
```

Volley avec coroutines

```
suspend fun downloadHTMLVolley(urlParam: String): String = suspendCoroutine
{ cont ->
  val textRequest = StringRequest(Request.Method.GET, urlParam,
    { response -> cont.resume(response) },
    { e -> cont.resumeWithException(e) })
  queue.add(textRequest)
}
```

Ktor

Framework asynchrone client/serveur pour HTTP.

```
private val ktorClient = HttpClient(Android) {
  // Configuration du moteur Android
  engine {
    connectTimeout = 5_000
    socketTimeout = 5_000
    dispatcher = Dispatchers.IO
  }

  // Plugin pour parser le JSON
  install(ContentNegotiation) {
    json(Json {
      ignoreUnknownKeys = true
      prettyPrint = true
      isLenient = true
    })
  }
}
```

```
private suspend fun requestFruits(): List<FruitDTO> {
  // 1. On lance la requête GET
  // 2. .body() convertit automatiquement le JSON en List<FruitDTO>
  return ktorClient
        .get("https://www.fruityvice.com/api/fruit/all")
        .body()
}

fun getFruits() {
  // La requête s'exécute dans le scope du ViewModel
  viewModelScope.launch {
    val allFruits = requestFruits()
    ...
  }
}
```

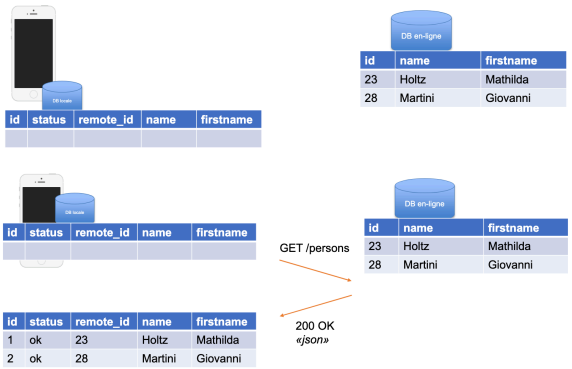
Synchronisation données

Contexte

- App fonctionne hors-ligne
- DB locale synchronisée avec serveur
- Algorithme sync complexe (conflits multiples devices)

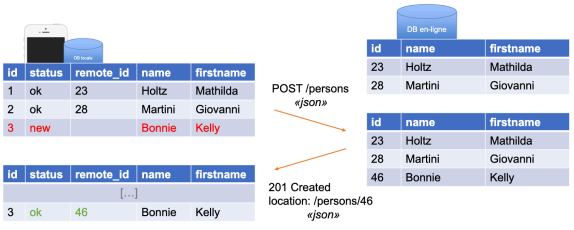
Structure DB locale

- `id` : identifiant local unique
- `remoteId` : identifiant serveur (null si nouveau)
- `status` : ok, new, mod, del



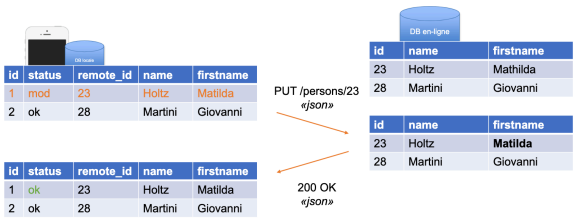
Création locale

`status = new, remoteId = null`
→ Sync : POST → remoteId reçu, status = ok



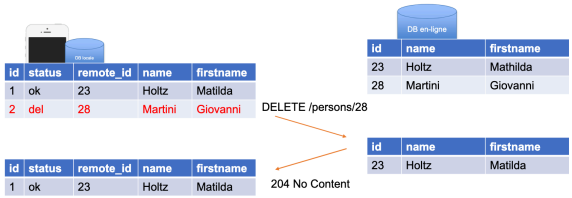
Modification

```
status = mod
→ Sync : PUT
```



Suppression

```
status = del
→ Sync : DELETE
```



Single Source of Truth (SSOT)

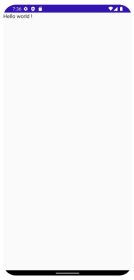
- DB locale = source unique de vérité
- Repository gère sync
- UI observe DB uniquement
- Pattern offline-first

Jetpack Compose

Caractéristiques

- API déclarative pour UI (Kotlin uniquement)
- Approche “Quoi” vs “Comment”
- Composants réutilisables et testables
- BOM pour gestion versions

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyComposeApplicationTheme {
                Text(text = "Hello world !")
            }
        }
    }
}
```



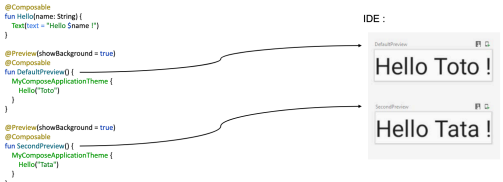
Fonctions @Composable

Règles importantes

- Exécution très fréquente (animations 60fps)
- Optimisation recomposition automatique
- Doit être rapide, sans effets de bord
- Idempotente (même entrée = même sortie)
- Éviter I/O et modifications externes

Prévisualisation

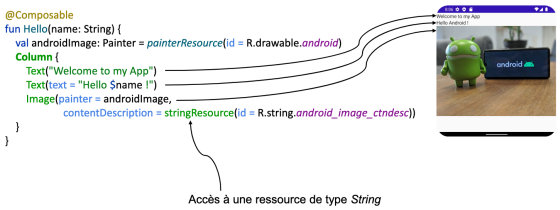
```
@Preview
@Composable
fun Hello(name: String = "World") {
  Text(text = "Hello $name!")
}
```



Layouts

Column

```
Column(
  verticalArrangement = Arrangement.SpaceBetween,
  horizontalAlignment = Alignment.CenterHorizontally
) {
  Text("Élément 1")
  Text("Élément 2")
}
```



Row

```
Row(
  horizontalArrangement = Arrangement.SpaceBetween,
  verticalAlignment = Alignment.CenterVertically
) { }
```

```
@Composable
fun Hello() {
  Row {
    Button(onClick = {}) {
      Text(text = "Un")
    }
    Button(onClick = {}) {
      Text(text = "Deux")
    }
    Button(onClick = {}) {
      Text(text = "Trois")
    }
  }
}
```

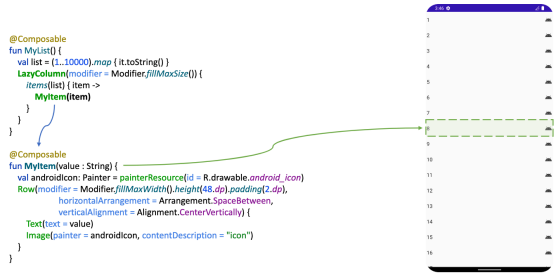
Composants de base

Scaffold

```
Scaffold(
  topBar = { TopAppBar(...) },
  floatingActionButton = { FloatingActionButton(...) },
  content = { }
)
```

Listes paresseuses

- `LazyColumn` : liste verticale
- `LazyRow` : liste horizontale
- `LazyVerticalGrid` : grille



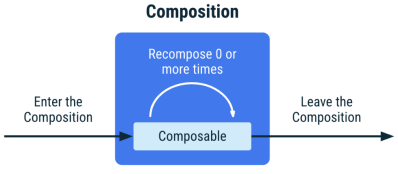
Gestion événements

```
Row(modifier = Modifier.clickable {
  Toast.makeText(context, "Cliqué", Toast.LENGTH_SHORT).show()
}) { }
```

Gestion états

State / MutableState

- `remember` : conserve état entre recompositions
- `rememberSaveable` : survit recréation Activity



Compteur

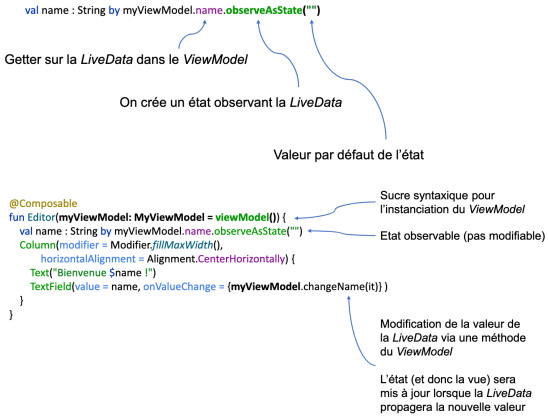
```
@Composable
fun Counter() {
  var counter by remember { mutableStateOf(0) }
  Button(onClick = { ++counter }) {
    Text("+")
  }
```

```
  Text("$counter")
}
```

TextField

```
@Composable
fun Editor() {
  var name by remember { mutableStateOf("") }
  TextField(value = name, onValueChange = {name = it})
}
```

ViewModel et LiveData



Flux exécution

1. Saisie TextField
2. Appel `changeName()` ViewModel
3. Modification `_name.value`
4. `observeAsState()` détecte changement
5. Recomposition avec nouvelle valeur

Le *ViewModel* et sa *Factory* :

```
class MyViewModel(initialName: String) : ViewModel() {
  private val _name = MutableLiveData(initialName)

  val name : LiveData<String> get() = _name

  fun changeName(newName : String) {
    _name.value = newName
  }
}

class MyViewModelFactory(private val initialName: String) : ViewModelProvider.Factory {
  override fun <T : ViewModel> create(modelClass: Class<T>): T {
    if(modelClass.isAssignableFrom(MyViewModel::class.java))
      return MyViewModel(initialName) as T
    throw IllegalArgumentException("Unknown ViewModel class")
  }
}
```

La fonction composable utilisant la *Factory* :

```
@Composable
fun Editor(myViewModel: MyViewModel = viewModel(factory = MyViewModelFactory("Toto"))) {
  val name : String by myViewModel.name.observeAsState("")
  Column(modifier = Modifier.fillMaxWidth(),
      horizontalAlignment = Alignment.CenterHorizontally) {
    Text("Bienvenue $name !")
    TextField(value = name, onValueChange = {myViewModel.changeName(it)} )
  }
}
```

State Hoisting

Déplacer gestion état vers composant parent.

Avantages

- Single Source of Truth
- Encapsulation
- Interceptable
- Partage entre composants
- Découplage (facilite tests)

```
@Composable //stateful
fun EditorScreen() {
  var name by remember { mutableStateOf("") }
  EditorContent(name = name, onNameChange = { name = it })
}

@Composable //stateless
fun EditorContent(name: String, onNameChange: (String) -> Unit) {
  Column(modifier = Modifier.fillMaxWidth(),
      horizontalAlignment = Alignment.CenterHorizontally) {
    Text("Bienvenue $name !")
    TextField(value = name, onValueChange = onNameChange )
  }
}
```

> ⚠ **Warning**
>
> Créer nouvelle instance avec `copy()` pour détecter changements (comparaison par référence).

StateFlow

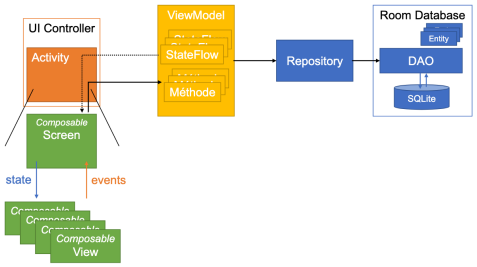
Recommandé avec Compose à la place de LiveData.

ViewModel

```
class PersonViewModel : ViewModel() {
  private val _person = MutableStateFlow(Person("", ""))
  val person: StateFlow<Person> = _person.asStateFlow()

  fun changePerson(name: String? = null, firstName: String? = null) {
    _person.update { currentPerson ->
      currentPerson.copy(
        name = name ?: currentPerson.name,
        firstName = firstName ?: currentPerson.firstName
      )
    }
  }
}
```

Room avec StateFlow

```
// DAO
@Query("SELECT * FROM Contact")
fun getAllContacts(): Flow<List<Contact>>

// Repository
val allContacts: Flow<List<Contact>> = contactsDao.getAllContacts()

// ViewModel
val allContacts: StateFlow<List<Contact>> = repository.allContacts
  .stateIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed(5000L),
    initialValue = emptyList()
  )

// Compose
val contacts by contactViewModel.allContacts.collectAsStateWithLifecycle()
```

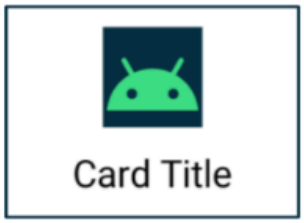


Layout adaptatif

BoxWithConstraints

```
@Composable
fun Card() {
  BoxWithConstraints {
    if (maxWidth < 400.dp) {
      Column { /* Layout mobile */ }
    } else {
      Row { /* Layout tablette */ }
    }
  }
}
```

```
val windowSizeClass = calculateWindowSizeClass(activity = this)
windowSizeClass.widthSizeClass
```



Hes·so
Haute Ecole Spécialisée de Suisse occidentale

UI Hybride
- Possible mais non recommandé
- `ComposeView` : Compose dans XML
- `AndroidView` : XML dans Compose
- Usage : migration progressive

Layout activity_main.xml :

```
<LinearLayout
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Texte XML"/>

  <androidx.compose.ui.platform.ComposeView
    android:id="@+id/compose_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

</LinearLayout>
```

Activité :

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    findViewById<ComposeView>(R.id.compose_view).setContent {
      AppCompatTheme {
        Text(text = "Texte Compose")
      }
    }
  }
}
```

Tests automatisés

3 types
- **Tests unitaires** : classes/méthodes individuelles
- **Tests d'intégration** : interactions composants
- **Tests bout en bout** : comportement utilisateur

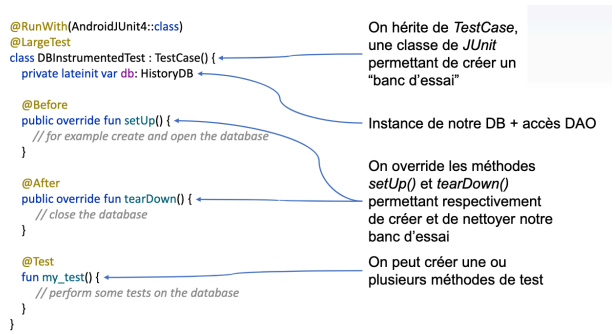
Tests unitaires (JUnit)

```
class CalculatorTest {
  private val calculator = Calculator()

  @Test
  fun testPow() {
    assertEquals(8.0, calculator.pow(2.0, 3.0), 0.001)
  }

  @Test
  fun testFactorial() {
    assertEquals(120, calculator.factorial(5))
  }
}
```

Tests instrumentalisés

Configuration
- `@RunWith(AndroidJUnit4::class)`
- Héritage `TestCase`
- `@Before setUp()` : initialisation
- `@After tearDown()` : nettoyage

Tests Room




```
@Before
override fun setUp() {
  val context = ApplicationProvider.getApplicationContext<Context>()
  db = Room.inMemoryDatabaseBuilder(context, HistoryDB::class.java).build()
  dao = db.historyDao()
}

@After
override fun tearDown() {
  db.close()
}
```

Tests Compose

Configuration

```
@get:Rule
val composeTestRule = createComposeRule()
```

Tests basés texte (limité)

```
@Test
fun editorScreenTest() {
  composeTestRule.setContent { EditorScreen(emptyTestPerson) }
  composeTestRule.onNodeWithText("Name").performTextInput(name)
  composeTestRule.onNodeWithText("Bienvenue", substring = true)
    .assertTextEquals("Bienvenue $fname $name !")
}
```

Limitations texte
- Sensible refactoring
- Problèmes traduction
- Ambiguïté éléments multiples

Solution : testTag

```
// Composable
Text(
  modifier = Modifier.testTag("welcome-msg"),
  text = "Bienvenue"
)

// Test
composeTestRule.onNodeWithTag("welcome-msg")
  .assertTextEquals("Bienvenue Jean Neige !")
```

Cheat-sheet : https://developer.android.com/jetpack/compose/testing-cheatsheet

CI/CD

Prérequis
- SDK Android
- Émulateur pour tests instrumentalisés

Docker
1. Image `openjdk:11-jdk`
2. Télécharger SDK Android
3. Accepter licences CLI
4. Télécharger versions nécessaires
5. Gradle : `app:lintDebug`, `app:assembleDebug`, `app:testDebug`

Tutoriel : https://howtodoandroid.com/setup-gitlab-ci-android/

Configuration émulateur
- Installation CLI automatisable
- Accélération matérielle requise
- Options : `--no-audio --no-windows`
- Démarrage : plusieurs minutes
- Désactiver animations avant tests

```
gradlew app:connectedAndroidTest
```

Tests supplémentaires

Monkey Testing (Play Store)
- Installation + interactions aléatoires
- Identifiants pour connexion
- Tests multi-appareils
- Vérification accessibilité
- Détection failles sécurité

Firebase Robo Tests
- Cartographie automatique app
- Captures écran/vidéos
- Logs détaillés
- Profilage
- Vérification niveaux API
- Version gratuite limitée
- Intégration CI/CD possible