ChatGPT

Tutoriel : Base de projet Android (Kotlin, Jetpack Compose) pour une application de puzzles

Introduction

Ce tutoriel guide pas à pas la création d'une base de projet Android en Kotlin utilisant **Jetpack Compose** pour développer une application de puzzles. Le contexte est un TP noté destiné à des débutants, et nous allons couvrir les aspects demandés : structure du projet favorisant l'ajout de composants, écran de démarrage (*SplashScreen*) affichant un drapeau spécifique, composant personnalisé de barre de progression (*FillBar*), et plan de réalisation des principales fonctionnalités (SplashScreen, FillBar, écran de jeu, etc.).

L'objectif est de fournir un guide clair et structuré, avec des extraits de code commentés et des bonnes pratiques, afin que vous puissiez construire progressivement votre application de puzzles. Assurez-vous d'avoir **Android Studio** (dans une version récente supportant Compose) installé.

1. Configuration du projet et structure recommandée

Avant d'implémenter les écrans et composants, configurons le projet Android pour **Jetpack Compose** et réfléchissons à une structure de code propre. Cela facilitera l'ajout ultérieur de nouveaux composants et fonctionnalités.

Étapes de configuration du projet :

1. **Créer le projet Android:** Dans Android Studio, créez un nouveau projet en choisissant le modèle "Activité vide Compose" (Empty Compose Activity). Donnez-lui un nom (par ex. "PuzzleApp") et un **namespace** (ex. `com.example.puzzleapp`). Assurez-vous que le langage est Kotlin et le **SDK minimum** au moins 21 (Compose nécessite un SDK >= 21) [1].
2. **Vérifier le `build.gradle` (Module: app):** Si vous n'avez pas utilisé le template Compose, vous devrez activer Compose manuellement dans Gradle. Dans le bloc `android` de votre module app, activez Compose et configurez le compilateur Kotlin dédié :

```
android {
    ...
    buildFeatures {
        compose true        // Active Jetpack Compose [2]
    }
    composeOptions {
        kotlinCompilerExtensionVersion "1.4.7"  // Version du Compiler Ext.
Jetpack Compose
    }
    ...
}
```

Dans la section `dependencies`, ajoutez les bibliothèques Jetpack Compose nécessaires. Il est recommandé d'utiliser le **BOM (Bill of Materials)** de Compose pour gérer les versions de manière cohérente [3]. Par exemple :

```
dependencies {
    // BOM Compose pour aligner les versions
    implementation platform("androidx.compose:compose-bom:2025.05.00")
    // Dépendances Jetpack Compose principales
    implementation "androidx.compose.ui:ui"                     // UI de
base Compose
    implementation "androidx.compose.material3:material3"       //
Composants Material Design 3
    implementation "androidx.activity:activity-compose:1.7.2"    //
Intégration Activity <-> Compose [4]
    implementation "androidx.navigation:navigation-compose:2.5.3"//
Navigation Compose pour gérer les écrans [5]
    // (Ajoutez d'autres dépendances Compose si besoin, ex: ui-tooling,
icons, etc.)
}
```

Explication: le **BOM Compose** importe une version cohérente pour l'ensemble de la suite Compose (UI, Material, etc.), évitant les incompatibilités de version. On importe ensuite les librairies nécessaires : ici Material3 pour l'UI moderne, *activity-compose* pour lancer Compose dans une Activity, et *navigation-compose* pour la gestion des écrans (navigation). 3. **Configurer l'AndroidManifest et le thème:** Ouvrez le fichier `AndroidManifest.xml` et assurez-vous que l'**activité principale** y est déclarée. Par exemple :

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.PuzzleApp">
    <activity android:name=".MainActivity"
              android:exported="true"
              android:theme="@style/Theme.PuzzleApp.NoActionBar">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Ici, nous appliquons le thème de l'application à `MainActivity`. Le template Compose crée généralement un thème Material3 (`Theme.PuzzleApp`) défini dans `themes.xml` et/ou dans du code Kotlin (fichiers `*Theme.kt` générés). Le thème **NoActionBar** est utilisé pour cacher la barre d'application par défaut, car nous utiliserons notre propre UI en Compose. 4. **Structurer les packages Kotlin:** Pour un projet clair, créez des packages pour séparer les composantes. Par exemple, sous `app/src/main/java/com/example/puzzleapp/` : - `ui/theme` – contient les fichiers liés au thème

Compose (couleurs, typographie, thème MaterialTheme). - `ui/screens` – contient les écrans de l'application sous forme de fonctions composables (SplashScreen, GameScreen, etc.). - `ui/components` – contient les composants UI réutilisables comme la FillBar. - `model` ou `data` – (optionnel) pour la logique du puzzle, structures de données, ViewModel, etc.

Cette organisation n'est pas strictement imposée, mais elle permet de s'y retrouver facilement et **d'ajouter de nouveaux composants** dans des fichiers séparés de façon cohérente. Par exemple, on pourra ajouter un nouvel écran en créant un fichier dans `ui/screens` sans tout mélanger dans l'activité principale.

1. **Intégrer la Navigation Compose (facultatif mais recommandé):** Pour gérer plusieurs écrans (SplashScreen, écran de jeu, écran de fin, etc.), il est pratique d'utiliser la bibliothèque Navigation Compose. Elle permet de naviguer entre des composables en conservant un historique de navigation. Nous l'avons ajoutée aux dépendances plus haut. Vous pouvez l'intégrer dans `MainActivity` comme suit :

   ```
   class MainActivity : ComponentActivity() {
       override fun onCreate(savedInstanceState: Bundle?) {
           super.onCreate(savedInstanceState)
           setContent {
               PuzzleAppTheme { // Applique le thème Material3
                   val navController = rememberNavController()
                   NavHost(navController = navController, startDestination
   = "splash") {
                       composable("splash") { SplashScreen(navController) }
                       composable("game")   { GameScreen(navController) }
                   }
               }
           }
       }
   }
   ```

   Explication: On crée un `NavHost` avec une route initiale `"splash"` (notre écran de démarrage). Deux destinations sont définies : `"splash"` affiche `SplashScreen` et `"game"` affiche `GameScreen`. Le `navController` permet de commander la navigation (nous l'utiliserons par exemple pour aller du splash vers le jeu).

Si vous préférez éviter la navigation pour simplifier, vous pouvez gérer un état booléen dans `MainActivity` pour afficher soit le splash soit l'écran principal. Cependant, adopter Navigation dès le début est une bonne pratique pour structurer l'appli dès qu'il y a plus d'un écran.

Une fois ces étapes réalisées, votre projet est configuré pour Compose et prêt à accueillir les fonctionnalités du TP puzzle.

2. Écran de démarrage (SplashScreen) avec le drapeau Sierra

La première étape du TP consiste à créer un **SplashScreen** qui s'affiche au lancement de l'application. Cet écran doit montrer un **drapeau correspondant à la lettre S** (puisque l'utilisateur s'appelle "Seer"). En alphabet international des codes, la lettre *S* est représentée par le mot *"Sierra"* et correspond à un pavillon maritime blanc avec un carré bleu central [6] .

Le pavillon maritime "Sierra" (lettre S) consiste en un drapeau blanc à carré bleu central. Ce drapeau correspond à l'initiale S du prénom "Seer" et sera affiché dans notre écran de démarrage (SplashScreen). Ce choix apporte une touche ludique et personnalisée à l'application. Nous allons voir comment intégrer cette image dans l'interface Compose.

Étapes pour implémenter le SplashScreen :

1. **Ajouter l'image du drapeau dans le projet:** Trouvez une image représentant le drapeau *Sierra* (format PNG ou SVG). Vous pouvez par exemple créer un fichier `flag_sierra.png` représentant un carré bleu sur fond blanc. Placez ce fichier dans le répertoire `app/src/main/res/drawable/`. Cela permettra d'y accéder via `R.drawable.flag_sierra` dans le code.
2. **Créer le composable SplashScreen:** Dans le package `ui/screens`, créez un fichier Kotlin `SplashScreen.kt`. Définissez une fonction annotée `@Composable` nommée `SplashScreen`. Cette fonction affichera l'image du drapeau, idéalement centrée sur l'écran. Utilisons un conteneur `Box` pour centrer le contenu. Par exemple :

```
@Composable
fun SplashScreen(navController: NavController) {
    // Occupy tout l'écran et centre le contenu
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(Color.White),         // fond blanc (ou utilisez
MaterialTheme.colorScheme.background)
        contentAlignment = Alignment.Center
    ) {
        Image(
            painter = painterResource(R.drawable.flag_sierra),
            contentDescription = "Drapeau Sierra (lettre S)",
            modifier = Modifier.size(200.dp)  // dimension de l'image,
ajustez si besoin
        )
    }
    // Après un délai, navigation vers l'écran principal
    LaunchedEffect(Unit) {
        delay(2000L) // attend 2 secondes
        navController.navigate("game") {
            popUpTo("splash") { inclusive = true }
        }
    }
}
```

   Dans cet extrait de code, on:
3. Utilise `Modifier.fillMaxSize()` pour que le conteneur occupe tout l'écran, et `contentAlignment = Alignment.Center` pour centrer l'image.
4. Affiche l'image du drapeau avec `Image` et un `painterResource` pointant vers notre ressource `flag_sierra`. Le `contentDescription` est important pour l'accessibilité (décrire l'image aux utilisateurs malvoyants).
5. Utilise `LaunchedEffect(Unit)` pour exécuter un effet de côté au lancement de ce composable : ici une suspension de 2 secondes (`delay(2000L)`) puis la navigation vers l'écran

de jeu `"game"`. Le paramètre `Unit` fait que cet effet ne s'exécute **qu'une seule fois** à l'entrée dans la composition (un `LaunchedEffect` sans variable dépendante s'exécute une seule fois lors de la première composition) [7].

6. Le `popUpTo("splash"){ inclusive = true }` permet de retirer l'écran splash de la backstack de navigation, ainsi l'utilisateur ne revient pas dessus en appuyant "Back".
7. **Configurer le thème du Splash (Optionnel avancé):** Sur Android 12+, le système affiche automatiquement un écran de lancement. Pour un résultat plus fluide, vous pourriez configurer le thème de lancement (`Theme.PuzzleApp.Splash`) avec l'image du drapeau en background, via le style `windowBackground`. Cependant, cette approche utilise XML et le *Android SplashScreen API*. Pour notre cas simplifié en Compose, le code ci-dessus suffit car il affiche rapidement l'image. Assurez-vous juste que le fond de l'activité (thème) est blanc pour éviter un flash noir au démarrage.
8. **Tester le SplashScreen:** Exécutez l'application. Vous devriez voir apparaître le drapeau Sierra pendant 2 secondes, puis la navigation se fait vers l'écran suivant (que nous appellerons l'écran de jeu, même s'il est vide pour l'instant). Si cela ne fonctionne pas, vérifiez que:
9. Le `NavHost` dans `MainActivity` utilise bien `"splash"` en startDestination et que `SplashScreen(navController)` y est intégré.
10. L'image est correctement référencée (pas d'erreur de ressource manquante).
11. Vous avez importé les fonctions nécessaires (`rememberNavController`, `painterResource`, etc.).

☞ **Bonnes pratiques:** Même pour un écran simple, on sépare la logique (ici la temporisation et navigation) de la définition visuelle. Nous utilisons Compose Navigation pour la transition d'écran plutôt qu'un `Handler.postDelayed` classique, ce qui rend le code déclaratif et propre. De plus, nous respectons l'accessibilité en fournissant un contentDescription à l'image, et nous maintenons le code UI réactif (si plus tard on voulait conditionner la durée ou skipper le splash, il suffirait de modifier le `LaunchedEffect` ou la navigation).

3. Composant barre de progression (FillBar)

La deuxième partie du TP porte sur la création d'une **barre de progression personnalisée**, appelée *FillBar*. L'idée est de fournir un indicateur visuel de progression, par exemple le pourcentage de puzzle complété ou le score. Nous allons créer ce composant en Compose afin de comprendre sa construction, bien que Jetpack Compose offre déjà des composants de progression (LinearProgressIndicator, CircularProgressIndicator) [8].

Notre FillBar sera une barre horizontale dont la portion remplie représente un pourcentage. Pour un débutant, cela exercera la gestion de layout basique et l'utilisation de paramètres Composables.

Étapes pour créer le composant FillBar :

1. **Définir le composable FillBar:** Dans un fichier `ui/components/FillBar.kt`, créez une fonction `@Composable fun FillBar(progress: Float)`. Le paramètre `progress` représentera le niveau de remplissage entre 0.0 et 1.0 (0% à 100%). C'est la convention également utilisée par les composants de progression Material [9]. Vous pouvez ajouter d'autres paramètres optionnels, par exemple la couleur de la barre ou sa taille, avec des valeurs par défaut.
2. **Construire la vue de la FillBar:** Pour dessiner la barre, on peut imbriquer deux `Box`. La *Box parent* servira de fond (la "track" en gris clair), et la *Box enfant* à l'intérieur aura une largeur proportionnelle à `progress` pour représenter la partie remplie. Par exemple :

```
@Composable
fun FillBar(progress: Float, modifier: Modifier = Modifier, color:
Color = MaterialTheme.colorScheme.primary) {
    // Constrain progress between 0f and 1f
    val clampedProgress = progress.coerceIn(0f, 1f)
    Box(
        modifier = modifier
            .fillMaxWidth()
            .height(20.dp)
            .background(MaterialTheme.colorScheme.surfaceVariant, shape
= RoundedCornerShape(50)) // fond arrondi
    ) {
        Box(
            modifier = Modifier
                .fillMaxWidth(fraction = clampedProgress) // largeur
proportionnelle
                .fillMaxHeight()
                .background(color, shape = RoundedCornerShape(50)) //
partie remplie colorée
        )
    }
}
```

Explication: On utilise `fillMaxWidth(fraction = progress)` pour que la largeur du second Box soit un pourcentage de la largeur totale du composant parent. Le fond utilise `surfaceVariant` du thème Material3 pour un gris clair, et la barre remplie utilise la couleur primaire du thème (on aurait pu paramétrer la couleur). On ajoute `RoundedCornerShape(50)` pour arrondir la barre (50% de hauteur, donc bords arrondis type "capsule").
On veille à *clamping* la valeur de `progress` entre 0 et 1 pour éviter les débordements (par exemple si on passe 1.2 ou -0.5 par erreur).

3. **Utiliser/Prévisualiser le composant:** Vous pouvez ajouter une fonction `@Preview` pour voir le rendu de la FillBar. Par exemple :

```
@Preview(showBackground = true)
@Composable
fun FillBarPreview() {
    Column(modifier = Modifier.padding(16.dp)) {
        Text("Progression du puzzle : 75%")
        FillBar(progress = 0.75f, modifier = Modifier.padding(vertical
= 8.dp))
    }
}
```

L'aperçu devrait montrer une barre remplie aux trois quarts en bleu (couleur primaire par défaut). N'hésitez pas à ajuster la hauteur, la couleur ou ajouter une bordure si souhaité.

4. **Intégrer la FillBar dans l'écran de jeu:** Une fois le composant disponible, placez-le dans l'UI de votre écran principal (GameScreen). Par exemple, on peut le mettre en haut d'écran pour indiquer la progression du joueur. Vous passerez en paramètre le pourcentage de progression

calculé. Au début du jeu, ce sera peut-être 0%, et cela augmentera au fur et à mesure que le puzzle se résout.

5. **Lier la progression à l'état du puzzle:** Pour rendre la FillBar dynamique, liez son `progress` à un état `MutableState` ou à un état dans un ViewModel représentant la progression. Par exemple, si vous suivez combien de pièces du puzzle sont bien placées, calculez `progress = piecesBienPlacees.toFloat() / totalPieces`. Mettez à jour cet état à chaque coup de l'utilisateur pour que la barre se **recompose** automatiquement. Compose rendra la barre plus remplie grâce au binding de l'état.
6. *Bonne pratique:* Vous pourriez animer la progression pour un rendu plus fluide. Jetpack Compose propose `animateFloatAsState` pour animer la valeur flottante de progression automatiquement lorsque celle-ci change. Cela donnera un effet de remplissage progressif plutôt qu'un changement brusque.

☞ **Remarque:** Il est tout à fait possible d'utiliser directement `LinearProgressIndicator` de Material à la place de créer FillBar. Par exemple, `LinearProgressIndicator(progress = progress, color = ..., trackColor = ...)` fournit déjà une barre prête à l'emploi conforme aux guides Material Design. Cependant, l'exercice de coder *FillBar* vous fait pratiquer la composition de bases (Box imbriqués) et la gestion de paramètres. À l'avenir, pour des besoins standards, n'hésitez pas à utiliser les composants existants qui intègrent déjà les bonnes pratiques Material.

4. Écran principal du puzzle (GameScreen) et logique de jeu

Passons à l'écran de jeu lui-même. Cette partie du TP vous demande sans doute de créer l'interface du puzzle et d'y intégrer la logique (par exemple, mélanger les pièces, gérer les interactions du joueur, détecter la fin du puzzle). Nous allons proposer une progression logique pour construire cet écran *pas à pas*, en se concentrant d'abord sur l'UI puis en évoquant la gestion d'état.

Étapes pour construire l'écran de puzzle (GameScreen) :

1. **Déterminer la représentation du puzzle:** Choisissez comment représenter les données du puzzle en Kotlin. Par exemple, pour un puzzle de taquin (15-puzzle, 4x4 avec une case vide) vous pouvez utiliser une liste de 16 entiers de 0 à 15, où 0 représente la case vide. Pour un puzzle de type image découpée, cela pourrait être une grille de tuiles.
   Définissez une variable d'état qui contiendra ce jeu de données. Le plus simple est d'utiliser un `remember` dans le composable (ou idéalement un `ViewModel` pour séparer UI et données). Par exemple :

   ```
   val puzzleState = remember { mutableStateOf(listOf(1, 2, 3, ..., 15, 0)) }
   ```

   Ici, on initialise l'état avec la liste ordonnée (pièce 1 à 15 et 0 pour vide à la fin). Plus tard, on écrira une fonction pour mélanger (shuffle) cette liste au début de la partie.
2. **Construire l'UI des pièces:** Utilisez des composables de disposition pour afficher la grille du puzzle. Vous pouvez soit utiliser un **GridLayout** via `LazyVerticalGrid` (Compose Foundation), soit plus simplement une **colonne de lignes** vu que la taille est fixe (4x4 par exemple). Pour chaque case, affichez un composant visuel (une carte, un bouton, etc.) avec le contenu approprié (numéro ou image de la pièce).
3. Si la case représente la pièce vide (par ex. valeur 0), on peut afficher un carré vide (transparent) pour matérialiser l'espace.

4. Sinon, affichez un **Card** ou **Box** avec un texte (le numéro) au centre. Ajoutez un style visuel (couleur de fond, bordure) pour ressembler à une pièce de puzzle.
   Exemple simplifié pour un puzzle 4x4 de chiffres :

```
@Composable
fun GameBoard(tiles: List<Int>, onTileClick: (Int) -> Unit) {
    Column {
        // 4 lignes
        for (i in 0 until 4) {
            Row {
                // 4 colonnes
                for (j in 0 until 4) {
                    val index = 4 * i + j
                    val value = tiles[index]
                    if (value != 0) {
                        // Tuile non vide
                        Box(
                            modifier = Modifier
                                .size(80.dp)
                                .padding(4.dp)
                                .background(MaterialTheme.colorScheme.primaryContainer,
RoundedCornerShape(8.dp))
                                .clickable { onTileClick(index) }, //
action sur clic
                            contentAlignment = Alignment.Center
                        ) {
                            Text(
                                text = value.toString(),
                                style =
MaterialTheme.typography.headlineMedium,
                                color =
MaterialTheme.colorScheme.onPrimaryContainer
                            )
                        }
                    } else {
                        // Case vide
                        Box(modifier =
Modifier.size(80.dp).padding(4.dp)) { /* empty */ }
                    }
                }
            }
        }
    }
}
```

Explication: On parcourt la liste de tuiles pour en afficher une grille 4x4. Chaque tuile non vide est affichée comme un carré cliquable avec un texte. On utilise le thème Material3 pour les couleurs (primaryContainer pour la tuile, onPrimaryContainer pour le texte) afin de respecter la cohérence visuelle de l'application. La case vide est juste un espace vide pour conserver l'alignement.

Le paramètre `onTileClick` permettra de notifier lorsqu'une tuile est cliquée (voir étape suivante).

5. **Gérer les interactions (mouvements du puzzle):** Maintenant, implémentez la logique lorsque l'utilisateur clique sur une tuile. Dans un puzzle glissant, cliquer une tuile adjacente à la case vide doit l'échanger avec le vide (pour la déplacer). Vous pouvez implémenter la fonction `onTileClick(index: Int)` dans `GameScreen`. Cette fonction vérifiera si la tuile cliquée est déplaçable (adjacente au 0). Si oui, permute sa position avec le 0 dans la liste d'état du puzzle.
6. Pour trouver le voisinage, calculer l'index du 0 dans la liste (`emptyIndex = tiles.indexOf(0)`). Comparer avec l'index cliqué : il est déplaçable si c'est la case directement au-dessus, en dessous, à gauche ou à droite. En indices, cela veut dire soit `index == emptyIndex ± 1` dans la même ligne, soit `index == emptyIndex ± 4` (au-dessus ou en dessous).
7. Si déplaçable, faites une copie mutable de la liste d'état, échangez les deux éléments, puis faites `puzzleState.value = nouvelleListe`. Grâce à Compose, le UI se mettra à jour automatiquement pour refléter la nouvelle configuration.
8. **Utiliser la FillBar pour indiquer la progression:** À chaque mise à jour de l'état du puzzle, recalculer le pourcentage de complétion. Par exemple, comptez combien de pièces sont à la bonne place. Dans le puzzle de taquin, on peut dire qu'une pièce est à sa place si la valeur == index+1 (sauf la dernière case qui est le vide). Calculez `progress = nombre_de_pieces_bien_placees / 15f` (15 pièces à placer). Mettez à jour un état `progressState` de type `Float` et passez-le à votre composant `FillBar(progressState)`. Ainsi, la barre se remplira au fur et à mesure que le puzzle se rapproche de la solution.
9. **Détecter la fin du puzzle:** Vous devriez aussi vérifier après chaque mouvement si le puzzle est résolu (par exemple si la liste est dans l'ordre 1,2,...,15,0). Si oui, vous pouvez :
10. Afficher un message de félicitations (un simple Text ou un **Dialog** Compose).
11. Naviguer vers un écran de fin/victoire si vous avez prévu un autre écran (ex: `navController.navigate("victory")`).
12. Réinitialiser ou proposer de rejouer.

Pour rester dans l'esprit du TP, afficher un message ou changer visuellement l'écran (feux d'artifice en ASCII ou autre) peut suffire. Vous pouvez utiliser un état boolean `isFinished` pour conditionner l'affichage de ce message ou la navigation. 6. **Composer GameScreen avec les éléments:** Assemblons maintenant dans le composable `GameScreen` les différentes parties : la barre de progression en haut, la grille du puzzle, et éventuellement des éléments d'UI supplémentaires (bouton "Recommencer", compteur de coups, etc. si désiré). Par exemple :

```
@Composable
fun GameScreen(navController: NavController) {
    // Etats du puzzle
    var tiles by remember { mutableStateOf(shuffledTiles()) }  // liste
mélangée de tuiles au démarrage
    var movesCount by remember { mutableStateOf(0) }
    // Calcul de progression (dans [0,1])
    val progress = remember(tiles) {

calculateCompletionPercent(tiles) // ex: renvoie un Float entre 0f et 1f
    }

    Column(modifier = Modifier.fillMaxSize().padding(16.dp)) {
```

```
        // Barre de progression en haut
        Text(text = "Progression : ${(progress * 100).toInt()}%", style =
MaterialTheme.typography.titleMedium)
        FillBar(progress = progress, modifier = Modifier.padding(vertical =
8.dp))
        // Grille du puzzle
        GameBoard(tiles = tiles) { clickedIndex ->
            // Logique de déplacement
            val emptyIndex = tiles.indexOf(0)
            if (isAdjacent(clickedIndex, emptyIndex)) {
                tiles = tiles.toMutableList().also {
                    it[emptyIndex] = tiles[clickedIndex]
                    it[clickedIndex] = 0
                }
                movesCount += 1
            }
            // Vérifier si terminé
            if (isPuzzleSolved(tiles)) {
                // Exemple: affichage d'un message de victoire
                Toast.makeText(LocalContext.current, "Puzzle résolu en
$movesCount coups!", Toast.LENGTH_LONG).show()
                // ou navigation vers un autre écran de fin, etc.
            }
        }
    }
}
```

Dans ce code: - `shuffledTiles()` serait une fonction utilitaire qui génère une liste 1-15,0 mélangée de façon solvable (pour un vrai taquin, il faut une permutation paire). - `calculateCompletionPercent(tiles)` calcule le pourcentage de pièces bien placées. - `isAdjacent(a, b)` vérifie si deux index de la liste sont voisins sur la grille (à implémenter selon les dimensions). - `isPuzzleSolved(tiles)` renvoie vrai si la liste est dans l'ordre de victoire.

Le `GameBoard` est appelé avec la liste courante et une lambda sur clic. À chaque clic valide, on met à jour la liste `tiles` (ce qui déclenche la recomposition de `GameBoard` et de `FillBar` via Compose car on a utilisé des états `remember`). On incrémente aussi un compteur de mouvements `movesCount` (juste informatif). Enfin, si le puzzle est terminé, on présente un feedback (ici un Toast pour faire simple, mais on pourrait mettre un Text visible conditionnellement dans la colonne, ou naviguer ailleurs).

1. **Tester l'application complète:** Compilez et exécutez. Vous devriez voir le SplashScreen, puis l'écran de puzzle avec la barre de progression à 0% (ou très bas) et la grille mélangée. En cliquant sur les tuiles, elles se déplacent si la logique est correcte, la barre de progression augmente petit à petit, et le pourcentage affiché aussi. En fin de puzzle, le message de victoire apparaît.
   Si quelque chose ne fonctionne pas, utilisez **Android Studio Debugger** ou des logs (`Log.d`) pour inspecter l'état des listes lors des clics. Vérifiez particulièrement la logique d'adjacence et la recomposition de l'état (le puzzle ne se mettrait pas à jour si vous n'avez pas utilisé un type mutable observé par Compose, d'où l'utilisation de `by remember { mutableStateOf(...) }` qui délègue à un MutableState).

☞ **Bonnes pratiques architecture:** Pour un vrai projet, la logique du puzzle (mélange, déplacements, condition de victoire) devrait idéalement résider dans un **ViewModel** (du package androidx.lifecycle) et non dans le composable directement. Le composable `GameScreen` se contenterait d'observer l'état exposé par le ViewModel (via des `MutableStateFlow` ou `LiveData`) et d'appeler des méthodes du ViewModel pour les actions (comme déplacer une tuile). Cela rend le code plus testable et maintenable. Étant donné qu'il s'agit d'un TP pour débutant, vous pouvez gérer l'état localement comme on l'a fait pour comprendre le fonctionnement de Compose, mais ayez conscience qu'on tend vers une séparation UI/Logique pour des applications plus complexes.

Conclusion

En suivant ce tutoriel, vous avez créé une base de projet Android moderne en Kotlin et Jetpack Compose pour une application de puzzles, comprenant : un **SplashScreen** personnalisé affichant un drapeau lié à la lettre S, un composant **FillBar** pour indiquer la progression, et une esquisse d'**écran de jeu** avec une grille de puzzle interactive. Tout au long du chemin, nous avons appliqué de bonnes pratiques : - Structuration du code en fichiers et packages clairs (séparation des écrans, composants, thème, etc.). - Utilisation de **Jetpack Compose** pour construire l'UI de manière déclarative et réactive, en profitant de sa gestion de l'état (@Composable, remember, mutableStateOf). - Adoption de composants Material3 (thème, couleurs) et attention à l'accessibilité (contentDescription). - Introduction de la **Navigation Compose** pour gérer les écrans de l'application de façon propre. - Ajout de commentaires et d'explications dans le code pour faciliter la compréhension, ce qui est essentiel en contexte pédagogique.

N'hésitez pas à faire évoluer cette base de projet. Par exemple, vous pouvez ajouter un écran d'accueil avec menu, un sélecteur de niveau de puzzle, ou améliorer la convivialité (animations, sons, etc.). Avec Jetpack Compose, vous avez un outil puissant pour créer des interfaces utilisateur richement et plus facilement qu'avec les layouts traditionnels Android. Bonne continuation dans vos puzzles et bon code !

Sources : Pour aller plus loin avec Jetpack Compose, vous pouvez consulter la documentation officielle Android (en particulier sur la [configuration de Compose dans Gradle](#) [2] [3] et [la navigation Compose](#) [5]), ainsi que les guides sur les composants Material (ex. indicateurs de progression [8]). Ces ressources vous aideront à approfondir les bonnes pratiques que nous avons effleurées dans ce tutoriel. Bon développement !

---

[1] [2] [3] [4] Quick start | Jetpack Compose | Android Developers
https://developer.android.com/develop/ui/compose/setup

[5] Navigation with Compose | Jetpack Compose | Android Developers
https://developer.android.com/develop/ui/compose/navigation

[6] International maritime signal flags - Wikipedia
https://en.wikipedia.org/wiki/International_maritime_signal_flags

[7] Splash Screen with Jetpack Compose: Side-Effects & How to Use Them | by Daniel Dimovski | Medium
https://dimovski-d.medium.com/splash-screen-with-jetpack-compose-side-effects-in-compose-how-to-use-them-2a90eb6e1d34

[8] [9] Progress indicators | Jetpack Compose | Android Developers
https://developer.android.com/develop/ui/compose/components/progress