ChatGPT

Documentation du projet de jeu de puzzle (Android, Kotlin, Jetpack Compose)

Introduction

Ce document présente un jeu de puzzle **Android** développé en **Kotlin** avec **Jetpack Compose**. L'objectif de ce projet est de proposer un petit jeu de puzzle (de type *jigsaw puzzle*) dont les pièces sont récupérées dynamiquement depuis une **API web** en fonction d'une adresse email utilisateur et d'un niveau de difficulté choisi. L'application illustre l'utilisation de Jetpack Compose pour construire une interface moderne sans *XML*, en incluant un écran d'accueil animé et une interface de jeu interactive. Elle démontre également de bonnes pratiques de développement avec Compose, telles que la **navigation déclarative**, la **gestion d'état** et les **effets de bord contrôlés**.

Concrètement, le flux de l'application est le suivant : un écran d'accueil (*SplashScreen*) affiche le drapeau de la Sierra Leone dessiné en Compose et une barre de chargement animée, avec un bouton permettant de démarrer. Une fois le jeu lancé, l'écran principal (*GameScreen*) permet à l'utilisateur de saisir son email, de sélectionner la difficulté (par exemple la taille du puzzle), puis de récupérer un puzzle depuis l'API (*PuzzleFetcher*). Le puzzle est représenté par une image découpée en plusieurs pièces (*Puzzle* et *JigsawPiece*). Ces pièces sont affichées mélangées sur une grille (*Jigsaw*), et le joueur doit les échanger pour reconstituer l'image. La logique de permutation des pièces, la vérification des pièces bien placées (avec bordures colorées) et la détection de la fin du jeu sont gérées par une classe dédiée (*JigsawManager*). L'interface utilise **Jetpack Compose Navigation** pour passer de l'écran d'accueil au jeu, et exploite des composants Composables réutilisables et sans état pour une meilleure maintenabilité.

Le document est structuré de façon pédagogique : chaque section correspond à un composant ou fichier du projet, avec son code source complet, des explications claires et des commentaires en français. Les bonnes pratiques utilisées (telles que `LaunchedEffect`, composables sans état, utilisation de `remember` / `mutableStateOf`, etc.) sont soulignées au fil du texte et récapitulées en fin de document. L'ensemble vise à guider un débutant pas à pas dans la compréhension du code et des concepts mis en œuvre.

Sommaire

8. **JigsawPiece.kt** – Composable d'une pièce du puzzle (chargement d'image avec état de chargement)
9. **Jigsaw.kt** – Affichage de la grille de pièces du puzzle selon un ordre (permutation)
10. **JigsawManager.kt** – Logique du puzzle (gestion d'état, échanges de pièces, bordures, validation de fin)
11. **GameScreen.kt** – Écran principal du jeu intégrant PuzzleFetcher et la grille du puzzle
12. **Fichiers build.gradle** – Configuration Gradle pour Jetpack Compose et la navigation
13. **Bonnes pratiques appliquées** – Récapitulatif des pratiques de développement utilisées
14. **Conclusion** – Synthèse et pistes d'amélioration

MainActivity.kt – Navigation avec Jetpack Compose

La classe **MainActivity** initialise l'interface Compose de l'application et définit la navigation entre les écrans (SplashScreen et GameScreen). Grâce à **Jetpack Compose Navigation**, on peut déclarer un **NavHost** avec des routes correspondant aux écrans de l'application. Dans ce projet, nous avons deux destinations : `"splash"` pour l'écran d'accueil, et `"game"` pour l'écran du jeu. Le code suivant montre comment l'`Activity` utilise `setContent` pour définir le contenu de l'application en Compose, et configure le NavController et le NavHost. Un **thème Material** enveloppe l'ensemble pour appliquer le style Material Design par défaut.

```
package com.example.puzzlegame

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material3.MaterialTheme
import androidx.compose.runtime.Composable
import androidx.compose.ui.platform.LocalContext
import androidx.navigation.compose.rememberNavController
import androidx.navigation.NavType
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Initialisation de la composante principale de l'application
        setContent {
            MaterialTheme {
                // Création du contrôleur de navigation Compose
                val navController = rememberNavController()
                // Définition du graph de navigation avec deux destinations:
Splash et Game
                NavHost(navController = navController, startDestination =
"splash") {
                    // Écran d'accueil (SplashScreen)
```

```
                    composable(route = "splash") {
                        SplashScreen(
                            onStart = {
                                // Navigation vers l'écran de jeu lorsqu'on
appuie sur "Démarrer"
                                navController.navigate("game")
                            }
                        )
                    }
                    // Écran principal du jeu (GameScreen)
                    composable(route = "game") {
                        GameScreen()
                    }
                }
            }
        }
    }
}
```

Explications : Dans `onCreate`, au lieu d'utiliser un layout XML, on appelle `setContent` pour définir l'UI en Compose. Le contenu est englobé dans `MaterialTheme` (ici on utilise le thème Material3 par défaut) afin de bénéficier des styles Material Design. On crée ensuite un `navController` via `rememberNavController()`. Le `NavHost` configure les routes de l'application : la route initiale `"splash"` affiche le composable `SplashScreen`, et la route `"game"` affiche le composable `GameScreen`. Le paramètre `onStart` passé à `SplashScreen` est une lambda qui utilise `navController.navigate("game")` pour passer à l'écran du jeu lorsque l'utilisateur clique sur le bouton démarrer. Ainsi, aucune transaction de fragment ou d'`Activity` n'est requise : la navigation entre écrans est déclarative et gérée par Compose Navigation.

SplashScreen.kt – Écran d'accueil animé

Le composant **SplashScreen** correspond à l'écran de démarrage de l'application. Son rôle est d'afficher une animation d'introduction de manière ludique avant d'accéder au jeu. Dans notre cas, le SplashScreen montre le **drapeau de la Sierra Leone** dessiné grâce à Compose, une **barre de progression animée** simulant un chargement, et un bouton *Commencer* (ou *Démarrer*) pour passer à l'écran suivant. Ce composable utilise un `onStart` passé en paramètre pour notifier l'activité ou le contrôleur de navigation que l'utilisateur souhaite passer à la suite.

Le code ci-dessous illustre l'implémentation de `SplashScreen`. On y retrouve l'utilisation de `LaunchedEffect` pour animer progressivement la barre de chargement. La barre (composable `FillBar`) va de 0% à 100% en quelques secondes, puis le texte du bouton "Commencer" apparaît pleinement (on pourrait choisir de n'afficher le bouton qu'après le chargement, selon le design souhaité). L'utilisateur peut cliquer sur le bouton à tout moment pour appeler la lambda `onStart` et naviguer vers l'écran de jeu.

```kotlin
package com.example.puzzlegame.ui

import androidx.compose.foundation.layout.*
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp

@Composable
fun SplashScreen(onStart: () -> Unit) {
    // État local pour la progression de la barre (0.0f à 1.0f)
    var progress by remember { mutableStateOf(0f) }

    // Effet lancé une seule fois à l'entrée dans la Composition pour animer la
barre
    LaunchedEffect(Unit) {
        // Simulation d'un chargement progressif
        for (i in 0..100) {
            progress = i / 100f       // met à jour la progression
            kotlinx.coroutines.delay(20)  // petite pause pour animer (20ms par
étape)
        }
        // Optionnel: on pourrait naviguer automatiquement après le chargement
si désiré
    }

    // Mise en page de l'écran d'accueil
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(32.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        // Affiche le drapeau de la Sierra Leone
        SierraFlag(modifier = Modifier.fillMaxWidth().padding(16.dp))
        Spacer(modifier = Modifier.height(24.dp))
        // Barre de progression animée avec le pourcentage
        FillBar(progress = progress, modifier = Modifier.fillMaxWidth(0.8f))
        Spacer(modifier = Modifier.height(48.dp))
        // Bouton "Commencer" pour démarrer le jeu
        Button(onClick = { onStart() }) {
            Text("Commencer le puzzle")
        }
```

```
    }
}
```

Explications : On utilise une colonne (`Column`) centrée verticalement et horizontalement pour disposer les éléments du SplashScreen. Le composable personnalisé `SierraFlag` est appelé pour dessiner le drapeau (ce composant est détaillé dans la section suivante). Ensuite, `FillBar` est utilisé pour afficher une barre de chargement dont la longueur est déterminée par la variable d'état `progress`. Grâce à `LaunchedEffect`, cette variable est incrémentée progressivement de 0 à 1 sur une durée d'environ 2 secondes (100 * 20ms). L'effet `LaunchedEffect` permet de lancer une coroutine liée au cycle de vie du composable afin d'exécuter ce type d'animation ou d'action asynchrone une fois à l'affichage [1]. Enfin, un bouton **Material3** (`Button`) avec le texte "Commencer le puzzle" est affiché. Lorsque l'utilisateur clique dessus, la lambda `onStart()` est exécutée, ce qui déclenchera la navigation vers l'écran de jeu (comme configuré dans `MainActivity`).

Notons que le SplashScreen est **stateless** vis-à-vis de l'extérieur : il ne gère pas de navigation directement, il se contente d'appeler `onStart` fourni par l'appelant. Il gère uniquement un état interne local (`progress`) pour l'animation. Ce composant est ainsi réutilisable et facile à tester car il n'a pas de dépendances externes directes, mis à part le callback de navigation.

SierraFlag.kt – Dessin du drapeau de la Sierra Leone

Le fichier **SierraFlag.kt** contient un composable qui dessine le drapeau de la Sierra Leone. C'est un exemple de dessin simple avec Jetpack Compose, utilisant l'API Canvas pour dessiner des formes géométriques. Le drapeau de la Sierra Leone se compose de trois bandes horizontales de tailles égales (verte en haut, blanche au milieu, bleue en bas). Ce composable n'a pas d'état interne : il se contente de dessiner le drapeau dans l'espace qui lui est alloué.

Voici le code source de `SierraFlag` :

```
package com.example.puzzlegame.ui

import androidx.compose.foundation.Canvas
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.geometry.Size
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.drawscope.drawRect

@Composable
fun SierraFlag(modifier: Modifier = Modifier) {
    // Canvas permet de dessiner directement des formes
    Canvas(modifier = modifier) {
        // Calcul de la hauteur d'une bande (1/3 de la hauteur totale)
        val stripeHeight = size.height / 3f
        // Bande verte (en haut)
```

```
        drawRect(
            color = Color(0xFF1EB53A),            // vert (couleur hexadécimale
du drapeau)
            size = Size(size.width, stripeHeight),// pleine largeur, 1/3 hauteur
            topLeft = androidx.compose.ui.geometry.Offset(0f, 0f)
        )
        // Bande blanche (au milieu)
        drawRect(
            color = Color.White,
            size = Size(size.width, stripeHeight),
            topLeft = androidx.compose.ui.geometry.Offset(0f, stripeHeight)
        )
        // Bande bleue (en bas)
        drawRect(
            color = Color(0xFF0072C6),            // bleu
            size = Size(size.width, stripeHeight),
            topLeft = androidx.compose.ui.geometry.Offset(0f, 2 * stripeHeight)
        )
    }
}
```

Explications : Le composable utilise `Canvas` de Compose pour dessiner directement sur une surface. À l'intérieur du lambda de dessin, on utilise la taille disponible (`size`) pour calculer la hauteur de chaque bande (un tiers de la hauteur totale). On appelle `drawRect` trois fois pour dessiner trois rectangles de la largeur totale du canvas et de hauteur `stripeHeight` : le premier en vert (positionné à l'origine `(0,0)`), le second en blanc (positionné à `y = stripeHeight`), et le dernier en bleu (positionné à `y = 2 * stripeHeight`). Les couleurs utilisées correspondent aux couleurs officielles du drapeau de la Sierra Leone. Ce composable est purement visuel et **sans état** : quel que soit l'écran ou le conteneur où on le place, il dessinera toujours le même drapeau. Son `Modifier` peut être spécifié à l'appel pour ajuster sa taille (par exemple, on l'a appelé dans le SplashScreen avec `fillMaxWidth().height(...)` pour qu'il prenne la largeur de l'écran et une hauteur proportionnelle).

FillBar.kt – Barre de progression personnalisée avec texte

La composante **FillBar** réalise une barre de progression horizontale affichant en son centre le pourcentage d'avancement. C'est un composable réutilisable qui prend en paramètre un niveau de progression (`progress`) sous forme de `Float` (allant de 0.0 à 1.0). Ici, nous l'utilisons dans le SplashScreen pour montrer une animation de chargement. La barre est dessinée de façon personnalisée pour illustrer la flexibilité de Compose : on utilise deux `Box` imbriquées pour créer une barre grise en arrière-plan et une barre colorée par-dessus dont la largeur dépend de la progression. Un texte est centré pour afficher le pourcentage (arrondi à l'entier le plus proche).

```
package com.example.puzzlegame.ui

import androidx.compose.foundation.background
import androidx.compose.foundation.layout.*
```

```
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.MaterialTheme.colorScheme
import androidx.compose.material3.MaterialTheme.typography
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp

@Composable
fun FillBar(progress: Float, modifier: Modifier = Modifier) {
    // Contrainte pour que la valeur de progression soit entre 0 et 1
    val clampedProgress = progress.coerceIn(0f, 1f)
    // Conteneur de la barre (barre de fond grise arrondie)
    Box(
        modifier = modifier
            .height(20.dp) // hauteur fixe de la barre
            .background(color = Color.LightGray, shape =
RoundedCornerShape(10.dp))
    ) {
        // Barre de remplissage colorée par-dessus, largeur proportionnelle à la
progression
        Box(
            modifier = Modifier
                .fillMaxHeight()
                .fillMaxWidth(fraction = clampedProgress) // fraction de la
largeur totale
                .background(color = Color(0xFF0066CC), shape =
RoundedCornerShape(10.dp))
        )
        // Texte affichant le pourcentage (centré dans la Box par défaut)
        Text(
            text = "${(clampedProgress * 100).toInt()}%",
            modifier = Modifier.align(Alignment.Center),
            color = Color.White,
            style = MaterialTheme.typography.bodySmall
        )
    }
}
```

Explications : La fonction `FillBar` utilise un `Box` principal qui sert de fond de barre (en gris clair, avec des coins arrondis de rayon 10dp). À l'intérieur, un second `Box` représente la partie remplie de la barre : on lui applique `fillMaxWidth(fraction = progress)` pour que sa largeur soit un pourcentage de la largeur du parent, égal à la progression. Sa hauteur remplit la hauteur du parent (`fillMaxHeight()`) et il a le même shape arrondi, de sorte que la barre remplie épouse les coins arrondis du fond. On lui donne une couleur bleue personnalisée (ici un bleu moyen, code hex `#0066CC`). Ensuite, un composant `Text` est

placé avec `Modifier.align(Alignment.Center)` pour se superposer au centre du `Box` parent. Il affiche la valeur de `progress` en pourcentage (convertie en entier). Le texte est en blanc pour être lisible sur le fond coloré, et utilise le style `bodySmall` du thème Material3 pour être de taille appropriée.

Ce composable est **stateless** : il ne stocke aucun état en interne, il se contente d'afficher la vue en fonction du paramètre `progress` reçu. Ainsi, c'est le parent (par exemple SplashScreen) qui gère l'évolution de `progress` et provoque la recomposition de `FillBar`. Ce principe est conforme à la bonne pratique de *state hoisting* (remontée d'état) : la logique de calcul du pourcentage est externalisée, et `FillBar` n'est qu'une vue pure [2].

PuzzleFetcher.kt – Formulaire de récupération d'un puzzle

Le composant **PuzzleFetcher** est un élément clé de l'écran de jeu : il fournit une petite interface permettant à l'utilisateur de **saisir son email**, de **choisir la difficulté du puzzle** via un slider, puis de **lancer la récupération** du puzzle auprès de l'API. Ce composant gère plusieurs états : le texte de l'email saisi, la valeur de difficulté sélectionnée, un état de validation (email valide ou non), et l'état de chargement (en cours de récupération ou non). Il est conçu de manière à ne pas contenir l'état final du puzzle lui-même, mais à transmettre celui-ci via un callback `onNewPuzzle` lorsque l'API a répondu. Ainsi, le puzzle chargé sera « remonté » au niveau supérieur (GameScreen) qui pourra alors l'afficher.

Le code suivant implémente `PuzzleFetcher`. Pour simplifier, nous simulons l'appel réseau avec un délai (comme si on attendait la réponse de l'API) et nous générons un objet `Puzzle` factice. Dans une application réelle, la fonction `fetchPuzzleFromApi(email, difficulty)` serait implémentée pour effectuer une requête HTTP et récupérer les données du puzzle.

```
package com.example.puzzlegame.ui

import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.text.input.TextFieldValue
import androidx.compose.ui.unit.dp
import kotlinx.coroutines.delay

@Composable
fun PuzzleFetcher(onNewPuzzle: (Puzzle) -> Unit) {
    // État pour l'email saisi
    var email by remember { mutableStateOf("") }
    // État pour la difficulté (nombre de pièces par côté du puzzle)
    var difficulty by remember { mutableStateOf(3) }
    // État pour indiquer si on est en cours de chargement (appel API en cours)
    var isLoading by remember { mutableStateOf(false) }
    // État pour un éventuel message d'erreur
    var errorMessage by remember { mutableStateOf<String?>(null) }
```

```
    // Validation basique de l'email (doit contenir "@" et un ".")
    val isEmailValid = email.contains("@") && email.contains(".")

    Column(modifier = Modifier.fillMaxWidth().padding(16.dp)) {
        // Champ de texte pour l'email
        OutlinedTextField(
            value = email,
            onValueChange = {
                email = it
                errorMessage = null  // réinitialise l'erreur lorsqu'on modifie
l'email
            },
            label = { Text("Email") },
            isError = !isEmailValid && email.isNotEmpty(),
            modifier = Modifier.fillMaxWidth()
        )
        if (!isEmailValid && email.isNotEmpty()) {
            Text(
                text = "Adresse email invalide",
                color = MaterialTheme.colorScheme.error,
                style = MaterialTheme.typography.bodySmall
            )
        }
        Spacer(modifier = Modifier.height(16.dp))
        // Slider pour la difficulté (de 2 à 5)
        Text(text = "Difficulté : $difficulty x $difficulty pièces")
        Slider(
            value = difficulty.toFloat(),
            onValueChange = { difficulty = it.toInt() },
            valueRange = 2f..5f,
            steps = 3  // valeurs entières 2, 3, 4, 5
        )
        Spacer(modifier = Modifier.height(8.dp))
        // Bouton de chargement du puzzle
        Button(
            onClick = {
                if (!isEmailValid) {
                    errorMessage = "Veuillez entrer un email valide."
                    return@Button
                }
                // Démarre le chargement du puzzle (appel réseau simulé)
                isLoading = true
                errorMessage = null
            },
            enabled = !isLoading,
            modifier = Modifier.fillMaxWidth()
        ) {
```

```
            Text(if (isLoading) "Chargement..." else "Charger le puzzle")
        }
        // Affichage d'un indicateur de progression pendant le chargement
        if (isLoading) {
            LaunchedEffect(Unit) {
                try {
                    // Simulation: attente de 2 secondes pour imiter un appel
réseau
                    delay(2000)
                    // Génération d'un puzzle factice une fois "récupéré"
                    val puzzle = Puzzle.generateDummyPuzzle(difficulty)
                    onNewPuzzle(puzzle)    // envoie le puzzle récupéré vers
l'extérieur
                } catch (e: Exception) {
                    errorMessage = "Erreur lors de la récupération du puzzle."
                } finally {
                    isLoading = false
                }
            }
            LinearProgressIndicator(modifier = Modifier.fillMaxWidth())
        }
        // Affichage d'un message d'erreur s'il y a lieu
        errorMessage?.let { msg ->
            Text(text = msg, color = MaterialTheme.colorScheme.error)
        }
    }
}
```

Explications : Le composable est structuré en colonne avec du padding. Le champ de texte (`OutlinedTextField`) est lié à la variable d'état `email`. Une validation sommaire vérifie que l'email contient au moins un "`@`" et un "`.`". Si l'utilisateur a commencé à saisir quelque chose et que la validation échoue, on affiche un texte d'erreur sous le champ. Le slider de difficulté utilise une valeur entière entre 2 et 5 (ces valeurs correspondent au nombre de pièces par côté du puzzle, par exemple 3 signifie un puzzle 3x3 donc 9 pièces). Le texte au-dessus du slider indique la taille actuelle sélectionnée.

Le bouton **"Charger le puzzle"** est cliquable seulement si on n'est pas déjà en cours de chargement (`enabled = !isLoading`). Au clic, on vérifie la validité de l'email : si invalide, on met à jour un message d'erreur et on ne poursuit pas. Si tout est bon, on met `isLoading = true`, on réinitialise les anciens messages d'erreur, et on déclenche l'opération de chargement du puzzle.

Lorsque `isLoading` passe à `true`, un `LaunchedEffect` est lancé (avec `Unit` en clé pour qu'il s'exécute immédiatement une seule fois) afin de faire l'appel réseau de manière asynchrone sans bloquer l'UI. Ici, on utilise `delay(2000)` pour simuler un temps de réponse. Après ce délai, on crée un puzzle fictif via `Puzzle.generateDummyPuzzle(difficulty)` (nous définirons cette fonction dans la classe Puzzle pour générer une liste de pièces factices). Puis on invoque `onNewPuzzle(puzzle)` afin de transmettre le puzzle récupéré au niveau supérieur (GameScreen). Quoi qu'il arrive (succès ou exception), on met `isLoading = false` dans le bloc `finally` pour cacher l'indicateur de chargement et réactiver l'UI. En

parallèle, pendant que `isLoading` est vrai, on affiche un `LinearProgressIndicator` (barre de progression horizontale indéterminée) sous le bouton pour signaler visuellement que le chargement est en cours. Si une erreur s'est produite (par exemple, une exception dans l'appel réseau), on affiche le message d'erreur stocké dans `errorMessage` en rouge en bas du formulaire.

Ce composant **gère son état local** (email, difficulté, chargement, erreur) mais il **externalise l'état global du puzzle**. En effet, une fois le puzzle obtenu, il est passé via le callback `onNewPuzzle` plutôt que d'être affiché directement ici. Cette conception respecte la séparation des responsabilités : `PuzzleFetcher` s'occupe de la saisie et du fetch, puis c'est le parent qui décidera comment utiliser le `Puzzle`. On applique ici le principe de **state hoisting** (remontée d'état) et de composable sans état interne pour la donnée métier principale [2]. Le composable en lui-même reste réutilisable pour n'importe quelle logique de récupération de puzzle similaire, en passant une fonction de traitement différente si nécessaire.

Puzzle.kt – Représentation des données du puzzle

La classe de données **Puzzle** sert à modéliser le puzzle récupéré depuis l'API. Elle contient les informations nécessaires pour afficher et résoudre le puzzle : typiquement la liste des **morceaux de l'image** (ici représentés par des URLs d'images qui seront chargées), et la dimension de la grille du puzzle (par exemple 3 pour un puzzle 3x3). Selon l'API réelle, on pourrait avoir d'autres champs (un identifiant, le nom du puzzle, etc.), mais pour notre projet nous nous limitons à ces éléments.

Ci-dessous le code de `Puzzle.kt`. On y inclut également une fonction utilitaire `generateDummyPuzzle` pour générer un puzzle factice à partir d'une difficulté donnée (cette fonction nous aide à simuler une réponse de l'API dans notre démonstration sans accès réseau effectif).

```
package com.example.puzzlegame.model

data class Puzzle(
    val imageUrls: List<String>,  // URLs de chaque pièce du puzzle (ordonnées
par position correcte)
    val gridSize: Int             // taille de la grille (ex: 3 pour 3x3)
) {
    companion object {
        // Génère un puzzle factice de dimension n x n pour test (images
placeholder)
        fun generateDummyPuzzle(n: Int): Puzzle {
            // On utilise des images de placeholder en ligne (par exemple via un
service public)
            val totalPieces = n * n
            val urls = List(totalPieces) { index ->
                // Génère une URL d'image arbitraire avec l'index pour
différencier (ici images placeholder via https://picsum.photos)
                "https://picsum.photos/seed/puzzle$index/300/300"
            }
            return Puzzle(imageUrls = urls, gridSize = n)
        }
```

```
    }
}
```

Explications : `Puzzle` est une **data class** simple. Le champ `imageUrls` est une liste de chaînes de caractères représentant les liens vers les images de chaque pièce. On suppose que ces images sont carrées et toutes de la même taille (par exemple 300x300 pixels comme indiqué dans l'URL placeholder). L'ordre de cette liste correspond à l'ordre *correct* des pièces (c'est-à-dire que l'image à l'index 0 correspond à la position [0,0] dans la grille une fois le puzzle résolu, l'index 1 correspond à la position [0,1], etc.). Le champ `gridSize` indique combien de pièces par côté compose le puzzle (ce qui définit la dimension de la grille : gridSize x gridSize pièces).

La fonction `generateDummyPuzzle(n)` est dans un `companion object` pour pouvoir être appelée sans instance. Elle crée une liste de `totalPieces` URLs en utilisant un service d'images aléatoires (ici *Picsum Photos*, qui retourne une image aléatoire différente pour chaque URL unique). On incorpore l'index du morceau dans l'URL pour varier chaque image. Ainsi, si l'on appelle `Puzzle.generateDummyPuzzle(3)`, on obtiendra un Puzzle de 9 pièces (3x3) avec 9 URL distinctes pointant vers des images aléatoires. **NB:** Dans un contexte réel, l'API fournirait probablement déjà soit l'image globale à découper, soit directement les images découpées. Ici, on simule le second cas où chaque pièce est une image séparée disponible via une URL.

JigsawPiece.kt – Composable pour afficher une pièce du puzzle

JigsawPiece est un composable responsable de l'affichage individuel d'une pièce de puzzle, c'est-à-dire d'une image carrée. Il doit également gérer l'état de chargement de l'image (afficher un indicateur visuel tant que l'image n'est pas chargée). Pour cela, on utilise la bibliothèque **Coil** qui s'intègre bien avec Compose pour charger des images à partir d'URL. En particulier, Coil fournit le composable `SubcomposeAsyncImage` permettant de spécifier un contenu de remplacement (placeholder) pendant le chargement.

Notre `JigsawPiece` prend en paramètre l'URL de l'image de la pièce, ainsi que deux indicateurs booléens optionnels : `isSelected` et `isCorrect`. Ces drapeaux permettent de savoir si la pièce est actuellement sélectionnée par le joueur et si elle est à sa position correcte, afin d'afficher des bordures colorées en conséquence (par exemple une bordure jaune pour la pièce sélectionnée, une bordure verte pour une pièce bien placée). Le composable ajuste son apparence en fonction de ces états mais ne gère pas lui-même la logique de sélection ou de vérification (cela est du ressort du JigsawManager).

Voici l'implémentation de `JigsawPiece` :

```
package com.example.puzzlegame.ui

import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.shape.RectangleShape
import androidx.compose.material3.Card
```

```
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import coil.compose.SubcomposeAsyncImage

@Composable
fun JigsawPiece(
    imageUrl: String,
    isSelected: Boolean,
    isCorrect: Boolean,
    modifier: Modifier = Modifier
) {
    // Choix de la couleur de bordure selon l'état
    val borderColor: Color = when {
        isSelected -> Color.Yellow  // pièce sélectionnée (jaune)
        isCorrect -> Color.Green    // pièce à la bonne place (vert)
        else -> Color.Transparent   // pas de bordure visible autrement
    }

    // On peut utiliser un Card de Material3 pour bénéficier d'une bordure et
d'un fond
    Card(
        modifier = modifier,
        shape = RectangleShape, // pièces carrées sans arrondi
        border = BorderStroke(width = 2.dp, color = borderColor),
        colors = CardDefaults.cardColors(containerColor = Color.LightGray) //
fond gris clair par défaut
    ) {
        Box {
            // Chargement asynchrone de l'image de la pièce
            SubcomposeAsyncImage(
                model = imageUrl,
                contentDescription = "Pièce du puzzle",
                loading = {
                    // Affiche un indicateur de chargement tant que l'image
n'est pas chargée
                    Box(modifier = Modifier.fillMaxSize(), contentAlignment =
Alignment.Center) {
                        CircularProgressIndicator(color = Color.DarkGray)
                    }
                },
                modifier = Modifier.fillMaxSize(),
                contentScale = ContentScale.Crop
            )
        }
```

```
    }
}
```

Explications : Le composable utilise un composant `Card` de Material3 comme conteneur de la pièce. Les `Card` proposent facilement des bordures (`border`) et une couleur de fond (`containerColor` via `CardDefaults.cardColors`). Ici, on définit la forme (`RectangleShape`) sans arrondis pour que les pièces soient bien carrées. La bordure a une épaisseur de 2dp et sa couleur dépend de l'état de la pièce : jaune si la pièce est sélectionnée, verte si elle est correctement placée, sinon transparente (pas de bordure visible). Le fond par défaut est gris clair, ce qui sera visible uniquement si l'image met du temps à charger ou en cas d'erreur, afin d'avoir un aplat neutre.

À l'intérieur de la carte, on utilise `SubcomposeAsyncImage` (fournie par Coil Compose) pour charger l'image à partir de `imageUrl`. Le paramètre `loading` définit le contenu affiché pendant que l'image se charge : nous mettons un `CircularProgressIndicator` centré qui tournera pour indiquer le chargement en cours. Une fois l'image chargée, celle-ci remplacera le contenu de `loading`. Nous utilisons `contentScale = ContentScale.Crop` pour que l'image remplisse bien l'espace de la pièce en recadrant si nécessaire (ainsi chaque pièce reste carrée et couvre toute la carte).

Le composable `JigsawPiece` est **dépendant de l'état qui lui est fourni** (`isSelected` et `isCorrect`), mais il ne gère pas d'état en interne. C'est le parent (la grille de puzzle) qui décide quelle pièce est sélectionnée et quelles pièces sont à la bonne place, et qui appelle `JigsawPiece` en conséquence. Cette séparation permet de changer la logique de sélection sans modifier le composant d'affichage de la pièce.

Jigsaw.kt – Affichage de la grille de puzzle

Le composant **Jigsaw** est chargé de disposer l'ensemble des pièces du puzzle sur une grille carrée. Il reçoit en paramètre le puzzle à afficher (une instance de `Puzzle`) ainsi qu'une instance de `JigsawManager` qui contient l'état courant de la disposition des pièces et la logique pour échanger les pièces. `Jigsaw` va s'appuyer sur les données de `JigsawManager` pour savoir dans quel ordre afficher les pièces et comment réagir aux interactions de l'utilisateur (par exemple, lorsqu'une pièce est cliquée pour être sélectionnée ou échangée).

Dans une implémentation Compose moderne, on pourrait utiliser une LazyVerticalGrid pour générer la grille, mais pour plus de pédagogie et de contrôle nous allons construire la grille "à la main" en utilisant des colonnes et des lignes (`Column` contenant des `Row`). Cela permet de parcourir toutes les positions de 0 à `gridSize*gridSize - 1` et de placer un composable `JigsawPiece` à chaque position correspondante.

Le code de `Jigsaw` est présenté ci-dessous :

```
package com.example.puzzlegame.ui

import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.runtime.Composable
```

```
import androidx.compose.ui.Modifier

@Composable
fun Jigsaw(puzzle: Puzzle, manager: JigsawManager, modifier: Modifier =
Modifier) {
    val gridSize = puzzle.gridSize
    Column(modifier = modifier.fillMaxWidth()) {
        // Parcours des rangées du puzzle
        for (row in 0 until gridSize) {
            Row(modifier = Modifier.fillMaxWidth()) {
                // Parcours des colonnes pour chaque pièce de la rangée
                for (col in 0 until gridSize) {
                    val positionIndex = row * gridSize + col
                    // L'index de l'image à afficher à cette position selon la
permutation courante
                    val imageIndex = manager.arrangement[positionIndex]
                    // Détermination des états de bordure
                    val isCorrect = (imageIndex == positionIndex)
                    val isSelected = (manager.firstSelectedIndex ==
positionIndex)
                    // Composable de la pièce avec gestion du clic
                    JigsawPiece(
                        imageUrl = puzzle.imageUrls[imageIndex],
                        isSelected = isSelected,
                        isCorrect = isCorrect,
                        modifier = Modifier
                            .weight(1f)                // chaque pièce partage
équitablement l'espace horizontal
                            .aspectRatio(1f)           // chaque pièce est
carrée (hauteur = largeur)
                            .clickable {
                                // Gestion du clic sur une pièce : délègue au
JigsawManager
                                manager.onTileClicked(positionIndex)
                            }
                    )
                }
            }
        }
    }
}
```

Explications : On récupère `gridSize` depuis le puzzle pour savoir combien de lignes et colonnes afficher. On crée une `Column` occupant toute la largeur disponible. Pour chaque numéro de ligne de 0 à gridSize-1, on génère une `Row`. À l'intérieur de chaque ligne, on itère sur chaque colonne de 0 à gridSize-1 et on calcule l'index linéaire de la position (`positionIndex = row * gridSize + col`).

Ensuite, on consulte `manager.arrangement` à cet index pour connaître `imageIndex`, c'est-à-dire l'index réel de l'image qui doit être placée à cette position dans l'état actuel du puzzle. Par exemple, si `manager.arrangement[0] == 5`, cela signifie que la pièce qui devrait être à la position 0 (coin haut gauche) est actuellement la pièce d'index 5 (donc la mauvaise pièce est là, le puzzle est mélangé).

On détermine deux booléens : `isCorrect` est vrai si la pièce à cette position est la bonne (c'est-à-dire si `imageIndex == positionIndex`), et `isSelected` est vrai si cette position correspond à l'index de la pièce sélectionnée actuellement par le joueur (`manager.firstSelectedIndex`). Ces indicateurs sont passés à `JigsawPiece` afin qu'il affiche éventuellement une bordure colorée.

Pour l'affichage, chaque `JigsawPiece` est contenu dans un `Modifier.weight(1f)` à l'intérieur d'une Row, ce qui fait que les pièces se répartissent équitablement sur la largeur de l'écran. L'ajout de `aspectRatio(1f)` assure que chaque pièce est rendue dans un conteneur carré (la hauteur de la Row s'adapte pour respecter le ratio 1:1 en fonction de la largeur disponible pour chaque pièce). Ainsi, la grille sera toujours carrée et les pièces uniformément dimensionnées.

Le `Modifier.clickable` entoure chaque pièce pour gérer les interactions utilisateur. Au clic sur une pièce, on appelle `manager.onTileClicked(positionIndex)` – c'est une méthode du `JigsawManager` qui encapsule la logique de sélection/échange des pièces (décrite dans la section suivante). En confiant la gestion du clic au manager, on isole la logique du jeu en dehors de l'UI. L'UI (`Jigsaw`) se contente de déduire l'état visuel (`isSelected`, `isCorrect`) en fonction de l'état du manager, et de demander au manager d'actualiser l'état lorsque l'utilisateur agit.

En résumé, `Jigsaw` est un composable **dumb UI** (*composable présentoir*) qui affiche la grille de pièces en se basant sur l'état du puzzle fourni par `JigsawManager`. Il n'a pas d'état propre et reflète simplement le contenu de `manager.arrangement`. Cela facilite la compréhension : tout le comportement interactif se trouve centralisé dans `JigsawManager`.

JigsawManager.kt – Logique et état du puzzle

La classe **JigsawManager** est le cœur de la logique du puzzle. Elle n'est pas un composant UI (pas un Composable), mais une classe Kotlin standard qui gère : - L'état courant de la permutation des pièces (`arrangement`), c'est-à-dire l'ordre dans lequel les images sont disposées dans la grille. - La sélection d'une première pièce à échanger (`firstSelectedIndex`). - La logique d'échange de deux pièces sélectionnées. - La vérification de la résolution du puzzle (toutes les pièces à la bonne place). - D'éventuelles informations supplémentaires comme le nombre de mouvements effectués (on peut le conserver à titre indicatif).

Cette classe interagit avec l'UI via ses propriétés observables. En utilisant des types mutables observables de Compose (`MutableState` ou `mutableStateListOf`), on s'assure que l'UI (les composables) se mettra à jour automatiquement quand l'état du puzzle change. `JigsawManager` peut être initialisé à chaque nouveau puzzle, et exposer les informations nécessaires à l'UI (par exemple, pour mettre une bordure verte quand une pièce est bien placée, il suffit de comparer index et valeur dans `arrangement` côté UI, comme on l'a fait).

Voici le code de `JigsawManager` :

```
package com.example.puzzlegame.ui

import androidx.compose.runtime.mutableStateListOf

class JigsawManager(puzzle: Puzzle) {
    // Liste observable représentant l'ordre actuel des pièces (permutation des
indices)
    val arrangement = mutableStateListOf<Int>()
    // Index de la première pièce sélectionnée (ou null si aucune pièce n'est en
cours de sélection)
    var firstSelectedIndex: Int? = null
        private set

    init {
        // Initialisation de l'arrangement avec une permutation aléatoire des
indices 0..n-1
        val indices = (0 until puzzle.imageUrls.size).toList()
        arrangement.addAll(indices.shuffled())
        // S'assurer de ne pas démarrer sur un puzzle déjà résolu par hasard
        if (isSolved()) {
            arrangement.shuffle()
        }
    }

    // Appelé lorsqu'une pièce à la position index est cliquée
    fun onTileClicked(positionIndex: Int) {
        if (firstSelectedIndex == null) {
            // Aucune pièce encore sélectionnée, on sélectionne celle cliquée
            firstSelectedIndex = positionIndex
        } else {
            // Une première pièce était déjà sélectionnée, on échange avec la
nouvelle
            swapTiles(firstSelectedIndex!!, positionIndex)
            // Réinitialiser la sélection
            firstSelectedIndex = null
        }
    }

    // Permute deux pièces dans l'arrangement
    private fun swapTiles(index1: Int, index2: Int) {
        if (index1 == index2) return  // si on clique deux fois la même pièce,
on ne fait rien
        val temp = arrangement[index1]
        arrangement[index1] = arrangement[index2]
        arrangement[index2] = temp
    }
```

```
    // Vérifie si le puzzle est résolu (toutes les pièces à leur place)
    fun isSolved(): Boolean {
        // Le puzzle est résolu si chaque valeur est égale à son index (position
correcte)
        return arrangement.indices.all { pos -> arrangement[pos] == pos }
    }
}
```

Explications : Dans le constructeur (`init`), on crée d'abord une liste `indices` contenant tous les indices possibles des pièces (0 à nombreDePieces-1). On mélange cette liste aléatoirement (`shuffled()`) et on l'ajoute à `arrangement`. Ainsi, initialement, les pièces sont dans un ordre aléatoire. On ajoute une petite vérification : si par hasard ce mélange était déjà la solution (très peu probable, mais possible statistiquement), on mélange à nouveau (`shuffle()`) pour s'assurer de ne pas commencer avec le puzzle déjà résolu.

La propriété `arrangement` est une `mutableStateListOf<Int>` – c'est une liste observable par Compose. Toute modification (ajout, échange, etc.) entraîne une notification de Compose et donc une recomposition des composables qui l'utilisent. Cela permet à l'UI (le composant `Jigsaw`) de se mettre à jour automatiquement lorsque l'on échange des pièces.

`firstSelectedIndex` garde la trace de la première pièce que l'utilisateur a cliquée pour un échange en cours. Au début, il n'y a pas de pièce sélectionnée, donc c'est `null`. La méthode `onTileClicked(positionIndex)` est appelée par l'UI lorsqu'une pièce est cliquée. Si aucune pièce n'était sélectionnée (`firstSelectedIndex == null`), on enregistre l'index cliqué comme première sélection. Si au contraire une pièce était déjà sélectionnée, cela signifie que l'utilisateur a cliqué sur une seconde pièce : on appelle alors `swapTiles` pour échanger les deux pièces dans l'`arrangement`. Après l'échange, on remet `firstSelectedIndex` à `null` (l'échange est terminé, plus aucune pièce n'est « active »).

La méthode privée `swapTiles(index1, index2)` effectue l'échange des éléments de la liste `arrangement` aux positions données. On ajoute une condition pour ne rien faire si `index1 == index2` (par exemple, l'utilisateur a double-cliqué la même pièce, ou cliqué accidentellement deux fois la même : dans ce cas pas d'échange à faire). L'échange se fait en trois opérations : sauvegarde temporaire d'une valeur, assignations croisées. Étant donné que `arrangement` est une liste observable, ces changements vont déclencher la recomposition de `Jigsaw`, ce qui mettra à jour la grille affichée.

Enfin, la fonction `isSolved()` retourne `true` si le puzzle est résolu, c'est-à-dire si pour chaque position dans la grille, l'indice de pièce correspond à la même valeur (ex : à la position 0 on a la pièce 0, position 1 la pièce 1, etc.). On l'utilise dans l'initialisation pour éviter un état résolu initial, et on pourrait l'utiliser pour, par exemple, afficher un message de félicitations lorsque `isSolved()` passe à vrai après un échange.

Remarque sur l'architecture : `JigsawManager` illustre le principe de séparation logique/visuelle. C'est en quelque sorte notre **ViewModel/contrôleur** pour le puzzle (bien qu'ici on ne fasse pas appel à la bibliothèque `ViewModel` d'Android, on pourrait tout à fait intégrer `JigsawManager` dans un `ViewModel` Compose). L'UI (`Jigsaw` composable) ne fait qu'appeler les méthodes du manager et lire ses

propriétés. Cela rend le code plus clair et facilite les tests unitaires de la logique de puzzle sans dépendre de l'UI.

GameScreen.kt – Écran principal intégrant le puzzle

Le composant **GameScreen** représente l'écran de jeu complet. C'est sur cet écran que l'utilisateur va : 1. Utiliser `PuzzleFetcher` pour entrer son email, choisir la difficulté et charger un puzzle. 2. Une fois le puzzle chargé, voir apparaître la grille du puzzle (`Jigsaw`) et pouvoir interagir pour résoudre le puzzle.

`GameScreen` orchestre ces deux phases en gardant un état pour le puzzle courant. Au départ, aucun puzzle n'est chargé, on affiche donc l'interface de `PuzzleFetcher`. Quand `PuzzleFetcher` invoque le callback `onNewPuzzle` avec un puzzle, `GameScreen` met à jour son état interne pour enregistrer le puzzle. Dès lors, l'interface recomposée affichera le puzzle et plus le formulaire.

On propose également d'afficher un message de réussite lorsque le puzzle est complété, ainsi qu'un bouton pour éventuellement rejouer (charger un nouveau puzzle). Cela apporte une meilleure expérience utilisateur.

Voici le code de `GameScreen` :

```
package com.example.puzzlegame.ui

import androidx.compose.foundation.layout.*
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.*

@Composable
fun GameScreen() {
    // État du puzzle courant (null si pas encore chargé)
    var currentPuzzle by remember { mutableStateOf<Puzzle?>(null) }
    // Le manager du puzzle, initialisé lorsqu'un Puzzle est chargé
    val jigsawManager = remember(currentPuzzle) {
        currentPuzzle?.let { JigsawManager(it) }
    }

    Column(modifier = Modifier.fillMaxSize()) {
        if (currentPuzzle == null) {
            // Afficher le formulaire de chargement si aucun puzzle n'est
présent
            PuzzleFetcher(onNewPuzzle = { puzzle ->
                currentPuzzle = puzzle
            })
        } else {
            // Un puzzle est chargé, on affiche la grille de jeu
            Text(
```

```
                text = "Puzzle ${currentPuzzle!!.gridSize}x$
{currentPuzzle!!.gridSize}",
                style = MaterialTheme.typography.titleMedium,
                modifier = Modifier.padding(16.dp)
            )
            // Affichage de la grille du puzzle
            Jigsaw(puzzle = currentPuzzle!!, manager = jigsawManager!!,
modifier = Modifier.weight(1f))
            // Si le puzzle est résolu, affichage d'un message de félicitations
et bouton rejouer
            if (jigsawManager.isSolved()) {
                Text(
                    text = "  Puzzle complété !  ",
                    style = MaterialTheme.typography.titleMedium,
                    modifier = Modifier.padding(16.dp)
                )
                Button(onClick = {
                    // Réinitialiser l'état pour rejouer (revenir au formulaire)
                    currentPuzzle = null
                }, modifier = Modifier.padding(16.dp)) {
                    Text("Nouveau puzzle")
                }
            }
        }
    }
}
```

Explications : `GameScreen` utilise un `var currentPuzzle` de type `Puzzle?` initialisé à `null`. Grâce à `remember`, cette variable conservera sa valeur à travers les recompositions (ce qui est nécessaire car lors d'un appel à `currentPuzzle = puzzle`, la fonction recomposera). On crée ensuite un `jigsawManager` via `remember(currentPuzzle) { ... }`. Le paramètre `currentPuzzle` dans remember fait que chaque fois qu'un nouveau puzzle est affecté, un nouveau `JigsawManager` sera créé (pour gérer ce puzzle spécifique). Si `currentPuzzle` est null, on garde `jigsawManager` null (on n'en a pas besoin avant d'avoir un puzzle).

L'UI est une colonne qui occupe tout l'écran. Si `currentPuzzle` est null, on affiche le composant `PuzzleFetcher`. On passe la lambda `onNewPuzzle` qui assigne la valeur reçue à `currentPuzzle`. Notez qu'en assignant `currentPuzzle = puzzle`, on change l'état, ce qui cause la recomposition de `GameScreen` et donc la condition `currentPuzzle == null` deviendra fausse.

Une fois un puzzle chargé, la branche `else` est affichée : on montre un titre avec la taille du puzzle (par exemple "Puzzle 3x3") pour contextualiser. Puis on affiche la grille du puzzle via `Jigsaw`, en passant `currentPuzzle!!` (on force le non-null car on est dans le cas else où currentPuzzle n'est pas null) et un `jigsawManager!!` (également non-null dans ce contexte). On donne à `Jigsaw` un `Modifier.weight(1f)` pour qu'il prenne tout l'espace restant disponible sous le titre, permettant à la

grille de s'étendre (sans `weight(1f)`, la grille prendrait juste la place de son contenu et on pourrait avoir un grand vide en bas si l'écran est plus grand que nécessaire).

Ensuite, on vérifie `if (jigsawManager.isSolved())` : si le puzzle est résolu, on affiche un message de félicitations avec des émojis confettis, ainsi qu'un bouton **"Nouveau puzzle"**. Ce bouton remet `currentPuzzle` à null, ce qui a pour effet de réafficher le formulaire `PuzzleFetcher` (donc l'utilisateur peut charger un autre puzzle).

Ainsi, l'application offre la possibilité d'enchaîner les parties sans redémarrer l'app, en repassant par l'écran de sélection dès qu'un puzzle est fini. On pourrait améliorer en conservant l'email saisi précédemment, ou en proposant directement un autre puzzle de même difficulté, mais cela sort du cadre de base.

Points à noter : `GameScreen` ne connaît pas les détails de comment `JigsawManager` fonctionne, ni comment `PuzzleFetcher` obtient le puzzle – il orchestre simplement les sous-composants en fonction de l'état. C'est un bon exemple de haut niveau de **gestion d'état avec Compose**. En particulier, on voit l'utilisation de `remember` pour maintenir l'état `currentPuzzle` et l'objet `JigsawManager` associé à travers les recompositions. L'utilisation de `remember(currentPuzzle)` assure qu'un nouveau manager est créé uniquement quand le puzzle change, et pas à chaque recomposition, évitant de réinitialiser la permutation du puzzle par inadvertance.

Fichiers build.gradle – Configuration du projet pour Compose et Navigation

Pour que ce projet fonctionne, il faut s'assurer que la configuration Gradle prend en charge Jetpack Compose et les bibliothèques utilisées (Material3, Navigation Compose, Coil, etc.). Deux fichiers sont à configurer : le *build.gradle* au niveau du projet (Project) et celui du module de l'application (Module *app*).

build.gradle (Projet)

Le fichier de configuration du projet doit inclure les dépôts Maven de Google et Maven Central (qui contiennent les artefacts Jetpack). Cela se fait généralement dans le `settings.gradle` ou le build.gradle de projet selon la version de Gradle, mais l'important est d'avoir `google()` et `mavenCentral()` dans la liste des dépôts. De plus, on s'assure d'appliquer le plugin Kotlin Android et d'avoir la bonne version du plugin Compose.

```
// build.gradle (Project-level)
buildscript {
    dependencies {
        classpath "com.android.tools.build:gradle:8.0.2"
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.8.21"
    }
}
allprojects {
    repositories {
```

```
        google()
        mavenCentral()
    }
}
```

Note : Les versions ci-dessus (Gradle plugin 8.0.2, Kotlin 1.8.21) sont données à titre d'exemple et devraient correspondre à une configuration compatible avec Compose en 2025. L'essentiel est d'avoir le plugin Gradle Android récent et le plugin Kotlin correspondant.

build.gradle (Module app)

Le fichier de configuration de l'application est plus important pour intégrer Compose. On doit y activer Compose, définir la version du compilateur Compose, et ajouter les dépendances nécessaires.

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}

android {
    compileSdk 33

    defaultConfig {
        applicationId "com.example.puzzlegame"
        minSdk 21
        targetSdk 33
        versionCode 1
        versionName "1.0"
    }
    // Activation de Jetpack Compose
    buildFeatures {
        compose true
    }
    composeOptions {
        kotlinCompilerExtensionVersion
"1.4.8"  // version du Compiler Extension à adapter selon Compose
    }
    packagingOptions {
        resources {
            excludes += "/META-INF/{AL2.0,LGPL2.1}" // Exclusion d'éventuels
duplicats de licences communes
        }
    }
}

dependencies {
```

```
    // Dépendances Jetpack Compose
    implementation "androidx.activity:activity-compose:1.7.2"
    implementation "androidx.compose.ui:ui:1.4.3"
    implementation "androidx.compose.ui:ui-tooling-preview:1.4.3"
    implementation "androidx.compose.material3:material3:1.1.0"
    // Navigation Compose pour la navigation entre écrans
    implementation "androidx.navigation:navigation-compose:2.6.0"
    // Coil pour le chargement d'images dans Compose
    implementation "io.coil-kt:coil-compose:2.2.2"
    // (Optionnel) Intégration ViewModel avec Compose
    implementation "androidx.lifecycle:lifecycle-viewmodel-compose:2.6.1"
}
```

Détails :

- On applique les plugins `com.android.application` et `org.jetbrains.kotlin.android`.
- `compileSdk` est défini à 33 (ou plus, selon le niveau Android visé). `minSdk 21` est le minimum supporté par Compose (on pourrait mettre 23+ selon les besoins).
- Dans `buildFeatures`, on active `compose = true` pour informer Gradle qu'on utilise Jetpack Compose.
- On précise la version de l'extension du compilateur Compose via `composeOptions.kotlinCompilerExtensionVersion`. Cette version doit correspondre à la version des bibliothèques Compose utilisées. Dans l'exemple, on utilise Compose UI 1.4.3 et Material3 1.1.0, ce qui requiert une version du compiler extension ~1.4.8 (à ajuster en fonction des notes de version Compose).
- Dans les dépendances, on ajoute :
- `activity-compose` qui fournit l'activité optimisée pour Compose (ComponentActivity avec support Compose).
- `compose.ui` et `compose.material3` pour les composants UI et le thème Material3.
- `compose.ui-tooling-preview` pour avoir la fonction preview (et des outils de debug).
- `navigation-compose` pour la navigation entre écrans en Compose.
- `coil-compose` pour charger les images depuis le web dans les composables.

- En option, `lifecycle-viewmodel-compose` si on compte utiliser des ViewModel avec Compose (ici non utilisé explicitement, mais souvent utile en architecture MVVM).

- On n'oublie pas d'inclure `google()` et `mavenCentral()` dans les repositories du module également si nécessaire. Souvent, cela est pris en charge au niveau projet comme montré plus haut.

Avec ces configurations, lors de la compilation, Gradle saura tirer les dépendances de Compose et les compiler correctement. Jetpack Compose n'utilise pas de vue XML, donc pas besoin de fichiers layout ou de `findViewById`. Tout le rendu UI est décrit dans le code Kotlin via les fonctions Composables.

Bonnes pratiques appliquées

Ce projet met en œuvre un certain nombre de **bonnes pratiques** de développement avec Jetpack Compose et Kotlin, que nous résumons ici :

- **Séparation de la logique et de l'UI :** La logique du jeu de puzzle est isolée dans `JigsawManager` (classe Kotlin non-UI) tandis que l'UI (composables `Jigsaw`, `JigsawPiece`, etc.) se contente de refléter l'état fourni par ce manager et de lui déléguer les actions de l'utilisateur. Cette séparation rend les composables plus simples (sans logique métier complexe) et facilite les tests unitaires de la logique (puisqu'on peut tester `JigsawManager` sans interface). Cela s'apparente au pattern MVVM où le UI observe un ViewModel.

- **State Hoisting (remontée d'état) et Composables sans état interne :** La plupart des composables définis (`SierraFlag`, `FillBar`, `JigsawPiece`, `Jigsaw`) ne stockent pas d'état en interne. Ils reçoivent les données dont ils ont besoin via leurs paramètres. Par exemple, `FillBar` affiche une progression qu'on lui passe, `JigsawPiece` affiche une image et une bordure selon des indicateurs passés, etc. Si un composable a besoin d'un état (par exemple `PuzzleFetcher` pour l'email et la difficulté), cet état est géré localement mais le **résultat final** (comme le puzzle récupéré) est remonté au niveau supérieur (`GameScreen`). C'est une application du principe de *stateless composables*, encouragé par la documentation officielle [2], pour maximiser la réutilisabilité et la testabilité. Un composable stateless peut être réutilisé dans d'autres contextes facilement car il n'a pas de dépendance cachée.

- **Gestion de l'état avec `remember` et types MutableState :** On utilise `remember { mutableStateOf(...) }` pour créer des états locaux dans les composables (`progress` dans SplashScreen, `email` / `difficulty` dans PuzzleFetcher, etc.). Le mécanisme `remember` permet de conserver la valeur à travers les recompositions successives d'un même composable. Cela évite de réinitialiser ces valeurs à chaque fois que l'UI se rafraîchit. De même, on utilise `mutableStateListOf` pour la liste observable des pièces dans `JigsawManager`. Les types `MutableState` et `SnapshotStateList` sont observables par Compose : toute modification provoque automatiquement la mise à jour de l'interface aux endroits où ces états sont utilisés. Il est important de noter que l'on évite les collections mutables classiques non observables (`ArrayList` etc.) pour stocker l'état UI, conformément aux recommandations Android [3].

- **Effets side-effect bien encapsulés (LaunchedEffect & rememberCoroutineScope)** : Nous avons eu recours à `LaunchedEffect` pour gérer des opérations asynchrones liées au cycle de vie des composables:

- Dans `SplashScreen`, un `LaunchedEffect(Unit)` démarre l'animation de la barre de progression lorsque l'écran s'affiche. Ce coroutine context est lié à la composable SplashScreen et sera automatiquement annulé si l'écran est recomposé hors de l'arbre (par exemple si on quitte l'écran avant la fin de l'animation), évitant ainsi des actions inutiles ou des fuites de coroutine [1].
- Dans `PuzzleFetcher`, un `LaunchedEffect(Unit)` à l'intérieur du `if (isLoading)` est déclenché lorsqu'on passe en mode chargement. Il exécute la simulation d'appel réseau en coroutine, puis remet à jour l'état (via `onNewPuzzle` ou message d'erreur). Utiliser

`LaunchedEffect` ici garantit que la logique de chargement est exécutée au bon moment et annulée proprement si l'utilisateur navigue ailleurs.

- Nous aurions pu utiliser `rememberCoroutineScope` pour lancer la coroutine du fetch directement dans l'onClick du bouton, ce qui est une autre approche. Ici nous avons choisi `LaunchedEffect` couplé à l'état `isLoading` pour bien découpler la logique d'appel réseau de l'événement d'onClick. Les deux approches sont valides, l'essentiel étant de respecter le cycle de Compose.

- **Compose Navigation pour la navigation d'écran :** Au lieu d'utiliser des Intents ou fragment transactions, nous avons utilisé la navigation Compose pour gérer l'écran de splash et l'écran de jeu. Cela permet de conserver un seul Activity (MainActivity) et de ne recharger que les composables nécessaires. C'est en ligne avec l'architecture single-Activity recommandée. L'utilisation de la navigation Compose assure également la **conservation d'état** des écrans déjà visités si on revenait en arrière (non illustré ici car notre navigation va dans un seul sens sans retour arrière prévu).

- **Utilisation de bibliothèques Jetpack modernes** : Le projet utilise Material3 (la dernière version de Material Design), Coil pour le chargement d'images (privilégié par rapport à Glide/Picasso en contexte Compose pour sa simplicité d'intégration), et les dernières versions de Compose UI. Cela garantit des performances optimales et un support des bonnes pratiques (par exemple, Material3 apporte les composants `OutlinedTextField`, `Card`, etc., adaptés au nouveau design).

- **Commentaires et code clair** : Bien que ce point ne soit pas technique, il s'agit d'une bonne pratique de maintenir un code lisible. Nous avons ajouté des commentaires explicatifs dans le code source pour clarifier l'intention de chaque section. Les noms de fonctions et de variables sont choisis pour être explicites (`PuzzleFetcher`, `isEmailValid`, `firstSelectedIndex`, etc.), ce qui améliore la lisibilité du code, surtout pour un débutant.

En suivant ces bonnes pratiques, l'application gagne en **fiabilité** et en **maintenabilité**. Les composables stateless peuvent être adaptés, la logique peut évoluer sans impacter l'UI et vice-versa, et les comportements asynchrones sont bien maîtrisés. La documentation officielle de Jetpack Compose insiste sur l'importance de ces principes, par exemple en recommandant d'écrire des composables sans effet de bord et en utilisant l'architecture déclarative pour l'état et la navigation [4] [2].

Conclusion

Ce tutoriel a présenté en détail la construction d'une application de puzzle en utilisant **Jetpack Compose**. À travers les différentes sections, nous avons vu comment définir des composables pour chaque partie de l'UI (écran d'accueil, formulaire, puzzle), gérer l'état de l'application de manière réactive, et appliquer des bonnes pratiques de développement moderne sur Android. Le résultat est un projet structuré où l'**interface utilisateur** est décrite de façon déclarative et concise, et où la **logique métier** du puzzle est isolée et testable.

En résumé, les points clés à retenir sont : - La mise en place de la **navigation Compose** pour structurer l'application en écrans sans recourir aux fragments. - La création de **composables sur mesure** (`SierraFlag`, `FillBar`, `JigsawPiece`) pour des éléments UI spécifiques, ce qui démontre la flexibilité de Compose (dessin avec Canvas, custom view). - La gestion du **formulaire** avec Compose (`TextField`, `Slider`, boutons) et la gestion de la validation utilisateur en temps réel. - L'utilisation de

Coil pour le chargement asynchrone d'images dans Compose, avec indications de chargement. - L'implémentation de la logique d'un **jeu de puzzle** (mélange, sélection et échange de pièces, détection de victoire) en Kotlin pur, intégrée de façon réactive avec l'UI. - Les **bonnes pratiques** comme le *state hoisting*, les composables sans état, l'usage approprié de `remember` et des APIs d'effet (`LaunchedEffect`), qui facilitent le développement d'interfaces déclaratives robustes.

Pour aller plus loin, on pourrait améliorer ce projet en intégrant une véritable source de puzzles (une API REST réelle), en gérant la persistance de l'état (par exemple en cas de rotation d'écran on pourrait sauver/restaurer `currentPuzzle` via un ViewModel ou `rememberSaveable`), ou en ajoutant des fonctionnalités comme un chrono, le comptage de coups, etc. Cependant, même dans sa forme actuelle, ce projet sert de base solide pour comprendre comment construire une application complète avec Jetpack Compose en 2025, en tirant parti de l'ensemble de l'écosystème Android moderne.

Nous espérons que ce document vous aura aidé à comprendre la structure et le code de ce projet de puzzle. Bon développement sous Android avec Compose !

---

[1] [4] Side-effects in Compose | Jetpack Compose | Android Developers
https://developer.android.com/develop/ui/compose/side-effects

[2] [3] State and Jetpack Compose | Android Developers
https://developer.android.com/develop/ui/compose/state