ChatGPT

Cours de développement Android avec Kotlin & Jetpack Compose (débutant)

Bienvenue dans ce cours accéléré qui vous apprendra les bases du développement d'une application Android en Kotlin en utilisant Jetpack Compose. L'objectif est de vous fournir en 3 heures les connaissances nécessaires pour coder une application simple (par exemple pour un examen). Nous aborderons pas à pas les concepts essentiels : de la syntaxe Kotlin aux composants d'interface Compose, en passant par la navigation multi-écrans, la gestion d'état, l'utilisation d'un ViewModel, la création de formulaires, les listes dynamiques et même une introduction au stockage local avec Room. Chaque section inclut des explications pédagogiques pour débutants, des extraits de code commentés, des illustrations de l'interface attendue et des conseils pour réussir rapidement lors d'un projet d'examen.

1. Introduction à Kotlin (variables, fonctions, classes, collections)

Kotlin en bref : Kotlin est un langage moderne et concis, officiellement supporté par Google pour Android. Il est statiquement typé (les types sont vérifiés à la compilation) et 100% interopérable avec Java [1]. Kotlin est orienté objet et fonctionnel, ce qui le rend multi-paradigme. Pour nos besoins, nous allons nous concentrer sur les bases de sa syntaxe.

Variables : `var` vs `val`

En Kotlin, on déclare une variable avec les mots-clés `var` ou `val`. La différence est cruciale :
- `var` définit une variable **mutable**, dont la valeur peut être modifiée après l'initialisation [2] [3]. C'est l'équivalent d'une variable normale en Java. Par exemple : `var compteur: Int = 10` déclare un entier modifiable initialisé à 10, et on pourrait plus tard faire `compteur = 15` [3].
- `val` définit une variable **immutable**, qu'on ne peut pas réassigner une fois initialisée (semblable à un `final` en Java) [4] [5]. Si on tente de changer sa valeur, le compilateur produira une erreur. Par exemple : `val langue: String = "Kotlin"` crée une constante qui vaudra toujours "Kotlin" [6].

En résumé, utilisez `val` par défaut pour toutes les valeurs qui ne devraient pas changer, et `var` pour les variables dont le contenu doit évoluer. Cela évite des bugs et clarifie l'intention du code [6]. *Conseil examen :* ne vous trompez pas entre `val` et `var`. Si une variable ne doit pas être modifiée, utilisez `val` – le compilateur vous aidera ainsi à attraper les réassignations non souhaitées.

Types de base et inférence de type

Kotlin gère la plupart des types de base similaires à Java : `Int`, `Long`, `Double`, `String`, etc. Il supporte l'inférence de type, ce qui signifie que vous pouvez souvent omettre le type lors de la déclaration si la valeur initiale est évidente. Par exemple, `val compteur = 10` suffit pour que Kotlin comprenne que c'est un `Int`. Le langage étant statiquement typé, le type d'une variable est figé à la compilation et ne changera pas ; cela sécurise votre code [7].

Kotlin introduit aussi la **null-safety** (sécurité vis-à-vis des valeurs nulles) pour éviter le classique *NullPointerException*. Par défaut, une variable ne peut pas être `null`. Si vous avez besoin d'autoriser

`null`, il faut expliciter un type **nullable** en ajoutant `?` après le type, par exemple : `var adresse: String? = null`. Vous devrez alors gérer les cas nullité (opérateur `?.`, etc.) lorsque vous utiliserez cette variable, ce qui évite bien des crashes [8] [9]. *Conseil :* en cas d'examen pratique, faites attention aux types nullables lors de l'accès à des données potentiellement nulles (par ex. des champs non initialisés) pour éviter des exceptions inattendues.

Fonctions

La déclaration d'une fonction en Kotlin se fait avec le mot-clé `fun` suivi du nom, des paramètres entre parenthèses (chacun avec nom et type), et éventuellement du type de retour après `:`. Exemple d'une fonction qui additionne deux entiers :

```
fun somme(a: Int, b: Int): Int {
    return a + b
}
```

Ici `somme(3, 5)` renverrait 8. Kotlin permet aussi d'**inférer** le type de retour et de conciser la syntaxe pour les fonctions simples. Par exemple, on peut écrire la même fonction sur une seule ligne : `fun somme(a: Int, b: Int) = a + b` [10]. Si une fonction ne renvoie rien (procédure), son type de retour est `Unit` (analogue à `void`), et on peut même omettre `Unit` dans la déclaration.

Les fonctions sont des citoyens de première classe en Kotlin : on peut les stocker dans des variables, les passer en paramètre, etc., mais ceci dépasse le cadre de l'introduction. *Conseil examen :* n'hésitez pas à utiliser des fonctions pour organiser votre code et éviter les répétitions. Même pour une petite app, découper le code en fonctions claires (par ex. `calculerScore()`, `afficherResultat()`, etc.) rendra votre logique plus lisible et facile à déboguer sous la pression du temps.

Classes et objets

Définir une classe en Kotlin est concis. On utilise le mot-clé `class` suivi du nom de la classe. Les propriétés (attributs) et un **constructeur primaire** peuvent être déclarés directement dans l'entête de la classe. Par exemple :

```
class Rectangle(val hauteur: Double, val longueur: Double) {
    val perimetre = (hauteur + longueur) * 2
}
```

Cette classe `Rectangle` possède deux propriétés immuables `hauteur` et `longueur` et calcule une propriété `perimetre` à partir de celles-ci [11]. Vous pouvez créer une instance avec `val rect = Rectangle(5.0, 2.0)` et accéder à `rect.perimetre`. Par défaut, les classes Kotlin ne peuvent pas être héritées (elles sont *final*). Si vous voulez permettre l'héritage, déclarez la classe avec `open class`.

Pour des classes qui servent surtout à transporter des données (comme un modèle avec juste des propriétés), Kotlin propose les **data classes**. En ajoutant le mot-clé `data` devant la classe, le compilateur génère automatiquement pour vous des méthodes utilitaires comme `toString()`, `equals()`, `hashCode()` et `copy()` [12]. Exemple : `data class Tache(val titre: String, val fait: Boolean)`. Une *data class* est idéale pour représenter les entités de votre application (tâche, utilisateur, message, etc.) car elle fournit d'emblée un comparateur structurel et peut être

facilement copiée. *Conseil :* utilisez les data classes sans hésiter pour vos modèles de données, cela vous évitera d'écrire du code passe-partout inutile lors d'un exam.

Collections (listes et autres)

Kotlin dispose d'une panoplie de collections : listes (`List`), listes modifiables (`MutableList`), ensembles (`Set`), maps (`Map`), etc. La syntaxe littérale permet de créer rapidement des collections. Par exemple : `val fruits = listOf("pomme", "banane", "kiwi")` crée une liste immutable de chaînes. Vous pouvez itérer facilement dessus avec une boucle `for` :

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

Vous pouvez aussi vérifier la présence d'un élément avec l'opérateur `in` : `if ("banana" in items) println("La banane est dans la liste")` [13]. Les collections Kotlin offrent des opérations de haute niveau très pratiques, comme `filter`, `map`, `sortedBy`, etc., utilisant souvent des **lambdas** (fonctions anonymes). Par exemple, pour filtrer et transformer une liste :

```
val nombres = listOf(0,1,2,3,4,5,6,7)
val resultats = nombres.filter { it > 5 }.map { it * 2 }
println(resultats)  // Affiche [12, 14]
```

Ici on a filtré les nombres >5 puis multiplié chacun par 2, le tout en une seule ligne [14]. *À savoir :* la plupart des collections en Kotlin existent en version immutable (par défaut) et mutable. Une `List` créée par `listOf` est non modifiable (pas d'ajout/retrait d'élément). Pour une liste mutable, utilisez `mutableListOf`. De même, `mapOf` vs `mutableMapOf`, etc.

Conseils pour l'examen : - Gérez bien vos imports (Kotlin regroupe beaucoup de fonctions utilitaires en extensions, souvent il suffit d'ajouter `import kotlin.collections.*` ou d'autres selon besoin). Android Studio aide généralement en auto-complétant les imports.
- Utilisez les boucles et conditions Kotlin qui sont plus expressives. Par exemple, le `when` remplace avantageusement les switch/cascade de if.

Entraînez-vous un peu sur ces bases Kotlin avant le jour J. Comprendre ces fondamentaux vous fera gagner du temps lors du développement de l'app Android.

2. Introduction à Android Studio et à la structure d'un projet Android

Avant de plonger dans Jetpack Compose, il faut maîtriser l'outil et l'organisation d'un projet Android. **Android Studio** est l'IDE officiel pour développer sur Android. Assurez-vous de l'avoir installé et configuré. Vous créerez un nouveau projet en choisissant le template **"Empty Compose Activity"** (Activité Compose vide). Ce gabarit génère une application de base utilisant Compose avec une activité principale toute prête.

Structure d'un projet Android : dans Android Studio, l'explorateur de projets vous montre typiquement les dossiers suivants pour le module "app" : - `manifests/` : contient le fichier **AndroidManifest.xml**, qui déclare les composants de l'application (activités, permissions, etc.). Par exemple, la `<application>` y englobe une ou plusieurs balises `<activity>`. L'activité principale (MainActivity) doit y être déclarée avec un intent-filter pour le LAUNCHER (point d'entrée) [15]. Si une activité n'est pas listée dans le manifeste, elle ne pourra pas être lancée par le système [16]. - `java/` **ou** `kotlin/` : contient le code source Kotlin de l'application, organisé par packages. C'est ici que se trouve votre `MainActivity.kt` généré par le template, et où vous créerez vos autres classes (activités supplémentaires, data classes, ViewModel, etc.). - `res/` : contient les ressources. On y trouve notamment le sous-dossier `layout/` (pour les layouts XML classiques, toutefois avec Compose on en utilisera peu), `drawable/` (images et formes graphiques), `values/` (fichiers XML définissant des valeurs réutilisables telles que couleurs, styles, dimensions, strings de localisation, etc.). Dans un projet Compose, on aura surtout un fichier `themes.xml` dans `values/` pour le thème de l'appli, même si la plupart du thème est aussi géré en Kotlin via MaterialTheme.

Lorsque vous créez une **Empty Compose Activity**, Android Studio génère une activité Kotlin (par ex. `MainActivity`) qui étend `ComponentActivity`. Au lieu de définir un layout XML dans `onCreate`, elle utilise `setContent { ... }` pour définir l'UI via des composables Compose. Par exemple, le template appelle souvent une fonction `Greeting("Android")` à l'intérieur de `setContent` et fournit un thème Material3 par défaut. Il crée également un fichier `Theme.kt` (dans le package `ui.theme`) contenant la définition du thème Material de l'application (couleurs, typographies, shapes), et une fonction `MyApplicationTheme` ou similaire. Selon le template, vous pourriez aussi voir une fonction annotée `@Preview` pour afficher un aperçu de l'interface directement dans l'IDE.

Lancement et tests sur émulateur/appareil : assurez-vous de savoir exécuter votre projet. Branchez un appareil Android en mode développeur *ou* configurez un émulateur virtuel (AVD) depuis l'AVD Manager d'Android Studio. En appuyant sur le bouton "Run" (triangle vert) ou Maj+F10, l'application est compilée, déployée et lancée. Sur Compose, l'**outil d'aperçu** est très utile durant le développement : dans Android Studio, en disposant des fonctions `@Preview`, une fenêtre Preview affiche en temps réel le rendu de vos composables sans avoir à lancer l'appli complète [17] [18]. C'est un gain de temps précieux.

Conseils pour le projet rapide en examen : - *Préparation de l'IDE* : Avant l'examen, assurez-vous qu'Android Studio fonctionne correctement sur votre machine, que le SDK est à jour et qu'un émulateur est configuré. Vous ne voulez pas perdre 30 minutes à résoudre un problème d'environnement le moment venu. - *Squelettes de code* : N'hésitez pas à créer un projet de test à l'avance avec une Compose Activity vide pour vous familiariser avec la structure. Repérez où se trouve la fonction `setContent` et comment sont organisés les fichiers de thème. Le jour J, vous pourrez partir de ce squelette plus sereinement. - *Gradle et dépendances* : Le template Compose inclut normalement tout le nécessaire (Compose UI, Material, etc.). Si vous devez ajouter une bibliothèque (par ex. Room ou Navigation), souvenez-vous que cela se fait dans `app/build.gradle(.kts)` en ajoutant la dépendance et en synchronisant. Si possible, ayez sous la main les versions de dépendances requises ou utilisez le **BOM Compose** qui gère les versions pour vous (c'est souvent déjà configuré dans les projets récents). - *Organisation du code* : Même pour un petit projet d'examen, organisez vos fichiers : gardez par exemple les écrans UI dans un fichier ou package `ui/`, les modèles de données dans un fichier séparé, etc. Android Studio permet de créer des **packages** pour structurer (clic droit sur le dossier java → New > Package). Cela peut sembler du détail, mais retrouver rapidement où est telle fonction sous stress vous fera gagner de précieuses minutes.

En résumé, Android Studio est votre allié : maîtrisez-en les bases (création de projet, exécution, usage du preview Compose) et comprenez l'arborescence d'un projet Android. Une fois cela en place, on peut se concentrer sur Jetpack Compose pour construire l'interface utilisateur.

3. Fondamentaux de Jetpack Compose (Composable, Column, Row, Text, Button, etc.)

Jetpack Compose est le **nouveau toolkit déclaratif** pour construire des interfaces Android de manière plus simple et plus rapide. Plutôt que d'écrire du XML, on décrit l'UI directement en Kotlin via des fonctions dites *composables*. Compose utilise bien moins de code *boilerplate* et offre des API Kotlin intuitives [19].

Fonctions Composables : toute fonction d'interface que vous voulez rendre réutilisable ou affichable doit être annotée `@Composable`. Par exemple, une fonction simple pour afficher un texte peut être :

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

Ici `Text` est lui-même une fonction Composable fournie par la bibliothèque Compose UI qui affiche du texte à l'écran. Vous pouvez appeler `MessageCard("Android")` depuis une autre fonction composable (par exemple dans votre `setContent`). Compose se charge alors de convertir cela en éléments d'UI réels. **Important** : on ne peut appeler une composable que depuis une autre composable ou depuis la lambda de `setContent`. Vous ne pouvez pas invoquer directement une fonction @Composable depuis du code impératif classique.

Dans votre `MainActivity.onCreate`, vous verrez typiquement :

```
setContent {
    // on définit ici le contenu de l'Activity via Compose
    MyApplicationTheme {
        // Par exemple :
        MessageCard(name = "Android")
    }
}
```

Le bloc `setContent` déclare la **hiérarchie d'interface** de l'écran en appelant vos composables. Compose utilise un moteur de rendu qui **recompose** automatiquement l'UI quand les données changent, en ne mettant à jour que ce qui est nécessaire.

Exemple d'interface créée avec Jetpack Compose : une carte avec une image et du texte. Avec Compose, il suffit de quelques composables (Image, Text, Row, Column) pour arriver à ce résultat, le tout sans utiliser de layout XML [20] [21].

Les composables de base fournis par Compose incluent notamment : - **Text** – pour afficher du texte (équivalent d'un TextView). Exemple : `Text("Bonjour le monde")`. On peut ajuster son style via

`style = MaterialTheme.typography.bodyMedium` ou sa couleur via `color = Color.Red`, etc. - **Button** – pour un bouton cliquable. On l'utilise en fournissant une action onClick et un contenu. Exemple :

```
Button(onClick = { /* action */ }) {
    Text("Cliquez-moi")
}
```

Compose fournit aussi des variantes comme `TextButton`, `OutlinedButton` selon le style Material Design souhaité. - **Row** et **Column** – les conteneurs de mise en page. Une `Row` dispose ses enfants horizontalement côte à côte, tandis qu'une `Column` les dispose verticalement [22] [23]. Ce sont l'équivalent déclaratif des LinearLayout en orientation horizontale ou verticale. On peut leur ajouter des `Modifier` pour la taille, padding, etc. (nous y reviendrons). - **Image** – pour afficher une image. On utilise généralement un `painterResource` pour charger une ressource drawable. Exemple :

```
Image(painter = painterResource(R.drawable.mon_image), contentDescription =
"Description")
```

On peut modifier sa forme (cercle, coin arrondis) avec `.clip(shape)` et sa taille avec `.size(dp)` via des modificateurs [24] [25]. - **Spacer** – un composable invisible utilisé pour insérer un espace vide (vertical ou horizontal) entre des éléments, avec un `Modifier.width()` ou `.height()` [26] [27]. - **Surface** – un conteneur souvent utilisé pour appliquer un fond ou une élévation à un bloc d'UI (par exemple pour une carte). On l'utilise en Material3 pour créer une surface avec une couleur de fond du thème.

Composition hiérarchique : Compose vous fait construire l'interface en imbriquant des composables. Par exemple, pour faire une carte de message avec une photo de profil et deux textes (auteur et message), on peut écrire :

```
@Composable
fun MessageCard(msg: Message) {
    Row(modifier = Modifier.padding(8.dp)) {
        Image(
            painter = painterResource(R.drawable.profile_picture),
            contentDescription = null, // image décorative
            modifier = Modifier
                .size(40.dp)
                .clip(CircleShape)
                .border(1.5.dp, MaterialTheme.colorScheme.primary,
CircleShape)
        )
        Spacer(modifier = Modifier.width(8.dp))
        Column {
            Text(text = msg.author, style =
MaterialTheme.typography.titleSmall)
            Spacer(modifier = Modifier.height(4.dp))
            Text(text = msg.body, style =
MaterialTheme.typography.bodyMedium, maxLines = 1)
```

```
        }
    }
}
```

Dans ce code, on combine Row, Column, Text, Image, Spacer pour structurer le contenu [28] [29]. L'image est en cercle avec bordure, suivie d'un espace, puis d'une colonne contenant deux textes. Sans Compose, cela aurait nécessité un layout XML complexe, ici tout est dans une fonction Kotlin lisible.

Modifier et mises en forme : Le paramètre `modifier` disponible sur de nombreux composables permet de leur appliquer des **modificateurs** pour la mise en page ou le style. Par exemple, `Modifier.padding(8.dp)` ajoute une marge de 8dp autour d'un élément [30], `Modifier.fillMaxWidth()` ferait s'étendre un composable sur toute la largeur disponible, etc. Les modificateurs se chênent via le `.` et sont appliqués dans l'ordre d'écriture. Ce système remplace les attributs XML layout_width, layout_height, margin, etc., par une approche fluide et programmatique.

MaterialTheme : Jetpack Compose est orienté Material Design par défaut. Le template de projet crée un thème (Material3) que vous pouvez utiliser via `MaterialTheme` pour styliser vos composants (couleurs, typographie). Par exemple, `MaterialTheme.colorScheme.primary` vous donne la couleur primaire du thème, utilisable sur vos textes, fonds, etc. Le thème Material3 inclut des styles prédéfinis de texte (h1, body1, etc.) accessibles via `MaterialTheme.typography`. Dans notre code ci-dessus, on a utilisé `MaterialTheme.typography.titleSmall` pour le nom d'auteur. Pensez à envelopper l'ensemble de votre UI dans le thème, typiquement en appelant `MyApplicationTheme { Surface { ... } }` dans le setContent, afin que tout hérite du design choisi [31] [32].

Aperçu dans Android Studio : grâce aux annotations `@Preview`, vous pouvez prévisualiser vos composables sans lancer l'app. Exemple :

```
@Preview(showBackground = true)
@Composable
fun PreviewMessageCard() {
    MyApplicationTheme {
        MessageCard(Message("Lexi", "Jetpack Compose est génial !"))
    }
}
```

Cela va afficher dans l'IDE un rendu de `MessageCard` avec un thème appliqué, très utile pour vérifier l'apparence [33] [34]. *Conseil examen :* utilisez les Previews pendant que vous codez l'UI. Cela vous permet de repérer rapidement un problème d'affichage (texte qui se chevauche, etc.) sans perdre de temps à relancer l'émulateur.

Interactions de base : Pour rendre un composable interactif, Compose propose des modificateurs comme `clickable` (pour rendre cliquable n'importe quel composable) ou des composables prêts à l'emploi comme `Button`, `IconButton`, `TextField` (pour la saisie texte), des cases à cocher (`Checkbox`), etc. Par exemple, pour rendre notre carte de message cliquable :

```
Row(modifier = Modifier
        .padding(8.dp)
```

```
        .clickable { /* action au clic */ }
) { ... }
```

Ainsi, toute la ligne devient cliquable.

Cycle de vie Compose : Compose adopte un **paradigme déclaratif**. Vous n'avez plus à manuellement trouver des vues par ID ni à appeler `myTextView.setText()` dans votre activité. À la place, vous déclarez "voici à quoi doit ressembler l'UI pour tel état de données". Compose se charge d'appeler vos composables et de les reconstruire (recomposer) automatiquement quand les données sous-jacentes changent. Nous verrons dans la section sur la gestion d'état comment cela fonctionne.

Conseils en vrac pour Compose : - *Moins de code impératif :* évitez de penser en termes de "mettre à jour l'UI" manuellement. Concentrez-vous sur la description de l'écran en fonction des données. - *Organisez vos composables :* n'hésitez pas à créer de petites fonctions composables pour des éléments récurrents (un widget de profil utilisateur, un champ d'entrée personnalisé, etc.). Cela améliore la lisibilité et la réutilisabilité. - *Préfixe @Composable :* Toute fonction composable doit avoir le décorateur `@Composable`. Si vous oubliez, Android Studio vous le signalera. De même, vous verrez qu'une composable ne peut pas appeler directement une fonction standard qui retourne un UI (car elle n'est pas @Composable). Respectez bien cette contrainte, sinon l'IDE va râler. - *Erreurs communes :* Si rien ne s'affiche à l'écran, vérifiez que vous appelez bien votre composable principal dans `setContent`. Par exemple, si vous avez fait une fonction `AccueilScreen()`, assurez-vous que `setContent { AccueilScreen() }` est présent. Une autre erreur classique en Compose est d'oublier un Modifier.fillMaxSize() ou un Scroll, ce qui peut tronquer votre contenu. Apprenez à repérer ces soucis en observant l'aperçu ou l'UI sur l'émulateur.

En maîtrisant ces bases de Compose (Text, Button, Row/Column, etc.), vous pourrez rapidement construire l'interface de votre application. Dans la prochaine section, nous verrons comment naviguer entre plusieurs écrans composables.

4. Navigation entre écrans (Jetpack Navigation Compose)

La plupart des applications réelles comportent plusieurs écrans entre lesquels l'utilisateur peut naviguer (pages de contenu, formulaires, écran de détails, etc.). En **Modern Android Development**, le composant Jetpack **Navigation** simplifie la gestion de la navigation et de la pile d'écrans. Jetpack Navigation Compose est l'extension qui permet d'utiliser ce composant directement avec Compose, de façon déclarative et type-safe.

Principe : On définit un **graph de navigation** composé de destinations, et un **NavController** pour piloter la navigation. En Compose, une *destination* correspond généralement à une fonction @Composable représentant l'écran. Le NavController s'occupe de changer l'écran affiché et de gérer la "Back Stack" (pile d'historique). Voici les éléments clés :
- **NavController** : C'est la classe centrale qui orchestre la navigation entre vos écrans composables [35]. Il sait quels écrans sont disponibles et permet de passer de l'un à l'autre via des méthodes comme `navigate()`, `popBackStack()`, etc. Dans Compose, on obtient une instance via `val navController = rememberNavController()` au sein d'un composable (souvent au niveau de l'activité ou du composant racine).
- **NavHost** : C'est un composable fourni par la librairie Navigation Compose. Il sert de conteneur pour afficher le bon écran en fonction de l'état du NavController [36]. En définissant un NavHost, on lui associe le navController, une destination de départ, et on déclare à l'intérieur toutes les routes (composables) possibles de l'app. - **NavGraph / routes** : On peut nommer chaque écran par un

identifiant de route (une simple chaîne de caractère) ou utiliser une classe scellée/enum pour plus de sûreté. Chaque route est reliée à une composable. Le NavGraph est la structure qui mappe ces routes aux écrans [35].

Concrètement, avec la bibliothèque *navigation-compose*, on va écrire quelque chose comme :

```
NavHost(navController = navController, startDestination = "accueil") {
    composable("accueil") { AccueilScreen(navController) }
    composable("details") { DetailsScreen(navController) }
}
```

Ici on a deux écrans, "accueil" et "details". En exécutant `navController.navigate("details")`, on demandera à NavHost d'afficher la DetailsScreen, en empilant l'écran précédent dans le back stack.

Mise en place : Pour utiliser Navigation Compose, il faut ajouter la dépendance `androidx.navigation:navigation-compose`. Assurez-vous de l'avoir dans votre build.gradle (le template "Empty Compose Activity" ne l'inclut pas par défaut). Ensuite, typiquement on crée un composable racine qui gère la nav, par exemple :

```
@Composable
fun MyAppNavHost() {
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination = "screen1") {
        composable("screen1") { Ecran1(navController) }
        composable("screen2") { Ecran2(navController) }
        // etc. Ajoutez toutes les routes nécessaires
    }
}
```

On appellera `MyAppNavHost()` dans le `setContent` de l'activité principale pour initialiser la nav.

Navigation et actions utilisateur : Pour passer d'un écran à l'autre, on utilise le NavController. Par exemple, dans Ecran1, un bouton "Aller à l'écran 2" pourrait être :

```
Button(onClick = { navController.navigate("screen2") }) {
    Text("Suivant")
}
```

Cet appel empile l'écran2. Le NavController gère automatiquement le bouton retour (Back) d'Android pour revenir en arrière dans la pile. Si vous voulez gérer explicitement le retour, vous pouvez appeler `navController.popBackStack()` (pour dépiler un écran).

Si vous avez plusieurs étapes, vous pouvez aussi naviguer en passant des **arguments** aux composables. Navigation Compose permet de passer des paramètres typés via la fonction `composable(route) { backStackEntry -> ... }`, ou plus simplement de stocker l'état partagé dans un ViewModel commun (nous en parlerons plus loin). Pour un début, sachez qu'il est possible d'ajouter `/{param}`

dans une route et de le récupérer, mais cela peut être complexe sous pression, donc on conseille de limiter les arguments ou d'utiliser des ViewModel partagés pour l'examen.

Exemple simple : Supposons une app de quiz à deux écrans : l'écran de question et l'écran de résultat. On peut définir :

```
NavHost(navController, startDestination = "question") {
    composable("question") { QuizQuestionScreen(onQuizEnd = { score ->
        // Naviguer vers l'écran résultat en passant le score
        navController.navigate("resultat/$score")
    })}
    composable(
      route = "resultat/{score}",
      arguments = listOf(navArgument("score") { type = NavType.IntType })
    ) { backStackEntry ->
        val score = backStackEntry.arguments?.getInt("score") ?: 0
        QuizResultScreen(score, onRetry = {
            navController.popBackStack("question", inclusive = false)
        })
    }
}
```

Ici, on passe un paramètre score via la route. À l'examen, si la gestion d'arguments vous paraît compliquée, une approche plus simple est d'utiliser un ViewModel qui stocke le score globalement.

Navigation et AppBar : Pensez à mettre à jour le titre du TopAppBar ou le bouton de retour en fonction de l'écran courant. Vous pouvez observer le `navController.currentBackStackEntryAsState()` pour adapter l'UI (par exemple afficher une flèche de retour sur les écrans secondaires). Toutefois, cela entre dans des détails qu'on peut éviter dans un mini-projet d'examen en gardant une interface simple.

Exemple d'application multi-écrans : une conversation chat affichée via Compose. L'utilisateur peut naviguer dans les messages. Avec Navigation Compose, chaque écran (liste de conversations, détail d'une conversation, etc.) est un composable dans le NavHost. La LazyColumn n'affiche ici que les éléments visibles, offrant de hautes performances pour les longues listes [37] [38].

Conseils pour la navigation en examen : - **Simplifiez les chemins :** N'utilisez que quelques écrans (2 ou 3 maximum) pour limiter la complexité. Par exemple, un écran principal et un écran de formulaire/détails. - **Préparez du code de navigation générique :** Vous pouvez mémoriser un extrait de code NavHost comme squelette et l'adapter rapidement (pensez à importer `androidx.navigation.compose.*` et à ajouter la dépendance). - **Testez vos transitions :** Vérifiez en runnant que cliquer sur vos boutons change bien d'écran et que le bouton "Back" du téléphone fonctionne (sinon, c'est souvent parce que vous avez mal configuré le NavController ou que vous empilez plusieurs fois le même écran). - **Pas de panique si bloqué :** En cas de bug de navigation, un contournement rapide peut être de tout mettre sur un seul écran et de gérer l'affichage de pseudo-pages via des `if` en Kotlin (exemple : `if (showResult) { ResultUI() } else { QuestionUI() }`). Ce n'est pas architecturalement idéal, mais en situation d'examen, cela peut sauver la fonctionnalité si la navigation vous pose problème. Cependant, essayez de suivre les bonnes pratiques avec NavController si possible, car c'est plus propre et souvent attendu.

Jetpack Navigation Compose vous évite d'écrire du code spaghetti pour passer d'une activité à l'autre ou gérer des fragments. Une fois configuré, il suffit d'appeler `navController.navigate(route)` et Compose s'occupe du reste. Profitez-en pour structurer clairement vos écrans et rendre votre app navigable intuitivement.

5. Gestion d'état avec `remember` et `mutableStateOf`

La **gestion d'état** est au cœur de Jetpack Compose. Dans l'approche déclarative, l'interface se met à jour automatiquement en fonction de l'état de vos données. Il faut donc savoir créer et manipuler cet état de manière appropriée.

`mutableStateOf` **:** c'est une fonction qui prend une valeur initiale et retourne un **objet état observable**. En d'autres termes, c'est un conteneur dont la valeur peut changer et qui informera Compose de la nécessité de recompose l'UI quand ça arrive. Exemple : `val nom = mutableStateOf("Jean")`. On obtient un `State<String>` dont la valeur initiale est "Jean". Pour accéder à la valeur, on fait `nom.value`, et pour la modifier, `nom.value = "Pierre"`. Cependant, dans Compose on préfère souvent utiliser le délégateur `by` pour plus de concision.

`remember` **:** cette fonction est utilisée à l'intérieur d'une composable pour **se souvenir de l'état à travers les recompositions**. Si vous créez un objet d'état sans `remember`, il sera recréé à chaque recomposition, ce qui n'est pas le comportement voulu pour conserver une valeur. En combinant les deux :

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

Ici on déclare une variable `compteur` qui est un état mutable se souvenant de sa valeur. À chaque fois que la composable est ré-évaluée (par Compose), la valeur précédente sera retenue au lieu de réinitialiser à 0.

Recomposition automatique : Lorsque la valeur d'un `mutableStateOf` change, Compose marque les fonctions composables qui l'utilisent pour être recomposées (c'est-à-dire exécutées de nouveau afin de mettre l'UI à jour) [39] . Par exemple, si on a :

```
var texte by remember { mutableStateOf("Bonjour") }
Text(texte)
Button(onClick = { texte = "Bonsoir" }) { Text("Changer") }
```

Au clic du bouton, on change la variable `texte`. Compose détecte que `texte` est un état utilisé dans un composable (le Text), et va re-appeler la composable englobante pour rafraîchir l'écran. Résultat : le Text affichera "Bonsoir" sans que vous ayez eu à manipuler directement la vue.

Exemple pratique : Imaginons un bouton qui, à chaque clic, incrémente un compteur affiché à l'écran. Avec Compose, cela donne :

```
@Composable
fun CompteurScreen() {
    var count by remember { mutableStateOf(0) }
```

```
    Column(horizontalAlignment = Alignment.CenterHorizontally) {
        Text("Compteur : $count")
        Button(onClick = { count++ }) {
            Text("Incrementer")
        }
    }
}
```

Au départ, `count` vaut 0. On affiche "Compteur : 0". Quand on appuie sur le bouton, `count++` modifie l'état. Compose recompose automatiquement la fonction `CompteurScreen`, donc le Text est redessiné avec la nouvelle valeur de count ("Compteur : 1", puis 2, etc.). Ce mécanisme est très puissant car il vous évite d'écrire du code de liaison UI/valeur manuellement – tout est *réactif*.

Le mot-clé `by` et les imports : Vous noterez l'usage de `by` pour déléguer l'accès à la valeur d'un State. C'est purement du sucre syntaxique pour éviter d'écrire `.value` partout. Pour que cela fonctionne, assurez-vous d'importer `import androidx.compose.runtime.getValue` et `import androidx.compose.runtime.setValue` (Android Studio le propose normalement automatiquement) [40].

Plusieurs états : Vous pouvez bien sûr avoir plusieurs variables d'état dans une même interface. Par exemple, un champ texte et une case à cocher auraient chacun leur `remember { mutableStateOf(...) }`. Veillez juste à les initialiser à l'intérieur de la composable (ou dans un ViewModel, cf section suivante).

`rememberSaveable` **:** Il existe une variante de remember qui permet de sauvegarder l'état à travers les changements de configuration (comme la rotation d'écran), c'est `rememberSaveable`. Il fonctionne comme remember mais en plus, il sauvegarde la valeur dans un SavedInstanceState. Pour un examen de 3h, ce détail n'est pas forcément crucial, sauf si vous savez que l'évaluateur va tourner l'écran pour tester ! Par prudence, vous pouvez remplacer la plupart des remember par rememberSaveable, surtout pour des données simples (types primitifs, String, etc. supportés par Bundle). Sinon, documentez dans votre copie que en cas de rotation l'état se réinitialise, mais qu'on pourrait utiliser rememberSaveable pour y remédier.

Exemple d'utilisation dans une UI Compose : Reprenons la carte de message de la section précédente, en y ajoutant la capacité de se déplier pour afficher tout le texte. On peut utiliser un état booléen *isExpanded* pour chaque message, initialisé à false :

```
@Composable
fun MessageCard(msg: Message) {
    var isExpanded by remember { mutableStateOf(false) }
    Column(modifier = Modifier.clickable { isExpanded = !isExpanded }) {
        Text(text = msg.author)
        Text(
            text = msg.body,
            maxLines = if (isExpanded) Int.MAX_VALUE else 1
        )
    }
}
```

Ici, on se *souvient* de la variable isExpanded pour chaque MessageCard. Par défaut non étendu. Le Column a un modifier clickable qui inverse isExpanded au clic [29] [41]. Grâce à Compose, cliquer sur le message modifie isExpanded, ce qui déclenche une recomposition du Text secondaire avec un `maxLines` différent, révélant ainsi tout le texte. C'est fluide et sans code impératif de manipulation de TextView. Compose gère l'animation d'expansion si on le souhaite (via `animateContentSize()` par exemple, en modifiant le Modifier) [42], mais c'est optionnel.

Portée de l'état : Un piège classique est de définir un remember à un niveau trop bas ou trop haut. Règle d'or : l'état minimal requis pour un **sous-composant** devrait être hoisté (remonté) au composant parent si plusieurs composables en dépendent. Par exemple, si vous avez une liste de tâches avec des cases à cocher, vous pouvez gérer l'état "coché/pas coché" soit individuellement dans chaque item (remember dans l'item composable), soit de manière centralisée dans la liste (une liste de booléens dans le ViewModel ou le composant parent). Pour un petit projet, on peut faire au plus simple, mais sachez que soulever l'état permet de le partager.

Conseils d'utilisation de l'état en examen : - **Toujours initialiser l'état correctement** : si vous utilisez `mutableStateOf`, pensez à la valeur initiale adéquate (ex: une chaîne vide `""` pour un champ texte, false pour un switch off, etc.). - **Éviter les états non nécéssaires** : Ne dupliquez pas l'information. Si un état peut être dérivé d'un autre (par ex, vous stockez déjà une liste, pas besoin d'un state séparé pour "count" = `list.size`), utilisez la source directe. Trop d'états rend la logique confuse. - **Tester la reactivité** : jouez avec votre UI – si une valeur change mais rien ne se met à jour, c'est souvent que vous n'avez pas utilisé un mutableStateOf/remember correctement. À l'inverse, si ça recomposé de façon infinie ou inattendue, vérifiez de ne pas recréer un state à chaque recomposition par erreur (d'où l'importance du remember). - **Nettoyage** : `remember` ne persiste l'état qu'au sein du cycle de vie du composable. Si le composable disparaît (ex: on navigue ailleurs), l'état est perdu. Si vous avez des états qu'il faut vraiment conserver plus globalement (ex: panier d'achat), envisagez un ViewModel ou un état hoisté plus haut dans l'arbre de composables.

En résumé, Compose élimine le besoin de gérer manuellement les changements d'UI : on manipule juste des états Kotlin, et l'interface "suit". C'est très confortable une fois qu'on a pris le pli. Avec `remember { mutableStateOf(...) }`, vous avez 90% des cas d'usage d'interactivité couverts (champs modifiables, toggles, compteurs, etc.). Dans la section suivante, nous verrons comment intégrer un ViewModel pour gérer l'état de manière encore plus propre, surtout quand l'application grossit.

6. Intégration de ViewModel (basique)

Le **ViewModel** est un composant du pattern MVVM (Model-View-ViewModel) qui fait partie des bibliothèques Android Jetpack. C'est un objet qui a vocation à *conserver l'état de l'interface* et la logique métier associée, indépendamment du cycle de vie des activités/composables. En clair, il sert de tampon entre les données et l'UI : il fournit à l'UI les données prêtes à afficher et récupère les actions de l'UI pour mettre à jour ces données, le tout en survivant aux rotations d'écran et autres recreations d'activité.

Dans le contexte de Compose, l'utilisation du ViewModel est fortement encouragée pour tout état non éphémère ou partagé entre plusieurs composables. **Jetpack Compose supporte totalement les ViewModels Jetpack** [43], via une intégration directe.

Créer un ViewModel : On définit une classe qui hérite de `ViewModel` (du package androidx.lifecycle). Exemple basique :

```
class MonViewModel : ViewModel() {
    // Un état de UI stocké dans le VM
    var compteur by mutableStateOf(0)
        private set  // accès en lecture seule de l'extérieur

    fun incrementer() {
        compteur++
    }
}
```

Ici, `compteur` est un état (on peut même le déclarer en MutableLiveData ou StateFlow, mais Compose permet l'utilisation directe de mutableStateOf dans un ViewModel). On met `private set` pour que seul le ViewModel modifie sa valeur. La fonction `incrementer()` encapsule la logique de mise à jour. Un ViewModel peut également initier des chargements de données, appeler des repositories, etc., mais pour un exam simple on aura surtout de petits morceaux de logique.

Associer un ViewModel à l'UI Compose : Compose fournit l'API `viewModel()` pour récupérer une instance de ViewModel associée à l'activité ou à une navigation destination. Vous pouvez l'appeler directement dans une composable. Par exemple :

```
@Composable
fun CompteurScreen(viewModel: MonViewModel = viewModel()) {
    val compteur = viewModel.compteur  // comme c'est un State<Int>, Compose
le rend observable
    Column {
        Text("Valeur : $compteur")
        Button(onClick = { viewModel.incrementer() }) {
            Text("Cliquez")
        }
    }
}
```

Ici, `viewModel()` va par défaut fournir une instance de MonViewModel en se basant sur le `ViewModelStoreOwner` courant (l'activité hôte par défaut, ou la NavBackStackEntry si on est dans NavHost). Cela veut dire que si plusieurs écrans utilisent le même MonViewModel, ils partageront la même instance (ce qui peut être utile pour un état global). **Attention :** on **ne peut pas scope un ViewModel à une simple fonction composable** non liée à une navigation ou une Activity [44]. En gros, il faut que Compose sache à quel "scope" de vie attacher le ViewModel (Activity, Fragment, Navigation graph). Si vous appelez `viewModel()` dans une composable très profondément imbriquée, il utilisera toujours l'Activity par défaut, sauf si vous êtes dans NavHost. Si vous voulez un ViewModel par écran de NavHost, il est conseillé d'appeler `val vm: MonViewModel = viewModel()` **à l'intérieur de la lambda de composable{ } de NavHost**, ainsi le ViewModel sera scoped à cette destination de navigation.

Dans l'exemple ci-dessus, `viewModel.compteur` est de type `Int` (en fait `mutableStateOf` s'utilise comme un State ici). Compose va observer ce `compteur` pour recomposer CompeurScreen dès qu'il change. Ainsi, quand on appelle `viewModel.incrementer()`, la valeur augmente et l'UI se met à jour automatiquement.

Avantages du ViewModel : - Il survit aux rotations et recréations d'activité (contrairement aux variables stockées directement dans une composable sans rememberSaveable). Si l'utilisateur tourne l'écran, le ViewModel n'est pas recréé, donc `compteur` gardera sa valeur et on n'aura pas de reset à 0 [45] [46]. - Il sépare les responsabilités : la UI ne fait que afficher et déléguer les actions, le ViewModel gère la logique (calculer un score, valider une entrée, charger des données d'une base, etc.). Cela rend le code plus testable et clair. - On peut partager un même ViewModel entre plusieurs écrans (par ex, un ViewModel "Panier" accessible depuis l'écran liste produits et l'écran détail produit). - Le ViewModel peut exposer l'état sous forme de LiveData ou Flow, que Compose peut observer via `collectAsState()` ou `observeAsState()`. Cependant, pour de la simplicité on peut aussi utiliser des `mutableStateOf` directement dans le ViewModel comme montré, ce qui évite d'introduire la notion de LiveData.

Exemple concret : Si on reprend notre concept d'application de tâches (to-do list) : - On peut avoir un `TodoViewModel` avec une liste de tâches en état (`var tasks = mutableStateListOf<Tache>()` par exemple, ou un SnapshotStateList). - Le ViewModel fournirait des fonctions `ajouterTache(tache: Tache)`, `supprimerTache(tache)`, etc., qui modifient la liste. - L'écran Compose (la liste des tâches) récupérerait `val tasks = todoViewModel.tasks` et l'afficherait (voir LazyColumn section suivante), et appellerait `todoViewModel.ajouterTache(...)` quand l'utilisateur valide le formulaire d'ajout. Compose rendra la LazyColumn réactive à toute modification de la liste (les *SnapshotStateList* émettent des recompositions sur changement).

Importer le ViewModel : Assurez-vous d'ajouter la dépendance `implementation("androidx.lifecycle:lifecycle-viewmodel-compose:X.Y.Z")` (et éventuellement `lifecycle-runtime-ktx`). Dans les derniers BOM Android, elle est souvent incluse. Sans cela, la fonction `viewModel()` dans Compose pourrait ne pas être reconnue.

Conseils usage ViewModel en examen : - *Ne perdez pas de temps inutile :* Si votre application est très simple (par ex, juste un écran formulaire+résultat), vous pouvez techniquement vous passer de ViewModel en gérant l'état avec remember. Mais si la consigne de l'examen mentionne ou attend un ViewModel, montrez-en un usage basique comme ci-dessus. C'est généralement bien vu de structurer en MVVM même un petit projet. - *Pas de contexte Android dans le VM :* Rappelez-vous qu'un ViewModel ne doit pas contenir de référence directe à UI ou contexte (pas d'Activity, pas de View). Il doit uniquement manipuler des données. Compose permet parfois d'accéder à `LocalContext`, mais ne passez pas ça à un ViewModel. Si vous avez besoin d'une ressource (ex: string), mieux vaut la injecter ou l'exposer autrement. Cependant, pour un petit exam, ce point ne devrait pas trop se poser. - *Nettoyage :* Un ViewModel peut implémenter `onCleared()` si des ressources doivent être libérées quand il est détruit (ex: fermer une connexion). Dans un contexte 3h, c'est rare d'en avoir besoin. - *Communication UI <-> VM :* Utilisez soit des fonctions du VM pour que l'UI lui envoie des événements (comme `incrementer()`), soit modifiez directement les propriétés du VM si elles sont var publiques (mais c'est moins encapsulé). L'approche idiomatique est de garder les propriétés en lecture seule et d'avoir des méthodes dans le VM. - *Observabilité :* Si vous utilisez LiveData ou Flow dans le VM, Compose peut les observer via `collectAsState()`. Mais on peut éviter cette couche pour un exam de base et utiliser directly mutableStateOf comme montré.

En somme, **ViewModel** vous aidera à gérer l'état de manière robuste dans Compose. Pour notre besoin (maîtriser une app en 3h), un seul ViewModel peut souvent suffire à gérer l'essentiel de l'état de l'application (par ex, toutes les données de votre ToDo ou quiz). Vous centralisez ainsi la logique et vous laissez l'UI Compose se rafraîchir en fonction. Cela réduit le risque de bugs lors des changements de configuration et clarifie le code. On va maintenant appliquer tout cela pour créer des formulaires et des listes dynamiques.

7. Création de formulaires simples (ex : login, inscription)

Les **formulaires** (écran de login, d'inscription, de saisie de données) sont un cas très courant et un bon moyen de tester vos compétences Compose. Un formulaire typique comprend des champs de texte, éventuellement des sélecteurs (checkbox, radio) et des boutons pour soumettre. Voyons comment gérer cela en Compose pour un débutant.

Champs de texte (`TextField` / `OutlinedTextField`) : Compose fournit des composables pour la saisie utilisateur. Le plus utilisé est `TextField` (style Material filled) et sa variante `OutlinedTextField` (avec bordure). Ils nécessitent un paramètre principal pour le texte et un lambda pour gérer la modification : - Ancienne approche (value/onValueChange) :

```
var email by remember { mutableStateOf("") }
OutlinedTextField(
    value = email,
    onValueChange = { email = it },
    label = { Text("Email") }
)
```

Ici, on lie le contenu du champ `email` à une variable d'état locale. À chaque saisie (touche frappée), Compose appelle `onValueChange` avec la nouvelle valeur et on met à jour notre état, ce qui recompose le TextField avec le nouveau texte. **Important :** sans cet état, le TextField ne pourra pas être édité (compose a besoin de la source de vérité).
- Nouvelle approche (Material3 1.4+ avec TextFieldState) : plus avancée, on peut utiliser `rememberTextFieldState()`. Pour un exam, on peut rester sur l'approche classique value/onValueChange qui est bien comprise.

Pour un **champ de mot de passe**, on veut masquer le texte saisi. On peut utiliser `visualTransformation = PasswordVisualTransformation()` sur un OutlinedTextField pour remplacer les caractères par des points. Exemple :

```
var password by remember { mutableStateOf("") }
OutlinedTextField(
    value = password,
    onValueChange = { password = it },
    label = { Text("Mot de passe") },
    visualTransformation = PasswordVisualTransformation(),
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password)
)
```

Ici on indique aussi au clavier virtuel qu'il s'agit d'un champ de mot de passe (ça peut changer l'affichage de la touche de validation, etc.). On pourrait ajouter une icône à droite pour afficher/masquer le mot de passe en jouant sur `visualTransformation` dynamique, mais c'est du bonus.

Boutons de soumission : Un formulaire a souvent un bouton "Valider" ou "Se connecter". Ce sera un composable `Button` standard. Dans son onClick, on va vérifier les champs et agir en conséquence (par exemple, envoyer une requête de login ou simplement naviguer vers un autre écran si c'est un examen hors-ligne). Dans un contexte d'examen, vous pouvez simuler la validation de login (pas besoin

d'une vraie authentification). Par exemple, on peut afficher un Toast ou juste naviguer vers un écran "Bienvenue".

Validation de champs : Pour un simple exam, on peut implémenter des validations basiques du style "champ requis" ou "email doit contenir @". Ceci peut se faire en ajustant l'état. Par exemple, on peut avoir un état pour l'erreur et afficher un texte rouge si erreur. Par simplicité :

```
var errorMessage by remember { mutableStateOf("") }
Button(onClick = {
    if (email.isBlank() || password.isBlank()) {
        errorMessage = "Veuillez remplir tous les champs"
    } else {
        errorMessage = ""
        // Poursuivre la logique de connexion
    }
}) { Text("Connexion") }
if (errorMessage.isNotEmpty()) {
    Text(errorMessage, color = Color.Red)
}
```

Ainsi l'erreur s'affiche dynamiquement si le bouton est pressé sans remplir les champs [47] .

Exemple concret de formulaire de login : Un écran de login minimal en Compose pourrait ressembler à ceci :

```
@Composable
fun LoginScreen(onLoginSuccess: () -> Unit) {
    var email by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }
    var error by remember { mutableStateOf("") }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text("Connexion", style = MaterialTheme.typography.headlineMedium)
        Spacer(Modifier.height(24.dp))
        OutlinedTextField(
            value = email,
            onValueChange = { email = it },
            label = { Text("Email") },
            modifier = Modifier.fillMaxWidth(),
            singleLine = true,
            keyboardOptions = KeyboardOptions(keyboardType =
KeyboardType.Email, imeAction = ImeAction.Next)
        )
```

```
        Spacer(Modifier.height(16.dp))
        OutlinedTextField(
            value = password,
            onValueChange = { password = it },
            label = { Text("Mot de passe") },
            modifier = Modifier.fillMaxWidth(),
            singleLine = true,
            visualTransformation = PasswordVisualTransformation(),
            keyboardOptions = KeyboardOptions(keyboardType =
KeyboardType.Password, imeAction = ImeAction.Done)
        )
        Spacer(Modifier.height(16.dp))
        Button(
            onClick = {
                if (email.isBlank() || password.isBlank()) {
                    error = "Veuillez renseigner email et mot de passe"
                } else {
                    // Ici, on pourrait vérifier la correspondance avec un
utilisateur fictif
                    error = ""
                    onLoginSuccess()
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text("Se connecter")
        }
        if (error.isNotEmpty()) {
            Spacer(Modifier.height(8.dp))
            Text(error, color = MaterialTheme.colorScheme.error)
        }
    }
}
```

Ce code met en œuvre les points clés : champs contrôlés par état, bouton avec validation et message d'erreur. On utilise quelques **KeyboardOptions** pour améliorer l'UX (par exemple passer au champ suivant automatiquement avec `imeAction`). **Note:** `imeAction = ImeAction.Done` permet de customiser la touche Entrée du clavier (ici pour soumettre).

Capture d'écran d'un simple formulaire de connexion construit avec Compose, comportant deux champs (`OutlinedTextField` pour le nom d'utilisateur et le mot de passe) et un bouton de validation [48] [49]. L'interface est épurée : chaque champ affiche un label et le texte saisi, et le bouton "Login" permet de soumettre le formulaire.

Une fois ce composant LoginScreen prêt, on peut l'intégrer dans la NavHost (ex: route "login") ou le lancer comme écran principal. S'il y avait une navigation à faire après connexion, on appellerait `onLoginSuccess()` pour naviguer vers l'écran suivant (par exemple : `navController.navigate("home")`).

Autres entrées utilisateur courantes : - *CheckBox*: pour une case à cocher, utilisez `Checkbox(checked = valeur, onCheckedChange = { valeur = it })` avec un état booléen via remember. - *RadioButton*: Compose propose RadioButton, généralement on les groupe via un Row/Column. On garde un état pour la sélection (par ex. une variable `choix: String`) et on fait `RadioButton(selected = choix == "Option1", onClick = { choix = "Option1" })`. - *Switch*: similaire à Checkbox, pour on/off.

Conseils formulaires en examen : - Gérez le focus du clavier : Compose gère automatiquement le focus entre champs si vous utilisez `ImeAction.Next` comme dans l'exemple (il faut aussi un `keyboardActions = KeyboardActions(onNext = { focusManager.moveFocus(...) })` si on veut le faire proprement). Pour un exam, ce n'est pas grave si l'utilisateur doit toucher le second champ manuellement, concentrez-vous sur la fonctionnalité. - Limitez la validation : Par manque de temps, faites des validations simples (champs requis). Si vous avez le temps, montrez-en une plus, par exemple vérifier que l'email contient "@" ou que le mot de passe a une certaine longueur, mais ce n'est pas prioritaire. - Accessibilité / ContentDescription : Indiquez des contentDescription sur les éléments non textuels (icônes, images) surtout si critique. Ce niveau de détail est bien mais souvent optionnel dans un contexte exam court. - **Ne stockez pas le mot de passe en clair** : bon, dans un exam local ce n'est pas très grave, mais c'est une mauvaise pratique en vrai. Vous pourriez mentionner oralement ou dans un commentaire que dans un vrai contexte, on ne ferait pas ça ainsi.

En maîtrisant les TextField et Buttons, vous pourrez réaliser d'autres formulaires comme une inscription (pratiquement les mêmes éléments qu'un login, avec champs supplémentaires). Compose simplifie la création de formulaires car tout est lié directement au state, ce qui évite d'écrire du code "retrouve l'editText et lis sa valeur".

8. Utilisation de listes (LazyColumn) avec données dynamiques

Afficher des listes de données est extrêmement fréquent (listes d'articles, de tâches, de messages...). Avec Jetpack Compose, on utilise principalement **LazyColumn** (pour une liste verticale) ou **LazyRow** (horizontale) pour afficher une collection d'éléments de façon performante. *Lazy* signifie que la liste est rendue de manière paresseuse : seuls les éléments visibles à l'écran sont composés, ce qui assure de bonnes performances même avec de longues listes [50] [51].

LazyColumn de base : son utilisation rappelle le RecyclerView + Adapter d'autrefois, mais en beaucoup plus simple. Exemple minimal :

```
val fruits = listOf("Banane", "Pomme", "Orange", "Kiwi")
LazyColumn {
    items(fruits) { fruit ->
        Text("Fruit : $fruit")
    }
}
```

Ici, `items(fruits)` va itérer sur la liste et pour chaque élément appeler le contenu lambda en passant l'élément (qu'on nomme ici `fruit`). Compose génère autant de `Text` que nécessaire pour afficher les 4 fruits, et si la liste était très longue, il n'en créerait d'abord que suffisamment pour remplir l'écran puis les suivants au scroll. Vous pouvez aussi spécifier un **index** si besoin en utilisant l'autre signature `itemsIndexed`.

Performance : LazyColumn n'affiche que les items visibles, ce qui le rend **très efficace pour de longues listes** (scroll infini, etc.) [52] . Vous n'avez plus besoin de ViewHolder ou d'optimisation manuelle. Compose recycle/détruit et crée les composables au fil du scroll automatiquement.

Gestion de données dynamiques : Si les données de la liste peuvent changer (par exemple on ajoute/enlève des éléments), utilisez de préférence une collection *State* comme **SnapshotStateList**. Si vous faites `val maListe = remember { mutableStateListOf(1,2,3) }`, cette liste est observable : ajouter ou retirer un élément déclenchera la recomposition de LazyColumn. Vous pouvez directement passer `items(maListe)` et Compose détectera les changements (il compare les éléments via leur **key** éventuellement). Pour assurer un bon suivi des modifications, surtout si vos éléments ne sont pas uniques ou la liste peut bouger, il est recommandé de fournir un paramètre `key` à items. Par exemple, si vous avez une data class Tache avec un champ id unique, faites `items(tasks, key = { it.id }) { ... }`. Ainsi Compose saura mieux identifier chaque item (utile pour animations ou juste performance de diff).

Exemple : liste de tâches : Supposons qu'on a un état dans le ViewModel : `val tasks = mutableStateListOf<Tache>()`. On veut afficher la liste et pouvoir cocher/décocher les tâches. On peut faire :

```
@Composable
fun TaskListScreen(viewModel: TodoViewModel = viewModel()) {
    val tasks = viewModel.tasks  // SnapshotStateList<Tache>
    LazyColumn {
        items(tasks, key = { it.id }) { task ->
            Row(modifier = Modifier.fillMaxWidth().padding(8.dp),
                verticalAlignment = Alignment.CenterVertically) {
                Checkbox(
                    checked = task.fait,
                    onCheckedChange = { viewModel.toggleDone(task) }
                )
                Text(
                    text = task.titre,
                    style = if (task.fait) TextStyle(textDecoration =
TextDecoration.LineThrough) else TextStyle()
                )
            }
            Divider()
        }
    }
    // On pourrait ajouter un bouton flottant + (FloatingActionButton) pour
ajouter une tâche
}
```

Ici, chaque item de la LazyColumn est une Row contenant une Checkbox et un Text. La Checkbox appelle `viewModel.toggleDone(task)` pour mettre à jour la tâche (par exemple inverser son booléen *fait*). Grâce à la nature mutableStateList, cocher la case modifie la liste (ou l'élément) et Compose recomposera cet item avec le nouveau état (le Text sera barré si fait = true). **Remarque :** il faut que le fait de cocher modifie un State observé, or ici *task.fait* est une propriété d'un élément de la liste. Dans un SnapshotStateList, si la data class n'est pas observable elle-même, il vaut mieux faire `tasks[index] = tasks[index].copy(fait = true)` pour que Compose capte le changement.

Ou déclarer Tache.fait comme `var fait by mutableStateOf(false)` à l'intérieur de la data class, ce qui est possible. Pour un exam, vous pouvez simplifier en recréant la liste ou en émettant une nouvelle liste, même si moins optimal.

Sectionnement, en-têtes, etc. : LazyColumn permet aussi d'insérer des séparateurs ou en-têtes. Vous pouvez utiliser directement des composables *hors* de items, par exemple :

```
LazyColumn {
    item { Text("Ma Liste de Fruits") }  // un seul élément en-tête
    items(fruits) { ... }
}
```

Ou intercaler des `Divider()` entre items comme dans l'exemple ci-dessus.

Scrolling : Par défaut, LazyColumn est scrollable. Vous pouvez modifier son comportement avec des paramètres (par ex. `verticalArrangement` pour l'espacement entre items, etc.). Si la liste est à l'intérieur d'un autre composant scrollable, attention à la **nested scrolling** (un LazyColumn dans un Column scrollable peut poser problème, il vaut mieux éviter deux scroll vertical imbriqués).

Affichage conditionnel d'une liste vide : Si votre liste peut être vide, prévoyez un petit `if` avant la LazyColumn :

```
if (tasks.isEmpty()) {
    Text("Aucune tâche pour le moment", modifier = Modifier.padding(16.dp))
} else {
    LazyColumn { items(tasks) { ... } }
}
```

Histoire d'informer l'utilisateur quand il n'y a rien.

Optimisation : Compose gère beaucoup de choses automatiquement. Cependant, pour d'énormes listes, vous pouvez activer la **pagination** ou le chargement à la volée. Dans un contexte de 3h, c'est peu probable qu'on vous demande ça. Mentionnez simplement que LazyColumn ne charge que les éléments visibles [50], ce qui suffit souvent.

Conseils pour les listes en examen : - *Mock data* : Si nécessaire, créez des données factices pour démontrer la liste (ex: une liste de 10 tâches en dur). Ne perdez pas trop de temps à faire un système d'ajout complet si ce n'est pas demandé – sauf si le sujet l'implique (par ex, "faire une liste où l'utilisateur peut ajouter des éléments"). - *Scrollable Column vs LazyColumn* : Si la liste a très peu d'éléments fixes, vous pourriez être tenté d'utiliser une Column simple avec `.verticalScroll()`. Mais montrez que vous savez utiliser LazyColumn, car c'est la bonne pratique dès qu'on a du contenu en liste, même modeste. - *UI des items* : personnalisez un minimum l'affichage de chaque item pour montrer que vous savez combiner composables (comme l'exemple avec Checkbox + Text). Si c'est une liste d'objets plus complexes, n'hésitez pas à créer un composable dédié pour l'item (ex: `@Composable fun TacheItem(task: Tache) { ... }`) et l'appeler dans items: `items(tasks) { TacheItem(it) }`. Cela améliore la lisibilité. - *Performance note* : vous pouvez mentionner que c'est plus besoin de ViewHolder, etc. Mais dans le code c'est déjà évident.

Avec LazyColumn, afficher un ensemble dynamique devient aisé. Compose s'occupe du recyclage et du diff. Vous, vous décrivez juste comment rendre un item. Ce sera particulièrement utile dans l'application complète qu'on va envisager (ToDo, quiz, notes – toutes ont des listes).

9. Stockage local de données avec Room (introduction simple)

Certaines applications nécessitent de **stocker des données localement** sur l'appareil de l'utilisateur, par exemple une liste de notes ou de tâches qui persiste entre les lancements de l'application. Android propose la base de données SQLite en natif, mais le framework **Room** (bibliothèque Jetpack) facilite grandement son utilisation en offrant une couche d'abstraction et des DAO (Data Access Object) puissants. Vu le temps imparti de 3h, il est possible que l'intégration complète de Room soit un peu ambitieuse, mais une *courte introduction* ne fait pas de mal et peut impressionner positivement si bien gérée.

Principe de Room : Room est une surcouche de SQLite qui utilise la réflexion et les annotations pour générer le code de base de données. Les éléments principaux sont [53] : - Une **classe de base de données** (annotée `@Database`) qui étend `RoomDatabase`. Elle définit les entités (tables) qu'elle contient et offre des méthodes d'accès (DAO). - Des **entités** (annotées `@Entity`) qui représentent les tables de la base, généralement ce sont des data classes Kotlin. Chaque propriété correspond à une colonne. Il faut une clé primaire (annotée `@PrimaryKey`) et on peut spécifier des infos de colonnes (nom, index, etc.) [54]. - Des **DAO** (interfaces annotées `@Dao`) qui contiennent les méthodes pour interagir avec la base (requêtes SQL via `@Query`, insertion via `@Insert`, mise à jour via `@Update`, suppression via `@Delete`) [55] [56]. Room génère automatiquement l'implémentation de ces interfaces.

Exemple minimal : Supposons une application de notes. On peut créer une entité Note :

```
@Entity
data class Note(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val contenu: String,
    val date: Long
)
```

Ici `id` sera la clé primaire auto-générée (0 signifie que Room la remplira à l'insertion). Ensuite un DAO :

```
@Dao
interface NoteDao {
    @Query("SELECT * FROM Note")
    fun getAll(): List<Note>

    @Insert
    fun insert(note: Note)

    @Delete
    fun delete(note: Note)
}
```

Enfin la classe database :

```
@Database(entities = [Note::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun noteDao(): NoteDao
}
```

Pour utiliser cette base, il faut créer une instance de AppDatabase. Room fournit une méthode builder :

```
val db = Room.databaseBuilder(context, AppDatabase::class.java, "ma-
database.db").build()
val noteDao = db.noteDao()
```

On peut ensuite appeler `noteDao.getAll()` pour récupérer les notes, mais attention, cela doit se faire en dehors du thread principal (Room force l'appel en asynchrone pour les queries longues). En pratique, on utiliserait des coroutines (suspend functions) ou LiveData/Flow pour ne pas bloquer l'UI.

Dans un examen court, on ne vous en voudra pas de simplifier (par ex, appeler sur un thread IO ou simuler l'opération). L'important est de montrer la structure.

Intégration avec Compose/ViewModel : Le ViewModel peut contenir le `noteDao` ou mieux, le référencer via un Repository. On peut appeler `getAll()` dans init du ViewModel pour peupler un état initial des notes. Ou utiliser un `Flow` et collectAsState en Compose. Cela commence à faire beaucoup de choses à mettre en place en 3h, donc il faut jauger. Si l'exam attend explicitement Room, vous pouvez préparer du code ou le pseudo-code comme ci-dessus.

Conseils de simplification pour l'exam : - Vous pouvez limiter l'implémentation. Par exemple, juste démontrer la création de l'entité et du DAO sans forcément utiliser effectivement la DB dans l'app, mais en expliquant que dans un vrai contexte on ferait noteDao.insert(note) quand l'utilisateur ajoute une note. - Ou alors, utiliser Room *seulement* pour stocker une petite info (par ex. les tâches) et montrer une requête simple.

Quand persister les données : Si l'application le justifie (ToDo, notes), c'est bien de sauvegarder en base pour retrouver les données au redémarrage. Dans le cadre de l'examen, on ne pourra sans doute pas démontrer la persistance sur plusieurs lancements (sauf si examinateur teste l'app après fermeture). Mais mentionner la persistance est un plus théorique.

Room et threads : Rappel : par défaut, les méthodes DAO ne sont pas suspendues, donc il faut les appeler dans un `CoroutineScope(Dispatchers.IO)`. Si on utilise Flow (en mettant `@Query("SELECT * FROM Note") fun getAll(): Flow<List<Note>>`), Room gère l'asynchronisme automatiquement et on collectera dans Compose. Ce niveau de détail peut être omis si c'est trop.

Conseils en examen pour Room : - *Ne vous embourbez pas dans la config* : ajouter Room nécessite souvent d'ajouter le plugin KSP ou annotationProcessor. Si vous ne l'avez jamais fait, faites-y attention. Ex: dans gradle, ajouter `ksp("androidx.room:room-compiler:2.x.x")`. Sans cette configuration, Room ne générera rien et vous aurez des erreurs. Il faut aussi `implementation("androidx.room:room-ktx:2.x.x")` et runtime. - *Plan B :* Si Room pose

problème, mentionnez dans votre rapport comment vous *auriez* stocké les données. Montrez l'entité et dites "faute de temps, l'implémentation d'insertion n'a pas été finalisée, mais la structure Room est en place". Mieux vaut ça que de tout casser. - *Ne pas bloquer l'UI* : Si vous appelez une query directement, l'app pourrait crasher (Room jette une exception si appel sur main thread). Si vous n'avez pas le temps de mettre en place coroutines, vous pouvez temporairement autoriser mainThread queries avec `.allowMainThreadQueries()` sur le builder Room (à n'utiliser qu'en debug !). C'est sale en prod, mais toléré en contexte d'apprentissage pour tester [57]. Vous pourriez justifier cela par le contexte restreint de l'examen.

TL;DR Room : 3 annotations principales (@Entity, @Dao, @Database) et votre base est prête à l'emploi sans écrire de SQL vous-même. Par exemple, l'image ci-dessous illustre l'architecture de Room : la classe Database donne accès aux DAO, qui eux permettent de faire des requêtes sur les tables entités [58] [59].

(Pas de capture d'écran ici pour Room car c'est conceptuel, mais vous pouvez imaginer un schéma avec l'AppDatabase, le DAO et l'entité.)

10. Exemple d'application complète réalisable en 3 heures

Pour conclure, mettons tout ensemble dans un **exemple concret d'application** qu'on pourrait réaliser en environ 3 heures de code : une application de gestion de tâches (ToDo app) simple. C'est un choix judicieux car elle combine plusieurs concepts que nous avons vus : un écran avec une liste de tâches (LazyColumn), la possibilité d'ajouter une tâche via un formulaire (TextField + Button), de marquer des tâches comme effectuées (gestion d'état + interactions), et la persistance locale (on peut y intégrer Room pour sauvegarder les tâches).

Remarque : Vous pourriez tout aussi bien implémenter une mini app de quiz (écrans enchaînés avec Navigation et calcul de score) ou une app de notes. Nous détaillons l'exemple ToDo, mais les autres suivent des schémas similaires en utilisant les mêmes briques.

Fonctionnalités de la ToDo App : - Afficher la liste des tâches existantes (avec leur statut fait / pas fait). - Permettre d'ajouter une nouvelle tâche via un champ de texte et un bouton "Ajouter". - Optionnel : Permettre de supprimer une tâche ou de la marquer terminée (par un checkbox). - L'UI se met à jour en temps réel quand on ajoute/termine une tâche. - Bonus si le temps : stocker la liste dans une base Room pour la retrouver à la relance de l'app.

Structure générale : On aura deux écrans principaux : 1. **TaskListScreen** – l'écran d'accueil affichant la liste des tâches et un bouton pour aller à l'écran d'ajout. 2. **AddTaskScreen** – un écran avec un champ texte pour la description de la tâche et un bouton de validation. (On pourrait aussi faire ce formulaire d'ajout en bas de la liste sur le même écran pour simplifier, mais utilisons la navigation pour démonstration).

On utilisera Navigation Compose pour passer de l'un à l'autre, et un ViewModel (TodoViewModel) pour conserver la liste des tâches et gérer les opérations.

Data model : une data class `Tache(val id: Int, val titre: String, val fait: Boolean = false)`. Si on intègre Room, on lui mettrait @Entity et @PrimaryKey. Sinon, on la traite en mémoire seulement.

ViewModel :

```
class TodoViewModel : ViewModel() {
    private var nextId = 0  // pour auto-incrémenter les ids en cas de non-
Room
    var tasks = mutableStateListOf<Tache>()
        private set

    fun ajouterTache(titre: String) {
        if (titre.isBlank()) return
        tasks.add(Tache(id = nextId++, titre = titre, fait = false))
    }

    fun toggleDone(task: Tache) {
        val index = tasks.indexOf(task)
        if (index >= 0) {
            val ancienne = tasks[index]
            tasks[index] = ancienne.copy(fait = !ancienne.fait)
        }
    }

    fun supprimerTache(task: Tache) {
        tasks.remove(task)
    }
}
```

Ici, on gère une liste mutable d'état (SnapshotStateList). `ajouterTache` crée une nouvelle tâche avec un ID unique et l'ajoute. `toggleDone` remplace la tâche par une copie inversant le booléen *fait* (cette opération notifie Compose que l'item a changé). `supprimerTache` enlève l'élément. Si on utilisait Room, ces méthodes appelleraient en plus le DAO correspondant (`dao.insert`, `dao.update`...). Mais on peut le faire plus tard.

On crée une instance de ce ViewModel au niveau du NavHost ou de l'activité (via `val todoViewModel: TodoViewModel = viewModel()`).

Navigation : Deux routes : "liste" et "ajout". Dans NavHost :

```
NavHost(navController, startDestination = "liste") {
    composable("liste") {
       TaskListScreen(
          tasks = todoViewModel.tasks,
          onToggle = { todoViewModel.toggleDone(it) },
          onDelete = { todoViewModel.supprimerTache(it) },
          onNavigateToAdd = { navController.navigate("ajout") }
       )
    }
    composable("ajout") {
       AddTaskScreen(
          onAdd = { titre ->
             todoViewModel.ajouterTache(titre)
             navController.popBackStack() // revenir à l'écran liste
```

```
            },
            onCancel = { navController.popBackStack() }
        )
    }
}
```

On injecte le viewModel ou ses données dans les écrans via les lambdas. On pourrait aussi obtenir `TodoViewModel` dans chaque écran avec `val todoViewModel = viewModel(LocalContext.current as ComponentActivity)` mais passons par paramètres pour plus de clarté.

UI TaskListScreen :

```
@Composable
fun TaskListScreen(
    tasks: List<Tache>,
    onToggle: (Tache) -> Unit,
    onDelete: (Tache) -> Unit,
    onNavigateToAdd: () -> Unit
) {
    Scaffold(
        topBar = { TopAppBar(title = { Text("Mes Tâches") }) },
        floatingActionButton = {
            FloatingActionButton(onClick = onNavigateToAdd) {
                Icon(Icons.Default.Add, contentDescription = "Ajouter")
            }
        }
    ) { padding ->
        if (tasks.isEmpty()) {
            Box(modifier = Modifier.fillMaxSize().padding(padding),
contentAlignment = Alignment.Center) {
                Text("Aucune tâche. Cliquez sur + pour en ajouter.")
            }
        } else {
            LazyColumn(modifier = Modifier.padding(padding)) {
                items(tasks, key = { it.id }) { task ->
                    Row(modifier = Modifier
                            .fillMaxWidth()
                            .padding(8.dp),
                        verticalAlignment = Alignment.CenterVertically
                    ) {
                        Checkbox(checked = task.fait, onCheckedChange = {
onToggle(task) })
                        Text(
                            text = task.titre,
                            modifier = Modifier.weight(1f).padding(start =
8.dp),
                            style = if (task.fait) TextStyle(textDecoration
= TextDecoration.LineThrough) else TextStyle()
                        )
```

```
                        IconButton(onClick = { onDelete(task) }) {
                            Icon(Icons.Default.Delete, contentDescription =
"Supprimer")
                        }
                    }
                    Divider()
                }
            }
        }
    }
}
```

Explications : On utilise un `Scaffold` qui offre un design de base avec une TopAppBar (titre) et un FloatingActionButton. Le FAB appelle `onNavigateToAdd` pour aller à l'écran d'ajout. Dans le corps, on affiche soit un message de liste vide, soit la LazyColumn des tâches. Chaque ligne affiche une Checkbox (coche ou décoche via onToggle), le titre de la tâche (avec style barré si terminée), et une icône de poubelle pour supprimer la tâche. On a ajouté une petite marge et on utilise `Modifier.weight(1f)` sur le texte pour qu'il prenne l'espace disponible entre la checkbox et l'icône de suppression. On sépare chaque item par un Divider pour l'esthétique.

Cette UI doit se rafraîchir automatiquement quand `tasks` change, car `tasks` est une SnapshotStateList observable. Comme on passe `tasks: List<Tache>` en param, Compose va sous le capot convertir cela en State<List<Tache>> si c'est un état (ce qui est le cas). Ainsi, cocher une case modifie tasks et la LazyColumn se recomposera sur l'item modifié.

UI AddTaskScreen :

```
@Composable
fun AddTaskScreen(onAdd: (String) -> Unit, onCancel: () -> Unit) {
    var texte by remember { mutableStateOf("") }
    Column(modifier = Modifier.fillMaxSize().padding(16.dp)) {
        Text("Nouvelle tâche", style =
MaterialTheme.typography.headlineSmall)
        OutlinedTextField(
            value = texte,
            onValueChange = { texte = it },
            label = { Text("Intitulé de la tâche") },
            modifier = Modifier.fillMaxWidth().padding(vertical = 16.dp)
        )
        Row {
            Button(onClick = { onCancel() }, modifier = Modifier.weight(1f))
{
                Text("Annuler")
            }
            Spacer(Modifier.width(8.dp))
            Button(onClick = {
                onAdd(texte)
            }, modifier = Modifier.weight(1f)) {
                Text("Ajouter")
            }
```

```
        }
    }
}
```

C'est un écran très simple : un champ pour le titre de la tâche (contrôlé par `texte` state), et deux boutons côte à côte pour Annuler et Ajouter. En cliquant Ajouter, on appelle `onAdd(texte)` qui, rappelez-vous, dans NavHost est relié à `todoViewModel.ajouterTache(texte)` puis `popBackStack` (retour à la liste). **Astuces UX** : on pourrait désactiver le bouton Ajouter si `texte` est vide pour éviter les tâches sans nom (via `enabled = texte.isNotBlank()` sur le Button), ou nettoyer le champ après ajout (ici on ne revient pas sur cet écran donc pas crucial). On pourrait aussi gérer le clavier (cacher le clavier au retour), mais pas nécessairement dans le temps imparti.

Test du flux complet : - L'application démarre sur TaskListScreen. Au début, la liste est vide, donc un message invite à ajouter. - L'utilisateur clique le FAB (+). NavController navigue vers AddTaskScreen. - Il entre un titre et appuie "Ajouter". Cela déclenche `todoViewModel.ajouterTache(titre)` et revient à TaskListScreen. - TaskListScreen se recompose car la liste dans le ViewModel a changé (1 tâche ajoutée). La tâche apparaît dans la LazyColumn. - L'utilisateur coche la checkbox : `onToggle` appelle viewModel.toggleDone -> modifie la liste. Compose met à jour l'UI, le texte se barre. - Il ajoute d'autres tâches etc., éventuellement supprime via la corbeille. - Si on avait la persistance Room, on aurait initialisé `tasks` depuis la DB et mis à jour la DB dans chaque opération. Sans Room, les données sont en mémoire et perdues si on tue l'app, mais c'est acceptable pour un test rapide.

Ce qui est réalisable en ~3h : Ce projet ToDo est d'ampleur réduite mais couvre nos 10 points : 1. Kotlin de base : utilisation de data class, listes, variables. 2. Android Studio/projet : on a créé un projet Compose. 3. Compose UI fondamentaux : Text, Row, Column, Button, Icon, Checkbox, etc. utilisés. 4. Navigation : NavHost avec deux écrans, usage de navController.navigate et popBackStack. 5. État remember/mutableStateOf : on s'en sert pour le champ texte, et dans ViewModel on utilise SnapshotStateList. 6. ViewModel : TodoViewModel gère l'état de l'app. 7. Formulaire : AddTaskScreen est un mini formulaire avec champ et boutons. 8. Liste LazyColumn : TaskListScreen affiche LazyColumn de tâches avec items dynamiques. 9. Room : on a montré comment on l'intégrerait (on peut ajouter, en imagination, l'annotation @Entity sur Tache, etc.). 10. Exemple complet : c'est bien notre ToDo app en entier.

Conseils finaux pour réussir un projet rapide lors d'un examen :

- **Priorisez les fonctionnalités** : commencez par mettre en place la structure (écrans + navigation + ViewModel) et une fonctionnalité de base qui marche (par ex., ajouter et lister des éléments). Assurez-vous d'avoir quelque chose de fonctionnel le plus tôt possible, puis itérez pour ajouter des détails (cases à cocher, suppression, validation de formulaire, etc.). Cela vous évite de vous retrouver avec un squelette non fonctionnel par manque de temps.
- **Utilisez le debugger et l'aperçu** : En Compose, l'aperçu est votre ami pour l'UI, et vous pouvez utiliser Logcat ou des `Log.d()` pour voir ce qui se passe sur des actions (par exemple logguer le contenu de la liste après un ajout) afin de vérifier la logique rapidement.
- **Ne restez pas bloqué** : Si une partie vous prend trop de temps (ex: configuration de Room, un bug de navigation), envisagez de la contourner temporairement (stocker en mémoire plutôt qu'en DB, utiliser une variable globale en secours, etc.) afin de présenter une application qui tourne. Vous pourrez expliquer que "faute de temps, la persistance n'est pas implémentée, mais le reste fonctionne".
- **S'appuyer sur les docs/exemples** : Durant la préparation, n'hésitez pas à avoir sous la main quelques extraits courants (pattern NavHost, instantiation Room, etc.). Cela n'est pas de la triche,

c'est de l'efficacité. En examen en conditions réelles, vous auriez accès à la documentation (souvent on l'autorise pour code, ou au moins les docs officielles).

- **Soin de l'UI minimale** : Compose facilite la mise en page, donc essayez d'aligner correctement vos éléments, de mettre des espacements (`Spacer`, `padding`) pour une UI propre. Même si le design n'est pas l'objectif principal, une app bien présentée fait meilleure impression. Le MaterialTheme par défaut donne déjà un style correct, utilisez-le (comme nos TopAppBar et FloatingActionButton).
- **Tests rapides** : Prenez le temps de tester les cas limites de votre appli : ajouter rien (devrait être ignoré ou message d'erreur), cocher/décocher plusieurs fois, supprimer la première tâche, etc. Corrigez les petites erreurs (par ex., j'ai fait attention dans toggleDone à créer une nouvelle instance de tâche pour notifier Compose).
- **Commentaires et explications** : En situation d'examen, commentez votre code pour montrer que vous comprenez ce que vous faites. Par exemple, un petit commentaire `// Utilisation de remember pour conserver le texte saisi pendant la recomposition` ou `// Navigation vers l'écran d'ajout lorsque l'utilisateur clique sur le FAB`. Cela prouve au correcteur que ce n'est pas du code cargo-cult mais bien réfléchi.

En suivant ces conseils et en s'appuyant sur tout ce qu'on a vu (Kotlin, Compose, Architecture MVVM basique, etc.), vous devriez être capable de livrer une application Android simple mais complète en 3 heures. Bonne programmation et n'oubliez pas de respirer : Compose est là pour vous simplifier la vie, faites-lui confiance .

---

[1] [4] [5] [8] [9] [12] [14] Introduction à Kotlin pour Android
http://blog.ippon.fr/2017/12/11/introduction-a-kotlin-pour-android/

[2] [3] [6] [7] Learn the Kotlin programming language | Android Developers
https://developer.android.com/kotlin/learn

[10] [11] [13] Basic syntax | Kotlin Documentation
https://kotlinlang.org/docs/basic-syntax.html

[15] | App architecture - Android Developers
https://developer.android.com/guide/topics/manifest/activity-element

[16] Activity Declaration in AndroidManifest.xml - Stack Overflow
https://stackoverflow.com/questions/19122386/activity-declaration-in-androidmanifest-xml

[17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [33] [34] [37] [38] [39] [40] [41] [42] [50] [51] [52] Tutoriel Android Compose | Jetpack Compose | Android Developers
https://developer.android.com/develop/ui/compose/tutorial?hl=fr

[31] [32] Android Compose Tutorial | Jetpack Compose | Android Developers
https://developer.android.com/develop/ui/compose/tutorial

[35] Navigate between screens with Compose | Android Developers
https://developer.android.com/codelabs/basic-android-kotlin-compose-navigation

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

[43] [44] [45] [46] ViewModel overview | App architecture | Android Developers
https://developer.android.com/topic/libraries/architecture/viewmodel

[47] [48] [49] How to Validate TextFields in a Login Form in Android using Jetpack Compose? - GeeksforGeeks
https://www.geeksforgeeks.org/how-to-validate-textfields-in-a-login-form-in-android-using-jetpack-compose/

[53] [54] [55] [56] [57] [58] [59] Save data in a local database using Room | App data and files | Android Developers
https://developer.android.com/training/data-storage/room