Sujet de TP N°9 - Application de Jeu de Scrambler

VUE D'ENSEMBLE

Dans ce TP, vous allez créer une application de jeu de scrambler, où l'utilisateur devra retrouver des mots à partir de lettres mélangées. Le jeu fonctionne avec un ensemble de mots à deviner et propose un score basé sur les réponses correctes. L'application sera structurée autour du concept de séparation des responsabilités dans l'architecture, avec un état maintenu par un `ViewModel` (state holder), un flux de données unidirectionnel et l'utilisation de Jetpack Compose pour l'interface utilisateur.

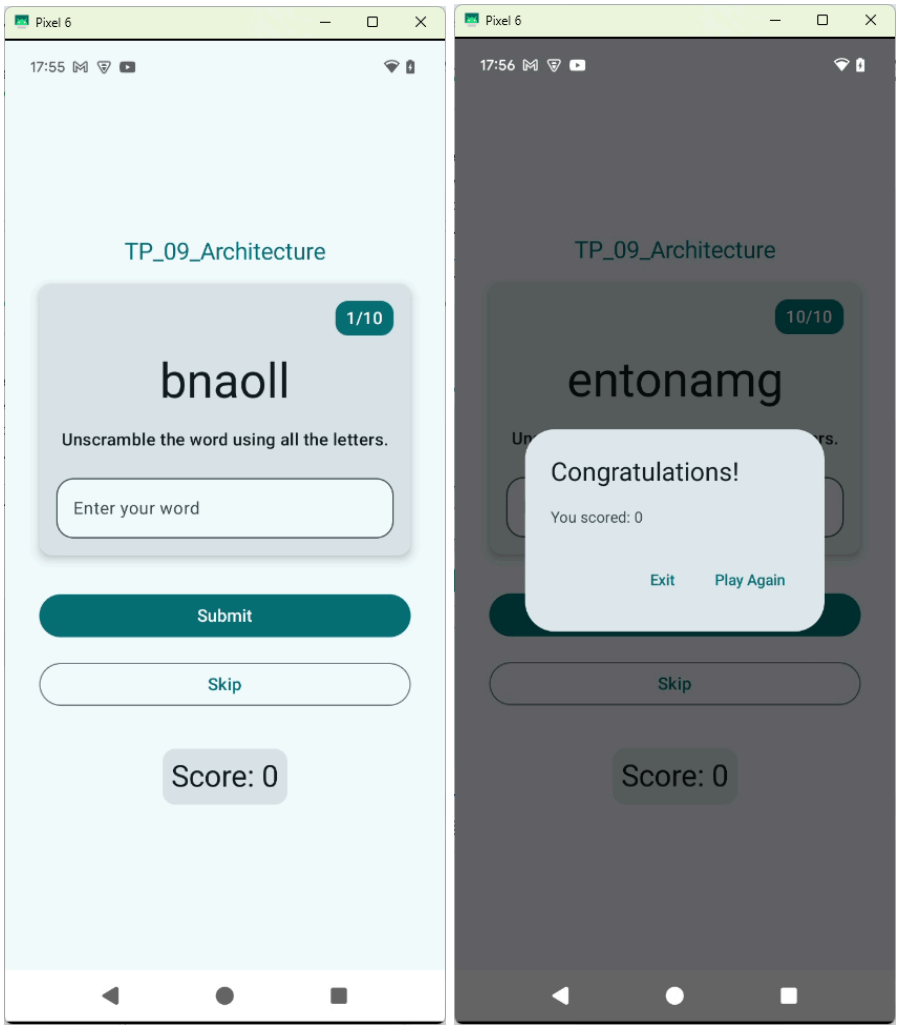
OBJECTIFS

- Apprendre à **séparer l'état de l'interface utilisateur** : Utiliser un `ViewModel` pour gérer l'état de l'application et garantir un flux de données unidirectionnel.
- Implémenter une **architecture MVVM** : Assurer une séparation claire entre la logique de l'UI, le modèle et la gestion des données.
- Gérer l'état du jeu : Développer la logique de gestion des mots, de la sélection des lettres mélangées, et du score.
- Apprendre à **utiliser StateFlow** pour rendre l'état observable dans les composables Compose.

Fonctionnalités à mettre en œuvre :

1. **Jeu de Scrambler** :
    - L'utilisateur voit un mot avec des lettres mélangées.
    - Il doit reconstituer le mot correct à partir des lettres proposées.
    - Si la réponse est correcte, il marque des points, sinon il peut retenter sa chance.
    - Proposer une fonctionnalité pour ignorer un mot.
2. **Affichage du nombre de mots proposés** :
    - Dans le coin supérieur droit de l'interface, afficher le nombre de mots déjà proposés durant la partie.
3. **Gestion du score** :
    - À chaque mot deviné correctement, le score du joueur est mis à jour.
4. **Séparation de l'interface et de la logique de jeu** :

- Utiliser un `ViewModel` pour gérer l'état du jeu.
- Assurer une séparation claire entre l'interface utilisateur et la logique du jeu (le modèle).



Étapes du TP

Étape 1 : Configuration

Vous pouvez partir du squelette d'application fourni dans les ressources (TP - 9 - Ressources), qui contient :

- Une **UI prédéfinie avec des composables** : champs de saisie, boutons, popup de fin de jeu.
- Une **liste statique de mots** pour la couche de données.

Ajout des dépendances

Ajoutez la dépendance suivante dans le fichier `libs.versions.toml` et `build.gradle (module)` :

Dans `libs.versions.toml` :

```
androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }
```

Dans `build.gradle (module)` :

```
implementation(libs.androidx.lifecycle.viewmodel.compose)
```

Synchroniser le projet

Une fois les modifications appliquées, synchronisez votre projet pour télécharger et intégrer la nouvelle dépendance.

Cette dépendance permet d'intégrer un `ViewModel` dans une application Jetpack Compose en assurant la gestion de l'état tenant compte du cycle de vie.

Étape 2 : Ajouter un ViewModel

Dans cette étape, vous allez ajouter un **ViewModel** à votre application pour gérer l'état de l'interface utilisateur (UI) de votre jeu (mot mélangé, nombre de mots, score, etc.).

1. **Création du GameViewModel**

- Créez une nouvelle classe `GameViewModel` dans le package `ui`. Elle doit étendre la classe `ViewModel`.

2. **Définir l'état de l'UI (`GameUiState`)**

- Créez une classe de données `GameUiState` pour représenter l'état de l'interface utilisateur (UI), avec au moins une propriété pour le mot mélangé.

```
data class GameUiState(
    val currentScrambledWord: String = ""
)
```

3. **Gestion de l’état avec `StateFlow`**

Utilisez StateFlow pour gérer et observer l'état de l'UI de manière réactive.

- Déclarez une propriété privée _uiState de type MutableStateFlow dans le ViewModel pour stocker l'état.

`StateFlow` permet de propager des mises à jour d'état vers vos composables Compose tout en garantissant que les données soient immuables et observables.

Dans notre cas, le StateFlow prend un `GameUiState`.

Une bonne pratique consiste à avoir une propriété privée de type `MutableStateFlow` et une propriété publique de type `StateFlow`. Ainsi, vous garantissez que seules les modifications internes de votre `ViewModel` peuvent affecter les données.

4. **Afficher un mot aléatoire**

- Dans `GameViewModel,` ajoutez une propriété appelée `currentWord` de type String qui servira à enregistrer le mot à deviner.
- Ajoutez une méthode qui permet de choisir aléatoirement un mot de la liste et qui renverra ce mot mélangé. (Attention, un mot ne doit pas pouvoir tomber plusieurs fois).
- Créez une méthode `reset()` qui permet d'initialiser et de réinitialiser le jeu. Elle permet de remettre à 0 les mots déjà joués et d'initialiser la valeur de _uiState.
- Ajoutez un bloc `init { reset() }` au ViewModel.

Étape 3 : Transmettre les données

- Transmettez l'instance du ViewModel à l'interface utilisateur, c'est-à-dire de `GameViewModel` à `MainScreen()` dans le fichier **`MainScreen.kt`**. Dans `MainScreen()`, utilisez l'instance du ViewModel pour accéder à `uiState` à l'aide de `collectAsState()`.

La fonction `collectAsState()` collecte les valeurs de ce StateFlow et représente sa dernière valeur. Chaque fois qu'une nouvelle valeur est ajoutée dans StateFlow, la valeur renvoyée State se met à jour, ce qui entraîne la recomposition.

```
import androidx.lifecycle.viewmodel.compose.viewModel
@Composable
fun MainScreen(
   gameViewModel: GameViewModel = viewModel()
) {
   val uiState by gameViewModel.uiState.collectAsState()
}
```

- Vous pouvez maintenant transmettre `currentScrambledWord` de votre UiState au composable `GameLayout().` Vous pouvez ensuite l'afficher dans le Composable.
- Exécutez l'application. Le mot mélangé doit s'afficher.

Étape 4 : Gestion des évènements

Pour respecter le principe UDF, le ViewModel transmet les données à la UI, et la UI remonte des évènements au ViewModel qui s'occupe de mettre à jouer le UiState.

- Ajoutez 2 paramètres au Composable `GameLayout()`

```
@Composable
fun GameLayout(
      currentScrambledWord: String,
      onUserGuessChanged: (String) -> Unit,
      onKeyboardDone: () -> Unit,
      modifier: Modifier = Modifier
) { … }
```

  Ces deux paramètres s'utilisent sur le Composable `OutlinedTextField()`.

- Dans le ViewModel, ajoutez deux méthodes :
  - `updateUserGuess(word: String)` qui est appelée quand l'utilisateur entre du texte. Vous pouvez stocker cette information au niveau du ViewModel dans une propriété `var userGuess by mutableStateOf("")`.
  - `checkUserGuess()` qui est appelée quand l'utilisateur valide sa proposition, soit depuis le clavier, soit depuis le bouton valider. Cette méthode sera complétée plus tard.

- Transmettez `userGuess` au Composable responsable d'afficher la saisie de l'utilisateur.
- Exécutez l'application. Vous devriez pouvoir saisir du texte.

Étape 5 : Valider le mot

- Ajoutez dans `GameUiState` une propriété qui indique si la dernière tentative est un échec.
- Dans la méthode `checkUserGuess()`, ajoutez un bloc if/else qui vérifie que la proposition est correcte.
  - Réinitialiser `userGuess` pour vider la chaîne.
  - En cas d'erreur, mettez à jour le StateFlow grâce à la méthode `update` puis `copy`.
- Transmettez ce Boolean jusqu'au paramètre `isError` de `OutlinedTextField` pour afficher l'erreur dans le champ de texte.

- Mettez à jour le label du champ de texte en fonction du Boolean indiquant l'erreur.
- Exécutez l'application et saisissez une proposition incorrecte. Vérifiez que le champ de texte devient rouge.

Étape 6 : Gestion du score et du tour

- Dans `GameUiState` ajoutez deux propriétés pour gérer le score ainsi que le tour de jeu.
- Modifiez la méthode `checkUserGuess()` pour augmenter le score en cas de réussite, incrémenter le tour de jeu et choisir un nouveau mot.
- Utilisez ces nouvelles propriétés dans les composables appropriés pour afficher les données.
- Implémentez une nouvelle méthode `skip()` qui permet de passer le mot actuel, le score ne doit pas être augmenté.
- Exécutez l'application, vous devriez à présent pouvoir jouer, il ne devrait rester que la gestion du dernier tour du jeu.

Étape 7 : Gestion du dernier tour

- Dans le ViewModel, ajoutez la logique permettant de gérer le dernier tour de jeu, remontez cette information à la UI pour afficher la popup de fin de jeu.
- Remontez les évènements de la Popup au ViewModel.
- Vérifiez que l'application fonctionne correctement.

Étape 8 : Pour aller plus loin (Bonus)

Dans ce code, la couche de données est représentée par une simple liste statique. On pourrait envisager d'ajouter une base de données à l'application, préchargée avec cette liste. Un dépôt (repository) serait alors chargé d'agir comme la **Single Source of Truth (SSOT)** pour gérer l'accès et la manipulation des données.