Projet - Application de quiz guidée avec Kotlin et Room en clean architecture

Table des matières

I. Structure du projet

Durée : 1 h 30

Environnement de travail : Avoir installé sur son ordinateur Android Studio.

Prérequis :

- Avoir les connaissances de base du langage Kotlin et de ces innovations par rapport à Java,
- Avoir les bases du langage SQL,
- Avoir des bases en programmation orientée objets.

Contexte

Ce cours aborde la création d'une application de quiz, les questions s'afficheront sous forme de liste, la réponse d'une question entraînant l'affichage de la suivante.

L'utilisateur pourra se connecter via un écran de connexion et pourra augmenter son score personnel à chaque bonne réponse.

Les points gagnés dépendent de la difficulté de la question et / ou de la rapidité de réponse.

Les questions ainsi que les utilisateurs de notre application seront stockés en local via la librairie Room.

Les notions abordées seront les suivantes : la gestion d'un projet en suivant les préceptes d'une « *clean architecture* », l'affichage d'un *recycleView* pour gérer la liste de questions, l'utilisation des coroutines de Kotlin pour récupérer nos données et la mise en pratique de la POO dans un cas concret.

Objectif

- Comprendre la structure du projet
- Comprendre la circulation des données du projet

Contexte

Il est indispensable avant de se lancer dans le code de choisir une architecture globale de fichier et une structure à suivre pour votre projet.

Cela assure une navigation naturelle des informations au sein de votre projet et diminue les possibilités d'anti-pattern et de fuites mémoires.

Fondamental

Pour ce projet nous adopterons une circulation de données qui suivra les codes de la clean architecture présenté par Robert C. Martin qui est majoritairement adopté par la communauté pour développer des projets de grande envergure sans finir avec un code qui s'entremêle et qui perd en fiabilité.

L'architecture tourne autour de 5 étapes par lesquelles les données transitent :

La Datasource : c'est l'endroit où sont stockées les données c'est le plus souvent une base de données qui peut être locale ou externe. Dans notre cas, ce sera la base de données local Room à laquelle il faudra accéder grâce à des *Data Access Object* (DAO).

Les Repositories : ce sont les classes qui sont souvent des classes *companion* unique, qui permettent d'offrir des fonctions pour accéder aux données, à travers une abstraction de la datasource qui peut être soit une locale soit externe.

Les UseCase : aussi appelés *interactors*, ils permettent de faire le pont entre la Vue et les repositories, c'est ici que s'effectue tout le code logique qui accompagne les requêtes.

Les ViewModels : ils accompagnent chaque fragment et contiennent toutes les fonctions dont le fragment a besoin pour fonctionner. Cela peut être des appels aux useCase pour nourrir le fragment d'information, cela peut aussi être du simple code logique sans requête.

Les Fragments : les fragments sont la base de la chaîne. Ils doivent rester le plus ignorant possible, leur rôle est seulement d'afficher ce qui doit être affiché, ils ne doivent pas contenir de code logique ou de requêtes.

Le circuit des informations au sein de l'application sera donc le suivant :

`Dao → Repositories → UseCase → ViewModel → Fragment`

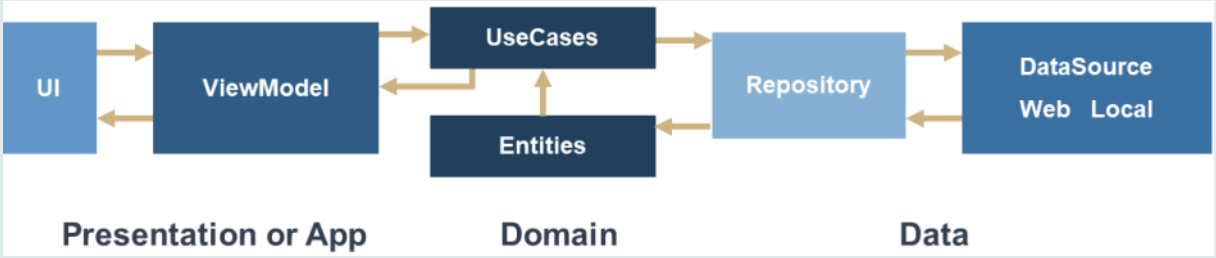
Les données de la datasource sont envoyées au fragment et le chemin inverse est adopté pour les requêtes.

En ce qui concerne la séparation des fichiers, on adoptera la séparation « *By layer* », nous aurons ainsi trois dossiers qui contiennent nos types de classes présentés au-dessus :

Présentation : cette couche contiendra tous les fragments dont le but est uniquement d'afficher des informations à l'utilisateur. Cette couche contiendra nos vues (fragments et activités) et nos viewModels.

Domain : c'est la couche qui contiendra le modèle et les classes objet de l'application ainsi que toutes les UseCases.

Data : cette couche aura pour rôle de faire la liaison entre notre application et le monde extérieur (dans notre cas ce sera la base de données Room). Cette couche sera composée de nos repositories.



Complément

The Clean Code Blog[1]

II. Dépendances

Objectifs

Ajouter les dépendances nécessaires

Contexte

Inutile de réinventer la roue, étape redondante mais nécessaire, l'ajout des dépendances nous permet d'utiliser toutes les librairies extérieures à notre projet.

Fondamental

Tout d'abord pour pouvoir interagir avec Room nous aurons besoin d'intégrer ses dépendances :

```
implementation 'android.arch.persistence.room:runtime:1.0.0-alpha1'
annotationProcessor 'android.arch.persistence.room:compiler:1.0.0-alpha1'
```

1 https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Ensuite nous aurons besoin des dépendances servant à utiliser les coroutines de Kotlin pour les données et les `ConstraintLayout` pour notre xml :

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-play-services:1.2.1'
implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
```

III. Développement de l'écran de connexion

Objectif

Mettre en place le design et le système de connexion utilisateur en local

Contexte

Pour que chaque utilisateur puisse disposer d'un score il doit pouvoir être identifiable indépendamment et doit donc pouvoir se connecter grâce à son identifiant.

Fondamental

Nous allons suivre un design simple permettant la connexion de l'utilisateur grâce à un identifiant et un mot de passe.

Dans un premier temps, les identifiants sont stockés en données brutes puis stockés sur Room quand l'application y sera connectée.

L'écran de connexion sera la première *activity* appelée au lancement de l'application et renverra l'utilisateur vers le quiz ou affichera une erreur en cas de mauvais identifiant.

Pour le bloc de connexion, nous utiliserons un `LinearLayout` qui contient nos deux `editext` et notre bouton de connexion.

Pour le code, nous suivons le modèle MVVM pour garder une séparation des rôles bien distincts. Nous avons donc `LoginActivity` qui a le rôle de View en affichant les éléments de l'UI et `LoginViewModel`. Il se charge de contrôler les identifiants de connexion.

LoginViewModel :

```
class LoginViewModel : ViewModel() {

    val loginResult: MutableLiveData<Boolean> = MutableLiveData()

    fun login(username: String, pass: String){
        if (username == "Josh" && pass == "0415"){
            loginResult.postValue( value: true)
        }else{
            loginResult.postValue( value: false)
        }
    }
}
```

On définit la LiveData recevant un boolean qui dépend de la validité des informations de connexion (ici pour l'utilisateur « *Josh* » avec le pass « *0415* »).

LoginActivity :

Gestion du clic de connexion

```
login_btn.setOnClickListener { it: View!
    loginViewModel.login(
        username_input.text.toString(),
        pass_input.text.toString()
    )
}
```

En cliquant sur le bouton de connexion on envoie les identifiants à la fonction `login` définie précédemment dans notre viewModel.

Observation de la livedata

```
loginViewModel = ViewModelProvider( owner: this).get(LoginViewModel::class.java)
loginViewModel.loginResult.observe( owner: this, Observer { t -> processLogin(t) })
```

On observe la `LiveData` définie dans le viewmodel, chaque nouvelle connexion *update* la valeur de `loginResult` qui est transmise à `processLogin()` pour procéder aux changements côté UI.

Traitement du résultat de connexion

```
private fun processLogin(isLogOk: Boolean) {
    if (isLogOk) {
        val intent = Intent( packageContext: this, MainActivity::class.java)
        startActivity(intent)
        } else {
        Toast.makeText( context: this,  text: "Wrong log", Toast.LENGTH_SHORT).show()
    }
}
```

Le résultat de la connexion est traité ici. On lance l'activité suivante si la connexion est bonne, dans le cas contraire, on affiche un *Toast*.

IV. Mise en place de la librairie Room

Objectifs

- Créer la base de donnée Room
- Définir les objets qui la composeront
- Définir les données brutes à mettre en base

Contexte

Pour que nos informations puissent circuler naturellement nous devons les encapsuler dans les classes objets qui représentent les entités de notre base de données room.

Fondamental

Dans notre cas, toutes les données qui circulent à travers l'application sont, soit des données liées aux questions de notre quiz (intitulé de la question, réponse, etc.), soit des données liées à l'utilisateur connecté (score, identifiants, etc.).

Base de données room :

```
@TypeConverters(RoomConverter::class)
@androidx.room.Database(
    entities = [UserEntity::class, QuestionEntity::class],
    version = 1,
    exportSchema = false
)
abstract class Database : RoomDatabase() {

    abstract fun questionDao(): QuestionDao
    abstract fun userDao(): UserDao

    companion object {
        var INSTANCE: Database? = null

        fun getDatabase(context: Context): Database? {
            if (INSTANCE == null) {
                synchronized(Database::class) {
                    if (INSTANCE == null) {
                        INSTANCE = Room.databaseBuilder(
                            context.applicationContext,
                            Database::class.java,
                            name: "database"
                        ).build()
                    }
                }
            }
            return INSTANCE
        }
    }
}
```

La base de donnée est définie avec un compagnon *object* pour s'assurer de n'avoir qu'une instance.

On inclut également nos DAO (Data Access Object) vu précédemment qui permettent d'accéder à nos données.

On précise dans l'annotation « `entities` », les entités qui composeront la base.

Puis on définit nos deux entités :

Entité utilisateurs

```kotlin
@Entity(tableName = "user_table")
data class UserEntity(
    @PrimaryKey(autoGenerate = true) val id: Long,
    val username: String,
    val pass: String,
    var score: Int
)
```

Entité Question

```kotlin
@Entity(tableName = "question_table")
data class QuestionEntity(
    @PrimaryKey(autoGenerate = true) val id: Long,
    val text: String,
    val answer : String
)
```

L'annotation `@Entity` de room nous permet de changer le nom de la table en bdd qui aurait été par défaut égal au nom de la classe.

`@PrimaryKey` nous permet de préciser que le champ suivant représente la clé primaire de la table et qu'elle s'auto-incrémente grâce à « *`autogenerate = true`* ».

À noter que nous utilisons ici directement les entités issues de notre base de données locale partout dans notre application.

Dans le contexte d'un plus gros projet, il est préférable d'avoir une classe objet pour la base de données et une classe objet pour la partie *model* de l'application. En effet, ces deux classes peuvent avoir des définitions et des besoins différents.

Remarque

Ces objets seront ensuite utilisés dans notre écran de quiz pour afficher le score actuel de l'utilisateur ainsi que les questions.

V. Rendre nos objets accessible : les DAO

Objectifs

- Rendre nos objets Room accessibles
- Créer nos DAO (Data Access Objects) permettant de communiquer avec nos objets
- Créer des repository qui feront la transition entre la couche domaine et la couche data

Contexte

Maintenant que nos objets sont créés il nous faut les rendre accessibles. Dans la librairie Room nos objets entités sont rendus accessibles par des DAO (Data Access Objects). Ce sont des classes qui permettent d'accéder à nos données grâce à des requêtes SQL de la même manière que sur une base de données embarquée traditionnelle comme SQLlite.

Fondamental

Une fois nos DAO créés nous y accédons grâce à des classes repository qui ont pour seul but de faire la liaison en lecture et écriture avec le reste de notre application.

Nous avons un DAO et un repository par entité pour respecter la séparation des préoccupations.

Les DAO et les repository sont dans la couche data de notre architecture car ces classes ont pour seul rôle d'interagir avec des données et n'ont aucun lien ou dépendance avec la logique de notre application.

Nous pourrions si besoin copier la couche data dans un autre projet sans avoir à modifier quoi que ce soit.

DAO question :

```
@Dao
interface QuestionDao {

    @Query( value: "DELETE  FROM question_table")
    fun deleteAll()

    @Query( value: "SELECT * FROM question_table")
    fun getAllQuestions(): Flow<MutableList<QuestionEntity>>

    @Query( value: "SELECT count(*) FROM question_table")
    fun count(): Int

    @Insert
    fun insertQuestion(user: QuestionEntity)
}
```

L'annotation `@Dao` permet d'indiquer à Room que cette classe est utilisée pour accéder à une entité.

Les fonctions de notre DAO sont présentées avec la query SQL précédée de l'annotation `@Query` suivie d'une simple définition de la fonction Kotlin qui lui est associée.

Room propose des annotations telles que `@Insert` pour simplifier et éviter d'écrire la requête SQL d'insertion.

Room nous permet de renvoyer directement des *Flow* à la sortie de notre DAO, ce qui nous permet de traiter les données de notre base de données Room comme des Flow classiques et donc d'actualiser, par exemple, notre liste de questions automatiquement si des éléments sont ajoutés en base de données.

QuestionRepository :

```
class QuestionRepository(private val questionDao: QuestionDao) {

    fun getAllQuestions(): Flow<MutableList<QuestionEntity>> {
        return questionDao.getAllQuestions()
    }

    fun insertQuestion(question: QuestionEntity){
        questionDao.insertQuestion(question)
    }
}
```

Notre repository fait appel au DAO et reprend simplement les fonctions dont nous aurons besoin.

VI. Développement de la classe du quiz et du recyclerView

Objectifs

- Créer notre layout qui intégrera chaque question
- Créer la recyclerView qui affichera nos questions

Contexte

Pour l'écran principal de quiz, nous composons simplement le layout d'un RecyclerView qui affiche le layout de chaque question et de deux textView. Un premier pour le pseudo de l'utilisateur et un autre qui comptabilise les scores au fur et à mesure des réponses.

Fondamental

La première chose à faire est de définir notre « *QuestionAdapter* » qui reçoit notre liste de questions et se charge de créer un layout par question et de les afficher sous forme de liste horizontale.

QuestionAdapter :

```
class QuestionAdapter(private val questions: MutableList<Question>) :
    RecyclerView.Adapter<QuestionAdapter.ViewHolderQuestion>() {

    var processAnswer: ((Question, String, Int) -> Unit)? = null

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): ViewHolderQuestion {
        val v: View = LayoutInflater.from(parent.context).inflate(R.layout.basic_question_item_view, parent, attachToRoot: false)
        return  ViewHolderQuestion(v)
    }

    class ViewHolderQuestion(v: View) : RecyclerView.ViewHolder(v) {
        val libelle: TextView = itemView.findViewById(R.id.question)
        val valider: TextView = itemView.findViewById(R.id.validationBtn)
        val reponse: TextView = itemView.findViewById(R.id.answerInput)
    }

    override fun getItemCount(): Int {
        return questions.size
    }

    override fun onBindViewHolder(holder: ViewHolderQuestion, position: Int) {
        holder.libelle.text = questions[position].text
        holder.valider.setOnClickListener { it: View!
            processAnswer?.invoke(questions[position], holder.reponse.text.toString(), position)
        }
    }
}
```

Le layout « *basic_question_item_view* » est un simple layout avec un `textView` pour l'intitulé de la question, un `editText` pour entrer la réponse de la question et un bouton pour valider la réponse.

Rien d'inhabituel dans l'adapter, « *processAnswer* » est une fonction lambdas qui est définie depuis le fragment de quiz et qui nous permet de traiter les résultats des questions en dehors de l'adapter.

L'équivalent en java serait de faire implémenter « *QuizFragment* » d'une interface et de la passer dans le constructeur de l'adapter.

VII. Développement de l'écran de quiz : fragment

Objectifs

- Créer notre écran de quiz
- Traiter les réponses aux questions
- Initialiser le recyclerView

Fondamental

Pour la classe du fragment, elle a deux activités distinctes : l'initialisation du recyclerView et le traitement des réponses aux questions.

processInput :

```
private fun processInput(question: Question, input: String, position: Int) {
    val isCorrect: Boolean = question.answer == input
    if (isCorrect) {
        Toast.makeText(context, text: "Bonne réponse !", Toast.LENGTH_SHORT).show()
    } else {
        Toast.makeText(context, text: "Mauvaise réponse, dommage !", Toast.LENGTH_SHORT).show()
    }
    recyclerview.smoothScrollToPosition( position: position + 1)
}
```

`processInput` vérifie si le champ entré par l'utilisateur est égal à la réponse, et affiche un Toast correspondant, puis place le RecyclerView à la prochaine question.

InitQuestions :

```
private fun initQuestions(questions: MutableList<Question>) {
    recyclerview.apply { this: QuestionsRecyclerView!
        setHasFixedSize(true)
        adapter = QuestionAdapter(
            questions
        ).apply { this: QuestionAdapter
            processAnswer = { question, answer, position ->
                processInput(question, answer, position)
            }
        }
        layoutManager = NoScrollLayoutManager(context)
    }
}
```

On initialise l'adapter en passant la liste des questions et en définissant la fonction lambdas « *processAnswer* » qui exécute « *processInput* » sur l'invoke.

« *NoScrollLayoutManager* » est une classe custom qui étend LayoutManager et qui nous permet de bloquer le scroll manuel du RecyclerView.

VIII. Développement du viewModel et des Usecases

Objectifs

- Créer notre viewModel pour le fragment quiz
- Créer notre useCase qui récupérera les questions à afficher
- Initialiser le recyclerView

Contexte

Comme vu précédemment le rôle du viewModel est de faire appel aux useCase pour que ces dernières récupèrent les données dont on a besoin.

Fondamental

Pour notre fragment « *Quiz* » la seule tâche de notre viewModel sera de récupérer la liste de questions en base de données.

Pour cela notre ViewModel exécute une coroutines dans laquelle un useCase fait appel à `questionRepository` qui nous renvoie la liste de questions.

Nous mettons une sécurité supplémentaire pour gérer la réussite ou l'échec de la requête de notre useCase qui, dans notre cas, sera une requête sur la base de données room, mais pourrait aussi être un appel à une librairie http comme retrofit pour récupérer des données sur une base de données externe.

Result :

```
sealed class Result<T>{
    data class Success<T>(val value: T?): Result<T>()
    data class Failure<T>(val throwable: Throwable): Result<T>()
}
```

La classe `sealed Result` représente le statut de notre requête, elle comporte deux sous classes :

La requête peut être de type « *Success* » et prendre en paramètre la valeur récupérée ; ou de type « *Failure* » et comporter un « *Throwable* » représentant l'erreur arrivée pendant la requête.

« *T* » est un générique qui représente le type de la valeur à récupérer.

UseCase :

```
class GetQuestionsToDoUseCase(dao: QuestionDao) :
    IFetchUseCase<MutableList<QuestionEntity>> {
    val questionRepository: QuestionRepository = QuestionRepository(dao)
    override suspend fun execute(): Result<MutableList<QuestionEntity>> {
        return withContext(Dispatchers.IO) { this: CoroutineScope
            return@withContext try {
                val questions = questionRepository.getAllQuestions().first()
                Result.Success(questions)
            } catch (ex: Exception) {
                Result.Failure<MutableList<QuestionEntity>>(Throwable(ex.message))
            }
        }
    }
}
```

Notre useCase prend en paramètre le DAO pour pouvoir instancier notre repository, il implémente une interface « *IFetchUseCase* » qui comporte une seul fonction « *execute* » commune à tous nos useCase qui renvoie un objet de type « *Result* ».

On exécute ensuite la requête dans notre coroutines avec un try qui nous renvoie notre « *Result.Sucess* » avec notre value en paramètre ou un « *Result.Failure* » avec un throwable dans notre catch.

viewModel :

```
class QuizzViewModel : ViewModel() {

    val questions: MutableLiveData<MutableList<QuestionEntity>> = MutableLiveData()

    fun getAllQuestions(context: Context) {
        CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
            val questionsResult = GetQuestionsToDoUseCase(
                Database.getDatabase(context)!!.questionDao()
            ).execute()
            if (questionsResult is Result.Success) {
                questions.postValue(questionsResult.value)
            }
        }
    }
}
```

Le ViewModel fait appel à « `execute` » de notre useCase, vérifie ensuite si le résultat est un succès puis envoie la « `value` » de notre objet « `Result` » dans une liveData qui déclenche automatiquement le *refresh* de notre recyclerview.

IX. Mise en place de notre UseCase de connexion

Objectifs

- Mettre en place le useCase qui permettra la vérification des identifiants
- Récupérer l'utilisateur et le stocker en variable global
- Câbler notre viewModel au useCase

Nous avions laissé notre viewModel de connexion avec des données brut, nous allons maintenant mettre en place un useCase de connexion permettant de vérifier les identifiants entrés lors de la connexion et de permettre à l'utilisateur de se connecter.

Nous stockons ensuite notre utilisateur récupéré dans une classe *static* afin de le rendre unique au travers de l'application.

ConnectUseCase :

```
class ConnectUseCase(val username: String, val pass: String, userDao: UserDao) : IFetchUseCase<Nothing> {
    val userRepository: UserRepository = UserRepository(userDao)
    override suspend fun execute(): Result<Nothing> {
        return withContext(Dispatchers.IO) { this: CoroutineScope
            return@withContext try {
                val user = userRepository.getUser(username, pass)
                SessionManager.apply { this: SessionManager
                    currentUser = user.first()
                    isConnected = true
                }
                Result.Success( value: null)
            } catch (ex: Exception) {
                Result.Failure<Nothing>(Throwable(ex.message))
            }
        }
    }
}
```

Même structure que pour récupérer nos questions, notre useCase prend en paramètre le userDao pour instancier notre repository. Il implémente « *`IFetchUseCase`* » et renvoie un objet « *`Result`* » avec le generic a « *`Nothing`* » car nous enregistrons notre user directement dans le useCase.

On enregistre l'utilisateur de la session et un boolean pour la connexion dans une classe companion object pour pouvoir récupérer ces valeurs dans l'application si besoin.

```
class SessionManager {
    companion object{
        var currentUser: UserEntity? = null
        var isConnected: Boolean = false
    }
}
```

LoginViewModel :

```
class LoginViewModel : ViewModel() {

    val loginResult: MutableLiveData<Boolean> = MutableLiveData()

    fun login(username: String, pass: String, context: Context){
        CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
            val connectResult = ConnectUseCase(username, pass, Database.getDatabase(context)!!.userDao()).execute()
            if (connectResult is Result.Success){
                loginResult.postValue( value: true)
            }else{
                loginResult.postValue( value: false)
            }
        }
    }
}
```

Comme pour nos questions, on vérifie si le résultat est un succès ou un échec et on enregistre le `boolean` dans une `liveData` pour la transmettre à notre fragment.

X. Initialisation des données en base de données

Objectifs

Insérer nos données brutes dans room

Contexte

Maintenant que tout est mis en place, nous pouvons insérer nos données dans notre base de données room. Dans notre cas, ce seront de simples données insérées au lancement de l'*app* en faisant directement appel au repository. Dans le cas d'un vrai projet, ces données seraient sûrement un échantillon récupéré d'une base de données externe.

Fondamental

Nous allons simplement mettre en place une fonction qui crée un *array* de « `QuestionEntity` » ainsi que notre utilisateur puis placer cette fonction dans notre activity de login au démarrage de l'application.

```
private fun populateDatabase() {
    val questionRepository = QuestionRepository(Database.getDatabase(baseContext)!!.questionDao())
    val userRepository = UserRepository(Database.getDatabase(baseContext)!!.userDao())

    var questions : MutableList<QuestionEntity> = ArrayList()

    questions.add(QuestionEntity( id: 0, text: "Quel est le seul pays qui possede un drapeau carré ?", answer: "La Suisse"))
    questions.add(QuestionEntity( id: 0, text: "Quel est la capital de l'ukraine ?", answer: "Kiev"))
    questions.add(QuestionEntity( id: 0,
        text: "Depuis quelle année Kotlin est le language officiel de devellopement sur Android", answer: "2019"))

    CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
        questions.forEach{questionRepository.insertQuestion(it)}
        userRepository.insertQuestion(UserEntity( id: 0, username: "JohnDoe", pass: "4558", score: 0))
    }
}
```

On boucle ensuite sur le tableau pour insérer chaque question grâce au repository.

XI. Gestion du score

Objectifs

- Mettre à jour le score de l'utilisateur après la réponse à une question
- Update le score de l'utilisateur en base de données

Contexte

Une fois que l'utilisateur a répondu à une question nous souhaitons que l'application affiche son score qui incrémente dans le cas d'une bonne réponse et perd un point dans le cas inverse.

Nous devons aussi implémenter une fonction pour mettre à jour l'utilisateur en base de données pour mettre à jour le score.

Pour cela nous devons garder le même sens de circulation des données :

`Dao → repository → usecase → viewModel → View`

DAO :

```
@Update
fun update(user: UserEntity)
```

Room propose aussi une annotation update qui nous évite d'écrire la requête SQL.

Repository :

```
fun updateUser(user: UserEntity){
    userDao.update(user)
}
```

useCase :

```
class UpdateUserUseCase(val user: UserEntity, userDao: UserDao) : IFetchUseCase<Nothing> {
    val userRepository: UserRepository = UserRepository(userDao)
    override suspend fun execute(): Result<Nothing> {
        return try {
            userRepository.updateUser(user)
            Result.Success( value: null)
        } catch (ex: Exception) {
            Result.Failure(Throwable(ex.message))
        }
    }
}
```

ViewModel :

```
fun updateUser(user: UserEntity, context: Context){
    CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
        UpdateUserUseCase(user , Database.getDatabase(context)!!.userDao()).execute()
    }
}
```

Fragment :

```
private fun processInput(question: QuestionEntity, input: String, position: Int) {
    val isCorrect: Boolean = question.answer == input
    if (isCorrect) {
        Toast.makeText(context, text: "Bonne réponse !", Toast.LENGTH_SHORT).show()
        SessionManager.currentUser!!.score ++
    } else {
        Toast.makeText(context, text: "Mauvaise réponse, dommage !", Toast.LENGTH_SHORT).show()
        SessionManager.currentUser!!.score --
    }
    context?.let { quizzViewModel.updateUser(SessionManager.currentUser!!, it) }
    recyclerview.smoothScrollToPosition( position: position + 1)
}
```

On augmente ou on diminue le score de l'utilisateur actuel en fonction d'une bonne ou d'une mauvaise réponse, puis on appelle `updateUser` du viewmodel pour mettre à jour notre utilisateur en bdd.

XII. Finir avec un peu de design

Objectifs

- Styliser nos buttons et nos Editext
- Changer les polices et l'alignement pour un meilleur rendu

Contexte

Après nous être penché sur l'aspect technique et architectural de notre application, il faudrait maintenant prendre un peu de temps pour donner un rendu acceptable à nos éléments visuels en modifiant leur emplacement et leur design.

Fondamental

Dans un premier temps nous centrons tous nos layout pour donner un aspect ordonné à notre vue et pour garder un espacement et une marge similaire entre les éléments de la vue de connexion.

Ensuite, nous uniformisons la police d'écriture pour avoir la même taille et la même police, puis nous mettons en gras les titres.

Enfin, nous adoptons un style spécifique similaire pour notre bouton de validation de l'écran de connexion et de l'écran de quiz. Nous faisons de même avec un style spécifique pour les *editext username* et mot de passe, et l'*editext* de la réponse.

XIII. Pour aller plus loin...

Notre quiz est maintenant terminé, l'utilisateur peut se connecter via ses identifiants et il peut tenter d'accumuler des points en répondant juste aux questions qui s'enchaînent. Toutes les informations sont bien insérées et récupérées en base de données.

Du fait que notre application respecte les préceptes de la clean architecture, elle est ouverte à toutes modifications chaque « *feature* » suivra le même chemin que les fonctions que nous venons d'implémenter.

Des idées de développement supplémentaires pourraient être :

- Ajouter un champs difficulté à chaque question et donner plus ou moins de point au joueur en fonction de la difficulté de la question.
- Rajouter de la même façon un *timer* pour chronométrer le temps de réponse et donner plus ou moins de point en fonction de la rapidité de réponse du joueur.
- On pourrait également enregistrer l'id des bonnes et des mauvaises réponses du joueur pour éviter de lui reposer les mêmes questions ou pour développer un écran d'historique des dernières questions répondues.