Développement Android

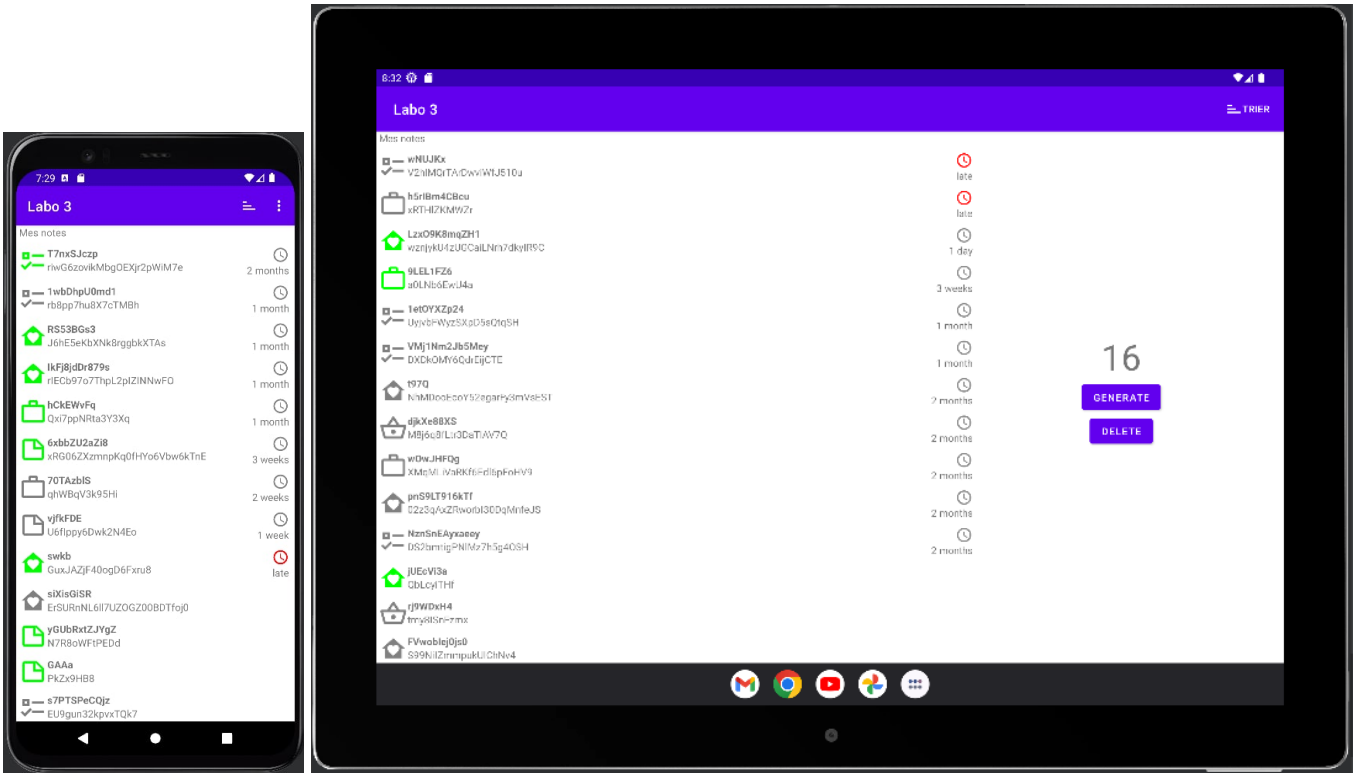
Laboratoire n°3: Architecture MVVM, utilisation d'une base de données Room et d'un RecyclerView

Friedli Jonathan, Marengo Stéphane, Silvestri Géraud

07.12.2022

Introduction

Le but de ce laboratoire est de développer une application android basée sur une architechture `MVVM` et ayant une base de données `Room`. Cette application permet d'afficher une liste de notes et doit proposer une interface graphique différentes pour les smartphones et les tablettes:



1. Détails d'implémentation

1.1. Layout

Puisque l'inferface doit être différente entre la version téléphone et la version tablette, nous avons créé un dossier `layout` et un autre dossier `layout-sw600dp`. Les deux dossiers contiennent un fichier `activity_main.xml` contenant le layout de l'application. De cette manière, le layout de l'application est différent selon la taille de l'appareil. A partir de 600dp, l'appareil est considéré comme une tablette.

Layout téléphone:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    [...] >

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/notes_fragment"
        android:name="ch.heigvd.daa_lab3.fragments.NotesFragment"
        [...] />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Layout tablette:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    [...] >

    <androidx.fragment.app.FragmentContainerView
            android:id="@+id/notes_fragment"
            android:name="ch.heigvd.daa_lab3.fragments.NotesFragment"
            app:layout_constraintWidth_percent=".66"
            [...] />

    <androidx.fragment.app.FragmentContainerView
            android:id="@+id/controle_fragment"
            android:name="ch.heigvd.daa_lab3.fragments.ControlsFragment"
            app:layout_constraintWidth_percent=".33"
            [...] />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Les deux layouts sont très similaires, la seule différence est que dans le layout de la tablette, nous avons un autre `FragmentContainerView` pour le `ControlsFragment`. Les fragments sont référencés dans le layout grâce à la propriété `android:name` des `FragmentContainerView`.

Menu:

Pour les menus, nous avons procédé de la même manière que précédemment, en créant donc un dossier `menu` et un autre dossier `menu-sw600dp` contenant tout deux un fichier `main_menu.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/menu_show_sorting_option"
        android:title="@string/menu_sort"
        android:icon="@drawable/sort"
        app:showAsAction="ifRoom|withText">
        <menu>
            <item
                android:id="@+id/menu_sort_creation_date"
                android:title="@string/menu_sort_creation_date" />
            <item
                android:id="@+id/menu_sort_eta"
                android:title="@string/menu_sort_eta" />
        </menu>
    </item>
    <!-- Les deux éléments ci-dessous ne sont pas présents dans le main_menu.xml
pour tablette -->
    <item
        android:id="@+id/menu_generate"
        android:title="@string/menu_generate" />
    <item
        android:id="@+id/menu_delete_all"
        android:title="@string/menu_delete_all" />
</menu>
```

Les ids des éléments sont les mêmes dans les deux menus, permettant ainsi de ne pas faire de différence lors du traitement des événements de clic sur les éléments du menu.

1.2. Fragments

Nous avons créé plusieurs fragments pour l'application. Le fragment `NotesFragment` contient la `RecyclerView` et le fragment `ControlsFragment` contient les boutons pour générer une note et pour supprimer toutes les notes. Le `ControlsFragment` n'est pas présent dans la version téléphone.

Dans les deux fragments, le `ViewModel` est stocké dans un champ privé et est initialisé de la manière suivante:

```
private val viewModel: NotesViewModel by activityViewModels {
    NotesViewModelFactory((requireActivity().application as MyApp).repository)
}
```

NotesFragment:

Ce fragment possède un second champ, `notesAdapter`, initialisé de la manière suivante:

```
private val notesAdapter: NotesAdapter by lazy { NotesAdapter() }
```

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    restoreSortType()

    view.findViewById<RecyclerView>(R.id.notes_list).apply {
        adapter = notesAdapter
        layoutManager = LinearLayoutManager(context)
    }

    viewModel.allNotes.observe(viewLifecycleOwner) { notes ->
        notesAdapter.items = notes
    }
}

fun sortByEta() {
    NotesAdapter.SortType.ETA.let {
        notesAdapter.sortedBy = it
        saveSortType(it)
    }
}

fun sortByCreationDate() {
    NotesAdapter.SortType.CREATION_DATE.let {
        notesAdapter.sortedBy = it
        saveSortType(it)
    }
}
```

Lorsque la vue est créée, nous assignons simplement à la `RecyclerView` l'adapter ainsi qu'un `LinearLayoutManager` pour avoir un affichage liste. Nous observons ensuite les notes du `ViewModel` et nous les assignons à l'adapter.

Les deux méthodes publiques `sortByEta` et `sortByCreationDate` permettent de trier la liste de la `RecyclerView` et seront appelées par `MainActivity`.

Les méthodes `saveSortType()` et `restoreSortType()` seront expliquées dans la section **Question complémentaire** décrite plus bas.

ControlsFragment:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    view.findViewById<Button>(R.id.generate_button).setOnClickListener {
        viewModel.generateANote()
    }

    view.findViewById<Button>(R.id.delete_all_button).setOnClickListener {
        viewModel.deleteAllNotes()
    }

    val counterText = view.findViewById<TextView>(R.id.nb_notes_text)
    viewModel.countNotes.observe(viewLifecycleOwner) { count ->
        counterText.text = count.toString()
    }
}
```

Lorsque la vue est créée, nous assignons les listeners aux boutons et nous observons le nombre de notes du `ViewModel` pour mettre à jour le `TextView` affichant le nombre de notes.

Les deux boutons appellent simplement les méthodes `generateANote` et `deleteAllNotes` du `ViewModel`.

1.3. Base de données

La base de données a été implémentée comme vu en cours.

MyDatabase

```
@Database(
    entities = [Note::class, Schedule::class],
    version = 1,
    exportSchema = true
)
@TypeConverters(CalendarConverter::class)
abstract class MyDatabase : RoomDatabase() {
    abstract fun noteDAO(): NoteDAO

    companion object {
        private const val NB_NOTES_TO_CREATE_IF_EMPTY = 10

        private var INSTANCE: MyDatabase? = null

        fun getDatabase(context: Context): MyDatabase {
            return INSTANCE ?: synchronized(this) {
                INSTANCE = Room.databaseBuilder(
                    context.applicationContext,
                    MyDatabase::class.java, "mydatabase.db"
                )
                    .fallbackToDestructiveMigration()
                    .addCallback(populateCallBack())
                    .build()
                INSTANCE!!
            }
        }

        private fun populateCallBack(): Callback {
            return object : Callback() {
                override fun onOpen(db: SupportSQLiteDatabase) {
                    INSTANCE?.let { database ->
                        if (db.query("SELECT * FROM Note").count > 0) {
                            return
                        }

                        thread {
                            repeat(NB_NOTES_TO_CREATE_IF_EMPTY) {
                                val note = Note.generateRandomNote()
                                val schedule = Note.generateRandomSchedule()

                                val noteId = database.noteDAO().insert(note)
                                if (schedule != null) {
                                    schedule.ownerId = noteId
                                    database.noteDAO().insert(schedule)
                                }
                            }
```

```
                        }
                    }
                }
            }
        }
    }
}
```

Comme vu en cours, la liste des entités gérées par la base de données est définie dans l'annotation `@Database`. Nous avons également défini une classe `CalendarConverter` qui permet de convertir un `Calendar` en `Long` et vice-versa. Cela nous permettra de stocker les dates dans la base de données.

La méthode statique `getDatabase` permet de récupérer une instance de la DB grâce au pattern Singleton.

Pour ajouter des données à la base de données, nous avons implémenté un `Callback` qui va vérifier si la DB est vide à son ouverture. Si c'est le cas, il va créer un nombre de notes aléatoires (défini par la constante `NB_NOTES_TO_CREATE_IF_EMPTY`) et les insérer dans la base de données.

Plutôt que de créer une classe interne pour le `Callback`, nous avons créé méthode statique `populateCallBack` qui retourne un objet de type `Callback` implémentant la méthode `onOpen`. Cette dernière vérifie si la DB est vide en exécutant une requête directement, sans passer par le DAO, pour éviter de recevoir un `LiveData` et attendre son résultat. L'insertion, quant à elle, est effectuée au travers du DAO car nous avons besoin de récupérer l'ID de la note insérée.

NoteDAO:

```
@Dao
interface NoteDAO {
    @Insert
    fun insert(note: Note): Long

    @Insert
    fun insert(schedule: Schedule): Long

    @Query("SELECT COUNT(*) FROM Note")
    fun getCount(): LiveData<Long>

    @Transaction
    @Query("SELECT * FROM Note")
    fun getAllNotes(): LiveData<List<NoteAndSchedule>>

    @Query("DELETE FROM Note")
    fun deleteAllNotes()
}
```

Nous retournons directement des `NoteAndSchedule` sans distinction.

Les méthodes d'insertion retournent l'ID de l'entité insérée pour pouvoir ajouter facilement les `Schedule` aux notes.

DataRepository

```
class DataRepository(
    private val noteDAO: NoteDAO,
    private val applicationScope: CoroutineScope
) {
    val allNotes = noteDAO.getAllNotes()
    val notesCount = noteDAO.getCount()

    fun insert(note: Note, schedule: Schedule?) {
        applicationScope.launch {
            val noteId = noteDAO.insert(note)

            if (schedule != null) {
                schedule.ownerId = noteId
                noteDAO.insert(schedule)
            }
        }
    }

    fun insertRandomNote() {
        applicationScope.launch {
            val note = Note.generateRandomNote()
            val schedule = Note.generateRandomSchedule()
            insert(note, schedule)
        }
    }

    fun deleteAllNotes() {
        applicationScope.launch {
            noteDAO.deleteAllNotes()
        }
    }
}
```

Le `ViewModel` utilise ces quelques méthodes pour interagir avec les données. Chacune de ces méthodes est exécutée dans un `CoroutineScope` afin de ne pas bloquer le thread principal.

CalendarConverter

```
class CalendarConverter {
    @TypeConverter
    fun toCalendar(dateLong: Long): Calendar =
        Calendar.getInstance().apply {
            time = Date(dateLong)
        }

    @TypeConverter
    fun fromCalendar(date: Calendar) =
```

```
        date.time.time // Long
}
```

Ce converter permet de convertir un `Calendar` en `Long` et vice-versa afin de pouvoir stocker les dates dans la base de données.

MyApp

```
class MyApp : Application() {
    private val applicationScope = CoroutineScope(SupervisorJob())

    val repository by lazy {
        val database = MyDatabase.getDatabase(this)
        DataRepository(database.noteDAO(), applicationScope)
    }
}
```

La classe `MyApp` permet de créer une seule instance de `DataRepository` qui sera accessible depuis n'importe quelle activité de l'application.

1.4. ViewModel

L'implémentation du `ViewModel` est identique à celle proposée dans la donnée.

NoteViewModel

```
class NotesViewModel(private val repository: DataRepository) : ViewModel() {
    val allNotes = repository.allNotes
    val countNotes = repository.notesCount

    fun generateANote() {
        repository.insertRandomNote()
    }

    fun deleteAllNotes() {
        repository.deleteAllNotes()
    }
}
```

NoteViewModelFactory

```
class NotesViewModelFactory(private val repository: DataRepository) :
ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(NotesViewModel::class.java)) {
            return NotesViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Nous utilisons une factory pour instancier le `ViewModel` avec le `Repository` en paramètre.

1.5. MainActivity

```
class MainActivity : AppCompatActivity() {
    private val notesFragment by lazy {
        supportFragmentManager.findFragmentById(R.id.notes_fragment) as
NotesFragment
    }

    private val viewModel: NotesViewModel by viewModels {
        NotesViewModelFactory((application as MyApp).repository)
    }

    // Utilisation du menu décrit plus haut
    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.main_menu, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_generate -> {
                viewModel.generateANote()
                true
            }
            R.id.menu_delete_all -> {
                viewModel.deleteAllNotes()
                true
            }
            R.id.menu_sort_eta -> {
                notesFragment.sortByEta() // Appel de méthode publique du fragment
                true
            }
            R.id.menu_sort_creation_date -> {
                notesFragment.sortByCreationDate() // Appel de méthode publique du
fragment
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }
}
```

Le `ViewModel` est instancié avec la factory créée précédemment.

La réécriture de la méthode `onCreateOptionsMenu` permet d'ajouter le menu décrit plus haut. Les actions de chaque élément du menu sont définies dans la méthode `onOptionsItemSelected`.

Le champ `notesFragment` permet de récupérer une référence vers le fragment des notes. Ceci nous permet ensuite d'appeler ses méthodes publiques de tris.

1.6. Adapters

NoteAdapter

```
class NotesAdapter(items: List<NoteAndSchedule> = listOf()) :
    RecyclerView.Adapter<NotesAdapter.ViewHolder>() {
    companion object {
        private const val NOTE = 1
        private const val NOTE_WITH_SCHEDULE = 2
        private val SCHEDULE_UNITS = arrayOf(
            ChronoUnit.MONTHS,
            ChronoUnit.WEEKS,
            ChronoUnit.DAYS,
            ChronoUnit.HOURS,
            ChronoUnit.MINUTES
        )
    }

    var items: List<NoteAndSchedule> = items
        set(value) {
            val sortedValue = sort(value)
            val diffCallback = NotesDiffCallback(items, sortedValue)
            val diffItems = DiffUtil.calculateDiff(diffCallback)
            field = sortedValue
            diffItems.dispatchUpdatesTo(this)
        }

    var sortedBy = SortType.CREATION_DATE
        set(value) {
            if (field != value) {
                field = value
                items = items // On force le tri
            }
        }

    enum class SortType {
        CREATION_DATE, ETA
    }

    private fun sort(values: List<NoteAndSchedule>): List<NoteAndSchedule> {
        return when (sortedBy) {
            SortType.CREATION_DATE -> values.sortedBy { it.note.creationDate }
            SortType.ETA -> values.sortedWith(compareBy<NoteAndSchedule> {
it.schedule == null }
                .thenBy { it.schedule?.date })
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder
{
        val layout = when (viewType) {
            NOTE -> R.layout.list_item_note
```

```
            NOTE_WITH_SCHEDULE -> R.layout.list_item_note_with_schedule
            else -> throw IllegalArgumentException("Unknown view type $viewType")
        }

        return ViewHolder(
            LayoutInflater.from(parent.context).inflate(layout, parent, false)
        )
    }

    [...]

    override fun getItemViewType(position: Int): Int {
        return if (items[position].schedule == null) NOTE else NOTE_WITH_SCHEDULE
    }

    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        // Récupération des vues
        private val typeIcon by lazy { view.findViewById<ImageView>
(R.id.item_note_icon_type) }
        [...]

        fun bind(noteAndSchedule: NoteAndSchedule) {
            val note = noteAndSchedule.note
            val schedule = noteAndSchedule.schedule

            // Remplissage des champs présents dans les deux layouts
            title.text = note.title
            [...]

            // Remplissage des champs spécifiques au layout avec schedule
            if (schedule != null) {
                val (text, color) = getScheduleTextAndColor(schedule.date,
scheduleIcon.context)
                scheduleText.text = text
                scheduleIcon.setColorFilter(color)
            }
        }

        private fun getScheduleTextAndColor(
            scheduleDate: Calendar,
            context: Context
        ): Pair<String, Int> {
            // Si la date est passée, on retourne directement "late"
            if (scheduleDate.before(Calendar.getInstance())) {
                return context.getString(R.string.schedule_late) to
                        ContextCompat.getColor(context, R.color.schedule_late)
            }

            val color = ContextCompat.getColor(context, R.color.schedule_in_time)
            val today = LocalDateTime.now()
            val targetDate = LocalDateTime.ofInstant(
                scheduleDate.toInstant(),
                TimeZone.getDefault().toZoneId()
            )
```

```
            val formatter =
                MeasureFormat.getInstance(Locale.getDefault(),
MeasureFormat.FormatWidth.WIDE)

            for (unit in SCHEDULE_UNITS) {
                val delta = unit.between(today, targetDate)
                if (delta <= 0) {
                    continue
                }
                return formatter.formatMeasures(Measure(delta,
unit.toMeasureUnit())) to color
            }

            return context.getString(R.string.schedule_now) to color
        }

        private fun ChronoUnit.toMeasureUnit(): MeasureUnit {
            return when (this) {
                ChronoUnit.MONTHS -> MeasureUnit.MONTH
                [...]
                else -> throw UnsupportedOperationException("Unsupported unit
$this")
            }
        }
    }
}
```

Les objets de la liste seront affichés à l'aide de deux layout différents (`list_item_note` et `list_item_note_with_schedule`), comme indiqué dans la donnée. Pour ce faire, la méthode `getItemViewType` doit retourner le type de vue à utiliser pour l'object à la position donnée. Nous avons donc deux constantes `NOTE` et `NOTE_WITH_SCHEDULE` qui correspondent aux deux types de vue. Dans la méthode `onCreateViewHolder`, nous retournons un `ViewHolder` possédant le layout correspondant au type de vue voulu.

La méthode `bind` du `ViewHolder` va remplir les vues avec les données de l'objet. Les références aux vues sont récupérées dans un bloc `lazy` pour éviter des recherches inutiles. Par exemple, si l'objet à afficher est de type `NOTE_WITH_SCHEDULE`, il n'est pas nécessaire d'essayer de récupérer une référence vers le `TextView item_note_schedule_text`.

Pour la gestion du tri, nous avons un enum `SortType` ainsi qu'un champ `sortedBy` possédant un setter qui va réaffecter la liste à elle-même pour forcer un tri en cas de changement. La méthode privée `sort` retourne la liste triée en fonction de la valeur de `sortedBy`. Dans le setter d'`items`, la liste est d'abord triée puis les différences sont calculées avec `DiffUtil` grâce à la classe `NoteDiffCallback` décrite plus bas.

Pour calculer le temps restant, nous utilisons la classe `ChronoUnit` permettant d'obtenir des différences dans plusieurs unités de temps (jours, semaines, mois, ...). Le code va trouver la différence entre deux dates dans l'unité la plus grande possible (ici, le mois) en parcourant le tableau `SCHEDULE_UNITS`. Par exemple, si il reste 5 mois et 2 jours, on obtiendra une différence de 5.

Ce résultat est ensuite formaté en utilisant la classe `MeasureFormat`. Cette classe permet de formatter un nombre et une unité en une chaîne de caractères localisée. Cela permet, par exemple, de retourner `1 month`

en anglais et `1 mois` en français. De plus, cela prend en compte le pluriel.

Pour combiner les deux, nous avons créé une méthode d'extension `ChronoUnit.toMeasureUnit` permettant de faire le lien entre `ChronoUnit` et `MeasureUnit`.

NotesDiffCallback

```
class NotesDiffCallback(
    private val oldList: List<NoteAndSchedule>,
    private val newList: List<NoteAndSchedule>
) : DiffUtil.Callback() {
    override fun getOldListSize(): Int = oldList.size

    override fun getNewListSize(): Int = newList.size

    override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int):
Boolean {
        return oldList[oldItemPosition].note.noteId ==
newList[newItemPosition].note.noteId
    }

    override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int):
Boolean {
        return oldList[oldItemPosition] == newList[newItemPosition]
    }
}
```

Dans la méthode `areContentsTheSame`, nous comparons directement les objets `NoteAndSchedule` avec l'opérateur `==` car il s'agit de `data class` et que leurs propriétés sont toutes définies dans le constructeur. En effet, Kotlin génère automatiquement, entre autre, une méthode `equals` qui compare les propriétés définies dans le constructeur des `data class`. Le contructeur de `NoteAndSchedule` contient les propriétés `note` et `schedule` qui sont elles-mêmes des `data class`, permettant ainsi à la comparaison de s'effectuer sur l'entier des propriétés.

2. Questions complémentaires

2.1 **Quelle est la meilleure approche pour sauver le choix de l'option de tri de la liste des notes ? Vous justifierez votre réponse et l'illustrez en présentant le code mettant en œuvre votre approche.**

Nous avons décidé de gérer le tri entièrement dans l'adapter c'est à dire au niveau des vues. Avec une telle approche, nous estimons que l'état du tri ne doit pas être stocké dans le `ViewModel` car il est spécifique à la vue. C'est donc au fragment `NotesFragment` de gérer cet état.

Nous avons décidé d'utiliser les `SharedPreferences` de l'application permettant ainsi de conserver l'état du tri même si l'application est fermée. Pour cela, nous avons créé une méthode `saveSortType` dans la classe `NotesFragment` qui est appelée à chaque fois que l'utilisateur change le tri:

```
private fun saveSortType(sortType: NotesAdapter.SortType) {
    requireActivity().getPreferences(Context.MODE_PRIVATE).edit {
        putString(SORTED_BY_KEY, sortType.name)
    }
}

fun sortByEta() {
    NotesAdapter.SortType.ETA.let {
        notesAdapter.sortedBy = it
        saveSortType(it)
    }
}

fun sortByCreationDate() {
    NotesAdapter.SortType.CREATION_DATE.let {
        notesAdapter.sortedBy = it
        saveSortType(it)
    }
}
```

La valeur est stockée sous forme de `String` car il est facile de créer un `enum` à partir d'une `String` avec la méthode `valueOf`.

La restauration, quant à elle, est effectuée à l'aide de la méthode `restoreSortType` lors de la création de la vue:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    restoreSortType()

    [...]
}
```

```
private fun restoreSortType() {
    requireActivity().getPreferences(Context.MODE_PRIVATE)
        .getString(SORTED_BY_KEY, null)
        ?.let { // Seulement si une valeur a été trouvée
            notesAdapter.sortedBy = NotesAdapter.SortType.valueOf(it)
        }
}
```

La clé associée à la valeur stockée dans les `SharedPreferences` est définie dans une constante de la classe:

```
companion object {
    private const val SORTED_BY_KEY = "SORTED_BY_KEY"
}
```

2.2 **L'accès à la liste des notes issues de la base de données `Room` se fait avec une `LiveData`. Est ce que cette solution présente des limites ? Si oui, quelles sont-elles ? Voyez-vous une autre approche plus adaptée ?**

Oui, cette solution présente des limites. En effet, la requête suivante va charger toutes les données dans la `LiveData` ce qui peut rapidement surcharger la mémoire. Cela n'est donc clairement pas adapté pour des grandes bases de données.

```
@Query("SELECT * FROM Note")
fun getAllNotes(): LiveData<List<NoteAndSchedule>>
```

Une meilleure approche serait d'utiliser un `Flow` qui permet de récupérer les données au fur et à mesure. Cela permet de ne pas surcharger la mémoire et de ne charger que les données nécessaires.

Il serait également possible de paginer les données à l'aide de la librairie `Paging`.

Au final, la solution à privilégier dépendra du besoin de l'application. Si l'on souhaite afficher toutes les données, il est préférable d'utiliser un `Flow`. Si l'on souhaite afficher une liste paginée, il est préférable d'utiliser la librairie `Paging`. Pour les cas simples comme celui-ci, les `LiveData` sont suffisantes.

2.3 **Les notes affichées dans la `RecyclerView` ne sont pas sélectionnables ni cliquables. Comment procéderiez-vous si vous souhaitiez proposer une interface permettant de sélectionner une note pour l'éditer ?**

Il faut ajouter un paramètre au constructeur de l'adapter permettant de passer un callback prenant en paramètre l'élément cliqué:

```
class NotesAdapter(
    private val callback: (NoteAndSchedule) -> Unit,
    items: List<NoteAndSchedule> = listOf()
)
```

Ensuite, il faut modifier la méthode `onBindViewHolder` de l'adapter pour ajouter un listener lors du clic sur une vue (c'est à dire un élément de la liste)) et appeler le callback:

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val item = items[position]
    holder.bind(item)
    holder.itemView.setOnClickListener { callback(item) }
}
```

Finalement, lors de la création de l'adapter, il est possible de définir le callback:

```
NotesAdapter(
    { item ->
        println(item.note.title)
    }
)
```

Evidemment, il faudrait ajouter pas mal d'autre chose afin que l'Interface Utilisateur reste intuitive et agréable. Par exemple, ajouter des animations lorsqu'un élément est cliqué. L'idée globale reste cependant la même.