Laboratoire 9

Les composants d'architecture (Room, ViewModel, LiveData)

Objectifs d'apprentissage

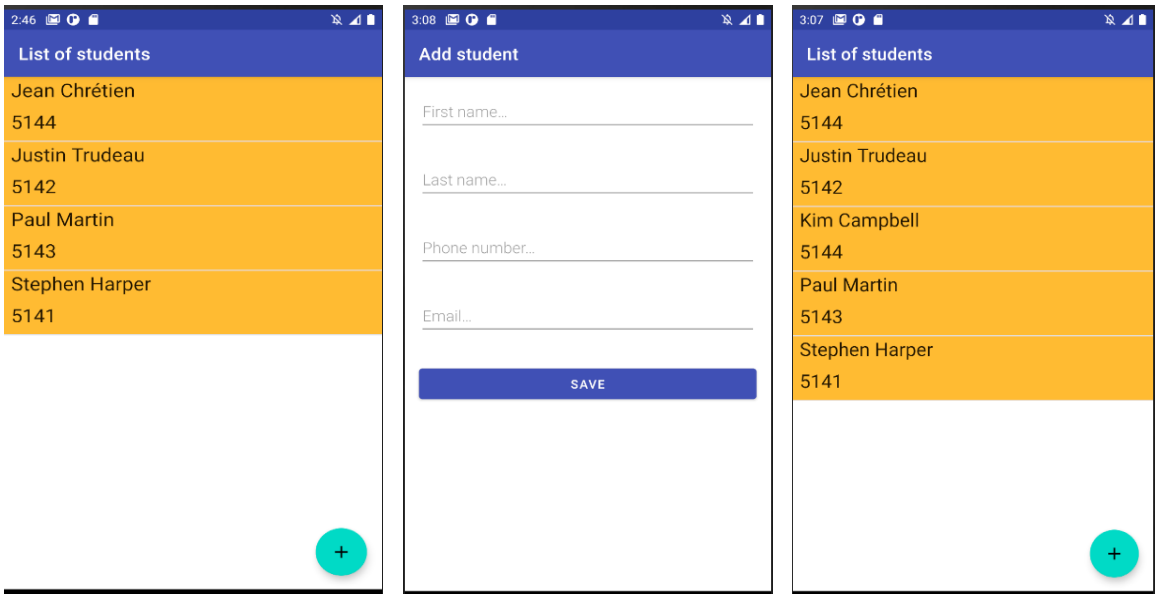
- Comprendre les composants d'architecture d'Android de la librairie Jetpack
- Approfondir l'utilisation des RecyclerView et des adapteurs.
- Utiliser un ORM (Room) pour accéder aux données SQLite
- Comprendre les modèles d'architecture logicielle séparant les données de l'UI.

L'objectif des composants d'architecture est de fournir des conseils sur l'architecture des applications, avec des bibliothèques pour les tâches courantes telles que la gestion du cycle de vie et la persistance des données. Les composants d'architecture vous aident à structurer votre application de manière robuste, testable et maintenable avec moins de code standard. Les bibliothèques de composants d'architecture font partie d'Android Jetpack.

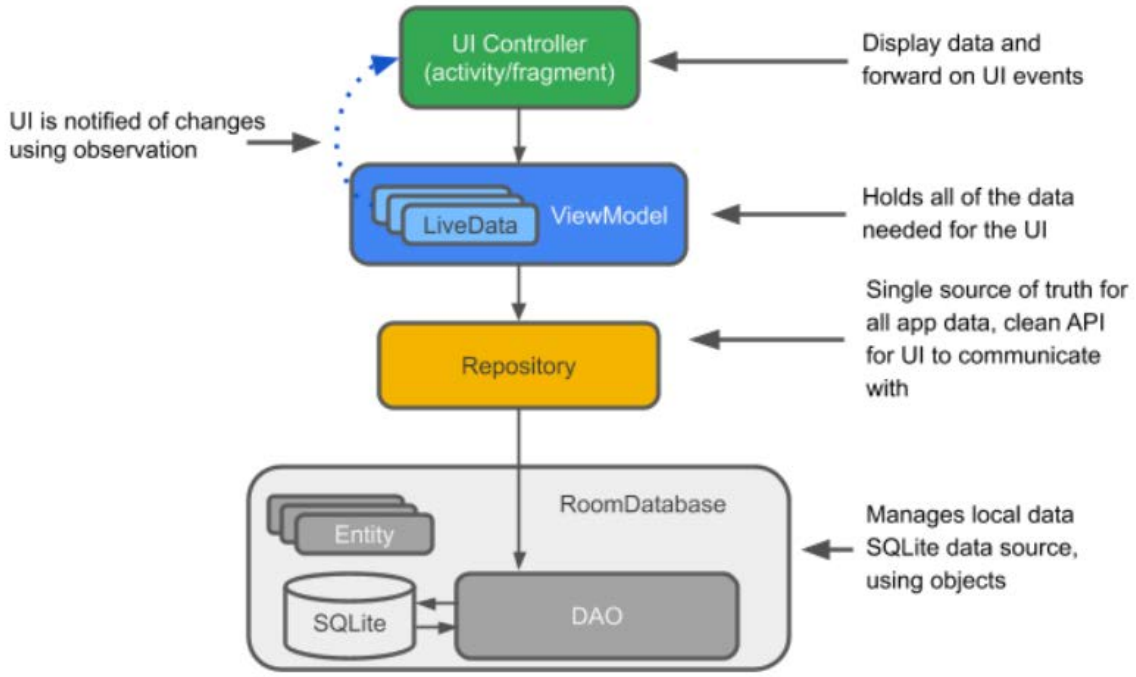
Dans ce laboratoire, vous allez apprendre à concevoir et construire une application en utilisant les composants d'architecture Room, ViewModel, LiveData.
Vous allez

1. Créer une base de données (student-database) SQLite en utilisant Room
2. Rechercher des données pour les afficher dans un recyclerview.
3. Sauvegarder des données dans la base de données.

Voici un aperçu de ce que vous allez réaliser:

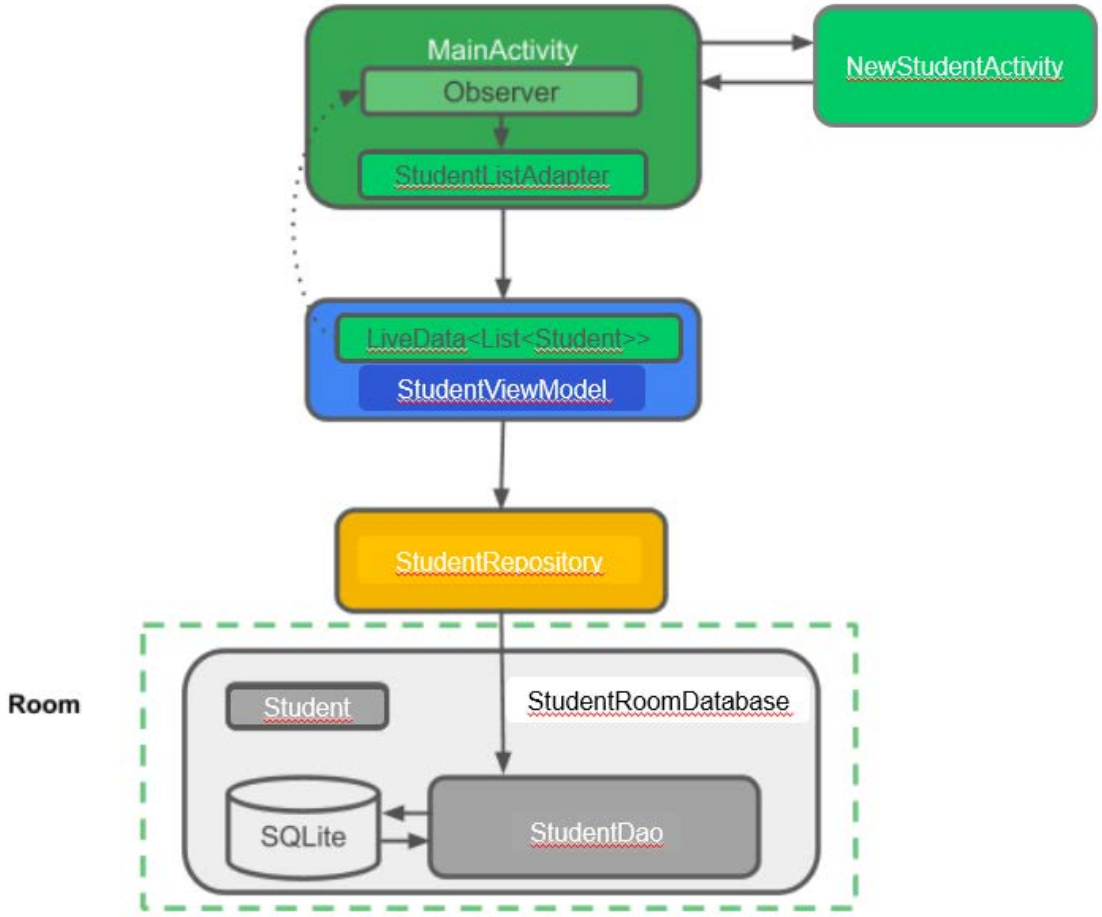


Cette application se rapproche de ce que vous avez réalisé au labo 7, mais l'architecture est vraiment différente et utilise d'autre composants provenant de Android Jetpack.
Il y aura de nombreuses étapes à mettre en œuvre pour réaliser l'architecture recommandée. Le plus important est de créer un modèle mental de ce qui se passe, de comprendre comment les éléments interagissent et comment les données circulent entre eux. Ne vous contentez pas de copier et coller le code. Essayez de bien comprendre ce que vous êtes en train de faire.
Le diagramme suivant montre une forme de base de l'architecture:



On va passer en revue rapidement les différents composants qu'on va utiliser.
Entité: classe annotée décrivant une table de la base de données quand on utilise Room.
SQLite: Bibliothèque de persistance permettant de stocker les données en local, gérant aussi la base de données.
DAO: objet d'accès aux données. Un mappage des requêtes SQL à des fonctions. Quand vous utilisez un DAO, vous appelez la fonction et Room s'occupe de tout.
Base de données Room: Point d'accès à la base de données SQLite. Elle utilise Room pour envoyer des requêtes à SQLite.
Repository: Une classe que utilisée pour gérer plusieurs sources de données.
ViewModel: Intermédiaire entre le repository et l'UI. Ce dernier n'a plus à se soucier de l'origine des données. À noter que les instances d'un ViewModel survivent à la destruction/création d'un fragment ou d'une activité.
LiveData: Une classe encapsulant vos données et qui les met sous observation en utilisant le patron de conception Observateur. Elle contient et met toujours en cache la dernière

version des données et avertit ses observateurs lorsque les données changent. LiveData est conscient du cycle de vie des fragments/activités ou autres. Les composants de l'UI observent seulement les données pertinentes. Ils n'arrêtent ni ne recommencent les observations en cours. LiveData en fait la gestion de façon automatique parce qu'il est au courant des modifications du statut des éléments qui observent les données.
Le diagramme suivant montre toutes les pièces de l'application **de gestion des étudiants avec Room**. Chacune des boîtes englobantes (à l'exception de la base de données SQLite) représente une classe que vous allez créer.



Création de l'application

Créer un nouveau projet **RoomStudent** dans Android Studio en utilisant le template **Empty Activity**.
Vous allez maintenant ajouter les librairies de composants que vous allez utiliser dans le fichier Gradle.
Ouvrez le fichier build.gradle (**Module:app**).
Ajoutez le plugin de processeur d'annotations **kapt** au début du fichier build.gradle (**Module: app**) après les lignes apply plugin.

```
apply plugin: 'kotlin-kapt'
```

Ajoutez les dépendances suivantes dans le fichier build.gradle.

Création de l'entité Student

Vous devez créer une classe Student avec les champs suivants et ensuite ajouter les annotations nécessaires pour que Room puisse en assurer la gestion. Room utilise ces informations pour générer le code nécessaire. Cette classe décrit l'entité representant la table SQLite pour les étudiants. Chacune des propriétés représente une colonne dans la table table-student.

```
@Entity(tableName = "student_table")
data class Student(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name="id")
    val id:Int,
    @ColumnInfo(name = "firstName")
    val firstName: String,
    @ColumnInfo(name = "lastName")
    val lastName: String,
    @ColumnInfo(name = "phoneNumber")
    val phoneNumber: String,
    @ColumnInfo(name = "email")
    val email: String
)
```

L'annotation @Entity(tableName = "table-student) indique que c'est une entité. Vous pouvez spécifier le nom de la table si vous voulez qu'il soit différent du nom de la classe ou le nom de la classe sera utilisé par défaut. Ici le nom de la table (student-table) est différent du nom de la classe (Student).

L'annotation @PrimaryKey vous permet de définir une clé primaire pour votre table. Chaque entité doit obligatoirement avoir une clé primaire. Vous pouvez générer les clés primaires de façon automatique en utilisant le mot clé autoGenerate (`@PrimaryKey(autoGenerate = true)`)

L'annotation @ColumnInfo(name = "id") vous permet de spécifier un nom pour une colonne s'il est différent du nom de la propriété. Sinon le nom de la propriété sera par défaut utiliser comme nom de colonne.

Création de l'objet d'accès aux données (DAO)

Dans le DAO (objet d'accès aux données), vous spécifiez des requêtes SQL et les associez à des appels de méthode. Le compilateur vérifie le SQL et génère des requêtes à partir d'annotations pratiques pour les requêtes courantes, telles que @Insert. Room utilise le DAO pour créer une API propre pour votre code.

Le DAO doit être une interface ou une classe abstraite.

Par défaut, toutes les requêtes doivent être exécutées sur un thread distinct.

Room prend en charge les coroutines, permettant à vos requêtes d'être annotées avec le mot-clé suspend, puis appelées à partir d'une coroutine ou d'une autre fonction de suspension. Le DAO pour les étudiants fournira un ensemble de requêtes CRUD.

Créez une interface StudentDao comme suit:

```
@Dao
interface StudentDao {
    @Query("SELECT * from student_table ORDER BY firstName ASC")
    fun getStudents(): LiveData<List<Student>>

    @Query("SELECT * FROM student_table WHERE id=(:id)")
    fun getStudent(id: Int): LiveData<Student?>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    fun insert(student: Student)

    @Update
    fun updateStudent(student: Student)

    @Query("DELETE FROM student_table")
    fun deleteAll()
}
```

L'annotation Dao définit la classe comme un objet d'accès aux données (DAO) pour Room.

Les annotations Insert, Update, Delete vous évite à écrire du SQL.

L'annotation Query nécessite que vous fournissiez une requête SQL, comme son nom l'indique, en paramètre.

L'annotation `onConflict = OnConflictStrategy.IGNORE` un nouvel étudiant s'il existe déjà dans la base de données.

LiveData est une classe qui vous permet d'observer les données. Vous utilisez une valeur de retour pour les méthodes getStudent et getStudents encapsulées dans LiveData, ce qui permet à Room de mettre à jour automatiquement ces données. Si vous souhaitez mettre à jour des données, vous devez utiliser la classe MutableLiveData (méthodes setValue(T) et postValue(T)). D'habitude, MutableLiveData est utilisé dans le ViewModel qui expose les objets LiveData aux observateurs.

Création de la base de données Room

Room est une couche se trouvant au-dessus de la base de données SQLite, s'occupant des tâches que vous aviez vu dans SQLiteOpenHelper au laboratoire 7. Il utilise le DAO pour émettre des requêtes vers la base de données. Il ne permet pas d'émettre des requêtes sur le thread principal. Quand Room utilise LiveData, les requêtes sont automatiquement exécutées dans un thread en arrière-plan.

Vous devez avoir une seule instance de RoomDatabase dans votre aplication. Votre classe héritant de RoomDatabase doit être abstraite.

Créez la classe singleton StudentRoomDatabase, pour éviter d'ouvrir plusieurs instances de la base de données en même temps, comme suit:

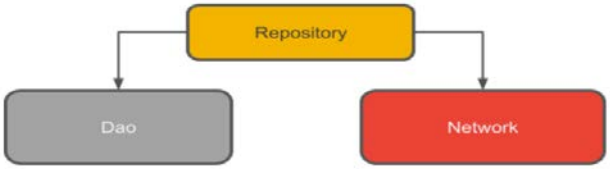
```
@Database(entities = arrayOf(Student::class), version = 1, exportSchema = false)
abstract class StudentRoomDatabase: RoomDatabase() {
    abstract fun studentDao(): StudentDao

    companion object{
        //Singleton to prevent multiple instances of database openint at the same time
        @Volatile
        private var INSTANCE: StudentRoomDatabase? = null

        fun getDatabase(context: Context): StudentRoomDatabase {
            val tempInstance = INSTANCE
            if(tempInstance != null)
            {
                return tempInstance
            }
            INSTANCE = Room.databaseBuilder(
                context.applicationContext,
                StudentRoomDatabase::class.java,
                "student_database"
            ).build()
            return INSTANCE as StudentRoomDatabase
        }
    }
}
```

La fonction getDatabase renvoie le singleton créé. Il créera la base de données "student_database" lors de son premier accès, en utilisant le générateur de base de données de Room pour créer un objet RoomDatabase, utilisant le contexte de l'application applicationContext, à partir de la classe StudentRoomDatabase.

Vous annotez la classe en tant que base de données Room avec `@Database` et utilisez les paramètres d'annotation pour déclarer les entités qui appartiennent à la base de données et définir le numéro de version. Chaque entité correspond à une table qui sera créée dans la base de données. Vous définissez `exportSchema` à false ici pour éviter un avertissement de construction. Dans une vraie application, vous devriez envisager de définir un répertoire que Room utilisera pour exporter le schéma afin de pouvoir vérifier le schéma actuel dans votre système de contrôle de version.

Création du repository

Une classe repository fait abstraction de l'accès à plusieurs sources de données. Le référentiel ne fait pas partie des bibliothèques de composants d'architecture, mais constitue une meilleure pratique suggérée pour la séparation du code et l'architecture. Une classe Repository fournit une API propre pour l'accès aux données au reste de l'application.



Créez la classe StudentRepository comme suit:

```
class StudentRepository(private val studentDao: StudentDao) {
    // Room execute toutes les requêtes dans un thread séparé.
    // Observed LiveData will notify the observer when the data has changed.
    val allStudents: LiveData<List<Student>> = studentDao.getStudents()

    suspend fun insert(student: Student){
        studentDao.insert(student)
    }
}
```

Le mot clé suspend indique au compilateur que la fonction insert doit être appelée à partir d'une coroutine ou d'une autre fonction de suspension.

Création du ViewModel

Le role d'un ViewModel est de fournir des données à l'interface utilisateur et de survivre aux changements de configuration. Il agit comme un relai de communication entre le Repository et l'interface utilisateur. Un ViewModel peut être aussi utiilisé pour partager des données entre des fragments. Il fait partie de la bibliothèque du cycle de vie. Il conserve les données de manière consciente du cycle de vie (des fragments/activités) et leur permet de survivre aux changements de configuration. Séparer les données de l'interface utilisateur de vos classes `Activity` et `Fragment` vous permet de mieux suivre le principe de responsabilité unique: vos activités et vos fragments sont responsables du dessin des données à l'écran, tandis que votre ViewModel peut prendre en charge le stockage et le traitement de toutes les données nécessaires à l'interface utilisateur. Utilisez LiveData dans les ViewModel pour que l'interface utilisateur soit au courant des modifications qui sont apportées.

Créez une classe StudentViewModel comme suit:

```
class StudentViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: StudentRepository
    // - Using LiveData and caching what getStudents returns has several benefits:
    // - We can put an observer on the data (instead of polling for changes) and only
update the UI when the data actually changes.
    // - Repository is completely separated from the UI through the ViewModel.
    val allStudents: LiveData<List<Student>>

    init {
        val studentsDao = StudentRoomDatabase.getDatabase(application,
viewModelScope).studentDao()
        repository = StudentRepository(studentsDao)
        allStudents = repository.allStudents
    }

     // Launching a new coroutine to insert the data in a non-blocking way
    fun insert(student: Student) = viewModelScope.launch(Dispatchers.IO) {
        repository.insert(student)
    }
}
```

La classe StudentViewModel recoit un objet de type Application et hérite de la classe AndroidViewModel. Elle contient une variable de type StudentRepository et une variable de type LiveData pour gérer les étudiants. Dans le bloc init, on se crée un objet DAO utilisé pour créer l'objet repository pou accéder aux données. Une fonction insert permet de faire l'insertion d'un étudiant en utilisant une coroutine dans le scope ViewModel pour ne pas bloquer le thread principial. Les ViewModels ont une portée de coroutine basée sur leur cycle de vie appelée viewModelScope qu'on pourrait aussi utiliser.

En Kotlin, toutes les coroutines s'exécutent dans un CoroutineScope. Un scope contrôle la durée de vie d'une coroutine via son job. Une fois le job gérant la coroutine annulée, la coroutine s'annule également.

Modification de l'interface utilisateur

Modifiez le fichier styles.xml comme suit:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme"
parent="Theme.MaterialComponents.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

    <!-- The default font for RecyclerView items is too small.
    The margin is a simple delimiter between the students. -->
    <style name="student_info">
        <item name="android:layout_width">match_parent</item>
        <item name="android:paddingBottom">8dp</item>
        <item name="android:paddingLeft">8dp</item>
        <item
name="android:background">@android:color/holo_orange_light</item>
        <item
name="android:textAppearance">@android:style/TextAppearance.Large</item>
    </style>
</resources>
```

Créer dans le repertoire layout un fichier recyclerview_item.xml avec le contenu suivant:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/fullNameView"
        style="@style/student_info"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/holo_orange_light" />
```

```
    <TextView
        android:id="@+id/phoneNumberView"
        style="@style/student_info"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/holo_orange_light"/>
</LinearLayout>
```

Modifiez le fichier activity_main.xml comme suit:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerview"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:src="@drawable/ic_baseline_add_24" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Observez le code du bouton flottant ajouté. Ajoutez le fichier vector asset correspondant à l'image dans drawable.

Vous allez afficher les données dans un recycler view. Vous devez créer la classe StudentListAdapter pour l'exploitation des données à afficher.

```
package ca.qc.cgodin.roomstudents

import android.content.Context
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView


class StudentListAdapter constructor(
        context: Context
) : RecyclerView.Adapter<StudentListAdapter.StudentViewHolder>() {
```

```
    private val inflater: LayoutInflater = LayoutInflater.from(context)
    private var students = emptyList<Student>() // Cached copy of students

    inner class StudentViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        val fullNameItemView: TextView =
itemView.findViewById(R.id.fullNameView)
        val phoneNumberItemView: TextView =
itemView.findViewById(R.id.phoneNumberView)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
StudentViewHolder {
        val itemView = inflater.inflate(R.layout.recyclerview_item, parent,
false)
        return StudentViewHolder(itemView)
    }

    override fun onBindViewHolder(holder: StudentViewHolder, position: Int) {
        val current = students[position]
        holder.fullNameItemView.text = "${current.firstName}
${current.lastName}"
        holder.phoneNumberItemView.text = "${current.phoneNumber}"
    }

    fun setStudents(students: List<Student>) {
        this.students = students
        notifyDataSetChanged()
    }

    override fun getItemCount() = students.size
}
```

Exécutez votre application et assurez-vous que tout fonctionne normalement. Vous n'allez pas encore voir des données.
Vous allez ajouter une nouvelle activité pour insérer des étudiants.
Créez une nouvelle activité **NewStudentActivity** en utilisant le template **Empty Activity**.
Modifiez votre fichier strings.xml pour avoir ce qui suit.

```
<resources>
    <string name="app_name">RoomStudentsManagement</string>
    <string name="display_students">List of students</string>
    <string name="action_settings">Settings</string>
    <string name="hint_student">Student…</string>
    <string name="hint_firstname">First name…</string>
    <string name="hint_lastname">Last name…</string>
    <string name="hint_phone_number">Phone number…</string>
    <string name="hint_email">Email…</string>
    <string name="buttonSave">Save</string>
    <string name="empty_not_saved">Student not saved because it is
empty.</string>
```

```
    <string name="add_student">Add student</string>
</resources>
```

Ajoutez la ligne suivante dans votre fichier colors.xml:

```
<color name="buttonLabel">#FFFFFF</color>
```

Ajoutez les lignes suivantes dans un nouveau fichier dimens.xml que vous aurez créé.

```
<resources>
    <dimen name="small_padding">8dp</dimen>
    <dimen name="big_padding">16dp</dimen>
    <dimen name="min_height">48dp</dimen>
</resources>
```

Modifiez votre fichier activity_new_student.xml pour avoir ce qui suit:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText
        android:id="@+id/editFirstNameView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/big_padding"
        android:fontFamily="sans-serif-light"
        android:hint="@string/hint_firstname"
        android:inputType="textAutoComplete"
        android:minHeight="@dimen/min_height"
        android:textSize="18sp" />

    <EditText
        android:id="@+id/editLastNameView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:minHeight="@dimen/min_height"
        android:fontFamily="sans-serif-light"
        android:hint="@string/hint_lastname"
        android:inputType="textAutoComplete"
        android:layout_margin="@dimen/big_padding"
        android:textSize="18sp" />

    <EditText
        android:id="@+id/editPhoneNumberView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/big_padding"
        android:fontFamily="sans-serif-light"
        android:hint="@string/hint_phone_number"
        android:inputType="textAutoComplete"
        android:minHeight="@dimen/min_height"
        android:textSize="18sp" />

    <EditText
        android:id="@+id/editEmailView"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/big_padding"
        android:fontFamily="sans-serif-light"
        android:hint="@string/hint_email"
        android:inputType="textAutoComplete"
        android:minHeight="@dimen/min_height"
        android:textSize="18sp" />

    <Button
        android:id="@+id/button_save"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@color/colorPrimary"
        android:text="@string/button_save"
        android:layout_margin="@dimen/big_padding"
        android:textColor="@color/buttonLabel" />

</LinearLayout>
```

Ajoutez le code suivant vous permettant de définir un listener sur votre bouton Save dans le fichier NewStudentActivity comme suit:

```
buttonSave.setOnClickListener {
        val replyIntent = Intent()
        if (TextUtils.isEmpty(editFirstNameView.text)) {
            setResult(Activity.RESULT_CANCELED, replyIntent)
        } else {
            val firstName = editFirstNameView.text.toString()
            replyIntent.putExtra(EXTRA_FIRSTNAME, firstName)

            val lastName = editLastNameView.text.toString()
            replyIntent.putExtra(EXTRA_LASTNAME, lastName)

            val phoneNumber = editPhoneNumberView.text.toString()
            replyIntent.putExtra(EXTRA_PHONENUMBER, phoneNumber)

            val email = editEmailView.text.toString()
            replyIntent.putExtra(EXTRA_EMAIL, email)

            setResult(Activity.RESULT_OK, replyIntent)
        }
        finish()
    }
}

companion object {
    const val EXTRA_FIRSTNAME = "FIRSTNAME"
    const val EXTRA_LASTNAME = "LASTNAME"
    const val EXTRA_PHONENUMBER = "PHONENUMBER"
    const val EXTRA_EMAIL = "EMAIL"
}
```

La dernière étape consiste à connecter l'interface utilisateur à la base de données en affichant les données saisies par l'utilisateur dans le recyclerview.
Pour afficher le contenu actuel de la base de données, ajoutez un observateur de LiveData dans le ViewModel. A chaque fois qu'il y a modificaton de données, la fonction onChanged() sera appelée. Dans cette fonction, on fait appel à la fonction setStudents définie dans la classe StudentListAdapter.
Dans la classe MainActivity, créez une variable studentViewModel comme suit:

```
private lateinit var studentViewModel: StudentViewModel
```

Utilisez ViewModelProvider pour associer votre ViewModel à votre activite.
Lors du premier démarrage de votre activité, le ViewModelProviders créera le fichier ViewModel. A la destruction de l'activité, via un changement de configuration par exemple, le ViewModel va persister. Quand l'activité sera créé à nouveau, le ViewModelProviders va permettre d'utiliser le ViewModel existant à nouveau.

Ajoutez le code suivant à la fin de onCreate de la classe MainActivity.

```
studentViewModel = ViewModelProvider(this).get(StudentViewModel::class.java)
```

Ajoutez ensuite un observateur pour la propriété LiveData allStudents de votre ViewModel comme suit:

```
studentViewModel.allStudents.observe(this, Observer { students ->
    // Update the cached copy of the students in the adapter.
    students?.let { adapter.setStudents(it) }
})
```

Vous devez ajouter une variable newStudentActivityRequestCode au début de la classe MainActivity comme suit:

```
private val newStudentActivityRequestCode = 1
```

Dans MainActivity, ajoutez la fonction onActivityResult() qui sera appelée à la création d'un nouvel étudiant. Quand l'activité revient avec RESULT_OK, insérez l'étudiant dans la base de données en appelant la fonction insert() de StudentViewModel.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, intentData:
Intent?) {
    super.onActivityResult(requestCode, resultCode, intentData)

    if (requestCode == newStudentActivityRequestCode && resultCode ==
Activity.RESULT_OK) {
        intentData?.let { data ->
            val student = Student(
                    0,
                    data.getStringExtra(NewStudentActivity.EXTRA_FIRSTNAME),
                    data.getStringExtra(NewStudentActivity.EXTRA_LASTNAME),
                    data.getStringExtra(NewStudentActivity.EXTRA_PHONENUMBER),
                    data.getStringExtra(NewStudentActivity.EXTRA_EMAIL)
            )
            Log.i("TAG", student.toString())
            studentViewModel.insert(student)
            Unit
        }
    } else {
        Toast.makeText(
```

```
                applicationContext,
                R.string.empty_not_saved,
                Toast.LENGTH_LONG
        ).show()
    }
}
```

Ajoutez le code suivant à la fin de la fonction onCreate de MainActivity pour pouvoir lancer l'activité NewStudentActivity quand on clique sur le bouton FloatingActionButton.

```
fab.setOnClickListener {
    val intent = Intent(this@MainActivity, NewStudentActivity::class.java)
    startActivityForResult(intent, newStudentActivityRequestCode)
}
```

Lancez votre application. Insérez des étudiants et vérifiez que vous pouvez les voir dans votre recyclerview.

Lorsque vous ajoutez un étudiant à la base de données dans NewStudentActivity, l'interface utilisateur se met à jour automatiquement.

Travail à faire

Modifiez l'application en permettant de faire des mises à jour d'un étudiant.

Modifiez l'application en permettant de supprimer un étudiant.

Créer une interface utilisateur aussi permettant de retrouver un étudiant à partir de son nom.

A remettre le 20 octobre 2020

Le programme zippé.

Bon travail!