SOUTENANCE PROJET 9

Devenez un as de la gestion immobilière

OPENCLASSROOMS

Lien de l’application finie sur GitHub :

Lamine MESSACI

13/09/2020

1. Présentation

- L’application doit permettre à un agent immobilier de pouvoir créer un nouveau bien depuis l’application, en précisant tout ou une partie des informations demandées.

- Une fois l’ajout d’un bien correctement effectué, un message de notification doit apparaitre sur le téléphone de l’utilisateur afin de lui indiquer que tout s’est bien passé.

- La géo-localisation d'un bien est automatiquement effectuée à partir de son adresse, afin d'afficher la vignette de carte correspondante dans le détail du bien.

- Les biens existants peuvent être édités pour mettre à jour leurs informations (ajout, modification, suppression).

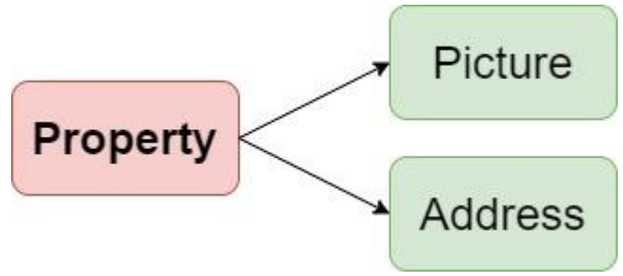
- Il n’est pas possible de supprimer un bien, en revanche il est possible de préciser qu’un bien a été vendu, en précisant obligatoirement sa date de vente.

- L’application doit être écrite en **Java** ou **Kotlin** et supporte Android à partir de la version **4.4** (**API 19 KitKat**)

2. Classe Modèle

L'application **RealEstateManager** s'appuie sur trois Objets principaux.
J'ai donc créé ces trois « data class model » :

```kotlin
@Parcelize
@Entity
@SuppressWarnings(RoomWarnings.PRIMARY_KEY_FROM_EMBEDDED_IS_DROPPED)
data class Property(
    @PrimaryKey(autoGenerate = true)
    var id: Long,
    var type: String,
    var description: String?,
    var price: Double?,
    var livingSpace: Int?,
    var rooms: Int? = 0,
    var shops: Boolean?,
    var trainStation: Boolean?,
    var park: Boolean?,
    var airport: Boolean?,
    var subway: Boolean?,
    var school: Boolean?,
    var status: Boolean?,
    var dateOfEntry: String?,
    var dateOfSale: String?,
    var realtor: String?,
    var numOfBath: Int?,
    var numOfBed: Int?,
    @Embedded
    var address: Address?,

    @TypeConverters
    var pictures: List<Picture>? = arrayListOf()
) : Parcelable {
    constructor() : this(
        id: 0, type: "", description: "", price: 0.0, livingSpace: 0,
        rooms: 0, shops: false, trainStation: false, park: false, airport: false, subway: false, school: false, status: true, dateOfEntry: "", dateOfSale: "",
        realtor: "", numOfBath: 0, numOfBed: 0, address: null, emptyList()
    )

    fun fromContentValues(values: ContentValues): Property {...}
}
```



3. ROOM

ROOM est une librairie fournissant des outils pour créer, requêter et manipuler plus facilement des bases de données SQLite.

Une fois les entités sont définies grâce aux annotations @Entity, il serait intéressant de pouvoir les **manipuler** à travers notre base de données SQLite.

Réaliser les différentes actions CRUD ! et c'est grâce au **Design Pattern DAO**

```kotlin
@Dao
interface PropertyDao {


    @Query( value: "SELECT * from property")
    fun getAllProperties(): LiveData<List<Property>>

    @Query( value: "SELECT * from property WHERE id = :propertyId")
    fun getProperty(propertyId: Long): LiveData<Property>

    @RawQuery(observedEntities = [Property::class])
    fun getPropertyByArgs(query: SupportSQLiteQuery): LiveData<List<Property>>

    @Insert(onConflict = REPLACE)
    fun insertProperty(property: Property): Long

    @Update
    fun updateProperty(property: Property): Int

    @Query( value: "DELETE FROM property WHERE id = :index")
    fun deleteProperty(index: Long)

    //// ---- FOR CONTENT PROVIDER ---- ////

    @Query( value: "SELECT * FROM property WHERE id = :idProperty")
    fun getPropertyWithCursor(idProperty: Long): Cursor
}
```

RealEstateDatabase : rôle sera de **lier** toutes les classes/interfaces que nous avons précédemment créées ensemble, et surtout de **configurer** La base de données !

A l'intérieur de cette classe, j'ai déclaré l'**interfaces de DAO**. Ainsi qu'un companion object pour créer un **singleton** de la base de données

```kotlin
// RealEstateManager Database configuration
@Database(
    entities = [(Property::class), (Picture::class), (Address::class)],
    version = 2,
    exportSchema = false
)
@TypeConverters(Converters::class)
abstract class RealEstateDatabase : RoomDatabase() {

    abstract fun propertyDao(): PropertyDao

    companion object {
        private var INSTANCE: RealEstateDatabase? = null

        fun getInstance(context: Context): RealEstateDatabase {
            if (INSTANCE == null) {
                synchronized( lock: this) {
                    INSTANCE = Room.databaseBuilder(
                        context.applicationContext,
                        RealEstateDatabase::class.java,
                        name: "RealEstateManager.db"
                    ).build()
                }
            }
            return INSTANCE as RealEstateDatabase
        }
    }
}
```

4. API Google

A. Maps

Pour utiliser la carte, l'application intègre le SDK *Maps Android*.

```
implementation 'com.google.android.gms:play-services-maps:17.0.0'
```

Une fois la clé générée sur le site de *Google Cloud Plateform*, le fragment affiche la carte grâce au *SupportMapFragment.*

```
<fragment
    xmlns:map="http://schemas.android.com/apk/res-auto"
    android:id="@+id/fragment_map"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    map:layout_constraintBottom_toBottomOf="parent"
    map:layout_constraintTop_toTopOf="parent"
    tools:context=".controllers.activities.MainActivity" />
```

Enfin, pour pouvoir interagir avec celle-ci, le fragment implémente *OnMapReady()*. Et bien sûr en vérifiant Les permissions de localisation.

```
override fun onMapReady(googleMap: GoogleMap?) {
    if (googleMap != null) {
        map = googleMap
        if (activity?.let { it: FragmentActivity
                ActivityCompat.checkSelfPermission(
                    it,
                    Manifest.permission.ACCESS_FINE_LOCATION
                )
            } != PackageManager.PERMISSION_GRANTED && activity?.let { it: FragmentActivity
                ActivityCompat.checkSelfPermission(
                    it,
                    Manifest.permission.ACCESS_COARSE_LOCATION
                )
            } != PackageManager.PERMISSION_GRANTED
        ) {

            return
        }

            map.setMyLocationEnabled(true)
    }
}
```

4. API Google

B.Geocode

Afin d'avoir les coordonnées(location) à partir d'une Adresse, L'application s'appuie sur l'API *Google geocode For Android*.

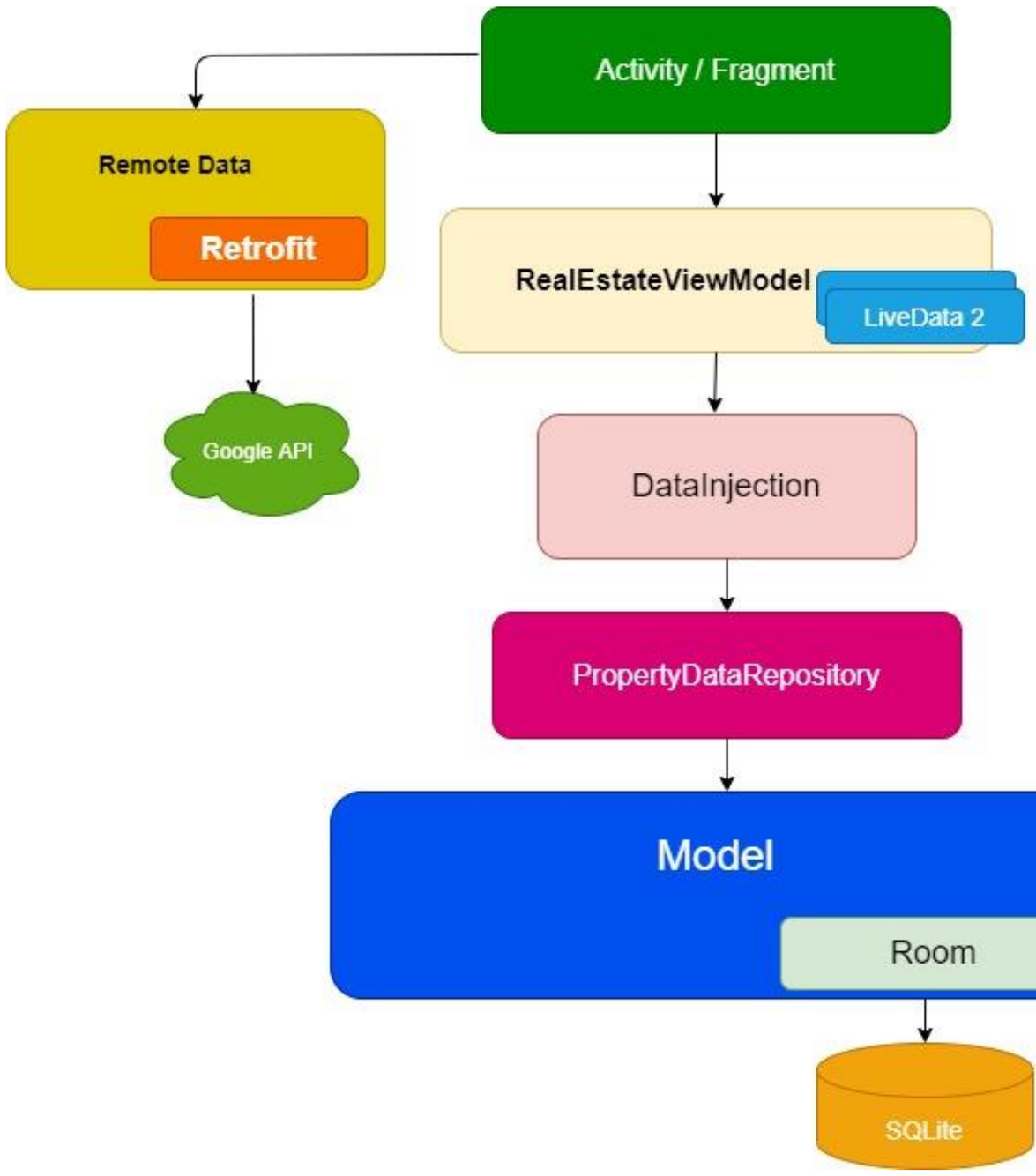
Pour réaliser les appels réseau, ***Retrofit*** couplé à ***RxJava*** sont utilisés.

```
*/

class RealEstateStream {

    // Stream function
    fun streamFetchGeocodeInfo(address: String, key: String): Observable<GeocodeInfo> =
        RealEstateService.create().getGeocodeInfo(address, key)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .timeout( timeout: 10, TimeUnit.SECONDS)
}
```

```
interface RealEstateService {

    // Http service configuration
    @GET( value: "json")
    fun getGeocodeInfo(
        @Query( value: "address") address: String,
        @Query( value: "key") key: String
    ): Observable<GeocodeInfo>

    companion object {
        fun create(): RealEstateService {
            val retrofit : Retrofit = Retrofit.Builder()
                .baseUrl( baseUrl: "https://maps.google.com/maps/api/geocode/")
                .addConverterFactory(GsonConverterFactory.create())
                .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
                .client(
                    OkHttpClient.Builder().addInterceptor(
                        HttpLoggingInterceptor().setLevel(
                            HttpLoggingInterceptor.Level.BASIC
                        )
                    ).build()
                )
                .build()

            return retrofit.create(RealEstateService::class.java)
        }
    }
}
```

Les informations renvoyées par ***geocode*** arrivent au format json. Elles sont retranscrites (*générées via le site http://www.jsonschema2pojo.org/*).

5.MVVM


Activity / Fragment
Remote Data
Retrofit
RealEstateViewModel
LiveData 2
Google API
DataInjection
PropertyDataRepository
Model
Room
SQLite

6.CreateEditActivity

1. Vérification des inputs de l'utilisateur (Adresse)
2. Récupération des flux via **Retrofit** en passant en paramètres l'adresse et key
3. Si tout va bien on sauvegarde le premier résultat a la base de données
4. **getLocationFromAddress**() renvoie une location a partir d'une adresse.

Elle utilise l'objet **Geocoder** d'android et sa méthode **getFromLocationName()**

```kotlin
// To save geolocation in database
private fun storeLocationToDatabase() {
    progressBar_create.visibility = View.VISIBLE
    val addressStr : String = checkClass.checksAddressElements(address, city, postalCode)
    if (addressStr.isNotEmpty()) {
        val realEstateStream = RealEstateStream()
        disposable =
            realEstateStream.streamFetchGeocodeInfo(addressStr, BuildConfig.GoogleSecAPIKEY)
                .subscribeWith(object : DisposableObserver<GeocodeInfo?>() {
                    override fun onNext(t: GeocodeInfo) {
                        geoLocation = t
                    }

                    override fun onError(e: Throwable) {
                        showAlertDialog()
                        progressBar_create.visibility = View.GONE
                    }

                    override fun onComplete() {
                        setValuesInProperty()
                    }
                })
    }
}
```

```kotlin
// To set values in object
private fun setValuesInProperty() {
    progressBar_create.visibility = View.GONE
    val latLng: LatLng = this?.getLocationFromAddress(address.toString())!!
    lat = latLng.latitude
    lng = latLng.longitude

    property = checkClass.setValuesInProperty(
        lat, lng, airport, school, subway, shops, trainStation, park, additionalAddress,
        pictures, address, description, entryDate, apartNumber, sold, soldDate, property
    )
    propertyViewModel.createProperty(property)
    showNotification()
    returnToHome()
}
```

7. MapsFragment

Le ***MapsFragment*** permet d'afficher la carte configurée précédemment (voir page 5) et d'y placer des ***markers*** symbolisant les **Properties** environnants.

Chaque *marker* contient un *tag* avec son ***PropertyId***, grâce à celui-ci et à la méthode *()*, un double clic sur un *marker* lance ***DetailEstateFragment***.

```
// Markers listener
override fun onMarkerClick(p0: Marker?): Boolean {
    p0?.let { getPropertyId(it) }
    return true
}
```

```
// To display items on map
private fun displayList(properties: List<Property>) {
    geoCodeList = arrayListOf()
    propertiesList = properties
    for (p: Property in properties) {
        if (p.address?.lat != 0.0 && p.address?.lng != 0.0) {
            addMarker(p, p.address?.lat!!, p.address?.lng!!)
        } else {
            if (activity?.let { Utils.isInternetAvailable(it) }!!) {
                executeRequestToGetAddresses(p)
            } else {
                Toast.makeText(
                    activity,
                    text: "Sorry but internet isn't available. We can't get addresses for properties.",
                    Toast.LENGTH_LONG
                ).show()
            }
        }
    }
}
```

```
// To request location if not exist in database
private fun executeRequestToGetAddresses(property: Property) {
    val addressStr : String =
        property.address?.address + "+" + property.address?.city + property.address?.postalCode
    Log.e( tag: "test address", addressStr)
    val realEstateStream = RealEstateStream()
    disposable =
        realEstateStream.streamFetchGeocodeInfo(addressStr, BuildConfig.GoogleSecAPIKEY)
            .subscribeWith(object : DisposableObserver<GeocodeInfo?>() {
                override fun onNext(location: GeocodeInfo) {
                    geoLocation = location
                }

                override fun onError(e: Throwable) {
                }

                override fun onComplete() {
                    geoCodeList.add(geoLocation)
                    val lat: Double = geoLocation.results?.get(0)?.geometry?.location?.lat!!
                    val lng: Double = geoLocation.results?.get(0)?.geometry?.location?.lng!!
                    addMarker(property, lat, lng)
                    storeLocation(property, lat, lng)
                }
            })
}
```

8.SearchFragment

Après avoir entrer les critères recherchées le clique sur bouton de recherche déclenche la fonction **makeSearchQuery(),**

Cette dernière utilse la classe SearchUtils() de recherche pour demander des propriétés dans la base de données à l'aide de la methode **makeQuery()** qui retourn a son tour une **query** prés configurer par rapport critères entrées par l'utilisateur et initialisé le **ViewModel**

En passant cette **query** en paramètres

```
Log.e( tag: "***test args: ", query)
propertyViewModel.getPropertyByArgs(query).observe( owner: this, androidx.lifecycle.Observer { it: List<Property>!
    if (it!!.isNotEmpty()) {
        getResult(it)
```

ViewModel

```
//// --- SEARCH --- ////
fun getPropertyByArgs(queryString: String): LiveData<List<Property>> {
    val query = SimpleSQLiteQuery(queryString)
    Log.e( tag: "get properties by args", msg: "Query : ${query.sql}")
    return mPropertyDataRepository.getPropertyByArgs(query)
}
```

PropertyDataRepository

```
fun getPropertyByArgs(query: SimpleSQLiteQuery): LiveData<List<Property>> {
    return propertyDao.getPropertyByArgs(query)
}
```

PropertyDao

```
@RawQuery(observedEntities = [Property::class])
fun getPropertyByArgs(query: SupportSQLiteQuery): LiveData<List<Property>>
```

9.Capture d'écran