Lab 8 - Android Architecture Components

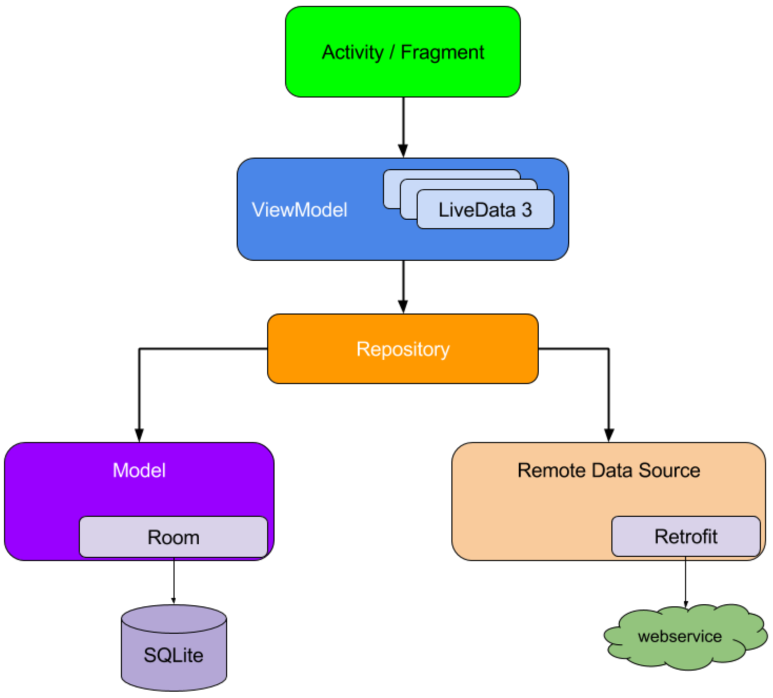
[based on Android Studio 3.3. Development Essentials by Neil Smyth]

Android Architecture Components are a new set of classes and libraries designed to make certain common tasks in Android programming more efficient and safer (i.e. fewer crashes). While so far we used to put pretty much all our programme's logic in Activity and Fragment classes, with the new classes we can achieve a higher level of *the separation of concerns*. Thus, different sections of our programme (different classes) will handle completely different roles. This is very important in mobile programming, as Activities and Fragments can be destroyed any time - you do not have any control over their lifecycle, the way a user interacts with your app and the way OS manages the memory decide the life of your Activites and Fragments. Thus, we want to minimize our dependency on Activities and Fragments and use other classes for, say, controlling the data that a user sees on the screen.

We have the following new classes for this purpose:

- **ViewModel** - to provide data for UI elements, without being aware of their existence
- **LiveData** - to allow data values to be observable by other components
- **Repository** - technically, not a new Android framework class, but a one that you create and that is used to provide access to the data, irrespective of where the data physically resides
- **Room** - an ORM database

The figure below displays the relationship among the mentioned classes (as well as remote data fetching using Retrofit, but we will not deal with it in this lab).

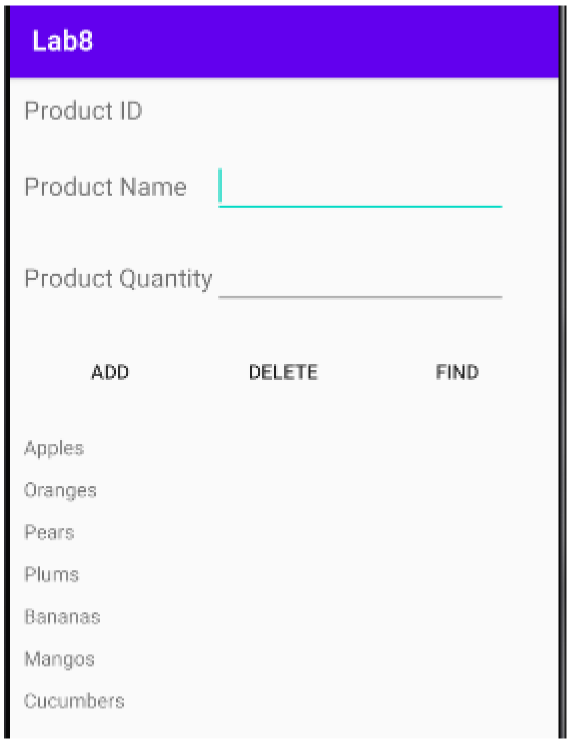


Task overview

We will develop an inventory application that lets a user store items in the inventory, find them there, and delete them, if needed. The inventory is persisted in a Room database on the device.

To shortcut the development process, download code from https://bitbucket.org/pbdfrita/pbd2023-lab-8

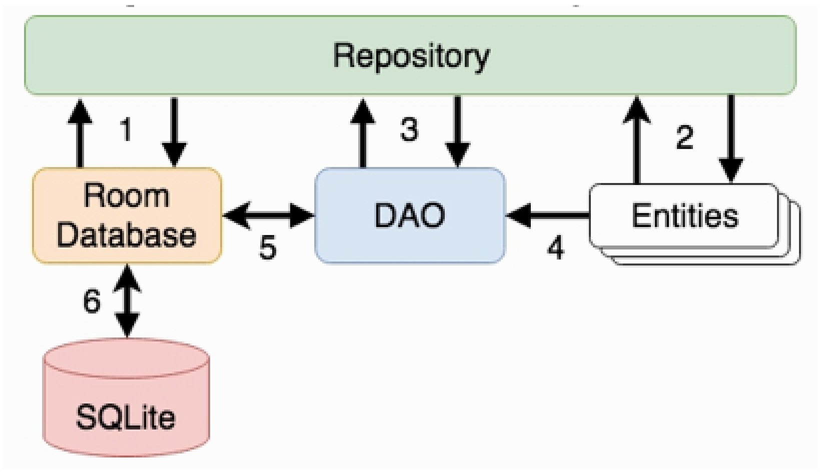
Our app will have a single Activity/Fragment where a user can enter the product, its quantity, and store it in the inventory. The interface also allows the user to search for a product by name or to delete a product. The list of products is shown at the bottom and is updated immediately after any changes in the inventory. The data should persist even when the application is killed.



The Database

The Room database allows us to store the data without writing tedious SQL queries. The access is provided through Database Access Objects (DAOs), which represent an interface between Kotlin objects and the database. The “shape” of the database, the tables, are provided by Entities. In the figure below you see the usual way of interaction with a Room database [from Android Studio 3.3. Development Essentials by Neil Smyth]:

1. The repository interacts with the Room Database to get a database instance which, in turn, is used to obtain references to DAO instances.
2. The repository creates entity instances and configures them with data before passing them to the DAO for use in search and insertion operations.
3. The repository calls methods on the DAO passing through entities to be inserted into the database and receives entity instances back in response to search queries.
4. When a DAO has results to return to the repository it packages those results into entity objects.
5. The DAO interacts with the Room Database to initiate database operations and handle results.
6. The Room Database handles all of the low level interactions with the underlying SQLite database, submitting queries and receiving results.



Entity

We need to create Entities (data tables) for our project. We are storing only one type of information - products - thus we need to create a single entity. Create a new Kotlin class called **Product** in package `si.uni_lj.fri.pbd.lab8`. The class should contain the following fields:

- `var id = 0`
- `var name: String? = null`
- `var quantity = 0`

The class is not yet an Entity class, to become that, we need to add `@Entity(tableName = "products")` just above the "`class Product`" line. Now, this class can be used to create a new table in a Room database and that table will be named "`products`".

The underlying SQLite database is a relational database, which means we must at the minimum define the primary key. You can do that with more annotations: `@PrimaryKey(autoGenerate = true)` above the declaration of the field you would like to use for the primary key.

Finally, annotation `@ColumnInfo(name = "SOMETHING")` above the declaration of the field you want to use in your queries tells SQLite that the field should be stored under the `SOMETHING` column name in the database. Put this annotation above each of the three fields and name these fields' columns "`productId`", "`productName`" and "`productQuantity`" respectively.

Data Access Object

The next step is creating the Data Access Object (DAO). Create a new public interface called **ProductDao**. Adding `@Dao` annotation above the "`interface ProductDao`" line tells Android that this class will be used as a DAO. You now need to create a specific access methods:

- For inserting a Product in the database add:
  ```
  @Insert

  fun insertProduct(product: Product)
  ```
- For finding a product with a specific name in a database:
  ```
  @Query("SELECT * FROM products WHERE productName = :name")

  fun findProduct(name: String): List<Product>
  ```
- For deleting a product with a specific name from the database:
  ```
  @Query("DELETE FROM products WHERE productName = :name")

  fun deleteProduct(name: String)
  ```
- For returning a LiveData object with a list of all products in the database:
  ```
  @get:Query("SELECT * FROM products")

  val allProducts: LiveData<List<Product>>
  ```

Spend some time trying to understand how the annotations and the function declarations tell Android what exactly to do when these functions are called.

Room Database

A Room Database is handled through an abstract class that extends **RoomDatabase**. Create a new abstract class **ProductRoomDatabase** that extends **RoomDatabase**. Putting `@Database(entities = [Product::class], version = 1)` above the "`abstract class ProductRoomDatabase`" tells Android that this class will define the database. Note how it tells that **Product.class** defines the (only) entity/table we have in the database.

We need to add a static reference to this database. "Static" because you want to make sure you have only one such database. Add the following as a companion object:

```
private var INSTANCE: ProductRoomDatabase? = null
```

The field is private, so we need to create an access method. Add:

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

`getDatabase` returns a reference to the database instance and it makes sure that the instance is created if it is not already. Note "`synchronized`" - this ensures that the code is not executed in parallel via multiple calls, which may have catastrophic consequences for the database.

We also need to provide our DAO for modifying the database. All you need to do is to add one more abstract function:

```
abstract fun productDao(): ProductDao
```

Finally, the code for accessing the database. Doing this on the main thread is not the best practice. To access it in the background, you can use the **Executors**. Add the following lines to the **ProductRoomDatabase** class (as companion objects):

```
private const val NUMBER_OF_THREADS = 4

val databaseWriteExecutor = Executors.newFixedThreadPool(NUMBER_OF_THREADS)
```

We will call database queries on this executor.

Repository

The repository will be responsible for interacting with the database on behalf of the ViewModel. The class **ProductRepository** is partly provided for you. However, you need to add a reference to a DAO you will use to access the database:

```
private val productDao: ProductDao?
```

Further, when we instantiate the repository we should also instantiate the database. Put the following code in an init block:

```
val db: ProductRoomDatabase? = application?.let {
ProductRoomDatabase.getDatabase(it.applicationContext) }

productDao = db?.productDao()

allProducts = productDao?.allProducts
```

The functions to insert, delete, and find a product need to be completed. Note, the queries on the database need to be run in the background using the executor. In `insertProduct` function add:

```
ProductRoomDatabase.databaseWriteExecutor.execute(Runnable {
   productDao?.insertProduct(newproduct)
})
```

Completing `deleteProduct` should be straightforward.

Completing `findProduct` is a bit different - you run a query, but it might take some time to find the result and it should then report it back to the `searchResults` object (of type **MutableLiveData**), something like this should go in `run()` method of the **Runnable**:

```
searchResults.postValue(productDao?.findProduct(name))
```

The User Interface

We have already created the **MainActivity**, **MainFragment**, **ProductListAdapter**, and the layouts for you.

MainFragment has three fields (see the screenshot above) corresponding to the three views: `productId`, `productName`, and `productQuantity`.

- If the views corresponding to `productName` and `productQuantity` are set by the user, who then clicks on “`Add`” button, a new **Product** instance should be created and added to the database;
- If the view corresponding to `productName` is set by the user who then clicks on “`Delete`” button, the product with the given name is deleted from the database;
- If the view corresponding to `productName` is set by the user who then clicks on “`Find`” button, the product details (ID, name, quantity) are shown, if the product with this name exists in the database;

`main_fragment.xml` also contains a **RecyclerView**. This **RecyclerView** should show a list of all the inventory. Check `recyclerSetup()` function in **MainFragment** to see how the **RecyclerView** is tied to a **ProductListAdapter**.

ViewModel and LiveData

The main novelty we will encounter here is the **ViewModel**. The **ViewModel** will hold the data so that even when the **Fragment** is stopped or killed the data stays in the memory and can be quickly retrieved back. Open **MainViewModel**. It should hold two kinds of data (you should add these as private fields):

- `var allProducts: LiveData<List<Product>>?` - a list of all Products that will be shown in the RecyclerView

- `var searchResults: MutableLiveData<List<Product>>` - details about the product that we search for (NOTE: technically, this is a list of Products because an SQL select statement returns a list, but we will handle only the first element of this list)

These fields are of type **LiveData**, which allows them to notify an observer (which will reside in **Fragment**) that there is a change in data. Thus, the UI does not need to be refreshed all the time - any changes will be reported through **LiveData** immediately! Note that the `searchResults` field is **MutableLiveData**. This allows it to be modified outside the **MainViewModel** class. This is needed as the `findProduct` function in **ProductRepository** is run on a separate thread and reports the results when they are ready (`postValue`), rather than returning them immediately.

We usually prefer that the **ViewModel** does not interact with the database directly, but through a repository. Create `private val repository: ProductRepository` in **MainViewModel**.

Now, in the **MainViewModel** init block set the repository to a `ProductRepository` (give application as an argument), set `allProducts` to `repository.allProducts` and `searchResults` to `repository.searchResults`.

ViewModel should also use the repository to find, insert, and delete the data in the database. Create the following functions as well:

```
fun insertProduct(product: Product) {

   repository.insertProduct(product)

}

fun findProduct(name: String) {

   repository.findProduct(name)

}

fun deleteProduct(name: String) {

   repository.deleteProduct(name)

}
```

Setting the UI and LiveData Observers

We have put a ViewModel in our Fragment (the field has already been put for you) and we set interaction through button click events. Open **MainFragment** and follow TODO instructions in the `listenerSetup` function.

Finally, we set observers to monitor data changes. First, set an observer for `mViewModel.allProducts`. When this LiveData object changes, the adapter should set the product list to the new data. E.g.

```
mViewModel?.allProducts?.observe(viewLifecycleOwner) { products ->

   adapter?.setProductList(products)

}
```

Second, we set an observer for `mViewModel.searchResults`. When this result changes, we should populate the upper part of the UI (TextViews and EditTexts) with the found product data. Your code should look something like this:

```
mViewModel?.searchResults?.observe(viewLifecycleOwner) { products ->

    if (products.isNotEmpty()) {

        productId?.text = String.format(Locale.US, "%d", products[0].id)

        productName?.setText(products[0].name)

        productQuantity?.setText(String.format(Locale.US, "%d", products[0].quantity))

    } else {

        productId?.text = "No Match"

    }

}
```

That’s the whole lab! Run it and check whether adding, deleting, and searching for a product works as expected.

If you did not attend this lab assignment in-person at FRI you can commit your solution to a private repository named **PBD2023-LAB-8** in your Bitbucket account. User **pbdfrita (pbdfrita@gmail.com)** must be added as a read only member of this repository. The code must be committed by Sunday (May 7th) 23:59.

Happy coding!