Android Applications Development

10 – ViewModel and Room

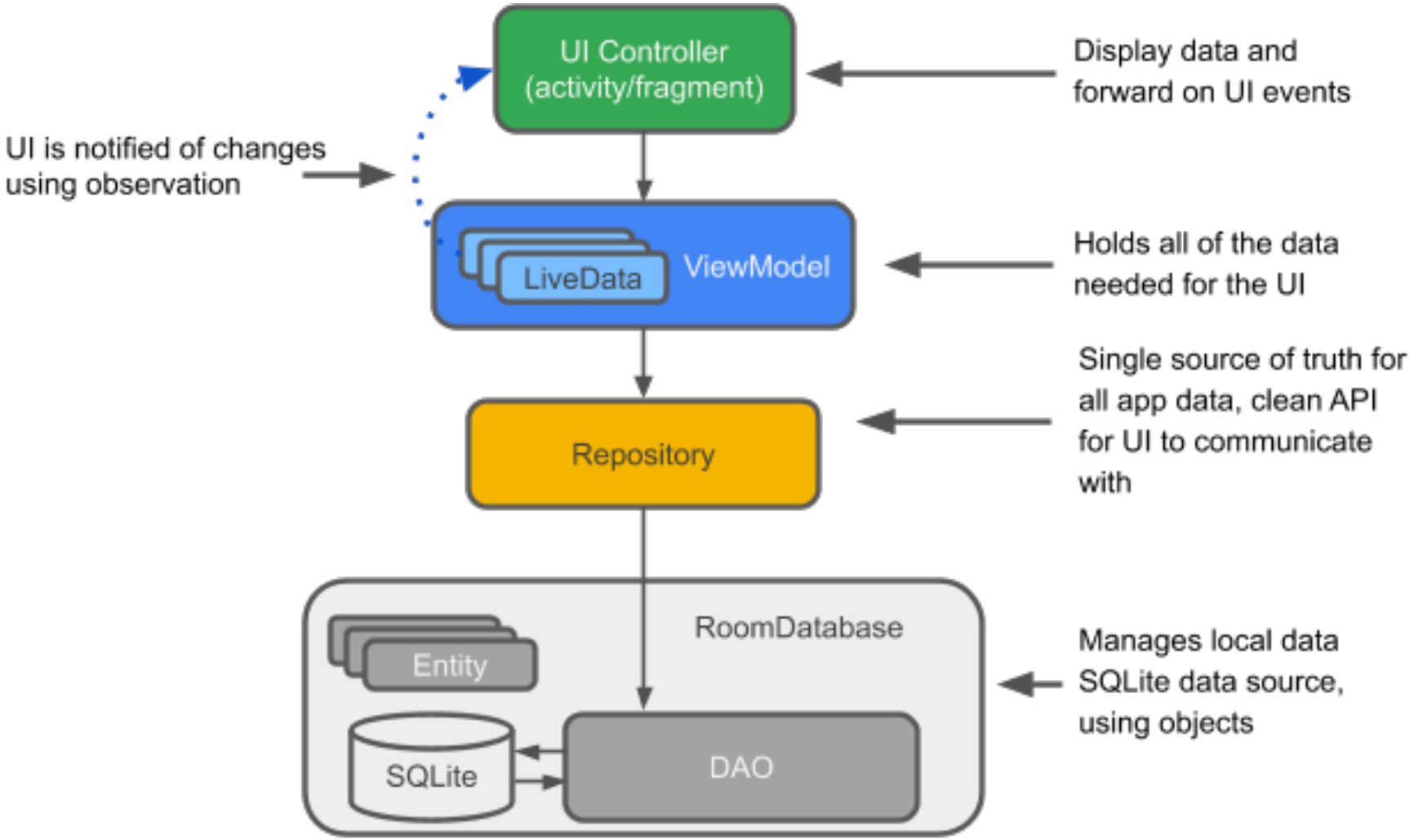
Objectives

- Introduction to View Model and Room

Credits

- The following slides are extracted from :
  - https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0

The big picture



The big picture

- Entity: Annotated class that describes a database table when working with Room.
- SQLite database: On device storage. The Room persistence library creates and maintains this database for you.
- DAO: Data access object. A mapping of SQL queries to functions. When you use a DAO, you call the methods, and Room takes care of the rest.
- Room database: Simplifies database work and serves as an access point to the underlying SQLite database (hides SQLiteOpenHelper). The Room database uses the DAO to issue queries to the SQLite database.
- Repository: A class that you create that is primarily used to manage multiple data sources.
- ViewModel: Acts as a communication center between the Repository (data) and the UI. The UI no longer needs to worry about the origin of the data. ViewModel instances survive Activity/Fragment recreation.
- LiveData: A data holder class that can be observed. Always holds/caches the latest version of data, and notifies its observers when data has changed. LiveData is lifecycle aware. UI components just observe relevant data and don't stop or resume observation. LiveData automatically manages all of this since it's aware of the relevant lifecycle status changes while observing.

ROOM

Room

- Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally.
- Common use is to cache relevant pieces of data so that when the device cannot access the network, the user can browse that content while they are offline.

Room

- Room persistence library provides an abstraction layer over existing SQLite
- Room provides
  - Compile time verification of SQL queries
  - Convenience annotations that minimize repetitive and error-prone boilerplate code
  - Streamlined database migration paths
- Recommend to use Room instead of using SQLite APIs directly.

Entities

```
@Entity
data class User(
    @PrimaryKey var id: Int,
    var firstName: String?,
    var lastName: String?
)
```

- Uses @Entity annotation to state table names
- Each entity must define 1 field as primary key.
- Primary key is annoted with the @PrimaryKey.
- Variable name will be used as column name by default

Entities

```
@Entity(primaryKeys = arrayOf("firstName", "lastName"))
data class User(
    val firstName: String?,
    val lastName: String?
)
```

- If entities have composite primary key, you can declare it as shown above

```
@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    val lastName: String?,
    @Ignore val picture: Bitmap?
)
```

- By default, all column for each field will be created. If you don’t wish, you can annotate them using @Ignore

Entities

```
@Entity(tableName = "contacts_table")
data class Contacts(@PrimaryKey(autoGenerate = true) @ColumnInfo(name = "id") val id : Int,

                    @ColumnInfo(name="name") val name: String,

                    @ColumnInfo(name="num") val num : String)
```

- Above example states the following:
  - Table name
  - Primary key is “id”. It is autogenerated by Room
  - Each column name is stated using the @ColumnInfo

Data access object (DAO)

- When you use the Room persistence library to store your app's data, you interact with the stored data by defining data access objects, or DAOs.
- Each DAO includes methods that offer abstract access to your app's database.
- At compile time, Room automatically generates implementations of the DAOs that you define.

DAO Anatomy

```
@Dao
interface UserDao {
    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)

    @Query("SELECT * FROM user")
    fun getAll(): List<User>
}
```

- DAO does not have any implementation. Mainly just definitions of methods for interacting with the data in your app's database.

- There are two types of DAO methods that define database interactions:
  - Convenience methods that let you insert, update, and delete rows in your database without writing any SQL code.
  - Query methods that let you write your own SQL query to interact with the database.

Insert

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUsers(vararg users: User)

    @Insert
    fun insertBothUsers(user1: User, user2: User)

    @Insert
    fun insertUsersAndFriends(user: User, friends: List<User>)
}
```

- The @Insert annotation allows you to define methods that insert their parameters into the appropriate table in the database.
- Each parameter for an @Insert method must be either an instance of a Room data entity class annotated with @Entity or a collection of data entity class instances.
- When an @Insert method is called, Room inserts each passed entity instance into the corresponding database table.

Update

```
@Dao
interface UserDao {
    @Update
    fun updateUsers(vararg users: User)
}
```

- The @Update annotation allows you to define methods that update specific rows in a database table.
- Similarly to @Insert methods, @Update methods accept data entity instances as parameters.

Delete

```
@Dao
interface UserDao {
    @Delete
    fun deleteUsers(vararg users: User)
}
```

- The @Delete annotation allows you to define methods that delete specific rows from a database table.
- Similarly to @Insert methods, @Delete methods accept data entity instances as parameters.

Query

```
@Query("SELECT * FROM user")
fun loadAllUsers(): Array<User>
```

- The @Query annotation allows you to write SQL statements and expose them as DAO methods.
- Use these query methods to query data from your app's database, or when you need to perform more complex inserts, updates, and deletes.

Query

```
@Query("SELECT * FROM user WHERE age > :minAge")
fun loadAllUsersOlderThan(minAge: Int): Array<User>
```

```
@Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int): Array<User>

@Query("SELECT * FROM user WHERE first_name LIKE :search " +
       "OR last_name LIKE :search")
fun findUserWithName(search: String): List<User>
```

- Most of the time, your DAO methods need to accept parameters so that they can perform filtering operations.
- Room supports using method parameters as bind parameters in your queries.

Query

```
data class NameTuple(
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

```
@Query("SELECT first_name, last_name FROM user")
fun loadFullName(): List<NameTuple>
```

- Most of the time, you only need to return a subset of the columns from the table that you are querying.
- Room allows you to return a simple object from any of your queries as long as you can map the set of result columns onto the returned object.

Example

```
@Dao
interface ContactsDao{


    @Query( value: "Select * from contacts_table")

    fun retrieveAllContacts() : List<Contacts>

    @Insert(onConflict = OnConflictStrategy.ABORT)
    fun insert(newContacts : Contacts)


    @Delete
    fun delete(delContact : Contacts)

}
```

- Example states the following:
  - Dao interface
  - Query from contacts table.
  - Insertion of Contacts and abort if record exists
  - Deleting from contacts table

Database observation

- When data changes, you usually want to take some action, such as displaying the updated data in the UI. This means you have to observe the data so when it changes, you can react.
- To observe data changes we will use Flow from kotlinx-coroutines.
- Use a return value of type Flow in your method description, and Room generates all necessary code to update the Flow when the database is updated.

```
@Query( value: "Select * from contacts_table")
fun retrieveAllContacts() : Flow<List<Contacts>>
```

Database

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

- The class must be annotated with a @Database annotation that includes an entities array that lists all of the data entities associated with the database.
- The class must be an abstract class that extends RoomDatabase.
- For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.

Database

- After you have defined the data entity, the DAO, and the database object, you can use the following code to create an instance of the database:

```
val db = Room.databaseBuilder(
            applicationContext,
            AppDatabase::class.java, "database-name"
        ).build()
```

- If your app runs in a single process, you should follow the singleton design pattern when instantiating an AppDatabase object.
- Each RoomDatabase instance is fairly expensive, and you rarely need access to multiple instances within a single process

Example

```
@Database(entities = arrayOf(Contacts::class), version = 1, exportSchema = false)
abstract class ContactsRoomDatabase : RoomDatabase() {

    abstract fun contactsDao(): ContactsDao

    companion object {

        @Volatile
        private var INSTANCE: ContactsRoomDatabase? = null

        fun getDatabase(context: Context,scope : CoroutineScope): ContactsRoomDatabase? {

            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context,
                    ContactsRoomDatabase::class.java,
                     name: "contacts_database"
                ).build()
                INSTANCE = instance

                instance  ^synchronized
            }
        }
    }
}
```

getDatabase returns the singleton. It'll create the database the first time it's accessed, using Room's database builder to create a RoomDatabase object

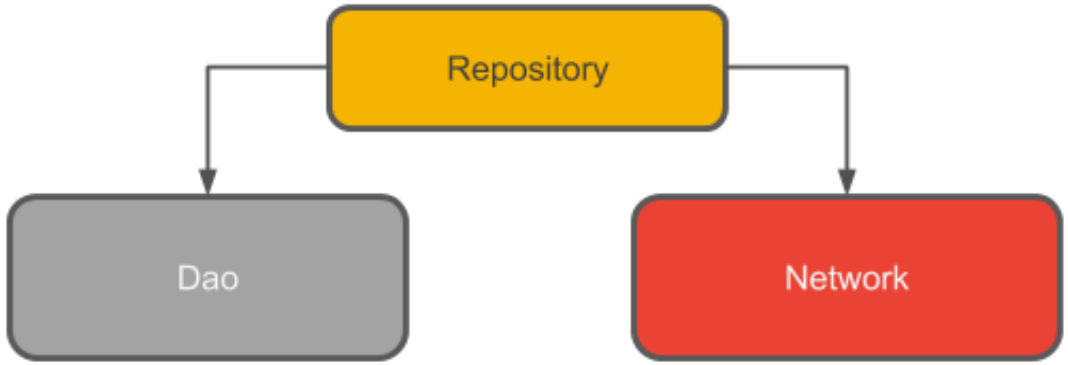
Example

```
@Database(entities = arrayOf(Contacts::class), version = 1, exportSchema = false)
abstract class ContactsRoomDatabase : RoomDatabase() {

    abstract fun contactsDao(): ContactsDao

    companion object {

        @Volatile
        private var INSTANCE: ContactsRoomDatabase? = null

        fun getDatabase(context: Context,scope : CoroutineScope): ContactsRoomDatabase? {

            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context,
                    ContactsRoomDatabase::class.java,
                     name: "contacts_database"
                ).build()
                INSTANCE = instance

                instance  ^synchronized
            }
        }
    }
}
```

getDatabase returns the singleton. It'll create the database the first time it's accessed, using Room's database builder to create a RoomDatabase object

Repository

- A repository class abstracts access to multiple data sources.
- The repository is not part of the Architecture Components libraries, but is a suggested best practice for code separation and architecture.
- A Repository class provides a clean API for data access to the rest of the application.



Why use a Repository?

A Repository manages queries and allows you to use multiple backends. In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in a local database.

Repository

- The DAO is passed into the repository constructor as opposed to the whole database.
- This is because it only needs access to the DAO, since the DAO contains all the read/write methods for the database.

```
class ContactsRepository(private val contactsDao : ContactsDao){

    val allContacts = contactsDao.retrieveAllContacts()

    suspend fun insert(contact : Contacts){
        contactsDao.insert(contact)
    }

    suspend fun delete(contact : Contacts){
        contactsDao.delete(contact)
    }
}
```

VIEW MODEL

Overview

- The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way.
- The ViewModel class allows data to survive configuration changes such as screen rotations.
- The Android framework manages the lifecycles of UI controllers, such as activities and fragments. The framework may decide to destroy or re-create a UI controller in response to certain user actions or device events that are completely out of your control.
- If the system destroys or re-creates a UI controller, any transient UI-related data you store in them is lost.
- Another problem is that UI controllers frequently need to make asynchronous calls that may take some time to return

Overview

- UI controllers such as activities and fragments are primarily intended to display UI data, react to user actions, or handle operating system communication, such as permission requests. Requiring UI controllers to also be responsible for loading data from a database or network adds bloat to the class. Assigning excessive responsibility to UI controllers can result in a single class that tries to handle all of an app's work by itself, instead of delegating work to other classes. Assigning excessive responsibility to the UI controllers in this way also makes testing a lot harder.
- It's easier and more efficient to separate out view data ownership from UI controller logic.

Implementation

- Architecture Components provides ViewModel helper class for the UI controller that is responsible for preparing data for the UI.
- ViewModel objects are automatically retained during configuration changes so that data they hold is immediately available to the next activity or fragment instance.
- For example, if you need to display a list of users in your app, make sure to assign responsibility to acquire and keep the list of users to a ViewModel, instead of an activity or fragment

Implementation

- Architecture Components provides ViewModel helper class for the UI controller that is responsible for preparing data for the UI.
- ViewModel objects are automatically retained during configuration changes so that data they hold is immediately available to the next activity or fragment instance.

Implementation

```
class MyViewModel : ViewModel() {
    private val users: MutableLiveData<List<User>> by lazy {
        MutableLiveData().also {
            loadUsers()
        }
    }

    fun getUsers(): LiveData<List<User>> {
        return users
    }

    private fun loadUsers() {
        // Do an asynchronous operation to fetch users.
    }
}
```

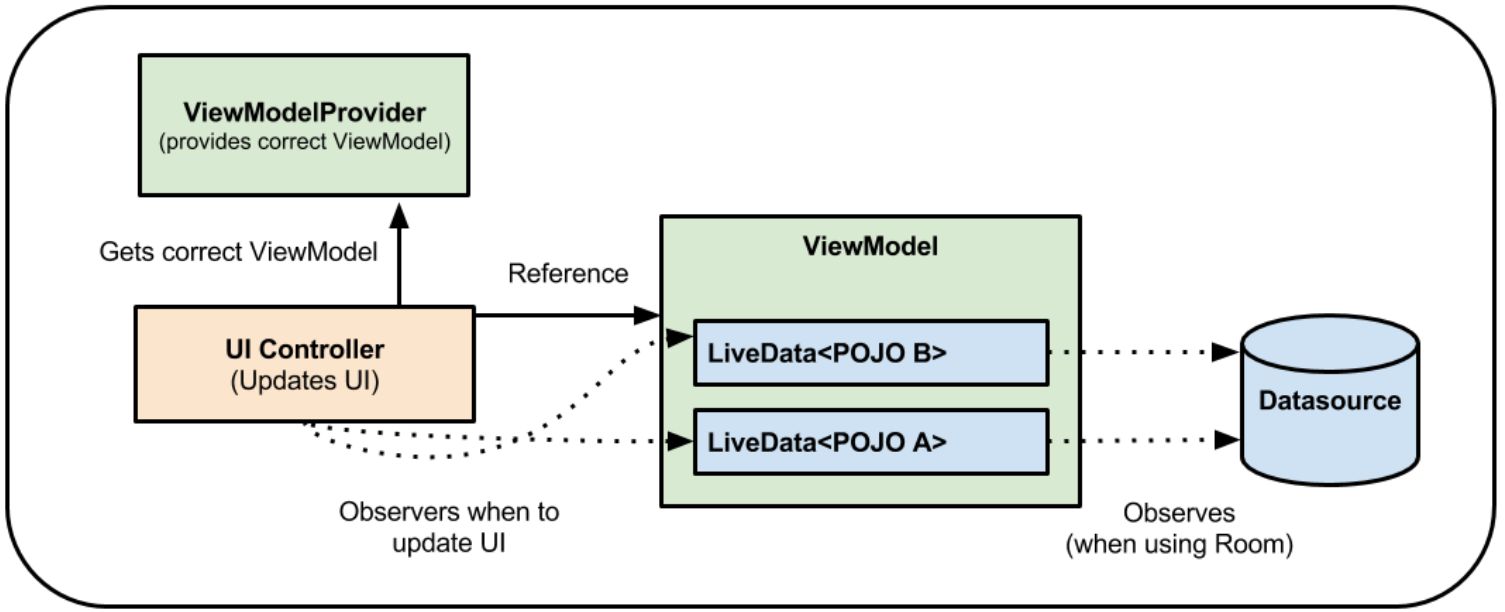
For example, if you need to display a list of users in your app, make sure to assign responsibility to acquire and keep the list of users to a ViewModel, instead of an activity or fragment

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        // Create a ViewModel the first time the system calls an activity's onCreate() method.
        // Re-created activities receive the same MyViewModel instance created by the first activity.

        // Use the 'by viewModels()' Kotlin property delegate
        // from the activity-ktx artifact
        val model: MyViewModel by viewModels()
        model.getUsers().observe(this, Observer<List<User>>{ users ->
            // update UI
        })
    }
}
```

If the activity is re-created, it receives the same MyViewModel instance that was created by the first activity.

ViewModel and Room

- ViewModel works with Room and LiveData.
  The ViewModel ensures that the data survives a device configuration change. Room informs your LiveData when the database changes, and the LiveData, in turn, updates your UI with the revised data.



ViewModelScope

- ViewModel includes support for Kotlin coroutine.
- A ViewModelScope is defined for each ViewModel in your app. Any coroutine launched in this scope is automatically cancelled if the ViewModel is cleared.
- Coroutines are useful here for when you have work that needs to be done only if the ViewModel is active.
- You can access the CoroutineScope of a ViewModel through the viewModelScope property of the ViewModel,

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch {
            // Coroutine that will be canceled when the ViewModel is cleared.
        }
    }
}
```

Example

```
class ContactsViewModel(val repo: ContactsRepository) : ViewModel() {

    val allContacts: LiveData<List<Contacts>> = repo.allContacts.asLiveData()

    fun insert(contactItem: Contacts) = viewModelScope.launch(Dispatchers.IO) {  this: CoroutineScope

        repo.insert(contactItem)

    }

    fun remove(contactItem: Contacts) = viewModelScope.launch(Dispatchers.IO) {  this: CoroutineScope

        repo.delete(contactItem)
    }

}
```

- ContactsViewModel takes in repository as argument
- ContactsViewModel uses Live data from repository
- Insert and remove is done using viewModelScope
- Repository functions are used to perform insertion and deletion.

Instantiate repository and database

- We want to only have one instance of the database and of the repository in our application.
- An easy way to achieve this is by creating them as members of the Application class. Like this, whenever they're needed, they will just be retrieved from the Application, rather than constructed every time.

```
class MyContacts() : Application(){

    val appScope = CoroutineScope(SupervisorJob())
    val db by lazy {ContactsRoomDatabase.getDatabase( context: this,appScope)}
    val repo by lazy{ContactsRepository(db!!.contactsDao())}


}
```

- We created a database instance using RoomDatabase
- We created a repository instance, based on the database DAO.

Connect with data In the Activity

- To display the current contents of the database, add an observer that observes the LiveData in the ViewModel

```
class MainActivity : AppCompatActivity() {


    private val contactsViewModel: ContactsViewModel by viewModels() {

        ContactsViewModelFactory((application as MyContacts).repo)
    }
```

```
contactsViewModel.allContacts.observe( owner: this, Observer {  it: List<Contacts>!

    var contactsConvertList = mutableListOf<String>()

    allContacts = it
    for(contact in it){

        contactsConvertList.add("${contact.name} - ${contact.num}")

    }

    it?.let{  it: List<Contacts>

        contactsAdapter = ArrayAdapter( context: this,
            android.R.layout.simple_list_item_1, contactsConvertList)

        contactsLV.adapter = contactsAdapter

    }

    toggleVisibility()

})
```

Whenever the data changes, the onChanged() callback is invoked, which calls the adapter's setWords() method to update the adapter's cached data and refresh the displayed list.

Making use of ViewModel

```
var contact = allContacts!!.get(info.position)
contactsViewModel.remove(contact)
```

```
var name = data!!.getStringExtra( name: "name")

var num = data!!.getStringExtra( name: "num")

contactsViewModel.insert(Contacts( id: 0,name,num))
```

- Make use of ViewModel to manipulate with the data within your database.