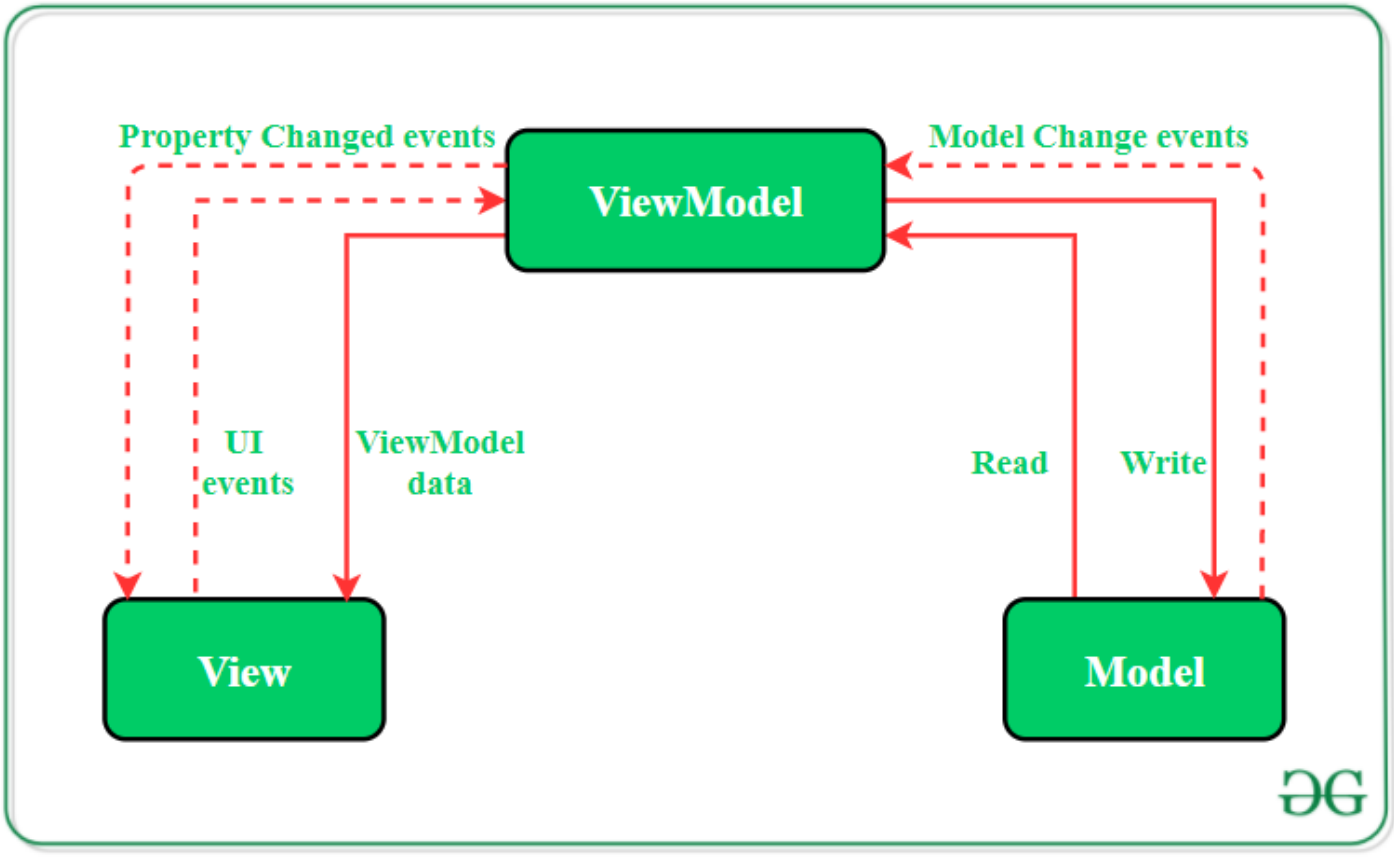
ANDROID INTERVIEW QUESTION PART 4

1) What is MVVM?

Model — View — ViewModel (MVVM) is the industry-recognized software architecture pattern that overcomes all drawbacks of MVP and MVC design patterns. MVVM suggests separating the data presentation logic(Views or UI) from the core business logic part of the application.

- **Model:** This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.
- **View:** The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.
- **ViewModel:** It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.



MVVM pattern has some similarities with the MVP(Model — View — Presenter) design pattern as ViewModel plays the Presenter role. However, the drawbacks of the MVP pattern have been solved by MVVM in the following ways:

1. ViewModel does not hold any kind of reference to the View.
2. Many to-1 relationships exist between View and ViewModel.
3. No triggering methods to update the View.

Advantages of MVVM Architecture

- Enhance the reusability of code.
- All modules are independent which improves the testability of each layer.
- Makes project files maintainable and easy to make changes.

Disadvantages of MVVM Architecture

- This design pattern is not ideal for small projects.
- If the data binding logic is too complex, the application debug will be a little harder.

2) Difference Between MVC, MVP and MVVM Architecture Patterns in Android?

MVC(MODEL VIEW CONTROLLER)	MVP(MODEL VIEW PRESENTER)	MVVM(MODEL VIEW VIEWMODEL)
One of the oldest software architecture	Developed as the second iteration of software architecture which is advance from MVC.	Industry-recognized architecture pattern for applications.
UI(**View**) and data-access mechanism(**Model**) are tightly coupled.	It resolves the problem of having a dependent **View** by using **Presenter** as a communication channel between **Model** and **View**.	This architecture pattern is more event-driven as it uses data binding and thus makes easy separation of **core business logic** from the **View**.
Controller and **View** exist with the one-to-many relationship. One Controller can select a different View based upon required operation.	The one-to-one relationship exists between **Presenter** and **View** as one Presenter class manages one View at a time.	Multiple **View** can be mapped with a single **ViewModel** and thus, the one-to-many relationship exists between View and ViewModel.
The **View** has no knowledge about the **Controller**.	The **View** has references to the **Presenter**.	The **View** has references to the **ViewModel**
Difficult to make changes and modify the app features as the code layers are tightly coupled.	Code layers are loosely coupled and thus it is easy to carry out modifications/changes in the application code.	Easy to make changes in the application. However, if **data binding logic** is too complex, it will be a little harder to debug the application.
User Inputs are handled by the **Controller**.	The **View** is the entry point to the Application	The **View** takes the input from the user and acts as the entry point of the application.
Ideal for small scale projects only.	Ideal for simple and complex applications.	Not ideal for small scale projects.
Limited support to **Unit testing**.	Easy to carry out **Unit testing** but a tight bond of View and Presenter can make it slightly difficult.	**Unit testability** is highest in this architecture.
This architecture has a high dependency on Android APIs.	It has a low dependency on the Android APIs.	Has low or no dependency on the Android APIs.
It does not follow the modular and single responsibility principle.	Follows modular and single responsibility principle.	Follows modular and single responsibility principle.

MVC, MVP, MVVM

3) What is Design Patterns in Android?

A design pattern is a repeatable solution to a software engineering problem. Unlike most program-specific solutions, design patterns are used in many programs. Design patterns are not considered finished products; rather, they are templates that can be applied to multiple situations and can be improved over time, making a very robust software engineering tool.

Design patterns is basically a solution or blueprint for a problem that we get over and over again in programming, so they are just typical types of problems we can encounter as programmers, and these design patterns are just a good way to solve those problems, there is a lot of design pattern in android. So basically, they are three categories as below:

1. **Creational patterns**
2. **Structural patterns**
3. **Behavioral patterns**

1. Creational Patterns

These are the design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation. These are the design patterns are come under this category.

- Singleton
- Builder
- Dependency Injection
- Factory

Singleton:

- The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance. There are very few applications, do not overuse this pattern.
- It's very easy to accomplish in Kotlin but not in java. because in java there is no native implementation of the singleton.

- By using the ***object***, you'll know you're using the same instance of that class throughout your app.

```
object eSingleton {
  fun doing() {
    // ...
  }
}
```

Builder:

- The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.

Factory:

- As the name suggests, Factory takes care of all the object creational logic. In this pattern, a factory class controls which object to instantiate. Factory pattern comes in handy when dealing with many common objects. You can use it where you might not want to specify a concrete class.

Dependency Injection:

- In software terms, dependency injection has you provide any required objects to instantiate a new object. This new object doesn't need to construct or customize the objects themselves.
- In Android, you might find you need to access the same complex objects from various points in your app, such as a network client, image loader, or SharedPreferences for local storage. You can inject these objects into your activities and fragments and access them right away.
- Currently, there are three main libraries for dependency injection: Dagger '2', Dagger Hilt, and Koin.

2. Structural Patterns

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object patterns define ways to compose objects to obtain new functionality.

- Facade
- Adapter
- Decorator

- Composite
- Protection Proxy

Facade:

- The facade pattern is used to define a simplified interface to a more complex subsystem.

Adapter:

- The adapter pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

Decorator:

- The decorator pattern is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behavior.

Composite:

- The composite pattern is used to compose zero-or-more similar objects so that they can be manipulated as one object.

Protection Proxy:

- The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. Protection proxy is restricting access.

3. Behavioral Patterns

- Command
- Observer
- Strategy
- State

- Visitor
- Mediator
- Memento
- Chain of Responsibility

Command:

- The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.

Observer:

- The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.

Strategy:

- The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.

State:

- The state pattern is used to alter the behavior of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.

Visitor:

- The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

Mediator:

- The mediator design pattern is used to provide a centralized communication medium between different objects in a system. This pattern is very helpful in an enterprise application where multiple objects are interacting with each other.

Memento:

- The memento pattern is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

Chain of Responsibility:

- The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler.

4) what is Android Dependency Injection?

The term **"Dependency Injection"** or in short **"DI"** is independent of Android. It is a design pattern in Software Engineering.
The term is explaining itself as Inject Dependency.

To understand it in more detail, let's break the term Dependency Injection. In two parts Dependency and Injection.

What is Dependency?

Consider the following Java class.

```
public class User{

 private Database mDatabase;

 public User(){
 mDatabase = new Database();
 }
}
```

In the above class, we are creating the object of class **Database** inside the constructor of class **User**. So here, we can say that; the User is dependent on Database. So here, **Database** is a dependency for the class **User**.

What is Injection?

As we seen the **User** is Dependent on **Database**, now think how User can get its dependency, which is the Database.

5) What is Dagger 2 ?

Dagger 2 is a compile-time android dependency injection framework that uses **Java Specification Request 330** and Annotations. Some of the basic annotations that are used in dagger 2 are:

1. **@Module** This annotation is used over the class which is used to construct objects and provide the dependencies.
2. **@Provides** This is used over the method in the module class that will return the object.
3. **@Inject** This is used over the fields, constructor, or method and indicate that dependencies are requested.
4. **@Component** This is used over a component interface which acts as a bridge between @Module and @Inject. (Module class doesn't provide dependency directly to requesting class, it uses component interface)
5. **@Singleton** This is used to indicate only a single instance of dependency object is created.

6) What is Dagger-Hilt ?

Dagger Hilt provides a smooth dependency injection way in android.
It reduces various steps in comparison with Dagger2 and providing the better features so definitely we should learn Dagger-hilt to use this advanced cutting edge tool in dependency Injection.

@HiltAndroidApp : Annotation which is required to keep top of the Android Application class
@AndroidEntryPoint : This annotation is required to keep top of the Activity class.
@Inject:This annotation is required to inject field or constructor like repository, services etc.
@HiltViewModel : This is class level annotation that need to keep top of the view model class.
@Module : This is annotation that is require to keep top of the module object class.
@InstallIn : This annotation is require to set up the module with component like Singleton Component.
@Provides: This annotation require to return feature of each module method.
@Binds: This annotation require to return feature of each module method but here module class must be abstract.

7) What is Room Database?

Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

Room is now considered as a better approach for data persistence than SQLiteDatabase. It makes it easier to work with SQLiteDatabase objects in your app, decreasing the amount of **boilerplate** code and verifying SQL queries at compile time.

Why use Room?

- Compile-time verification of SQL queries. each @Query and @Entity is checked at the compile time, that preserves your app from crash issues at runtime and not only it checks the only syntax, but also missing tables.
- Boilerplate code
- Easily integrated with other Architecture components (like LiveData)

Major problems with SQLite usage are

- There is no compile-time verification of raw SQL queries. For example, if you write a SQL query with a wrong column name that does not exist in real database then it will give exception during run time and you can not capture this issue during compile time.
- As your schema changes, you need to update the affected SQL queries manually. This process can be time-consuming and error-prone.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects (POJO).

Room vs SQLite

Room is an ORM, Object Relational Mapping library. In other words, Room will map our database objects to Java objects. Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

Difference between **SQLite** and **Room** persistence library:-

- In the case of SQLite, There is no compile-time verification of raw SQLite queries. But in Room, there is SQL validation at compile time.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects. But, Room maps our database objects to Java Object without boilerplate code.
- As your schema changes, you need to update the affected SQL queries manually. Room solves this problem.
- Room is built to work with LiveData and RxJava for data observation, while SQLite does not.

Room has three main components of Room DB :

- Entity
- Dao

- Database

1. Entity

Represents a table within the database. Room creates a table for each class that has `@Entity` annotation, the fields in the class correspond to columns in the table. Therefore, the entity classes tend to be small model classes that don't contain any logic.

Entities annotations

Before we get started with modeling our entities, we need to know some useful annotations and their attributes.

@Entity — every model class with this annotation will have a mapping table in DB

- foreignKeys — names of foreign keys
- indices — list of indicates on the table
- primaryKeys — names of entity primary keys
- tableName

@PrimaryKey — as its name indicates, this annotation points the primary key of the entity. autoGenerate — if set to true, then SQLite will be generating a unique id for the column

```
@PrimaryKey(autoGenerate = true)
```

@ColumnInfo — allows specifying custom information about column

```
@ColumnInfo(name = "column_name")
```

@Ignore — field will not be persisted by Room

@Embeded — nested fields can be referenced directly in the SQL queries.

2. Dao

DAOs are responsible for defining the methods that access the database. In the initial SQLite, we use the `Cursor` objects. With Room, we don't need all the `Cursor` related code and can simply define our queries using annotations in the `Dao` class.

3. Database

Contains the database holder and serves as the main access point for the underlying connection to your app's persisted, relational data.

To create a database we need to define an abstract class that extends `RoomDatabase`. This class is annotated with `@Database`, lists the entities contained in the database, and the DAOs which access them.

The class that's annotated with `@Database` should satisfy the following conditions:

- Be an abstract class that extends `RoomDatabase`.
- Include the list of entities associated with the database within the annotation.
- Contain an abstract method that has 0 arguments and returns the class that is annotated with Dao

8) What is RxJava?

RxJava is a JVM library for doing asynchronous and executing event-based programs by using observable sequences. It's main building blocks are triple O's, Operator, Observer, and Observables. And using them we perform asynchronous tasks in our project. It makes multithreading very easy in our project. It helps us to decide on which thread we want to run the task.

9) What is RxAndroid?

RxAndroid is an extension of RxJava for Android which is used only in Android application.

RxAndroid introduced the **Main Thread** required for Android.

To work with the multithreading in Android, we will need the **Looper and Handler** for Main Thread execution.

RxAndroid provides **AndroidSchedulers.mainThread()** which returns a scheduler and that helps in performing the task on the main UI thread that is mainly used in the Android project. So,

here **AndroidSchedulers.mainThread()** is used to provide us access to the main thread of the application to perform actions like updating the UI.

In Android, updating UI from background thread is technically not possible, so using **AndroidSchedulers.mainThread()** we can update anything on the main thread. Internally it utilizes the concept of Handler and Looper to perform the action on the main thread.

RxAndroid uses RxJava **internally** and compiles it. But while using RxAndroid in our project we still add the dependency of RxJava to work with like,

```
implementation 'io.reactivex.rxjava3:rxjava:3.0.0'
implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
```

Use-Cases in Android

RxJava has the power of operators and as the saying goes by, " **RxJava has an operator for almost everything** ".

Case 1:

Consider an example, where we want to do an API call and save it to some storage/file. It would be a long-running task and doing a long-running task on the main thread might lead to unexpected behavior like App Not Responding.

So, to do the above-mentioned task we might think to use AsyncTask as our goto solution. But with Android R, AsyncTask is going to be deprecated, and then libraries like RxJava will be the solution for it.

Using RxJava over AsyncTask helps us to write less code. It provides better management of the code as using AsyncTask might make the code lengthy and hard to manage.

Case 2:

Consider a use-case where we might want to fetch user details from an API and from the user's ID which we got from the previous API we will call another API and fetch the user's friend list.

Doing it using AsyncTask we might have to do use multiple Asynctask and manage the results in was way where we want to combine all the AsyncTask to return the result as a single response.

But using RxJava we can use the power of **zip** operator to combine the result of multiple different API calls and return a single response.

Case 3:

Consider an example of doing an API call and getting a list of users and from that, we want only the data which matches the given current condition.

A general approach is to do the API call, and from the Collection, we can then filter the content of that specific user based on the condition and then return the data.

But using RxJava we can directly filter out the data while returning the API response by using the **filter** operator and we do all of this by doing the thread management.

These are a few use cases to understand RxJava for Android and why we need RxAndroid in our project.

10) What is Android WorkManager?

Android WorkManager is an API introduced by Google to simplify and manage background tasks in Android applications. It serves as a unified solution that abstracts away the differences between various versions of Android and their limitations regarding background processing. WorkManager enables developers to schedule and execute tasks reliably, even across device reboots.

What are the key features of WorkManager?

WorkManager offers several essential features, including:

- Support for one-time and periodic tasks.
- Ability to define constraints for task execution, such as network availability or device charging status.
- Guaranteed task execution, even across device reboots.
- Seamless integration with other Jetpack components, such as LiveData and ViewModel.

11) How can you handle orientation changes in an Android application?

Orientation changes, such as rotating the device, can cause an Activity to restart. To handle orientation changes properly, you can override the `onSaveInstanceState()` method to save important data. The saved data can be retrieved in the `onCreate()` or `onRestoreInstanceState()` method to restore the previous state of the Activity. Additionally, using Fragments and ViewModel can provide a more robust solution for managing orientation changes.

12) What is SSL Pinning?

SSL (Secure Sockets Layer) pinning involves hard-coding or "pinning" a specific SSL certificate or its public key within the app. This means that the app will only trust connections to the server if the presented certificate matches the pinned certificate. This helps protect against attacks that involve using fraudulent certificates to intercept or manipulate the communication between the app and the server.

Why SSL Pinning?

1. **MitM Attacks Prevention:** SSL pinning prevents attackers from intercepting communication by presenting a different, unauthorized certificate.
2. **Certificate Authority Compromise:** If a Certificate Authority (CA) is compromised, attackers can issue fraudulent certificates. SSL pinning reduces reliance on CAs for trust.
3. **Enhanced Security:** Pinning reduces the attack surface, making it difficult for attackers to exploit vulnerabilities in the certificate infrastructure.

13) What is Retrofit?

Retrofit is a popular HTTP client library for Android and Java-based applications, which simplifies making HTTP requests and processing their responses. It uses annotations to define the request parameters and the response structure.

Retrofit is based on the four pillars of HTTP request-response cycle:

1. **Request method** *The HTTP request method, also known as a verb, is used to specify the desired action to be performed on the resource. Retrofit supports all HTTP request methods such as GET, POST, PUT, DELETE, etc.*
2. **Endpoint** *An endpoint is a URL that specifies the location of a resource. Retrofit uses annotations to specify the endpoint for each API call.*
3. **Request body** *The request body is the data sent to the server as part of the API call. In Retrofit, you can use annotations to specify the type of data being sent in the request body.*

4. **Response body** *The response body is the data received from the server as a result of the API call. In Retrofit, you can define the expected response type using annotations.*

These four pillars work together to provide a simple and powerful way to interact with REST APIs in Android applications. By using Retrofit, developers can easily define their API calls and handle the request and response data with minimal boilerplate code.