**Android platform**

Android is a software stack that includes an operating system, middleware and key applications. While Google is the main actor which comes to mind, the Open Handset Alliancealso collaborates on Android's development and release. The operating system is based on the Linux kernel

**Android SDK**

Android development starts with the Android SDK (Software Development Kit). While there are many different programming languages and a host of IDEs (Integrated Development Environments) you can use to create an app, the SDK is a constant.

SDK provides a selection of tools required to build Android apps or to ensure the process goes as smoothly as possible. Whether you end up creating an app with Java, Kotlin or C#, you need the SDK to get it to run on an Android device and access unique features of the OS.

**Anatomy of the Android SDK**

The Android SDK can be broken down into several components. These include:

- **Platform-tools**

The Platform tools are more specifically suited to the version of Android that you want to target.

- **Build-tools-**

needed to build your Android apps. This includes the zipalign tool for instance, which optimizes the app to use minimal memory when running prior to generating the final APK, and the apksigner which signs the APK (surprise!) for subsequent verification.

- **SDK-tools-**

These are what will actually create the APK – turning your Java program into an Android app that can be launched on a phone. These include a number of build tools, debugging tools, and image tools. An example is DDMS, which is what lets us use the Android Device Monitor to check the status of an Android device.

- **The Android Debug Bridge (ADB)**

The Android Debug Bridge (ADB) is a program that allows you to communicate with any Android device. It relies on Platform-tools in order to understand the Android version that is being used on said device and hence it is included in the Platform-tools package.

- **Android Emulator**

The Android emulator is what lets you test and monitor apps on a PC, without necessarily needing to have a device available.

**Android Version and its Feature:-**

- **Android 9-Pie**
  - Api Lvl 28
  - Rounded corners across the UI
  - Quick settings menu change.
  - Notification bar, the clock has moved to the left.
  - The "dock" now has a semi-transparent background.
  - New transitions when switching between apps, or within apps
  - Volume slider updated
  - Richer messaging notifications: with full conversation, large images, smart replies
  - The power options now has a "screenshot" button

- **Android 8- Oreo**
  - Api Lvl 27
  - Android Instant apps
  - Improved notifications system
  - Improved system settings
  - Lock screen redesign
- **Android 7- Nougat**
  - Api Lvl 25
  - Long press on the app icon enable new launch actions
  - The default keyboard allows now to send GIFs directly
  - Multi-window mode
- **Android 6- MarshMallow**
  - Api Lvl 23
  - USB Type-C support
  - Fingerprint Authentication support
  - Better battery life with "deep sleep"

**Manage Different Screen Size**

For Different screen size, The following is a list of resource directories in an application that provides different layout designs for different screen sizes and different bitmap drawables for small, medium, high, and extra high density screens.

```
res/layout/my_layout.xml            // layout for normal screen size ("default")
res/layout-small/my_layout.xml      // layout for small screen size
res/layout-large/my_layout.xml      // layout for large screen size
res/layout-xlarge/my_layout.xml     // layout for extra large screen size
res/layout-xlarge-land/my_layout.xml // layout for extra large in landscape orientation

res/drawable-mdpi/my_icon.png       // bitmap for medium density
res/drawable-hdpi/my_icon.png       // bitmap for high density
res/drawable-xhdpi/my_icon.png      // bitmap for extra high density
```

The following code in the Manifest supports all dpis.

```
<supports-screens android:smallScreens="true"
        android:normalScreens="true"
        android:largeScreens="true"
        android:xlargeScreens="true"
        android:anyDensity="true" />
```

**RESTFul API & JSON**

REST is a simple way to organize interactions between independent systems.

1. **Enabling Internet Access**

Making use of a REST API obviously involves using the Internet. However, Android applications can access the Internet only if they have the `android.permission.INTERNET` permission.

**2. Creating Background Threads**

The Android platform does not allow you to run network operations on the main thread of the application. Therefore, all your networking code must belong to a background thread. The easiest way to create such a thread is to use the execute() method of the AsyncTask class. As its only argument, execute() expects a Runnable object.

```
syncTask.execute(new Runnable() {
```

```
@Override
public void run() {
   // All your networking logic
   // should be here
}
;
```

**3. Creating an HTTP Connection**

By using the openConnection() method of the URL class, you can quickly set up a connection to any REST endpoint. Both HttpURLConnection and HttpsURLConnection allow you to perform operations such as adding request headers and reading responses.

The following code snippet shows you how to set up a connection with the GitHub API's root endpoint:

```
′ Create URL
RL githubEndpoint = new URL("https://api.github.com/");

′ Create connection
ttpsURLConnection myConnection = (HttpsURLConnection) githubEndpoint.openConnection();
```

**5. Reading Responses**

Once you have passed all the request headers, you can check if you have a valid response using the getResponseCode() method of the HttpURLConnection object.

```
(myConnection.getResponseCode() == 200) {
 // Success
 // Further processing here
else {
 // Error handling code goes here
```

**6. Parsing JSON Responses**

The Android SDK has a class called **JsonReader**, which makes it very easy for you to parse JSON documents. You can create a new instance of the JsonReader class by passing the InputStreamReaderobject to its constructor.

```
onReader jsonReader = new JsonReader(responseBodyReader);
```

**Android Activity Lifecycle**

| Method | Description |
|---|---|
| onCreate | called when activity is first created. |
| onStart | called when activity is becoming visible to the user. |
| onResume | called when activity will start interacting with the user.<br>OnResume method gets called everytime when an activity moves between background to foreground state. |
| onPause | called when activity is not visible to the user. |
| onStop | called when activity is no longer visible to the user.<br>The user leaves the current activity. |
| onRestart | called after your activity is stopped, prior to start. |

| | |
|---|---|
| | **onRestart gets called only when onStop method is called.** |
| onDestroy | called before the activity is destroyed.<br>The user takes out the activity from the "recent apps" screen. |

**Fragment LifeCycle**

- **onAttach()**The fragment instance is associated with an activity instance.
- **onCreate()** The system calls this method when creating the fragment
- **onCreateView()** The system calls this callback when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a **View** component from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.
- **onActivityCreated()**The onActivityCreated() is called after the onCreateView() method when the host activity is created.
- **onStart()**The onStart() method is called once the fragment gets visible.
- **onResume()**Fragment becomes active.
- **onPause()** The system calls this method as the first indication that the user is leaving the fragment. This is usually where you should commit any changes that should be persisted beyond the current user session.
- **onStop()**Fragment going to be stopped by calling onStop()
- **onDestroyView()**Fragment view will destroy after call this method
- **onDestroy()**onDestroy() called to do final clean up of the fragment's state but Not guaranteed to be called by the Android platform.

**RECYCLER VIEW**

Android RecyclerView is more advanced version of ListView with improved performance

```
public class MoviesAdapter extends RecyclerView.Adapter<MoviesAdapter.MyViewHolder> {
  private List<Movie> moviesList;
  public class MyViewHolder extends RecyclerView.ViewHolder {
    public TextView title, year, genre;

    public MyViewHolder(View view) {
      super(view);
      title = (TextView) view.findViewById(R.id.title);
      genre = (TextView) view.findViewById(R.id.genre);
      year = (TextView) view.findViewById(R.id.year);
    }
  }
```

```
    public  MoviesAdapter(List<Movie> moviesList) {
      this.moviesList = moviesList;
    }
    @Override
    public  MyViewHolder onCreateViewHolder(ViewGroup parent, int  viewType) {
      View itemView = LayoutInflater.from(parent.getContext())
            .inflate(R.layout.movie_list_row, parent, false);

      return  new  MyViewHolder(itemView);
    }

    @Override
    public  void  onBindViewHolder(MyViewHolder holder, int  position) {
      Movie movie = moviesList.get(position);
      holder.title.setText(movie.getTitle());
      holder.genre.setText(movie.getGenre());
      holder.year.setText(movie.getYear());
    }

    @Override
    public  int  getItemCount() {
      return  moviesList.size();
    }
}
```

**In MainActivity**

```
recyclerView = (RecyclerView) findViewById(R.id.recycler_view);
      mAdapter = new  MoviesAdapter(movieList);
      RecyclerView.LayoutManager mLayoutManager = new  LinearLayoutManager(getApplicationContext());
      recyclerView.setLayoutManager(mLayoutManager);
      recyclerView.setItemAnimator(new  DefaultItemAnimator());
      recyclerView.setAdapter(mAdapter);
```

To feed all your data to the list, you must extend the RecyclerView.Adapter class. This object creates views for items, and replaces the content of some of the views with new data items when the original item is no longer visible.

The layout manager calls the adapter's onCreateViewHolder() method. That method needs to construct a RecyclerView.ViewHolder and set the view it uses to display its contents. The type of the ViewHolder must match the type declared in the Adapter class signature. Typically, it would set the view by inflating an XML layout file. Because the view holder is not yet assigned to any particular data, the method does not actually set the view's contents.

The layout manager then binds the view holder to its data. It does this by calling the adapter's onBindViewHolder() method, and passing the view holder's position in the RecyclerView. The onBindViewHolder() method needs to fetch the appropriate data, and use it to fill in the view holder's layout. If the list needs an update, call a notification method on the RecyclerView.Adapter object, such as notifyItemChanged(). The layout manager then rebinds any affected view holders, allowing their data to be updated.

**Android Services**

**Android service** is a component that is *used to perform operations on the background* such as playing music, handle network transactions, interacting content providers etc. It doesn't has any UI (user interface).
The service runs in the background indefinitely even if application is destroyed.

There can be two forms of a service.The lifecycle of service can follow two different paths: started or bound.

*1) Started Service*

A service is started when component (like activity) calls **startService()** method, now it runs in the background indefinitely. It is stopped by **stopService()** method. The service can stop itself by calling the **stopSelf()** method.

*2) Bound Service*

A service is bound when another component (e.g. client) calls **bindService()** method. The client can unbind the service by calling the **unbindService()** method.

**Services vs IntentServices**

**When to use?**
- The *Service* can be used in tasks with no UI, but shouldn't be too long. If you need to perform long tasks, you must use threads within Service.
- The *IntentService* can be used in long tasks usually with no communication to Main Thread. If communication is required, can use Main Thread handler or broadcast intents. Another case of use is when callbacks are needed (Intent triggered tasks).

**How to trigger?**
- The *Service* is triggered by calling method startService().
- The *IntentService* is triggered using an Intent, it spawns a new worker thread and the method onHandleIntent() is called on this thread.

**Triggered From**
- The *Service* and *IntentService* may be triggered from any thread, activity or other application component.

**Runs On**
- The *Service* runs in background but it runs on the Main Thread of the application.
- The *IntentService* runs on a separate worker thread.

**Limitations / Drawbacks**
- The *Service* may block the Main Thread of the application.
- The *IntentService* cannot run tasks in parallel. Hence all the consecutive intents will go into the message queue for the worker thread and will execute sequentially.

**When to stop?**
- If you implement a *Service*, it is your responsibility to stop the service when its work is done, by calling stopSelf() or stopService(). (If you only want to provide binding, you don't need to implement this method).
- The *IntentService* stops the service after all start requests have been handled, so you never have to call stopSelf().

**BroadCast Receivers**

**Broadcast Receivers** simply respond to broadcast messages from other applications or from the system itself.

There are following two important steps to make BroadcastReceiver works for the system broadcasted intents −

- Creating the Broadcast Receiver.

```
public class MyReceiver extends BroadcastReceiver {

  @Override

  public void onReceive(Context context, Intent intent) {

    Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();

  }
```

```
}
```

- Registering Broadcast Receiver

An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file.

**FireBase**

**Firebase** is a platform for mobile developers to develop awesome-quality apps, quickly grow a user base, and monetize apps.
It provides the following features for the Development phase

1. Integration with Cloud Messaging.
2. Robust Authentication for added security.
3. Realtime Database for realtime storage of app data.
4. Storage support for files.
5. Support for on the fly Remote Configuration.
6. Test Lab to deliver high quality apps.
7. Crash Reporting to keep your apps stable and free from bugs.

**FiresBase Listners**

**ValueEventListener**: A ValueEventListener listens for data changes to a specific location in your database - i.e a node. ValueEventListener has one event callback method, onDataChange() to read a static snapshot of the contents at a given path, as they existed at the time of the event

**ChildEventListener**: A ChildEventListener listens for changes to the children of a specific database reference, for example the root node of a database.

**Firebase Cloud Messaging (FCM)**

is a cross-platform messaging solution that lets you reliably deliver messages at no cost.

- A service that extends FirebaseMessagingService. This is required if you want to do any message handling beyond receiving notifications on apps in the background. To receive notifications in foregrounded apps, to receive data payload, to send upstream messages, and so on, you must extend this service.

```
<service
    android:name=".java.MyFirebaseMessagingService"
    android:exported="false">
    <intent-filter>
       <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
 </service>
```

### *How to handle notifications in foreground*

When the app is in foreground, all received messages are processed by the app and if you need to do some logic with them (*i.e get the message details and locally save them in a database*) or change the UI (*use a larger icon or change the status bar icon*) of the notification in the ***onMessageReceived*** method from the class which extends from ***FirebaseMessagingService*** is the place we need to do it:

```
public class MyFirebaseMessagingService extends FirebaseMessagingService {
    public static final String TAG = "MsgFirebaseServ";
    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
      //check for the data/notification entry from the payload
```

```
    }
}
```

# AsyncTask

The `AsyncTask` class allows to run instructions in the background and to synchronize again with the main thread. It also reporting progress of the running tasks. AsyncTasks should be used for short background operations which need to update the user interface.

To use `AsyncTask` you must subclass it. The parameters are the following `AsyncTask <TypeOfVarArgParams, ProgressValue, ResultValue>`.

An `AsyncTask` is started via the `execute()` method. This `execute()` method calls the `doInBackground()` and the `onPostExecute()` method.

TypeOfVarArgParams is passed into the `doInBackground()` method as input. ProgressValue is used for progress information and ResultValue must be returned from `doInBackground()` method. This parameter is passed to `onPostExecute()` as a parameter.

The `doInBackground()` method contains the coding instruction which should be performed in a background thread. This method runs automatically in a separate `Thread`.

The `onPostExecute()` method synchronizes itself again with the user interface thread and allows it to be updated. This method is called by the framework once the `doInBackground()` method finishes.

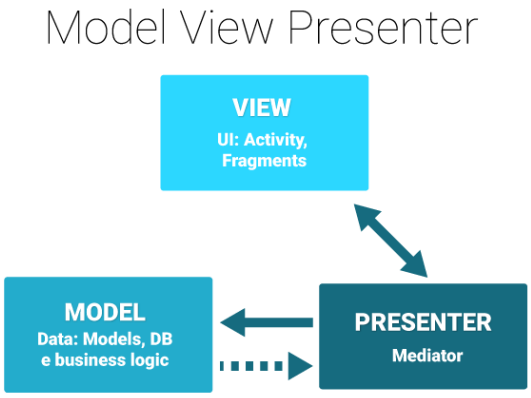
**What is Handler?**

1. Handler allows to add messages to the thread which creates it and It also enables you to schedule some runnable to execute at some time in future.
2. The Handler is associated with the application's main thread. It handles and schedules messages and runnables sent from background threads to the app main thread.
3. If you are doing multiple repeated tasks, for example downloading multiple images which are to be displayed in ImageViews (like downloading thumbnails) upon download, use a task queue with Handler.
4. There are two main uses for a Handler. First is to schedule messages and runnables to be executed as some point in the future; and second Is to enqueue an action to be performed on a different thread than your own.
5. Scheduling messages is accomplished with the the methods like post(Runnable), postAtTime(Runnable, long), postDelayed(Runnable, long), sendEmptyMessage(int), sendMessage(Message), sendMessageAtTime(Message, long), and sendMessageDelayed(Message, long) methods.
6. When a process is created for your application, its main thread is dedicated to running a message queue that takes care of managing the top-level application objects (activities, broadcast receivers, etc) and any windows they create.
7. You can create your own threads, and communicate back with the main application thread through a Handler.

**What is AsyncTask ?**

1. Async task enables you to implement multi threading without get hands dirty into threads. AsyncTask enables proper and easy use methods that allows performing background operations and passing the results back to the UI thread.
2. If you are doing something isolated related to UI, for example downloading data to present in a list, go ahead and use AsyncTask.
3. AsyncTasks should ideally be used for short operations (a few seconds at the most.)
4. An asynchronous task is defined by 3 generic types, called Params, Progress and Result, and 4 steps, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.
5. In onPreExecute you can define code, which need to be executed before background processing starts.
6. doInBackground have code which needs to be executed in background, here in doInBackground we can send results to multiple times to event thread by publishProgress() method, to notify background processing has been completed we can return results simply.
7. onProgressUpdate() method receives progress updates from doInBackground method, which is published via publishProgress method, and this method can use this progress update to update event thread
8. onPostExecute() method handles results returned by doInBackground
9. The generic types used are
    - Params, the type of the parameters sent to the task upon execution,
    - Progress, the type of the progress units published during the background computation.
    - Result, the type of the result of the background computation.
10. If an async task not using any types, then it can be marked as Void type.
11. An running async task can be cancelled by calling cancel(boolean) method.

## MVP in Android




- **View** = a passive interface that displays data and routes user actions to the Presenter. In Android, it is represented by Activity, Fragment, or View.
- **Model** = a layer that holds business logic, controls how data is created, stored, and modified. In Android, it is a data access layer such as database API or remote server API.

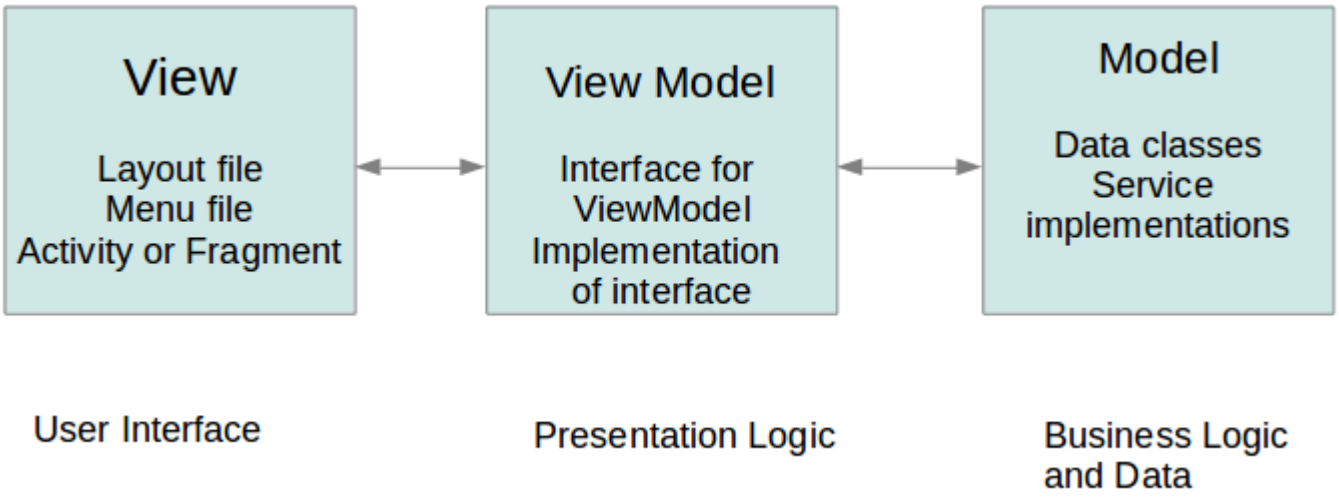
- **Presenter** = A middle man which retrieves data from Model and show it in the View. It also process user action forwarded to it by the View.

**Important points of MVP are:**

- View can not access Model.
- Presenter is tied to a single View. (One-to-One relationship)
- View is completely dumb and passive (only retrieve user action and leave all other things for Presenter to handle).

## The Model View View Model architecture for Android

The Model View View Model design pattern is also known as Model View Binder.




## The view

A view component in MVP contains a visual part of the application.

The view binds to observable variables and actions exposed by the view model typically using the data binding framework.

The view is responsible for handling for example:

- Menus
- Permissions
- Event listeners
- Showing dialogs, Toasts, Snackbars
- Working with Android `View` and `Widget`
- Starting activities
- All functionality which is related to the Android `Context`

## The view model

The view model contains the data required for the view. It is an abstraction of the view and exposes public properties and commands. It uses observable data to notify the view about changes. It also allows to pass events to the model. It is also a value converter from the raw model data to presentation-friendly properties)

The view model has the following responsibilities:

- Exposing data
- Exposing state (progress, offline, empty, error, etc)

- Handling visibility
- Input validation
- Executing calls to the model
- Executing methods in the view

The view model should only know about the application context. the application context can:

- Start a service
- Bind to a service
- Send a broadcast
- Register a broadcast receiver
- Load resource values

It cannot:

- Show a dialog
- Start an activity
- Inflate a layout

## The model

Contains a data provider and the code to fetch and update the data. The data can be retrieved from different sources, for example:

- REST API
- Realm db
- SQLite db
- Handles broadcast
- Shared Preferences
- Firebase
- etc.

Basically the same as the model in the MVP.

## Differences to MVP

MVVM uses data binding and is therefore a more event driven architecture. MVP typically has a one to one mapping between the presenter and the view, while MVVM can map many views to one view model In MVVM the view model has no reference to the view, while in MVP the view knows the presenter.