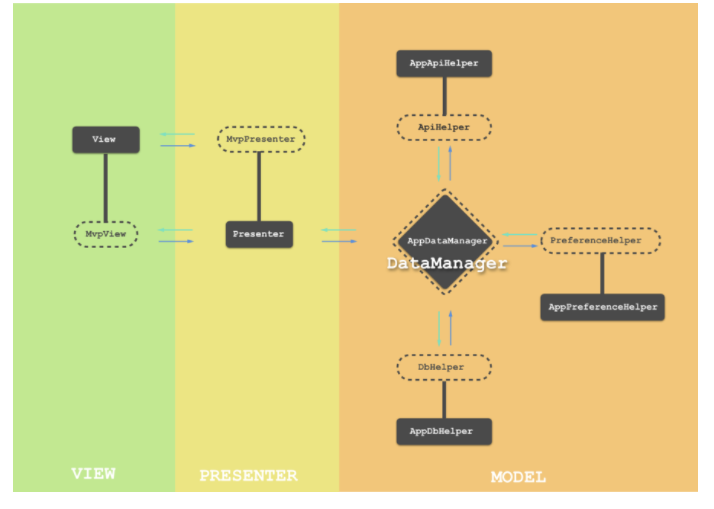
**Best Practices**

* **Always maintain the code quality.** Quality matters a lot. It’s not only about MVP/MVVM/MVC/etc., but also about each piece of code in each part of your app.
* **Detecting and fixing memory leaks in android.** [Read this article to understand](https://mindorks.com/blog/detecting-and-fixing-memory-leaks-in-android).
* **Always include unit tests.** This is the most important part of application development. I recommend running unit tests on the JVM, because it’s much faster than running them on the Android device itself or an emulator. If you require any android dependencies, use [Robolectric](http://robolectric.org/). If you need to mock some objects while testing, use [Mockito](http://mockito.org/).
* Use Dependency Injector to make testing easier. Learn how to use dependency injector from [**here**](https://blog.mindorks.com/introduction-to-dagger-2-using-dependency-injection-in-android-part-1-223289c2a01b) and [**here**](https://blog.mindorks.com/the-new-dagger-2-android-injector-cbe7d55afa6a).
* **Always include functional UI tests.** Functional tests check the functionality of your app from the user’s point of view. They launch your app and test its functionality. Here you can use [Android Instrumentation](https://developer.android.com/studio/test/index.html) if your application is not going to interact with other applications, as it runs only with your application. If there is an interaction with other apps use [UIAutomator](https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html) for testing this functionality.
* As I said earlier, **Always write code for better performance.**
* **Use** [**Proguard**](https://developer.android.com/studio/build/shrink-code.html) **in your release version.** This will remove all your unused code, which will reduce APK size.
* **Use debugging tools.** I highly recommend to use [**Android Debug Database**](https://github.com/amitshekhariitbhu/Android-Debug-Database) . This library will be your best friend. [**Android Debug Database**](https://github.com/amitshekhariitbhu/Android-Debug-Database) is a powerful library for debugging databases and shared preferences in Android applications. It’s a very simple tool for viewing databases and shared preferences directly in your browser.
* **Use strings.xml.** Adding text as string resources is always useful in the long-run, especially when support for new languages needs to be added.
* **Create separate layouts for UI elements that will be re-used.** Then use the include tag in xml. Another handy tag is the <merge/> tag. It acts as a pseudo parent, and helps get rid of an unneeded root ViewGroup. Read [here](https://developer.android.com/training/improving-layouts/reusing-layouts.html) about it.
* **Place launcher icons in mipmap-folders.** When building separate APKs for different densities, drawable folders for other densities get stripped. This will make the icons appear blurry on devices that use launcher icons of higher density. Since mipmap folders do not get stripped, it’s always best to use them for including the launcher icons.
* **Use shapes and selectors instead of images as much as possible.** This further reduces APK size.
* **Avoid deep levels in layouts.** A deep hierarchy of views makes your UI slow, not to mention making it harder to manage your layouts. Deep hierarchies can mostly be avoided by using the correct ViewGroup. Use [Constraint Layout](https://medium.com/@amitshekhar/using-constraint-layout-in-android-531e68019cd).
* Use a HTTP library like [**Fast Android Networking**](https://github.com/amitshekhariitbhu/Fast-Android-Networking), Volley, or Retrofit, depending on your use case.
* **Use the Parcelable class instead of Serializable when passing data in Intents or Bundles.** Serialization of an object that implements the Parcelable interface is much faster than using Java’s default serialization. A class that implements the Serializable interface is marked as serializable, and Java serializes it using reflection (which makes it slow). When using the Parcelable interface, the whole object doesn’t get serialized automatically. Rather, you can selectively add data from the object to a Parcel using which the object is later deserialized.
* **Perform file operations off the UI thread.** File operations should always be performed on a worker thread, typically by using an AsyncTask/Loader. They take time, and if done on the UI thread can make the interface feel sluggish. In situations where they block the UI thread for 5 seconds, an Application Not Responding warning will be triggered and shown to the user.
* **Understand Bitmaps.** As it takes a huge amount of memory. can lead to OOM easily. Users love content! Especially when the content is well formatted and looks nice. Images, for instance, are extremely nice content, mainly due to their property of conveying a thousand words per image. They also consume a lot of memory. A lot of memory! —[Learn From Here](https://developer.android.com/training/displaying-bitmaps/index.html).
* **Understand Context.** Learn from [here](https://blog.mindorks.com/understanding-context-in-android-application-330913e32514).
* Use styles to avoid duplicate attributes in layout XMLs.
* Use [Fragment](https://developer.android.com/guide/components/fragments.html) when it is required.
* Learn RxJava. RxJava is a Java VM implementation of Reactive Extensions. It has become the single most important skill for Android development.

**Best Practice-MVP**



The blueprint of the Android architecture drawn above contains all these features and is based on the principles of MVP.

Let’s understand each part of the sketched architecture.

* **View**: It is the part of the application which renders the UI and receives interactions from the user. Activity, Fragment, and CustomView constitute this part.
* **MvpView**: It is an interface, that is implemented by the View. It contains methods that are exposed to its Presenter for the communication.
* **Presenter**: It is the decision-making counterpart of the View and is a pure java class, with no access to Android APIs. It receives the user interactions passed on from its View and then takes the decision based on the business logic, finally instructing the View to perform specific actions. It also communicates with the DataManager for any data it needs to perform business logic.
* **MvpPresenter**: It is an interface, that is implemented by the Presenter. It contains methods that are exposed to its View for the communication.
* **AppDbHelper**: Database management and all the data handling related to a database is done in this part of the application.
* **DbHelper**: It is an interface implemented by the AppDbHelper and contains methods exposed to the application components. This layer decouples any specific implementation of the DbHelper and hence makes AppDbHelper as plug and play unit.
* **AppPreferenceHelper**: It is like AppDbHelper but is given the job to read and write the data from android shared preferences.
* **PreferenceHelper**: Interface just like DbHelper but implemented by AppPreferenceHelper.
* **AppApiHelper**: It manages the network API calls and API data handling.
* **ApiHelper**: It is an interface just like DbHelper but implemented by AppApiHelper.
* **DataManager**: It is an interface that is implemented by the AppDataManager. It contains methods, exposed for all the data handling operations. Ideally, it delegates the services provided by all the Helper classes. For this DataManager interface extends DbHelper, PreferenceHelper and ApiHelper interfaces.
* **AppDataManager**: It is the one point of contact for any data related operation in the application. DbHelper, PreferenceHelper, and ApiHelper only work for DataManager. It delegates all the operations specific to any Helper.

**Android SDK features**

The Android SDK (software development kit) is a set of development tools used to develop applications for Android platform. The Android SDK includes the following:

* Required libraries
* Debugger
* An emulator
* Relevant documentation for the Android application program interfaces (APIs)
* Sample source code
* Tutorials for the Android OS
* **Offline maps:** Your app can now download arbitrary regions of the globe for offline use.
* **Telemetry:** The world is a constantly changing place, and telemetry allows the map to keep up with it.
* **Camera API :** The previous [Camera API](https://www.mapbox.com/blog/android-camera-api/) exposed only one method to change the camera position.
* **Dynamic markers:** You can now update the position and the icon of a marker dynamically without having to fall back to removing and re-adding it.
* **Map padding:** By default, a MapboxMap fills the entire region defined by its container element. With map padding, you can add a frame-like padding around the map to help you design UIs that can overlap the map
* **Improved API compatibility:** With this release, we’ve made it much easier to help you migrate from the Google Maps Android API.

Although the SDK can be used to write Android programs in the command prompt, the most common method is by using an integrated development environment (IDE). The recommended IDE is Eclipse with the Android Development Tools (ADT) plug-in. However, other IDEs, such as NetBeans or IntelliJ, will also work. Most of these IDEs provide a graphical interface enabling developers to perform development tasks faster. Since Android applications are written in Java code, a user should have the Java Development Kit (JDK) installed.