## 

This guide is for developers who are past the basics of building an app, and now want to know the best practices and recommended architecture for building robust, production-quality apps. This page covers the following information:

[**Common problems faced by app developers**](#_bi6aoqrj7zph) **1**

[**Common Architectural Principles**](#_ap447yr2y592) **3**

[**How to Architect an App**](#_w3bae8s9762b) **3**

[User Interface](#_wvx6u1ucl8ud) 4

[Fetching Data](#_csm4rgkgvf62) 7

[Dependencies Between Components:](#_x6c3f1mgmg7e) 9

[Connecting ViewModel and The Repository](#_c60t032sxfb1) 9

[Caching Data](#_egx6kj32d8ry) 10

[Handling Application Restarts](#_tsvy3a69ize) 11

[Single Source Of Truth](#_6s0h8uf7g37u) 15

[Testing](#_5jzg85dqvyjs) 15

[The Final Architecture](#_ghkbwu2m4rm8) 16

[**Guiding principles**](#_127n8oliht7a) **16**

[**Addendum**](#_szmgyvro4ck4) **18**

[Exposing Network Status](#_grs5yqs1jjca) 18

**Target audience:** This document assumes that the reader has familiarity with the Android Framework. If you are new to the platform, you should first check out the [Getting Started](https://developer.android.com/training/index.html) training series, which covers prerequisite topics for this guide.

## Common problems faced by app developers

*This document is a sign, not a cop. It is impossible to have one way of writing apps that will be the best for every scenario. That being said, choices mentioned here should be a good starting point for most use cases. If you already have a good way of writing Android applications, you don’t need to change, but this document may help you improve your architecture.*

Unlike their traditional desktop counterparts which, in the majority of cases, have a single entry point from the launcher shortcut and run as a single monolithic process, Android applications have a much more complex structure. A typical Android application is constructed out of multiple moving pieces - activities, fragments, services, content providers and broadcast receivers, just to name a few.

Most of these pieces are declared in the [application manifest](https://developer.android.com/guide/topics/manifest/manifest-intro.html) which is used by the Android OS to decide how to integrate your application into the overall user experience with their devices. While, as mentioned earlier, a desktop app is traditionally running as a monolithic process, a properly written Android application needs to be much more flexible as the user weaves their way through the multitude of applications on their device, constantly switching flows and tasks.

As one example out of many, consider what happens when you want to share a photo in your favorite social network application. It triggers a camera intent from which the Android OS launches a camera application to handle the request. At this point, the user leaves the social network application but their experience is seamless. The camera application in return may trigger other intents, like opening the file chooser, from which the user may launch another application. Eventually the user comes back to the social media application and shares the photo. The user may receive a phone call during any step in this process and come back to the photo-sharing flow minutes later.

This level of application hopping is the common case, not the exception, so your application should handle these flows correctly. Keep in mind that mobile devices are resource constrained, so at any time, the operating system may need to kill some applications to make room for new ones.

The Android Framework has well-defined APIs to handle these use cases. Activities, services, content providers, and more allow your application to participate in these flows but they are [**not** the building blocks for your application architecture](https://plus.google.com/+DianneHackborn/posts/FXCCYxepsDU). They are merely entry points to your application from the OS.

## Common Architectural Principles

If activities, services and content providers are not the building blocks for applications, how should the applications be structured?

The most important thing you should focus on is the [separation of concerns](https://en.wikipedia.org/wiki/Separation_of_concerns) in your applications. It is a **common mistake** to write all your code in an Activity or a Fragment. Any code that does not handle a UI or operating system interaction should **not** be in these classes. Keeping them as lean as possible will allow you to avoid many lifecycle related problems. Don’t forget that you don’t **own** those classes, they are just glue classes that embody the contract between the OS and your app. The Android OS may destroy them at any time based on user interactions or other factors like low memory. It is best to minimize your dependency on them to provide a solid user experience.

The second important principle is that you should drive your UI from a model, preferably a persistent model. Persistence is important because it is common for apps to be torn down and recreated, like when the OS kills apps to regain resources. Models are components that are responsible for handling the data for the application. They are independent from the Views or operating system components of your application, hence they are isolated from the lifecycle concerns of those components. Keeping UI code simple and free of app logic makes it easier to manage. Basing your app on model classes with well defined responsibility of managing the data will make them testable and your app consistent.

## Recommended App Architecture

Let’s start with an example of how we can structure an app around a specific use case.

**Note:** To make it easier, we will refer to actual libraries in this guide. This **does not** mean that you must use these libraries, take them as recommendations suitable for cases like this. Your use case may have a better alternative but for the common case, you can be assured that libraries mentioned here are good choices.

Imagine we have a UI in an application that shows a user profile. This user profile will be fetched from our own private back end using a REST API.

### User Interface

We can start by creating *UserProfileFragment.java* and the related *user\_profile\_layout.xml*layout file.

There are 2 data elements we need for this UI.

**The User ID:** The identifier for the user. It is best to pass this information into the fragment using the fragment arguments. If the Android OS needs to destroy your process, this information will be preserved by the OS so the next time your application is restarted, it can recover the user id.

**The User Object**:The actual user data. We will create a *UserProfileViewModel* class to keep this information.

* **ViewModel** - provides the data for a specific UI component (fragment, activity) and handles the communication with the business part of data handling - such as calling other components to load the data or forwarding user modifications. The ViewModel does not know about the View and is not affected by configuration changes such as recreating an Activity due to rotation.

Now we have 3 files.

* *user\_profile.xml:* The UI definition for the screen.
* *UserProfileViewModel.java:* The class that prepares the data for the UI.
* *UserProfileFragment.java*: The UI controller that displays the data in the ViewModel and reacts to user interactions.

Below are our starting implementations (the layout file is left out for simplicity):

|  |
| --- |
| public class UserProfileViewModel extends ViewModel {  private String userId;  private User user;   public void init(String userId) {  this.userId = userId;  }  public User getUser() {  return user;  } } |

## 

|  |
| --- |
| public class UserProfileFragment extends LifecycleFragment {  private static final String UID\_KEY = "uid";  private UserProfileViewModel viewModel;   @Override  public void onActivityCreated(@Nullable Bundle savedInstanceState) {  super.onActivityCreated(savedInstanceState);  String userId = getArguments().getString(UID\_KEY);  viewModel = ViewModelProviders.of(this).get(UserProfileViewModel.class);  viewModel.init(userId);  }  @Override  public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {  return inflater.inflate(R.layout.user\_profile, container, false);  } } |

**Note:** The example above extends LifecycleFragment instead of a the Fragment class. After lifecycles project reaches api stable, the Fragment class will implement LifecycleOwner.

Now we have these three pieces, how do we connect them? After all, when the *ViewModel’s* *user* field is set, we need a way to inform the UI. This is where the *LiveData* class comes in.

* **LiveData** is an observable data holder. It allows observing changes to the data it holds across multiple components of your application without creating explicit and rigid dependency paths between them. It respects the complex lifecycle paths of your application components (activities, fragments, services) and does the right thing to prevent object leaking so that your application does not constantly consume more memory and lead to overall system problems due to memory thrashing.

**Note:** If you are already using a library like [RxJava](https://github.com/ReactiveX/RxJava) or [Agera](https://github.com/google/agera), you can keep using them instead of LiveData. But when you use them or other approaches, make sure you are handling the lifecycle properly such that your data streams pause when the related LifecycleOwner is stopped and the streams are destroyed when the LifecycleOwner is destroyed. You can also add the *android.arch.lifecycle:reactivestreams* artifact to use **LiveData** with another reactive streams library (e.g. RxJava2).

We are going to replace the *user* field in the *UserProfileViewModel* with a *LiveData<User>* so that the Fragment can be informed when the data is updated. The great thing about *LiveData* is that it is lifecycle aware and will automatically clean up references when it is not needed anymore.

|  |
| --- |
| public class UserProfileViewModel extends ViewModel {  ...  ~~private User user;~~  private LiveData<User> user;  public LiveData<User> getUser() {  return user;  } } |

Now we will connect it to the Fragment.

|  |
| --- |
| @Override public void onActivityCreated(@Nullable Bundle savedInstanceState) {  super.onActivityCreated(savedInstanceState);  viewModel.getUser().observe(this, user -> {  // update UI  }); } |

Now every time the *user* data is updated, the *onChanged* callback will be invoked and the UI will be refreshed.

* If you are familiar with other libraries where observable callbacks are used, you might have realized that we didn’t have to override *Fragment.onStop*to stop observing the data. This is not necessary with LiveData as it is lifecycle aware, which means it will not invoke the callback unless the fragment is in an active state (received *onStart* but did not receive *onStop*). It will also automatically remove the observer when the Fragment receives *onDestroy*.
* We also didn’t do anything special to handle configuration changes (e.g. user rotating the screen). The ViewModel is automatically restored when the configuration changes, so as soon as the new fragment comes to life, it will receive the same ViewModel and the callback will be called instantly with the current data. This is **why** the ViewModel should not reference Views directly; it outlives the View’s lifecycle.

### Fetching Data

Now we have connected the ViewModel with the Fragment, but how does the ViewModel obtain the user data? For this example, we will assume that our back end

provides a REST API and we have written a *Webservice* class that allows us to communicate with the back end. In this example, we are using the [Retrofit](http://square.github.io/retrofit/) library to access the back end services. It is a good library, but you are free to use a different library that serves the same purpose.

The class has the following interface:

|  |
| --- |
| public interface Webservice {  // declares a HTTP GET request  @GET("/users/{user}")  // @Path("user") annotation on the userId parameter marks it as a  // replacement for the {user} placeholder in the @GET path  Call<User> getUser(@Path("user") String userId); } |

A naive implementation of the *ViewModel* could directly call the *Webservice* to fetch the data and assign it back to the user object. Even though it will help you quickly show the data, the short term benefit will quickly cost you more maintenance and instability as your application grows. It gives too much responsibility to the ViewModel class which goes against the separation of concerns that we’ve mentioned earlier. In addition to that, ViewModels are tied to an Activity or Fragment scope and losing all of the data when Activity/Fragment is finished is a bad user experience. Instead, our ViewModel will delegate this work to a new class called **Repository.**

* **Repository** classes are responsible for handling data operations. They provide a clean API to the rest of the application. They know where to get the data from and what API calls to make when data is updated. You can consider them as mediators between different data sources (persistent model, web service, cache, etc.).

Below, we create a Repository class that uses the WebService directly to fetch the user data item.

|  |
| --- |
| public class UserRepository {  private Webservice webservice;  // ...  public LiveData<User> getUser(int userId) {  // This is not an optimal implementation, we’ll fix it below  final MutableLiveData<User> data = new MutableLiveData<>();  webservice.getUser(userId).enqueue(new Callback<User>() {  @Override  public void onResponse(Call<User> call, Response<User> response) {  // error case is left out for brevity  data.setValue(response.body());  }  });  return data;  } } |

Even though this Repository class looks unnecessary, it serves an important purpose; it abstracts the data sources from the rest of the application. Now our ViewModel does not know that the data arrives from the Web service, which means we can swap it as necessary

**Note:** We’ve left out the error case for the sake for simplicity. You can see an [alternative implementation](#_grs5yqs1jjca) in the addendum that exposes loading status as well.

#### Dependencies Between Components:

The *UserRepository* class above needs an instance of the *WebService* to do its work. It could simply create it but to do that, It would also need to know the dependencies of the WebService class to properly construct it. This would significantly complicate and duplicate the code (e.g. each class that needs a WebService would need to know how to construct it with its dependencies). In addition to this, UserRepository is probably not the only class that needs a *WebService*. If each such class creates a new *WebService,* it would be very resource heavy.

There are 2 major patterns you can use to tackle this problem:

* [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_injection): Dependency injection allows classes to define their dependencies without constructing them. At runtime, another class is responsible to provide these dependencies. Google provides [Dagger 2](https://google.github.io/dagger/) library for managing / providing these dependencies and using Dagger 2 is the recommended approach. Dagger 2 does the heavy lifting and provides compile time guarantees on dependencies.
* [Service Locator:](https://en.wikipedia.org/wiki/Service_locator_pattern) Service Locator provides a Registry where classes can obtain their dependencies instead of constructing them. It is relatively easier to implement than Dependency Injection (DI), so if you are not familiar with DI, you can use a Service Locator instead.

These patterns will allow you to scale your code because they provide clear patterns to add new functionalities to your codebase without affecting other classes. Both of them also allow swapping implementations for testing; which is one of the main benefits of using them.

For the rest of this example, we are going to use [Dagger 2](https://google.github.io/dagger/) for managing / providing these dependencies.

### Connecting ViewModel and The Repository

|  |
| --- |
| public class UserProfileViewModel extends ViewModel {  private LiveData<User> user;  private UserRepository userRepo;   @Inject // injected by Dagger 2  public UserProfileViewModel(UserRepository userRepo) {  this.userRepo = userRepo;  }   public void init(String userId) {  if (this.user != null) {  // ViewModel is created per Fragment so  // we know the userId won't change  return;  }  user = userRepo.getUser(userId);  }   public LiveData<User> getUser() {  return this.user;  } } |

### Caching Data

The Repository implementation above was good for abstracting the details but since it had only 1 data source, it was not very functional.

There was an obvious problem with the *UserRepository* implementation above. After fetching the data, it did not keep it anywhere. If the user exits the *ProfileFragment* and comes back to it, the application re-fetches the data. This is a bad idea because it is wasting valuable user bandwidth while also causing a bad user experience, since the user has to wait for the new query to complete. To address this, we will add a new data source to our Repository which will cache the User objects.

|  |
| --- |
| @Singleton // informs Dagger that this class should be constructed once public class UserRepository {  private Webservice webservice;  // simple in memory cache, details omitted for brevity  private UserCache userCache;  public LiveData<User> getUser(String userId) {  LiveData<User> cached = userCache.get(userId);  if (cached != null) {  return cached;  }   final MutableLiveData<User> data = new MutableLiveData<>();  userCache.put(userId, data);  // this is still suboptimal but better than before.  // a complete implementation must also handle the error cases.  webservice.getUser(userId).enqueue(new Callback<User>() {  @Override  public void onResponse(Call<User> call, Response<User> response) {  data.setValue(response.body());  }  });  return data;  } } |

### Persisting Data Between App Restarts

Now we have have shown a decent setup to show the UI. If the user rotates the screen, the existing UI will be visible instantly, since the ViewModel is automatically restored. If the user leaves the application and comes back soon, they will see cached data. But what happens if the user leaves the application and comes back hours later, after the Android OS has killed the process?

With the current implementation, we will need to fetch the data again from the network. This is not only a bad user experience, but also wasteful since it will use mobile data to re-fetch the same data. You could simply fix this by caching the web requests, but it creates new problems. What happens if the same User data shows up from another type of request (e.g., a list of friends)? Then your application will possibly show inconsistent data, which is a confusing user experience at best. For instance, the same User’s data may show up differently in friendsrequest and *user* request since they are executed in different times. Your app needs to merge them to avoid showing inconsistent data.

There is a decent way to handle this, using a persistent model. This is where **Room** comes to the rescue.

**Room** is an object mapping library that provides local data persistence with minimal boilerplate code. At compile time, it validates each query against the schema, so that broken SQL queries result in compile time errors instead of runtime failures. It abstracts away some of the underlying implementation details of working with raw SQL tables and queries. It also allows observing changes to the database data (including collections and join queries), exposing such changes via *LiveData* objects. In addition, it explicitly defines thread constraints that address common issues such as accessing storage on the main thread.

**Note:** If you are familiar with another persistence solution like an SQLite ORM or a different database like [Realm](https://realm.io/products/realm-mobile-database/), you don’t need to replace it with Room unless Room’s feature set is more relevant to your use case.

To use Room, we need to define our local schema. We are going to annotate the *User* class with *@Entity* and create a database class for your application.

|  |
| --- |
| @Entity class User {  @PrimaryKey  private int id;  private String name;  private String lastName;  // getters and setters for fields } |

|  |
| --- |
| @Database(entities = {User.class}, version = 1) public abstract class MyDatabase extends RoomDatabase { } |

Notice that *MyDatabase*class is abstract. Room automatically provides an implementation of it: see Room documentation for details.

Now we need a way to save the user data into the database. For this, we’ll create a [data access object](https://en.wikipedia.org/wiki/Data_access_object) and reference it from our database class.

|  |
| --- |
| @Dao public interface UserDao {  @Insert(onConflict = REPLACE)  void save(User user);  @Query("SELECT \* FROM user WHERE id = :userId")  LiveData<User> load(String userId); } |

|  |
| --- |
| @Database(entities = {User.class}) public abstract class MyDatabase extends RoomDatabase {  public abstract UserDao userDao(); } |

Notice that in the *load* function, we return a *LiveData<User>*. Room knows when the database is modified and it will automatically call the observers when the data changes. Because it is using *LiveData*, this will be efficient because it will update the data only if there is an active observer.

**Note:** As of the alpha 1 release, Room checks invalidations based on table modifications which means it may dispatch false positive notifications. There is future work to better analyze queries to avoid such false positives.

Now we can change our *UserRepository* to incorporate this.

|  |
| --- |
| @Singleton public class UserRepository {  private final Webservice webservice;  private final UserDao userDao;  private final Executor executor;   @Inject  public UserRepository(Webservice webservice, UserDao userDao, Executor executor) {  this.webservice = webservice;  this.userDao = userDao;  this.executor = executor;  }   public LiveData<User> getUser(String userId) {  refreshUser(userId);  // return a LiveData directly from the database.  return userDao.load(userId);  }   private void refreshUser(final String userId) {  executor.execute(() -> {  // running in a background thread  // check if user was fetched recently  boolean userExists = userDao.hasUser(FRESH\_TIMEOUT);  if (!userExists) {  // refresh the data  Response response = webservice.getUser(userId).execute();  // TODO check for error etc.  // Update the database.The LiveData will automatically refresh so  // we don’t need to do anything else here besides updating the database  userDao.save(response.body());  }  });  } } |

Notice that even though we changed where the data comes from (in the *UserRepository* class), we didn’t change our *UserProfileViewModel*. This is the flexibility provided by the abstraction. You can see that this is great for testing since you can provide a fake *UserRepository* while testing your *UserProfileViewModel.* Throughout this exercise, we showed three revisions of the *UserRepository* that first fetched from the network directly, then used an in-memory cache and then used a persistent cache, all without changing a single line of code in our ViewModel or Fragment classes.

Now our code is complete. If the user comes back to the same UI days later, they will instantly see the user information because we’ve persisted it. Meanwhile, our Repository will update the data in the background if the data is stale. Of course, depending on your use case, you may prefer not to show the persisted data if it is too old.

In some use cases, you would like to notify the user if there is a network operation happening. For instance, in the case of pull-to-refresh, it is important for a responsive UI to know that a network operation is happening. It is usually good practice to separate the UI action from the actual data since it might be updated for various reasons (e.g. if we fetch a list of friends, there is a good chance that the same user will be fetched so LiveData<User> will update). From the UI’s perspective, the fact that there is a request in flight is just another data point, similar to any other data (like the User object).

There are 2 common solutions for this use case:

* Change *getUser* to return a LiveData that includes the status of the network operation. An example implementation is provided in the [Exposing Network Status](#_grs5yqs1jjca) section.
* Provide another public function in the *Repository* class that can return the refresh status of the User. This option is usually better if you want to show the network status in your UI only in response to explicit user action (like pull-to-refresh).

If you are using a reactive streams library like RxJava, error handling is already built in, so you can use that instead.

#### Single Source Of Truth

It is very common for different API endpoints to return the same data. For instance, if we had another endpoint that returns the friend list, the same user object would come from 2 different API endpoints, maybe in different granularity. If we were to return the webservice request response as-is from the *UserRepository,* our UIs could potentially show inconsistent data since the data might change on the server side between these requests. This is why in the *UserRepository* implementation, the web service callback just saves the data into the database. Then the database will invoke callbacks on active *LiveData* objects if their results are affected by the change.

In this model, the database serves as the single point of truth, and other parts of the app access it via the repository. Regardless of whether you use a disk cache, we recommend using the repository model to provide a single source of truth to the rest of your application.

### Testing

We’ve mentioned the benefits of separation to achieve testability. Lets see how we can test the pieces above.

User Interface & Interactions:

* This will be the only time you need an Android UI Instrumentation test. The best way to test UI code is to create an [Espresso](https://developer.android.com/training/testing/ui-testing/espresso-testing.html) test. You can create the fragment and provide it a mock ViewModel. Since the fragment only talks to the ViewModel, mocking it will be sufficient to fully test this UI.

ViewModel:

* The ViewModel can be tested using an Android JUnit test. You only need to mock the *UserRepository* to test it.

UserRepository:

* You can test the *UserRepository* using a JUnit test as well. You need to mock the Webservice and the DAO. You can test that it makes the right Webservice calls, saves the result into the database and does not make any unnecessary requests if the data is cached and up to date. Since both *Webservice* and *UserDao* are interfaces, you can mock them or create fake implementations for more complex test cases..

UserDao:

* The recommended approach for testing DAO classes is using instrumentation tests. Since these instrumentation tests do not require any UI, they will still run fast. For each test, you can create an in-memory database to ensure that the test does not have any side effects (like changing the database files on the disk).
* Room also allows specifying the database implementation so you can test it by providing it the JUnit implementation of the SupportSQLiteOpenHelper. This approach is usually not recommended because the SQLite version running on the device may differ from the SQLite version on your host machine.

Webservice:

* It is important to make tests independent from the outside world so even your Webservice tests should avoid making network calls. There are plenty of libraries that help with this. For instance, [MockWebService](https://github.com/square/okhttp/tree/master/mockwebserver) is a great library that can help you create a fake local server for your tests.

### The Final Architecture

## 

## Guiding principles

Programming is a creative field, and writing Android applications is not an exception. There are almost always more than one way to solve a specific problem, be it communicating data between multiple activities or fragments, retrieving remote data and persisting it locally for offline mode, or any number of other common scenarios that non-trivial applications encounter, sooner or later.

While the following recommendations are not mandatory, it has been our experience that following them will make your code base more robust, testable and maintainable in the long run.

* The entry points you define in your manifest - activities, services, broadcast receivers, etc. - are not the source of data. Instead, they should only be coordinating the subset of data that is relevant to that entry point. Since each such entry point is rather short lived, depending on the user’s interaction with their device and the overall current health of the runtime, you do not want any of these entry points to be the source of data.
* Be merciless in creating well defined boundaries of responsibility between various modules of your application. For example, don’t spread the code that loads data from the network across multiple classes or packages in your code base. Similarly, don’t stuff unrelated responsibilities - such as data caching and data binding - into the same class.
* Expose as little as possible from each such module. Do not be tempted to create “just that one” shortcut that exposes internal implementation detail from one module. You may gain a bit of time now, but that will be erased many times over as your codebase evolves.
* As you define these interaction seams between the modules, always think about how to make each one testable in isolation. For example, having a well-defined API for fetching data from the network will make it much easier to test the module that persists that data in a local database. If, instead, you mix the logic from these two modules in one place, or sprinkle your networking code across your entire code base, it will be much harder - if not impossible - to test.
* The core of your application is what makes it stand out from the rest. Don’t spend time reinventing wheels that are already there. Don’t write the same boilerplate code again and again. Focus your mental energy on what makes your application unique, and let the Android Architecture Components and other recommended libraries handle the repetitive boilerplate.
* Persist as much relevant and fresh data as possible so that your application is usable when the device is in offline mode. While you may enjoy constant and high speed connectivity in your office, your users won’t necessarily have it while using your application.

## Addendum

### Exposing Network Status

TODO