

Guide to app architecture

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 assumes you are familiar with the Android Framework. If you are new to Android app development, check out our [Developer guides](https://developer.android.com/guide) (<https://developer.android.com/guide>), which cover prerequisite topics for this guide.

Mobile app user experiences

In the majority of cases, desktop apps have a single entry point from a desktop or program launcher, then run as a single, monolithic process. Android apps, on the other hand, have a much more complex structure. A typical Android app contains multiple [app components](https://developer.android.com/guide/components/fundamentals.html#Components) (<https://developer.android.com/guide/components/fundamentals.html#Components>), including activities, fragments, services, content providers, and broadcast receivers.

You declare most of these app components in your [app manifest](https://developer.android.com/guide/topics/manifest/manifest-intro.html) (<https://developer.android.com/guide/topics/manifest/manifest-intro.html>). The Android OS then uses this file to decide how to integrate your app into the device's overall user experience. Given that a properly-written Android app contains multiple components and that users often interact with multiple apps in a short period of time, apps need to adapt to different kinds of user-driven workflows and tasks.

For example, consider what happens when you share a photo in your favorite social networking app:

1. The app triggers a camera intent. The Android OS then launches a camera app to handle the request.

At this point, the user has left the social networking app, but their experience is still seamless.

2. The camera app might trigger other intents, like launching the file chooser, which may launch yet another app.
3. Eventually, the user returns to the social networking app and shares the photo.

At any point during the process, the user could be interrupted by a phone call or notification. After acting upon this interruption, the user expects to be able to return to, and resume, this photo-sharing process. This app-hopping behavior is common on mobile devices, so your app must handle these flows correctly.

Keep in mind that mobile devices are also resource-constrained, so at any time, the operating system might kill some app processes to make room for new ones.

Given the conditions of this environment, it's possible for your app components to be launched individually and out-of-order, and the operating system or user can destroy them at any time. Because these events aren't under your control, **you shouldn't store any app data or state in your app components**, and your app components shouldn't depend on each other.

Common architectural principles

If you shouldn't use app components to store app data and state, how should you design your app?

Separation of concerns

The most important principle to follow is **separation of concerns**

(https://en.wikipedia.org/wiki/Separation_of_concerns). It's a common mistake to write all your code in an **Activity** (<https://developer.android.com/reference/android/app/Activity.html>) or a **Fragment** (<https://developer.android.com/reference/android/app/Fragment.html>). These UI-based classes should only contain logic that handles UI and operating system interactions. By keeping these classes as lean as possible, you can avoid many lifecycle-related problems.

Keep in mind that you don't *own* implementations of **Activity** and **Fragment**; rather, these are just glue classes that represent the contract between the Android OS and your app. The OS can destroy them at any time based on user interactions or because of system conditions like low memory. To provide a satisfactory user experience and a more manageable app maintenance experience, it's best to minimize your dependency on them.

Drive UI from a model

Another important principle is that you should **drive your UI from a model**, preferably a persistent model. *Models* are components that are responsible for handling the data for an app. They're independent from the **View**

(<https://developer.android.com/reference/android/view/View>) objects and app components in your app, so they're unaffected by the app's lifecycle and the associated concerns.

Persistence is ideal for the following reasons:

- Your users don't lose data if the Android OS destroys your app to free up resources.
- Your app continues to work in cases when a network connection is flaky or not available.

By basing your app on model classes with the well-defined responsibility of managing the data, your app is more testable and consistent.

Recommended app architecture

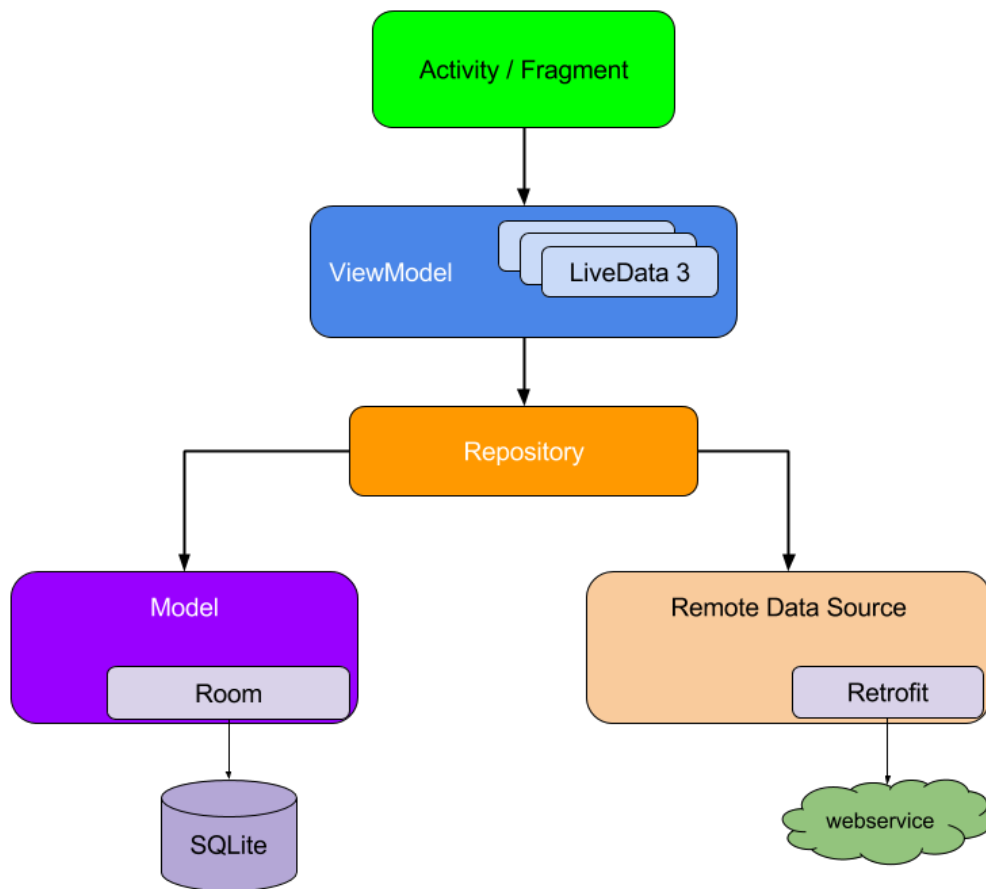
In this section, we demonstrate how to structure an app using [Architecture Components](https://developer.android.com/jetpack/#architecture-components) (<https://developer.android.com/jetpack/#architecture-components>) by working through an end-to-end use case.

Note: It's impossible to have one way of writing apps that works best for every scenario. That being said, this recommended architecture is a good starting point for most situations and workflows. If you already have a good way of writing Android apps that follows the [common architectural principles](#) ([#common-principles](#)), you don't need to change it.

Imagine we're building a UI that shows a user profile. We use a private backend and a REST API to fetch the data for a given profile.

Overview

To start, consider the following diagram, which shows how all the modules should interact with one another after designing the app:



Notice that each component depends only on the component one level below it. For example, activities and fragments depend only on a view model. The repository is the only class that depends on multiple other classes; in this example, the repository depends on a persistent data model and a remote backend data source.

This design creates a consistent and pleasant user experience. Regardless of whether the user comes back to the app several minutes after they've last closed it or several days later, they instantly see a user's information that the app persists locally. If this data is stale, the app's repository module starts updating the data in the background.

Build the user interface

The UI consists of a fragment, `UserProfileFragment`, and its corresponding layout file, `user_profile_layout.xml`.

To drive the UI, our data model needs to hold the following data elements:

- **User ID:** The identifier for the user. It's best to pass this information into the fragment using the fragment arguments. If the Android OS destroys our process, this information is preserved, so the ID is available the next time our app is restarted.

- **User object:** A data class that holds details about the user.

We use a `UserProfileViewModel`, based on the *ViewModel* architecture component, to keep this information.

A **ViewModel** (<https://developer.android.com/topic/libraries/architecture/viewmodel>) object provides the data for a specific UI component, such as a fragment or activity, and contains data-handling business logic to communicate with the model. For example, the `ViewModel` can call other components to load the data, and it can forward user requests to modify the data. The `ViewModel` doesn't know about UI components, so it isn't affected by configuration changes, such as recreating an activity when rotating the device.

We've now defined the following files:

- `user_profile.xml`: The UI layout definition for the screen.
- `UserProfileFragment`: The UI controller that displays the data.
- `UserProfileViewModel`: The class that prepares the data for viewing in the `UserProfileFragment` and reacts to user interactions.

The following code snippets show the starting contents for these files. (The layout file is omitted for simplicity.)

`UserProfileViewModel`

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



`UserProfileFragment`

```
public class UserProfileFragment extends Fragment {  
    private static final String UID_KEY = "uid";  
    private UserProfileViewModel viewModel;  
  
    @Override  
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
```



```

        super.onCreate(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);
    }
}

```

Now that we have these code modules, how do we connect them? After all, when the `user` field is set in the `UserProfileViewModel` class, we need a way to inform the UI. This is where the *LiveData* architecture component comes in.

LiveData (<https://developer.android.com/topic/libraries/architecture/livedata>) is an observable data holder. Other components in your app can monitor changes to objects using this holder without creating explicit and rigid dependency paths between them. The *LiveData* component also respects the lifecycle state of your app's components—such as activities, fragments, and services—and includes cleanup logic to prevent object leaking and excessive memory consumption.

Note: If you're already using a library like [RxJava](https://github.com/ReactiveX/RxJava) (<https://github.com/ReactiveX/RxJava>) or [Agera](https://github.com/google/agera) (<https://github.com/google/agera>), you can continue using them instead of *LiveData*. When you use libraries and approaches like these, however, make sure you handle your app's lifecycle properly. In particular, make sure to pause your data streams when the related **LifecycleOwner** is stopped and to destroy these streams when the related **LifecycleOwner** is destroyed. You can also add the **android.arch.lifecycle:reactivestreams** artifact to use *LiveData* with another reactive streams library, such as *RxJava2*.

To incorporate the *LiveData* component into our app, we change the field type in the `UserProfileViewModel` to `LiveData<User>`. Now, the `UserProfileFragment` is informed when the data is updated. Furthermore, because this **LiveData** (<https://developer.android.com/reference/android/arch/lifecycle/LiveData.html>) field is lifecycle aware, it automatically cleans up references after they're no longer needed.

```
UserProfileViewModel
```

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



Now we modify `UserProfileFragment` to observe the data and update the UI:

`UserProfileFragment`

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



Every time the user profile data is updated, the `onChanged()`.

([https://developer.android.com/reference/android/arch/lifecycle/Observer.html#onChanged\(T\)](https://developer.android.com/reference/android/arch/lifecycle/Observer.html#onChanged(T))) callback is invoked, and the UI is refreshed.

If you're familiar with other libraries where observable callbacks are used, you might have realized that we didn't override the fragment's `onStop()`.

([https://developer.android.com/reference/android/app/Fragment.html#onStop\(\)](https://developer.android.com/reference/android/app/Fragment.html#onStop())) method to stop observing the data. This step isn't necessary with LiveData because it's lifecycle aware, which means it doesn't invoke the `onChanged()` callback unless the fragment is in an active state; that is, it has received `onStart()`.

([https://developer.android.com/reference/android/app/Fragment.html#onStart\(\)](https://developer.android.com/reference/android/app/Fragment.html#onStart())) but hasn't yet received `onStop()`. ([https://developer.android.com/reference/android/app/Fragment.html#onStop\(\)](https://developer.android.com/reference/android/app/Fragment.html#onStop()))).

LiveData also automatically removes the observer when the fragment's `onDestroy()`.

([https://developer.android.com/reference/android/app/Fragment.html#onDestroy\(\)](https://developer.android.com/reference/android/app/Fragment.html#onDestroy())) method is called.

We also didn't add any logic to handle configuration changes, such as the user rotating the device's screen. The `UserProfileViewModel` is automatically restored when the configuration changes, so as soon as the new fragment is created, it receives the same instance of `ViewModel`, and the callback is invoked immediately using the current data. Given that `ViewModel` objects are intended to outlast the corresponding `View` objects that they update, you shouldn't include direct references to `View` objects within your

implementation of **ViewModel**. For more information about the lifetime of a **ViewModel** corresponds to the lifecycle of UI components, see [The lifecycle of a ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel.html#the_lifecycle_of_a_viewmodel) (https://developer.android.com/topic/libraries/architecture/viewmodel.html#the_lifecycle_of_a_viewmodel)

.

Fetch data

Now that we've used **LiveData** to connect the **UserProfileViewModel** to the **UserProfileFragment**, how can we fetch the user profile data?

For this example, we assume that our backend provides a REST API. We use the [Retrofit](http://square.github.io/retrofit/) (<http://square.github.io/retrofit/>) library to access our backend, though you are free to use a different library that serves the same purpose.

Here's our definition of **Webservice** that communicates with our backend:

Webservice

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



A first idea for implementing the **ViewModel** might involve directly calling the **Webservice** to fetch the data and assign this data to our **LiveData** object. This design works, but by using it, our app becomes more and more difficult to maintain as it grows. It gives too much responsibility to the **UserProfileViewModel** class, which violates the [separation of concerns](#) (#separation-of-concerns) principle. Additionally, the scope of a **ViewModel** is tied to an [Activity](https://developer.android.com/reference/android/app/Activity.html) (<https://developer.android.com/reference/android/app/Activity.html>) or [Fragment](https://developer.android.com/reference/android/app/Fragment.html) (<https://developer.android.com/reference/android/app/Fragment.html>) lifecycle, which means that the data from the **Webservice** is lost when the associated UI object's lifecycle ends. This behavior creates an undesirable user experience.

Instead, our **ViewModel** delegates the data-fetching process to a new module, a *repository*.

Repository modules handle data operations. They provide a clean API so that the rest of the app can retrieve this data easily. They know where to get the data from and what API

calls to make when data is updated. You can consider repositories to be mediators between different data sources, such as persistent models, web services, and caches.

Our `UserRepository` class, shown in the following code snippet, uses an instance of `WebService` to fetch a user's data:

`UserRepository`

```
public class UserRepository {  
    private Webservice webservice;  
    // ...  
    public LiveData<User> getUser(int userId) {  
        // This isn't an optimal implementation. We'll fix it later.  
        final MutableLiveData<User> data = new MutableLiveData<>();  
        webservice.getUser(userId).enqueue(new Callback<User>() {  
            @Override  
            public void onResponse(Call<User> call, Response<User> response) {  
                data.setValue(response.body());  
            }  
  
            // Error case is left out for brevity.  
        });  
        return data;  
    }  
}
```

Even though the repository module looks unnecessary, it serves an important purpose: it abstracts the data sources from the rest of the app. Now, our `UserProfileViewModel` doesn't know how the data is fetched, so we can provide the view model with data obtained from several different data-fetching implementations.

Note: We've left out the network error case for the sake of simplicity. For an alternative implementation that exposes errors and loading status, see [Addendum: exposing network status \(#addendum\)](#).

Manage dependencies between components

The `UserRepository` class above needs an instance of `Webservice` to fetch the user's data. It could simply create the instance, but to do that, it also needs to know the dependencies of the `Webservice` class. Additionally, `UserRepository` is probably not the only class that needs a `Webservice`. This situation requires us to duplicate code, as each class that needs a reference to `Webservice` needs to know how to construct it and its

dependencies. If each class creates a new `WebService`, our app could become very resource heavy.

You can use the following design patterns to address this problem:

- Dependency injection (DI) (https://en.wikipedia.org/wiki/Dependency_injection): Dependency injection allows classes to define their dependencies without constructing them. At runtime, another class is responsible for providing these dependencies. We recommend the Dagger 2 (<https://google.github.io/dagger/>) library for implementing dependency injection in Android apps. Dagger 2 automatically constructs objects by walking the dependency tree, and it provides compile-time guarantees on dependencies.
- Service locator (https://en.wikipedia.org/wiki/Service_locator_pattern): The service locator pattern provides a registry where classes can obtain their dependencies instead of constructing them.

It's easier to implement a service registry than use DI, so if you aren't familiar with DI, use the service locator pattern instead.

These patterns allow you to scale your code because they provide clear patterns for managing dependencies without duplicating code or adding complexity. Furthermore, these patterns allow you to quickly switch between test and production data-fetching implementations.

Our example app uses Dagger 2 (<https://google.github.io/dagger/>) to manage the `WebService` object's dependencies.

Connect ViewModel and the repository

Now, we modify our `UserProfileViewModel` to use the `UserRepository` object:

`UserProfileViewModel`

```
public class UserProfileViewModel extends ViewModel {  
    private LiveData<User> user;  
    private UserRepository userRepo;  
  
    // Instructs Dagger 2 to provide the UserRepository parameter.  
    @Inject  
    public UserProfileViewModel(UserRepository userRepo) {  
        this.userRepo = userRepo;  
    }  
}
```



```

public void init(int userId) {
    if (this.user != null) {
        // ViewModel is created on a per-Fragment basis, so the userId
        // doesn't change.
        return;
    }
    user = userRepo.getUser(userId);
}

public LiveData<User> getUser() {
    return this.user;
}
}

```

Cache data

The **UserRepository** implementation abstracts the call to the **Webservice** object, but because it relies on only one data source, it's not very flexible.

The key problem with the **UserRepository** implementation is that after it fetches data from our backend, it doesn't store that data anywhere. Therefore, if the user leaves the **UserProfileFragment**, then returns to it, our app must re-fetch the data, even if it hasn't changed.

This design is suboptimal for the following reasons:

- It wastes valuable network bandwidth.
- It forces the user to wait for the new query to complete.

To address these shortcomings, we add a new data source to our **UserRepository**, which caches the **User** objects in memory:

UserRepository

```

// Informs Dagger that this class should be constructed only once.
@Singleton
public class UserRepository {
    private Webservice webservice;

    // Simple in-memory cache. Details omitted for brevity.
    private UserCache userCache;

    public LiveData<User> getUser(int userId) {
        LiveData<User> cached = userCache.get(userId);
        if (cached != null) {

```



```

        return cached;
    }

    final MutableLiveData<User> data = new MutableLiveData<>();
    userCache.put(userId, data);

    // This implementation is still suboptimal but better than before.
    // A complete implementation also handles 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;
}
}

```

Persist data

Using our current implementation, if the user rotates the device or leaves and immediately returns to the app, the existing UI becomes visible instantly because the repository retrieves data from our in-memory cache.

However, what happens if the user leaves the app and comes back hours later, after the Android OS has killed the process? By relying on our current implementation in this situation, we need to fetch the data again from the network. This refetching process isn't just a bad user experience; it's also wasteful because it consumes valuable mobile data.

You could fix this issue by caching the web requests, but that creates a key new problem: What happens if the same user data shows up from another type of request, such as fetching a list of friends? The app would show inconsistent data, which is confusing at best. For example, our app might show two different versions of the same user's data if the user made the list-of-friends request and the single-user request at different times. Our app would need to figure out how to merge this inconsistent data.

The proper way to handle this situation is to use a persistent model. This is where the Room (<https://developer.android.com/training/data-storage/room/index.html>) persistence library comes to the rescue.

Room (<https://developer.android.com/training/data-storage/room/index.html>) is an object-mapping library that provides local data persistence with minimal boilerplate code. At compile time, it validates each query against your data schema, so broken SQL queries

result in compile-time errors instead of runtime failures. Room abstracts away some of the underlying implementation details of working with raw SQL tables and queries. It also allows you to observe changes to the database's data, including collections and join queries, exposing such changes using *LiveData* objects. It even explicitly defines execution constraints that address common threading issues, such as accessing storage on the main thread.

Note: If your app already uses another persistence solution, such as a SQLite object-relational mapping (ORM), you don't need to replace your existing solution with [Room](https://developer.android.com/training/data-storage/room/index.html) (<https://developer.android.com/training/data-storage/room/index.html>). However, if you're writing a new app or refactoring an existing app, we recommend using Room to persist your app's data. That way, you can take advantage of the library's abstraction and query validation capabilities.

To use Room, we need to define our local schema. First, we add the `@Entity` (<https://developer.android.com/reference/android/arch/persistence/room/Entity.html>) annotation to our `User` data model class and a `@PrimaryKey` ([https://developer.android.com/reference/android/arch/persistence/room/PrimaryKey](https://developer.android.com/reference/android/arch/persistence/room/PrimaryKey.html)) annotation to the class's `id` field. These annotations mark `User` as a table in our database and `id` as the table's primary key:

`User`

```
@Entity
class User {
    @PrimaryKey
    private int id;
    private String name;
    private String lastName;

    // Getters and setters for fields.
}
```



Then, we create a database class by implementing `RoomDatabase` (<https://developer.android.com/reference/android/arch/persistence/room/RoomDatabase.html>) for our app:

`UserDatabase`

```
@Database(entities = {User.class}, version = 1)
public abstract class UserDatabase extends RoomDatabase {
}
```



Notice that `UserDatabase` is abstract. Room automatically provides an implementation of it. For details, see the [Room](https://developer.android.com/training/data-storage/room/) (https://developer.android.com/training/data-storage/room/) documentation.

We now need a way to insert user data into the database. For this task, we create a [data access object \(DAO\)](https://en.wikipedia.org/wiki/Data_access_object). (https://en.wikipedia.org/wiki/Data_access_object).

`UserDao`

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



Notice that the `load` method returns an object of type `LiveData<User>`. Room knows when the database is modified and automatically notifies all active observers when the data changes. Because Room uses *LiveData*, this operation is efficient; it updates the data only when there is at least one active observer.

Note: Room checks invalidations based on table modifications, which means it may dispatch false positive notifications.

With our `UserDao` class defined, we then reference the DAO from our database class:

`UserDatabase`

```
@Database(entities = {User.class}, version = 1)
public abstract class UserDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```



Now we can modify our `UserRepository` to incorporate the Room data source:

```
@Singleton
public class UserRepository {
    private final Webservice webservice;
    private final UserDao userDao;
    private final Executor executor;

    @Inject
    public UserRepository(Webservice webservice, UserDao userDao, Executor ex
```



```

        this.webservice = webservice;
        this.userDao = userDao;
        this.executor = executor;
    }

    public LiveData<User> getUser(String userId) {
        refreshUser(userId);
        // Returns a LiveData object directly from the database.
        return userDao.load(userId);
    }

    private void refreshUser(final String userId) {
        // Runs in a background thread.
        executor.execute(() -> {
            // Check if user data was fetched recently.
            boolean userExists = userDao.hasUser(FRESH_TIMEOUT);
            if (!userExists) {
                // Refreshes the data.
                Response<User> response = webservice.getUser(userId).execute()

                // Check for errors here.

                // Updates the database. The LiveData object automatically
                // refreshes, so we don't need to do anything else here.
                userDao.save(response.body());
            }
        });
    }
}

```

Notice that even though we changed where the data comes from in `UserRepository`, we didn't need to change our `UserProfileViewModel` or `UserProfileFragment`. This small-scoped update demonstrates the flexibility that our app's architecture provides. It's also great for testing, because we can provide a fake `UserRepository` and test our production `UserProfileViewModel` at the same time.

If users wait a few days before returning to an app that uses this architecture, it's likely that they'll see out-of-date information until the repository can fetch updated information. Depending on your use case, you may not want to show this out-of-date information. Instead, you can display *placeholder* data, which shows dummy values and indicates that your app is currently fetching and loading up-to-date information.

Single source of truth

It's common for different REST API endpoints to return the same data. For example, if our backend has another endpoint that returns a list of friends, the same user object could come from two different API endpoints, maybe even using different levels of granularity. If the `UserRepository` were to return the response from the `Webservice` request as-is, without checking for consistency, our UIs could show confusing information because the version and format of data from the repository would depend on the endpoint most recently called.

For this reason, our `UserRepository` implementation saves web service responses into the database. Changes to the database then trigger callbacks on active `LiveData` objects. Using this model, **the database serves as the single source of truth**, and other parts of the app access it using our `UserRepository`. Regardless of whether you use a disk cache, we recommend that your repository designate a data source as the single source of truth for the rest of your app.

Show in-progress operations

In some use cases, such as pull-to-refresh, it's important for the UI to show the user that there's currently a network operation in progress. It's good practice to separate the UI action from the actual data because the data might be updated for various reasons. For example, if we fetched a list of friends, the same user might be fetched again programmatically, triggering a `LiveData<User>` update. From the UI's perspective, the fact that there's a request in flight is just another data point, similar to any other piece of data in the `User` object itself.

We can use one of the following strategies to display a consistent data-updating status in the UI, regardless of where the request to update the data came from:

- Change `getUser()` to return an object of type `LiveData`. This object would include the status of the network operation.

For an example, see the [NetworkBoundResource](https://github.com/google/android-architecture-components/blob/88747993139224a4bb6dbe985adf652d557de621/GithubBrowserSample/app/src/main/java/com/android/example/github/repository/NetworkBoundResource.kt)

(<https://github.com/google/android-architecture-components/blob/88747993139224a4bb6dbe985adf652d557de621/GithubBrowserSample/app/src/main/java/com/android/example/github/repository/NetworkBoundResource.kt>)

implementation in the android-architecture-components GitHub project.

- Provide another public function in the `UserRepository` class that can return the refresh status of the `User`. This option is better if you want to show the network status in your UI only when the data-fetching process originated from an explicit user action, such as pull-to-refresh.

Test each component

In the [separation of concerns](#) (#separation-of-concerns) section, we mentioned that one key benefit of following this principle is testability.


The following list shows how to test each code module from our extended example:

- **User interface and interactions:** Use an [Android UI instrumentation test](https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html) (<https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html>). The best way to create this test is to use the [Espresso](https://developer.android.com/training/testing/ui-testing/espresso-testing.html) (<https://developer.android.com/training/testing/ui-testing/espresso-testing.html>) library. You can create the fragment and provide it a mock `UserProfileViewModel`. Because the fragment communicates only with the `UserProfileViewModel`, mocking this one class is sufficient to fully test your app's UI.
- **ViewModel:** You can test the `UserProfileViewModel` class using a [JUnit test](https://developer.android.com/training/testing/unit-testing/local-unit-tests.html) (<https://developer.android.com/training/testing/unit-testing/local-unit-tests.html>). You only need to mock one class, `UserRepository`.
- **UserRepository:** You can test the `UserRepository` using a JUnit test, as well. You need to mock the `Webservice` and the `UserDao`. In these tests, verify the following behavior:
 - The repository makes the correct web service calls.
 - The repository saves results into the database.
 - The repository doesn't make unnecessary requests if the data is cached and up to date.

Because both `Webservice` and `UserDao` are interfaces, you can mock them or create fake implementations for more complex test cases.

- **UserDao:** Test DAO classes using instrumentation tests. Because these instrumentation tests don't require any UI components, they run quickly.

For each test, create an in-memory database to ensure that the test doesn't have any side effects, such as changing the database files on disk.

-  **Caution:** Room allows specifying the database implementation, so it's possible to test your DAO by providing the JUnit implementation of [SupportSQLiteOpenHelper](https://developer.android.com/reference/android/arch/persistence/db/SupportSQLiteOpenHelper.html) (<https://developer.android.com/reference/android/arch/persistence/db/SupportSQLiteOpenHelper.html>). This approach isn't recommended, however, because the SQLite version running on the device might differ from the SQLite version on your development machine.

- **Webservice:** In these tests, avoid making network calls to your backend. It's important for all tests, especially web-based ones, to be independent from the outside world.

Several libraries, including MockWebServer

(<https://github.com/square/okhttp/tree/master/mockwebserver>), can help you create a fake local server for these tests.

- **Testing Artifacts:** Architecture Components provides a maven artifact to control its background threads. The `android.arch.core:core-testing` artifact contains the following JUnit rules:
 - **InstantTaskExecutorRule:** Use this rule to instantly execute any background operation on the calling thread.
 - **CountingTaskExecutorRule:** Use this rule to wait on background operations of Architecture Components. You can also associate this rule with Espresso as an idling resource (<https://developer.android.com/training/testing/espresso/idling-resource>)

Best practices

Programming is a creative field, and building Android apps isn't an exception. There are many ways to solve a 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 nontrivial apps encounter.

Although the following recommendations aren't mandatory, it has been our experience that following them makes your code base more robust, testable, and maintainable in the long run:

Avoid designating your app's entry points—such as activities, services, and broadcast receivers—as sources of data.

Instead, they should only coordinate with other components to retrieve the subset of data that is relevant to that entry point. Each app component is rather short-lived, depending on the user's interaction with their device and the overall current health of the system.

Create well-defined boundaries of responsibility between various modules of your app.

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 define multiple unrelated responsibilities—such as data caching and data binding—into the same class.

Expose as little as possible from each module.

Don't be tempted to create "just that one" shortcut that exposes an internal implementation detail from one module. You might gain a bit of time in the short term, but you then incur technical debt many times over as your codebase evolves.

Consider how to make each module testable in isolation.

For example, having a well-defined API for fetching data from the network makes it 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 distribute your networking code across your entire code base, it becomes much more difficult—if not impossible—to test.

Focus on the unique core of your app so it stands out from other apps.

Don't reinvent the wheel by writing the same boilerplate code again and again. Instead, focus your time and energy on what makes your app unique, and let the Android Architecture Components and other recommended libraries handle the repetitive boilerplate.

Persist as much relevant and fresh data as possible.

That way, users can enjoy your app's functionality even when their device is in offline mode. Remember that not all of your users enjoy constant, high-speed connectivity.

Assign one data source to be the single source of truth.

Whenever your app needs to access this piece of data, it should always originate from this single source of truth (#truth).

Addendum: exposing network status

In the recommended app architecture (#recommended_app_architecture) section above, we omitted network error and loading states to keep the code snippets simple.

This section demonstrates how to expose network status using a `Resource` class that encapsulate both the data and its state.

The following code snippet provides a sample implementation of `Resource`:

```
// A generic class that contains data and status about loading this data
public class Resource<T> {
```

```

@NonNull public final Status status;
@Nullable public final T data;
@Nullable public final String message;
private Resource(@NonNull Status status, @Nullable T data,
    @Nullable String message) {
    this.status = status;
    this.data = data;
    this.message = message;
}

public static <T> Resource<T> success(@NonNull T data) {
    return new Resource<>(Status.SUCCESS, data, null);
}

public static <T> Resource<T> error(String msg, @Nullable T data) {
    return new Resource<>(Status.ERROR, data, msg);
}

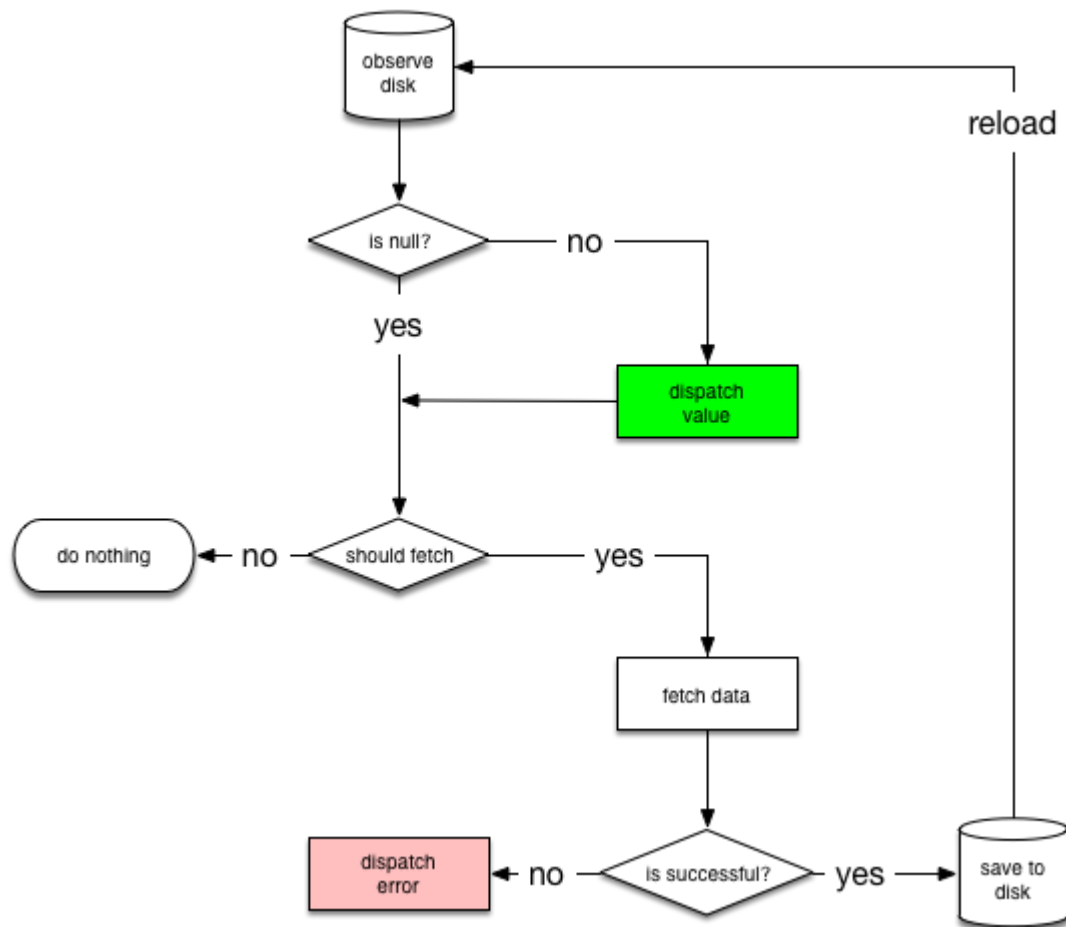
public static <T> Resource<T> loading(@Nullable T data) {
    return new Resource<>(Status.LOADING, data, null);
}

public enum Status { SUCCESS, ERROR, LOADING }
}

```

Because it's common to load data from the network while showing the disk copy of that data, it's good to create a helper class that you can reuse in multiple places. For this example, we create a class called **NetworkBoundResource**.

The following diagram shows the decision tree for **NetworkBoundResource**:



It starts by observing the database for the resource. When the entry is loaded from the database for the first time, `NetworkBoundResource` checks whether the result is good enough to be dispatched or that it should be re-fetched from the network. Note that both of these situations can happen at the same time, given that you probably want to show cached data while updating it from the network.

If the network call completes successfully, it saves the response into the database and re-initializes the stream. If network request fails, the `NetworkBoundResource` dispatches a failure directly.

Note: After saving new data to disk, we re-initialize the stream from the database. We usually don't need to do that, however, because the database itself happens to dispatch the change.

Keep in mind that relying on the database to dispatch the change involves relying on the associated side effects, which isn't good because undefined behavior from these side effects could occur if the database ends up not dispatching changes because the data hasn't changed.

Also, don't dispatch the result that arrived from the network because that would violate the [single source of truth \(#truth\)](#) principle. After all, maybe the database includes triggers that change data values during a "save" operation. Similarly, don't dispatch `SUCCESS` without the new data, because then the client receives the wrong version of the data.

The following code snippet shows the public API provided by **NetworkBoundResource** class for its children:



```
// ResultType: Type for the Resource data.
// RequestType: Type for the API response.
public abstract class NetworkBoundResource<ResultType, RequestType> {
    // Called to save the result of the API response into the database.
    @WorkerThread
    protected abstract void saveCallResult(@NonNull RequestType item);

    // Called with the data in the database to decide whether to fetch
    // potentially updated data from the network.
    @MainThread
    protected abstract boolean shouldFetch(@Nullable ResultType data);

    // Called to get the cached data from the database.
    @NonNull @MainThread
    protected abstract LiveData<ResultType> loadFromDb();

    // Called to create the API call.
    @NonNull @MainThread
    protected abstract LiveData<ApiResponse<RequestType>> createCall();

    // Called when the fetch fails. The child class may want to reset component
    // like rate limiter.
    @MainThread
    protected void onFetchFailed();

    // Returns a LiveData object that represents the resource that's implemented
    // in the base class.
    public final LiveData<Resource<ResultType>> getAsLiveData();
}
```

Note these important details about the class's definition:

- It defines two type parameters, **ResultType** and **RequestType**, because the data type returned from the API might not match the data type used locally.
- It uses a class called **ApiResponse** for network requests. **ApiResponse** is a simple wrapper around the **Retrofit2.Call** class that convert responses to instances of **LiveData**.

The full implementation of the **NetworkBoundResource** class appears as part of the [android-architecture-components GitHub project](#)

(<https://github.com/googlesamples/android-architecture-components/blob/88747993139224a4bb6dbe985adf652d557de621/GithubBrowserSample/app/src/main/java/com/android/example/github/repository/NetworkBoundResource.kt>)

After creating the **NetworkBoundResource**, we can use it to write our disk- and network-bound implementations of **User** in the **UserRepository** class:

UserRepository

```
class UserRepository {  
    Webservice webservice;  
    UserDao userDao;  
  
    public LiveData<Resource<User>> loadUser(final int userId) {  
        return new NetworkBoundResource<User,User>() {  
            @Override  
            protected void saveCallResult(@NonNull User item) {  
                userDao.insert(item);  
            }  
  
            @Override  
            protected boolean shouldFetch(@Nullable User data) {  
                return rateLimiter.canFetch(userId)  
                    && (data == null || !isFresh(data));  
            }  
  
            @NonNull @Override  
            protected LiveData<User> loadFromDb() {  
                return userDao.load(userId);  
            }  
  
            @NonNull @Override  
            protected LiveData<ApiResponse<User>> createCall() {  
                return webservice.getUser(userId);  
            }  
        }.getAsLiveData();  
    }  
}
```



Content and code samples on this page are subject to the licenses described in the [Content License](#) (/license).
Java is a registered trademark of Oracle and/or its affiliates.

Last updated January 22, 2019.



Twitter

Follow @AndroidDev on
Twitter



YouTube

Check out Android Developers
on YouTube