ViewModel Overview

Part of **Android Jetpack** (https://developer.android.com/jetpack?authuser=1).

The `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) class is designed to store and manage UI-related data in a lifecycle conscious way. The `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) class allows data to survive configuration changes such as screen rotations.

Note: To import `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) into your Android project, see the instructions for declaring dependencies in the Lifecycle release notes (https://developer.android.com/jetpack/androidx/releases/lifecycle?authuser=1#declaring_dependencies)
.

The Android framework manages the lifecycles of UI controllers, such as activities and fragments. The framework may decide to destroy or re-create a UI controller in response to certain user actions or device events that are completely out of your control.

If the system destroys or re-creates a UI controller, any transient UI-related data you store in them is lost. For example, your app may include a list of users in one of its activities. When the activity is re-created for a configuration change, the new activity has to re-fetch the list of users. For simple data, the activity can use the `onSaveInstanceState()` (https://developer.android.com/reference/android/app/Activity?authuser=1#onSaveInstanceState(android.os.Bundle)) method and restore its data from the bundle in `onCreate()` (https://developer.android.com/reference/android/app/Activity?authuser=1#onCreate(android.os.Bundle)) , but this approach is only suitable for small amounts of data that can be serialized then deserialized, not for potentially large amounts of data like a list of users or bitmaps.

Another problem is that UI controllers frequently need to make asynchronous calls that may take some time to return. The UI controller needs to manage these calls and ensure the system cleans them up after it's destroyed to avoid potential memory leaks. This management requires a lot of maintenance, and in the case where the object is re-created for a configuration change, it's a waste of resources since the object may have to reissue calls it has already made.

UI controllers such as activities and fragments are primarily intended to display UI data, react to user actions, or handle operating system communication, such as permission

requests. Requiring UI controllers to also be responsible for loading data from a database or network adds bloat to the class. Assigning excessive responsibility to UI controllers can result in a single class that tries to handle all of an app's work by itself, instead of delegating work to other classes. Assigning excessive responsibility to the UI controllers in this way also makes testing a lot harder.

It's easier and more efficient to separate out view data ownership from UI controller logic.

Implement a ViewModel

Architecture Components provides `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) helper class for the UI controller that is responsible for preparing data for the UI. `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) objects are automatically retained during configuration changes so that data they hold is immediately available to the next activity or fragment instance. For example, if you need to display a list of users in your app, make sure to assign responsibility to acquire and keep the list of users to a `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1), instead of an activity or fragment, as illustrated by the following sample code:

KOTLIN JAVA (#JAVA)

```
class MyViewModel : ViewModel() {
    private val users: MutableLiveData<List<User>> by lazy {
        MutableLiveData<List<User>>().also {
            loadUsers()
        }
    }

    fun getUsers(): LiveData<List<User>> {
        return users
    }

    private fun loadUsers() {
        // Do an asynchronous operation to fetch users.
    }
}
```

You can then access the list from an activity as follows:

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        // Create a ViewModel the first time the system calls an activit
        // Re-created activities receive the same MyViewModel instance c

        // Use the 'by viewModels()' Kotlin property delegate
        // from the activity-ktx artifact
        val model: MyViewModel by viewModels()
        model.getUsers().observe(this, Observer<List<User>>{ users ->
            // update UI
        })
    }
}
```

If the activity is re-created, it receives the same `MyViewModel` instance that was created by the first activity. When the owner activity is finished, the framework calls the `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) objects's `onCleared()` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1#onCleared()) method so that it can clean up resources.

Caution: A `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) must never reference a view, `Lifecycle` (https://developer.android.com/reference/androidx/lifecycle/Lifecycle?authuser=1), or any class that may hold a reference to the activity context.

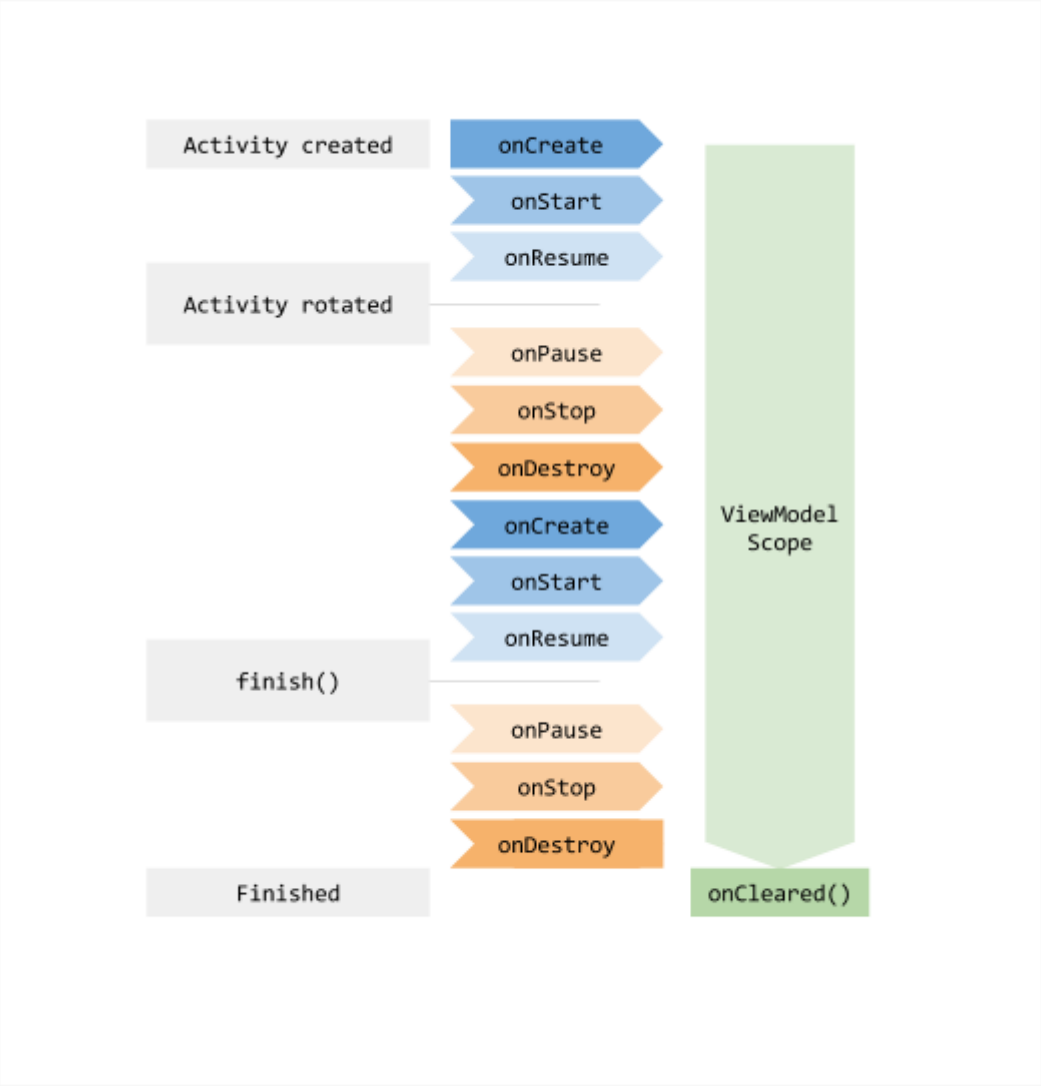
`ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) objects are designed to outlive specific instantiations of views or `LifecycleOwners` (https://developer.android.com/reference/androidx/lifecycle/LifecycleOwner?authuser=1). This design also means you can write tests to cover a `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) more easily as it doesn't know about view and `Lifecycle` (https://developer.android.com/reference/androidx/lifecycle/Lifecycle?authuser=1) objects. `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) objects can contain `LifecycleObservers` (https://developer.android.com/reference/androidx/lifecycle/LifecycleObserver?authuser=1), such as `LiveData` (https://developer.android.com/reference/androidx/lifecycle/LiveData?authuser=1) objects.

However `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) objects must never observe changes to lifecycle-aware observables, such as `LiveData` (https://developer.android.com/reference/androidx/lifecycle/LiveData?authuser=1) objects. If the `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) needs the `Application` (https://developer.android.com/reference/android/app/Application?authuser=1) context, for example to find a system service, it can extend the `AndroidViewModel` (https://developer.android.com/reference/androidx/lifecycle/AndroidViewModel?authuser=1) class and have a constructor that receives the `Application` (https://developer.android.com/reference/android/app/Application?authuser=1) in the constructor, since `Application` (https://developer.android.com/reference/android/app/Application?authuser=1) class extends `Context` (https://developer.android.com/reference/android/content/Context?authuser=1).

The lifecycle of a ViewModel

`ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) objects are scoped to the `Lifecycle` (https://developer.android.com/reference/androidx/lifecycle/Lifecycle?authuser=1) passed to the `ViewModelProvider` (https://developer.android.com/reference/androidx/lifecycle/ViewModelProvider?authuser=1) when getting the `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1). The `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) remains in memory until the `Lifecycle` (https://developer.android.com/reference/androidx/lifecycle/Lifecycle?authuser=1) it's scoped to goes away permanently: in the case of an activity, when it finishes, while in the case of a fragment, when it's detached.

Figure 1 illustrates the various lifecycle states of an activity as it undergoes a rotation and then is finished. The illustration also shows the lifetime of the `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) next to the associated activity lifecycle. This particular diagram illustrates the states of an activity. The same basic states apply to the lifecycle of a fragment.



You usually request a `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) the first time the system calls an activity object's `onCreate()` (https://developer.android.com/reference/android/app/Activity?authuser=1#onCreate(android.os.Bundle)) method. The system may call `onCreate()` (https://developer.android.com/reference/android/app/Activity?authuser=1#onCreate(android.os.Bundle)) several times throughout the life of an activity, such as when a device screen is rotated. The `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) exists from when you first request a `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) until the activity is finished and destroyed.

Share data between fragments

It's very common that two or more fragments in an activity need to communicate with each other. Imagine a common case of split-view (`master-detail`) fragments, where you have a fragment in which the user selects an item from a list and another fragment that displays the contents of the selected item. This case is never trivial as both fragments need to define some interface description, and the owner activity must bind the two together. In addition, both fragments must handle the scenario where the other fragment is not yet created or visible.

This common pain point can be addressed by using `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) objects. These fragments can share a `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) using their activity scope to handle this communication, as illustrated by the following sample code:

KOTLIN JAVA (#JAVA)

```
class SharedViewModel : ViewModel() {
    val selected = MutableLiveData<Item>()

    fun select(item: Item) {
        selected.value = item
    }
}

class MasterFragment : Fragment() {

    private lateinit var itemSelector: Selector

    // Use the 'by activityViewModels()' Kotlin property delegate
    // from the fragment-ktx artifact
    private val model: SharedViewModel by activityViewModels()

    override fun onViewCreated(view: View, savedInstanceState: Bundle?)
        super.onViewCreated(view, savedInstanceState)
        itemSelector.setOnClickListener { item ->
            // Update the UI
        }
    }
}

class DetailFragment : Fragment() {

    // Use the 'by activityViewModels()' Kotlin property delegate
    // from the fragment-ktx artifact
    private val model: SharedViewModel by activityViewModels()
```

```
    override fun onViewCreated(view: View, savedInstanceState: Bundle?)
        super.onViewCreated(view, savedInstanceState)
        model.selected.observe(viewLifecycleOwner, Observer<Item> { item
            // Update the UI
        })
    }
}
```

Notice that both fragments retrieve the activity that contains them. That way, when the fragments each get the `ViewModelProvider` (https://developer.android.com/reference/androidx/lifecycle/ViewModelProvider?authuser=1), they receive the same `SharedViewModel` instance, which is scoped to this activity.

This approach offers the following benefits:

- The activity does not need to do anything, or know anything about this communication.
- Fragments don't need to know about each other besides the `SharedViewModel` contract. If one of the fragments disappears, the other one keeps working as usual.
- Each fragment has its own lifecycle, and is not affected by the lifecycle of the other one. If one fragment replaces the other one, the UI continues to work without any problems.

Replacing Loaders with ViewModel

Loader classes like `CursorLoader` (https://developer.android.com/reference/android/content/CursorLoader?authuser=1) are frequently used to keep the data in an app's UI in sync with a database. You can use `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1), with a few other classes, to replace the loader. Using a `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) separates your UI controller from the data-loading operation, which means you have fewer strong references between classes.

In one common approach to using loaders, an app might use a `CursorLoader` (https://developer.android.com/reference/android/content/CursorLoader?authuser=1) to observe the contents of a database. When a value in the database changes, the loader automatically triggers a reload of the data and updates the UI:

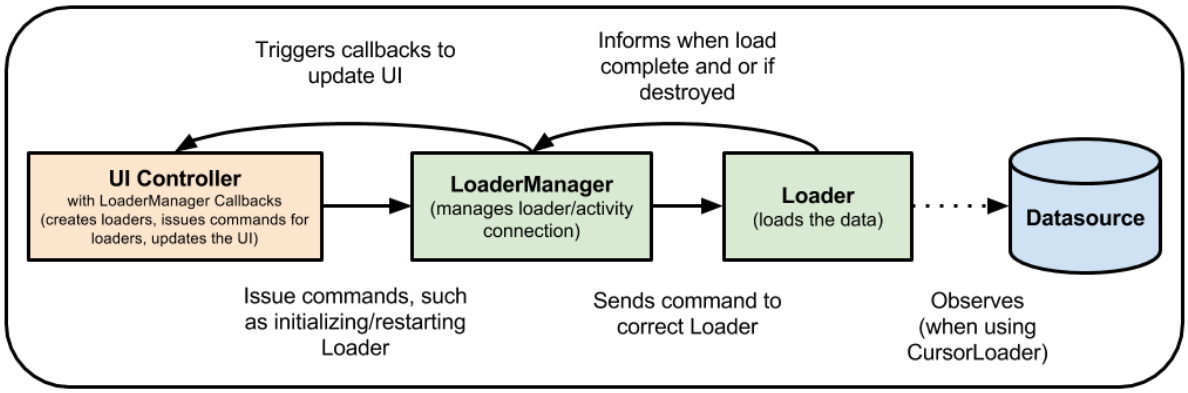


Figure 2. Loading data with loaders

`ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) works with Room (https://developer.android.com/topic/libraries/architecture/room?authuser=1) and LiveData (https://developer.android.com/topic/libraries/architecture/livedata?authuser=1) to replace the loader. The `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) ensures that the data survives a device configuration change. Room (https://developer.android.com/topic/libraries/architecture/room?authuser=1) informs your `LiveData` (https://developer.android.com/reference/androidx/lifecycle/LiveData?authuser=1) when the database changes, and the LiveData (https://developer.android.com/topic/libraries/architecture/livedata?authuser=1), in turn, updates your UI with the revised data.

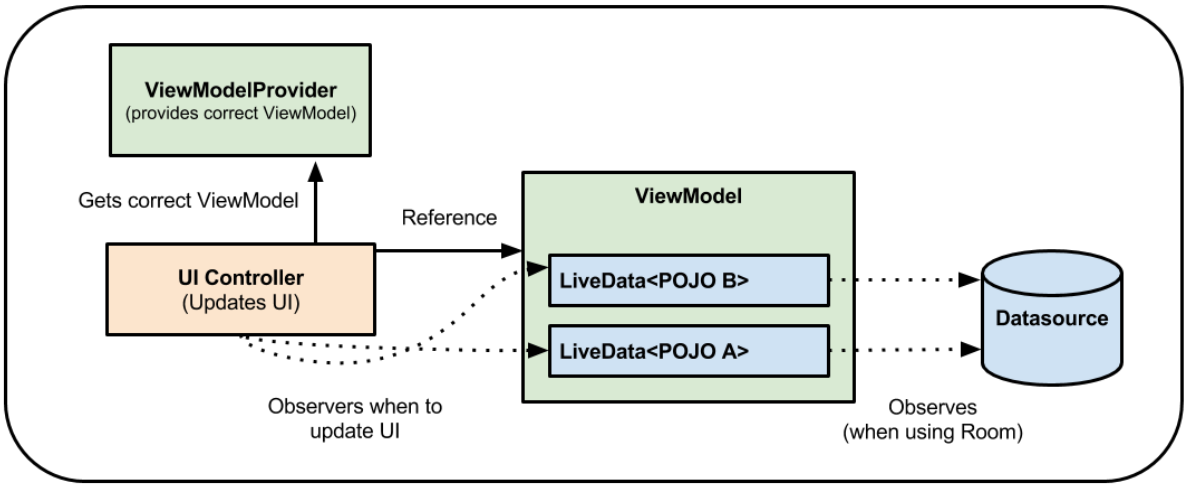


Figure 3. Loading data with ViewModel

Use coroutines with ViewModel

`ViewModel` includes support for Kotlin coroutines. For more information, see Use Kotlin coroutines with Android Architecture Components (https://developer.android.com/topic/libraries/architecture/coroutines?authuser=1).

Further information

As your data grows more complex, you might choose to have a separate class just to load the data. The purpose of `ViewModel` (https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) is to encapsulate the data for a UI controller to let the data survive configuration changes. For information about how to load, persist, and manage data across configuration changes, see Saving UI States (https://developer.android.com/topic/libraries/architecture/saving-states?authuser=1).

The Guide to Android App Architecture (https://developer.android.com/topic/libraries/architecture/guide?authuser=1#fetching_data) suggests building a repository class to handle these functions.

Additional resources

For further information about the `ViewModel` class, consult the following resources.

Samples

- Android Architecture Components basic sample (https://github.com/android/architecture-components-samples/tree/main/BasicSample)
- Sunflower (https://github.com/android/sunflower), a gardening app illustrating Android development best practices with Android Jetpack.

Codelabs

- Android Room with a View (Java) (https://codelabs.developers.google.com/codelabs/android-room-with-a-view?authuser=1) (Kotlin) (https://codelabs.developers.google.com/codelabs/android-room-with-a-view-kotlin?authuser=1)
- Android lifecycle-aware components codelab (https://codelabs.developers.google.com/codelabs/android-lifecycles/?authuser=1#0)

Blogs

- ViewModels : A Simple Example (https://medium.com/androiddevelopers/viewmodels-a-simple-example-ed5ac416317e)
- ViewModels: Persistence, onSaveInstanceState(), Restoring UI State and Loaders (https://medium.com/androiddevelopers/viewmodels-persistence-onsaveinstancestate-restoring-ui-state-and-loaders-fc7cc4a6c090)
- ViewModels and LiveData: Patterns + AntiPatterns (https://medium.com/androiddevelopers/viewmodels-and-livedata-patterns-antipatterns-21efaef74a54)
- Kotlin Demystified: Understanding Shorthand Lambda Syntax (https://medium.com/androiddevelopers/kotlin-demystified-understanding-shorthand-lamba-syntax-74724028dcc5)
- Kotlin Demystified: Scope functions (https://medium.com/androiddevelopers/kotlin-demystified-scope-functions-57ca522895b1)
- Kotlin Demystified: When to use custom accessors (https://medium.com/androiddevelopers/kotlin-demystified-when-to-use-custom-accessors-939a6e998899)
- Lifecycle Aware Data Loading with Architecture Components (https://medium.com/google-developers/lifecycle-aware-data-loading-with-android-architecture-components-f95484159de4)

Videos

- Android Jetpack: ViewModel (https://www.youtube.com/watch?v=5qlIPTDE274&t=30s&authuser=1)

Last updated 2021-04-28 UTC.