



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

Day 3

(ViewModels & Configuration Changes - Philosophy, Implementation, and Roots in AOSP) 🚀



Esmaeeil Moradi

@esmaeeil-moradi700

Day 3



ViewModels & Configuration Changes - Philosophy, Implementation, and Roots in AOSP

**Welcome to the community of Android architects
and seasoned enthusiasts!**

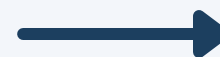
Senior Colleagues,

**In the third step of this expert challenge, we embark
on a comprehensive exploration of the concept of
ViewModels and the management of Configuration
Changes in Android. Our goal extends beyond
examining the technical aspects and interaction
with the Android Open Source Project (AOSP); it
also encompasses understanding the design
philosophy behind ViewModels and why they have
become a fundamental component in modern
Android architecture. Tangible examples of
ViewModel usage will illuminate these abstract
concepts for us.**



Esmaeel Moradi

@esmaeel-moradi700



The Philosophy Behind ViewModel: Separation, Persistence, and UI Facilitation

The core philosophy of ViewModel rests on three pillars:

1. Separation of UI from Logic:

ViewModel acts as a robust intermediary layer between the View (Activity/Composable) and the Model (data layer). This separation establishes clear boundaries between the responsibilities of each part. The View focuses exclusively on displaying data and handling user events, while the ViewModel takes on the task of receiving raw data from the Model, processing it, and preparing it for consumption by the UI. This leads to more organized, readable code with improved maintainability and testability.



Esmaeel Moradi
@esmaeel-moradi700



2. Persistence of State Across Configuration Changes:

Configuration changes, such as screen rotations, language switching, or theme changes, can lead to the complete destruction and recreation of Activities. This process, by default, causes the loss of any state stored within the Activity, resulting in a poor user experience. By having a lifecycle that extends beyond the Activity (and is tied to the LifecycleOwner), ViewModel preserves state throughout these changes. For example, if an Activity has loaded data from a service, the ViewModel retains this data, and there's no need to reload it after the Activity is rebuilt.

3. Facilitating UI Development:

ViewModel provides data in an optimal format for display in the UI. This may involve performing calculations, formatting data, or combining multiple data sources. Furthermore, by using observable mechanisms (such as LiveData or StateFlow), the ViewModel automatically notifies the UI of data changes. This reactive approach minimizes the need for manual UI update management and facilitates the development of dynamic and responsive user interfaces.



Esmaeel Moradi

@esmaeel-moradi700

ViewModel:

The Guardian of State Across Configuration Changes

The crucial ability of a **ViewModel** to retain data across configuration changes stems from its unique lifecycle. Unlike **Activities** and **Fragments**, which are destroyed and recreated, a **ViewModel** persists for the entire lifespan of its associated **ViewModelStoreOwner** (typically an **Activity** or a **Fragment**).

- Here's a breakdown of how this data retention works:

1. ViewModelStore:

Each **ViewModelStoreOwner** (like your **Activity** or **Fragment**) has an associated **ViewModelStore**. This **ViewModelStore** is essentially a container – a **HashMap** under the hood – that holds references to the **ViewModels** associated with that owner.



Esmaeel Moradi

@esmaeel-moradi700



2. Association via ViewModelProvider:

You don't directly interact with the ViewModelStore. Instead, you use a ViewModelProvider to obtain instances of your ViewModels. The ViewModelProvider is lifecycle-aware. When you request a ViewModel for the first time, the ViewModelProvider creates a new instance and stores it in the ViewModelStore associated with your Activity or Fragment.

3. Recreation of the ViewModelStoreOwner:

When a configuration change occurs, the Activity or Fragment is destroyed and a new instance is created. However – and this is the key point – the ViewModelStore associated with the previous instance is not immediately destroyed. The Android framework retains this ViewModelStore.



Esmaeel Moradi

@esmaeel-moradi700



4. Re-obtaining the ViewModel:

When the new Activity or Fragment instance needs its ViewModel, you again use the ViewModelProvider. This time, instead of creating a new ViewModel, the ViewModelProvider checks the retained ViewModelStore. Since the ViewModel instance from the previous Activity/Fragment is still in the ViewModelStore, the ViewModelProvider simply returns the existing instance.

5. ViewModel Remains Untouched:

Because the ViewModel instance itself was never destroyed, any data it holds in its member variables (LiveData, StateFlow, simple variables, etc.) remains intact throughout the configuration change. The new Activity or Fragment instance simply re-connects to the same ViewModel instance and can continue to access the preserved data.



Esmaeel Moradi

@esmaeeil-moradi700



6. ViewModel Destruction:

The ViewModel is only cleared (and its `onCleared()` method is called) when the `ViewModelStoreOwner` is finally destroyed for good – for example, when the Activity finishes and is removed from the back stack, or when a Fragment is detached. This ensures that the ViewModel's resources are cleaned up only when they are no longer needed.



Esmaeel Moradi
@esmaeeil-moradi700

Deeper Dive:



Configuration Changes in AOSP and the Role of AMS:

The Android operating system continuously monitors changes in the device configuration. When a significant change occurs, the system notifies the running application. At the AOSP level, the ActivityManagerService (AMS) plays a crucial role in managing this process. AMS decides which Activities need to be destroyed and recreated, coordinating this process by invoking the relevant lifecycle methods (`onSaveInstanceState()`, `onPause()`, `onStop()`, `onDestroy()`, and then `onCreate()`, `onStart()`, `onResume()`). Understanding this cycle and the limitations of `onSaveInstanceState()` in preserving complex state underscores the importance of using ViewModel.



Esmaeel Moradi

@esmaeeil-moradi700



ViewModel as a Lifecycle-Aware Solution in AndroidX:

The ViewModel component in the AndroidX Lifecycle library is intelligently tied to the lifecycle of UI components (Activities, Fragments, and Composables). The ViewModelStore exists within each LifecycleOwner and is responsible for holding the ViewModel instances associated with it. The ViewModel remains alive as long as its corresponding LifecycleOwner is not completely destroyed (e.g., when the Activity is closed with the Back button), and it is not destroyed during Configuration Changes. This persistence allows for state preservation without the need for complex manual management.



Esmaeel Moradi
@esmaeel-moradi700



Implementation of Lifecycle-Aware Components in AOSP:

By examining the internal architecture of AOSP, we can gain insights into how the `androidx.lifecycle` library mirrors or interacts with similar concepts at the system level. While the exact implementation details are complex and subject to change, understanding the design principles within AOSP for managing the lifecycles of UI components can guide our use of the AndroidX counterparts. This might involve looking at how system services manage the lifecycle of UI-related components and how these principles have inspired the design of the Lifecycle and ViewModel components we use in our applications.



Esmaeel Moradi
@esmaeel-moradi700



State Management at the System Level:

Investigating how state is managed within core system components can provide valuable lessons for our application-level state management strategies within ViewModels. For example, how does the System UI preserve its state during Configuration Changes? What architectural patterns do system services employ to maintain data integrity? Drawing parallels and learning from these system-level approaches can lead to more robust and efficient state management within our ViewModels, especially when dealing with complex data or interactions



Esmaeel Moradi
@esmaeeil-moradi700



ViewModel Factories and Interaction with Service Manager:

For a ViewModel to perform its tasks, it often requires dependencies such as Repositories or Use Cases. Creating and managing these dependencies can be intricate. Patterns like ViewModel Factories help us standardize the process of creating ViewModels with their required dependencies. Frameworks like Hilt also automate this process, facilitating the management of the lifecycle and scope of dependencies. At the AOSP level, although there isn't a direct interaction between ViewModel Factories and Service Manager at the application level, the concept of managing and accessing services across the operating system can inspire similar patterns for managing dependencies within the application layer.



Esmaeel Moradi
@esmaeeil-moradi700



Performance and Resource Management in AOSP:

At the AOSP level, performance and resource management are critical. We will discuss how improper handling of state and lifecycle events, including those related to Configuration Changes, can lead to memory leaks and performance degradation. Understanding the principles employed within AOSP to prevent such issues (e.g., careful resource allocation and deallocation, avoiding long-running operations on the main thread) can inform our best practices when using ViewModels, especially in long-lived applications or those dealing with substantial amounts of data.



Esmaeel Moradi
@esmaeeil-moradi700

? Key Interview Questions:

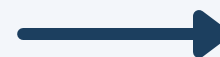


- **What is ViewModel, and what problem does it solve in Android architecture?**
- **How does ViewModel preserve data during Configuration Changes? Describe the ViewModelStore and ViewModelProvider mechanism.**
- **What is the difference between onSaveInstanceState() and ViewModel for managing state? In what scenarios would you use each?**
- **What is the role of a ViewModel Factory, and why do we need it when injecting dependencies into a ViewModel?**
- **How can you share a ViewModel between multiple Fragments?**
- **What are Lifecycle-Aware Components in Android, and how does ViewModel interact with them?**
- **What are the challenges of managing complex state in a ViewModel, and what patterns exist to solve them (e.g., using LiveData, StateFlow, Redux-like patterns)?**



Esmaeel Moradi

@esmaeel-moradi700



? Challenge Question for Experts:

Based on your experience in AOSP, what specific considerations exist regarding state and lifecycle management in operating system UI components that could inspire the design and use of ViewModels at the application level? Are there similar mechanisms for preserving state at the Framework level, and how does ViewModel interact with these concepts?



Esmaeel Moradi

@esmaeeil-moradi700

We eagerly await your
valuable insights and
experiences! Let's deepen
our understanding of
ViewModels &
Configuration Changes
together. 🚀

LEAVE A COMMENT BELOW



Esmaeel Moradi

@esmaeel-moradi700