



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

Day 13

State Management in UI

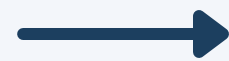
From Simple remember to Robust Architectures with
ViewModel and UDF 🚀



Esmaeail Moradi

@esmaeail-moradi700

Day 13



Mastering State Management in Jetpack Compose: From Simple remember to Robust Architectures with ViewModel and UDF 🚀

Greetings again to all professional colleagues, Android engineers, and architects!

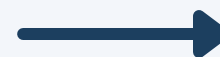
In our Day 12 article, we took our first steps into the exciting world of Jetpack Compose and familiarized ourselves with its basics. We saw how Composable functions declaratively describe our UI and how `remember` and `mutableStateOf` help us manage simple states. However, as any experienced developer knows, managing state in real-world applications, especially as complexity grows, can quickly become a major challenge.

Today, we're moving beyond the fundamentals and diving into the beating heart of a well-crafted Compose application: Effective State Management. This is precisely where the difference between a Compose app that merely "works" and one that is "properly architected, maintainable, testable, and performant" becomes clear. If you thought `remember { mutableStateOf(...) }` was the whole story of state management in Compose, prepare for a deep dive that will transform your app architectures! As someone who has grappled with state management challenges in various UI architectures (from MVP and MVVM in the View System to modern approaches in Compose) for over a decade, I want to share practical experiences and patterns with you.



Esmaeel Moradi

@esmaeel-moradi700



Introduction:

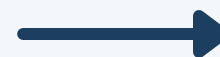
State – The Heartbeat of Your Compose UI.

As we saw the previous day, UI in Compose is a function of state ($UI = f(state)$). Any change in state potentially leads to a change in the UI. Therefore, how we define, hold, pass, and mutate these states directly impacts the readability, stability, performance, and testability of our code. Effective state management is the key to building complex yet understandable Compose applications. This is where we transition from being a "Compose coder" to a "Compose architect."



Esmaeel Moradi

@esmaeel-moradi700



Revisiting Core State Concepts (Briefly, with an Expert's Nuance):

- **remember and mutableStateOf:** These are your fundamental tools for creating and remembering state that survives Recompositions. `mutableStateOf` creates an observable `State<T>` object whose value changes notify Compose to trigger Recomposition for dependent parts.
- **Recomposition:** The smart and efficient process of rebuilding the UI by Compose in response to state changes. Remember: Compose only Recomposes the Composables that directly read the changed state. By managing state correctly and logically structuring your Composables, you can prevent unnecessary Recompositions and maintain high performance. "Excessive Recomposition" is a myth if you follow the principles.



Esmaeel Moradi
@esmaeel-moradi700



State Hoisting – The Cornerstone of Reusable Composables (A Full Dissection):

This is one of the most important and fundamental patterns in Compose that should become second nature.

"Why" Should We Hoist State?

- **Stateless Composables:** By moving state out of a Composable, we turn it into a "Stateless" function. Stateless Composables are highly reusable because their behavior depends solely on their inputs.
- **Easier Testability:** Testing Stateless Composables is significantly simpler.
- **Centralized Logic:** State management and modification logic becomes centralized in a higher, common ancestor, which improves code readability and maintainability.
- **Sharing State:** If multiple Composables need to access or modify a shared state, that state must be hoisted to their nearest common ancestor.



Esmaeel Moradi

@esmaeel-moradi700



"How" to Hoist State?

The common pattern is for the child Composable to receive the current state value as an input parameter and an event callback (usually a lambda function) to notify the parent of a request to change the state.

```
// Stateless Composable
@Composable
fun MyTextField(
    text: String, // State passed down
    onChange: (String) -> Unit, // Event flows up
    modifier: Modifier = Modifier
) {
    TextField(
        value = text,
        onChange = onChange, // Notify parent of change
        modifier = modifier
    )
}

// Stateful Composable (Parent)
@Composable
fun MyScreenWithInput() {
    var name by remember { mutableStateOf("") } // State owned by parent

    Column {
        Text("Your name: $name")
        MyTextField(
            text = name, // Pass state down
            onChange = { newName -> name = newName } // Handle event up
        )
    }
}
```



Esmaeel Moradi
@esmaeel-moradi700



When and How Far to Hoist State?

A good rule of thumb is to hoist state to the level where it's needed by all Composables that read or modify it, but no higher. Hoisting too far can reduce readability.

State Types and State Holders:

Differentiating State Types:

- **UI Element State:** Internal, visual state of a specific Composable (e.g., expanded/collapsed state of a DropdownMenu, scroll position of a LazyListState). This state is usually managed close to the UI element itself.
- **Screen/Feature State:** The overall state of a screen or a specific application feature, potentially used by multiple Composables. This often includes business logic.
- **Application State:** States that are global to the entire application (e.g., user login status, theme settings).



Esmaeel Moradi

@esmaeel-moradi700



- **Simple State Holders:**

To encapsulate several related state variables and their modification logic for a Composable or a small UI section, you can create a plain Kotlin class. An instance of this class can be remembered in the Composition.

```
// Simple State Holder class
class MyCounterState(initialCount: Int = 0) {
    var count by mutableStateOf(initialCount)
    private set // Only expose increment

    fun increment() {
        count++
    }
}

@Composable
fun rememberMyCounterState(initialCount: Int = 0): MyCounterState =
    remember(initialCount) { MyCounterState(initialCount) }

@Composable
fun AnotherCounter() {
    val counterState = rememberMyCounterState()
    Button(onClick = { counterState.increment() }) {
        Text("Counter: ${counterState.count}")
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700



ViewModel as a Screen/Feature Level State Holder:

For more complex screen-level states that require business logic and need to survive configuration changes (like screen rotation), `AndroidViewModel` or `ViewModel` from the Jetpack `ViewModel` library is the standard and recommended choice.

- Integration with Compose: Using the `viewModel()` function from the `androidx.lifecycle:lifecycle-viewmodel-compose` library.
- Consuming State from ViewModel: Typically, `StateFlow` or `SharedFlow` (or `LiveData`) is used in the `ViewModel` to hold and emit state. In the `Composable`, this state is converted to a Compose-usable `State<T>` using `collectAsStateWithLifecycle()` (for `Flows`) or `observeAsState()` (for `LiveData`). Using `collectAsStateWithLifecycle` is highly recommended as it aligns `Flow` collection with the `Composable`'s lifecycle, preventing resource waste when the UI is in the background.



Esmaeel Moradi

@esmaeel-moradi700



```
// MyScreenViewModel.kt
class MyScreenViewModel : ViewModel() {
    private val _name = MutableStateFlow("")
    val name: StateFlow<String> = _name.asStateFlow()

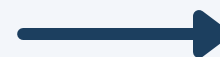
    fun onNameChange(newName: String) {
        _name.value = newName
    }
}

// MyScreenComposable.kt
@Composable
fun MyScreenWithViewModel(viewModel: MyScreenViewModel = viewModel()) {
    // Collects the StateFlow as State, respecting lifecycle
    val name by viewModel.name.collectAsStateWithLifecycle()

    Column {
        Text("Entered name: $name")
        TextField(value = name, onValueChange = { viewModel.onNameChange(it) })
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700



Unidirectional Data Flow (UDF) – The Guiding Principle:

UDF is an architectural pattern where:

- **State flows down:** Parent Composables pass state down to their children.
- **Events flow up:** Children notify their parents of changes or events via callback functions. Compose is inherently designed to work best with UDF. State Hoisting and the use of ViewModels facilitate the implementation of UDF.
- **Benefits:** High predictability of UI behavior, easier testability of components, and simplification of the debugging process.



Esmaeel Moradi
@esmaeel-moradi700

Managing Complex State Scenarios:

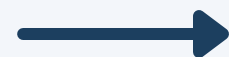
- **Lists and Dynamic Content (LazyColumn / LazyRow):** Use `rememberLazyListState()` to remember and control scroll state.
- **Critical Importance of key:** Define a unique and stable key for each item in a LazyColumn/Row. This helps Compose correctly preserve the internal state of items (like remembered values within them) and optimize Recomposition when the list changes (items added/removed/reordered).
- **Form Handling & Validation:** Typically handled by hoisting the state of all fields to a State Holder (like a ViewModel or custom class) and implementing validation logic and corresponding error state display there.
- **State Across Multiple Screens (Navigation):** State can be passed via Compose Navigation arguments (for simple data) or managed by shared ViewModels scoped to the Navigation Graph (`hiltViewModel()` or `navGraphViewModels()`) for more complex data or shared state between multiple screens.



Esmaeel Moradi

@esmaeel-moradi700

Side Effects in Compose (Revisited with State Focus):



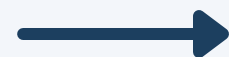
Sometimes we need to perform actions in response to state changes or a Composable's lifecycle that are not directly related to UI rendering (e.g., logging, network requests, registering a listener).

- **LaunchedEffect(key1 = stateVariable):** To launch a coroutine when stateVariable changes or the Composable first enters the Composition.
- **rememberCoroutineScope():** To get a CoroutineScope tied to the Composable's lifecycle, which can be used to launch coroutines in response to events (like button clicks).
- **rememberUpdatedState(value):** If you need the latest value of a state within a long-running effect (like LaunchedEffect) but don't want the effect to restart when that value changes, use this.
- **DisposableEffect(key1):** For effects that require cleanup when the Composable leaves the Composition (e.g., unregistering a listener).



Esmaeel Moradi
@esmaeel-moradi700

Saving and Restoring UI State (rememberSaveable):



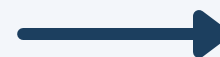
To preserve simple UI state across Activity/Process recreation (e.g., due to low memory or a configuration change not handled by ViewModel), use rememberSaveable instead of remember. rememberSaveable saves data in the Bundle.

- **When to Use?** For small, serializable UI states (like text in a TextField, a Switch's state, or scroll position). For more complex states or large data, ViewModel (which survives process death) remains the better solution as it's more suited for holding business logic.

```
// rememberSaveable example  
var query by rememberSaveable { mutableStateOf("") }  
TextField(value = query, onValueChange = { query = it })
```



Esmaeel Moradi
@esmaeel-moradi700



Testing Stateful Composables & State Holders:

- Stateless Composables (with hoisted state) are easily testable by passing different input values.
- For testing Composables using remember or rememberSaveable, use ComposeTestRule and Compose UI testing APIs.
- ViewModels and other state holder classes can be unit tested as regular Kotlin classes.



Esmaeel Moradi
@esmaeeil-moradi700



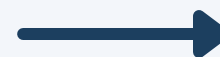
Performance Considerations Related to State:

- **Scope of Recomposition:** Try to limit states to the lowest necessary level in the Composable tree so that a state change causes Recomposition of only a small part of the UI. Proper State Hoisting helps with this.
- **Stability of Inputs:** If a Composable's inputs are "stable" (meaning the Compose compiler can determine if they've actually changed), Compose can skip unnecessary Recompositions. Data classes with only **val properties** of stable types are usually considered stable. For other types, you might need to use **@Stable** or **@Immutable** annotations as hints to the compiler (this is a more advanced topic).
- Avoid reading state in inappropriate phases (e.g., reading state in the Composition phase for tasks that should be done in the Layout or Draw phase).



Esmaeel Moradi

@esmaeel-moradi700



Conclusion:

Mastering State is Mastering Compose.

State management is, without a doubt, one of the most challenging yet critical aspects of building modern UIs with Jetpack Compose. While `remember` and `mutableStateOf` are good starting points, building real-world, complex, and maintainable applications requires a deep understanding of patterns like State Hoisting, proper use of state holders like `ViewModel`, and implementing Unidirectional Data Flow (UDF). Mastering these concepts is what separates a Compose app that merely works from a professional, high-performance, testable, and production-ready application. Embrace this learning journey, because every step you take in mastering state management brings you closer to becoming a true Compose artisan.



Esmaeel Moradi
@esmaeel-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



LEAVE A COMMENT BELOW



Esmaeeil Moradi

@esmaeeil-moradi700