📌 Snackbars, DrawerState & Coroutine Scopes in Jetpack Compose

---

1️⃣ UI Must Run on Main Thread

- In Android (including **Jetpack Compose**), **all UI updates must run on the main (UI) thread**.
- If you call UI APIs (like `SnackbarHostState.showSnackbar()` or `DrawerState.open()`) from a **background/worker thread**:
  - ❌ You'll get an exception (`Called on a wrong thread`)
  - ❌ Or the call may silently fail.

```
// ❌ Wrong: running on IO thread will crash or misbehave
scope.launch(Dispatchers.IO) {
    snackbarHostState.showSnackbar("Hi")
}
```

✅ **Rule:** Always touch UI from the **Main** thread.

2️⃣ Why We Need a Coroutine Scope

- `SnackbarHostState.showSnackbar()` and `DrawerState.open()` are **suspend functions**.
  → They cannot be called directly; they must run inside a **coroutine**.
- In Compose, we use a **CoroutineScope** to launch a coroutine on the main thread:

```
val scope = rememberCoroutineScope()

Button(onClick = {
    scope.launch {
        snackbarHostState.showSnackbar("Hello!")  // ✅ Safe
    }
})
```

- `scope.launch {}` is **main-safe** by default.
- This is the same reason you use a coroutine scope for:
  - **Snackbars** → `snackbarHostState.showSnackbar()`

- **Drawer actions** → `drawerState.open()` / `drawerState.close()`

---

3 Switching Threads with `withContext`

If you have to do **heavy work** first (e.g., network, DB, file I/O), do it in **IO**, then switch back to **Main** for UI:

```
val scope = rememberCoroutineScope()

Button(onClick = {
    scope.launch { // starts on Main
        val result = withContext(Dispatchers.IO) {
            heavyWork() // network, DB, file I/O
        }
        snackbarHostState.showSnackbar("Result: $result") // back on Main
    }
})
```

- `launch {}` → Main by default
- `withContext(Dispatchers.IO)` → temporarily switch to IO
- Resumes automatically on **Main** after the block.

⚡ If the work is **light and UI-only**, you **don't need** `withContext`.

---

4 Common Coroutine Scopes in Android

Scope	Owner / Lifetime	Default Dispatcher	Best For
`viewModelScope`	ViewModel (survives config changes)	`Dispatchers.Main`	Business/data work (API calls, DB, update UI state)
`lifecycleScope`	LifecycleOwner (Activity / Fragment)	`Dispatchers.Main`	UI tasks tied to Activity/Fragment lifecycle
`rememberCoroutineScope()`	A single Composable instance	`Dispatchers.Main`	UI actions (snackbar, animations, drawers)
`LaunchedEffect {}`	A single Composable instance (auto-launches)	`Dispatchers.Main`	Run suspend code once when Composable enters
`GlobalScope` ⚠️	Whole app process	—	❌ Avoid (lifecycle leaks)

🔷 ViewModelScope vs Compose Scope

- **`viewModelScope`**
  - Survives rotation/config changes.
  - Best for **long-lived tasks**: API calls, database, and updating UI state via `StateFlow` / `LiveData`.

- `rememberCoroutineScope / LaunchedEffect`
    - **Tied to a single Composable**.
    - Cancels when the Composable leaves the screen.
    - Ideal for **short-lived UI actions** like:
        - Showing a **Snackbar**
        - Opening/closing a **Navigation Drawer**
        - Running animations

5️⃣ Typical Safe Pattern

```
val scope = rememberCoroutineScope()

Button(onClick = {
    scope.launch(Dispatchers.IO) {
        val data = heavyWork() // Heavy I/O work
        withContext(Dispatchers.Main) {
            snackbarHostState.showSnackbar("Done: $data")
            drawerState.open() // Safe because now on Main
        }
    }
})
```

- Start in **IO** if your first work is heavy.
- Switch back to **Main** with `withContext(Dispatchers.Main)` for any UI updates.

✅ Best Practices Summary

- 🟢 **UI calls must run on Main** (`showSnackbar()` , `drawerState.open()` , etc.).
- 🟡 Use `withContext(Dispatchers.IO)` for heavy work, then return to Main.
- 🟢 For simple UI suspend calls, just `scope.launch { ... }` (already Main).
- 🟢 Pick the right scope:
    - **ViewModelScope** → long-lived, survives rotation, for data/business.
    - **rememberCoroutineScope / LaunchedEffect** → short-lived UI actions.
- 🔴 Avoid `GlobalScope` — can cause leaks & run after the UI is gone.

💡 Key Takeaway

> **Snackbars and Drawer actions are suspend functions** — they **require a coroutine scope** and must be called from the **Main thread**.

> Use `rememberCoroutineScope()` or `LaunchedEffect` for UI actions; use `viewModelScope` for long-running data tasks.