Flutter Hooks — Complete Rulebook

🎯 Flutter Hooks — Complete Rulebook

Flutter Hooks = **UI lifecycle & widget-local state manager**
They make UI code cleaner — but **must stay inside widgets only.**

🧱 1. Where Hooks Are Allowed

Layer	Hooks Allowed?	Reason
presentation/screens/	✅ YES	Hooks manage widget state & lifecycle
presentation/widgets/	✅ YES	For local animations, forms, etc.
presentation/controllers/	❌ NO	Controller is pure logic
providers/	❌ NO	Providers are stateless, pure functions
data/	❌ NO	Pure Dart files (API, repository)
domain/	❌ NO	Business logic, not UI
core/	❌ NO	Constants & utilities only

🧩 2. Base Rules of Using Hooks

Rule	Description
✅ Only inside a `HookWidget` or `HookConsumerWidget`	Hooks require widget lifecycle
❌ Never in pure Dart classes	No lifecycle context there
✅ Keep Hooks inside the `build` method	They rebuild with the widget
❌ Don’t call Hooks conditionally	Must always run same order every build
✅ Use Hooks for local UI state	animation, scroll, form, text fields, focus
❌ Don’t store global state in Hooks	Use Riverpod providers for that

💡 3. Common Hooks and Their Purpose

Hook	Purpose	Example
`useState<T>()`	Local reactive variable	`final counter = useState(0);`
`useEffect()`	Run side effects on mount/update	`useEffect(() { fetchData(); }, []);`
`useTextEditingController()`	Manage text fields	`final controller = useTextEditingController();`
`useAnimationController()`	Run animations	`final anim = useAnimationController(duration: 1s);`
`useScrollController()`	Control scroll behavior	`final scroll = useScrollController();`
`useFocusNode()`	Control focus	`final focus = useFocusNode();`
`useMemoized()`	Cache expensive values	`final data = useMemoized(() => heavyCalc());`
`useStream()`	Subscribe to a stream	`final data = useStream(myStream);`
`useFuture()`	Handle one-time async ops	`final snapshot = useFuture(fetch());`
`useRef()`	Keep stable object between rebuilds	`final counter = useRef(0);`

⚙️ 4. Hook Execution Rules

Hooks follow **strict order and lifecycle**:

Rule	Example	Why
✅ All Hooks must be called every build	`useState()`, `useEffect()` in same order	Flutter tracks Hook order
❌ Don't call Hooks in `if`, `for`, `switch`	`if (cond) useState()` ❌	Will break lifecycle
✅ Cleanup in `useEffect`	`return () => controller.dispose();`	Prevent memory leaks
✅ Depend on specific values	`useEffect(() { fetch(); }, [id]);`	Avoid unnecessary reruns

❌ Don't trigger rebuilds inside Hooks	Infinite rebuild loop risk	

🔁 5. `useEffect()` Lifecycle Rules

Type	Example	Behavior
On Mount	`useEffect(() { init(); return null; }, []);`	Runs once at widget load
On Value Change	`useEffect(() { load(); }, [id]);`	Runs when `id` changes
On Dispose	`useEffect(() { return () => controller.dispose(); }, []);`	Cleanup
❌ No async directly in useEffect	`useEffect(() async { ... });` ❌	Use inside normal function

✅ Correct pattern:

```
useEffect(() {
  Future.microtask(() async {
    await ref.read(userControllerProvider.notifier).fetchUser();
  });
  return null;
}, []);
```

📦 6. Hooks vs Providers — Clear Boundary

Task	Use Hook	Use Provider
UI animation	✅	❌
Text field controller	✅	❌
Simple toggle (widget-only)	✅	❌
Global theme toggle	❌	✅
Fetch data from API	❌	✅
Manage login state	❌	✅
Keep global counter	❌	✅

Handle UI scroll position	✅	❌

🚫 7. Common Mistakes with Hooks

Mistake	Why Wrong	Fix
Calling Hooks in a function outside `build`	Breaks lifecycle	Move inside `build`
Calling Hooks conditionally	Reorder error	Always call unconditionally
Using Hook to fetch API directly	Logic in wrong layer	Use provider/controller
Forgetting cleanup in `useEffect`	Memory leak	Return cleanup callback
Using `setState()` with Hooks	Conflicting states	Use `useState()` instead
Using `useEffect()` to rebuild UI repeatedly	Infinite rebuilds	Add proper dependency array
Reading `ref.watch()` inside `useEffect()` incorrectly	Might cause loop	Use `ref.listen()` or check dependencies

🎮 8. Example — Hooks in Correct Context

✅ **Good usage:**

```
class LoginScreen extends HookConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final email = useTextEditingController();
    final password = useTextEditingController();
    final isVisible = useState(false);

    ref.listen(authControllerProvider, (_, next) {
      next.whenOrNull(
        data: (_) => ScaffoldMessenger.of(context)
            .showSnackBar(const SnackBar(content: Text("Success"))),
        error: (e, _) => ScaffoldMessenger.of(context)
            .showSnackBar(SnackBar(content: Text(e.toString()))),
      );
    });

    return Column(
      children: [
        TextField(controller: email),
        TextField(controller: password),
```

```
        ElevatedButton(
          onPressed: () => ref
              .read(authControllerProvider.notifier)
              .login(email.text, password.text),
          child: const Text('Login'),
        ),
      ],
    );
  }
}
```

🚫 **Wrong usage:**

```
class AuthController extends StateNotifier {
  final emailController = useTextEditingController(); // ❌
  ...
}
```

❌ **Why wrong:** Controllers and data layers are not part of the widget tree — no lifecycle → can't use Hooks.

🧠 9. Hooks Lifecycle Summary

Lifecycle	Hook	
Widget created	`useEffect(() {}, [])` runs	
Widget rebuilt	Hooks re-evaluate in same order	
Widget removed	Cleanup in `useEffect` runs	
Hook order changes	Error: "Hook order mismatch"	

📋 10. Best Practice Summary

Category	Rule
Scope	Hooks are widget-local only
Placement	Must be inside HookWidget build()
Structure	Call in same order, no conditions
Lifecycle	Use `useEffect` for init/dispose
Cleanup	Always dispose controllers/focus

Logic Separation	API → providers, not hooks
Global State	Providers, not hooks
Performance	Use `useMemoized()` for heavy calcs
Safety	Don’t use async directly in hooks
Testing	Hooks only in UI, test logic separately

🧩 11. Final Golden Rules (Remember These 5)

1. **Hooks live only in widgets**
2. **Hooks manage only UI state or lifecycle**
3. **Hooks must never call APIs or repositories**
4. **Never use Hooks conditionally or outside build()**
5. **Use Riverpod for logic and shared/global states**