

Practical Android

A HANDS-ON GUIDE

Practical Android

Kotlin, Jetpack Compose and Clean Architecture

Author Name
with Co-Author Name

PUBLISHER IMPRINT

Practical Android: Kotlin, Jetpack Compose and Clean Architecture

Copyright © 2026 Author Name and Co-Author Name. All rights reserved.

No part of this book may be reproduced in any form without written permission from the authors, except for brief quotations in reviews.

Android is a trademark of Google LLC. Kotlin is a trademark of the Kotlin Foundation. Other names are the property of their respective owners and are used for identification only.

The code in this book is provided for teaching purposes, as is, without warranty.

Contact the authors

Author Name — author@example.com

Co-Author Name — coauthor@example.com

ISBN (paperback): 978-0-000000-00-0

ISBN (hardcover): 978-0-000000-01-7

Interior typeset in EB Garamond, Space Grotesk and JetBrains Mono (SIL Open Font License).

Published through Notion Press · HARDCOVER edition · Printed in India

Contents

Introduction	I
How to read this book	I
I State and Recomposition in Jetpack Compose	3
I.1 What counts as state	3
I.2 remember and rememberSaveable	4
I.3 State hoisting	5
I.4 Moving state into a ViewModel	6
I.5 Summary	8

Introduction

Android has changed more in the last five years than in the decade before it. Views and XML layouts gave way to Jetpack Compose; callbacks gave way to coroutines and flows; and the question of where state should live moved from an afterthought to the centre of every design decision.

This book is written for developers who already know a little Kotlin and want to build Android apps the way current teams build them. Every chapter is organised around one running example, a small water-tracking app called *Hydrate*, which grows from a single screen into a tested, multi-module project.

How to read this book

Chapters build on each other, but each one starts with a short recap so you can jump in where you need to. Code listings are complete and compile against the versions listed in Appendix A. Where a listing shows only part of a file, the omitted lines are marked with a comment.

CONVENTIONS

Kotlin code is set in a monospaced font with line numbers. File names appear above each listing. Terms are set in *italics* the first time they are defined.

State and Recomposition in Jetpack Compose

A Compose screen is a function of its state. When the state changes, Compose calls the affected functions again and redraws only what changed. This process is called *recomposition*, and understanding it is the difference between an app that feels instant and one that stutters, loses input, or forgets what the user was doing.

In this chapter we turn the static Hydrate screen from Chapter 2 into a live one. By the end you will know where each piece of state belongs, how to keep it across configuration changes, and how to move it into a `ViewModel` when it grows beyond a single screen.

1.1 What counts as state

State is any value that can change over time and that the UI reads to decide what to show. In Hydrate, the number of glasses drunk today is state; so is the selected glass size, and so is whether a network request is in flight. A constant such as the daily goal of eight glasses is not state, unless the user can edit it.

Compose observes state through the `State<T>` type. When a composable reads `state.value`, Compose records that read. When the value later changes, every composable that read it is scheduled for recomposition, and nothing else is.

```

UI/TODAYSCREEN.KT
1  @Composable
2  fun GlassCounter() {
3      var glasses by remember { mutableIntStateOf(0) }
4
5      Column(horizontalAlignment = Alignment.CenterHorizontally)
6      {
7          Text(
8              text = "$glasses of 8 glasses",
9              style = MaterialTheme.typography.headlineMedium,
10             )
11         Button(onClick = { glasses++ }) {
12             Text("Add glass")
13         }
14     }

```

Line 3 does two things. `mutableIntStateOf(0)` creates an observable integer, and `remember` keeps that object alive across recompositions, so the counter does not reset to zero every time the function runs. Without `remember`, each recomposition would create a fresh state object and the button would appear to do nothing.

1.2 remember and rememberSaveable

`remember` survives recomposition, but not a configuration change. Rotate the device and the activity is recreated; everything stored with `remember` is lost. For values the user would expect to keep, use `rememberSaveable`, which writes the value into the saved-instance-state bundle.

```

UI/TODAYSCREEN.KT
1  var glasses by rememberSaveable {
2      mutableIntStateOf(0)
3  }
4  var glassSize by rememberSaveable {
5      mutableStateOf(GlassSize.Medium)
6  }

```

Figure 1.1 shows the result: the same screen before and after rotation, with the count intact.

`rememberSaveable` can only store types that fit in a `Bundle`: primitives, strings, `Parcelable` and `Serializable` objects. For anything else, supply a custom `Saver`. Table 1.1 summarises the three places state usually lives.

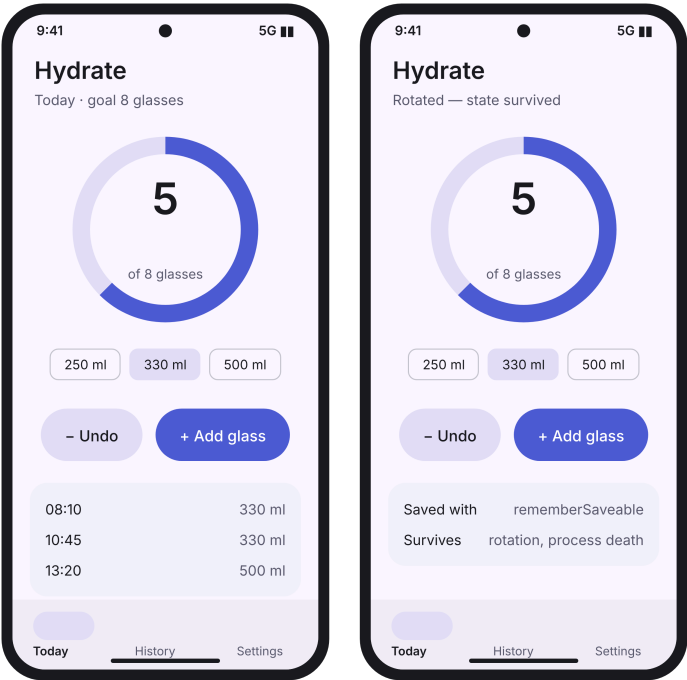


FIGURE 1.1 Hydrate before and after a configuration change. Because the counter uses `rememberSaveable`, the value 5 survives rotation and process death.

TABLE 1.1 Where to keep UI state, and what each option survives.

Holder	Recomposition	Rotation	Process death
<code>remember</code>	Yes	No	No
<code>rememberSaveable</code>	Yes	Yes	Yes (Bundle)
<code>ViewModel</code>	Yes	Yes	With <code>SavedStateHandle</code>
Repository / DB	Yes	Yes	Yes

1.3 State hoisting

A composable that owns its state is easy to write but hard to reuse and test. *State hoisting* moves the state up to the caller and replaces it with two parameters: the current value, and an event callback. The composable becomes *stateless*: it shows what it is given and reports what happened.

```

UI/GLASSCOUNTER.KT
1  @Composable
2  fun GlassCounter(
3      glasses: Int,
4      goal: Int,
5      onAddGlass: () → Unit,
6      modifier: Modifier = Modifier,
7  ) {
8      Column(modifier, horizontalAlignment =
9          Alignment.CenterHorizontally) {
10         ProgressRing(progress = glasses / goal.toFloat())
11         Text("$glasses of $goal glasses")
12         Button(onClick = onAddGlass) { Text("Add glass") }
13     }
14 }

```

The pattern produces a *unidirectional data flow*: state travels down the tree, events travel up. Figure 1.2 shows the loop for Hydrate.

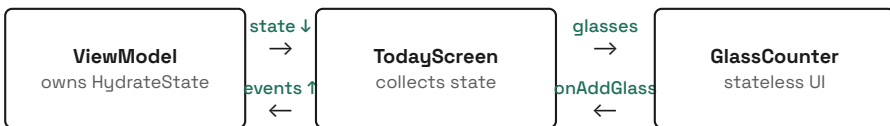


FIGURE 1.2 Unidirectional data flow in Hydrate. State moves toward the UI; events move back to the owner, which is the only place state changes.

TIP

Hoist state to the lowest common ancestor of every composable that reads it. Hoisting higher than that makes recomposition wider than it needs to be.

1.4 Moving state into a ViewModel

Once state is shared by several screens, or depends on data from disk or network, it belongs in a `ViewModel`. The `ViewModel` exposes a read-only `StateFlow`, and the screen collects it with lifecycle awareness so that no work happens while the app is in the background.

```

TODAY/TODAYVIEWMODEL.KT
1  data class HydrateState(
2      val glasses: Int = 0,
3      val goal: Int = 8,
4      val glassSize: GlassSize = GlassSize.Medium,
5  )
6
7  class TodayViewModel(
8      private val repository: IntakeRepository,
9  ) : ViewModel() {
10
11      val state: StateFlow<HydrateState> = repository.today()
12          .map { log → HydrateState(glasses = log.count, goal =
log.goal) }
13          .stateIn(
14              scope = viewModelScope,
15              started = SharingStarted.WhileSubscribed(5_000),
16              initialValue = HydrateState(),
17          )
18
19      fun onAddGlass() = viewModelScope.launch {
20          repository.addGlass(state.value.glassSize)
21      }
22  }

```

`SharingStarted.WhileSubscribed(5_000)` keeps the upstream flow alive for five seconds after the last collector disappears. That is long enough to survive a rotation without restarting the database query, and short enough that nothing runs while the app sits in the background.

```

TODAY/TODAYSCREEN.KT
1  @Composable
2  fun TodayScreen(viewModel: TodayViewModel = hiltViewModel()) {
3      val state by viewModel.state.collectAsStateWithLifecycle()
4      GlassCounter(
5          glasses = state.glasses,
6          goal = state.goal,
7          onAddGlass = viewModel::onAddGlass,
8      )
9  }

```

Common mistakes

- Exposing `MutableStateFlow` publicly, which lets any caller change state and breaks the one-owner rule.
- Calling `collectAsState()` instead of `collectAsStateWithLifecycle()`, which keeps collecting while the app is stopped.

- Creating objects inside a composable without `remember`, which allocates on every recomposition.

1.5 Summary

State drives everything a Compose screen shows. Keep it in `remember` for throw-away UI details, `rememberSaveable` for values the user expects to keep, and a `ViewModel` once it is shared or comes from data. Hoist state so composables stay stateless, and let events flow back to a single owner. In Chapter 2 we connect `IntakeRepository` to `Room` and write the first tests for `TodayViewModel`.