08 · Widget Lifecycle

Flutter Practice Notes

Contents

08 · Widget Lifecycle

The `State` Lifecycle (Overview)

`initState` vs `didChangeDependencies`

`didUpdateWidget` (Reacting to New Config)

`setState` Mechanics & Pitfalls

`dispose` & Leak Prevention

App Lifecycle (`AppLifecycleState` , `WidgetsBindingObserver`)

08 · Widget Lifecycle

Introduction

A `StatefulWidget`'s `State` object goes through a well-defined lifecycle: created, initialized, built, updated, and disposed. Knowing *which method runs when* — and what belongs in each — is what separates correct, leak-free Flutter code from the buggy kind. This module also covers **app-level** lifecycle (foreground/background).

Why this module exists

Most Flutter bugs — leaks, "setState after dispose", "no initialized value", stale subscriptions, work done at the wrong time — are lifecycle mistakes. This module makes the sequence and the responsibilities of each method precise, building on the three-trees model (Module 06).

Module map

#	File	Topic	Level
1	01_state_lifecycle_overview.md	The full `State` lifecycle sequence	🔵
2	02_initstate_and_dependencies.md	`initState` vs `didChangeDependencies`	🔵
3	03_didupdatewidget.md	Reacting to new widget config	🔵
4	04_setstate_mechanics.md	How `setState` works + pitfalls	🟢
5	05_dispose_and_leaks.md	`dispose`, cleanup, leak prevention	🔴
6	06_app_lifecycle.md	`AppLifecycleState`, `WidgetsBindingObserver`	🔵

> **Cross-references:** Three trees / state persistence: Module 06. Memory/leaks: 02 · memory_and_gc. Streams/subscriptions: 02 · streams. Controllers/forms: 07 · input_and_forms. State management alternatives to raw `setState`: Module 11.

Prerequisites

06 Flutter Fundamentals (three trees, stateless/stateful, `BuildContext`) and 02 Advanced Dart (streams, memory/GC).

What you'll be able to do after this module

- Recite the `State` lifecycle and put each responsibility in the right method.
- Choose `initState` vs `didChangeDependencies` correctly.

- React to config changes via `didUpdateWidget`.
- Use `setState` safely (no post-dispose calls, no async pitfalls).
- Prevent leaks by disposing controllers/subscriptions/timers.
- Respond to app foreground/background transitions.

Capstones

Tier	Build
Beginner	A timer widget that starts in `initState` and cancels in `dispose`.
Intermediate	A widget subscribing to a stream, resubscribing on `didUpdateWidget`.
Advanced	A screen that pauses/resumes work on app background/foreground.
Enterprise	A `LifecycleAwareMixin` that standardizes subscription/controller cleanup.

Summary

The lifecycle is a contract: initialize in `initState`, react to inherited data in `didChangeDependencies`, react to config in `didUpdateWidget`, rebuild via `setState`, and **always clean up in `dispose`**. Master it and a whole class of bugs disappears.

The `State` Lifecycle (Overview)

> A `State` object is created once, initialized (`initState`), built many times (`build`), updated on config/dependency changes (`didUpdateWidget` / `didChangeDependencies`), and finally torn down (`deactivate` → `dispose`). Each method has a specific job.

Introduction

This file lays out the full lifecycle sequence and what each callback is for, so the detailed files (init, update, dispose) have a map. The `State` object persists in the Element tree across rebuilds while the `StatefulWidget` config is recreated (Module 06).

Why this concept exists

Flutter needs defined hook points so you can initialize resources, react to changes, and release resources at the right moments. This is a **Template Method** pattern (05 · template_method): the framework calls fixed lifecycle steps in order; you override the ones you need.

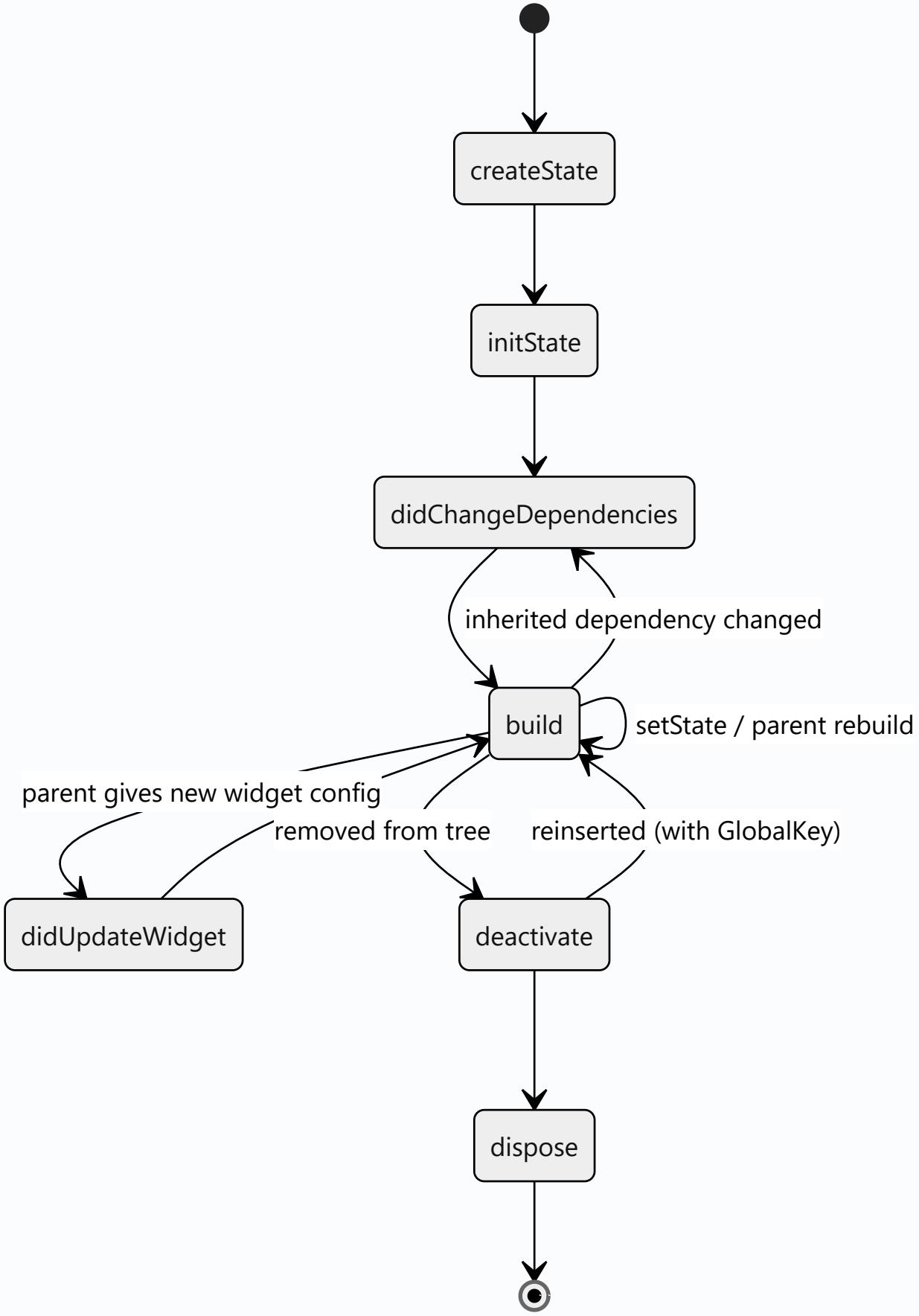
Real-world analogy

An **employee's tenure**: hired (`createState`), onboarded once (`initState`), does daily work repeatedly (`build`), adapts when their manager/role changes (`didUpdateWidget` / `didChangeDependencies`), and off-boarded with equipment returned (`dispose`). Skipping off-boarding (not disposing) = keycards still active = a leak.

Problem Statement

Where do you start a timer, read an inherited theme, react to a changed `userId` prop, and cancel a subscription? By the end you'll place each in the correct lifecycle method.

Internal Working



Method	Called	Use for
`createState()`	Once, when the element is created	Return the `State` instance
`initState()`	Once, after `createState`	One-time init: controllers, subscriptions (no `context` inherited lookups needing dependencies)
`didChangeDependencies()`	After `initState`, and when an inherited dependency changes	React to `InheritedWidget` data (theme, provider)
`build()`	Every rebuild	Return the UI (pure)
`didUpdateWidget(old)`	When the parent rebuilds this widget with a new config	React to changed widget properties
`setState()`	You call it	Mark dirty → schedule rebuild
`deactivate()`	When removed from the tree (may be reinserted)	Rare; temporary removal
`dispose()`	Once, when permanently removed	Release: cancel subscriptions/timers, dispose controllers
`mounted`	property	True between `initState` and `dispose`; guard async

Memory Representation

The `State` lives with its Element (persistent across rebuilds); it holds your controllers/subscriptions until `dispose` (02 · memory_and_gc).

Compiler Behavior

Not applicable.

Runtime Behavior

The framework invokes these in order; `build` runs frequently, init/dispose exactly once. Calling `setState` after `dispose` throws.

Flutter Engine Behavior

Lifecycle drives the build phase feeding the render pipeline (Module 09).

Dart VM Behavior

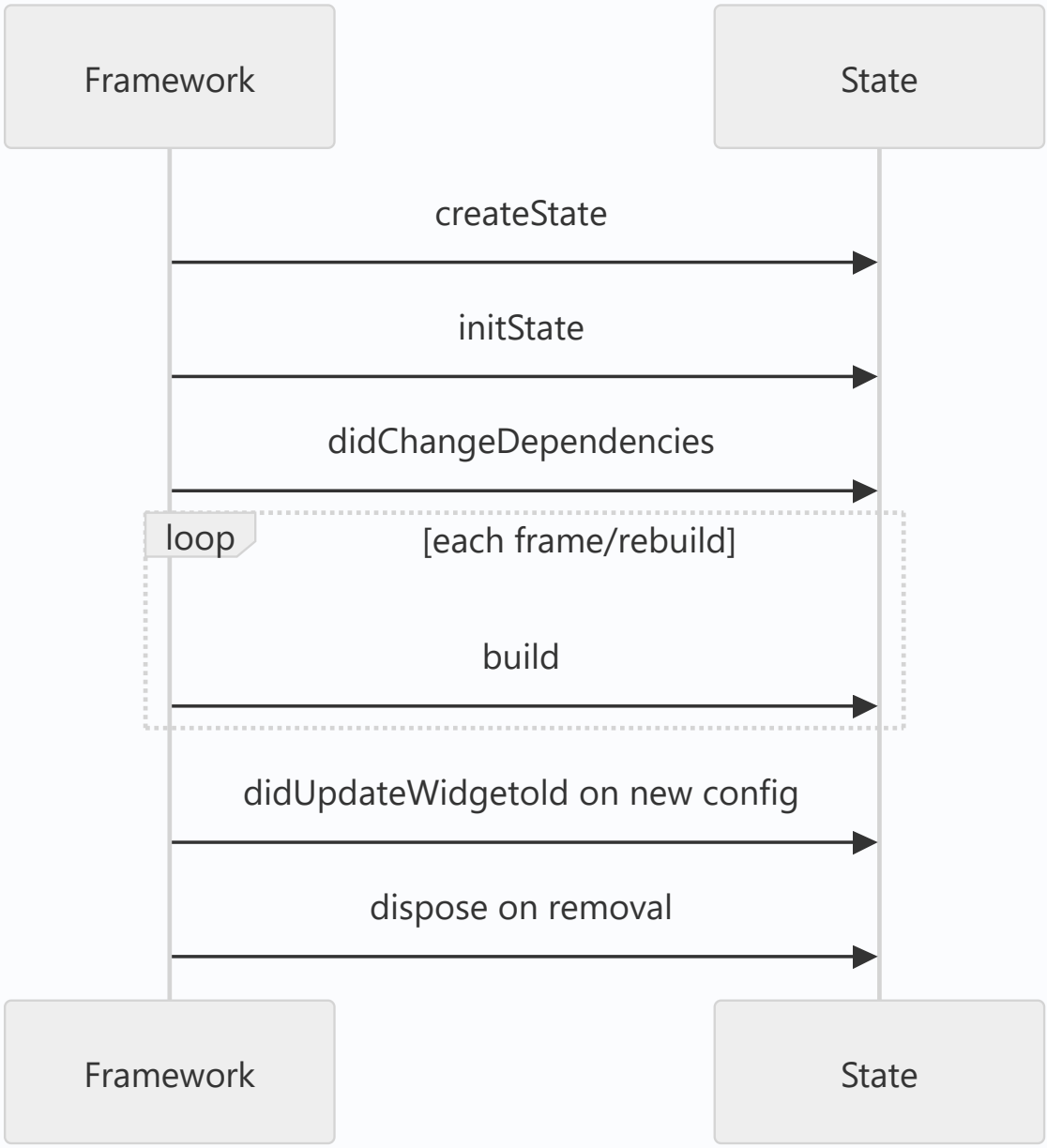
Not applicable.

Examples

```
import 'dart:async';
import 'package:flutter/material.dart';

class Ticker extends StatefulWidget {
  final int intervalSeconds;
  const Ticker({super.key, required this.intervalSeconds});
  @override
  State<Ticker> createState() => _TickerState();
}

class _TickerState extends State<Ticker> {
  int _ticks = 0;
  Timer? _timer;

  @override
  void initState() {
    super.initState();
    _start(); // one-time init
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    // react to inherited data if needed (e.g., Theme/Localizations/Provider)
  }

  @override
  void didUpdateWidget(covariant Ticker oldWidget) {
    super.didUpdateWidget(oldWidget);
    if (oldWidget.intervalSeconds != widget.intervalSeconds) {
      _timer?.cancel();
      _start(); // react to changed config
    }
  }

  void _start() {
    _timer = Timer.periodic(Duration(seconds: widget.intervalSeconds), (_) {
      if (mounted) setState(() => _ticks++); // guard + rebuild
```

```
    });
  }

  @override
  void dispose() {
    _timer?.cancel(); // ALWAYS clean up
    super.dispose();
  }

  @override
  Widget build(BuildContext context) => Text('Ticks: $_ticks');
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Inherited lookups needing dependencies in `initState`	Dependencies not ready	Use `didChangeDependencies`
Forgetting `super.initState()` / `super.dispose()`	Framework invariants break	Always call super
Not disposing resources	Leaks	Cancel/dispose in `dispose`
`setState` after `dispose`	Throws	Guard with `mounted`
Heavy work in `build`	Runs often	Do it in `initState`/handlers

Best Practices

- Initialize once in `initState`; read inherited data in `didChangeDependencies`.
- React to prop changes in `didUpdateWidget` (compare old vs new).
- Keep `build` pure; trigger rebuilds via `setState`/state management.
- **Always** release in `dispose`; guard async with `mounted`.
- Call `super.*` in overridden lifecycle methods.

Performance

`build` frequency is fine if cheap; do expensive setup once in `initState` and cache. Leaks from missing `dispose` degrade the app over time (Module 21).

Advantages / Disadvantages

- + Precise hooks for init/react/cleanup; predictable order; enables correct resource management.
- − Easy to misuse (wrong method, missing dispose); more to manage than stateless.

Interview Questions

1. 🟢 **Outline the `State` lifecycle.** — `createState` → `initState` → `didChangeDependencies` → `build` (repeated), `didUpdateWidget` on new config, `deactivate` → `dispose` on removal.
2. 🟢 **What runs once vs many times?** — `initState`/`dispose` once; `build` many times; `didChangeDependencies`/`didUpdateWidget` on relevant changes.
3. 🟡 **Why not do inherited-widget lookups in `initState`?** — Dependencies aren't registered yet; do them in `didChangeDependencies`.
4. 🟡 **What is `mounted` and why guard with it?** — True while the `State` is in the tree; guarding prevents `setState`/context use after `dispose` (async callbacks).
5. 🟡 **Which pattern does the lifecycle resemble?** — Template Method: fixed framework-called steps you override.
6. 🔴 **When is `didUpdateWidget` called and why compare old vs new?** — When the parent rebuilds this widget with a new config; compare to react only to actual changes (e.g., re-subscribe when an id changed).
7. 🔴 **What's the difference between `deactivate` and `dispose`?** — `deactivate` = temporarily removed (may be reinserted, e.g., with a `GlobalKey`); `dispose` = permanent teardown.

Senior Engineer Tips

- Memorize "init once, react in the two `did*` methods, clean up in `dispose`."

- Pair every resource acquisition with its release in the same file (create in `initState`, free in `dispose`).
- Guard every async callback that touches state/context with `if (!mounted) return;`.

Architect Perspective

The lifecycle is the local resource-management contract; standardizing it (base classes/mixins for subscription cleanup, lint rules) prevents leaks at scale (02 · memory_and_gc). It's also why business logic belongs in state management (Module 11) — so widget lifecycle handles only UI resources.

Summary

- `State`: created once, `initState` once, `build` many, `did*` on changes, `dispose` once.
- Put each responsibility in the right method; always clean up; guard async with `mounted`.
- It's Template Method — override the hooks you need, call `super`.

Revision Notes

- Order: createState → initState → didChangeDependencies → build (×N); didUpdateWidget (new config); dispose (removal).
- Init once (`initState`), inherited data (`didChangeDependencies`), config changes (`didUpdateWidget`), cleanup (`dispose`).
- `mounted` guards async; call `super.*`; keep `build` pure.

Practice Questions

1. Where do you start a timer and where cancel it?
2. Why read `Theme.of(context)` in `didChangeDependencies` not `initState`?
3. Difference between `deactivate` and `dispose`?

Coding Questions

1. Build a widget logging each lifecycle method to observe the order.
2. Start/cancel an animation controller across `initState` / `dispose`.
3. Guard an async fetch's `setState` with `mounted`.

Mini Project

Lifecycle logger (Flutter): Build a `StatefulWidget` that prints each lifecycle callback, toggles its own config from a parent (to trigger `didUpdateWidget`), and manages a timer (start in `initState`, cancel in `dispose`). Acceptance: correct order observed; timer cancelled; `mounted`-guarded; `super.*` called.

`initState` vs `didChangeDependencies`

> Use `initState` for one-time setup that doesn't need inherited data; use `didChangeDependencies` for setup that reads `InheritedWidget`s (theme, provider, media query) — because those dependencies aren't ready in `initState`.

Introduction

Both run early, but they differ in *when* and *why*. `initState` is a single one-time hook; `didChangeDependencies` runs right after `initState` **and again** whenever an inherited dependency changes. Picking correctly avoids "dependencies not ready" bugs and stale data.

Why this concept exists

`InheritedWidget` dependencies (registered via `context.dependOn...`) aren't wired up when `initState` runs, so lookups there are unsafe/incomplete. `didChangeDependencies` exists as the safe place to read them — and to *re-read* when they change.

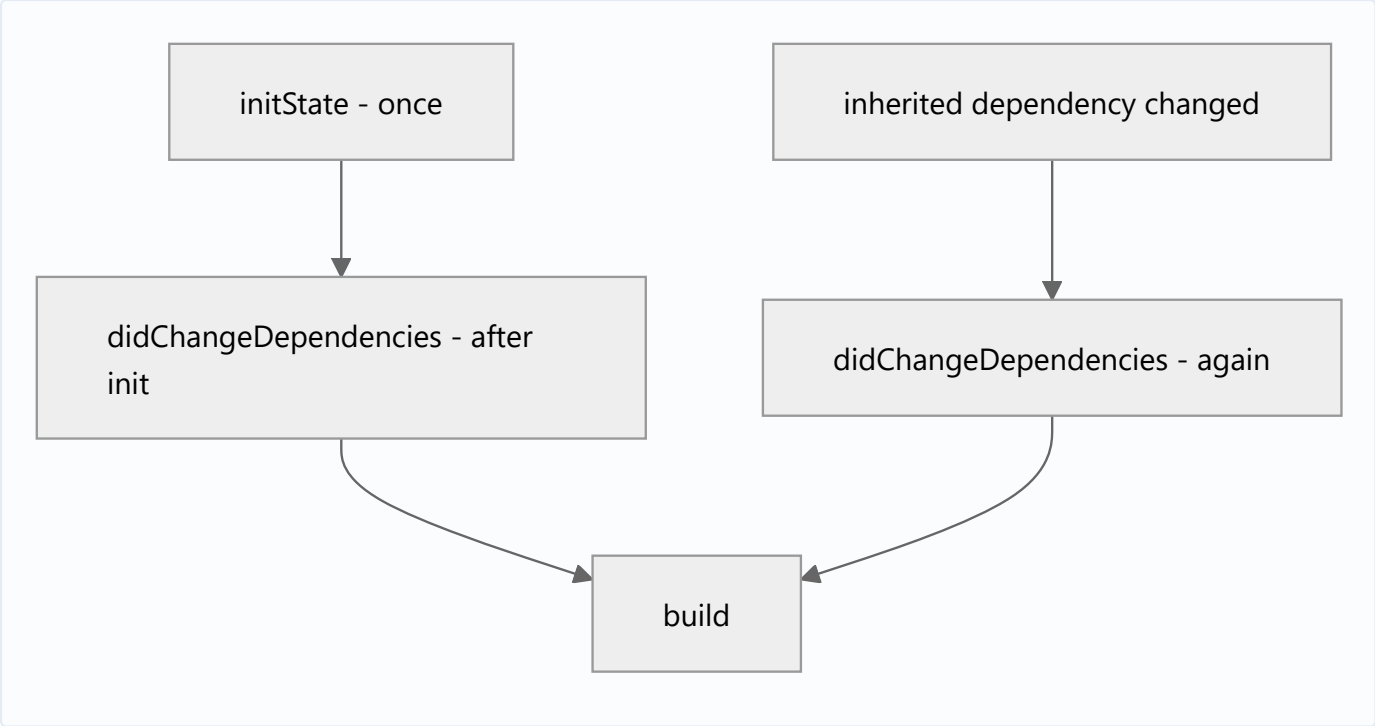
Real-world analogy

`initState` is **unpacking your own boxes** when you move in (things you brought). `didChangeDependencies` is **checking the building's shared utilities** (water pressure, electricity) — which you can only do once you're connected, and must recheck if the utility company changes something.

Problem Statement

You need to (a) create an `AnimationController` once, and (b) load data that depends on a `Locale` / `Provider` value, re-loading if that value changes. You'll put (a) in `initState` and (b) in `didChangeDependencies`.

Internal Working



- `initState`: called **once**, before dependencies are available. Safe: create controllers/timers, subscribe to non-context streams, init fields. Unsafe: `dependOnInheritedWidgetOfExactType` (theme/provider) that registers a dependency.
- `didChangeDependencies`: called **immediately after** `initState`, and **again** whenever an inherited widget this `State` depends on changes. Safe place for `Theme.of`, `MediaQuery.of`, `Provider.of(listen:true)`, `Localizations.of`, and reacting to their changes.
- `context` is usable in both, but *inherited-dependency* lookups belong in `didChangeDependencies`.

Memory Representation

Resources created here live in the `State` until `dispose` (05_dispose_and_leaks.md).

Compiler Behavior

Not applicable.

Runtime Behavior

`didChangeDependencies` can run multiple times; make it **idempotent** (don't leak by re-subscribing without cleaning up the previous subscription).

Flutter Engine Behavior / Dart VM Behavior

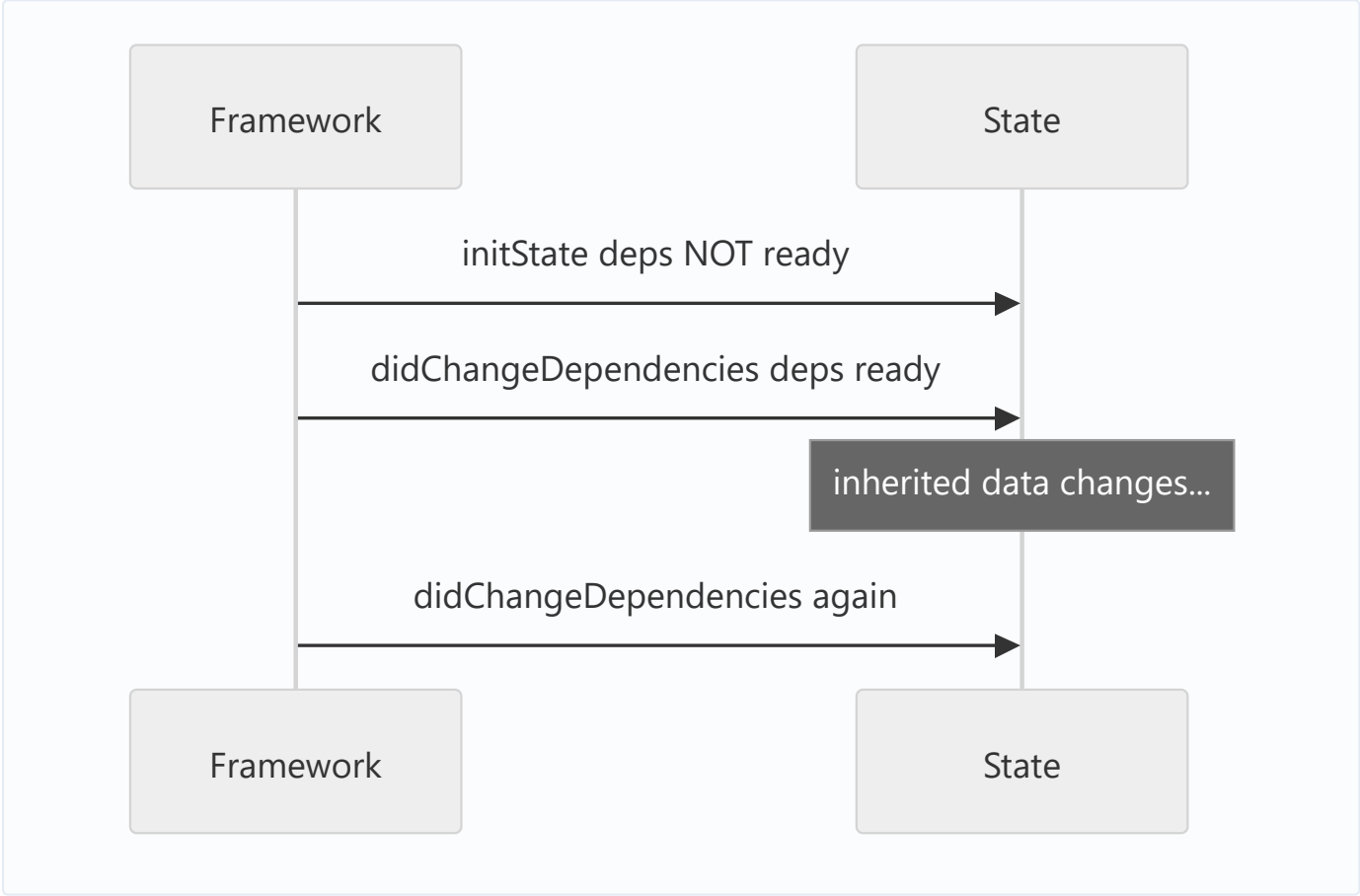
Not applicable.

Examples

```
import 'package:flutter/material.dart';

class LocalizedGreeting extends StatefulWidget {
  const LocalizedGreeting({super.key});
  @override
  State<LocalizedGreeting> createState() => _LocalizedGreetingState();
}

class _LocalizedGreetingState extends State<LocalizedGreeting>
    with SingleTickerProviderStateMixin {
  late final AnimationController _controller; // needs `this` (a TickerProvider)
  Locale? _locale;
  String _greeting = '';

  @override
  void initState() {
    super.initState();
    // one-time setup that does NOT need inherited data:
    _controller = AnimationController(vsync: this, duration: const Duration(seconds: 1))
      ..forward();
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    // reads inherited data; re-runs if the Locale changes:
    final newLocale = Localizations.localeOf(context);
    if (newLocale != _locale) {
      _locale = newLocale;
      _greeting = _greetingFor(newLocale); // reload based on dependency
    }
  }

  String _greetingFor(Locale l) => l.languageCode == 'fr' ? 'Bonjour' : 'Hello';

  @override
  void dispose() {
    _controller.dispose(); // release
```

```
    super.dispose();
  }

  @override
  Widget build(BuildContext context) => Text(_greeting);
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`Theme.of(context)` / `Provider.of` (listen) in `initState`	Dependencies not registered yet	Move to `didChangeDependencies`
Assuming `didChangeDependencies` runs once	It re-runs on dependency change	Make it idempotent (diff before acting)
Re-subscribing without cleanup in `didChangeDependencies`	Leaks/duplicate subscriptions	Cancel old before creating new
Heavy unconditional work each call	Re-runs waste work	Guard with a change check

Best Practices

- One-time, context-free setup → `initState`.
- Inherited-data reads and reactions → `didChangeDependencies`, made **idempotent** (compare old vs new before re-doing work).
- If you re-subscribe on dependency change, cancel the previous subscription first.
- Use `initState` for `AnimationController` (needs `vsync: this`), fields, and non-context subscriptions.

Performance

`didChangeDependencies` may run several times; guard expensive work behind a change check to avoid redundant reloads (Module 21).

Advantages / Disadvantages

- + Clear separation: own setup vs inherited-data setup; supports reacting to dependency changes.
- − Subtle timing rules; idempotency required; easy to misuse `initState` for context lookups.

Interview Questions

1. 🟢 **Difference between `initState` and `didChangeDependencies`?** — `initState` runs once before dependencies are ready (own setup); `didChangeDependencies` runs after it and again on inherited-dependency changes (context-dependent setup).
2. 🟢 **Why can't you call `Theme.of(context)` (as a dependency) in `initState`?** — The inherited-widget dependencies aren't registered yet; use `didChangeDependencies`.
3. 🟡 **Why must `didChangeDependencies` be idempotent?** — It can run multiple times; unconditional work leaks/duplicates. Diff before acting.
4. 🟡 **Where do you create an `AnimationController` and why?** — In `initState` (with `vsync: this`); it's one-time, context-free setup.
5. 🟡 **How do you react to a changed inherited value (e.g., Locale)?** — In `didChangeDependencies`, compare new vs stored and reload if different.
6. 🔴 **What triggers a re-run of `didChangeDependencies`?** — A change in an `InheritedWidget` this `State` depends on (e.g., theme/locale/provider it read with `listen:true`).
7. 🔴 **If you subscribe to a provider stream, where and how to avoid duplicates?** — Subscribe in `didChangeDependencies`, cancelling any previous subscription first; cancel finally in `dispose`.

Senior Engineer Tips

- Rule of thumb: "needs `context` inherited data? → `didChangeDependencies`. Otherwise → `initState`."
- Guard `didChangeDependencies` work with a stored-previous-value comparison to avoid redundant reloads.
- Prefer state management (Module 11) for data loading; these hooks are for wiring, not business logic.

Architect Perspective

Correctly separating own-setup from dependency-driven setup keeps widgets robust to theme/locale/provider changes and avoids leaks — important for reactive, localized, themable apps. Standardizing idempotent dependency handling (or pushing it to state management) reduces subtle lifecycle bugs across a large codebase.

Summary

- `initState`: one-time, context-free setup (controllers, fields, non-context subscriptions).
- `didChangeDependencies`: read/react to inherited data; runs again on changes — keep it idempotent.
- Cancel-before-resubscribe; dispose everything in `dispose`.

Revision Notes

- `initState` once, deps NOT ready → own setup (`AnimationController`, fields).
- `didChangeDependencies` after init + on inherited change → `Theme/Provider/Locale`; idempotent.
- Re-subscribe? cancel previous first. Dispose in `dispose`.

Practice Questions

1. Why does `didChangeDependencies` exist separately from `initState`?
2. What makes `didChangeDependencies` need to be idempotent?
3. Where to create an `AnimationController` and why?

Coding Questions

1. Load data based on a `Provider` value and reload when it changes (idempotent).
2. Create an `AnimationController` in `initState` and dispose it.
3. Log both methods to observe when each fires as a dependency changes.

Mini Project

Locale-aware panel (Flutter): Build a widget that creates a controller in `initState` and loads a locale-dependent message in `didChangeDependencies`, reloading only when the locale actually changes, with proper disposal. Acceptance: correct method placement; idempotent reload; no leaks; app runs.

`didUpdateWidget` (Reacting to New Config)

> When a parent rebuilds and gives this `State` a **new widget instance** of the same type, `didUpdateWidget(oldWidget)` runs — the place to react to *changed properties* (re-subscribe, restart animations, refetch).

Introduction

Because `State` persists while its `StatefulWidget` config is recreated each build (Module 06), the `State` must be told when its configuration changed. `didUpdateWidget(oldWidget)` is that hook: compare `oldWidget` to the current `widget` and react to differences.

Why this concept exists

A parent may pass a new `userId`, `url`, `duration`, or `controller` to the same widget. The `State`'s existing resources (subscriptions, animations) were set up for the *old* values; `didUpdateWidget` lets you tear down and re-setup for the new ones — otherwise you'd show stale data.

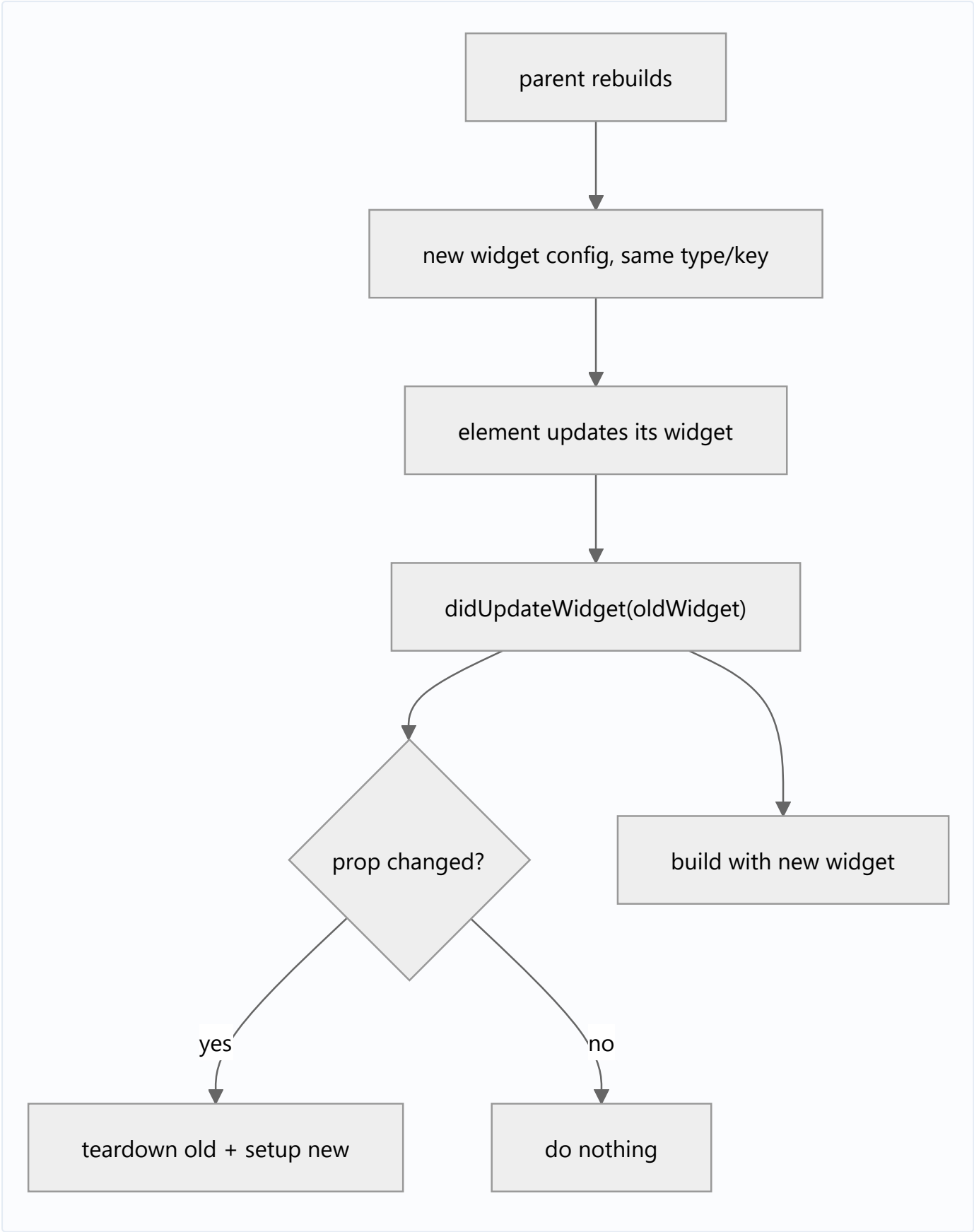
Real-world analogy

A **rented apartment with the same tenant but a new lease**: the tenant (State) stays, but the lease terms (widget config) changed. `didUpdateWidget` is reading the new lease and adjusting (new parking spot? re-register the car).

Problem Statement

A `UserAvatar(userId)` subscribes to a user's profile stream. When the parent passes a different `userId`, you must unsubscribe from the old and subscribe to the new. You'll do it in `didUpdateWidget`.

Internal Working



- Called when the framework updates the element with a **new widget of the same runtimeType (and key)** — i.e., reconciliation reused this element (Module 06).
- `widget` already refers to the **new** config inside `didUpdateWidget`; the parameter is the **old** one.

- Always call `super.didUpdateWidget(oldWidget)`.
- Compare specific fields; react only to real changes. Follows immediately by a `build`.

Memory Representation

You typically swap resources (cancel old subscription, create new) — ensure the old one is released to avoid leaks (05_dispose_and_leaks.md).

Compiler Behavior

Not applicable. Use `covariant` for the typed old-widget parameter.

Runtime Behavior

If the widget type/key changes, the element is **replaced** (new `State`, `initState` runs) rather than updated — so `didUpdateWidget` only fires for same-type-same-key updates.

Flutter Engine Behavior / Dart VM Behavior

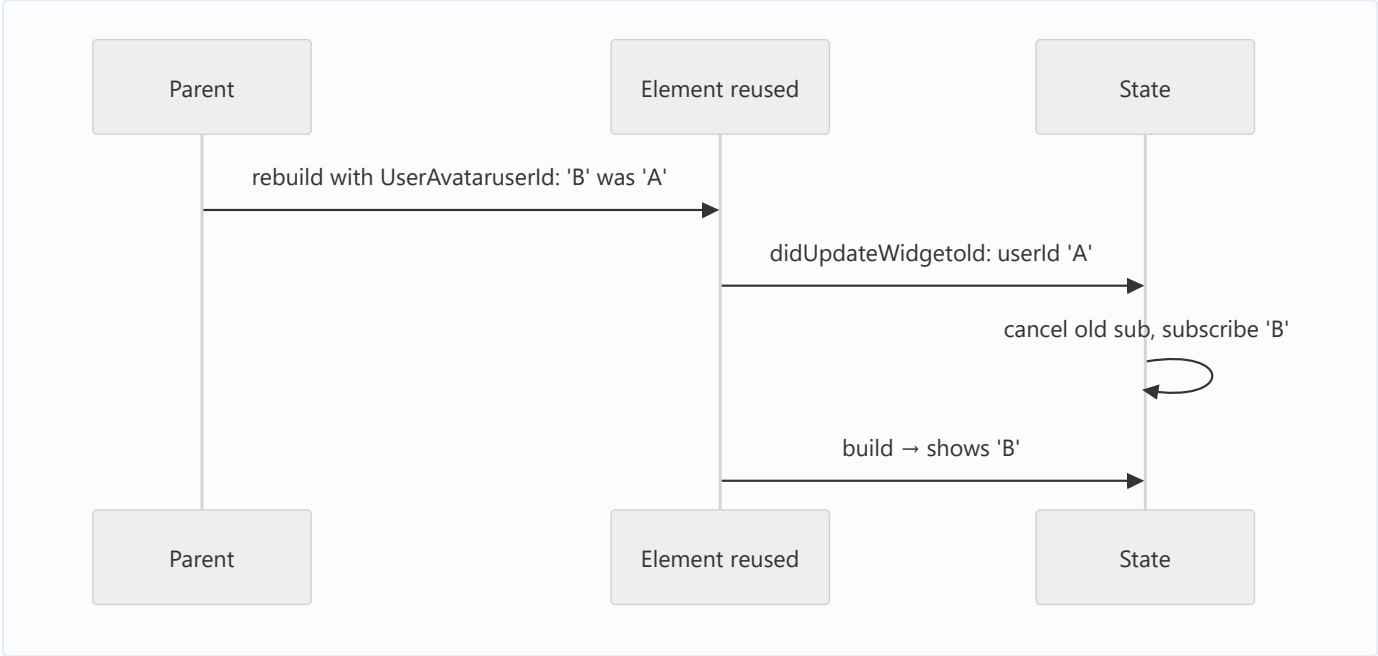
Not applicable.

Examples

```
import 'dart:async';
import 'package:flutter/material.dart';

class UserAvatar extends StatefulWidget {
  final String userId;
  const UserAvatar({super.key, required this.userId});
  @override
  State<UserAvatar> createState() => _UserAvatarState();
}

class _UserAvatarState extends State<UserAvatar> {
  StreamSubscription<String>? _sub;
  String _name = '...';

  @override
  void initState() {
    super.initState();
    _subscribe(widget.userId);
```

```
  }

  @override
  void didUpdateWidget(covariant UserAvatar oldWidget) {
    super.didUpdateWidget(oldWidget);
    if (oldWidget.userId != widget.userId) {
      // config changed -> re-subscribe for the new user
      _sub?.cancel();               // tear down old
      _subscribe(widget.userId);    // set up new
    }
  }

  void _subscribe(String id) {
    _sub = _profileStream(id).listen((name) {
      if (mounted) setState(() => _name = name);
    });
  }

  Stream<String> _profileStream(String id) async* {
    yield 'User $id';
  }

  @override
  void dispose() {
    _sub?.cancel(); // final cleanup
    super.dispose();
  }

  @override
  Widget build(BuildContext context) => Text(_name);
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Ignoring config changes	Stale data/subscriptions	React in `didUpdateWidget`
Not cancelling old resource before re-setup	Leak/duplicate	Cancel old, then create new
Reacting unconditionally (no field compare)	Wasteful/incorrect restarts	Compare old vs new fields
Expecting `didUpdateWidget` on key/type change	It's replaced instead (`initState` runs)	Handle init in `initState` too
Forgetting `super.didUpdateWidget`	Framework invariants	Always call super

Best Practices

- Compare **specific fields** (`oldWidget.x != widget.x`) and react only to real changes.
- **Tear down before re-setup** (cancel old subscription/controller, then create new).
- Keep the same acquisition logic reusable across `initState` and `didUpdateWidget` (a private `_setup(id)`).
- Remember `widget` = new config inside the method; the parameter = old.

Performance

Reacting only on actual change avoids needless re-subscribes/animation restarts; unconditional work causes flicker/waste (Module 21).

Advantages / Disadvantages

- + Keeps a reused `State` in sync with changing config; avoids stale data.
- − Easy to forget; must manage teardown/re-setup carefully to avoid leaks/duplicates.

Interview Questions

1. 🟢 **When is `didUpdateWidget` called?** — When the parent rebuilds this widget with a new instance of the **same type and key**, so the element reuses this `State`.
2. 🟢 **What does the `oldWidget` parameter represent?** — The previous configuration; `widget` already holds the new one.
3. 🟡 **Why not just handle everything in `build`?** — `build` runs constantly and shouldn't perform side effects like re-subscribing; `didUpdateWidget` fires only on config change.
4. 🟡 **What happens if the widget's type or key changes instead?** — The element is replaced: old `State` is disposed and a new one runs `initState` (no `didUpdateWidget`).
5. 🟡 **What must you do when a watched prop changes?** — Tear down the old resource (cancel subscription/controller) and set up for the new value.
6. 🔴 **How do you avoid leaks/duplicates across updates?** — Cancel the previous subscription/controller before creating the new one; final cleanup in `dispose`.
7. 🔴 **Why compare fields instead of reacting unconditionally?** — To avoid unnecessary restarts/refetches (flicker, wasted work) when unrelated props changed.

Senior Engineer Tips

- Factor resource setup into a `_setup(config)` used by both `initState` and `didUpdateWidget`, with a matching `_teardown()`.
- If a passed-in controller (e.g., `TextEditingController`) changes, swap listeners in `didUpdateWidget`.
- Remember key/type change ⇒ fresh `State`; design so both paths (init and update) are handled.

Architect Perspective

`didUpdateWidget` keeps reusable, parameterized widgets correct as inputs change — essential for list items, detail screens keyed by id, and controller-driven widgets. Combined with keys and reconciliation, it's how Flutter efficiently reuses elements while staying data-correct (Module 06).

Summary

- `didUpdateWidget(oldWidget)` fires when a reused `State` gets a new same-type-same-key config.
- Compare old vs new fields; tear down old resources before setting up new; call `super`.

- Type/key change replaces the `State` (runs `initState`) instead.

Revision Notes

- Fires on new config, same type+key (element reused); `widget`=new, param=old.
- React to changed fields: cancel old → setup new; compare before acting.
- Type/key change ⇒ replace (initState), not update.
- Factor `_setup` / `_teardown`; final cleanup in `dispose`; call super.

Practice Questions

1. Why does changing `userId` need `didUpdateWidget`, not `build`?
2. What happens if the widget's `key` changes instead of a field?
3. How do you avoid a leaked subscription across updates?

Coding Questions

1. Re-subscribe a stream when a `topic` prop changes.
2. Restart an animation when its `duration` prop changes.
3. Swap listeners when an injected `controller` instance changes.

Mini Project

Keyed detail loader (Flutter): Build a `DetailView(id)` that loads/streams data for `id`, re-loading when `id` changes via `didUpdateWidget` (teardown+setup), with a shared `_setup` / `_teardown` and full disposal. Acceptance: no stale data on id change; no leaked subscriptions; reacts only on real change; app runs.

`setState` Mechanics & Pitfalls

> `setState(fn)` runs `fn` synchronously to mutate state, then marks the element dirty so the framework rebuilds it on the next frame — it does **not** rebuild immediately, and calling it after `dispose` throws.

Introduction

`setState` is the primitive that turns state changes into rebuilds. Understanding *exactly* what it does (and doesn't) prevents the classic bugs: async misuse, post-dispose calls, heavy work inside it, and over-broad rebuilds.

Why this concept exists

In the declarative model (`UI = f(state)` — 06 · declarative_ui), the framework needs to know *which* elements changed so it can rebuild them efficiently. `setState` is how you signal "my state changed, schedule a rebuild of this element's subtree."

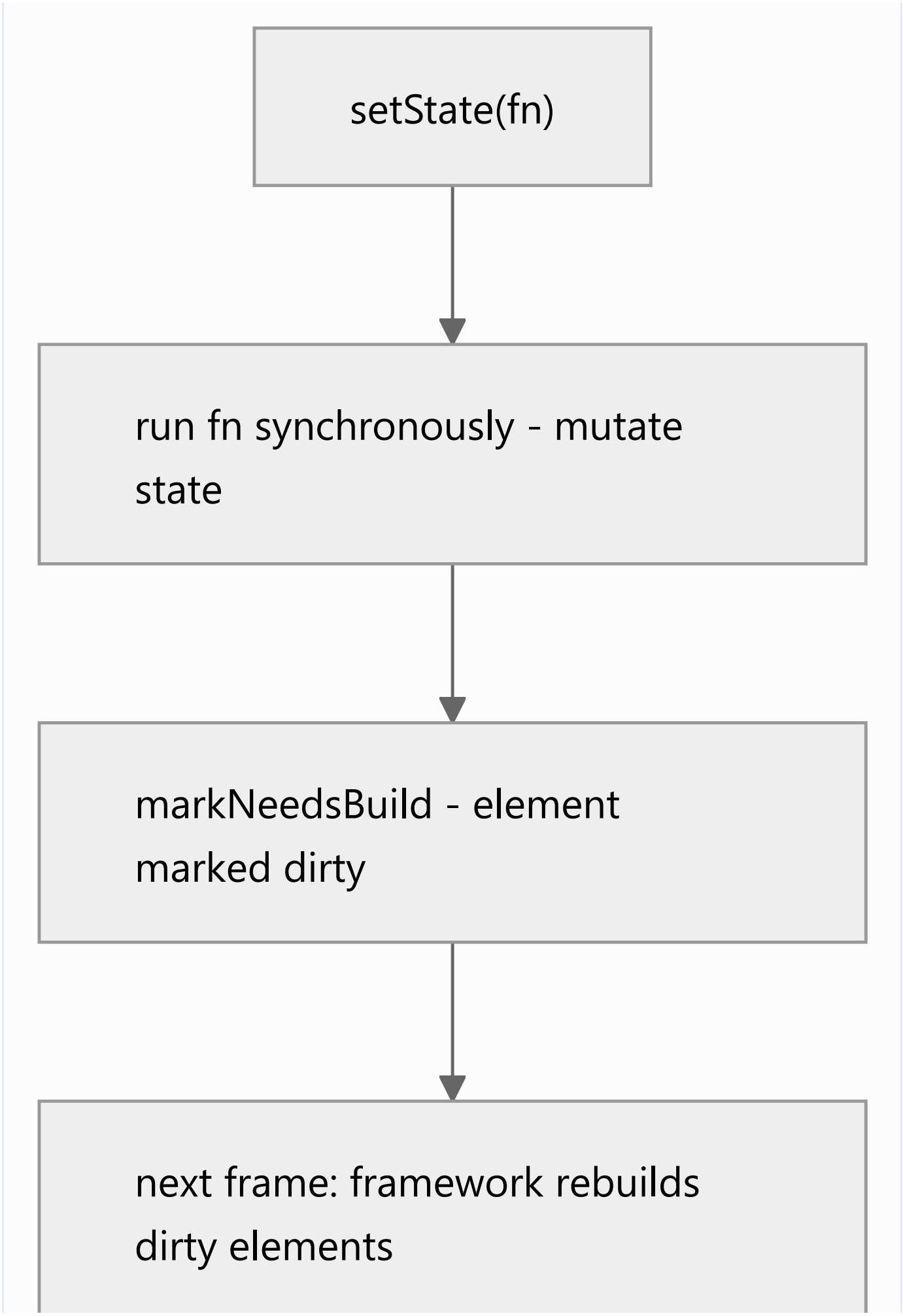
Real-world analogy

`setState` is **flipping a "needs cleaning" flag on a hotel room**: you update the room's status (mutate state) and raise the flag (mark dirty). Housekeeping (the framework) doesn't come *instantly* — it processes flagged rooms on its next round (next frame).

Problem Statement

You increment a counter, fetch data async then update UI, and wonder why a late callback crashes with "setState() called after dispose()". You'll learn `setState` 's exact behavior and safe async usage.

Internal Working


setState(fn)
run fn synchronously - mutate state
markNeedsBuild - element marked dirty
next frame: framework rebuilds dirty elements

- `setState(fn)`:
  1. Runs `fn` **synchronously** (do your state mutation here).
  2. Marks this element **dirty** (`markNeedsBuild`).
  3. The framework rebuilds it during the **next frame** — not immediately.
- Rebuild scope = **this element's subtree** (the whole `State` 's `build`), reconciled against the old tree.
- `setState` **asserts** `mounted` : calling after `dispose` throws.
- The `fn` should be **synchronous and cheap** — just the mutation. Do async/heavy work *before* calling `setState` .

Memory Representation

No allocation of note; it flips a dirty flag. Rebuilds recreate throwaway widgets (GC'd) while elements persist (Module 06).

Compiler Behavior

Not applicable.

Runtime Behavior

Multiple `setState` calls in one synchronous stretch coalesce into a single rebuild next frame. `setState` outside `build` ? fine (handlers/callbacks). Inside `build` ? causes an error/rebuild loop.

Flutter Engine Behavior

Marking dirty schedules a frame; the engine drives the build→layout→paint pipeline (Module 09).

Dart VM Behavior

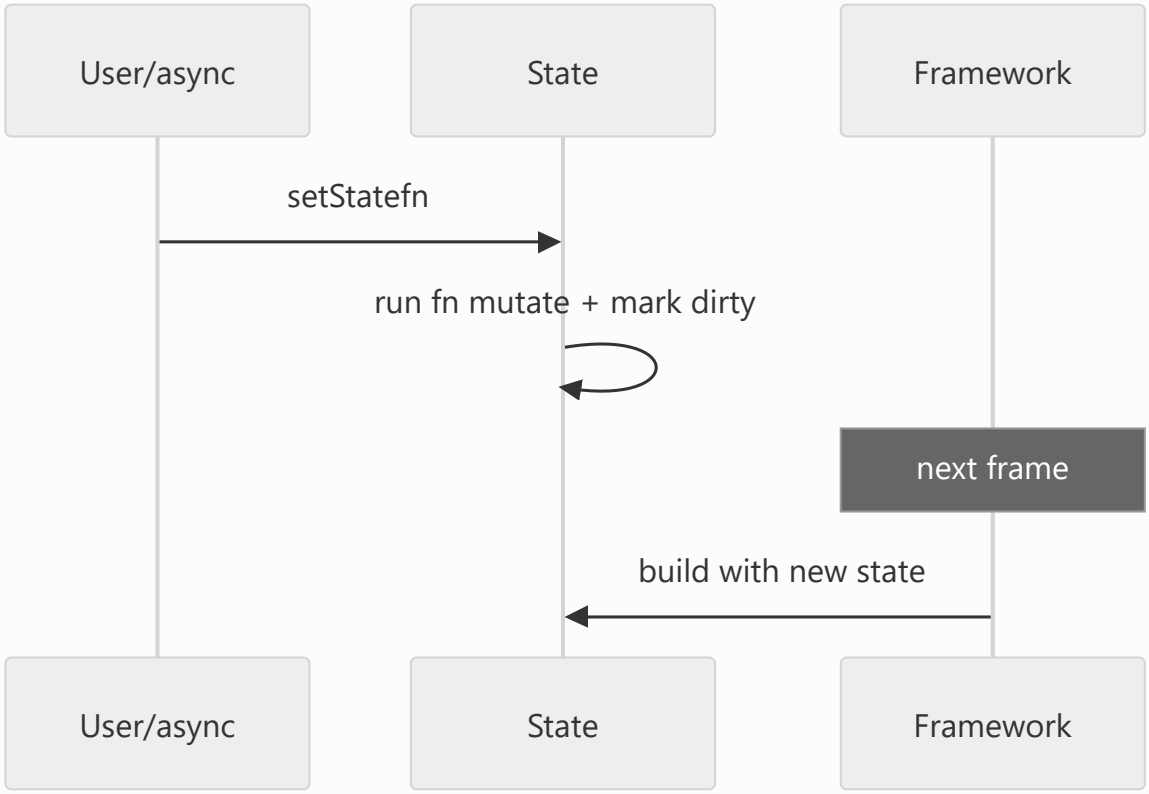
Not applicable directly; async callbacks run on the event loop (02 · event_loop).

Examples

```
import 'package:flutter/material.dart';

class Counter extends StatefulWidget {
  const Counter({super.key});
  @override
  State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;
  bool _loading = false;
  String _data = '';

  void _increment() {
    setState(() => _count++); // sync mutation -> rebuild next frame
  }

  Future<void> _load() async {
    setState(() => _loading = true);       // show spinner
    final result = await _fetch();         // async work OUTSIDE setState
    if (!mounted) return;                  // guard: widget may be disposed
    setState(() {                          // apply result
      _data = result;
      _loading = false;
    });
  }

  Future<String> _fetch() async {
    await Future.delayed(const Duration(milliseconds: 300));
    return 'loaded';
  }

  @override
  Widget build(BuildContext context) {
```

```
    return Column(
      mainAxisSize: MainAxisSize.min,
      children: [
        Text('Count: $_count'),
        if (_loading) const CircularProgressIndicator() else Text(_data),
        ElevatedButton(onPressed: _increment, child: const Text('+')),
        ElevatedButton(onPressed: _load, child: const Text('Load')),
      ],
    );
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`await` **inside** the `setState` callback	Callback must be sync	Await first, then `setState` with the result
`setState` after `dispose` (late async)	Asserts `mounted` → throws	Guard `if (!mounted) return;`
Heavy work inside `setState`	Runs synchronously, blocks	Do work before; only mutate in the callback
`setState` in `build`	Rebuild loop	Trigger from handlers/effects
Empty `setState(() {})` after mutating a field elsewhere	Works but obscures intent	Mutate inside the callback
Over-broad `setState` on a huge widget	Rebuilds too much	Split widgets / push state down (Module 11)

Best Practices

- Keep the `setState` callback **synchronous and minimal** — just the mutation.
- Do **async/heavy work outside**, then guard with `mounted` before `setState`.
- Never call `setState` in `build`; call it from event handlers/callbacks.
- **Scope rebuilds**: split large widgets so `setState` rebuilds the smallest subtree.
- For shared/complex state, prefer a state-management solution over raw `setState` (Module 11).

Performance

Rebuild scope = the calling `State`'s subtree; keep it small. Coalesced `setState`s = one rebuild. Cheap `build` + `const` keep frequent rebuilds affordable (Module 21).

Advantages / Disadvantages

- **+** Simple, built-in, precise per-widget rebuild trigger.
- **−** Manual; easy to misuse (async/dispose); rebuilds the whole `State` subtree; doesn't scale to shared state.

Interview Questions

1. 🟢 **What does `setState` do?** — Runs the callback synchronously to mutate state, marks the element dirty, and the framework rebuilds it next frame (not immediately).
2. 🟢 **Does `setState` rebuild immediately?** — No; it schedules a rebuild for the next frame.

3. 🟡 **Why is `setState` after `dispose` an error?** — It asserts `mounted`; the element is gone, so there's nothing to rebuild — guard with `mounted`.
4. 🟡 **Why shouldn't the `setState` callback be async?** — It must complete synchronously; do `await` before and call `setState` with the result.
5. 🟡 **What is the rebuild scope of `setState`?** — The calling `State`'s subtree (its `build`), reconciled against the previous tree.
6. 🔴 **What happens with multiple `setState` calls in one synchronous block?** — They coalesce into a single rebuild on the next frame.
7. 🔴 **When should you stop using `setState`?** — When state is shared across widgets or logic is complex — move to a state manager (Module 11).

Senior Engineer Tips

- The async pattern is fixed: `setState(loading=true)` → `await` → `if(!mounted) return;` → `setState(apply result)`.
- If `setState` rebuilds too much, that's a signal to split the widget or lift state, not to optimize `build`.
- Prefer `ValueListenableBuilder`/state management to rebuild only the tiny reactive part.

Architect Perspective

`setState` is fine for **local, ephemeral UI state**; beyond that, its whole-subtree rebuild and manual nature don't scale. Recognizing that boundary — local `setState` vs managed shared state — is the key architectural decision leading into Module 11 State Management.

Summary

- `setState` mutates synchronously, marks dirty, rebuilds next frame (this subtree).
- Keep the callback sync/minimal; do async outside + guard `mounted`; never in `build`.
- Scope rebuilds; graduate to state management for shared/complex state.

Revision Notes

- `setState(fn)`: run fn sync (mutate) → mark dirty → rebuild next frame (this subtree).
- Async: await first → `if(!mounted) return;` → setState. Never async callback, never in `build`.
- Multiple calls coalesce; scope rebuilds by splitting widgets.
- Local UI state only → else Module 11.

Practice Questions

1. Why does the UI update "next frame" not instantly?
2. Fix a "setState after dispose" from a delayed callback.
3. Why can't the `setState` callback be `async`?

Coding Questions

1. Implement an async load with spinner using the safe `setState` pattern.
2. Show two coalesced `setState` calls producing one rebuild (log builds).
3. Refactor an over-broad `setState` by splitting into a smaller stateful subwidget.

Mini Project

Safe async loader (Flutter): Build a widget that loads data with a spinner using the `setState(loading)` → `await` → `mounted`-guard→ `setState(result)` pattern, plus an error path. Split the reactive part into a small subwidget to scope rebuilds. Acceptance: no post-dispose crash; sync callbacks; scoped rebuild; app runs.

`dispose` & Leak Prevention

> `dispose()` is your one guaranteed chance to release resources — cancel subscriptions/timers and dispose controllers there, or they (and everything they capture) leak.

Introduction

`dispose()` runs once when a `State` is permanently removed from the tree. It's where you undo everything acquired in `initState` / `didChangeDependencies` / `didUpdateWidget`: stream subscriptions, timers, `AnimationController`s, `TextEditingController`s, `FocusNode`s, `ScrollController`s, and listeners. Missing cleanup is the #1 Flutter memory-leak source.

Why this concept exists

Flutter is GC-managed, but GC only reclaims **unreachable** objects (02 · memory_and_gc). An active subscription/timer/listener keeps a reference to your callback — which captures `this` (the `State`) — so the whole widget subtree stays reachable and can't be collected. `dispose` breaks those references.

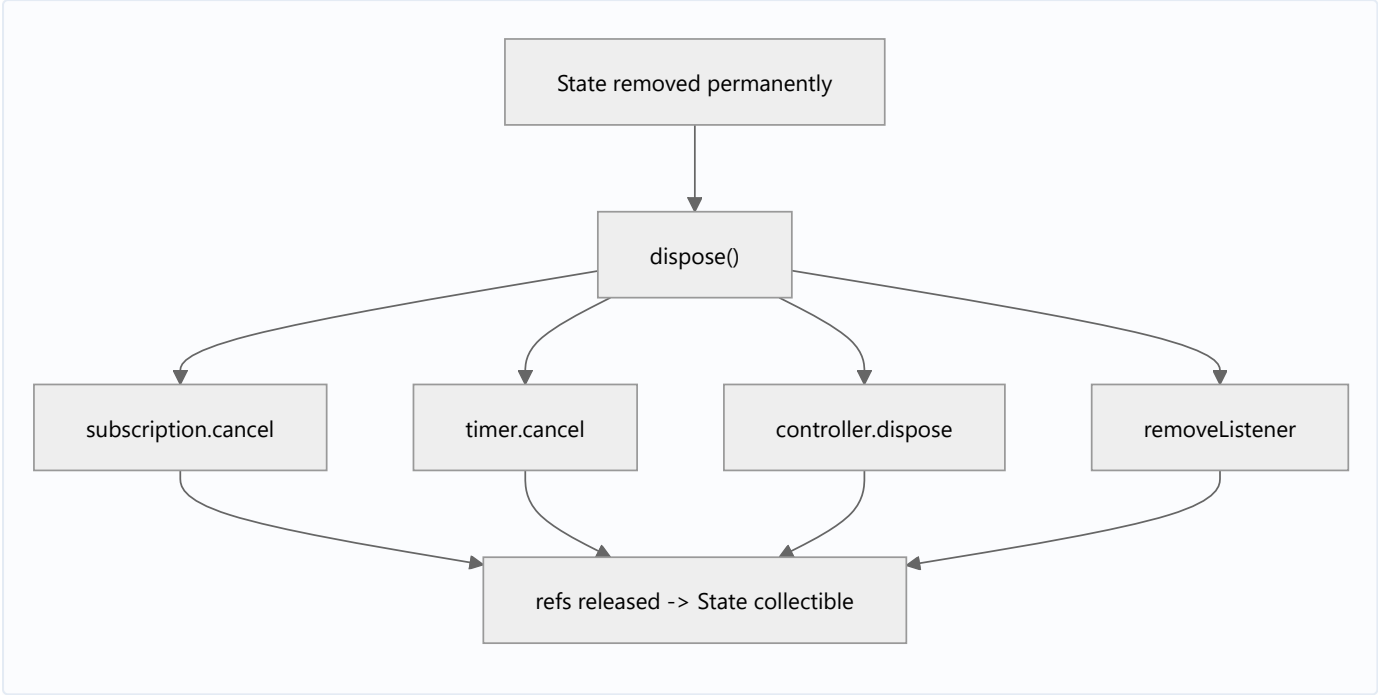
Real-world analogy

Moving out of an apartment: `dispose` is **returning the keys, cancelling utilities, and forwarding mail**. If you skip it, the utility company keeps billing (timer fires), the old address keeps receiving mail (callbacks), and you're still "reachable" — you never truly leave (leak).

Problem Statement

Your screen's memory climbs each time it's opened/closed. It has a stream subscription, a periodic timer, and an `AnimationController`. You'll release all three in `dispose` and verify the leak is gone.

Internal Working



- `dispose()` runs once, after `deactivate`, when the `State` won't be reused.
- Release everything with an owner in this `State`:
  - `StreamSubscription.cancel()`
  - `Timer.cancel()`
  - `AnimationController.dispose()`, `TextEditingController.dispose()`, `ScrollController.dispose()`, `FocusNode.dispose()`
  - `listenable.removeListener(...)`
- Call `super.dispose()` **last**.
- After `dispose`, `mounted` is false → never `setState`.

Memory Representation

Active subscriptions/timers/listeners are GC roots-by-reachability: they retain the `State` and its captured objects (including `BuildContext`, big data) until released (02 · memory_and_gc).

Compiler Behavior

Analyzer lints (e.g., `close_sinks`, package `leak_tracker` in tests) help catch some cases; not comprehensive.

Runtime Behavior

A fired timer/stream callback after removal that touches state throws (or leaks); cancel them in `dispose` to prevent both.

Flutter Engine Behavior / Dart VM Behavior

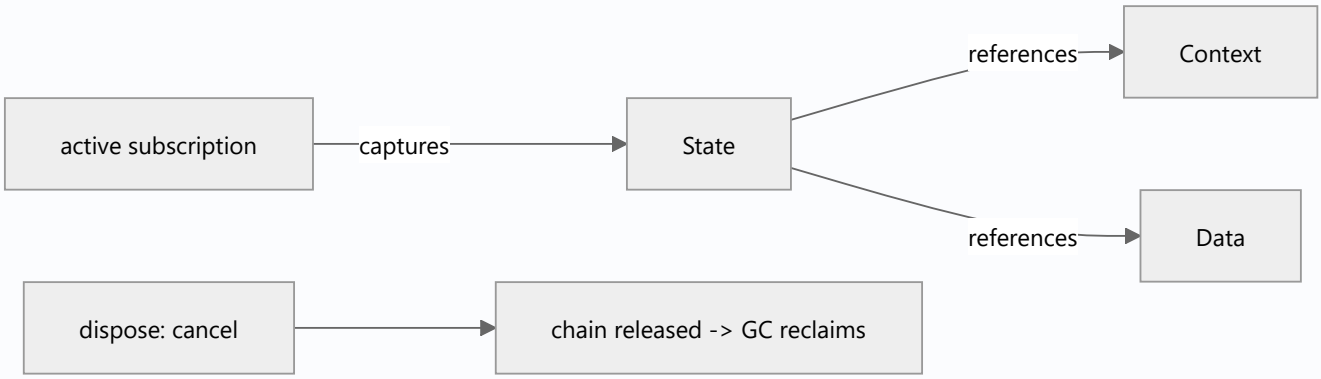
Not applicable directly; leaked objects increase heap pressure and GC work over time.

Examples

```
import 'dart:async';
import 'package:flutter/material.dart';

class LiveScreen extends StatefulWidget {
  final Stream<int> ticks;
  const LiveScreen({super.key, required this.ticks});
  @override
  State<LiveScreen> createState() => _LiveScreenState();
}

class _LiveScreenState extends State<LiveScreen>
    with SingleTickerProviderStateMixin {
  StreamSubscription<int>? _sub;
  Timer? _timer;
  late final AnimationController _anim;
  final _scroll = ScrollController();
  int _value = 0;

  @override
  void initState() {
    super.initState();
    _anim = AnimationController(vsync: this, duration: const Duration(seconds: 1));
    _sub = widget.ticks.listen((v) {
      if (mounted) setState(() => _value = v);
    });
    _timer = Timer.periodic(const Duration(seconds: 1), (_) {/* ... */});
  }

  @override
```

```
  void dispose() {
    _sub?.cancel();      // 1. cancel stream subscription
    _timer?.cancel();    // 2. cancel timer
    _anim.dispose();     // 3. dispose animation controller
    _scroll.dispose();   // 4. dispose scroll controller
    super.dispose();     // last
  }

  @override
  Widget build(BuildContext context) =>
      ListView(controller: _scroll, children: [Text('$_value')]);
}
```

Diagrams



Common Mistakes

Mistake	Why leaks	Fix
Not cancelling `StreamSubscription`	Stream retains callback → `State`	`cancel()` in `dispose`
Not disposing controllers	Framework/ticker retains them	`dispose()` each
Not cancelling `Timer.periodic`	Scheduler holds callback	`cancel()`
`addListener` without `removeListener`	Listenable retains listener	`removeListener` in `dispose`
`super.dispose()` first	Frees framework state too early	Call it **last**
Storing `BuildContext` /large data in long-lived objects	Retained via the object	Don't; release owners

Best Practices

- **Every acquire has a release**: create in `initState`, free in `dispose`, in reverse-ish order; `super.dispose()` last.
- Cancel subscriptions/timers; dispose all controllers/focus nodes; remove listeners.
- Guard async callbacks with `mounted` so a not-yet-cancelled callback can't `setState` post-dispose.
- Consider a **mixin/base** that tracks and disposes resources uniformly.
- Verify with **DevTools memory** (open/close cycles) and `leak_tracker` in tests.

Performance

Leaks cause gradual memory growth → more GC → eventual jank/OOM on low-end devices (Module 21). Proper disposal keeps memory flat across navigation.

Advantages / Disadvantages

- + Deterministic cleanup point; prevents leaks and post-dispose callbacks.
- − Manual and easy to forget; must mirror every acquisition.

Interview Questions

1. 🟢 **What is `dispose` for?** — Releasing resources when a `State` is permanently removed: cancel subscriptions/timers, dispose controllers, remove listeners.
2. 🟢 **Name three things you must dispose/cancel.** — `StreamSubscription` (cancel), `Timer` (cancel), controllers like `AnimationController` / `TextEditingController` (dispose).
3. 🟡 **Why does an uncancelled subscription leak the whole screen?** — The stream holds the callback, which captures `this` (the `State`), keeping the widget/state graph reachable.
4. 🟡 **Where should `super.dispose()` go?** — Last, after your own cleanup.
5. 🟡 **How do you prevent post-dispose `setState`?** — Guard with `if (!mounted) return;` in async callbacks and cancel their sources in `dispose`.
6. 🔴 **How do you detect a leak?** — DevTools memory: heap snapshots across open/close cycles, look for growing instances, inspect the retaining path; `leak_tracker` in tests.
7. 🔴 **How would you standardize cleanup across many widgets?** — A mixin/base class that registers subscriptions/controllers and disposes them all in one `dispose`.

Senior Engineer Tips

- Adopt a rule enforced in review: "if you `listen`/create a controller/start a timer, cancel/dispose it in `dispose`."

- Build a `DisposeBag` / `AutoDisposeMixin` to register-and-release resources uniformly.
- Prefer state management (`Bloc` /Riverpod auto-dispose) which handles much of this for you (Module 11).

Architect Perspective

Disposal discipline is a system-wide reliability concern; leaks are a leading cause of gradual slowdowns and crashes at scale. Standardizing lifecycle-resource management (mixins, lints, leak tests in CI) and leaning on auto-disposing state solutions is an architectural decision that keeps long-running apps healthy (02 · memory_and_gc, Module 49).

Summary

- `dispose` is the one guaranteed cleanup hook: cancel subscriptions/timers, dispose controllers, remove listeners; `super.dispose()` last.
- Uncancelled resources retain the `State` (and its captures) → leaks.
- Guard async with `mounted`; standardize with mixins; verify with DevTools/leak_tracker.

Revision Notes

- Release in `dispose`: subscription.cancel, timer.cancel, controller.dispose, removeListener; `super.dispose()` last.
- Leak cause: active ref captures `this` → State not collectible.
- Guard async with `mounted`; DisposeBag/mixin to standardize.
- Detect: DevTools retaining path / `leak_tracker`.

Practice Questions

1. Why does a forgotten `Timer.periodic` keep a screen alive?
2. Why call `super.dispose()` last?
3. How do you confirm a leak is fixed?

Coding Questions

1. Write an `AutoDisposeMixin` that tracks subscriptions/controllers and disposes all.
2. Reproduce a leak (uncancelled subscription) then fix it; verify in DevTools.
3. Add `mounted` guards to prevent post-dispose `setState`.

Mini Project

Leak-safe live screen (Flutter): Build a screen with a stream subscription, a periodic timer, an `AnimationController`, and a `ScrollController`; dispose all correctly and guard async. Add a `DisposeBag` mixin and refactor to use it. Verify (notes) via DevTools that memory is flat across open/close. Acceptance: all resources released; `super.dispose()` last; no post-dispose calls; app runs.

App Lifecycle (`AppLifecycleState`, `WidgetsBindingObserver`)

> Beyond individual widgets, the whole app moves between **resumed, inactive, paused, hidden, and detached** states; observe them with `WidgetsBindingObserver` to pause/resume work when the app goes to the background or foreground.

Introduction

Widget lifecycle handles a widget's tenure; **app lifecycle** handles the process moving in/out of the foreground. You observe it via `WidgetsBindingObserver.didChangeAppLifecycleState(AppLifecycleState)` (or `AppLifecycleListener`). This is where you pause animations/timers, stop the camera, save drafts, or resume streams.

Why this concept exists

When the user backgrounds the app (call, home button, app switcher), continuing to run animations, GPS, camera, or network polling wastes battery and may be killed by the OS. The app lifecycle lets you react — pausing on background, resuming on foreground, and persisting state before possible termination.

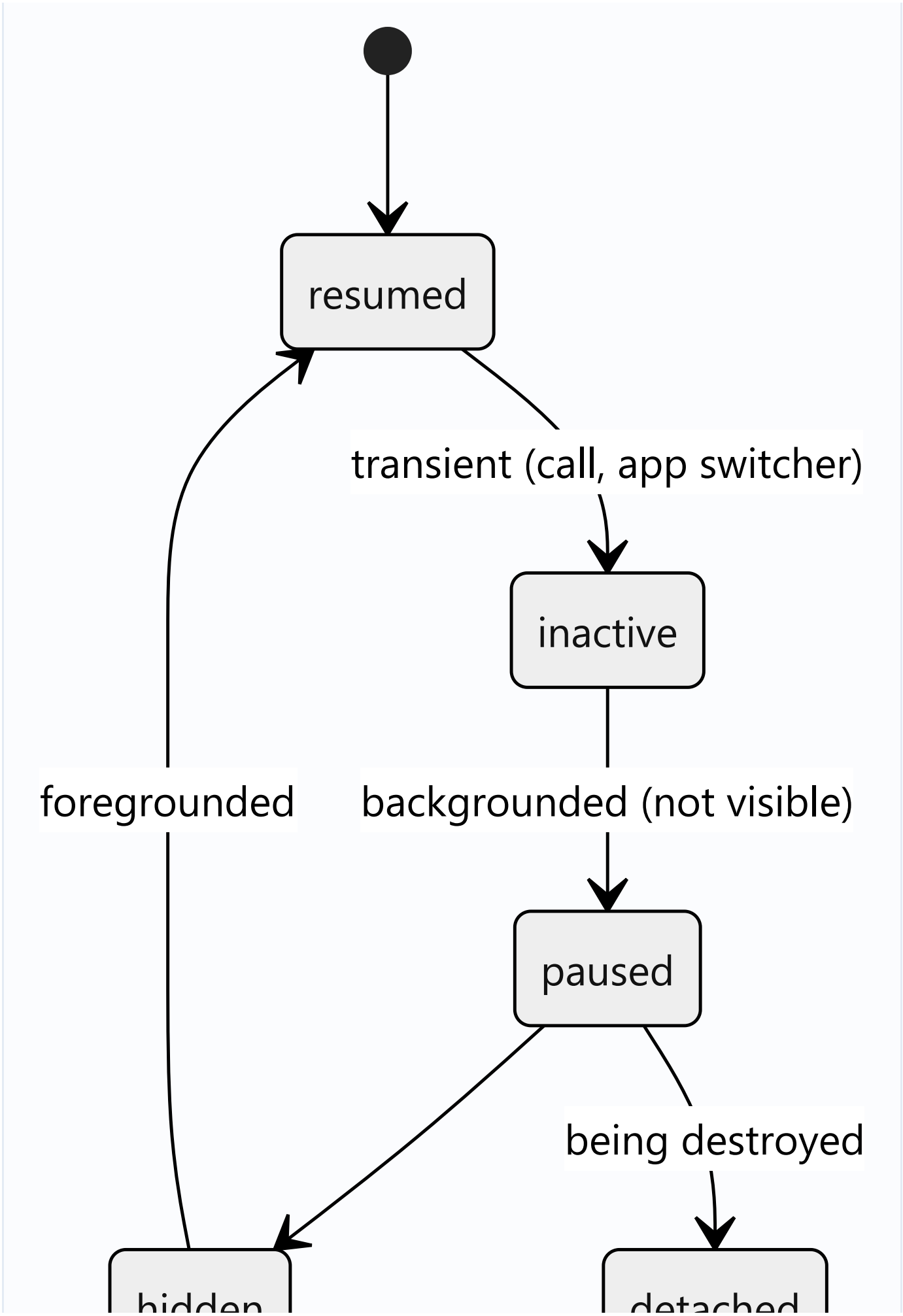
Real-world analogy

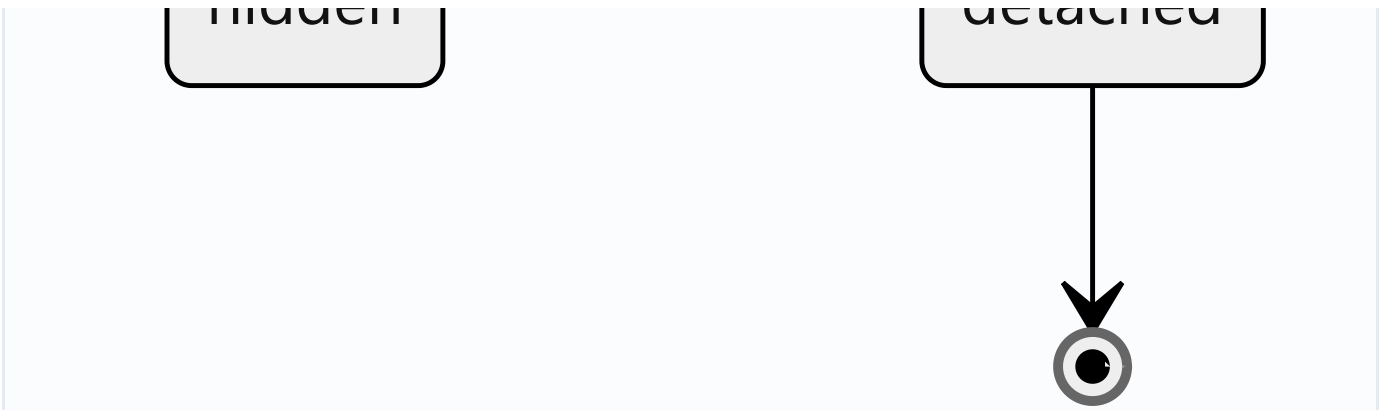
A **shop's open/closed status**: when open (`resumed`) staff serve customers; when the shutters come down (`paused`), you stop the coffee machine, lock the till (save state), and turn off lights (release resources) — then reverse it when reopening.

Problem Statement

Your app runs a location stream and an animation. When backgrounded, both should pause (battery); on return, resume; and if the OS may kill the app, persist a draft. You'll observe `AppLifecycleState`.

Internal Working


resumed
transient (call, app switcher)
inactive
foregrounded
backgrounded (not visible)
paused
being destroyed
hidden
detached



State	Meaning	Typical action
`resumed`	Visible + interactive (foreground)	Resume streams/animations
`inactive`	Transient, not receiving input (e.g., incoming call, app switcher)	Light pause
`paused`	Backgrounded, not visible	Pause timers/GPS/camera; save state
`hidden`	Hidden (newer unified state across platforms)	Similar to paused
`detached`	Engine running, no view; app being destroyed	Final cleanup

- Implement `WidgetsBindingObserver`, register with `WidgetsBinding.instance.addObserver(this)` in `initState`, remove in `dispose`, and override `didChangeAppLifecycleState`.
- Modern alternative: `AppLifecycleListener` (Flutter 3.13+) with granular callbacks (`onResume`, `onPause`, `onDetach`, etc.).

Memory Representation

The observer is held by the binding until removed — **remove it in** `dispose` or it leaks (same principle as 05_dispose_and_leaks.md).

Compiler Behavior

Not applicable.

Runtime Behavior

The framework invokes `didChangeAppLifecycleState` on transitions. `detached`/kill may not always give you time — persist important state on `paused`.

Flutter Engine Behavior

The embedder reports platform lifecycle events to the engine, surfaced as `AppLifecycleState` (06 · architecture_overview).

Dart VM Behavior

Not applicable.

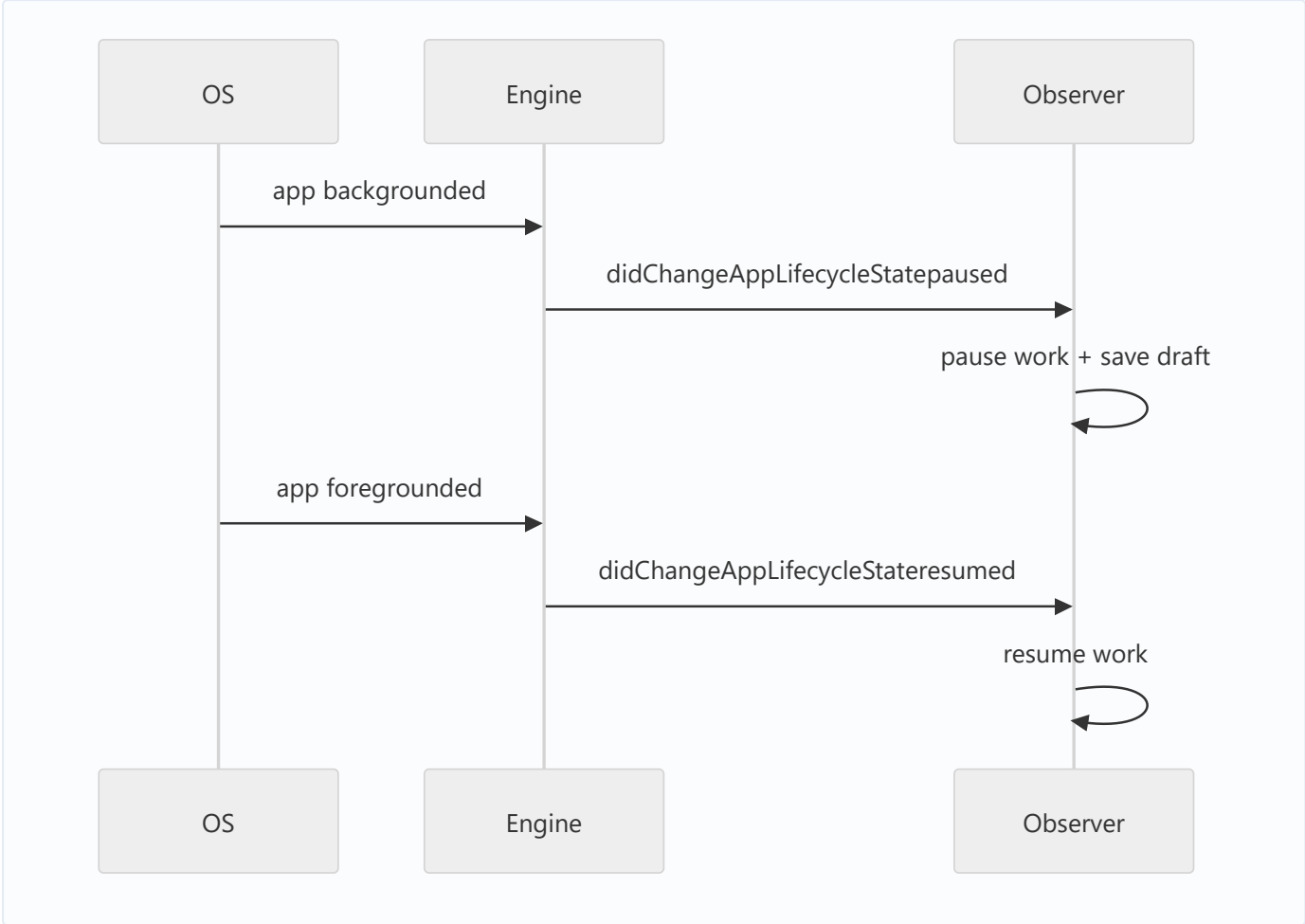
Examples

```
import 'package:flutter/material.dart';


class LifecycleAware extends StatefulWidget {
  const LifecycleAware({super.key});
  @override
  State<LifecycleAware> createState() => _LifecycleAwareState();
}


class _LifecycleAwareState extends State<LifecycleAware>
    with WidgetsBindingObserver {
  String _status = 'resumed';


  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this); // register
  }


  @override
  void didChangeAppLifecycleState(AppLifecycleState state) {
    setState(() => _status = state.name);
    switch (state) {
      case AppLifecycleState.resumed:
        _resumeWork();   // resume streams/animations
      case AppLifecycleState.inactive:
        break;           // transient
      case AppLifecycleState.paused:
      case AppLifecycleState.hidden:
        _pauseWork();    // pause timers/GPS/camera
        _saveDraft();    // persist in case of termination
      case AppLifecycleState.detached:
        _finalCleanup();
    }
  }
```

```
  void _resumeWork() {}
  void _pauseWork() {}
  void _saveDraft() {}
  void _finalCleanup() {}

  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(this); // MUST remove
    super.dispose();
  }

  @override
  Widget build(BuildContext context) => Text('App: $_status');
}
```

```
// Modern alternative (Flutter 3.13+):
// final listener = AppLifecycleListener(
//   onResume: _resumeWork, onPause: _pauseWork, onDetach: _finalCleanup,
// );
// ... listener.dispose() when done.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not removing the observer in `dispose`	Leak	`removeObserver(this)`
Relying on `detached` for critical saves	May not run/finish	Persist on `paused`
Keeping GPS/camera/animations running when paused	Battery drain, OS kill	Pause on `paused` / `hidden`
Doing heavy work synchronously in the callback	Jank on transition	Keep it light; offload
Ignoring `inactive` vs `paused` distinction	Wrong reaction timing	Treat `inactive` as transient

Best Practices

- Register in `initState`, **remove in** `dispose`.
- **Pause** battery-heavy work (GPS/camera/animations/polling) on `paused` / `hidden`; **resume** on `resumed`.
- **Persist important state on** `paused` (don't rely on `detached`).
- Keep the callback light; offload heavy save/cleanup.

- Consider `AppLifecycleListener` for granular, modern handling.

Performance / Battery

Pausing background work saves battery and avoids OS termination for misbehavior; resuming promptly keeps UX smooth (Module 21, Module 33).

Advantages / Disadvantages

- + React to foreground/background; save battery; persist before termination; resume cleanly.
- − Observer lifecycle to manage (leak risk); `detached` timing not guaranteed; platform nuances.

Interview Questions

1. 🟢 **What are the `AppLifecycleState` values?** — `resumed`, `inactive`, `paused`, `hidden`, `detached`.
2. 🟢 **How do you observe app lifecycle?** — Implement `WidgetsBindingObserver`, `addObserver` in `initState`, `removeObserver` in `dispose`, override `didChangeAppLifecycleState` (or use `AppLifecycleListener`).
3. 🟡 **What should you do on `paused`?** — Pause battery-heavy work (GPS/camera/animations) and persist important state.
4. 🟡 **`inactive` vs `paused`?** — `inactive` is a transient state (e.g., app switcher/incoming call) still partly visible; `paused` is fully backgrounded/not visible.
5. 🟡 **Why not rely on `detached` for saving?** — It runs during teardown and may not complete; save on `paused` instead.
6. 🔴 **Why must you remove the observer in `dispose`?** — The binding retains it; not removing leaks the `State` (same reachability issue as subscriptions).
7. 🔴 **How would you pause a location stream app-wide on background?** — In `didChangeAppLifecycleState`, on `paused` / `hidden` pause the location service; resume on `resumed`.

Senior Engineer Tips

- Treat `paused` as "save now" — persistence you rely on must happen here, not `detached`.
- Centralize app-lifecycle handling (one observer / an `AppLifecycleListener` in a service) rather than scattering it across widgets.
- Combine with background execution (Module 33) when work must continue while backgrounded.

Architect Perspective

App-lifecycle handling is a cross-cutting concern (battery, data integrity, resource release). Centralizing it in a lifecycle service that coordinates streams, sensors, and persistence — rather than ad-hoc per widget — yields consistent, battery-friendly, crash-resilient behavior, and integrates with background services and offline-first strategies (Modules 33, 19).

Summary

- App lifecycle: `resumed` / `inactive` / `paused` / `hidden` / `detached`; observe via `WidgetsBindingObserver` / `AppLifecycleListener`.
- Pause heavy work and persist state on `paused`; resume on `resumed`; final cleanup on `detached` (don't rely on it for saves).
- Register in `initState`, remove observer in `dispose` (leak risk).

Revision Notes

- States: resumed/inactive/paused/hidden/detached.
- Observe: `WidgetsBindingObserver` + `add/removeObserver` (init/dispose) + `didChangeAppLifecycleState`; or `AppLifecycleListener`.
- `paused` → pause GPS/camera/anim + SAVE state (not `detached`).
- Remove observer in `dispose` (leak); centralize handling.

Practice Questions

1. Where do you persist a draft to survive app kill, and why?
2. Why remove the observer in `dispose`?
3. `inactive` vs `paused` — how do you react differently?

Coding Questions

1. Pause/resume an animation on app background/foreground.
2. Save a text draft on `paused` and restore on `resumed`.
3. Reimplement using `AppLifecycleListener` with granular callbacks.

Mini Project

Battery-aware tracker (Flutter): Build a widget that runs a (fake) location stream + animation, pauses both on `paused` / `hidden`, saves state, and resumes on `resumed`, using `WidgetsBindingObserver` with proper observer removal. Acceptance: work pauses/resumes correctly; state persisted on

`paused`; observer removed; app runs.