22 · Animations

Flutter Practice Notes

Contents

22 · Animations

Introduction

Motion communicates state, hierarchy, and continuity. Flutter's animation system is built on `Animation` / `AnimationController` / `Tween` / `Curve`, driven per-frame by the scheduler (Module 09). This module covers implicit animations, explicit control, staggered/choreographed sequences, physics/gesture-driven motion, and animation performance.

Why this module exists

Good motion elevates UX; bad motion janks or distracts. Knowing when to use the simple implicit widgets vs full explicit control — and how to keep animations at 60/120fps — is the difference between polish and pain.

Module map

#	File	Topic	Level
1	01_animation_fundamentals.md	`Animation` / `AnimationController` / `Tween` / `Curve`	🔵
2	02_implicit_animations.md	`AnimatedFoo` / `TweenAnimationBuilder`	🟢
3	03_explicit_animations.md	Controllers, `AnimatedBuilder`, transitions	🔵
4	04_staggered_and_choreographed.md	`Interval`, sequences, `AnimatedList`	🔴
5	05_physics_and_gesture_driven.md	Springs/physics, gesture-driven motion	🔴
6	06_animation_performance.md	Keeping animations smooth	🔴

> **Cross-references:** Scheduler/ `Ticker` /vsync: 09 · scheduler_and_vsync. Raster/repaint isolation: 09 · compositing, 21 · jank_and_raster. Dispose controllers: 08 · dispose_and_leaks. Hero/route transitions: 12 · route_transitions. Custom painting: Module 23.

Prerequisites

08 Widget Lifecycle, 09 · scheduler_and_vsync, 21 · jank_and_raster.

What you'll be able to do after this module

- Explain the animation system (controller → tween/curve → value → rebuild).
- Use implicit animations for simple state transitions.
- Build explicit, controllable, choreographed animations.

- Add physics/gesture-driven motion.
- Keep animations smooth (isolation, cheap builds, avoid raster cost).

Capstone

Animated interactions: An implicit expand/collapse card, an explicit staggered list entrance, and a gesture-driven draggable/dismissible item with spring physics — all profiled to hold 60fps.

Summary

Flutter animation is value-driven: a controller produces a 0→1 value, tweens/curves shape it, and widgets rebuild from it. Prefer implicit for simple cases, explicit for control, physics for natural motion — and always dispose controllers and keep frames within budget.

Animation Fundamentals (`AnimationController`, `Tween`, `Curve`)

> Flutter animation is value-driven: an `AnimationController` produces a value (0→1) each frame (via a `Ticker`), a `Tween` maps that to your range/type, a `Curve` shapes the timing, and the UI rebuilds from the resulting `Animation<T>`.

Introduction

The core objects: `AnimationController` (drives a 0→1 value over a duration, tied to a vsync `Ticker`), `Animation<T>` (a listenable value + status), `Tween<T>` (maps 0→1 to a range/type), and `Curve` (non-linear timing). This file explains how they compose — the foundation for every animation.

Why this concept exists

Animating manually per frame is error-prone. Flutter factors it: the controller handles the clock (frames/duration), tweens handle interpolation (any type), and curves handle easing — a composable system that plugs into the scheduler (09 · scheduler_and_vsync).

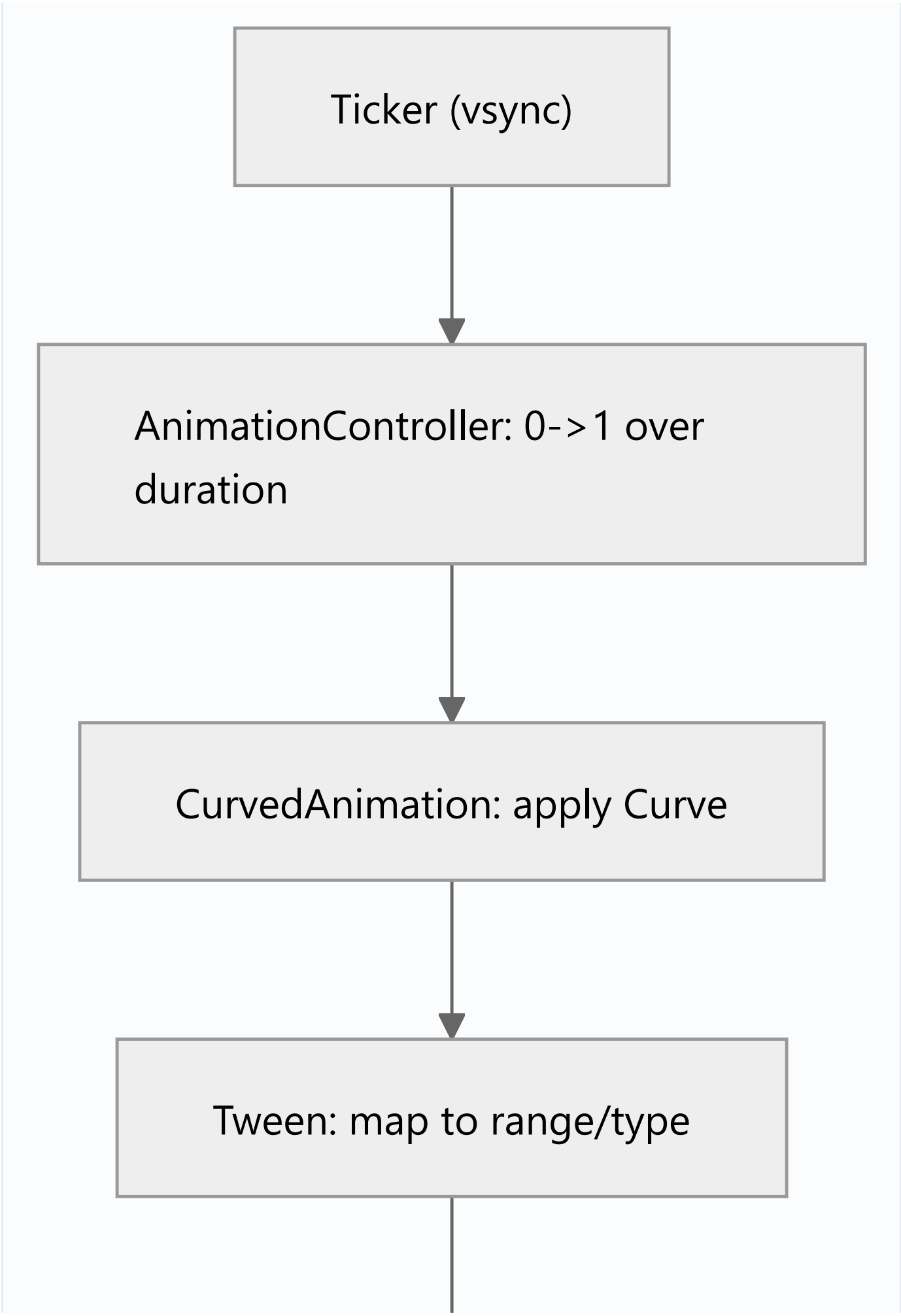
Real-world analogy

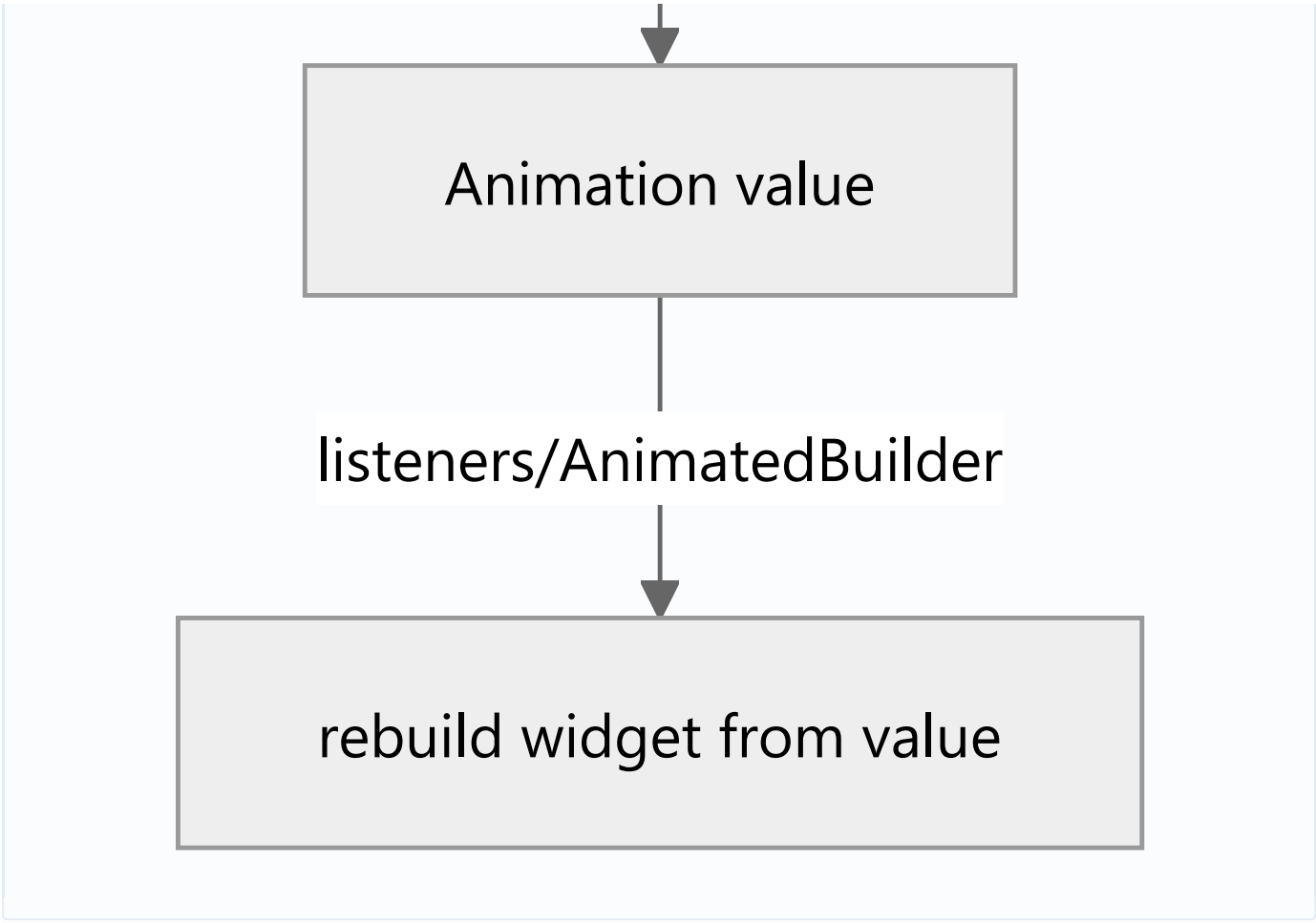
A **film projector**: the controller is the motor turning the reel at a set speed (duration); the tween is the **frames drawn** for each position (start→end); the curve is the **motor easing in/out** rather than instant full speed. Together they play smooth motion.

Problem Statement

Fade and slide a widget in over 400ms with an ease-out curve, controllable (play/reverse), and correctly disposed. You'll wire a controller + tweens + curve + an `AnimatedBuilder`.

Internal Working


Ticker (vsync)
AnimationController: 0->1 over duration
CurvedAnimation: apply Curve
Tween: map to range/type



- `AnimationController(vsync:, duration:)` : produces a value from `lowerBound` .. `upperBound` (default 0..1) driven by a `Ticker` each frame; methods `forward()` / `reverse()` / `repeat()` / `stop()` ; `status` (`forward` / `completed` / `reverse` / `dismissed`). Needs a `TickerProvider` (`SingleTickerProviderStateMixin`) and must be **disposed**.
- `Animation<T>` : a `Listenable` exposing `.value` + `.status` ; the controller *is* an `Animation<double>` .
- `Tween<T>(begin, end)` : `.animate(controller)` maps 0→1 to `[begin, end]` — works for `double` , `Offset` , `Color` (`ColorTween`), `Size` , etc.
- `Curve` (`Curves.easeOut` , etc.): non-linear timing via `CurvedAnimation(parent:, curve:)` (chain before the tween).
- **Consuming the value**: `AnimatedBuilder` / `ListenableBuilder` rebuild on each tick (preferred, scoped), or `addListener` + `setState` (coarser). `*Transition` widgets (`FadeTransition` , `SlideTransition`) consume an `Animation` directly.

Memory Representation

The controller + ticker are objects held by `State` ; not disposing leaks and keeps frames scheduled (08 · dispose_and_leaks). Tweens/curves are lightweight.

Compiler Behavior

Not applicable.

Runtime Behavior

`forward()` advances the value each frame (transient scheduler phase — 09 · scheduler_and_vsync); listeners fire; the widget rebuilds from `.value`. `status` transitions drive chaining/reversal.

Flutter Engine Behavior

Each tick triggers a rebuild → the animated widget re-paints; keep the rebuilt subtree small and isolate with `RepaintBoundary` to bound raster cost (06_animation_performance.md).

Dart VM Behavior

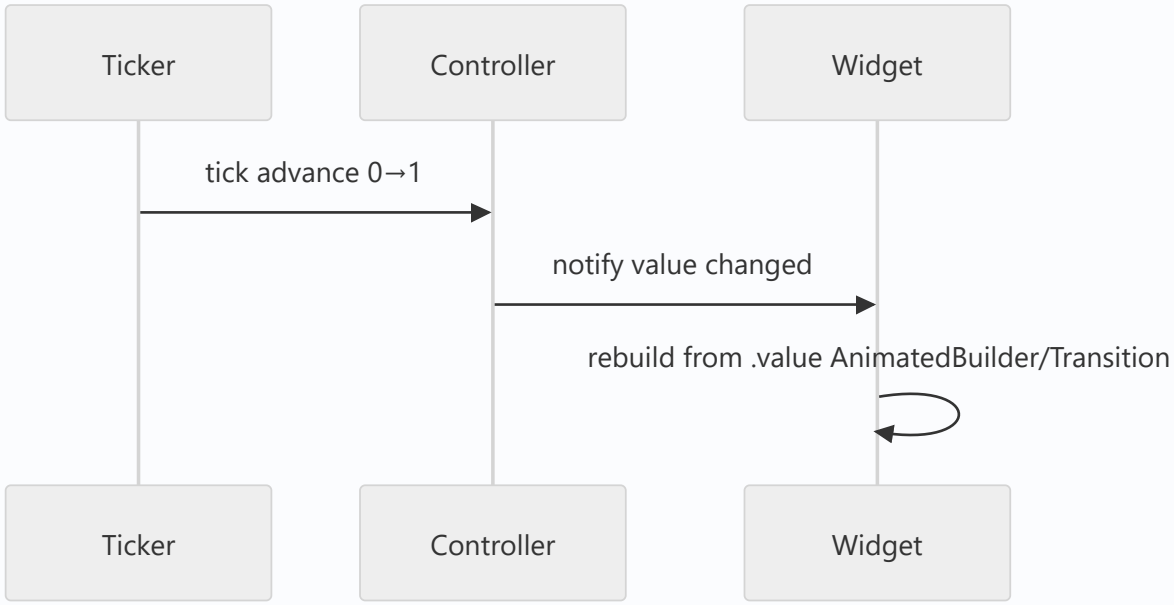
Ticker callbacks run on the isolate's frame; heavy per-tick work blocks the UI thread (02 · isolates).

Examples

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


class FadeSlideIn extends StatefulWidget {
  final Widget child;
  const FadeSlideIn({super.key, required this.child});
  @override State<FadeSlideIn> createState() => _FadeSlideInState();
}
class _FadeSlideInState extends State<FadeSlideIn> with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, duration: const Duration(milliseconds: 400));
  late final Animation<double> _fade =
      CurvedAnimation(parent: _c, curve: Curves.easeOut);           // curve
  late final Animation<Offset> _slide = Tween(
    begin: const Offset(0, 0.1), end: Offset.zero,                  // tween<Offset>
  ).animate(CurvedAnimation(parent: _c, curve: Curves.easeOut));


  @override
  void initState() { super.initState(); _c.forward(); }             // play
  @override
  void dispose() { _c.dispose(); super.dispose(); }                 // MUST dispose
```

```
  @override
  Widget build(BuildContext context) {
    // *Transition widgets consume the Animation directly (efficient):
    return FadeTransition(
      opacity: _fade,
      child: SlideTransition(position: _slide, child: widget.child),
    );
    // Equivalent with AnimatedBuilder (custom rebuild from value):
    // return AnimatedBuilder(animation: _c, builder: (_, child) =>
    //   Opacity(opacity: _fade.value, child: child), child: widget.child);
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not disposing the controller	Leak; keeps frames scheduled	`dispose()` in `State.dispose`
No `TickerProvider` mixin	Controller needs `vsync`	`SingleTickerProviderStateMixin`
`addListener` + `setState` for large subtrees	Rebuilds too much each tick	`AnimatedBuilder` / `*Transition` (scoped) + `child`
Applying curve after tween incorrectly	Wrong easing	`CurvedAnimation(parent, curve)` then `.animate`
Heavy work in the per-tick builder	UI-thread jank	Keep builder cheap; precompute

Best Practices

- Create the controller in `initState` (with `vsync: this`) and **dispose** it.
- Use `CurvedAnimation` for easing, `Tween` for range/type, and `AnimatedBuilder` / `*Transition` to rebuild only the animated part (pass a `child` for static content).
- Keep per-tick builders **cheap**; isolate with `RepaintBoundary` (06_animation_performance.md).
- Drive chaining/reversal off `status`; prefer implicit animations for simple cases (02_implicit_animations.md).

Performance

Every tick rebuilds/repaints the animated subtree — keep it small and cheap; use `*Transition` (rebuilds only the transition) and `RepaintBoundary`. Heavy builders or effects cause jank (21 · jank_and_raster).

Advantages / Disadvantages

- **+** Composable (controller/tween/curve), any type/range, precise control, integrates with scheduler.
- **−** Boilerplate + lifecycle (dispose) vs implicit animations; easy to over-rebuild if misused.

Interview Questions

1. 🟢 **What does an `AnimationController` do?** — Drives a value (default 0→1) over a duration via a vsync `Ticker`, with play/reverse/repeat/status control.
2. 🟢 **What are `Tween` and `Curve` for?** — `Tween` maps 0→1 to a range/type (`Offset` / `Color` /…); `Curve` shapes the timing (easing) via `CurvedAnimation`.

3. 🟡 **Why must you dispose the controller?** — It holds a `Ticker`; not disposing leaks it and keeps frames scheduled (wasting battery).
4. 🟡 **`AnimatedBuilder` vs `addListener` + `setState`?** — `AnimatedBuilder` rebuilds only its builder (scoped, efficient, with a `child` slot); listener+`setState` rebuilds the whole `State` subtree.
5. 🟡 **What does `status` give you?** — Animation state (`forward` / `completed` / `reverse` / `dismissed`) for chaining/reversing.
6. 🔴 **How does animation tie into the scheduler?** — The `Ticker` fires each frame in the transient phase, advancing the controller value (09 · scheduler_and_vsync).
7. 🔴 **How do you keep a value-driven animation efficient?** — Scope rebuilds with `AnimatedBuilder` / `*Transition` + `child`, keep the builder cheap, and isolate with `RepaintBoundary`.

Senior Engineer Tips

- Prefer `*Transition` widgets (`FadeTransition` / `SlideTransition` / `ScaleTransition`) — they rebuild only the transition and are efficient by design.
- Always pass a `child` to `AnimatedBuilder` so static content isn't rebuilt each tick.
- Use one controller to drive multiple tweens/curves (fade + slide together) rather than several controllers.

Architect Perspective

The controller→tween/curve→value→rebuild model is Flutter's uniform animation substrate; understanding it lets you build any motion and reason about its cost. It underpins implicit widgets, transitions, staggering, and physics — and its performance (scoped rebuilds, isolation) ties directly to the rendering pipeline (Module 09, 21).

Summary

- Animation = controller (0→1 via ticker) + tween (range/type) + curve (easing) → `Animation<T>` → rebuild.
- Dispose the controller; use `CurvedAnimation` / `Tween` and `AnimatedBuilder` / `*Transition` (with `child`).
- Keep per-tick builds cheap and isolated; foundation for all later animation techniques.

Revision Notes

- `AnimationController(vsync, duration)` (needs `TickerProvider`, dispose!); `forward/reverse/repeat/status`.
- `Tween(begin,end).animate(CurvedAnimation(parent, curve))` → `Animation<T>`.
- Consume via `AnimatedBuilder` / `*Transition` (+ `child`), not whole-subtree `setState`.

- Ticks = transient scheduler phase; keep builders cheap + `RepaintBoundary`.

Practice Questions

1. How do controller, tween, and curve compose into an animation?
2. Why and where do you dispose the controller?
3. Why prefer `AnimatedBuilder` / `*Transition` over `addListener` + `setState`?

Coding Questions

1. Fade+slide a widget in over 400ms with `easeOut`, one controller, disposed.
2. Animate a `Color` via `ColorTween` and drive a container.
3. Chain a second animation when the first `completed` (via `status`).

Mini Project

Reusable entrance animation (Flutter): Build a `FadeSlideIn` widget wrapping any child, driven by one `AnimationController` + `CurvedAnimation` + `Tween<Offset>`/opacity, using `*Transition` / `AnimatedBuilder` with a `child`, disposed correctly. Acceptance: smooth fade+slide; one controller; disposed; scoped rebuilds; runs.

Implicit Animations (`AnimatedFoo`, `TweenAnimationBuilder`)

> Implicit animations animate **automatically** when a property changes: give an `AnimatedContainer` / `AnimatedOpacity` /etc. a new target value + `duration`, and Flutter tweens from old to new for you — no controller, no lifecycle, ideal for simple state-driven transitions.

Introduction

Implicit animation widgets (`AnimatedContainer`, `AnimatedOpacity`, `AnimatedPositioned`, `AnimatedAlign`, `AnimatedDefaultTextStyle`, and the general `TweenAnimationBuilder` / `AnimatedSwitcher`) animate to whatever new value you build with. This file covers them and when they're the right, low-boilerplate choice.

Why this concept exists

Many animations are just "smoothly go from the old value to the new one" on a state change (size, color, opacity, position). Managing a controller for that is overkill. Implicit widgets do it declaratively: change the value, set a duration, done — perfectly matching the declarative UI model.

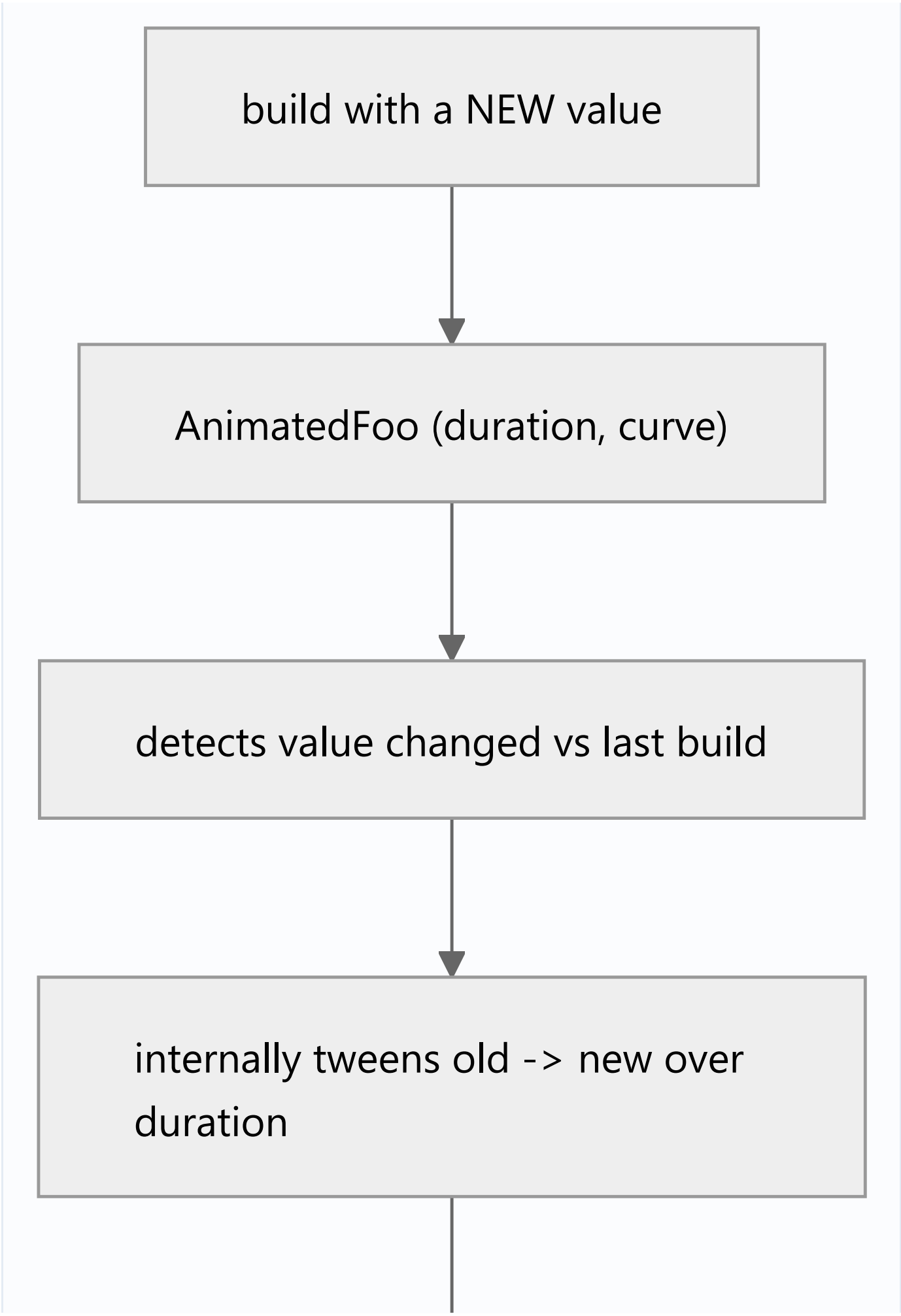
Real-world analogy

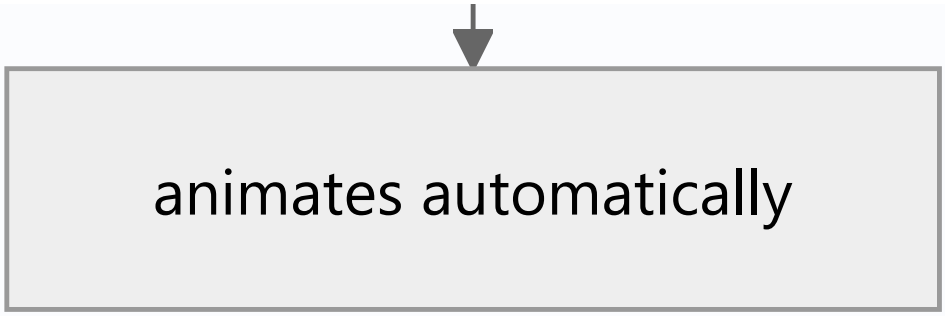
Implicit animation is a **thermostat with a smooth dial**: you set a new target temperature (property), and it eases there over time automatically — you don't manually turn the dial degree by degree (no controller).

Problem Statement

A card should smoothly resize/recolor when expanded, an icon should cross-fade when toggled, and a badge should animate its count — all from simple state changes, no controllers. You'll use `AnimatedContainer`, `AnimatedSwitcher`, and `TweenAnimationBuilder`.

Internal Working


build with a NEW value
AnimatedFoo (duration, curve)
detects value changed vs last build
internally tweens old -> new over duration



- `AnimatedFoo` **widgets**: hold animatable props + `duration` (+ optional `curve`). On rebuild with a **different** value, they animate from the previous to the new value internally (their own controller). Examples: `AnimatedContainer` (size/color/padding/decoration), `AnimatedOpacity`, `AnimatedPositioned` (in a `Stack`), `AnimatedAlign`, `AnimatedPadding`, `AnimatedDefaultTextStyle`, `AnimatedPhysicalModel`.
- `AnimatedSwitcher`: cross-fades (or custom-transitions) between **different child widgets** when the child changes (use a `Key` to signal "different").
- `TweenAnimationBuilder<T>`: animates to a target value via a builder — implicit animation for arbitrary values/one-offs (e.g., animate a number, a custom paint value) without a controller.
- **Trigger**: just build with a new value (via `setState`/state management) — the animation is automatic.
- **When to use**: simple, single-shot, state-driven transitions. For **control** (play/reverse/repeat/precise sequencing) use **explicit** animations (03_explicit_animations.md).

Memory Representation

Implicit widgets manage their own controller/ticker internally (disposed for you). Lightweight; no manual lifecycle (08 · dispose_and_leaks).

Compiler Behavior

Not applicable.

Runtime Behavior

On rebuild, the widget compares the new value to its last and animates the delta over `duration`/`curve`; interrupting with a newer value re-targets smoothly. `AnimatedSwitcher` animates on child-key change.

Flutter Engine Behavior

Each animating frame repaints the widget; keep the animated subtree small and isolate if expensive (06_animation_performance.md).

Dart VM Behavior

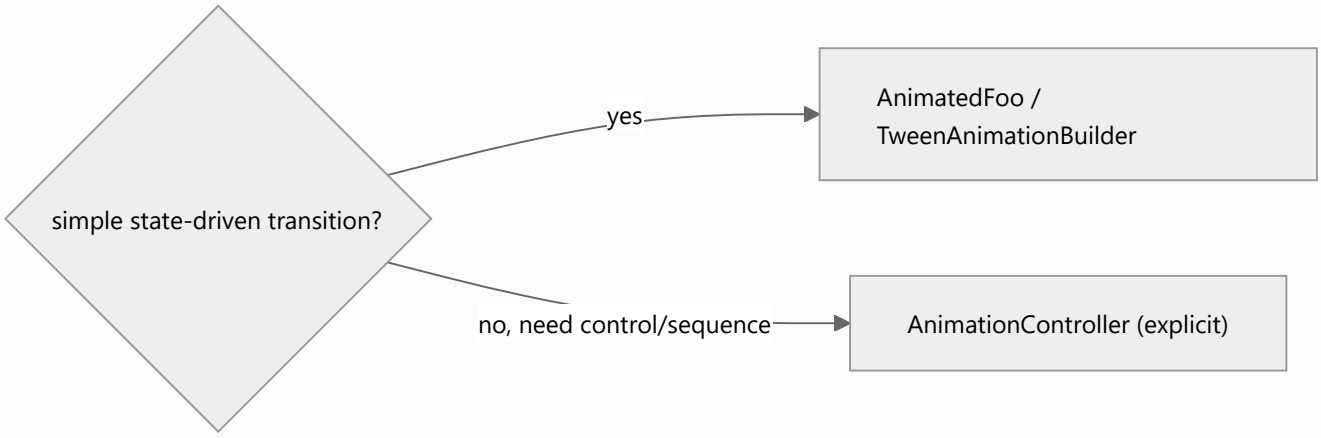
Not applicable.

Examples

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


class ImplicitDemo extends StatefulWidget {
  const ImplicitDemo({super.key});
  @override State<ImplicitDemo> createState() => _ImplicitDemoState();
}
class _ImplicitDemoState extends State<ImplicitDemo> {
  bool _expanded = false;
  bool _dark = false;
  int _count = 0;


  @override
  Widget build(BuildContext context) {
    return Column(children: [
      // AnimatedContainer: animates size + color on state change (no controller)
      GestureDetector(
        onTap: () => setState(() => _expanded = !_expanded),
        child: AnimatedContainer(
          duration: const Duration(milliseconds: 300),
          curve: Curves.easeInOut,
          width: _expanded ? 200 : 100,
          height: _expanded ? 120 : 60,
          color: _dark ? Colors.indigo : Colors.teal,
        ),
      ),


      // AnimatedSwitcher: cross-fade between DIFFERENT children (key signals change)
      AnimatedSwitcher(
        duration: const Duration(milliseconds: 250),
```

```
        child: Icon(
          _dark ? Icons.dark_mode : Icons.light_mode,
          key: ValueKey(_dark),   // different key -> transition
          size: 48,
        ),
      ),
      Switch(value: _dark, onChanged: (v) => setState(() => _dark = v)),

      // TweenAnimationBuilder: animate an arbitrary value (a number) implicitly
      TweenAnimationBuilder<double>(
        tween: Tween(begin: 0, end: _count.toDouble()),
        duration: const Duration(milliseconds: 400),
        builder: (_, value, __) => Text('Count: ${value.toStringAsFixed(0)}'),
      ),
      ElevatedButton(onPressed: () => setState(() => _count += 10), child: const Text('+10')),
    ]);
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using explicit controllers for simple transitions	Needless boilerplate/lifecycle	Use implicit `AnimatedFoo`
No `Key` on `AnimatedSwitcher` children	Won't detect "different" child	Give distinct `Key` s
Expecting play/reverse/repeat control	Implicit is fire-on-change only	Use explicit animations
Animating a huge subtree implicitly	Repaint cost	Isolate; keep animated part small
Same value each build	No animation triggers	Ensure the value actually changes

Best Practices

- Use **implicit** widgets for simple, state-driven, single-shot transitions (size/color/opacity/position/child-swap) — least code.
- Give `AnimatedSwitcher` children distinct `Key` **s**; supply a `transitionBuilder` for custom effects.
- Use `TweenAnimationBuilder` to implicitly animate arbitrary values without a controller.
- Set a sensible **duration/curve** (~200–400ms, standard curves); keep the animated subtree small.
- Escalate to **explicit** animations only when you need control/sequencing/repetition.

Performance

Cheap for small subtrees; the widget manages its controller efficiently. Cost is the per-frame repaint of the animated subtree — keep it small, isolate expensive content (21 · jank_and_raster).

Advantages / Disadvantages

- **+** Minimal boilerplate, no lifecycle to manage, declarative, great for common transitions.
- **−** No fine control (play/reverse/repeat/precise timing), fire-on-change only, less suited to complex choreography.

Interview Questions

1. 🟢 **What is an implicit animation?** — A widget that animates automatically from the old to the new value when a property changes (e.g., `AnimatedContainer`), with no explicit controller.
2. 🟢 **How do you trigger one?** — Rebuild with a different value + a `duration`; the widget tweens the delta.
3. 🟡 **Implicit vs explicit — when each?** — Implicit for simple, single-shot, state-driven transitions (least code); explicit when you need control/sequencing/repetition.

4. 🟡 **How does `AnimatedSwitcher` know to animate?** — When its child changes identity (different `Key`), it transitions (default cross-fade) between old and new.
5. 🟡 **What is `TweenAnimationBuilder` for?** — Implicitly animating an arbitrary value to a target via a builder, without managing a controller.
6. 🔴 **Do implicit widgets leak?** — No; they manage/dispose their internal controller — a benefit over hand-rolled controllers.
7. 🔴 **When do implicit animations *not* fit?** — Complex choreography, staggering, reversible/repeating, or physics/gesture-driven motion — use explicit/physics (03_explicit_animations.md, 05_physics_and_gesture_driven.md).

Senior Engineer Tips

- Reach for implicit first — most UI motion (expand/collapse, color/opacity/position changes, icon swaps) needs nothing more.
- Remember the `Key` rule for `AnimatedSwitcher`; without a changing key it won't animate.
- Keep durations consistent with your design system; wrap common implicit transitions in reusable widgets.

Architect Perspective

Implicit animations align motion with the declarative model (animate on value change) and eliminate controller lifecycle bugs — ideal for the bulk of everyday UI motion. Reserving explicit/physics for genuinely complex cases keeps animation code minimal and maintainable across a design system (Module 25).

Summary

- Implicit widgets (`AnimatedFoo` / `AnimatedSwitcher` / `TweenAnimationBuilder`) animate automatically on value/child change — no controller, no lifecycle.
- Best for simple, single-shot, state-driven transitions; give `AnimatedSwitcher` children keys.
- Escalate to explicit/physics for control, choreography, or gesture/physics-driven motion.

Revision Notes

- `AnimatedFoo(duration, curve)` animates old→new on rebuild (self-managed controller).
- `AnimatedSwitcher` (child key change → transition); `TweenAnimationBuilder<T>` (implicit arbitrary value).
- No play/reverse/repeat control → use explicit for that.
- Keep animated subtree small; ~200–400ms; standard curves.

Practice Questions

1. When choose implicit over explicit?
2. Why does `AnimatedSwitcher` need a changing `Key`?
3. What does `TweenAnimationBuilder` enable without a controller?

Coding Questions

1. Build an expand/collapse `AnimatedContainer` (size + color).
2. Cross-fade an icon with `AnimatedSwitcher` (keys) + a custom `transitionBuilder`.
3. Animate a counter number with `TweenAnimationBuilder`.

Mini Project

Implicit interactions (Flutter): Build a card that expands/recolors (`AnimatedContainer`), a theme icon that cross-fades (`AnimatedSwitcher`), and a stat that animates its number (`TweenAnimationBuilder`) — all driven by simple state, no controllers. Acceptance: smooth transitions on state change; correct switcher keys; no lifecycle management; runs.

Explicit Animations (Controllers, `AnimatedBuilder`, Transitions)

> Explicit animations give you a **controller you drive** — play, reverse, repeat, stop, chain on status — with the value consumed by `AnimatedBuilder` or `*Transition` widgets; use them when you need control, repetition, or coordination that implicit animations can't provide.

Introduction

Where implicit animations fire-on-change (02_implicit_animations.md), explicit animations put an `AnimationController` in your hands. This file covers driving controllers (forward/reverse/repeat), consuming values (`AnimatedBuilder` / `*Transition`), coordinating multiple animations from one controller, and status-driven chaining.

Why this concept exists

Real interactions need control: a loading spinner that **repeats**, a like button that plays **and reverses**, an animation that starts on a gesture and **chains** into another. Implicit widgets can't express these; explicit controllers can — with precise timing and lifecycle.

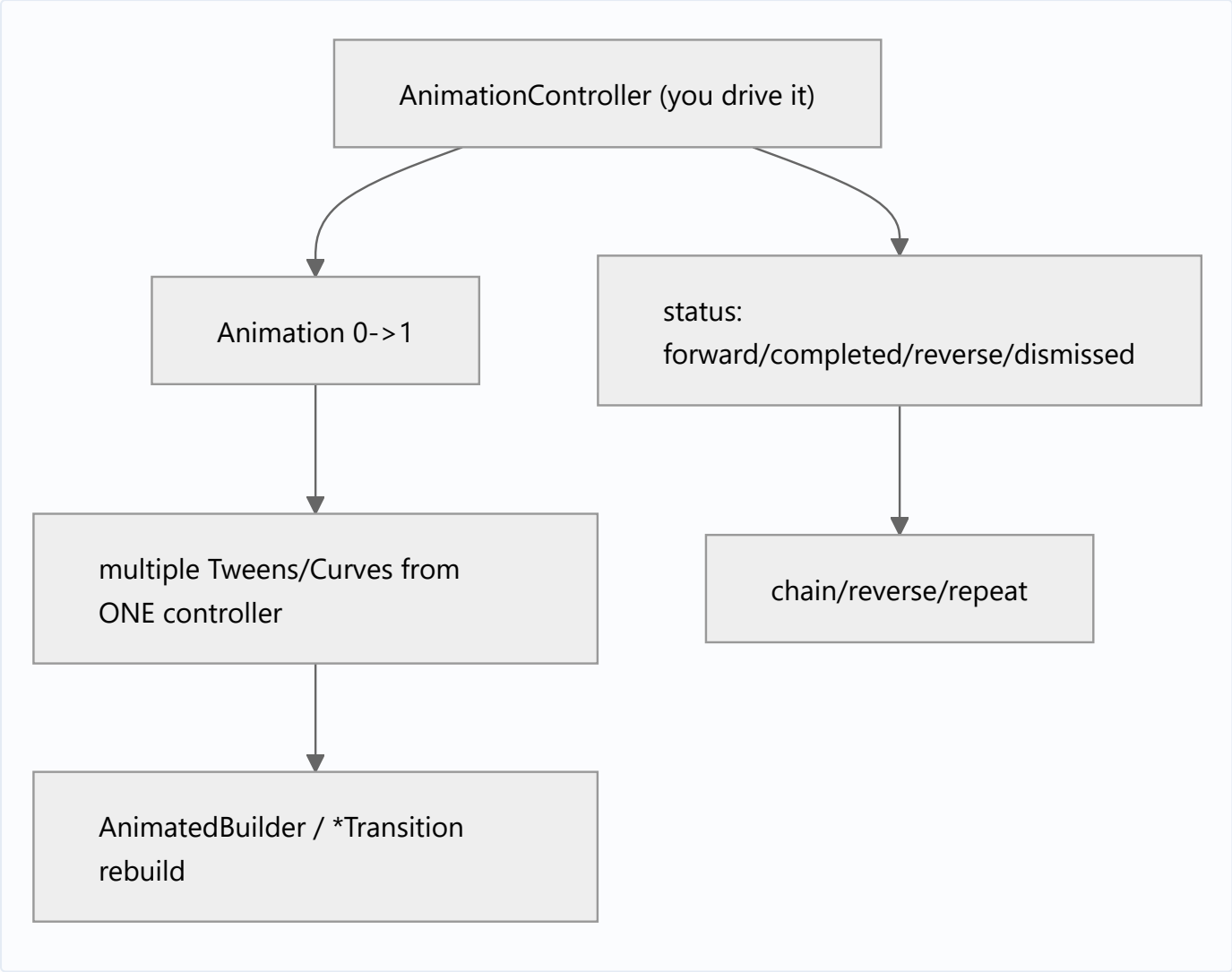
Real-world analogy

Implicit is a **thermostat** (set target, it eases there). Explicit is a **DJ at the controls**: you start, pause, rewind, loop, and cue the next track (chain) exactly when you want — full manual control.

Problem Statement

Build a repeating loading spinner, a like button that plays forward on tap and reverses on un-tap, and a two-step sequence (scale then fade) — all from explicit controllers. You'll drive `forward` / `reverse` / `repeat` and coordinate multiple tweens.

Internal Working



- **Drive the controller**: `forward()`, `reverse()`, `repeat(reverse: true)`, `stop()`, `animateTo()`, `fling()`. Control `duration` / `reverseDuration`; read/react to `status`.
- **Consume the value**:
  - `AnimatedBuilder(animation:, builder:, child:)` — rebuild the builder each tick from `.value` (pass static content via `child`).
  - `*Transition` (`FadeTransition`, `ScaleTransition`, `RotationTransition`, `SlideTransition`, `SizeTransition`) — efficient, purpose-built consumers of an `Animation`.
- **One controller, many tweens/curves**: drive scale + fade + color from a single controller with different `Tween` / `CurvedAnimation`s (coordinated, cheap).
- **Chaining**: use `status` listeners (`addStatusListener`) or `TweenSequence` for multi-step; reverse on completion, etc.
- **Lifecycle**: create in `initState`, **dispose** in `dispose`; use `SingleTickerProviderStateMixin` (or `TickerProviderStateMixin` for multiple controllers) (08 · dispose_and_leaks).

Memory Representation

Controller(s) + tickers held by `State`; dispose them (leak + wasted frames otherwise). `AnimatedBuilder`'s `child` avoids rebuilding static content (06_animation_performance.md).

Compiler Behavior

Not applicable.

Runtime Behavior

Each tick updates the value → consumers rebuild; `repeat` loops until stopped; `status` transitions drive your chaining logic. `dispose` stops the ticker.

Flutter Engine Behavior

Per-tick rebuild → repaint of the animated subtree; isolate with `RepaintBoundary` and keep the subtree small (21 · jank_and_raster).

Dart VM Behavior

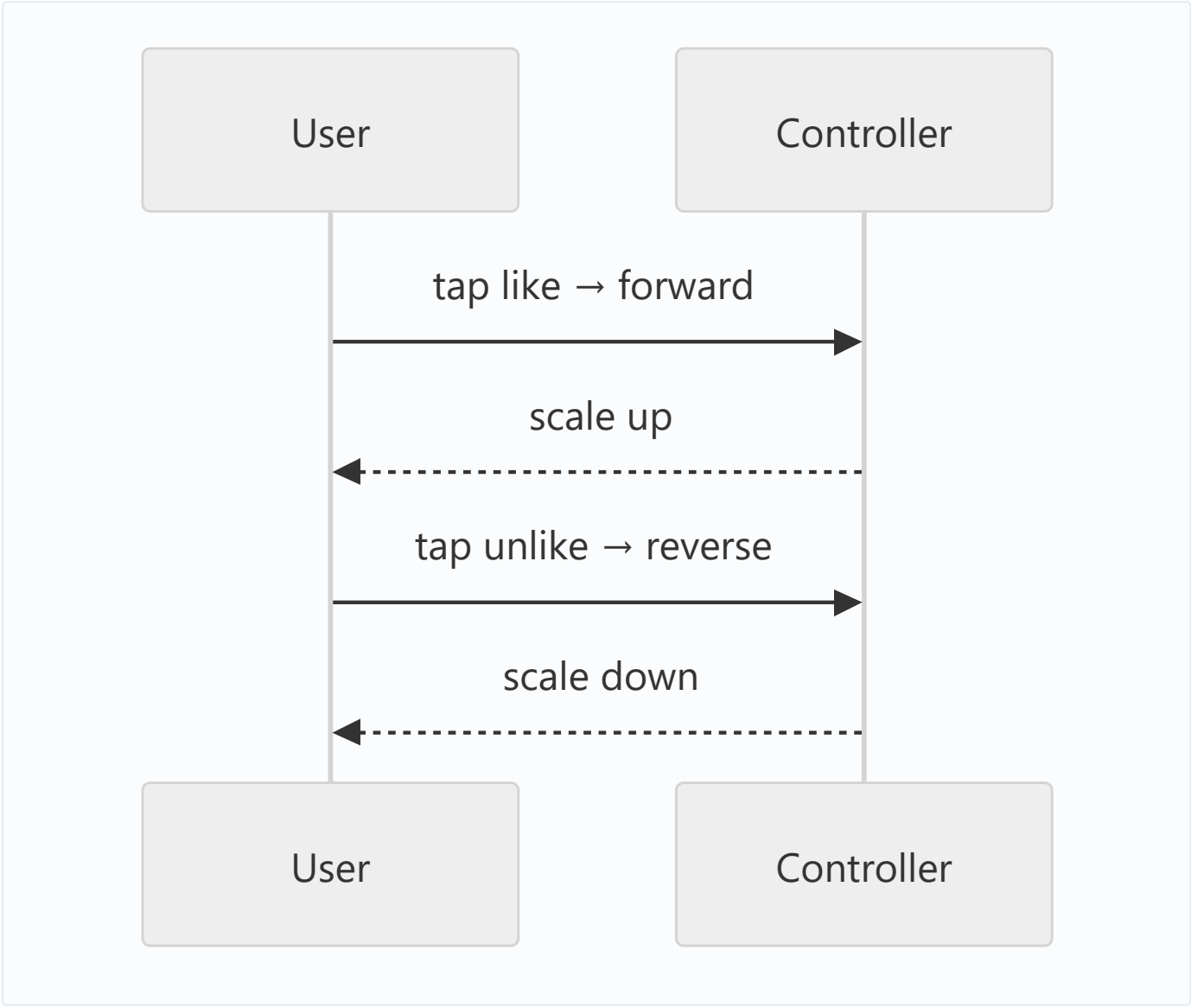
Not applicable.

Examples

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


// Repeating spinner (explicit, controlled)
class Spinner extends StatefulWidget {
  const Spinner({super.key});
  @override State<Spinner> createState() => _SpinnerState();
}
class _SpinnerState extends State<Spinner> with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, duration: const Duration(seconds: 1))..repeat();
  @override void dispose() { _c.dispose(); super.dispose(); }
  @override
  Widget build(BuildContext context) =>
      RotationTransition(turns: _c, child: const Icon(Icons.autorenew)); // efficient transition
}
```

```
// Like button: play forward on like, reverse on unlike (one controller, coordinated tweens)
class LikeButton extends StatefulWidget {
  const LikeButton({super.key});
  @override State<LikeButton> createState() => _LikeButtonState();
}
class _LikeButtonState extends State<LikeButton> with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, duration: const Duration(milliseconds: 300));
  late final Animation<double> _scale =
      Tween(begin: 1.0, end: 1.4).chain(CurveTween(curve: Curves.easeOut)).animate(_c);
  bool _liked = false;

  @override void dispose() { _c.dispose(); super.dispose(); }

  void _toggle() {
    setState(() => _liked = !_liked);
    _liked ? _c.forward() : _c.reverse(); // control direction
  }

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _toggle,
      child: AnimatedBuilder(
        animation: _c,
        builder: (_, child) => Transform.scale(scale: _scale.value, child: child),
        child: Icon(_liked ? Icons.favorite : Icons.favorite_border,
            color: _liked ? Colors.red : null),
      ),
    );
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not disposing controller(s)	Leak + endless frames	`dispose()` all controllers
Multiple controllers where one suffices	Overhead/complexity	Drive multiple tweens from one controller
`AnimatedBuilder` without `child`	Rebuilds static content each tick	Pass static content via `child`
`setState` per tick for big subtrees	Over-rebuild/jank	Use `AnimatedBuilder` / `*Transition`
`repeat()` left running off-screen	Wasted frames/battery	Stop when not visible
Wrong ticker mixin for N controllers	Errors	`TickerProviderStateMixin` for multiple

Best Practices

- Create controllers in `initState`, **dispose** in `dispose`; use the right ticker mixin (`Single...` vs `TickerProviderStateMixin`).
- Prefer `*Transition` widgets; use `AnimatedBuilder` + `child` for custom rebuilds; keep builders cheap.
- Drive **multiple coordinated tweens from one controller**; chain via `status` / `TweenSequence`.
- **Stop/repeat deliberately**; don't leave animations running off-screen.
- Isolate with `RepaintBoundary` for expensive/independent animations (21 · jank_and_raster).

Performance

Per-tick rebuild cost scales with the animated subtree — keep it small, use `child`, prefer `*Transition`, isolate with `RepaintBoundary`, and stop off-screen animations. Mind raster cost of animated effects (21 · jank_and_raster).

Advantages / Disadvantages

- + Full control (play/reverse/repeat/stop/chain), coordination, precise timing, reusable.
- − Boilerplate + lifecycle (dispose), easy to over-rebuild or leave running; more than simple cases need.

Interview Questions

1. 🟢 **When do you need explicit animations?** — When you need control (play/reverse/repeat/stop), coordination, chaining, or precise timing that implicit animations can't provide.
2. 🟢 **How do you consume a controller's value efficiently?** — `*Transition` widgets or `AnimatedBuilder` with a `child` (rebuild only the animated part).
3. 🟡 **How do you coordinate multiple animations?** — Drive multiple `Tween` / `CurvedAnimation`s from a single `AnimationController`.
4. 🟡 **How do you chain animations?** — `addStatusListener` (react to `completed` / `dismissed`) or `TweenSequence` for multi-step sequences.
5. 🟡 **Which ticker mixin for multiple controllers?** — `TickerProviderStateMixin` (use `SingleTickerProviderStateMixin` for one).
6. 🔴 **How do you keep an explicit animation efficient?** — Scope rebuilds (`*Transition` / `AnimatedBuilder` + `child`), keep builders cheap, isolate with `RepaintBoundary`, and stop when off-screen.
7. 🔴 **What lifecycle bugs are common?** — Not disposing controllers (leak + endless scheduled frames) and leaving `repeat()` running off-screen.

Senior Engineer Tips

- One controller, many tweens — coordinate related motion (scale+fade+color) from a single clock.
- Always pass a `child` to `AnimatedBuilder`; it's the difference between rebuilding a leaf vs a subtree each tick.
- Pause/stop animations that aren't visible (route changes, off-screen list items) to save frames/battery.

Architect Perspective

Explicit animations are the controllable tier for rich interactions and micro-interactions. Encapsulating them in reusable widgets (like/spinner/reveal) with correct lifecycle + scoped rebuilds keeps motion consistent and performant across a design system, building on the fundamentals and feeding staggering/physics (01_animation_fundamentals.md, 04_staggered_and_choreographed.md).

Summary

- Explicit = a controller you drive (forward/reverse/repeat/stop/chain), value consumed by `*Transition` / `AnimatedBuilder`.
- One controller drives many coordinated tweens; chain via `status` / `TweenSequence`.
- Dispose controllers, scope rebuilds (`child` /transition), stop off-screen; use for control/coordination beyond implicit.

Revision Notes

- Drive: `forward/reverse/repeat/stop/animateTo`; react to `status`; `Single` / `TickerProviderStateMixin`; dispose!
- Consume: `*Transition` (efficient) or `AnimatedBuilder` + `child`.
- One controller → many tweens/curves; chain via status/ `TweenSequence`.
- Isolate + keep builders cheap; stop off-screen animations.

Practice Questions

1. Why can implicit widgets not build a reversible like animation?
2. How do you coordinate scale + fade from one controller?
3. Why pass a `child` to `AnimatedBuilder`?

Coding Questions

1. Build a repeating spinner with `RotationTransition`.
2. Build a like button that forwards/reverses on tap (coordinated scale + icon).
3. Chain scale→fade via a `status` listener or `TweenSequence`.

Mini Project

Micro-interaction kit (Flutter): Build reusable explicit animations — a repeating loader, a reversible like button (one controller, coordinated tweens), and a two-step reveal (`TweenSequence`) — with correct lifecycle, scoped rebuilds (`*Transition` / `AnimatedBuilder` + `child`), and off-screen stopping. Acceptance: full control (play/reverse/repeat/chain); disposed; scoped/efficient; runs at 60fps.

Staggered & Choreographed Animations

> Choreograph multiple animations from **one controller** by giving each an `Interval` (a slice of the 0→1 timeline) so they start/end at different times — producing staggered entrances, sequenced steps, and coordinated motion; `AnimatedList` / `TweenSequence` help with list and multi-phase choreography.

Introduction

Complex motion = many animations timed relative to each other. Rather than juggling multiple controllers, drive them from **one** controller and carve the timeline with `Interval`s (and `Curve`s). This file covers staggered animations, `TweenSequence`, and animated lists.

Why this concept exists

Choreography (a card's icon, title, and button entering in sequence; a list staggering in) needs shared timing. One controller + intervals guarantees synchronization and simplicity; separate controllers drift and complicate lifecycle. It's the standard technique for polished, multi-part motion.

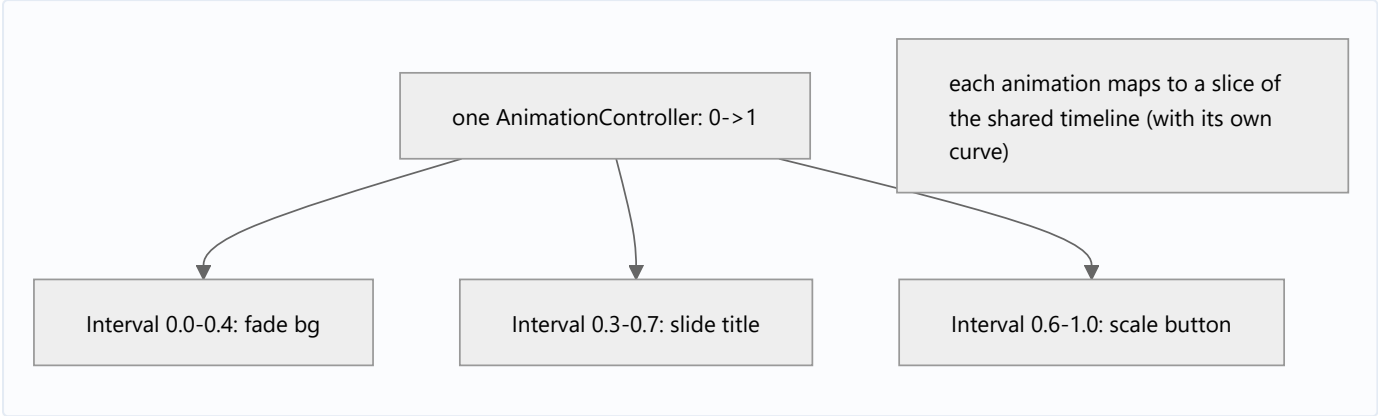
Real-world analogy

A **synchronized dance** to one piece of music (the controller's timeline). Each dancer (animation) has a **cue** — when to start and stop within the song (`Interval`) — so the whole routine is coordinated to a single clock.

Problem Statement

A card should animate in as: background fade (0–40%), title slide (30–70%), then button scale (60–100%) — all from one controller; and a list should stagger items in as they're added. You'll use `Interval`s and `AnimatedList`.

Internal Working



- `Interval(begin, end, curve:)`: a `Curve` that is flat (0) before `begin`, animates `begin` → `end`, and flat (1) after — so a `CurvedAnimation(parent: controller, curve: Interval(...))` runs only during its slice of the shared 0→1.
- **Stagger**: assign overlapping/sequential intervals to each animation from **one controller**; `forward()` plays the whole choreography.
- `TweenSequence`: chain multiple tweens with weights over one animation (multi-phase single value, e.g., up-then-down, color A→B→C).
- `AnimatedList` / `SliverAnimatedList`: animate item insertion/removal (`insertItem` / `removeItem` + a per-item transition builder) — staggered/animated lists (07 · scrolling_and_slivers).
- **List stagger**: give each item an interval based on its index (delay ∝ index) for a cascading entrance.

Memory Representation

One controller + several derived animations (lightweight). `AnimatedList` tracks items + in-flight transitions. Dispose the controller (08 · dispose_and_leaks).

Compiler Behavior

Not applicable.

Runtime Behavior

Each tick advances the shared value; each animation reflects its interval slice (flat outside it). `AnimatedList` runs per-item enter/exit transitions on insert/remove.

Flutter Engine Behavior

All parts repaint during their active slices; isolate heavy parts and keep item counts/effects reasonable (06_animation_performance.md).

Dart VM Behavior

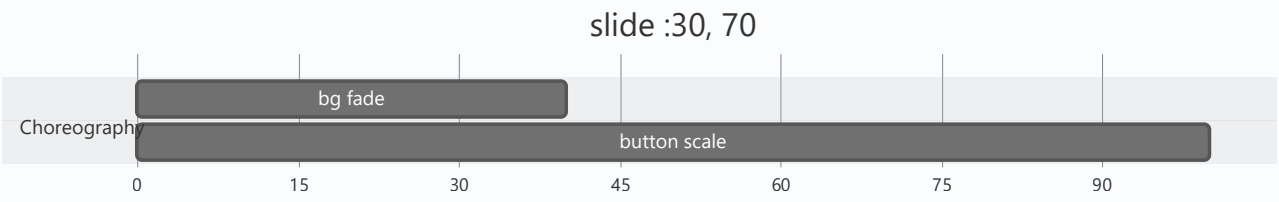
Not applicable.

Examples

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

// Staggered card entrance from ONE controller via Intervals
class StaggeredCard extends StatefulWidget {
  const StaggeredCard({super.key});
  @override State<StaggeredCard> createState() => _StaggeredCardState();
}
class _StaggeredCardState extends State<StaggeredCard> with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, duration: const Duration(milliseconds: 900))..forward();

  late final Animation<double> _bgFade =
      CurvedAnimation(parent: _c, curve: const Interval(0.0, 0.4, curve: Curves.easeOut));
  late final Animation<Offset> _titleSlide = Tween(begin: const Offset(-0.2, 0), end: Offset.zero)
      .animate(CurvedAnimation(parent: _c, curve: const Interval(0.3, 0.7, curve: Curves.easeOut)));
  late final Animation<double> _btnScale =
      CurvedAnimation(parent: _c, curve: const Interval(0.6, 1.0, curve: Curves.easeOutBack));

  @override void dispose() { _c.dispose(); super.dispose(); }

  @override
  Widget build(BuildContext context) {
    return FadeTransition(
      opacity: _bgFade,                                  // 0-40%
      child: Column(mainAxisSize: MainAxisSize.min, children: [
        SlideTransition(position: _titleSlide,           // 30-70%
            child: const Text('Welcome', style: TextStyle(fontSize: 24))),
        ScaleTransition(scale: _btnScale,                // 60-100%
            child: ElevatedButton(onPressed: () {}, child: const Text('Start'))),
      ]),
    );
  }
}
```

```
// TweenSequence: multi-phase single value (grow then settle)
final bounce = TweenSequence<double>([
  TweenSequenceItem(tween: Tween(begin: 1.0, end: 1.3), weight: 50),
  TweenSequenceItem(tween: Tween(begin: 1.3, end: 1.0), weight: 50),
]);
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Multiple controllers for one choreography	Drift, complex lifecycle	One controller + `Interval`s
Intervals summing/overlapping wrongly	Janky timing	Plan intervals on the 0→1 timeline
Manually rebuilding an animated list	Bugs/no exit animations	Use `AnimatedList` insert/remove
Staggering a huge list all at once	Perf hit	Stagger only visible/bounded items
Not disposing the controller	Leak	Dispose in `dispose`

Best Practices

- Drive choreography from **one controller** using `Interval`**s** (with per-slice curves); map the timeline deliberately.
- Use `TweenSequence` for multi-phase single values (bounce, A→B→C color).
- Use `AnimatedList` / `SliverAnimatedList` for animated insert/remove; stagger by index for entrances (bound to visible items).
- Keep total durations tasteful (~600–1000ms for entrances); dispose the controller; isolate heavy parts.

Performance

One controller is efficient; cost is the sum of active parts repainting. Stagger only bounded/visible items; isolate expensive parts with `RepaintBoundary`; keep durations reasonable (21 · jank_and_raster, 06_animation_performance.md).

Advantages / Disadvantages

- **+** Synchronized, polished multi-part motion from one clock; simple lifecycle; list enter/exit animations.
- **−** Interval planning overhead; over-choreography feels slow/busy; list staggering costs if unbounded.

Interview Questions

1. 🟢 **How do you stagger multiple animations?** — Drive them from one `AnimationController` and give each an `Interval` slice of the 0→1 timeline (with its own curve).
2. 🟢 **Why one controller instead of several?** — Guarantees synchronization to a single clock and simplifies lifecycle; separate controllers drift.
3. 🟡 **What does `Interval` do?** — It's a curve that's flat before `begin` and after `end`, animating only within its slice — timing an animation to part of the shared timeline.
4. 🟡 **What is `TweenSequence` for?** — Chaining multiple tweens (with weights) over one animation for multi-phase single values (e.g., grow-then-settle).
5. 🟡 **How do you animate list insert/remove?** — `AnimatedList` / `SliverAnimatedList` with `insertItem` / `removeItem` and a per-item transition builder.
6. 🔴 **How do you stagger a list without hurting performance?** — Stagger only bounded/visible items (delay ∝ index within the viewport), not all items at once.
7. 🔴 **How do you plan overlapping intervals?** — Map each animation's begin/end on the 0→1 timeline (they can overlap for smooth handoff); ensure the total reads as coordinated.

Senior Engineer Tips

- Sketch the timeline (Gantt-style) first; choreography is a timing-design problem, then a coding one.
- Overlap intervals slightly for smooth handoffs (title starts before bg finishes) rather than strictly sequential.
- For lists, stagger the *visible* entrance only; unbounded staggering janks and feels sluggish.

Architect Perspective

Choreography via one controller + intervals is the scalable way to build complex, synchronized motion (onboarding, hero screens, list entrances) with clean lifecycle. It composes the fundamentals into polished sequences and, done tastefully and bounded, delivers premium feel without perf cost (01_animation_fundamentals.md, 03_explicit_animations.md).

Summary

- Choreograph multiple animations from one controller using `Interval` timeline slices (+curves).
- `TweenSequence` for multi-phase single values; `AnimatedList` for animated insert/remove; stagger lists by index (bounded).
- Plan the timeline, overlap for smooth handoffs, keep it tasteful, dispose the controller.

Revision Notes

- One controller + `Interval(begin,end,curve)` per animation = staggered/choreographed.
- `TweenSequence` (weighted multi-phase single value); `AnimatedList` / `SliverAnimatedList` (insert/remove).
- Stagger lists by index but only visible/bounded; overlap intervals for handoffs.
- Plan timeline first; dispose controller; isolate heavy parts.

Practice Questions

1. How do intervals let one controller stagger many animations?
2. When use `TweenSequence` vs multiple interval'd animations?
3. How do you stagger a list performantly?

Coding Questions

1. Build a staggered card entrance (bg/title/button) from one controller via intervals.
2. Implement a grow-then-settle bounce with `TweenSequence`.
3. Animate list insertions with `AnimatedList`, staggering visible items by index.

Mini Project

Choreographed onboarding (Flutter): Build an onboarding screen where several elements animate in staggered from one controller (intervals + curves), plus an `AnimatedList` section that staggers item entrances. Sketch the timeline in comments. Acceptance: single controller; coordinated intervals; animated list insert/remove; bounded stagger; disposed; smooth.

Physics-Based & Gesture-Driven Animation

> Natural motion follows physics (springs, friction, gravity) rather than fixed durations; drive an `AnimationController` with a `Simulation` (e.g., `SpringSimulation`) or tie its value directly to a **gesture** (drag) so animations feel responsive and organic — like `Draggable`, `Dismissible`, and swipe-to-dismiss sheets.

Introduction

Duration-based curves feel mechanical for interactive motion. This file covers **physics simulations** (spring/friction/gravity via `controller.animateWith(Simulation)`) and **gesture-driven** animation (map a drag to the controller value; `fling()` with velocity), plus the built-in gesture widgets (`Draggable`, `Dismissible`).

Why this concept exists

When users fling/drag, motion should respond to their **velocity** and settle naturally (spring), not play a fixed 300ms tween. Physics makes interactions feel alive and predictable to the hand; gesture-driving makes the UI track the finger 1:1, then continue with momentum.

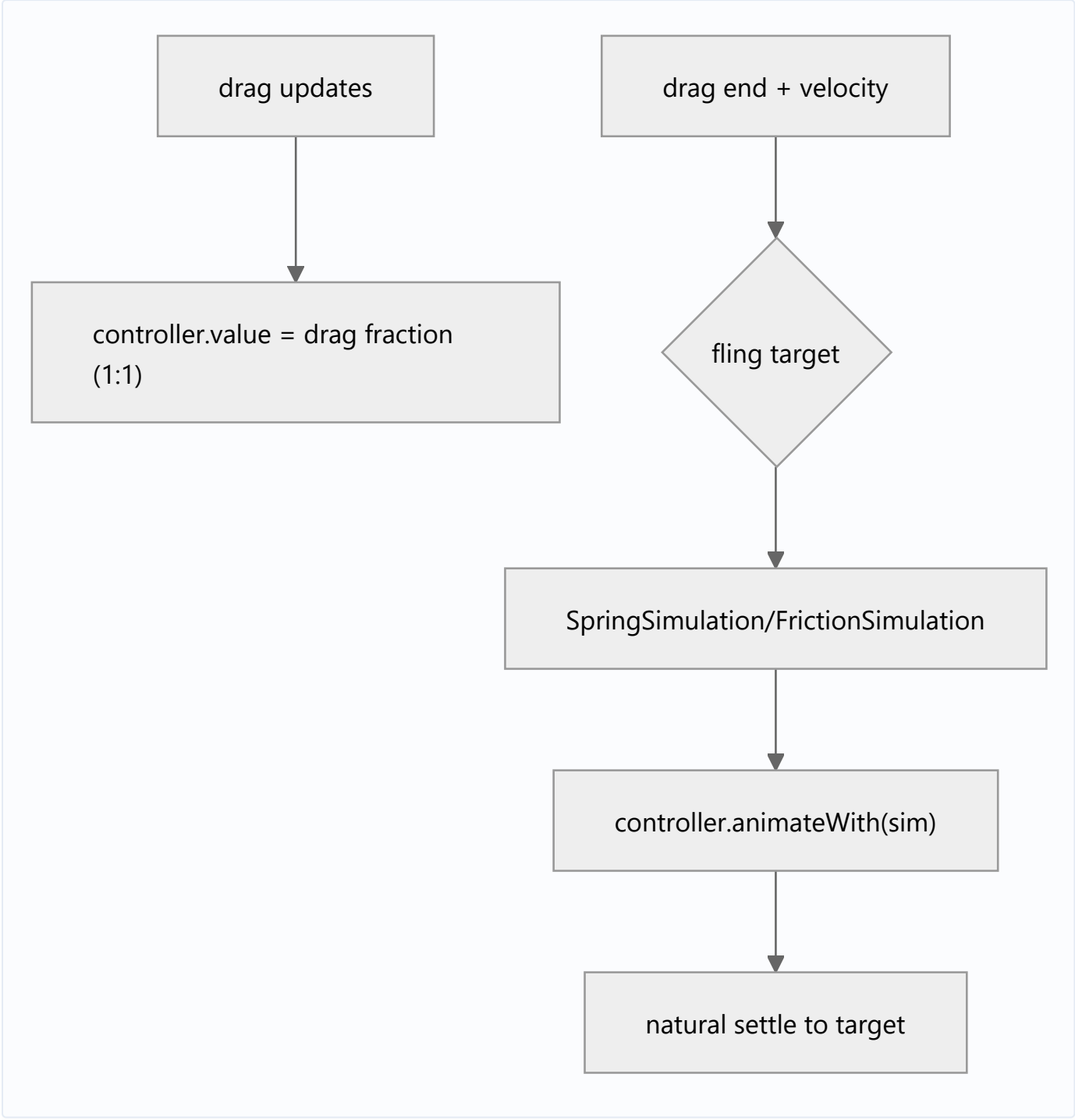
Real-world analogy

A **real drawer**: you push it and it glides with momentum, then eases to a stop (friction); a **spring-loaded lid** snaps back when released. Physics animations reproduce that felt-realism; gesture-driving is the drawer following your hand exactly while you hold it.

Problem Statement

A bottom sheet should follow the drag 1:1, then **fling** to open/closed based on release velocity and **spring** into place; and a card should be swipe-to-dismiss. You'll drive a controller with a spring simulation and from gesture velocity.

Internal Working



- **Simulations**: a `Simulation` defines position/velocity over time. `controller.animateWith(simulation)` drives the value by physics instead of a fixed duration.
  - `SpringSimulation(SpringDescription, start, end, velocity)` : spring toward a target with mass/stiffness/damping — natural snap/bounce.
  - `FrictionSimulation` : decelerating glide (momentum scrolling feel).
  - `GravitySimulation` : fall under gravity.
- **Gesture-driven**: in `onPanUpdate` / `onVerticalDragUpdate` , set `controller.value` from the drag delta (1:1 tracking); on `onDragEnd` , use the **velocity** to `fling()` or pick a target and

`animateWith(spring)` to settle.

- `fling(velocity:)`: drive the controller with a spring-like simulation using the release velocity.
- **Built-in gesture widgets**: `Draggable` / `DragTarget` (drag-and-drop), `Dismissible` (swipe-to-dismiss with velocity), `DraggableScrollableSheet` (draggable sheets) — encapsulate common gesture+physics patterns.
- `ScrollPhysics` (`BouncingScrollPhysics` / `ClampingScrollPhysics`) is physics-driven scrolling — a Strategy (05 · strategy).

Memory Representation

Controller + simulation objects (lightweight); dispose the controller. Gesture recognizers are managed by the gesture widgets (08 · dispose_and_leaks).

Compiler Behavior

Not applicable.

Runtime Behavior

During a drag the value tracks the finger; on release the simulation runs to completion (position/velocity → settle). `fling` uses velocity for a natural throw. Interruption (new drag) re-targets.

Flutter Engine Behavior

Per-frame value updates repaint the moving widget; isolate/keep it cheap (06_animation_performance.md).

Dart VM Behavior

Not applicable.

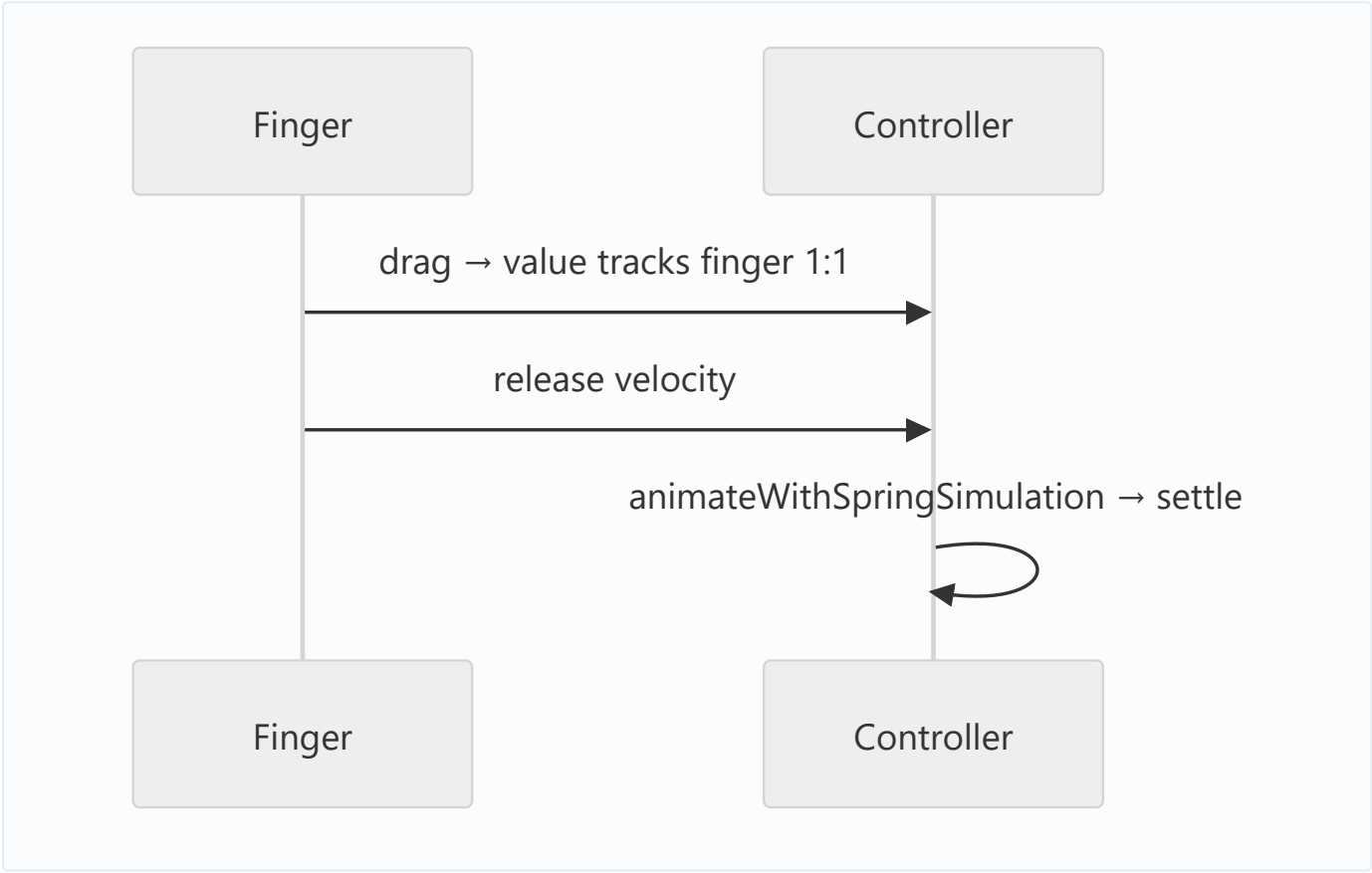
Examples

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

// Gesture-driven, spring-settled draggable panel (vertical)
class SpringPanel extends StatefulWidget {
  const SpringPanel({super.key});
  @override State<SpringPanel> createState() => _SpringPanelState();
```

```
}
class _SpringPanelState extends State<SpringPanel> with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, lowerBound: 0, upperBound: 1)..value = 0;
  @override void dispose() { _c.dispose(); super.dispose(); }

  void _onDragUpdate(DragUpdateDetails d, double height) {
    _c.value -= d.primaryDelta! / height; // 1:1 follow the finger
  }

  void _onDragEnd(DragEndDetails d, double height) {
    // Use release velocity to decide + spring toward a target:
    final v = -d.velocity.pixelsPerSecond.dy / height;
    final target = (_c.value + v * 0.2) > 0.5 ? 1.0 : 0.0;
    final sim = SpringSimulation(
      const SpringDescription(mass: 1, stiffness: 200, damping: 20),
      _c.value, target, v,
    );
    _c.animateWith(sim); // natural spring settle
  }

  @override
  Widget build(BuildContext context) {
    final height = MediaQuery.of(context).size.height;
    return GestureDetector(
      onVerticalDragUpdate: (d) => _onDragUpdate(d, height),
      onVerticalDragEnd: (d) => _onDragEnd(d, height),
      child: AnimatedBuilder(
        animation: _c,
        builder: (_, child) => Align(
          alignment: Alignment(0, 1 - _c.value * 2), // map value -> position
          child: child,
        ),
        child: Container(height: 200, color: Colors.teal),
      ),
    );
  }
}

// Built-in: swipe-to-dismiss (velocity-aware) — no manual physics needed
```

```
Widget dismissible(Widget child, VoidCallback onDismissed) => Dismissible(
      key: const ValueKey('item'),
      onDismissed: (_) => onDismissed(),
      child: child,
    );
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Fixed-duration tween for fling/drag	Feels mechanical/unresponsive	Physics (spring/friction) + velocity
Ignoring release velocity	Unnatural settle/decision	Use `d.velocity` for fling/target
Not clamping controller bounds	Out-of-range values	Set `lowerBound` / `upperBound`; clamp
Not disposing controller	Leak	Dispose
Reinventing swipe-to-dismiss	Bugs	Use `Dismissible` / `Draggable` /sheets
Heavy per-frame builder during drag	Jank	Keep builder cheap; isolate

Best Practices

- Use **physics simulations** (`SpringSimulation` / `Friction` / `fling`) for interactive/release motion; reserve fixed-duration curves for non-interactive transitions.
- **Track gestures 1:1** into the controller value during drag; use **release velocity** to fling/choose a target and **spring** to settle.
- Prefer **built-in gesture widgets** (`Draggable` , `Dismissible` , `DraggableScrollableSheet` , `ScrollPhysics`) before hand-rolling.
- Clamp controller bounds; dispose; keep the drag builder cheap/isolated.

Performance

Physics/gesture animations repaint per frame during interaction — keep the moving subtree small/cheap and isolated. Velocity-based settling avoids overlong tweens. Built-ins are optimized (21 · jank_and_raster).

Advantages / Disadvantages

- + Natural, responsive, velocity-aware motion; 1:1 gesture tracking; realistic settle; built-ins cover common cases.
- − More complex than duration tweens; tuning spring params; per-frame cost during interaction; needs care to feel right.

Interview Questions

1. 🟢 **Why use physics instead of fixed-duration curves for interactions?** — Interactive motion should respond to the user's velocity and settle naturally (spring/friction), which fixed durations can't express.
2. 🟢 **How do you drive a controller with physics?** — `controller.animateWith(simulation)` (e.g., `SpringSimulation`) instead of `forward()` with a duration.
3. 🟡 **How do you make an animation follow a drag?** — Set `controller.value` from the drag delta (1:1) in `onDragUpdate` ; on end, use velocity to `fling` /spring to a target.
4. 🟡 **What does `fling(velocity:)` do?** — Drives the controller with a velocity-based (spring-like) simulation for a natural throw.
5. 🟡 **Which built-in widgets encapsulate gesture+physics?** — `Draggable` / `DragTarget` , `Dismissible` , `DraggableScrollableSheet` , and `ScrollPhysics` (bouncing/clamping).
6. 🔴 **What is a `Simulation` and its common types?** — An object defining position/velocity over time; `SpringSimulation` (spring), `FrictionSimulation` (glide/decelerate), `GravitySimulation` (fall).
7. 🔴 **How do you keep gesture-driven animation smooth?** — Cheap/isolated per-frame builder, clamp bounds, use velocity to avoid long tweens, and reuse built-ins where possible.

Senior Engineer Tips

- Reach for built-ins (`Dismissible` / `DraggableScrollableSheet`) first — they encode the physics/gesture nuances you'd otherwise get wrong.
- For custom sheets/panels, track 1:1 then `animateWith(SpringSimulation)` using release velocity — this is the recipe for "feels native."
- Tune `SpringDescription` (stiffness/damping) to taste; over-bouncy springs feel toy-like.

Architect Perspective

Physics/gesture-driven motion is what makes interactions feel native and responsive (sheets, dismiss, drag-drop, scroll). Building it from `Simulation` + gesture-to-value (or leaning on built-ins) is a UX-quality decision; it composes with the animation fundamentals and scroll physics, and its per-frame cost must respect the budget (01_animation_fundamentals.md, 21).

Summary

- Physics simulations (spring/friction/gravity via `animateWith`) give natural, velocity-aware motion.
- Gesture-drive by mapping drag→controller value (1:1) and flinging/springing on release using velocity.
- Prefer built-in gesture widgets; clamp bounds; dispose; keep the moving subtree cheap/isolated.

Revision Notes

- `controller.animateWith(Simulation)`: `SpringSimulation` / `FrictionSimulation` / `GravitySimulation`; `fling(velocity)`.
- Gesture: `onDragUpdate` → set `value` (1:1); `onDragEnd` → velocity → target + spring.
- Built-ins: `Draggable` / `Dismissible` / `DraggableScrollableSheet` / `ScrollPhysics`.
- Clamp bounds; dispose; cheap/isolated builder; use velocity to settle.

Practice Questions

1. Why do fixed-duration tweens feel wrong for a fling?
2. How do you tie an animation value to a drag and settle with a spring?
3. Which built-ins should you use before hand-rolling gesture physics?

Coding Questions

1. Build a draggable panel that follows the finger and springs to open/closed by velocity.
2. Use `Dismissible` for swipe-to-dismiss list items.
3. Implement a `fling`-based toggle using release velocity.

Mini Project

Spring bottom sheet (Flutter): Build a draggable bottom sheet that tracks the drag 1:1 and, on release, flings/springs (`SpringSimulation` with release velocity) to open or closed; plus a swipe-to-dismiss list via `Dismissible`. Acceptance: 1:1 tracking; velocity-aware spring settle; built-in dismiss; disposed; smooth (60fps).

Animation Performance

> Animations run every frame, so they're unforgiving of cost: keep the per-tick rebuild **small** (`AnimatedBuilder` + `child` , `*Transition`), **isolate** the moving part (`RepaintBoundary`), avoid **raster-expensive** effects (opacity/clip/blur = `saveLayer`), address **shader jank** (Impeller), and **stop off-screen** animations.

Introduction

An animation that drops frames ruins the effect it was meant to create. This file consolidates the performance rules for animation, tying the animation system to the rendering-pipeline and performance modules (Module 09, Module 21).

Why this concept exists

Because animations rebuild/repaint ~60–120×/second, small per-frame costs multiply into jank. The fixes are specific: scope rebuilds, isolate repaints, cut raster cost, precompile shaders, and don't animate when unseen. Knowing them keeps motion smooth on real devices.

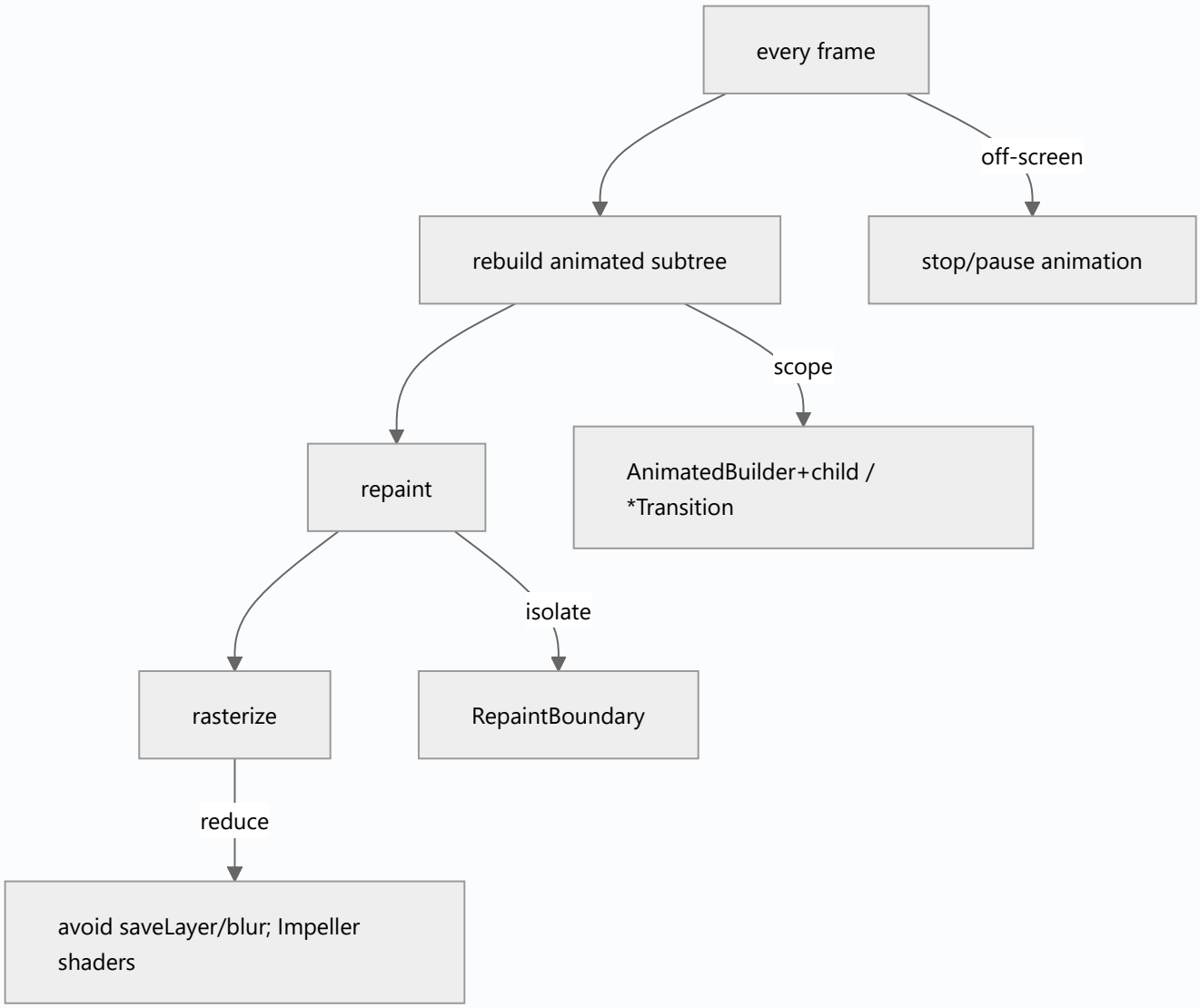
Real-world analogy

A **flipbook**: if each page takes too long to draw, the animation stutters. You keep pages simple (cheap builds), only redraw the moving character (isolation), avoid elaborate shading (raster cost), and stop flipping when no one's watching (off-screen).

Problem Statement

A spinning, blurred badge over a complex screen janks (raster-bound), and a list-item animation rebuilds the whole tile. You'll scope rebuilds, isolate repaints, cut raster cost, and confirm 60fps in release.

Internal Working



- **Scope the rebuild**: use `*Transition` widgets (rebuild only the transition) or `AnimatedBuilder` **with a** `child` (static content passed via `child` isn't rebuilt). Never `addListener` + `setState` on a big subtree (01_animation_fundamentals.md).
- **Isolate the repaint**: wrap the animating widget (and cache the expensive-static background) in `RepaintBoundary` so a moving part doesn't repaint neighbors (09 · compositing, 21 · jank_and_raster).
- **Cut raster cost**: animating **opacity/clips/blur** uses `saveLayer` (offscreen buffers) — prefer `FadeTransition` (still `saveLayer` but scoped), tint paints, or bake effects; limit blur area/radius. Custom painters: correct `shouldRepaint` (09 · paint_phase).
- **Shader jank**: first-run animation stutter is shader compilation (Skia) → **Impeller** precompiles (21 · jank_and_raster).
- **Stop off-screen**: pause/stop controllers when the widget isn't visible (route change, off-screen list item) to save frames/battery; dispose on removal.

- **Profile**: check `rasterDuration` vs `buildDuration` during the animation on a low-end release build (21 · profiling_and_frame_budget).

Memory Representation

Controllers/tickers + any `saveLayer`/boundary GPU buffers. Off-screen running animations waste CPU/GPU/battery; boundaries cost GPU memory (use where measured) (09 · compositing).

Compiler Behavior

Impeller precompiles shaders; `const` reduces rebuild allocation.

Runtime Behavior

Each tick: rebuild → paint → raster of the animated subtree; costs compound over frames. Off-screen `repeat()` keeps scheduling frames until stopped.

Flutter Engine Behavior

Raster-thread cost (effects/shaders) is separate from UI-thread rebuild cost — an animation can be raster-bound even with a trivial build (09 · rasterization).

Dart VM Behavior

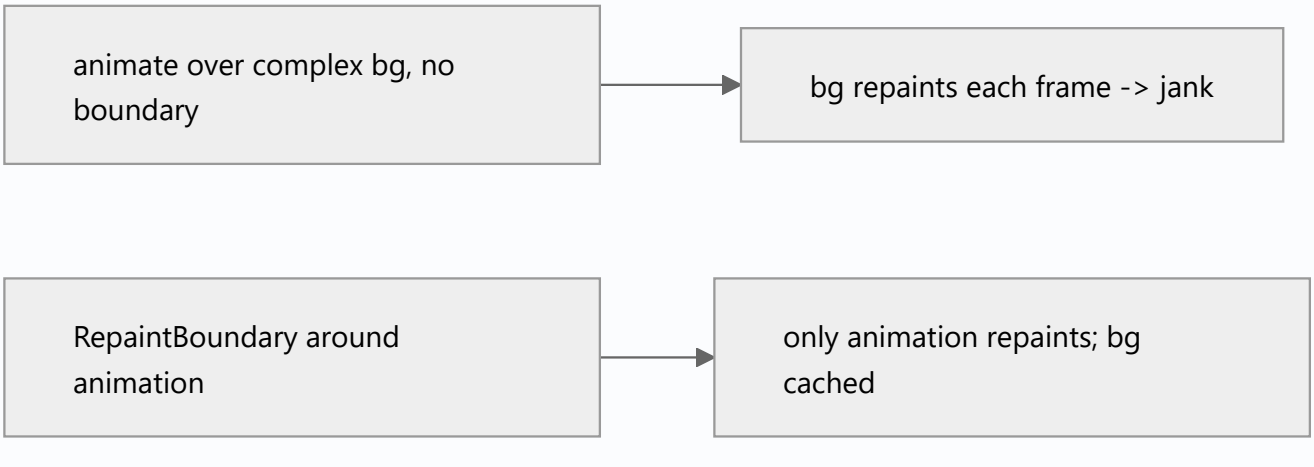
Heavy per-tick Dart work blocks the UI thread → dropped frames (02 · isolates).

Examples

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


class PerfAnimation extends StatefulWidget {
  const PerfAnimation({super.key});
  @override State<PerfAnimation> createState() => _PerfAnimationState();
}
class _PerfAnimationState extends State<PerfAnimation> with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, duration: const Duration(seconds: 1))..repeat();
  @override void dispose() { _c.dispose(); super.dispose(); }


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

```
    return Stack(children: [
      const RepaintBoundary(child: _ExpensiveStaticBackground()), // cache static
      // Isolate the animated part; AnimatedBuilder rebuilds only the transform, child stays put:
      RepaintBoundary(
        child: AnimatedBuilder(
          animation: _c,
          builder: (_, child) => Transform.rotate(angle: _c.value * 6.28, child: child),
          child: const Icon(Icons.refresh, size: 48), // NOT rebuilt each tick
        ),
      ),
    ]);
  }
}
class _ExpensiveStaticBackground extends StatelessWidget {
  const _ExpensiveStaticBackground();
  @override
  Widget build(BuildContext context) => const DecoratedBox(
      decoration: BoxDecoration(gradient: LinearGradient(colors: [Colors.indigo, Colors.teal])),
      child: SizedBox.expand());
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`addListener` + `setState` on big subtree	Whole subtree rebuilds each tick	`AnimatedBuilder` + `child` / `*Transition`
No `child` in `AnimatedBuilder`	Static content rebuilt each tick	Pass it via `child`
No `RepaintBoundary` around animation	Neighbors repaint	Isolate the moving part
Heavy opacity/clip/blur animation	`saveLayer` raster cost	Reduce/bake; limit area; Impeller
First-run stutter blamed on rebuilds	Shader compilation	Impeller / warm-up
`repeat()` running off-screen	Wasted frames/battery	Stop/pause when not visible
Not profiling in release	Miss raster/shader reality	Profile low-end release

Best Practices

- **Scope rebuilds**: `*Transition` or `AnimatedBuilder` + `child`; never `setState` a big subtree per tick.
- **Isolate** the animating part (and cache expensive static) with `RepaintBoundary`.
- **Minimize raster-expensive effects** (opacity/clip/blur); bake/limit them; correct `shouldRepaint`; adopt **Impeller** for shader jank.
- **Stop/pause off-screen** animations; **dispose** controllers.
- **Profile the animation** in release on a low-end device (`buildDuration` vs `rasterDuration`).

Performance

The rules attack each frame phase: rebuild (scope), paint/composite (isolate), raster (cheap effects/Impeller). Off-screen stopping saves battery. Verify with frame timings during the animation (21 · profiling_and_frame_budget).

Advantages / Disadvantages

- **+** Smooth 60/120fps motion; battery-friendly; scalable across many animations.
- **−** Requires discipline (isolation/scoping/effect budgeting); `RepaintBoundary` GPU-memory cost; profiling effort.

Interview Questions

1. 🟢 **Why are animations sensitive to cost?** — They rebuild/repaint every frame (~60–120×/s), so small per-tick costs multiply into jank.
2. 🟢 **How do you scope an animation's rebuild?** — `*Transition` widgets or `AnimatedBuilder` with a `child` so only the animated part rebuilds.

3. 🟡 **Why wrap an animation in `RepaintBoundary`?** — To isolate its repaints into its own layer so it doesn't repaint neighbors (and expensive static content can be cached).
4. 🟡 **Why is animating opacity/blur expensive?** — They use `saveLayer` (offscreen buffers) on the raster thread; minimize/bake them.
5. 🟡 **What causes first-run animation stutter, and the fix?** — Shader compilation (Skia); fix with Impeller (precompiled shaders).
6. 🔴 **How can an animation be janky with a trivial build?** — It's raster-bound (effects/shaders) — check `rasterDuration`, not just `buildDuration`.
7. 🔴 **Why stop off-screen animations?** — A running/`repeat()` controller keeps scheduling frames, wasting CPU/GPU/battery when nothing's visible.

Senior Engineer Tips

- Default recipe: `AnimatedBuilder` + `child` (or `*Transition`) **inside** a `RepaintBoundary`, over a cached static background — solves most animation jank.
- Profile the *specific animation* in release; attribute to UI vs raster before optimizing.
- Pause animations on route push / off-screen list items; it's an easy battery/perf win.

Architect Perspective

Animation performance is where the animation system meets the rendering pipeline: scope rebuilds (build phase), isolate repaints (compositing), budget raster (effects/shaders/Impeller), and stop when unseen. Baking these into reusable animation widgets keeps motion premium *and* performant across the app (Module 09, Module 21).

Summary

- Keep per-tick rebuilds small (`AnimatedBuilder` + `child` / `*Transition`), isolate with `RepaintBoundary`, minimize raster-heavy effects, fix shader jank (Impeller), stop off-screen, dispose.
- Profile the animation in release (UI vs raster) and verify 60/120fps.
- These tie the animation system to the pipeline/performance modules.

Revision Notes

- Scope: `*Transition` / `AnimatedBuilder` + `child` (never big-subtree `setState`).
- Isolate: `RepaintBoundary` (animation + cache static bg).
- Raster: avoid/limit opacity/clip/blur (`saveLayer`); `shouldRepaint`; Impeller for shader jank.
- Stop off-screen; dispose controllers; profile animation in release (build vs raster).

Practice Questions

1. What's the default recipe for a smooth animation over a complex background?
2. How can an animation be raster-bound despite a cheap build?
3. Why stop animations that aren't visible?

Coding Questions

1. Convert a `setState`-per-tick animation to `AnimatedBuilder` + `child` inside a `RepaintBoundary`.
2. Reduce an animated blur's raster cost; measure `rasterDuration`.
3. Stop a repeating animation when its route is not on top.

Mini Project — Module capstone

Smooth animated screen (Flutter): Combine techniques — an implicit expand card, an explicit staggered entrance, and a gesture-driven spring sheet — each scoped (`*Transition` / `AnimatedBuilder` + `child`), isolated (`RepaintBoundary`), effect-budgeted, and stopped off-screen. Profile in release (UI vs raster) to confirm 60fps. Acceptance: scoped/isolated animations; raster budget respected; off-screen stopping; disposed; measured 60fps.