# 07 · Widgets

Flutter Practice Notes

## Contents

# 07 · Widgets

## Introduction

Widgets are Flutter's building blocks — *everything* is a widget: layout, styling, gestures, even the app itself. This module is the practical catalog: how to lay out, size, scroll, style, and compose widgets, and how to build your own. It builds directly on the three-trees model from Module 06.

## Why this module exists

Knowing *what widget to reach for* and *how layout/constraints work* is the day-to-day skill of Flutter UI development. Most layout frustration ("unbounded height", "overflow", "why won't this expand?") comes from not understanding constraints — which this module makes concrete.

## Module map

| # | File | Topic | Level |
|---|---|---|---|
| 1 | 01_widget_categories.md | The kinds of widgets and how to navigate the catalog | 🟢 |
| 2 | 02_layout_flex.md | `Row` / `Column` / `Flex` / `Expanded` / `Flexible` / `Spacer`, alignment | 🟢 |
| 3 | 03_constraints_and_sizing.md | Constraints go down, sizes go up; `Container` / `SizedBox` / `ConstrainedBox` | 🔵 |
| 4 | 04_stack_and_positioning.md | `Stack` / `Positioned` / `Align` / `Alignment` (overlap layout) | 🟢 |
| 5 | 05_scrolling_and_slivers.md | `ListView` / `GridView` / `CustomScrollView` /slivers | 🔵 |
| 6 | 06_text_and_theming.md | `Text` / `TextStyle` / `Theme` / `TextTheme` | 🟢 |
| 7 | 07_images_and_assets.md | `Image`, assets, network, caching, `pubspec` | 🟢 |
| 8 | 08_input_and_forms.md | `TextField` / `Form` /validation/buttons/gestures | 🔵 |
| 9 | 09_custom_composite_widgets.md | Building your own widgets by composition | 🔵 |

> **Cross-references:** The layout *pipeline* (constraints/layout/paint internals) is Module 09. Custom drawing is Module 23. Responsive/adaptive layout are Modules 24/25. Animations are Module 22.

## Prerequisites

06 Flutter Fundamentals — especially the three trees, stateless/stateful, and `BuildContext`.

## What you'll be able to do after this module

- Build any static layout with `Row` / `Column` / `Stack` + flex.
- Predict and fix layout errors using the constraints model.
- Build performant scrolling lists/grids with builders and slivers.
- Style text and theme an app; load images/assets correctly.
- Build forms with validation and compose reusable custom widgets.

## Capstones

| Tier | Build |
|---|---|
| Beginner | A profile card (avatar + text + buttons) via composition. |
| Intermediate | A scrollable feed with a `ListView.builder` + custom item widget. |
| Advanced | A collapsing-header screen using `CustomScrollView` + slivers. |
| Enterprise | A reusable, themed **design-system widget kit** (buttons, cards, inputs). |

## Summary

Master layout via the constraints model, reach for the right catalog widget, and compose small custom widgets. This is the practical core of building Flutter UIs; rendering internals (Module 09) explain *why* it behaves as it does.

# Widget Categories (Navigating the Catalog)

> Widgets fall into a few functional categories — layout, single-child, multi-child, styling/painting, input, and app-structure — and knowing the categories turns "which widget?" from guesswork into a lookup.

## Introduction

Flutter ships hundreds of widgets, but they group into a handful of **roles**. This file maps those roles so you can navigate the catalog: layout vs content, single-child vs multi-child, Material vs Cupertino vs foundational, and stateless vs stateful.

## Why this concept exists

Beginners drown in the widget list. Categories give a mental index: to overlay things you want a *multi-child positioning* widget (`Stack`); to add padding you want a *single-child layout* widget (`Padding`); to scroll you want a *scrollable*. Knowing the role narrows hundreds of options to a few.

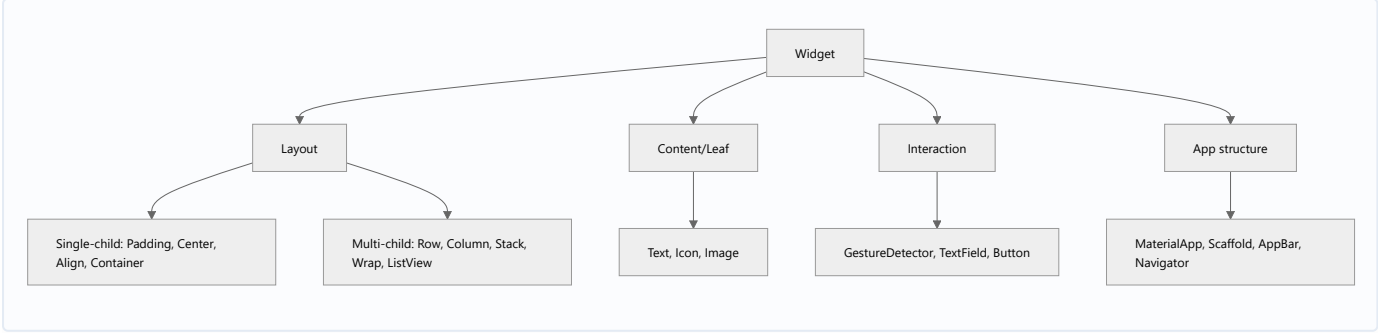
## Real-world analogy

Like a **hardware store organized by aisle**: fasteners, tools, plumbing, electrical. You don't memorize every SKU — you learn the aisles, then find the specific item. Widget categories are the aisles.

## Problem Statement

You need to: pad content, arrange items in a row, overlay a badge, scroll a list, and style text. Which category each? By the end you'll route each need to the right kind of widget instantly.

## Internal Working



| Category | Examples | Purpose |
|---|---|---|
| **Single-child layout** | `Padding`, `Center`, `Align`, `SizedBox`, `Container`, `ConstrainedBox` | Position/size one child |
| **Multi-child layout** | `Row`, `Column`, `Flex`, `Stack`, `Wrap`, `ListView`, `GridView` | Arrange multiple children |
| **Content/leaf** | `Text`, `Icon`, `Image`, `RichText` | Render actual content |
| **Painting/styling** | `DecoratedBox`, `Opacity`, `ClipRRect`, `Transform` | Visual effects |
| **Input/interaction** | `GestureDetector`, `TextField`, `ElevatedButton`, `InkWell` | Handle user input |
| **App structure** | `MaterialApp`, `Scaffold`, `AppBar`, `Navigator`, `Drawer` | App/screen scaffolding |
| **Sliver** | `SliverList`, `SliverAppBar`, `SliverGrid` | Scroll-effect building blocks |

Also orthogonal: **Material** ( `ElevatedButton` ) vs **Cupertino** ( `CupertinoButton` ) vs **foundational** ( `Text`, `Container` ); and **Stateless** vs **Stateful** (06).

## Memory Representation

Not applicable — categories are conceptual. (All are widgets → elements → render objects, 06.)

## Compiler Behavior / Runtime Behavior

Not special; category is a design lens, not a language feature.

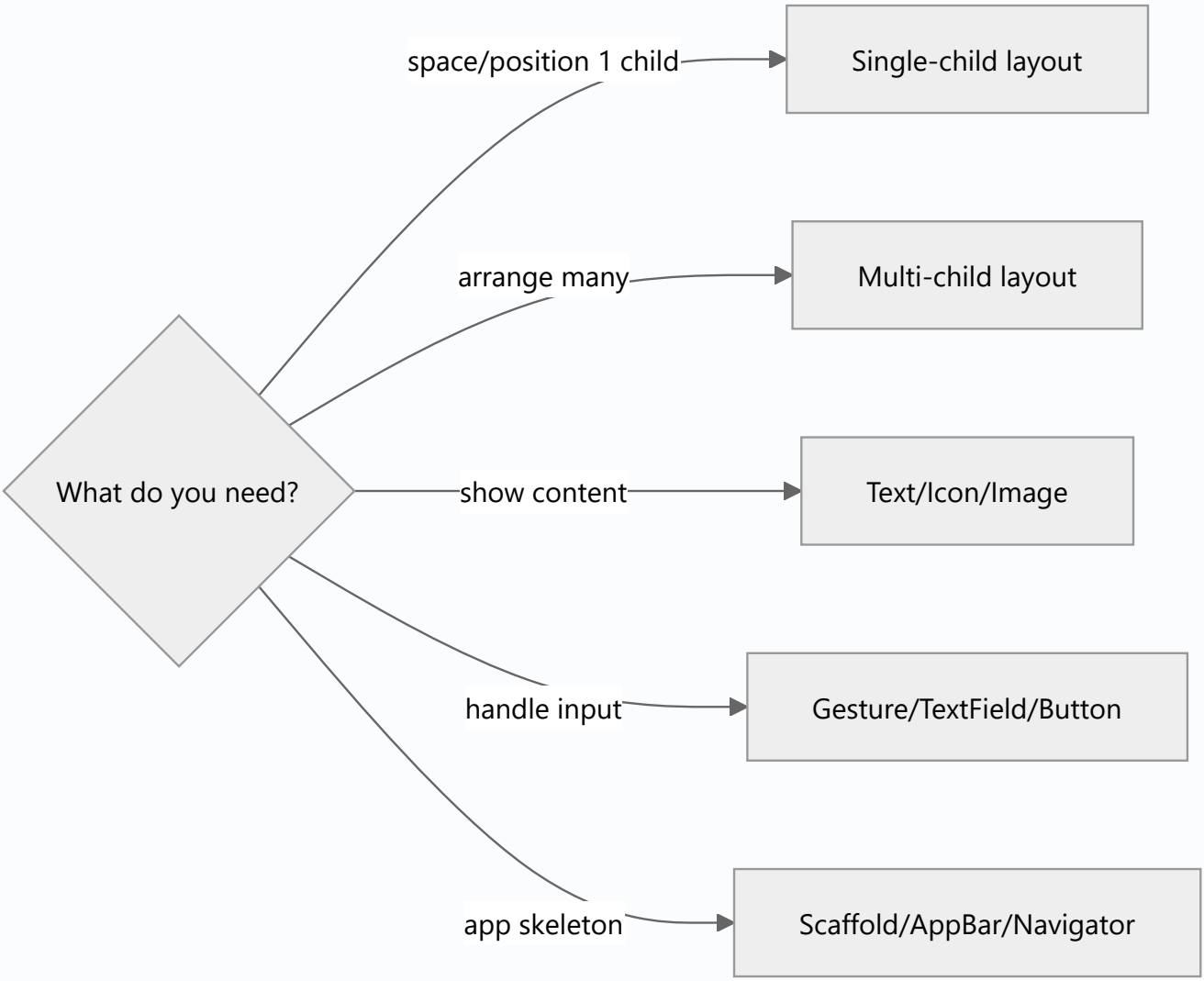
## Flutter Engine Behavior / Dart VM Behavior

Not applicable.

## Examples

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


class CategoryDemo extends StatelessWidget {
  const CategoryDemo({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(                          // app structure
      appBar: AppBar(title: const Text('Categories')), // app + content
      body: Padding(                          // single-child layout
        padding: const EdgeInsets.all(16),
        child: Column(                        // multi-child layout
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            const Text('Title', style: TextStyle(fontSize: 24)), // content
            const SizedBox(height: 8),        // single-child (spacing)
            Row(children: const [             // multi-child
              Icon(Icons.star),               // content/leaf
              SizedBox(width: 4),
              Text('4.5'),
            ]),
            GestureDetector(                  // interaction
              onTap: () {},
              child: const Text('Tap me'),
            ),
          ],
        ),
      ),
    );
  }
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Using `Container` for everything | Bloated, unclear intent | Use the specific widget ( `Padding` , `SizedBox` , `Align` ) |
| Reaching for a widget by memory | Slow, error-prone | Route by category first |
| Mixing Material/Cupertino inconsistently | Inconsistent UX | Pick a design language; adapt deliberately (Module 25) |
| Stateful where stateless works | Extra complexity | Default stateless (06) |

## Best Practices

- Route needs by **category** → then pick the specific widget.
- Prefer the **narrowest** widget that expresses intent ( `Padding` over `Container(padding:)` ).
- Keep a consistent design language (Material or Cupertino) per platform strategy.

- Compose small widgets rather than one giant `build`.

## Performance

Prefer `const` widgets; choose builder-based scrollables for long lists (05_scrolling_and_slivers.md); avoid unnecessarily heavy widgets (Module 21).

## Advantages / Disadvantages

- + A mental index that makes the huge catalog navigable; intent-revealing widget choices.
- − Categories overlap (e.g., `Container` spans several); requires familiarity to internalize.

## Interview Questions

1. 🟢 **"Everything is a widget" — what does that mean?** — Layout, styling, gestures, and app structure are all widgets composed into a tree; there's no separate "layout system" vs "view system."
2. 🟢 **Single-child vs multi-child layout widgets?** — Single-child (`Padding`, `Center`) wrap one child to position/size it; multi-child (`Row`, `Column`, `Stack`) arrange several.
3. 🟡 **When would you avoid `Container`?** — When a specific widget states intent better/cheaper (`Padding`, `SizedBox`, `Align`, `DecoratedBox`).
4. 🟡 **Material vs Cupertino vs foundational widgets?** — Material (Android/Material Design), Cupertino (iOS-style), foundational (design-agnostic like `Text`, `Container`, `Row`).
5. 🟡 **How do you decide which widget to use?** — Identify the *role* (layout/content/input/structure), then pick the specific widget in that category.
6. 🔴 **Why does Flutter favor many small widgets over big configurable ones?** — Composition (03): small single-purpose widgets compose flexibly and enable `const`/rebuild scoping.
7. 🔴 **How do categories map to the three trees?** — All categories are widgets → elements → render objects; layout widgets typically back onto layout render objects, leaves onto paint render objects.

## Senior Engineer Tips

- Learn the *aisles*, not the SKUs — you'll look up specifics but should know the category instantly.
- Prefer intent-revealing widgets; a screen full of nested `Container`s hides structure.
- Bookmark the official **Widget Catalog** and **Widget of the Week**; the catalog is your reference, not memorization.

## Architect Perspective

Consistent widget vocabulary and a chosen design language are the seeds of a design system ([Module 07 capstone], Module 25). Composing small, intent-revealing widgets keeps UIs maintainable and rebuild-efficient at scale.

## Summary

- Widgets group into layout (single/multi-child), content, styling, input, app-structure, and slivers.
- Route needs by category, then pick the narrowest intent-revealing widget.
- Everything is a widget; compose small pieces; prefer `const`.

## Revision Notes

- Categories: single-child layout / multi-child layout / content / styling / input / app-structure / sliver.
- Material vs Cupertino vs foundational; stateless vs stateful.
- Route by role → specific widget; prefer narrowest intent-revealing widget over `Container`.

## Practice Questions

1. Which category do you reach for to overlay a badge on an avatar?
2. Why prefer `Padding` over `Container(padding:)`?
3. How do widget categories relate to the three trees?

## Coding Questions

1. Build a screen using one widget from each major category, labeled in comments.
2. Refactor a `Container`-heavy widget into specific intent-revealing widgets.
3. Recreate a simple UI once with Material and once with Cupertino widgets.

## Mini Project

**Category tour (Flutter):** Build a single screen that intentionally uses each category (single/multi-child layout, content, styling, input, app-structure) with comments naming the category and why. Acceptance: correct category use; intent-revealing widgets; `const` where possible; app runs.

# Flex Layout (`Row`, `Column`, `Flex`, `Expanded`, `Flexible`, `Spacer`)

> `Row` and `Column` lay children along a main axis; `Expanded` / `Flexible` divide the leftover space; `mainAxisAlignment` / `crossAxisAlignment` position them — this is 80% of everyday Flutter layout.

## Introduction

`Row` (horizontal) and `Column` (vertical) are `Flex` widgets that arrange children along a **main axis** and size them along a **cross axis**. `Expanded` / `Flexible` / `Spacer` control how free space is shared. Mastering these + alignment covers most layouts.

## Why this concept exists

Linear arrangement (toolbars, lists of fields, button rows, stacked sections) is the most common UI need. Flex provides a declarative, constraint-driven way to distribute children and space without manual coordinates.

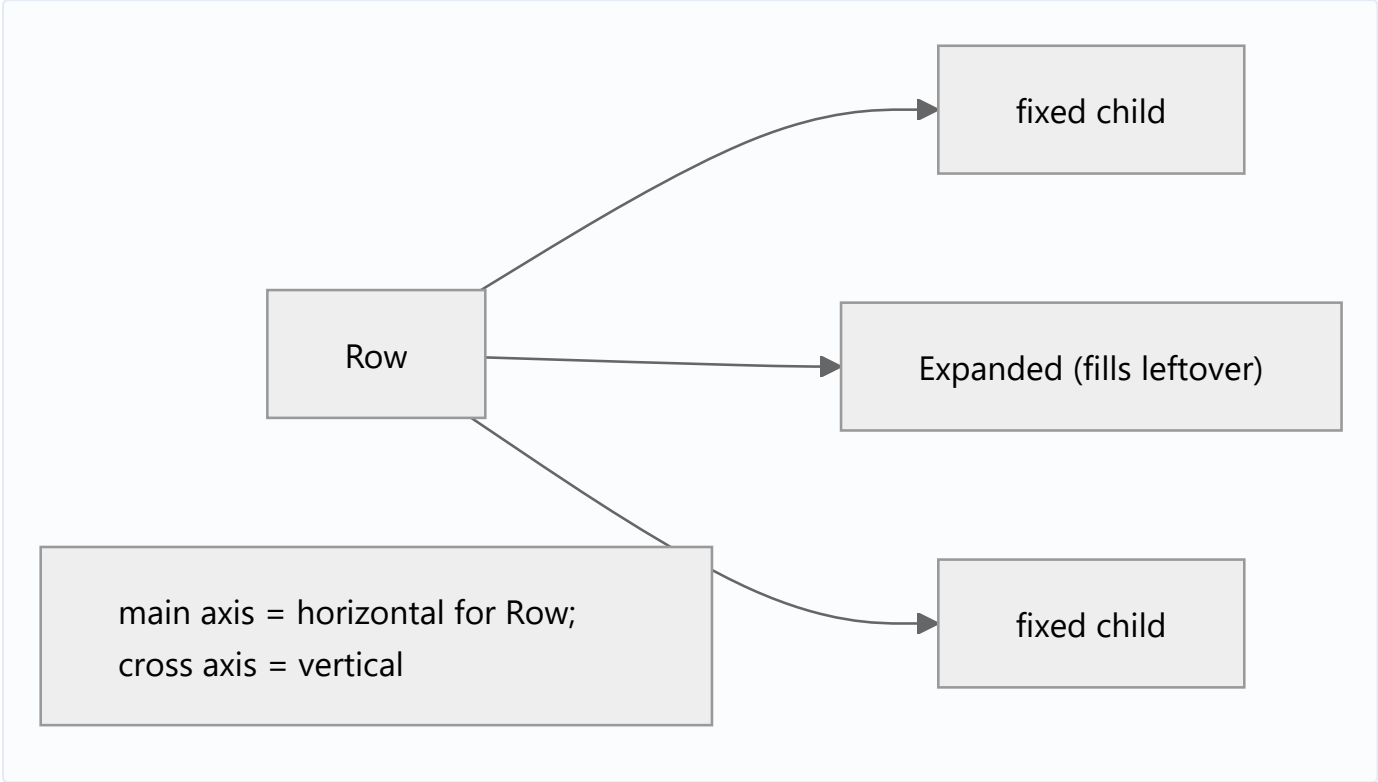
## Real-world analogy

A **row of books on an adjustable shelf**: some books have fixed width (unflexed), some stretch to fill gaps (`Expanded`), and you can push them left, center, right, or space them evenly (`mainAxisAlignment`).

## Problem Statement

Build a toolbar: an icon on the left, a title that takes remaining space, and an action on the right; plus a form column of full-width fields. You'll use `Row` / `Column`, `Expanded`, `Spacer`, and alignment.

## Internal Working




- **Main axis:** horizontal for `Row`, vertical for `Column`. **Cross axis** is perpendicular.
- `mainAxisAlignment`: `start` / `end` / `center` / `spaceBetween` / `spaceAround` / `spaceEvenly` — distributes free space along main axis.
- `crossAxisAlignment`: `start` / `end` / `center` / `stretch` / `baseline`.
- `mainAxisSize`: `max` (fill available) vs `min` (shrink to children).
- `Expanded` (flex, fills remaining, forces fill) vs `Flexible` (may be smaller than its share) vs `Spacer` (flexible empty gap).
- Flex first lays out **non-flex** children, then divides remaining space among flex children by their `flex` factor.

## Memory Representation

Backed by `RenderFlex` render objects (06); layout computed via constraints (03_constraints_and_sizing.md).

## Compiler Behavior

Not applicable. Use `const` for static children.

## Runtime Behavior

`RenderFlex` sizes inflexible children first, then allocates leftover main-axis space to flex children proportionally; overflow (children exceed space, no flex) produces the yellow-black overflow stripes.

## Flutter Engine Behavior

Layout (constraints down, sizes up) happens in the render layer; details in Module 09.

## Dart VM Behavior

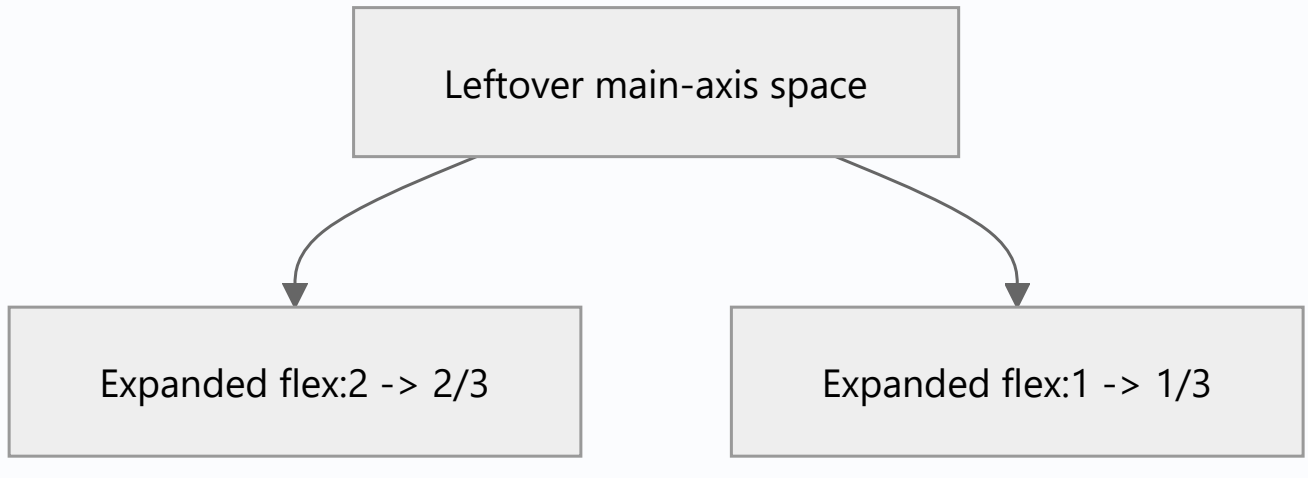
Not applicable.

## Examples

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


class FlexDemo extends StatelessWidget {
  const FlexDemo({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Column(
        crossAxisAlignment: CrossAxisAlignment.stretch, // fill width
        children: [
          // Toolbar: icon | title (fills) | action
          Row(
            children: [
              const Icon(Icons.menu),
              const SizedBox(width: 8),
              const Expanded(                         // takes ALL leftover space
                child: Text('Title', overflow: TextOverflow.ellipsis),
              ),
              IconButton(icon: const Icon(Icons.search), onPressed: () {}),
            ],
          ),
          const SizedBox(height: 16),
          // Proportional split: 2:1
          Row(
            children: [
```

```
              Expanded(flex: 2, child: Container(height: 40, color: Colors.teal)),
              const SizedBox(width: 8),
              Expanded(flex: 1, child: Container(height: 40, color: Colors.orange)),
            ],
          ),
          const SizedBox(height: 16),
          // Push apart with Spacer
          Row(
            children: const [Text('Left'), Spacer(), Text('Right')],
          ),
          const SizedBox(height: 16),
          // Even distribution
          const Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [Text('A'), Text('B'), Text('C')],
          ),
        ],
      ),
    );
  }
}
```

## Diagrams



## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Overflow (yellow/black stripes) | Children exceed available space with no flex | Wrap flexible child in `Expanded` / `Flexible`, or scroll |
| `Row` with an unbounded-width child (e.g., another `Row` / `Text` too long) | Infinite width constraint | Constrain or `Expanded` / `Flexible` |
| Using `Expanded` outside a Flex | Only valid in `Row` / `Column` / `Flex` | Put it inside a flex parent |
| `mainAxisSize.max` inside a scroll view | Fights unbounded axis | Use `min` or intrinsic sizing |
| Confusing main vs cross alignment | Wrong axis controlled | Row: main=horizontal; Column: main=vertical |

## Best Practices

- Use `Expanded` to fill remaining space; `Flexible` when the child may be smaller; `Spacer` for gaps.
- Set `mainAxisSize: min` when a `Row` / `Column` should hug its children (e.g., inside a `Center`).
- Guard long text with `Expanded` + `overflow: TextOverflow.ellipsis`.
- Prefer `SizedBox` for fixed spacing over padding hacks.

## Performance

Flex layout is cheap; deeply nested flex + intrinsic sizing can cost more — keep trees reasonable (Module 21).

## Advantages / Disadvantages

- **+** Declarative linear layout, flexible space distribution, alignment control.
- **−** Overflow/unbounded-constraint errors confuse beginners; requires the constraints mental model.

## Interview Questions

1. 🟢 **`Row` vs `Column`?** — `Row` arranges children horizontally (main axis = horizontal); `Column` vertically. Both are `Flex`.
2. 🟢 **`Expanded` vs `Flexible`?** — `Expanded` forces the child to fill its share of remaining space; `Flexible` lets it be smaller (up to its share). `Expanded` = `Flexible(fit: FlightFit.tight)`.
3. 🟡 **What causes the overflow stripes?** — Children's total main-axis size exceeds available space with no flex to absorb it; fix with `Expanded` / `Flexible` or a scroll view.

4. 🟡 `mainAxisAlignment` **vs** `crossAxisAlignment`**?** — Main aligns along the layout direction (distributing free space); cross aligns perpendicular (including `stretch`).
5. 🟡 **What does** `mainAxisSize` **control?** — Whether the flex fills the available main-axis extent (`max`) or shrinks to its children (`min`).
6. 🔴 **How does** `RenderFlex` **allocate space?** — Lays out inflexible children first, then divides leftover main-axis space among flex children by their `flex` factors.
7. 🔴 **Why does a** `Row` **inside a** `Row` **sometimes throw an unbounded-width error?** — The inner needs a bounded width; wrap it in `Expanded` / `Flexible` or constrain it (03_constraints_and_sizing.md).

## Senior Engineer Tips

- "Unbounded" errors mean a child wants infinite space on an axis; the fix is almost always `Expanded` / `Flexible` or an explicit size.
- Use `Spacer` and `spaceBetween` instead of manual padding math for distribution.
- Extract repeated `Row` / `Column` structures into small widgets (09_custom_composite_widgets.md).

## Architect Perspective

Flex is the workhorse of responsive layouts; combined with `LayoutBuilder` / `MediaQuery` it adapts to size (Module 24). A solid grasp of flex + constraints prevents the majority of layout bugs and is prerequisite to building a consistent design system.

## Summary

- `Row` / `Column` / `Flex` arrange along a main axis; `Expanded` / `Flexible` / `Spacer` split leftover space.
- `mainAxisAlignment` / `crossAxisAlignment` / `mainAxisSize` control positioning.
- Overflow/unbounded errors come from the constraints model — wrap flexible children.

## Revision Notes

- Row=horizontal main axis; Column=vertical. Cross axis = perpendicular.
- `Expanded` fills share (tight); `Flexible` may be smaller; `Spacer` = flexible gap.
- `mainAxisAlignment` distributes free space; `crossAxisAlignment` incl. `stretch`.
- Overflow/unbounded → wrap in `Expanded` / `Flexible` or constrain/scroll.

## Practice Questions

1. Why does long text in a `Row` overflow, and how do you fix it?

2. `Expanded` vs `Flexible` — give a case for each.
3. How does a 2:1 space split work with `flex`?

## Coding Questions

1. Build a toolbar: leading icon, expanding ellipsized title, trailing action.
2. Split a row into 3:2 colored panels with a gap.
3. Use `Spacer` / `spaceBetween` to place items at the two ends.

## Mini Project

**Responsive toolbar + form (Flutter):** Build a `Column` form of full-width fields under a `Row` toolbar (icon | expanding title | action), plus a proportional colored panel row. Handle long titles with ellipsis. Acceptance: no overflow errors; correct flex distribution; alignment intentional; app runs.

# Constraints & Sizing (`Container`, `SizedBox`, `ConstrainedBox`)

> Flutter layout has one rule: **constraints go down, sizes go up, parent sets position.** Understand it and every "unbounded height", "infinite width", and overflow error becomes obvious.

## Introduction

A parent passes **constraints** (min/max width & height) to each child; the child chooses its **size** within them and reports back; the parent then **positions** it. This single-pass model governs all layout. This file makes it concrete and covers the sizing widgets (`SizedBox`, `Container`, `ConstrainedBox`, `FractionallySizedBox`, `AspectRatio`).

## Why this concept exists

Flutter uses a fast, single-pass layout (no multi-pass constraint solving). To keep it O(n), children must size themselves *given* a parent's constraints — they can't "measure everything." Knowing this explains why widgets behave as they do and why certain combinations error.

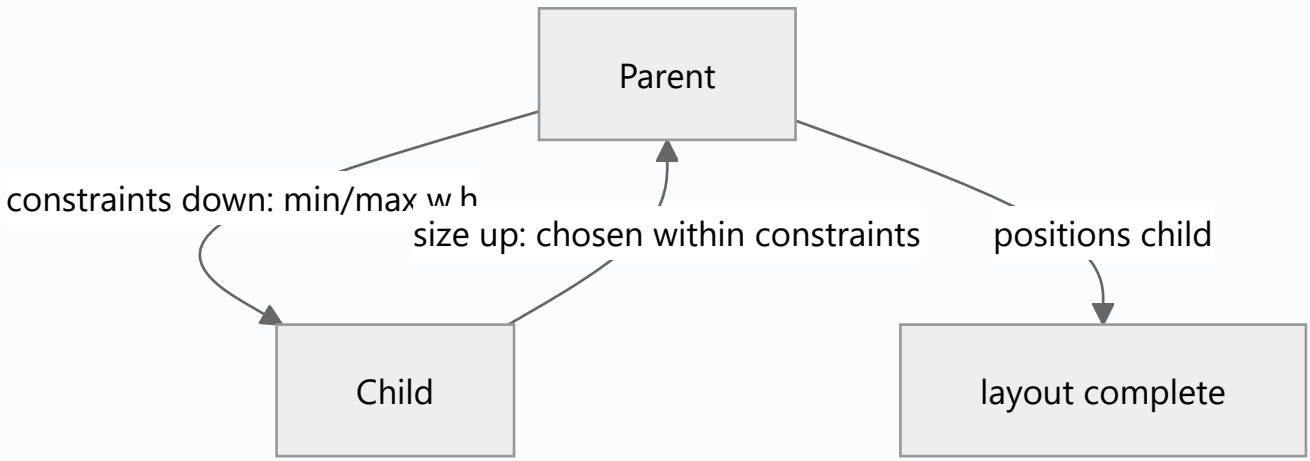
## Real-world analogy

A landlord (parent) tells a tenant (child): "your room can be **between 10 and 20 $m^2$**" (constraints). The tenant picks a size in that range (say 15 $m^2$) and reports it; the landlord then decides **where** to place the room. The tenant can't demand a size outside the allowed range.

## Problem Statement

You get "BoxConstraints forces an infinite height", or a `Container` doesn't size as expected, or a child won't fill its parent. You'll diagnose each via the constraints rule and pick the right sizing widget.

## Internal Working




**The rule (memorize):**

1. Parent gives child **constraints** (a `BoxConstraints`: minW/maxW/minH/maxH).
2. Child picks its **size** within those constraints.
3. Parent **positions** the child (child can't know/choose its own position).

Sizing widgets:

| Widget | Effect |
|---|---|
| `SizedBox(width, height)` | Fixed size (or spacing) |
| `ConstrainedBox(constraints)` | Impose min/max constraints |
| `UnconstrainedBox` | Let child pick its natural size (removes constraints) |
| `Container` | Convenience combo (padding/margin/color/constraints/child) |
| `FractionallySizedBox(widthFactor)` | Size relative to parent |
| `AspectRatio(aspectRatio)` | Size to a ratio |
| `Expanded` / `Flexible` | Fill within a Flex (02_layout_flex.md) |
| `LayoutBuilder` | Read incoming constraints to build adaptively |

**Tight vs loose constraints:** *tight* fixes an exact size (min==max); *loose* allows anything up to max (min 0). E.g., a `Center` gives loose constraints; `Expanded` gives tight.

## Memory Representation

Constraints/sizes are computed per frame in the render tree; no persistent extra allocation beyond render objects (06).

## Compiler Behavior

Not applicable.

## Runtime Behavior

Layout runs once per frame top-down (constraints) then bottom-up (sizes). Violations (e.g., an unbounded constraint given to a widget that needs a bound) throw descriptive layout errors.

## Flutter Engine Behavior

This is the layout phase of the pipeline; deep detail in Module 09.

## Dart VM Behavior

Not applicable.

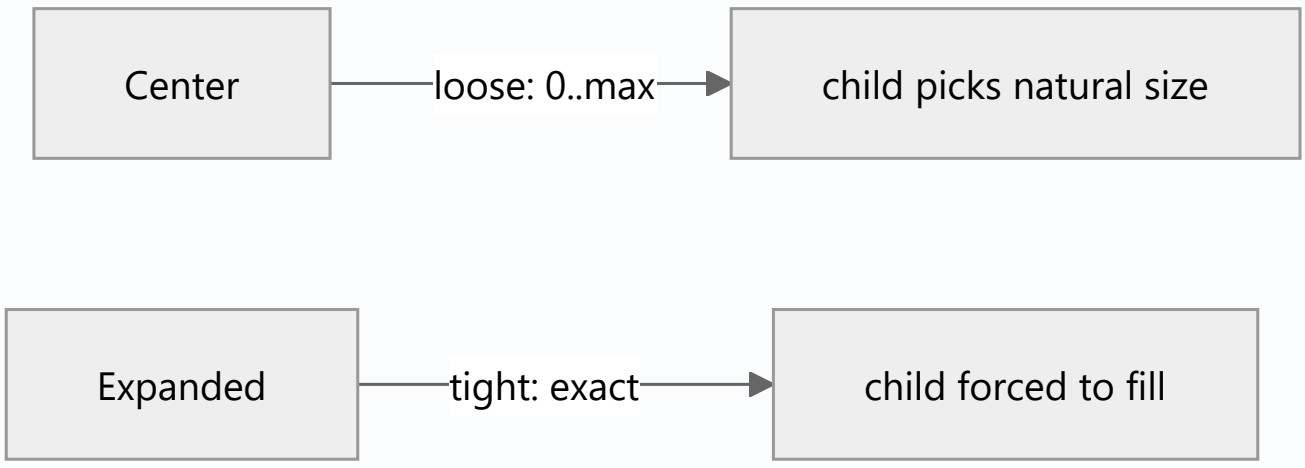
## Examples

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


class SizingDemo extends StatelessWidget {
  const SizingDemo({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Column(
        children: [
          // Fixed size box
          const SizedBox(width: 100, height: 40, child: ColoredBox(color: Colors.teal)),


          // Container tries to be as big as parent allows unless constrained:
          Container(height: 50, color: Colors.orange), // full width, 50 tall


          // Impose max width regardless of parent:
          ConstrainedBox(
            constraints: const BoxConstraints(maxWidth: 150),
            child: Container(height: 30, color: Colors.purple),
          ),


          // Read incoming constraints to adapt:
```

```
          LayoutBuilder(
            builder: (context, constraints) => Text(
              'maxW=${constraints.maxWidth.toStringAsFixed(0)}',
            ),
          ),

          // Fill remaining vertical space (tight constraint from Expanded):
          Expanded(child: Container(color: Colors.indigo.shade100)),
        ],
      ),
    );
  }
}
```

```
Classic error: putting a ListView (wants unbounded height) inside a Column
without bounding it -> "vertical viewport was given unbounded height".
Fix: wrap the ListView in Expanded (gives it a bounded height).
```

## Diagrams



## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| `ListView` / `Column` unbounded in a `Column` | Scrollables want unbounded main-axis; parent can't provide | Wrap in `Expanded` / `SizedBox(height:)` / `shrinkWrap` |
| Expecting `Container` to shrink-wrap when parent is tight | Tight constraints force size | Use `Align` / `UnconstrainedBox` or change parent |
| `width` / `height` ignored | Parent constraints override child's wishes | Understand constraints beat requested size |
| Infinite width in `Row` child | Unbounded main axis | `Expanded` / `Flexible` or fixed width |
| Overusing `Container` for one property | Bloat | Use `SizedBox` / `Padding` / `ColoredBox` / `Align` |

## Best Practices

- Recite the rule when debugging: **constraints down, sizes up, parent positions.**
- Use `LayoutBuilder` to build based on **incoming constraints**; `MediaQuery` for screen size (Module 24).
- Bound scrollables ( `Expanded` , fixed height, or `shrinkWrap` ) inside flexes.
- Prefer specific sizing widgets ( `SizedBox` , `ConstrainedBox` , `AspectRatio` ) over `Container` when you need one thing.

## Performance

Single-pass layout is O(n); avoid excessive `IntrinsicWidth` / `IntrinsicHeight` (they force extra passes) — expensive in big trees (Module 21).

## Advantages / Disadvantages

- **+** Fast single-pass layout, predictable once understood, powerful adaptive tools.
- **−** Steep beginner learning curve; unbounded/tight errors are cryptic until the rule clicks.

## Interview Questions

1. 🟢 **State Flutter's layout rule.** — Constraints go down (parent→child), sizes go up (child→parent), and the parent sets the child's position.
2. 🟢 **Tight vs loose constraints?** — Tight: min==max (exact size forced, e.g., `Expanded` ). Loose: min 0..max (child picks, e.g., `Center` ).
3. 🟡 **Why does a `ListView` in a `Column` throw "unbounded height"?** — `Column` gives its children unbounded vertical space, but a scrollable needs a bounded viewport. Wrap it in

`Expanded` (or set a height / `shrinkWrap`).

4. 🟡 **Why might a `Container`'s `width` be ignored?** — If the parent passes tight constraints, they override the child's requested size.
5. 🟡 **`LayoutBuilder` vs `MediaQuery`?** — `LayoutBuilder` gives the *parent's* constraints (local); `MediaQuery` gives *screen/window* metrics (global).
6. 🔴 **Why is Flutter layout single-pass and why does that matter?** — For O(n) performance; it means children size themselves given constraints rather than being measured multiple times — hence the rule and its errors.
7. 🔴 **What do `IntrinsicHeight` / `IntrinsicWidth` cost?** — Extra layout passes to compute intrinsic sizes; avoid in large/scrolling trees.

## Senior Engineer Tips

- When layout misbehaves, add a `LayoutBuilder` to print the incoming constraints — it reveals what the parent is actually offering.
- "It won't fill" → the parent isn't giving tight/bounded constraints; "unbounded" → wrap in something that bounds it.
- Reserve intrinsic sizing for small, non-scrolling cases.

## Architect Perspective

The constraints model is the foundation of responsive, adaptive, and performant layouts across form factors (Modules 24, 25). Teaching it early prevents the most common UI bugs and is prerequisite to custom layout/render work (Module 09, Module 23).

## Summary

- Layout = constraints down, sizes up, parent positions (single pass, O(n)).
- Tight forces size; loose lets the child choose. Use `SizedBox` / `ConstrainedBox` / `LayoutBuilder` deliberately.
- Bound scrollables in flexes; avoid excessive intrinsic sizing.

## Revision Notes

- Rule: constraints ↓, sizes ↑, parent positions.
- Tight (min==max) vs loose (0..max); `Expanded` =tight, `Center` =loose.
- `ListView` in `Column` → unbounded → `Expanded` /height/ `shrinkWrap`.
- `LayoutBuilder` = parent constraints; `MediaQuery` = screen; avoid heavy intrinsics.

## Practice Questions

1. Why is a child's requested width sometimes ignored?
2. How do you fix "vertical viewport was given unbounded height"?
3. `LayoutBuilder` vs `MediaQuery` — when each?

## Coding Questions

1. Reproduce and fix the `ListView`-in-`Column` unbounded error three ways.
2. Use `LayoutBuilder` to switch layout above/below 600px width.
3. Constrain a child to `maxWidth: 400` centered on wide screens.

## Mini Project

**Constraints playground (Flutter):** Build a screen demonstrating tight vs loose (Expanded vs Center), a bounded ListView-in-Column, a `ConstrainedBox` max-width card, and a `LayoutBuilder` that adapts at a breakpoint. Comment each with the constraints rule. Acceptance: no layout errors; each case demonstrates the rule; app runs.

# Stack & Positioning (`Stack`, `Positioned`, `Align`, `Alignment`)

> `Stack` overlaps children on top of each other; `Positioned` / `Align` place them within it — the tool for badges, overlays, layered backgrounds, and free-form placement.

## Introduction

Where `Row` / `Column` arrange children *beside* each other, `Stack` layers them *on top* of each other. Children are positioned with `Positioned` (edges/size) or `Align` / `Alignment` (relative anchor). This file covers overlap layout and its sizing quirks.

## Why this concept exists

Some UIs need layering: a badge on an avatar, text over an image, a floating button over content, gradient overlays. Flex can't overlap; `Stack` provides z-ordering and precise/relative placement in a shared coordinate space.

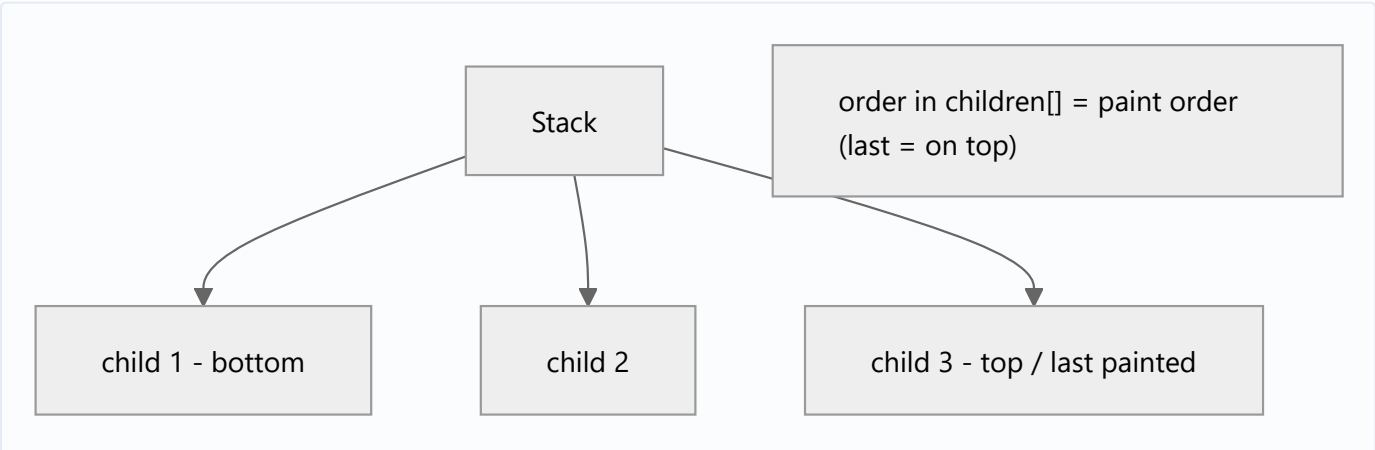
## Real-world analogy

A **stack of transparent sheets on an overhead projector**: each sheet (child) sits above the previous; you slide each to a spot (position). The last sheet added is on top (paint order).

## Problem Statement

Put a notification badge on the top-right of an avatar, and caption text at the bottom-left of an image. You'll use `Stack` + `Positioned` / `Align`.

## Internal Working



- `Stack` sizes itself to its **non-positioned** children (largest), constrained by parent; `Positioned` children don't affect stack size.
- `Positioned` places a child by `top` / `right` / `bottom` / `left` / `width` / `height` (only valid inside a `Stack`). `Positioned.fill` stretches to all edges.
- `Align` places a non-positioned child by an `Alignment` (e.g., `Alignment.bottomLeft`); `Alignment(x, y)` uses -1..1 coordinates.
- `fit`: `StackFit.loose` (children size themselves) vs `StackFit.expand` (fill the stack).
- **Paint/z-order:** later children in the list paint on top.

## Memory Representation

Backed by `RenderStack`; positioned children are laid out relative to the stack (06).

## Compiler Behavior / Runtime Behavior

`Positioned` outside a `Stack` is a layout error. Non-positioned children are aligned via the stack's `alignment`.

## Flutter Engine Behavior

Layered painting in paint order; overlays composite in the pipeline (Module 09).

## Dart VM Behavior

Not applicable.

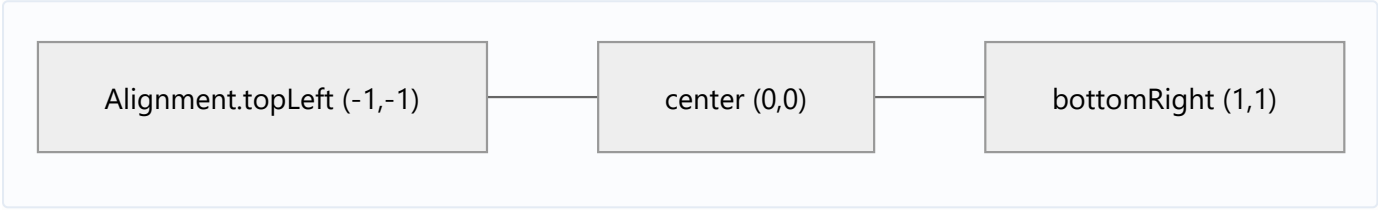
## Examples

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


class StackDemo extends StatelessWidget {
  const StackDemo({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            // Avatar with a badge (top-right)
```

```
SizedBox(
  width: 64,
  height: 64,
  child: Stack(
    clipBehavior: Clip.none,
    children: [
      const CircleAvatar(radius: 32, child: Icon(Icons.person)),
      Positioned(
        top: -2,
        right: -2,
        child: Container(
          padding: const EdgeInsets.all(4),
          decoration: const BoxDecoration(
              color: Colors.red, shape: BoxShape.circle),
          child: const Text('3',
              style: TextStyle(color: Colors.white, fontSize: 10)),
        ),
      ),
    ],
  ),
),
const SizedBox(height: 24),
// Image with caption overlay (bottom-left)
SizedBox(
  width: 200,
  height: 120,
  child: Stack(
    fit: StackFit.expand,
    children: [
      Container(color: Colors.teal), // stand-in for an image
      const Align(
        alignment: Alignment.bottomLeft,
        child: Padding(
          padding: EdgeInsets.all(8),
          child: Text('Caption',
              style: TextStyle(color: Colors.white)),
        ),
      ),
    ],
  ),
```

```
            ),
          ],
        ),
      ),
    );
  }
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| `Positioned` outside a `Stack` | Only valid in `Stack` | Put it inside a `Stack` |
| Stack collapses to zero size | Only positioned children → no size driver | Add a non-positioned child or wrap in `SizedBox` |
| Content clipped at edges | Default clips overflow | `clipBehavior: Clip.none` (for badges) |
| Expecting flex behavior in Stack | Stack overlaps, not distributes | Use `Row` / `Column` for beside-each-other |
| Wrong z-order | Paint order = list order | Reorder `children` |

## Best Practices

- Give the `Stack` a size (via a sized parent or a non-positioned child) so positioned children have a frame.
- Use `Positioned.fill` for backgrounds/overlays; `Align` for simple anchored placement.
- `clipBehavior: Clip.none` when badges must exceed bounds.
- Keep stacks shallow; deep overlays hurt readability and can complicate hit-testing.

## Performance

Cheap for small stacks; many overlapping semi-transparent layers add compositing cost (Module 21).

## Advantages / Disadvantages

- **+** Layering/overlays, precise + relative placement, z-order control.
- **−** Sizing quirks (needs a size driver), easy to misuse for layouts flex should do, hit-testing subtleties.

## Interview Questions

1. 🟢 **What does `Stack` do?** — Overlaps children on top of each other in a shared coordinate space (z-ordered by list order).
2. 🟢 **How do you position a child in a `Stack`?** — `Positioned` (edges/size) for absolute-ish placement, or `Align` / `Alignment` for relative anchoring.
3. 🟡 **How does a `Stack` decide its size?** — It sizes to its largest **non-positioned** child (within parent constraints); positioned children don't drive size.
4. 🟡 **What determines z-order?** — Paint order = order in `children` (last is on top).
5. 🟡 **`Positioned.fill` vs `StackFit.expand`?** — `Positioned.fill` stretches one child to all edges; `StackFit.expand` makes the stack expand and forces non-positioned children to fill.
6. 🔴 **Why does my `Stack` have zero size?** — All children are `Positioned`, so there's no size driver; add a non-positioned child or constrain the stack.
7. 🔴 **How do you let a badge overflow the stack bounds?** — Set `clipBehavior: Clip.none` (default clips).

## Senior Engineer Tips

- Always give a Stack a definite size via a sized ancestor/child; "zero-size Stack" is a frequent gotcha.
- Prefer `Align` for simple anchored overlays; reserve `Positioned` for precise offsets.
- For tappable overlapping areas, mind hit-test order (top child wins) and use `IgnorePointer` / `AbsorbPointer` deliberately.

## Architect Perspective

Stack underpins overlay systems (badges, tooltips, custom dialogs, media controls). Combined with `Overlay` / `OverlayEntry` it powers app-wide floating UI. Used judiciously it's essential; overused it hides structure that flex would express more clearly.

## Summary

- `Stack` layers children (z-order = list order); `Positioned` / `Align` place them.
- Stack sizes to non-positioned children; give it a size driver.

- Use `Positioned.fill` for overlays, `Clip.none` for overflowing badges.

## Revision Notes

- `Stack` overlaps; last child on top; sizes to non-positioned children.
- `Positioned` (edges/size, Stack-only) vs `Align` / `Alignment` (relative -1..1).
- Zero-size stack → add non-positioned child / constrain.
- `Positioned.fill` overlays; `Clip.none` to overflow.

## Practice Questions

1. Why might a Stack render with zero size?
2. How is z-order determined?
3. When use `Align` vs `Positioned`?

## Coding Questions

1. Put a count badge on the top-right of an icon, overflowing bounds.
2. Overlay a gradient + caption on an image via `Positioned.fill` + `Align`.
3. Build a card with a "NEW" ribbon in a corner.

## Mini Project

**Media card with overlays (Flutter):** Build a fixed-size card with a background, a bottom gradient + title ( `Positioned.fill` / `Align` ), and a top-right badge that overflows ( `Clip.none` ). Acceptance: correct sizing/z-order; badge overflows; no `Positioned` -outside-Stack errors; app runs.

# Scrolling & Slivers (`ListView`, `GridView`, `CustomScrollView`)

> Use **builder** constructors (`ListView.builder`) for long/lazy lists, and **slivers** (`CustomScrollView` + `SliverAppBar` / `SliverList`) when you need custom scroll effects like collapsing headers — never build thousands of widgets eagerly.

## Introduction

Scrollables show more content than fits on screen. This file covers `ListView` / `GridView` (and their lazy `.builder` forms), `SingleChildScrollView`, and the **sliver** system (`CustomScrollView`) for advanced scroll behaviors. The key theme: **laziness** — build only what's visible.

## Why this concept exists

Rendering a 10,000-item list eagerly would allocate 10,000 widgets/elements and jank hard. Lazy builders construct items on demand as they scroll into view. Slivers generalize this: composable scrollable "slices" that can implement app bars, grids, lists, and effects in one scroll view.

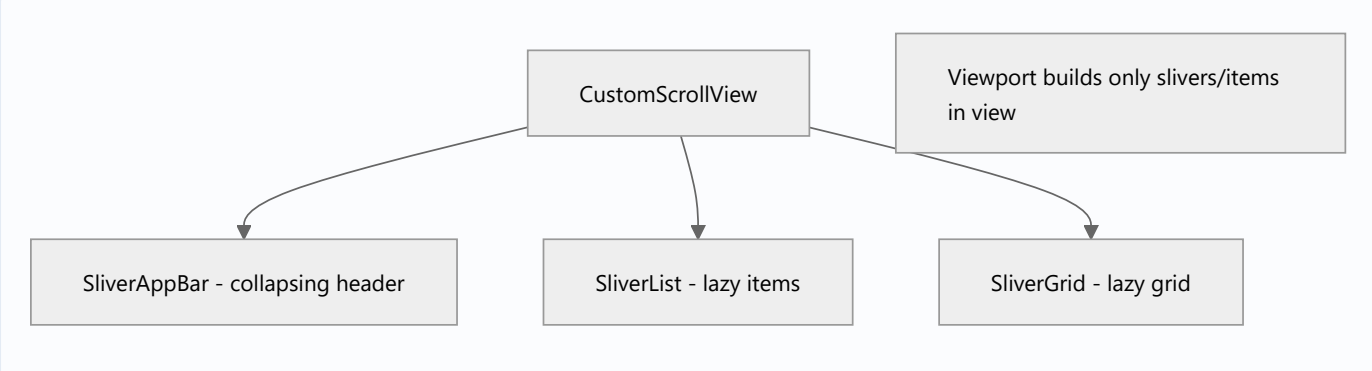
## Real-world analogy

A **conveyor belt sushi bar**: plates (items) are prepared as they approach you (lazy building), not all at once. Slivers are like **different stations on one belt** — an app-bar station, a list station, a grid station — all moving together.

## Problem Statement

Show a 5,000-item feed smoothly, then a screen with a large image header that collapses into an app bar as you scroll. You'll use `ListView.builder` and a `CustomScrollView` with slivers.

## Internal Working



- `ListView.builder(itemBuilder, itemCount)`: builds items **lazily** as they scroll into view (the default for real lists).
- `ListView(children: [...])`: builds **all** children eagerly — fine for a few, bad for many.
- `SingleChildScrollView`: scrolls a single (often `Column`) child — no laziness; only for small content.
- `GridView.builder`: lazy grid with a `SliverGridDelegate` (fixed count or max extent).
- **Slivers**: `CustomScrollView` hosts `SliverAppBar` (collapsing), `SliverList` / `SliverChildBuilderDelegate`, `SliverGrid`, `SliverToBoxAdapter` (wrap a normal widget), `SliverPadding`.
- **Physics**: `ScrollPhysics` (bouncing iOS, clamping Android) — a Strategy (05 · strategy).

## Memory Representation

Lazy scrollables keep only visible (+cache extent) items in the element/render trees; off-screen items are recycled → bounded memory (02 · memory_and_gc).

## Compiler Behavior

Not applicable. Use `const` item content where possible.

## Runtime Behavior

The viewport requests items for the visible range + a cache area; scrolling builds/disposes items at the edges. Eager `ListView(children:)` builds everything up front regardless of visibility.

## Flutter Engine Behavior

Scrolling drives repeated layout/paint of the viewport; smoothness depends on cheap item builds (Module 21).

## Dart VM Behavior

Not applicable.

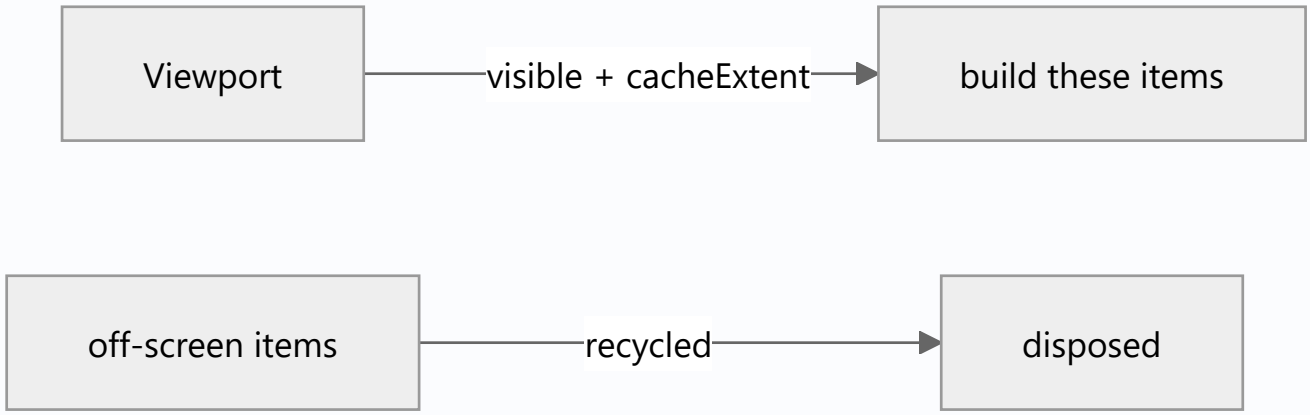
## Examples

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


class FeedList extends StatelessWidget {
  const FeedList({super.key});
  @override
  Widget build(BuildContext context) {
```

```
    // LAZY: builds only visible items among 5000
    return ListView.builder(
      itemCount: 5000,
      itemBuilder: (context, index) => ListTile(
        leading: CircleAvatar(child: Text('$index')),
        title: Text('Item $index'),
      ),
    );
  }
}

class CollapsingHeader extends StatelessWidget {
  const CollapsingHeader({super.key});
  @override
  Widget build(BuildContext context) {
    return CustomScrollView(
      slivers: [
        const SliverAppBar(
          expandedHeight: 200,
          pinned: true,
          flexibleSpace: FlexibleSpaceBar(title: Text('Gallery')),
        ),
        SliverList(
          delegate: SliverChildBuilderDelegate(
            (context, index) => ListTile(title: Text('Row $index')),
            childCount: 50,
          ),
        ),
        SliverGrid(
          gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
            crossAxisCount: 3,
          ),
          delegate: SliverChildBuilderDelegate(
            (context, index) => Card(child: Center(child: Text('$index'))),
            childCount: 30,
          ),
        ),
      ],
    );
```

```
  }
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| `ListView(children: [huge list])` | Eager build → jank/memory | Use `ListView.builder` |
| `Column` of many items in `SingleChildScrollView` | No laziness | Use a lazy list |
| `ListView` inside `Column` unbounded | Constraints error | `Expanded` /height/ `shrinkWrap` (03_constraints_and_sizing.md) |
| Heavy work in `itemBuilder` | Per-item jank | Keep builds cheap; precompute |
| Nested scrollables same axis | Conflicting scroll | Use slivers in one `CustomScrollView` |
| Missing keys on dynamic lists | State jumps on reorder | Add `ValueKey` (06) |

## Best Practices

- **Always use** `.builder` for lists that can grow; reserve `children:[]` for a few fixed items.
- Combine multiple scroll effects with **one** `CustomScrollView` + slivers, not nested scrollables.
- Keep `itemBuilder` cheap; use `const` item subtrees; add stable keys for dynamic data.
- For infinite scroll/pagination, load more near the end (Module 21).
- Wrap a plain widget in a scroll view via `SliverToBoxAdapter`.

## Performance

Laziness bounds memory/CPU to the visible window; jank comes from expensive item builds or eager lists. Tune `cacheExtent`; profile scrolling with DevTools (Module 21).

## Advantages / Disadvantages

- **+** Lazy = smooth + bounded memory for huge data; slivers compose rich scroll effects.
- **−** Sliver API is verbose/steeper; nested/misused scrollables cause bugs.

## Interview Questions

1. 🟢 **`ListView` vs `ListView.builder`?** — `ListView(children:)` builds all children eagerly; `.builder` builds items lazily on demand — required for long/large lists.
2. 🟢 **When use `SingleChildScrollView`?** — For a small amount of content that might overflow; it's not lazy, so not for long lists.
3. 🟡 **What are slivers?** — Composable scrollable slices (`SliverAppBar`, `SliverList`, `SliverGrid`) hosted by a `CustomScrollView` to build custom scroll effects in one viewport.
4. 🟡 **How do you build a collapsing header?** — `CustomScrollView` with a `SliverAppBar` (`expandedHeight`, `pinned`, `flexibleSpace`) followed by content slivers.
5. 🟡 **Why does a `ListView` in a `Column` error?** — Unbounded height; bound it via `Expanded`/height/`shrinkWrap`.
6. 🔴 **How does lazy building bound memory?** — The viewport builds only visible items + a cache extent and recycles off-screen ones, so element/render counts stay proportional to what's visible.
7. 🔴 **How do you combine a list and a grid that scroll together?** — Put a `SliverList` and a `SliverGrid` in the same `CustomScrollView` (don't nest separate scrollables).

## Senior Engineer Tips

- Default to `.builder`; treat eager `children:[]` lists as a smell for anything data-driven.
- Use one `CustomScrollView` for multi-section screens (header + list + grid) instead of nesting scroll views.
- Profile scroll jank: it's almost always expensive `itemBuilder` work — cache/const/simplify items.

## Architect Perspective

Lazy scrolling + slivers are essential for feed-heavy, data-rich apps (social, commerce, media). The pattern — build only what's visible, compose sections as slivers, paginate at the edge — is a core performance and UX decision that scales to large datasets (Modules 21, 16).

## Summary

- Use `.builder` for lazy long lists/grids; `SingleChildScrollView` only for small content.
- Slivers ( `CustomScrollView` ) compose custom scroll effects (collapsing headers, mixed sections).
- Bound scrollables inside flexes; keep item builds cheap; key dynamic lists.

## Revision Notes

- `ListView.builder` / `GridView.builder` = lazy; `children:[]` = eager (few items only).
- Slivers: `CustomScrollView` + `SliverAppBar` / `SliverList` / `SliverGrid` / `SliverToBoxAdapter` .
- `ListView` in `Column` → bound it ( `Expanded` /height/ `shrinkWrap` ).
- Cheap `itemBuilder` ; keys for dynamic lists; paginate near end.

## Practice Questions

1. Why is `ListView.builder` required for a 5,000-item list?
2. How do you build a collapsing app-bar header?
3. Why nest slivers instead of scroll views for multi-section screens?

## Coding Questions

1. Build a lazy 10k-item `ListView.builder` with cheap items.
2. Build a `CustomScrollView` with a pinned `SliverAppBar` + `SliverList` + `SliverGrid` .
3. Add infinite-scroll pagination (load more near the bottom).

## Mini Project

**Gallery feed (Flutter):** Build a `CustomScrollView` with a collapsing `SliverAppBar` , a lazy `SliverList` section, and a `SliverGrid` section, plus pagination on the list. Keep items `const` /cheap and keyed. Acceptance: smooth scroll; lazy building (bounded memory); no unbounded/nested-scroll errors; app runs.

# Text & Theming (`Text`, `TextStyle`, `Theme`, `TextTheme`)

> Style text via `TextStyle`, but centralize look-and-feel in a `ThemeData` so the whole app stays consistent — read styles from `Theme.of(context)` instead of hardcoding.

## Introduction

`Text` renders strings; `TextStyle` styles them; `RichText` / `Text.rich` mix styles. But hardcoding styles everywhere causes drift. **Theming** centralizes colors, typography (`TextTheme`), and component styles in `ThemeData`, which widgets read via `Theme.of(context)`. This file covers both.

## Why this concept exists

Consistent typography and color are the essence of a polished app. Hardcoding `TextStyle(fontSize: 16, color: ...)` in 200 places makes rebranding or dark mode a nightmare. A central theme is the single source of truth; widgets consume it contextually.

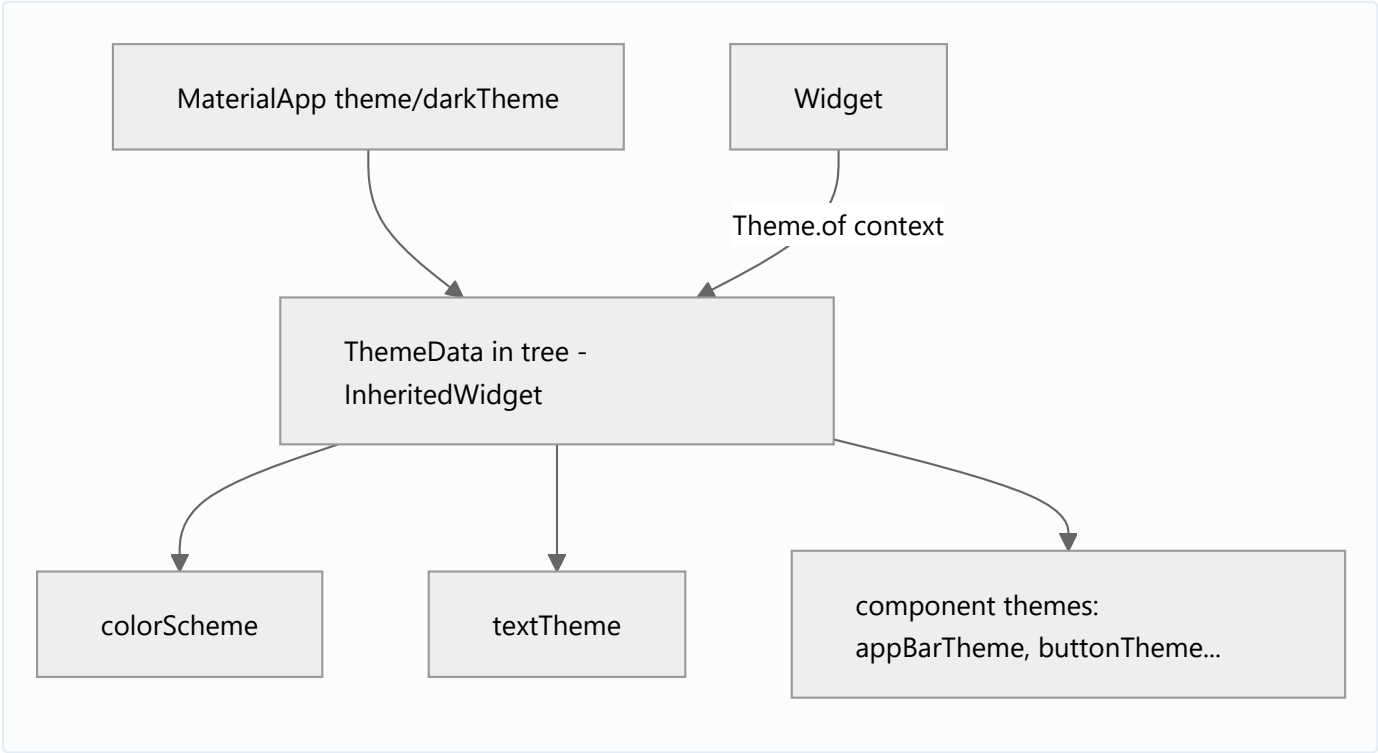
## Real-world analogy

A **company brand guide**: instead of each designer picking fonts/colors ad hoc, everyone follows one guide (theme). Change the guide once and all materials update. `Theme.of(context)` is "look it up in the brand guide."

## Problem Statement

Style headings/body consistently, support light/dark themes, and change the brand color in one place. You'll define a `ThemeData`, use `TextTheme`, and read styles from context.

## Internal Working




- **`TextStyle`**: `fontSize`, `fontWeight`, `color`, `letterSpacing`, `height` (line height), `fontFamily`, etc. Combine/override with `style.copyWith(...)`.
- **`ThemeData`**: app-wide `colorScheme` (Material 3: `colorSchemeSeed`), `textTheme`, and component sub-themes (`appBarTheme`, `elevatedButtonTheme`…).
- **`TextTheme`**: named roles (`displayLarge`, `headlineMedium`, `titleLarge`, `bodyMedium`, `labelSmall`…). Use `Theme.of(context).textTheme.titleLarge`.
- **Provided via `InheritedWidget`**: `Theme.of(context)` looks up the nearest `Theme` (06 · build_context).
- **Dark mode:** provide `theme` + `darkTheme` + `themeMode`; the framework picks based on system/user.

## Memory Representation

`ThemeData` is an immutable object in the tree; lookups are cheap (06).

## Compiler Behavior

Not applicable. Prefer `const TextStyle` where values are constant.

## Runtime Behavior

`Theme.of(context)` resolves the nearest theme; changing `themeMode` or theme triggers dependent rebuilds.

## Flutter Engine Behavior

Text layout/shaping happens in the engine (glyphs, line breaking); the framework provides the style.

## Dart VM Behavior

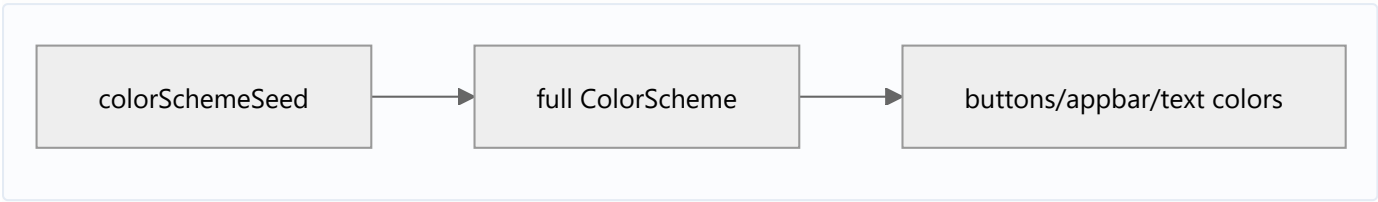
Not applicable.

## Examples

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


void main() => runApp(const MyApp());


class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      themeMode: ThemeMode.system, // follow device
      theme: ThemeData(
        useMaterial3: true,
        colorSchemeSeed: Colors.indigo, // one seed derives the palette
        textTheme: const TextTheme(
          titleLarge: TextStyle(fontWeight: FontWeight.bold),
        ),
      ),
      darkTheme: ThemeData(
        useMaterial3: true,
        brightness: Brightness.dark,
        colorSchemeSeed: Colors.indigo,
      ),
      home: const TextDemo(),
    );
  }
}


class TextDemo extends StatelessWidget {
  const TextDemo({super.key});
```

```
  @override
  Widget build(BuildContext context) {
    final theme = Theme.of(context);
    return Scaffold(
      appBar: AppBar(title: const Text('Text & Theme')),
      body: Padding(
        padding: const EdgeInsets.all(16),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            // Read styles from the theme, not hardcoded:
            Text('Heading', style: theme.textTheme.headlineMedium),
            const SizedBox(height: 8),
            Text('Body text goes here.', style: theme.textTheme.bodyMedium),
            const SizedBox(height: 8),
            // Override just one property via copyWith:
            Text('Accent',
                style: theme.textTheme.titleLarge
                    ?.copyWith(color: theme.colorScheme.primary)),
            const SizedBox(height: 8),
            // Mixed styles in one paragraph:
            Text.rich(TextSpan(children: [
              const TextSpan(text: 'Normal '),
              TextSpan(
                  text: 'bold',
                  style: theme.textTheme.bodyMedium
                      ?.copyWith(fontWeight: FontWeight.bold)),
            ])),
          ],
        ),
      ),
    );
  }
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Hardcoding `TextStyle`/colors everywhere | Drift, hard to rebrand/dark-mode | Read from `Theme.of(context)` |
| Duplicating a style with tweaks | Divergence | `theme.textTheme.x.copyWith(...)` |
| Ignoring dark mode | Poor UX | Provide `darkTheme` + `themeMode` |
| Fixed colors that break in dark | Contrast issues | Use `colorScheme` roles (`primary`, `onSurface`) |
| Custom fonts not declared | Font not applied | Declare in `pubspec.yaml` fonts section |

## Best Practices

- Centralize typography/colors in `ThemeData`; read via `Theme.of(context)`.
- Use `colorScheme` **roles** (`primary`, `surface`, `onSurface`) so dark mode & contrast work.
- Derive one-off styles with `copyWith` from theme text styles.
- Support light + dark from day one; test both.
- Declare custom fonts in `pubspec.yaml`; prefer Material 3 (`useMaterial3: true`).

## Performance

Theme lookups are cheap; `const TextStyle`/widgets avoid rebuild cost. Excessive theme-dependent rebuilds can be scoped (Module 21).

## Advantages / Disadvantages

- **+** Consistency, easy rebrand/dark mode, single source of truth, accessible contrast via roles.
- **−** Learning the `TextTheme`/`colorScheme` roles; over-theming can obscure local intent.

## Interview Questions

1. 🟢 **How do you style text?** — With `TextStyle` on a `Text` widget; combine styles with `copyWith`; mix styles via `Text.rich`/`RichText`.

2. 🟢 **How do you keep styling consistent app-wide?** — Define a `ThemeData` (colors + `textTheme` + component themes) and read via `Theme.of(context)` instead of hardcoding.
3. 🟡 **What is `TextTheme`?** — Named typography roles (`headlineMedium`, `bodyMedium`, etc.) provided by the theme.
4. 🟡 **How do you support dark mode?** — Provide `theme`, `darkTheme`, and `themeMode`; use `colorScheme` roles so colors adapt.
5. 🟡 **How is the theme delivered to widgets?** — Via an `InheritedWidget`; `Theme.of(context)` looks up the nearest one.
6. 🔴 **Why use `colorScheme` roles over raw colors?** — Roles (`primary` / `onSurface`) adapt across light/dark and maintain contrast/accessibility; raw colors break in one mode.
7. 🔴 **How do you add a custom font?** — Add the font files and declare them under `flutter: fonts:` in `pubspec.yaml`, then set `fontFamily`.

## Senior Engineer Tips

- Build a small **design-system layer** (extension getters like `context.textTheme`, semantic styles) so screens read intent, not raw values.
- Prefer seed-based `ColorScheme` (M3) for a coherent palette and easy theming.
- Always verify contrast in dark mode; use `on*` roles for foreground colors.

## Architect Perspective

Theming is your design-system foundation: centralizing tokens (color/typography/spacing) enables rebranding, white-labeling, dark mode, and accessibility at scale. Wrapping the theme in semantic app-level extensions decouples screens from raw values — the base of a maintainable UI layer (Module 25).

## Summary

- Style with `TextStyle` / `copyWith` / `Text.rich`; centralize in `ThemeData` and read via `Theme.of(context)`.
- Use `TextTheme` roles and `colorScheme` roles for consistency + dark mode + contrast.
- Support light/dark from the start; declare custom fonts in `pubspec`.

## Revision Notes

- `Text` + `TextStyle` (+ `copyWith`, `Text.rich`); prefer `const`.
- Centralize in `ThemeData` (colorScheme + textTheme + component themes); read via `Theme.of(context)`.
- Dark mode: `theme` / `darkTheme` / `themeMode`; use `colorScheme` roles.

- Custom fonts declared in `pubspec.yaml`; M3 `colorSchemeSeed`.

## Practice Questions

1. Why prefer `colorScheme.primary` over a hardcoded color?
2. How do you override one property of a theme text style?
3. How is `ThemeData` delivered to a deep widget?

## Coding Questions

1. Define a themed app (seed color + custom `titleLarge`) and render headings/body from the theme.
2. Add light/dark themes with `themeMode: system` and verify both.
3. Create semantic extension getters (`context.textTheme`, `context.colors`).

## Mini Project

**Themed typography kit (Flutter):** Build a screen showing all major `TextTheme` roles, an accent style via `copyWith`, and light/dark support driven by `colorSchemeSeed`. Add `context.textTheme` / `context.colors` extensions. Acceptance: no hardcoded colors/sizes; dark mode works with good contrast; app runs.

# Images & Assets (`Image`, assets, network, caching)

> Load bundled images with `Image.asset` (declared in `pubspec.yaml`), remote ones with `Image.network` (cached, with loading/error builders), and always control sizing with `fit` to avoid layout surprises.

## Introduction

Images come from assets (bundled), the network, files, or memory. This file covers declaring assets, the `Image` constructors, `BoxFit`, caching, placeholders/error handling, and resolution-aware assets.

## Why this concept exists

Images are heavy (bytes + decode + GPU memory) and come from varied sources with different failure modes (missing asset, network error, slow load). Flutter provides source-specific constructors, an image cache, and hooks for placeholders/errors so you handle these correctly and performantly.

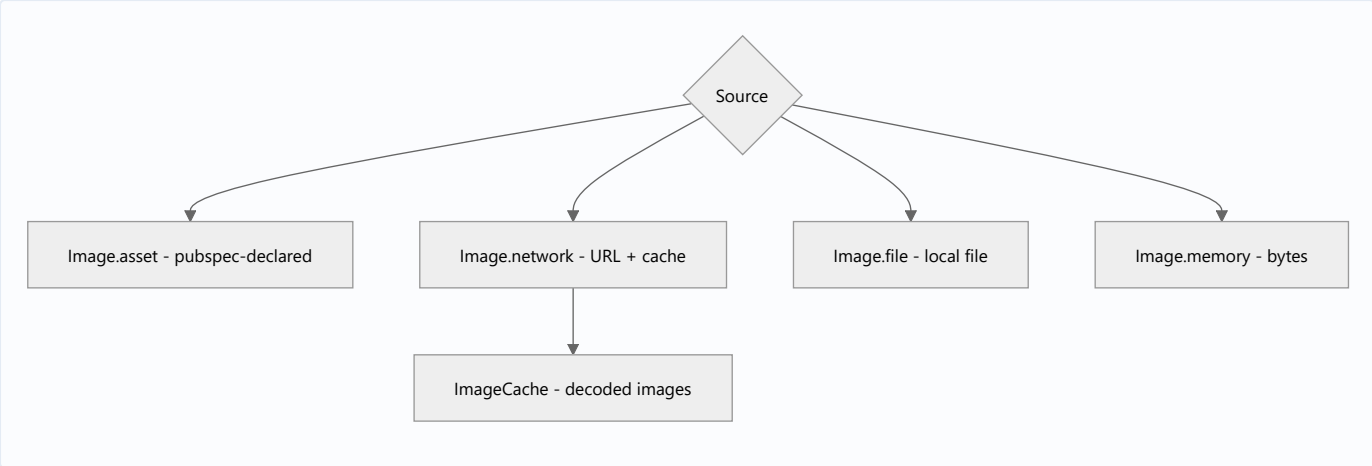
## Real-world analogy

Images are like **framed pictures**: `fit` decides how a picture fills its frame (stretch, crop, letterbox); the frame size is the layout box. Network images are pictures **ordered by mail** — you show a placeholder while waiting and a fallback if delivery fails.

## Problem Statement

Show a bundled logo, a remote avatar with a spinner while loading and a fallback on error, and control cropping. You'll declare assets, use `Image.network` builders, and set `BoxFit`.

## Internal Working




- `Image.asset('assets/logo.png')` — must be declared under `flutter: assets:` in `pubspec.yaml`.
- `Image.network(url, loadingBuilder:, errorBuilder:)` — remote; provide placeholder + fallback; uses the image cache.
- `Image.file` / `Image.memory` — local file / raw bytes.
- `BoxFit`: `cover` (fill, crop), `contain` (letterbox), `fill` (stretch), `fitWidth` / `fitHeight`, `none`, `scaleDown`.
- **Resolution-aware assets**: `1.5x` / `2.0x` / `3.0x` variants folders; Flutter picks by device pixel ratio.
- **Caching**: decoded images cached in `ImageCache` (bounded); `cacheWidth` / `cacheHeight` decode at target size to save memory.
- For robust network caching/placeholders, `cached_network_image` is the common package.

## Memory Representation

Decoded images consume memory ∝ pixels (not file size). Large images decoded at full resolution blow memory — use `cacheWidth` / `cacheHeight` or `ResizeImage` (Module 21).

## Compiler Behavior

Asset paths are validated at build; missing declared assets fail the build/asset bundle.

## Runtime Behavior

Network images load asynchronously (show placeholder), may error (show fallback). The cache reuses decoded frames across widgets.

## Flutter Engine Behavior

The engine decodes images (often off the UI isolate) and uploads to GPU textures; oversized images cost decode time + GPU memory (Module 21).

## Dart VM Behavior

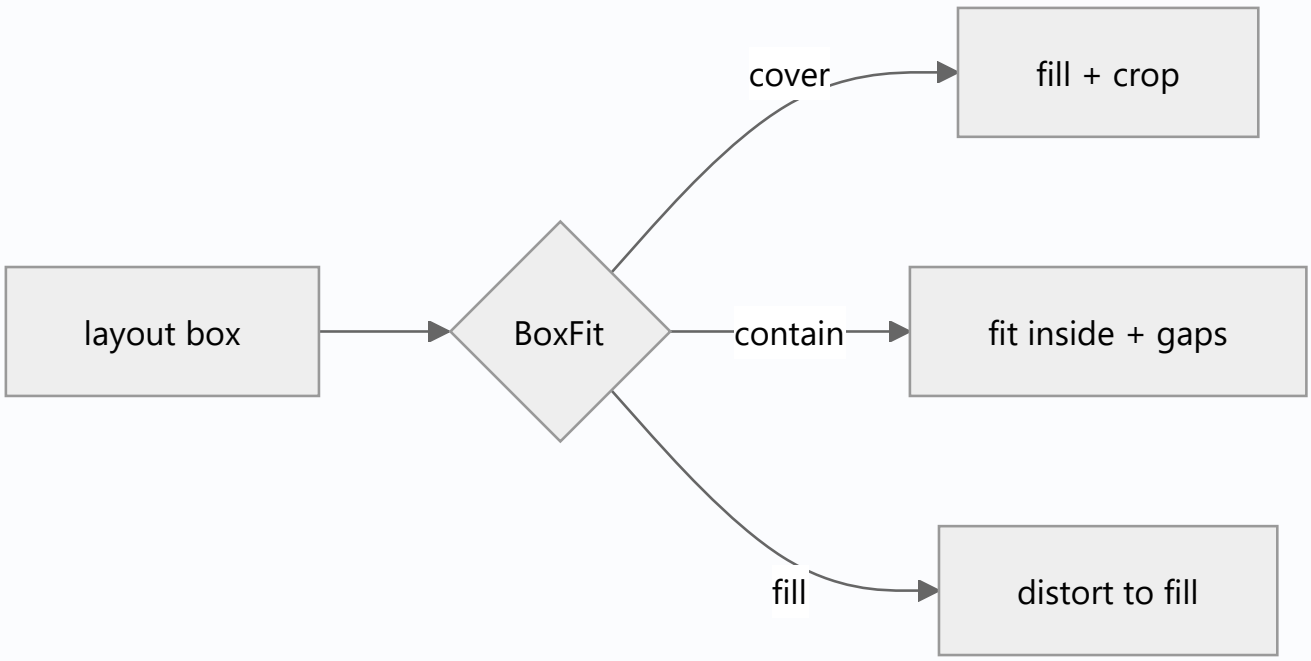
Not applicable directly; decode is engine-side.

## Examples

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


class ImageDemo extends StatelessWidget {
  const ImageDemo({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Column(
        children: [
          // Bundled asset (declared in pubspec.yaml)
          Image.asset('assets/logo.png', height: 80),


          // Network image with placeholder + error fallback, cropped to fill
          SizedBox(
            width: 120,
            height: 120,
            child: Image.network(
              'https://example.com/avatar.jpg',
              fit: BoxFit.cover, // fill the box, cropping as needed
              cacheWidth: 240,   // decode at ~2x display size, not full-res
              loadingBuilder: (context, child, progress) =>
                  progress == null ? child : const Center(child: CircularProgressIndicator()),
              errorBuilder: (context, error, stack) =>
                  const Icon(Icons.broken_image, size: 48),
            ),
          ),
        ],
      ),
    );
  }
}
```

```yaml
# pubspec.yaml — declare assets so Image.asset can find them
flutter:
  assets:
    - assets/logo.png
    - assets/images/   # a whole folder
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
| --- | --- | --- |
| Asset not declared in `pubspec.yaml` | Not bundled → error | Declare under `flutter: assets:` |
| No `errorBuilder` / `loadingBuilder` on network images | Broken/blank UI on failure/slow net | Provide placeholder + fallback |
| Decoding huge images at full res | Memory spikes/jank | `cacheWidth` / `cacheHeight` / `ResizeImage` |
| Wrong `BoxFit` | Distortion or unexpected crop | Choose `cover` / `contain` intentionally |
| Unbounded image in a `Row` / `Column` | Layout error | Constrain with `SizedBox` / `Expanded` |

## Best Practices

- Declare assets in `pubspec.yaml`; provide resolution variants for crispness.
- Always give network images **loading + error** builders (or use `cached_network_image`).
- Constrain image size and pick `BoxFit` deliberately (`cover` for avatars/thumbnails).
- Decode at display size (`cacheWidth` / `cacheHeight`) to bound memory.
- Prefer vector (`flutter_svg`) for icons/illustrations where appropriate.

## Performance

Memory ∝ decoded pixels; downscale on decode. The image cache reuses decodes; oversized images are a top memory/jank cause (Module 21).

## Advantages / Disadvantages

- **+** Source-specific constructors, caching, placeholders/errors, resolution awareness.
- **−** Easy to blow memory with full-res decodes; network needs explicit loading/error handling.

## Interview Questions

1. 🟢 **How do you show a bundled image?** — `Image.asset('path')` with the path declared under `flutter: assets:` in `pubspec.yaml`.
2. 🟢 **How do you handle network image loading/errors?** — `Image.network` with `loadingBuilder` (placeholder/progress) and `errorBuilder` (fallback), or use `cached_network_image`.
3. 🟡 **What does `BoxFit.cover` do vs `contain`?** — `cover` fills the box and crops overflow; `contain` fits entirely inside, possibly leaving gaps.
4. 🟡 **Why can images cause memory spikes?** — Decoded memory is proportional to pixel dimensions; full-res decodes of large images are huge. Use `cacheWidth` / `cacheHeight`.
5. 🟡 **How does Flutter pick resolution variants?** — By device pixel ratio, choosing `2.0x` / `3.0x` asset folders when declared.
6. 🔴 **Where does image decoding happen and why does it matter?** — In the engine (often off the UI isolate); oversized decodes cost time + GPU memory, causing jank if not downscaled.
7. 🔴 **When would you use `flutter_svg` instead of raster images?** — For scalable icons/illustrations that must stay crisp at any size and keep asset size small.

## Senior Engineer Tips

- Always downscale-on-decode for thumbnails/avatars; full-res decode of list images is a classic memory bug.
- Use `cached_network_image` in real apps for disk caching + placeholders/errors out of the box.
- Reserve space (fixed box) for images to avoid layout jumps while they load.

## Architect Perspective

Image handling is a performance-critical, failure-prone concern in media/commerce/social apps: caching strategy (memory + disk), decode sizing, and graceful placeholders/errors materially affect memory, smoothness, and perceived quality — decisions that scale with feed size (Modules

21, 16, 34).

## Summary

- Load images by source ( `asset` / `network` / `file` / `memory` ); declare assets in `pubspec` .
- Handle network loading/errors; control cropping with `BoxFit` ; decode at display size to bound memory.
- Use resolution variants and caching; consider `cached_network_image` / `flutter_svg` .

## Revision Notes

- `Image.asset` (declare in `pubspec` ), `Image.network` (loading/error builders), `.file` / `.memory` .
- `BoxFit` : cover(crop)/contain(letterbox)/fill(stretch).
- Memory ∝ decoded pixels → `cacheWidth` / `cacheHeight` .
- Resolution variants by DPR; `cached_network_image` / `flutter_svg` in real apps.

## Practice Questions

1. Why must assets be declared in `pubspec.yaml` ?
2. How do you prevent a large image from blowing memory?
3. `cover` vs `contain` — when each?

## Coding Questions

1. Show an asset logo + a network avatar with spinner + error fallback.
2. Build an image grid that decodes thumbnails at reduced size.
3. Compare `cover` / `contain` / `fill` visually in one screen.

## Mini Project

**Media gallery item (Flutter):** Build a reusable image tile: fixed box, `BoxFit.cover` , downscaled decode, loading spinner, and error fallback, plus a bundled asset header. Acceptance: assets declared; network loading/error handled; memory-conscious decode; no layout jumps; app runs.

# Input & Forms (`TextField`, `Form`, validation, buttons, gestures)

> Collect input with `TextField` / `TextFormField`, group and validate with `Form` + `GlobalKey<FormState>`, and handle taps with buttons or `GestureDetector` / `InkWell` — always disposing controllers.

## Introduction

This file covers user input: text fields (+controllers/focus), the `Form` system (validation, save), buttons (`ElevatedButton` / `TextButton` / `OutlinedButton`), and gesture handling (`GestureDetector`, `InkWell`). It's the interactive half of UI.

## Why this concept exists

Apps need to capture and validate user input reliably: manage text state, focus, keyboard, validation errors, and submit. Flutter's `Form` / `FormField` system coordinates validation/saving across fields, and controllers manage field state and cursor.

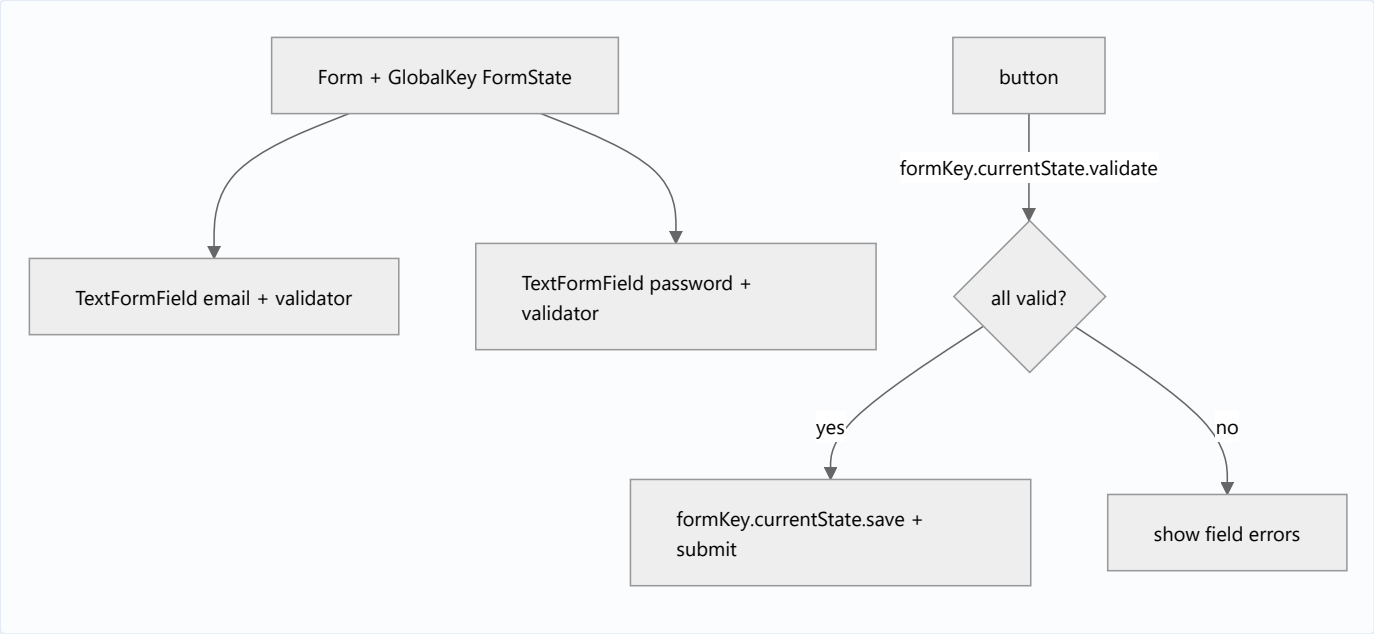
## Real-world analogy

A **paper form with a clerk**: each blank is a field, the clerk (FormState) checks all blanks are valid before accepting (validate), then files the answers (save). Buttons are the "submit" and "cancel" stamps.

## Problem Statement

Build a login form: email + password fields with validation, a submit button enabled logic, focus movement, and proper controller disposal. You'll use `Form`, `TextFormField`, controllers, and `GlobalKey<FormState>`.

## Internal Working




- `TextField` : raw input; use a `TextEditingController` to read/set text and a `FocusNode` for focus.
- `TextFormField` : `TextField` integrated with `Form` — has `validator` and `onSaved` .
- `Form` + `GlobalKey<FormState>` : `formKey.currentState!.validate()` runs all validators; `.save()` triggers `onSaved` ; `AutovalidateMode` controls when validation runs.
- **Buttons**: `ElevatedButton` / `FilledButton` / `TextButton` / `OutlinedButton` / `IconButton` ; disabled when `onPressed: null` .
- **Gestures**: `GestureDetector` (raw taps/drags), `InkWell` (Material ripple), `onTap` / `onLongPress` .
- **Controllers/FocusNodes must be disposed** in `State.dispose` (Module 08, 02 · memory_and_gc).

## Memory Representation

Controllers/FocusNodes are objects held by `State` ; not disposing them leaks (02 · memory_and_gc).

## Compiler Behavior / Runtime Behavior

Validators run on `validate()` /autovalidate; returning a non-null string shows an error under the field. `onPressed: null` renders a disabled button.

## Flutter Engine Behavior

Text input coordinates with the platform keyboard/IME via the engine/embedder (06 · architecture_overview).

## Dart VM Behavior

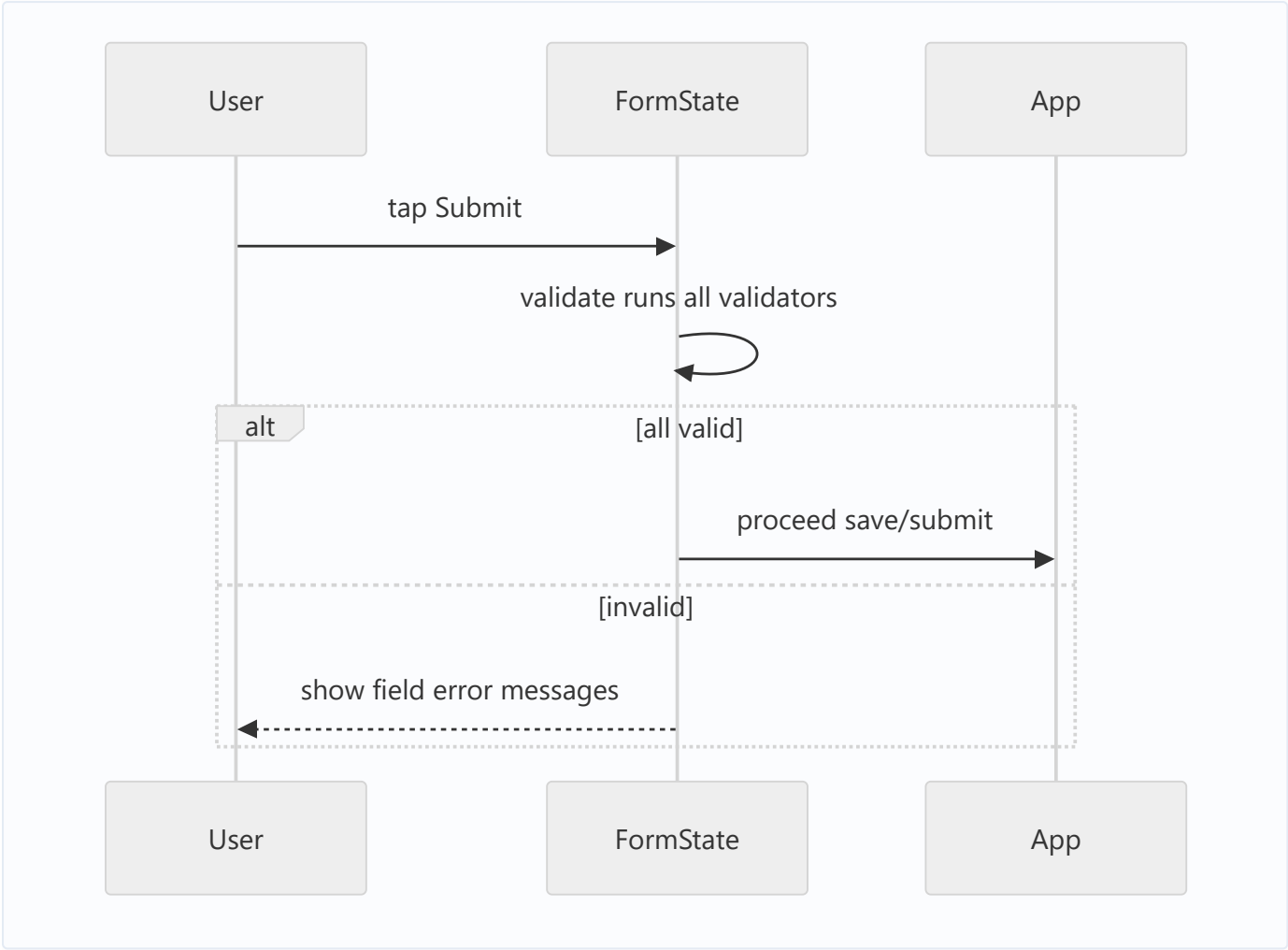
Not applicable.

## Examples

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


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


class _LoginFormState extends State<LoginForm> {
  final _formKey = GlobalKey<FormState>();
  final _email = TextEditingController();
  final _password = TextEditingController();
  final _passwordFocus = FocusNode();


  @override
  void dispose() {
    _email.dispose();          // MUST dispose controllers/focus nodes
    _password.dispose();
    _passwordFocus.dispose();
    super.dispose();
  }


  void _submit() {
    if (_formKey.currentState!.validate()) {
      // valid -> proceed (call a view model / repository)
      debugPrint('login ${_email.text}');
    }
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Login')),
      body: Padding(
```

```
        padding: const EdgeInsets.all(16),
        child: Form(
          key: _formKey,
          child: Column(
            children: [
              TextFormField(
                controller: _email,
                decoration: const InputDecoration(labelText: 'Email'),
                keyboardType: TextInputType.emailAddress,
                textInputAction: TextInputAction.next,
                onFieldSubmitted: (_) => _passwordFocus.requestFocus(),
                validator: (v) =>
                    (v == null || !v.contains('@')) ? 'Enter a valid email' : null,
              ),
              TextFormField(
                controller: _password,
                focusNode: _passwordFocus,
                obscureText: true,
                decoration: const InputDecoration(labelText: 'Password'),
                validator: (v) =>
                    (v == null || v.length < 6) ? 'Min 6 characters' : null,
                onFieldSubmitted: (_) => _submit(),
              ),
              const SizedBox(height: 16),
              FilledButton(onPressed: _submit, child: const Text('Sign in')),
              TextButton(onPressed: () {}, child: const Text('Forgot password?')),
            ],
          ),
        ),
      ),
    );
  }
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Not disposing controllers/focus nodes | Memory leak | `dispose()` them in `State.dispose` |
| Managing text via `setState` on every keystroke | Rebuild churn | Use the controller; read on submit |
| No `Form` /validators for multi-field input | Manual, error-prone | Use `Form` + `TextFormField` validators |
| Business logic in the widget | Untestable | Delegate to a view model/BLoC (Module 11) |
| Using `context` after async submit without `mounted` | Crash | Guard with `if (!mounted) return;` (06 · build_context) |

## Best Practices

- Use `Form` + `TextFormField` + `GlobalKey<FormState>` for multi-field validation.
- **Dispose** controllers/focus nodes; create them in `initState` /as fields.

- Manage focus with `FocusNode` + `textInputAction`; move to next on submit.
- Keep submit logic in a view model; the widget only collects/validates + calls it.
- Choose the right button semantics ( `Filled` / `Elevated` / `Text` / `Outlined` ); disable via `onPressed: null` .

## Performance

Reading text from controllers on submit avoids per-keystroke rebuilds. Gesture/ink handling is cheap (Module 21).

## Advantages / Disadvantages

- **+** Coordinated validation/save, focus/keyboard management, rich input widgets, Material feedback (ink).
- **−** Controller/focus lifecycle to manage (leak-prone); easy to over- `setState` on input.

## Interview Questions

1. 🟢 **`TextField` vs `TextFormField` ?** — `TextFormField` integrates with `Form` (has `validator` / `onSaved` ); `TextField` is standalone.
2. 🟢 **How do you validate a form?** — Wrap fields in a `Form` with a `GlobalKey<FormState>` , give each a `validator` , and call `formKey.currentState!.validate()` .
3. 🟡 **Why dispose `TextEditingController` / `FocusNode` ?** — They're long-lived objects held by `State` ; not disposing leaks memory (02).
4. 🟡 **How do you move focus between fields?** — Use `FocusNode` s + `textInputAction: next` and `onFieldSubmitted` to `requestFocus` the next node.
5. 🟡 **How do you disable a button?** — Set `onPressed: null` .
6. 🔴 **`GestureDetector` vs `InkWell` ?** — Both detect taps; `InkWell` adds Material ripple feedback and must be inside a `Material` . Use `InkWell` for Material tap surfaces.
7. 🔴 **Where should submit/business logic live?** — In a view model/BLoC, not the widget; the form collects+validates and delegates, keeping UI testable.

## Senior Engineer Tips

- Create controllers as `final` fields (or in `initState` ) and dispose them — a top source of Flutter leaks.
- Keep the widget dumb: validate locally, delegate the actual sign-in/save to injected logic.
- Use `autovalidateMode: onUserInteraction` for friendlier validation timing.

## Architect Perspective

Forms are where UI meets domain validation. Keeping presentation-level validation (format) in the form and business rules in the domain/use case (Modules 40, 46) — with the widget delegating submit to a view model — yields testable, maintainable input flows. Controller lifecycle discipline prevents a common leak class at scale.

## Summary

- Collect input with `TextField` / `TextFormField`; validate via `Form` + `GlobalKey<FormState>`.
- Dispose controllers/focus nodes; manage focus + keyboard actions; disable buttons with `null`.
- Delegate submit/business logic to a view model; guard async `context` use.

## Revision Notes

- `Form` + `GlobalKey<FormState>` + `TextFormField.validator`; `validate()` / `save()`.
- Dispose `TextEditingController` / `FocusNode` (leaks!).
- Focus: `FocusNode` + `textInputAction` + `onFieldSubmitted`.
- Disable button: `onPressed: null`; `InkWell` = Material ripple.
- Logic → view model, not widget; guard async with `mounted`.

## Practice Questions

1. Why must controllers be disposed, and where?
2. How does `Form` coordinate multi-field validation?
3. `GestureDetector` vs `InkWell` — when each?

## Coding Questions

1. Build a signup form (email/password/confirm) with cross-field validation.
2. Implement focus traversal that moves to the next field on "next".
3. Make a submit button enable only when the form is valid.

## Mini Project

**Login form (Flutter):** Build a validated `Form` (email + password), focus traversal, a submit that validates and calls an injected `AuthService` fake, proper disposal, and `mounted`-guarded async. Acceptance: validation works; controllers disposed; logic delegated (not in widget); no post-dispose context use; app runs.

# Building Custom Composite Widgets

> Build your own widgets by **composing** existing ones into small, reusable, `const`-friendly units — Flutter's "composition over inheritance" philosophy in daily practice.

## Introduction

Most custom widgets in Flutter are **composite**: a `StatelessWidget` / `StatefulWidget` whose `build` combines existing widgets. This file covers designing reusable widgets — parameters, `const` constructors, callbacks, `child`/builder slots — and when to reach for a custom `RenderObject` instead (rarely; see Module 23).

## Why this concept exists

Copy-pasting the same `Container` + `Row` + `Text` block across screens causes drift and bloats `build` methods. Extracting a named composite widget (`UserCard`, `PrimaryButton`) gives reuse, a clear API, `const` rebuild-skipping, and testability — the practical payoff of composition (03 · composition).

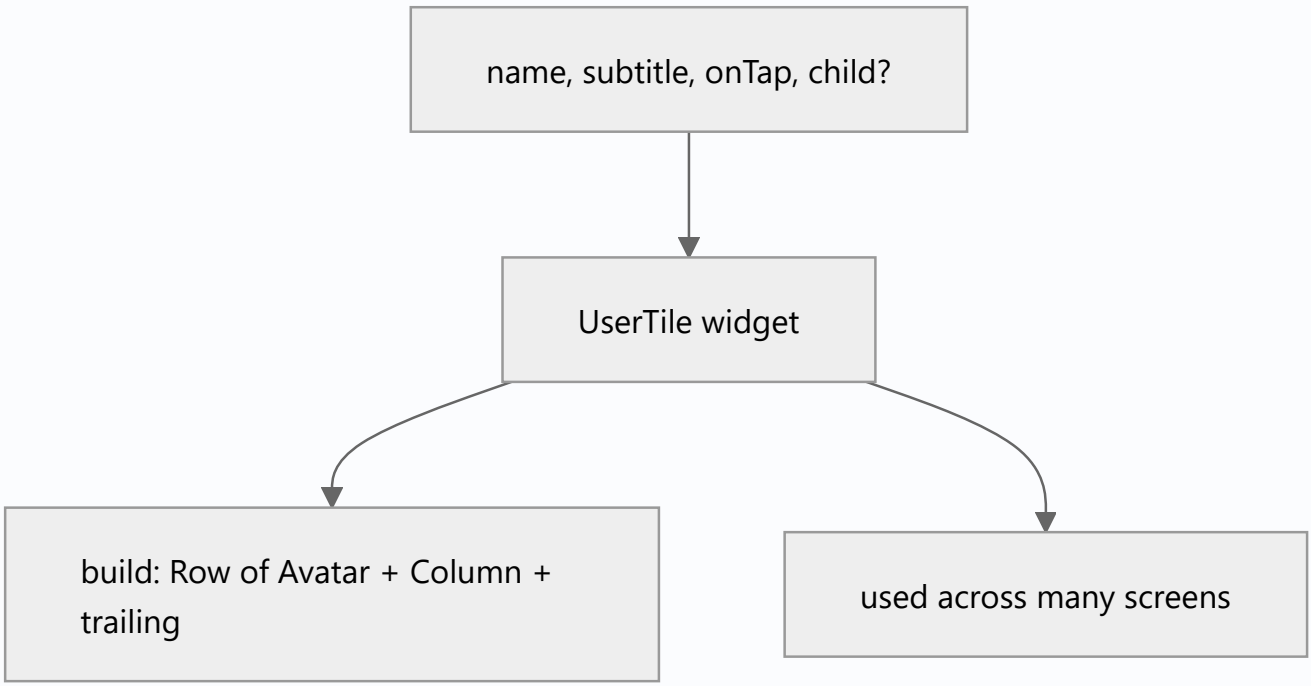
## Real-world analogy

**Prefab components** in construction: instead of building each wall from raw bricks every time, you make a reusable wall panel (custom widget) with defined connectors (parameters/callbacks) and snap it in wherever needed.

## Problem Statement

You keep repeating an avatar+name+subtitle+trailing-action row. You'll extract a configurable, `const`-friendly `UserTile` widget with parameters and an `onTap` callback, and learn to expose a `child` slot for flexibility.

## Internal Working




- Extract a widget when a UI block is **repeated**, **complex**, or **named-concept** worthy.
- Expose a **clear API**: required/optional named params, callbacks ( `VoidCallback` , `ValueChanged<T>` ), and optionally a `child` / `Widget Function(...)` builder slot for flexibility.
- Make it `const` **-constructible** (all fields `final` ) so parents can `const` it and skip rebuilds.
- Prefer `StatelessWidget` unless it owns local state; keep business logic out (06 · stateless_vs_stateful).
- Custom `RenderObject` widgets are for bespoke layout/painting only — rare; see Module 23.

## Memory Representation

Composite widgets are ordinary widgets → elements → render objects; `const` instances are shared/skip rebuilds (06).

## Compiler Behavior

`const` constructors require all- `final` fields and enable canonicalization/rebuild-skipping.

## Runtime Behavior

A composite rebuilds when its inputs change or its parent rebuilds it; `const` instances are skipped when unchanged.

## Flutter Engine Behavior / Dart VM Behavior

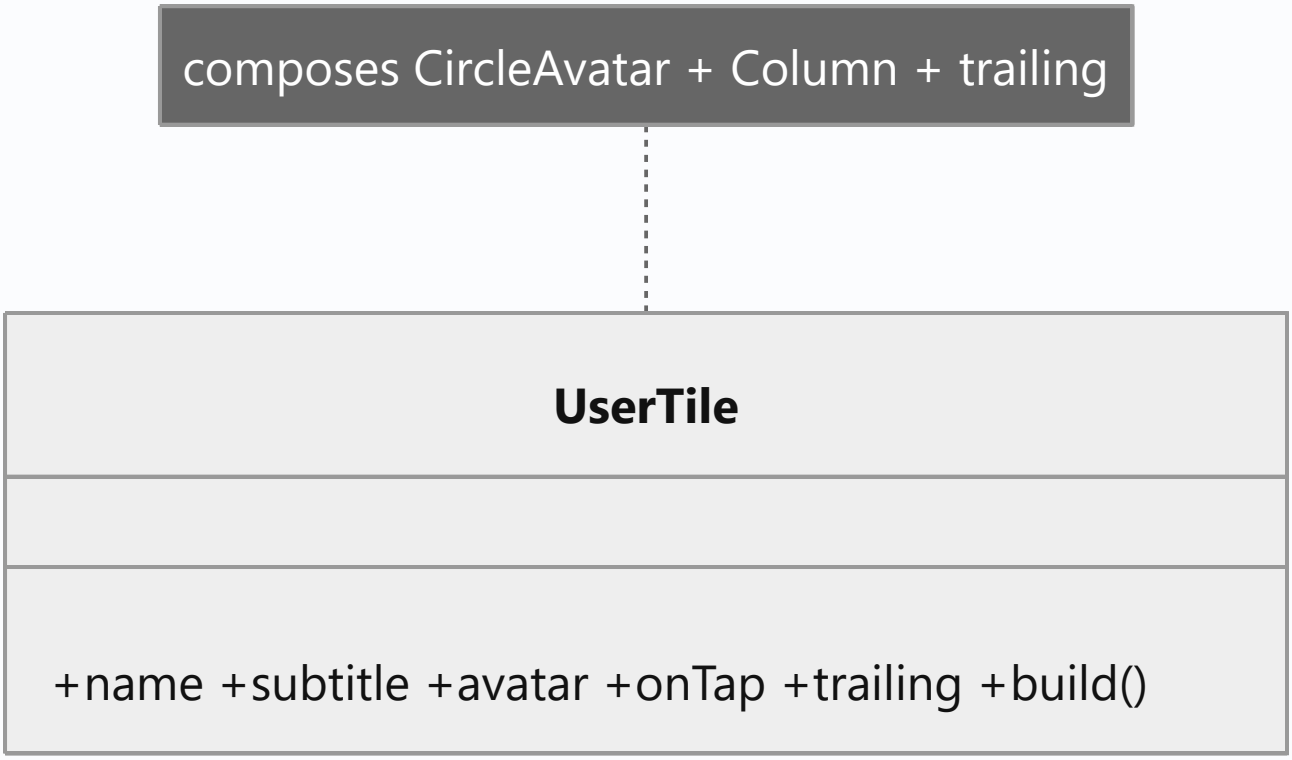
Not applicable beyond normal widget behavior.

## Examples

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


// A reusable, const-friendly composite widget with a clear API.
class UserTile extends StatelessWidget {
  final String name;
  final String subtitle;
  final ImageProvider? avatar;
  final VoidCallback? onTap;   // callback slot
  final Widget? trailing;      // widget slot for flexibility


  const UserTile({
    super.key,
    required this.name,
    required this.subtitle,
    this.avatar,
    this.onTap,
    this.trailing,
  });


  @override
  Widget build(BuildContext context) {
    final theme = Theme.of(context);
    return InkWell(
      onTap: onTap,
      child: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 8),
        child: Row(
          children: [
            CircleAvatar(backgroundImage: avatar, child: avatar == null ? Text(name[0]) : null),
            const SizedBox(width: 12),
            Expanded(
              child: Column(
                crossAxisAlignment: CrossAxisAlignment.start,
                children: [
```

```
                  Text(name, style: theme.textTheme.titleMedium),
                  Text(subtitle, style: theme.textTheme.bodySmall),
                ],
              ),
            ),
            if (trailing != null) trailing!,
          ],
        ),
      ),
    );
  }
}

// Usage — reusable across screens, const where inputs are constant:
class Demo extends StatelessWidget {
  const Demo({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ListView(
        children: [
          UserTile(
            name: 'Ada',
            subtitle: 'Engineer',
            trailing: const Icon(Icons.chevron_right),
            onTap: () {},
          ),
          const UserTile(name: 'Bob', subtitle: 'Designer'), // const!
        ],
      ),
    );
  }
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
| --- | --- | --- |
| Giant `build` methods, copy-pasted blocks | Drift, unreadable | Extract named composite widgets |
| Non-`const` constructors (mutable fields) | Lose rebuild-skipping | Make fields `final`, add `const` ctor |
| Business logic inside the widget | Untestable, coupled | Delegate via callbacks / view models |
| Extracting into a *method* returning a widget | Doesn't get its own element/ `const` /rebuild scope | Extract a **widget class**, not a helper method |
| Over-parameterizing | Confusing API | Provide sensible defaults; use `child` /builder slots |

## Best Practices

- Extract a **widget class** (not a `Widget _buildX()` method) so it gets its own element, `const`-ability, and rebuild scoping.
- Design a **small, clear API**: required/optional named params, callbacks, and `child` /builder slots.

- Make it `const`-**constructible**; keep it **stateless** unless it owns local state.
- Keep logic out (delegate via callbacks); theme via `Theme.of(context)` not hardcoded.
- Compose small widgets; each has a single responsibility (04 · SRP).

## Performance

Widget classes (vs helper methods) enable `const` + independent rebuild scoping, reducing unnecessary rebuilds — a real, common perf win (Module 21).

## Advantages / Disadvantages

- + Reuse, clear APIs, `const`/rebuild scoping, testability, readable `build`s.
- − More files/types; risk of over-abstracting tiny one-offs.

## Interview Questions

1. 🟢 **How do you build a custom widget in Flutter?** — Usually by composing existing widgets in a `Stateless`/`StatefulWidget`'s `build` (composition over inheritance).
2. 🟢 **Why extract a widget class instead of a `_buildX()` method?** — A class gets its own element, can be `const`, and rebuilds independently; a method inlines into the parent's build (no scoping/`const` benefit).
3. 🟡 **How do you make a custom widget flexible?** — Expose parameters, callbacks (`VoidCallback`/`ValueChanged`), and `child`/builder slots.
4. 🟡 **Why make custom widgets `const`-constructible?** — `const` instances are canonicalized and skipped during rebuilds, improving performance.
5. 🟡 **Stateless or stateful for a custom widget?** — Stateless unless it owns local mutable state; keep business logic out regardless.
6. 🔴 **When do you need a custom `RenderObject` widget?** — Only for bespoke layout/painting not expressible by composition — rare; see Module 23.
7. 🔴 **How does extracting widgets help rebuild performance?** — It scopes rebuilds to the smallest subtree and enables `const`, so unrelated parts don't rebuild.

## Senior Engineer Tips

- Prefer widget classes over build-helper methods — it's a genuine performance and clarity win, not just style.
- Design custom widgets like a public API: minimal required params, sensible defaults, escape-hatch `child`/builder slots.
- Build a small design-system package of composites (buttons, cards, tiles) for consistency across teams.

## Architect Perspective

Custom composite widgets are the atoms of a **design system**: reusable, themed, tested components with clear APIs. Investing in them early yields UI consistency, faster feature work, and rebuild efficiency across a large app — the UI-layer parallel to a well-factored domain (Modules 25, 47).

## Summary

- Build custom widgets by composing existing ones into small, `const`-friendly, single-purpose classes.
- Expose clear parameters/callbacks/slots; keep logic out; extract widget *classes*, not helper methods.
- This is composition-over-inheritance in practice and the basis of a design system.

## Revision Notes

- Compose existing widgets into a `Stateless` / `Stateful` class (not a `_buildX()` method).
- Clear API: named params + callbacks + `child`/builder slots; `const` ctor (final fields).
- Stateless unless owns state; logic → callbacks/view models; theme via `Theme.of`.
- Widget class ⇒ `const` + rebuild scoping (perf). RenderObject only for bespoke layout/paint.

## Practice Questions

1. Why is a widget class better than a build-helper method?
2. How do slots (`child`/builder) improve a widget's flexibility?
3. When would you actually need a custom `RenderObject`?

## Coding Questions

1. Extract a repeated card block into a `const`-constructible `InfoCard` with params + `onTap`.
2. Add a `trailing` widget slot and a builder slot to a custom list tile.
3. Convert three `_buildX()` helper methods into widget classes and note the benefits.

## Mini Project

**Design-system starter kit (Flutter):** Build `PrimaryButton`, `UserTile`, and `SectionCard` as reusable, themed, `const`-friendly composite widgets with clear APIs (params, callbacks, slots), and a demo screen using them. Acceptance: widget classes (not helper methods); `const` where possible; logic delegated via callbacks; consistent theming; app runs.