24 · Responsive UI

Flutter Practice Notes

Contents

24 · Responsive UI

Introduction

Responsive UI adapts a **single layout** to any screen size/orientation — phone, tablet, desktop, web, split-screen — using the constraints model, `MediaQuery` / `LayoutBuilder`, flexible widgets, and breakpoints. This module builds directly on the constraints system (07 · constraints_and_sizing).

Why this module exists

Flutter runs everywhere; a phone-only layout breaks on tablets/desktop/web. Responsive design (fluid + breakpoint-based) is essential for reach and quality. This is distinct from **adaptive** UI (platform conventions/widgets), which is Module 25.

Module map

#	File	Topic	Level
1	01_responsive_fundamentals.md	Principles, breakpoints, mobile→desktop	🟢
2	02_mediaquery_vs_layoutbuilder.md	Global vs local size/constraints	🔵
3	03_responsive_layout_widgets.md	`Flexible` / `Wrap` / `AspectRatio` / `FractionallySizedBox`	🔵
4	04_adaptive_grids_and_split_views.md	Responsive grids, master-detail/split views	🔴
5	05_responsive_typography_and_orientation.md	Text scaling, spacing, orientation, safe areas	🔵

> **Cross-references:** Constraints model: 07 · constraints_and_sizing. Flex layout: 07 · layout_flex. Adaptive (platform) UI: Module 25. Web/desktop: Modules 53/54. Theming: 07 · text_and_theming.

Prerequisites

07 Widgets — especially constraints and flex.

What you'll be able to do after this module

- Design fluid + breakpoint-based layouts for phone/tablet/desktop/web.
- Choose `MediaQuery` vs `LayoutBuilder` correctly.
- Use flexible/wrapping widgets and responsive grids.
- Build master-detail/split-view layouts that adapt to width.
- Handle text scaling, spacing, orientation, and safe areas.

Capstone

Adaptive product screen: A screen that shows a single scrolling list on phones and a master-detail split view on tablet/desktop, with a responsive grid, fluid spacing/typography, and orientation/safe-area handling — one codebase, many sizes.

Summary

Responsive = one layout, many sizes. Combine fluid sizing (constraints/flex/fractions) with breakpoints (`MediaQuery` / `LayoutBuilder`), reflow grids and switch to split views on large screens, and respect text scale/orientation/safe areas.

Responsive Fundamentals (Breakpoints & Principles)

> Responsive UI = a single layout that flows and reconfigures across sizes: use **fluid sizing** (flex/fractions/constraints) for continuous adaptation and **breakpoints** (width thresholds) to switch layout structure (e.g., list → split view) between phone, tablet, and desktop.

Introduction

Before tools, the principles: think in **constraints and available space**, not fixed pixels; combine **fluid** layout (stretches with size) with **breakpoint** decisions (change structure at thresholds); and design mobile-first, enhancing for larger screens. This grounds the `MediaQuery` / `LayoutBuilder` /grid files.

Why this concept exists

Fixed pixel layouts break outside the device they were designed for. Flutter targets phones→desktop→web with wildly different sizes; responsive principles produce one codebase that looks right everywhere, improving reach and quality.

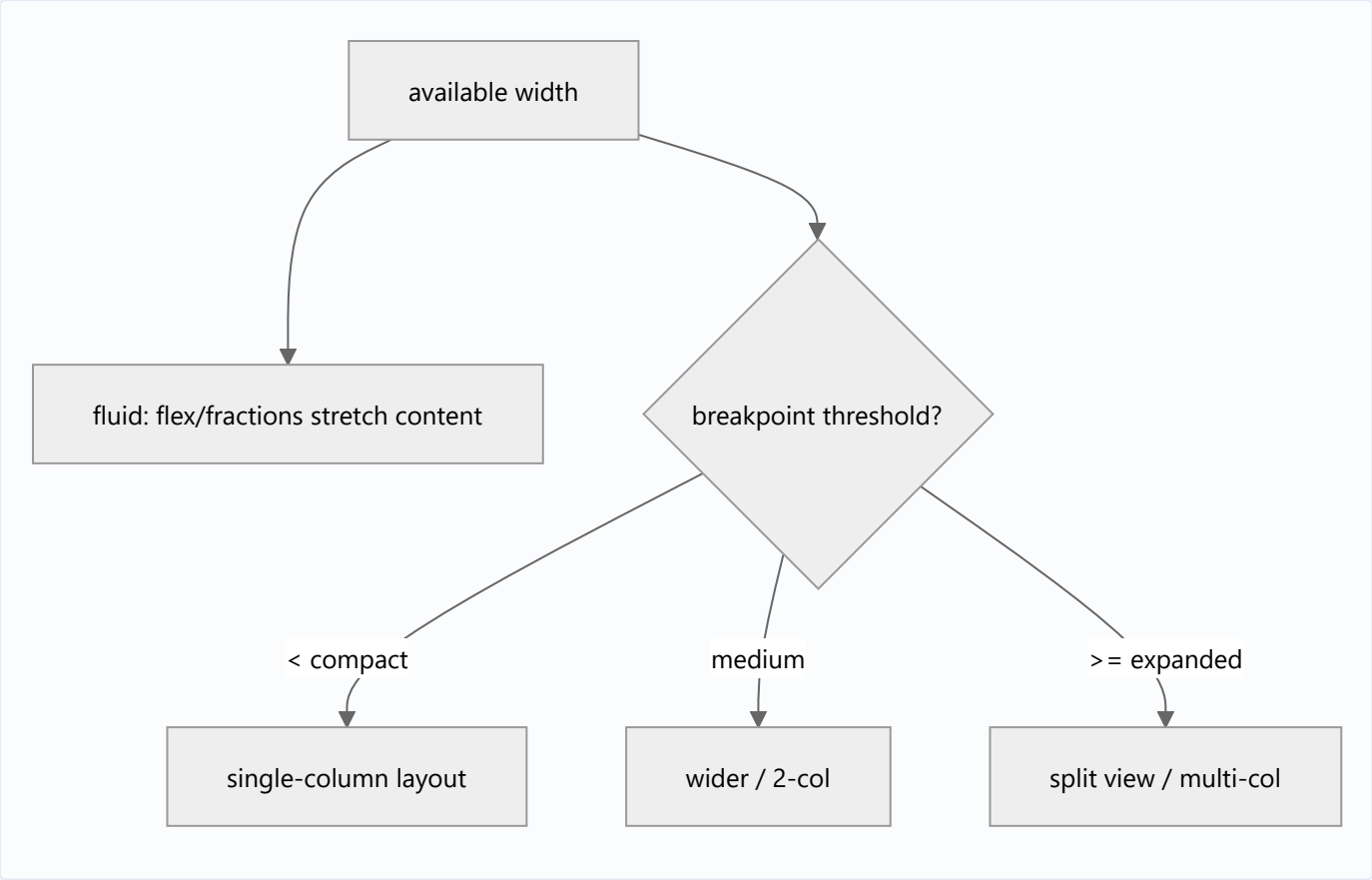
Real-world analogy

Responsive design is **water taking the shape of its container**: it fills small and large glasses (fluid), but at a certain size you switch containers entirely — a cup vs a pitcher with a spout (breakpoint structural change). Same water, right form per size.

Problem Statement

A product screen should be a single scrolling list on phones but a two-pane master-detail on tablets/desktop, with content that fluidly fills the width in between. You'll define breakpoints + fluid behavior and a mobile-first approach.

Internal Working



- **Two mechanisms**:
  - **Fluid**: content stretches/shrinks continuously with available space (flex, fractions, constraints — 03_responsive_layout_widgets.md). No thresholds.
  - **Breakpoints**: at width thresholds, change **structure/layout** (columns, split views, navigation pattern). Common tiers: compact (< ~600), medium (~600–840/1024), expanded (≥ ~840/1024) — roughly Material's window size classes; pick sensible values for your app.
- **Mobile-first**: design the smallest layout first, then *add* structure/columns for larger screens (progressive enhancement).
- **Base on layout width, not device**: decide by the **space the layout has** (from `LayoutBuilder` / `MediaQuery`), not "is it a phone" — because of split-screen, foldables, web resizing, and embedded panes.
- **Navigation adapts too**: bottom nav (compact) → navigation rail/drawer (expanded).
- Distinct from **adaptive** (platform conventions/widgets per OS — Module 25): responsive is about *size*, adaptive about *platform*.

Memory Representation

Not applicable; layout decisions are per-frame based on size (07 · constraints_and_sizing).

Compiler Behavior / Runtime Behavior

Layout re-evaluates on size/orientation changes (resize, rotate, split-screen); breakpoint branches pick a structure at build/layout time.

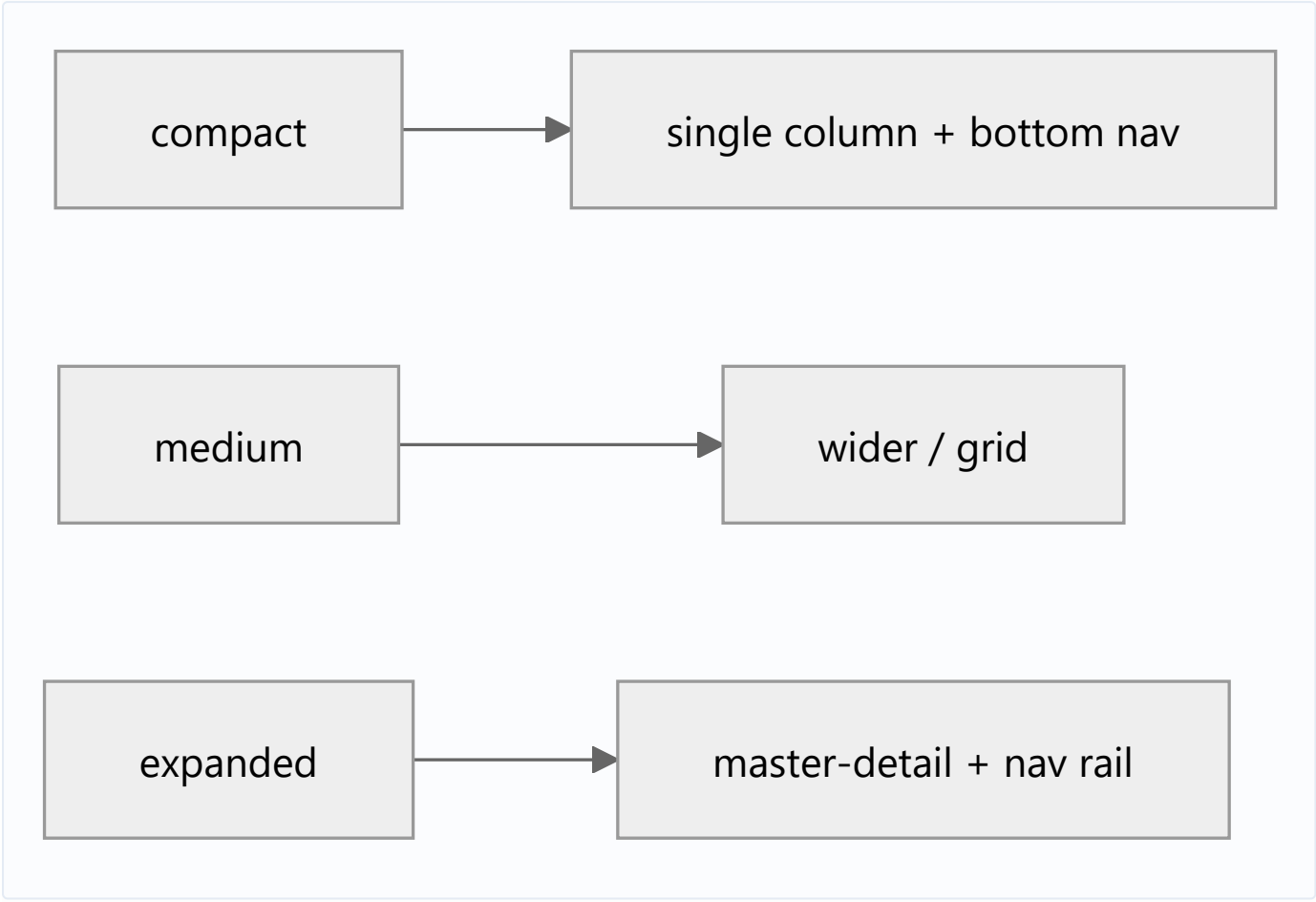
Flutter Engine Behavior / Dart VM Behavior

Not applicable.

Examples

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

// Breakpoint tiers (choose values to fit your design)
enum WindowSize { compact, medium, expanded }
WindowSize windowSizeFor(double width) =>
    width < 600 ? WindowSize.compact
    : width < 1024 ? WindowSize.medium
    : WindowSize.expanded;

// Mobile-first: base layout, enhanced for larger widths
class ProductScreen extends StatelessWidget {
  const ProductScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return LayoutBuilder(
      builder: (context, constraints) {
        final size = windowSizeFor(constraints.maxWidth); // decide by layout WIDTH
        return switch (size) {
          WindowSize.compact => const _ListOnly(),           // phone: single column
          WindowSize.medium => const _WiderList(),           // tablet: wider/2-col
          WindowSize.expanded => const _MasterDetail(),      // desktop: split view
        };
      },
    );
  }
}
class _ListOnly extends StatelessWidget { const _ListOnly(); @override Widget build(_) => const
Placeholder(); }
class _WiderList extends StatelessWidget { const _WiderList(); @override Widget build(_) => const
Placeholder(); }
class _MasterDetail extends StatelessWidget { const _MasterDetail(); @override Widget build(_) => const
Placeholder(); }
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Fixed pixel sizes	Breaks across sizes	Fluid (flex/fractions/constraints) + breakpoints
Deciding by "device type"	Ignores split-screen/foldables/web resize	Decide by available **width**
Only breakpoints (no fluid)	Rigid between thresholds	Combine fluid + breakpoints
Desktop-first design	Hard to shrink to mobile	Mobile-first + enhance
Not adapting navigation	Bottom nav on wide screens feels wrong	Rail/drawer on expanded
Hardcoding many device sizes	Fragile	A few semantic tiers

Best Practices

- Combine **fluid sizing** (continuous) with **breakpoints** (structural switches at width tiers).
- Design **mobile-first**; progressively enhance for medium/expanded.
- Decide by **available layout width** (`LayoutBuilder` / `MediaQuery`), not device type.
- Use a small set of **semantic breakpoints** (compact/medium/expanded) app-wide.
- Adapt **navigation** (bottom nav → rail/drawer) and structure (list → split view) per tier.

- Keep responsive logic centralized (helpers/extensions) for consistency.

Performance

Layout re-evaluation on resize/rotate is cheap; branching by tier is negligible. Avoid heavy work in size-dependent builders (21 · rebuild_optimization).

Advantages / Disadvantages

- + One codebase across sizes, better UX/reach, future-proof (foldables/web/desktop), quality on all form factors.
- − Design/testing effort across sizes; branching complexity; must handle continuous *and* threshold behavior.

Interview Questions

1. 🟢 **What is responsive UI?** — A single layout that adapts to size/orientation via fluid sizing and breakpoint-based structural changes.
2. 🟢 **Fluid vs breakpoint responsiveness?** — Fluid stretches continuously with space (flex/fractions); breakpoints switch layout structure at width thresholds.
3. 🟡 **Why decide by layout width, not device type?** — Split-screen, foldables, embedded panes, and web resizing mean "phone/tablet" is unreliable; the available width is what matters.
4. 🟡 **What is mobile-first design?** — Start with the smallest layout and progressively add structure/columns for larger screens.
5. 🟡 **Responsive vs adaptive?** — Responsive adapts to *size*; adaptive conforms to *platform conventions/widgets* (Material/Cupertino — Module 25).
6. 🔴 **How should navigation change across tiers?** — Bottom navigation on compact → navigation rail/drawer on expanded; list → master-detail split on large widths.
7. 🔴 **Why use semantic breakpoints (compact/medium/expanded)?** — Fewer, meaningful tiers are maintainable and align with window size classes vs hardcoding many device dimensions.

Senior Engineer Tips

- Define breakpoints and a `windowSizeFor(width)` helper (or extension) once; use it everywhere for consistent, testable responsiveness.
- Combine both mechanisms: fluid within a tier, structural switch between tiers.
- Test at continuous sizes (resize a desktop/web window) and split-screen — not just a few device presets.

Architect Perspective

Responsive strategy is a UI-architecture decision spanning navigation, layout structure, and content flow across all form factors. Centralizing breakpoints + a mobile-first, width-driven approach yields a maintainable single codebase for phone→desktop→web, and pairs with adaptive platform conventions (Module 25) for polished cross-platform apps.

Summary

- Responsive = one layout adapting to size via fluid sizing + breakpoints (compact/medium/expanded).
- Mobile-first, decide by available width (not device), adapt navigation and structure per tier.
- Distinct from adaptive (platform); centralize breakpoints for consistency.

Revision Notes

- Fluid (flex/fractions/constraints, continuous) + breakpoints (structure switch at width tiers).
- Tiers: compact/medium/expanded (semantic, not device-hardcoded); decide by layout width.
- Mobile-first + progressive enhancement; adapt navigation (bottom→rail) + structure (list→split).
- Responsive (size) ≠ adaptive (platform, Module 25); centralize breakpoints.

Practice Questions

1. Why combine fluid and breakpoint responsiveness?
2. Why decide by width instead of device type?
3. How does navigation adapt from compact to expanded?

Coding Questions

1. Write a `windowSizeFor(width)` helper with compact/medium/expanded tiers.
2. Build a screen that switches list→split view at a breakpoint via `LayoutBuilder`.
3. Adapt navigation (bottom nav → nav rail) by tier.

Mini Project

Breakpoint scaffold (Flutter): Build a screen using a `windowSizeFor(width)` helper that renders a single column (compact), a wider/2-col layout (medium), and a master-detail split (expanded), plus adaptive navigation (bottom nav ↔ nav rail). Test by resizing. Acceptance: width-driven tiers; mobile-first; fluid within tiers; navigation adapts; runs across sizes.

`MediaQuery` vs `LayoutBuilder`

> `MediaQuery` gives **global** screen/window metrics (size, orientation, text scale, insets); `LayoutBuilder` gives the **local** constraints a specific widget was handed — use `LayoutBuilder` to respond to the space a widget actually has, and `MediaQuery` for device-wide info.

Introduction

Both inform responsive decisions, but at different scopes. `MediaQuery.of(context)` = the whole screen/window; `LayoutBuilder(builder: (context, constraints))` = *this* widget's available box. This file clarifies when to use which — a frequent source of responsive bugs.

Why this concept exists

A widget inside a sidebar/split-pane/card has far less width than the screen. Deciding its layout from `MediaQuery` (screen size) would be wrong; it should decide from its **local constraints** (`LayoutBuilder`). Conversely, device orientation/insets/text-scale are screen-wide — `MediaQuery`.

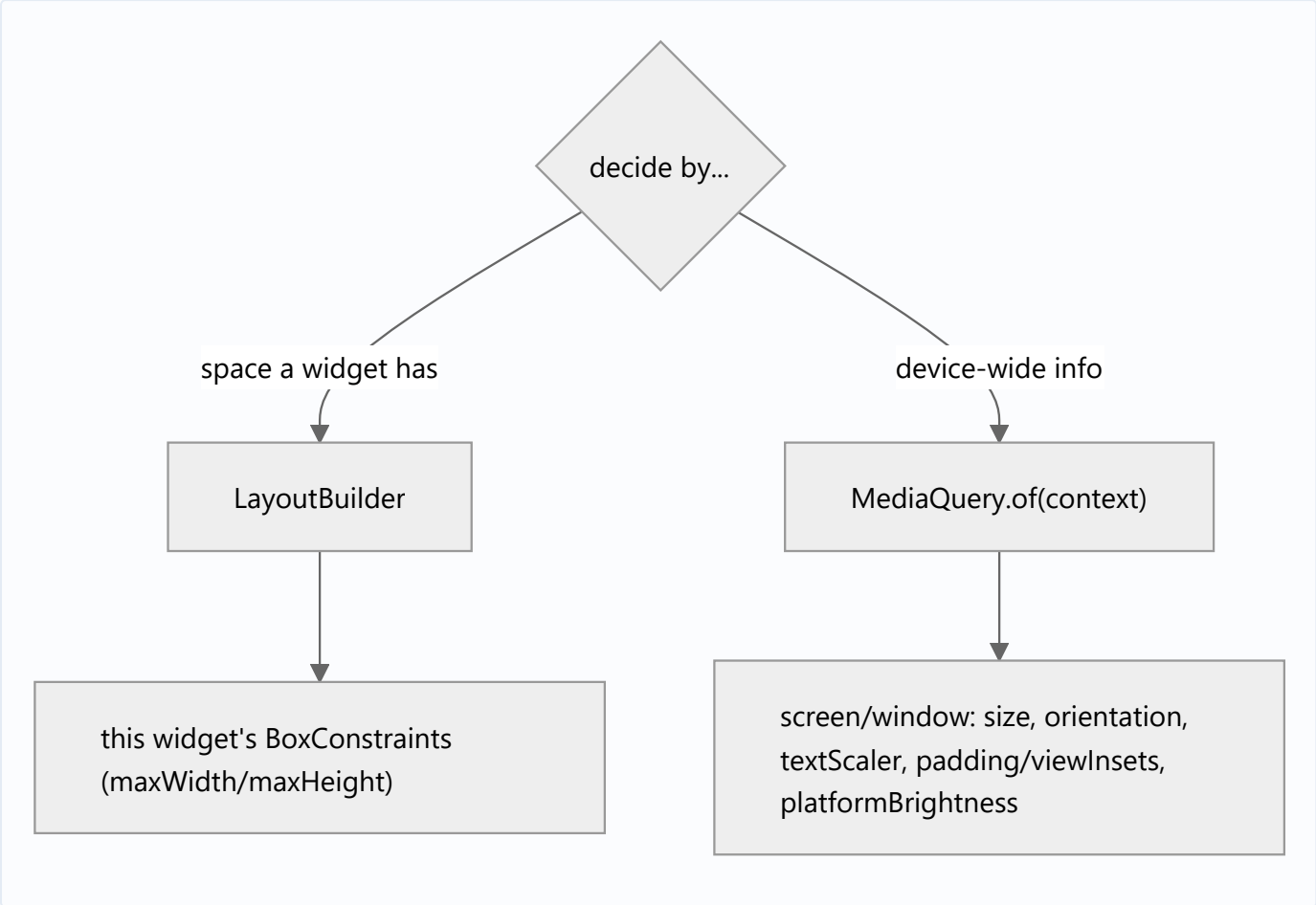
Real-world analogy

`MediaQuery` is knowing the **whole building's dimensions**; `LayoutBuilder` is knowing **the size of the room you're standing in**. To arrange furniture, you use the room's size (local), not the building's — but for building-wide facts (which floor, is it daytime) you ask the building.

Problem Statement

A card must show 1 or 2 columns based on **its own** width (it lives in a variable-width pane), while the app switches navigation based on **screen** width and respects the device's text scale. You'll use `LayoutBuilder` for the card and `MediaQuery` for screen-level decisions.

Internal Working



- `MediaQuery.of(context)` : `size` (screen/window logical px), `orientation`, `textScaler` / `textScaleFactor` (user font scaling), `padding` / `viewInsets` / `viewPadding` (safe areas, keyboard), `platformBrightness`, `devicePixelRatio`. **Global** to the window; subscribes the widget to changes (06 · build_context).
- `LayoutBuilder` : builds with the `BoxConstraints` this widget received from its parent — the *actual* available space (respects sidebars, split panes, cards). **Local**.
- **Rule of thumb**: layout a widget by **local constraints** (`LayoutBuilder`) so it's correct wherever it's placed; use `MediaQuery` for screen-wide facts (orientation, insets, text scale, brightness) and top-level structure.
- `MediaQuery.sizeOf(context)` / `orientationOf` etc. (newer, granular) subscribe only to that aspect — fewer rebuilds than whole- `MediaQuery.of` .
- **Don't** decide inner-widget layout from `MediaQuery.size` — it's the screen, not the widget's box (a classic bug).

Memory Representation

`MediaQuery` data is provided via an `InheritedWidget` (11 · inherited_widget); `LayoutBuilder` reads incoming constraints during layout. Both cheap.

Compiler Behavior

Not applicable.

Runtime Behavior

`MediaQuery.of` rebuilds the widget when *any* media data changes (use `sizeOf` / `orientationOf` to scope); `LayoutBuilder` rebuilds when its incoming constraints change (parent resizes it).

Flutter Engine Behavior / Dart VM Behavior

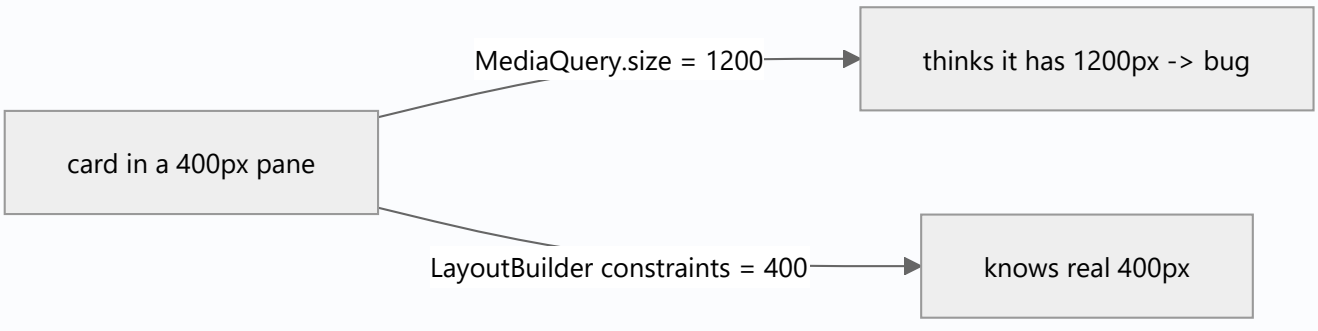
Not applicable.

Examples

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


class ResponsiveDemo extends StatelessWidget {
  const ResponsiveDemo({super.key});
  @override
  Widget build(BuildContext context) {
    // Screen-level: orientation + text scale (global) via MediaQuery
    final orientation = MediaQuery.orientationOf(context);   // scoped subscription
    final screenWidth = MediaQuery.sizeOf(context).width;


    return Scaffold(
      // Top-level navigation decided by SCREEN width:
      body: Row(children: [
        if (screenWidth >= 1024) const NavigationRail(destinations: [
          NavigationRailDestination(icon: Icon(Icons.home), label: Text('Home')),
          NavigationRailDestination(icon: Icon(Icons.person), label: Text('Me')),
        ], selectedIndex: 0),
        Expanded(
          child: Padding(
            padding: const EdgeInsets.all(16),
            child: Column(children: [
              Text('Orientation: ${orientation.name}'),
              // A card that decides ITS OWN columns by LOCAL width (not screen):
              Expanded(child: _AdaptiveCard()),
            ]),
          ),
```

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

class _AdaptiveCard extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return LayoutBuilder(
      builder: (context, constraints) {
        // Decide by the WIDTH THIS WIDGET has (correct inside panes/sidebars):
        final twoColumns = constraints.maxWidth >= 500;
        return twoColumns
            ? const Row(children: [Expanded(child: _Pane('A')), Expanded(child: _Pane('B'))])
            : const Column(children: [_Pane('A'), _Pane('B')]);
      },
    );
  }
}
class _Pane extends StatelessWidget {
  final String label; const _Pane(this.label);
  @override Widget build(_) => Card(child: Center(child: Text(label)));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using `MediaQuery.size` for inner-widget layout	It's the screen, not the widget's box	Use `LayoutBuilder` constraints
Whole `MediaQuery.of` for one metric	Rebuilds on any media change	Use `sizeOf` / `orientationOf` /etc.
`LayoutBuilder` for device-wide facts	It doesn't know orientation/insets	Use `MediaQuery` for those
Reading `MediaQuery` above `MaterialApp`	Not provided yet	Read below `MaterialApp`
Ignoring insets/text scale	Overlap/clipping	Use padding/ `textScaler` (see typography file)

Best Practices

- **Layout widgets by local constraints** (`LayoutBuilder`) so they're correct in any container (pane/card/sidebar).
- Use `MediaQuery` for **screen-wide** info: orientation, insets/safe areas, text scale, brightness, and top-level structure/navigation.
- Prefer **scoped accessors** (`MediaQuery.sizeOf` / `orientationOf`) to reduce rebuilds.
- Don't decide inner layout from `MediaQuery.size` ; combine both (screen-level structure + widget-level fluidity).

Performance

`MediaQuery.of` subscribes to all media changes → broader rebuilds; scoped `*Of` accessors are cheaper. `LayoutBuilder` rebuilds only when its constraints change — efficient and correct for local decisions (21 · rebuild_optimization).

Advantages / Disadvantages

- **+ MediaQuery:** device-wide metrics for global decisions. **+ LayoutBuilder:** correct local sizing anywhere.
- **− MediaQuery:** wrong for inner-widget sizing; whole- `of` over-rebuilds. **− LayoutBuilder:** doesn't know screen-wide facts; extra nesting.

Interview Questions

1. `MediaQuery` **vs** `LayoutBuilder` **?** — `MediaQuery` = global screen/window metrics; `LayoutBuilder` = the local constraints (available box) this widget received.

2. 🟢 **Which do you use to size a widget inside a sidebar?** — `LayoutBuilder` — it knows the widget's actual width, not the screen's.
3. 🟡 **What does `MediaQuery` provide?** — Size, orientation, text scale, padding/view insets (safe areas/keyboard), platform brightness, device pixel ratio.
4. 🟡 **Why prefer `MediaQuery.sizeOf` over `MediaQuery.of`?** — It subscribes only to size changes, avoiding rebuilds when unrelated media data changes.
5. 🟡 **Why is `MediaQuery.size` wrong for inner-widget layout?** — It reports the whole screen, not the constrained box the widget occupies (split panes/cards make them differ).
6. 🔴 **When would `LayoutBuilder` and `MediaQuery` give different widths?** — Whenever the widget isn't full-screen — inside padding/split views/sidebars/cards, its local width < screen width.
7. 🔴 **How do you combine them for a responsive screen?** — `MediaQuery` for top-level structure/navigation (screen size/orientation) + `LayoutBuilder` for each component's fluid layout by its own space.

Senior Engineer Tips

- Default to `LayoutBuilder` for component layout — it's correct regardless of where the widget is placed; reserve `MediaQuery` for genuinely screen-wide concerns.
- Use scoped `MediaQuery.*Of` accessors to keep rebuilds tight.
- When a responsive bug appears in a pane/card, suspect `MediaQuery.size` misuse first.

Architect Perspective

Choosing the right scope (local constraints vs global metrics) is foundational to a correct, composable responsive system: components decide their own layout (portable), while the shell decides structure/navigation from screen metrics. This separation makes components reusable across split views, sidebars, and form factors (01_responsive_fundamentals.md, 04_adaptive_grids_and_split_views.md).

Summary

- `MediaQuery` = global (screen size/orientation/insets/text scale/brightness); `LayoutBuilder` = local (widget's constraints).
- Layout components by local constraints (portable/correct); use `MediaQuery` for screen-wide structure/info.
- Prefer scoped `MediaQuery.*Of`; don't size inner widgets from `MediaQuery.size`.

Revision Notes

- `MediaQuery.of/sizeOf/orientationOf` = screen/window (size/orientation/textScaler/padding/viewInsets/brightness).
- `LayoutBuilder` = this widget's `BoxConstraints` (real available space).
- Component layout → `LayoutBuilder`; screen-wide facts/structure → `MediaQuery`; scoped accessors to limit rebuilds.
- Don't use `MediaQuery.size` for inner-widget sizing (panes/cards differ).

Practice Questions

1. Why size a card by `LayoutBuilder`, not `MediaQuery.size`?
2. What screen-wide facts require `MediaQuery`?
3. Why prefer `MediaQuery.sizeOf` over `.of`?

Coding Questions

1. Build a widget that switches 1↔2 columns by its own width (`LayoutBuilder`).
2. Add a nav rail on wide screens using `MediaQuery.sizeOf`.
3. Show a bug: card using `MediaQuery.size` inside a narrow pane, then fix it with `LayoutBuilder`.

Mini Project

Scope-correct responsive screen (Flutter): Build a screen where top-level structure/navigation is decided by `MediaQuery` (screen width/orientation) and each component decides its own layout via `LayoutBuilder`; place a card inside a narrow pane to prove it uses local width. Acceptance: components layout by local constraints; screen structure by `MediaQuery`; scoped accessors used; correct in panes; runs.

Responsive Layout Widgets (`Flexible`, `Wrap`, `AspectRatio`, `FractionallySizedBox`)

> A toolkit of widgets makes layouts fluid without manual math: `Expanded` / `Flexible` share space, `Wrap` flows to new lines, `FractionallySizedBox` sizes by a fraction of the parent, `AspectRatio` keeps proportions, and `ConstrainedBox` caps max width for readability.

Introduction

Responsive layout is mostly composing the right fluid widgets. This file covers the everyday responsive toolkit — flex distribution, wrapping, fractional/aspect sizing, and max-width constraints — so content adapts to space without breakpoints for the continuous cases.

Why this concept exists

Fluid adaptation (fill remaining space, reflow chips, keep a 16:9 ratio, cap line length) shouldn't require measuring pixels or breakpoints. These widgets express intent ("take the rest," "wrap," "half the width," "keep ratio") and the layout engine computes sizes (07 · constraints_and_sizing).

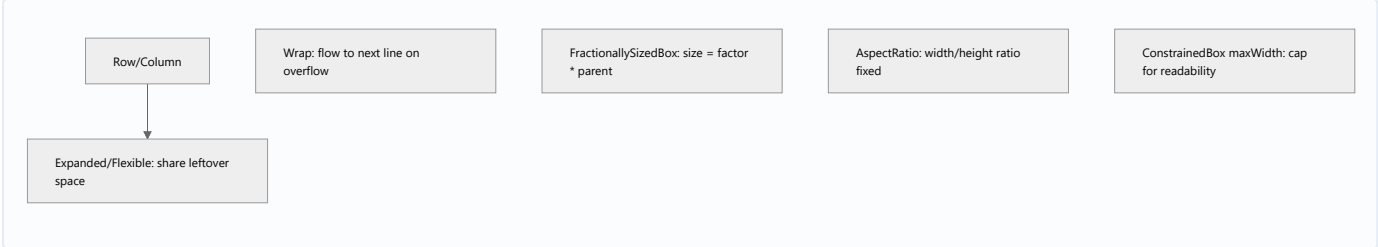
Real-world analogy

These are **adjustable furniture**: `Expanded` is a couch that stretches to fill the room; `Wrap` is shelving that adds rows when full; `FractionallySizedBox` is "make it half the wall"; `AspectRatio` is a framed picture that keeps its shape; `ConstrainedBox(maxWidth)` is a reading lamp positioned so text lines aren't too long.

Problem Statement

Build a toolbar (icon + expanding title + action), a chip cloud that wraps, a 16:9 media box, a half-width panel, and body text capped to a readable max width. You'll use the fluid widgets — mostly without breakpoints.

Internal Working



- `Expanded` / `Flexible` / `Spacer`: distribute leftover main-axis space in a `Row` / `Column` by `flex` — the core of fluid horizontal/vertical layout (07 · layout_flex).
- `Wrap`: lays children in a run, moving to the next line when out of space (chips, tags, responsive button rows) — reflows automatically by width.
- `FractionallySizedBox(widthFactor/heightFactor)`: sizes a child to a fraction of the parent (e.g., 0.5 = half width) — fluid proportional sizing.
- `AspectRatio(aspectRatio)`: constrains the child to a width:height ratio (16/9 media, square tiles) — proportion-preserving across sizes.
- `ConstrainedBox(BoxConstraints(maxWidth: ...))` (often centered): caps content width for **readability** on wide screens (line length) — a key responsive move for text/forms.
- `LayoutBuilder` / `MediaQuery`: for the discrete structural switches (02_mediaquery_vs_layoutbuilder.md); these fluid widgets handle continuous adaptation.
- Also useful: `Flexible(fit: loose/tight)`, `IntrinsicWidth/Height` (sparingly — cost), `Table`, `Wrap.spacing/runSpacing`.

Memory Representation

Not applicable; standard layout render objects (07 · constraints_and_sizing).

Compiler Behavior

Not applicable. Use `const` where possible.

Runtime Behavior

`Wrap` reflows on width change; `Expanded` / `Flexible` recompute shares; `FractionallySizedBox` / `AspectRatio` recompute from the parent each layout. All respond to resize/rotate automatically.

Flutter Engine Behavior / Dart VM Behavior

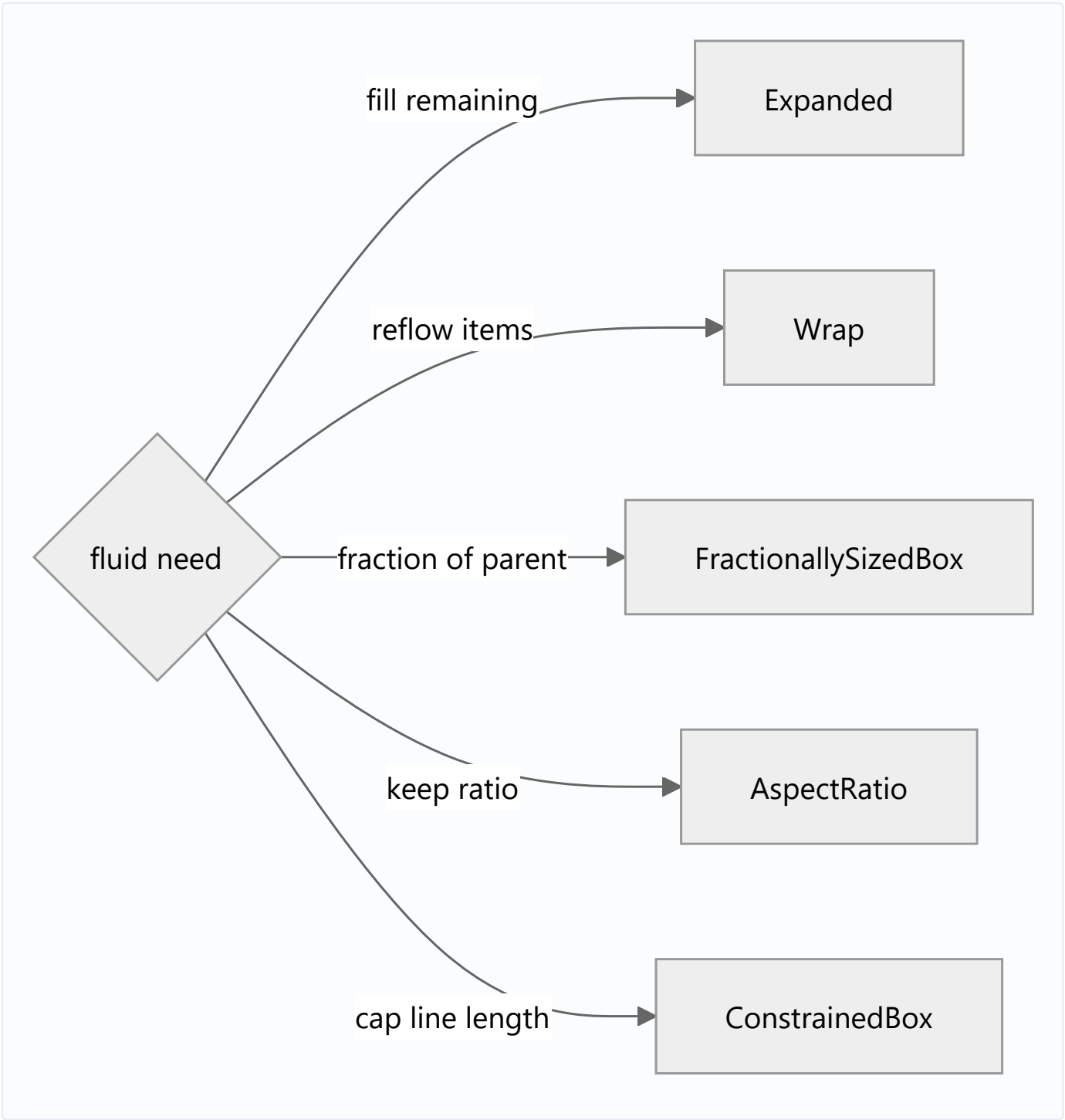
Not applicable.

Examples

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


class FluidDemo extends StatelessWidget {
  const FluidDemo({super.key});
  @override
```

```
  Widget build(BuildContext context) {
    return Center(
      child: ConstrainedBox(
        constraints: const BoxConstraints(maxWidth: 720), // cap width for readability
        child: Column(children: [
          // Toolbar: icon | expanding title | action (fluid)
          Row(children: [
            const Icon(Icons.menu),
            const SizedBox(width: 8),
            const Expanded(child: Text('Title', overflow: TextOverflow.ellipsis)),
            IconButton(icon: const Icon(Icons.search), onPressed: () {}),
          ]),
          const SizedBox(height: 12),

          // Chip cloud that wraps to new lines by available width
          Wrap(
            spacing: 8, runSpacing: 8,
            children: List.generate(12, (i) => Chip(label: Text('Tag $i'))),
          ),
          const SizedBox(height: 12),

          // 16:9 media box (proportion preserved at any width)
          AspectRatio(aspectRatio: 16 / 9, child: Container(color: Colors.teal)),
          const SizedBox(height: 12),

          // Half-width panel (fluid fraction of parent)
          FractionallySizedBox(
            widthFactor: 0.5, alignment: Alignment.centerLeft,
            child: Container(height: 40, color: Colors.orange),
          ),
        ]),
      ),
    );
  }
}
```

Diagrams


fluid need
fill remaining
Expanded
reflow items
Wrap
fraction of parent
FractionallySizedBox
keep ratio
AspectRatio
cap line length
ConstrainedBox

Common Mistakes

Mistake	Why	Fix
Fixed widths instead of flex/fractions	Breaks across sizes	`Expanded` / `FractionallySizedBox`
Full-width text on wide screens	Unreadable long lines	`ConstrainedBox(maxWidth)` + center
`Row` overflow with long content	Yellow/black stripes	`Expanded` / `Flexible` + ellipsis
Using breakpoints for continuous sizing	Rigid	Fluid widgets adapt continuously
`Wrap` vs `Row` confusion	Overflow vs reflow	`Wrap` when items should flow to new lines
Overusing `IntrinsicWidth/Height`	Extra layout passes (cost)	Avoid in big/scrolling trees (07)

Best Practices

- Use **`Expanded`** / **`Flexible`** / **`Spacer`** for fluid space distribution; **`Wrap`** for reflowing item groups.
- Size proportionally with **`FractionallySizedBox`** and keep media/tiles with **`AspectRatio`**.
- **Cap content width** (`ConstrainedBox(maxWidth)` + `Center`) for readability on large screens (text/forms/reading views).
- Prefer these **fluid** widgets for continuous adaptation; add **breakpoints** only for structural switches.
- Guard long text with `overflow: ellipsis` ; avoid fixed pixel widths and excessive intrinsics.

Performance

Cheap standard layout; `Wrap` /flex recompute on resize. Avoid `IntrinsicWidth/Height` in large/scrolling trees (extra passes) (07 · constraints_and_sizing, 21).

Advantages / Disadvantages

- **+** Fluid adaptation without math/breakpoints; intent-revealing; composable; resize/rotate-safe.
- **−** Requires understanding constraints; misuse causes overflow/unbounded errors; not a substitute for structural breakpoints.

Interview Questions

1. 🟢 **How do you make a row's item fill remaining space?** — Wrap it in `Expanded` (or `Flexible`) inside the `Row` .
2. 🟢 **What does `Wrap` do?** — Lays children in runs and flows to the next line when out of horizontal space — reflows by width.
3. 🟡 **How do you keep media at 16:9 across sizes?** — `AspectRatio(aspectRatio: 16/9)` .

4. 🟡 **How do you cap text width for readability on wide screens?** — `ConstrainedBox(constraints: BoxConstraints(maxWidth: N))`, usually centered.
5. 🟡 `FractionallySizedBox` **— what does it do?** — Sizes a child to a fraction of its parent (e.g., 0.5 width) — fluid proportional sizing.
6. 🔴 **When use these fluid widgets vs breakpoints?** — Fluid widgets for continuous adaptation within a layout; breakpoints for discrete structural changes between size tiers.
7. 🔴 **Why avoid** `IntrinsicWidth/Height` **in responsive lists?** — They add extra layout passes (cost); prefer flex/fractions/fixed sizing.

Senior Engineer Tips

- Reach for `Expanded` / `Wrap` / `FractionallySizedBox` / `AspectRatio` / `ConstrainedBox` before writing any width math — they express intent and adapt automatically.
- Always cap reading content width on wide screens; full-bleed text is a common desktop/web readability failure.
- Handle overflow (ellipsis/ `Flexible`) proactively; test at narrow and very wide widths.

Architect Perspective

These fluid widgets are the continuous-adaptation layer of responsive design; combined with breakpoint-driven structure (01_responsive_fundamentals.md) and correct scope (02_mediaquery_vs_layoutbuilder.md), they produce components that look right at any width — reusable across split views, grids, and platforms.

Summary

- Fluid toolkit: `Expanded` / `Flexible` / `Spacer` (share space), `Wrap` (reflow), `FractionallySizedBox` (fraction), `AspectRatio` (ratio), `ConstrainedBox(maxWidth)` (readability).
- Handles continuous adaptation without math/breakpoints; cap content width on wide screens; guard overflow.
- Combine with breakpoints for structural switches.

Revision Notes

- `Expanded` / `Flexible` / `Spacer` (fill/share), `Wrap` (reflow), `FractionallySizedBox` (fraction of parent), `AspectRatio` (ratio), `ConstrainedBox(maxWidth)` (line-length).
- Fluid = continuous; breakpoints = structural. Guard overflow (ellipsis/ `Flexible`).
- Avoid fixed widths + excessive intrinsics; `const` where possible.

Practice Questions

1. Which widget reflows chips onto new lines?
2. How do you cap text width on wide screens, and why?
3. Fluid widgets vs breakpoints — when each?

Coding Questions

1. Build a toolbar (icon | expanding ellipsized title | action).
2. Make a wrapping chip cloud + a 16:9 media box.
3. Center body text capped at `maxWidth: 700`.

Mini Project

Fluid article layout (Flutter): Build an article screen using `ConstrainedBox(maxWidth)` for readable body text, a `Wrap` tag cloud, an `AspectRatio` hero image, a `FractionallySizedBox` callout, and an `Expanded`-based toolbar — all fluid (no breakpoints). Test narrow→very-wide. Acceptance: readable capped width; reflowing tags; preserved image ratio; no overflow; fluid across widths; runs.

Adaptive Grids & Split Views (Master-Detail)

> On large screens, reflow content into **responsive grids** (`SliverGridDelegateWithMaxCrossAxisExtent`) and switch single-page navigation into **master-detail split views** (list + detail side-by-side) — the two highest-impact large-screen patterns.

Introduction

The signature tablet/desktop upgrades: grids that pick their column count by width, and master-detail layouts that show list + detail together instead of navigating between them. This file covers responsive grids and the split-view (master-detail) pattern, including how navigation adapts.

Why this concept exists

Wide screens waste space with phone layouts (single column, full-screen detail). Grids fill width with more columns; master-detail uses the extra width to show context (list) + content (detail) simultaneously — the expected large-screen experience (mail, settings, file browsers).

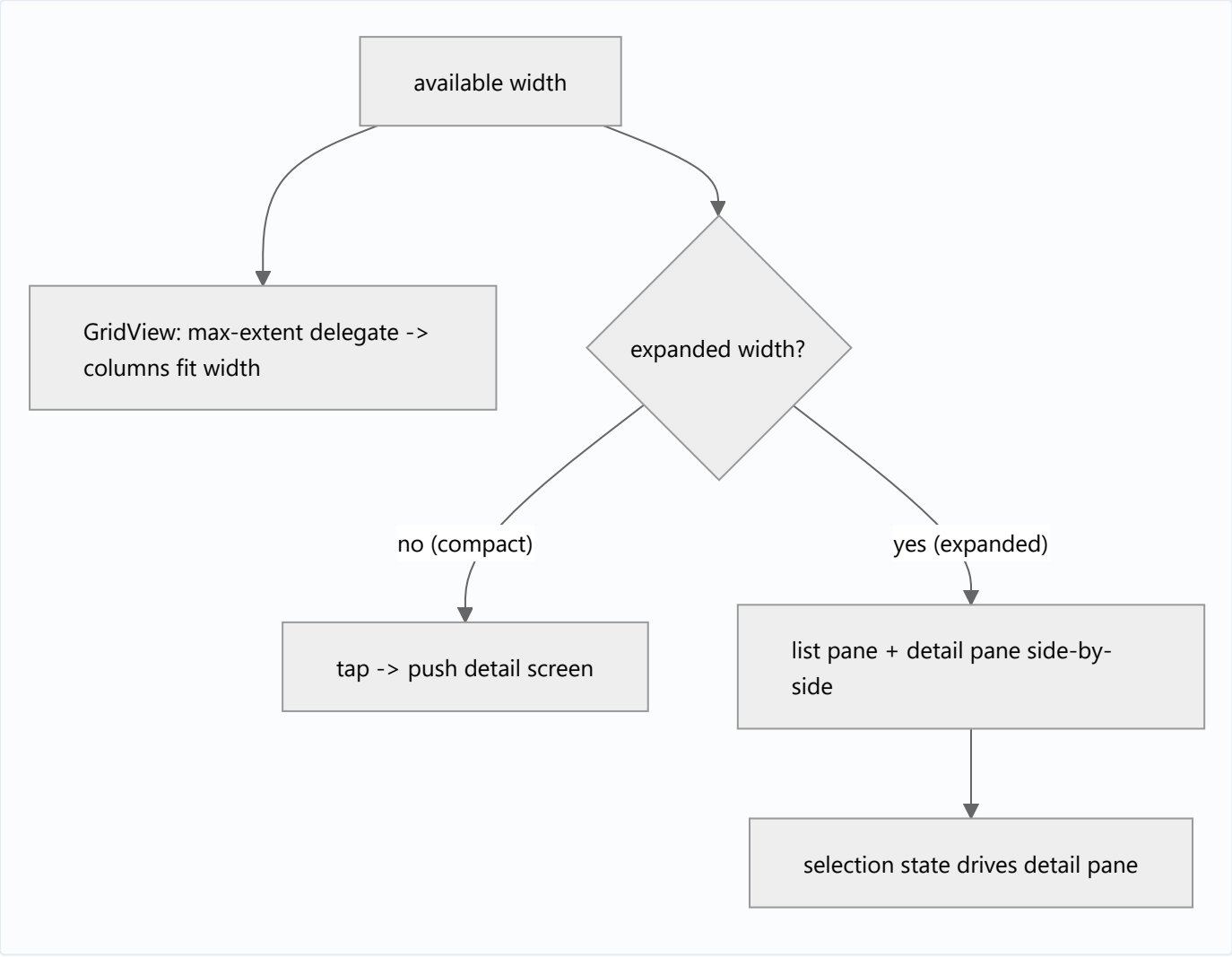
Real-world analogy

A **filing cabinet vs a desk**: on a phone you pull one folder at a time (navigate to detail); on a wide desk you lay the index (list) on the left and the open document (detail) on the right — see both at once (master-detail). A grid is arranging items to fill the desk's width in as many columns as fit.

Problem Statement

A products screen: a grid whose columns grow with width, and — on tablet/desktop — a master-detail where tapping a list item shows its detail in the right pane (instead of pushing a new screen on phones). You'll build a responsive grid and an adaptive split view.

Internal Working



- **Responsive grid**: prefer `GridView.builder` with `SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent:)` — it fits as many columns as the width allows (columns auto-adjust) — over a fixed `crossAxisCount` (which doesn't adapt). Set `childAspectRatio`/spacing.
- **Master-detail (split view)**:
  - **Compact**: a list; tapping **navigates** to a detail screen (12 · imperative_navigation).
  - **Expanded**: a `Row` with a **list pane** (fixed/flex width) + **detail pane** (`Expanded`), both visible; tapping updates a **selection** that the detail pane renders — no navigation.
  - Decide the split by width (`LayoutBuilder`/`MediaQuery` — 02_mediaquery_vs_layoutbuilder.md); keep **selection state** lifted so both layouts share it (11).
- **Navigation adapts**: bottom nav (compact) → **navigation rail** (expanded); list-push → split-select.
- Route-integrated version: `go_router` `StatefulShellRoute` for deep-linkable panes (13 · shell_routes_and_nested).

Memory Representation

Grid uses lazy building (only visible cells) — bounded (07 · scrolling_and_slivers). Split view keeps both panes alive; selection state is small/lifted.

Compiler Behavior / Runtime Behavior

Layout branches by width; grid recomputes columns on resize; the detail pane rebuilds on selection change (scope it).

Flutter Engine Behavior / Dart VM Behavior

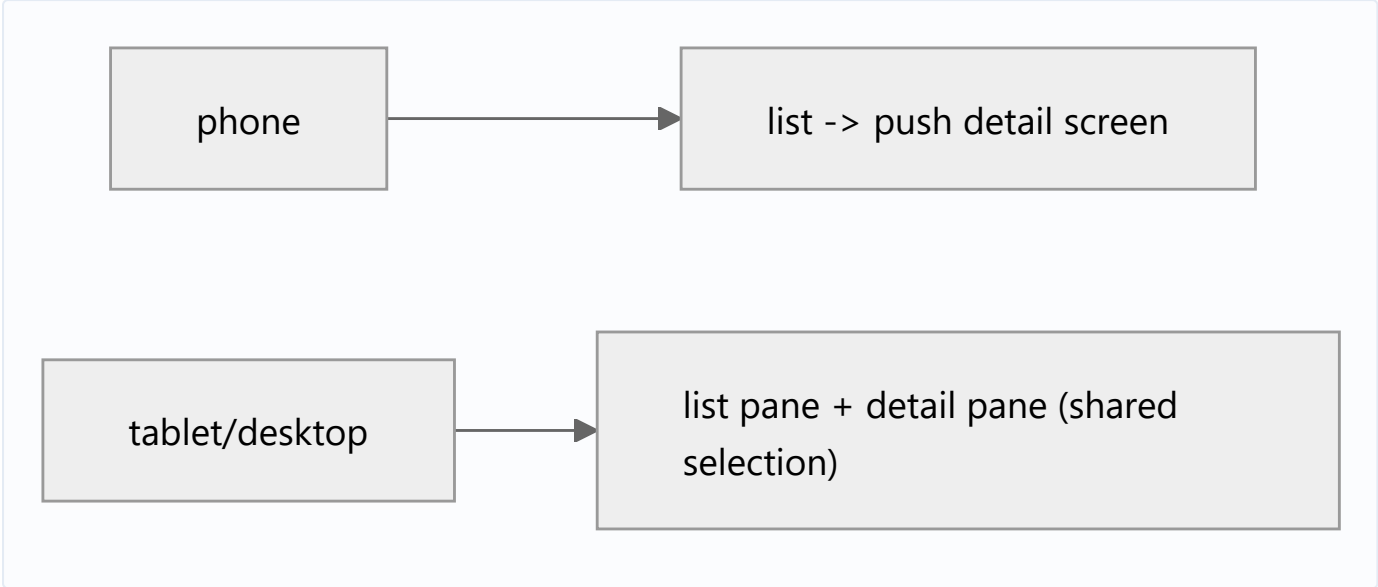
Not applicable.

Examples

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

// Responsive grid: columns fit the width via max cross-axis extent
class ProductGrid extends StatelessWidget {
  final List<String> items;
  const ProductGrid({super.key, required this.items});
  @override
  Widget build(BuildContext context) => GridView.builder(
        gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(
          maxCrossAxisExtent: 200, // as many ~200px columns as fit
          childAspectRatio: 3 / 4, mainAxisSpacing: 8, crossAxisSpacing: 8,
        ),
        itemCount: items.length,
        itemBuilder: (_, i) => Card(child: Center(child: Text(items[i]))),
      );
}

// Master-detail: split on wide screens, navigate on narrow
class Catalog extends StatefulWidget {
  const Catalog({super.key});
  @override State<Catalog> createState() => _CatalogState();
}
class _CatalogState extends State<Catalog> {
  final _items = List.generate(20, (i) => 'Item $i');
  String? _selected; // lifted selection shared by both layouts
```

```
  @override
  Widget build(BuildContext context) {
    return LayoutBuilder(builder: (context, c) {
      final expanded = c.maxWidth >= 900;
      final list = ListView(
        children: _items.map((it) => ListTile(
          title: Text(it),
          selected: it == _selected,
          onTap: () {
            setState(() => _selected = it);
            if (!expanded) {
              Navigator.of(context).push(MaterialPageRoute(
                builder: (_) => _Detail(it))); // compact: navigate
            }
          },
        )).toList(),
      );

      if (!expanded) return Scaffold(appBar: AppBar(title: const Text('Catalog')), body: list);

      // Expanded: list pane + detail pane side-by-side (no navigation)
      return Scaffold(
        body: Row(children: [
          SizedBox(width: 320, child: list),
          const VerticalDivider(width: 1),
          Expanded(child: _selected == null
              ? const Center(child: Text('Select an item'))
              : _Detail(_selected!)),
        ]),
      );
    });
  }
}
class _Detail extends StatelessWidget {
  final String item; const _Detail(this.item);
  @override Widget build(_) => Scaffold(appBar: AppBar(title: Text(item)), body: Center(child:
Text('Detail: $item')));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Fixed `crossAxisCount` grid	Doesn't adapt to width	`SliverGridDelegateWithMaxCrossAxisExtent`
Separate state for list vs split layouts	Selection lost on resize	Lift shared selection state
Eager grid children	Jank/memory	`GridView.builder` (lazy)
Split view on narrow screens	Cramped panes	Switch to navigate on compact
Detail pane rebuilding whole screen	Perf	Scope detail-pane rebuild to selection
No empty/placeholder detail state	Blank pane	Show "select an item" placeholder

Best Practices

- Use **max-cross-axis-extent grids** so column count adapts to width (over fixed counts); lazy `GridView.builder`.
- Build **master-detail**: navigate on compact, split panes on expanded; **lift the selection state** so both share it.
- Adapt **navigation** (bottom nav → rail) alongside the layout switch.
- Provide a **placeholder** for the empty detail pane; scope detail rebuilds to selection.
- For deep-linkable panes, use `go_router` `StatefulShellRoute` (13).

Performance

Lazy grids bound memory/CPU; split view keeps both panes alive (fine for two panes) — scope detail rebuilds. Resize recomputes columns/branch cheaply (21 · list_and_scroll_performance).

Advantages / Disadvantages

- **+** Uses large-screen space well (more columns, context+content), expected tablet/desktop UX, one codebase.
- **−** More layout branches + shared-state management; split view adds complexity vs single-page; must handle empty/selection states.

Interview Questions

1. 🟢 **How do you make a grid's columns adapt to width?** — Use `GridView.builder` with `SliverGridDelegateWithMaxCrossAxisExtent` (fits as many columns as the width allows) instead of a fixed `crossAxisCount`.
2. 🟢 **What is a master-detail (split view)?** — A layout showing a list (master) and the selected item's detail side-by-side on large screens, instead of navigating to a separate detail screen.
3. 🟡 **How does navigation differ between compact and expanded in master-detail?** — Compact: tapping pushes a detail screen; expanded: tapping updates a selection that the detail pane renders (no navigation).
4. 🟡 **Why lift the selection state?** — So the same selection drives both the navigate-based (compact) and split (expanded) layouts, surviving resize between them.
5. 🟡 **Why prefer max-extent over fixed column count?** — Column count then adapts to any width automatically; fixed counts don't respond to size.
6. 🔴 **How do you make split-view panes deep-linkable?** — Use `go_router`'s `StatefulShellRoute` (nested navigators + URLs) so each pane/route is addressable.
7. 🔴 **How do you keep the split view performant?** — Lazy grid, scoped detail-pane rebuilds on selection, and placeholders for empty state.

Senior Engineer Tips

- Standardize a reusable adaptive-scaffold (list↔split + nav bar↔rail) driven by a breakpoint helper; screens plug content in.
- Keep the master-detail selection in a shared store/state so resizing between phone/tablet doesn't lose it.
- Use max-extent grids by default; they "just work" across sizes without magic column numbers.

Architect Perspective

Responsive grids + master-detail are the defining large-screen patterns; combined with adaptive navigation and deep-linkable shells (13), they deliver a first-class tablet/desktop/web experience from one codebase. Centralizing the adaptive scaffold + shared selection state is the scalable way to apply them across an app (01_responsive_fundamentals.md).

Summary

- Responsive grids (max-cross-axis-extent, lazy) fill width with adaptive columns.
- Master-detail: navigate on compact, split panes on expanded, with lifted shared selection and adaptive navigation.
- Provide placeholders, scope detail rebuilds, and consider `StatefulShellRoute` for deep-linkable panes.

Revision Notes

- Grid: `GridView.builder` + `SliverGridDelegateWithMaxCrossAxisExtent` (adaptive columns), lazy.
- Master-detail: compact→push detail; expanded→list pane + `Expanded` detail; lift selection state.
- Adapt nav (bottom→rail); placeholder detail; scope detail rebuild.
- Deep-linkable panes → `go_router` `StatefulShellRoute` (Module 13).

Practice Questions

1. Why use max-extent over fixed `crossAxisCount`?
2. How does master-detail behave differently on compact vs expanded?
3. Why lift the selection state?

Coding Questions

1. Build a responsive grid with adaptive columns via max cross-axis extent.
2. Build a master-detail that navigates on narrow and splits on wide (shared selection).
3. Add adaptive navigation (bottom nav ↔ rail) to the split scaffold.

Mini Project

Adaptive catalog (Flutter): Build a catalog with a responsive product grid (adaptive columns) and a master-detail layout — navigate to detail on phones, split panes on tablet/desktop — with lifted selection state, adaptive navigation (bottom nav ↔ rail), and an empty-detail placeholder. Acceptance: adaptive grid columns; split on wide/navigate on narrow; shared selection survives resize; runs across sizes.

Responsive Typography, Spacing, Orientation & Safe Areas

> Responsiveness isn't just layout: respect the user's **text scale** (accessibility), scale **spacing** by size, handle **orientation** changes, and lay out within **safe areas** (`SafeArea`, insets) so content isn't clipped by notches, system bars, or the keyboard.

Introduction

The finishing details of responsive UI: honoring `textScaler` (never hardcode font sizes rigidly), scaling spacing/typography across tiers, reacting to orientation, and using `SafeArea` / `MediaQuery` insets so content avoids notches/bars/keyboard. This file rounds out the module.

Why this concept exists

Users set larger fonts (accessibility); devices have notches, rounded corners, gesture bars, and keyboards; screens rotate. Ignoring these clips text, overlaps system UI, or breaks on rotation — failing accessibility and quality bars. Handling them is essential responsiveness.

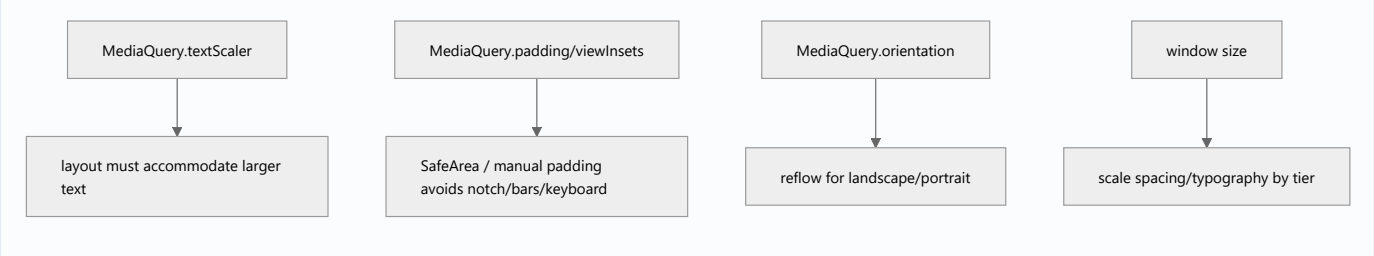
Real-world analogy

Safe areas are like **keeping important content inside the "title-safe" zone** of a TV broadcast (not behind the bezel/overscan). Text scaling is **honoring a reader's chosen font size** rather than forcing your own — the room must accommodate the reader, not the other way around.

Problem Statement

A form must remain readable when the user enables large text, not be clipped by a notch or the keyboard, scale its spacing up on tablets, and reflow sensibly in landscape. You'll use `textScaler`, `SafeArea`, insets, and orientation.

Internal Working



- **Text scaling / accessibility**: the OS/user font scale is in `MediaQuery.textScaler` (newer) / `textScaleFactor` (older). `Text` scales automatically; your **layout** must not assume fixed text heights — use flexible/wrapping layouts and avoid clipping. Don't disable scaling; **test at large scales**. Use theme text styles (07 · text_and_theming) and let them scale.
- **Responsive typography/spacing**: pick base sizes/spacing per size tier (larger on expanded), ideally from theme/tokens; scale gaps/paddings with a spacing helper. Cap line length for readability (03_responsive_layout_widgets.md).
- **Safe areas & insets**: wrap content in `SafeArea` to avoid notches/status/gesture bars; `MediaQuery.viewInsets.bottom` gives keyboard height (pad/scroll so fields aren't hidden); `viewPadding` / `padding` distinguish system insets. Use `resizeToAvoidBottomInset` and scrollable forms.
- **Orientation**: `MediaQuery.orientation` / `OrientationBuilder` to reflow (e.g., stack in portrait, row in landscape); preserve state across rotation (state lives above the widget).
- Combine with breakpoints (structure) and fluid widgets (continuous) from earlier files.

Memory Representation

Not applicable; these are layout/metric-driven decisions per frame (02_mediaquery_vs_layoutbuilder.md).

Compiler Behavior / Runtime Behavior

Text-scale/orientation/inset changes trigger rebuilds; layouts must adapt. Keyboard appearance changes `viewInsets.bottom` at runtime.

Flutter Engine Behavior

Insets/notches/keyboard come from the platform via the embedder into `MediaQuery` (10 · embedder_and_startup).

Dart VM Behavior

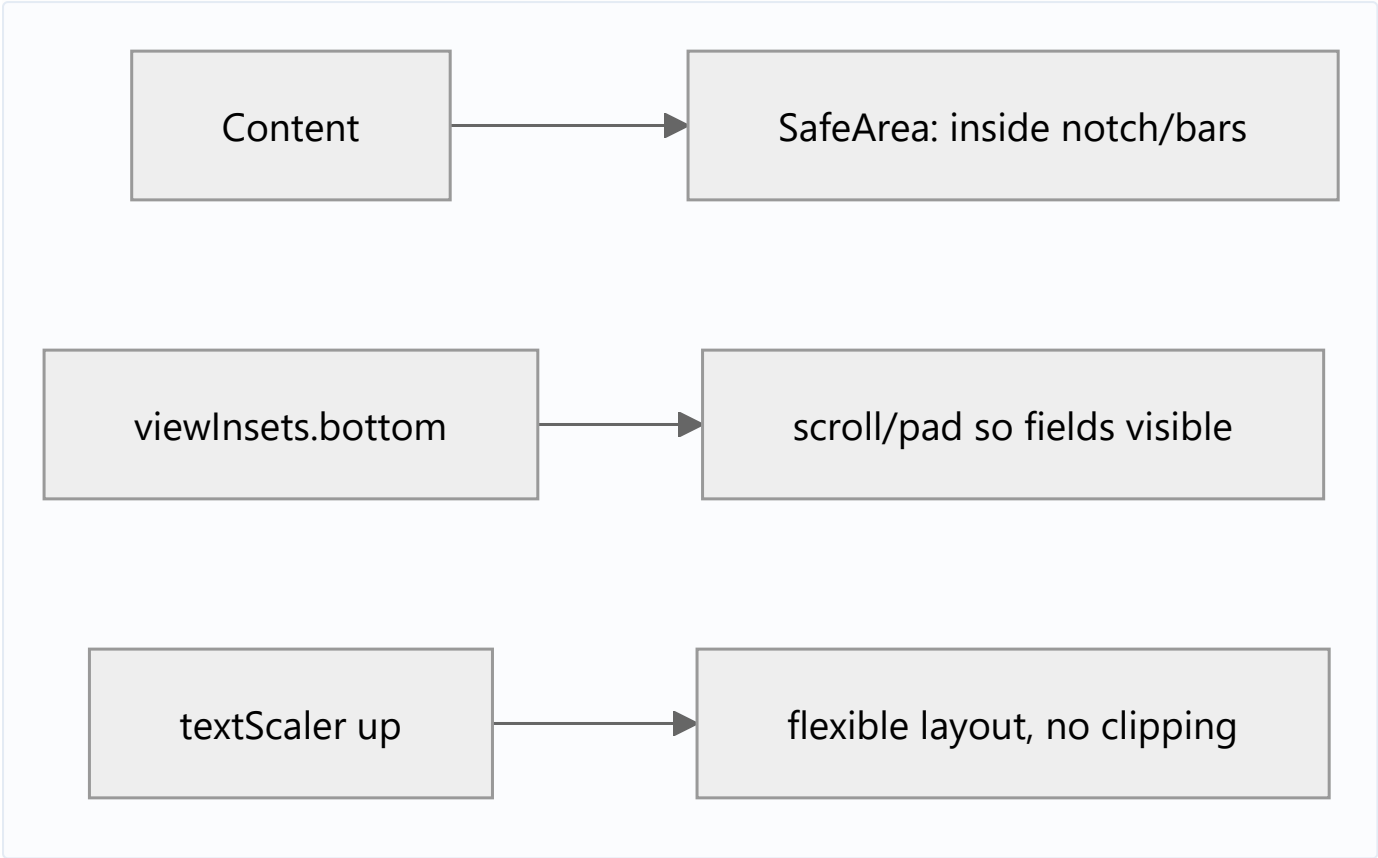
Not applicable.

Examples

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


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

```
  @override
  Widget build(BuildContext context) {
    final tier = MediaQuery.sizeOf(context).width >= 900 ? 24.0 : 16.0; // spacing by tier
    return Scaffold(
      resizeToAvoidBottomInset: true, // resize/scroll when keyboard shows
      body: SafeArea(                  // avoid notch/status/gesture bars
        child: Center(
          child: ConstrainedBox(
            constraints: const BoxConstraints(maxWidth: 520), // readable width
            child: SingleChildScrollView( // scroll so fields aren't hidden by keyboard
              padding: EdgeInsets.all(tier),
              child: Column(children: [
                // Uses theme text styles -> honors user text scale automatically
                Text('Sign up', style: Theme.of(context).textTheme.headlineMedium),
                SizedBox(height: tier),
                const TextField(decoration: InputDecoration(labelText: 'Email')),
                SizedBox(height: tier),
                const TextField(decoration: InputDecoration(labelText: 'Password')),
                SizedBox(height: tier),
                FilledButton(onPressed: () {}, child: const Text('Create account')),
              ]),
            ),
          ),
        ),
      ),
    );
  }
}

// Orientation-aware reflow
Widget orientationDemo() => OrientationBuilder(
      builder: (context, orientation) => orientation == Orientation.portrait
          ? const Column(children: [Placeholder(), Placeholder()])
          : const Row(children: [Expanded(child: Placeholder()), Expanded(child: Placeholder())]),
    );
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Disabling/ignoring text scale	Breaks accessibility	Honor `textScaler`; test at large scales
Fixed text heights/tight rows	Clip at large text	Flexible/wrapping layouts; scrollable
No `SafeArea`	Content under notch/bars	Wrap in `SafeArea`
Keyboard hides fields	Poor form UX	Scrollable + `resizeToAvoidBottomInset` / `viewInsets`
Same spacing on all sizes	Cramped/wasteful	Scale spacing by tier
State lost on rotation	Re-entry annoyance	Keep state above the widget

Best Practices

- **Honor the user's text scale**: use theme text styles, flexible layouts, and scrollable content; **test at large** `textScaler` — never hardcode-lock font sizes.
- Wrap screens in `SafeArea`; handle the **keyboard** (scrollable forms + `resizeToAvoidBottomInset` / `viewInsets.bottom`).
- **Scale spacing/typography by tier** (bigger on expanded) via tokens/helpers; **cap reading width**.

- Use `OrientationBuilder` / `MediaQuery.orientation` to reflow; keep state above the widget so rotation preserves it.
- Combine with breakpoints (structure) + fluid widgets (continuous).

Performance

Negligible; these are layout choices. Rebuilds on text-scale/orientation/keyboard changes are expected — scope them (21 · rebuild_optimization).

Advantages / Disadvantages

- **+** Accessible (text scale), device-safe (notch/keyboard), rotation-robust, size-appropriate spacing — full-quality responsiveness.
- **−** More cases to test (scales/orientations/devices); insets/keyboard handling nuances.

Interview Questions

1. 🟢 **How do you support user font scaling?** — Rely on `MediaQuery.textScaler` (text scales automatically); build flexible/scrollable layouts that don't clip at large scales — never lock font sizes.
2. 🟢 **What is `SafeArea` for?** — Insetting content to avoid notches, status/navigation bars, and gesture areas.
3. 🟡 **How do you keep fields visible when the keyboard appears?** — Use a scrollable form + `resizeToAvoidBottomInset` and/or account for `MediaQuery.viewInsets.bottom`.
4. 🟡 **How do you handle orientation changes?** — `OrientationBuilder` / `MediaQuery.orientation` to reflow; keep state above the widget so it survives rotation.
5. 🟡 **Why scale spacing/typography by size tier?** — Fixed spacing looks cramped on large screens and oversized on small; scale by tier (tokens/helpers) for balance.
6. 🔴 **Why is disabling text scaling an accessibility problem?** — It ignores the user's chosen font size (a11y setting); layouts must accommodate larger text instead of preventing it.
7. 🔴 **`padding` vs `viewInsets` vs `viewPadding` in `MediaQuery`?** — `padding` = system intrusions (notch/bars) currently affecting layout; `viewInsets` = obscured regions like the keyboard; `viewPadding` = intrusions ignoring current insets.

Senior Engineer Tips

- Test every screen at the **largest text scale** and with the **keyboard open** — these catch the most real-world responsive bugs.
- Centralize spacing as tokens scaled by tier; wrap screens in `SafeArea` by default in your scaffold.

- Keep form state (and scroll position) above the widget so rotation/keyboard don't reset it.

Architect Perspective

Typography scaling, spacing tokens, orientation, and safe-area handling are the accessibility-and-polish layer of responsiveness. Baking `SafeArea`, tier-scaled spacing, and text-scale-safe layouts into your base scaffold/design system ensures every screen is accessible and device-safe across form factors (07 · text_and_theming, Module 25).

Summary

- Honor user text scale (flexible/scrollable, test large), wrap in `SafeArea`, handle keyboard insets, scale spacing/typography by tier, and reflow on orientation.
- Keep state above widgets for rotation; cap reading width.
- The accessibility/polish layer completing responsive design.

Revision Notes

- Text scale: `MediaQuery.textScaler`; flexible/scrollable layouts; test large scales (don't lock sizes).
- `SafeArea` (notch/bars); keyboard via `resizeToAvoidBottomInset` / `viewInsets.bottom` + scroll.
- Spacing/typography scale by tier (tokens); cap reading width.
- Orientation via `OrientationBuilder` / `orientation`; keep state above widget. `padding` / `viewInsets` / `viewPadding` differ.

Practice Questions

1. Why must layouts accommodate large text scale?
2. How do you prevent the keyboard from hiding a field?
3. `padding` vs `viewInsets` in `MediaQuery`?

Coding Questions

1. Build a form that's safe-area-wrapped, keyboard-aware (scroll), and text-scale-safe.
2. Reflow a layout portrait↔landscape with `OrientationBuilder`.
3. Scale spacing by size tier via a helper/token.

Mini Project — Module capstone

Fully responsive screen (Flutter): Combine the module: breakpoint structure (list↔split), fluid widgets + adaptive grid, capped reading width, tier-scaled spacing/typography, `SafeArea` + keyboard-aware scrollable form, and orientation reflow — tested at large text scale, with keyboard open, and across sizes/orientations. Acceptance: adaptive structure + fluid content; accessible at large text; keyboard/notch-safe; orientation-robust; runs everywhere. (Builds on the adaptive product screen in README.)