25 · Adaptive UI

Flutter Practice Notes

Contents

25 · Adaptive UI

Introduction

Adaptive UI conforms to **platform conventions** — Material on Android, Cupertino on iOS, desktop idioms on Windows/macOS/Linux, web patterns on the browser — so the app feels native everywhere. This is the *platform* dimension, complementary to responsive design's *size* dimension (Module 24).

Why this module exists

Flutter renders one consistent UI by default, which can feel "non-native" (Material dialogs on iOS, wrong scroll physics/back gestures). Adaptive UI selectively conforms to each platform's expectations where it matters — a deliberate, bounded effort, not blanket duplication.

Module map

#	File	Topic	Level
1	01_adaptive_fundamentals.md	Adaptive vs responsive; platform detection; when to adapt	🔵
2	02_material_vs_cupertino.md	Design languages; `.adaptive` constructors; theming	🔵
3	03_platform_adaptive_widgets_and_navigation.md	Adaptive widgets, dialogs, navigation, scroll/back	🔴
4	04_cross_platform_design_system.md	Abstraction layer + adaptive components	🔴

> **Cross-references:** Responsive (size): Module 24. Widgets/theming: 07 · widget_categories, 07 · text_and_theming. Navigation/back: 12. Web/desktop: Modules 53/54. Platform detection: 10 · layered_architecture.

Prerequisites

07 Widgets, 24 Responsive UI, 12 Navigation.

What you'll be able to do after this module

- Distinguish adaptive from responsive and decide *when* to adapt.
- Detect platform correctly and use Material/Cupertino + `.adaptive` widgets.
- Adapt dialogs, navigation, scroll physics, and back behavior per platform.
- Build a cross-platform design system that centralizes adaptation.

Capstone

Platform-adaptive app: A screen that uses platform-correct scaffolding, dialogs, switches, scroll physics, and navigation (Material vs Cupertino vs desktop), driven by a small adaptive design-system layer — one codebase, native feel on each platform.

Summary

Adaptive UI = conform to platform conventions where it matters (design language, dialogs, navigation, scroll/back), via detection + `.adaptive` widgets + a design-system abstraction. Combine with responsive (size) for apps that feel right on every platform *and* form factor.

Adaptive Fundamentals (Adaptive vs Responsive; Platform Detection)

> **Adaptive** UI conforms to the *platform* (Material/Cupertino/desktop/web conventions); **responsive** UI adapts to *size*. Detect the platform correctly (`Theme.of(context).platform` / `defaultTargetPlatform`, not raw `Platform.isX`), and adapt **selectively** where conventions matter — not everywhere.

Introduction

Before adapting, get the concepts and detection right: the adaptive-vs-responsive distinction, how to detect platform properly, and the principle of *selective* adaptation (adapt what users notice; keep the rest consistent). This grounds the Material/Cupertino and design-system files.

Why this concept exists

Flutter's consistency is a feature, but some things *should* differ per platform (dialog style, switches, scroll physics, back navigation, date pickers) — mismatches feel wrong/non-native. Blanket per-platform UIs, though, double the work. The skill is deciding *when* and *how much* to adapt, with correct detection.

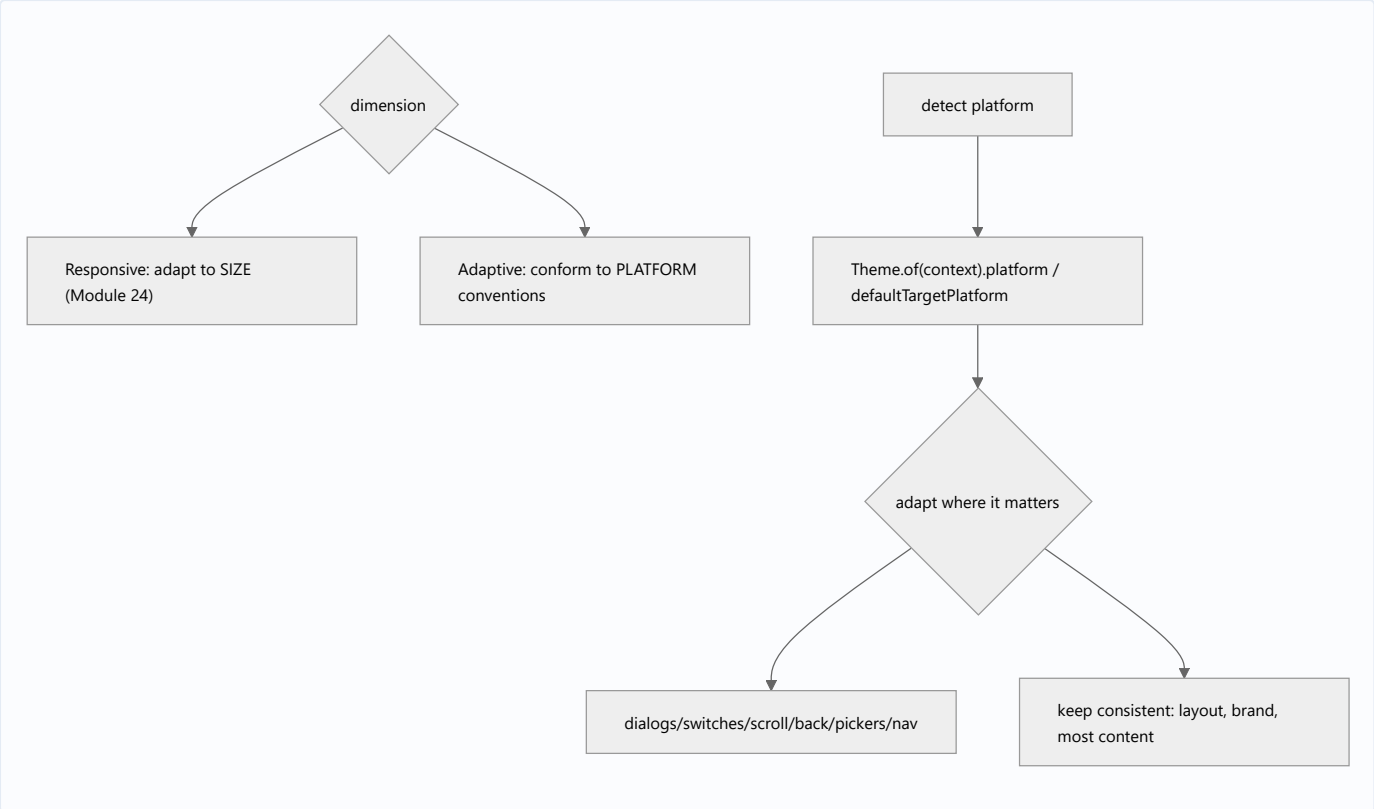
Real-world analogy

Adaptive is **speaking the local dialect and customs** wherever you travel (greetings, etiquette) while keeping your identity; responsive is **dressing for the weather** (size/conditions). You adapt the *manners that locals notice*, not everything — and you first need to correctly identify *which country you're in* (detection).

Problem Statement

Your app shows a Material dialog and switch on iOS (feels wrong) and uses Android scroll physics everywhere. Decide what to adapt, and detect the platform correctly (including web/desktop and testability). You'll set up proper detection + a selective-adaptation policy.

Internal Working



- **Adaptive vs responsive**: adaptive = *platform* (which design language/idioms); responsive = *size/orientation* (Module 24). They're orthogonal — real apps do both.
- **Platform detection (correctly)**:
  - Use `Theme.of(context).platform` or `defaultTargetPlatform` (a `TargetPlatform` enum: android/iOS/macOS/windows/linux/fuchsia). These are **overridable** (great for testing/previews) and work in widget code.
  - Avoid raw `dart:io` `Platform.isIOS` in UI: it's not overridable, throws on **web** (`dart:io` unavailable), and is harder to test. Use `kIsWeb` to detect web.
- **Selective adaptation**: adapt what users perceive as native (dialogs, switches/sliders, scroll physics, back gesture, date/time pickers, navigation patterns); keep **layout, branding, and most content consistent** to avoid doubling work.
- **Consistency vs nativeness tradeoff**: decide your product stance — some apps stay fully Material everywhere (brand consistency), others adapt heavily (native feel). Be intentional.

Memory Representation

Not applicable; platform is a metric read at build time.

Compiler Behavior

`defaultTargetPlatform` / `kIsWeb` are compile-time-friendly constants/enums; `dart:io` `Platform` isn't available on web (build/runtime issue).

Runtime Behavior

Detection returns the current (or overridden) platform; adaptive branches choose platform-appropriate widgets. Overriding `TargetPlatform` (in `MaterialApp` /tests) changes what adapts — enabling previews/tests.

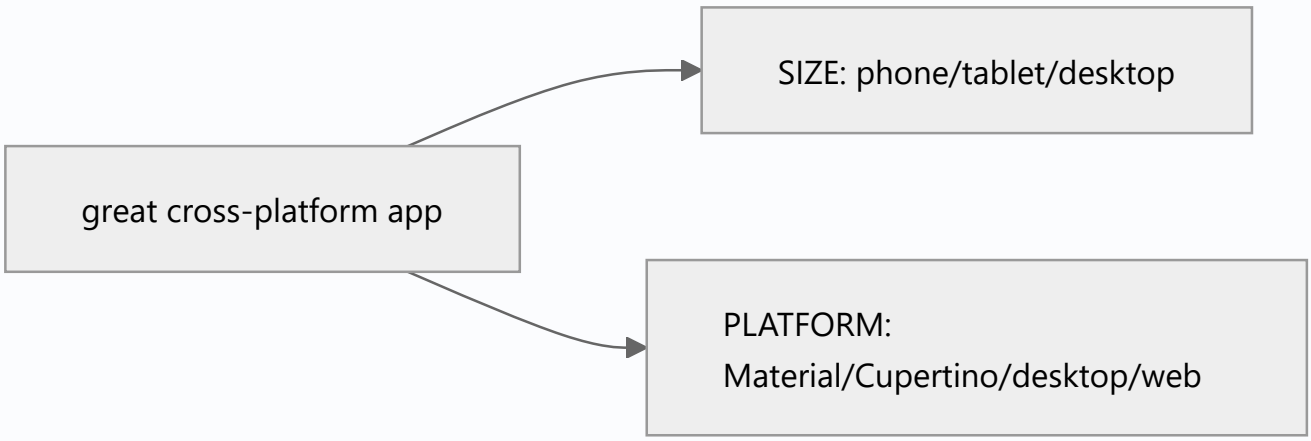
Flutter Engine Behavior / Dart VM Behavior

Not applicable.

Examples

```
import 'package:flutter/foundation.dart'; // defaultTargetPlatform, kIsWeb, TargetPlatform
import 'package:flutter/material.dart';

// ✅ Correct, testable, web-safe platform detection:
bool isApplePlatform(BuildContext context) {
  final p = Theme.of(context).platform; // overridable + works in widgets
  return p == TargetPlatform.iOS || p == TargetPlatform.macOS;
}

bool get isWeb => kIsWeb; // never use dart:io Platform on web

// ❌ Anti-pattern (avoid in UI):
// import 'dart:io';
// if (Platform.isIOS) ...   // not overridable, throws on web, hard to test

// Selective adaptation policy (what to adapt vs keep consistent):
// ADAPT:  dialogs, switches/sliders, scroll physics, back gesture, date/time pickers, nav pattern
// KEEP:   overall layout, brand colors/typography, business content
class AdaptiveExample extends StatelessWidget {
  const AdaptiveExample({super.key});
  @override
  Widget build(BuildContext context) {
    final apple = isApplePlatform(context);
    return Text(apple ? 'Cupertino-style here' : 'Material-style here');
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`dart:io` `Platform.isIOS` in UI	Crashes on web; not overridable/testable	`Theme.of(context).platform` / `defaultTargetPlatform` + `kIsWeb`
Confusing adaptive with responsive	Different dimensions	Adaptive=platform, responsive=size
Adapting everything	Doubles work/maintenance	Adapt selectively (what users notice)
Ignoring web/desktop platforms	Wrong idioms there	Handle web (`kIsWeb`) + desktop targets
Hardcoding platform checks everywhere	Scattered, fragile	Centralize in a design-system/helper

Best Practices

- Detect platform with `Theme.of(context).platform` / `defaultTargetPlatform` (overridable, testable) + `kIsWeb`; avoid `dart:io` `Platform` in UI.
- **Adapt selectively**: dialogs, switches/sliders, scroll physics, back gesture, date/time pickers, navigation — keep layout/brand/content consistent.
- Decide a **product stance** (fully-consistent vs heavily-adaptive) and apply it intentionally.
- **Centralize** adaptation in helpers/a design system (04_cross_platform_design_system.md); make platform overridable for testing/previews.
- Combine with **responsive** (size) for full cross-platform/form-factor coverage.

Performance

Negligible; detection is a constant/enum read and branching is cheap.

Advantages / Disadvantages

- + Native feel where it matters, better UX per platform, correct/testable detection, bounded effort.
- − Extra branches/testing across platforms; risk of over-adapting; must maintain a consistency policy.

Interview Questions

1. 🟢 **Adaptive vs responsive?** — Adaptive conforms to *platform* conventions (Material/Cupertino/desktop/web); responsive adapts to *size/orientation*. They're orthogonal.
2. 🟢 **How should you detect platform in Flutter UI?** — `Theme.of(context).platform` or `defaultTargetPlatform` (overridable, testable) + `kIsWeb` for web.
3. 🟡 **Why avoid `dart:io` `Platform.isIOS` in widgets?** — It's unavailable on web (throws), not overridable, and harder to test than `defaultTargetPlatform`.
4. 🟡 **Why adapt selectively rather than everywhere?** — Full per-platform UIs double effort; adapt only what users perceive as native (dialogs/switches/scroll/back/pickers/nav), keep the rest consistent.
5. 🟡 **What things typically warrant adaptation?** — Dialogs, switches/sliders, scroll physics, back gesture, date/time pickers, and navigation patterns.
6. 🔴 **Why is overridable platform detection valuable?** — It lets you preview/test each platform's UI (set `TargetPlatform` in `MaterialApp`/tests) without running on that device.
7. 🔴 **How do adaptive and responsive combine?** — Adaptive picks platform idioms; responsive picks layout by size — together they deliver native feel on every platform *and* form factor.

Senior Engineer Tips

- Standardize on `defaultTargetPlatform` / `Theme.of(context).platform` + `kIsWeb`; ban `dart:io` `Platform` in UI code (lint/review).
- Write down a **consistency-vs-nativeness policy** per product; it prevents ad-hoc, inconsistent adaptation.
- Centralize platform branching so screens stay clean and adaptation is testable/overridable.

Architect Perspective

Adaptive strategy is a product/UX-architecture decision: how native to feel vs how consistent to stay, and where to draw that line. Correct, centralized, overridable detection + selective adaptation (in a design-system layer) keeps a single codebase feeling native across platforms while remaining maintainable — complementing responsive design for full coverage (Module 24, 04_cross_platform_design_system.md).

Summary

- Adaptive = platform conventions; responsive = size — orthogonal, do both.
- Detect via `Theme.of(context).platform` / `defaultTargetPlatform` + `kIsWeb` (not `dart:io`).
- Adapt selectively (dialogs/switches/scroll/back/pickers/nav), keep layout/brand consistent, centralize the logic.

Revision Notes

- Adaptive (platform) vs responsive (size); orthogonal.
- Detect: `Theme.of(context).platform` / `defaultTargetPlatform` (overridable/testable) + `kIsWeb`; avoid `dart:io Platform` in UI (web-unsafe).
- Adapt selectively: dialogs/switches/scroll/back/pickers/nav; keep layout/brand consistent.
- Centralize + define product stance; combine with responsive.

Practice Questions

1. Why is `defaultTargetPlatform` better than `Platform.isIOS` in UI?
2. What's the difference between adaptive and responsive?
3. Which UI elements typically warrant platform adaptation?

Coding Questions

1. Write a web-safe, testable `isApplePlatform(context)` helper.
2. List, for a given app, what to adapt vs keep consistent (with rationale).
3. Override `TargetPlatform` in a test to preview iOS UI.

Mini Project

Adaptation policy + detection (docs + helper): Write `ADAPTATION.md` defining your product's adapt-vs-consistent policy (which elements adapt and why), and implement a centralized, web-safe, overridable platform-detection helper (`platform`, `isApple`, `isWeb`) used by a demo widget.

Acceptance: correct detection (no `dart:io` in UI); documented selective policy; overridable for tests; runs on mobile + web.

Material vs Cupertino (and `.adaptive` Constructors)

> Flutter ships two design languages — **Material** (`package:flutter/material.dart` , Android/web default) and **Cupertino** (`package:flutter/cupertino.dart` , iOS-style) — plus `.adaptive` **constructors** (`Switch.adaptive` , `Slider.adaptive` , `showAdaptiveDialog` , etc.) that render the right style per platform automatically.

Introduction

This file covers the two widget kits, their theming, when to use each, and the built-in `.adaptive` helpers that pick Material/Cupertino for you — the lowest-effort way to adapt common components without hand-branching.

Why this concept exists

iOS users expect Cupertino visuals/behaviors (switches, dialogs, nav bars, pickers); Android/web expect Material. Rather than manually branch every component, `.adaptive` constructors and `MaterialApp` 's built-in adaptations give platform-correct rendering with minimal code, while full Cupertino widgets exist when you want a deeply iOS-native experience.

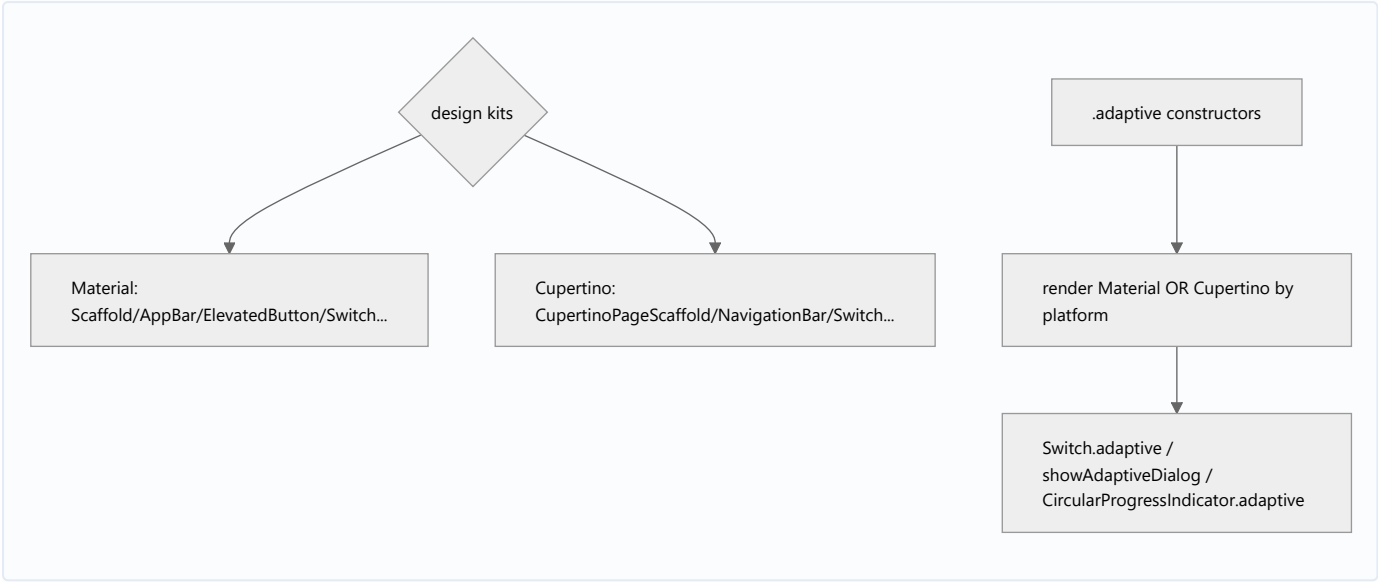
Real-world analogy

Material and Cupertino are **two style guides** (British vs American English). `.adaptive` constructors are a **smart editor** that auto-applies the right dialect based on your audience — you write once, it localizes the style.

Problem Statement

Show a platform-correct switch, dialog, and loading indicator without hand-branching, and know when to reach for full Cupertino widgets/theming for an iOS-native screen. You'll use `.adaptive` constructors and Cupertino equivalents.

Internal Working



- **Material** (`material.dart`): `MaterialApp` , `Scaffold` , `AppBar` , `ElevatedButton` , `Switch` , `TextField` , Material 3 theming (`ThemeData` , `colorScheme` — 07 · text_and_theming). Default for Android/web.
- **Cupertino** (`cupertino.dart`): `CupertinoApp` , `CupertinoPageScaffold` , `CupertinoNavigationBar` , `CupertinoButton` , `CupertinoSwitch` , `CupertinoTextField` , `CupertinoTheme` — iOS look/behavior.
- `.adaptive` **constructors** (in Material): render the platform-appropriate variant automatically:
  - `Switch.adaptive` , `Slider.adaptive` , `Checkbox.adaptive` , `Radio.adaptive`
  - `CircularProgressIndicator.adaptive` (spinner style per platform)
  - `showAdaptiveDialog` / `AlertDialog.adaptive` (Material vs Cupertino alert)
  - `Icon` /theme adaptations; `MaterialApp` also applies some platform adaptations (e.g., page transitions, scroll physics).
- **Choosing**: use **Material as the base** + `.adaptive` for common components (least effort, decent nativeness); use **full Cupertino** widgets/ `CupertinoApp` when you want a deeply iOS-native app or specific iOS components (pickers, action sheets, segmented controls).
- **Theming both**: Material via `ThemeData` ; Cupertino via `CupertinoTheme` / `MaterialApp` 's cupertino overrides; keep brand colors/typography consistent across both.

Memory Representation

Not applicable; both are widget trees. `.adaptive` picks a variant at build using platform.

Compiler Behavior

Not applicable.

Runtime Behavior

`.adaptive` reads the platform (`Theme.of(context).platform`) and builds the matching widget; overriding `TargetPlatform` changes the rendered style (previews/tests).

Flutter Engine Behavior / Dart VM Behavior

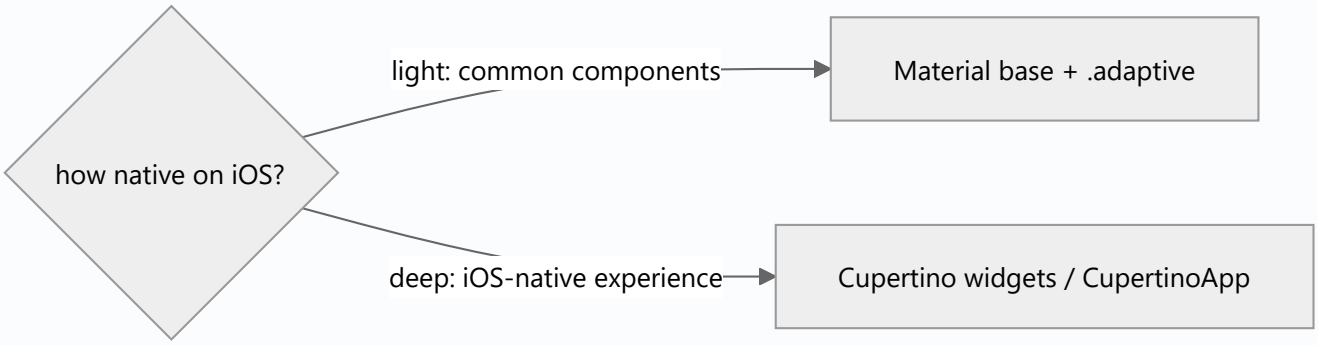
Not applicable.

Examples

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

class AdaptiveComponents extends StatefulWidget {
  const AdaptiveComponents({super.key});
  @override State<AdaptiveComponents> createState() => _S();
}
class _S extends State<AdaptiveComponents> {
  bool _on = true;
  @override
  Widget build(BuildContext context) {
    return Column(children: [
      // .adaptive: Material switch on Android, Cupertino switch on iOS — no branching
      Switch.adaptive(value: _on, onChanged: (v) => setState(() => _on = v)),

      // Adaptive spinner (Material circular vs Cupertino activity indicator)
      const CircularProgressIndicator.adaptive(),

      // Adaptive dialog (Material AlertDialog vs Cupertino alert)
      ElevatedButton(
        onPressed: () => showAdaptiveDialog(
          context: context,
          builder: (_) => AlertDialog.adaptive(
            title: const Text('Delete?'),
            content: const Text('This cannot be undone.'),
            actions: [
              TextButton(onPressed: () => Navigator.pop(context), child: const Text('Cancel')),
              TextButton(onPressed: () => Navigator.pop(context), child: const Text('Delete')),
            ],
```

```
          ),
        ),
        child: const Text('Show adaptive dialog'),
      ),
    ]);
  }
}

// Full Cupertino when you want a deeply iOS-native screen:
Widget cupertinoScreen() => const CupertinoPageScaffold(
      navigationBar: CupertinoNavigationBar(middle: Text('iOS Screen')),
      child: Center(child: CupertinoActivityIndicator()),
    );
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Material dialogs/switches on iOS	Feels non-native	`.adaptive` constructors / Cupertino
Hand-branching every component	Verbose/error-prone	Prefer `.adaptive`; centralize the rest
Full Cupertino for one small element	Over-work	Use `.adaptive` for common bits
Inconsistent brand across kits	Disjointed look	Share colors/typography in both themes
Cupertino-only widgets on Android	Wrong idiom there	Use adaptive/Material fallback
Forgetting web/desktop	No `.adaptive` iOS on web	Decide web/desktop styling explicitly

Best Practices

- Use **Material as the base** + `.adaptive` constructors for common components (switches/sliders/spinner/dialogs) — best effort-to-nativeness ratio.
- Reach for **full Cupertino** widgets/ `CupertinoApp` only when you want a **deeply iOS-native** screen or iOS-specific components (pickers/action sheets/segmented controls).
- **Theme both** kits with shared brand tokens (colors/typography) for consistency.
- **Centralize** platform choices in a design-system layer (04_cross_platform_design_system.md); keep detection overridable.
- Explicitly decide **web/desktop** styling (usually Material).

Performance

No meaningful cost; `.adaptive` is a build-time branch. Both kits are optimized framework widgets.

Advantages / Disadvantages

- **+** `.adaptive` **:** minimal-code platform correctness. **+ Cupertino:** deep iOS nativeness.
- **–** `.adaptive` **:** limited to widgets that provide it. **– Full Cupertino:** more code/duplication; must theme both; wrong on non-iOS if misused.

Interview Questions

1. 🟢 **What are Material and Cupertino?** — Flutter's two design-language widget kits: Material (Android/web) and Cupertino (iOS-style).
2. 🟢 **What do** `.adaptive` **constructors do?** — Render the platform-appropriate variant automatically (e.g., `Switch.adaptive`, `showAdaptiveDialog`, `CircularProgressIndicator.adaptive`).
3. 🟡 **When use full Cupertino vs** `.adaptive` **?** — `.adaptive` for common components with minimal effort; full Cupertino for deeply iOS-native experiences or iOS-specific widgets (pickers/action sheets).
4. 🟡 **How do you keep branding consistent across kits?** — Share color/typography tokens in both `ThemeData` and `CupertinoTheme`.
5. 🟡 **How does** `.adaptive` **decide which to render?** — By the current platform (`Theme.of(context).platform`), which is overridable for tests/previews.
6. 🔴 **What about web/desktop with** `.adaptive` **?** — `.adaptive` mainly toggles iOS(Cupertino) vs others(Material); decide web/desktop styling explicitly (usually Material) — don't assume Cupertino there.
7. 🔴 **What's the tradeoff of full per-platform Cupertino/Material screens?** — Native feel vs duplicated code/maintenance and dual theming — adapt selectively.

Senior Engineer Tips

- Default to **Material +** `.adaptive`; it covers most nativeness needs cheaply. Escalate to Cupertino widgets only where iOS users clearly expect them.
- Keep a single source of brand tokens applied to both themes so Material and Cupertino look like *your* app, not stock kits.
- Wrap platform choices behind design-system widgets so screens don't branch inline.

Architect Perspective

Material/Cupertino + `.adaptive` are the building blocks of a platform-adaptive UI. A design-system layer that defaults to Material, applies `.adaptive` for common components, and uses Cupertino selectively — all themed with shared brand tokens — delivers native feel with controlled effort, and is where you centralize the adapt-vs-consistent policy (01_adaptive_fundamentals.md, 04_cross_platform_design_system.md).

Summary

- Material (Android/web) and Cupertino (iOS) are the two kits; `.adaptive` constructors auto-pick per platform.
- Base on Material + `.adaptive`; use full Cupertino for deeply iOS-native needs; theme both with shared brand tokens.
- Decide web/desktop styling explicitly; centralize choices in a design system.

Revision Notes

- Material (`material.dart`) vs Cupertino (`cupertino.dart`); `.adaptive` : `Switch/Slider/Checkbox.adaptive` , `CircularProgressIndicator.adaptive` , `showAdaptiveDialog` / `AlertDialog.adaptive` .
- Base Material + `.adaptive` (low effort); full Cupertino for deep iOS.
- Theme both with shared brand tokens; `.adaptive` = iOS vs others (decide web/desktop explicitly).
- Centralize in design system; detection overridable.

Practice Questions

1. When use `.adaptive` vs full Cupertino widgets?
2. How do you keep both kits on-brand?
3. What does `.adaptive` do on web/desktop?

Coding Questions

1. Build a form with `Switch.adaptive` + adaptive spinner + `showAdaptiveDialog`.
2. Create a Cupertino-native screen (`CupertinoPageScaffold` / `CupertinoNavigationBar`).
3. Share brand colors/typography across `ThemeData` and `CupertinoTheme`.

Mini Project

Adaptive component set (Flutter): Build a settings screen using Material as the base with `.adaptive` switches/dialog/spinner (platform-correct), plus one full-Cupertino detail screen, theming both kits with shared brand tokens. Preview iOS via `TargetPlatform` override. Acceptance: platform-correct common components; deep-Cupertino where chosen; consistent branding; web/desktop styling decided; runs.

Platform-Adaptive Widgets & Navigation

> Beyond dialogs/switches, adapt the **interaction patterns** users feel: scroll physics (bouncing vs clamping), the **back gesture/behavior**, navigation structure (bottom tabs vs nav rail/menu bar), scrollbars, context menus, and date/time pickers — so each platform behaves natively.

Introduction

This file covers platform-adaptive *behaviors and navigation*, not just visuals: scroll physics, back handling, navigation patterns per platform/form factor, scrollbars, selection/context menus, and pickers. These are the behavioral details that make an app feel native (or not).

Why this concept exists

Users notice *behavior* even more than looks: iOS bouncing scroll, the iOS edge-swipe back, desktop right-click menus and always-visible scrollbars, Android's system back. Getting these right (mostly automatic, sometimes manual) is what separates "a Flutter app" from "a native-feeling app."

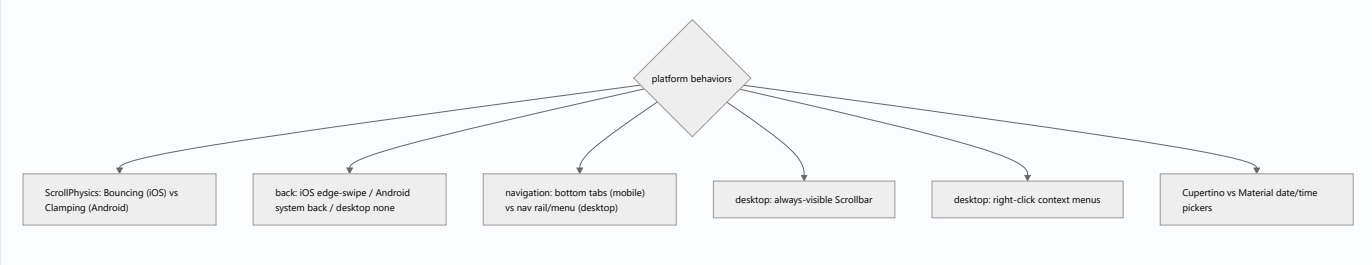
Real-world analogy

Adapting behaviors is like **driving on the correct side of the road** in each country — not just repainting the car. Visitors immediately feel when the controls behave "wrong"; matching local driving conventions (behaviors) matters more than the paint job.

Problem Statement

Your app scrolls with Android physics on iOS, ignores the iOS back-swipe, shows bottom tabs on desktop (should be a rail/menu), lacks desktop scrollbars, and uses a Material date picker on iOS. You'll adapt scroll physics, back, navigation, scrollbars, and pickers.

Internal Working



- **Scroll physics**: `MaterialApp` applies platform-appropriate `ScrollPhysics` by default (bouncing on iOS, clamping on Android); override via `ScrollConfiguration` / `physics:` when needed. It's a Strategy (05 · strategy).
- **Back behavior**: iOS gets the **edge-swipe back** (via `CupertinoPageRoute`/adaptive page transitions); Android uses the **system back** button; desktop/web use app-provided back. Use `PopScope` for confirm-exit (12 · navigator_stack).
- **Navigation patterns** (platform + size): mobile → **bottom navigation/tabs**; large/desktop → **navigation rail / side menu / menu bar**; iOS → `CupertinoTabScaffold`; combine with responsive tiers (24 · responsive_fundamentals).
- **Scrollbars**: desktop/web want **always-visible, draggable** scrollbars (`Scrollbar(thumbVisibility: true)`); mobile hides them.
- **Selection/context menus**: desktop/web want **right-click context menus** and hover states; mobile uses long-press/toolbar.
- **Pickers/inputs**: iOS wheel date/time pickers vs Material calendar; adapt where users expect the native picker.
- Much is **automatic** via `MaterialApp` + `.adaptive`; the rest is **selective manual** adaptation.

Memory Representation

Not applicable; behaviors are configured widgets/physics chosen by platform/size.

Compiler Behavior / Runtime Behavior

Physics/nav/back chosen at build by platform (overridable); scrollbars/menus respond to input device (mouse vs touch).

Flutter Engine Behavior

Input devices (mouse/touch/keyboard), back gestures, and system bars are surfaced by the embedder; the framework maps them to platform behaviors (10 · embedder_and_startup).

Dart VM Behavior

Not applicable.

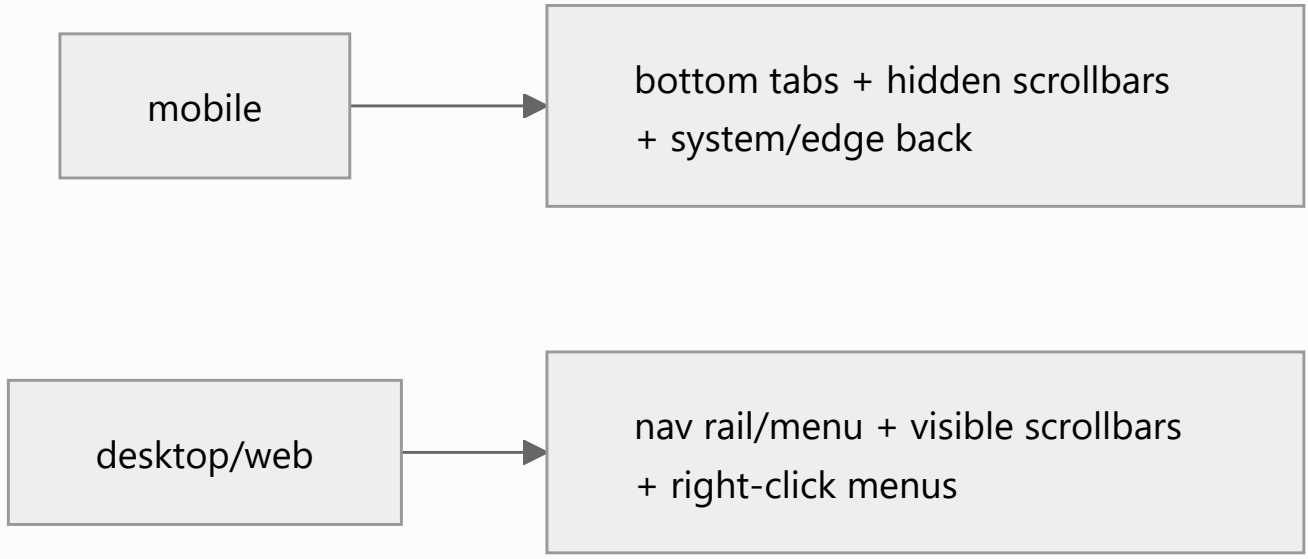
Examples

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


// Navigation adapts by size/platform (bottom nav on mobile, rail on desktop)
```

```
class AdaptiveNavScaffold extends StatelessWidget {
  final int index; final Widget body; final ValueChanged<int> onSelect;
  const AdaptiveNavScaffold({super.key, required this.index, required this.body, required this.onSelect});

  @override
  Widget build(BuildContext context) {
    final wide = MediaQuery.sizeOf(context).width >= 900;
    final destinations = const [
      (icon: Icons.home, label: 'Home'),
      (icon: Icons.search, label: 'Search'),
      (icon: Icons.person, label: 'Profile'),
    ];
    if (wide) {
      // Desktop/tablet: navigation rail + always-visible scrollbar on content
      return Scaffold(
        body: Row(children: [
          NavigationRail(
            selectedIndex: index, onDestinationSelected: onSelect,
            destinations: [for (final d in destinations)
              NavigationRailDestination(icon: Icon(d.icon), label: Text(d.label))],
          ),
          const VerticalDivider(width: 1),
          Expanded(child: Scrollbar(thumbVisibility: true, child: body)), // desktop scrollbar
        ]),
      );
    }
    // Mobile: bottom navigation
    return Scaffold(
      body: body,
      bottomNavigationBar: NavigationBar(
        selectedIndex: index, onDestinationSelected: onSelect,
        destinations: [for (final d in destinations)
          NavigationDestination(icon: Icon(d.icon), label: d.label)],
      ),
    );
  }
}

// Scroll physics: MaterialApp adapts by default; override explicitly if needed
```

```
// ScrollConfiguration(behavior: const MaterialScrollBehavior(), child: ...)
// or set physics: const BouncingScrollPhysics() / ClampingScrollPhysics() deliberately.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Android scroll physics on iOS	Non-native feel	Let `MaterialApp` adapt / `BouncingScrollPhysics` on iOS
No edge-swipe back on iOS	Breaks iOS expectation	Cupertino/adaptive page routes
Bottom tabs on desktop	Wrong idiom	Navigation rail/menu on wide/desktop
No visible scrollbars on desktop/web	Hard to scroll with mouse	`Scrollbar(thumbVisibility: true)`
No right-click/hover on desktop	Misses desktop UX	Context menus + hover states
Material date picker on iOS	Non-native	Cupertino picker where expected

Best Practices

- Rely on `MaterialApp` + `.adaptive` for automatic behaviors (scroll physics, page transitions), then **selectively** adapt navigation, scrollbars, context menus, and pickers.
- **Adapt navigation by platform + size**: bottom tabs (mobile) → rail/side menu/menu bar (desktop) — combine with responsive tiers.

- Provide **desktop/web affordances**: visible scrollbars, hover states, right-click context menus, keyboard shortcuts.
- Honor **back conventions** (iOS edge-swipe, Android system back) via adaptive routes; `PopScope` for confirm-exit.
- Use **native pickers/inputs** where users expect them; centralize behavior choices in a design system.

Performance

Negligible; these are configuration/branching choices. Scroll physics/scrollbars are framework-optimized.

Advantages / Disadvantages

- + Native-feeling behavior per platform/device; better UX; meets user expectations.
- − More cases (mobile/desktop/web, touch/mouse) to handle/test; risk of inconsistency without centralization.

Interview Questions

1. 🟢 **What behaviors should you adapt per platform?** — Scroll physics, back gesture/behavior, navigation pattern, scrollbars, context menus/hover, and date/time pickers.
2. 🟢 **How does scroll physics adapt by default?** — `MaterialApp` applies platform-appropriate `ScrollPhysics` (bouncing on iOS, clamping on Android); override via `ScrollConfiguration` / `physics:`.
3. 🟡 **How should navigation change across platform/size?** — Bottom tabs on mobile → navigation rail/side menu/menu bar on desktop/wide; iOS `CupertinoTabScaffold` for deep iOS.
4. 🟡 **What desktop/web affordances are expected?** — Always-visible draggable scrollbars, hover states, right-click context menus, and keyboard shortcuts.
5. 🟡 **How do you honor back conventions?** — iOS edge-swipe (Cupertino/adaptive page routes), Android system back; `PopScope` for exit confirmation.
6. 🔴 **Why adapt behavior even more than visuals?** — Users feel wrong *behavior* (scroll/back/menus) immediately; behavioral mismatches break the native feel more than styling.
7. 🔴 **How do you keep adaptive behavior maintainable?** — Centralize choices in a design-system/adaptive-scaffold layer; rely on automatic adaptations, add manual ones selectively.

Senior Engineer Tips

- Build one **adaptive scaffold** (nav bar ↔ rail/menu, scrollbars, back handling) driven by platform + size; screens just supply content.
- Don't forget **desktop/web input** (mouse hover, right-click, keyboard) — a common gap for mobile-first Flutter teams.
- Let the framework's defaults do the heavy lifting (physics/transitions); reserve manual adaptation for navigation, scrollbars, menus, and pickers.

Architect Perspective

Behavioral adaptation (scroll/back/navigation/scrollbars/menus/pickers) is what truly makes an app feel native across platforms and input devices. Centralizing it in an adaptive scaffold/design system — combined with responsive size tiers (Module 24) — delivers native behavior everywhere from one codebase, and is essential for web/desktop targets (Modules 53/54).

Summary

- Adapt behaviors: scroll physics, back gesture/behavior, navigation (tabs↔rail/menu), scrollbars, context menus/hover, pickers.
- Framework + `.adaptive` handle much automatically; adapt navigation/scrollbars/menus/pickers selectively.
- Provide desktop/web input affordances; centralize in an adaptive scaffold; combine with responsive tiers.

Revision Notes

- Adapt: ScrollPhysics (bouncing iOS/clamping Android, `MaterialApp` default), back (iOS edge-swipe/Android system), nav (bottom↔rail/menu), scrollbars (`thumbVisibility` desktop), right-click/hover, pickers.
- Automatic via MaterialApp/ `.adaptive` ; manual selective for nav/scrollbars/menus/pickers.
- Desktop/web: visible scrollbars + hover + context menus + shortcuts.
- Centralize in adaptive scaffold; combine with responsive size tiers.

Practice Questions

1. How does scroll physics differ by platform and how is it set?
2. How should navigation adapt from mobile to desktop?
3. What desktop/web behaviors do mobile-first apps often miss?

Coding Questions

1. Build an adaptive scaffold: bottom nav (mobile) ↔ navigation rail (desktop).
2. Add always-visible scrollbars + hover/right-click context menu for desktop.
3. Use a Cupertino date picker on iOS and Material on Android.

Mini Project

Adaptive navigation scaffold (Flutter): Build a reusable scaffold that renders bottom navigation on mobile and a navigation rail/side menu on desktop/wide, with platform-correct scroll physics, iOS edge-swipe back, desktop scrollbars, and right-click context menus on desktop. Acceptance: navigation adapts by platform+size; native scroll/back; desktop affordances (scrollbar/hover/right-click); centralized/reusable; runs on mobile + desktop/web.

Cross-Platform Design System

> Centralize adaptation (and branding) in a **design system**: shared tokens (color/typography/spacing) applied to Material *and* Cupertino, plus reusable adaptive components and an adaptive scaffold — so screens use consistent, platform-correct building blocks without inline platform/size branching.

Introduction

The synthesis of this module (and the UI-craft cluster): a design system that turns responsive + adaptive rules into **reusable components + tokens**. This file covers structuring tokens, adaptive component wrappers, an adaptive scaffold, and how it keeps a large app consistent, native-feeling, and maintainable.

Why this concept exists

Without a design system, platform/size branching and styling scatter across screens — inconsistent, hard to change, error-prone. A design system centralizes the *adapt-vs-consistent policy* (01_adaptive_fundamentals.md), tokens, and components, so screens compose clean, on-brand, platform-correct pieces.

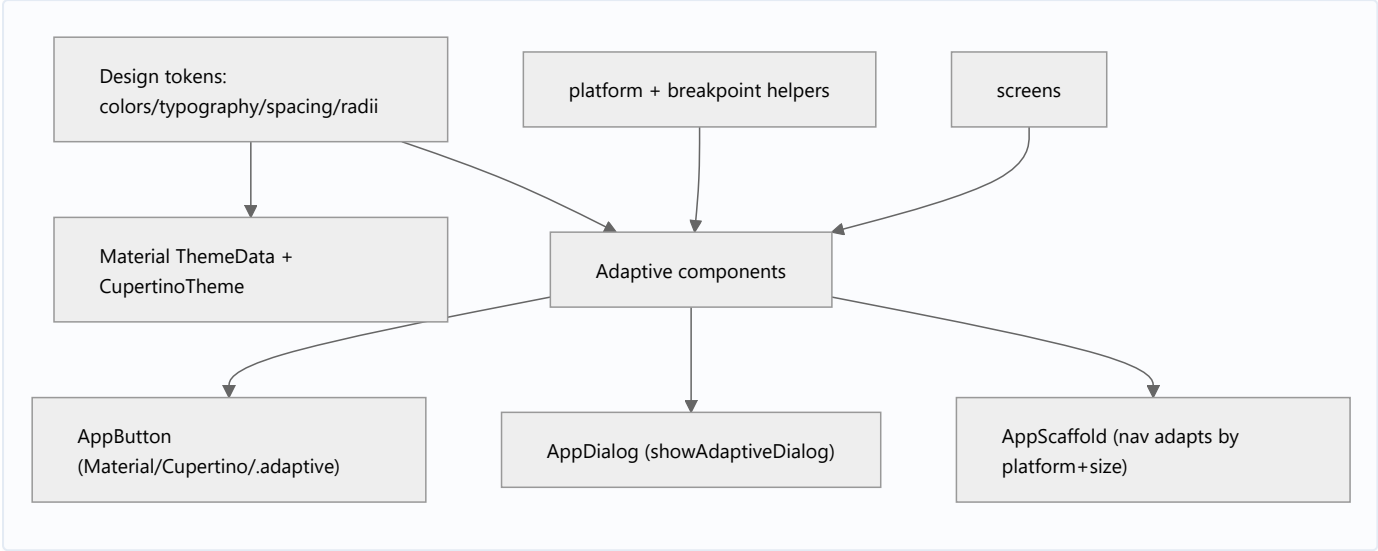
Real-world analogy

A design system is a **brand style guide + a kit of pre-approved, localized components**: designers/engineers assemble screens from vetted parts (that already handle platform/size), rather than hand-crafting each one — consistent output, faster work, easy global changes.

Problem Statement

Your app has inconsistent buttons/dialogs, scattered platform checks, and duplicated styling. Build a design-system layer — tokens, adaptive components (`AppButton` , `AppDialog` , `AppScaffold`), and helpers — so screens are clean, consistent, and platform/size-correct. You'll structure the tokens + components.

Internal Working



- **Tokens** (single source of truth): colors (semantic roles), typography scale, spacing scale, radii, durations — as constants/theme extensions. Applied to **both** `ThemeData` (Material) and `CupertinoTheme` so both kits look like *your* brand (07 · text_and_theming).
- **Adaptive components** (reusable widgets that encapsulate branching): e.g., `AppButton` (uses `.adaptive` /Material/Cupertino), `AppDialog` (`showAdaptiveDialog`), `AppTextField` , `AppSwitch` (`Switch.adaptive`) — screens use these, never raw platform branches.
- **Adaptive scaffold**: `AppScaffold` that adapts navigation (bottom nav ↔ rail/menu) by platform + size, handles scrollbars/back — combining adaptive (03_platform_adaptive_widgets_and_navigation.md) + responsive (Module 24).
- **Helpers/extensions**: centralized `context.platform` , `context.windowSize` , `context.spacing` , `context.colors` so screens read intent, not raw metrics — and detection stays overridable/testable.
- **Consistency policy** lives here: the design system enforces the adapt-vs-consistent decisions once.

Memory Representation

Tokens are constants/theme objects (shared, cheap); components are ordinary widgets. No special cost.

Compiler Behavior

Theme extensions/tokens are type-safe; components centralize platform/size logic (testable, overridable).

Runtime Behavior

Components read tokens + platform/size and render appropriately; a token/theme change updates the whole app consistently.

Flutter Engine Behavior / Dart VM Behavior

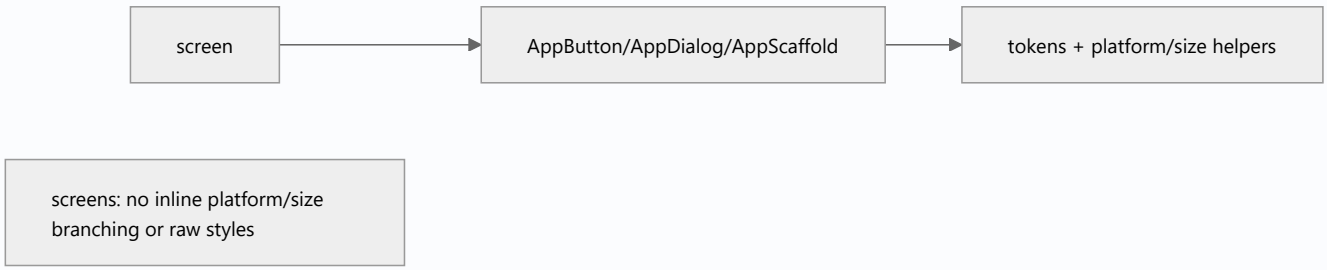
Not applicable.

Examples

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


// 1) Tokens (single source of truth) — as a ThemeExtension for type-safe access
@immutable
class AppTokens extends ThemeExtension<AppTokens> {
  final double spacingUnit;
  final double radius;
  const AppTokens({this.spacingUnit = 8, this.radius = 12});
  @override AppTokens copyWith({double? spacingUnit, double? radius}) =>
      AppTokens(spacingUnit: spacingUnit ?? this.spacingUnit, radius: radius ?? this.radius);
  @override AppTokens lerp(AppTokens? other, double t) => this; // simplified
}


extension AppContextX on BuildContext {
  AppTokens get tokens => Theme.of(this).extension<AppTokens>()!;
  bool get isApple {
    final p = Theme.of(this).platform;
    return p == TargetPlatform.iOS || p == TargetPlatform.macOS;
  }
  bool get isWide => MediaQuery.sizeOf(this).width >= 900;
}


// 2) Adaptive component (screens use THIS, not raw platform branches)
class AppButton extends StatelessWidget {
  final String label; final VoidCallback? onPressed;
  const AppButton(this.label, {super.key, this.onPressed});
  @override
  Widget build(BuildContext context) {
    // Could branch to CupertinoButton on Apple; here Material with brand radius:
```

```
    return FilledButton(
      style: FilledButton.styleFrom(
        shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(context.tokens.radius)),
      ),
      onPressed: onPressed,
      child: Text(label),
    );
  }
}

// 3) Adaptive scaffold (nav adapts by platform+size) — see platform_adaptive_widgets file
// class AppScaffold extends StatelessWidget { ... bottom nav <-> rail by context.isWide ... }

// 4) Theme wiring (both kits share tokens)
ThemeData appTheme() => ThemeData(
      useMaterial3: true,
      colorSchemeSeed: Colors.indigo,        // brand
      extensions: const [AppTokens()],
    );
// CupertinoTheme derived from the same brand colors for consistency.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Platform/size branching in screens	Scattered, inconsistent, untestable	Encapsulate in design-system components
Hardcoded colors/spacing everywhere	Inconsistent, hard to rebrand	Centralize tokens; access via theme/extensions
Styling only Material (Cupertino off-brand)	Disjointed on iOS	Apply tokens to both kits
No adaptive scaffold	Repeated nav/scroll/back logic	One `AppScaffold`
Over-abstracting tiny one-offs	Ceremony	Componentize repeated/branching pieces
Non-overridable platform detection in components	Hard to test/preview	Use `Theme.of(context).platform` (overridable)

Best Practices

- Define **tokens** (color/typography/spacing/radii/durations) as the single source of truth; apply to **both** Material and Cupertino themes.
- Build **adaptive components** (`AppButton` / `AppDialog` / `AppScaffold` /…) that encapsulate `.adaptive` /platform/size branching — screens use these, never raw branches.
- Expose **helpers/extensions** (`context.platform` , `context.windowSize` , `context.tokens`) so screens read intent; keep detection **overridable/testable**.
- Centralize the **adapt-vs-consistent policy** in the design system.
- Combine adaptive (platform) + responsive (size) at the component level; document and version the system.

Performance

Tokens/components are lightweight; a global token change updates the app cheaply. Centralization reduces duplicated (and buggy) branching (21 · rebuild_optimization).

Advantages / Disadvantages

- **+** Consistency, native feel, single place to change branding/adaptation, testable, faster feature work at scale.
- **−** Upfront investment; governance/maintenance; risk of over-engineering small apps.

Interview Questions

1. 🟢 **What is a design system in this context?** — A centralized layer of shared tokens + reusable adaptive components that encapsulate platform/size branding and behavior.

2. 🟢 **Why centralize adaptation in components?** — To keep screens free of scattered, inconsistent platform/size branching and make changes/testing single-point.
3. 🟡 **How do you keep both Material and Cupertino on-brand?** — Apply the same tokens (colors/typography/spacing) to both `ThemeData` and `CupertinoTheme`.
4. 🟡 **What belongs in tokens?** — Colors (semantic roles), typography scale, spacing/radii, durations — the brand's design decisions.
5. 🟡 **How do screens access adaptation without branching?** — Via design-system components (`AppButton` / `AppScaffold`) and helpers (`context.platform` / `windowSize` / `tokens`).
6. 🔴 **How does the design system combine adaptive + responsive?** — Components read both platform (adaptive) and size (responsive) to render correctly, so screens compose one adaptive/responsive kit.
7. 🔴 **When is a design system overkill?** — For very small/short-lived apps; the investment pays off with scale, multiple platforms, and team size.

Senior Engineer Tips

- Start tokens + a handful of adaptive components early; retrofitting a design system into a branch-everywhere codebase is painful.
- Make components the *only* place platform/size branching lives; enforce "no raw `Platform`/breakpoint checks in screens" in review.
- Version and document the system; treat it as a product the whole team consumes.

Architect Perspective

A cross-platform design system is the architectural capstone of UI craft: it operationalizes the adapt-vs-consistent policy, unifies responsive + adaptive + theming into reusable components, and gives one codebase native feel + brand consistency across platforms and form factors — at scale, with maintainability. It's where Modules 07, 22, 23, 24, and 25 converge (Module 24, Module 47).

Summary

- Centralize tokens (applied to Material + Cupertino) + adaptive components + an adaptive scaffold + helpers.
- Screens compose design-system components — no inline platform/size branching or raw styles.
- Combines adaptive (platform) + responsive (size) + branding; the maintainable, scalable UI foundation.

Revision Notes

- Tokens (color/typography/spacing/radii/durations) → both `ThemeData` + `CupertinoTheme`.
- Adaptive components (`AppButton` / `AppDialog` / `AppScaffold`) encapsulate `.adaptive` /platform/size; screens use them.
- Helpers/extensions (`context.platform` / `windowSize` / `tokens`); overridable detection.
- Centralizes adapt-vs-consistent policy; combines adaptive + responsive; invest early, govern/version.

Practice Questions

1. Why encapsulate platform/size branching in components?
2. How do you keep Material and Cupertino on-brand?
3. What goes in tokens, and how are they accessed?

Coding Questions

1. Define an `AppTokens` `ThemeExtension` + `context.tokens` accessor and use it.
2. Build an `AppButton` and `AppDialog` that adapt per platform.
3. Build an `AppScaffold` that adapts navigation by platform + size.

Mini Project — Module capstone

Adaptive design system (Flutter): Build a small design system: tokens (color/typography/spacing/radii) applied to Material + Cupertino, adaptive components (`AppButton` , `AppSwitch` , `AppDialog` , `AppScaffold` with nav adapting by platform+size), and `context` helpers (`platform` / `windowSize` / `tokens`). Build a demo screen using only these components — no inline platform/size branching. Preview iOS via `TargetPlatform` override and resize for size tiers. Acceptance: consistent branding across kits; platform+size adaptation centralized in components; screens branch-free; overridable/testable; runs on mobile + desktop/web.