Interview Practice

Flutter Practice Notes

Contents

Interview Practice — Flutter/Dart Question Bank

Dart Core — Interview Questions

Advanced Dart & Asynchrony — Interview Questions

OOP, SOLID & Design Patterns (in Dart) — Interview Questions

Flutter Internals & Architecture — Interview Questions

Widgets, Layout & Lifecycle — Interview Questions

Rendering Pipeline & Performance — Interview Questions

State Management — Interview Questions

Navigation & Routing — Interview Questions

Networking, Data & Storage — Interview Questions

Dependency Injection — Interview Questions

Animations & Custom UI — Interview Questions

App Architecture — Interview Questions

Testing — Interview Questions

Platform Integration, Native & Device — Interview Questions

Security, Error Handling & Logging — Interview Questions

Mobile System Design (Flutter) — Interview Questions

CI/CD, Deployment & Monitoring — Interview Questions

Behavioral & Scenario Questions — Interview Questions

Rapid-Fire & Mixed Mock Rounds — Interview Questions

Interview Practice — Flutter/Dart Question Bank

> A consolidated, **topic-wise** interview question bank spanning 🟢 **Basic** → 🟡 **Intermediate** → 🔴 **Advanced**, with crisp model answers. Complements the per-file *Interview Questions* sections in the handbook modules by pulling everything into a single drill-ready place.

How to use this

1. **Pick a topic file** below matching what you're revising or what the role emphasizes.
2. Read top-to-bottom: each file goes **Basic** → **Intermediate** → **Advanced**, so you always know your depth.
3. **Answer out loud first**, then check the model answer. Interviews test articulation, not recognition.
4. For depth on any answer, follow the cross-link to the matching handbook module.
5. The night before an interview: skim **19_rapid_fire_mixed.md** and the Advanced tier of your target topics.

Difficulty legend

Tag	Level	Who it's for
🟢	**Basic**	0–1.5 yrs · fundamentals, definitions, "what is X"
🟡	**Intermediate**	1.5–4 yrs · "how does X work", tradeoffs, common bugs
🔴	**Advanced**	4+ yrs / senior · internals, scaling, design, "why" and "when NOT"

Topic index

#	File	Covers	Handbook modules
01	01_dart_core.md	Types, null safety, collections, functions, records & patterns	01
02	02_dart_advanced_async.md	Futures, async/await, Streams, isolates, event loop, generics, mixins	02
03	03_oop_solid_patterns.md	OOP, SOLID, GoF design patterns in Dart	03, 04, 05
04	04_flutter_internals.md	Widget/Element/RenderObject trees, BuildContext, keys, how Flutter runs	06, 10
05	05_widgets_layout_lifecycle.md	Widgets, layout & constraints, Stateless/Stateful, State lifecycle	07, 08
06	06_rendering_performance.md	Rendering pipeline, repaint boundaries, jank, performance tuning	09, 21
07	07_state_management.md	setState, Provider, Riverpod, BLoC/Cubit, GetX, selection	11
08	08_navigation_routing.md	Navigator 1.0/2.0, go_router, deep links, guards	12, 13
09	09_networking_data_storage.md	REST/GraphQL, JSON, storage, SQLite/Drift/Hive, offline, caching	15, 16, 19, 20
10	10_dependency_injection.md	DI principles, get_it/injectable, provider/riverpod as DI, testing	14
11	11_animations_custom_ui.md	Implicit/explicit/physics animations, CustomPainter, responsive/adaptive	22, 23, 24, 25
12	12_architecture.md	Clean, MVC/MVP/MVVM, feature-first, modular, DDD, scalable	40–47
13	13_testing.md	Unit/widget/integration/golden, mocking, TDD	49
14	14_platform_native_device.md	Platform channels, FFI, native Android/iOS, device, notifications, background	26–33
15	15_security_error_logging.md	Security, secure storage, error handling, logging	37, 38, 39
16	16_system_design_mobile.md	Mobile HLD/LLD, scenario design, scaling a Flutter app	48
17	17_cicd_deployment_monitoring.md	CI/CD, flavors, signing, store release, monitoring	50, 51, 52
18	18_behavioral_and_scenarios.md	Behavioral, situational, ownership, mentoring, conflict	58, 60
19	19_rapid_fire_mixed.md	One-line rapid-fire + mixed mock rounds across topics	all

Company focus (quick map)

- **Product companies (Google/Uber/Atlassian):** heavy on 04, 06, 12, 16 + 18.
- **Fintech (Razorpay/PhonePe/CRED):** heavy on 09, 15, 13, 12.
- **Consumer/social (Swiggy/Flipkart/Airbnb):** heavy on 06, 07, 11, 14.

> Company-specific and format-specific prep (rounds, timing, machine-coding) lives in handbook modules **55 Flutter Interview Preparation** and **56 Machine Coding Rounds**. This folder is the **topic drill**; those are the **format drill**.

Summary

- 19 topic files, each **Basic** → **Intermediate** → **Advanced**, model answers included.
- Consolidated drill companion to the per-module interview sections.
- Pair with Modules 55/56 for format/company prep.

Dart Core — Interview Questions

> Language fundamentals: variables & finality, the type system, sound null safety, collections, functions & closures, Dart 3 records & patterns, and enums. For depth see 01 Dart Fundamentals and 02 Advanced Dart.

This topic tests whether you actually understand Dart as a language — not just Flutter APIs. Interviewers probe it because null-safety mistakes, `const` misuse, and pattern-matching gaps are the most common source of runtime crashes and rebuild bugs in real apps.

🟢 Basic

1. What is the difference between `var`, `final`, and `const`? `var` declares a mutable variable whose type is inferred once at the declaration. `final` means the binding is assigned exactly once and can't be reassigned, but the object it points to can still be mutated. `const` is a stronger `final`: the value must be a *compile-time constant*, so it's frozen deeply and computed before the program runs.

```
final list = [1, 2];
list.add(3);   // OK — the reference is final, the List is mutable
const list2 = [1, 2];
list2.add(3);  // runtime error — const collections are deeply immutable
```

2. When would you choose `final` over `const`? Use `const` when the value is fully known at compile time (literals, `const` constructors) — you gain canonicalization and cheaper rebuilds. Use `final` when the value is known only at runtime (e.g. `DateTime.now()`, a value from a function, a constructor arg). If you *can* make it `const`, prefer it; otherwise `final`.

3. What are Dart's core built-in types? `num` (superclass of `int` and `double`), `String`, `bool`, `List`, `Set`, `Map`, `Runes`, and `Symbol`. `int` and `double` are both `num`; note that on the web all numbers are IEEE-754 doubles, so `int` there is a double under the hood. `null` is its own type (`Null`).

4. What's the difference between `dynamic`, `Object`, and `Object?`? `Object?` is the top type — everything, including `null`, is an `Object?`. `Object` is every non-null value. `dynamic` is also a top type but it *disables static type checking*: calls on a `dynamic` are resolved at runtime and can throw `NoSuchMethodError`. Prefer `Object?` when you want "anything but keep type safety"; reserve `dynamic` for genuine interop/JSON cases.

5. What is sound null safety? Types are non-nullable by default; `String` can never hold `null`, only `String?` can. "Sound" means the guarantee is enforced end-to-end — if the type system says a value is non-null, the runtime trusts it (no null checks compiled in), so you can't get a

surprise null through a non-nullable variable. This eliminates most `NoSuchMethodError: null` crashes.

6. Explain `?`, `!`, `??`, and `??=`.

- `?` — nullable type (`int?`) or null-aware access (`user?.name`, returns null if `user` is null).
- `!` — non-null assertion: `value!` throws if `value` is null, otherwise narrows to non-null.
- `??` — if-null: `a ?? b` returns `a` unless it's null, then `b`.
- `??=` — assign-if-null: `x ??= 5` sets `x` to 5 only when it's currently null.

7. What does `late` do? `late` promises the compiler a non-nullable variable will be initialized before first use, deferring initialization past the declaration. For a `late` with an initializer, it also makes the init *lazy* — computed on first read. Reading a `late` before assignment throws `LateInitializationError`.

8. How do you write list, set, and map literals?

```
final list = [1, 2, 3];              // List<int>
final set  = {1, 2, 3};              // Set<int>
final map  = {'a': 1, 'b': 2};       // Map<String, int>
final empty = <String>{};            // empty Set (bare {} is a Map)
```

An empty `{}` is a `Map`, not a `Set` — annotate the type to disambiguate.

9. What is string interpolation? Embedding expressions in a string with `$` — `'$name'` for a simple identifier, `'${expr}'` for any expression. It calls `toString()` on the value. Prefer it over concatenation for readability and fewer allocations.

10. What are positional vs named parameters? Positional params are matched by order; named params by name in `{}`. Named params are optional by default unless marked `required`. Positional optionals use `[]` with optional defaults.

```
void f(int a, [int b = 0]) {}          // b optional positional
void g({required int a, int b = 0}) {} // a required named, b optional
```

11. What is an arrow function? `=>` is shorthand for a function body that's a single expression: `int sq(int x) => x * x;` is equivalent to `{ return x * x; }`. It can only wrap one expression, not statements.

12. What is an enum in Dart? A fixed set of named constant values: `enum Status { active, paused, done }`. Each has `.name`, `.index`, and the type gets `.values`. Enums are commonly used in `switch` for exhaustive handling.

🟡 Intermediate

13. What is flow analysis / type promotion in null safety? The compiler tracks control flow to *promote* a nullable variable to non-nullable within a scope where it can't be null.

```
void f(String? s) {
  if (s == null) return;
  print(s.length); // s promoted to String here
}
```

Promotion only works on **local variables and parameters**, never on non-final fields or getters (they could change between checks) — a top reason `!` shows up in real code.

14. Why can't you promote a class field, and what are the workarounds? A field could be overridden by a getter or mutated by another isolate/callback between the null check and the use, so the compiler can't prove it stays non-null. Workarounds: copy to a local (`final x = field; if (x != null) x.use();`), make the field `final`, or use `?.` / `!`.

15. What's wrong with over-using `!`? `!` silences the compiler but reintroduces the exact runtime null crashes null safety was designed to prevent. Each `!` is a claim "I know better than the type system"; if you're wrong it throws. Prefer `?.`, `??`, promotion, or fixing the type. Reserve `!` for cases you can genuinely prove non-null (e.g. right after a check the compiler can't follow).

16. Explain spread, collection-if, and collection-for. They build collections declaratively inside a literal:

```
final base = [1, 2];
final all = [
  0,
  ...base,                // spread
  ...?maybeNullList,      // null-aware spread
  if (isAdmin) 99,        // collection-if
  for (final n in base) n * 10, // collection-for
];
```

This avoids imperative `.add()` chains and keeps `const` collections possible.

17. How do you make a collection immutable? Use `List.unmodifiable(source)` / `Map.unmodifiable` / `Set.unmodifiable`, or `const []` for compile-time constants, or `UnmodifiableListView` for a live read-only wrapper. `const` collections are deeply immutable; `unmodifiable` creates a fixed copy at runtime. Returning unmodifiable views from getters prevents callers from mutating your internal state.

18. What is a closure? A function that captures and retains variables from its lexical scope, even after that scope has exited.

```
Function counter() {
  var count = 0;
  return () => ++count; // captures `count`
}
final c = counter();
c(); c(); // 1, 2 — state persists in the closure
```

Closures are how callbacks, `Future.then`, and event handlers carry state.

19. What's the classic closure-in-a-loop pitfall? With `var`, all closures could share one variable; Dart avoids this because a `for`-loop variable is fresh per iteration.

```
final fns = [for (var i = 0; i < 3; i++) () => i];
print(fns.map((f) => f())); // (0, 1, 2) — each closure captured its own i
```

This differs from older JS `var` semantics — worth stating in an interview.

20. What are higher-order functions? Give Dart examples. Functions that take or return functions. Dart's iterables lean on them: `map`, `where`, `fold`, `reduce`, `sort` (with a comparator), `expand`.

```
final evens = [1, 2, 3, 4].where((n) => n.isEven).toList(); // [2, 4]
```

21. What are records (Dart 3)? Anonymous, immutable, fixed-size bundles of values with positional and/or named fields — a lightweight tuple that lets a function return multiple values without a class.

```
(int, String) pair() => (1, 'a');
({int x, int y}) point() => (x: 1, y: 2);
final r = pair();
print(r.$1);       // positional field access
print(point().x);  // named field access
```

Records have structural equality — two records with equal fields are `==`.

22. What is destructuring / pattern matching? Patterns pull values out of records, lists, maps, and objects in one step.

```
final (a, b) = (1, 2);            // record destructure
final [first, _, third] = [1, 2, 3]; // list pattern, _ discards
final {'id': id} = json;          // map pattern
var Point(x: px, y: py) = point; // object pattern
```

Patterns work in variable declarations, `switch`, and `if-case`.

23. What are enhanced enums? Since Dart 2.17, enums can have fields, constructors, methods, and implement interfaces.

```
enum Planet {
  earth(9.8), mars(3.7);
  const Planet(this.gravity);
  final double gravity;
  bool get isHeavy => gravity > 5;
}
```

This replaces the old "enum + extension" workaround for attaching data/behavior.

24. Difference between `==` and `identical()`? `==` is value/logical equality and can be overridden. `identical(a, b)` checks reference identity — whether they're the exact same object in memory, and cannot be overridden. For two separate list literals with equal contents, `==` is false (default identity) but `identical` is also false; for canonicalized `const` values, both are true.

🔴 Advanced

25. Explain `const` canonicalization. Identical `const` expressions are *canonicalized* to a single shared instance at compile time. So `const Point(1, 2)` used in ten places allocates one object, and `identical(const [1], const [1])` is `true`. In Flutter this is why `const` widgets are cheap: the framework can skip rebuilding a subtree when it receives the *same instance* it already has.

26. Why does making Flutter widgets `const` improve performance? A `const` widget is a canonicalized, immutable instance. During rebuilds, when a parent hands its child the identical `const` object, `Element.updateChild` short-circuits — same `Widget` instance means no `updateShouldNotify` / `build` work for that subtree. It also reduces allocations. Hence the `prefer_const_constructors` lint.

27. What are the requirements for an expression to be a compile-time constant? All inputs must themselves be constants: literals, other `const` values, `const` constructors invoked with `const` args, and a limited set of operations (arithmetic, string concat, `==` on primitives, ternaries, `identical`). No method calls on runtime objects, no `DateTime.now()`, no non-const constructor. A class needs `const` constructor with all-`final` fields to be constructible as `const`.

28. What makes a class support a `const` constructor? Every instance field must be `final`, and the constructor must be declared `const` with no body that mutates state (initializer list only). Subclasses can add their own `const` constructors only if the superclass has one. This immutability is what allows canonicalization.

29. How does `switch` exhaustiveness work with sealed classes? Marking a class `sealed` makes it abstract and restricts subclassing to the same library, so the compiler knows the *complete* set of subtypes. A `switch` over a sealed type is then checked for exhaustiveness — miss a case and you get a **compile-time** error, no `default` needed.

```
sealed class Shape {}
class Circle extends Shape {}
class Square extends Shape {}

double area(Shape s) => switch (s) {
  Circle() => 3.14,
  Square() => 4.0,
  // add a third subtype → this switch stops compiling until handled
};
```

This gives you compiler-verified handling of every state — huge for modeling API/UI states.

30. What are guard clauses and `when` in patterns? A `when` clause adds a boolean condition to a pattern case; it must match *and* the guard must be true.

```
final result = switch (value) {
  int n when n > 0 => 'positive',
  int n when n < 0 => 'negative',
  int _ => 'zero',
  _ => 'not an int',
};
```

Unlike a nested `if`, a failed guard falls through to the next case rather than exiting the switch.

31. Explain `if-case` and where it shines. `if (x case Pattern)` matches and destructures in one condition, binding variables scoped to the branch.

```
if (json case {'user': {'id': int id}}) {
  print(id); // id is int, only reachable when the shape matches
}
```

It's the cleanest way to validate-and-extract nested JSON without a cascade of null checks.

32. What's the difference between `late final` and `final`? `final` must be initialized at declaration or in the constructor initializer list. `late final` defers that single assignment to any point before first read, still enforcing write-once. Useful for fields whose value depends on `this` or on async setup, or for lazy computation. Beware: `late final` without an initializer throws if read before assignment, and offers no compile-time guarantee of initialization.

33. What are the runtime costs and hazards of `late`? Each read of a `late` variable compiles to a check "has this been initialized?", so there's a small per-access overhead and the risk of `LateInitializationError`. A `late` field with an initializer runs that initializer lazily on first access — meaning side effects fire at an unpredictable time. Don't use `late` to paper over a design where a value genuinely might be absent; use `?` instead.

34. How does numeric behavior differ between the VM and web? On the native VM, `int` is a true 64-bit integer. On the web (compiled to JS), all numbers are IEEE-754 doubles, so `int` is emulated — large integers lose precision beyond 2^53, and `is int` / `is double` checks can behave unexpectedly (`1.0 is int` can be true on web). For cross-platform correctness with big IDs, use `BigInt` or strings.

35. When should you actually use `dynamic`, and what's the safer alternative? Use `dynamic` only for genuine dynamism: decoded JSON before typing, reflection-like interop, or `noSuchMethod` forwarding. Everywhere else prefer `Object?` — you keep the "any value" flexibility but the compiler forces you to check the type (`is`) or cast before calling methods, catching mistakes at compile time instead of runtime `NoSuchMethodError`.

36. How do you implement value equality correctly for a class? Override both `==` and `hashCode` consistently — equal objects must have equal hash codes, or `Set` / `Map` break.

```
class P {
  final int x, y;
  const P(this.x, this.y);
  @override
  bool operator ==(Object other) =>
      other is P && other.x == x && other.y == y;
  @override
  int get hashCode => Object.hash(x, y);
}
```

Use `Object.hash` / `Object.hashAll` for combining. In practice, prefer records or a codegen tool (freezed/equatable) to avoid mistakes.

37. Records vs a data class — when to pick which? Records are ideal for *ephemeral, local* groupings: multiple return values, a temporary key, destructuring. They give free structural equality and less boilerplate. Choose a named class when the bundle has identity/behavior, is

part of your domain model, needs documentation, named construction validation, or will be widely referenced — a record's positional fields (`$1` , `$2`) hurt readability across a codebase.

Rapid-fire (one-liners)

Question	Answer
Default value of an uninitialized nullable field?	`null`
Is bare `{}` a Set or Map?	Map (empty)
Does `final` make the object immutable?	No — only the binding
Can you promote a non-final field?	No
Top type in Dart?	`Object?`
`identical(const [1], const [1])` ?	`true` (canonicalized)
Operator you must override alongside `==` ?	`hashCode`
Keyword for compile-time exhaustive switch?	`sealed`
Access second positional record field?	`.$2`
What throws if a `late` var is read too early?	`LateInitializationError`
Null-aware spread operator?	`...?`
`a ??= b` means?	assign `b` to `a` only if `a` is null
Is `int` a subtype of `num` ?	Yes
Can enums have methods in modern Dart?	Yes (enhanced enums)
Difference `Object` vs `Object?` ?	`Object?` includes `null`

Follow-up drills

1. Design an immutable `Money` value type with correct `==` / `hashCode` , a `const` constructor, and arithmetic that returns new instances — then justify each `const` / `final` choice.
2. Model a network request result as a `sealed` class hierarchy (`Loading` / `Success` / `Error`) and write an exhaustive `switch` UI mapper; show how adding a new state fails compilation until handled.
3. Refactor a JSON-parsing function riddled with `!` into safe `if-case` pattern matching and explain each null-safety improvement.
4. Debug: a widget rebuilds every frame despite "looking const" — walk through how a missing `const` , a non- `const` field, or a broken `==` could cause it.
5. Explain to a junior why type promotion works on a local copy of a field but not the field itself, with a concrete race/override example.

6. Optimize a hot list-building path using spread/collection-for and `const` sub-collections; measure what canonicalization saves.

Advanced Dart & Asynchrony — Interview Questions

> Concurrency, the event loop, Futures/Streams/isolates, and the power features that separate a Dart *engineer* from a Flutter *user*. Depth lives in 02 Advanced Dart — start with 01_event_loop.md, and the runtime/threading foundations in 59 Computer Science Foundations.

This topic tests whether you understand *what actually runs your code*. Interviewers lean hard here because the event loop, isolates, and stream lifecycle expose the difference between someone who sprinkles `await` and someone who can debug jank, leaks, and race conditions.

🟢 Basic

1. Is Dart single-threaded? Then how does it do async work? A single isolate runs on one thread with one event loop. It never runs two pieces of your Dart code at the same instant. Async work isn't parallelism — it's *interleaving*: I/O, timers, and futures register callbacks that the event loop runs one at a time when the current synchronous chunk finishes. True parallelism needs a second isolate. So "asynchronous" means "not now, later on this same thread," not "on another core."

2. What is a `Future`? A `Future<T>` is a handle to a value that isn't available yet — it will complete once with either a value or an error. Creating a future starts the work (or schedules it); you attach continuations with `then` / `catchError` or `await`. It's Dart's equivalent of a JS Promise. A future can be in one of three states: uncompleted, completed-with-value, or completed-with-error.

3. What's the difference between the microtask queue and the event queue? Both are FIFO queues the event loop drains, but the microtask queue has strict priority. After each event-queue task, the loop fully empties the microtask queue *before* touching the next event. Microtasks come from `scheduleMicrotask` and from resolved-future continuations; events come from timers, I/O, gestures, and `Timer.run`. Practical rule: microtasks are for short internal follow-ups; abusing them can starve the event queue and freeze the UI.

4. Predict the output.

```
void main() {
  print('A');
  Future(() => print('B'));                 // event queue
  Future.microtask(() => print('C'));        // microtask queue
  Future.value(0).then((_) => print('D'));   // microtask (already complete)
  scheduleMicrotask(() => print('E'));       // microtask queue
  print('F');
}
// output: A F C D E B
```

Synchronous `A` and `F` first, then *all* microtasks in registration order (C, D, E), then the event-queue `B`.

5. `async` / `await` vs `.then()` — what does `await` actually do? `await` pauses the enclosing `async` function and returns control to the caller/event loop; when the awaited future completes, the *rest* of the function is scheduled as a continuation (a microtask). It's syntactic sugar over `.then()`, but reads sequentially and gives you normal `try/catch`. `.then()` is still useful for fire-and-forget chaining or when you don't want to make the function `async`.

6. What does an `async` function return, and when? It always returns a `Future`. It returns that future *synchronously* the moment execution first hits an `await` (or the end of the function). Code before the first `await` runs synchronously in the caller's turn — a common surprise:

```
Future<void> f() async { print('1'); await null; print('2'); }
void main() { f(); print('3'); }
// output: 1 3 2
```

7. What is a `Stream`? A `Stream<T>` is a sequence of asynchronous events delivered over time — zero or more data events, and either a done or an error event. Think "a Future that can complete many times." You consume it with `await for`, `.listen()`, or stream operators (`map`, `where`, `take`). Futures are for one-shot results; streams are for ongoing sources like sockets, sensor readings, or UI events.

8. `listen()` vs `await for` — when do you use each? `await for` reads a stream sequentially inside an `async` function and is clean for consuming until done, but it *blocks that function* and you can't easily manage the subscription. `.listen()` returns a `StreamSubscription` you can `pause`, `resume`, and `cancel`, and lets you keep running — which is what you almost always want in Flutter (e.g. cancel in `dispose`). Prefer `.listen()` when lifecycle control matters.

9. What's a `StreamController` for? It's the standard way to *create* a stream you push events into manually via `add`, `addError`, and `close`. You expose `controller.stream` to consumers. Use it to bridge callback-based or imperative sources into the stream world (e.g. a WebSocket wrapper, a

custom event bus). Always `close()` it to avoid leaks.

10. What is an isolate, in one sentence? An isolate is Dart's unit of concurrency: an independent worker with its *own memory heap and event loop*, communicating only by message passing — so there's no shared mutable state and therefore no locks or data races.

11. What is `compute()` and when do you reach for it? `compute(fn, message)` is a Flutter helper that spawns a one-off isolate, runs a top-level/static `fn` on `message`, returns the result as a `Future`, and tears the isolate down. Use it to move CPU-heavy work (parsing a large JSON, image processing, crypto) off the UI isolate so frames keep rendering. On modern Dart, `Isolate.run` is the more general equivalent.

12. What is a mixin and why not just use inheritance? A `mixin` lets you inject a bundle of methods/fields into a class without a subclass relationship, and a class can mix in *many* of them (`class A extends B with M1, M2`). It's how Dart gets reuse across unrelated hierarchies without multiple inheritance's ambiguity. Example: `SingleTickerProviderStateMixin` gives a `State` the `vsync` capability.

13. What is an extension method? Extensions add methods/getters to an existing type *you don't own* without subclassing or wrapping it, resolved statically at compile time. Example:

```
extension StringX on String {
  bool get isBlank => trim().isEmpty;
}
'  '.isBlank; // true
```

They're syntactic reach, not runtime polymorphism — dispatch is based on the *static* type.

14. What does `const` give you that `final` doesn't? `final` is a runtime single-assignment; `const` is a *compile-time* constant, fully computed and canonicalized during compilation. Two identical `const` values are the same instance (canonicalization), which is why `const` widgets are cheap and skip rebuilds. Everything in a `const` object graph must itself be `const`.

🟡 Intermediate

15. Walk through `Future.wait`, `Future.value`, and error handling. `Future.wait([...])` runs futures concurrently and completes with a `List` of results when *all* finish — but it completes with an *error* as soon as *any* one fails (use `eagerError` / catch per-future to change that). `Future.value(x)` wraps an already-available value in a completed future (its `then` fires on the microtask queue). Standard error handling: `try/catch` around `await`, or `.catchError` / `.then(onError:)` in chains.

16. `catchError` vs `whenComplete` vs `try/catch`?

Tool	Fires on	Analogy
`.then(onValue)`	success	`try` body
`.catchError(fn)`	error (optionally `test:`)	`catch`
`.whenComplete(fn)`	success *or* error	`finally`
`await` + `try/catch/finally`	same, but linear	the whole thing

`whenComplete` runs cleanup regardless of outcome and does not swallow the error (unless it throws its own). `catchError` 's optional `test:` predicate lets you catch only specific error types.

17. How does `async` / `await` desugar? The compiler rewrites an `async` function into a state machine driven by the `Future` /microtask machinery. Each `await` becomes a suspension point: the function registers a continuation on the awaited future and returns; on completion the continuation resumes at the saved state as a microtask. That's why (a) the function returns a future immediately, (b) resumption is asynchronous even for already-complete futures, and (c) exceptions propagate through the returned future rather than up the sync stack.

18. Predict the output — nested futures and microtasks.

```
void main() {
  Future(() => print('1'));
  Future(() { print('2'); Future(() => print('3')); scheduleMicrotask(() => print('4')); });
  Future(() => print('5')).then((_) => print('6'));
  scheduleMicrotask(() => print('7'));
  print('8');
}
// output: 8 7 1 2 4 5 6 3
```

Sync `8` ; microtask `7` ; then event queue in order — `1` ; `2` (which enqueues event `3` and microtask `4`); `4` drains before next event; `5` then its `then` continuation `6` (microtask); finally the deferred `3` .

19. What's `unawaited` and why does it exist? `unawaited(future)` (from `dart:async` / `package:meta`) explicitly signals "I'm intentionally not awaiting this," silencing the `discarded_futures` / `unawaited_futures` lint. It documents intent and keeps you from *accidentally* dropping a future whose errors would otherwise go uncaught. Note: unawaited errors still surface to the isolate's error handler / zone — `unawaited` doesn't swallow them.

20. Single-subscription vs broadcast streams.

	Single-subscription	Broadcast
Listeners	exactly one, ever	many, concurrently
Buffers before listen	yes (pauses source)	no — events fire whether or not anyone listens
Re-listen after cancel	no	yes
Typical source	file/socket read	UI/events, `Stream.periodic().asBroadcast...`

Default `StreamController()` is single-subscription; use `StreamController.broadcast()` for many listeners. Listening twice to a single-subscription stream throws.

21. How do you write your own stream with `async*` / `yield`? An `async*` function is a *stream generator*: each `yield` emits a data event, `yield*` delegates to another stream, and the stream completes when the function returns. It respects backpressure — it pauses at `yield` while the consumer is paused.

```
Stream<int> countTo(int n) async* {
  for (var i = 1; i <= n; i++) { await Future.delayed(Duration(milliseconds: 100)); yield i; }
}
```

(`sync*` + `yield` produces a lazy `Iterable` instead — synchronous, pull-based.)

22. What are stream transformers, and name common operators. A transformer converts one stream into another. Built-in operators: `map`, `where`, `expand`, `take` / `skip`, `distinct`, `asyncMap` (awaits per event), `handleError`, `asBroadcastStream`. For custom logic use `.transform(StreamTransformer.fromHandlers(...))` or `package:rxdart` (`debounceTime`, `switchMap`, `combineLatest`) — heavily used in BLoC. `asyncMap` preserves order and awaits each result, unlike firing overlapping futures yourself.

23. Why must you cancel stream subscriptions, and where in Flutter? An un-cancelled subscription keeps the callback (and its captured `this` / `State`) alive, leaking memory and — worse — calling `setState` on a disposed widget, which throws. Cancel in `State.dispose()`:

```
StreamSubscription? _sub;
@override void initState() { super.initState(); _sub = stream.listen(_onData); }
@override void dispose() { _sub?.cancel(); super.dispose(); }
```

Or use `StreamBuilder`, which manages the subscription for you.

24. Why can't isolates just share objects like threads do? Because Dart's memory model is deliberately share-nothing: each isolate owns its heap, so there's no shared mutable state, no need for locks/mutexes, and no data races — the entire class of concurrency bugs that plague

Java/C++ threads is designed out. The tradeoff is that communication is by *copying* messages across ports (structured clone), which costs serialization but keeps things safe. See 59 Computer Science Foundations for the threads-vs-message-passing tradeoff.

25. How do isolates communicate — ports? Via `SendPort` / `ReceivePort` pairs. You create a `ReceivePort`, pass its `sendPort` to the spawned isolate (through `Isolate.spawn`'s message), and each side sends messages the other receives as stream events. Messages must be sendable (primitives, lists/maps of sendables, `TransferableTypedData`; not closures capturing state or most native handles). `Isolate.run` / `compute` hide all this boilerplate for the common request/response case.

26. Bounded generics — what and why? A bound constrains a type parameter with `extends`, so you can call the bound's members inside:

```
T maxOf<T extends Comparable<T>>(T a, T b) => a.compareTo(b) >= 0 ? a : b;
```

Without `extends Comparable<T>`, the compiler couldn't guarantee `compareTo` exists. Bounds give you compile-time safety *and* the ability to use the constrained API, instead of falling back to `dynamic`.

27. Explain `mixin ... on` and linearization. `mixin M on Base` restricts `M` to classes that extend/implement `Base`, letting `M` call `Base`'s members and `super`. When multiple mixins are applied, Dart *linearizes* them into a single chain — rightmost mixin wins for overrides, and `super` calls walk left along that chain. This deterministic ordering is how Dart resolves the "diamond problem" without ambiguity.

28. Extension methods vs extension types — what's the difference? Extension *methods* add behavior to an existing type; the object is unchanged and dispatch is static. An extension *type* (Dart 3.3+) is a *zero-cost wrapper*: a compile-time-only type over a representation object with no runtime allocation, used to give a distinct, safer API to an existing type (e.g. `extension type UserId(int id)`), erasing to the underlying type at runtime. Extensions add methods; extension types create a new (compile-time) type.

29. How do you make a truly immutable value object in Dart? Mark fields `final`, give a `const` constructor, and implement value equality (`==` / `hashCode`) — plus a `copyWith` for evolving state. For deep immutability, hold unmodifiable collections (`List.unmodifiable` / `UnmodifiableListView`). Codegen (`freezed`) automates equality/`copyWith`/immutability. Immutability makes state predictable, enables cheap `const` reuse, and is why state-management libraries lean on it.

🔴 Advanced

30. An exception thrown inside a future's callback isn't caught by your surrounding `try/catch`. Why, and how do you catch it? `try/catch` only catches synchronous throws in the *current* execution turn. Once work is deferred to a future callback, it runs in a later microtask with a fresh stack, so a sync `try/catch` around the *scheduling* code never sees it. You must either `await`

the future inside the try (turning the async error into a catchable one), attach `.catchError`, or install a `runZonedGuarded` / `Zone` error handler. Uncaught async errors otherwise bubble to `Isolate.current.addErrorListener` / `PlatformDispatcher.onError` / `FlutterError.onError`.

31. What is a `Zone` and when do you actually need one? A zone is an execution context that can intercept uncaught errors, override `print`, and control how microtasks/timers are scheduled — async callbacks run in the zone that scheduled them. The main real-world use is `runZonedGuarded(() => runApp(...), (error, stack) => report(error, stack))` to funnel *all* uncaught async errors to Crashlytics/Sentry. Beyond error handling and test fakes (fake async, captured prints), you rarely author zones by hand — they're powerful but subtle.

32. `Future.wait` fails fast on the first error and you lose the other results. How do you get all outcomes? `Future.wait` completes with an error as soon as one future rejects; the others keep running but their results are dropped (and can cause unhandled-error warnings). To collect every outcome, wrap each future so it never rejects — e.g. `future.then((v) => Result.ok(v)).catchError((e) => Result.err(e))` — then `await Future.wait([...])` yields a list of results you can inspect individually. This is the manual equivalent of `Promise.allSettled`.

33. Design a debounced search that cancels stale in-flight requests. What operators? Debounce the query stream (`debounceTime(300ms)`), then use `switchMap`: when a new query arrives, `switchMap` cancels the previous inner request stream and subscribes to the new one — so a slow earlier request can never overwrite a newer result (the classic race). With plain `asyncMap` you'd preserve order but still await stale requests; with un-managed futures you'd get out-of-order results. rxdart's `switchMap` is the idiomatic answer.

34. `compute` shows overhead for small tasks — why, and when is an isolate the wrong tool? Spawning an isolate costs memory (a fresh heap) and startup time, and every message crosses the port by *copying* (serialization). For small or frequent tasks the copy + spawn cost dwarfs the work, so you regress performance. Isolates are wrong for I/O-bound work (that's what `async` already handles without a thread) and for anything needing tight shared state. Reach for them only for genuinely CPU-bound, coarse-grained chunks; reuse a long-lived isolate (or a pool) instead of `compute`-per-item.

35. Explain covariance in Dart generics and the hole it opens. Dart generics are *covariant*: `List<Cat>` is assignable to `List<Animal>`. That's convenient for reads but unsound for writes — you could add a `Dog` into a list statically typed `List<Animal>` that's actually a `List<Cat>`. Dart accepts this at compile time and inserts a *runtime* type check that throws on the bad `add`. So covariance trades a bit of static soundness for ergonomics, backed by runtime checks. (`covariant` keyword separately lets you narrow a parameter type in an override, also runtime-checked.)

36. Two mixins define the same method and both call `super`. What runs? Resolution follows the linearization order. `class C extends B with M1, M2` builds the chain `B → M1 → M2 → C`; a call resolves to the *rightmost* definition, and each `super.method()` walks one step left. So `M2`'s version

runs first and its `super` reaches `M1`'s, then `B`'s. This deterministic left-to-right composition (rightmost overrides) is exactly how Flutter stacks mixins like `TickerProviderStateMixin` predictably.

37. JIT vs AOT in Dart — what runs when, and why two modes? Debug builds use the JIT VM: source compiled just-in-time with a kernel snapshot, enabling *hot reload* and fast iteration but with warm-up cost. Release builds use AOT: Dart compiled ahead-of-time to native machine code, giving fast startup, predictable performance, and no runtime compiler — at the cost of no hot reload. You get developer velocity in debug and production performance in release from the same source.

38. What is tree shaking and what defeats it? Tree shaking is dead-code elimination during AOT compilation: code (functions, classes) provably unreachable from `main` is dropped, shrinking the binary. It relies on *static* reachability, so it's defeated by dynamic dispatch it can't resolve — notably reflection (`dart:mirrors`, which is why it's banned in Flutter) and `@pragma('vm:entry-point')` code invoked from native. Keep entry points static so the compiler can prune aggressively.

39. A `setState` fires "called after dispose" from a stream/future callback. Root cause and fixes. The async callback outlived the widget: the future/subscription completed after the `State` was disposed, and its captured closure called `setState` on a dead element. Fixes, in order of preference: cancel the subscription (and ideally the request) in `dispose`; guard with `if (!mounted) return;` before `setState`; or use `StreamBuilder` / `FutureBuilder` which own the subscription lifecycle. The guard is a band-aid — cancelling the work is the real fix (it also stops wasted network/CPU).

40. `scheduleMicrotask` in a loop can freeze your UI even though "async." Explain. The event loop fully drains the microtask queue before rendering the next frame or handling input. If each microtask schedules another, the queue never empties, so the loop never returns to the event queue — timers, gestures, and frame callbacks starve, and the app hangs despite no synchronous long loop. This is *microtask starvation*: for repeated deferred work, use the event queue (`Future(...)`, `Timer`) so the loop can breathe between items.

41. How does an `await` in a hot render path cause a dropped frame even without heavy compute? Even a trivial `await` defers the continuation to a microtask *after* the current turn, so any work after it can't complete within the same frame — and if that work rebuilds/relayouts, you've pushed it into a later frame. More importantly, code *before* the first `await` runs synchronously on the UI isolate; if that prelude (or a synchronous JSON decode you forgot to move to an isolate) is heavy, it blocks the 16ms budget directly. The fix is offloading CPU work to an isolate, not adding more `await`s.

⚡ Rapid-fire (one-liners)

Q	A
Microtask vs event queue priority?	Microtasks drain completely before each event-queue task.
Does an `async` function return before or after its first `await`?	Returns the future synchronously *at* the first await.
`Future.value(x).then(...)` runs on which queue?	Microtask.
One-shot vs many results?	`Future` vs `Stream`.
Default `StreamController` type?	Single-subscription.
How to allow many listeners?	`StreamController.broadcast()`.
Emit from a generator?	`async*` + `yield` (stream); `sync*` + `yield` (iterable).
Move CPU work off UI thread?	`compute()` / `Isolate.run`.
Isolates share memory?	No — message passing over ports.
`finally` for futures?	`.whenComplete()`.
Silence "unawaited future" intentionally?	`unawaited(f)`.
Catch only specific future errors?	`.catchError(fn, test: (e) => ...)`.
Funnel all uncaught async errors?	`runZonedGuarded`.
Constrain a generic type param?	`<T extends Bound>`.
Restrict a mixin to a supertype?	`mixin M on Base`.
Zero-cost wrapper type?	`extension type` (Dart 3.3+).
Hot reload works in which mode?	JIT (debug), not AOT (release).
What kills tree shaking?	Reflection / `dart:mirrors`.

Follow-up drills

1. **Predict & justify:** given a snippet mixing `print`, `Future`, `Future.microtask`, `scheduleMicrotask`, and a nested future, write the exact output and explain each queue transition.
2. **Design:** a rate-limited, cancellable download manager exposing a `Stream<Progress>` — decide single vs broadcast, backpressure strategy, and cleanup.
3. **Optimize:** the app janks while parsing a 12MB JSON on startup. Diagnose with the event-loop model and refactor using isolates without regressing small payloads.
4. **Debug:** intermittent "setState called after dispose" from a search screen that fires overlapping requests — identify the two distinct bugs (lifecycle + stale response) and fix both.
5. **Whiteboard:** implement `allSettled` semantics on top of `Future.wait`, then extend it with a per-future timeout and retry-with-backoff.

6. **Reason:** explain, using linearization, what `super.build` / `super`-chained mixins resolve to for a `State` using `WidgetsBindingObserver` + `TickerProviderStateMixin`.

OOP, SOLID & Design Patterns (in Dart) — Interview Questions

> Object-oriented modeling, the SOLID principles, and the design patterns you actually meet in Flutter code. Depth: 03 Object Oriented Programming, 04 SOLID Principles, 05 Design Patterns.

This topic tests whether you can structure non-trivial code so it stays testable and changeable. Interviewers use it to separate people who memorized pattern names from people who know *which* problem each pattern removes — and when applying one is over-engineering.

🟢 Basic

1. What are the four pillars of OOP, in one line each?

- **Encapsulation** — bundle state with the behavior that guards it, and hide the internals (`_private` in Dart).
- **Inheritance** — a subtype reuses and specializes a supertype (`class Dog extends Animal`).
- **Polymorphism** — one interface, many runtime implementations; the correct override is dispatched dynamically.
- **Abstraction** — expose *what* an object does, hide *how*; model the essential and drop the rest.

2. How does encapsulation work in Dart specifically? Dart has no `private` / `public` / `protected` keywords. Privacy is **library-level**: an identifier prefixed with `_` is visible only within its own `.dart` library, not just the class. So two classes in the same file can touch each other's `_fields`. You expose controlled access through getters/setters instead of public fields, which lets you add validation or computed values later without changing the call site.

3. What is the difference between a class, an abstract class, and an interface in Dart? A **concrete class** can be instantiated. An **abstract class** (`abstract class`) can declare methods without bodies and cannot be instantiated directly — it's meant to be extended or implemented. Dart has **no separate `interface` keyword**: *every class implicitly defines an interface*. When you `implements SomeClass`, you adopt its interface (all members) but inherit none of its implementation and must re-supply everything.

4. `extends` vs `implements` vs `with` — when do you use each?

Keyword	Meaning	Inherits implementation?	Count
`extends`	Subclass a class	Yes	One only
`implements`	Adopt a class's interface as a contract	No — you re-implement all members	Many
`with`	Mix in reusable behavior	Yes (from the mixin)	Many

Use `extends` for a true is-a with shared code, `implements` for a pure contract or to fake/mock a type in tests, `with` (mixins) to share behavior across unrelated hierarchies.

5. What is polymorphism in practice, with a Dart example? Calling a method on a supertype reference and having the subtype's version run at runtime.

```
abstract class Shape { double area(); }
class Circle implements Shape { final double r; Circle(this.r); double area() => 3.14 * r * r; }
class Square implements Shape { final double s; Square(this.s); double area() => s * s; }

double total(List<Shape> shapes) => shapes.fold(0, (sum, s) => sum + s.area());
// total([Circle(1), Square(2)]) works without total() knowing the concrete types
```

Adding a `Triangle` later requires zero changes to `total()`.

6. What is composition, and why "composition over inheritance"? Composition means an object *holds* other objects and delegates to them (has-a) instead of inheriting from them (is-a). It's preferred because inheritance is rigid: it couples you to the parent's implementation, allows only one superclass, and deep hierarchies become fragile (a change up top ripples down). Composition lets you swap collaborators, mix capabilities freely, and test each piece in isolation. Flutter itself is built on composition — you *nest* widgets, you don't subclass a mega-widget.

7. Why is `abstract class` used so much in Flutter/Dart architecture? To define contracts. A repository or service is declared as an abstract class; the concrete implementation lives elsewhere and is injected. Code depends on the abstraction, so you can swap a real HTTP implementation for a fake in tests without touching callers. It's the enabler for the D in SOLID.

8. What does SOLID stand for?

- **S** — Single Responsibility Principle
- **O** — Open/Closed Principle
- **L** — Liskov Substitution Principle
- **I** — Interface Segregation Principle
- **D** — Dependency Inversion Principle

They're guidelines for managing coupling and change, not laws — see 04 SOLID Principles.

9. Explain the Single Responsibility Principle with the smell it fixes. A class should have one reason to change. The smell is a **god class** that does networking, parsing, caching, and UI formatting — a bug fix in parsing risks breaking caching, and the class is impossible to test in isolation. Split it: a `UserApi` (fetch), a `UserMapper` (parse), a `UserCache` (store). Each changes for one reason.

10. What is a design pattern, and why not just write ad-hoc code? A design pattern is a named, reusable solution to a recurring design problem. The value is twofold: it's a **vocabulary** ("this is a Repository behind a Factory") that compresses communication, and it encodes a structure that's already been proven to keep coupling low. The risk is cargo-culting — applying a pattern where a plain function would do adds indirection for no benefit.

11. What is the Singleton pattern and how do you write one in Dart? A class with exactly one shared instance. The idiomatic Dart form uses a factory constructor returning a cached static instance:

```
class ApiClient {
  ApiClient._internal();                    // private named ctor
  static final ApiClient _instance = ApiClient._internal();
  factory ApiClient() => _instance;         // every ApiClient() returns the same object
}
```

Because top-level and `static final` fields in Dart are **lazily initialized** on first access, `static final ApiClient instance = ApiClient._();` is an even simpler idiom.

12. What is a factory constructor and how does it differ from a generative one? A generative constructor always creates a *new* instance and initializes fields. A `factory` constructor doesn't have to — it can return a cached instance, a subtype, or an instance built from a computed value. That's why it's used for singletons, caches, and `fromJson` deserialization where the concrete type depends on the data.

🟡 Intermediate

13. Abstract class vs implicit interface — when do you `extends` vs `implements` the same abstract class? `extends` when the abstract class provides shared implementation you want to inherit and you only override the abstract bits. `implements` when you want *only the contract* and will supply every member yourself — common in tests to build a fake, or when a class already extends something else (single-inheritance forces you to `implements`). Note: `implements` on an abstract class with concrete methods forces you to re-implement even those methods.

14. How do you correctly implement value equality in Dart? Override **both** `==` and `hashCode` — they must agree (equal objects must have equal hashCodes), or the object misbehaves in `Set` / `Map`.

```
class Point {
  final int x, y;
  const Point(this.x, this.y);
  @override bool operator ==(Object other) =>
      other is Point && other.x == x && other.y == y;
  @override int get hashCode => Object.hash(x, y);
}
```

Use `Object.hash(...)` / `Object.hashAll(...)` rather than hand-rolling. In practice, prefer generating this with `freezed` / `equatable` to avoid mistakes. See 07 Equality and Copying.

15. Why must equal objects have equal hashCodes? Hash-based collections (`HashSet` , `HashMap`) first bucket an object by `hashCode` , then compare with `==` only within that bucket. If two `==` objects have different hashCodes they land in different buckets, so a lookup for one won't find the other — you get "I just added it but `contains` says false" bugs. The reverse isn't required: unequal objects *may* share a hashCode (a collision), that's fine.

16. What is `copyWith` and why is it central to immutable state? `copyWith` returns a **new** instance with some fields replaced and the rest copied — the idiomatic way to "mutate" an immutable object.

```
User copyWith({String? name, int? age}) =>
    User(name: name ?? this.name, age: age ?? this.age);
```

It's central to state management (BLoC/Riverpod/ `setState`) because immutable state means change detection is a cheap identity/ `==` check, and old snapshots stay valid (undo, time-travel). Caveat: `name ?? this.name` can't distinguish "leave unchanged" from "set to null" — `freezed` solves this with sentinel values.

17. Deep copy vs shallow copy — what breaks if you get it wrong? A **shallow copy** duplicates the top-level object but shares references to nested objects. A **deep copy** recursively clones nested objects too. If a `copyWith` shallow-copies a `List` field (`this.items`) and a caller mutates that list, *both* the old and new "immutable" objects change — silently corrupting state and breaking equality-based rebuilds. Fix: copy the collection (`List.of(items)`) or, better, use genuinely immutable collections.

18. Explain the Open/Closed Principle with a Dart example. Open for extension, closed for modification: add behavior by adding code, not editing tested code. The smell is a growing `switch` :

```
// Violates OCP — every new type edits this method
double price(String type) { switch (type) { case 'a': ...; case 'b': ...; } }
```

Fix with polymorphism — each type implements a `Priceable` interface, and new types are new classes. The pricing engine never changes. See 02 Open/Closed.

19. Explain the Liskov Substitution Principle and a classic violation. A subtype must be usable anywhere its supertype is expected without breaking correctness. Classic violation: `Square extends Rectangle` overriding `setWidth` to also change height — code that sets width and height independently (valid for a Rectangle) now breaks. Also violated when an override **strengthens preconditions** (rejects inputs the parent accepted) or throws `UnsupportedError` for an inherited method. If you find yourself type-checking `if (x is Square)` to work around a subtype, LSP is broken.

20. Explain Interface Segregation with a Dart smell. Don't force a class to implement methods it doesn't use. A fat `Worker` interface with `work()` and `eat()` forces a `RobotWorker` to implement a meaningless `eat()`. Split into `Workable` and `Feedable`; a robot implements only `Workable`. In Dart this maps to keeping abstract classes small and role-focused rather than one giant service interface. See 04 Interface Segregation.

21. Explain the Dependency Inversion Principle — and how it differs from dependency injection. DIP (the principle): high-level modules and low-level modules should both depend on **abstractions**, not on each other. Your `LoginBloc` depends on an `AuthRepository` interface, not on `FirebaseAuth` directly. **Dependency Injection** (the technique/pattern) is *how* you satisfy DIP — you pass the concrete implementation in from outside (constructor, `get_it`, Provider) instead of the class `new`-ing it internally. DIP is the goal; DI is a means. See 05 Dependency Inversion and 14 Dependency Injection.

22. What is the Strategy pattern and where does it show up in Flutter? Encapsulate interchangeable algorithms behind a common interface and pick one at runtime.

```
abstract class SortStrategy { List<int> sort(List<int> data); }
class QuickSort implements SortStrategy { ... }
class Sorter { SortStrategy strategy; Sorter(this.strategy); ... }
```

It's OCP in action. In Flutter you see it as pluggable validators, sort/filter selectors, or a `ScrollPhysics` you swap on a `ListView`.

23. What is the Factory pattern, and how does Flutter/Dart use it? A factory centralizes *object creation* so callers don't hardcode concrete types. Dart's `factory` constructor is the language-level form; `Model.fromJson` is the everyday example — it decides what to build from the data. A **Factory Method** returns a subtype based on input (`Notification.from(type)` returning `EmailNotification` or `PushNotification`). See 01 Factory.

24. What is the Builder pattern and where is it in Flutter? Construct a complex object step by step, separating construction from representation. Flutter's `...builder` callbacks (`ListView.builder`, `StreamBuilder`, `LayoutBuilder`) are a variant — they defer creating widgets until context/index is

known. The classic Builder (a chainable `QueryBuilder().where().limit().build()`) shows up in query/HTTP request builders. See 02 Builder.

25. What is the Repository pattern and why is it standard in Flutter apps? A Repository is an abstraction over data sources that gives the domain layer a clean, source-agnostic API (`getUser(id)`), hiding whether data comes from REST, cache, or a local DB. It's standard because it (a) satisfies DIP — BLoCs depend on `UserRepository`, not Dio; (b) makes tests trivial with a fake repo; and (c) localizes the "network-first vs cache-first" decision in one place. See 40 Clean Architecture.

🔴 Advanced

26. Walk through the Observer pattern and where Flutter already implements it. A subject maintains a list of observers and notifies them on state change. Flutter is *saturated* with it: `ChangeNotifier` / `notifyListeners()` + `AnimatedBuilder` / `ListenableBuilder`, `ValueNotifier` / `ValueListenableBuilder`, and `Stream` / `StreamBuilder` are all Observer. `setState` is a scoped form. When you use Provider or `flutter_bloc`, the widget subscribes as an observer and rebuilds on notification. Knowing this reframes state management as "who observes what, and how narrowly."

27. What is the Decorator pattern and how does it differ from inheritance? Decorator wraps an object to add behavior *at runtime* while keeping the same interface — composition, not a compile-time subclass. Flutter's widget tree is decoration by nesting: `Padding(child: DecoratedBox(child: ...))` each adds one responsibility around the child. The advantage over inheritance is combinatorial: you compose `Padding + Opacity + GestureDetector` in any order instead of needing a `PaddedOpaqueTappableWidget` subclass for every combination. See 06 Decorator.

28. What is the Adapter pattern and when do you reach for it? Adapter converts one interface into another so incompatible classes can collaborate — you reach for it at boundaries you don't control. Example: a third-party analytics SDK exposes `logEvent(Map)`, but your app defines an `AnalyticsService.track(Event)` interface; a `FirebaseAnalyticsAdapter implements AnalyticsService` translates between them. This keeps the vendor SDK out of your domain, so swapping vendors touches only the adapter. See 05 Adapter.

29. Contrast Adapter, Decorator, and Proxy — they all wrap an object. All three hold a reference to a wrapped object, but their *intent* differs:

Pattern	Interface vs wrapped	Intent
Adapter	Different	Make an incompatible interface fit
Decorator	Same	Add responsibilities without changing the type
Proxy	Same	Control access — lazy-load, cache, authorize, log

Same mechanics, three different problems. Naming the intent is what interviewers listen for.

30. Give a concrete Dart example of the Dependency Injection pattern with a testable seam.

```
abstract class Clock { DateTime now(); }
class SystemClock implements Clock { DateTime now() => DateTime.now(); }


class SessionService {
  final Clock _clock;
  SessionService(this._clock);            // injected — the seam
  bool isExpired(DateTime issued) => _clock.now().difference(issued).inMinutes > 30;
}
// prod:  SessionService(SystemClock());
// test:  SessionService(FakeClock(fixed: someInstant));  // deterministic
```

The injected `Clock` turns an untestable `DateTime.now()` into a controllable dependency. `get_it` / `injectable` and Provider automate wiring this at scale — see 14 Dependency Injection.

31. Is Singleton an anti-pattern? Defend your answer. It's a *smell* more than an outright anti-pattern. Problems: it's global mutable state (hard to reason about), it hides dependencies (a class secretly uses `ApiClient()` instead of declaring it), and it's hard to reset/mock between tests. The fix isn't "never one instance" — it's registering a single instance in a DI container (`get_it.registerSingleton`) and *injecting* it, so lifetime is controlled and tests can override it. So: one instance = fine; global static access to it = the smell.

32. What is a god class / god object and how do you refactor one? A class that accumulates too many responsibilities and knows too much — hundreds of lines, dozens of fields, imports from every layer. It violates SRP, is a merge-conflict magnet, and can't be unit-tested. Refactor by (1) grouping methods/fields that change together, (2) extracting each group into a focused collaborator, (3) having the original delegate, then (4) inverting dependencies so it holds abstractions. In Flutter this is the 2,000-line `StatefulWidget` that does fetching, business logic, and layout — split into repository, controller/BLoC, and dumb widgets.

33. What is tight coupling, how do you detect it, and how do you loosen it? Tight coupling is when a change in one class forces changes in another. Detection signals: `new`/direct construction of concrete collaborators inside a class, reaching through object chains (`a.b.c.d` — Law of Demeter violation), and widespread `import` of a concrete type. Loosen it by depending on abstractions (DIP), injecting collaborators, and using events/streams so producer and consumer don't reference each other. Measure of success: you can unit-test the class with fakes and no real network/DB.

34. How do abstract classes, `sealed`, and mixins differ for modeling a closed hierarchy?

- `abstract class` — a contract/base; open for anyone to extend or implement.

- `sealed class` (Dart 3) — abstract *and* the subtypes are known at compile time within the library, so `switch` over them is **exhaustively checked** — the compiler errors if you miss a case. Ideal for state unions and the modern alternative to a `Result` / `Either` hierarchy.
- `mixin` — reusable behavior grafted onto classes via `with`, without an is-a relationship.

Reach for `sealed` when you want the compiler to enforce that every case is handled — it makes OCP-vs-exhaustiveness a deliberate choice rather than an accident.

35. When is applying a design pattern the wrong call? When it adds indirection that outlives no real variation point. A Strategy with exactly one implementation, a Factory that just calls one constructor, or an interface with a single forever-implementation are speculative generality — they cost readability now for flexibility you may never need (YAGNI). Introduce the pattern at the moment a *second* case appears, guided by the smell it removes, not preemptively. Senior signal: you can name both the pattern *and* the condition under which it's premature.

36. How does immutability interact with equality and Flutter rebuild performance? Immutable objects with value equality let Flutter skip work: `const` widgets are canonicalized (one instance reused), and equality-based diffing (BLoC's `buildWhen`, Riverpod `select`, `Equatable`) rebuilds only when a value actually changed rather than on every new-but-equal instance. But it only holds if equality is correct and copies are deep enough — a shared mutable list inside a "value" object defeats both the `==` check and `const`-ness, causing missed or excess rebuilds. This is why `freezed` (immutable + generated `==` / `hashCode` / `copyWith`) is the default for state classes. See 21 Performance.

37. Design a payment system that's open to new providers but closed to modification — which principles and patterns apply?

- **DIP + Strategy**: define `abstract class PaymentGateway { Future<Result> pay(Amount a); }`; each provider (`StripeGateway`, `RazorpayGateway`) is a strategy implementing it.
- **Factory**: a `PaymentGatewayFactory.forMethod(method)` selects the implementation, so callers stay decoupled from concretes.
- **OCP**: adding PayPal = a new class + one factory registration; the checkout flow never changes.
- **Adapter**: each gateway class adapts the vendor SDK to your `PaymentGateway` interface.
- **DI**: the factory/gateway is injected so tests use a `FakeGateway`.

That's four patterns collaborating, each removing a specific coupling — and it's the kind of layered answer senior interviews reward.

⚡ Rapid-fire (one-liners)

Question	Answer
Dart's access modifier?	`_` prefix = library-private; no `public` / `protected` keywords
Is there an `interface` keyword?	No — every class defines an implicit interface
`extends` count limit?	One superclass; `implements` / `with` allow many
Override `==`, must also override...?	`hashCode` (they must agree)
Idiomatic hashCode combine?	`Object.hash(a, b)` / `Object.hashAll(list)`
Why `copyWith`?	Immutable "mutation" — new instance, rest copied
Shallow copy risk?	Shared nested references mutate both objects
S in SOLID?	Single Responsibility — one reason to change
OCP mechanism in Dart?	Polymorphism replaces a growing `switch`
LSP red flag?	Overrides that throw or need `is` type-checks
DIP vs DI?	DIP = depend on abstractions (goal); DI = pass them in (means)
Dart singleton idiom?	`factory` returning a `static final` instance
`fromJson` is which pattern?	Factory
`ListView.builder` is which pattern?	Builder (lazy construction)
`ChangeNotifier` / `Stream` is which pattern?	Observer
Nested widgets are which pattern?	Decorator (composition)
Repository solves?	Source-agnostic data access + DIP + testability
God class fix?	Extract responsibilities, delegate, invert deps
`sealed class` benefit?	Exhaustive `switch` checked at compile time

Follow-up drills

1. **Design** a notification system supporting email, SMS, and push that a new channel can join without editing existing code — name the patterns and principles at each seam.
2. **Refactor** a 1,500-line `StatefulWidget` that fetches data, holds business rules, and builds UI into a testable structure; describe the extraction order and the seams you introduce.
3. **Debug this scenario:** a value object is added to a `Set`, but `set.contains(equalObject)` returns `false`. Diagnose the root cause and the exact fix.
4. **Decide:** you have one implementation of a service and no second one in sight — justify whether to introduce an interface + DI now or later, using YAGNI vs testability.
5. **Contrast** using `sealed class` + exhaustive `switch` versus polymorphic dispatch for modeling a set of app states — trade-offs of each for adding new states.

6. **Design** a caching layer in front of a remote API without changing any caller — pick between Proxy and Decorator and defend the choice.

Flutter Internals & Architecture — Interview Questions

> How Flutter actually runs: the three trees, BuildContext, keys, the layered architecture, and the Dart/hot-reload machinery. Depth in 06 Flutter Fundamentals and 10 Flutter Architecture.

This topic separates people who *use* Flutter from people who *understand* it. Interviewers probe here to see if you can reason about rebuilds, performance, and weird bugs (lost state, wrong list item) from first principles instead of guessing.

🟢 Basic

1. What is Flutter? Flutter is an open-source UI toolkit from Google for building natively-compiled apps for mobile, web, desktop, and embedded from a single Dart codebase. The key idea: Flutter does **not** use the platform's native (OEM) widgets. It ships its own rendering engine and draws every pixel itself onto a canvas, so a Flutter app looks and behaves identically across platforms unless you deliberately branch.

2. How does Flutter differ from React Native? React Native maintains a JS thread that talks over a *bridge* to real native OEM widgets — your `<View>` becomes a `UIView` / `android.view.View`. Flutter has no bridge for UI and no OEM widgets; it compiles Dart to native code and renders its own widgets directly via Skia/Impeller. Result: fewer thread-hops, no "bridge serialization" bottleneck, and pixel-identical UI, at the cost of not automatically inheriting native look-and-feel.

	Flutter	React Native
UI widgets	Own, self-rendered	Real native OEM widgets
Bridge	No UI bridge	JS↔native bridge (JSI newer)
Language	Dart (AOT to native)	JS (JIT/Hermes)
Consistency	Pixel-identical everywhere	Varies with OS version

3. If Flutter doesn't use native widgets, how does anything appear on screen? Flutter's engine owns a texture/canvas provided by the platform (via the embedder). The framework builds a description of the frame, the engine rasterizes it with Skia (or Impeller) using the GPU, and hands the finished pixels to the OS to composite. A `Text` widget is not an OEM label — it's Flutter drawing glyphs onto that canvas.

4. What are the three trees in Flutter?

- **Widget tree** — immutable configuration/blueprint you write in `build`.
- **Element tree** — the mutable runtime instances that hold state and position in the tree; the "glue" between widgets and render objects.
- **RenderObject tree** — the objects that do layout, painting, and hit-testing (the actual pixels).

Every widget of consequence has a corresponding element, and render-producing widgets also have a render object.

5. Why is a Widget described as "immutable configuration"? All a `Widget`'s fields are `final`; it's a lightweight, throwaway description of "what the UI should look like given this config." Flutter recreates widgets constantly (every `build`) because they're cheap. They hold no long-lived state and never mutate — mutation lives in `Element` / `State`.

6. Then where does mutable state live if widgets are immutable? In the **Element** (and for `StatefulWidget`, the `State` object the element holds). Widgets get thrown away and rebuilt; the element persists across rebuilds as long as it can be matched to a new widget, and it carries the `State`. That's why your counter value survives a `setState` even though a brand-new `MyWidget()` was created.

7. What is `BuildContext`, really? `BuildContext` **is the Element**. `BuildContext` is just an interface that `Element` implements, handed to you in `build(BuildContext context)`. So "context" is your widget's location in the element tree — which is why you use it to walk the tree (`Theme.of(context)`, `Navigator.of(context)`) to find ancestors/inherited data.

8. What does `setState` actually do? `setState` runs your callback (so you mutate state synchronously) and then marks the element **dirty** via `markNeedsBuild`. On the next frame, the framework rebuilds that element by calling `build` again and reconciles the new widget subtree against the old element tree. `setState` doesn't repaint immediately and doesn't rebuild the whole app — only the dirty element's subtree.

9. What's the difference between `StatelessWidget` and `StatefulWidget` in this model? A `StatelessWidget` creates a `StatelessElement` and has nothing to persist. A `StatefulWidget` creates a `StatefulElement` that holds a `State` object via `createState()`; that `State` survives rebuilds and can call `setState`. Choose stateful only when there's mutable data that must outlive a rebuild.

10. What is a Key at a basic level? A `Key` is an identity tag on a widget that helps Flutter decide, during reconciliation, whether a new widget should reuse an existing element (and its state) or get a fresh one. Without keys, Flutter matches by **position + runtimeType**. Keys let you match by **identity** instead.

11. What language does Flutter use and why Dart? Dart. It's chosen because it supports both **JIT** (fast compile → stateful hot reload in dev) and **AOT** (compiled to native ARM/x64 for fast release), has a mature isolate-based concurrency model, sound null safety, and a UI-friendly, non-blocking single-threaded-per-isolate event loop that suits a declarative framework.

12. What is hot reload? Hot reload injects updated Dart source into the running Dart VM (Just-In-Time), rebuilds the widget tree, and repaints — **without restarting the app**, so your current state (logged-in user, scroll position) is preserved. It's the dev-loop feature that makes Flutter iteration fast.

🟡 Intermediate

13. Why does Flutter need three trees instead of one? Separation of concerns and performance. Widgets are cheap, immutable descriptions you regenerate freely. RenderObjects are expensive (they own layout/paint state) so you want to *keep* them and only mutate what changed. The Element tree is the diffing layer in between: it persists, maps widgets↔render objects, and lets Flutter update the minimum by comparing new widgets to old elements. One tree can't be both cheap-and-disposable and expensive-and-persistent.

14. Walk through what happens on the first build vs a rebuild.

- **First build:** framework calls `build`, gets widgets, then for each widget calls `createElement()` → mounts elements → elements call `createRenderObject()` → render tree built → layout → paint.
- **Rebuild:** `build` produces new widgets; the framework calls `updateChild` on each element. If the new widget's `runtimeType` **and** `key` match the old one, the element is *reused* — it just gets the new config via `update()` and the render object is mutated in place. If they don't match, the old element/subtree is unmounted and a new one is created.

15. What is the exact matching rule during reconciliation? For each child position, Flutter compares the new widget to the element's current widget. It reuses the element **iff** `Widget.canUpdate(old, new)` is true, which means `old.runtimeType == new.runtimeType && old.key == new.key`. Otherwise it deactivates the old element and inflates a new one. This single rule explains almost every "state jumped to the wrong item" bug.

16. When do you actually need a Key? When you're **reordering, inserting, or removing** stateful widgets in a collection of the **same type**, and state must follow the logical item rather than the position. Common triggers: reorderable/dismissible lists, swapping two stateful widgets, animated lists. If widgets are stateless, or the list is static, keys usually aren't needed.

17. Explain the classic "keys" bug with a list of stateful items. Say you have a `Column` / `ListView` of stateful `ColorTile`s and you remove the first one. Because Flutter matches by position + type, element[0] (which held the *first* tile's state) is now matched to the *second* tile's widget — so the second tile inherits the first's state (color, controller, animation). Visually the wrong item keeps the state. Adding a `ValueKey(item.id)` makes Flutter match by identity, so each element follows its logical item.

18. LocalKey vs GlobalKey — what's the difference?

- **LocalKey** — unique only among siblings; used for reconciliation within one parent (`ValueKey`, `ObjectKey`, `UniqueKey`).
- **GlobalKey** — unique across the *entire* app; lets you reference an element/state from anywhere (`myKey.currentState`, `currentContext`), and it lets a subtree's element (and state) be *moved* to a different parent without being rebuilt.

GlobalKeys are heavier (a global registry, must be app-unique) — use them sparingly.

19. ValueKey vs ObjectKey vs UniqueKey — when each?

- `ValueKey(v)` — identity by a **value** with proper `==` (an id, a string). Most common.
- `ObjectKey(obj)` — identity by the object's `identical()` reference; use when two objects can be equal by value but are logically distinct.
- `UniqueKey()` — never equal to anything, even itself across rebuilds; forces a *fresh* element every build (useful to force a reset), but defeats reuse so don't put it in a `build` you rebuild often.

20. Name three legitimate uses of GlobalKey.

1. `GlobalKey<FormState>` to call `formKey.currentState!.validate()`.
2. Reading a widget's size/position via `currentContext.findRenderObject()`.
3. Preserving a stateful subtree while moving it to a different parent (e.g., a video that keeps playing when reparented across layouts).

21. What are the three architectural layers of Flutter?

- **Framework** (Dart) — Material/Cupertino, Widgets, Rendering, Animation, Painting, foundation. What you write against.
- **Engine** (C/C++) — Skia/Impeller rendering, Dart runtime, text layout, the compositor; exposes `dart:ui`.
- **Embedder** (platform-specific) — the native shell that gets the app running on Android/iOS/web/desktop: surface setup, event loop, input, packaging as a native app.

Your Dart calls flow down: Framework → `dart:ui` → Engine → Embedder → OS.

22. What is `dart:ui`? It's the lowest Dart-level library, the thin binding that exposes engine primitives to the framework — `Canvas`, `Picture`, `Scene`, `Paragraph`, images, and the hooks that let Flutter schedule and submit frames. The whole widget/rendering framework is built on top of `dart:ui`.

23. Skia vs Impeller — what changed and why? Skia is the long-standing 2D graphics library Flutter used; it compiles shaders **at runtime** on first use, which caused "shader compilation jank" (first-run stutter). Impeller is Flutter's newer rendering engine that **precompiles** shaders at build/engine time and is designed around modern GPU APIs (Metal on iOS, Vulkan on Android), eliminating that jank and giving more predictable frame times. Impeller is the default on iOS and Android in current stable Flutter.

	Skia	Impeller
Shaders	Compiled at runtime	Precompiled
Main pain solved	—	Shader-jank on first run
Backends	GL/Metal/Vulkan	Metal (iOS), Vulkan (Android)
Status	Legacy default / fallback	Current default

24. JIT vs AOT in Flutter — when is each used?

- **Debug builds** use the Dart VM's **JIT** — code compiled on the fly, which enables hot reload and assertions. Larger, slower, dev-only.
- **Release/profile builds** use **AOT** — Dart compiled ahead-of-time to native ARM/x64 machine code. Fast startup, no runtime compilation, small and optimized. No hot reload.

25. Hot reload vs hot restart — compare.

	Hot reload	Hot restart
What it does	Injects changed code into running VM, rebuilds widget tree	Kills the Dart isolate and restarts app from `main()`
State	**Preserved**	**Lost** (reset to initial)
Speed	~sub-second	A few seconds
Handles	UI/ `build` changes	Changes to `main`, global/ `static` inits, enums, some class shape changes
Doesn't handle	`initState`/global initializers won't re-run; some structural changes	—

26. Why doesn't `initState` re-run on hot reload? Hot reload preserves the existing `State` objects and element tree; it only re-runs `build`. `initState` runs once when a `State` is *created*, and no new state is created on reload, so it's skipped. If your change lives in `initState`, you need a **hot restart** (or trigger a rebuild that recreates the widget, e.g. via a key change).

🔴 Advanced

27. Describe the frame pipeline from `setState` to pixels. `setState` → `markNeedsBuild` flags the element dirty and schedules a frame with the `SchedulerBinding`. On the next vsync the engine calls back and `WidgetsBinding.drawFrame` runs the phases in order:

1. **Build** — rebuild dirty elements (`buildScope`).
2. **Layout** — dirty render objects run `performLayout` (constraints down, sizes up).
3. **Paint** — produce a layer tree via `paint` / `Canvas`.
4. **Composite** — layers assembled into a `Scene`.
5. **Rasterize** — engine (Skia/Impeller) turns the scene into pixels on the GPU raster thread.

Only *dirty* nodes at each stage are reprocessed; that's the source of Flutter's efficiency.

28. "Constraints go down, sizes go up, parent sets position." Explain. Flutter layout is a single-pass box protocol. A parent passes `BoxConstraints` (min/max width/height) down to each child. Each child sizes itself **within** those constraints and returns its chosen `Size` up. The parent

then **positions** each child (sets its `offset`). A widget can't know its own size until its parent constrains it — which is why `MediaQuery` / `LayoutBuilder` exist and why "unbounded height" errors happen (parent gave infinite constraints, child couldn't pick a size).

29. How does an `InheritedWidget` propagate changes efficiently, and how does `context` participate? When an element calls `dependOnInheritedWidgetOfExactType` (what `.of(context)` does), the framework registers that element as a **dependent** of the nearest matching `InheritedElement` — walking up via the element (context) tree and caching the lookup (O(1) afterward). When the `InheritedWidget` is rebuilt with `updateShouldNotify` returning true, only the registered dependents are marked dirty — not the whole subtree. This is the mechanism behind Provider/Theme/MediaQuery.

30. Why can calling `context.of()` in `initState` fail or be unsafe? `initState` runs before the element is fully wired into the tree for dependency purposes; more importantly, dependencies established there won't be re-triggered correctly. Inherited lookups that create dependencies belong in `didChangeDependencies` or `build`. A one-off read is fine with `context.read` (Provider) or a `listen: false` lookup, but registering a dependency in `initState` is the classic mistake.

31. What is an `Element`'s lifecycle (states)? `initial` → **mounted/active** (after `mount`, it's in the tree and can build) → **inactive** (removed from tree but might be reinserted this frame, e.g. via GlobalKey move) → **defunct** (permanently unmounted, `unmount` called, can never be reused). Understanding "inactive" explains how a GlobalKey subtree can be moved: the element is deactivated then reactivated under a new parent instead of being destroyed.

32. Why are RenderObjects expensive, and how does `RepaintBoundary` exploit the tree design? RenderObjects cache layout results, paint state, and layer info; recreating them means redoing layout/paint. The element tree lets Flutter mutate a render object in place instead. `RepaintBoundary` inserts a boundary that gives its subtree its **own layer**, so when that subtree repaints (e.g. an animation) it doesn't force the parent/siblings to repaint — isolating raster work. It's a direct lever on the paint/composite phases.

33. How does hot reload work internally, and what are its hard limits? The tooling computes changed libraries, compiles them to kernel (`.dill`) deltas, and sends them to the running VM's isolate, which **swaps method bodies** of existing classes in place. It then invalidates the widget tree and reassembles (`reassemble`) so `build` re-runs. Limits stem from "swap bodies, keep objects": it can't rerun `main`/global initializers, can't reshape live objects for changes to field layout, enum values, generic bounds, or `const` evaluation — those need a hot **restart** to recreate objects from scratch.

34. A widget's state resets unexpectedly when you wrap/unwrap it in another widget. Why, and the fix? Wrapping changes the widget's **position and/or the runtimeType of an ancestor**, so at that position `canUpdate` returns false, the old element is discarded, and a new one (with fresh

`State`) is created. The fix is a `GlobalKey` on the stateful widget: the element is matched by global identity and *moved* to the new location (deactivated → reactivated) instead of rebuilt, preserving state.

35. Two `StatefulWidget`s of the same type swap places in a `Row` and their state follows the wrong one. Diagnose. Same root cause as the list bug: position-based matching. Element[0] holds widget-A's state; after the swap, position 0 now hosts widget-B's widget, but element[0] (and its `State`) is reused because type matches. State is bound to *position*, not *logical widget*. Add distinct `ValueKey`s so `canUpdate` fails on the mismatched key and elements are re-paired to the correct logical widgets.

36. Why is Flutter's architecture called "layered" and why does that matter to a senior engineer? Each layer is built purely on the public API of the layer below, with no back-references, so you can **drop down** a level when you need to: build custom render objects (bypass Material), draw directly with `Canvas` / `CustomPainter`, or even swap the top layer entirely (Flutter's own widgets are not privileged). This composability — no monolithic "core" you must go through — is why Flutter is extensible and why "there's no magic, just widgets" holds.

37. On which threads does Flutter run, and why does that matter for jank? The engine uses several runners: the **UI/platform (Dart) thread** runs your Dart, build, layout, paint; the **raster thread** (formerly GPU) executes the layer tree with Skia/Impeller; plus IO and platform threads. Jank is either a **build/layout** overrun (fix on the UI thread: fewer rebuilds, `const`, RepaintBoundary) or a **raster** overrun (expensive shaders/large layers: Impeller, simpler effects). Diagnosing jank starts with *which thread* blew the 16ms (at 60Hz) budget in the timeline.

38. How does Flutter achieve identical UI across platforms, and what's the trade-off? Because it renders everything itself onto a canvas via its own engine, the output doesn't depend on OEM widget implementations or OS versions — a button looks the same on Android 8 and iOS 17. Trade-offs: (1) you must *opt in* to platform conventions (Cupertino vs Material, `.adaptive` constructors) since you don't inherit them; (2) larger binary (ships the engine); (3) accessibility and text-editing behaviors must be re-bridged by the framework rather than inherited from the OS.

⚡ Rapid-fire (one-liners)

Q	A
What is `BuildContext`?	The `Element` — your widget's spot in the element tree.
Which tree holds state?	Element (and its `State`).
Which tree does layout/paint?	RenderObject.
Widget mutability?	Immutable (`final` fields).
`canUpdate` rule?	Same `runtimeType` **and** same `key`.
Default matching without keys?	Position + runtimeType.
GlobalKey superpower?	App-wide identity; move a subtree keeping its state.
Key to force a full reset?	`UniqueKey()`.
Debug build compiler?	Dart JIT.
Release build compiler?	Dart AOT (native).
Hot reload preserves state?	Yes. Hot restart? No.
Why `initState` skipped on reload?	State isn't recreated; only `build` reruns.
Rendering engine (current)?	Impeller (Skia legacy/fallback).
Impeller solves?	Shader-compilation jank.
Three architecture layers?	Framework, Engine, Embedder.
Layout protocol?	Constraints down, sizes up, parent sets position.
What isolates a repaint?	`RepaintBoundary` (own layer).
Does Flutter use OEM widgets?	No — draws its own pixels.

Follow-up drills

1. **Design:** Sketch how you'd implement a reorderable list of stateful cards that keep their animation state while being dragged — specify exactly which keys go where and why.
2. **Debug:** Given a `ListView` where deleting an item leaves the *next* item showing the deleted item's checkbox state, walk the interviewer through the element-matching failure and the one-line fix.
3. **Optimize:** An animation drops frames only on first play, then runs smoothly. Explain the likely cause on Skia and how Impeller changes the picture; then explain a jank that Impeller would *not* fix.
4. **Explain:** Trace a single `setState` all the way to pixels, naming each pipeline phase and which thread it runs on.

5. **Reason:** You wrap a live `VideoPlayer` in a new `Container` conditionally and playback restarts. Explain the element lifecycle transitions and how a `GlobalKey` prevents the restart.
6. **Contrast:** Argue when React Native's native-widget model is actually preferable to Flutter's self-rendered model, and what that costs you.

Widgets, Layout & Lifecycle — Interview Questions

> How Flutter widgets are built, composed, laid out, and torn down. For depth see 07 Widgets and 08 Widget Lifecycle.

This topic tests whether you understand Flutter's declarative model beyond the surface: the widget/element/render-object split, the State lifecycle and where side-effects belong, the "constraints go down, sizes go up" layout protocol, and how to avoid leaks and needless rebuilds. Interviewers use it to separate people who memorized widget names from people who understand the framework.

🟢 Basic

1. What is a Widget in Flutter? A Widget is an immutable configuration/blueprint for a piece of UI. It is not the thing on screen — it's a lightweight description that Flutter inflates into an `Element` (which holds the tree position and lifecycle) and, for rendering widgets, a `RenderObject` (which does layout and paint). Because widgets are cheap and immutable, Flutter can rebuild them freely and diff the new tree against the old.

2. StatelessWidget vs StatefulWidget — when do you use each? Use `StatelessWidget` when everything the widget needs is passed in via its constructor and never changes internally over its lifetime (it can still change when the parent rebuilds it with new inputs). Use `StatefulWidget` when the widget must hold mutable state that changes over time in response to user interaction, timers, streams, or animations, and it needs to trigger its own rebuilds via `setState`.

```
// Stateless: pure function of its inputs
class Price extends StatelessWidget {
  const Price(this.amount, {super.key});
  final double amount;
  @override
  Widget build(BuildContext c) => Text('\$$amount');
}
```

3. Why is `build()` allowed to be called many times, and what does that imply? `build()` can run on every frame, every `setState`, every parent rebuild, and on hot reload. So it must be cheap, pure, and side-effect free — no network calls, no timers, no controller creation inside `build`. Side-effects go in lifecycle methods (`initState`) or event handlers, never in `build`.

4. Why is `State` separate from the `StatefulWidget` itself? The `StatefulWidget` is immutable and gets thrown away and recreated on every rebuild. The `State` object is long-lived — it persists across rebuilds as long as the element stays in the tree — so mutable data lives there. That split is

what lets the framework rebuild widgets cheaply while keeping your state intact.

5. What is the difference between a widget's constructor being `const` and not being `const`? A `const` constructor lets Dart canonicalize the widget at compile time so the exact same instance is reused. During rebuilds Flutter can then short-circuit: if the new widget is `identical` to the old one, that subtree is skipped entirely. Non-const widgets are new instances every build, forcing a diff.

6. What does `setState` actually do? `setState` synchronously runs the callback you pass (where you mutate fields) and then marks the element dirty so it's rebuilt on the next frame. It does not rebuild immediately, and it only rebuilds this widget's subtree. Mutating state without `setState` changes the data but never schedules a rebuild, so the UI goes stale.

7. What is `BuildContext`? `BuildContext` is a handle to the location of a widget in the element tree. You use it to look up inherited widgets (`Theme.of` , `MediaQuery.of`), find ancestors, and get the render object. It's essentially the `Element` itself exposed through a narrower interface.

8. What's the core layout rule in Flutter (one sentence)? Constraints go down, sizes go up, and the parent sets the position. A parent passes `BoxConstraints` to each child, the child picks its own size within those constraints and reports it back, then the parent decides where to place the child.

9. What is a `BoxConstraints`? `BoxConstraints` is the min/max width and height a parent imposes on a child during layout (`minWidth` , `maxWidth` , `minHeight` , `maxHeight`). The child must choose a size that satisfies them. This single downward-flowing object is the whole layout contract.

10. `Row` vs `Column` vs `Flex`? `Row` lays children out horizontally, `Column` vertically. Both are just `Flex` with a fixed `direction` — `Flex` is the general form where you pass `direction: Axis.horizontal/vertical` . You control alignment with `mainAxisAlignment` (along the layout direction) and `crossAxisAlignment` (perpendicular).

11. What does `Expanded` do? `Expanded` forces its child to fill the available free space along a `Flex` 's main axis, dividing space by `flex` factor when there are several. It applies a *tight* constraint on the main axis, so the child has no say in its main-axis size.

12. What is a `Stack` / `Positioned` for? `Stack` overlays children on top of each other in paint order (later children on top). `Positioned` (only valid directly inside a `Stack`) pins a child by `top` / `right` / `bottom` / `left` / `width` / `height` . Non-positioned children are sized and aligned by the stack's `alignment` .

13. Why must you call `super.dispose()` and dispose controllers? Controllers (`AnimationController` , `TextEditingController` , `ScrollController` , `StreamSubscription` , timers) hold resources and often register with tickers or listeners. If you don't `dispose()` them when the `State` is removed, they keep firing and holding references — a memory/CPU leak. `dispose` is the mirror of `initState` .

🟡 Intermediate

14. Walk through the `State` lifecycle in order and say what belongs where. | Method | When | What goes here | |---|---|---| | `createState()` | once, when widget inserted | return your `State` instance only | | `initState()` | once, before first build | one-time setup: controllers, subscriptions, listeners. No `InheritedWidget` lookups | | `didChangeDependencies()` | after `initState`, and whenever an inherited dependency changes | react to `InheritedWidget` data (Theme/MediaQuery/Provider) | | `build()` | every frame it's dirty | return the widget subtree, pure | | `didUpdateWidget(old)` | parent rebuilds with a new widget instance | react to changed config; e.g. re-subscribe if a callback/id changed | | `deactivate()` | element removed from tree (may be reinserted) | rarely used; cleanup before possible move | | `dispose()` | permanently removed | tear down everything created in `initState` |

15. Why can't you use `context` for an `InheritedWidget` in `initState`? In `initState` the element is inserted but the framework has not yet registered this widget as a dependent of any ancestor `InheritedWidget`, so calling `Theme.of(context)` / `MediaQuery.of(context)` either throws in debug or won't rebuild you when that ancestor later changes. `didChangeDependencies` runs right after `initState` *and* every time a dependency changes — that's the correct place, because the dependency link exists and you'll be re-notified.

16. `didChangeDependencies` vs `didUpdateWidget` — what triggers each? `didChangeDependencies` fires when an `InheritedWidget` this state depends on changes (e.g. theme/locale switch). `didUpdateWidget(oldWidget)` fires when the *parent* rebuilds and hands this element a new widget of the same type — you compare `widget` vs `oldWidget` and react to changed constructor params (e.g. the parent swapped the `animation` or `url`).

17. Why does a `ListView` inside a `Column` throw an unbounded-height error? `Column` gives its children *unbounded* height on the main axis (it wants each child to be as tall as it wants). A scrollable like `ListView` wants to expand to fill all available height — but "all available height" is infinite here, which is undefined. Fix by giving it a bounded height: wrap in `Expanded` (take the remaining space), give it a fixed-height `SizedBox`, or set `shrinkWrap: true` (only for small lists — it measures all children, defeating lazy building).

18. `Expanded` vs `Flexible` vs `Spacer`? | | Constraint on child | Use | |---|---|---| | `Expanded` | tight — child *must* fill its share | fill remaining space | | `Flexible` | loose (`fit: FlexFit.loose`) — child may be *up to* its share | let a child shrink but not forced to grow | | `Spacer` | takes flexible empty space | push widgets apart with no visible child | `Expanded` is literally `Flexible(fit: FlexFit.tight)`. `Spacer` is an `Expanded` wrapping an empty `SizedBox`.

19. What is the difference between tight and loose constraints? A *tight* constraint has `minWidth == maxWidth` (and same for height) — the child has exactly one legal size, it's forced. A *loose* constraint has `min == 0` — the child may be any size up to the max, so it sizes to its content.

`BoxConstraints.tight(size)` vs `BoxConstraints.loose(size)`. `Center`, for example, loosens the constraints it passes to its child.

20. What is `LayoutBuilder` and when do you need it (vs `MediaQuery`)? `LayoutBuilder` gives you the `BoxConstraints` from the *parent* at build time, so you can build differently based on the space actually available to this widget. `MediaQuery.of(context).size` gives the whole screen/window. Use `LayoutBuilder` for responsive layout inside a panel/card that isn't full-screen; use `MediaQuery` for screen-level decisions.

21. Name a few "builder" widgets and why they exist. `LayoutBuilder` (build from parent constraints), `StreamBuilder` / `FutureBuilder` (rebuild on async data), `ValueListenableBuilder` / `AnimatedBuilder` (rebuild only on a listenable, keeping the rest const), `Builder` (get a fresh `context` below the current widget, e.g. to access a `Scaffold` provided above). They exist to scope rebuilds tightly and/or to defer building until some value/context is available.

22. How do `const` constructors help performance concretely? When a parent rebuilds, it produces new child widgets. If a child is `const`, the new instance is `identical` to the old one, so `Element.updateChild` returns immediately without rebuilding that subtree or touching its render objects. Marking leaf widgets `const` (and enabling the `prefer_const_constructors` lint) is the cheapest widespread rebuild optimization.

23. What is intrinsic sizing and why is it expensive? Intrinsic sizing (`IntrinsicWidth` / `IntrinsicHeight`, or a parent that queries `getMinIntrinsicWidth` etc.) asks a child "how big would you be ideally?" *before* the real layout pass. Answering usually requires an extra speculative layout of the subtree, turning a single-pass O(n) layout into effectively $O(n^2)$ in the worst case. Use it only when unavoidable; prefer fixed sizes, `Flexible`, or `Table` / `Wrap` alternatives.

24. Why is `key` important, and when do you need one? Keys let Flutter match elements to widgets correctly when reordering, inserting, or removing children of the same type in a list. Without keys, Flutter matches by position and can attach the wrong `State` to the wrong item (e.g. after removing an item, the checkbox state "shifts"). Use `ValueKey` / `ObjectKey` on list items with identity; use `GlobalKey` sparingly to access a state/element across the tree.

25. What happens on hot reload with respect to the lifecycle? Hot reload re-runs `build()` and preserves `State` (fields keep their values), but it does *not* re-run `initState`. So changes to `initState` logic won't take effect until a hot *restart*. This is why controllers set up once in `initState` survive a hot reload unchanged.

26. How do you rebuild only part of a large widget on a value change? Wrap just the reactive slice in `ValueListenableBuilder` / `AnimatedBuilder` / `StreamBuilder` and pass the static rest via the `child` parameter so it's built once and reused. This confines `build` work to the small subtree instead of rebuilding the whole page with `setState`.

🔴 Advanced

27. Explain the three trees: Widget, Element, RenderObject. The **Widget** tree is the immutable configuration you write. The **Element** tree is the mutable, persistent middle layer: each element holds a widget, its position in the tree, and its lifecycle, and it decides whether to update or replace its render object on rebuild. The **RenderObject** tree does the real work — layout (constraints/size), painting, and hit-testing. On rebuild Flutter diffs new widgets against the existing elements; elements are reused (updating their render objects) whenever the widget's `runtimeType` and `key` match, which is what makes rebuilds cheap.

28. Precisely, how does the element decide to update vs recreate a child? `Element.updateChild(oldChild, newWidget)` checks: if `newWidget == null`, deactivate the child; if `oldChild != null` and `Widget.canUpdate(old.widget, newWidget)` is true (same `runtimeType` **and** same `key`), it calls `child.update(newWidget)` and keeps the element (and its `State`/render object); otherwise it deflates the old element and inflates a new one. This is why changing a widget's `key` blows away its `State`.

29. Why does `initState` run before `didChangeDependencies`, and what would break if you did inherited lookups in `initState`? The framework mounts the element, then calls `initState`, then registers dependencies and calls `didChangeDependencies` for the first time. If you read an `InheritedWidget` in `initState`, you read it *without* creating a dependency, so you won't get `didChangeDependencies` callbacks when it changes — your widget would silently use stale theme/locale/provider data forever. Debug mode asserts against this for the common `.of(context)` helpers.

30. Trace a full layout pass for `Center(child: Text('hi'))` inside a sized box. The box passes tight constraints down to `Center`. `Center` (a loosening parent) passes *loose* constraints (`0..maxW`, `0..maxH`) down to `Text`. `Text` measures its content and returns its intrinsic size up. `Center` sizes itself to the incoming constraints (fills the box) and positions the child at the center by setting the child's offset. Sizes flowed up, position was set by the parent — the protocol end to end.

31. When does a `RenderObject` decide it is or isn't a "relayout boundary," and why does it matter? If a render object receives *tight* constraints (its size can't change regardless of children) and doesn't size to its parent, it becomes a relayout boundary: when it marks itself dirty, layout doesn't have to propagate up past it. Boundaries limit how far `markNeedsLayout` walks the tree, so tight constraints and `RepaintBoundary` (its paint analog) are how you contain the cost of frequent updates.

32. What subtle bug does `didUpdateWidget` prevent with listeners/controllers? If your state subscribes to something derived from a widget parameter (e.g. `widget.controller` or `widget.stream`) in `initState`, and the parent later passes a *different* controller/stream, `initState` won't run again

— so you're still listening to the old one. In `didUpdateWidget` you compare `oldWidget.controller != widget.controller`, remove the listener from the old, and add it to the new. Forgetting this is a classic "stale listener / wrong data" bug.

```
@override
void didUpdateWidget(covariant MyWidget old) {
  super.didUpdateWidget(old);
  if (old.controller != widget.controller) {
    old.controller.removeListener(_onChange);
    widget.controller.addListener(_onChange);
  }
}
```

33. `deactivate` vs `dispose` — is there a case where `deactivate` doesn't lead to `dispose`? Yes. `deactivate` runs when an element is removed from the tree, but if it's reinserted elsewhere in the *same frame* (e.g. moved via a `GlobalKey`), it's reactivated and never disposed. `dispose` only runs when the element is not readopted by end of frame and is permanently removed. So put reversible/move-safe cleanup in `deactivate` and permanent teardown in `dispose`.

34. How exactly does an undisposed `AnimationController` leak, and how would you catch it? An `AnimationController` registers with the `Ticker` provider (via `TickerProviderStateMixin`), so it keeps receiving frame callbacks and holds a reference to its `State` (and its subtree). Not disposing it means the ticker stays active, wasting frames, and the `State` can't be garbage-collected. Flutter's debug mode actively throws "A TickerProvider was disposed with an active Ticker" / leak warnings; also watch for `setState called after dispose` from late async callbacks — guard with `if (!mounted) return;`.

35. Why can `shrinkWrap: true` on a `ListView` hurt, and what's the right fix for "list inside a column"? `shrinkWrap` makes the list measure *all* its children to size itself, defeating lazy/viewport building and causing jank/O(n) work for long lists. When you need a list inside a `Column`, prefer `Expanded(child: ListView(...))` so the list gets a bounded height and stays lazy, or restructure using a `CustomScrollView` with slivers so headers and the list share one scrollable.

36. What is a `RepaintBoundary` and when do you add one? A `RepaintBoundary` isolates a subtree onto its own layer so that when it repaints (e.g. an animation) it doesn't force ancestors/siblings to repaint, and vice-versa. Add it around frequently-animating widgets or expensive-to-paint static content next to churning UI. Overusing it costs memory (extra layers), so measure with the "repaint rainbow" in DevTools first.

37. How do `Flexible` / `Expanded` interact with a child that has its own intrinsic/min size? `Expanded` applies a tight main-axis constraint equal to the child's flex share, overriding the child's preferred size — so a min-size child gets stretched or, if the share is smaller than its content, can

overflow. `Flexible` (loose) lets the child stay at its intrinsic size up to the share. If content might exceed the space, combine with `FittedBox`, `overflow` handling, or make the child scrollable — don't rely on flex alone to prevent overflow.

38. Why is mutating widget fields wrong, and where does mutable state legitimately live? Widgets are immutable by contract; the framework assumes it can compare and discard them, and it may reuse a widget instance across builds. Mutating a widget field breaks the diffing assumptions and won't schedule a rebuild. All mutable state lives in the `State` object (or a state-management holder outside the tree), which is persistent and whose changes are funneled through `setState`/listeners.

39. A screen janks on every keystroke in a `TextField`. Diagnose with lifecycle/rebuild reasoning. Likely the whole page rebuilds because the `TextEditingController`/form state change triggers `setState` high in the tree, or a heavy widget isn't `const`/isn't behind a `RepaintBoundary`. Fixes: listen to the controller with `ValueListenableBuilder` and pass the static subtree via `child`; mark unchanging children `const`; move the field into its own small `StatefulWidget` so `setState` is scoped; ensure no expensive work (parsing, layout with intrinsics) runs in `build`.

⚡ Rapid-fire (one-liners)

Question	Answer
Is a Widget mutable?	No — immutable config; `Element` holds the mutable position/lifecycle.
Where do you create controllers?	`initState`, and dispose them in `dispose`.
Where do you read `Theme.of(context)` for the first time safely?	`didChangeDependencies` (or `build`), not `initState`.
Constraints flow which way?	Down; sizes flow up; parent sets position.
`Expanded` = ?	`Flexible(fit: FlexFit.tight)`.
Tight constraint means?	`min == max` — one legal size.
Why does ListView-in-Column crash?	Column gives unbounded height; scrollable can't fill infinity. Wrap in `Expanded`.
`Spacer` is?	An `Expanded` around empty space.
Cost of `IntrinsicHeight`?	Extra speculative layout pass — up to $O(n^2)$.
What makes `const` widgets fast?	`identical` check skips rebuilding the subtree.
`didUpdateWidget` is for?	Reacting to changed constructor params from the parent.
Does hot reload call `initState`?	No — only hot restart does.
`setState` after `dispose`?	Guard with `if (!mounted) return;`.
`LayoutBuilder` gives you?	The parent's `BoxConstraints`.
`Stack` paint order?	Later children paint on top.
Changing a widget's `key`?	Destroys and recreates its `State`.

Follow-up drills

1. Design a responsive dashboard tile that switches between a 1-column and 2-column internal layout based on the space its parent gives it (not the screen size). Which builder and why?
2. Debug this: after deleting the second item in a list, the *third* item's expand/collapse state is now wrong. Explain the root cause and fix.
3. Optimize a page where a 1-second-interval clock in the app bar is causing the entire scaffold body to rebuild every tick.
4. You have a `StatefulWidget` whose `widget.userId` can change when the parent rebuilds. Write the lifecycle code that keeps a stream subscription correct across `initState`, `didUpdateWidget`, and `dispose`.
5. Explain why wrapping a whole page in a single `AnimatedBuilder` is worse than wrapping only the animating leaf, in terms of the element/render trees.

6. Given a `Row` that overflows on small screens, walk through three structurally different fixes (`Flexible`, `FittedBox`, scroll) and the trade-offs of each.

Rendering Pipeline & Performance — Interview Questions

> How Flutter turns widgets into pixels each frame, and how to keep that under the frame budget. Depth in 09 Rendering Pipeline and 21 Performance.

This topic tests whether you understand what actually happens per frame (build → layout → paint → composite → rasterize), where jank comes from (UI vs raster thread), and whether you optimize by *measuring* rather than by sprinkling `const` and `RepaintBoundary` everywhere. Senior interviews live here.

🟢 Basic

1. What are the phases of a single frame in Flutter? For a frame Flutter runs, in order: **build** (run `build()` on dirty widgets, reconcile the Element tree), **layout** (RenderObjects size/position themselves under parent constraints — constraints go down, sizes come up), **paint** (RenderObjects emit paint commands into layers), **composite** (assemble the layer tree into a scene), and **rasterize** (the raster thread turns the scene into GPU commands and pixels). Build/layout/paint/composite run on the UI thread; rasterize runs on the raster thread. Understanding this order tells you *which* work to cut when a frame is slow.

2. What triggers a new frame? What is vsync? The OS display emits a **vsync** signal (~every 16.6 ms at 60 Hz). Flutter's `SchedulerBinding` registers a frame callback with the engine, which is driven by vsync — this is Flutter's analogue of Android's **Choreographer**. When something marks the tree dirty (`setState`, animation tick, `markNeedsBuild/Layout/Paint`), Flutter schedules a frame; on the next vsync it runs the pipeline. Nothing renders "as fast as possible" — work is paced to the display refresh so you don't waste effort producing frames the screen can't show.

3. What's the frame budget and why 16 ms? At 60 Hz you get one frame every ~16.6 ms, so *all* UI-thread work (build+layout+paint+composite) for that frame must finish inside that window, and the raster thread must also finish its part. On a 120 Hz display the budget halves to ~8.3 ms. Blow the budget and the previous frame stays on screen an extra refresh — that dropped frame is **jank**. The budget is shared, not per-phase.

4. What is jank? Jank is a visibly dropped or late frame — a stutter during scrolling or animation. It happens when a frame can't be produced within the budget, so the display repeats the old frame. It comes in two flavors: **build/UI jank** (too much Dart work on the UI thread) and **raster jank** (the GPU/raster thread takes too long to rasterize the scene). Fixing them requires opposite tactics, so you must first identify which one you have.

5. What is the difference between the UI thread and the raster thread? The **UI (Dart) thread** runs your Dart code: build, layout, paint (recording), and composite. The **raster thread** (historically "GPU thread") takes the layer tree and issues actual draw calls to the GPU. They run in parallel and pipeline across frames. A frame is only smooth if *both* finish their share in time. This is why "my code is fast but it still janks" is usually raster-side (expensive shaders, huge images, saveLayer).

6. What is `const` on a widget and why does it help performance? A `const` widget is canonicalized at compile time — the same instance is reused. During reconciliation, when Flutter compares the new widget with the old, `const` widgets short-circuit via `identical()`: same instance ⇒ the Element and its subtree are not rebuilt. So `const` widgets skip build work for their whole subtree. Cheap and effective; enable `prefer_const_constructors` lint.

7. What is `ListView.builder` and why prefer it over `ListView(children: [...])`? `ListView(children: [...])` builds *all* children eagerly, even off-screen ones — O(n) work and memory up front. `ListView.builder` is lazy: it builds only the items in (and near) the viewport via `itemBuilder`, recycling as you scroll. For long or infinite lists this is the difference between a smooth scroll and an out-of-memory / long-build-jank list. Use `.builder` whenever the item count is large or unknown.

8. What is `RepaintBoundary` at a high level? `RepaintBoundary` inserts a separate compositing layer so its subtree paints independently. When something *outside* the boundary repaints, the boundary's cached layer is reused instead of repainting its subtree — and vice versa. It isolates repaint work. It's a tool for *paint*-phase cost, not build cost, and it's not free (extra layer + memory), so it must be justified.

9. What is a key and how does it relate to performance? Keys control how Flutter matches new widgets to existing Elements/State during reconciliation. When a list of same-type widgets reorders, keys let Flutter *reuse* the right Elements and their State instead of tearing down and rebuilding — preserving State and avoiding wasted rebuild/layout. Use `ValueKey` / `ObjectKey` on items that reorder or get inserted/removed in the middle of a list.

10. What tool do you use to diagnose performance problems? Flutter DevTools — specifically the Performance/Timeline view (per-frame flame chart split into UI and raster), the Frame chart (bars colored by which thread overran), the CPU profiler, the widget **rebuild counter**, and the memory view. Always profile in **profile mode** on a **real device**, never debug mode (debug adds asserts/overhead and is not representative).

11. Why must you profile in profile mode, not debug? Debug mode runs unoptimized JIT code with assertions, service extensions, and no AOT compilation, so timings are meaningless (often 5–10x slower). **Profile mode** is AOT-compiled like release but keeps tracing/DevTools hooks, so numbers reflect reality while remaining observable. Release mode has no tracing at all. Rule: profile in profile mode on a physical device representative of your low-end target.

12. What does "measure before optimizing" mean here? Don't guess. Reproduce the jank, capture a timeline, and read whether frames overrun on the UI thread or the raster thread — then optimize *that*. Blindly adding `const`, `RepaintBoundary`, or micro-tweaks often adds complexity and memory for no measured gain, and can even make things worse. Optimization without a before/after measurement is superstition.

🟡 Intermediate

13. How do you tell build jank from raster jank in DevTools? In the Frame chart each frame is a bar split into a **blue/UI** portion and a **green/raster** portion. If the UI portion is over budget, it's build/layout/paint jank — look at your Dart work (expensive `build()`, big layouts, synchronous work on the UI thread). If the raster portion is over budget, it's raster jank — expensive painting, `saveLayer`, big/unscaled images, or **shader compilation**. Enable the performance overlay to see the two graphs live.

14. When does `RepaintBoundary` help, and when does it hurt? It **helps** when a small, frequently-repainting subtree (an animation, a progress spinner, a blinking cursor) sits inside a larger static one — the boundary stops the animation from dirtying the whole screen's paint, and stops screen repaints from redrawing the animation. It **hurts** when applied indiscriminately: each boundary is a new layer costing memory and a composite step, and wrapping cheap or rarely-painting widgets just adds overhead. Use DevTools' "Highlight repaints" to find what actually repaints, then wrap only that. `ListView` already inserts RepaintBoundaries around its children, so don't double-wrap them.

15. Why doesn't `const` help if I pass a non-const argument? `const` requires *all* constructor arguments to be compile-time constants. The moment you pass a runtime value (a variable, a value from state, a closure, `MediaQuery.of(context)`), the constructor can't be `const`, so a fresh instance is created every build and the subtree can rebuild. That's why hoisting truly-static subtrees into `const` (or into fields) matters: you're preserving the `identical()` short-circuit. A closure or `Theme.of(context)` in the args silently defeats it.

16. Compare `itemExtent`, `prototypeItem`, and neither in a scroll list.

Approach	What it does	When
`itemExtent: h`	Fixes each item's main-axis size, so the list can compute scroll positions and off-screen extent *without laying out items*	All items same height — fastest
`prototypeItem`	Lays out one prototype to derive the extent, then treats items as fixed	Uniform but height not known as a literal
Neither	List must lay out items to know their size — more layout work, slower scroll-to and extent math	Genuinely variable heights

Giving the list a known extent avoids per-item layout during fling and makes scrollbar/`jumpTo` math cheap.

17. What is `cacheExtent` and how does tuning it cut jank? `cacheExtent` is how many logical pixels *beyond* the visible viewport the list keeps built/painted (default ~250 px). A larger cacheExtent builds upcoming items earlier, so they're ready before they scroll in — smoothing fast flings — at the cost of more memory and build work off-screen. Too large wastes memory and can *cause* build jank; too small causes items to pop in / build late at the viewport edge. Tune it against a measured fling, don't max it out.

18. My whole screen rebuilds on every tick. How do you stop that? Narrow the rebuild scope:

- **Split widgets** so `setState`/state changes live in the smallest subtree that actually changes; extract the static parts into `const` siblings.
- Use a **selector** (`Selector` in Provider, `select` in Riverpod, `context.select`, `BlocSelector`) so a widget rebuilds only when the specific slice it reads changes.
- For animations, use `AnimatedBuilder` / `ValueListenableBuilder` with a `child:` argument so only the animated part rebuilds and the static `child` is passed through untouched.
- Verify with the **rebuild counter** that the count actually dropped.

19. Why is `AnimatedBuilder`'s `child` parameter a performance feature? The `builder` runs every animation frame, but anything you pass as `child` is built **once** and handed into the builder as an already-built widget. So you keep the expensive static subtree out of the per-frame rebuild and only rebuild the tiny animated wrapper. Same pattern for `ValueListenableBuilder` and `TweenAnimationBuilder`. Forgetting it means rebuilding the whole subtree 60–120 times a second.

20. What is shader compilation jank? The first time a particular shader is needed (a gradient, a specific `BackdropFilter`, some `Path` effects), the engine compiles it *on the raster thread*, mid-frame — a one-time but very visible stutter, classically on the first run of an animation. It's raster jank you can't fix with Dart optimizations. Historically mitigated with **SkSL warm-up** (bundling precompiled shaders captured via `--cache-sksl`); the modern fix is **Impeller**.

21. What is Impeller and how does it address shader jank? Impeller is Flutter's rendering engine that replaces the runtime Skia shader-compilation path. It **precompiles a fixed set of shaders at build time** (offline), so there's no on-the-fly shader compilation during animation — killing shader-compilation jank by design. It's the default on iOS and is the default/rolling out on Android. With Impeller the whole SkSL warm-up workaround becomes unnecessary.

22. How does Flutter cache images and why does that matter? Decoded images live in `ImageCache` (via `PaintingBinding.instance.imageCache`), keyed by provider, with limits on count (default 1000) and bytes (default ~100 MB). Re-using the same `ImageProvider` serves a decoded bitmap instead of re-decoding/re-downloading. The killer detail: images are cached at their **decoded** resolution, so decoding a 4000×3000 photo to show a 100×100 thumbnail wastes huge memory — use `cacheWidth` / `cacheHeight` (or `ResizeImage`) to decode at display size.

23. How do you reduce app (binary) size? Build with `--split-per-abi` (or App Bundle) so each device downloads only its ABI; use `--obfuscate --split-debug-info` for release; run `flutter build --analyze-size` to find the heaviest assets/packages; compress/right-size images and prefer vector/ `WebP` ; drop unused fonts/locales and unused dependencies; and use **deferred loading** to split rarely-used features out of the initial download. Tree-shaking of icon fonts happens automatically in release.

24. What is deferred loading and when do you use it? `import '...' deferred as x;` plus `await x.loadLibrary();` splits a library into a chunk loaded on demand instead of at startup. It shrinks the initial download / startup cost for features most users never open (an admin panel, a heavy report/PDF module). It's most impactful on **Flutter Web** (real network chunks); on mobile the benefit is smaller. The trade-off is an async load point you must handle in the UI.

🔴 Advanced

25. Walk through exactly what happens between `setState` and pixels on screen. `setState` calls `Element.markNeedsBuild` , adding the element to the dirty list and calling `scheduleFrame` on the `SchedulerBinding` . On the next **vsync**, `WidgetsBinding.drawFrame` runs: (1) **build** — `buildScope` rebuilds dirty elements, reconciling widgets against the element tree; (2) **layout** — `flushLayout` visits `markNeedsLayout` RenderObjects, passing constraints down and receiving sizes up; (3) **paint** — `flushPaint` records paint ops into the layer tree for `markNeedsPaint` objects; (4) **compositing** — the layer tree is flattened and a `Scene` is built via `SceneBuilder` ; (5) **semantics** flush; then the scene is handed to the engine, and the **raster thread** rasterizes it to the GPU. Relayout boundaries and repaint boundaries limit how far steps 2 and 3 propagate.

26. What are relayout boundaries and how do they bound layout cost? A RenderObject is a **relayout boundary** when its size can't be affected by its children's changes — this holds when it's given **tight constraints**, when `parentUsesSize` is false, or when `sizedByParent` is true. When a child calls `markNeedsLayout` , dirtiness propagates *up* only until it hits a relayout boundary, so layout re-runs on that bounded subtree rather than the whole tree. That's why tightly-constraining a subtree (e.g. a fixed-size box) is a real perf lever: it caps how much layout a deep change can trigger.

27. Why does `Opacity` (or `ClipRRect` / `saveLayer`) cause raster jank, and what are the alternatives? `Opacity` , non-trivial clips, `ShaderMask` , and `BackdropFilter` force the engine to allocate an **offscreen buffer** via `saveLayer` — render the subtree into it, then composite — which is expensive on the raster thread, especially animated. Alternatives: use `AnimatedOpacity` only where needed; for a fading image use `FadeInImage` ; prefer `Container(color:)` over `Opacity` on a color; use `clipBehavior: Clip.hardEdge` instead of anti-aliased clips when acceptable; and for a whole-widget fade, `Opacity` on a `RepaintBoundary` 'd subtree so the offscreen is at least cached. Watch the raster line in the overlay to confirm.

28. Where do `RepaintBoundary`s already exist for free, and why does that change your strategy? Flutter auto-inserts `RepaintBoundary`s in several spots — around each `ListView`/`Scrollable` child, in `TabBarView` pages, and around some transitions — precisely because scrolling content constantly repaints and must be isolated from the static frame. So you rarely need to wrap list items yourself (double-wrapping wastes layers). Your job is to add boundaries where the *framework can't know* your intent: a hand-rolled animation inside a static page, a `CustomPaint` that repaints often, a video/canvas surface.

29. How do you diagnose "the UI thread is fine but frames still drop"? That's raster-thread bound. In the timeline, the raster track overruns while the UI track is comfortable. Usual suspects: shader compilation (first-run stutter → check with `--trace-skia`/switch to Impeller), too many `saveLayer`s (Opacity/clip/blur), large unscaled images decoding/uploading, an unbounded number of layers, or expensive `CustomPainter` paint. DevTools' **Raster Stats** shows per-layer rasterization cost and lets you spot the one layer eating the budget. Fix the layer, not the Dart.

30. How does `RepaintBoundary` interact with the `debugRepaintRainbowEnabled` and profiling flags? `debugRepaintRainbowEnabled = true` recolors each layer's border on every repaint, so a region that flashes new colors constantly is repainting every frame — that's your candidate to wrap (or to investigate why it repaints). Pair it with the rebuild counter (build-phase repaint cause) and Raster Stats (paint cost). The workflow: rainbow to find *what* repaints → decide if a boundary isolates it → measure raster time before/after. Adding a boundary that doesn't reduce measured repaint area is pure overhead.

31. What causes memory growth / OOM in a Flutter app and how do you find it? Common causes: unbounded image cache from full-res decodes (fix with `cacheWidth`), retained large objects via un-cancelled `StreamSubscription`s / `AnimationController`s / `Timer`s not disposed, `GlobalKey` misuse holding subtrees, and caching lists that never evict. Use DevTools **Memory** view: watch the heap over an interaction, take snapshots, diff allocations, and look for classes whose instance count only grows. The `dispose()` audit (every controller/subscription cancelled) fixes most leaks. Set `imageCache.maximumSizeBytes` down on low-RAM targets.

32. When should you move work off the UI thread, and how? Move CPU-heavy synchronous work (JSON parsing of large payloads, image processing, crypto, parsing/compression) off the UI thread when it would exceed the build budget. Use `compute()` for a one-shot, or `Isolate.run` (Dart 2.19+) / a long-lived isolate with `SendPort` for repeated work. Note isolates don't share memory, so you pay serialization cost — only worth it above a threshold you should *measure*. `async/await` alone does **not** help here: it doesn't add parallelism, so a long synchronous computation inside an `await`ed function still blocks the UI thread.

33. How do you optimize a `CustomPainter` that repaints during animation? Implement `shouldRepaint` to return `false` when inputs are unchanged (a wrong `shouldRepaint` forces repaint every frame). Isolate the painter behind a `RepaintBoundary` so its repaints don't dirty siblings. Drive it with a `Listenable` passed to `CustomPaint(painter: ...)` / repaint arg so only the paint runs, not a

widget rebuild. Avoid allocating `Paint` / `Path` objects inside `paint()` — hoist them to fields. Minimize `saveLayer`, and cache static geometry (e.g. a `Picture` recorded once) if only a transform changes.

34. Explain the pipeline's back-pressure: what happens if the raster thread falls behind the UI thread? The two threads pipeline: the UI thread produces a layer tree for frame N while the raster thread rasterizes frame N-1. If the raster thread consistently takes longer than the budget, produced frames queue and the engine throttles frame scheduling — so even a fast UI thread can't help, and you see sustained jank tied to raster cost. This is why raster-bound problems (shaders, saveLayer, image upload) feel like "everything is slow" and can't be profiled away on the Dart side. The fix must reduce raster work or precompile shaders (Impeller).

35. Give a principled optimization workflow for a janky scrolling screen.

1. Reproduce on a **real low-end device in profile mode**. 2) Capture a timeline while scrolling; read UI vs raster split. 3) If **UI-bound**: check rebuild counts (selectors/ `const` /split widgets), ensure `ListView.builder` with `itemExtent` / `prototypeItem`, tune `cacheExtent`, move heavy parsing to an isolate. 4) If **raster-bound**: find the costly layer in Raster Stats, remove `Opacity`/clip/ `saveLayer`, decode images at display size, confirm Impeller is on. 5) **Re-measure** and keep only changes with a proven before/after win. 6) Add a regression guard if possible (e.g. a golden/perf test). The discipline — measure, change one thing, re-measure — is the actual answer interviewers want.

36. `const` vs keys vs `RepaintBoundary` — which phase does each optimize?

Tool	Phase it saves	Mechanism
`const` widgets	**Build**	`identical()` short-circuit skips rebuild of unchanged subtree
Keys	**Build/layout**	Correct element+state reuse across reorders, avoids teardown/rebuild
Selectors	**Build**	Rebuild only on the specific state slice that changed
`itemExtent` / `cacheExtent`	**Layout**	Skip per-item layout / control off-screen build volume
`RepaintBoundary`	**Paint/composite**	Isolate a subtree's layer so repaints don't cascade
Impeller / precompiled shaders	**Rasterize**	No runtime shader compilation on the raster thread

Knowing *which phase* a tool targets is what stops you from applying `const` to a raster problem.

⚡ Rapid-fire (one-liners)

Q	A
Frame pipeline order?	build → layout → paint → composite → rasterize
Which phases on UI thread?	build, layout, paint(record), composite
Which on raster thread?	rasterize (draw calls to GPU)
60 Hz budget?	~16.6 ms; 120 Hz ~8.3 ms — shared across all work
Flutter's Choreographer analogue?	`SchedulerBinding` + engine vsync frame callback
Two kinds of jank?	UI/build jank vs raster jank
Why `const` helps?	`identical()` short-circuit skips subtree rebuild
`ListView` vs `.builder`?	eager (all children) vs lazy (viewport only)
`itemExtent` for?	fixed item size → skip per-item layout
`cacheExtent` for?	pixels built beyond viewport to smooth flings
When RepaintBoundary hurts?	over-wrapping cheap/static widgets → extra layers
Shader jank fix?	Impeller (precompiled shaders) / SkSL warm-up
Big image, tiny slot fix?	`cacheWidth` / `cacheHeight` / `ResizeImage`
Profile in which mode?	profile mode, real device, never debug
Reduce initial download?	deferred loading + `--split-per-abi` /App Bundle
Heavy JSON parse blocks UI — fix?	`compute` / `Isolate.run`, not just `await`
DevTools for repaints?	Highlight repaints / repaint rainbow + Raster Stats
`AnimatedBuilder` child arg?	build static subtree once, skip per-frame rebuild

Follow-up drills

1. **Optimize** a 60-item feed that janks on fling: given a timeline showing raster overrun, list the exact changes you'd make and how you'd verify each.
2. **Debug this scenario:** an animation stutters *only the first time* it plays, then is smooth. Name the cause, prove it in DevTools, and give two fixes.
3. **Design** a rebuild strategy for a dashboard where one card updates every second but 20 others are static — minimize rebuilds and justify each tool.
4. **Explain** to a mid-level engineer when adding `RepaintBoundary` makes things *worse*, with a concrete example.
5. **Optimize** app startup and download size for a feature-heavy app where 70% of users only use the home tab.

6. **Debug** an OOM that only reproduces after scrolling an image-heavy gallery for a minute — outline your DevTools memory workflow end to end.

State Management — Interview Questions

> How Flutter apps hold, share, and rebuild on data. For depth see the handbook module 11 State Management, especially 06_riverpod.md.

State management is the highest-signal topic for mid/senior Flutter interviews: it exposes whether you understand the widget/element tree, rebuild costs, and testability — not just which library you memorized. Interviewers push from "what is setState" toward "why does everything rebuild" and "when would you *not* use BLoC".

🟢 Basic

1. What is "state" in Flutter, and what are its two kinds? State is any data that can change during the app's lifetime and affects the UI. Flutter distinguishes two kinds: **ephemeral (UI/local) state** — a `TabController` index, a checkbox toggle, a text-field's current value — that lives in one widget and no one else needs; and **app (shared) state** — auth status, cart contents, user settings — that multiple widgets across the tree read and mutate. The distinction is the first design question: ephemeral state belongs in `setState`; app state needs a shared solution (Provider/Riverpod/BLoC).

2. When is `setState` the right choice? When the state is ephemeral and confined to a single `StatefulWidget`. `setState` marks the element dirty so the framework rebuilds that subtree on the next frame. It is not "wrong" or amateur — for a toggle, an animation flag, or an expansion tile, it's the simplest correct tool. It becomes a problem only when the same state must be shared upward or across siblings.

3. What does `setState` actually do? It runs the callback you pass (to mutate fields synchronously) and then calls `markNeedsBuild()` on the element, scheduling it to rebuild in the next frame. It does *not* rebuild immediately, and it does *not* do a diff of your data — Flutter reruns `build()` and diffs the returned widget tree. Calling `setState` with an empty body still triggers a rebuild; mutating a field *without* `setState` updates the data but never repaints.

4. What is "lifting state up"? Moving state from a child to the nearest common ancestor when siblings need to share it. The ancestor owns the state and passes data down via constructor params and passes callbacks down to let children request changes. It's the vanilla Flutter answer before reaching for a library — see 02_setstate_and_lifting_state.md. The pain point it eventually creates ("prop drilling" through many layers) is exactly what InheritedWidget and Provider solve.

5. What is `InheritedWidget` and why does it matter? `InheritedWidget` is the framework primitive that lets a widget expose data to its entire subtree in O(1) lookup, without passing it through every constructor. Descendants call `context.dependOnInheritedWidgetOfExactType<T>()` (usually via a static `.of(context)`), which registers the calling element as a dependent. When the inherited

widget is rebuilt with new data and its `updateShouldNotify` returns true, *only the registered dependents* rebuild. **Provider, Riverpod, `Theme`, `MediaQuery`, and `Navigator` all build on it** — this is the "how does Provider work under the hood" answer.

6. What is `ValueNotifier` / `ValueListenableBuilder`? `ValueNotifier<T>` is a lightweight `ChangeNotifier` holding a single value; assigning `.value` notifies listeners. `ValueListenableBuilder<T>` subscribes to it and rebuilds only its `builder` when the value changes. It's the zero-dependency way to get surgical rebuilds for one piece of state — great for a counter, a loading flag, or form field validity. See 04_value_change_notifier.md.

```
final counter = ValueNotifier<int>(0);
ValueListenableBuilder<int>(
  valueListenable: counter,
  builder: (context, value, _) => Text('$value'), // only this rebuilds
);
counter.value++; // triggers the rebuild
```

7. What is `ChangeNotifier`? A class that maintains a list of listeners and calls them when you invoke `notifyListeners()`. You extend it, mutate your fields, then call `notifyListeners()` to broadcast "I changed". It's the model class Provider's `ChangeNotifierProvider` and `Consumer` are built around. Remember to `dispose()` it to remove listeners and avoid leaks.

8. What problem does Provider solve? Provider is a wrapper around `InheritedWidget` that removes its boilerplate and adds lifecycle handling (creation, disposal) and easy consumption. Instead of hand-writing an `InheritedWidget` + `updateShouldNotify` + a `.of` method, you write `ChangeNotifierProvider(create: ...)` and read it with `context.watch/read` or `Consumer`. It gives you dependency injection + change notification with a clean API. See 05_provider.md.

9. What's the difference between `context.watch`, `context.read`, and `context.select`?

- `context.watch<T>()` — subscribes; the widget rebuilds when `T` notifies. Use in `build`.
- `context.read<T>()` — one-time read, no subscription. Use inside callbacks (`onPressed`) to *call* methods.
- `context.select<T, R>((t) => t.field)` — subscribes to a *derived slice*; rebuilds only when that slice changes. Use to avoid rebuilding on unrelated changes.

Calling `watch` inside a callback, or `read` inside `build` for values you display, are the two classic mistakes.

10. What is BLoC in one line, and what is Cubit? BLoC (Business Logic Component) maps a stream of **events** to a stream of **states** — UI dispatches events, the bloc emits new states, the UI rebuilds via `BlocBuilder`. **Cubit** is a simplified BLoC without events: you call methods directly that `emit` new states. Cubit is less boilerplate; BLoC's event log gives you traceability and easier handling of complex, event-driven flows. See 07_bloc.md and 08_cubit.md.

11. What is Riverpod in one line? A rewrite of Provider by the same author that moves providers out of the widget tree into top-level globals, so state access is **compile-safe** (no `BuildContext`, no runtime `ProviderNotFound`) and providers are composable and testable. See 06_riverpod.md.

12. What is GetX known for? An all-in-one package bundling state management (`Obx`, `.obs`), routing, and dependency injection with minimal boilerplate and no `BuildContext` needed for many operations. It's popular for speed of development but heavily criticized (see the Advanced tier) for encouraging poor architecture and hiding magic.

🟡 Intermediate

13. Walk through what happens on a rebuild when you call `notifyListeners()` in a `ChangeNotifierProvider`. `notifyListeners()` iterates the notifier's listeners. Provider's element is one such listener; when notified, it rebuilds the `InheritedWidget` (the internal `_InheritedProviderScope`) with the same notifier instance but a bumped state. Because `updateShouldNotify` returns true, every element that called `watch` / `Consumer` / `select` on it is marked dirty. `select` dependents are re-evaluated and only rebuild if their selected slice actually changed. `read` dependents are not subscribed and are untouched.

14. Why does `context.watch` rebuild "too much", and how do you fix it? `watch` subscribes to the *whole* notifier, so any `notifyListeners()` rebuilds the widget even if the field it displays didn't change. Fixes: use `context.select` to depend on a specific slice; wrap only the truly-dynamic subtree in a `Consumer` (or `Selector`) so the rest of `build` stays static; or split one fat notifier into several focused ones. The mental model: subscribe to the smallest slice you actually render.

15. What causes `ProviderNotFound` and how do you avoid it? It's thrown when `context.read/watch<T>()` walks up the tree and finds no matching provider above that element. Common causes: the provider is declared *below* or as a *sibling* of the consumer; you're reading in the same `build` that created the provider (wrong `context`); or you read a provider across a route boundary that doesn't inherit it. Fixes: hoist the provider above all consumers (often above `MaterialApp`), or use a `Builder` / `Consumer` to get a context beneath the provider. **This entire class of runtime bug is what Riverpod eliminates by design.**

16. Why is calling `notifyListeners()` (or `setState`) inside `build` a bug? `build` must be a pure function of state. Notifying/setting state during build schedules another build, which runs, notifies again — an infinite rebuild loop, or a "setState/markNeedsBuild called during build" assertion. It usually happens when you mutate a notifier while reading it, or trigger a fetch in `build`. Fix: move side effects to `initState`, event handlers, or `addPostFrameCallback`; do data mutation in response to user actions, not during rendering.

17. Concretely, how does Riverpod fix Provider's problems? | Problem in Provider | Riverpod's fix | |---|---| | `ProviderNotFound` at runtime | Providers are top-level globals — resolved at compile time, can't be "missing" | | Can't have two providers of same type | Each provider is a unique object, not keyed by type | | Needs `BuildContext` to read | `ref` reads anywhere, including outside widgets | | Logic coupled to the widget tree | Providers live outside the tree; fully unit-testable | | Manual disposal | `autoDispose` frees state when no longer listened |

18. What are `ref.watch`, `ref.read`, and `ref.listen` in Riverpod? `ref.watch(p)` subscribes — the widget/provider rebuilds when `p`'s value changes (use in `build` and to compose providers). `ref.read(p)` reads once without subscribing (use in callbacks). `ref.listen(p, cb)` runs a side effect (navigation, snackbar) on change *without* rebuilding. Mirrors Provider's watch/read but decoupled from context.

19. What are `family` and `autoDispose` in Riverpod? `family` parameterizes a provider — `userProvider(userId)` creates a distinct provider per argument, so you can fetch by id without a global mutable variable. `autoDispose` destroys the provider's state once nothing is listening (e.g., you leave a screen), preventing stale data and memory leaks. They compose: `FutureProvider.autoDispose.family<User, String>(...)` is a per-id, self-cleaning async fetch — a very common pattern.

20. When should you choose BLoC over Provider/Riverpod? When flows are genuinely event-driven and you value an explicit, testable, replayable event→state history: complex forms, multi-step wizards, apps needing strict separation of business logic, or teams wanting one enforced pattern at scale. BLoC's structure and `bloc_test` / `HydratedBloc` ecosystem shine on large teams. For simple shared state it's over-engineering — that's where Cubit or Riverpod fit better.

21. Contrast BLoC and Cubit — when Cubit? Cubit drops events; you expose methods that `emit` states directly. Use Cubit when the transition is a simple "do X → new state" and you don't need an event trail. Use full BLoC when you need to (a) log/trace what triggered each transition, (b) transform events (debounce search, `concurrent` / `droppable` with `bloc_concurrency`), or (c) model complex event interplay. Many teams default to Cubit and escalate to BLoC only where events add value.

22. How does `BlocBuilder` avoid unnecessary rebuilds? `BlocBuilder` rebuilds on every emitted state by default, but its `buildWhen: (prev, curr) => ...` lets you gate rebuilds to meaningful transitions. Pair with `BlocListener` (side effects, no rebuild) and `BlocSelector` (rebuild on a derived slice) — the same "subscribe to the minimum" principle as Provider's `select`. `context.select` / `buildWhen` / `Selector` are the cross-library rebuild-minimization tools.

23. Why must you `dispose` notifiers/controllers, and what happens if you don't? Listeners hold references; an undisposed `ChangeNotifier` / `ValueNotifier` / `StreamController` / `Bloc` / `TextEditingController` keeps its listeners and closures alive, leaking memory and sometimes firing callbacks on dead widgets (`setState called`

`after dispose`). In `StatefulWidget` dispose in `State.dispose()`. Provider disposes what it created automatically; Riverpod's `autoDispose` and `ref.onDispose` handle it declaratively; BLoC closes its stream in `close()`.

24. How do you handle async/loading/error state cleanly? Model it as data, not scattered booleans. Riverpod's `AsyncValue<T>` (`data` / `loading` / `error`) and its `.when(...)` is the canonical example; BLoC uses sealed state classes (`Loading`, `Loaded`, `Error`); with Provider you expose an enum/sealed status on the notifier. The anti-pattern is a `bool isLoading` + `String? error` + `T? data` combo that can represent impossible states (loading *and* error). Prefer a single union type.

🔴 Advanced

25. Explain precisely how `InheritedWidget` propagates changes — why is it O(1) to read but targeted on update? Reading via `dependOnInheritedWidgetOfExactType` is O(1) because each element caches a map of its inherited widgets by type, populated as it's mounted — a hash lookup, not a tree walk at read time. On the *first* dependency, the element registers itself in the inherited element's dependents set. When the `InheritedWidget` is rebuilt and `updateShouldNotify(old)` returns true, the framework calls `notifyClients`, which marks *exactly* the registered dependent elements dirty — not the whole subtree. This is why "everything rebuilds" is a myth for InheritedWidget itself; over-rebuilding comes from subscribing too coarsely at a higher layer.

26. What's the difference between `InheritedWidget` and `InheritedModel`? `InheritedWidget` notifies all dependents when `updateShouldNotify` is true — dependents can't say "I only care about part of your data". `InheritedModel<T>` adds *aspects*: dependents subscribe to named aspects, and `updateShouldNotifyDependent(old, changedAspects)` lets you rebuild only the widgets whose aspect changed. `MediaQuery` uses this idea so a widget depending on text-scale isn't rebuilt by a size-only change. It's the framework-level version of `select`.

27. A screen rebuilds fully on every keystroke in a search field wired to a Provider. Diagnose and fix. The consumer is almost certainly `context.watch`-ing the whole notifier (or a `Consumer` wraps too much of the tree), and each keystroke calls `notifyListeners()`. Fixes, in order: (1) narrow the subscription with `context.select` / `Selector` / `buildWhen` to the exact rendered slice; (2) shrink the `Consumer` / `builder` to only the dynamic widget so `const` and static subtrees are excluded; (3) debounce the input so you're not emitting on every character; (4) if the field's text is only ephemeral UI state, keep it in a local `TextEditingController` and don't push every keystroke into app state at all.

28. Why is Riverpod called "compile-safe", and what class of bugs does that remove? Because providers are typed top-level objects resolved by reference, not looked up by runtime type through `BuildContext`. That removes: `ProviderNotFound` (a missing provider is a missing symbol — won't compile / obvious at wire-up), ambiguous "two providers of the same type" resolution, and reading the wrong scope's value. The tradeoff is global provider declarations, which some find less encapsulated — but they're lazily instantiated and, with `autoDispose`, not truly long-lived.

29. What are the main criticisms of GetX?

- **Hidden global state & service-locator magic** (`Get.put` / `Get.find`) make dependencies implicit and hard to trace or test.
- `GetxController` **outliving its widget** can leak or hold stale state if lifecycle isn't managed carefully.
- **Encourages putting logic in controllers coupled to Get's navigation/DI**, so it's hard to migrate off.
- `Obx` **reactivity is implicit** — it tracks whatever `.value` you read during build, which can silently rebuild too much or too little.
- **Does too much** (state + routing + DI + utils), violating separation of concerns and making the "one package" a lock-in.

It optimizes for initial velocity at the cost of testability and long-term maintainability — that's the senior take. Detail in 09_getx.md.

30. How do you unit-test state logic across these libraries? The goal is testing logic *without* pumping widgets. **Provider/ChangeNotifier:** instantiate the notifier, call methods, assert on fields / listen for notifications. **Riverpod:** create a `ProviderContainer` (with `overrides` to inject fakes), `container.read(provider)` , act, assert — no widget tree needed; `overrideWith` swaps dependencies. **BLoC/Cubit:** use `bloc_test` 's `blocTest(build/act/expect/verify)` to assert the emitted state sequence. **ValueNotifier:** listen and assert `.value` . The library that makes logic testable *outside* the tree (Riverpod, BLoC) is generally easier to test than one requiring a `BuildContext` .

31. How would you inject a fake repository for a widget/screen test? Provider: wrap the widget under `ChangeNotifierProvider(create: (_) => Vm(fakeRepo))` in the test's `pumpWidget` . Riverpod: wrap in `ProviderScope(overrides: [repoProvider.overrideWithValue(fakeRepo)])` — clean and per-test. BLoC: pass a mocked bloc via `BlocProvider.value` or mock the repo the bloc depends on. This overlaps with DI — see 10_dependency_injection.md.

32. `setState() called after dispose()` **— root cause and fix.** An async callback (a `Future` , a stream event, a timer) completes *after* the widget was removed and calls `setState` . The element is defunct, so the assertion fires. Fixes: guard with `if (!mounted) return;` before `setState` ; cancel `StreamSubscription` s/ `Timer` s and complete-guard futures in `dispose()` ; or move the state out of the widget into a notifier/bloc whose lifecycle you control (which is partly *why* you lift long-lived async state out of `StatefulWidget`).

33. How do you decide when app state should be *persisted* vs. held only in memory, and how does that interact with your state solution? Persist state that must survive restarts (auth token, cart, settings, draft) via `HydratedBloc` , Riverpod + a storage layer, or manual `SharedPreferences` / `Drift` reads in the notifier's init. Keep transient/derived state in memory. The

state solution should treat persistence as a *repository dependency*, not bake storage calls into UI — the notifier/bloc reads/writes a repo, keeping the state layer testable and the storage swappable. See offline/persistence in 09_networking_data_storage.md.

34. Two providers/notifiers depend on each other's data (derived state). How do you model it without loops or stale reads? Prefer *derived* providers over cross-mutation. In Riverpod, a provider `ref.watch`es its inputs and recomputes — the dependency graph is explicit and updates propagate automatically, no manual syncing. With Provider, use `ProxyProvider` / `ChangeNotifierProxyProvider` to feed one into another. Avoid two mutable notifiers each calling the other's setter — that's how you get rebuild loops and order-dependent bugs. The rule: single source of truth, everything else is a pure function of it.

35. Give a decision framework for choosing a state solution on a new project. Weigh: team size/experience, app complexity, how much of the state is shared vs. ephemeral, testability needs, and whether you need an enforced pattern. Ephemeral → `setState` / `ValueNotifier`. Small–medium shared state, familiar team → Provider. Want compile-safety, composable async, best testability → Riverpod (the current default recommendation for greenfield). Large team, complex event-driven domain, want one rigid pattern + tooling → BLoC. Avoid choosing GetX for anything you expect to maintain for years. Don't mix three solutions in one app — pick one primary. Full matrix in 10_comparison_and_selection.md.

36. Comparison table — the one interviewers ask you to draw.

Dimension	setState	ValueNotifier	Provider	Riverpod	BLoC/Cubit	GetX
Built on	Element dirty-mark	ChangeNotifier	InheritedWidget	Own (no context)	Streams	Own rea
Needs BuildContext	n/a	No	Yes	No	For consuming	No
Boilerplate	Lowest	Very low	Low	Low–medium	Highest (BLoC)	Low
Compile-safe lookup	n/a	n/a	No (`ProviderNotFound`)	**Yes**	Partial	No
Testability (no tree)	Poor	OK	OK	**Excellent**	**Excellent**	Poor
Async built-in	No	No	Manual	`AsyncValue`/FutureProvider	Sealed states	`.obs` /\
Best for	Ephemeral	One value	Small/medium shared	Greenfield, composable async	Complex event-driven	Rapid prototyp
Main risk	Doesn't scale	Manual wiring	Runtime not-found	Global providers	Verbosity	Architec rot

⚡ Rapid-fire (one-liners)

Q	A
Ephemeral vs app state?	Local-to-one-widget vs shared-across-tree.
What primitive underlies Provider/Theme/MediaQuery?	`InheritedWidget`.
`watch` vs `read`?	Subscribe-and-rebuild vs one-shot read (use `read` in callbacks).
Tool for surgical Provider rebuilds?	`context.select` / `Selector`.
BLoC input/output?	Events in → States out.
Cubit vs BLoC?	Cubit = methods+emit, no events.
Why Riverpod over Provider?	Compile-safe, no `BuildContext`, no `ProviderNotFound`.
Riverpod per-argument provider?	`family`.
Riverpod auto-cleanup?	`autoDispose`.
Riverpod tri-state async type?	`AsyncValue`.
Rebuild only on value change, no deps?	`ValueListenableBuilder`.
Two Provider subscribe mistakes?	`read` in build / `watch` in callback.
`notifyListeners()` in `build`?	Infinite rebuild loop — never.
Fix `setState after dispose`?	Guard with `if (!mounted)` + cancel subs.
Main GetX criticism?	Hidden global magic, poor testability.
bloc_test helper?	`blocTest(build/act/expect)`.
Riverpod test entry point?	`ProviderContainer` with `overrides`.
Default greenfield recommendation?	Riverpod (or BLoC for big event-driven teams).

Follow-up drills

1. **Design** the state layer for a shopping-cart feature (add/remove, quantities, promo codes, persisted across restart) in both Riverpod and BLoC — justify each layer's responsibilities.
2. **Optimize** a product-list screen that stutters while a header timer ticks every second — identify every over-rebuild and cut it without changing behavior.
3. **Debug** an app where leaving a details screen and returning shows stale data — pin it to a missing `autoDispose` / `dispose` and explain the lifecycle.
4. **Migrate** a mid-size Provider app to Riverpod incrementally — sequence the steps so the app stays shippable throughout.
5. **Refactor** three overlapping booleans (`isLoading`, `hasError`, `data`) into a single sealed/`AsyncValue` state and update all consumers.

6. **Justify** to a skeptical team lead why you would *not* adopt GetX for a 3-year enterprise app — and what you'd use instead and why.

Navigation & Routing — Interview Questions

> How Flutter moves between screens, from `Navigator.push` to declarative routing. For depth see the handbook modules 12 Navigation and 13 Routing.

Navigation is where interviewers probe whether you understand the difference between *imperative* screen pushing and *declarative* URL-driven state — the split that Navigator 2.0 and go_router exist to bridge. Expect it to escalate from "how do you pass an argument" to "how do you keep bottom-nav tab state and still support deep links".

🟢 Basic

1. What is the `Navigator` and how does the stack model work? `Navigator` is a widget that manages a stack of `Route` objects — one per screen. `push` adds a route on top (the new screen slides in), `pop` removes the top one (revealing the screen beneath and optionally returning a result). It's a LIFO stack, exactly like a browser history or an activity back-stack. The `Navigator` lives above your screens (installed by `MaterialApp` / `WidgetsApp`) and is reached via `Navigator.of(context)` or the static `Navigator.push(context, ...)` helpers.

2. How do you push and pop a screen?

```
Navigator.push(
  context,
  MaterialPageRoute(builder: (_) => const DetailsPage()),
);
Navigator.pop(context); // back to the previous screen
```

`push` takes a `Route`; `MaterialPageRoute` gives you the platform-correct transition (slide on Android, cupertino swipe on iOS) and a full-screen opaque page. `pop` unwinds one entry.

3. What is `MaterialPageRoute` and how does it differ from `PageRouteBuilder`? `MaterialPageRoute` is a ready-made modal route with Material transitions and a barrier. `PageRouteBuilder` is the low-level escape hatch that lets you define `pageBuilder`, `transitionsBuilder`, and `transitionDuration` yourself — use it for custom transitions (fade, scale, shared-axis). `CupertinoPageRoute` is the iOS-styled equivalent. All extend `PageRoute` → `ModalRoute`.

4. What are named routes and how do you register them? Named routes map a string to a builder, declared once in `MaterialApp`:

```
MaterialApp(
  routes: {
    '/': (_) => const HomePage(),
    '/details': (_) => const DetailsPage(),
  },
);
Navigator.pushNamed(context, '/details');
```

They centralize your route table and read like a sitemap. The downside: passing typed arguments is stringly-typed and awkward, which is why teams outgrow them.

5. How do you pass arguments to a named route, and read them back? Pass via the `arguments` parameter and read them with `ModalRoute.of(context)!.settings.arguments`:

```
Navigator.pushNamed(context, '/details', arguments: Product(id: 7));
// in DetailsPage:
final product = ModalRoute.of(context)!.settings.arguments as Product;
```

The argument is `Object?`, so you cast it — there's no compile-time safety. This is the classic pain point that `onGenerateRoute` and typed routers fix.

6. How do you return a result from a screen? `push` returns a `Future` that completes when the route is popped, and `pop` can carry a value:

```
final picked = await Navigator.push<Color>(
  context,
  MaterialPageRoute(builder: (_) => const ColorPicker()),
);
// in ColorPicker:
Navigator.pop(context, Colors.blue);
```

So navigation is naturally `async`: you `await` the pushed route to get what the user chose. Always guard `context` after the `await` with a `mounted` check.

7. What is `onGenerateRoute` and when do you use it? `onGenerateRoute` is a callback on `MaterialApp` that turns a `RouteSettings` (name + arguments) into a `Route`. It's more powerful than the static `routes` map because you can parse the name, extract path parameters, build typed arguments, and choose the transition — all in one place:

```
onGenerateRoute: (settings) {
  if (settings.name == '/details') {
    final id = settings.arguments as int;
    return MaterialPageRoute(builder: (_) => DetailsPage(id: id));
  }
  return null; // falls through to onUnknownRoute
},
```

It's the "one router function" pattern before go_router — useful for validation and centralized argument decoding.

8. What's the difference between `push`, `pushReplacement`, and `pushNamedAndRemoveUntil`? `push` adds on top. `pushReplacement` swaps the current top route for a new one (common for splash → home, so back doesn't return to the splash). `pushNamedAndRemoveUntil` pushes and pops everything until a predicate is true — e.g. after login, clear the whole stack and land on home with `(route) => false`.

9. What does `Navigator.pop` return to, and how do you go back multiple screens? `pop` returns to the route directly beneath. To skip several, use `popUntil((route) => route.isFirst)` or `popUntil(ModalRoute.withName('/home'))`. `maybePop` pops only if there's something to pop (and respects `onWillPop`-style callbacks), which is what the system back button triggers.

10. What is the difference between imperative and declarative navigation? Imperative (Navigator 1.0): you *command* the stack — "push this, pop that" — from event handlers. Declarative (Navigator 2.0 / go_router): you *describe* the stack as a function of app state — the list of pages is rebuilt whenever state changes, and the framework diffs it. Imperative is simpler for linear flows; declarative is necessary when navigation must sync with URLs, deep links, and restorable app state.

11. Why is the URL / browser address bar relevant to Flutter navigation? On Flutter web (and for deep links on mobile), the stack must be reconstructable from a URL string. Imperative pushes don't naturally serialize to a URL, so a user pasting `/products/7` or hitting the browser back button can't be handled cleanly. Declarative routing exists precisely to make "URL ⇄ page stack" a two-way, restorable mapping.

12. What is a `RouteSettings` object? It's the metadata attached to a route: its `name` and `arguments`. `MaterialPageRoute(settings: ...)` carries it, and `ModalRoute.of(context)!.settings` reads it. Named routes and `onGenerateRoute` both work by inspecting `RouteSettings`.

🟡 Intermediate

13. Why was Navigator 2.0 introduced — what could 1.0 not do? Navigator 1.0 is imperative and stack-only: the app can push/pop, but the *system* (browser, OS) can't tell the app "the user wants to be at this URL/route now". 1.0 couldn't cleanly handle web URL sync, browser

forward/back, or programmatically rewriting the entire stack from app state. Navigator 2.0 adds a declarative **Pages API** and a `Router` widget so the OS/browser can push route information *into* the app, and the app describes its whole stack declaratively.

14. Explain the core pieces of Navigator 2.0. | Piece | Job | |---|---| | `Router` | Widget that wires everything together; listens to route info | | `RouteInformationProvider` | Source of route info (the platform / browser URL) | | `RouteInformationParser` | Parses a `RouteInformation` (URL) into a typed app state, and back | | `RouterDelegate` | Builds the `Navigator` with a `pages` list from app state; handles pops | | `BackButtonDispatcher` | Routes the system back button to the right delegate |

The `RouterDelegate` holds the app's navigation state and calls `notifyListeners()` to rebuild the `Navigator` with a new `pages` list.

15. What is the Pages API? Instead of imperatively pushing routes, you give `Navigator` a declarative `pages: [...]` list of `Page` objects (`MaterialPage` , `CupertinoPage`) plus an `onDidRemovePage` (formerly `onPopPage`) callback. When app state changes, you rebuild the list; Flutter diffs old vs new pages by their `key` and animates the difference. This is the declarative model — the stack is derived from state, not mutated directly.

16. Why do people find raw Navigator 2.0 painful? It's extremely verbose and error-prone: you hand-write a `RouterDelegate` (extending `ChangeNotifier` + `PopNavigatorRouterDelegateMixin`), a `RouteInformationParser` , a state model, and page-diffing logic — dozens of lines of boilerplate for what 1.0 did in one call. Edge cases (nested routers, back dispatch, restoring state) are fiddly. That boilerplate is exactly why `go_router` (and others) were built on top of it.

17. What is `go_router` and why prefer it? `go_router` is the Flutter-team-endorsed declarative router that wraps Navigator 2.0. You declare a route tree with URL-style paths; it gives you deep-link/URL sync, redirects, nested navigation, and error handling without touching `RouterDelegate` . You keep the declarative benefits and lose the boilerplate:

```
final router = GoRouter(
  routes: [
    GoRoute(path: '/', builder: (_, __) => const HomePage()),
    GoRoute(
      path: '/product/:id',
      builder: (_, state) => ProductPage(id: state.pathParameters['id']!),
    ),
  ],
);
MaterialApp.router(routerConfig: router);
```

18. `context.go` vs `context.push` in go_router? `go` sets the location declaratively — it *replaces* the current navigation state with the stack implied by that path (used for tab switches, post-login redirects). `push` adds a page on top imperatively, keeping the current stack (used for drill-down like list → detail). Mixing them thoughtfully is common; overusing `push` breaks URL/state consistency.

19. How do you pass data and read path/query params in go_router? Path params come from the pattern (`/product/:id` → `state.pathParameters['id']`), query params from `state.uri.queryParameters` , and arbitrary objects via `extra` :

```
context.go('/product/7?ref=home');
context.push('/checkout', extra: cart);
// reader: state.extra as Cart
```

`extra` is not serialized into the URL, so it won't survive a deep link or web refresh — use it only for in-app object passing, and use path/query params for anything that must be linkable.

20. What is a redirect / route guard in go_router? `redirect` is a callback (global on `GoRouter` or per-route) that returns a new location string to bounce the user, or `null` to allow. It's how you implement auth gating:

```
redirect: (context, state) {
  final loggedIn = auth.isLoggedIn;
  final goingToLogin = state.matchedLocation == '/login';
  if (!loggedIn && !goingToLogin) return '/login';
  if (loggedIn && goingToLogin) return '/';
  return null;
},
```

Combine with `refreshListenable` so the router re-evaluates redirects when auth state changes.

21. What is `refreshListenable` and why is it needed? By itself, a `redirect` only runs on navigation events. If auth state changes *while the user sits on a page* (token expires, logout elsewhere), nothing re-triggers the guard. `refreshListenable: someListenable` makes go_router re-run redirects whenever that `Listenable` notifies — so logging out anywhere immediately bounces to `/login` .

22. What is `ShellRoute` and when do you use it? `ShellRoute` (and `StatefulShellRoute`) wraps a set of child routes in a persistent shell UI — typically a `Scaffold` with a `BottomNavigationBar` — so the shell stays mounted while the body swaps. The shell's `builder` receives the child navigator; child routes render into it. This is the idiomatic way to get a bottom nav where the app bar / nav bar persist across tabs.

23. How do you handle unknown routes / 404s? In Navigator 1.0, `onUnknownRoute` on `MaterialApp` catches names no route matched. In go_router, use `errorBuilder` (or `errorPageBuilder`) to render a not-found screen, and per-route exceptions surface there too. Always have one — otherwise a bad deep link crashes or shows a blank page.

24. What are typed routes in go_router? Typed routes (`go_router_builder`) generate route classes from annotated definitions so you navigate with compile-time-safe objects instead of raw strings:

```
@TypedGoRoute<ProductRoute>(path: '/product/:id')
class ProductRoute extends GoRouteData {
  const ProductRoute({required this.id});
  final int id;
  @override
  Widget build(BuildContext c, GoRouterState s) => ProductPage(id: id);
}
// navigate:
const ProductRoute(id: 7).go(context);
```

You get refactor-safe, typo-proof navigation and automatic param parsing — the answer to the "stringly-typed arguments" pain from Navigator 1.0.

🔴 Advanced

25. Explain the full Navigator 2.0 URL round-trip.

1. The platform reports a new URL via `RouteInformationProvider`.
2. `RouteInformationParser.parseRouteInformation` turns that `RouteInformation` into a typed app state (e.g. `AppRoutePath.details(7)`).
3. `RouterDelegate.setNewRoutePath` receives that state, updates its fields, and `notifyListeners()`.
4. `RouterDelegate.build` returns a `Navigator` whose `pages` list is derived from the state.
5. For the reverse (app → URL), `RouteInformationParser.restoreRouteInformation` serializes current state back into a URL so the address bar / OS stays in sync.

Being able to narrate this both directions is the senior-level Navigator 2.0 answer.

26. How do nested navigators work, and what breaks with them? A nested `Navigator` is a child `Navigator` widget below the root — used for tab-local stacks, wizards, or modal flows so back stays within that subtree. Pitfalls: (a) the system back button targets the root unless you wire a `BackButtonDispatcher` / `Router.of` correctly; (b) `Navigator.of(context)` picks the *nearest* navigator, so

you may push onto the wrong one — use `rootNavigator: true` when you need the root; (c) deep links must resolve to the correct inner navigator. `StatefulShellRoute` in go_router manages these nested navigators for you.

27. How do you preserve independent stack + scroll state per bottom-nav tab? Each tab needs its own `Navigator` so its back-stack survives tab switches, and the inactive tab's subtree must stay alive. Options: `IndexedStack` of per-tab navigators (keeps all mounted — simple but builds everything eagerly), or `StatefulShellRoute.indexedStack` in go_router (the modern answer — it manages a branch navigator per tab, preserving each branch's stack and state). Wrap scroll views with `PageStorageKey` so scroll offset is retained. The interviewer is checking you know a naive `body: pages[index]` throws away state on every switch.

28. How do deep links and app links actually get set up (high level)?

- **Android:** declare an `intent-filter` in `AndroidManifest.xml` for your scheme/host with `android:autoVerify="true"` for **App Links** (verified `https://` links, no chooser dialog). Verification needs an `assetlinks.json` at `https://yourdomain/.well-known/assetlinks.json` containing your app's SHA-256 fingerprint.
- **iOS:** enable **Associated Domains** (`applinks:yourdomain`) in Xcode capabilities and host an `apple-app-site-association` (AASA) file at `https://yourdomain/.well-known/` .
- **Flutter side:** with go_router, incoming links arrive as route information automatically; otherwise use a package like `app_links` / `uni_links` to receive the initial and streamed URIs. Custom URL schemes (`myapp://`) work without web verification but are less secure and can be hijacked.

29. Difference between deep links, universal/app links, and custom-scheme links? A *deep link* is any URL that opens a specific in-app location. A *custom-scheme* link (`myapp://product/7`) is app-owned but unverified — any app can claim the scheme. *Universal Links (iOS) / App Links (Android)* use real `https://` URLs verified against your domain via AASA/assetlinks, so the OS opens your app directly (or the web page if the app isn't installed) with no ambiguity. Interviewers want the verification/security distinction.

30. `WillPopScope` vs `PopScope` — what changed and why? `WillPopScope(onWillPop: () async => bool)` let you *asynchronously veto* a pop (e.g. "discard unsaved changes?"). It's deprecated because it's incompatible with **predictive back** (Android's back-gesture preview): the OS needs to know *up front* whether the pop will be blocked, but an async callback can't answer synchronously. `PopScope` replaces it with a synchronous `canPop` flag plus an `onPopInvokedWithResult(didPop, result)` callback:

```
PopScope(
  canPop: !hasUnsavedChanges,
  onPopInvokedWithResult: (didPop, result) {
    if (didPop) return;
    _showDiscardDialog(); // handle the blocked pop
  },
  child: form,
);
```

31. How do you implement auth-gated navigation robustly? Use a global `redirect` keyed on an auth `Listenable` wired via `refreshListenable`. Rules: unauthenticated + not on `/login` → `/login`; authenticated + on `/login` → home. Store the intended destination (`state.uri` in `?from=`) so you can bounce the user back after login. Keep the auth state outside the router (a notifier/BLoC) so both UI and redirects read one source of truth. This avoids flicker and race conditions from doing guarding imperatively inside `build`.

32. Why must you check `mounted` (or capture objects) after an `await` on navigation? `Navigator.push` returns a `Future`; while you `await` the result, the pushing widget may be disposed (user navigated away). Using its `context` afterward (for another push, `ScaffoldMessenger`, etc.) throws or leaks. Guard with `if (!mounted) return;` after the await, or capture `Navigator` / `ScaffoldMessenger` references *before* awaiting. The analyzer's `use_build_context_synchronously` lint flags exactly this.

33. How does the framework decide which pages to animate when the `pages` list changes? `Navigator` diffs the old and new `pages` lists by each `Page`'s `key` (a `LocalKey` / `ValueKey`). New keys → a push-in transition; removed keys → a pop-out; reordering by key reshuffles without recreating state. Forgetting stable keys causes wrong or missing transitions and lost state — so unique, stable page keys are essential in the declarative model.

34. How do you build a custom page transition and keep it consistent app-wide? Subclass `PageRouteBuilder` (or `CustomTransitionPage` in go_router) with your own `transitionsBuilder`, then apply it centrally — a shared factory or a `PageTransitionsTheme` in `ThemeData` so every `MaterialPageRoute` uses it per platform. Centralizing avoids one-off transitions scattered across `push` calls and keeps motion coherent.

35. What are the risks of using `extra` for navigation data, and the alternatives? `extra` is a plain in-memory `Object?` — it is **not** encoded in the URL, so it's `null` after a web refresh, a cold deep link, or state restoration, and it isn't type-safe. Use it only for transient in-app hand-offs. For anything linkable or restorable, encode identity in path/query params and re-fetch the object by id at the destination (single source of truth), or use typed routes for compile-time safety.

⚡ Rapid-fire (one-liners)

Question	Answer
What data structure is the Navigator?	A LIFO stack of `Route` s
How to get a result from a screen?	`await Navigator.push(...)` ; return via `pop(context, value)`
Push without keeping current screen?	`pushReplacement`
Clear the whole stack after login?	`pushNamedAndRemoveUntil(..., (r) => false)`
Read named-route arguments?	`ModalRoute.of(context)!.settings.arguments`
Turn a URL into a Route centrally?	`onGenerateRoute`
Two halves of Navigator 2.0 parser?	`RouteInformationParser` + `RouterDelegate`
Declarative stack list is called?	The Pages API (`pages: [...]`)
go_router: replace stack vs add page?	`context.go` vs `context.push`
Persistent bottom nav in go_router?	`StatefulShellRoute` / `ShellRoute`
Auth guard in go_router?	`redirect` + `refreshListenable`
Compile-time-safe routes?	Typed routes via `go_router_builder`
Replacement for `WillPopScope` ?	`PopScope` (`canPop` + `onPopInvokedWithResult`)
Verified https deep links called?	App Links (Android) / Universal Links (iOS)
Files that verify app links?	`assetlinks.json` / `apple-app-site-association`
Push onto the root navigator?	`Navigator.of(context, rootNavigator: true)`

Follow-up drills

1. **Design** a routing layer for a 3-tab app where each tab keeps its own back-stack and scroll position, and a deep link like `/orders/42` opens inside the correct tab — using `StatefulShellRoute` .
2. **Migrate** a legacy app from Navigator 1.0 named routes to go_router without breaking existing `pushNamed` call sites during the transition.
3. **Implement** end-to-end auth gating: unauthenticated users bounced to `/login` , redirected back to their intended destination after sign-in, and instantly ejected on token expiry from any screen.
4. **Set up** verified App Links / Universal Links for `https://shop.example.com/product/:id` and explain what breaks if the `assetlinks.json` fingerprint is wrong.
5. **Debug** a scenario where switching bottom-nav tabs resets each tab to its first screen and loses scroll — identify the cause and fix it.

6. **Optimize** a nested-navigator flow where the system back button closes the whole app instead of popping the inner stack.

Networking, Data & Storage — Interview Questions

> How a Flutter app talks to the network, serializes data, persists it, and stays usable offline. For depth see 16 Networking, 15 Local Storage, 20 Database, and 19 Offline First.

This topic tests whether you can move data across an unreliable boundary and store it correctly: HTTP mechanics, choosing a client, JSON codegen, error mapping to your domain, picking the right storage tier, and designing offline-first sync. Interviewers push from "how do you call an API" toward "how do you cache, retry, paginate, and resolve conflicts".

🟢 Basic

1. What is HTTP and what do the common methods/status codes mean? HTTP is a stateless request/response protocol over TCP (or QUIC for HTTP/3). Methods carry intent: `GET` (read, safe/idempotent), `POST` (create, non-idempotent), `PUT` (replace, idempotent), `PATCH` (partial update), `DELETE` (remove, idempotent). Status classes: `2xx` success (`200 OK` , `201 Created` , `204 No Content`), `3xx` redirect (`304 Not Modified`), `4xx` client error (`400` , `401` unauth, `403` forbidden, `404` , `409` conflict, `429` rate-limited), `5xx` server error. Knowing idempotency matters because it decides what's safe to retry.

2. How do you make a basic GET request with the `http` package?

```
import 'package:http/http.dart' as http;


final res = await http.get(Uri.parse('https://api.example.com/users/1'));
if (res.statusCode == 200) {
  final user = User.fromJson(jsonDecode(res.body) as Map<String, dynamic>);
}
```

`http` is a thin, official wrapper. You check `statusCode` yourself, decode `res.body` (a `String`) with `jsonDecode` , and map to a model. Anything non-2xx does *not* throw — you must branch on the code.

3. `http` vs `dio` — when do you reach for each? `http` is minimal and official: great for simple apps and packages that don't want a heavy dependency. `dio` is a full-featured client: interceptors, global config (`BaseOptions`), automatic JSON decoding, timeouts, `CancelToken` , retries, multipart/form-data, download progress, and better error objects (`DioException`). Reach for `dio` the moment you need cross-cutting concerns (auth token injection, logging, refresh-on-401) — building those by hand on `http` reinvents `dio` .

	`http`	`dio`
Interceptors	No	Yes
Auto JSON decode	No	Yes
Cancellation	No	`CancelToken`
Timeouts	Manual `.timeout()`	Built-in per-request
Best for	Simple/lightweight, packages	Real apps with cross-cutting concerns

4. What is JSON serialization and why can't you just use the decoded map directly? `jsonDecode` gives you `Map<String, dynamic>` / `List<dynamic>` — untyped, error-prone, and stringly-keyed. Serialization means converting that into typed Dart objects (`fromJson`) and back (`toJson`). You want typed models so the compiler catches field typos and type mismatches, and so the rest of the app never touches raw maps.

5. What does a manual `fromJson` / `toJson` look like?

```
class User {
  final int id;
  final String name;
  const User({required this.id, required this.name});

  factory User.fromJson(Map<String, dynamic> json) =>
      User(id: json['id'] as int, name: json['name'] as String);

  Map<String, dynamic> toJson() => {'id': id, 'name': name};
}
```

Fine for a handful of small models. The problems appear at scale: it's tedious, easy to get wrong, and every schema change means hand-editing two methods per model.

6. What is `shared_preferences` and what should it store? A key-value store backed by the platform's native prefs (`NSUserDefaults` / `SharedPreferences` /web `localStorage`). It holds *small, non-sensitive* primitives: a theme flag, onboarding-seen boolean, last tab, feature toggles. It is **not** encrypted, not for large data, and not a database. See 15 Local Storage.

7. When do you use `flutter_secure_storage` instead? For **secrets**: auth/refresh tokens, API keys, PII. It stores values in the platform Keychain (iOS) / Keystore-backed `EncryptedSharedPreferences` (Android), so data is encrypted at rest and survives outside plain prefs. Never put a JWT in `shared_preferences` — that's the classic security-review finding. See [37 Security].

8. When would you store data as plain files instead of prefs or a DB? For blobs and documents: cached images, downloaded PDFs, exported CSV/Excel, large JSON snapshots. Use `path_provider` to get the right directory (`getApplicationDocumentsDirectory` for user data that persists,

`getTemporaryDirectory` for disposable cache the OS can evict). Files are right when the data is large or binary and you don't need to query it.

9. What is SQLite / `sqflite` and when do you need a database at all? SQLite is an embedded relational DB; `sqflite` is the Flutter plugin exposing it via raw SQL. You need a DB (not prefs/files) when you have **structured, queryable, relational data at volume** — lists you filter/sort/paginate, entities with relationships, data you mutate transactionally. If you're serializing a growing list into a prefs string, that's the signal you outgrew prefs.

10. What's the difference between `jsonDecode` returning a `Map` vs a `List`? `jsonDecode` returns `dynamic`: an object `{...}` decodes to `Map<String, dynamic>`, an array `[...]` to `List<dynamic>`. For a list of models you map over it:

```
final data = jsonDecode(res.body) as List<dynamic>;
final users = data.map((e) => User.fromJson(e as Map<String, dynamic>)).toList();
```

Getting the cast wrong (`as Map` on an array) throws at runtime — a common bug.

11. What is a REST API in one line, and what makes it "RESTful"? REST models the server as **resources** addressed by URLs (`/users/1/orders`), manipulated with HTTP verbs, using stateless requests and standard status codes. "RESTful" means using verbs/codes semantically and keeping the server stateless (each request self-contained, auth carried per request), rather than tunneling everything through `POST /doStuff`.

12. How do you set a request timeout, and why must you? Networks stall; without a timeout a request hangs forever and your loading spinner never resolves. With `http`: `future.timeout(Duration(seconds: 10))`. With `dio`: `BaseOptions(connectTimeout:, receiveTimeout:, sendTimeout:)`. Always set both connect and receive timeouts — a fast connection that then stalls mid-body still needs to fail.

🟡 Intermediate

13. Why prefer `json_serializable` / `build_runner` over hand-written serialization? Code generation eliminates the tedium and the class of bugs manual mapping creates. You annotate `@JsonSerializable()`, declare fields, and `build_runner` generates `fromJson`/`toJson` in a `.g.dart` part. Benefits: fields and JSON stay in sync automatically, handles nested objects/lists/enums, supports renaming (`@JsonKey(name:)`), defaults, and custom converters — and it's compile-checked, so a schema change surfaces as a build error, not a runtime crash. The cost is a build step; worth it past a few models.

14. Show a `json_serializable` model and explain the workflow.

```
import 'package:json_annotation/json_annotation.dart';
part 'user.g.dart';

@JsonSerializable()
class User {
  final int id;
  @JsonKey(name: 'full_name') final String name;
  @JsonKey(defaultValue: false) final bool isActive;
  const User({required this.id, required this.name, required this.isActive});

  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

Run `dart run build_runner build --delete-conflicting-outputs` to generate `user.g.dart`. Use `watch` during development to regenerate on save.

15. What are interceptors and what do you use them for? Interceptors are hooks that run on every request/response/error, letting you centralize cross-cutting concerns instead of repeating them per call. Common uses: **attach the auth header**, **log** requests, **transform/normalize errors**, **refresh a token on 401 and retry**, add correlation IDs. In `dio`:

```
dio.interceptors.add(InterceptorsWrapper(
  onRequest: (o, h) { o.headers['Authorization'] = 'Bearer $token'; h.next(o); },
  onError: (e, h) async {
    if (e.response?.statusCode == 401) { /* refresh + retry */ }
    h.next(e);
  },
));
```

16. How do you implement token refresh on a 401 without a stampede? In an error interceptor, on 401 you refresh the token and retry the original request. The trap is *concurrent* 401s all firing refresh at once. Fix: serialize refresh with a single in-flight `Future`/lock so parallel failures await the *same* refresh, then retry with the new token. If refresh itself fails, clear the session and route to login. Keep the refresh token in `flutter_secure_storage`.

17. How do you cancel an in-flight request, and when do you need to? With `dio` you pass a `CancelToken`; calling `token.cancel()` aborts the request. You need it when a screen is disposed mid-fetch, or on **search-as-you-type** where each keystroke supersedes the last request (combine with debounce). Cancelling avoids wasted bandwidth, out-of-order responses overwriting newer state, and `setState after dispose`.

```
final token = CancelToken();
dio.get('/search', queryParameters: {'q': q}, cancelToken: token);
// on next keystroke / dispose:
token.cancel('superseded');
```

18. How should you handle and map errors from the network layer? Never leak `DioException` / `SocketException` /status codes into UI or business logic. Catch at the data-source/repository boundary and map to a **domain error type** (a sealed class or `Failure`): `NetworkFailure` (no connectivity/timeout), `ServerFailure(code)` (5xx), `UnauthorizedFailure` (401), `NotFoundFailure` (404), `ValidationFailure` (422 with field errors), `UnknownFailure`. The UI then switches on typed failures — testable, localizable, and decoupled from the transport. See [38 Error Handling].

19. Compare REST, GraphQL, gRPC, and WebSockets — when each?

Style	Shape	Best when
REST	Resource + verbs over HTTP, JSON	Default CRUD APIs, caching via HTTP, broad tooling
GraphQL	Single endpoint, client-specified query	Many clients with different data needs; avoid over/under-fetching; aggregating sources
gRPC	Binary Protobuf over HTTP/2, contract-first	Low-latency service-to-service, strict schemas, streaming, mobile-to-backend perf
WebSockets	Full-duplex persistent connection	Real-time push: chat, live prices, presence, collaborative editing

REST is the default. GraphQL solves over/under-fetching but you own caching/complexity. gRPC wins on payload size and speed but needs HTTP/2 and codegen. WebSockets are for server-initiated streams, not request/response CRUD.

20. `sqflite` vs Drift vs Isar vs Hive — what's the trade-off?

DB	Model	Type-safety	Relations/queries	Reactive	Best for
Hive	NoSQL key-value/box	Low (dynamic or adapters)	No joins	Watch boxes	Fast simple local cache, no SQL
Isar	NoSQL object DB	High (codegen)	Links, rich queries, indexes	Watchers	High-perf typed store, full-text search
sqflite	Raw SQL	Low (manual maps)	Full SQL, manual	No (poll)	You want plain SQL / know SQLite
Drift	SQL + typed Dart DSL on SQLite	High (codegen)	Full SQL, typed, joins	Streamed queries	Relational data with compile-safe queries + reactivity

Rule of thumb: simple key-value cache → Hive; typed object store with speed → Isar; relational/complex queries with SQL comfort and type safety → Drift; thin SQL wrapper → sqflite. See 20 Database.

21. Why not just store everything in `shared_preferences`? It's synchronous-feeling but actually async, unindexed, and loads the whole store into memory. There's no querying, no transactions, no relations, and no encryption. Serializing large/growing lists into a JSON string in prefs is an anti-pattern: reads/writes get slower, you lose partial updates, and you race concurrent writes. Past small primitives, move to a DB.

22. What does "single source of truth" mean and why does it matter for storage? Every piece of state has exactly one authoritative owner; everyone else derives from it. For offline-first, the **local database is the source of truth** — the UI reads from the DB (via a stream), and the network only *updates* the DB. This prevents the classic bug of UI reading sometimes-from-cache-sometimes-from-network with divergent shapes, and makes offline behavior automatic: no network just means the DB isn't refreshed. See 19 Offline First.

23. What is the repository pattern's role in the data layer? The repository is the seam between domain and data sources. It exposes domain-typed methods (`Future<List<Order>> getOrders()`), hides *where* data comes from (remote API, local DB, cache), decides the fetch strategy (cache-aside, network-then-cache), and maps DTOs↔domain entities plus errors↔failures. It lets you swap `dio` for something else, test the domain with a fake repo, and centralize caching/offline logic in one place.

24. What is cache-aside, and how do you implement it? Cache-aside (lazy loading): on read, check the cache first; on miss, fetch from source, populate the cache, return. On write, update the source and invalidate/update the cache. In a repository: read local DB → if present and fresh, return it → else fetch remote, write to DB, return. It's the default because it's simple and only caches what's actually requested.

🔴 Advanced

25. Design an offline-first read flow that stays live. Make the DB the single source of truth and have the UI subscribe to it, not the network:

1. UI watches a **stream** from the local DB (Drift/Isar watcher) → renders whatever's stored.
2. Repository triggers a background fetch; on success it **writes into the DB**, which pushes a new value down the stream automatically.
3. On network failure, the stream just keeps emitting the last stored data — the app works offline with no special casing. This decouples "showing data" from "refreshing data" and gives instant loads + eventual freshness. See 19 Offline First.

26. What is optimistic UI and what must you handle? Optimistic UI applies the user's change *locally and immediately* (before the server confirms), then reconciles. E.g. tapping "like" increments instantly, and you queue the network call. You must handle **rollback**: if the request

fails, revert the local change and surface an error; and **reconciliation**: replace the optimistic value with the server's authoritative one on success. It makes the app feel instant but requires you to track the pre-change state to undo.

27. How do you design a sync engine for offline writes? Queue mutations locally and replay them when online:

- Persist each local write as a pending operation (op type, payload, timestamp, entity id) in an **outbox** table, and apply it to the local DB optimistically.
- A sync worker (triggered by connectivity regain / periodic / on app resume) drains the outbox in order, calling the API.
- On success, mark the op done and store the server's canonical result; on failure, retry with backoff, and after N failures flag for conflict/manual resolution.
- Use **idempotency keys** (client-generated ids) so retries don't double-create. `connectivity_plus` detects reachability; but treat "connected" as a hint — the request can still fail.

28. How do you resolve sync conflicts? A conflict is when local and server versions of the same record both changed. Strategies: **last-write-wins** (compare timestamps/version numbers — simple, can lose data); **server-wins/client-wins** (policy-based); **field-level merge** (merge non-overlapping field changes); **CRDTs** (conflict-free types that merge deterministically, for collaborative apps); or **surface to the user** to pick. Version each record (monotonic version or `updatedAt`) so you can *detect* the conflict at all — optimistic concurrency: send the version you based your edit on, server rejects with 409 if it moved.

29. How does TTL-based caching work and where do you store the metadata? Attach an expiry to cached data: store `fetchedAt` alongside each record (or per collection). On read, if `now - fetchedAt > ttl`, treat as stale → refetch (either block, or serve-stale-then-revalidate). TTL suits data with a tolerable staleness window (product lists, feeds). Store the timestamp in the same DB row or a metadata table so eviction/freshness is queryable. Combine with cache-aside: cache-aside decides *where* to look, TTL decides *when it's too old*.

30. Explain ETag / `If-None-Match` conditional requests and their payoff. The server returns an `ETag` (a content hash/version) with a response. On the next request you send `If-None-Match: <etag>`; if the resource is unchanged the server replies `304 Not Modified` with **no body**, and you reuse the cached copy. Payoff: you still make the round-trip (so you know it's fresh) but save bandwidth and parsing on unchanged data. `Last-Modified` / `If-Modified-Since` is the timestamp-based equivalent. Great for large, occasionally-changing resources.

31. How do you implement pagination correctly, and offset vs cursor? Fetch pages on demand as the user scrolls (detect near-end via `ScrollController` or a sentinel item). Two schemes:

- **Offset/limit** (`?page=2&limit=20`): simple, allows jumping to a page, but **breaks under inserts/deletes** — items shift, causing duplicates or skips — and gets slow at large offsets.

- **Cursor/keyset** (`?after=<lastId/token>`): pass the last item's stable key; the server returns the next slice. Stable under concurrent writes and efficient — the standard for infinite feeds. Handle the states explicitly: loading-first-page, loading-more, end-reached, error-on-page (retry just that page). Dedupe by id when merging pages.

32. How do you keep pagination working offline / with a cache? Persist fetched pages into the DB keyed by their position/cursor and serve from the DB while background-refreshing. The UI pages over the local store; the network fills gaps. Tools like `infinite_query` /custom repositories cache page windows and reconcile with pull-to-refresh (which typically invalidates and refetches from the first page/cursor). The single-source-of-truth rule still holds: paginate the DB, sync the DB.

33. How do you handle large responses and JSON parsing without jank? `jsonDecode` on a big payload blocks the UI isolate and drops frames. Options: parse on a background isolate via `compute(parseUsers, body)` (or `Isolate.run` in modern Dart) so decode + model mapping happen off the main thread; stream/paginate so payloads stay small; use binary formats (Protobuf) to cut parse cost; and store parsed results in the DB so you parse once, not every read. Measure first — small payloads don't need an isolate.

34. How do you make the networking/data layer testable? Depend on abstractions and inject them: the repository takes a `RemoteDataSource` and `LocalDataSource` interface. Test with a mocked `dio` (via `MockAdapter` / `http_mock_adapter` or `mockito`) or fakes returning canned JSON/failures. Assert that: 4xx/5xx map to the right `Failure` , cache-aside reads local before remote, offline path serves stored data, and the outbox replays in order. Keeping HTTP details behind the data source means domain tests never touch the wire. See [49 Testing].

35. What are the security concerns in the networking/storage layer? Use HTTPS/TLS always; consider **certificate pinning** for high-value apps (pin in `dio` 's `HttpClientAdapter` , accept rotation). Store tokens in `flutter_secure_storage` , never in prefs or logs. Scrub sensitive fields from logging interceptors. Don't cache sensitive responses to plain files/DB unencrypted (Isar/Drift/SQLCipher support encryption). Validate and treat all server data as untrusted when deserializing. See [37 Security].

36. Walk through a complete "load a list" flow in a clean, offline-first architecture.

1. **UI** watches a `Stream<List<Order>>` from the repository → shows cached data instantly.
2. **Repository** returns the DB stream, and kicks off a refresh: calls `RemoteDataSource.fetchOrders()` .
3. **RemoteDataSource** (dio) hits the API; interceptor adds auth; response DTOs are parsed (on an isolate if big).
4. On success, repository writes DTOs→entities into the **local DB** (single source of truth) → the stream re-emits → UI updates.
5. On failure, map to `Failure` ; the stream keeps serving stored data, and the UI shows a non-blocking error/retry.

6. TTL/ETag decide whether the refresh was even necessary; pagination appends further pages into the same DB. This is the synthesis interviewers want — networking, serialization, storage, caching, and offline in one coherent path.

⚡ Rapid-fire (one-liners)

Q	A
Which HTTP methods are idempotent?	GET, PUT, DELETE (and safe: GET).
Does `http.get` throw on 404?	No — check `statusCode` yourself.
Main reason to pick `dio`?	Interceptors + built-in timeouts/cancel.
Cancel a request in dio?	`CancelToken` → `token.cancel()`.
Why codegen for JSON?	Kills boilerplate, stays in sync, compile-checked.
Command to generate models?	`dart run build_runner build`.
Where do auth tokens go?	`flutter_secure_storage`, never prefs.
What's `shared_preferences` for?	Small non-sensitive key-value primitives.
Store large binaries?	Files via `path_provider`.
Offset vs cursor pagination?	Offset shifts under writes; cursor is stable.
Single source of truth in offline-first?	The local database.
Cache-aside in one line?	Read cache → miss → fetch → populate → return.
What triggers a 304?	Matching `If-None-Match` / `If-Modified-Since`.
TTL caching stores what?	`fetchedAt` timestamp per record.
Optimistic UI must support?	Rollback on failure.
Prevent double-create on retry?	Idempotency key.
Real-time push protocol?	WebSockets.
Binary contract-first RPC?	gRPC (Protobuf/HTTP2).
Relational + typed + reactive DB?	Drift.
Fast NoSQL key-value cache?	Hive.
Parse huge JSON without jank?	Decode on an isolate (`compute` / `Isolate.run`).
Map transport errors to?	Domain `Failure` types at the repo boundary.

Follow-up drills

1. **Design** the full data layer for a news-feed app: infinite scroll, offline reading, pull-to-refresh, and article-read state that syncs — specify tables, cache strategy, and sync engine.

2. **Implement** dio token refresh that handles 20 concurrent 401s without firing 20 refresh calls — describe the locking.
3. **Debug** a search bar where fast typing shows results for an earlier query — identify the fix (cancellation + debounce + latest-wins).
4. **Optimize** a screen that janks for 400ms every time it loads a 2 MB JSON list — cut the frame drops without changing the API.
5. **Resolve** a sync conflict where a note was edited offline on two devices — compare last-write-wins, field merge, and CRDT approaches for this case.
6. **Choose** between Hive, Isar, Drift, and sqflite for (a) a 50-field relational reporting app and (b) a simple settings + favorites cache — justify both.

Dependency Injection — Interview Questions

> How objects get the collaborators they need instead of building them. For depth see the handbook module 14 Dependency Injection, especially 01_di_fundamentals.md and 02_get_it_and_injectable.md.

DI is a mid/senior signal: it exposes whether you understand testability, decoupling, and object lifetimes — not just which package you imported. Interviewers push from "what is DI" toward "why is a service locator a smell" and "how do you swap a fake in a test without touching production code".

🟢 Basic

1. What is dependency injection, in one line? Giving an object its dependencies from the outside instead of having it create them itself. Rather than `final api = ApiClient();` inside a class, you accept `ApiClient` as a constructor parameter and someone else decides which concrete instance to pass. DI is a *technique*; it does not require any library — passing a collaborator into a constructor is DI.

2. What is Inversion of Control (IoC), and how does DI relate to it? IoC is the general principle that a component does not control the creation or wiring of its collaborators — that control is *inverted* to an outer layer (a composition root, a framework, a container). DI is the most common concrete form of IoC: instead of the class pulling its dependencies, they are pushed to it. The Hollywood principle — "don't call us, we'll call you" — is IoC in a sentence.

3. Why bother with DI? What does it buy you?

- **Testability** — you can pass a fake/mock instead of the real network client, so tests are fast and deterministic.
- **Decoupling** — the class depends on an abstraction (`AuthRepository`) not a concrete type, so implementations can change without touching consumers.
- **Single wiring point** — object graphs are assembled in one place (the composition root) instead of scattered `new` calls.
- **Lifetime control** — one place decides whether a dependency is a singleton or created fresh each time.

4. What are the three kinds of injection?

- **Constructor injection** — dependencies passed as constructor params. Preferred: makes them explicit, required, and immutable (`final`).
- **Setter/property injection** — assigned after construction via a setter. Use only for genuinely optional dependencies.

- **Method injection** — passed into the specific method that needs them.

Constructor injection is the default answer; the others are exceptions.

5. Why is constructor injection preferred over the others? Because the dependencies become part of the type's public contract — you cannot construct the object in an invalid, half-wired state, and the compiler enforces that everything is provided. They can be `final`, so the object is immutable and thread-safe to read. And reading the constructor tells you exactly what the class needs, which surfaces "this class has 9 dependencies" as the design smell it is.

6. What is a service locator, and how does it differ from injection? A service locator is a global registry you *ask* for dependencies: `getIt<ApiClient>()`. With injection, dependencies are *pushed* to the object (it stays passive); with a locator, the object *pulls* them (it reaches out to a global). `get_it` is a service locator. The distinction matters because pulling hides dependencies — the constructor no longer tells you what the class needs.

7. What is `get_it`? A simple, fast service locator for Dart/Flutter. You register factories/singletons against a type at startup, then resolve them anywhere with `getIt<T>()` — no `BuildContext` needed. It is not a full DI container (no automatic constructor resolution); you wire the graph yourself in the registration calls. See 02_get_it_and_injectable.md.

```
final getIt = GetIt.instance;


void setup() {
  getIt.registerLazySingleton<ApiClient>(() => ApiClient());
  getIt.registerFactory<LoginBloc>(() => LoginBloc(getIt<AuthRepo>()));
}
```

8. What is a "composition root"? The single place — usually near `main()` — where the entire object graph is assembled and lifetimes are decided. Keeping wiring here means the rest of the code depends only on abstractions and never calls `new` on its collaborators. In a `get_it` app it's your `setupLocator()` / `configureDependencies()` function.

9. Does DI require a package in Flutter? No. Passing repositories and services down through constructors — or through the widget tree with `Provider` / `InheritedWidget` — is DI with zero extra dependencies. Packages like `get_it` and `injectable` reduce boilerplate for large graphs, but "DI" is a design idea, not a library.

10. How is Provider a form of DI? `Provider` injects a value into the widget subtree; descendants resolve it with `context.read<T>()` instead of receiving it through their own constructors. It's scoped DI keyed to the widget tree — the value lives exactly as long as the provider's position in the tree. It doubles as change-notification, but the "expose a dependency to a subtree" half is pure DI.

🟡 Intermediate

11. `registerSingleton` vs `registerLazySingleton` vs `registerFactory` in get_it?

Method	When instance is created	How many instances	Use for
`registerSingleton<T>(inst)`	**Eagerly**, at registration	One, shared	Something needed immediately / expensive to defer, must exist at startup
`registerLazySingleton<T>(() => ...)`	**Lazily**, on first `get<T>()`	One, shared thereafter	Most singletons — repositories, clients you may not touch on every launch
`registerFactory<T>(() => ...)`	**Every** `get<T>()` call	New each time	Stateful objects you want fresh — BLoCs/ViewModels tied to a screen

The default choice for shared, stateless services is `registerLazySingleton`; for per-use stateful objects, `registerFactory`.

12. What's the difference between `registerFactory` and `registerFactoryParam`?
`registerFactoryParam` lets you pass up to two runtime arguments into the factory at resolve time, e.g. `getIt<DetailBloc>(param1: productId)`. Plain `registerFactory` takes no runtime args. Use the param variant when an object needs data only known at the call site (a route argument) rather than at registration.

13. How do you make DI testable with get_it — resetting between tests? Call `await getIt.reset()` in `tearDown` (or `setUp`) so each test starts with a clean container, then register fakes for that test. For a single override you can `unregister<T>()` then re-register, or use `allowReassignment = true`. Because `get_it` is a global, forgetting to reset leaks registrations across tests and causes order-dependent failures.

```
setUp(() {
  getIt.registerSingleton<AuthRepo>(FakeAuthRepo());
});
tearDown(() => getIt.reset());
```

14. What are get_it scopes, and why use them? Scopes let you push a named layer of registrations onto a stack (`pushNewScope`) and pop it later (`popScope`), overriding or adding registrations for a bounded lifetime. Classic use: a **user session** — push a scope at login holding user-specific singletons, pop it at logout to dispose them all at once. Resolutions look through the stack top-down, so a scoped registration shadows a lower one.

15. What is `injectable` and how does it relate to get_it? `injectable` is a code-generation layer *on top of* `get_it`. You annotate classes (`@injectable`, `@singleton`, `@lazySingleton`) and it generates the `getIt.registerX` boilerplate via `build_runner`, resolving constructor dependencies for you. You still resolve through `getIt<T>()` at runtime — `injectable` only automates the registration wiring.

```
@module
abstract class RegisterModule {
  @lazySingleton
  Dio get dio => Dio();
}

@LazySingleton(as: AuthRepository)
class AuthRepositoryImpl implements AuthRepository {
  AuthRepositoryImpl(this._dio);
  final Dio _dio;
}
```

16. How do you register an interface-to-implementation binding? Register the concrete class *against the abstract type*: `getIt.registerLazySingleton<AuthRepository>(() => AuthRepositoryImpl(...))`. Consumers resolve `getIt<AuthRepository>()` and never know the implementation. In `injectable`, `@LazySingleton(as: AuthRepository)` does the same. This is what makes swapping a fake in tests a one-line change.

17. How does Riverpod act as a DI container? Every provider *is* a dependency registration: `final apiProvider = Provider((ref) => ApiClient());`, and other providers pull it with `ref.watch(apiProvider)`. Resolution is compile-safe (no runtime "not found"), lifetimes are controlled with `autoDispose`, and — crucially — tests inject fakes via `ProviderContainer(overrides: [...])` with no global mutation. It's a service locator that is type-safe and override-friendly. See 03_provider_riverpod_as_di.md.

18. Compare get_it and Riverpod as DI mechanisms.

Aspect	get_it	Riverpod
Kind	Global service locator	Provider graph (compile-safe locator)
Lookup safety	Runtime — throws if unregistered	Compile-time wiring, no `BuildContext` needed
Lifetime control	Singleton / lazy / factory / scopes	`autoDispose`, `family`, `keepAlive`
Test override	Global `reset` + re-register	`ProviderContainer(overrides:)` — isolated
Coupling to tree	None	None (providers are top-level)
Best for	Simple flat wiring, non-widget code	Apps already using Riverpod for state

19. Where do you resolve dependencies in Clean Architecture layers? Only at the **composition root** and the presentation entry points — never deep inside domain/data classes. Use cases receive repositories via constructor; repositories receive data sources via constructor; the locator wires `DataSource → Repository → UseCase → Bloc` once. Inner layers stay pure and locator-free, so the domain layer has no dependency on `get_it` at all. See 40 Clean Architecture.

20. How do you swap a fake/mock for a real dependency in a test? Because everything depends on an abstraction, you register (or override) the fake at the seam. With constructor injection you just pass it: `LoginBloc(FakeAuthRepo())`. With `get_it` you re-register `AuthRepository` with a fake after `reset()`. With Riverpod you add `authRepoProvider.overrideWithValue(FakeAuthRepo())`. No production code changes — that's the whole point of injecting the abstraction. See 05_testing_with_di.md.

21. What does object "lifetime" mean in DI and who should own it? Lifetime is how long a resolved instance lives and when it's disposed — per-resolve (factory), whole-app (singleton), or bounded (scope/session). The DI container owns lifetimes so consumers don't guess. Getting this wrong is a common bug: a factory-registered stream controller never disposed leaks; a singleton holding request-specific state bleeds data between users.

🔴 Advanced

22. Why is a service locator often called an anti-pattern despite get_it's popularity? Because it **hides dependencies**. A constructor tells you what a class needs; a class that calls `getIt<X>()` internally hides that it needs `X`, defeating the compiler and making the graph invisible until it fails at runtime. It also creates a global that couples every consumer to the locator, complicates test isolation (global state), and lets you resolve anything anywhere, which erodes layering. The pragmatic stance: use `get_it` *at the composition root and presentation edges only*, and still constructor-inject into your domain/data classes — get the ergonomics without smearing the locator through the codebase.

23. Constructor injection vs service locator — when is each actually appropriate? Constructor injection wherever a type is unit-tested or lives in a layer you want pure (domain, data) — dependencies stay explicit and mockable. Service locator at the outer boundary where wiring would otherwise be tedious (getting a BLoC into a widget without prop-drilling, accessing services from non-widget code like background isolates or notification handlers). The senior answer is "both": locate at the edge, inject through the core.

24. What problems do global mutable singletons cause?

- **Shared mutable state** — one caller mutates it, another reads unexpected data; the source of "works in isolation, fails together" bugs.
- **Test pollution** — state persists across tests unless explicitly reset, causing order-dependent flakiness.
- **Hidden coupling** — anything can reach the singleton, so the dependency graph is invisible.
- **Lifetime bugs** — a session-scoped value living as an app singleton leaks the previous user's data after logout.

Mitigations: make singletons *stateless* (or immutable), scope stateful ones to a session, and always `reset()` in tests.

25. What are "hidden dependencies" and why do they hurt? Dependencies a type acquires without declaring them — via a service locator call, a global, or reading a static. They hurt because the type's true contract is invisible: you can't tell what it needs to run, tests fail with cryptic "not registered" errors, and refactoring is dangerous because the compiler can't help. The fix is to surface them as constructor parameters so the type is honest about its needs.

26. How do you detect a circular dependency in a DI graph, and how do you break it? With `get_it`, a lazy singleton that resolves its own type during construction blows the stack or throws on first `get`. Break the cycle by: introducing an abstraction one side depends on instead of the concrete other; using lazy/deferred resolution (pass a `() => getIt<T>()` factory or resolve inside a method rather than the constructor); or — usually the real fix — extracting the shared concern into a third type both depend on. A cycle almost always signals a missing seam in the design.

27. How do you handle async dependencies that must be ready before the app runs? Register them with `registerSingletonAsync<T>(() async => await T.init())` and gate startup on `await getIt.allReady()`, or await the init in `main()` before `runApp`. Use `registerSingletonWithDependencies` to order async singletons that depend on each other. Example: `SharedPreferences.getInstance()` or a database `open()` must complete before repositories that use them are resolved.

28. How do you scope dependencies to a user session and dispose them on logout? Push a get_it scope at login holding user-specific singletons (`getIt.pushNewScope(scopeName: 'session', init: (g) {...})`) and `popScope()` at logout, which disposes everything registered in it. In Riverpod, put session state in `autoDispose` providers or a `family` keyed by user, and invalidate on logout. The goal is that no logged-in user's data can survive into the next session. See 04_scopes_and_lifetimes.md.

29. Why prefer `ProviderContainer(overrides:)` over resetting a global in Riverpod tests? Because each `ProviderContainer` is an isolated graph — overrides apply only to that container, so tests can't leak state into one another and can run in parallel. There's no global to forget to reset. This is the structural advantage Riverpod has over `get_it` for testing: isolation is the default, not something you bolt on with `tearDown`.

30. How does `injectable`'s environment feature help with DI? `@Environment('dev')` / `@Environment('prod')` (and the `@dev` / `@prod` / `@test` shortcuts) let you register different implementations per environment, selected by the environment string you pass to `configureDependencies`. So a `@Environment('test')` fake repository replaces the real one without conditional code in production. It moves environment branching into declarative annotations resolved at wiring time.

31. A widget deep in the tree needs a service. Constructor-inject it down, use Provider, or use get_it — how do you decide? Prop-drilling through many layers is noise and couples intermediate widgets to a type they don't use — avoid past a level or two. `Provider` / `InheritedWidget` is right when the value is genuinely tree-scoped (theme, a screen-scoped controller) and you want it disposed with that subtree. `get_it`/Riverpod is right for app-

wide services (repositories, clients) that aren't tied to any widget's position — resolve them at the point of use without threading them through the tree. Decision axis: is the lifetime tied to the widget tree (Provider) or the app/session (locator)?

⚡ Rapid-fire (one-liners)

Q	A
DI in one line?	Pass dependencies in instead of creating them inside.
DI vs IoC?	DI is a concrete form of IoC (control of wiring inverted outward).
Preferred injection type?	Constructor injection.
Injection vs service locator?	Push (passive object) vs pull (object asks a global).
Is get_it DI or a locator?	A service locator.
Lazy vs eager singleton?	Created on first `get` vs at registration.
Factory registration gives you?	A new instance every resolve.
Register a BLoC per screen as?	`registerFactory`.
Reset get_it between tests?	`await getIt.reset()` in tearDown.
get_it scopes are for?	Bounded lifetimes, e.g. a user session.
What is injectable?	Codegen registration layer on top of get_it.
Bind interface to impl in injectable?	`@LazySingleton(as: AbstractType)`.
Async singleton before startup?	`registerSingletonAsync` + `await allReady()`.
Riverpod as DI?	Providers are registrations; `ref.watch` resolves.
Override a dep in Riverpod test?	`ProviderContainer(overrides:)`.
Main service-locator smell?	Hidden dependencies / global coupling.
Where to resolve in Clean Arch?	Composition root + presentation edge only.
Swap a fake in a test?	Override the abstraction at the seam — no prod change.
Fix a circular dependency?	Add an abstraction or defer resolution; extract shared type.

Follow-up drills

1. **Design** the DI wiring for a Clean Architecture feature (data source → repository → use case → bloc) using `get_it` + `injectable` — show the composition root and where each lifetime is chosen.
2. **Refactor** a class that calls `getIt<X>()` in three methods into constructor injection, and explain what testability you just gained.

3. **Debug** a test suite that passes individually but fails when run together — trace it to un-reset global singleton state and fix it two different ways.
4. **Migrate** an app's DI from a hand-rolled `get_it` setup to `injectable` codegen incrementally without a big-bang rewrite.
5. **Implement** session-scoped dependencies that are created at login and fully disposed at logout, and prove no data leaks into the next session.
6. **Justify** to a teammate why you'd keep `get_it` only at the edges while constructor-injecting through the core — with a concrete example where the pure-locator approach bites.

Animations & Custom UI — Interview Questions

> How Flutter animates over time, paints arbitrary pixels, and adapts to every screen and platform. Depth in 22 Animations, 23 Custom Painting, 24 Responsive UI, and 25 Adaptive UI.

This topic tests whether you know when to reach for implicit vs explicit animations, can manage an `AnimationController` lifecycle without leaking tickers, understand what actually makes an animation jank, and can drop to `CustomPainter` / `RenderObject` or reshape a UI across breakpoints and platforms when the framework widgets aren't enough.

🟢 Basic

1. What is the difference between implicit and explicit animations? Implicit animations (`AnimatedContainer`, `AnimatedOpacity`, `AnimatedPadding`, `TweenAnimationBuilder`) animate automatically whenever you rebuild with a new target value — you set the new value and a `duration`, and Flutter interpolates for you. **Explicit** animations (`AnimationController` + `Tween` + `AnimatedBuilder`/transitions) give you a handle you drive manually: forward, reverse, repeat, stop, seek. Rule of thumb: use implicit for simple one-shot A→B transitions triggered by state, explicit when you need control over playback, looping, coordination, or reuse.

2. How does `AnimatedContainer` work? You give it a `duration` (and optional `curve`) and rebuild it with different property values (color, width, padding, alignment…). On each rebuild it detects which properties changed and animates from the old value to the new over the duration. Internally it's an `ImplicitlyAnimatedWidget` backed by a hidden `AnimationController`, so you get animation with zero controller boilerplate. The trigger is just `setState` with new values.

3. What is `TweenAnimationBuilder` and when do you use it? It's the implicit animation for a *single custom value* when there's no ready-made `AnimatedX` widget. You give it a `Tween`, a `duration`, and a `builder(context, value, child)`; whenever the tween's `end` changes it animates from the current value to the new end and rebuilds via the builder. Use it for one-off custom interpolations (e.g. animating a progress arc or a number counting up) without wiring an `AnimationController`.

4. What are the core objects in an explicit animation?

- `AnimationController` — drives a value from 0.0→1.0 over a duration; produces ticks, controls playback.
- `Tween<T>` — maps that 0→1 into a real range (e.g. `Tween<Color>`, `Tween<Offset>`, `Tween<double>`), via `tween.animate(controller)`.
- `Curve` — reshapes the 0→1 timeline (ease-in, bounce…) via `CurvedAnimation`.
- `Animation<T>` — the read-only value + status you listen to.
- `AnimatedBuilder` / `*Transition` — rebuilds only the animating subtree each tick.

5. What is a `Tween` and what does `.animate()` do? A `Tween<T>` defines a `begin` and `end` and knows how to `lerp` between them. `tween.animate(controller)` returns an `Animation<T>` whose value tracks the controller's 0→1 progress mapped into `[begin, end]`. Tweens hold no state themselves — they're just interpolation recipes, so you can reuse one across controllers.

6. What is a `Curve` and why use one? A `Curve` remaps the linear 0→1 progress into a non-linear one — `Curves.easeInOut`, `Curves.bounceOut`, `Curves.elasticIn`, etc. Real motion is rarely linear; curves make animations feel natural (accelerate then settle). Apply with `CurvedAnimation(parent: controller, curve: Curves.easeOut)` or pass `curve:` to implicit widgets. You can also give a separate `reverseCurve`.

7. What is `AnimatedBuilder` and why is it preferred over listening + `setState`? `AnimatedBuilder(animation: ..., builder: ...)` rebuilds only its `builder` on every tick, scoping the rebuild to the animating widget instead of the whole `State`. Its `child` argument lets you pass a subtree that's built **once** and reused each frame (not rebuilt), so expensive static children aren't rebuilt 60 times a second. Manually calling `setState` in a listener rebuilds the entire `build()` method — wasteful.

8. What is a Hero animation? A Hero animation is a shared-element transition between two routes: wrap a widget in `Hero(tag: 'x', child: ...)` on both the source and destination pages with the **same tag**, and during navigation Flutter flies the element from its old position/size to the new one. The `Navigator` and `HeroController` handle it automatically. Tags must be unique per screen; a duplicate tag on one screen throws.

9. What is `vsync` and why does an `AnimationController` need it? `vsync` takes a `TickerProvider`. The controller needs a `Ticker` that fires once per frame (driven by vsync) to advance the animation. Passing `vsync: this` binds the ticker to the widget so it's **muted when the widget is off-screen** (e.g. in an inactive route), preventing animations from burning CPU when nobody can see them. Without vsync there'd be no frame signal to step the animation.

10. `SingleTickerProviderStateMixin` vs `TickerProviderStateMixin` — which do you use? Add the mixin to your `State` so it can be the `TickerProvider` (`vsync: this`). Use `SingleTickerProviderStateMixin` when the State drives exactly **one** `AnimationController` — it's lighter. Use `TickerProviderStateMixin` when you have **multiple** controllers in the same State. Using the single variant with two controllers throws at runtime.

11. Why must you dispose an `AnimationController`? The controller holds a `Ticker` registered with the framework. If you don't call `controller.dispose()` in the State's `dispose()`, the ticker keeps ticking after the widget is gone — a leak that Flutter flags in debug ("A Ticker was being disposed but was still active" / disposed-controller-still-in-use). Always: create in `initState`, dispose in `dispose`.

12. What's the difference between MediaQuery and LayoutBuilder for responsive UI? `MediaQuery.of(context)` gives you **screen/window-level** info: full size, orientation, padding/insets (safe area, keyboard), text scale, device pixel ratio. `LayoutBuilder` gives you the **constraints of the parent slot** the widget actually occupies — which can be much smaller than the screen. Use MediaQuery for global decisions (phone vs tablet layout, keyboard avoidance); use LayoutBuilder to adapt a widget to the space it's *given* (e.g. a card that reflows inside a column).

13. What does "adaptive" UI mean versus "responsive" UI? Responsive = adjusting layout to *size* (columns, spacing, breakpoints) — same platform, different screen. **Adaptive** = adjusting behavior/look to the *platform or device conventions* — Material on Android/web, Cupertino on iOS; a swipe-back gesture on iOS, a mouse hover state on desktop. Responsive is about geometry; adaptive is about platform idioms. Good apps do both.

🟡 Intermediate

14. Walk through the full lifecycle of an `AnimationController`.

1. In `initState`, create it: `_c = AnimationController(vsync: this, duration: ...)`.
2. Build `Animation`s from it via tweens/curves (`_c.drive(tween)` or `tween.animate(_c)`).
3. Start playback: `forward()`, `reverse()`, `repeat()`, `animateTo()`, `fling()`.
4. React via `addListener` (value changed) and `addStatusListener` (completed/dismissed/forward/reverse).
5. In `dispose`, call `_c.dispose()`. The controller's value stays in `[lowerBound, upperBound]` (default 0..1) and `status` reports where it is.

15. How do you build a staggered animation? Use **one** `AnimationController` and give each animated property its own `Interval` curve so pieces start/finish at different fractions of the same timeline:

```
final fade = CurvedAnimation(parent: _c, curve: const Interval(0.0, 0.5, curve: Curves.easeIn));
final slide = CurvedAnimation(parent: _c, curve: const Interval(0.4, 1.0, curve: Curves.easeOut));
```

Sharing one controller keeps the phases perfectly in sync and cheap. Driving each with its own controller is possible but harder to coordinate and easier to desync.

16. How do physics-based animations differ from duration-based ones? Duration-based animations run for a fixed time regardless of state. **Physics-based** ones (`SpringSimulation`, `FrictionSimulation`, `GravitySimulation` via `controller.animateWith(simulation)`) run until the *simulation settles* — duration emerges from physics (mass, stiffness, velocity), not a clock. They feel natural for gesture continuations because you can hand off the gesture's release velocity into a spring so motion continues seamlessly.

17. How do you make a gesture-driven animation (drag to dismiss / swipe)? Bind an `AnimationController` to the gesture instead of playing it on a timer: in `onPanUpdate` set `_c.value += delta/extent` to track the finger, and in `onPanEnd` call `_c.fling(velocity: ...)` or `animateWith(SpringSimulation(...))` using the release velocity to settle to the nearest snap point. The controller becomes the single source of truth for position, so the finger-follow and the release animation are the same value stream. `Draggable`, `Dismissible`, and `DraggableScrollableSheet` package this pattern.

18. What causes an animation to jank, and how do you fix it? Jank = a frame missed the budget (~16 ms at 60 Hz). Common causes:

- **Rebuilding too much per tick** — use `AnimatedBuilder` with a cached `child`, not `setState` on the whole tree.
- **Expensive work inside `build` / `paint`** running every frame — precompute outside the tick.
- **Repaint spilling into neighbors** — wrap the animating subtree in a `RepaintBoundary` so it composites on its own layer.
- **`saveLayer`, big images, blurs, shader compile jank** on the raster thread. Diagnose in DevTools: is the UI thread or the raster thread over budget? Fix that one.

19. How does `RepaintBoundary` help an animation? It isolates the animating subtree onto its **own compositing layer**. When the animation repaints, only that layer is re-rasterized; the static content around it reuses its cached layer, and vice-versa. This cuts raster-thread work for localized animation (a spinner, a pulsing badge). It costs extra memory per layer, so add it where an animation repaints frequently — not everywhere.

20. Why prefer transition widgets like `FadeTransition` / `SlideTransition` / `ScaleTransition` over animating in `setState`? The `*Transition` widgets listen to the `Animation` and rebuild/repaint only what's necessary — often just re-applying an opacity or transform at the **paint/composite** layer without rebuilding the child. That avoids re-running `build()` for the subtree each frame. `setState`-per-tick rebuilds the entire widget and its children 60×/s. Same visual result, far less work.

21. What is `shouldRepaint` in a `CustomPainter` and why does it matter? When a `CustomPaint` rebuilds, Flutter calls `shouldRepaint(oldDelegate)`; return `true` only if the painter's inputs changed, else `false` to **skip repainting** and reuse the cached picture. Getting this wrong is a classic bug: return `true` always and you repaint needlessly (jank); capture inputs incorrectly and you paint stale visuals. Compare the fields that actually affect the drawing.

22. How do you drive a `CustomPainter` from an animation efficiently? Pass the `Animation` to the painter's `repaint:` constructor argument (`CustomPainter(repaint: animation)`), and base `shouldRepaint` on it. The painter then repaints on each tick **without rebuilding** the widget tree at all — the `CustomPaint` subscribes to the listenable directly. This is the cheapest way to animate custom drawing.

23. How do you make a layout responsive with breakpoints? Pick width thresholds (common: ~600 for tablet, ~840/1024 for desktop) and switch layouts:

```
final w = MediaQuery.sizeOf(context).width;
if (w >= 1024) return DesktopLayout();
if (w >= 600)  return TabletLayout();
return MobileLayout();
```

Prefer `MediaQuery.sizeOf(context)` (Flutter 3.10+) over `MediaQuery.of` so you only rebuild on size changes. For per-widget adaptation use `LayoutBuilder`. Flutter's own `NavigationRail` vs `BottomNavigationBar` swap is a canonical breakpoint use.

24. How do you build an adaptive (platform-aware) UI? Branch on the platform and use the matching design language: `Theme.of(context).platform`, `Platform.isIOS` (mobile), or `defaultTargetPlatform`. Options:

- Use `.adaptive` constructors (`Switch.adaptive`, `Checkbox.adaptive`, `CircularProgressIndicator.adaptive`) that render Cupertino on iOS and Material elsewhere.
- Swap whole widgets (`CupertinoPageScaffold` / `CupertinoButton` vs `Scaffold` / `ElevatedButton`).
- Use `defaultTargetPlatform` (not `Platform`) so it works on web too, and so you can override it in tests.

25. `MediaQuery.of(context)` vs `MediaQuery.sizeOf(context)` — why does it matter? `MediaQuery.of` subscribes the widget to the **entire** `MediaQueryData`, so it rebuilds on any change (keyboard insets, text scale, size…). The `.sizeOf`, `.paddingOf`, `.orientationOf` aspect accessors subscribe to just that slice, so the widget rebuilds only when *that* value changes. Use the narrow accessor to avoid needless rebuilds — e.g. a layout that only cares about width shouldn't rebuild when the keyboard opens.

🔴 Advanced

26. When would you drop from `CustomPainter` to a custom `RenderObject`? `CustomPainter` only participates in the **paint** phase — it can't influence layout (it paints within the size it's given). Write a custom `RenderObject` (`RenderBox`) when you need custom **layout** (measuring children, non-standard sizing under constraints), custom **hit testing**, multiple children with bespoke positioning, or intrinsic dimensions — things paint alone can't do. It's the most work and least ergonomic, so reserve it for genuinely novel layout/interaction (custom charts, editors, layout algorithms) after `CustomPaint`, `Flow`, and composition fail.

27. Explain the `Canvas` operations you use in a `CustomPainter` and their cost. `canvas.drawPath/drawRect/drawCircle/drawRRect` with a `Paint` (style, color, `strokeWidth`, `shader`, `maskFilter`). Build complex shapes with `Path` (moveTo/lineTo/cubicTo/arcTo/close). Gradients go

on `Paint.shader` via `ui.Gradient.linear/radial/sweep` (or `LinearGradient(...).createShader(rect)`). Costs: solid fills are cheap; `MaskFilter` blurs, `saveLayer`, and large paths are expensive on the raster thread. Prefer drawing directly over `saveLayer` when possible.

28. What does `saveLayer` do and why is it expensive? `canvas.saveLayer(bounds, paint)` allocates an **offscreen buffer**, draws subsequent commands into it, then composites that buffer back with the paint's opacity/blend/filter. It's needed for group opacity, blend modes across multiple draws, and some clips. It's costly because it's an extra allocation + render target switch + composite on the raster thread — a frequent source of raster jank. Widgets like `Opacity` and `ShaderMask` use it internally, which is why animating `Opacity` over a big subtree janks (prefer `FadeTransition` / `AnimatedOpacity`, or `opacity` on an image).

29. How do fragment shaders work in modern Flutter? You author a **GLSL fragment shader** (`.frag`), declare it under `flutter: shaders:` in `pubspec.yaml`, load it with `FragmentProgram.fromAsset`, set its uniforms, and use the resulting `FragmentShader` as a `Paint.shader` inside a `CustomPainter` (or via `ShaderMask`). It runs per-pixel on the GPU — great for gradients, noise, ripples, dissolves, and effects too expensive to compute per-pixel in Dart. Uniforms (time, resolution, pointer) let you animate them. Under Impeller shaders are compiled ahead-of-time, which also removes the old "shader compilation jank" on first run.

30. What is shader compilation jank and how is it addressed? On the older Skia backend, a shader used for the first time was compiled **at draw time on the raster thread**, stalling that frame — a visible hitch the first time an effect/animation appeared. Historically mitigated with **SkSL warm-up** (bundling precompiled shaders captured via `--cache-sksl`). The modern fix is **Impeller** (default on iOS and Android), which precompiles its shaders during build, largely eliminating the class of jank.

31. How do you coordinate multiple animations that must stay in sync? Prefer a **single `AnimationController`** as the clock and derive each animation from it with different `Tween`s + `Interval`s/`Curve`s (the staggered pattern). One ticker, one timeline, guaranteed sync, minimal overhead. Reach for multiple controllers only when phases are truly independent (different triggers/durations/lifecycles); then you coordinate via status listeners (`chain` on completion) or `TickerFuture` (`await _c.forward()`). Multiple controllers = multiple tickers = more per-frame cost and desync risk.

32. How does the `Hero` flight actually work, and how do you customize it? On a route push, `HeroController` (installed by `MaterialApp`) finds matching tags on the outgoing and incoming routes, measures both rects, and inserts an **overlay** widget that animates (via a `RectTween` and the route's transition) from source to destination rect, hiding the real widgets during flight. Customize with `flightShuttleBuilder` (control what's drawn mid-flight — e.g. cross-fade), `createRectTween` (arc vs linear path — `MaterialRectArcTween` gives a curved flight), and `placeholderBuilder`. Pitfalls: duplicate tags on one screen throw; heroes need the same tag and should wrap similarly-shaped widgets.

33. Your list scroll janks only on low-end devices during a fade-in animation. How do you approach it? Reproduce in **profile mode on that class of device**, open DevTools Timeline. Determine thread: if the **UI thread** overruns, the animation is rebuilding too much — move to `AnimatedBuilder` with cached `child`, cut work in `build`, ensure item widgets are `const` where possible. If the **raster thread** overruns, suspect `Opacity` / `saveLayer` over large subtrees (switch to `FadeTransition`), unbounded image sizes (add `cacheWidth`), missing `RepaintBoundary` around the animating cell, or shader jank (verify Impeller). Fix the measured bottleneck, then re-measure — don't sprinkle boundaries blindly.

34. How do you animate between two entirely different layouts (not just a property)? Options by complexity: `AnimatedSwitcher` (cross-fade/scale swapping child widgets by key), `AnimatedCrossFade` (between exactly two children with size interpolation), `Hero` (shared elements across routes), or `Flow` / `CustomMultiChildLayout` driven by a controller for bespoke reflow. For implicit multi-property morphs, `AnimatedContainer` / `TweenAnimationBuilder`. When the two states share elements, prefer a shared-element approach so items move rather than fade — it reads as continuity, not replacement.

35. How do `AnimatedList` / `SliverAnimatedList` handle insertions and removals? They animate items in/out by requiring you to keep the list and a `GlobalKey<AnimatedListState>` in sync: call `listKey.currentState!.insertItem(index)` / `removeItem(index, builder)` **alongside** mutating your backing data. On removal you must supply a builder because the data is already gone — the widget needs to know how to paint the departing item during its exit animation. Getting the data and the animated-list calls out of sync causes range errors or ghost items.

36. When is a custom `RenderObject`'s `performLayout` the right tool, and what must it respect? When your sizing/positioning logic can't be expressed by existing layout widgets. In `performLayout` you must: read the incoming `constraints`, lay out each child with `child.layout(childConstraints, parentUsesSize: true)`, position children via their `parentData.offset`, and set your own `size` **within** the constraints (`constraints.constrain(...)`). Respect the constraint contract (constraints down, sizes up, parent sets position) or you get overflow/assertion errors. Also implement `hitTest`, `paint`, and often `computeDryLayout`/intrinsics.

⚡ Rapid-fire (one-liners)

Q	A
Implicit animation for one custom value?	`TweenAnimationBuilder`.
What drives an explicit animation's time?	`AnimationController` (ticker/vsync).
One controller, multiple properties, different timing?	`Interval` curves (staggered).
Mixin for one controller?	`SingleTickerProviderStateMixin`.
Where do you dispose a controller?	`State.dispose()`.
Widget that rebuilds only the animating subtree?	`AnimatedBuilder` (with cached `child`).
Shared-element route transition?	`Hero` (same `tag`).
Run animation until physics settles?	`animateWith(SpringSimulation(...))`.
Cheapest way to repaint a `CustomPainter` per tick?	Pass `repaint: animation`.
`shouldRepaint` returns...?	`true` only if paint inputs changed.
Group opacity offscreen buffer op?	`canvas.saveLayer`.
Isolate animation repaint onto its own layer?	`RepaintBoundary`.
Screen size vs parent slot size?	`MediaQuery.sizeOf` vs `LayoutBuilder`.
Rebuild only on size change?	`MediaQuery.sizeOf(context)`.
iOS/Material auto-switching control?	`.adaptive` constructors.
Platform check that also works on web/tests?	`defaultTargetPlatform`.
Backend that kills shader-compile jank?	Impeller.
Need custom layout, not just paint?	Custom `RenderObject`.

Follow-up drills

1. Design a pull-to-refresh with a custom `CustomPainter` indicator whose arc grows with drag distance and spins during load — wire the gesture, controller, and painter.
2. Implement a staggered "cards fly in" onboarding screen using a single `AnimationController` and `Interval` curves; explain how you'd tune the timing.
3. Debug: an `Opacity`-wrapped image gallery janks on scroll only on a 3-year-old Android phone. Walk through your DevTools investigation and the fix.
4. Build a screen that shows a `NavigationRail` on desktop, a bottom nav on phone, and Cupertino tab bar on iOS — responsive *and* adaptive in one widget.
5. Optimize a `CustomPainter` charting 10k points that repaints on every animation tick; get it under the frame budget.

6. Implement a swipe-to-dismiss card stack (Tinder-style) with gesture-driven position and spring settle on release, handing the fling velocity into the simulation.

App Architecture — Interview Questions

> How you structure a Flutter app so it stays testable, changeable, and scalable as teams and features grow. For depth see 40 Clean Architecture, 43 MVVM, 44 Feature First Architecture, 45 Modular Architecture, and 46 Domain Driven Design.

Architecture is the highest-leverage senior/lead signal: it tests whether you can reason about *change* — where a new requirement lands, what a UI-framework swap costs, how a team of ten avoids stepping on each other. Interviewers push from "what is Clean Architecture" toward "when would you *not* use it" and "how do you enforce boundaries at scale".

🟢 Basic

1. Why does architecture matter — isn't a working app enough? A working app is table stakes; architecture is about the cost of the *next* change. Good structure makes features land in predictable places, makes code testable without a device, isolates volatile decisions (which HTTP client, which DB, which state library) from stable business rules, and lets multiple people work without merge chaos. The payoff is not day-one velocity — it's month-twelve velocity, when an unarchitected app has calcified into a big ball of mud where every change risks breaking three unrelated screens.

2. What is "separation of concerns"? Each part of the code has one reason to exist and one axis of change. UI code worries about pixels and gestures; business logic worries about rules and workflows; data code worries about I/O, serialization, and caching. When these are tangled — an HTTP call inside a widget's `build`, a JSON key referenced in the UI — a change to one concern forces edits across many files and makes isolated testing impossible. Separation is the precondition for every other architectural benefit.

3. What is the dependency rule? Source-code dependencies point *inward*, toward higher-level policy, and never outward toward detail. Concretely: the domain (business rules) knows nothing about the UI or the database; the data layer depends on the domain, not vice versa. Inner layers define interfaces (e.g. a `UserRepository` abstract class); outer layers implement them. This is what lets you swap Dio for `http`, or REST for GraphQL, or Provider for BLoC, without touching your business logic.

4. What are the layers in Clean Architecture? Three concentric layers:

- **Presentation** — widgets + state holders (ViewModel/BLoC/Notifier). Renders state, dispatches user intent.
- **Domain** — entities, use cases, and repository *interfaces*. Pure Dart, no Flutter, no packages. The stable core.

- **Data** — repository *implementations*, remote/local data sources, DTOs, and mappers. Talks to the outside world.

Presentation and data both depend on domain; domain depends on nothing.

5. What is a use case (interactor)? A single, named unit of application-specific business logic — `LoginUser`, `AddToCart`, `FetchOrderHistory`. It orchestrates entities and repositories to accomplish one user goal, exposing something like a `call()` method. Use cases make the app's capabilities readable as a list, keep the ViewModel thin, and are trivially unit-testable with mocked repositories. In simpler apps you may collapse them and let the ViewModel call the repository directly — that's a legitimate right-sizing choice.

6. What is an entity vs a DTO/model? An **entity** is a domain object expressing a business concept and its invariants (a `User` with an `id` and rules about what a valid user is) — it lives in the domain layer and has no serialization concerns. A **DTO** (data transfer object / data model) mirrors an external shape — a JSON payload or DB row — with `fromJson` / `toJson`. Keeping them separate means an API rename (`user_name` → `login`) changes only the DTO and its mapper, never the entity or the UI that consumes it.

7. What is a repository? An abstraction that hides *where* data comes from behind a domain-friendly interface. The domain declares `abstract class ProductRepository { Future<List<Product>> getProducts(); }`; the data layer implements it, coordinating a remote data source, a local cache, and mappers. The presentation layer asks for products and neither knows nor cares whether they came from the network, SQLite, or memory. It's the seam where the dependency rule is enforced.

8. What is a mapper and why not skip it? A mapper converts between DTOs and entities (`UserDto.toEntity()` / `User.fromDto()`). Skipping it and using JSON models straight through the app couples your UI to the wire format — every backend change ripples to widgets, and you can't shape data to what the domain actually needs. The mapper is a small, cheap boundary that localizes the blast radius of external change.

9. What does MVC stand for, and what's the problem with it in Flutter? Model–View–Controller: Model holds data, View renders, Controller handles input and updates the model. In Flutter the View (widget tree) and Controller boundary is blurry — widgets already handle both layout and gesture callbacks — so pure MVC maps awkwardly. It tends to produce fat controllers and views that still know too much about models. Most Flutter teams reach for MVVM instead.

10. What is MVVM and why does it fit Flutter well? Model–View–ViewModel: the View (widgets) binds to a ViewModel that exposes observable state and commands; the ViewModel talks to Models/repositories and holds no widget references. It fits Flutter because Flutter is already reactive and declarative — a widget rebuilding from an observable ViewModel *is* data binding. `ChangeNotifier` + `ListenableBuilder`, or a Riverpod `Notifier`, are idiomatic ViewModels. See 43 MVVM.

11. What's the difference between feature-first and layer-first folder structure?

- **Layer-first** groups by technical role: top-level `presentation/`, `domain/`, `data/`, each containing all features. Simple early, but a single feature's files scatter across the tree and it doesn't scale.
- **Feature-first** groups by feature: `features/auth/`, `features/cart/`, each with its own `presentation/domain/data` inside. A feature is self-contained, easy to find, delete, or hand to a team.

Feature-first is the default for anything non-trivial. See 44 Feature First Architecture.

12. What is dependency injection's role in architecture? DI is *how* the dependency rule is realized at runtime: the outer layer's concrete implementations are supplied to the inner layer through its interfaces, rather than constructed inline. A ViewModel receives a `UserRepository` (interface); at composition root you bind it to `UserRepositoryImpl`. This is what makes swapping real for fake possible — and therefore what makes unit testing possible. `get_it`, `injectable`, or Riverpod providers all serve this role.

🟡 Intermediate

13. Contrast MVC, MVP, and MVVM.

Aspect	MVC	MVP	MVVM
Mediator	Controller	Presenter	ViewModel
View↔mediator coupling	Controller may know View	Presenter holds a View *interface*	View observes ViewModel; VM knows nothing of View
UI updates	Controller mutates model, view reads	Presenter calls `view.showX()`	View binds to observable state
Testability	Controller often UI-coupled	Good (mock the view interface)	Best (VM is plain, no view ref)
Flutter fit	Awkward	Verbose (manual view interface)	Natural (reactive binding)

MVVM wins in Flutter because the framework supplies the binding mechanism for free; MVP's explicit view interface is redundant boilerplate here.

14. Show a minimal MVVM ViewModel with `ChangeNotifier`.

```
class ProductsViewModel extends ChangeNotifier {
  ProductsViewModel(this._repo);
  final ProductRepository _repo;

  var _state = const AsyncValue<List<Product>>.loading();
  AsyncValue<List<Product>> get state => _state;

  Future<void> load() async {
    _state = const AsyncValue.loading();
    notifyListeners();
    _state = await AsyncValue.guard(() => _repo.getProducts());
    notifyListeners();
  }
}
// View: ListenableBuilder(listenable: vm, builder: (_, __) => switch on vm.state)
```

The VM holds no `BuildContext` and no widgets — that's what makes it unit-testable and reusable.

15. Why is Riverpod a good fit for MVVM in modern Flutter? A Riverpod `Notifier` / `AsyncNotifier` *is* a ViewModel: it holds state, exposes methods for intent, and its dependencies (repositories) are injected via `ref.watch`. It gives compile-safe DI, automatic disposal, built-in `AsyncValue` for loading/error/data, and testability via provider overrides — without the `context`-coupling of `ChangeNotifierProvider`. So Riverpod delivers MVVM *and* the DI/composition-root role in one tool.

16. Where does state management sit relative to architecture? State management (BLoC, Riverpod, Provider) is a *presentation-layer concern* — it's the machinery of the ViewModel/state-holder, not the architecture itself. A common mistake is thinking "we use BLoC" *is* the architecture. Clean Architecture is agnostic to which you pick; the BLoC/Notifier lives in presentation and depends inward on use cases or repositories. Keep business rules out of your bloc's event handlers.

17. What is the Result/Either pattern and why prefer it over exceptions? It models failure as a value in the return type instead of a thrown control-flow jump: `Future<Result<User, Failure>>` or `Either<Failure, User>`. Benefits: failures become part of the type signature so the compiler forces you to handle them; you distinguish expected domain failures (invalid credentials) from bugs; and repository boundaries return typed failures instead of leaking `DioException` upward. Dart's sealed classes make this clean:

```
sealed class Result<T> {}
class Ok<T> extends Result<T> { final T value; Ok(this.value); }
class Err<T> extends Result<T> { final Failure failure; Err(this.failure); }

final r = await repo.getUser();
switch (r) {
  case Ok(:final value): showUser(value);
  case Err(:final failure): showError(failure);
}
```

18. When should exceptions still be used, not Result? For *unrecoverable* or *programmer-error* conditions — a null assertion failing, a bug, an out-of-memory — where there's no meaningful local handling and you want it to crash loudly (and be logged). Result is for *expected, recoverable* domain outcomes the caller must branch on. Wrapping every null check in `Either` is over-engineering; reserve it for the domain/data boundary where failure is a real business outcome.

19. What is modular (multi-package) architecture? Splitting the app into multiple Dart/Flutter packages — e.g. `core`, `feature_auth`, `feature_cart`, `design_system` — each with its own `pubspec.yaml`, tests, and public API. Benefits: enforced boundaries (a package can only import what it declares as a dependency), faster incremental builds, independent testing, and clear ownership. It's feature-first taken to the package level. See 45 Modular Architecture.

20. What is melos and why use it for a monorepo? `melos` is a tool for managing a Dart/Flutter **monorepo** of many local packages: it bootstraps and links inter-package dependencies, and runs commands (test, analyze, build) across all packages with one invocation. It lets you keep modular packages in a single repo with shared CI and atomic cross-package changes, avoiding the versioning pain of publishing each package separately.

21. How do you actually *enforce* module boundaries? Several mechanisms, layered:

- **Package dependencies** — a package physically can't import code it doesn't depend on in `pubspec.yaml`.
- **Public API surface** — expose only through a barrel/ `lib/<pkg>.dart`; keep the rest under `lib/src/` which other packages shouldn't import.
- **Lint rules** — custom lints or `import_lint` / `dart_code_metrics` rules banning cross-feature imports.
- **Layer direction** — `feature_x` may depend on `core`, never on `feature_y`; shared needs go into `core`.

Convention alone erodes; the package graph is the enforcement that survives turnover.

22. What is a "composition root"? The single place — typically `main()` or a bootstrap/DI setup — where the concrete object graph is wired: interfaces bound to implementations, singletons created, providers overridden. It's the *only* place allowed to know every concrete class. Keeping wiring here means the rest of the code depends on abstractions, and swapping implementations (real vs mock, dev vs prod) is a one-file change.

23. How do you decide what goes in `core` / `shared` vs a feature? Ask "does more than one feature need this, and is it stable?" Cross-cutting infrastructure (networking client, logger, theme, base failure types, DI setup) goes in `core`. Feature-specific logic stays in the feature. Beware the anti-patterns: a bloated `core` that becomes a dumping ground, and premature sharing of code that two features happen to resemble but will diverge — duplicate first, extract when the third use appears.

🔴 Advanced

24. Walk a request through a Clean Architecture app. User taps "Load orders" →

1. **Widget** calls a method/event on the **ViewModel/BLoC** (presentation).
2. VM invokes a **use case** `GetOrders()` (domain).
3. Use case calls `OrderRepository.getOrders()` — an interface (domain).
4. `OrderRepositoryImpl` (data) hits `OrderRemoteDataSource`, gets `OrderDto`s, checks the cache, and **maps** DTOs → `Order` entities.
5. It returns `Result<List<Order>, Failure>` up the chain.
6. VM converts the result into UI state (`AsyncValue`); the widget rebuilds.

Every arrow crosses inward via an interface; nothing inner imports anything outer.

25. How do you architect an app with 100+ screens and a team of 10+?

- **Feature-first, modular packages** so features are independently ownable and buildable.
- **A `core`/design-system package** for shared UI and infrastructure; enforce that features never import each other.
- **Consistent per-feature skeleton** (presentation/domain/data) so any engineer can navigate an unfamiliar feature.
- **Typed navigation** (`go_router` with typed routes) and a route registry per feature to decouple navigation from feature internals.
- **A composition root** aggregating each feature's DI module.
- **CI running melos** across packages; codeowners per package; contract tests at repository boundaries.

The goal is that adding feature #101 doesn't require understanding features #1–100.

26. What is Domain-Driven Design, and how much of it is worth it on mobile? DDD is designing software around a rich model of the business domain, using a *ubiquitous language* shared with domain experts. Its tactical building blocks — entities, value objects, aggregates, domain services, repositories — map well to a Flutter domain layer. Pragmatically on mobile you adopt the *tactical* patterns (especially entities + value objects + repositories) where the domain is genuinely complex; the full strategic apparatus (bounded contexts, context maps, event storming) is usually overkill for a client app whose complexity lives server-side. See 46 Domain Driven Design.

27. Entity vs value object vs aggregate — define each.

- **Entity** — has identity that persists through change; two users with the same name are different if their `id`s differ. Equality is by identity.
- **Value object** — defined entirely by its attributes, no identity, immutable; `Money(100, 'USD')`, `EmailAddress`, `DateRange`. Equality is by value. Great place to enforce validity ("an EmailAddress cannot be constructed invalid").
- **Aggregate** — a cluster of entities/value objects with one **aggregate root** as the sole entry point, enforcing invariants across the cluster (an `Order` root guarding its `OrderLine`s so totals stay consistent). You reference and load aggregates as a unit.

28. Why are value objects underrated in Flutter apps? They push validation and invariants into the type system instead of scattering `if` checks across UI and services. An `EmailAddress` that can only exist if valid means every function taking one is spared re-validation, and impossible states become unrepresentable. They also give meaningful equality for free (via `==` / `Equatable` or Dart records), which matters for rebuild-diffing and testing. The cost is a bit of boilerplate — worth it for concepts with real rules, skippable for a bare `String label`.

29. How do you keep the domain layer pure in Flutter? The domain package/folder must not import `package:flutter`, `dio`, `hive`, `json_annotation`, etc. Enforce it by putting domain in its own package with only `meta` / `collection`-type deps, or via lint rules banning those imports. Entities are plain immutable Dart; no `fromJson` (that's the DTO's job); no `BuildContext`. Purity is what makes the domain portable, framework-swappable, and testable at pure-Dart speed with no widget or device.

30. How does architecture help manage technical debt over time? Boundaries *contain* debt. When data lives behind repository interfaces, a hacked-together data source is quarantined — you can rewrite it without touching domain or UI. Feature packages let you rewrite one feature without destabilizing others. Clear seams also make debt *visible and measurable* (this repo impl is the messy one), and give safe refactoring points. Without boundaries, debt is systemic: any shortcut can entangle unrelated code, and there's nowhere safe to cut.

31. When should you deliberately NOT use Clean Architecture? When the ceremony costs more than it saves:

- **Prototypes, MVPs, throwaway spikes** — speed matters, longevity doesn't.
- **Genuinely simple apps** — a handful of CRUD screens; the layers/mappers/use-cases become pure overhead.
- **Solo, short-lived projects** — no team-coordination benefit to pay for.

Right-sizing is itself a senior skill: start with feature-folders + a ViewModel + repositories, and *introduce* use cases, value objects, and package splits only when complexity or team size demands them. Over-architecting a to-do app is as much a mistake as under-architecting a banking app.

32. How do you evolve architecture incrementally instead of a big rewrite? Refactor at the seams you already have. Introduce a repository interface in front of an existing direct API call; move business logic out of a fat widget into a ViewModel; extract a stable, reused chunk into a `core` package; add use cases only where a VM has grown complex orchestration. Each step is independently shippable and test-covered. The dependency rule tells you the *direction* to refactor toward (push detail outward, policy inward); you don't need a green-field rewrite to get there.

33. How do error models flow across layers? The data layer catches raw, technology-specific errors (`DioException` , `SocketException` , `HiveError`) at its boundary and maps them to domain `Failure` types (`NetworkFailure` , `AuthFailure` , `CacheFailure`), returning `Result` / `Either` . The domain and presentation layers only ever see `Failure` s — they never import Dio. The presentation layer maps `Failure` → user-facing message/UI state. This keeps the inner layers ignorant of transport details and gives you one exhaustive place per layer to handle problems. Pair with 38 Error Handling.

34. How do you unit-test each layer in isolation?

- **Domain** — pure Dart tests; construct a use case with a *fake* repository, assert behavior. No Flutter binding needed.
- **Data** — test repository impls with mocked data sources; assert DTO→entity mapping, cache logic, and failure mapping.
- **Presentation** — test the ViewModel/BLoC with mocked use cases/repositories; assert emitted states for given intents; widget-test the View against a fake VM.

The whole point of the layering is that each of these needs no real network, DB, or device. See 49 Testing.

⚡ Rapid-fire (one-liners)

Question	Answer
Direction of dependencies in Clean Arch?	Inward — toward domain/policy
What does the domain layer depend on?	Nothing (pure Dart)
Where do `fromJson` / `toJson` live?	DTOs in the data layer
Entity vs DTO?	Business concept vs wire/DB shape
Best-fit MV* for Flutter?	MVVM
Is BLoC "the architecture"?	No — it's a presentation-layer state holder
Feature-first vs layer-first default?	Feature-first for non-trivial apps
What enforces module boundaries?	The package dependency graph + lints
Tool for a Dart monorepo?	melos
Value object equality is by?	Value, not identity
Aggregate entry point is called?	The aggregate root
Result/Either models failure as?	A return value, not a throw
Where is the object graph wired?	The composition root
When to skip Clean Architecture?	Prototypes, tiny/short-lived apps
Use case (interactor) is?	One unit of app-specific business logic
A repository hides?	Where the data comes from

Follow-up drills

1. **Design** the package/folder structure for a 120-screen fintech app with 4 squads — justify feature-first vs layer-first, what goes in `core`, and how you enforce that squads don't import each other's features.
2. **Refactor** a 900-line `HomeScreen` `StatefulWidget` that does HTTP, JSON parsing, caching, and UI into a Clean-ish structure — describe the sequence of safe, shippable steps.
3. **Right-size**: given a 6-screen internal tool built by one engineer in 3 weeks, argue what architecture you'd *deliberately omit* and why.
4. **Model** a `Cart` domain with an aggregate root enforcing "total never negative" and "max 20 line items" — show entities, value objects, and where the invariants live.
5. **Design** the error-handling contract for a repository: enumerate the `Failure` types, where raw exceptions are caught and mapped, and how the ViewModel surfaces each to the UI.
6. **Debug**: a teammate put business logic (discount calculation) inside a BLoC event handler and API JSON keys inside widgets — explain the concrete problems this causes and how you'd untangle it.

Testing — Interview Questions

> How you prove Flutter/Dart code works, at unit, widget, and integration levels. For depth see the handbook module 49 Testing.

Testing is a senior-signal topic: it exposes whether you write code that can *be* tested (DI, pure functions, seams) and whether you know the cost/value trade-off of each test type. Interviewers escalate from "what does `expect` do" to "why is your golden test flaky in CI" and "how do you test a debounced stream".

🟢 Basic

1. What is the testing pyramid, and how does it map to Flutter? The pyramid says: have many fast, cheap tests at the bottom and few slow, expensive tests at the top. In Flutter that's **unit tests** (bottom — pure Dart, logic, no widgets, milliseconds), **widget tests** (middle — a single widget or screen in a headless `WidgetTester`, no real device), and **integration tests** (top — the whole app on a device/emulator, slow). You want the bulk of coverage in unit tests because they run in milliseconds and pinpoint failures; integration tests are few because they're slow and flaky. The anti-pattern is the "ice-cream cone" — mostly manual/E2E tests and almost no unit tests.

2. What are the three test types Flutter ships and how do you run them? Unit and widget tests both use the `flutter_test` package and run headlessly with `flutter test` (on the Dart VM, no device). Integration tests use the `integration_test` package and run on a real device/emulator with `flutter test integration_test/...` or `flutter drive`. Rule of thumb: unit = pure logic, widget = UI in isolation, integration = full app flows.

3. What do `test`, `expect`, and matchers do? `test('description', () { ... })` declares a single test case. Inside it, `expect(actual, matcher)` asserts a condition; if the matcher fails, the test fails. A **matcher** describes the expected value — `equals(3)`, `isTrue`, `isNull`, `throwsA(isA<FormatException>())`, `contains('x')`, `greaterThan(0)`. You can pass a raw value (`expect(x, 3)`) and it's implicitly wrapped in `equals`.

```
test('adds two numbers', () {
  expect(add(2, 3), equals(5));
  expect(() => divide(1, 0), throwsA(isA<ArgumentError>()));
});
```

4. What are `group`, `setUp`, and `tearDown` for? `group('...', () {...})` bundles related tests and namespaces their descriptions. `setUp` runs before *each* test in its scope (fresh fixtures — a new repository, a new mock), and `tearDown` runs after each (cleanup — close a stream, clear a DB).

`setUpAll` / `tearDownAll` run once for the whole group. Prefer `setUp` over `setUpAll` so tests stay isolated and don't leak state between each other.

5. What is a unit test, and what should it *not* touch? A unit test verifies one unit of logic — a function, a method, a class — in isolation, with no I/O, no network, no file system, no `MethodChannel`, and ideally no clock or randomness. Those dependencies are replaced with test doubles. If a "unit test" hits a real API or database, it's an integration test wearing the wrong label — it'll be slow and flaky.

6. What is a widget test and what is `WidgetTester`? A widget test pumps a widget into a headless test environment and drives it, verifying layout, text, and interaction without a real device. `WidgetTester` is the handle you get from `testWidgets((tester) async {...})`: it builds the widget (`pumpWidget`), advances frames (`pump`), sends gestures (`tap`, `enterText`, `drag`), and queries the tree via finders.

```
testWidgets('increments counter', (tester) async {
  await tester.pumpWidget(const MyApp());
  expect(find.text('0'), findsOneWidget);
  await tester.tap(find.byIcon(Icons.add));
  await tester.pump();               // rebuild after setState
  expect(find.text('1'), findsOneWidget);
});
```

7. What are finders, and name the common ones? Finders locate widgets in the tree for assertions or actions. Common ones: `find.text('Login')`, `find.byType(ElevatedButton)`, `find.byIcon(Icons.add)`, `find.byKey(const Key('email'))`, `find.byWidgetPredicate(...)`, `find.widgetWithText(ElevatedButton, 'Save')`. You assert existence with matchers: `findsOneWidget`, `findsNothing`, `findsNWidgets(3)`, `findsWidgets`. Keys are the most robust finder because they survive copy changes and localization.

8. `pump` vs `pumpAndSettle` — what's the difference? `await tester.pump(duration)` triggers a single frame (or advances the clock by `duration` and renders one frame). `await tester.pumpAndSettle()` repeatedly pumps frames until there are no more scheduled — i.e., all animations and transitions have completed. Use `pump()` when you know exactly how many frames you need (and to avoid hanging); use `pumpAndSettle()` after navigation or an animation. **`pumpAndSettle` times out (default 10 min) if something animates forever** — e.g. an infinite `CircularProgressIndicator` — which is a classic hang.

9. How do you simulate a tap and check the result? `await tester.tap(find.byType(FloatingActionButton));` then `await tester.pump();` to process the resulting rebuild, then `expect(find.text('1'), findsOneWidget);`. The `pump` between the action and the assertion is essential — without it the frame hasn't rebuilt and your assertion checks stale UI.

10. What is code coverage and how do you generate it? Coverage measures which lines/branches your tests exercised. Generate it with `flutter test --coverage`, which writes `coverage/lcov.info`; view it as HTML with `genhtml`. It's a useful *floor* signal (untested critical paths are visible) but a poor *ceiling* goal — 100% coverage of trivial getters proves nothing. Track coverage on business logic, not on generated files or `main.dart`.

11. What is a mock, and why mock at all? A mock is a stand-in for a real dependency that you program to return canned responses and/or verify calls against. You mock to make tests **fast** (no network), **deterministic** (no flaky I/O or clock), and **isolated** (test *your* unit, not the collaborator). Example: mock a `Repository` so a `ViewModel` test doesn't hit HTTP.

12. What is TDD in one sentence? Test-Driven Development is the red-green-refactor loop: write a failing test first (red), write the minimum code to pass it (green), then refactor with the test as a safety net. It drives you toward small, testable units and forces you to define behavior before implementation.

🟡 Intermediate

13. mockito vs mocktail — which and why? Both create test doubles; the difference is code generation.

	mockito	mocktail
Setup	`@GenerateMocks` + `build_runner` codegen	No codegen — extend `Mock` directly
Null safety	Needs generated mocks for sound null safety	Null-safe by design
Arg matching	`any`, `argThat`	`any()`, `any(named:)`, `captureAny()`
Boilerplate	Generated `.mocks.dart` files	Zero

Mocktail is the modern favorite because it drops the `build_runner` step and reads cleaner. Mockito is still common in older/enterprise codebases. With mocktail you must `registerFallbackValue` for custom types used with `any()`.

14. Fakes vs mocks vs stubs — define each. All are test doubles but differ in intent:

- **Stub** — returns hardcoded canned answers; no logic, no verification. "When asked, return this."
- **Mock** — a stub plus recorded interactions you *verify* ("was `save()` called once with X?"). Behavior verification.
- **Fake** — a real, working lightweight implementation (e.g. an in-memory repository, `sqflite_common_ffi` in-memory DB). It actually runs logic, just not the production version.

Guideline: prefer fakes for state-holding collaborators (they're less brittle), use mocks when the *interaction itself* is what you're asserting.

15. When should you NOT mock? Don't mock value objects, `data` classes, or pure functions — just use the real thing. Don't mock what you don't own (third-party SDKs) directly; wrap them behind your own interface and mock that, so a library API change doesn't break dozens of tests. And don't over-mock so heavily that the test only proves your mocks agree with each other ("mockery") — that's coupling tests to implementation, not behavior.

16. How do you test `async` code — `Future`s? Make the test callback `async` and `await` the future, then assert. For errors use `expectLater(future, throwsA(...))`. `expect` also accepts `completes` / `completion(matcher)` matchers.

```
test('fetches user', () async {
  when(() => api.getUser(1)).thenAnswer((_) async => User('Ada'));
  final user = await repo.load(1);
  expect(user.name, 'Ada');
});
test('propagates error', () {
  expect(repo.load(-1), throwsA(isA<NotFoundException>()));
});
```

17. How do you test streams? Use `expectLater` with the `emitsInOrder` family of matchers: `emits`, `emitsInOrder([...])`, `emitsError`, `emitsDone`, `neverEmits`, `emitsThrough`. These subscribe and assert the emission sequence without manual `listen` bookkeeping.

```
expectLater(
  counterStream,
  emitsInOrder([1, 2, 3, emitsDone]),
);
```

18. What is `fakeAsync` / how do you test time-dependent code? Timers, `Future.delayed`, debounce, and throttle shouldn't make tests wait in real time. In widget tests, `tester.pump(Duration(...))` advances the fake clock. In pure Dart, use `fakeAsync((async) { ...; async.elapse(Duration(seconds: 5)); })` from the `fake_async` package to jump the virtual clock instantly. This makes a "wait 30s then retry" test run in microseconds and stay deterministic.

19. How do you pump a widget that needs `MaterialApp`, providers, or a `Navigator`? Wrap it in the ancestors it depends on. A screen using `Theme`, `MediaQuery`, `Navigator`, or localization needs `MaterialApp(home: ...)`. One using Provider/Riverpod needs the scope wrapped:

```
await tester.pumpWidget(
  ProviderScope(
    overrides: [repoProvider.overrideWithValue(FakeRepo())],
    child: const MaterialApp(home: ProfileScreen()),
  ),
);
```

Forgetting the ancestor gives the classic `No MediaQuery widget found` / `No Directionality` error. A bare widget can be wrapped in just `Directionality`.

20. How do you test BLoC/Cubit — what does `bloc_test` give you? `bloc_test`'s `blocTest` helper standardizes the arrange-act-assert for BLoCs: `build` the bloc (wiring mocked deps), `act` by adding events, and `expect` the ordered list of emitted states. `seed` sets an initial state; `wait` handles debounce; `verify` runs post-assertions on mocks.

```
blocTest<CounterBloc, int>(
  'emits [1] when Increment added',
  build: () => CounterBloc(),
  act: (bloc) => bloc.add(Increment()),
  expect: () => [1],
);
```

21. How do you test a Provider `ChangeNotifier` or a Riverpod provider? For `ChangeNotifier`: instantiate it with mocked dependencies, call methods, and assert on its exposed fields — or attach a listener and count `notifyListeners` calls. For Riverpod: create a `ProviderContainer(overrides: [...])` in the test, read the provider with `container.read(myProvider)`, and `addTearDown(container.dispose)`. Overrides let you inject fakes without any widget tree at all — that's Riverpod's big testability win.

22. What are golden tests? Golden (snapshot) tests render a widget and compare the pixels against a stored reference image (`.png` "golden file"). `await expectLater(find.byType(MyCard), matchesGoldenFile('goldens/card.png'));` and generate/update baselines with `flutter test --update-goldens`. They catch unintended visual regressions that assertion-based widget tests miss (spacing, color, layout).

23. What is DI's role in testability? Dependency Injection is what makes a unit *substitutable*. If a class news-up its own `HttpClient` internally, you can't replace it in a test. If it *accepts* a client (or repository, or clock) via constructor, you inject a fake. DI creates the "seam" that testing requires. See 14 Dependency Injection.

24. Why do pure functions make testing easy? A pure function's output depends only on its inputs and it has no side effects, so a test is just `expect(f(input), output)` — no setup, no mocks, no order dependence, no cleanup. Pushing logic into pure functions (and keeping I/O at the edges) is the single highest-leverage move for a testable codebase.

🔴 Advanced

25. What causes flaky tests, and how do you kill them? Flakiness = nondeterminism. Common causes and fixes:

- **Real time** (`Future.delayed`, timers) → use `fakeAsync` / `tester.pump(duration)`.
- **Ordering / shared state** between tests → fresh fixtures in `setUp`, avoid `setUpAll` mutable state, don't rely on test order.
- `pumpAndSettle` **on an infinite animation** → use bounded `pump()` counts.
- **Real network / clock / random** → inject fakes and seed randomness.
- **Async races** (asserting before a `Future` resolves) → `await` properly, use `expectLater`.
- **Environment** (locale, timezone, screen size, fonts) → pin them; set `tester.view.physicalSize`.

A flaky test is worse than no test — teams learn to ignore it, hiding real regressions.

26. Why are golden tests especially flaky, and how do you stabilize them for CI? Rendered pixels differ across platforms: font rendering and anti-aliasing differ between macOS/Linux/Windows, so a golden generated locally fails in CI. Fixes: generate goldens **in the same environment as CI** (often a Linux Docker image) or use a package like `golden_toolkit` / `alchemist` that loads a consistent test font (`Ahem`/Roboto) and disables platform font fallback. Pin the Flutter version. Many teams run goldens as a separate, non-blocking CI job or only on Linux to avoid cross-platform pixel drift.

27. How do integration tests differ from `flutter drive`, and when do you use each? The `integration_test` package runs test code *on the device* in the same process as the app, using the widget-test API (`WidgetTester`) but against the real app. `flutter drive` runs a *driver script on the host* that commands the app over a Dart VM service protocol — needed for things the on-device test can't do: capturing performance timelines/traces, driving from a CI harness like Firebase Test Lab, or screenshots on real hardware. Modern practice: write tests with `integration_test`, and use `flutter drive` (or `flutter test integration_test`) as the runner.

28. How do you measure performance in an integration test? Wrap the flow in `binding.traceAction(() async {...})` (using `IntegrationTestWidgetsFlutterBinding`) to capture a timeline, then write a `TimelineSummary` with jank metrics (`frame_build_times`, `missed_frames`, 90th/99th percentile frame times). Run via `flutter drive` so the results are written to disk for CI to assert against a budget. This is how you catch "this screen dropped from 60fps to 40fps" regressions.

29. What should you NOT test?

- The framework itself (don't test that `setState` rebuilds — Flutter's team already does).
- Third-party libraries (test your *usage*, behind your interface).
- Trivial code with no logic — plain getters, generated `copyWith`, DTO field assignments.
- Private implementation details / exact call sequences that aren't part of the contract — testing those makes refactoring break tests with no behavior change.
- Const/static UI with no logic (a golden test is overkill for a static label).

Test **behavior and contracts**, not implementation. The heuristic: "would a reasonable refactor that preserves behavior break this test?" If yes, you're testing the wrong thing.

30. How do you make legacy, untestable code testable without a big-bang rewrite? Introduce seams incrementally: extract the hard dependency behind an interface and inject it (constructor param, or a settable field / service-locator override for the worst cases). Pull business logic out of `build()` methods and `initState` into plain classes/functions you can unit-test. Wrap statics and singletons (`DateTime.now`, `Platform`, plugins) behind an injectable abstraction. Add a characterization test around current behavior *first*, then refactor under its protection.

31. How do you test code that uses platform channels / plugins? Register a mock handler: `TestDefaultBinaryMessengerBinding.instance.defaultBinaryMessenger.setMockMethodCallHandler(channel, (call) async {...})` to intercept `MethodChannel` calls and return fake responses — no native side needed. Better still, wrap the plugin behind your own interface and inject a fake in most tests, reserving the channel mock for the thin adapter class. See 26 Platform Channels.

32. How do you verify interactions and capture arguments with mocktail? Use `verify` / `verifyNever` for call assertions and `captureAny()` to grab the actual argument passed:

```
verify(() => analytics.log('checkout')).called(1);
final captured = verify(() => repo.save(captureAny())).captured.single as Order;
expect(captured.total, 42);
verifyNever(() => api.delete(any()));
```

Reserve interaction verification for genuine side effects (analytics fired, cache written). Over-verifying every call couples the test to implementation.

33. What is `WidgetTester.runAsync` and when do you need it? By default widget tests run in a `FakeAsync` zone, so real async work (real HTTP, actual timers, `precacheImage`, platform I/O) never completes. `await tester.runAsync(() async {...})` runs the callback with the real event loop so genuine async completes — needed when a widget does real I/O you can't fully fake. Use sparingly; it reintroduces real-time nondeterminism.

34. How do golden/widget tests handle images and network calls? Network images fail in tests (no real HTTP), returning a 400 and often throwing. Solutions: inject a fake `HttpClient` via `HttpOverrides` that returns a transparent 1×1 image, use `mockNetworkImagesFor` from

`network_image_mock`, or replace `Image.network` with an injectable image widget. For goldens specifically, ensure fonts and images are loaded before `expectLater`, otherwise you snapshot a half-rendered frame.

⚡ Rapid-fire (one-liners)

Q	A
Which command runs unit + widget tests?	`flutter test` (headless, Dart VM)
Which package for integration tests?	`integration_test`
`pump` vs `pumpAndSettle`?	one frame vs pump until no animations left
Most robust finder?	`find.byKey` — survives copy/locale changes
Assert widget count of 1?	`expect(find.text('x'), findsOneWidget)`
mockito's extra step vs mocktail?	`build_runner` codegen
Matcher for an async throw?	`throwsA(isA<T>())` with `expectLater` / `expect`
Test a stream's emissions?	`expectLater(s, emitsInOrder([...]))`
Advance virtual time in pure Dart?	`fakeAsync((a) => a.elapse(...))`
Update golden baselines?	`flutter test --update-goldens`
Generate coverage?	`flutter test --coverage` → `lcov.info`
Helper for BLoC tests?	`blocTest` from `bloc_test`
Override a Riverpod dep in tests?	`ProviderContainer(overrides: [...])`
Runs before each test?	`setUp` (per-test), `setUpAll` (once)
Real async inside a widget test?	`await tester.runAsync(() async {...})`
Biggest golden-test flakiness cause?	cross-platform font/anti-aliasing rendering

Follow-up drills

1. Design a test suite for a login flow: which cases go to unit, widget, and integration, and what do you mock at each level?
2. A widget test hangs forever. Walk through diagnosing it (hint: `pumpAndSettle` + an infinite spinner).
3. Your golden tests pass locally but fail on the CI Linux runner. Explain why and fix it.
4. Given a `ViewModel` that directly instantiates an `HttpClient` and calls `DateTime.now()`, refactor it to be unit-testable and write the key tests.
5. Test a search field that debounces input by 300ms and cancels in-flight requests — without any real waiting.

6. You have 92% coverage but production bugs keep shipping. Argue what's wrong with the test strategy and how you'd rebalance it.

Platform Integration, Native & Device — Interview Questions

> How Flutter talks to the OS: platform channels, Pigeon, FFI, native plugins, device features, notifications, and background work. For depth see 26 Platform Channels, 27 Native Android, 28 Native iOS, 29 Device Features, 32 Notifications, and 33 Background Services.

This topic tests whether you understand that Flutter renders its own UI but has *no* built-in access to the OS — everything native (camera, sensors, notifications, Bluetooth) crosses a language boundary. Interviewers probe how that bridge works, how it's serialized and threaded, and when to reach for channels vs Pigeon vs FFI.

🟢 Basic

1. Flutter draws its own pixels — so how does it access native OS features at all? Through a message-passing bridge called **platform channels**. Dart code and the host (Android/iOS) exchange asynchronous messages over a named channel; the native side runs real Kotlin/Swift (or Java/Obj-C) code that calls the platform SDK and sends a result back. Flutter itself only knows how to rasterize a UI via Skia/Impeller — anything OS-specific (GPS, camera, notifications) goes over this bridge, which is exactly why those features arrive as *plugins* rather than being in the core framework. See 26 Platform Channels.

2. What is a `MethodChannel` and what's the typical flow? A `MethodChannel` is a named channel for **request/response** calls: Dart invokes a named method with arguments, the platform handles it and returns a single result (or an error). The name must match exactly on both sides.

```
const channel = MethodChannel('app.dev/battery');
final level = await channel.invokeMethod<int>('getBatteryLevel');
```

On the native side you register a handler for that channel and `switch` on `call.method`, then call `result.success(...)`, `result.error(...)`, or `result.notImplemented()`. The Dart call returns a `Future` because the round trip is asynchronous.

3. What are the three channel types and when do you use each?

Channel	Direction / shape	Use for
`MethodChannel`	Dart ↔ native, one call → one result	Invoking a native function ("get battery level", "scan")
`EventChannel`	native → Dart, a **stream** of events	Continuous data: sensor readings, location updates, connectivity changes
`BasicMessageChannel`	bidirectional, **arbitrary messages** with a codec	Free-form/continuous two-way messaging, or custom serialization

`MethodChannel` is really a thin convention on top of `BasicMessageChannel`. `EventChannel` wraps the stream lifecycle (listen/cancel) so native pushes events via an `EventSink`.

4. How are channel messages serialized? By a **message codec**. The default is `StandardMessageCodec` (used by `StandardMethodCodec`), which binary-encodes a fixed set of types: null, bool, int, double, String, `Uint8List`/typed data lists, `List`, and `Map`. You don't send Dart objects — you send these primitives, and the codec maps them to the native equivalents (e.g. Dart `Map` ↔ Kotlin `HashMap` / Swift `Dictionary`). There's also `JSONMessageCodec`, `StringCodec`, and `BinaryCodec`. Anything richer (your own classes) you serialize to maps yourself — or use Pigeon.

5. Which thread do platform-channel handlers run on? By default, the platform side runs its handler on the **host platform's main (UI) thread** (Android main thread / iOS main thread), and the Dart side runs on the **root isolate's UI thread**. So heavy native work in a channel handler will jank the native UI — you must offload it to a background thread/executor natively and post the result back. The messages themselves are marshaled across the boundary asynchronously; your `await` on the Dart side never blocks the Dart UI thread.

6. Are platform channel calls synchronous or asynchronous, and why does that matter? Asynchronous — every `invokeMethod` returns a `Future`. This is deliberate: the message is serialized, sent over the bridge, handled, and the reply serialized back, all without blocking either UI thread. It matters because you cannot treat a channel call like a local function that returns instantly; you `await` it and handle latency/errors. Rapid or chatty calls add up in serialization overhead, which is one reason FFI exists for hot paths.

7. What is a Flutter *plugin* and how does it differ from a package? A **package** is pure Dart, reusable code (e.g. a formatting util). A **plugin** is a package that *also* contains platform-specific native code and wires it up over channels/FFI (e.g. `camera`, `geolocator`, `shared_preferences`). Plugins declare their native entry points in `pubspec.yaml` under `flutter.plugin.platforms`. You reach for a plugin whenever you need OS functionality Flutter doesn't provide itself.

8. How do you request runtime permissions, and why can't you just declare them? Declaring a permission only tells the OS you *might* use it — Android needs it in `AndroidManifest.xml`, iOS needs a usage-description key in `Info.plist`. But dangerous permissions (camera, location, mic, contacts) must *also* be granted by the user at runtime. The `permission_handler` package unifies this:

you check `.status`, call `.request()`, and handle `granted` / `denied` / `permanentlyDenied` (the last forces you to send the user to app settings). Missing the `Info.plist` string on iOS is an instant App Store rejection / crash. See 29 Device Features.

9. What's the difference between a local notification and a push (FCM) notification? A **local notification** is scheduled and fired by the app on-device (via `flutter_local_notifications`) — a reminder, a timer, a download-complete alert; no server, no network. A **push notification** originates from a server via **FCM** (Firebase Cloud Messaging) / APNs and is delivered by the OS even when your app is closed. Push needs backend + tokens + platform setup; local needs neither. Many apps use both: FCM delivers the data, and you render it with local notifications for richer control. See 32 Notifications.

10. What is an Android notification channel and why is it mandatory? Since Android 8 (API 26), every notification must belong to a **notification channel** — a category (e.g. "Messages", "Promotions") the user can independently configure (sound, importance, whether it's blocked). If you post a notification without creating its channel first, it silently won't show. You create channels once at startup with an id, name, and importance level. iOS has no channels but has the equivalent of authorization + categories.

11. Why do you need to request notification permission explicitly now? iOS always required the user to *authorize* notifications. Android added the same with **Android 13 (API 33)**: the `POST_NOTIFICATIONS` runtime permission. So on both platforms you must request permission before notifications appear — you can't assume they'll show just because you posted them. Web push also requires an explicit browser permission prompt.

12. Can you run Dart code in the background when the app is killed, and what's the catch? Partly — and the platforms differ sharply. Android allows real background execution via `WorkManager` (through `workmanager`) and background isolates; iOS is *far* more restrictive — it only grants short, OS-scheduled windows (BGTaskScheduler / background fetch) and will not let you run arbitrary long-lived work. The universal catch: background work runs in a **separate isolate** with no access to your app's in-memory state, so you must re-init plugins and pass data via serializable arguments or storage. See 33 Background Services.

🟡 Intermediate

13. Walk through the full round trip of a `MethodChannel` call, boundary by boundary. (1) Dart calls `invokeMethod('m', args)`; the `StandardMethodCodec` serializes the method name + args into a binary `ByteData`. (2) The message is handed to the engine and delivered to the platform side on its main thread via the `BinaryMessenger`. (3) Native decodes it, runs your handler, and produces a result (or error/notImplemented). (4) The result is re-encoded and sent back across the bridge. (5) The Dart `Future` completes with the decoded value, or throws a `PlatformException` if native returned an error. Every step is async and every payload must be codec-supported.

14. What's the difference between the root isolate and other isolates when it comes to channels? Platform channels are bound to the **root isolate** (the one Flutter starts). Historically, only the root isolate could use channels, so a spawned isolate couldn't call native code directly. Modern Flutter adds `BackgroundIsolateBinaryMessenger` (root isolate token) so a background isolate can register its own messenger and talk to plugins — you pass the `RootIsolateToken` into the isolate and call `ensureInitialized`. This is exactly what background-notification and WorkManager handlers must do.

15. What is Pigeon and what problem does it solve? Hand-written channels are **stringly-typed and fragile**: a typo in the method name or a mismatched argument type fails only at runtime, and you serialize objects into maps by hand on both sides. **Pigeon** is a code generator: you define your API in a Dart file (abstract classes with typed methods and data classes), run Pigeon, and it emits type-safe Dart + Kotlin/Swift/Obj-C glue. You then just implement the generated interface. Benefits: compile-time type safety, no string channel names to misspell, auto (de)serialization of your data classes, and refactor-safety. See 26 Platform Channels.

16. When would you use `dart:ffi` instead of platform channels? FFI (Foreign Function Interface) calls **C ABI functions directly** — no message serialization, no bridge hop, near-native speed, and *synchronous* by default. Use it when: you're integrating an existing C/C++ library (SQLite, image codecs, crypto, ML kernels), or you have a hot path where per-call channel serialization overhead is unacceptable. Use channels instead when you need platform *SDK* features (which are Kotlin/Swift APIs, not C) or push-style event streams. Rule of thumb: **channels for platform APIs, FFI for native libraries and performance-critical numeric code.**

17. Compare channels vs FFI directly.

Aspect	Platform channels	`dart:ffi`
Talks to	Kotlin/Swift/Java/Obj-C	C ABI (C/C++/Rust via C)
Sync/async	Asynchronous	Synchronous (can be made async)
Serialization	Codec marshals every message	None — direct memory/pointers
Overhead	Per-call encode/decode + bridge	Very low
Best for	Platform SDK APIs, event streams	Native libs, hot numeric paths
Thread safety	Runs on platform main thread	Runs on the calling isolate's thread

FFI's cost is manual memory management (`malloc` / `free` , `Pointer` , `Struct`) and losing access to platform SDKs.

18. How does an `EventChannel` actually stream data from native to Dart? Dart calls `.receiveBroadcastStream().listen(...)` . Under the hood this sends an `onListen` message to native, which hands you an `EventSink` / `EventChannel.StreamHandler` . Native pushes values by calling `sink.success(event)` (or `sink.error`), each of which is codec-encoded and delivered as a stream

event to Dart. When Dart cancels the subscription, an `onCancel` message tells native to stop producing (e.g. unregister the sensor listener). Forgetting to release the native resource in `onCancel` is a classic leak — sensors keep firing and draining battery.

19. What is a platform view and what's the cost of using one? A **platform view** embeds a *native* view (a Google Map, a WebView, a camera preview, an ad view) inside the Flutter widget tree via `AndroidView` / `UiKitView` (or `PlatformViewLink`). It exists because some things are genuinely native views you can't reproduce as Flutter widgets. The cost: composing a native view into Flutter's rendering is expensive — Android uses hybrid composition / texture layers, which adds overhead and can cause synchronization/jank, and it breaks the "everything is a Flutter widget" performance model. Use them only when necessary and keep them small/few.

20. On the native Android side, what do you touch to add a feature and wire a channel? You edit `AndroidManifest.xml` (permissions, `<service>` / `<receiver>`, metadata), and write Kotlin in the plugin/app that registers a channel in the `FlutterPlugin` 's `onAttachedToEngine` via `flutterPluginBinding.binaryMessenger`, setting a `MethodCallHandler`. Runtime permissions, foreground services, and background components are all declared here. Gradle config (min SDK, dependencies) lives in `build.gradle`. See 27 Native Android.

21. And on the native iOS side? You edit `Info.plist` (usage-description strings like `NSCameraUsageDescription`, background modes, URL schemes) and write Swift/Obj-C that registers a `FlutterMethodChannel` with the `registrar.messenger()` in the plugin's `register(with:)`. Capabilities (push, background fetch) are toggled in Xcode and reflected in the entitlements file. APNs setup lives here for push. See 28 Native iOS.

22. What are federated plugins and why do they exist? A **federated plugin** splits a plugin into separate packages: an **app-facing** package (the API you import), a **platform-interface** package (an abstract contract), and one **platform implementation** per platform (android, ios, web, windows...). This lets *different authors* add platform support without touching the core, lets an app override one platform's implementation, and keeps each platform's native code isolated. It's how big plugins like `url_launcher` and `camera` scale across 6 platforms — you can add desktop support by publishing just an endorsed implementation package.

23. How do you handle a deep link that arrives via a notification tap? The notification carries a payload (a route/URL). Two entry cases matter: the app is (a) foreground/background but alive, or (b) terminated. For a live app, the tap callback (`onDidReceiveNotificationResponse` / FCM `onMessageOpenedApp`) fires and you navigate. For a *terminated* app launched by the tap, you must read the launch payload at startup (`getNotificationAppLaunchDetails()` / FCM `getInitialMessage()`) and route accordingly *after* the navigator is ready. Missing the terminated-state path is the most common deep-link bug. See 32 Notifications.

24. How do FCM messages behave in foreground vs background vs terminated states? It depends on message type. A **notification message** is auto-displayed by the OS when the app is backgrounded/terminated; in the foreground it is *not* shown automatically — `onMessage` fires and

you render it yourself (usually via a local notification). A **data-only message** always invokes your handler and never auto-displays. Background/terminated data messages run in `onBackgroundMessage`, a top-level function in a **background isolate** — so it must be a top-level/static function and re-initialize anything it needs.

25. Why must the FCM background handler be a top-level or static function? Because it runs in a **separate background isolate** with a fresh memory space — it has none of your app's singletons, providers, or `main()` setup. Flutter needs a function it can look up and invoke standalone (annotated `@pragma('vm:entry-point')` so tree-shaking doesn't strip it). Inside it you must call `Firebase.initializeApp()` again and re-create any dependencies. Trying to reference app-level state from it simply won't work.

26. How do you get continuous location or sensor updates, and what's the resource concern? Via a stream — `geolocator`'s position stream or `sensors_plus`' accelerometer/gyroscope streams (both backed by `EventChannel`s). The concern is **battery and lifecycle**: high-accuracy GPS and high-frequency sensors drain battery fast, and you must cancel the subscription when the screen/app is no longer using it (in `dispose`) so the native listener unregisters. Request the *minimum* accuracy/frequency you need, and for background location you need extra permissions (`ACCESS_BACKGROUND_LOCATION` / iOS "Always") that trigger stricter store review.

🔴 Advanced

27. Your channel handler does heavy work (image processing) and the native UI freezes. Diagnose and fix. Channel handlers run on the platform **main thread**, so blocking work there janks native UI (and can ANR on Android). Fix natively: dispatch the work to a background executor/`DispatchQueue.global()`, then post the result back **on the main thread** before calling `result.success(...)` (the reply must return on the platform thread the handler was invoked on). Alternatively, if it's pure computation with no platform SDK, move it to a Dart `Isolate`/`compute`, or drop to FFI so it runs off the message pipeline entirely. The anti-pattern is doing CPU-bound work synchronously inside the handler.

28. You're sending large binary data (a captured frame) over a channel every frame and it's slow. What are your options? Per-frame channel serialization of large buffers is a known bottleneck — the `StandardMessageCodec` copies and encodes each payload. Options, best first: (1) use `Texture` **/external texture** so native writes pixels to a GPU texture Flutter composites directly, avoiding copying frames over the channel at all (this is how `camera`/video render previews); (2) use **FFI with shared memory / a** `Pointer` so Dart reads the buffer without a copy; (3) if you must use a channel, send `Uint8List` (which `BinaryCodec` passes efficiently) rather than lists of ints, and reduce resolution/frequency. Sending raw pixel `List<int>` per frame is the worst case.

29. Explain the threading model precisely: UI thread, platform thread, raster thread, and where channel messages flow. Flutter's engine uses several task runners: the **UI (Dart) thread** runs your Dart + builds the layer tree; the **raster thread** turns layers into GPU commands; the

platform thread is the host's main thread (Android/iOS UI thread) where plugins and channel handlers execute. A channel message originates on the Dart UI thread, crosses to the platform thread for handling, and the reply comes back to the Dart UI thread. This is why: channel calls can't block either UI thread (they're async), heavy native handler work janks the *platform* thread, and platform-view composition (on the platform/raster boundary) is expensive to synchronize.

30. When would you choose Pigeon over hand-written channels *and* over FFI — give the decision boundary. Pigeon vs hand-written channels: almost always prefer **Pigeon** for any non-trivial channel API — you get type safety and eliminate string/type-mismatch bugs for free, at the cost of a codegen step. Hand-written channels are fine only for a one-off single method. Pigeon vs FFI: Pigeon still generates *channel* code, so it targets **platform SDK APIs** (Kotlin/Swift). Choose **FFI** when the target is a **C library** or a performance-critical synchronous path where you can't afford serialization. So: C library / hot path → FFI; structured platform-API bridge → Pigeon; trivial single call → raw channel.

31. How does FFI manage memory, and what are the leak/safety pitfalls? FFI works with raw `Pointer`s into native memory. You allocate with `malloc.allocate` / `calloc`, must `free` it yourself, and marshal strings via `toNativeUtf8()` / `.toDartString()`. Pitfalls: **leaking** allocations you forgot to free; **use-after-free** if native frees memory Dart still holds a pointer to; passing a Dart-managed buffer that the GC may move; and blocking the isolate on a long native call (FFI is synchronous — use `Isolate`/async FFI for long calls). You also lose Dart's safety net: a bad pointer crashes the process, not a catchable exception. Wrap allocations in try/finally and prefer `ffigen` to generate correct bindings.

32. Design a robust FCM + local-notification pipeline that handles all app states and deep links. Layers: (1) request `POST_NOTIFICATIONS` (Android 13+) and iOS authorization on first launch; (2) obtain and refresh the FCM token, send it to your backend, and listen to `onTokenRefresh`; (3) create Android notification channels at startup; (4) foreground: `onMessage` → render via `flutter_local_notifications` with a payload; (5) background/terminated: a top-level `onBackgroundMessage` handler (re-init Firebase) for data messages; (6) taps: handle `onMessageOpenedApp` for warm starts and `getInitialMessage()` + `getNotificationAppLaunchDetails()` for cold starts, routing only after the navigator exists; (7) make routing idempotent so a re-delivered message doesn't double-navigate. See 32 Notifications.

33. Why is iOS background execution so limited, and how do you architect around it? iOS aggressively suspends apps to preserve battery; it does **not** allow arbitrary long-running background work. You get: silent push (`content-available`) to wake briefly, `BGAppRefreshTask` / `BGProcessingTask` scheduled *by the OS* (no guaranteed timing), and specific background modes (audio, location, VoIP). Architect around it by making the server authoritative: push data down, do minimal work in the short window, and **defer heavy processing to next**

foreground. Don't design a feature that assumes periodic background execution on iOS — Android's `WorkManager` can honor it but iOS can't, so the cross-platform design must tolerate "runs only when the OS allows." See 33 Background Services.

34. How does `WorkManager`-style background work interact with isolates and plugins? A scheduled task runs in a **background isolate** spawned by the engine, entered through a top-level `@pragma('vm:entry-point')` callback (`callbackDispatcher`). That isolate has no `BuildContext`, no app state, and — until you initialize it — no channel access. You must `WidgetsFlutterBinding.ensureInitialized()`, register the `RootIsolateToken` / `BackgroundIsolateBinaryMessenger`, and re-create any plugins you call. Constraints (network, charging, unmetered, periodic ≥15 min on Android) are enforced by the OS scheduler, not you. Work must be idempotent because the OS may retry it.

35. A plugin works on Android but throws `MissingPluginException` on a background isolate / add-to-app / a new platform. What's happening? `MissingPluginException` means the Dart side sent a channel message but **no handler is registered** on the receiving end. Common causes: (1) a **background isolate** that never registered its binary messenger (needs the root isolate token dance); (2) **add-to-app** where the engine wasn't set up so plugins never registered; (3) calling before `WidgetsFlutterBinding.ensureInitialized()`; (4) the plugin **doesn't implement that platform** (federated impl missing) so it's genuinely not there; (5) a hot-restart that lost native registration. Diagnose by checking whether native registration ran and whether the platform is actually supported.

36. How do you make a channel bridge testable and resilient to native errors? On the Dart side, wrap channel calls to translate `PlatformException` / `MissingPluginException` into your domain errors, and hide the channel behind a repository interface so business logic doesn't know about `MethodChannel`. Test it with `TestDefaultBinaryMessengerBinding` / `setMockMethodCallHandler` (or Pigeon's generated mocks) to fake native responses — no device needed. Always handle `notImplemented`, timeouts, and malformed payloads defensively, and prefer **Pigeon** so the contract is typed and both sides can't silently drift. This mirrors the "hide platform behind a repo" DI principle — see 14 Dependency Injection.

⚡ Rapid-fire (one-liners)

Q	A
Bridge Dart ↔ native?	Platform channels.
One call, one result channel?	`MethodChannel`.
Stream native → Dart?	`EventChannel`.
Free-form two-way + custom codec?	`BasicMessageChannel`.
Default channel codec?	`StandardMessageCodec` (binary).
Which thread runs channel handlers?	The platform main/UI thread.
Type-safe channel codegen tool?	Pigeon.
Direct C-library calls from Dart?	`dart:ffi`.
Channels vs FFI rule?	Platform APIs → channels; native libs / hot path → FFI.
Package vs plugin?	Plugin adds native platform code.
Split-by-platform plugin design?	Federated plugin.
Embed a native view?	Platform view (`AndroidView` / `UiKitView`).
iOS permission strings live where?	`Info.plist` (usage-description keys).
Android permission ≥API 26 grouping?	Notification channels.
Android 13 notification permission?	`POST_NOTIFICATIONS`.
Local vs push notification?	On-device app-fired vs server via FCM/APNs.
FCM data-message background entry?	Top-level `onBackgroundMessage` isolate.
Cold-start notification route source?	`getInitialMessage()` / launch details.
Android periodic background work?	`WorkManager` (min ~15 min).
Why background isolate can't see app state?	Separate isolate, fresh memory.
Efficiently render native camera frames?	External `Texture`, not per-frame channel copies.
`MissingPluginException` means?	No handler registered on the receiving side.

Follow-up drills

1. **Design** a cross-platform Bluetooth-scanner plugin: pick `MethodChannel` vs `EventChannel` per operation, define the Pigeon API, and lay out the federated package split.
2. **Optimize** a live camera-preview + real-time filter feature that drops frames — decide between platform view + texture, FFI image processing, and channel payloads, and justify.
3. **Debug** a notification that opens the correct screen when the app is warm but lands on the home screen when launched from a killed state.

4. **Architect** a "sync my data every hour" feature that behaves correctly on both Android `WorkManager` and iOS's restricted background model.
5. **Integrate** an existing C++ image-codec library via `dart:ffi` — specify the memory-management strategy, error handling, and how you'd keep long calls off the UI isolate.
6. **Justify** choosing Pigeon over hand-written channels to a team lead worried about the extra codegen step — quantify the failure modes it removes.

Security, Error Handling & Logging — Interview Questions

> How production Flutter apps stay safe and stay observable: secure storage, transport security, code hardening, robust error handling, and disciplined logging. For depth see 37 Security, 38 Error Handling, and 39 Logging.

This topic tests whether you treat a mobile app as an *untrusted client* running on a *hostile device* — someone can root the phone, MITM the network, and decompile your binary. Interviewers probe where you put secrets, how you protect tokens and traffic, how you fail gracefully, and whether your logs help you debug without leaking user data.

🟢 Basic

1. Why shouldn't you store auth tokens (or any secret) in `SharedPreferences`? `SharedPreferences` is **plaintext** — on Android it's an unencrypted XML file in the app sandbox, on iOS a plist in `NSUserDefaults`. On a rooted/jailbroken device, or from a device backup, anyone can read it. Tokens, passwords, PII, and encryption keys belong in `flutter_secure_storage`, which is backed by the OS-provided hardware-backed secure store. `SharedPreferences` is fine for non-sensitive UI state (theme, onboarding-seen flag). See 37 Security.

2. What does `flutter_secure_storage` actually use under the hood? It delegates to the platform's native secure store: the **iOS Keychain** and the **Android Keystore** (with `EncryptedSharedPreferences` for the encrypted blob). Keys are managed by the OS and can be hardware-backed (Secure Enclave / TEE / StrongBox), so key material never leaves secure hardware. That's the point — you're not rolling your own crypto, you're leaning on OS primitives designed for exactly this.

```
const storage = FlutterSecureStorage();
await storage.write(key: 'refresh_token', value: token);
final token = await storage.read(key: 'refresh_token');
```

3. Is data in the app sandbox already "secure" on modern phones? Only against *other apps* on a non-compromised device — app sandboxing isolates storage per app. It is **not** secure against a physically-possessed, rooted/jailbroken device, malware with root, or an unencrypted backup. Full-disk encryption (on by default) protects a powered-off/locked device, but once unlocked the sandbox files are readable to a privileged attacker. So "sandbox" ≠ "encrypted secret store" — sensitive data still needs the Keychain/Keystore.

4. What is TLS and what does it protect against? TLS (HTTPS) encrypts data in transit and authenticates the server via its certificate chain, defending against eavesdropping and tampering (man-in-the-middle). Always use `https://`; never ship an app that talks plaintext HTTP. On Android, cleartext traffic is blocked by default (API 28+) via network security config; on iOS, App Transport Security (ATS) enforces TLS. TLS alone trusts *any* CA the device trusts — which is why high-security apps add certificate pinning on top.

5. What is certificate pinning and why would you add it? Pinning means the app only trusts a **specific** certificate or public key (you embed its hash), rather than any cert signed by a trusted CA. It defends against a compromised/rogue CA and against users who install a custom root CA to MITM traffic (e.g. via Charles/mitmproxy or corporate proxies). You typically pin the leaf or intermediate public-key hash and validate it in an `HttpClient.badCertificateCallback` / Dio interceptor.

6. What's the difference between an access token and a refresh token? An **access token** is short-lived (minutes) and sent with each API request to prove authorization; if it leaks, the blast radius is small. A **refresh token** is long-lived and used *only* to obtain new access tokens from the auth server — it's more sensitive, so it must live in secure storage and never be sent to normal API endpoints. This split limits exposure: access tokens fly over the wire constantly but expire fast; refresh tokens are stored carefully and used rarely. See 17 Authentication.

7. Should you ever put API keys or secrets in your Dart code or `.env` bundled with the app? No. Anything shipped in the app binary — Dart strings, `.env` assets, `--dart-define` values — is extractable by decompiling the APK/IPA. There is no such thing as a "secret" on the client. Truly secret keys (payment provider secrets, admin keys) must stay server-side; the app calls your backend, which holds them. Client "keys" that must ship (like a Firebase/Maps API key) should be *restricted* (by package name, SHA fingerprint, referrer) so a leaked key is useless elsewhere.

8. What's the difference between an Error and an Exception in Dart? An `Exception` represents a condition you can reasonably anticipate and recover from (network failure, invalid input, file not found) — you catch and handle it. An `Error` represents a programming bug that shouldn't be caught in normal flow (`RangeError`, `StateError`, `TypeError`, failed `assert`) — you fix the code, not catch the symptom. The convention: throw `Exception` for recoverable runtime conditions, let `Error`s crash so you notice them. See 38 Error Handling.

9. Walk through `try` / `on` / `catch` / `finally`.

- `on SomeException` catches a **specific type** (preferred — you handle what you expect).
- `catch (e, st)` catches anything and gives you the object and **stack trace**.
- `finally` always runs (success or throw) — use it for cleanup like closing a stream or file.

```
try {
  await repo.fetch();
} on SocketException {
  showOffline();          // specific, expected
} on FormatException catch (e, st) {
  log.severe('bad payload', e, st);
} catch (e, st) {
  log.severe('unexpected', e, st);
  rethrow;                // don't swallow the unknown
} finally {
  hideSpinner();
}
```

Prefer `on Type` over a bare `catch`, and never swallow errors silently — at minimum log and `rethrow`.

10. Why is `catch (e) {}` (an empty catch) a code smell? It **silently swallows** failures — the app carries on in an inconsistent state and you get no crash report, no log, no signal that anything went wrong. These "silent failures" are the hardest bugs to diagnose in production. If you truly can ignore an error, add a comment explaining why and log it at `debug` / `fine`; otherwise handle it or `rethrow`.

11. What are log levels and why do they matter? Log levels rank messages by severity — commonly `trace/finest`, `debug/fine`, `info`, `warning`, `severe/error`, `fatal`. They let you emit verbose detail in development but filter to `warning` + in production, keeping noise and cost down while still capturing what matters. `print()` has no levels, no structure, and ships to release builds — use `dart:developer`'s `log()` or the `logging` / `logger` package instead. See 39 Logging.

12. Why must you never log passwords, tokens, or PII? Logs get aggregated to remote services (Crashlytics, Sentry, cloud logging), stored, and viewed by many people — a token or personal data in a log is a data breach waiting to happen and often a compliance violation (GDPR/PCI). Redact or omit sensitive fields *before* they hit any sink. Treat logs as potentially public.

13. What is obfuscation and how do you enable it in Flutter? Obfuscation renames Dart symbols (classes, methods) to meaningless identifiers so a decompiled binary is harder to read/reverse-engineer. Enable it at build time:

```
flutter build apk --obfuscate --split-debug-info=build/symbols
```

`--split-debug-info` extracts a symbol map so you can later **de-obfuscate** crash stack traces. Note it raises the bar for casual reverse-engineering; it is **not** encryption and doesn't hide strings/secrets — a determined attacker still gets through.

🟡 Intermediate

14. Where exactly should the access token vs refresh token live, and how does rotation work? Keep the **refresh token in secure storage** (Keychain/Keystore) always. The **access token** can sit in memory for the session and optionally in secure storage for warm starts. On a 401, an interceptor uses the refresh token to fetch a new access token, retries the failed request, and — with **refresh-token rotation** — the server also issues a *new* refresh token and invalidates the old one. Rotation limits the value of a stolen refresh token: reuse of an old one signals theft and lets the server revoke the whole family. Guard the refresh flow with a mutex so concurrent 401s trigger only one refresh.

15. What are the risks and operational costs of certificate pinning? The big one is **bricking your app on cert rotation**: when the server's cert/key changes and the app still pins the old one, every request fails until users update — you can't fix it server-side. Mitigations: pin the **public key** (survives cert renewal if the key is reused) or the intermediate CA, pin **multiple** keys (current + next/backup), and ship pins you can update via a controlled release. Pinning also complicates your own debugging/proxying. It's a real security win against MITM but demands rotation discipline, so reserve it for high-value apps (banking, health).

16. Give a concrete example of PII redaction in a logging pipeline. Scrub or mask fields before they reach a sink, ideally centrally in an interceptor/formatter so no call site can forget:

```
String redact(Map<String, dynamic> json) {
  const secret = {'password', 'token', 'authorization', 'ssn', 'email'};
  final safe = {
    for (final e in json.entries)
      e.key: secret.contains(e.key.toLowerCase()) ? '***' : e.value,
  };
  return jsonEncode(safe);
}
```

Also strip `Authorization` headers from logged requests, mask emails/phones (`a***@x.com`), and never log full request/response bodies of auth endpoints. Redaction at the boundary beats trusting every developer to remember.

17. Result/Either vs throwing exceptions — when do you use each? Throwing is idiomatic Dart and fine for truly exceptional/unexpected conditions. A `Result` / `Either<Failure, Success>` (via `dartz` , `fpdart` , or a sealed class) makes failure part of the **return type** — the compiler forces the caller to handle both branches, which is great for *expected* domain failures (validation, "user not found", offline) crossing layer boundaries in Clean Architecture.

```
sealed class Result<T> {}
class Ok<T> extends Result<T> { final T value; Ok(this.value); }
class Err<T> extends Result<T> { final Failure failure; Err(this.failure); }
```

Rule of thumb: **exceptions for the exceptional, Result for expected failures** you model as data. Mixing both everywhere just adds noise — pick a convention per layer. See 40 Clean Architecture.

18. How do you write and use a custom exception well? Model a specific failure with enough context to handle it, and implement `toString()` for logs:

```
class ApiException implements Exception {
  final int statusCode;
  final String message;
  const ApiException(this.statusCode, this.message);
  @override
  String toString() => 'ApiException($statusCode): $message';
}
```

Implement `Exception` (not extend it), keep fields immutable, and make types specific enough that callers can `on ApiException catch` and branch on `statusCode`. Convert low-level errors (`DioException`, `SocketException`) into your domain exceptions at the data-layer boundary so upper layers don't depend on transport details.

19. What global error handlers exist in Flutter, and what does each catch?

Handler	Catches
`FlutterError.onError`	Errors **inside the Flutter framework** (build/layout/paint), i.e. widget errors
`PlatformDispatcher.instance.onError`	Uncaught **async** errors outside a zone; returns `bool` (handled?)
`runZonedGuarded`	Uncaught errors (sync + async) in the **zone** wrapping your app

A robust setup forwards framework errors into your zone/reporter and reports everything to Crashlytics/Sentry:

```
void main() {
  runZonedGuarded(() {
    WidgetsFlutterBinding.ensureInitialized();
    FlutterError.onError = (details) {
      FlutterError.presentError(details);
      reporter.recordFlutterError(details);
    };
    PlatformDispatcher.instance.onError = (error, stack) {
      reporter.recordError(error, stack);
      return true;
    };
    runApp(const MyApp());
  }, (error, stack) => reporter.recordError(error, stack));
}
```

Modern Crashlytics can wire the first two for you (`FlutterError.onError = FirebaseCrashlytics.instance.recordFlutterFatalError`), but knowing what each layer covers is the interview signal.

20. What is an "error boundary" in Flutter and how do you build one? Flutter has no React-style error boundary widget, but you get two hooks. `ErrorWidget.builder` lets you replace the ugly red error screen with a friendly fallback in release. For *async/state* errors, catch them in your state layer and render an error state. For build-time isolation you can wrap a subtree and use `ErrorWidget.builder` globally:

```
ErrorWidget.builder = (details) => const FriendlyErrorScreen();
```

The pattern is: contain the failure to the smallest UI region, show a recoverable fallback (retry button), and report the error — don't let one widget's exception blank the whole app.

21. Structured logging vs plain string logs — what's the difference and why care? Plain logs are unsearchable text (`"user 42 failed login"`). **Structured logs** emit machine-parseable records — typically JSON with consistent fields (`level` , `timestamp` , `event` , `userId` , `traceId`) — so a log platform can index, filter, aggregate, and alert on them. In production you can then query "all `severe` events for `traceId=abc` in the last hour" instead of grepping. Attach a correlation/trace id to tie a user action across screens, network calls, and the backend together.

22. What's the difference between logging and observability? Logging is one signal — discrete event records. **Observability** is the broader ability to understand *why* a system behaves as it does from its outputs, built on three pillars: **logs** (what happened), **metrics** (aggregate numbers over time — crash-free rate, p95 latency), and **traces** (the path of one request across

components). Crashlytics gives you crash metrics; Sentry adds performance traces and breadcrumbs; Firebase Performance adds metrics. Interviewers want to hear that logs alone don't tell you the health of the system. See 52 Monitoring.

23. Crashlytics vs Sentry — how do you choose?

	Crashlytics	Sentry
Focus	Crash reporting (Firebase)	Errors + performance + tracing
Cost	Free	Free tier, paid scaling
Ecosystem	Deep Firebase/Google integration	Platform-agnostic, backend + frontend
Breadcrumbs/context	Custom keys, logs	Rich breadcrumbs, releases, traces

Both capture uncaught errors with de-obfuscated stack traces (upload your symbol files). Crashlytics is the default if you're already on Firebase; Sentry when you want unified error+performance across mobile *and* your backend. Either way, upload `--split-debug-info` symbols so obfuscated traces are readable.

24. What is root/jailbreak detection good for, and what are its limits? It flags a device where the OS security model is compromised (attacker can read your Keychain/Keystore, hook functions, MITM freely) so you can degrade gracefully — warn, block high-risk actions, or refuse to run (common in banking). The hard limit: it's a **cat-and-mouse** check running *on* the very device it's trying to assess, so a rooted device can spoof the check (Magisk Hide, Frida). Treat it as **defense-in-depth / risk signaling**, never as a security guarantee — the real protection is server-side validation and not trusting the client.

25. What is App Check / Play Integrity and what problem do they solve? They provide **attestation** that a request comes from a genuine, untampered instance of *your* app on a genuine device — not a script, emulator farm, or repackaged clone hitting your backend. **Play Integrity** (Android) and **App Attest / DeviceCheck** (iOS), surfaced through **Firebase App Check**, mint a token your backend verifies before serving sensitive APIs. Because the attestation is validated **server-side** by Google/Apple, it resists the client-side spoofing that defeats root detection. It reduces abuse/bot traffic but isn't user auth — it attests the *app*, not the *user*.

🔴 Advanced

26. Design a secure token lifecycle for a Flutter app from login to logout.

- **Login**: backend returns short-lived access token + long-lived refresh token. Store refresh (and optionally access) in `flutter_secure_storage`; keep access in memory.
- **Requests**: an interceptor attaches the access token. On **401**, a single-flight (mutex-guarded) refresh exchanges the refresh token for a new pair (**rotation**), updates secure storage, and retries queued requests.

- **Reuse detection**: server tracks refresh-token families; a replayed old token revokes the family (theft response).
- **Logout**: call server revoke endpoint, then wipe secure storage and in-memory state; clear any cached user data.
- **Hardening**: TLS + optional pinning, App Check on the refresh endpoint, biometric gate before releasing the token on app resume for sensitive apps. The theme: minimize token lifetime and exposure, store the sensitive one in hardware-backed storage, and make theft detectable and revocable.

27. Why does `runZonedGuarded` need `WidgetsFlutterBinding.ensureInitialized()` inside the same zone, and what's the common bug? The binding must be initialized in the **same zone** that runs the app, because Flutter asserts that the zone used to schedule frames is the zone the binding was created in. If you call `ensureInitialized()` outside `runZonedGuarded` (or in a different zone) you get a "Zone mismatch" assertion or silently lose error capture. The fix is to put *both* `ensureInitialized()` and `runApp()` inside the same `runZonedGuarded` callback. With `PlatformDispatcher.onError` now available, many teams skip the zone entirely for global capture — but if you use a zone, keep initialization inside it.

28. Uncaught async error handling changed across Flutter versions — explain the current recommended setup. Historically `runZonedGuarded` was the only way to catch uncaught *async* errors, because `FlutterError.onError` only covers the framework's synchronous build/layout/paint. Since Flutter 3.3, `PlatformDispatcher.instance.onError` catches uncaught async errors at the engine level with lower overhead and without a custom zone. The current recommendation: set `FlutterError.onError` for framework errors and `PlatformDispatcher.instance.onError` for everything else, reserving `runZonedGuarded` for cases where you must also capture errors from third-party code that reports into the current zone. Route all three to one reporter.

29. How do you make obfuscated production crashes debuggable end to end? Build with `--obfuscate --split-debug-info=<dir>`, which produces a per-build symbol map keyed by an app-id/build-id. Upload those symbols to your crash reporter (Crashlytics: the Flutter symbol upload / Gradle plugin; Sentry: `sentry-cli debug-files upload`) as part of CI. At runtime the device reports **obfuscated** frames; the backend **symbolicates** them using the matching build's symbols. Critical rule: symbol files are **per build** — a mismatched or missing upload yields unreadable traces, so wire the upload into your release pipeline, not a manual step. See 50 CI CD.

30. What do you actually implement to cover the OWASP Mobile Top 10 in a Flutter app? A pragmatic mapping:

- **M1 Improper credential/platform usage** → correct permissions, no misuse of platform APIs, biometrics done right.
- **M2 Inadequate supply chain** → audit/pin dependencies, verify plugin provenance.
- **M3 Insecure auth/authz** → short-lived tokens, server-side authz, refresh rotation.

- **M4 Insufficient input/output validation** → validate server-side; sanitize deep links/WebView input.
- **M5 Insecure communication** → TLS everywhere, optional pinning, no cleartext.
- **M6 Inadequate privacy** → minimize + redact PII, secure storage, clear on logout.
- **M7 Insufficient binary protection** → `--obfuscate`, integrity/attestation checks.
- **M8 Security misconfiguration** → no debug flags/verbose logs in release, locked-down network config.
- **M9 Insecure data storage** → Keychain/Keystore for secrets, never `SharedPreferences`.
- **M10 Insufficient cryptography** → use vetted libraries/OS crypto, never hand-rolled or hardcoded keys. The senior point: it's a checklist to *reason with*, and the recurring theme is "never trust the client — enforce on the server."

31. A screenshot of your app in the recent-apps switcher leaks a token/PII. How do you fix it, and what class of problem is this? This is **insecure data exposure via the OS**, not your storage. Prevent the OS from snapshotting sensitive screens: on Android set `FLAG_SECURE` on the window (blocks screenshots and hides the app in the recents thumbnail); on iOS overlay a blur/splash on `didEnterBackground` and remove it on resume. Also consider hiding sensitive fields when the app is backgrounded and clearing the clipboard. It's a reminder that "secure storage" doesn't cover *rendered* data — the screen, clipboard, backups, and logs are all exfiltration surfaces.

32. How do you prevent sensitive data from leaking through logs in a large team, structurally? Don't rely on discipline — build guardrails:

- **Centralize** logging behind one facade; forbid raw `print` / `debugPrint` via a lint rule (custom `analysis_options.yaml` rule or `avoid_print`).
- **Redact at the sink**: a single formatter masks known-sensitive keys and headers so no call site can leak them.
- **Strip in release**: gate verbose/debug logs behind `kDebugMode`; ship only `warning` + to remote.
- **Type your log events** (structured) so you log fields deliberately, not whole objects (`toString()` of a User model can dump everything).
- **Review + test**: a redaction unit test on the formatter, and PR review of new log statements. The structural fix is making the *safe* path the *only* easy path.

33. Your app must run on rooted devices (you can't just block them) but protect a payment flow. What's your layered strategy? Assume the device is compromised and push trust to the server:

- **Attestation**: require Play Integrity / App Attest tokens (via App Check) on payment endpoints — validated server-side.
- **Don't trust client checks**: root detection only *signals risk* (step-up auth, extra verification), never gates security alone.

- **Minimize on-device secrets**: keep payment secrets server-side; the client holds only short-lived, scoped tokens in secure storage.
- **Server-side enforcement**: authorize every payment on the backend with idempotency keys and anomaly detection; the client can't be the source of truth.
- **Transport**: TLS + pinning to stop MITM of the payment call.
- **Sensitive UI hardening**: `FLAG_SECURE`, biometric confirmation, no PANs in logs. The core answer interviewers want: *the client is untrusted; security is enforced and verified on the server.* See 31 Payments.

34. How do you handle errors across Clean Architecture layers so the UI shows meaningful messages without leaking internals? Map errors at each boundary. The **data layer** catches transport errors (`DioException`, `SocketException`, platform exceptions) and converts them into domain `Failure` types or domain exceptions — the domain never imports Dio. The **domain/use-case layer** returns `Either<Failure, T>` (or throws domain exceptions). The **presentation layer** maps each `Failure` to a user-facing message and state (offline banner, retry, "session expired → re-login"), and reports the original error + stack to Crashlytics. This keeps stack traces and internal details *out* of the UI while preserving them for observability, and gives users actionable, non-leaky messages. See 40 Clean Architecture.

35. `--dart-define` vs a bundled `.env` file for configuration and "secrets" — compare. `--dart-define` / `--dart-define-from-file` injects values at **compile time** as constants (tree-shakeable, usable in `const`), whereas a `.env` asset is read at **runtime** and shipped as a bundled file. `.env` files are trivially extractable from the asset bundle; `--dart-define` values are compiled in but still **recoverable from the binary** (strings/decompilation). So for *config* (base URLs, feature flags, environment name) either works and `--dart-define` is cleaner and per-flavor; for actual **secrets, neither is safe** — the only correct place is the server. The interview trap is claiming `--dart-define` "hides" secrets: it doesn't.

⚡ Rapid-fire (one-liners)

Question	Answer
Where do refresh tokens go?	`flutter_secure_storage` (Keychain/Keystore), never `SharedPreferences`.
`SharedPreferences` encrypted?	No — plaintext; UI state only.
Access token vs refresh token?	Short-lived, sent every request vs long-lived, only refreshes access.
Why is a client secret impossible?	Anything in the binary is extractable — keep it server-side.
Command to obfuscate?	`flutter build apk --obfuscate --split-debug-info=<dir>`.
Does obfuscation encrypt/hide strings?	No — just renames symbols; not a secret store.
Certificate pinning risk?	Bricks the app on cert rotation if pins aren't updated.
`Error` vs `Exception`?	Bug you fix vs recoverable condition you catch.
Catch a specific type?	`on ExceptionType catch (e, st)`.
What always runs?	`finally`.
Catch uncaught async errors?	`PlatformDispatcher.instance.onError` (or `runZonedGuarded`).
Catch framework build errors?	`FlutterError.onError`.
Result/Either vs throw?	Expected failures as data vs the truly exceptional.
Never log what?	Tokens, passwords, PII.
Structured logging?	Machine-parseable JSON records with consistent fields.
Three pillars of observability?	Logs, metrics, traces.
Root detection guarantee?	No — it's a risk signal, spoofable on-device.
App Check / Play Integrity verifies?	That the request is from a genuine app instance — server-side.
Hide sensitive screen in recents?	`FLAG_SECURE` (Android) / blur overlay (iOS).

Follow-up drills

1. **Design** the full network interceptor stack for a banking app: token attach, single-flight refresh with rotation, cert pinning, App Check header, retry, and PII-safe logging.
2. **Debug this scenario**: production crashes show only obfuscated frames like `MidsizeA.b()`. Walk through why and fix the pipeline end to end.
3. **Optimize** a logging setup that's costing too much and leaking PII into Sentry — redesign levels, redaction, and remote sampling.
4. **Design** an error-handling contract across data/domain/presentation layers so the UI never shows a raw exception yet every error reaches Crashlytics with a trace id.
5. **Threat-model** a Flutter app against the OWASP Mobile Top 10 and rank which mitigations you'd ship first for a fintech MVP.

6. **Debug this scenario**: after adding `runZonedGuarded`, the app throws a "Zone mismatch" assertion at startup — diagnose and fix.

Mobile System Design (Flutter) — Interview Questions

> How to design a Flutter client end-to-end under real mobile constraints: offline, flaky network, battery, app size, sync, and rollout. For depth see 48 System Design and 19 Offline First.

This topic tests whether you can scope a mobile feature and reason about the *client* as a system — not just draw boxes for servers. Interviewers push from "how would you call this API" toward "how does the app behave on a subway with 2% battery, on a 3-year-old phone, on app version N-3, with 50k queued writes".

🟢 Basic

1. How does a mobile system design interview differ from a backend one? Backend rounds optimize a server you fully control: throughput, sharding, consistency, availability. Mobile rounds optimize a client you *don't* control — it runs on someone else's device, on a network you can't trust, and on an app version you can't force-update instantly. The scarce resources flip: instead of CPU/DB, you're rationing battery, memory, disk, cellular data, cold-start time, and app-store binary size. Correctness now includes offline behavior and eventual consistency. A strong candidate leads with these client constraints instead of designing a distributed backend.

2. What are the core client-side constraints you should name up front?

- **Flaky/variable network** — connections drop, switch WiFi↔cellular, have high latency and packet loss; requests must be retryable and cancellable.
- **Offline** — the app must do something useful with no connectivity, not just show a spinner.
- **Battery & radio** — waking the cellular radio and GPS is expensive; batch and defer work.
- **Limited memory** — the OS kills backgrounded apps; large images/lists cause OOM.
- **App (binary) size** — every MB hurts install/update conversion; watch assets and native deps.
- **Versioning** — many app versions run in the wild simultaneously; your API and schema must tolerate old clients (and you need forced-update for the ones you can't).
- **Deep linking** — the app can be entered at any screen from a URL/notification.

3. Walk me through a framework for answering a mobile design question.

1. **Requirements** — functional (what screens/actions) and non-functional (offline? real-time? scale? latency budget?). Clarify, don't assume.
2. **API contract** — endpoints, request/response shapes, pagination, error model, auth.
3. **Data model** — domain entities and their relationships.
4. **Local cache / DB** — what persists on device, in what store, with what schema.
5. **Sync strategy** — how local and server state reconcile, including conflicts.

6. **State management / architecture** — layering (UI → domain → data), which state solution, DI.
7. **Non-functionals** — offline, performance, security, observability, rollout, testing. Say it out loud as a checklist; it signals structure and keeps you from rat-holing on the backend.

4. What does "requirements gathering" look like for a client feature? Ask scoping questions that change the design: *Does it work offline? Read-only or does the user create data? Real-time updates or pull-to-refresh? How large is the dataset (10 items vs infinite feed)? What's the latency expectation? Which platforms? Existing auth?* Each answer flips a decision — e.g. "must work offline + user writes" forces a local DB and a sync queue, while "read-only, small, online-only" is just an in-memory cache.

5. Why is the repository the key boundary in a mobile system? The repository is the single seam between the domain layer and *all* data sources (network, DB, cache, prefs). The UI/domain asks the repository for data and never knows whether it came from the server or SQLite. This lets you implement offline-first (serve cache, refresh in background), swap sources, retry, and test the domain with a fake repo. In an interview, draw the repository as the box where the offline/caching/sync policy lives.

6. What's a good default layered architecture for the client? `UI (widgets) → State (BLoC/Riverpod/ViewModel) → Domain (use cases + entities) → Data (repositories → data sources)`. Dependencies point inward; the domain has no Flutter or HTTP imports. This is Clean Architecture applied to a client. It matters here because it isolates the volatile parts (API shapes, DB) behind stable interfaces, which is exactly what you need when the backend and app versions drift.

7. What is offline-first and when do you need it? Offline-first means the local store is the source of truth for the UI: reads come from the DB, writes go to the DB immediately (optimistic), and a background process syncs with the server. You need it when users create/edit data on the go (notes, messaging, field apps) or when connectivity is unreliable. Contrast with "offline-tolerant" (cache-only reads, no local writes) which is much simpler. See 19 Offline First.

8. What is optimistic UI and why does it matter on mobile? Optimistic UI applies a user action to local state/UI *immediately*, before the server confirms, then reconciles when the response arrives (or rolls back on failure). It hides network latency and makes the app feel instant on flaky connections. The cost is you must handle rollback and conflicts. Classic examples: sending a chat message (shows as "sending"), liking a post, adding a to-do.

9. How do you handle pagination for long lists? Two common contracts: **offset/limit** (`?page=2&size=20`) — simple but breaks when items shift; and **cursor/keyset** (`?after=<opaque_cursor>`) — stable under inserts/deletes, preferred for feeds. On the client, fetch the next page when the scroll nears the end, append to the list, dedupe by ID, and show a footer loader. Cache pages so back-navigation is instant.

10. What are the options for getting real-time updates to a mobile client?

- **Polling** — client asks on an interval; simple, works everywhere, wastes battery/data and is stale between polls.
- **Long-polling** — request held open until data or timeout; less waste, more server load.
- **SSE (Server-Sent Events)** — one-way server→client stream over HTTP; great for feeds/notifications, simple to proxy.
- **WebSockets** — full-duplex; needed for chat, presence, collaborative editing.
- **Push (FCM/APNs)** — OS-level delivery even when the app is closed; use it to *wake* the app, not to carry payloads reliably.

11. Why can't push notifications be your source of truth? FCM/APNs are best-effort: messages can be dropped, delayed, coalesced, or throttled, and the OS may not wake a killed app. Treat a push as a *hint* ("something changed, come sync") — its payload should be minimal, and the app must fetch authoritative data from the API on receipt. Designing a chat app that relies on push payloads to deliver every message is a red flag.

12. What is a deep link and why does it affect your architecture? A deep link is a URL (or notification) that opens the app directly at a specific screen/state, e.g. `myapp://chat/42`. It forces your navigation/routing to be declarative and reconstructable from a URL — you must be able to build the whole back stack for an arbitrary destination, handle the not-logged-in case, and cold-start into it. This is why URL-based routers (go_router) are the interview-safe answer over imperative `Navigator.push`.

🟡 Intermediate

13. Design the API contract and error model before coding — what do you specify? Endpoints + verbs, request/response JSON shapes, pagination scheme, and a **consistent error envelope** (e.g. `{ "code": "RATE_LIMITED", "message": ..., "retryable": true }`). Specify auth (bearer token + refresh), idempotency keys for writes, and versioning (`/v1`). Client-side you map transport errors and status codes into a small domain `Failure` type (`NetworkFailure`, `AuthFailure`, `ServerFailure`, `ConflictFailure`) so the UI reasons about *meaning*, not HTTP codes. See 16 Networking.

14. Why do writes need idempotency keys, and where does the client generate them? On flaky networks a request may succeed on the server but time out on the client, which then retries — creating duplicates (double-posted message, double payment). The client generates a UUID per logical operation and sends it (header or body); the server dedupes on it. The client stores this key in its outbox so a retry reuses the *same* key. This is the single most important detail for correct retries.

15. How do you choose a local storage tier? | Need | Use | |---|---| | Small key/value flags, tokens | `shared_preferences` (non-sensitive) / secure storage (tokens) | | Structured, queryable, relational | SQLite via `drift` / `sqflite` | | Fast object store, simple | `hive` / `isar` | | Large blobs

(images, files) | File system + path, store path in DB | Don't put images in SQLite or tokens in prefs. Pick relational (drift) when you need joins, migrations, and complex queries — which most offline-first apps eventually do. See 15 Local Storage and 20 Database.

16. Describe a cache with a staleness policy. Store each entity with a `fetchedAt` timestamp. On read, the repository returns cached data immediately (stale-while-revalidate) and kicks off a network refresh if `now - fetchedAt > ttl`. Use `ETag` / `If-None-Match` or `Last-Modified` / `If-Modified-Since` so a refresh returns `304 Not Modified` with no body when nothing changed — saving battery and data. Distinguish TTL (soft, triggers refresh) from hard expiry (evict/never serve).

17. How does offline-first sync actually work — the outbox pattern? Local writes go into the DB and are appended to an **outbox** (a pending-operations queue) with an idempotency key and status. A sync worker drains the outbox when connectivity returns (listen to `connectivity_plus`), sending each op, marking it synced or failed-with-retry (exponential backoff). Reads always come from the DB, so the UI is consistent regardless of network. The server change feed (or a pull) brings remote changes back down. This decouples user actions from network availability.

18. How do you detect and pull remote changes efficiently? Delta sync: the client stores a high-water mark (a server `updatedAt` timestamp or a monotonic `sync_token`/cursor) and asks `GET /changes?since=<token>`. The server returns only rows changed since then, plus a new token. This avoids re-downloading everything and scales to large datasets. Tombstones (soft-delete rows with a `deleted` flag) let deletes propagate — a plain "row missing" is ambiguous over deltas.

19. What conflict-resolution strategies exist and when do you use each?

- **Last-write-wins (LWW)** — compare timestamps/versions, newest wins. Simple, lossy; fine for single-user-multi-device settings.
- **Server-wins / client-wins** — one side is authoritative; predictable but discards intent.
- **Merge / field-level** — combine non-conflicting fields; good for documents.
- **CRDTs** — conflict-free replicated data types converge automatically; used for real collaborative editing (notes, whiteboards) but complex.
- **Prompt the user** — surface the conflict (rare, high-value data). Pick based on data value and multi-writer likelihood. Most apps start with LWW + version numbers and escalate only where it hurts.

20. How do you version an API/schema so old clients don't break? Additive, backward-compatible changes: never remove or repurpose a field; add new optional fields; version the base path (`/v1`) for breaking changes and run versions in parallel during migration. Clients must ignore unknown fields (tolerant reader). For the local DB, ship ordered, tested migrations (drift/sqflite `onUpgrade`) and never assume a fresh install. Because users sit on old app versions for months, the server contract is effectively forever.

21. When and how do you force an app update? When an old client is unsafe or incompatible (breaking API change, security fix, broken protocol). Implement a **minimum-supported-version** check: the app sends its version, or fetches a `minVersion` from a config/feature-flag endpoint on launch; if below, show a blocking "Update required" screen linking to the store. Keep a soft-nudge tier for recommended updates. Never hard-block without a server-controlled kill switch — you may need to relax it.

22. Design the image/media pipeline for a media-heavy app.

- **Server** provides multiple sizes/thumbnails and a CDN URL (don't download a 4MB original for a 100px avatar); support `WebP/AVIF`.
- **Client** uses `cached_network_image` (memory + disk cache), requests the right resolution via `cacheWidth` / `cacheHeight` so it decodes downscaled and doesn't blow up memory.
- **Uploads** compress/resize on device first, upload in the background with resumable/multipart requests, show progress, and retry.
- Cap the disk cache and evict LRU. Placeholders + fade-in for perceived speed.

23. How do you keep app size and cold-start under control? Split ABIs / use app bundles (Play delivers per-device), tree-shake icons, compress/deduplicate assets, lazy-load features (deferred components / dynamic feature modules), and audit heavy native deps. For cold start, defer non-critical init off the first frame, avoid synchronous disk/network in `main()`, and pre-warm nothing you don't render. Measure with `--analyze-size` and startup traces. See 21 Performance.

24. What belongs in feature flags vs A/B tests, and how do they reach the client? Feature flags gate code paths (kill switch, gradual rollout, entitlement); **A/B tests** assign users to variants and measure a metric. Both come from a remote config service (Firebase Remote Config, LaunchDarkly) fetched on launch with a cached fallback so a config-server outage doesn't brick the app. Evaluate flags behind an interface so the rest of the app just reads `flags.isEnabled('newFeed')`, and default to the safe value when config is unavailable.

25. What client observability do you need and why? Crash reporting (Crashlytics/Sentry), structured logs, performance traces (cold start, frame drops, network latency), and analytics events for funnels. On mobile you're blind without it — you can't SSH into a user's phone. Tie events to app version and device class so you can spot a regression scoped to one rollout. Respect privacy: no PII in logs, sample high-volume events. See 52 Monitoring.

🔴 Advanced

26. How do you roll out a risky client change safely? Ship it dark behind a feature flag, do a **staged rollout** on the store (1% → 5% → 25% → 100%) while watching crash-free-sessions and key metrics, and keep a server-side kill switch to disable the feature without a new binary. Because you can't recall a bad build instantly, the flag + kill switch is your real rollback. Halt/roll back the store rollout on regression, and add a forced-update floor only for truly broken versions.

27. At scale, how do you keep sync efficient with tens of thousands of local records? Never full-sync: use cursor-based delta sync with tombstones and a per-entity high-water mark. Batch and paginate the change feed. Do writes through a bounded outbox with backoff and coalescing (collapse repeated edits to the same record before sending). Run sync in a background isolate / WorkManager job so it doesn't jank the UI, and index the DB columns you filter/sort on. Guard against thundering-herd on reconnect by jittering retry timers.

28. WebSocket vs SSE vs push — how do you combine them for a chat app? Use a **WebSocket** for the active conversation (low-latency bidirectional messages, typing/presence). Use **push (FCM/APNs)** to wake the app and notify when it's backgrounded/killed — the push just says "new message in room X", and the app fetches authoritative messages over REST/WS. SSE is a lighter alternative when you only need server→client (e.g. a live feed, not chat). The durable source of truth is always the server's message store fetched by ID/cursor, never the transport.

29. How do you guarantee message ordering and no-loss in chat over a flaky link? Server assigns a monotonic, gap-detectable sequence (per-room `seq` or Lamport/HLC clock); the client orders by it and can detect a gap → refetch the range. Outbound messages carry a client-generated idempotency/temp ID so retries dedupe and the server echo reconciles the optimistic bubble. On reconnect, the client sends its last-seen `seq` and pulls the delta. Acks move state `sending → sent → delivered → read`. This is delta sync applied to a message log.

30. How do you secure data and tokens on the client? Store tokens in platform secure storage (Keychain/Keystore via `flutter_secure_storage`), never in `shared_preferences`. Use short-lived access tokens + refresh tokens with silent refresh-on-401 in an interceptor. Pin certificates for high-value APIs, encrypt sensitive local DBs (SQLCipher), and assume the device can be rooted — don't trust client-side checks for anything security-critical; enforce on the server. See 37 Security and 17 Authentication.

31. How do you test an offline-first system design? Unit-test the repository and sync logic with fake data sources and a controllable clock/connectivity. Test conflict resolution with concrete divergent-edit scenarios. Integration-test the outbox: enqueue offline, toggle connectivity, assert the server received deduped, correctly-ordered ops. Simulate flaky network (latency, drops, 500s) to verify retries/backoff. The design is only as good as its behavior at the boundaries, so most bugs live in sync — test it hardest. See 49 Testing.

32. What are common mistakes candidates make in a mobile design round? Designing the backend instead of the client; ignoring offline and assuming connectivity; treating push payloads as reliable delivery; no idempotency on retried writes; putting business logic in widgets; unbounded caches (memory/disk) causing OOM; full re-sync instead of deltas; forgetting versioning/forced-update and deep linking; and no observability. Also: jumping to code before stating requirements and the API contract.

33. How would you architect for multiple app versions in the wild simultaneously? Assume version N-k is live for months. Keep the API additive and version-gated; feature-flag new behavior so the server can turn features on only for capable versions; use a `minVersion` gate for forced updates. Make the local DB migrations forward-only and tested against every prior schema. Design analytics/crash data to carry the app version so you can diff behavior per version. Never make a server change that assumes all clients upgraded overnight.

34. WORKED — Design an offline notes app. Sketch the design.

- **Requirements**: create/edit/delete notes offline, sync across the user's devices, search, near-instant UI. Multi-device single-user → conflicts possible.
- **API**: `GET /notes?since=<token>` (delta + tombstones), `POST /notes` / `PATCH /notes/:id` with idempotency key + `version`, `DELETE /notes/:id` (soft delete).
- **Data model**: `Note { id (client-generated UUID), title, body, updatedAt, version, deleted }`.
- **Local DB**: SQLite/drift; `notes` table + `outbox` table (op, payload, idempotencyKey, status). FTS index for search.
- **Reads**: always from DB → instant, works offline.
- **Writes**: write DB + enqueue outbox (optimistic). Sync worker drains outbox on connectivity, uses server `version` for **LWW/field-merge** conflict resolution; server bumps `version`.
- **Pull**: delta sync with `sync_token`; apply tombstones.
- **Architecture**: `NotesRepository` as the boundary; Riverpod/BLoC for state; go_router with deep links to a note.
- **Non-functionals**: encrypt DB if notes are sensitive, cap outbox with backoff, observability on sync failures.

35. WORKED — Design a news feed. Sketch the design.

- **Requirements**: infinite personalized feed, read-mostly, works semi-offline (show last feed), pull-to-refresh, media-heavy, huge scale (read latency + battery matter).
- **API**: `GET /feed?cursor=<opaque>&limit=20` returning items + `nextCursor` (keyset pagination, stable under inserts). Items reference media by CDN URL with multiple sizes.
- **Data model**: `FeedItem { id, authorId, text, mediaRefs[], createdAt, counters }`.
- **Local cache**: SQLite table of feed items keyed by id + a `feed_page` ordering table; `fetchedAt` for staleness. Serve cached feed on cold start (stale-while-revalidate), refresh top on pull.
- **Pagination**: cursor-based; prefetch next page near scroll end; dedupe by id.
- **Media**: `cached_network_image` with sized thumbnails + `cacheWidth`; bounded LRU disk cache.
- **Real-time**: lightweight — pull-to-refresh + optional SSE/push "new posts available" pill; no WebSocket needed.
- **Architecture**: `FeedRepository` merges cache + network; BLoC with `FeedState(items, hasMore, isLoadingMore)`.

- **Non-functionals**: rank server-side (client stays dumb), feature-flag ranking experiments (A/B), analytics on impressions/scroll depth, staged rollout of ranking changes, virtualized `ListView.builder` for memory.

⚡ Rapid-fire (one-liners)

Question	Answer
First thing to state in a mobile design round?	Requirements — functional + non-functional (offline? real-time? scale?).
Where does caching/offline/sync policy live?	The repository (the domain↔data boundary).
Best pagination for a feed?	Cursor/keyset — stable under inserts/deletes.
Make a retried write safe how?	Client-generated idempotency key, reused on retry.
Reliable message delivery via push?	No — push is a hint; fetch authoritative data from the API.
Chat transport?	WebSocket active + push to wake when backgrounded.
One-way live feed transport?	SSE (or polling as the fallback).
Store auth tokens where?	Platform secure storage, never `shared_preferences`.
Offline write mechanism?	Outbox/queue drained by a sync worker on reconnect.
Pull remote changes efficiently?	Delta sync with a cursor/ `since` token + tombstones.
Default conflict strategy?	LWW with version numbers; escalate to merge/CRDT if needed.
Handle breaking API changes?	Version the path (`/v1`), additive fields, tolerant reader.
Force an update how?	Server-driven `minVersion` gate + blocking screen.
Real rollback for a bad feature?	Feature flag / kill switch, not a new binary.
Avoid image OOM?	Sized thumbnails + `cacheWidth` / `cacheHeight`, bounded cache.
Reduce app size?	App bundles/split ABIs, tree-shake, lazy features, audit deps.

Follow-up drills

1. Design a **ride-hailing rider app**: live driver location on a map, request→match→trip lifecycle, ETA, payment — pick transports and offline behavior for each phase.
2. Design a **chat app** end-to-end: ordering, no-loss over flaky links, typing/presence, media, and multi-device read state.
3. Your feed shows **duplicate and out-of-order items** after a refresh on a slow network — diagnose the pagination/caching bug and fix the contract.
4. Users report **notes lost after editing on two devices offline** — design a conflict-resolution scheme that stops the data loss.
5. A **staged rollout** shows a crash spike at 5% scoped to Android 12 on one OEM — walk through your rollout controls and how you contain it without a new release.

6. Cut a media-heavy app's **cellular data usage by 50%** without hurting perceived quality — enumerate the levers.

CI/CD, Deployment & Monitoring — Interview Questions

> How you build, sign, ship, and observe a Flutter app in production — from `flutter build` flags to store rollout to crash dashboards. For depth see the handbook modules 50 CI CD, 51 Deployment, and 52 Monitoring.

This topic tests whether you can take a green test suite all the way to a monitored, rollback-able production release. Interviewers escalate from "AAB vs APK" to "your release is crashing 3% of users at 40% rollout — what do you do?" — and the honest answer is not "push a hotfix," because you can't un-install an app off someone's phone.

🟢 Basic

1. What are the three build modes and when is each used?

- **Debug** (`flutter run`): JIT-compiled, assertions on, hot reload, DevTools, service extensions. Slow and large — never ship it.
- **Profile** (`flutter run --profile`): AOT-compiled like release but keeps some tracing/DevTools hooks so you can measure real performance. Debugging aids are stripped. Used only for performance profiling on a real device (not an emulator).
- **Release** (`flutter build`): AOT-compiled, assertions off, no debugging/observatory, tree-shaken, minified. This is what ships to stores.

The key "why": profile exists because debug's JIT and asserts make timings meaningless, but release strips the tooling you'd need to inspect anything — profile is the honest middle.

2. AAB vs APK — what's the difference and which do you upload? An **APK** is an installable Android package. An **AAB** (Android App Bundle) is a *publishing* format you upload to Play; Google Play then generates and signs optimized, per-device APKs from it (Dynamic Delivery). AAB is mandatory for new Play submissions. Use `flutter build appbundle` for the store, `flutter build apk` for sideloading/CI artifacts/other stores. AABs enable **app thinning** — a device downloads only the density, ABI (arm64 vs x86), and language resources it needs, so download size shrinks.

3. What is app thinning / split APKs? Instead of one fat universal binary carrying every screen density, CPU architecture, and locale, the store delivers only the slices a given device needs. On Android this is Play's split APKs from an AAB (or `flutter build apk --split-per-abi` if you distribute yourself); on iOS it's App Thinning (slicing + on-demand resources). Result: smaller install, faster download, less user churn from "app too big."

4. What are flavors and why use them? Flavors are build variants of the *same* app — typically `dev`, `staging`, `prod` — each with its own app ID (`com.acme.app.dev`), name, icon, and backend config, so all three can sit on one device at once. You configure them natively (Android product flavors in `build.gradle`, iOS schemes/configurations) and select at build time: `flutter build apk --flavor prod -t lib/main_prod.dart`. Why: keep test and production data/keys fully separated and installable side by side.

5. What is `--dart-define` and what problem does it solve? `--dart-define=API_URL=https://api.prod.com` injects a compile-time constant read via `String.fromEnvironment('API_URL')` (also `int` / `bool.fromEnvironment`). It lets you vary config (base URLs, flags, non-secret keys) per build without editing source. Because the value is baked in at compile time, it's `const` and tree-shakeable — but it is **not secret**; anyone can extract it from the binary.

```
const apiUrl = String.fromEnvironment('API_URL', defaultValue: 'https://api.dev.com');
```

6. What is `--dart-define-from-file` and why is it better than many `--dart-define` flags? `--dart-define-from-file=config/prod.json` reads a JSON (or `.env`-style) file of key/values in one flag instead of a dozen brittle `--dart-define`s. It keeps CI commands short, lets you version or `.gitignore` config per environment, and pairs naturally with flavors (one file per flavor). Same compile-time semantics as `--dart-define`.

7. What is versioning in `pubspec.yaml` — version name vs build number? `version: 1.4.2+57` — the part before `+` is the **version name** (`1.4.2`, user-facing, semver) and after `+` is the **build number** (`57`, `versionCode` / `CFBundleVersion`). Stores require the build number to strictly increase with every upload even if the version name is unchanged; two uploads can't share a build number. Override at build time with `--build-name` and `--build-number`, which CI usually sets from the pipeline run number.

8. At a high level, why does an app need to be signed? Signing cryptographically proves the app came from you and hasn't been tampered with, and it's how the OS decides an update is "the same app" as what's installed. On **Android** you sign with a keystore (a `.jks` / `.keystore` holding your key); losing it (pre–Play App Signing) means you can never update that listing again. On **iOS** you need a signing **certificate** (identifies the developer) plus a **provisioning profile** (ties app ID + certificate + allowed devices/capabilities). Debug builds are auto-signed with a throwaway debug key; release builds must use your real signing config.

9. What is CI/CD in the context of a Flutter app? CI (Continuous Integration) runs automated checks on every push/PR — format, analyze, test, build — so broken code never merges. **CD** (Continuous Delivery/Deployment) takes a passing commit and produces/ships a release artifact (to a store track or a distribution service), ideally automatically. The point is a repeatable, auditable path from commit to users that a human can't forget a step in.

10. What does a typical Flutter CI pipeline run, in order? A gated sequence where each stage must pass before the next:

1. `flutter pub get`
2. `dart format --set-exit-if-changed .` (fail on unformatted code)
3. `flutter analyze` (static analysis / lints)
4. `flutter test --coverage` (unit + widget)
5. `flutter build apk/appbundle/ipa` (the build actually compiles)

Ordering is deliberate: cheap, fast checks first so you fail in seconds, not after a 10-minute build.

11. Name common CI/CD tools for Flutter and what each is good at.

- **GitHub Actions**: general CI, free-ish for open source, YAML workflows, huge action marketplace; you assemble the Flutter steps yourself.
- **Codemagic**: Flutter-specialized SaaS — understands flavors, signing, and store publishing out of the box; least setup for mobile.
- **Bitrise**: mobile-focused CI with a visual step library and strong signing/deploy support.
- **Fastlane**: not a CI host — a Ruby automation toolkit (lanes) that scripts signing, screenshots, and store uploads; you call it *from* any of the above.

12. What is Firebase Remote Config in one sentence? A cloud key/value store you fetch at runtime to change app behavior (values, thresholds, flags) without shipping an update — the backbone of feature flags and kill switches.

🟡 Intermediate

13. Why can't you just "roll back" a mobile release the way you roll back a web deploy? Because the artifact lives on the user's device, not your server. Once someone installs v2.0, you cannot reach into their phone and revert them to v1.9 — stores don't downgrade installed apps, and you can't force-uninstall. So "rollback" on mobile means containing the blast radius of what's already out and stopping it from spreading. Your levers are: **halt/pause the staged rollout** so new users stop getting the bad build, flip a **remote-config kill switch** to disable the broken feature in-place, and if it's truly unusable, ship a **forced-update** prompt pushing everyone to a fixed version. There is no instant undo — which is exactly why staged rollout and feature flags matter.

14. What is a staged / phased rollout and why use it? You release the new version to a small percentage of users first (Play: staged rollout at e.g. 1% → 5% → 20% → 50% → 100%; App Store: phased release over 7 days, auto-stepping daily). You watch crash-free rate and key metrics at each step; if release health degrades, you **halt** before most users ever get it. It converts a potential 100%-of-users incident into a 1%-of-users incident. Always pair it with monitoring — a staged rollout you don't watch is just a slow full rollout.

15. How do feature flags let you decouple deploy from release? Ship the code dark (behind a flag defaulting off), then turn it on via remote config for a cohort without a new build. This means the risky code is already through review and in the store, and "launching" is a config change you can also instantly reverse. It enables canaries, A/B tests, and gradual enablement, and gives you a **kill switch**: if the feature misbehaves, flip the flag off for everyone in seconds — no store review, no update download.

16. How do you implement a forced update? On startup (or resume) the app asks the backend/remote-config for a `minSupportedVersion` (and optionally `latestVersion`). Compare against the running build number/version:

- running `<` minSupported → show a **blocking**, non-dismissible dialog with a store link ("Update required").
- running `<` latest but `≥` minSupported → show a **soft**, dismissible "update available" nudge.

Keep the threshold server-controlled so you can raise the floor the moment a bad version is in the wild. Guard the check so a network failure doesn't lock users out.

17. `--dart-define` vs `--flavor` — when do you use which, and can you use both? They're orthogonal and usually combined. **Flavor** = a distinct build variant with its own native identity (app ID, icon, name) — heavier, defined in Gradle/Xcode. `--dart-define` = injecting Dart-side config values into a build — lightweight, no native change. Use flavors when you need separate installable apps or different app IDs; use `--dart-define(-from-file)` for the config values inside each. Typical: `flutter build appbundle --flavor prod -t lib/main_prod.dart --dart-define-from-file=env/prod.json`.

18. Why must you never put secrets (API keys, signing passwords) in `--dart-define` or source? `--dart-define` values are compiled into the binary as plain constants — trivially recoverable by decompiling or `strings` on the artifact. Client apps can't hold true secrets; anything on the device is public. Real secrets belong on the backend or in a secrets store (CI secret variables, KMS), and the app should authenticate to a server that holds them. `--dart-define` is fine for *non-secret* config (base URLs, feature toggles, public keys).

19. How is signing configured securely in CI? You never commit the keystore or certificates. Store the keystore/ `.p12` /provisioning profile and their passwords as **encrypted CI secrets**, decode them into the runner at build time, and point the build config at the decoded files. On Android, `key.properties` is generated from secrets, not checked in. Prefer **Play App Signing** (Google holds the app signing key; you keep only an upload key you can reset if lost) and iOS **fastlane match** (certs/profiles encrypted in a git repo, synced to CI) so signing is reproducible and a lost laptop isn't a catastrophe.

20. What's the difference between crash reporting and analytics? Crash reporting (Crashlytics, Sentry) captures *failures* — stack traces, device/OS, breadcrumbs — so you can diagnose and prioritize by impact (users affected, crash-free rate). Analytics (Firebase Analytics,

Amplitude) captures *intended behavior* — screen views, funnels, feature usage — to inform product decisions. Different questions: crash reporting answers "what's broken and for how many," analytics answers "what are people doing." You want both, plus a way to correlate them (e.g. did the crash tank the checkout funnel).

21. What is an ANR and how do you catch it? An **ANR** (Application Not Responding) is Android detecting that the main/UI thread was blocked too long (~5s for input) and offering to kill the app. In Flutter it usually means heavy work on the platform main thread or a jammed platform channel — not typically pure Dart on the UI isolate, but native plugin work or a synchronous channel call can cause it. You surface ANRs via Play Console's Android vitals and Crashlytics' ANR reporting. Fix by moving work off the main thread (background isolate / `compute`, async platform calls).

22. What is release health and which metric matters most? Release health is the aggregate stability of a specific version: **crash-free users** and **crash-free sessions**, plus ANR rate, adoption, and error volume — tracked per release. **Crash-free users %** is the headline metric for rollout decisions (e.g. "hold rollout if crash-free users drops below 99.5%"). It's per-*user* not per-*session* because one user crash-looping shouldn't look like thousands of independent failures.

23. What are performance traces and what do you monitor in production? Custom traces (Firebase Performance `Trace`, Sentry transactions) measure real-world durations you care about — app start, screen render, a network call, a checkout step — across real devices you'll never own. Firebase Performance auto-captures app start, screen rendering (slow/frozen frames), and HTTP request timing; you add custom traces for domain flows. You monitor p90/p99 (tails, not averages), slow/frozen frame rates, and network success/latency — because a 200ms median hides the users on cheap phones and bad networks who churn.

24. How do symbols/obfuscation interact with crash reporting? Release builds are AOT-compiled and can be obfuscated (`flutter build ... --obfuscate --split-debug-info=build/symbols`), which turns stack traces into meaningless addresses. To get readable traces you must **upload the symbol files** (Dart symbols, plus Android ProGuard/R8 mappings and iOS dSYMs) to your crash service for that exact build. If you obfuscate but forget to upload symbols, every crash report is undebuggable — a very common production footgun.

25. What is a canary / kill switch and how does remote config provide it? A kill switch is a remote-config boolean that instantly disables a feature (or the whole app path) for everyone, without a store update. Because the app reads it at runtime, flipping it in the Firebase console propagates on the next fetch — seconds to minutes. It's your fastest incident-response lever on mobile precisely because store review and forced updates are slow. Design risky features to fail *closed* (flag off = old safe behavior).

26. How do you set version name / build number in CI without editing pubspec by hand? Pass them at build time: `flutter build appbundle --build-name=1.4.2 --build-number=${{ github.run_number }}`. Deriving the build number from a monotonic CI counter (run number, or commit count)

guarantees it always increases and never collides on the store, and keeps `pubspec.yaml` from churning on every release. The version name typically comes from a git tag.

🔴 Advanced

27. Design the rollback/incident-response playbook for "v2.3 crashes 4% of users, currently at 20% staged rollout." There's no un-install, so contain and stop the spread:

1. **Halt the rollout immediately** (Play: pause staged rollout; App Store: pause phased release) — new users stop getting v2.3.
2. **Identify the cause** in Crashlytics — is the crash gated behind a feature you flagged?
3. If yes, **flip the remote-config kill switch** — existing v2.3 users stop hitting the crash path without any update.
4. If it's not flaggable, **prepare a hotfix** (v2.3.1) and, once verified, either resume rollout on the fixed build or push it. For a truly unusable build, raise `minSupportedVersion` so v2.3 users get a **forced-update** prompt.
5. Note that Play lets you resume the *previous* good version's rollout to new installs while you fix forward.
6. **Postmortem**: why did staged rollout + monitoring not catch it earlier (was the halt threshold too loose)?

The senior point: prevention (staged rollout %, flags, alert thresholds) is the real answer; once a bad build is installed, all remedies are partial.

28. Design a CI/CD pipeline from PR to production store for a flavored Flutter app.

- **On PR**: `pub get` → `dart format --set-exit-if-changed` → `flutter analyze` (fail on infos too, via `analysis_options`) → `flutter test --coverage` → build the dev flavor (proves it compiles). Block merge on any failure; upload coverage.
- **On merge to** `main`: build `staging` flavor, distribute to testers (Firebase App Distribution / TestFlight internal) via Fastlane.
- **On git tag** `v*`: build `prod` AAB/IPA with `--obfuscate --split-debug-info`, **upload symbols/dSYMs** to Crashlytics/Sentry, then Fastlane `supply` / `deliver` to the store on an **internal/testing track first**, promote to a **staged production rollout** (start at 5–10%) gated on a manual approval.
- **Cross-cutting**: signing from encrypted secrets, build number from CI run number, caching pub/gradle for speed, and a required status check so nothing ships unreviewed.

29. How do you correlate a spike in crashes with a specific release, device, or config? Attach context at capture time: set the release/version and flavor, and log **breadcrumbs**, custom keys (feature-flag states, experiment bucket, user tier), and `setUserIdentifier` (anonymized) on your crash SDK. Then in the dashboard you filter/group crashes by app version, OS, device model, and

custom key. This is how you discover "only crashes on Android 12 + flag X on = the new payments path," which turns a vague spike into a one-line fix. Also wire non-fatal error logging (`recordError`) for handled exceptions so you see degradation before it becomes a hard crash.

30. Set up alerting and release-health gating — what fires, and what does it trigger? Define SLO-style thresholds per release and alert on breach, not on raw counts:

- Crash-free users drops below, say, 99.5% for the active rollout → page on-call and **auto/assisted halt** the rollout.
- New "velocity"/regression alert (a fresh crash signature spiking) → Slack/PagerDuty.
- ANR rate or slow-frame rate above baseline → warning.
- p99 latency or API error rate regression from Performance Monitoring → warning.

Route via Crashlytics velocity alerts / Sentry alert rules → Slack + PagerDuty. The gate is the important part: an alert that doesn't stop the rollout just tells you calmly that you shipped a fire.

31. Play App Signing vs uploading your own signed AAB — what actually changes? With **Play App Signing**, Google holds the *app signing key* and re-signs the per-device APKs; you sign uploads with a separate **upload key**. Benefits: if you lose the upload key you can reset it (you can't reset a legacy standalone signing key — that ends the app), the app signing key is stored in Google's KMS, and it's required for AAB. The trade-off is Google technically holds the key, and the delivered APK's signature differs from your local build's — matters for anything that pins the signing certificate (e.g. some SDKs, API key restrictions). You register the app signing SHA in those consoles, not your upload SHA.

32. Why is per-user crash-free rate a better rollout gate than total crash count, and where does it mislead? Total count scales with traffic — a big release naturally has more absolute crashes without being worse, so you can't threshold on it. Crash-free *users* normalizes by audience and reflects felt pain. Where it misleads: (a) it hides severity — 0.5% of users each crashing 50 times a session is worse than the number suggests, so also watch crash-free *sessions*; (b) low-traffic early rollout stages have tiny denominators, so 2 crashes look like a 5% regression — wait for statistical significance before halting; (c) it ignores non-crash breakage (frozen UI, failed checkout) that needs analytics/ANR/error signals too.

33. How do you keep obfuscation, symbol upload, and versioning correct across an automated multi-flavor release? Each build must produce and archive its own symbol artifacts keyed to its exact version+flavor+platform: run with `--obfuscate --split-debug-info=symbols/<flavor>/<version>`, then upload Dart symbols + Android mapping.txt + iOS dSYMs to the crash service tagged with the identical version/build number the store shows. Automate it in the same CI job that builds (never manually) so the symbols always match the shipped binary, and store the symbol files as build artifacts for the retention window in case you need to symbolicate an old crash. The failure mode to prevent: a rebuilt binary with regenerated symbols no longer matching the store's copy — so build once, upload the symbols from that same build.

34. A staged rollout looks healthy on your metrics but users complain the app is "broken." What's likely happening and how do you catch it? Crash-free rate only measures *crashes* — an app can be fully broken while never crashing: a failing network call handled with an empty state, a feature flag misconfigured, a silent auth failure, or frozen (not crashed) frames. Catch it by monitoring beyond crashes: non-fatal/handled-error logging, funnel/conversion analytics (did checkouts drop?), API error-rate and latency from performance monitoring, ANR/frozen-frame rate, and in-app feedback. The lesson: release health is crashes **plus** business metrics **plus** performance — gate on the union, because the worst outages are the quiet ones.

⚡ Rapid-fire (one-liners)

Q	A
Store upload format for Android?	AAB (`flutter build appbundle`)
Build mode for perf profiling?	profile (AOT + tracing, real device)
Read a `--dart-define` value?	`String.fromEnvironment('KEY')`
One flag for many defines?	`--dart-define-from-file=env.json`
Are `--dart-define` values secret?	No — baked into the binary, extractable
`1.4.2+57` — what's `57` ?	build number (`versionCode`), must increase
iOS signing needs?	certificate + provisioning profile
Fastest incident lever on mobile?	remote-config kill switch
Can you un-install a bad release remotely?	No — halt rollout / kill switch / forced update
Headline rollout gate metric?	crash-free users %
What's an ANR?	Android main thread blocked too long
Flag to obfuscate a release?	`--obfuscate --split-debug-info=...`
Forgot to upload symbols → ?	unreadable/obfuscated crash traces
CI stage order?	format → analyze → test → build
Flutter-specialized CI SaaS?	Codemagic
Fastlane is…?	store-automation toolkit, not a CI host
Lets you deploy ≠ release?	feature flags (ship dark, toggle on)
iOS phased release length?	7 days, auto-stepping daily
Recover a lost Android signing key?	only with Play App Signing (reset upload key)

Follow-up drills

1. Design the full GitHub Actions workflow (jobs, secrets, caching) that takes a flavored Flutter app from PR to a staged Play production rollout, including symbol upload.
2. Your prod release is crashing 6% of users at 30% rollout, and the crash is *not* behind a feature flag. Walk through every option and their trade-offs.
3. Implement a forced-update + soft-update system: define the remote schema, the startup check, and the failure/offline behavior.
4. Optimize a 14-minute CI pipeline that runs on every PR — where do you cut time without losing safety?
5. Design release-health alerting for a payments app: thresholds, what auto-halts vs pages a human, and how you avoid false alarms in early low-traffic rollout stages.
6. You inherit an app with no monitoring. Prioritize what you instrument first (crash, ANR, analytics, performance, flags) and justify the order.
7. Explain how you'd manage signing for a team of 8 iOS developers plus CI so nobody's laptop is a single point of failure.

Behavioral & Scenario Questions — Interview Questions

> The non-code half of every Flutter/mobile interview: how you work, communicate, decide, and recover. For the seniority lens see 58 Senior Architect Notes and for process/craft see 60 Software Engineering Practices.

Behavioral rounds test whether a strong engineer is also a safe hire: can you own outcomes, collaborate under friction, and make defensible trade-offs. Answers are scored on *specificity and self-awareness*, not eloquence — so every story must be a real situation with a concrete result. Tiers below are by **experience level**, not difficulty: 🟢 common intro/experience questions, 🟡 situational/teamwork, 🔴 senior/leadership.

How to use the STAR method (read first)

STAR = **S**ituation → **T**ask → **A**ction → **R**esult. It forces a story arc instead of a vague opinion.

- **Situation** (1–2 sentences): set the scene — team size, product, the constraint. Keep it short; it's context, not the point.
- **Task**: what *you* specifically owned. Name your role explicitly so the interviewer can attribute the outcome to you.
- **Action** (the bulk, ~60%): the concrete steps *you* took and *why* you chose them. Use "I", not "we" — the interviewer is scoring you, not your team.
- **Result**: the measurable outcome (numbers > adjectives) plus what you learned or changed afterward.

Golden rules:

- **Quantify the Result.** "Cut cold-start jank" is weak; "cut jank frames from 18% to 3% on the mid-tier Android profile, measured in DevTools" is strong.
- **Pre-load 4–6 stories** that each flex a different muscle (a hard bug, a conflict, a failure, a leadership moment, an ambiguity call) — most questions map onto one of them.
- **Prefer recent, mobile-flavored stories.** A Flutter-specific example (a jank fix, a platform-channel decision, a Bloc-vs-Riverpod call) signals domain depth a generic story can't.
- **End every failure story on the learning + the change you made** so it never landed the same way twice.

🟢 Common (intro & experience)

1. "Tell me about yourself." *Assessing:* can you frame your own narrative concisely and steer it toward the role. This is a 60–90 second pitch, not a résumé readthrough. *Outline (present → past → future):*

- **Present:** "I'm a Flutter engineer focused on production mobile apps — currently owning the {payments / offline-sync} area of a {X}-user app."
- **Past:** one or two proof points that map to this job's needs — "shipped the migration from Provider to Riverpod across 40+ screens," "took app-startup TTI from 3.1s to 1.4s."
- **Future:** why *this* role now — "I want to go deeper on {scale / platform channels / team leadership}, which is exactly what this team does." *Red flag:* reciting a chronological life story, or going 4+ minutes. Practice it to time.

2. "Walk me through a project you're proud of." *Assessing:* depth of ownership and whether you understand impact, not just tasks. *STAR outline:*

- **S:** "Our app's checkout was losing users on flaky networks."
- **T:** "I owned making checkout resilient offline." (your scope)
- **A:** "I designed an offline-first queue with a local Drift DB, idempotent retry keys, and optimistic UI; I chose a command-queue pattern over naive caching because payments can't be replayed blindly."
- **R:** "Checkout completion on poor connections rose ~22%; the pattern became the team's template for other mutations." Then the learning. *Red flag:* can't explain *why* you made the technical choices, or claims sole credit for a team effort.

3. "Tell me about a difficult bug you fixed." *Assessing:* debugging methodology and rigor under pressure — do you guess or do you isolate. *STAR outline:*

- **S:** "Intermittent crash on Android only, no stack trace in Crashlytics beyond a native frame."
- **T:** "I had to root-cause a heisenbug we couldn't reproduce locally."
- **A:** "I formed hypotheses and eliminated them methodically — added breadcrumb logging, correlated it to a specific device/OS band, found a platform-channel call returning on a background isolate touching a plugin that required the platform thread." Emphasize the *method* (bisect, instrument, reproduce, fix, add a regression test).
- **R:** "Crash-free sessions went from 99.1% to 99.9%; I added a lint/CI check so the same misuse can't ship again." *Red flag:* "I just kept trying things until it worked" — shows no systematic approach.

4. "Why do you want to work here?" *Assessing:* genuine research and fit — is this a scattershot application. *Outline:* tie a specific thing about *their* product/tech/scale to your goals. "You're a Flutter-first team shipping to both stores from one codebase at millions of installs — I want to work on cross-platform consistency at that scale." Reference something concrete (their app, a talk, their tech blog). *Red flag:* generic answers that fit any company ("great culture, growth"). Interviewers hear these all day.

5. "What's your biggest strength / weakness?" *Assessing:* self-awareness — the weakness is the real question. *Outline:* pick a **real** weakness plus the concrete system you built to manage it. "I used to over-engineer abstractions early; now I default to the simplest thing that works and

only generalize on the second use — I even flag it in my own PRs." A strength should be backed by a one-line example, not just an adjective. *Red flag:* the humblebrag weakness ("I work too hard," "I'm a perfectionist"). It reads as evasive.

6. "Why are you leaving your current job / looking to move?" *Assessing:* professionalism and motivation — do you speak ill of employers. *Outline:* frame forward, not away. "I've learned a lot, but the mobile team is small and I want to work on {larger scale / a domain they don't have}." Keep it about growth and opportunity. *Red flag:* trashing your manager, team, or codebase. It predicts how you'll talk about *them* next.

7. "Tell me about a time you had to learn a new technology quickly." *Assessing:* learning agility — critical because the mobile stack churns fast. *STAR outline:*

- **S:** "We adopted Riverpod mid-project and I'd only used Provider/Bloc."
- **T:** "I had two weeks before I'd be reviewing others' Riverpod PRs."
- **A:** "I read the official docs end to end, rebuilt one real feature as a spike, and wrote a short team cheat-sheet on our conventions (when to use `ref.watch` vs `ref.read`, provider lifecycles)." Show a *repeatable* learning process.
- **R:** "Shipped the feature on time and the cheat-sheet cut onboarding questions for the next two hires." *Red flag:* framing learning as painful/reluctant, or no evidence you can self-serve from docs.

8. "How do you keep your Flutter skills current?" *Assessing:* intrinsic motivation and awareness of the ecosystem's pace. *Outline:* name concrete channels — release notes for each stable Flutter version, pub.dev trends, a side project, reading source of packages you depend on. Mention a *specific* recent change you tracked (e.g. impeller becoming default, a Dart language feature). Specificity proves it's real. *Red flag:* vague "I read blogs" with nothing concrete to point to.

9. "Describe your typical development workflow on a feature." *Assessing:* engineering discipline — see 60 Software Engineering Practices. *Outline:* clarify requirements → break into small PRs → write tests alongside (unit for logic, widget for UI) → self-review the diff → open a tight PR with context → address review → verify on both platforms before merge. The point is you have a *repeatable* process, not heroics. *Red flag:* "I just start coding" — no design, no tests, giant PRs.

10. "Tell me about a time you received tough feedback." *Assessing:* coachability and ego management. *STAR outline:*

- **S:** "A senior said my PRs were too large and slow to review."
- **T/A:** "I took it as valid, switched to stacked/small PRs, and asked them to check if it helped."
- **R:** "Review turnaround dropped and I now default to <400-line PRs." Show you acted on it and it stuck. *Red flag:* being defensive, or an example where you were "given feedback" but disagreed and did nothing.

🟡 Situational (teamwork & judgment)

11. "Tell me about a disagreement with a teammate." *Assessing:* can you disagree technically without making it personal, and commit once decided. *STAR outline:*

- **S:** "A teammate wanted Bloc for a simple settings screen; I thought it was overkill vs. a plain `ChangeNotifier`."
- **A:** "I asked what problem Bloc was solving here, laid out the trade-off (boilerplate vs. testability), and proposed we decide by our actual criterion — testability need and team consistency. We agreed to match the codebase's existing pattern."
- **R:** "We shipped without churn, and I disagreed-and-committed cleanly." Emphasize focusing on *the problem*, shared criteria, and reversible-vs-irreversible framing. *Red flag:* "I was right and they were wrong," or you won by seniority/volume rather than reasoning.

12. "Tell me about a time you missed a deadline." *Assessing:* accountability and communication under slippage. *STAR outline:*

- **S:** "A platform-channel integration for a hardware SDK was more complex than scoped."
- **A:** "As soon as I saw the slip coming, I flagged it *early* to the PM with options — cut scope for v1, or push the date two days. I didn't sit on it hoping to catch up."
- **R:** "We shipped a reduced v1 on time and the full version the next sprint." The signal is *early, honest communication + options*, not that you never miss. *Red flag:* the miss was a surprise to stakeholders, or you blame the estimate/others without owning the communication gap.

13. "Describe a code review conflict — as author or reviewer." *Assessing:* how you handle friction in the most frequent collaboration surface. *STAR outline:*

- **A (as reviewer):** "I distinguish blocking issues (a memory leak from an undisposed `StreamController`) from preferences (naming), label nits as nits, and explain the *why* with a link, not just an order."
- **A (as author):** "When I disagreed I moved it to a quick call instead of a comment war, and captured the decision in the PR for the next person."
- **R:** "We merged without resentment and codified the rule so it wasn't re-litigated." *Red flag:* treating every comment as blocking, or ego battles in the thread. See 60 Software Engineering Practices.

14. "How do you handle ambiguous or incomplete requirements?" *Assessing:* do you build the wrong thing confidently, or de-risk before coding. *STAR outline:*

- **S:** "A ticket said 'add offline support' with no detail on conflict handling."
- **A:** "I listed the open questions (which entities, last-write-wins vs. merge, what the user sees when syncing), took them to product with a recommendation, and built a thin prototype of the riskiest path first to make the ambiguity concrete."

- **R:** "We caught a conflict-resolution requirement before it was expensive to change." Show you reduce ambiguity with questions + a spike, not assumptions. *Red flag:* "I just build what I think is best and adjust later" — expensive rework, and it signals you skip stakeholder alignment.

15. "Tell me about a time you had to balance speed vs. quality." *Assessing:* pragmatism and conscious trade-off — not dogma in either direction. *STAR outline:*

- **S:** "A launch-critical feature vs. a hard marketing date."
- **A:** "I shipped a scoped, well-tested happy path and *explicitly logged* the deferred edge cases as tracked tech-debt tickets with owners — a deliberate, visible trade-off, not a silent shortcut."
- **R:** "Hit the date, paid the debt down the next sprint before it compounded." *Red flag:* absolutism ("I never cut corners" or "ship it, fix later always"), or cutting corners *silently*.

16. "Describe a time you had to give difficult feedback to a peer." *Assessing:* directness with empathy — a leadership precursor. *STAR outline:* private not public → specific behavior + impact ("the last three PRs had no tests, so QA caught regressions late") → ask for their view → agree on a concrete change → follow up. Result: relationship intact, behavior changed. *Red flag:* avoiding the conversation entirely, or being blunt without care/context.

17. "Tell me about a time you disagreed with a product/design decision." *Assessing:* can you push back constructively across functions and still deliver. *STAR outline:* "Design wanted an animation I knew would jank on low-end Androids. I didn't just say no — I built a quick profiled demo showing dropped frames on a target device, then proposed an alternative that kept the intent at 60fps." Result: shipped a version that satisfied design *and* performance. Evidence beats opinion. *Red flag:* "I just did what I was told" (no engineering voice) *or* "I refused" (not a team player).

18. "Tell me about working with a difficult stakeholder or teammate." *Assessing:* emotional maturity and whether you seek to understand. *STAR outline:* identify their underlying concern (a "difficult" PM often just fears a missed date) → adjust *your* communication (async written updates for someone who felt out of the loop) → find common ground. Result: relationship improved because you changed something you controlled. *Red flag:* the whole story is about how unreasonable the other person was, with no self-reflection.

19. "A teammate is blocked on you and getting frustrated — what do you do?" *Assessing:* responsiveness and prioritization of team throughput over personal focus. *Outline:* unblock small/cheap requests fast even if it interrupts you; for larger ones, give a clear ETA and a stopgap (a stub interface, a mock, a branch) so they can proceed in parallel. The signal is that you treat *team velocity* as your job, not just your own tickets. *Red flag:* "they should wait until I finish my task" — local optimization at the team's expense.

20. "Tell me about a time you made a mistake in production." *Assessing:* ownership and blameless response — do you hide or surface failures. *STAR outline:*

- **S:** "I shipped a release where a null cast crashed on a locale I didn't test."
- **A:** "I owned it immediately, rolled back / hotfixed via a feature flag, wrote a blameless postmortem, and added a test + a CI matrix for locales."
- **R:** "No repeat; the locale matrix caught two later bugs pre-release." *Red flag:* minimizing it, blaming QA, or no systemic fix so it could recur.

🔴 Senior & leadership

21. "Tell me about a time you mentored someone." *Assessing:* whether you scale through others, not just your own output. *STAR outline:*

- **S:** "A junior kept struggling with async/state bugs (setState after dispose, unawaited futures)."
- **A:** "I paired instead of fixing it for them, taught the mental model of the widget lifecycle and `mounted` checks, and set up a lightweight review rhythm. I optimized for *them* learning to fish."
- **R:** "Within a quarter they were reviewing others' async code; that's leverage." See 58 Senior Architect Notes. *Red flag:* "mentoring" that's actually just doing the work for them, or measuring your success by your commits instead of their growth.

22. "Describe a significant technical decision you drove (e.g., a state-management or architecture choice)." *Assessing:* structured decision-making, buy-in, and owning the outcome — good and bad. *STAR outline:*

- **S:** "Team was split on Bloc vs. Riverpod for a new app."
- **A:** "I ran a lightweight trade-off doc with explicit criteria (testability, boilerplate, team familiarity, ecosystem), built parallel spikes of one real feature in each, and made a *recommendation with a rationale*, not a decree. I looped in the seniors and the people who'd maintain it."
- **R:** "We chose Riverpod; I documented the decision + reasons (an ADR) so it wasn't re-litigated. Six months in I revisited whether it held up." Show a repeatable, evidence-based, buy-in-seeking process. *Red flag:* deciding by ego/hype/novelty, no criteria, or no follow-through to check if it was right.

23. "Tell me about a time you had to say no to a PM or stakeholder." *Assessing:* backbone plus the ability to say no *with* a path, not a wall. *STAR outline:* "PM wanted a feature in a sprint that required rushing our auth-token refresh logic — a security risk. I said no to *that timeline*, explained the risk in their terms (a token leak = user trust + store review risk), and offered a safe phased plan." Result: they chose the safe plan because you gave them a business-framed trade-off, not just resistance. *Red flag:* either can't ever say no (pushover) or says no without offering an alternative (blocker). Also: framing risk in jargon they can't act on.

24. "Walk me through how you owned an incident or outage." *Assessing:* calm under fire, coordination, and turning failure into system improvement. *STAR outline:*

- **S:** "A backend contract change caused a deserialization crash on app launch — users hard-stuck at splash."
- **A:** "I stabilized first (feature-flag kill-switch / forced-update gate), communicated status to stakeholders on a cadence, *then* root-caused. I ran the incident, didn't just fix code."
- **R:** "Restored within the hour; the blameless postmortem produced a contract-test in CI and a schema-version guard so a breaking change can't crash launch again." See 60 Software Engineering Practices. *Red flag:* heroics with no communication, or fixing the symptom with no prevention. Mitigate-then-diagnose order matters.

25. "How do you decide when to take on vs. pay down tech debt?" *Assessing:* treating tech debt as a managed trade-off, not a moral failing. *STAR outline:* "I treat debt like a loan — fine if it's deliberate, documented, and the interest (change-cost, bug-rate) is tracked. I quantify it: this legacy navigation stack cost us N bugs and slowed every feature by ~X. I bundle paydown into adjacent feature work rather than asking for a standalone 'refactor sprint' that PMs will never fund." Result: measurable drop in a metric (crash rate, cycle time). *Red flag:* "rewrite everything" absolutism, or never addressing debt because "there's no time" — both signal poor judgment. See 58 Senior Architect Notes.

26. "Tell me about a time you influenced without authority." *Assessing:* can you drive change across a team when you can't simply mandate it. *STAR outline:* "I wanted the team on a shared CI gate (format, analyze, test, golden diffs). I couldn't mandate it, so I built it, proved it on my own repo, showed the bugs it caught, and made adoption one-line easy. Peers opted in because it made *their* life easier." Result: org-wide adoption via demonstrated value. *Red flag:* influence via politics or pressure rather than evidence and making the right thing the easy thing.

27. "Describe leading a project that failed or was cancelled." *Assessing:* resilience, honesty, and extracting value from failure at scale. *STAR outline:* own the call honestly (a Flutter-web bet that didn't meet perf targets on the target hardware), what signals you missed, when you should have killed it sooner, and what the org kept (learnings, reusable components, a clearer bar for "is web viable here"). Result is the *learning* and any salvage. *Red flag:* no failure to share (unbelievable / low ownership), or blaming everyone but yourself.

28. "How do you handle a senior engineer whose code you think is wrong?" *Assessing:* courage + humility balance, and technical communication upward. *Outline:* assume you might be missing context → ask questions before asserting ("what happens to this `StreamSubscription` on rebuild — do we leak it?") → bring data/repro if you're confident → escalate on facts, not rank. Result: either you learn the context you missed, or the issue gets fixed — both are wins. *Red flag:* staying silent out of deference (real bug ships) or steamrolling with certainty (arrogant, no humility).

29. "How do you set technical direction or standards for a team?" *Assessing:* systems thinking — scaling quality through process, not per-PR policing. *Outline:* codify the important stuff (lint rules, architecture boundaries, ADRs for big calls), automate enforcement in CI so it's not about

personalities, and reserve human review for judgment. Lead by writing the first examples. Keep standards *living* — revisit when they cause more friction than value. *Red flag:* standards-by-decree with no buy-in, or gatekeeping everything manually (doesn't scale, becomes a bottleneck).

30. "Tell me about balancing individual delivery with team responsibilities as a senior." *Assessing:* understanding that senior impact is measured in team output, not personal LOC. *Outline:* deliberately trade some of your own throughput for reviews, unblocking, mentoring, and design docs when that multiplies the team; protect *some* focus time so you keep technical credibility. Name the trade-off explicitly. Result framed as team velocity/quality, not just your tickets shipped. *Red flag:* hoarding the "fun" work, or being so heads-down you never lift the team.

⚡ Rapid-fire (one-liners)

Prompt	One-line approach
Greatest professional achievement?	One story, quantified result, your specific role.
Where do you see yourself in 3–5 years?	Growth aligned to *this* role's trajectory; ambitious but plausible.
How do you prioritize competing tasks?	Impact × urgency; confirm priorities with the stakeholder, don't guess.
How do you handle stress / crunch?	Concrete tactics (scope-cut, communicate early) + sustainable boundaries.
Prefer working alone or in a team?	Both, with an example of each; adapt to what the work needs.
How do you handle being wrong?	Acknowledge fast, fix, extract the lesson — an example, not a platitude.
What motivates you?	Something specific and intrinsic (solving user pain, craft), not "money."
Describe your ideal manager.	Autonomy + context + candid feedback; avoid anything that trashes past bosses.
A time you went above and beyond?	Real story with impact — not just "worked late."
How do you onboard to an unfamiliar codebase?	Run it, trace one flow end-to-end, read tests, ask targeted questions.
Do you have questions for us?	Always yes — thoughtful ones about the team's real challenges.
How do you handle context-switching between platforms?	Checklists, verify on both iOS/Android, automate the repetitive parts.
A time you improved a process?	Named the friction, proposed a fix, measured the before/after.
How do you estimate work?	Break down, add uncertainty buffer, communicate ranges + assumptions.

Follow-up drills (self-practice — no answers)

1. **Escalating conflict:** "You disagreed and committed to a state-management choice. Six months later it's clearly hurting the team. How do you reopen that decision without

undermining the person who won the original call?"

2. **Incident ownership:** "Walk me through an outage where the root cause was *your* code, the fix was non-obvious, and stakeholders were watching live. Separate what you did to mitigate from what you did to diagnose, and to communicate."
3. **Saying no upward:** "A director personally wants a feature you believe is technically unsound for the app's scale. Your PM defers to them. How do you handle it?"
4. **Influence at scale:** "Convince a skeptical team of five to adopt a new architecture pattern (or CI gate) when you have no authority over any of them and two are more senior than you."
5. **Tech-debt tradeoff:** "You've inherited a codebase with no tests and a 4% crash rate under launch pressure. Make the case to product for the investment — in *their* language — and lay out your first 90 days."
6. **Mentoring under-performance:** "A mid-level engineer on your team is plateauing and morale is dipping. Describe how you'd diagnose, intervene, and know whether it's working."

See also: 58 Senior Architect Notes · 60 Software Engineering Practices

Rapid-Fire & Mixed Mock Rounds — Interview Questions

> Last-minute revision + timed mock practice spanning **every** topic in this question bank. For depth on any line, jump to the topic file or the handbook: 55 Flutter Interview Preparation and 56 Machine Coding Rounds.

This is the file you skim the night before and drill the morning of. It tests *recall speed* (can you fire the fact instantly?), *articulation* (can you explain it in 60 seconds?), and *composure under trick questions*. Use it after you've read the individual topic files — this is the compression layer, not the teaching layer.

How to use this

1. **Rapid-fire (revision mode):** cover the answer column, read the question, say the answer aloud, then reveal. If you stall, open the linked topic file. Do a full pass the night before.
2. **60-second drills (articulation mode):** set a phone timer to 60s per prompt. Speak — don't type. A strong answer hits the bullet points *and* stays inside the minute.
3. **Gotchas (composure mode):** these are the questions that separate memorizers from engineers. Predict the output / spot the bug before reading the answer.
4. **Mock sets (simulation mode):** pick the round matching your level, start a timer, and answer out loud with no notes. Record yourself. Grade on correctness *and* delivery.

Topic files for depth: 01_dart_core.md · 02_dart_advanced_async.md · 03_oop_solid_patterns.md · 04_flutter_internals.md · 05_widgets_layout_lifecycle.md · 06_rendering_performance.md · 07_state_management.md · 08_navigation_routing.md · 09_networking_data_storage.md · 10_dependency_injection.md · 11_animations_custom_ui.md · 12_architecture.md · 13_testing.md · 14_platform_native_device.md

---

⚡ Rapid-fire by topic

Dart

Q	A
`final` vs `const` ?	`final` = set once at runtime; `const` = compile-time, deeply immutable
Top type in Dart?	`Object?`
`dynamic` vs `Object?` ?	Both hold anything; `dynamic` disables static checks
What is a `sealed` class for?	Exhaustive `switch` — compiler forces all subtypes handled
Access 2nd positional record field?	`.$2`
Override alongside `==` ?	`hashCode`
What does `late` defer?	Initialization; lazy on first read
Null-aware spread?	`...?`
`int` on web is really?	An IEEE-754 double
`identical(const[1], const[1])` ?	`true` — canonicalized

Async & Isolates

Q	A
Single-value vs multi-value async type?	`Future` vs `Stream`
Microtask vs event queue priority?	Microtasks drain fully before the next event
What schedules a microtask?	`scheduleMicrotask` , `Future.value().then` , awaited completions
Do isolates share memory?	No — separate heaps, message-passing only
Cheapest way to run one function off-thread?	`Isolate.run` (Dart 2.19+) / `compute`
`async*` produces?	A `Stream` (generator)
Broadcast vs single-subscription stream?	Broadcast allows many listeners; single allows one
`await for` does what?	Iterates a stream sequentially
Does `async` move work off the UI thread?	No — same isolate, just non-blocking scheduling
Unhandled Future error goes where?	Zone error handler / `runZonedGuarded`

Flutter internals & the 3 trees

Q	A
The three trees?	Widget, Element, RenderObject
What is a Widget?	Immutable config / blueprint
What is an Element?	Mutable instance holding state + tree position
What is a BuildContext?	A handle to the Element's location in the tree
What is a RenderObject for?	Layout, painting, hit-testing
Why are widgets cheap to rebuild?	They're immutable configs; Elements are reused
What decides Element reuse?	`runtimeType` + `key` match at same position
Who holds `State`?	The `StatefulElement`
What is a `RenderObjectWidget`?	Widget that creates/updates a RenderObject
`InheritedWidget` gives you?	O(1) dependency lookup + targeted rebuilds

Widgets & lifecycle

Q	A
StatelessWidget lifecycle?	`build` only (recreated on config change)
Key `State` lifecycle order?	`initState` → `didChangeDependencies` → `build` → `dispose`
Where to subscribe to an `InheritedWidget`?	`didChangeDependencies`
When does `didUpdateWidget` fire?	Parent rebuilds with a new widget of same type
Safe place for one-time async init?	`initState` (guard with `mounted` before setState)
What must `dispose` clean up?	Controllers, streams, listeners, tickers
`const` constructor benefit?	Widget instance canonicalized → skipped rebuild
Why use a `Key`?	Preserve state/identity across reorders & type-stable swaps
`GlobalKey` cost?	Expensive; enables cross-tree access & state retention
`mounted` guards against?	`setState` after dispose

Rendering & performance

Q	A
Frame budget at 60fps?	~16.6 ms (build+layout+paint on UI thread)
Layout algorithm?	Constraints down, sizes up, parent sets position
`BoxConstraints` carries?	min/max width & height
Cheapest way to skip subtree rebuild?	`const` widgets
`RepaintBoundary` does?	Isolates a layer so repaints don't propagate
Jank = ?	A frame that misses the 16.6 ms budget
Tool to find rebuilds?	DevTools "Track widget builds" / rebuild counts
`ListView` vs `ListView.builder` ?	Builder lazily builds only visible items
`saveLayer` is expensive because?	Allocates offscreen buffer (opacity/clip layers)
Shader jank fix?	Shader warm-up / Impeller (precompiled)

State management

Q	A
Cheapest built-in shared state?	`InheritedWidget` / `InheritedNotifier`
Provider is a wrapper over?	`InheritedWidget`
BLoC core primitive?	Streams (events in → states out)
Cubit vs BLoC?	Cubit = methods emit; BLoC = events mapped to states
Riverpod advantage over Provider?	Compile-safe, no `BuildContext` , testable, no `ProviderNotFound`
`context.watch` vs `read` ?	`watch` rebuilds on change; `read` one-shot, no listen
`select` does?	Rebuilds only when a chosen slice changes
Golden rule for emitted state?	Emit new immutable instances (identity change)
Where does business logic live?	Outside widgets (notifier/bloc/controller)
`ChangeNotifier` notifies via?	`notifyListeners()`

Navigation

Q	A
Imperative API?	`Navigator.push` / `pop`
Declarative API?	`Router` + `Navigator(pages: [...])` (Nav 2.0)
Return a result from a route?	`await Navigator.push(...)` → `pop(result)`
Deep links need?	Declarative routing / `onGenerateRoute` mapping
`pushReplacement` does?	Swaps current route (no back to it)
`pushAndRemoveUntil` for?	Clearing stack (e.g. after login)
Popular declarative package?	`go_router`
Named vs `onGenerateRoute`?	Named = static map; generate = dynamic + args parsing
Nested navigators for?	Tab-local stacks / shell routes
Guard a route (auth)?	Redirect in router / check in `onGenerateRoute`

Networking & data

Q	A
Common HTTP clients?	`http`, `dio`
Why `dio` over `http`?	Interceptors, cancel tokens, timeouts, form-data
Parse JSON off the UI thread?	`compute(jsonDecode+map)` for big payloads
Retry/refresh-token best place?	An interceptor
Key-value local store?	`shared_preferences`
Structured local DB options?	`sqflite`, `drift`, `Isar`, `Hive`
Secure token storage?	`flutter_secure_storage` (Keychain/Keystore)
Offline-first pattern?	Cache-then-network / single source of truth = DB
Serialize models without boilerplate?	`json_serializable` / `freezed` codegen
Cancel an in-flight request?	`CancelToken` (dio)

Dependency injection

Q	A
What problem does DI solve?	Decouples construction from use → testable, swappable
Service locator package?	`get_it`
Codegen DI package?	`injectable`
`registerSingleton` vs `registerLazySingleton`?	Eager vs created on first resolve
`registerFactory` gives?	A new instance every resolve
Riverpod as DI?	Providers *are* the container (overridable)
Swap a real dep for a fake in tests?	Override registration / provider override
Constructor injection benefit?	Explicit deps, no hidden globals
Scoped DI use case?	Per-feature / per-route lifetime
Anti-pattern to avoid?	Reaching into a global singleton from widgets

Architecture

Q	A
Clean Architecture layers?	Presentation → Domain → Data
Dependency rule direction?	Inward — outer depends on inner, never reverse
What is a UseCase?	A single application action in the domain layer
Repository pattern hides?	Data sources behind a domain interface
MVVM binds view to?	A ViewModel exposing observable state
Where do entities live?	Domain (framework-independent)
Feature-first vs layer-first?	Group by feature vs by technical layer
DTO vs Entity?	DTO = data/transport shape; Entity = domain model
Why interfaces at layer boundaries?	Dependency inversion → testable, swappable
Modular architecture wins?	Build isolation, ownership, faster CI

Testing

Q	A
Three test tiers?	Unit, widget, integration
Widget test entry point?	`tester.pumpWidget`
Advance frames in a widget test?	`pump` / `pumpAndSettle`
Find widgets?	`find.byType/byKey/text`
Mock package?	`mockito` / `mocktail`
Test async without real delays?	`fakeAsync` / `tester.pump(duration)`
Golden test verifies?	Pixel-level rendering vs a reference image
Fake time in unit tests?	`fakeAsync` / injected `Clock`
Test coverage command?	`flutter test --coverage`
Why avoid `pumpAndSettle` with infinite anims?	It never settles → timeout

Platform & native

Q	A
Talk to native code via?	Platform channels
`MethodChannel` for?	Request/response native calls
`EventChannel` for?	Streaming native events (sensors, etc.)
New lower-overhead interop?	Pigeon (type-safe) / FFI
FFI is for?	Calling C/C++ directly, no channel hop
Channel calls are?	Async, message-passing, serialized
Detect platform at runtime?	`Platform.isAndroid` / `defaultTargetPlatform`
Adaptive vs responsive?	Adaptive = per-platform look; responsive = per-size layout
Run native UI inside Flutter?	`PlatformView`
Where does channel native code live?	Android (Kotlin/Java) & iOS (Swift/ObjC) host

---

🎯 "Explain in 60 seconds"

Each prompt: give a crisp elevator explanation. A strong answer hits the bullets *and* stays under a minute.

1. **How does Flutter render a frame?**
    - Build phase: rebuild dirty widgets → reconcile Element tree.

- Layout: constraints go down, sizes come up.
- Paint → composite layers → GPU rasterizes.
- All within the ~16.6 ms UI-thread budget; miss it = jank.

2. **Why is the Element tree the "middle man"?**
   - Widgets are immutable and thrown away every rebuild.
   - RenderObjects are expensive to create.
   - Elements persist, diff old vs new widget, and mutate/reuse RenderObjects.
   - This is what makes rebuilds cheap.
3. **`setState` vs a state-management solution — when to reach past `setState`?**
   - `setState` is fine for local, ephemeral, widget-owned state.
   - Reach past it when state is shared across screens, outlives the widget, or holds business logic.
   - Options scale: InheritedWidget → Provider → Riverpod/BLoC.
   - Goal: keep logic testable and out of `build`.
4. **What is sound null safety and why does "sound" matter?**
   - Types are non-nullable by default; `String?` opts into null.
   - "Sound" = guaranteed end-to-end, so the runtime trusts non-null types.
   - No hidden null checks compiled in → faster + safer.
   - Eliminates the classic `NoSuchMethodError: null`.
5. **Isolates vs async/await.**
   - `async` / `await` is concurrency on *one* thread (non-blocking scheduling).
   - Isolates are true parallelism — separate heaps, no shared memory.
   - Use async for I/O waits; use isolates for CPU-heavy work (parsing, crypto).
   - Communicate via message passing (`SendPort`) / `Isolate.run`.
6. **Provider vs Riverpod — pick one and defend it.**
   - Provider: simple, `InheritedWidget`-based, needs `BuildContext`, can throw `ProviderNotFound`.
   - Riverpod: compile-safe, context-free, easily testable, override-friendly.
   - Riverpod fixes Provider's runtime-lookup pitfalls.
   - Trade-off: Riverpod's learning curve vs Provider's ubiquity.
7. **Clean Architecture in a Flutter app.**
   - Three layers: Presentation, Domain, Data.
   - Domain (entities + use cases) is pure Dart, framework-free.
   - Dependencies point inward; data layer implements domain interfaces.

- Payoff: testability, swappable data sources, feature isolation.

8. **How would you diagnose jank?**
   - Open DevTools performance/timeline; look for frames > 16.6 ms.
   - Separate UI-thread jank (build/layout) from raster-thread jank (paint/shaders).
   - Fix with `const`, `RepaintBoundary`, `ListView.builder`, offloading work.
   - Confirm by re-profiling on a *real device* in profile mode.
9. **The widget lifecycle of a StatefulWidget.**
   - `createState` → `initState` → `didChangeDependencies` → `build`.
   - Updates: `didUpdateWidget` (new config) / `didChangeDependencies` (inherited change).
   - Teardown: `deactivate` → `dispose`.
   - Rule: allocate in `initState`, release in `dispose`, guard with `mounted`.
10. **How do platform channels work?**
    - Dart and native agree on a channel name + codec.
    - `MethodChannel` sends an async, serialized message across the boundary.
    - Native handler runs and returns a result (or streams via `EventChannel`).
    - For hot paths, prefer Pigeon (type-safe) or FFI (no channel hop).

🧠 Tricky / gotcha questions

1. **Why do `const` constructors actually matter for performance?** `const` widgets are canonicalized to a single shared instance, so on rebuild the framework sees the *same* object (`identical`), short-circuits the diff, and skips rebuilding that subtree entirely. It's not micro-optimization — it prunes whole branches of the rebuild.
2. **Why is calling `setState` inside `build` catastrophic?** `build` runs during the build phase; `setState` marks the element dirty and schedules another build → infinite rebuild loop (and often a "setState called during build" assertion). State changes belong to event handlers or lifecycle callbacks, never `build`.
3. **You reorder items in a `ListView` and the wrong ones keep their state. Why?** Without keys, Elements match by position + type, so state (scroll, animation, form input) sticks to the *slot*, not the item. Give each item a stable `ValueKey(id)` so the framework matches by identity across reorders.
4. **`final List<int> nums = [...]; nums.add(4);` — legal?** Yes. `final` freezes the *binding*, not the object. `nums = [...]` would fail; `nums.add(4)` mutates the same list and is fine. Use `const [...]` or `List.unmodifiable` for true immutability.

5. `==` **vs** `identical` **— when do they diverge?** `identical(a, b)` is reference equality; `a == b` is value equality (whatever `==` is overridden to be). Two different objects can be `==` but not `identical`. Two `const` literals with equal content *are* `identical` due to canonicalization.
6. **Predict the output:**

   ```
   print('A');
   Future(() => print('B'));            // event queue
   Future.microtask(() => print('C'));  // microtask queue
   print('D');
   // output: A D C B
   ```

   Sync code first (`A` , `D`), then the microtask queue drains fully (`C`) *before* any event-queue task (`B`).
7. `scheduleMicrotask` **vs** `Future(() => ...)` **— why does ordering flip intuition?** Microtasks always run before the next event-loop task, no matter when they were queued in the current turn. A `Future(...)` body goes to the event queue and yields to *all* pending microtasks first — so a later-scheduled microtask can run before an earlier-scheduled `Future`.
8. **Why can't you promote a nullable *field* with** `if (obj.field != null)` **?** A getter could return a different value on the second read (overridden getter, concurrent change), so the compiler can't prove non-null. Promote a local copy: `final f = obj.field; if (f != null) f.use();`.
9. **You** `await` **something after an** `if (!mounted) return;` **check and then call** `setState` **— still safe?** No. The `await` yields; the widget can be disposed *during* the gap. Re-check `mounted` **after** every `await` that precedes a `setState`, not just once at the top.
10. **Two objects are** `==` **but end up in a** `Set` **twice. What broke?** You overrode `==` but not `hashCode`. Hash-based collections bucket by `hashCode` first; unequal hashes never even compare with `==`. Always override both together (or use `Equatable` / `freezed`).

💉 Mock interview sets

Set a timer. Answer **out loud**, no notes. Record and grade on correctness *and* delivery. No answers here by design — self-check against the linked topic files.

Round 1 — Junior (45 min)

Emphasis: language fundamentals, core widgets, `setState`*, and clear articulation. Interviewers want correctness and evidence you understand what you type, not depth.*

1. Explain `var` vs `final` vs `const` with an example of when each is right.
2. What's the difference between `StatelessWidget` and `StatefulWidget` ?

3. Walk me through the lifecycle of a `StatefulWidget`.
4. What does `setState` do under the hood, and when is it the wrong tool?
5. `Row` / `Column` / `Expanded` — how does Flutter decide sizes here?
6. What is a `Future`, and what does `await` actually do?
7. How do you make and parse a simple HTTP GET request?
8. Why do we add a `key` to items in a list?
9. What's the difference between `ListView` and `ListView.builder`?
10. What is `BuildContext`, in one sentence?

Round 2 — Mid-level (60 min)

Emphasis: internals, state management trade-offs, async/streams, navigation, and testing. They probe why and expect you to compare options and justify a choice.

1. Explain the three trees and why the Element tree exists.
2. Compare Provider, Riverpod, and BLoC — which do you reach for and why?
3. Walk through how Flutter renders one frame and where jank comes from.
4. Streams: broadcast vs single-subscription, and where microtasks fit.
5. How do you return a result from a pushed route, and how does Navigator 2.0 differ?
6. Design the data layer for a screen that must work offline.
7. How do you structure and inject dependencies for testability?
8. Write (verbally) a widget test for a counter — what do you pump and assert?
9. When and how would you use `compute` / `Isolate.run`?
10. You see jank scrolling a list — how do you diagnose and fix it?

Round 3 — Senior (60 min)

Emphasis: architecture, scale, performance under pressure, and platform/native depth. They want system-level reasoning, trade-off ownership, and how you'd lead a codebase — depth beats breadth.

1. Design the architecture for a multi-feature app owned by 4 teams — layers, modules, boundaries.
2. Defend your state-management choice for a large app and its testing story.
3. How do you enforce the dependency rule and keep the domain framework-free at scale?
4. A production frame budget is blowing up only on low-end Android — how do you investigate end to end?
5. Design an offline-first sync engine with conflict resolution.
6. When do you choose FFI vs Pigeon vs `MethodChannel`, and what are the costs of each?
7. How do isolates change how you architect CPU-heavy features, and what are the messaging pitfalls?

8. Design a CI pipeline: what gets tested, gated, and how do you keep it fast?
9. How would you introduce modularization into a large monolithic Flutter codebase incrementally?
10. Walk me through a performance or architecture decision you'd defend to a skeptical staff engineer.

Revise topic-by-topic in the sibling Interview Practice files; go deep via 55 Flutter Interview Preparation and drill live-coding in 56 Machine Coding Rounds.