# 14 · Dependency Injection

Flutter Practice Notes

## Contents

# 14 · Dependency Injection

## Introduction

This module covers **how to do DI in Flutter apps** — the practical tooling that realizes the DI *pattern* from Module 05. It compares `get_it` / `injectable`, Provider/Riverpod as DI, and `InheritedWidget`-based DI, plus scopes/lifetimes and testing with injected fakes.

## Why this module exists

Every non-trivial app must wire repositories, services, clients, and view models to their consumers — testably and consistently. Choosing and using a DI approach determines testability, modularity, and how easily you swap implementations (real/fake/flavors). It's the connective tissue of clean architecture.

## Module map

| # | File | Topic | Level |
|---|---|---|---|
| 1 | 01_di_fundamentals.md | Composition root, DI vs service locator, in Flutter | 🔵 |
| 2 | 02_get_it_and_injectable.md | `get_it` service locator + `injectable` codegen | 🔵 |
| 3 | 03_provider_riverpod_as_di.md | Provider/Riverpod as DI containers | 🔵 |
| 4 | 04_scopes_and_lifetimes.md | Singleton/factory/lazy/scoped + disposal | 🔴 |
| 5 | 05_testing_with_di.md | Injecting fakes/mocks, overrides | 🔴 |

> **Cross-references:** DI pattern + DIP: 05 · dependency_injection, 04 · DIP. Providers as state: Module 11. Singletons: 05 · singleton. App entry/composition root: 06 · app_entry_point, 10 · embedder_and_startup. Testing: Module 49.

## Prerequisites

04 SOLID (DIP), 05 Design Patterns (DI/Singleton), 11 State Management.

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

- Set up a composition root and choose constructor injection vs service locator.
- Wire dependencies with `get_it` / `injectable`, Provider, or Riverpod.
- Manage lifetimes (singleton/factory/lazy/scoped) and disposal.
- Inject fakes/mocks to make code fully testable.

- Support build flavors (dev/prod/mock) via DI.

## Capstone

**Wired, testable app slice:** Wire a `Repository → UseCase → ViewModel` graph with a chosen DI approach, register real impls at a composition root, and run the same use case in tests with injected fakes — proving swap-ability and testability.

## Summary

DI in Flutter = choosing a mechanism ( `get_it` /Riverpod/Provider) to supply abstractions to consumers, wired at a composition root, with managed lifetimes and testable via fakes. The pattern (Module 05) is the *why*; this module is the *how*.

# DI Fundamentals (Composition Root, DI vs Service Locator)

> Dependency Injection supplies a class's dependencies from outside; in Flutter you wire the object graph at a **composition root** (app startup) and choose between **constructor injection** (explicit) and a **service locator** (global lookup) — usually a blend.

## Introduction

Recapping the pattern (05 · dependency_injection) in a Flutter context: what a composition root is, the DI-vs-service-locator tradeoff, and how the Flutter widget tree itself is a DI mechanism (`InheritedWidget`). This grounds the tool choices in the next files.

## Why this concept exists

Hard-wired `new`s make code untestable and rigid (DIP violation — 04). DI decouples "what a class needs" from "who provides it," enabling testing (inject fakes), swapping (flavors/providers), and a visible dependency graph. Flutter apps need a concrete strategy for this.

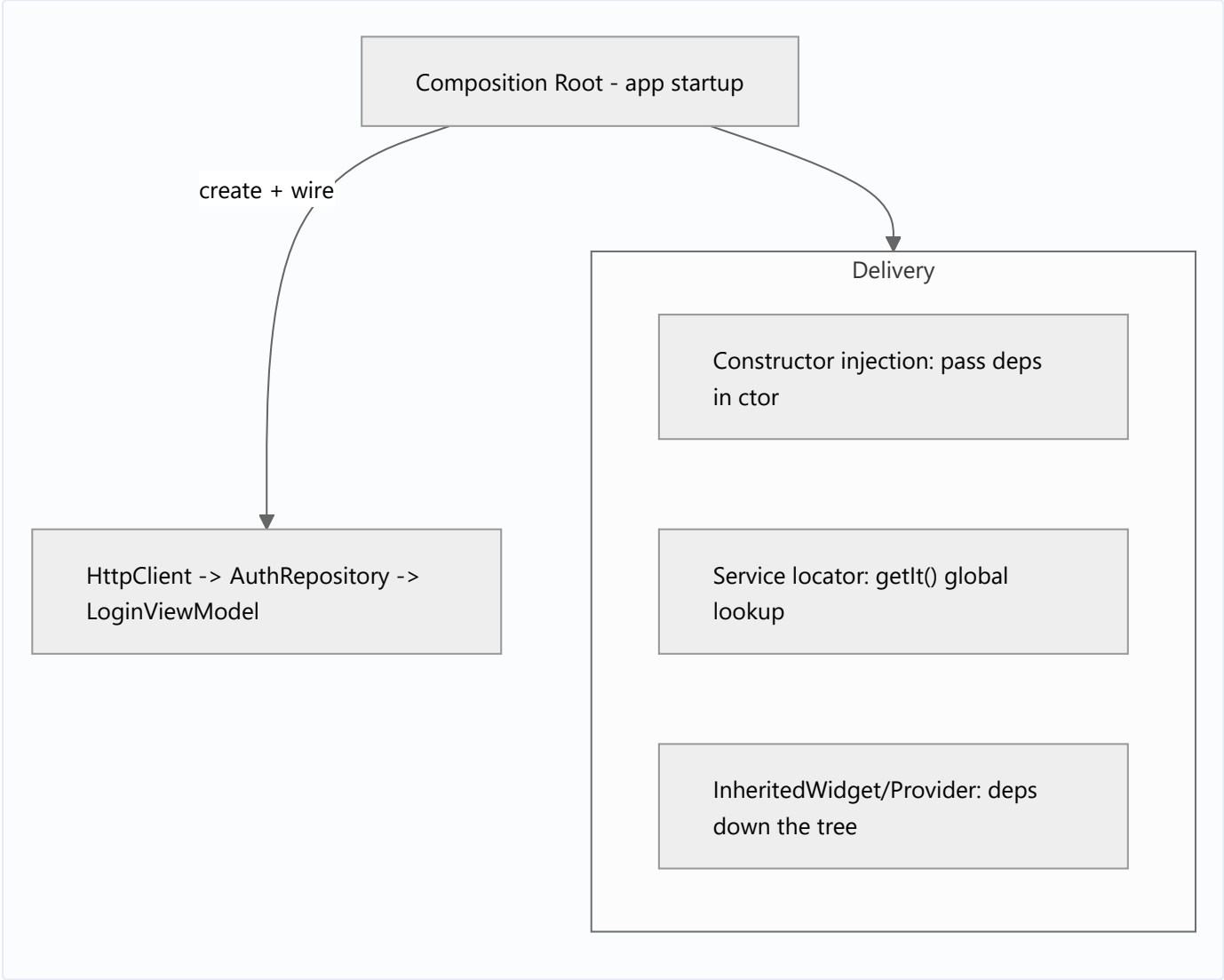
## Real-world analogy

A **restaurant kitchen**: ingredients (dependencies) are delivered to each station from a central pantry (composition root), not foraged by each cook (`new`). Constructor injection is **handing the cook exactly what they need**; a service locator is **a shared pantry cooks fetch from** — convenient, but you can't see from the recipe what a cook actually uses.

## Problem Statement

A `LoginViewModel` needs an `AuthRepository` which needs an `HttpClient`. Where do you create/wire these, how do consumers get them, and how do you swap real→fake in tests? You'll set up a composition root and pick injection styles.

## Internal Working




- **Composition root**: the single place (usually `main`/startup) where concrete implementations are created and wired to abstractions — the one spot allowed to know concretes (06 · app_entry_point).
- **Constructor injection** (preferred): dependencies passed to the constructor — explicit, immutable, testable; the graph is visible in signatures.
- **Service locator**: a global registry ( `get_it` ) where code calls `getIt<T>()` to fetch deps — convenient, less boilerplate, but hides dependencies (looks like a global) and errors at runtime.
- **Tree-based DI**: `InheritedWidget`/Provider/Riverpod supply deps to descendants via `context` / `ref` — natural for widget-scoped and reactive deps (Module 11).
- **Reality**: apps blend these — e.g., `get_it` for services + constructor injection into view models, or Riverpod providers for everything.

## Memory Representation

Container-registered singletons live for app lifetime; factories create per-request. Mind retained singletons (05 · singleton, 08 · dispose_and_leaks).

## Compiler Behavior

Constructor injection + abstractions give compile-time-safe substitution; service-locator lookups by type fail at runtime if unregistered (like Provider). Riverpod adds compile-time provider safety (11 · riverpod).

## Runtime Behavior

The composition root registers/creates; consumers receive (ctor) or fetch (locator/context). Tests substitute fakes at the root/override.

## Flutter Engine Behavior / Dart VM Behavior

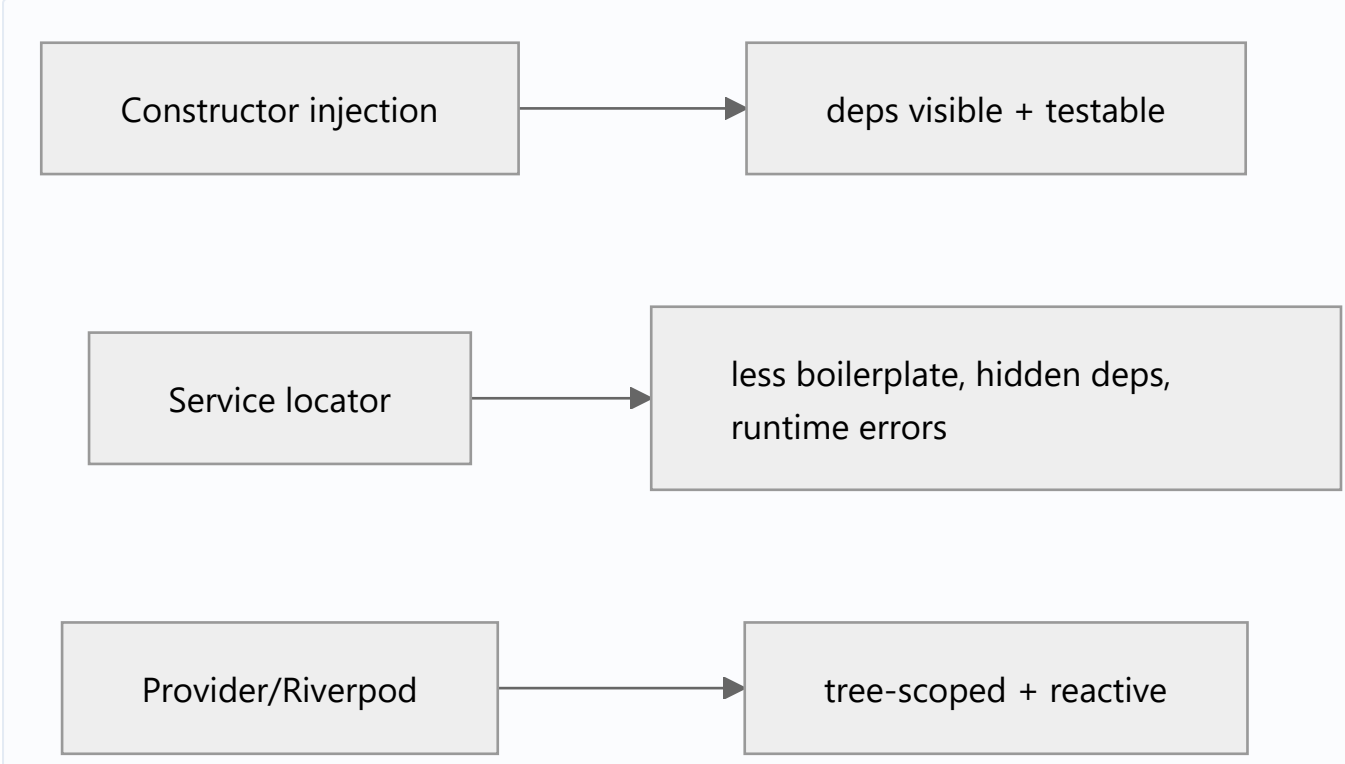
Not applicable.

## Examples

```
// Abstractions
abstract interface class HttpClient { Future<String> get(String url); }
abstract interface class AuthRepository { Future<bool> login(String u, String p); }

// Concrete impls
class RealHttpClient implements HttpClient {
  @override
  Future<String> get(String url) async => 'ok';
}
class HttpAuthRepository implements AuthRepository {
  final HttpClient client;
  HttpAuthRepository(this.client); // constructor injection
  @override
  Future<bool> login(String u, String p) async => (await client.get('/login')) == 'ok';
}

// Consumer receives its dependency (constructor injection)
class LoginViewModel {
  final AuthRepository repo;
```

```
  LoginViewModel(this.repo);
  Future<bool> submit(String u, String p) => repo.login(u, p);
}

// Composition root (wire the graph once, at startup)
LoginViewModel buildLoginViewModel() {
  final client = RealHttpClient();
  final repo = HttpAuthRepository(client);
  return LoginViewModel(repo);
}

// In tests: inject fakes, no real HTTP
class FakeAuthRepo implements AuthRepository {
  @override
  Future<bool> login(String u, String p) async => true;
}
Future<void> demo() async {
  final vm = LoginViewModel(FakeAuthRepo()); // swap at the boundary
  assert(await vm.submit('a', 'b'));
}
```

## Diagrams



## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| `new` ing deps inside classes | Untestable, coupled | Inject via constructor |
| Service locator everywhere | Hidden global deps | Prefer ctor injection; locate at edges |
| No single composition root | Wiring scattered | Centralize creation/wiring at startup |
| Injecting trivial pure values | Ceremony | Inject volatile/external deps only |
| Depending on concretes | Recoupling | Depend on abstractions |

## Best Practices

- Prefer **constructor injection of abstractions**; make deps `final`.
- Wire the graph at a **single composition root** (thin `main`).
- Use a container (`get_it`/Riverpod) to *reduce wiring boilerplate*, not to hide dependencies.
- Inject **volatile/external** deps (repos, clients, clock); not trivial values.
- Keep the "dirty" concrete knowledge in the composition root.

## Performance

Negligible; container lookups are cheap, singletons avoid re-creation (05 · dependency_injection).

## Advantages / Disadvantages

- **+** Testable (fakes), swappable (flavors), explicit graph, single wiring point, enables DIP/OCP/Clean Architecture.
- **−** Boilerplate/wiring; service-locator overuse hides deps; lifetime management needs care.

## Interview Questions

1. 🟢 **What is a composition root?** — The single startup place where concrete implementations are created and wired to abstractions; the rest of the app stays detail-agnostic.
2. 🟢 **Constructor injection vs service locator?** — Constructor injection passes deps explicitly (visible, testable); a service locator is a global registry fetched by type (convenient but hides deps, runtime errors).
3. 🟡 **How is the Flutter widget tree a DI mechanism?** — `InheritedWidget`/Provider/Riverpod supply dependencies to descendants via `context`/`ref` — tree-scoped DI.
4. 🟡 **What should you inject vs not?** — Volatile/external deps (repos, HTTP clients, clock, config); not trivial pure helpers/values.

5. 🟡 **How does DI enable testing?** — Substitute fakes/mocks for abstractions without changing consumers.
6. 🔴 **Why prefer constructor injection over a locator generally?** — It makes dependencies explicit and compile-checked; locators hide the graph and defer errors to runtime — but locators are handy at composition edges.
7. 🔴 **How does DI relate to build flavors?** — The composition root wires different implementations per flavor (dev/prod/mock) without touching feature code.

## Senior Engineer Tips

- Blend approaches deliberately: a locator/container for app services + constructor injection into view models/use cases keeps the graph both convenient and explicit.
- Keep the composition root thin and the only place that names concretes.
- Inject a `Clock` / `Uuid` / `Random` abstraction so time/randomness are testable — a common oversight.

## Architect Perspective

DI is the connective tissue of testable, modular architecture — the practical realization of DIP and the wiring layer of Clean Architecture (Module 40). A clear composition root, abstraction-based injection, and disciplined lifetimes enable testing, flavors, and modular boundaries at scale. The *mechanism* (get_it/Riverpod/Provider) matters less than the discipline.

## Summary

- DI supplies deps from outside; wire at a composition root; prefer constructor injection of abstractions.
- Service locators and tree-based DI (Provider/Riverpod) are alternatives/complements; blend deliberately.
- Inject volatile deps; keep concretes at the root; enables testing/flavors/DIP.

## Revision Notes

- Composition root = single startup wiring place (thin `main`).
- Ctor injection (explicit/testable) vs service locator (convenient/hidden/runtime) vs tree DI (Provider/Riverpod).
- Inject volatile/external deps (abstractions); keep concretes at root.
- Enables testing (fakes), flavors, DIP/Clean Arch.

## Practice Questions

1. Why is a `new`-ing class untestable, and how does DI fix it?
2. Constructor injection vs service locator tradeoffs?
3. What belongs in the composition root?

## Coding Questions

1. Wire a `Client→Repo→ViewModel` graph at a composition root.
2. Swap the repo for a fake in a test.
3. Introduce a `Clock` abstraction and inject a fake in a test.

## Mini Project

**Composition root (Dart/Flutter):** Wire `HttpClient → AuthRepository → LoginViewModel` at a single composition root with constructor injection of abstractions; provide a fake repo path for tests. Acceptance: no `new` inside consumers; graph wired once; swappable/testable; analyzer clean.

# `get_it` & `injectable`

> `get_it` is a lightweight **service locator** — register implementations against abstractions once, then fetch them anywhere with `getIt<T>()`; `injectable` adds annotation-based **codegen** so registrations are generated from your classes instead of hand-written.

## Introduction

`get_it` (package: `get_it`) is the most popular Flutter service locator. You register dependencies (singleton/lazy/factory) in a composition root and retrieve them without `BuildContext`. `injectable` (package: `injectable` + codegen) auto-generates the registration boilerplate from annotations. This file covers both.

## Why this concept exists

Passing dependencies through many constructors/widgets is tedious; `get_it` provides a central, context-free registry to fetch services from anywhere (great for repositories/services/blocs). `injectable` removes the manual registration boilerplate as the graph grows.

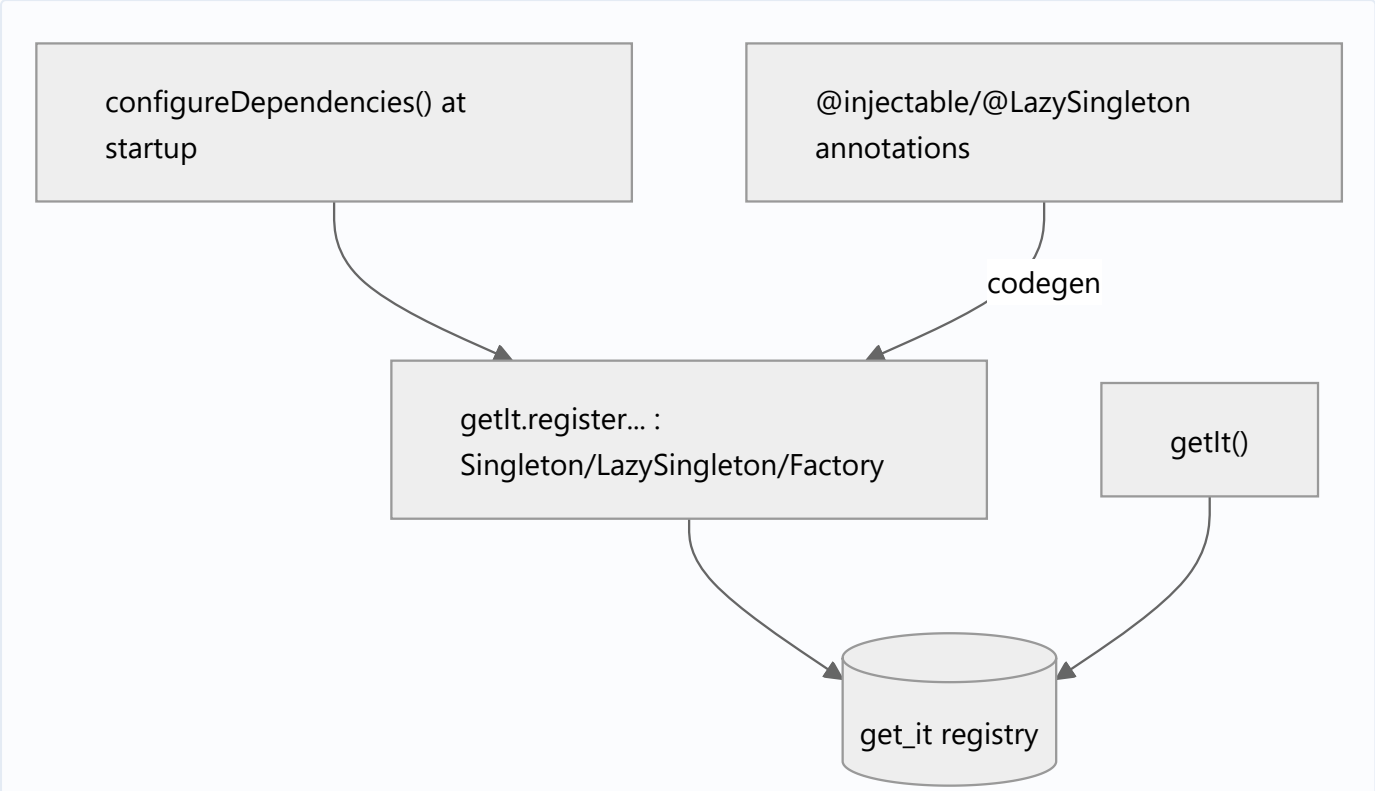
## Real-world analogy

`get_it` is a **tool crib** at a worksite: tools (services) are checked in once; any worker fetches what they need by name (`getIt<Drill>()`) without carrying everything. `injectable` is an **automated inventory system** that stocks the crib from labels on the tools, so you don't hand-list every item.

## Problem Statement

An app has an `HttpClient`, an `AuthRepository`, and an `AnalyticsService` used across many screens/blocs. You want to register them once and fetch anywhere, then automate registration as they multiply. You'll use `get_it`, then `injectable`.

## Internal Working




- **Register** (composition root):
  - `getIt.registerSingleton<T>(instance)` — one eager instance.
  - `getIt.registerLazySingleton<T>(() => impl)` — created on first `getIt<T>()`, then cached.
  - `getIt.registerFactory<T>(() => impl)` — new instance every fetch.
  - `getIt.registerFactoryParam` — factory with runtime params.
- **Resolve**: `getIt<T>()` (or `getIt.get<T>()`) anywhere — no `BuildContext` needed.
- **Register against the abstraction** (`registerLazySingleton<AuthRepository>(() => HttpAuthRepository(getIt()))`) so consumers depend on the interface (DIP).
- `injectable`: annotate classes (`@injectable`, `@LazySingleton(as: AuthRepository)`, `@singleton`, `@module` for third-party); run `build_runner` to generate `configureDependencies()` (the wiring) — no manual `register` calls.
- **Scopes/reset**: `getIt.pushNewScope()` / `popScope()` for scoped deps; `getIt.reset()` in tests.

## Memory Representation

Singletons/lazy-singletons persist for the app (or scope) lifetime; factories are transient. Registered disposables should be disposed on scope pop/reset (`dispose` param) (08 · dispose_and_leaks).

## Compiler Behavior

`getIt<T>()` for an unregistered type throws at **runtime** (not compile-time) — a service-locator tradeoff. `injectable` codegen surfaces some wiring issues at build time.

## Runtime Behavior

Lazy singletons construct on first access; factories construct each call. Fetching missing/unregistered types throws; ordering matters (register dependencies before dependents, or use lazy).

## Flutter Engine Behavior / Dart VM Behavior

Not applicable.

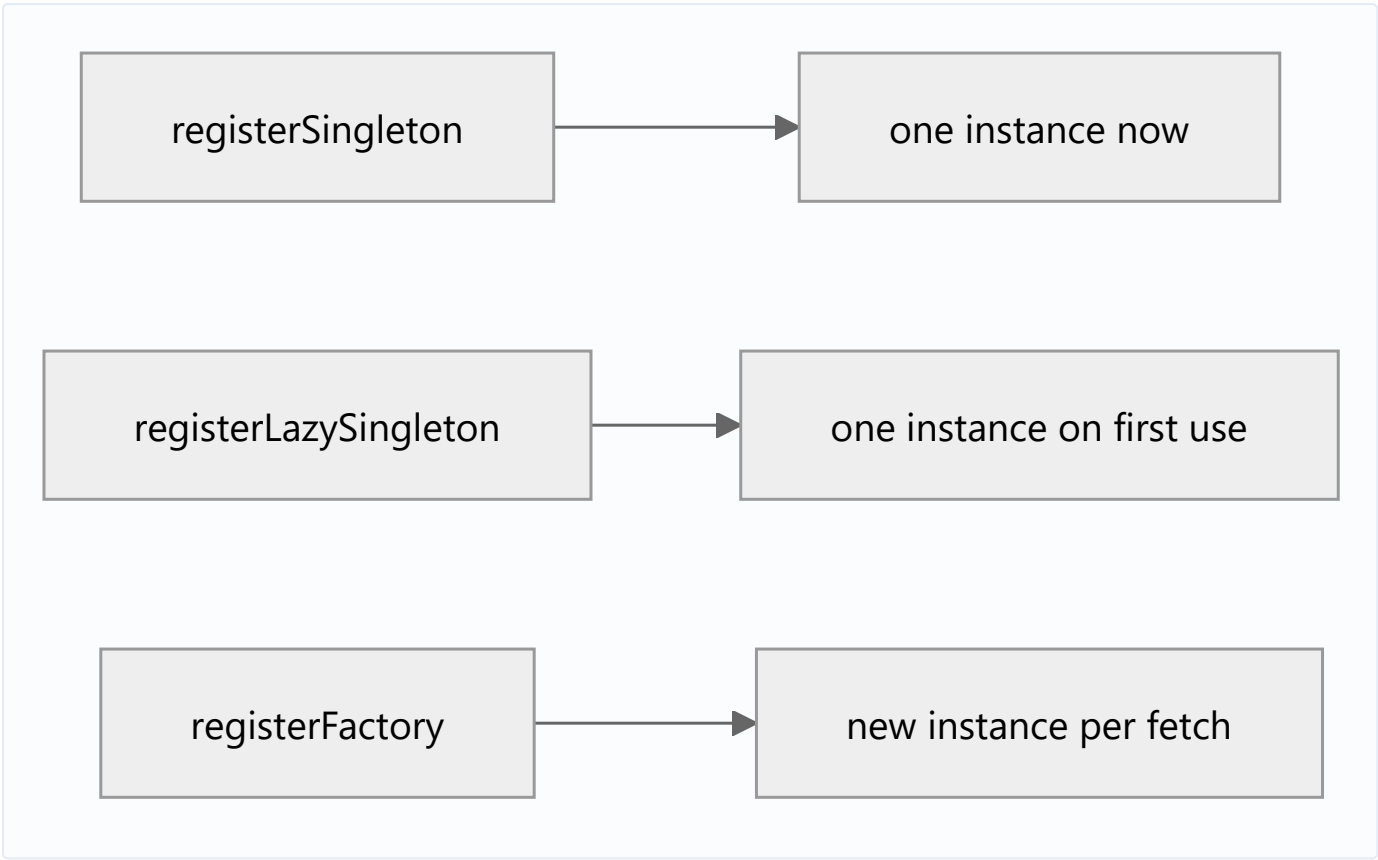
## Examples

```yaml
# pubspec.yaml
dependencies:
  get_it: ^7.6.0
  injectable: ^2.4.0
dev_dependencies:
  injectable_generator: ^2.6.0
  build_runner: ^2.4.0
```

```dart
import 'package:get_it/get_it.dart';

final getIt = GetIt.instance;

abstract interface class HttpClient { Future<String> get(String url); }
abstract interface class AuthRepository { Future<bool> login(String u, String p); }

class RealHttpClient implements HttpClient {
  @override
  Future<String> get(String url) async => 'ok';
}
class HttpAuthRepository implements AuthRepository {
  final HttpClient client;
  HttpAuthRepository(this.client);
  @override
  Future<bool> login(String u, String p) async => (await client.get('/login')) == 'ok';
}

// Manual registration (composition root):
void configureDependencies() {
  getIt.registerLazySingleton<HttpClient>(() => RealHttpClient());
  getIt.registerLazySingleton<AuthRepository>(
      () => HttpAuthRepository(getIt<HttpClient>())); // resolve dependency
  // getIt.registerFactory<LoginViewModel>(() => LoginViewModel(getIt()));
}

Future<void> main() async {
  configureDependencies();          // wire once at startup
  final repo = getIt<AuthRepository>(); // fetch anywhere, no context
  await repo.login('a', 'b');
}
```

```dart
// With injectable (codegen) — annotate, then `dart run build_runner build`:
// @LazySingleton(as: AuthRepository)
// class HttpAuthRepository implements AuthRepository { HttpAuthRepository(this.client); ... }
//
// @InjectableInit()
// void configureDependencies() => getIt.init(); // generated wiring
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
| --- | --- | --- |
| Registering against concretes | Recoupling | Register `<Abstraction>(() => Impl(...))` |
| `getIt<T>()` unregistered | Runtime crash | Register in the composition root; check ordering |
| Locator calls sprinkled deep in widgets | Hidden deps, hard tests | Fetch at edges; inject into constructors |
| Not disposing scoped/disposable singletons | Leaks | Provide `dispose`, pop scopes, `reset()` in tests |
| Forgetting `build_runner` (injectable) | Stale generated wiring | Run/watch codegen |

## Best Practices

- Register against **abstractions**; wire in one `configureDependencies()` composition root.
- Prefer **lazy singletons** for services (created on demand, cached); **factories** for per-use objects (e.g., a fresh bloc).
- Fetch from `get_it` at **composition edges** (e.g., provide a bloc), then use **constructor injection** downstream so the graph stays explicit/testable.
- Use `injectable` codegen once the graph is non-trivial; use `@module` for third-party types.
- Manage lifetimes/disposal (scopes, `dispose`, `reset()` in tests).

## Performance

Lazy singletons defer creation; factories allocate per call. Registry lookups are O(1)-ish. No notable runtime cost (04_scopes_and_lifetimes.md).

## Advantages / Disadvantages

- + Simple, context-free, popular, flexible lifetimes, `injectable` removes boilerplate, great for services/blocs.
- − Service-locator: hidden dependencies, runtime (not compile-time) errors, global-state risk if overused; codegen setup for `injectable`.

## Interview Questions

1. 🟢 **What is `get_it`?** — A service locator: a global registry where you register implementations (against abstractions) and fetch them via `getIt<T>()` without `BuildContext`.
2. 🟢 **Singleton vs lazySingleton vs factory?** — Eager single instance; single instance created on first access; a new instance per fetch.
3. 🟡 **Why register against an abstraction?** — So consumers depend on the interface (DIP) and you can swap implementations (incl. fakes in tests).
4. 🟡 **What does `injectable` add?** — Annotation-based codegen that generates the `get_it` registrations, removing manual boilerplate.
5. 🟡 **What's the main downside of a service locator?** — Hidden dependencies (not visible in signatures) and runtime errors for unregistered types.
6. 🔴 **How do you keep `get_it` from becoming global-state sprawl?** — Fetch at composition edges and use constructor injection downstream; register abstractions; scope where possible.
7. 🔴 **How do you handle disposal/scoping?** — Use `pushNewScope` / `popScope` and register `dispose` callbacks; `reset()` between tests.

## Senior Engineer Tips

- Treat `get_it` as your composition-root wiring, not an excuse to call `getIt<T>()` everywhere — fetch at the edge, inject downstream.
- Use lazy singletons by default for services; factories for stateful per-screen objects (blocs/view models).
- Adopt `injectable` when hand-registration becomes tedious; keep `@module` for third-party (Dio, SharedPreferences).

## Architect Perspective

`get_it` / `injectable` provide a pragmatic, context-free DI container that pairs well with Clean Architecture: register repositories/data sources/services at startup, inject into use cases/view models. The service-locator tradeoffs (hidden deps, runtime errors) are mitigated by edge-only lookups + constructor injection — a common, scalable enterprise setup (Module 40).

## Summary

- `get_it` = service locator: register (singleton/lazy/factory) against abstractions in a composition root, fetch via `getIt<T>()`.
- `injectable` generates registrations from annotations.
- Fetch at edges + inject downstream; manage lifetimes/disposal; runtime (not compile-time) errors.

## Revision Notes

- `getIt.registerSingleton/LazySingleton/Factory<Abstraction>(() => Impl(getIt()))`; fetch `getIt<T>()`.
- `injectable`: `@injectable` / `@LazySingleton(as:)` / `@module` + `build_runner` → generated `configureDependencies`.
- Fetch at edges, inject downstream; abstractions; lifetimes/disposal via scopes/ `reset()`.
- Runtime errors for unregistered types (service-locator tradeoff).

## Practice Questions

1. When use lazySingleton vs factory?
2. Why register against an abstraction, not the concrete?
3. What problem does `injectable` solve?

## Coding Questions

1. Register `HttpClient` + `AuthRepository` in `get_it` and resolve the repo.
2. Convert manual registrations to `injectable` annotations + codegen.
3. Register a factory bloc and fetch a fresh instance per screen.

## Mini Project

**Service registry (Flutter):** Set up `get_it` with lazy-singleton `HttpClient` / `AuthRepository` / `AnalyticsService` (registered against abstractions) in one `configureDependencies()`, plus a factory view model; fetch at a screen edge and inject downstream.

Add an `injectable` variant. Acceptance: abstractions registered; edge-only lookups; lifetimes correct; analyzer clean.

# Provider & Riverpod as DI

> Provider and Riverpod aren't just state solutions — they're **DI containers**: Provider injects objects down the widget tree (via `InheritedWidget`), and Riverpod exposes them as compile-safe, overridable, testable top-level providers accessed by `ref`.

## Introduction

You already met these as state solutions (Module 11); here we use them purely as **dependency injection**. Provider provides services scoped to the tree; Riverpod provides them via a container with compile-time safety and easy test overrides. Both can replace or complement `get_it`.

## Why this concept exists

Widget-tree-scoped and reactive dependencies are natural in Flutter. Provider/Riverpod inject services where they're needed (often scoped to a subtree/feature) and, crucially, make **test overrides** trivial — a big advantage over a global locator. Riverpod adds compile-time safety.

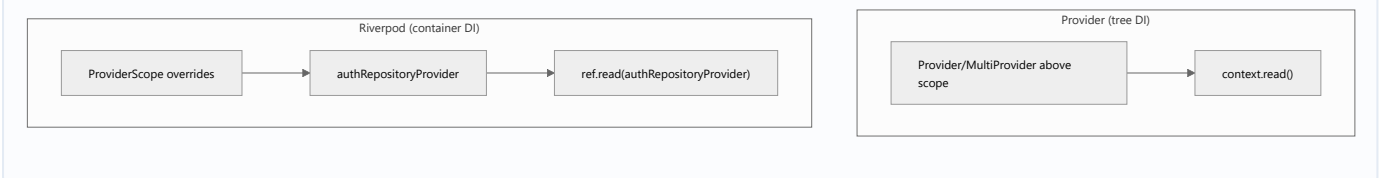
## Real-world analogy

Provider is **piping utilities to specific floors** of a building (tree-scoped delivery); Riverpod is a **service registry with a staging environment** — you declare services centrally and can spin up a test building with mocked services (overrides) effortlessly.

## Problem Statement

Inject an `AuthRepository` and `AnalyticsService` so screens/view models use them, scoped appropriately, and swap them for fakes in tests. You'll do it with Provider and with Riverpod, contrasting with `get_it`.

## Internal Working




- **Provider as DI**:
  - `Provider<AuthRepository>(create: (_) => HttpAuthRepository(...))` (plain service, not a `ChangeNotifier`).

    - `MultiProvider` to compose several.
    - Access via `context.read<AuthRepository>()` (in callbacks/wiring) — subscribe only if reactive.
    - **Scope**: place providers above the subtree that needs them (app-wide above `MaterialApp`, or feature-scoped).
    - Auto-disposes `ChangeNotifierProvider` / `Provider` with `dispose`.
- **Riverpod as DI**:
    - `final authRepositoryProvider = Provider<AuthRepository>((ref) => HttpAuthRepository(ref.read(httpClientProvider)));`
    - Access via `ref.read(authRepositoryProvider)` (or `watch` if reactive).
    - **Compile-safe** (typed top-level symbols); **overridable** in `ProviderScope(overrides: [...])` for tests/flavors; `ref.watch` composes providers.
- **vs** `get_it`: Provider/Riverpod are tree/container-scoped and (Riverpod) compile-safe + override-friendly; `get_it` is a flat global locator (context-free). Choose per app conventions.

## Memory Representation

Provider objects live in the tree (scope-bound); Riverpod state lives in the `ProviderContainer` (with `autoDispose` for cleanup) (11 · riverpod, 08 · dispose_and_leaks).

## Compiler Behavior

Riverpod: referencing a provider is compile-checked. Provider: `read<T>()` for an unprovided type throws at runtime (`ProviderNotFoundException`).

## Runtime Behavior

Provider resolves up the tree from `context`; Riverpod resolves from the container via `ref`. Overrides (Riverpod) or a different provider subtree (Provider) inject alternate implementations.

## Flutter Engine Behavior / Dart VM Behavior

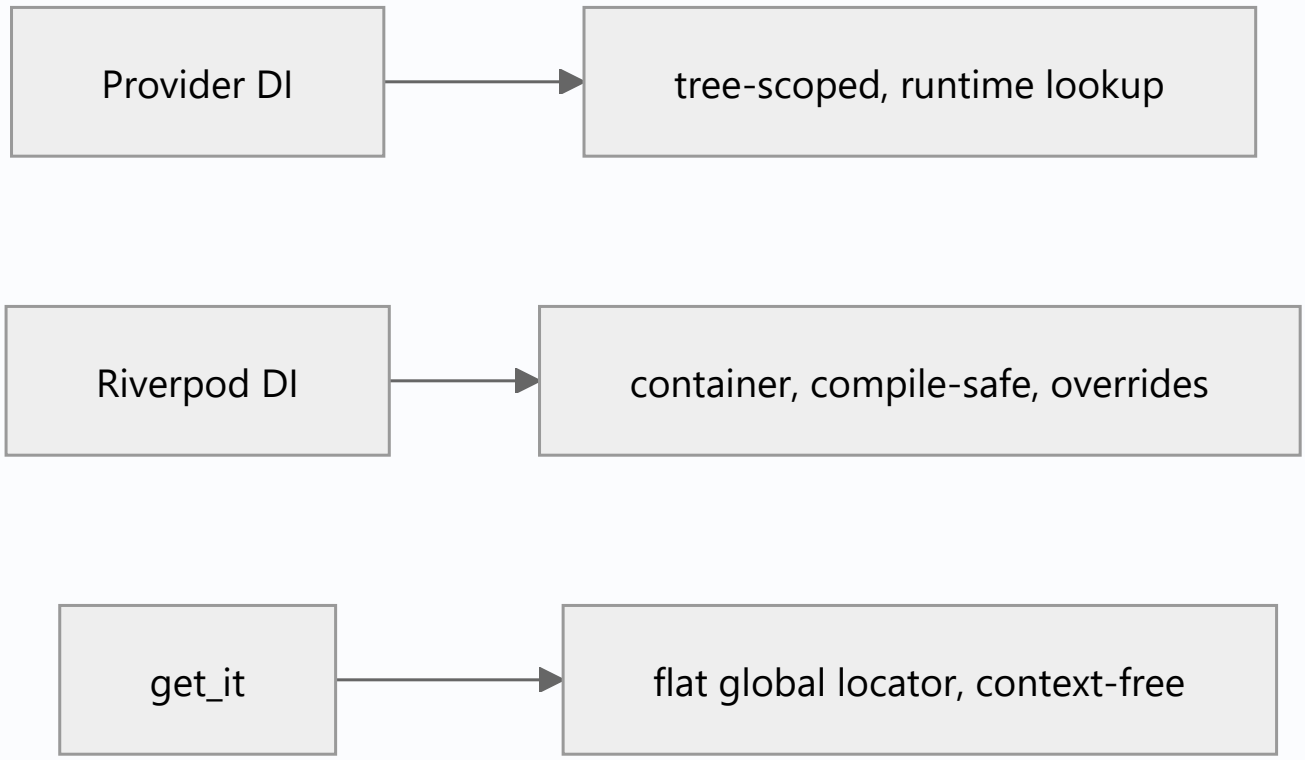
Not applicable.

## Examples

```
// ---- Provider as DI ----
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

abstract interface class AuthRepository { Future<bool> login(); }
class HttpAuthRepository implements AuthRepository {
  @override
  Future<bool> login() async => true;
}

void mainProvider() {
  runApp(
    MultiProvider(
      providers: [
        Provider<AuthRepository>(create: (_) => HttpAuthRepository()), // DI, plain service
      ],
      child: const MyApp(),
    ),
  );
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) => MaterialApp(
        home: Builder(builder: (context) {
          final repo = context.read<AuthRepository>(); // resolve dependency
          return Scaffold(body: Center(child: Text('repo: ${repo.runtimeType}')));
        }),
      );
}
```

```
// ---- Riverpod as DI ----
import 'package:flutter_riverpod/flutter_riverpod.dart';

final httpClientProvider = Provider<HttpAuthRepository>((ref) => HttpAuthRepository());
final authRepositoryProvider = Provider<AuthRepository>(
  (ref) => ref.read(httpClientProvider), // compose providers
);

// Test/flavor override (no widget tree needed):
// final container = ProviderContainer(overrides: [
//   authRepositoryProvider.overrideWithValue(FakeAuthRepo()),
// ]);
// container.read(authRepositoryProvider); // -> FakeAuthRepo
```

## Diagrams



## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Using `ChangeNotifierProvider` for a plain service | Unneeded reactivity | Use `Provider<T>` for services |
| Providing too low in the tree | Consumers can't see it | Provide above the needed subtree |
| `watch` for a non-reactive dependency | Extra rebuilds | Use `read` for pure DI access |
| Riverpod without `ProviderScope` | No container | Wrap app in `ProviderScope` |
| Mixing locator + tree DI inconsistently | Confusion | Standardize an approach |

## Best Practices

- Use `Provider<T>`/Riverpod `Provider` for **plain service DI** (not `ChangeNotifierProvider`, which is for reactive state).
- **Scope** providers to where they're needed (app-wide vs feature); Riverpod providers are global-but-scoped via overrides.
- Prefer **Riverpod** when you want **compile-time safety + trivial test overrides + composition**; Provider when you're already using it and want simplicity.
- Access with `read` for DI (no subscription); `watch` only for reactive needs.
- Standardize one DI approach across the codebase.

## Performance

DI access is cheap; Riverpod `autoDispose` bounds lifetime. No notable cost vs `get_it` (11 · riverpod).

## Advantages / Disadvantages

- **+ Provider:** simple, tree-scoped, familiar. **+ Riverpod:** compile-safe, context-independent, override-based testing, composition, autoDispose.
- **− Provider:** runtime lookup, context-bound. **− Riverpod:** more concepts/setup. Both: another dependency vs plain `get_it`.

## Interview Questions

1. 🟢 **How is Provider a DI mechanism?** — It provides objects to descendants via `InheritedWidget`; consumers resolve them with `context.read<T>()`.
2. 🟢 **How does Riverpod do DI?** — Services are top-level providers accessed via `ref.read/watch`, resolved from a `ProviderContainer` — compile-safe and overridable.

3. 🟡 **Provider vs `get_it` for DI?** — Provider is tree-scoped and context-bound (runtime lookup); `get_it` is a flat global, context-free locator. Choose by scoping needs/conventions.
4. 🟡 **How do you inject fakes in tests with Riverpod?** — `ProviderScope(overrides: [provider.overrideWithValue(fake)])` (or `ProviderContainer(overrides:)`) — no widget tree needed.
5. 🟡 **`read` vs `watch` for DI?** — `read` for one-off dependency access (no subscription); `watch` only when you need to rebuild on change.
6. 🔴 **Why is Riverpod's compile-safety a DI advantage?** — Missing/mistyped providers are compile errors, not runtime `ProviderNotFoundException`s — safer at scale.
7. 🔴 **When would you still use `get_it` alongside Riverpod/Provider?** — For context-free access (e.g., in pure-Dart services/blocs not tied to the tree) or existing conventions.

## Senior Engineer Tips

- For DI-heavy, testable apps, Riverpod's **overrides** are the cleanest way to swap dependencies per test/flavor.
- Use `Provider<T>` (not `ChangeNotifierProvider`) for stateless services; reserve reactive providers for state.
- Don't run two DI systems ad hoc; pick one primary (Riverpod *or* Provider *or* get_it) and be consistent.

## Architect Perspective

Provider/Riverpod fold DI and state into one tree/container model — reducing moving parts. Riverpod's compile-safe, override-driven DI is especially strong for Clean Architecture: inject repositories/data sources/use cases with easy mocking and flavor overrides (Module 40). The choice among these and `get_it` is a team/consistency decision more than a capability one.

## Summary

- Provider (tree, runtime) and Riverpod (container, compile-safe, overridable) are full DI mechanisms, not just state tools.
- Use `Provider<T>`/Riverpod `Provider` for services; scope appropriately; `read` for DI access.
- Riverpod's overrides make test/flavor injection trivial; standardize one approach (or combine with `get_it` deliberately).

## Revision Notes

- Provider DI: `Provider<T>(create:)` + `context.read<T>()` (tree-scoped, runtime lookup).
- Riverpod DI: top-level `Provider` + `ref.read/watch`; compile-safe; `ProviderScope(overrides:)` for tests/flavors; `autoDispose`.

- `read` for DI, `watch` for reactive; `Provider<T>` (not ChangeNotifierProvider) for services.
- Choose one primary DI (Riverpod/Provider/get_it); combine deliberately.

## Practice Questions

1. Why use `Provider<T>` not `ChangeNotifierProvider` for a service?
2. How do Riverpod overrides help testing?
3. Provider vs `get_it` scoping differences?

## Coding Questions

1. Inject an `AuthRepository` via `Provider` and resolve it in a screen.
2. Inject the same via Riverpod and override it with a fake in a container test.
3. Compose two Riverpod providers (client → repository).

## Mini Project

**DI two ways (Flutter):** Inject `AuthRepository` (+ `AnalyticsService`) using (a) Provider/ `MultiProvider` and (b) Riverpod providers; in each, write a test that swaps a fake (Provider: different provider subtree; Riverpod: override). Acceptance: services injected (not `new`ed) in consumers; fakes swapped in tests; consistent per-approach; app/tests run.

# Scopes & Lifetimes (Singleton / Factory / Lazy / Scoped) + Disposal

> Every injected dependency has a **lifetime** — app-wide singleton, per-request factory, lazily-created, or scoped to a feature/session — and choosing it correctly (plus disposing when it ends) prevents both stale-state bugs and memory leaks.

## Introduction

DI isn't just *what* to inject but *how long it lives*. This file classifies lifetimes (singleton, lazy singleton, factory/transient, scoped), maps them across `get_it`/Riverpod/Provider, and covers **disposal** so scoped/disposable dependencies are released.

## Why this concept exists

An `HttpClient` should usually be one shared instance (singleton); a per-screen bloc should be fresh (factory) and disposed on exit; a session-scoped cache should live only while logged in (scoped). Wrong lifetimes cause leaks (never-disposed) or bugs (unexpected sharing/re-creation).

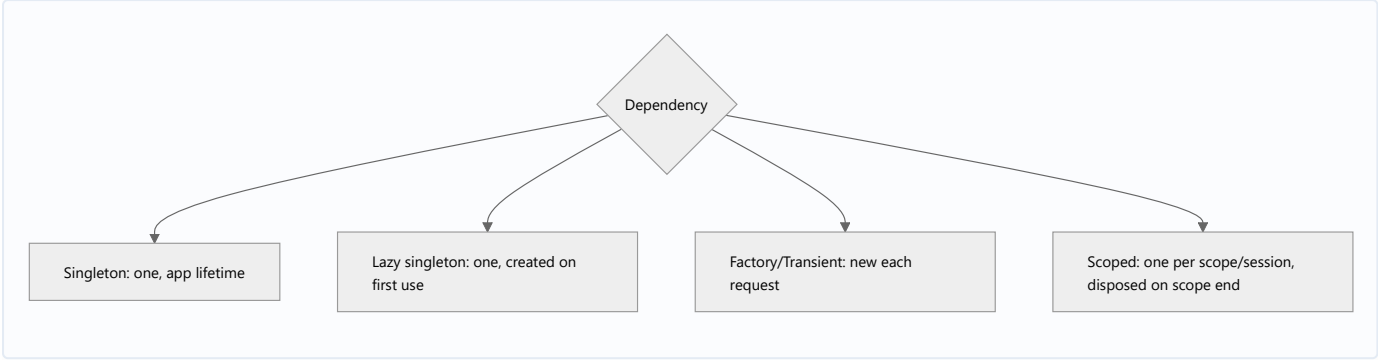
## Real-world analogy

Lifetimes are **rental terms**: buy-once shared equipment (singleton), single-use rentals returned after each job (factory), equipment leased for a project then returned (scoped). Forgetting to return leased gear = leak; sharing single-use gear = contamination.

## Problem Statement

Your app needs: one `HttpClient` (app-wide), a fresh `EditBloc` per edit screen (disposed on close), and a `SessionCache` that exists only while logged in (cleared on logout). You'll assign correct lifetimes and disposal.

## Internal Working




| Lifetime | Meaning | Use for | `get_it` | Riverpod | Provider |
|---|---|---|---|---|---|
| **Singleton** | One instance, app lifetime | Stateless shared services | `registerSingleton` | keep-alive `Provider` | app-level `P` |
| **Lazy singleton** | One, created on first use | Expensive shared services | `registerLazySingleton` | `Provider` (lazy by default) | `Provider(` |
| **Factory / transient** | New each fetch | Per-screen blocs/view models | `registerFactory` | `.autoDispose` + `.family` | `ChangeNot` per route |
| **Scoped** | One per scope/session | Session/feature state | `pushNewScope` / `popScope` | scoped `ProviderScope` / `autoDispose` | subtree pro |

- **Disposal**: singletons live until app end (rarely disposed); factories/scoped **must** be disposed when their owner/scope ends. `get_it` register `dispose:` + `popScope`; Riverpod `autoDispose` (and `ref.onDispose`); Provider auto-disposes `ChangeNotifierProvider` on removal (08 · dispose_and_leaks).
- **Scoped DI**: tie a dependency's life to a session/feature (e.g., a scope pushed at login, popped at logout) so it's created and torn down with that context.

## Memory Representation

Singletons are effectively GC roots for their lifetime (never collected while registered — watch what they retain). Factory instances are collected once unreferenced; scoped instances are disposed on scope end (05 · singleton, 02 · memory_and_gc).

## Compiler Behavior

Not applicable.

## Runtime Behavior

Lazy singletons construct on first resolve; factories construct per resolve; scoped instances construct on scope entry and dispose on exit. Missing disposal → leaks; over-eager singletons → wasted startup/memory.

## Flutter Engine Behavior / Dart VM Behavior

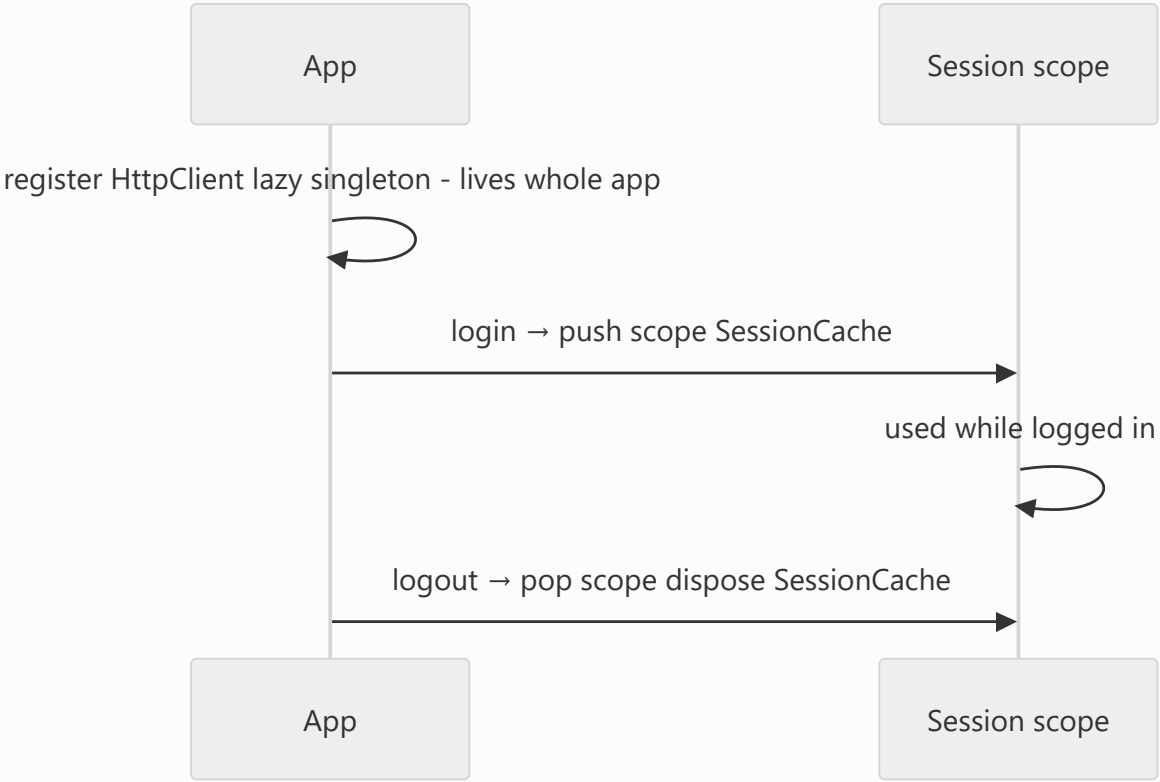
Not applicable.

## Examples

```
import 'package:get_it/get_it.dart';

final getIt = GetIt.instance;

abstract interface class HttpClient {}
class RealHttpClient implements HttpClient {}
class SessionCache { void clear() {} }
class EditBloc { void dispose() {} }

void configure() {
  // App-wide singleton (lazy): created on first use, lives forever
  getIt.registerLazySingleton<HttpClient>(() => RealHttpClient());

  // Factory: NEW EditBloc each time (per screen); dispose it when the screen closes
  getIt.registerFactory<EditBloc>(() => EditBloc());
}

// Session scope: create at login, dispose at logout
void onLogin() {
  getIt.pushNewScope(
    scopeName: 'session',
    init: (getIt) {
      getIt.registerSingleton<SessionCache>(SessionCache());
    },
    dispose: () {
      // called on popScope: release session-scoped deps
    },
  );
}
Future<void> onLogout() async {
  await getIt.popScope(); // disposes session scope (SessionCache)
}
```

```
// Riverpod lifetimes:
// final httpClientProvider = Provider<HttpClient>((ref) => RealHttpClient()); // kept alive
// final editBlocProvider = Provider.autoDispose<EditBloc>((ref) {
//   final bloc = EditBloc();
//   ref.onDispose(bloc.dispose); // disposed when no longer watched
//   return bloc;
// });
```

## Diagrams



## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Singleton for per-screen state | Shared/stale across screens | Use factory/ `autoDispose` |
| Factory for a shared service | Wasteful re-creation/inconsistency | Use (lazy) singleton |
| Never disposing scoped/factory deps | Leaks | `popScope` / `autoDispose` / `ref.onDispose` /dispose |
| Eager singletons for rarely-used deps | Slower startup/memory | Lazy singleton |
| Session data as app singleton | Leaks across logout | Scope to session; clear on logout |

## Best Practices

- Default services to **lazy singletons**; per-screen state to **factory/** `autoDispose`; session/feature state to **scoped**.
- **Always dispose** factory/scoped disposables (streams, blocs, controllers) at end-of-life (08 · dispose_and_leaks).
- Tie **session-scoped** deps to auth (push scope on login, pop on logout) so they clear correctly.
- Audit what **singletons retain** (they never get collected) — avoid holding large/transient data.
- Prefer Riverpod `autoDispose` for automatic lifetime management where suitable.

## Performance

Lazy creation avoids startup cost; correct disposal keeps memory flat. Wrong lifetimes cause leaks (growing memory) or churn (excess re-creation) (Module 21).

## Advantages / Disadvantages

- **+** Right lifetimes prevent leaks/stale-state, bound memory, and match real ownership (session/feature).
- **−** Requires deliberate choice + disposal discipline; scoped DI adds complexity.

## Interview Questions

1. 🟢 **Name the common DI lifetimes.** — Singleton, lazy singleton, factory/transient, and scoped.
2. 🟢 **When use a factory vs a singleton?** — Factory for per-use/per-screen instances (fresh, disposed); singleton for shared stateless services.

3. 🟡 **What is a lazy singleton and why prefer it?** — One instance created on first use (then cached); avoids paying creation cost until needed.
4. 🟡 **What is scoped DI, with an example?** — A dependency living for a scope's duration (e.g., a `SessionCache` created at login, disposed at logout via `pushNewScope` / `popScope`).
5. 🟡 **Why must factory/scoped deps be disposed?** — They own resources (streams/blocs/controllers); not disposing leaks them (singletons live for the app, so rarely disposed).
6. 🔴 **What's the risk of storing session data in an app singleton?** — It persists across logout (stale/leaked user data); scope it to the session instead.
7. 🔴 **How does Riverpod manage lifetimes?** — Providers are kept-alive by default; `.autoDispose` disposes when unwatched; `ref.onDispose` runs cleanup; `.family` parameterizes instances.

## Senior Engineer Tips

- Map each dependency to an **ownership/lifetime** explicitly; most leaks and stale-state bugs are lifetime mistakes.
- Use **session scopes** for anything tied to auth; pop on logout to guarantee cleanup.
- Prefer `autoDispose`/factory + dispose for anything holding resources; reserve singletons for truly shared, stateless services.

## Architect Perspective

Lifetime/scope design is a cross-cutting reliability and correctness concern: it governs memory, data freshness (session isolation), and resource cleanup across the app. Establishing conventions (services=lazy singleton, screen state=factory/autoDispose, session=scoped) and enforcing disposal is essential for leak-free, correct apps at scale (Modules 21, 40).

## Summary

- Lifetimes: singleton, lazy singleton, factory/transient, scoped — pick by ownership.
- Services → lazy singleton; per-screen → factory/`autoDispose`; session/feature → scoped.
- Always dispose factory/scoped disposables; scope session data to auth; audit singleton retention.

## Revision Notes

- Singleton (app), lazy singleton (first use), factory (per fetch), scoped (per session/feature).
- get_it: `registerSingleton/LazySingleton/Factory` + `pushNewScope/popScope(dispose:)`.
- Riverpod: kept-alive vs `.autoDispose` + `ref.onDispose`; Provider auto-disposes on removal.

- Dispose factory/scoped; scope session data; don't leak via singletons.

## Practice Questions

1. Which lifetime for an `HttpClient` vs a per-screen bloc vs a session cache?
2. Why lazy over eager singletons?
3. How do you guarantee session data clears on logout?

## Coding Questions

1. Register a lazy-singleton client + factory bloc in `get_it`; dispose the bloc on screen close.
2. Implement a session scope (push on login, pop on logout) with a `SessionCache`.
3. Use Riverpod `autoDispose` + `ref.onDispose` for a screen-scoped resource.

## Mini Project

**Lifetime-correct wiring (Flutter):** Wire a lazy-singleton `HttpClient`, a factory `EditBloc` (disposed on screen close), and a session-scoped `SessionCache` (cleared on logout), using `get_it` scopes (or Riverpod `autoDispose`). Prove (notes/tests) session data clears on logout and no factory leaks. Acceptance: correct lifetimes; disposal verified; session isolation; app runs.

# Testing with DI (Fakes, Mocks, Overrides)

> The payoff of DI is testability: because consumers depend on abstractions, you inject **fakes/mocks** in tests to run logic without real network/DB/platform — via constructor injection, `get_it.reset()` +re-register, or Riverpod/Provider overrides.

## Introduction

DI's whole point (beyond decoupling) is that you can substitute dependencies. This file shows how to test with injected doubles: constructor-injected fakes, `get_it` test setup, Riverpod/Provider overrides, and the fake-vs-mock distinction. It's the bridge to Module 49 Testing.

## Why this concept exists

Testing logic against real HTTP/DB/platform is slow, flaky, and often impossible in unit tests. DI lets you inject a controllable double (fake/mock) so tests are fast, deterministic, and isolated — verifying *your* logic, not the dependency's.

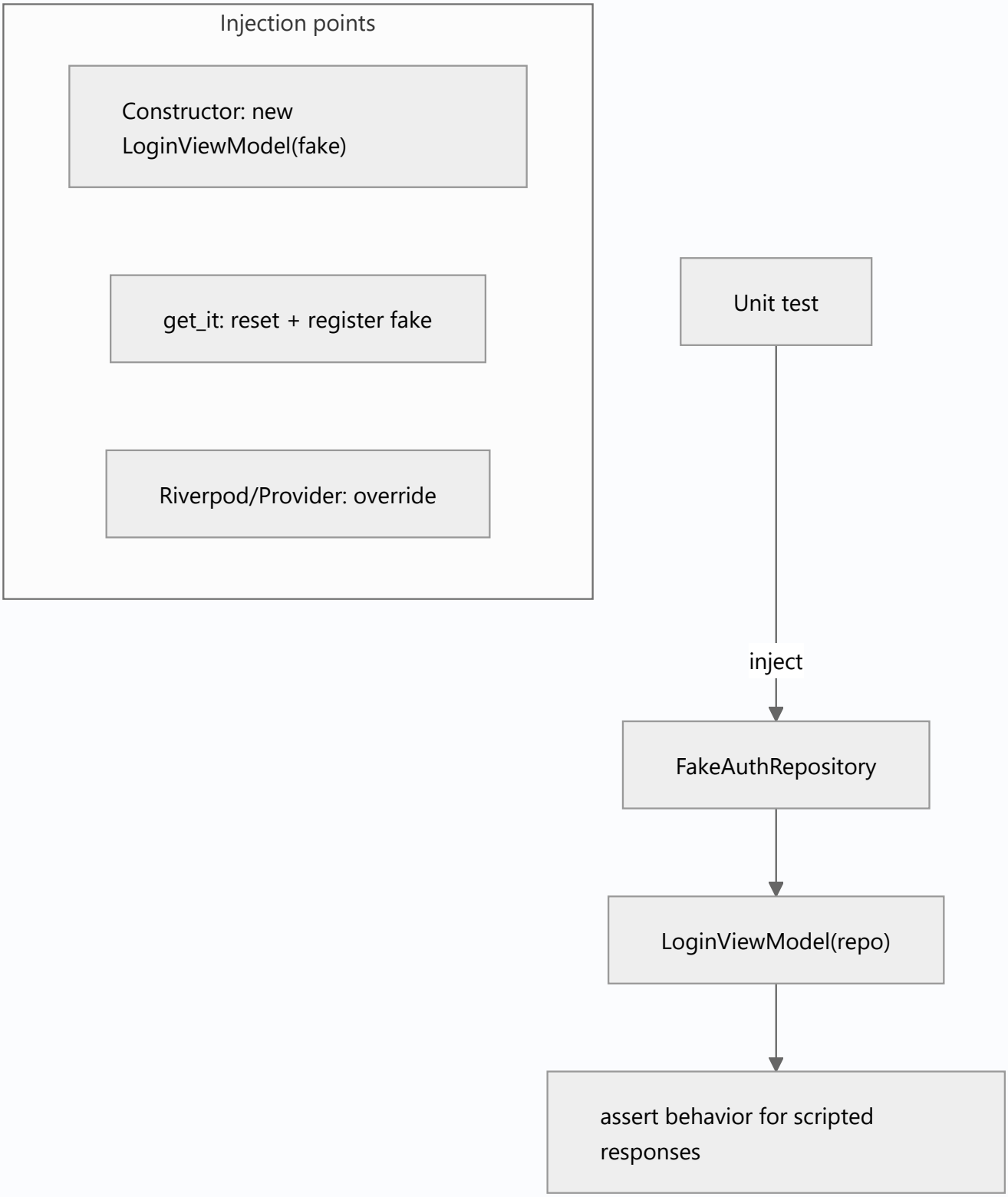
## Real-world analogy

A **flight simulator**: pilots (your logic) train against a simulated cockpit (fakes) instead of a real plane (production deps) — safe, repeatable, and you can script any scenario (errors, edge cases) on demand.

## Problem Statement

Test a `LoginViewModel` that calls an `AuthRepository` (which hits the network) — for success, failure, and exception cases — without any real network. You'll inject a fake/mock repository three ways.

## Internal Working




- **Fake vs mock**:
  - **Fake**: a working lightweight implementation (in-memory repo, canned responses) — you write it.

    - **Mock**: a generated/configured stub (via `mockito` / `mocktail`) where you stub methods and verify calls.
- **Injection in tests**:
    - **Constructor injection** (simplest): `LoginViewModel(FakeAuthRepo())` — no framework needed.
    - `get_it`: `getIt.reset()` then register fakes in test `setUp`; code that resolves `getIt<T>()` gets the fake.
    - **Riverpod**: `ProviderContainer(overrides: [authRepoProvider.overrideWithValue(fake)])` — pure Dart, no widgets.
    - **Provider**: wrap the widget-under-test in a `Provider` supplying the fake (widget tests).
- **Widget tests**: use `pumpWidget` with the DI wired to fakes; `bloc_test` for blocs; `mocktail` / `mockito` for verifications.

## Memory Representation

Test doubles are lightweight objects; reset DI (`getIt.reset()`, dispose containers) between tests to avoid state bleed (08 · dispose_and_leaks).

## Compiler Behavior

Abstractions make substitution type-safe; Riverpod overrides are compile-checked. `mocktail` avoids codegen; `mockito` uses codegen (`build_runner`).

## Runtime Behavior

The consumer runs its real logic against the double's scripted behavior; tests assert results/interactions. No real I/O occurs.

## Flutter Engine Behavior / Dart VM Behavior

Not applicable.

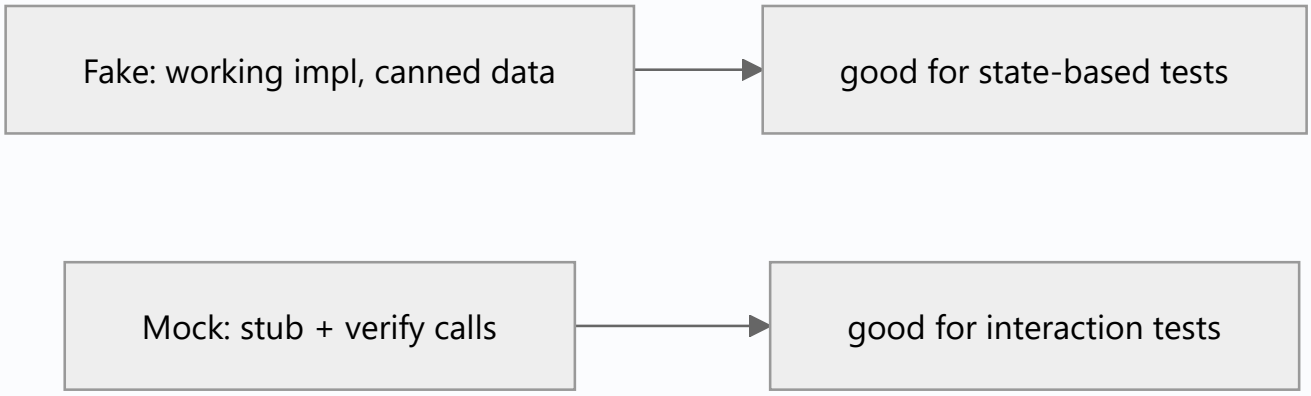
## Examples

```
import 'package:test/test.dart';


abstract interface class AuthRepository { Future<bool> login(String u, String p); }
class LoginViewModel {
  final AuthRepository repo;
  LoginViewModel(this.repo);
```

```
  Future<String> submit(String u, String p) async {
    try {
      return (await repo.login(u, p)) ? 'success' : 'invalid';
    } catch (_) {
      return 'error';
    }
  }
}

// 1) FAKE: hand-written, scriptable
class FakeAuthRepo implements AuthRepository {
  final bool result;
  final bool throwErr;
  FakeAuthRepo({this.result = true, this.throwErr = false});
  @override
  Future<bool> login(String u, String p) async {
    if (throwErr) throw Exception('network');
    return result;
  }
}

void main() {
  group('LoginViewModel', () {
    test('success', () async {
      final vm = LoginViewModel(FakeAuthRepo(result: true)); // constructor injection
      expect(await vm.submit('a', 'b'), 'success');
    });
    test('invalid', () async {
      final vm = LoginViewModel(FakeAuthRepo(result: false));
      expect(await vm.submit('a', 'b'), 'invalid');
    });
    test('error', () async {
      final vm = LoginViewModel(FakeAuthRepo(throwErr: true));
      expect(await vm.submit('a', 'b'), 'error');
    });
  });
}
```

```
// 2) get_it in tests:
// setUp(() { getIt.reset(); getIt.registerSingleton<AuthRepository>(FakeAuthRepo()); });
//
// 3) Riverpod override (pure Dart):
// final c = ProviderContainer(overrides: [
//   authRepositoryProvider.overrideWithValue(FakeAuthRepo()),
// ]);
// addTearDown(c.dispose);
//
// 4) mocktail:
// class MockAuthRepo extends Mock implements AuthRepository {}
// when(() => repo.login(any(), any())).thenAnswer((_) async => true);
// verify(() => repo.login('a', 'b')).called(1);
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Testing against real network/DB in unit tests | Slow, flaky | Inject fakes/mocks |
| Not resetting `get_it` between tests | State bleed | `getIt.reset()` in `setUp` |
| Not disposing `ProviderContainer` | Leaks across tests | `addTearDown(c.dispose)` |
| Over-mocking (mocking everything) | Brittle, tests the mock | Prefer fakes; mock only at boundaries |
| Depending on concretes | Can't substitute | Depend on abstractions |

## Best Practices

- **Depend on abstractions** so any double can be injected.
- Prefer **fakes** (working, readable) for most logic; use **mocks** (`mocktail` / `mockito`) when you must verify interactions.
- Use **constructor injection** for pure unit tests (no framework); **Riverpod overrides** / `get_it.reset()` for wired tests.
- **Reset/dispose DI** between tests to prevent state bleed.
- Keep logic **UI-free** (view models/use cases/blocs) so it's testable without the widget tree.

## Performance

Unit tests with fakes are fast/deterministic (no I/O). Reset/dispose keeps tests isolated; this is the foundation for a fast test suite (Module 49).

## Advantages / Disadvantages

- **+** Fast, deterministic, isolated tests; scriptable scenarios (errors/edges); verifies your logic; enables TDD.
- **−** Requires abstraction-based design + doubles; over-mocking creates brittle tests; some DI test setup boilerplate.

## Interview Questions

1. 🟢 **How does DI enable testing?** — Consumers depend on abstractions, so tests inject fakes/mocks to run logic without real dependencies.
2. 🟢 **Fake vs mock?** — Fake: a working lightweight implementation (canned data); mock: a stub where you configure returns and verify calls.
3. 🟡 **How do you inject a fake with Riverpod?** — `ProviderContainer(overrides: [provider.overrideWithValue(fake)])` (or `ProviderScope` overrides in widget tests) — no widget tree needed for the container case.
4. 🟡 **How do you inject fakes with `get_it` in tests?** — `getIt.reset()` in `setUp`, then register the fakes so `getIt<T>()` returns them.
5. 🟡 **Why prefer fakes over mocks generally?** — Fakes are readable, reusable, and less brittle; mocks are best for verifying interactions at boundaries.
6. 🔴 **Why reset/dispose DI between tests?** — To prevent state/registration bleed that causes order-dependent, flaky tests.
7. 🔴 **What design makes logic testable without the widget tree?** — Keeping logic UI-free (view models/use cases/blocs) with injected abstractions.

## Senior Engineer Tips

- Write reusable **fakes** for your repositories/services; they double as documentation and speed up all tests.
- Use `mocktail` (no codegen) for quick interaction verification; reserve heavy mocking for true boundaries.
- Riverpod overrides are the cleanest DI-for-tests story — prefer them if you're on Riverpod.

## Architect Perspective

Testability is the primary architectural dividend of DI. Abstraction-based dependencies + a clear injection strategy (constructor/overrides/reset) enable a fast, deterministic test pyramid (Module 49) and TDD/BDD. Combined with UI-free logic layers (view models/use cases), it makes the whole system verifiable and refactor-safe — the hallmark of Clean Architecture (Module 40).

## Summary

- DI's payoff is testability: inject fakes/mocks via constructor, `get_it.reset()`, or Riverpod/Provider overrides.
- Prefer fakes (state-based) over mocks (interaction-based); reset/dispose DI between tests.
- Keep logic UI-free + abstraction-based so it's testable without real deps or the widget tree.

## Revision Notes

- Inject doubles: constructor (pure unit), `get_it.reset()` +register, Riverpod/Provider overrides.
- Fake (working impl) vs mock ( `mocktail` / `mockito` : stub + verify).
- Depend on abstractions; reset/dispose between tests; logic UI-free.
- Fast/deterministic/isolated → foundation of the test pyramid (Module 49).

## Practice Questions

1. When use a fake vs a mock?
2. How do you inject a fake with Riverpod vs `get_it` ?
3. Why reset DI between tests?

## Coding Questions

1. Test a `LoginViewModel` with a fake repo for success/invalid/error.
2. Rewrite one test using `mocktail` (stub + `verify` ).
3. Write a widget test wiring a Provider/Riverpod override to a fake.

## Mini Project — Module capstone

**Wired + tested slice (Flutter):** Wire `HttpClient → AuthRepository → LoginViewModel` with your chosen DI (get_it or Riverpod), register real impls at a composition root, and write a test suite that injects fakes (constructor + container/reset override) covering success/failure/error — plus one `mocktail` interaction test. Acceptance: consumers depend on abstractions; real wiring at root; fakes injected in tests; DI reset/disposed between tests; suite passes.