15 · Local Storage

Flutter Practice Notes

Contents

15 · Local Storage

Introduction

Apps must persist data on-device: user preferences, auth tokens, cached responses, files. This module covers the **non-database** local storage options — `SharedPreferences`, secure storage, and files (`path_provider`) — plus caching strategies. (Full databases are Module 20; offline sync is Module 19.)

Why this module exists

Choosing the wrong storage (tokens in plain prefs, large blobs in key-value, no cache invalidation) causes security holes, bloat, and stale data. Knowing each option's purpose, limits, and security model is essential for correct, safe persistence.

Module map

#	File	Topic	Level
1	01_storage_options_overview.md	The options and how to choose	🟢
2	02_shared_preferences.md	Key-value preferences	🟢
3	03_secure_storage.md	`flutter_secure_storage`, keychain/keystore	🔴
4	04_file_storage.md	`path_provider`, files & directories	🔵
5	05_caching_strategies.md	TTL, cache-first, invalidation	🔴

> **Cross-references:** Databases (SQLite/Drift/Isar/Hive): Module 20. Offline-first/sync: Module 19. Security (encryption, threat model): Module 37. Networking/caching source: Module 16. File handling (upload/download/ZIP): Module 34. Repository pattern: 05 · repository.

Prerequisites

02 · json_and_serialization, 05 · repository, 14 Dependency Injection.

What you'll be able to do after this module

- Pick the right storage for a given data type (prefs vs secure vs file vs DB).
- Persist simple settings with `SharedPreferences`.
- Store secrets/tokens securely (keychain/keystore) — never in plain prefs.
- Read/write files and app directories with `path_provider`.

- Implement caching with TTL, cache-first strategies, and invalidation.

Capstone

Persistent settings + secure session + cached feed: Store theme/locale in prefs, the auth token in secure storage, and API responses in a file-based cache with TTL — all behind a repository so the rest of the app is storage-agnostic.

Summary

Match data to storage: small settings → prefs; secrets → secure storage; files/blobs → filesystem; structured/queryable data → a database (Module 20). Wrap storage behind repositories, and cache with explicit TTL/invalidation.

Storage Options Overview & How to Choose

> Match the data to the mechanism: **small key-value settings → SharedPreferences**, **secrets/tokens → secure storage**, **files/blobs → filesystem**, **structured/queryable data → a database** — using the wrong one causes security holes, bloat, or stale data.

Introduction

Flutter offers several on-device persistence options with distinct purposes and limits. This file is the decision framework so the detailed files (prefs/secure/files/cache) and the database module (20) are applied correctly.

Why this concept exists

Beginners default to `SharedPreferences` for everything — storing tokens (insecure), large JSON (bloat), or relational data (unqueryable). Each mechanism exists for a reason; choosing correctly is a correctness, security, and performance decision.

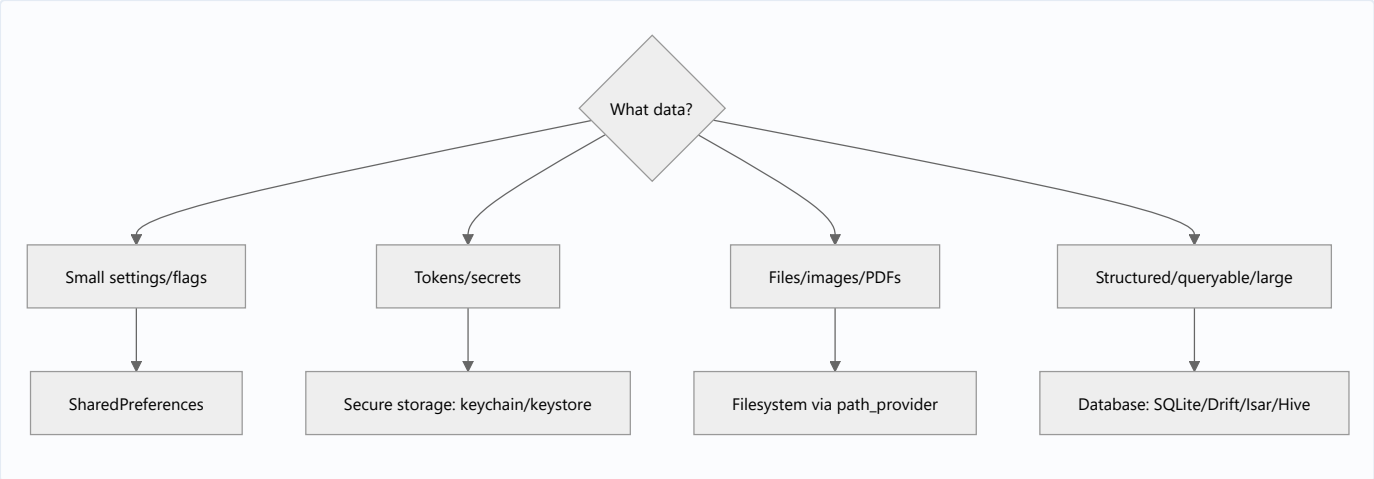
Real-world analogy

Storage options are like **places to keep belongings**: a sticky note for a quick reminder (prefs), a safe for valuables (secure storage), a filing cabinet for documents (files), and a library with an index for lots of searchable records (database). You don't keep cash on a sticky note or a library in a drawer.

Problem Statement

You must persist: theme choice, an auth token, a downloaded PDF, and 5,000 searchable products. Which storage each? By the end you'll route each correctly.

Internal Working



Option	Best for	Security	Capacity	Query	Module
SharedPreferences	Small key-value settings/flags	❌ plain text	Small	No	02_shared_preferences.md
Secure storage	Tokens, secrets, credentials	✅ keychain/keystore	Small	No	03_secure_storage.md
Files (`path_provider`)	Blobs, images, PDFs, exports, cache files	Optional (encrypt yourself)	Large	No	04_file_storage.md
Database	Structured, queryable, relational, large	Optional (encrypt)	Large	✅	Module 20

Decision heuristics:

- **Is it a secret?** → secure storage (never prefs).
- **Is it a small flag/setting?** → prefs.
- **Is it a file/large blob?** → filesystem.
- **Is it structured/queryable/lots of records?** → database.
- **Is it just a cache of network data?** → file/db cache with TTL (05_caching_strategies.md); consider offline-first (Module 19).

Memory Representation

Prefs/secure store small values (loaded into memory); files/db hold larger data on disk read on demand. Wrap access so the app holds only what it needs (02 · memory_and_gc).

Compiler Behavior / Runtime Behavior

All are async plugin APIs (platform channels); most are `Future`-based and need `WidgetsFlutterBinding.ensureInitialized()` before use in `main` (06 · app_entry_point).

Flutter Engine Behavior

Storage plugins cross the embedder boundary to native storage (UserDefaults/SharedPreferences, Keychain/Keystore, filesystem) (10 · embedder_and_startup).

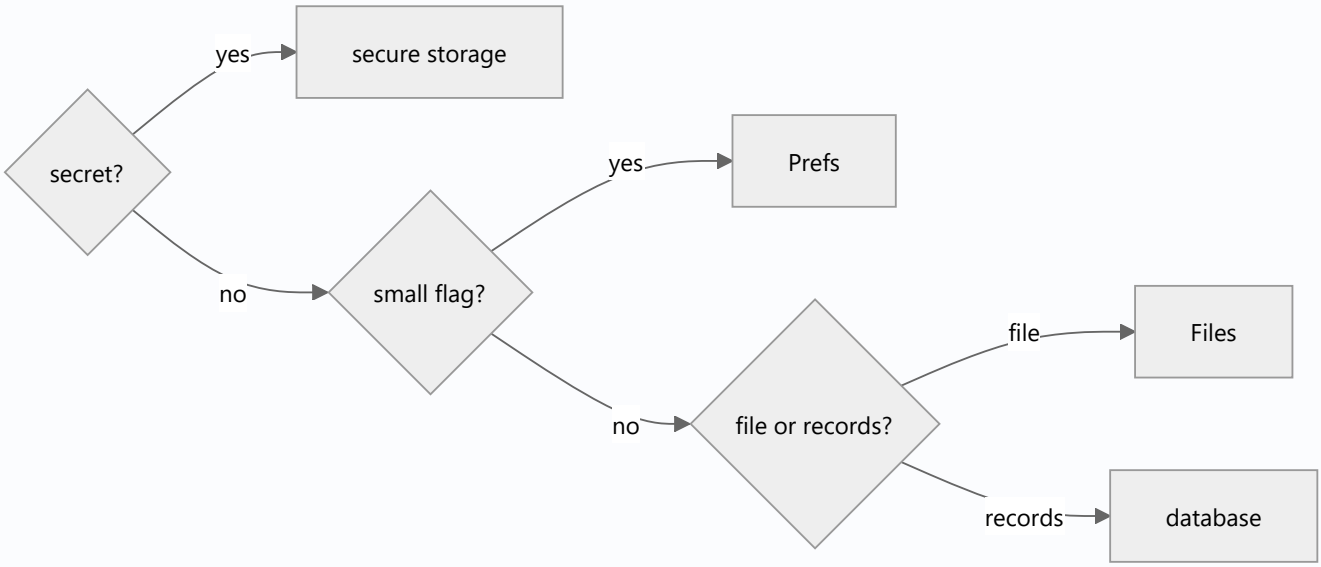
Dart VM Behavior

Not applicable.

Examples

```
// Routing data to storage (mental model):
// - themeMode: 'dark'            -> SharedPreferences (small flag)
// - authToken: 'eyJ...'          -> secure storage (secret!)
// - invoice.pdf (2MB)            -> filesystem (path_provider)
// - 5000 products (searchable)   -> database (Module 20)
// - cached /feed JSON            -> file/db cache with TTL (caching_strategies.md)
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Tokens/secrets in `SharedPreferences`	Plain text, readable	Use secure storage
Large JSON/blobs in prefs	Bloat, slow, size limits	Files or database
Relational/queryable data in prefs/files	No querying	Use a database (Module 20)
Cache without TTL/invalidation	Stale data	TTL + invalidation (05_caching_strategies.md)
Storage calls scattered in UI	Coupling, untestable	Wrap behind a repository (05 · repository)

Best Practices

- **Match data to mechanism** (secret→secure, flag→prefs, blob→file, records→db).
- **Never** store secrets in prefs; use secure storage.
- Wrap all storage behind **repositories/abstractions** so the app is storage-agnostic and testable (14 DI).
- Cache network data with explicit **TTL/invalidation**; consider offline-first for full sync (Module 19).
- Encrypt sensitive files/db as needed (Module 37).

Performance

Prefs/secure are small/fast; files/db scale but need on-demand reads and (for large data) off-isolate parsing (02 · isolates). Right-sizing avoids memory/latency issues.

Advantages / Disadvantages

- **+** A clear framework prevents insecure/bloated/unqueryable storage choices.
- **−** Multiple mechanisms to learn; the "right" choice is contextual.

Interview Questions

1. 🟢 **Where should you store an auth token?** — Secure storage (keychain/keystore) — never `SharedPreferences` (plain text).
2. 🟢 **What's `SharedPreferences` for?** — Small key-value settings/flags (theme, onboarding-seen), not secrets or large/structured data.
3. 🟡 **When files vs a database?** — Files for blobs/documents/exports/cache files; a database for structured, queryable, relational, or large record sets.
4. 🟡 **Why wrap storage behind a repository?** — To keep the app storage-agnostic, swappable, and testable (inject fakes), per DIP.

5. 🟡 **What's wrong with caching without TTL?** — Data goes stale with no expiry/invalidation; add TTL + invalidation strategy.
6. 🔴 **How do you decide storage for cached network data?** — By size/structure: small→prefs is wrong (no expiry logic), typically file/db cache with TTL, or offline-first sync for full offline support.
7. 🔴 **What are the security implications of the choice?** — Prefs/files are readable unless encrypted; secrets belong in OS secure storage; sensitive files/db should be encrypted (Module 37).

Senior Engineer Tips

- Treat "should this be secure?" as the first question — getting tokens/PII into secure storage prevents a common vulnerability.
- Abstract storage behind repositories from day one; it makes migrating prefs→db or adding encryption painless.
- For anything beyond a handful of key-values, reach for a database rather than stuffing JSON into prefs/files.

Architect Perspective

Storage strategy is a cross-cutting decision spanning security, performance, and offline capability. A repository-fronted, mechanism-appropriate design (secure for secrets, db for records, files for blobs, cache with TTL) keeps the app secure, fast, and swappable — and sets up offline-first (Module 19) and encryption (Module 37) cleanly.

Summary

- Options: prefs (small settings), secure storage (secrets), files (blobs), database (structured/queryable).
- Never store secrets in prefs; wrap storage behind repositories; cache with TTL/invalidation.
- Match data to mechanism; escalate to a database for records and offline-first for sync.

Revision Notes

- Secret→secure storage; small flag→prefs; blob/file→filesystem; records/queryable→database.
- Prefs/files are plain text unless encrypted; secrets never in prefs.
- Wrap behind repositories (testable/swappable); cache with TTL + invalidation.
- DBs = Module 20; sync = Module 19; encryption = Module 37.

Practice Questions

1. Route four data items to the correct storage with justification.
2. Why is a token in prefs a vulnerability?
3. When do you need a database over files/prefs?

Coding Questions

1. Design a `Storage` abstraction with prefs/secure/file backends and route calls appropriately.
2. Identify and fix a codebase that stores a token in prefs.
3. Sketch a cache decision (file vs db) for a paginated feed.

Mini Project

Storage router (docs + skeleton): For a described app, write `STORAGE.md` mapping each data item to its mechanism (with rationale + security notes), and sketch a repository interface fronting prefs/secure/file. Acceptance: correct, secure choices; repository abstraction; no secrets in prefs.

`SharedPreferences`

> `SharedPreferences` is a simple async key-value store for **small, non-sensitive settings** (theme, onboarding-seen, last tab) — backed by the platform's native prefs; never use it for secrets or large/structured data.

Introduction

`SharedPreferences` (package: `shared_preferences`) persists primitive key-value pairs (`bool`, `int`, `double`, `String`, `List<String>`) across app launches, wrapping platform storage (Android SharedPreferences, iOS UserDefaults). This file covers its API, correct use, and its (important) limits.

Why this concept exists

Apps need to remember tiny bits of state — the chosen theme, whether onboarding was seen, a feature flag. A full database is overkill; prefs give a trivial, cross-platform key-value API for exactly this.

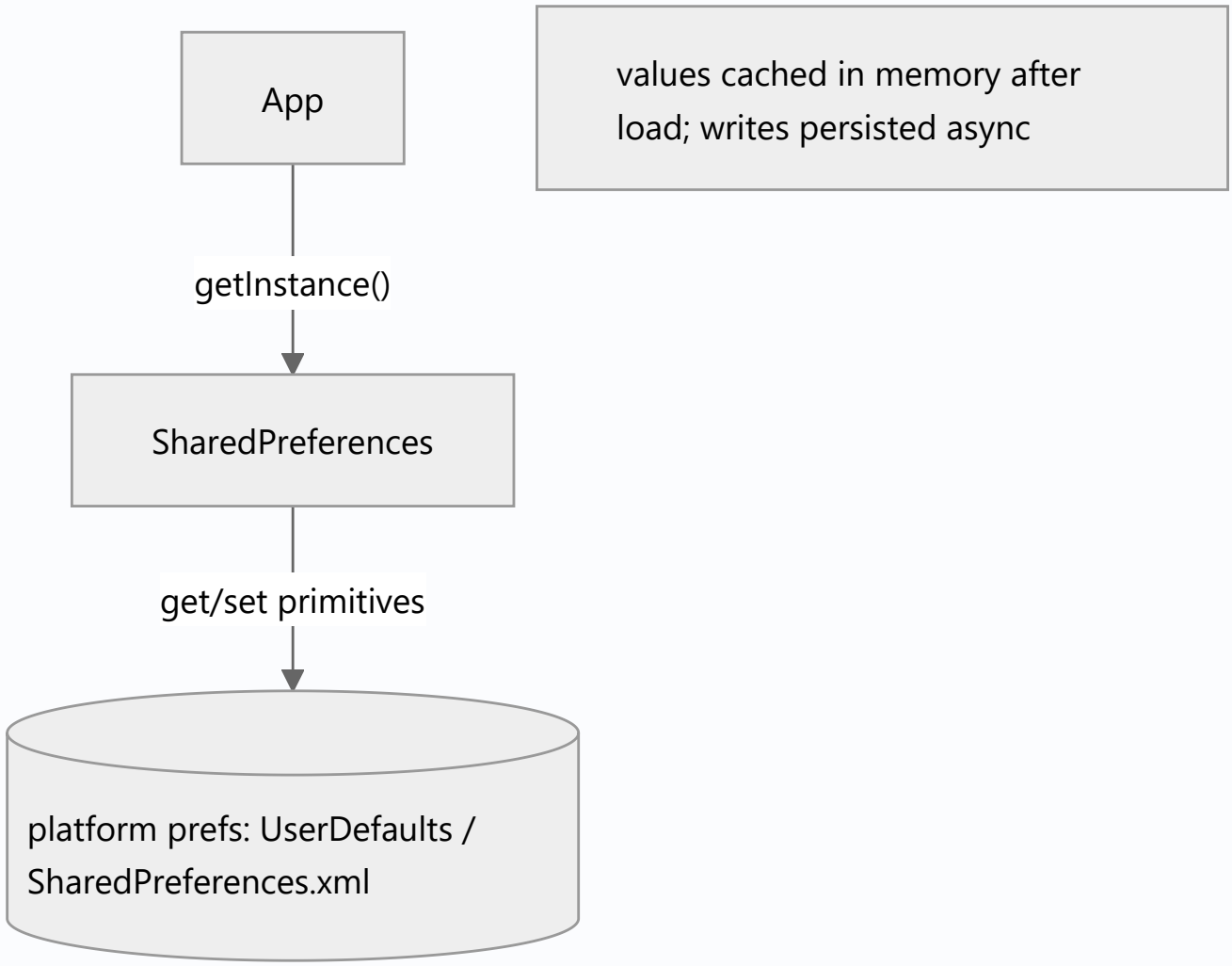
Real-world analogy

A **whiteboard by the door** where you jot quick reminders ("lights off," "buy milk"). Perfect for small notes you check on the way in; you wouldn't store your passport (secret) or filing cabinet (records) there.

Problem Statement

Persist the user's `themeMode` and an `onboardingSeen` flag across launches, read them at startup, and expose them behind a repository. You'll use `SharedPreferences` correctly and see why tokens/large data don't belong here.

Internal Working



- **Get instance**: `final prefs = await SharedPreferences.getInstance();` (async; often initialized once at startup).
- **Read/write**: `prefs.getString('key')` / `await prefs.setString('key', v)`; supported types: `bool` / `int` / `double` / `String` / `List<String>`.
- **Absent keys** return `null` (typed getters return `null` if unset) — provide defaults.
- **Remove/clear**: `prefs.remove('key')`, `prefs.clear()`.
- **Not for**: secrets (plain text — use secure storage), large/complex data (use files/db), or anything needing queries.
- Values are **cached in memory** after first load; writes are persisted asynchronously to native storage.

Memory Representation

All prefs are loaded into an in-memory map on first access; keep the dataset **small**. Native backing is a plist/XML file (readable on a rooted/jailbroken device) — hence no secrets (03_secure_storage.md).

Compiler Behavior

Not applicable. Typed getters return nullable values.

Runtime Behavior

`getInstance()` loads the store; getters read the in-memory cache (sync after load); setters update cache + persist async. Needs `WidgetsFlutterBinding.ensureInitialized()` if used before `runApp`.

Flutter Engine Behavior

Crosses the embedder to native prefs storage (10 · embedder_and_startup).

Dart VM Behavior

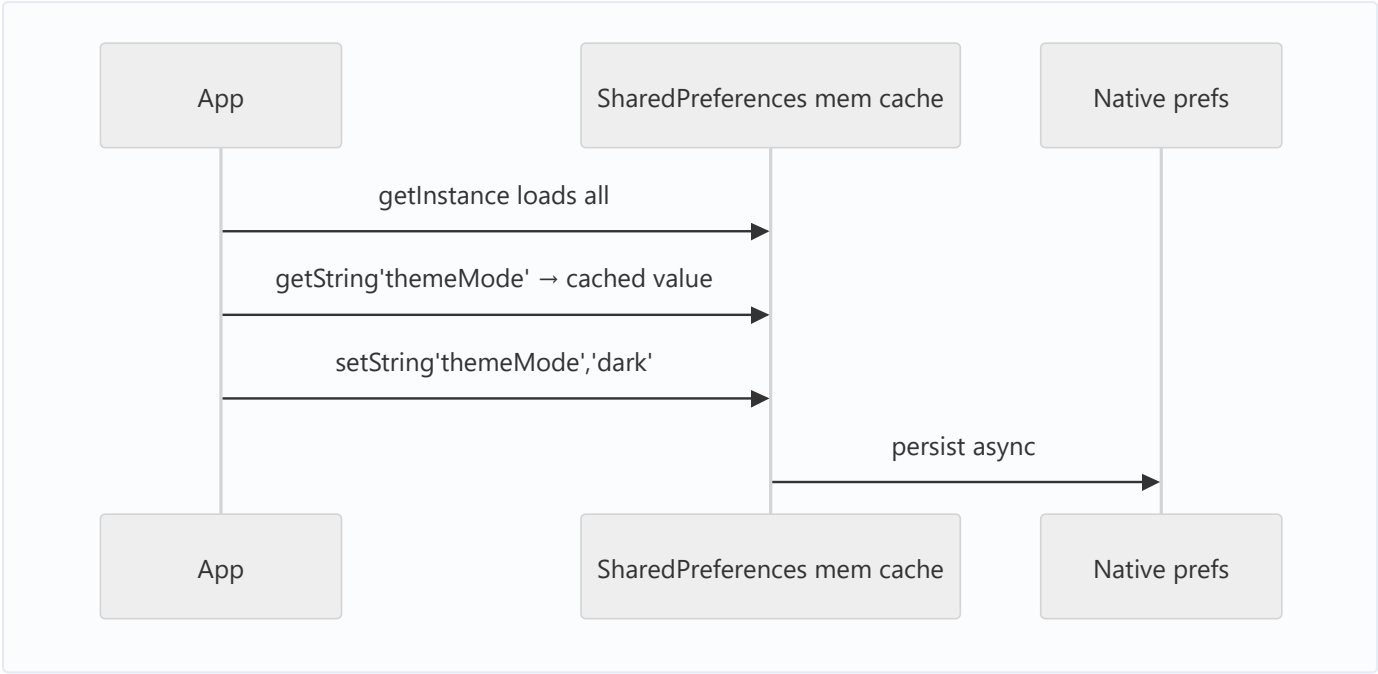
Not applicable.

Examples

```
# pubspec.yaml
dependencies:
  shared_preferences: ^2.2.0
```

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

// Wrap prefs behind a repository (storage-agnostic app, testable)
class SettingsRepository {
  final SharedPreferences _prefs;
  SettingsRepository(this._prefs);

  static const _kTheme = 'themeMode';
  static const _kOnboarded = 'onboardingSeen';

  String get themeMode => _prefs.getString(_kTheme) ?? 'system'; // default!
  Future<void> setThemeMode(String v) => _prefs.setString(_kTheme, v);

  bool get onboardingSeen => _prefs.getBool(_kOnboarded) ?? false;
  Future<void> setOnboardingSeen(bool v) => _prefs.setBool(_kOnboarded, v);
}

Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();          // needed before async in main
  final prefs = await SharedPreferences.getInstance(); // load once at startup
  final settings = SettingsRepository(prefs);

  print(settings.themeMode);        // 'system' first run
  await settings.setThemeMode('dark');
  await settings.setOnboardingSeen(true);
  print(settings.themeMode);        // 'dark' (persists next launch)
}

// (main uses ensureInitialized; import flutter/widgets in a real app)
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Storing tokens/secrets	Plain text, readable	Secure storage (03_secure_storage.md)
Storing large JSON/blobs	Loaded into memory, size limits	Files/database
No default for absent keys	`null` surprises	`?? default`
Assuming instant persistence	Writes are async	`await` sets when ordering matters
Prefs calls scattered in UI	Coupling/untestable	Wrap in a repository
Manual JSON in a single pref key	Unqueryable, error-prone	Use a database for structured data

Best Practices

- Use only for **small, non-sensitive** settings/flags.
- Provide **defaults** for absent keys; centralize **key constants**.
- Wrap behind a **repository** (inject `SharedPreferences`) for testability/swap-ability (14 DI).
- Initialize once at startup (`getInstance()`); `ensureInitialized()` before `runApp` if needed.
- Escalate to **secure storage** (secrets), **files** (blobs), or a **database** (structured) when outgrown.

Performance

Fast (in-memory after load); keep the store small since everything loads at once. Writes persist async — negligible cost for small values (Module 21).

Advantages / Disadvantages

- + Trivial cross-platform key-value API, fast reads (cached), perfect for small settings.
- − Plain text (no secrets), primitives only, no queries, loads everything into memory (not for large/structured data).

Interview Questions

1. 🟢 **What is `SharedPreferences` for?** — Persisting small key-value settings/flags (primitives) across launches.
2. 🟢 **What types can it store?** — `bool`, `int`, `double`, `String`, `List<String>`.
3. 🟡 **Why not store tokens in it?** — It's plain text (readable on a compromised device); use OS secure storage instead.
4. 🟡 **What happens for an absent key?** — Typed getters return `null`; always provide a default (`?? x`).
5. 🟡 **Is it synchronous?** — `getInstance()` is async (loads the store); after that, getters read an in-memory cache (effectively sync), writes persist async.
6. 🔴 **Why keep the prefs dataset small?** — All entries load into memory on first access; large/complex data bloats memory and hits size limits — use files/db.
7. 🔴 **How do you make prefs testable?** — Wrap behind a repository and inject `SharedPreferences` (use `setMockInitialValues` or a fake in tests).

Senior Engineer Tips

- Centralize keys as constants and expose a typed `SettingsRepository`; avoid raw `prefs.getX('literal')` across the app.
- In tests, `SharedPreferences.setMockInitialValues({})` gives a fake store — no platform needed.
- The moment you're serializing complex/growing JSON into a pref, switch to a database (Module 20).

Architect Perspective

`SharedPreferences` is the right tool for a narrow job: small, non-sensitive app settings. Fronting it with a repository keeps the app storage-agnostic (easy to migrate to a DB or add encryption) and testable — consistent with the storage-abstraction strategy (01_storage_options_overview.md, 05 · repository).

Summary

- `SharedPreferences`: small key-value primitives for non-sensitive settings, cached in memory, persisted async.
- Not for secrets (secure storage), large blobs (files), or structured data (database).
- Provide defaults, centralize keys, wrap in a repository, initialize at startup.

Revision Notes

- Small non-sensitive settings; types: bool/int/double/String/List.
- `getInstance()` async (loads all → mem cache); getters sync-ish, sets async; absent key → null (default it).
- NO secrets (plain text) / large / structured data.
- Wrap in repository (inject; `setMockInitialValues` in tests); centralize keys.

Practice Questions

1. Why provide a default for `getString`?
2. Why are tokens unsafe in prefs?
3. When do you outgrow prefs?

Coding Questions

1. Build a `SettingsRepository` (theme + onboarding flag) over injected `SharedPreferences`.
2. Test it using `setMockInitialValues` (no platform).
3. Migrate a "big JSON in one pref key" hack to a proper store.

Mini Project

Settings persistence (Flutter): Build a `SettingsRepository` (themeMode, onboardingSeen, lastTab) over `SharedPreferences`, with defaults and centralized keys, wired via DI, and a widget that reads/updates it. Add a unit test with mock initial values. Acceptance: small non-sensitive data only; defaults handled; repository-fronted; test passes; app runs.

Secure Storage (`flutter_secure_storage`, Keychain/Keystore)

> Store secrets — auth tokens, refresh tokens, credentials, encryption keys — in the OS-backed secure store (`flutter_secure_storage`), which uses the iOS **Keychain** and Android **Keystore/EncryptedSharedPreferences**, so data is encrypted at rest and not readable like plain prefs.

Introduction

`flutter_secure_storage` (package: `flutter_secure_storage`) provides a key-value API like prefs, but backed by platform secure enclaves: iOS **Keychain**, Android **Keystore**-encrypted storage. This file covers its API, the security model, and correct handling of tokens/secrets.

Why this concept exists

`SharedPreferences`/files are plain text — readable on rooted/jailbroken devices, backups, or via forensic tools. Secrets need OS-level encryption tied to the device/app. Secure storage delegates to the platform's hardened stores so your secrets aren't sitting in the clear.

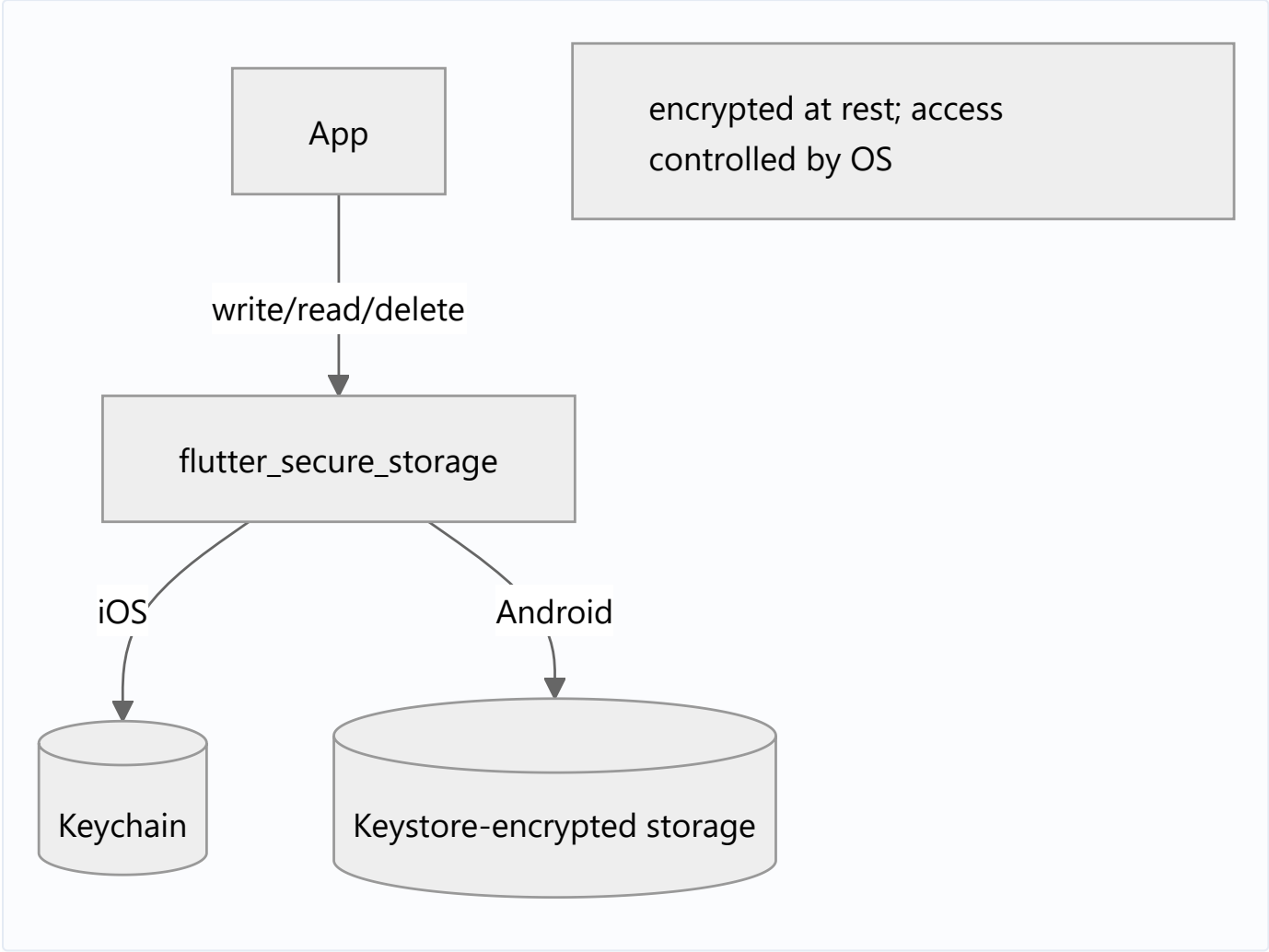
Real-world analogy

A **bank safe deposit box** vs a drawer: prefs are the drawer (anyone with access can read it); secure storage is the safe deposit box — locked by the bank (OS), tied to your identity (app/device), and far harder to break into.

Problem Statement

Store an auth token + refresh token securely, read them to authorize requests, and clear them on logout — never touching plain prefs. You'll use `flutter_secure_storage` behind a repository.

Internal Working



- **API**: `const storage = FlutterSecureStorage();` → `await storage.write(key:, value:)`, `read(key:)` (returns `String?`), `delete(key:)`, `deleteAll()`, `readAll()`.
- **Backing**: iOS **Keychain** (with configurable accessibility, e.g., `first_unlock`), Android **Keystore**-based encryption (EncryptedSharedPreferences).
- **All async**; values are `String` (serialize objects to JSON if needed — but avoid storing large data here).
- **Platform options**: iOS accessibility (`IOSOptions`), Android options (`AndroidOptions(encryptedSharedPreferences: true)`), biometric-gated access on some platforms.
- **Limits**: for secrets/credentials/keys only — not large data; secure storage can be slower than prefs and has platform quirks (e.g., Android backup/keystore edge cases).

Memory Representation

Secrets are encrypted at rest in the OS store; when read into your Dart heap they're plain `String`s — minimize their in-memory lifetime and don't log them (Module 37).

Compiler Behavior

Not applicable.

Runtime Behavior

Reads/writes go through native secure APIs (slower than prefs). `read` returns `null` if absent. Some configurations require device unlock/biometrics for access.

Flutter Engine Behavior

Crosses the embedder to native Keychain/Keystore APIs (10 · embedder_and_startup).

Dart VM Behavior

Not applicable.

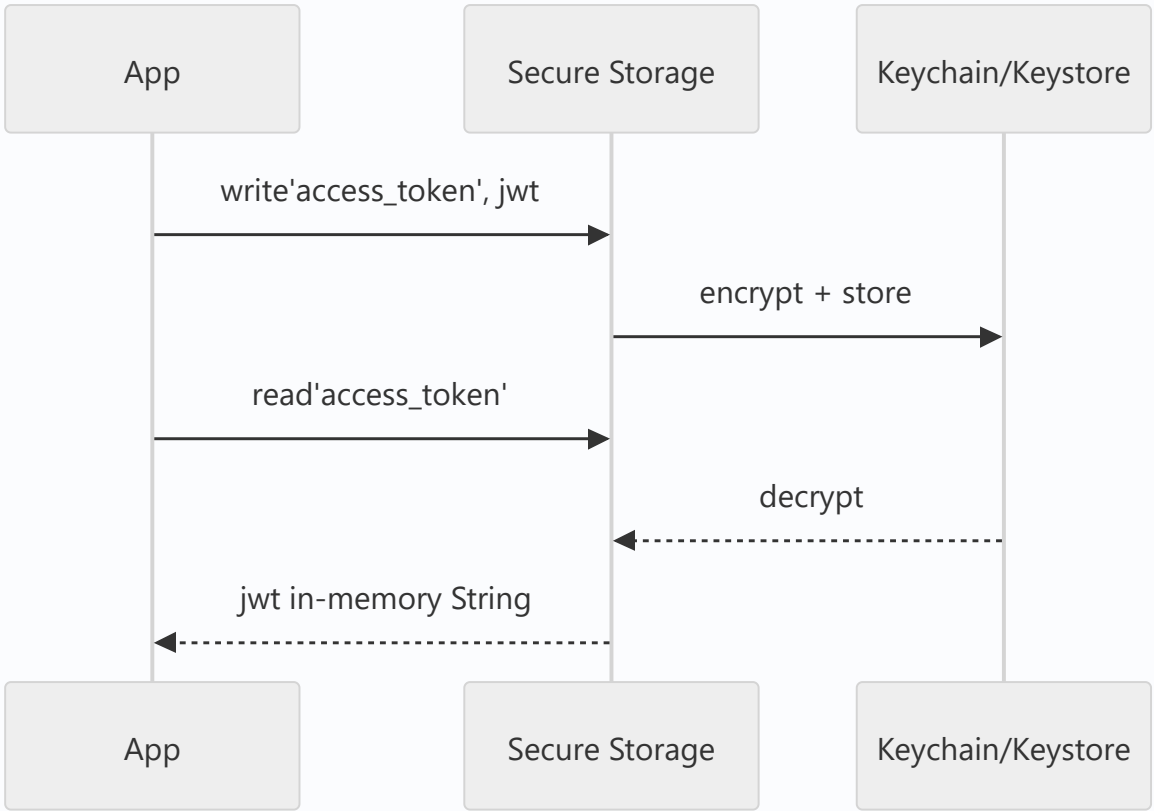
Examples

```
# pubspec.yaml
dependencies:
  flutter_secure_storage: ^9.0.0
```

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

// Repository fronting secure storage (tokens only)
class TokenStore {
  final FlutterSecureStorage _storage;
  TokenStore([FlutterSecureStorage? s])
      : _storage = s ??
            const FlutterSecureStorage(
              aOptions: AndroidOptions(encryptedSharedPreferences: true),
              iOptions: IOSOptions(accessibility: KeychainAccessibility.first_unlock),
            );

  static const _kAccess = 'access_token';
  static const _kRefresh = 'refresh_token';

  Future<void> saveTokens({required String access, required String refresh}) async {
    await _storage.write(key: _kAccess, value: access);   // encrypted at rest
    await _storage.write(key: _kRefresh, value: refresh);
```

```
  }

  Future<String?> get accessToken => _storage.read(key: _kAccess);

  Future<void> clear() async {
    await _storage.delete(key: _kAccess);   // on logout
    await _storage.delete(key: _kRefresh);
  }
}

Future<void> demo() async {
  final store = TokenStore();
  await store.saveTokens(access: 'eyJ...', refresh: 'r...');
  final token = await store.accessToken; // read for Authorization header
  print(token != null ? 'have token' : 'none');
  await store.clear(); // logout
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Tokens/secrets in prefs/files	Plain text, readable	Use secure storage
Storing large data here	Slow, not designed for it	Secure storage = secrets only
Logging secret values	Leaks to logs/crash reports	Never log tokens; redact
Assuming it's unbreakable	Rooted devices/OS bugs exist	Defense-in-depth; short-lived tokens, server checks
Ignoring platform options	Wrong accessibility/backup behavior	Configure iOS/Android options

Best Practices

- Store **only secrets** (tokens, refresh tokens, keys, credentials) — small string values.
- Front it with a **repository** (`TokenStore`); inject for testability (14 DI).
- Configure platform options (iOS accessibility, Android encrypted prefs); consider biometric gating for high-value secrets.
- **Clear tokens on logout**; use short-lived access tokens + refresh flow (Module 17).
- Never log/serialize secrets into analytics/crash reports; minimize their in-memory lifetime (Module 37).

Performance

Slower than prefs (native crypto/enclave calls) — fine for occasional token reads; don't use it as a general cache. Read once and hold briefly rather than per-request if hot (Module 21).

Advantages / Disadvantages

- **+** OS-backed encryption at rest, right home for secrets, simple key-value API, platform-hardened.
- **−** Secrets only (small), slower than prefs, platform quirks/edge cases, not a silver bullet (rooted devices).

Interview Questions

1. 🟢 **What is secure storage for?** — Persisting secrets (tokens, credentials, keys) encrypted at rest via the OS (Keychain/Keystore), unlike plain prefs.
2. 🟢 **What backs `flutter_secure_storage` per platform?** — iOS Keychain; Android Keystore-based encryption (EncryptedSharedPreferences).
3. 🟡 **Why not store tokens in `SharedPreferences`?** — Prefs are plain text and readable on compromised devices/backups; secrets need OS encryption.

4. 🟡 **What should you store in secure storage?** — Only small secrets; not large data (slow, not designed for it).
5. 🟡 **How do you handle logout?** — Delete the tokens (`delete` / `deleteAll`) so they don't persist.
6. 🔴 **Is secure storage unbreakable?** — No; rooted/jailbroken devices and OS bugs exist. Use defense-in-depth: short-lived tokens, server-side validation, and never log secrets.
7. 🔴 **What platform options matter?** — iOS Keychain accessibility (e.g., `first_unlock`) and Android `encryptedSharedPreferences`/backup behavior; biometric gating for sensitive secrets.

Senior Engineer Tips

- Keep a dedicated `TokenStore` / `SecretStore` repository — the only code touching secure storage — so security handling is centralized and auditable.
- Pair with short-lived access tokens + refresh; don't rely on device storage as your only defense.
- Audit that no secret is ever logged, put in analytics, or written to plain files/prefs.

Architect Perspective

Secure storage is a security-critical boundary. Centralizing secret handling behind a repository, configuring platform options, and combining with short-lived tokens + server validation forms a defense-in-depth strategy integral to auth and security architecture (Modules 17, 37).

Summary

- Store secrets in OS-backed secure storage (Keychain/Keystore) — never plain prefs/files.
- Small string secrets only; front with a repository; clear on logout; configure platform options.
- Not unbreakable — combine with short-lived tokens, server checks, and no-logging discipline.

Revision Notes

- Secrets → `flutter_secure_storage` (iOS Keychain / Android Keystore, encrypted at rest).
- API: `write/read/delete/deleteAll` (async, String values); secrets only (small).
- Never in prefs/files/logs; clear on logout; configure iOS/Android options.
- Defense-in-depth: short-lived tokens + server validation (rooted devices exist).

Practice Questions

1. Where do auth tokens belong and why?
2. What backs secure storage on iOS vs Android?

3. Why is secure storage not a complete security solution?

Coding Questions

1. Build a `TokenStore` (save/read/clear) over `flutter_secure_storage`.
2. Wire logout to clear tokens; verify a subsequent read is `null`.
3. Configure iOS accessibility + Android encrypted prefs options.

Mini Project

Secure session (Flutter): Build a `TokenStore` repository over secure storage that saves access/refresh tokens on login, exposes the access token for requests, and clears on logout — with platform options configured and no logging of secrets. Acceptance: secrets only in secure storage; cleared on logout; no secret logging; app runs.

File Storage (`path_provider`, Files & Directories)

> Use `path_provider` to get the correct platform directories, then `dart:io`'s `File` / `Directory` to read/write blobs — documents, images, exports, and cache files — choosing the right directory for persistence vs cache and never hardcoding paths.

Introduction

For data too big or unstructured for prefs (PDFs, images, downloaded files, JSON caches, exports), use the filesystem. `path_provider` (package) gives platform-correct directories; `dart:io` handles the actual read/write. This file covers directories, file operations, and the persistence-vs-cache distinction.

Why this concept exists

Apps can't hardcode filesystem paths (they differ per OS/sandbox). `path_provider` returns the sandboxed app directories (documents, support, temporary/cache) so you write to valid, correctly-treated locations (e.g., cache dirs the OS may purge).

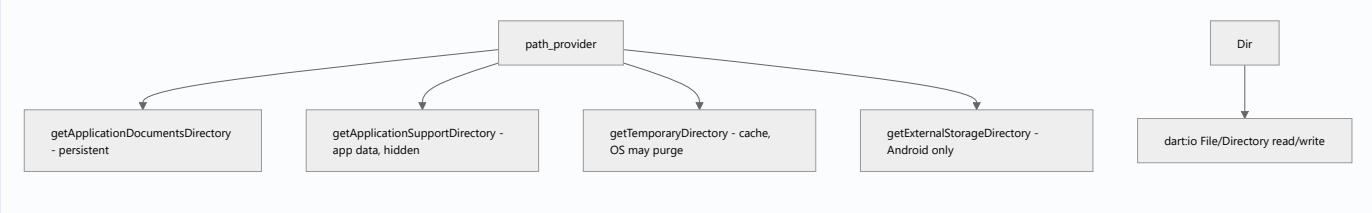
Real-world analogy

`path_provider` is a **building directory that tells you which room is which**: the permanent archive (documents), the staff-only back office (app support), and the recycling bin the janitor empties (temporary/cache). You store each thing in the appropriate room.

Problem Statement

Save a downloaded invoice PDF that must persist, cache an API JSON response that can be purged, and export a CSV the user keeps. You'll pick directories via `path_provider` and read/write with `dart:io`.

Internal Working



- **Directories** (via `path_provider`):

    - `getApplicationDocumentsDirectory()` — **persistent** user data (backed up); for files that must survive.
    - `getApplicationSupportDirectory()` — app-managed data, not user-visible.
    - `getTemporaryDirectory()` — **cache**; the OS may delete it anytime; for regenerable data.
    - `getExternalStorageDirectory()` (Android) / downloads dirs — platform-specific, permissions may apply.
- **File ops** (`dart:io`): `File('$dir/name').writeAsString/Bytes(...)`, `readAsString/Bytes()`, `exists()`, `delete()`; `Directory(...).create(recursive: true)`, `list()`.
- **Path building**: join with `package:path`'s `join(dir, 'sub', 'file.txt')` — never string-concat with `/`.
- **Large files/heavy parsing**: read/parse off the UI isolate (`Isolate.run`/streams) to avoid jank (02 · isolates).

Memory Representation

Files live on disk; `readAsBytes`/`readAsString` load into memory (mind large files — stream them). Cache dirs are OS-purgeable, so never store must-keep data there (02 · memory_and_gc).

Compiler Behavior

Not applicable. `dart:io` is unavailable on web — use conditional imports / web-specific storage for web (Module 53).

Runtime Behavior

File I/O is async (`Future`); large reads/writes should be chunked/streamed and possibly off-isolate. Missing files throw on read — check `exists()` or handle errors.

Flutter Engine Behavior

`path_provider` crosses the embedder to native path APIs; file I/O uses the IO task runner conceptually (10 · threading_model).

Dart VM Behavior

Not applicable beyond `dart:io`.

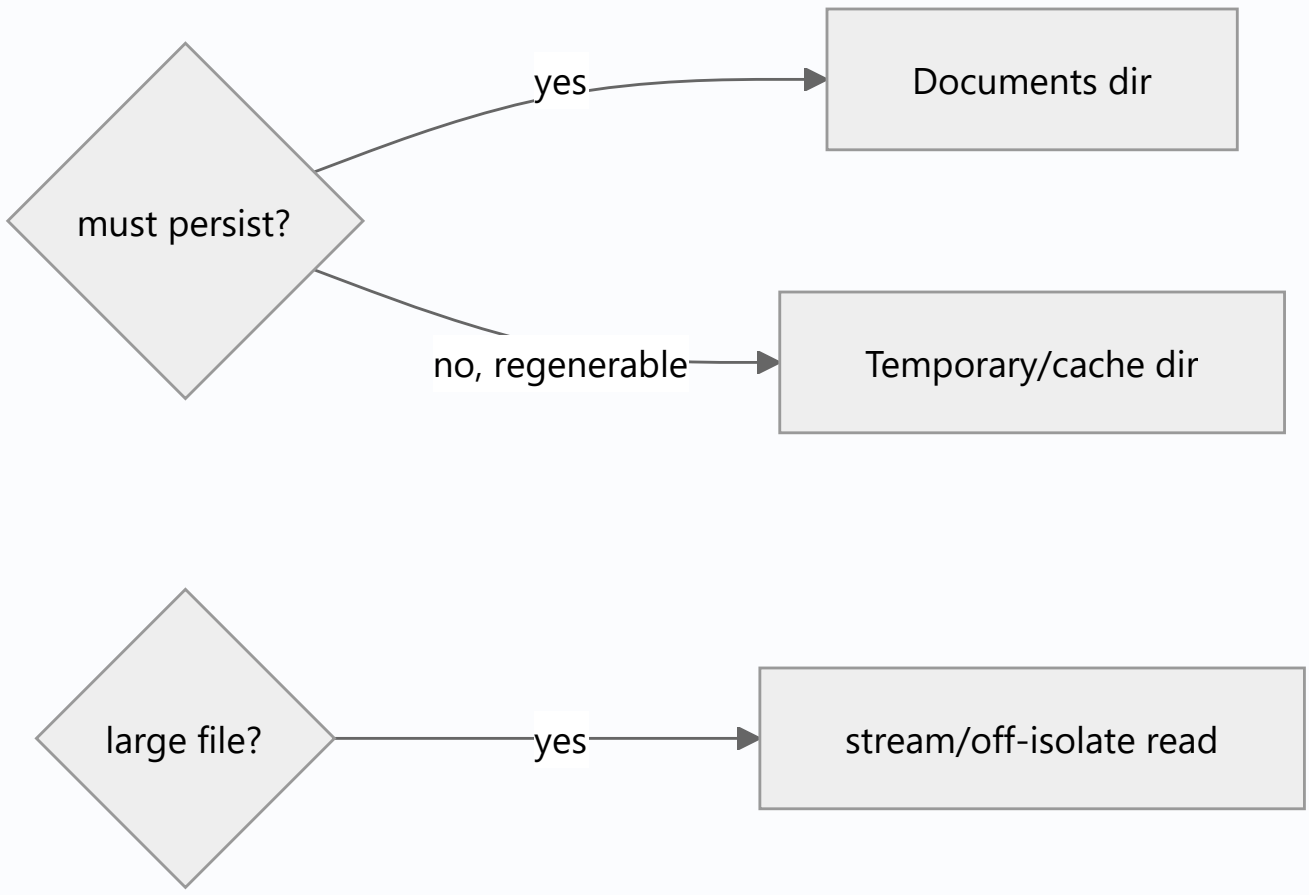
Examples

```yaml
# pubspec.yaml
dependencies:
  path_provider: ^2.1.0
  path: ^1.9.0
```

```dart
import 'dart:convert';
import 'dart:io';
import 'package:path/path.dart' as p;
import 'package:path_provider/path_provider.dart';

class FileStore {
  // Persistent file (survives; backed up)
  Future<File> _docFile(String name) async {
    final dir = await getApplicationDocumentsDirectory();
    return File(p.join(dir.path, name)); // use path.join, not string concat
  }

  // Cache file (OS may purge)
  Future<File> _cacheFile(String name) async {
    final dir = await getTemporaryDirectory();
    return File(p.join(dir.path, name));
  }

  Future<void> savePdf(String name, List<int> bytes) async {
    final f = await _docFile(name);
    await f.writeAsBytes(bytes); // persistent
  }

  Future<void> cacheJson(String name, Map<String, dynamic> json) async {
    final f = await _cacheFile(name);
    await f.writeAsString(jsonEncode(json)); // regenerable -> cache dir
  }

  Future<Map<String, dynamic>?> readCachedJson(String name) async {
    final f = await _cacheFile(name);
    if (!await f.exists()) return null;
    return jsonDecode(await f.readAsString()) as Map<String, dynamic>;
  }
```

```
}


Future<void> demo() async {
  final store = FileStore();
  await store.cacheJson('feed.json', {'items': [1, 2, 3]});
  print(await store.readCachedJson('feed.json')); // {items: [1,2,3]}
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Hardcoding paths	Wrong/invalid per platform	Use `path_provider` dirs + `path.join`
Persistent data in temporary dir	OS purges it	Use documents/support dir
String-concat paths (`dir+'/'+name`)	Breaks cross-platform	`p.join(...)`
Reading huge files on UI isolate	Jank	Stream/ `Isolate.run` (02)
Assuming `dart:io` on web	Unavailable	Conditional imports / web storage (Module 53)
No `exists()` /error handling	Crash on missing file	Check exists / try-catch

Best Practices

- Choose the **right directory**: documents/support for persistent, temporary for cache (regenerable).
- Build paths with `path.join`; never hardcode/concat.
- **Stream or off-isolate** large reads/writes/parsing to avoid jank.
- Front file access behind a **repository/** `FileStore`; handle missing files.
- On web, provide an alternative (`dart:io` unavailable); consider `path_provider` web support limits.
- Encrypt sensitive files if needed (Module 37); use files for cache with TTL (05_caching_strategies.md).

Performance

Disk I/O is async; large operations should stream/off-load to keep frames smooth. Cache dirs avoid backup bloat for regenerable data (Module 21).

Advantages / Disadvantages

- + Large/blob storage, full control, exports/documents, cache files, streaming for big data.
- − No querying, manual serialization, platform/web caveats, large-file/jank pitfalls, must pick correct dir.

Interview Questions

1. 🟢 **Why use** `path_provider`**?** — To get platform-correct, sandboxed app directories instead of hardcoding paths.
2. 🟢 **Documents vs temporary directory?** — Documents = persistent (backed up); temporary/cache = OS may purge (regenerable data only).

3. 🟡 **How do you build file paths correctly?** — With `package:path`'s `join`, not string concatenation.
4. 🟡 **How do you avoid jank with large files?** — Stream them and/or parse on a background isolate (`Isolate.run`).
5. 🟡 **What's different on web?** — `dart:io` File isn't available; use conditional imports / web-appropriate storage.
6. 🔴 **Where do you store a cache vs a must-keep document?** — Cache → temporary dir (purgeable); must-keep → documents/support dir (persistent).
7. 🔴 **How do you make file storage testable?** — Front it behind a repository and inject a directory/file-system abstraction (or a temp dir in tests).

Senior Engineer Tips

- Use the **temporary/cache directory** for anything regenerable (network caches, thumbnails) so it doesn't bloat backups and can be purged.
- Offload large JSON/image processing to isolates; file reads on the UI isolate are a common jank source.
- Centralize file access in a `FileStore` repository with error handling; it also eases web/desktop branching.

Architect Perspective

Filesystem storage handles blobs/caches/exports the other mechanisms can't. Correct directory choice (persistent vs purgeable), path safety, off-isolate I/O, and repository fronting make it robust and testable — and it's the backing for file-based caches (05_caching_strategies.md), file handling (Module 34), and offline data (Module 19).

Summary

- Use `path_provider` for platform directories (documents=persistent, temporary=cache) + `dart:io` for I/O; join paths with `path.join`.
- Stream/off-isolate large data; front behind a repository; mind web (`dart:io` unavailable).
- Files are for blobs/exports/caches — not queryable structured data (use a DB).

Revision Notes

- `path_provider`: documents/support (persistent) vs temporary (purgeable cache) vs external (Android).
- `dart:io` File/Directory (async); `path.join` for paths; `exists()`/try-catch.
- Large files → stream/`Isolate.run`; web → no `dart:io`.

- Front in a repository; cache in temp dir; encrypt sensitive files (Module 37).

Practice Questions

1. Which directory for a purgeable cache vs a kept document?
2. Why use `path.join` over string concatenation?
3. How do you keep large-file I/O from janking the UI?

Coding Questions

1. Write/read a persistent JSON file in the documents directory.
2. Cache an API response in the temporary directory and read it back.
3. Stream-read a large file off the UI isolate.

Mini Project

File-backed store (Flutter): Build a `FileStore` repository that saves a persistent export (documents dir) and caches API JSON (temporary dir), using `path.join`, `exists()` checks, and off-isolate parsing for large payloads. Acceptance: correct dir per data kind; safe paths; no UI-isolate jank on large data; repository-fronted; app runs.

Caching Strategies (TTL, Cache-First, Invalidation)

> A cache trades freshness for speed/offline resilience — but only if you control **staleness (TTL)**, choose a **read strategy** (cache-first vs network-first vs stale-while-revalidate), and have an **invalidation** plan; an uncontrolled cache is just a stale-data bug.

Introduction

Caching stores fetched data locally (in file/db/memory) to serve faster and offline. This file covers the strategies (cache-first, network-first, stale-while-revalidate), **TTL** (time-to-live), and **invalidation** — the discipline that separates a useful cache from a source of stale-data bugs.

Why this concept exists

Network is slow, costly, and sometimes unavailable. Caching improves perceived speed and enables offline reads. But cached data goes stale; without TTL and invalidation you show wrong data. Strategy makes caching correct, not just fast.

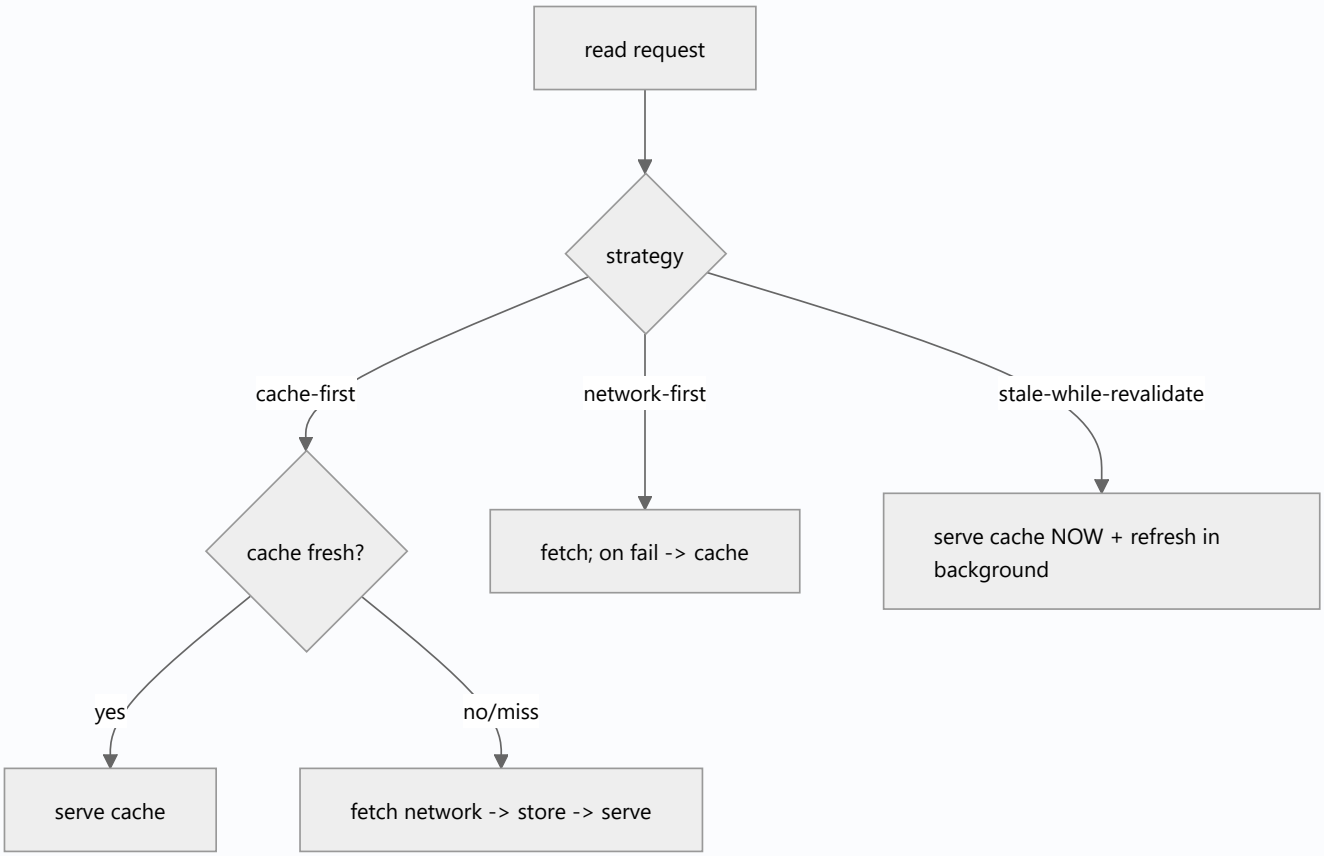
Real-world analogy

A cache is a **fridge**: keeping food (data) handy avoids trips to the store (network). But food expires (TTL) — eat it fresh (network-first), grab whatever's there first (cache-first), or eat the leftover now while restocking (stale-while-revalidate). And you must throw out spoiled items (invalidation).

Problem Statement

A product feed should load instantly from cache, refresh in the background, expire after an hour, and clear on logout/mutation. You'll implement cache-first + TTL + stale-while-revalidate + invalidation, behind a repository.

Internal Working



- **Read strategies**:
  - **Cache-first**: serve cache if present/fresh, else network (fast, may be stale).
  - **Network-first**: try network, fall back to cache on failure (fresh, offline-resilient).
  - **Stale-while-revalidate (SWR)**: serve cache immediately *and* refresh in the background, updating the UI when new data arrives (best perceived UX).
- **TTL (time-to-live)**: store a timestamp with each cache entry; treat it stale after `age > ttl`. "Fresh" = within TTL.
- **Invalidation**: remove/refresh cache on events — logout, mutation (create/update/delete), version change, manual pull-to-refresh, or TTL expiry.
- **Where**: memory (fastest, volatile), file/db (persistent). Often layered (memory + disk).
- Wrap in a **repository** that hides the strategy from callers (05 · repository).

Memory Representation

Cache entries hold data + metadata (timestamp/etag). Memory caches must be **bounded** (LRU/size cap) to avoid leaks; disk caches use the temporary dir (04_file_storage.md, 02 · memory_and_gc).

Compiler Behavior

Not applicable.

Runtime Behavior

On read, the repository checks freshness (TTL) and applies the strategy; SWR returns cached data then emits fresh data (often via a `Stream`). Mutations/logout trigger invalidation.

Flutter Engine Behavior / Dart VM Behavior

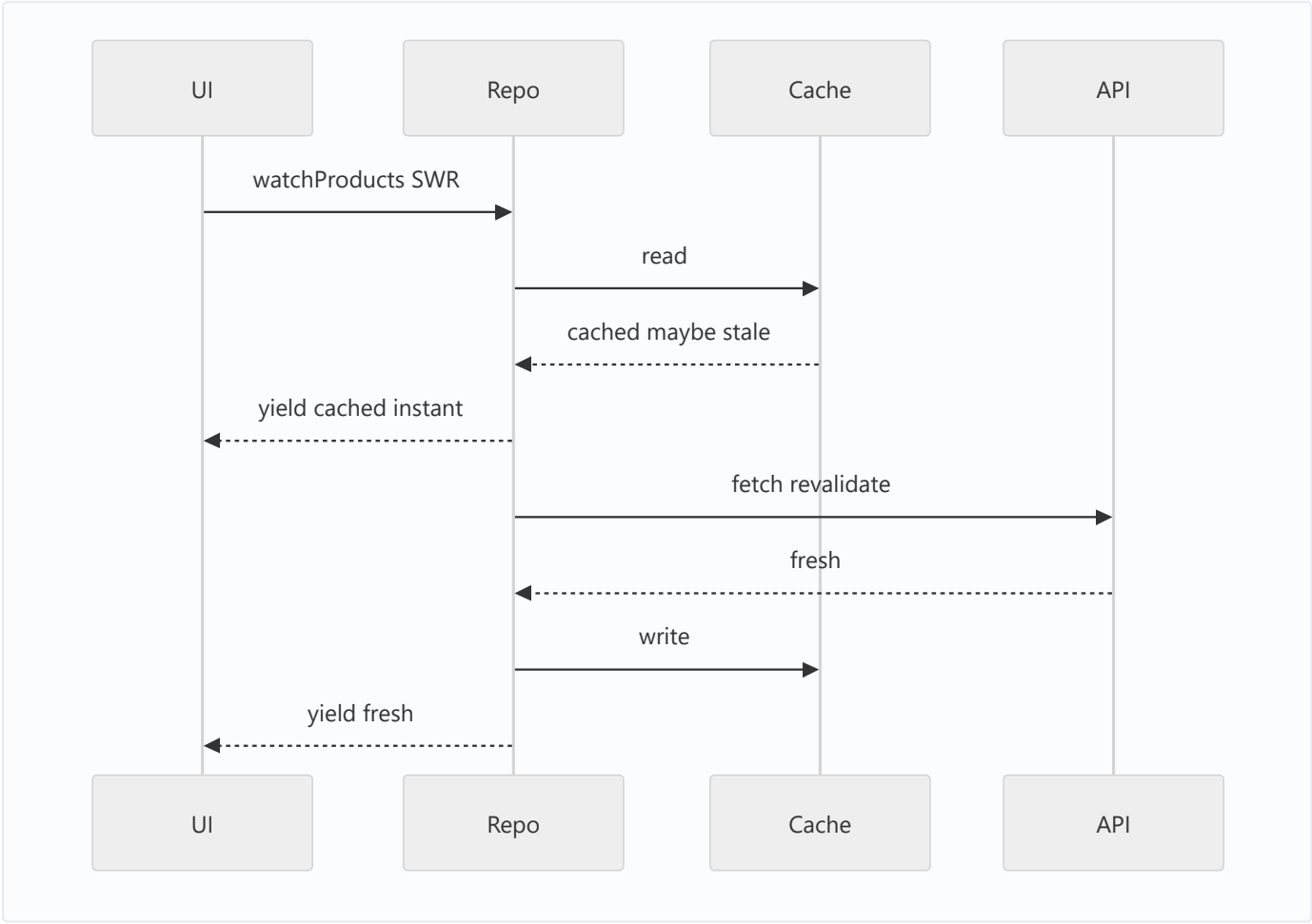
Not applicable.

Examples

```
import 'dart:convert';


class CacheEntry<T> {
  final T data;
  final DateTime storedAt;
  CacheEntry(this.data, this.storedAt);
  bool isFresh(Duration ttl, DateTime now) => now.difference(storedAt) < ttl;
}


abstract interface class ProductApi { Future<List<String>> fetch(); }
abstract interface class CacheBox {
  String? read(String key);
  Future<void> write(String key, String value);
  Future<void> remove(String key);
}


class ProductRepository {
  final ProductApi api;
  final CacheBox cache;
  final Duration ttl;
  final DateTime Function() now; // injectable clock (testable TTL)
  ProductRepository(this.api, this.cache, {this.ttl = const Duration(hours: 1), DateTime Function()?
clock})
      : now = clock ?? DateTime.now;


  static const _key = 'products';
```

```
  // Cache-first with TTL:
  Future<List<String>> getProducts({bool forceRefresh = false}) async {
    if (!forceRefresh) {
      final cached = _readCache();
      if (cached != null && cached.isFresh(ttl, now())) return cached.data; // fresh hit
    }
    final fresh = await api.fetch();       // miss/stale/forced -> network
    await _writeCache(fresh);
    return fresh;
  }

  // Stale-while-revalidate: serve cache now, refresh in background
  Stream<List<String>> watchProducts() async* {
    final cached = _readCache();
    if (cached != null) yield cached.data;     // instant (maybe stale)
    final fresh = await api.fetch();           // revalidate
    await _writeCache(fresh);
    yield fresh;                               // update UI
  }

  Future<void> invalidate() => cache.remove(_key); // on logout/mutation

  CacheEntry<List<String>>? _readCache() {
    final raw = cache.read(_key);
    if (raw == null) return null;
    final m = jsonDecode(raw) as Map<String, dynamic>;
    return CacheEntry(
      (m['data'] as List).cast<String>(),
      DateTime.parse(m['storedAt'] as String),
    );
  }
  Future<void> _writeCache(List<String> data) => cache.write(
      _key, jsonEncode({'data': data, 'storedAt': now().toIso8601String()}));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
No TTL	Data never expires → stale	Store timestamp; treat stale after TTL
No invalidation on mutation/logout	Wrong/leaked data	Invalidate on write/logout/version change
Unbounded memory cache	Leak/OOM	LRU/size cap (02)
Cache-first for critical fresh data	Shows stale (e.g., balance)	Network-first / short TTL
Strategy logic in the UI	Coupling	Hide strategy in the repository
Ignoring `DateTime.now()` in tests	Untestable TTL	Inject a clock

Best Practices

- Always attach a **TTL/timestamp**; define freshness explicitly.
- Choose the strategy per data criticality: **SWR** for feeds (great UX), **network-first** for critical/fresh data (balances, prices), **cache-first** for rarely-changing data.
- **Invalidate** on mutations, logout, and version changes; support pull-to-refresh (`forceRefresh`).

- **Bound** memory caches (LRU/size); persist disk caches in the temporary dir.
- Hide all of this behind a **repository**; inject a **clock** for testable TTL.
- For full offline read/write with conflict handling, graduate to **offline-first** (Module 19).

Performance

Caching cuts latency and network cost and enables offline reads; SWR gives instant-then-fresh UX. Bound caches to protect memory; parse large cached payloads off-isolate (02 · isolates, Module 21).

Advantages / Disadvantages

- **+** Faster loads, less network, offline reads, better UX (SWR), lower cost.
- **−** Staleness risk, invalidation complexity, memory bounds, potential wrong-data bugs if undisciplined.

Interview Questions

1. 🟢 **What is a cache TTL?** — Time-to-live: how long a cache entry is considered fresh; after it, the entry is stale and should be refreshed.
2. 🟢 **Cache-first vs network-first?** — Cache-first serves cache then network (fast, maybe stale); network-first fetches then falls back to cache (fresh, offline-resilient).
3. 🟡 **What is stale-while-revalidate?** — Serve cached data immediately, refresh in the background, then update the UI with fresh data — best perceived UX.
4. 🟡 **When must you invalidate a cache?** — On mutations (create/update/delete), logout, version changes, and TTL expiry; plus manual refresh.
5. 🟡 **Why bound a memory cache?** — Unbounded caches leak/OOM; use LRU or a size cap.
6. 🔴 **Which strategy for a bank balance vs a product feed?** — Balance: network-first / very short TTL (freshness critical); feed: SWR/cache-first (speed, tolerant of brief staleness).
7. 🔴 **How do you make TTL testable?** — Inject a clock (`DateTime Function() now`) so tests can advance time deterministically.

Senior Engineer Tips

- Decide caching per-endpoint by *how bad stale data is* — one global policy is wrong; criticality drives strategy/TTL.
- Prefer SWR for lists/feeds — users get instant content and fresh data arrives seamlessly.
- Always wire invalidation to mutations/logout; a "correct but stale" cache erodes trust fast.

Architect Perspective

Caching strategy is a core performance/UX/offline decision, best encapsulated in the repository/data layer so the app is oblivious to it. TTL + strategy + invalidation + bounded storage compose into a reliable data layer, and scale up into full offline-first sync with conflict resolution (Module 19) — a defining trait of resilient, fast apps.

Summary

- Cache = speed/offline at the cost of freshness; control it with **TTL**, a **read strategy** (cache-first/network-first/SWR), and **invalidation**.
- Choose strategy by data criticality; bound memory; hide behind a repository; inject a clock for testable TTL.
- Escalate to offline-first for full offline read/write with conflict handling.

Revision Notes

- Strategies: cache-first (fast/stale), network-first (fresh/fallback), SWR (cache now + refresh).
- TTL = freshness window (store timestamp); invalidate on mutation/logout/version/refresh.
- Bound memory caches (LRU/size); persist in temp dir; parse big payloads off-isolate.
- Repository-fronted; inject clock (testable TTL); full offline → Module 19.

Practice Questions

1. Which strategy and TTL for a chat list vs a bank balance?
2. When must you invalidate the cache?
3. Why inject a clock into a TTL cache?

Coding Questions

1. Implement cache-first-with-TTL in a repository over an injected cache + clock.
2. Add a stale-while-revalidate `Stream` variant.
3. Add invalidation on mutation and logout; test TTL expiry with a fake clock.

Mini Project — Module capstone

Cached feed data layer (Flutter): Build a `ProductRepository` fronting an API + file/db cache with: TTL-based freshness, cache-first `getProducts` + SWR `watchProducts`, invalidation on mutation/logout, a bounded memory layer, and an injected clock. Unit-test

freshness/expiry/invalidation with a fake clock. Acceptance: strategy hidden behind the repository; TTL + invalidation correct; bounded cache; tests pass; app runs. (Combines prefs/secure/file from this module; bridges to Module 19 Offline First.)