34 · File Handling

Flutter Practice Notes

Contents

34 · File Handling

Introduction

This module covers working with the file system in Flutter: **app directories & paths** (`path_provider` — temp vs documents vs support vs cache, and their platform meanings/lifetimes), **reading & writing files** (`dart:io` `File`, text/bytes/JSON, streaming large files off the UI thread), **picking & sharing files** (`file_picker`, `share_plus`, saving picked files into app storage), and **downloading & caching** (streamed downloads, cache directories, eviction/cleanup), tied together in a capstone. It builds on storage concepts (Module 15), networking (Module 16), and permissions (27/28).

Why this module exists

Apps constantly touch files — cached images, downloaded PDFs, exported reports, user-picked attachments, offline data. But mobile sandboxing makes *where* you write critical: the OS backs up some directories, clears others without warning, and hides most from the user. Writing to the wrong directory means data lost on cleanup, bloated backups, or files the user can't find. Getting directory choice, large-file streaming, and cleanup right is what separates robust file handling from silent data loss.

Module map

#	File	Topic	Level
1	01_app_directories_and_paths.md	`path_provider`: temp/documents/support/cache dirs, lifetimes, platforms	🔵
2	02_reading_writing_files.md	`dart:io` `File`: text/bytes/JSON, streaming, async off-UI-thread	🔵
3	03_file_picker_and_sharing.md	`file_picker`, `share_plus`, saving picked files, save dialogs	🟡
4	04_downloads_and_caching.md	Streamed downloads, cache dirs, TTL/eviction, cleanup	🔴
5	05_file_integration.md	Capstone: file repository, directory strategy, lifecycle	🔴

> **Cross-references:** Storage overview (prefs/secure/files): Module 15. Networking/downloads (`dio`): Module 16. Camera/gallery files: 29 · camera_and_gallery. Isolates (heavy file work): 02 · isolates. PDF/Excel generation: Module 35, Module 36. Offline caching: Module 19.

Prerequisites

15 Local Storage, 16 Networking, 02 Advanced Dart (async/isolates), permissions basics (27/28).

What you'll be able to do after this module

- Choose the correct app directory for each file (persistent vs cache vs temp) per platform.
- Read/write text, bytes, and JSON — and stream large files without blocking the UI.
- Let users pick and share files, and save picked files into app storage.
- Download files with progress, cache them, and implement eviction/cleanup.
- Wrap file access behind a repository with a clear directory & lifecycle strategy.

Capstone

File slice: A document manager that downloads PDFs with progress into the cache dir, saves user-picked attachments into the documents dir, reads/writes a JSON index, streams large files off the UI thread, lets the user share files, and evicts stale cache — all behind a `FileRepository` with an explicit directory strategy.

Summary

File handling = choosing the right sandboxed directory (`path_provider`), reading/writing efficiently (streaming large files off the UI thread), picking/sharing (`file_picker` / `share_plus`), and downloading/caching with cleanup. The core discipline is a deliberate directory & lifecycle strategy behind a repository so nothing is lost, bloated, or blocking.

App Directories & Paths (`path_provider`)

> You never hardcode paths on mobile — the app is **sandboxed**, so you ask `path_provider` for the right directory, and *which* one matters: **documents** (persistent, user data, may be backed up), **application support** (persistent, hidden app data), **cache/temporary** (the **OS can delete these anytime** — never store anything you can't re-fetch there), and **external storage** (Android, shared/large files). Choosing wrong means data lost on cleanup, bloated iCloud/Google backups, or files you can't recover.

Introduction

Every file operation starts with "where do I put this?" `path_provider` returns platform-correct, sandboxed directories with different **persistence guarantees, backup behavior, and visibility**. This file maps those directories, their lifetimes, and platform differences — the foundation for all file work.

Why this concept exists

Mobile OSes sandbox each app and manage storage aggressively (clearing caches under pressure, backing up user data, hiding internals). Paths differ per platform and OS version, so `path_provider` abstracts them into semantic directories. Picking the directory by **intent** (persistent user data vs disposable cache) is how you cooperate with the OS instead of fighting it.

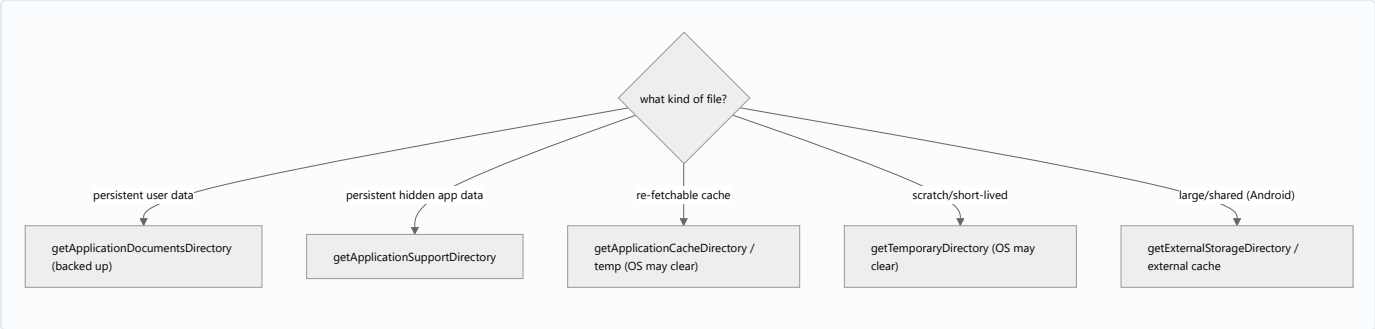
Real-world analogy

The directories are different **storage rooms**: **documents** is your **filing cabinet** (kept, sometimes copied to off-site backup), **application support** is a **locked back office** (kept, staff-only), **cache/temp** is a **recycling bin the janitor empties whenever** (don't leave anything important), and **external storage** is a **shared warehouse** for bulky/shareable goods. Putting your tax records in the recycling bin (cache) means they're gone next cleanup.

Problem Statement

Decide where to store: a user's saved documents, an app config/index, downloaded images that can be re-fetched, and a large exported file to share — on both platforms, respecting backup and cleanup. You'll call the right `path_provider` API per case.

Internal Working



- **Documents** (`getApplicationDocumentsDirectory`): **persistent** user-generated/important data. On **iOS** it maps to the app's Documents (often **included in iCloud/iTunes backup**) — don't dump large re-downloadable blobs here (bloats backups); mark large caches as excluded if you must. On Android, app-private files dir.
- **Application support** (`getApplicationSupportDirectory`): **persistent** app data **not** meant to be user-visible (config, databases). Preferred over Documents for internal files on iOS.
- **Cache** (`getApplicationCacheDirectory`) / **temporary** (`getTemporaryDirectory`): **the OS may delete these at any time** (low storage, cleanup). Use for **re-fetchable** downloads, thumbnails, scratch — **never** the sole copy of anything important. Temp is the shortest-lived.
- **External storage** (Android only: `getExternalStorageDirectory` , `getExternalCacheDirectories`): larger/shared storage. Scoped storage (Android 10+) limits access; for user-visible files prefer the system pickers/ `MediaStore` (03_file_picker_and_sharing.md). iOS has no direct equivalent.
- **Rule of thumb**: **persistent user data → documents/support; disposable → cache/temp.** Always build paths with `path.join(dir.path, name)` (via the `path` package), never string concatenation.
- **Async**: all these return `Future<Directory>` — cache the resolved dir; don't call repeatedly.

Memory Representation

Directories are just paths (strings). The persistence/backup/cleanup semantics are OS policy, not in-memory state. Cache the resolved `Directory` to avoid repeated async lookups.

Compiler Behavior

Not applicable; resolved at runtime per platform.

Runtime Behavior

Cache/temp contents can vanish between launches (OS cleanup). Documents/support persist across launches/updates (until uninstall). Backup restores may repopulate documents on a new device.

Flutter Engine Behavior

Not applicable; `path_provider` bridges to native directory APIs via platform channels (26).

Dart VM Behavior

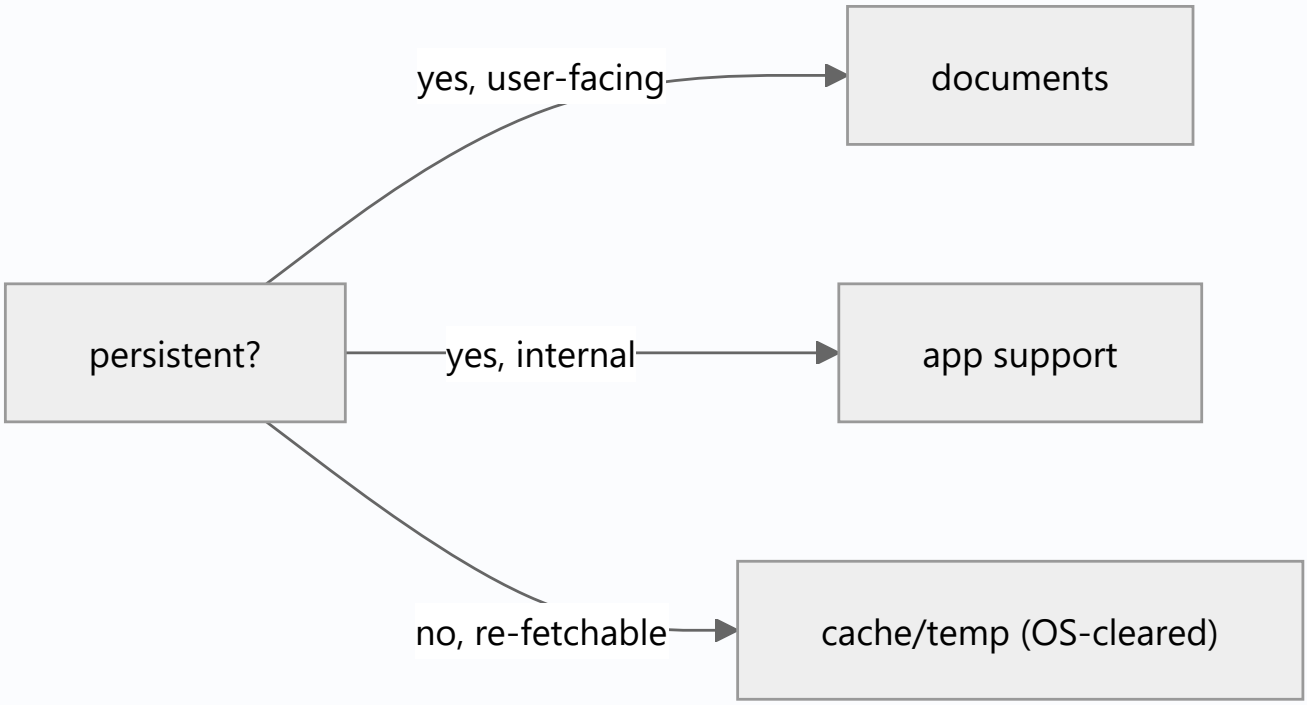
Not applicable.

Examples

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

class AppPaths {
  // Persistent user documents (careful: iOS backs these up)
  Future<File> documentFile(String name) async {
    final dir = await getApplicationDocumentsDirectory();
    return File(p.join(dir.path, name));                // always use path.join
  }

  // Persistent, hidden app data (config/db) — preferred for internal files
  Future<File> supportFile(String name) async {
    final dir = await getApplicationSupportDirectory();
    return File(p.join(dir.path, name));
  }

  // Re-fetchable cache — OS may delete anytime; never the only copy
  Future<File> cacheFile(String name) async {
    final dir = await getTemporaryDirectory();          // or getApplicationCacheDirectory
    return File(p.join(dir.path, name));
  }
}
```

```
Directory decision cheat-sheet:
  user-saved document / important export -> documents (persistent, backed up)
  app config / sqlite db / internal index -> application support (persistent, hidden)
  downloaded image/pdf you can re-fetch   -> cache/temporary (OS may clear)  <-- NOT the only copy
  scratch during an operation             -> temporary
  large shared file (Android)             -> external storage / system picker
```

Diagrams



Common Mistakes

Mistake	Why it's bad	Fix
Storing important data in cache/temp	OS deletes it anytime	Use documents/support for persistent data
Large re-downloadable blobs in documents (iOS)	Bloats iCloud backup	Use cache, or exclude from backup
Hardcoding paths	Sandbox/platform differences	Use `path_provider` + `path.join`
String-concatenating paths	Separator/edge bugs	`p.join(dir.path, name)`
Repeated async dir lookups	Wasteful	Cache the resolved directory
Assuming external storage on iOS	iOS has none	Guard Android-only APIs
Expecting cache to persist	It won't	Design for re-fetch

Best Practices

- Choose the directory by **intent**: **documents** (persistent user data), **application support** (persistent internal), **cache/temporary** (re-fetchable, OS may clear).
- Never store the **only copy** of important data in cache/temp; avoid **large re-downloadable blobs in iOS documents** (backup bloat) — cache or exclude from backup.

- Build paths with `path.join`; **cache** resolved directories; guard **Android-only** external-storage APIs.
- For user-visible/shared files, prefer **system pickers/share** over raw external paths (scoped storage) (03_file_picker_and_sharing.md).

Performance

Directory resolution is a cheap async call — cache it. The real cost is misplacement (re-downloading cleared "persistent" data, or bloated backups); correct directory choice is a performance/UX decision.

Advantages / Disadvantages

- + Correct sandboxed, platform-safe paths with clear persistence/backup/cleanup semantics.
- − Must understand each directory's lifetime, platform differences (external storage, iOS backup), scoped-storage limits.

Interview Questions

1. 🟢 **Why use `path_provider` instead of hardcoding paths?** — Apps are sandboxed and paths differ per platform/version; `path_provider` returns semantic, platform-correct directories.
2. 🟢 **Which directory for re-fetchable downloads vs important user data?** — Cache/temporary for re-fetchable (OS may clear); documents/support for persistent important data.
3. 🟡 **What's special about the temporary/cache directory?** — The OS can delete its contents anytime (storage pressure/cleanup) — never store the only copy of anything there.
4. 🟡 **Why avoid large blobs in the iOS documents directory?** — It's typically backed up to iCloud, so large re-downloadable files bloat backups — use cache or exclude from backup.
5. 🟡 **documents vs application support?** — Both persistent; documents is user-facing (backed up), application support is hidden internal data (preferred for db/config on iOS).
6. 🔴 **How does Android scoped storage affect file handling?** — Direct external-storage access is restricted (Android 10+); use app-private dirs or system pickers/ `MediaStore` for user-visible files.
7. 🔴 **Why build paths with `path.join`?** — To handle separators/platform differences correctly and avoid concatenation bugs.

Senior Engineer Tips

- Decide the directory strategy per file type up front and document it; "why did my data disappear" is almost always cache-vs-documents confusion.

- On iOS, keep big caches out of Documents (or set the do-not-backup flag) — backup bloat is a real user complaint and review concern.
- Cache resolved directories in your file repository and never concatenate paths by hand.

Architect Perspective

Directory choice is a data-lifecycle policy decision, not a detail. Encoding it once in a `FileRepository` / `AppPaths` (persistent vs cache vs temp per file type, platform-guarded, backup-aware) prevents an entire class of bugs — vanished data, bloated backups, scoped-storage failures — and gives the rest of the app intent-based file access (05_file_integration.md, Module 15).

Summary

- `path_provider` gives sandboxed, platform-correct directories: documents (persistent, user, backed up), application support (persistent, internal), cache/temporary (re-fetchable, OS may clear), external (Android).
- Choose by intent; never keep the only copy in cache/temp; avoid large blobs in iOS documents; use `path.join`.
- Cache resolved dirs; guard Android-only APIs; use system pickers for shared files.

Revision Notes

- `getApplicationDocumentsDirectory` (persistent, user, iOS-backed-up), `getApplicationSupportDirectory` (persistent, hidden), `getTemporaryDirectory` / `getApplicationCacheDirectory` (OS may clear), `getExternalStorageDirectory` (Android).
- Persistent → documents/support; disposable/re-fetchable → cache/temp; large iOS blobs → cache/exclude from backup.
- Build with `path.join`; cache resolved dirs; scoped storage (Android 10+) → system pickers for user files.

Practice Questions

1. Where do you store data the OS must never delete vs data it may?
2. Why avoid large downloads in the iOS documents directory?
3. Why use `path.join` instead of string concatenation?

Coding Questions

1. Build an `AppPaths` helper returning documents/support/cache file handles.

2. Decide + justify the directory for five different file types.
3. Guard an Android-only external-storage path.

Mini Project

Directory strategy (Flutter): Build an `AppPaths` helper exposing `documentFile`, `supportFile`, and `cacheFile` (via `path_provider` + `path.join`, cached dirs), and write a short doc mapping five file types (user doc, app config, re-fetchable download, scratch, large export) to directories with justification (persistence/backup/cleanup). Acceptance: correct APIs per intent; paths via `path.join`; dirs cached; iOS-backup + cache-clearing considerations documented; Android-only APIs guarded.

Reading & Writing Files (`dart:io` `File`)

> Read/write with `dart:io` `File`: `readAsString` / `writeAsString` (text/JSON), `readAsBytes` / `writeAsBytes` (binary), always **async** (never the `...Sync` variants on the UI thread). For **large files**, don't load the whole thing into memory — use **streams** (`file.openRead()` / `openWrite()`) and offload heavy parsing to an **isolate**; write **atomically** (temp file + rename) so a crash mid-write can't corrupt data.

Introduction

Once you have a path (01_app_directories_and_paths.md), you read/write bytes or text. This file covers the `File` API for text/bytes/JSON, async vs sync, streaming large files, atomic writes, and offloading heavy work — the mechanics of not blocking the UI or losing data.

Why this concept exists

File I/O is blocking and can be slow (large files, slow storage). Dart's `File` offers async APIs so I/O doesn't jank the UI, and streaming so you don't OOM on big files. Atomic writes exist because a partial write (crash/kill mid-write) otherwise corrupts the file.

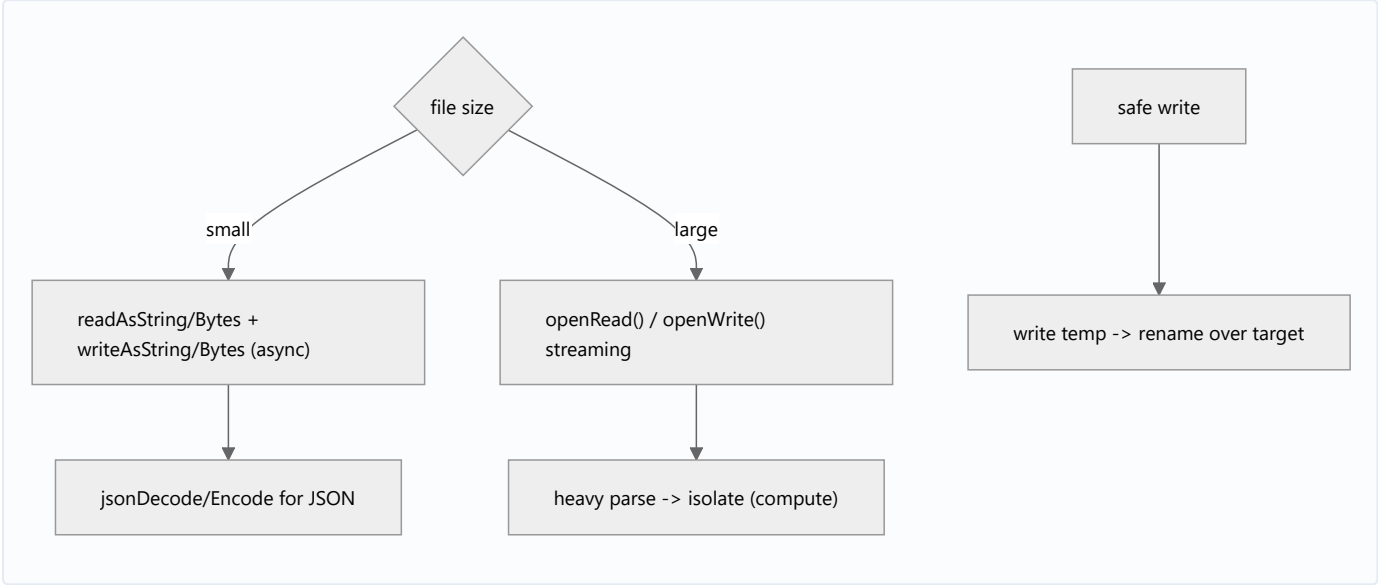
Real-world analogy

Reading a small file is **grabbing a note off the desk** (`readAsString`). A huge file is a **library you read page-by-page** (stream) rather than photocopying the whole thing into your arms (loading into memory → OOM). An atomic write is **writing the final copy on a fresh sheet, then swapping it for the old one in one move** — so no one ever sees a half-written page.

Problem Statement

Persist a JSON app-state index (small), append log lines efficiently, and import/parse a 200 MB data file without freezing the app or running out of memory — safely against crashes. You'll use text/bytes APIs, streaming, and atomic writes.

Internal Working



- **Small files (whole)**: `await file.writeAsString(jsonEncode(data))` / `final s = await file.readAsString()` ; `writeAsBytes` / `readAsBytes` for binary (images/blobs). JSON = `dart:convert` `jsonEncode` / `jsonDecode` .
- **Async only**: use the async methods; the `...Sync` variants block the calling isolate (UI jank) — avoid on the UI thread. Handle `FileSystemException` (missing/permission/full).
- **Existence/dirs**: `await file.exists()` , `await file.create(recursive: true)` (creates parent dirs), `file.delete()` , `dir.list()` to enumerate.
- **Append**: `writeAsString(s, mode: FileMode.append)` or an `IOSink` (`file.openWrite(mode: append)`) for many appends (logs) — flush/close it.
- **Large files (stream)**: `file.openRead()` yields `Stream<List<int>>` — process chunk-by-chunk (decode lines, hash, upload) without loading all into memory; `file.openWrite()` gives an `IOSink` to write incrementally.
- **Offload heavy CPU** (parsing/transform of big data) to an **isolate** via `compute()` / `Isolate.run` so decoding doesn't jank (02 · isolates).
- **Atomic write**: write to a temp file, then `rename()` over the target (rename is atomic on the same volume) — prevents corruption if interrupted; pair with the right directory.
- **Encoding**: default UTF-8 for text; specify explicitly if needed.

Memory Representation

`readAsBytes` / `readAsString` load the **entire** file into memory (fine for small, fatal for huge). Streaming holds only a chunk at a time. An `IOSink` buffers writes until flushed/closed.

Compiler Behavior

Not applicable.

Runtime Behavior

Async I/O runs off the main thread (event loop resumes on completion); sync I/O blocks. Streaming processes incrementally. A crash mid-non-atomic-write leaves a corrupt/partial file.

Flutter Engine Behavior

Not applicable; `dart:io` talks to the OS file system directly.

Dart VM Behavior

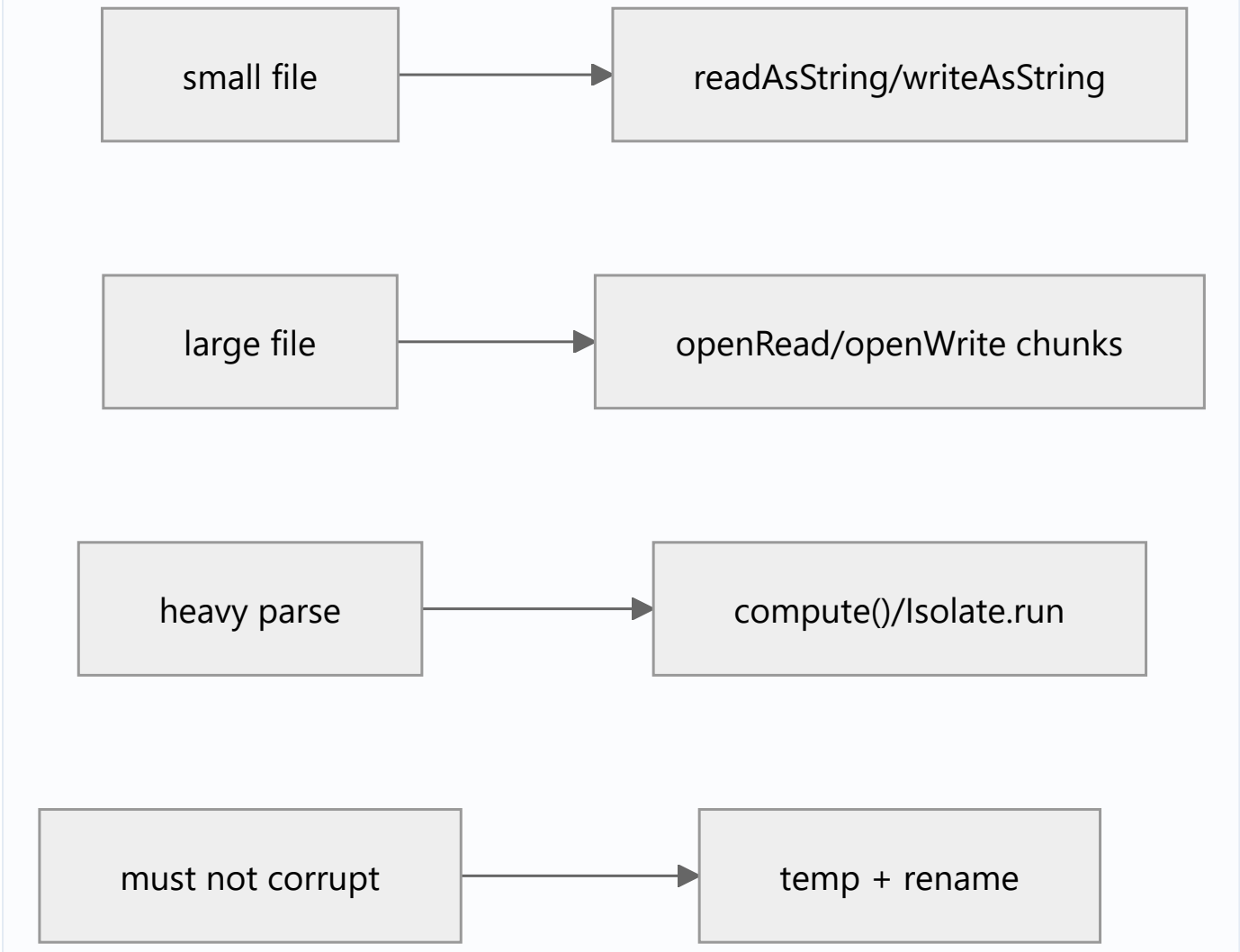
Async file ops use the VM's I/O thread pool; heavy CPU parsing still runs on the calling isolate unless you offload to another isolate.

Examples

```
import 'dart:io';
import 'dart:convert';
import 'package:flutter/foundation.dart'; // compute

// Small JSON: read/write whole (async)
Future<void> saveIndex(File file, Map<String, dynamic> index) async {
  await _atomicWriteString(file, jsonEncode(index));    // safe write
}
Future<Map<String, dynamic>> loadIndex(File file) async {
  if (!await file.exists()) return {};
  return jsonDecode(await file.readAsString()) as Map<String, dynamic>;
}

// Atomic write: temp -> rename (no corruption on crash)
Future<void> _atomicWriteString(File target, String contents) async {
  final tmp = File('${target.path}.tmp');
  await tmp.writeAsString(contents, flush: true);
  await tmp.rename(target.path);                        // atomic swap
}

// Append logs efficiently with an IOSink
Future<void> appendLog(File logFile, String line) async {
  final sink = logFile.openWrite(mode: FileMode.append);
  sink.writeln(line);
  await sink.flush();
```

```
  await sink.close();
}

// Large file: stream line-by-line, offload heavy parse to an isolate
Future<int> countMatches(File big, String needle) async {
  var count = 0;
  final lines = big.openRead().transform(utf8.decoder).transform(const LineSplitter());
  await for (final line in lines) {                    // one line at a time (low memory)
    if (line.contains(needle)) count++;
  }
  return count;
}
Future<Report> parseHeavy(String path) => compute(_parseInIsolate, path); // off UI thread
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`...Sync` on the UI thread	Jank/ANR	Use async APIs
`readAsBytes` on a huge file	OOM	Stream with `openRead()`
Non-atomic write	Corruption on crash	temp + `rename`
Heavy parse on UI isolate	Jank	Offload via `compute` /isolate
Not handling `FileSystemException`	Crash (missing/full/perm)	try/catch + fallbacks
Not closing `IOSink`	Data not flushed/leak	`flush()` + `close()`
Reopening file per append	Slow	Keep an `IOSink` / batch

Best Practices

- Use **async** `File` APIs; **stream** large files (`openRead` / `openWrite`) instead of loading them whole; **offload heavy parsing** to an isolate.
- Write **atomically** (temp + `rename`) for anything important; handle `FileSystemException` (missing/full/permission).
- Use `IOSink` for many appends (logs), flushing/closing it; JSON via `dart:convert` ; specify **encoding** when needed.
- Create parent dirs with `create(recursive: true)` ; check `exists()` ; pair with the correct directory (01_app_directories_and_paths.md).

Performance

Async keeps I/O off the UI thread; streaming caps memory at one chunk; isolates keep CPU-heavy parsing from janking. The wins are avoiding OOM (stream), avoiding jank (async + isolate), and avoiding corruption (atomic).

Advantages / Disadvantages

- **+** Flexible text/binary/streaming I/O, async non-blocking, atomic safety, JSON support.
- **−** Must manage memory (large files), sinks (flush/close), atomicity, exceptions, and isolate offloading yourself.

Interview Questions

1. 🟢 **Why prefer async file APIs over the** `Sync` **variants?** — Sync I/O blocks the isolate (UI jank/ANR); async runs off-thread and resumes via the event loop.

2. 🟢 **How do you read a very large file without OOM?** — Stream it with `openRead()` and process chunk/line by line rather than `readAsBytes` / `readAsString`.
3. 🟡 **What is an atomic write and why?** — Write to a temp file then `rename` over the target so an interrupted write can't corrupt the real file.
4. 🟡 **How do you efficiently append many log lines?** — Keep an `IOSink` (`openWrite(mode: append)`), write, then flush/close — avoid reopening per line.
5. 🟡 **How do you keep heavy parsing from janking the UI?** — Offload CPU-bound work to an isolate (`compute` / `Isolate.run`).
6. 🔴 **What exceptions must you handle and when?** — `FileSystemException` for missing files, permission denials, and full disks — with fallbacks.
7. 🔴 **What does `readAsBytes` do to memory vs streaming?** — Loads the whole file into memory (risky for large files); streaming holds only one chunk at a time.

Senior Engineer Tips

- Make important writes atomic by default (a tiny helper) — partial-write corruption is rare but catastrophic and easy to prevent.
- Reach for streaming + isolates the moment files get large; loading a big file whole is the classic OOM/jank bug.
- Wrap file ops in try/catch for `FileSystemException` (disk full/permission) and degrade gracefully rather than crashing.

Architect Perspective

Reading/writing is where correctness (atomicity, exceptions) and performance (async, streaming, isolates) meet. Encapsulating these in a `FileRepository` (atomic writes, streamed large-file helpers, isolate offloading) gives the app safe, non-blocking file access and keeps the tricky memory/corruption concerns in one tested place — feeding caching, downloads, and export features (04_downloads_and_caching.md, 05_file_integration.md, 02 · isolates).

Summary

- `dart:io` `File`: async text/bytes/JSON; stream large files (`openRead` / `openWrite`); offload heavy parse to isolates.
- Write atomically (temp + rename); handle `FileSystemException`; use `IOSink` for appends (flush/close).
- Async + streaming + isolates avoid jank/OOM; atomicity avoids corruption.

Revision Notes

- Small: `readAsString` / `writeAsString`, `readAsBytes` / `writeAsBytes` (async); JSON via `dart:convert`.
- Large: `openRead()` (Stream<List>) / `openWrite()` (IOSink) chunked; heavy parse → `compute`/isolate.
- Atomic write = temp + `rename`; handle `FileSystemException`; `create(recursive:true)`; flush/close sinks; avoid `...Sync` on UI.

Practice Questions

1. When must you stream a file instead of reading it whole?
2. How do you prevent file corruption on a crash mid-write?
3. How do you append log lines efficiently?

Coding Questions

1. Implement atomic JSON save/load helpers.
2. Stream a large file line-by-line counting matches (low memory).
3. Offload a heavy parse to an isolate with `compute`.

Mini Project

Safe file I/O helpers (Flutter): Build a small file utility: atomic JSON `saveIndex` / `loadIndex`, an `IOSink`-based `appendLog`, and a streamed large-file processor that offloads heavy parsing to an isolate — with `FileSystemException` handling. Acceptance: writes are atomic (temp+rename); large file processed with low memory (streamed); heavy parse off the UI thread; appends via a flushed/closed sink; exceptions handled; all async.

Picking & Sharing Files (`file_picker`, `share_plus`)

> Let users bring files **in** with `file_picker` (system picker → returns a path/bytes; may be a temp copy you must persist) and send files **out** with `share_plus` (the native share sheet). Key gotchas: a picked file often lives in a **temporary location** (copy it into app storage if you need it later), pickers respect **scoped storage** (no raw filesystem browsing), and you should **share via the OS share sheet**, not by exposing raw sandbox paths (which other apps can't read).

Introduction

Import/export is how files cross the app boundary. `file_picker` opens the OS document/media picker; `share_plus` opens the native share sheet. This file covers picking (single/multiple, type filters, bytes vs path), persisting picked files, saving files out, and sharing — the user-facing edges of file handling.

Why this concept exists

Sandboxing means apps can't freely browse the device or hand files to each other by path. The OS provides pickers (user grants access to a specific file) and share sheets (OS mediates file hand-off). These plugins wrap those native flows so users can attach documents and share exports safely.

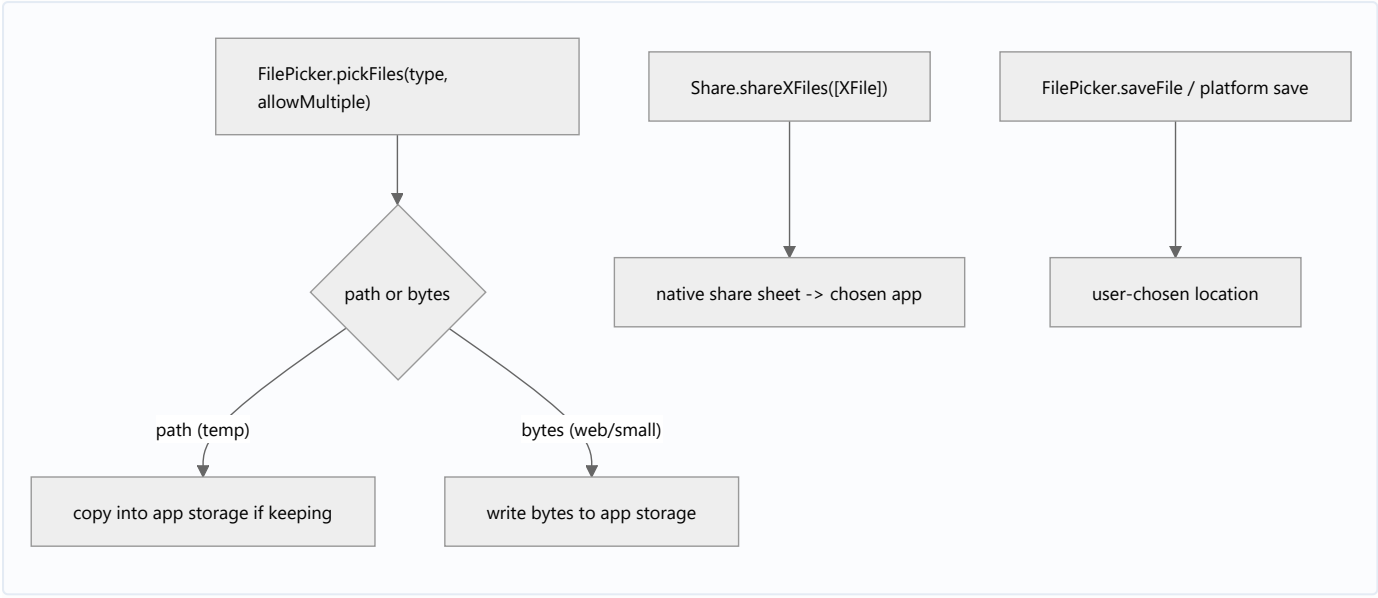
Real-world analogy

`file_picker` is the **reception desk** where a visitor hands you a **photocopy** of a document (a temp copy) — if you want to keep it, you must **file it in your own cabinet** (copy to app storage), because the original stays with them. `share_plus` is the **mailroom's outgoing tray**: you drop a file there and the OS decides which courier (app) delivers it — you don't hand your private cabinet key to strangers.

Problem Statement

Let the user attach a PDF/image (pick, then keep it across launches), export a generated report to share via WhatsApp/email/Drive, and let them save a file where they choose. You'll use `file_picker` (+ persist), `share_plus`, and a save dialog.

Internal Working



- **Picking** (`file_picker`): `FilePicker.platform.pickFiles(type: FileType.custom, allowedExtensions: ['pdf'], allowMultiple: false)` → `FilePickerResult?` (null if cancelled) with `files` each exposing `path` (mobile/desktop) and/or `bytes` (web/ `withData: true`).
- **Temp-copy gotcha**: on many platforms the picked file's `path` points to a **temporary/cache copy** the OS may clear. To use it later, **copy it into your documents/support dir** (01_app_directories_and_paths.md) and store *that* path — don't persist the picker's path.
- **Type filters/size**: filter by extension/media type; validate size/type after picking (don't trust the extension alone for security). Use `withData: true` when you need bytes (web) — heavy for large files.
- **Sharing** (`share_plus`): `Share.shareXFiles([XFile(path)], text: ...)` opens the native sheet; also `Share.share(text)` for text/links. On iPad, provide `sharePositionOrigin` (popover anchor) or it throws.
- **Saving out**: `FilePicker.platform.saveFile(...)` (desktop/Android) lets the user choose a destination; on iOS, "saving" is typically done via the **share sheet** → **Save to Files**.
- **Permissions**: modern pickers/share **don't need broad storage permissions** (the picker grants scoped access) — prefer them over requesting full storage access (27 · permissions).
- **Cleanup**: delete temp copies you created; don't leak duplicates.

Memory Representation

`path` -based results reference a file on disk (possibly temp); `bytes` -based results load content into memory (avoid for large files). Persisted copies live in your chosen app directory.

Compiler Behavior

Not applicable.

Runtime Behavior

Pickers/share sheets are native modal flows (app pauses while shown). A picked temp file may be cleared later — hence the copy-to-persist rule.

Flutter Engine Behavior

Pickers/share sheets are native UI over Flutter (platform channels — 26); `shareXFiles` hands file URIs to the OS.

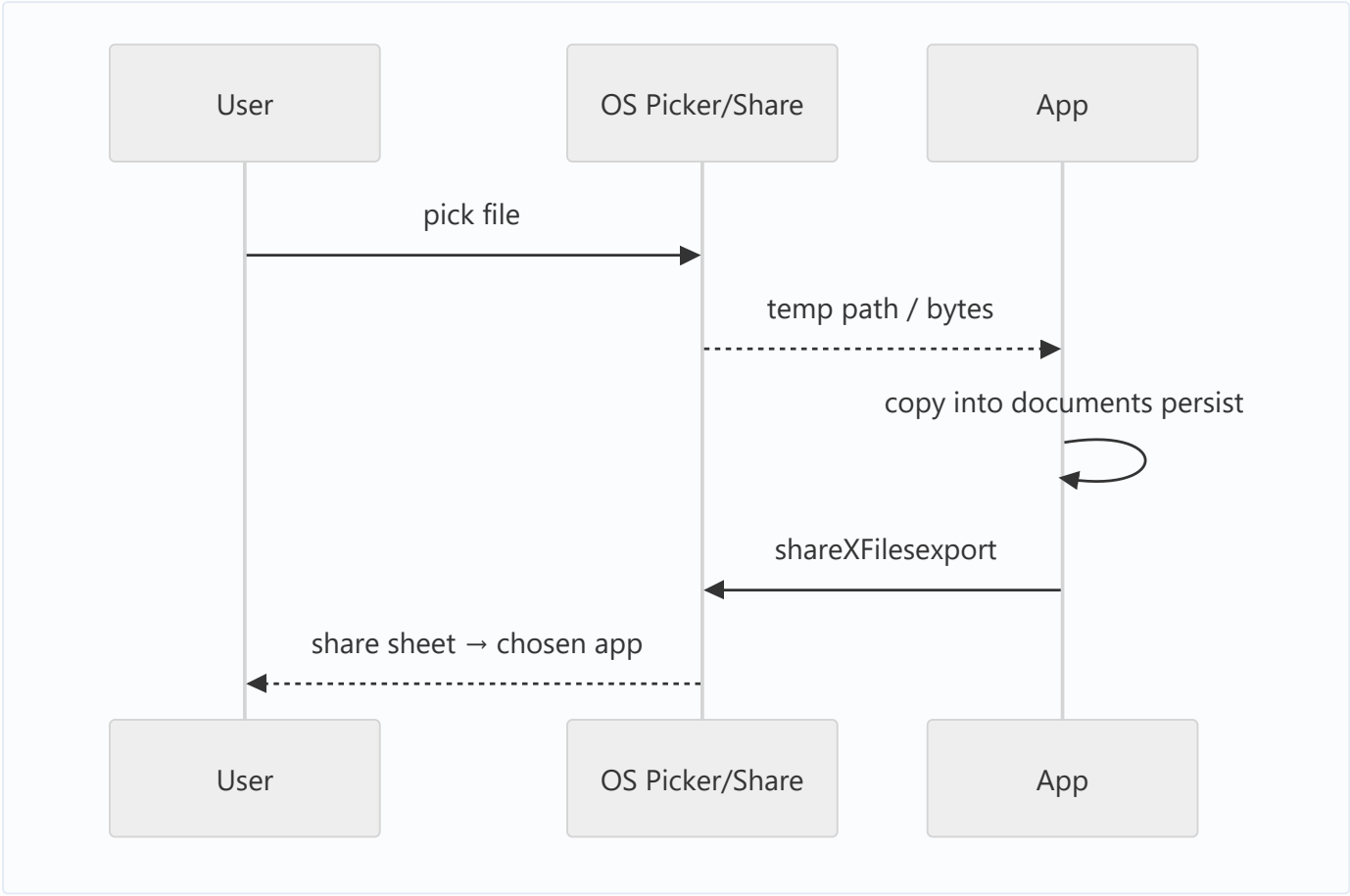
Dart VM Behavior

Not applicable.

Examples

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

// Pick a PDF and PERSIST it into app storage (picker path may be temporary)
Future<File?> pickAndKeepPdf() async {
  final result = await FilePicker.platform.pickFiles(
    type: FileType.custom, allowedExtensions: ['pdf'], allowMultiple: false,
  );
  final picked = result?.files.single.path;
  if (picked == null) return null;                    // cancelled
  final dir = await getApplicationDocumentsDirectory();
  final saved = File(p.join(dir.path, p.basename(picked)));
  return File(picked).copy(saved.path);               // keep our own copy
}

// Share a file via the native share sheet
Future<void> shareReport(File report) async {
  await Share.shareXFiles([XFile(report.path)], text: 'Here is your report');
  // iPad: pass sharePositionOrigin to avoid a crash.
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Persisting the picker's temp path	OS clears it → broken reference	Copy into app storage; store that path
`withData: true` for large files	High memory	Use `path` ; stream if needed
Trusting extension for type/security	Spoofable	Validate content/size after pick
Sharing raw sandbox path	Other apps can't read it	Use `shareXFiles` (OS mediates)
iPad share without origin	Throws	Provide `sharePositionOrigin`
Requesting broad storage perms	Unnecessary/rejected	Pickers grant scoped access
Leaking temp copies	Storage bloat	Clean up copies you make

Best Practices

- After picking, **copy the file into your app directory** and store that path — never rely on the picker's (often temporary) path.
- Filter by type and **validate content/size** after picking; use `path` over `bytes` for large files (bytes only when needed, e.g., web).

- **Share via** `share_plus` (`shareXFiles`), not raw paths; on **iPad** set `sharePositionOrigin` ; for save-out use `saveFile` /share-to-Files.
- Prefer **system pickers** (scoped access) over broad storage permissions; **clean up** temp copies you create.

Performance

Pickers/share are lightweight (native). The cost is loading `bytes` for large files (avoid) and leaking copies (clean up). Copying a picked file is a normal I/O cost — stream if very large (02_reading_writing_files.md).

Advantages / Disadvantages

- \+ User-friendly import/export, scoped (no broad permissions), native share targets, type filtering.
- − Temp-copy persistence gotcha, memory risk with `bytes` , iPad share origin, must validate/clean up, scoped-storage limits.

Interview Questions

1. 🟢 **What does** `file_picker` **return and what's the catch?** — A path and/or bytes; the path is often a temporary copy — persist it into app storage if you need it later.
2. 🟢 **How do you share a file from a Flutter app?** — `Share.shareXFiles([XFile(path)])` (share_plus) opens the native share sheet; don't expose raw sandbox paths.
3. 🟡 **Why copy a picked file into your app directory?** — The picker's temp location can be cleared by the OS; a persisted copy under documents/support survives.
4. 🟡 **When would you use** `bytes` **vs** `path` **?** — `bytes` for web or small in-memory needs; `path` for mobile/desktop and large files (avoid loading big files into memory).
5. 🟡 **Why prefer system pickers over storage permissions?** — Pickers grant scoped access to the chosen file without requesting broad, often-rejected storage permissions.
6. 🔴 **What's the iPad share gotcha?** — `shareXFiles` needs a `sharePositionOrigin` (popover anchor) on iPad or it throws.
7. 🔴 **Why not trust the picked file's extension for security?** — Extensions are spoofable; validate content/size (and scan if untrusted) after picking.

Senior Engineer Tips

- Make "pick → copy into app storage → return our path" a single helper; the temp-path bug is the most common file-picker regression.

- Use `path` and stream-copy large picks; loading `bytes` for a big attachment is a silent memory spike.
- Always share through the OS sheet and set the iPad origin; and validate type/size on import rather than trusting the source.

Architect Perspective

Picking/sharing are the app's file boundary with the OS and other apps — inherently scoped and mediated. Encapsulating "pick-and-persist" and "share" in the file repository (with validation, copy-to-app-storage, cleanup, and iPad handling) keeps features simple and safe, and integrates with generated exports (PDF/Excel) and attachments feeding uploads/offline (Module 35, Module 36, 05_file_integration.md).

Summary

- `file_picker`: system picker → path/bytes; **persist by copying into app storage** (picker path is often temporary); validate type/size; `bytes` only when needed.
- `share_plus`: `shareXFiles` via the native sheet (not raw paths); iPad needs `sharePositionOrigin`; save-out via `saveFile`/share-to-Files.
- Prefer scoped pickers over storage permissions; clean up temp copies.

Revision Notes

- `FilePicker.platform.pickFiles(type, allowedExtensions, allowMultiple, withData)` → `FilePickerResult?` (null = cancel); copy path into documents/support to persist.
- `Share.shareXFiles([XFile(path)], text:)` / `Share.share(text)`; iPad `sharePositionOrigin`; `saveFile` for save-out.
- `path` vs `bytes` (memory); validate type/size; scoped access (no broad perms); clean up copies.

Practice Questions

1. Why must you copy a picked file into app storage?
2. How do you share a generated file, and what's the iPad caveat?
3. When do you use `bytes` vs `path` from the picker?

Coding Questions

1. Implement `pickAndKeepPdf()` that persists the pick into documents.
2. Share a generated report via `share_plus` (handle iPad).
3. Validate a picked file's size/type before accepting it.

Mini Project

Attach & share (Flutter): Build import/export: `pickAndKeep` (pick a PDF/image, validate size/type, copy into documents, return the persisted path), and `share` (share a file via `share_plus`, iPad-safe). Clean up temp copies and avoid broad storage permissions. Acceptance: picked files persist across launches (copied); validation enforced; sharing works via the native sheet (iPad origin set); no broad permissions; temp copies cleaned up.

Downloads & Caching (Streamed Downloads, Eviction, Cleanup)

> Download files by **streaming the response to disk** (via `dio`'s `download`/response stream — never buffer a large file fully in memory), store them in a **cache directory** keyed by a stable id (URL hash), serve from cache on subsequent requests (cache-first), and **evict** by **TTL / LRU / size budget** with periodic cleanup — because the OS can clear the cache anytime, so cached files must always be **re-fetchable**, never the source of truth.

Introduction

Caching downloaded files (images, PDFs, media) avoids re-downloading and enables offline viewing, but an unmanaged cache grows unbounded and stores things it shouldn't. This file covers streamed downloads with progress, cache keys/lookup, eviction policies, and cleanup — the disciplined version of "download and keep."

Why this concept exists

Re-downloading wastes data/battery/time; caching to disk fixes that. But device storage is finite and the OS reclaims caches, so you need a **bounded, evictable, re-fetchable** cache — not an ever-growing pile. Streaming exists so large downloads don't OOM. (For image widgets, `cached_network_image` does much of this; this file is the general-file version.)

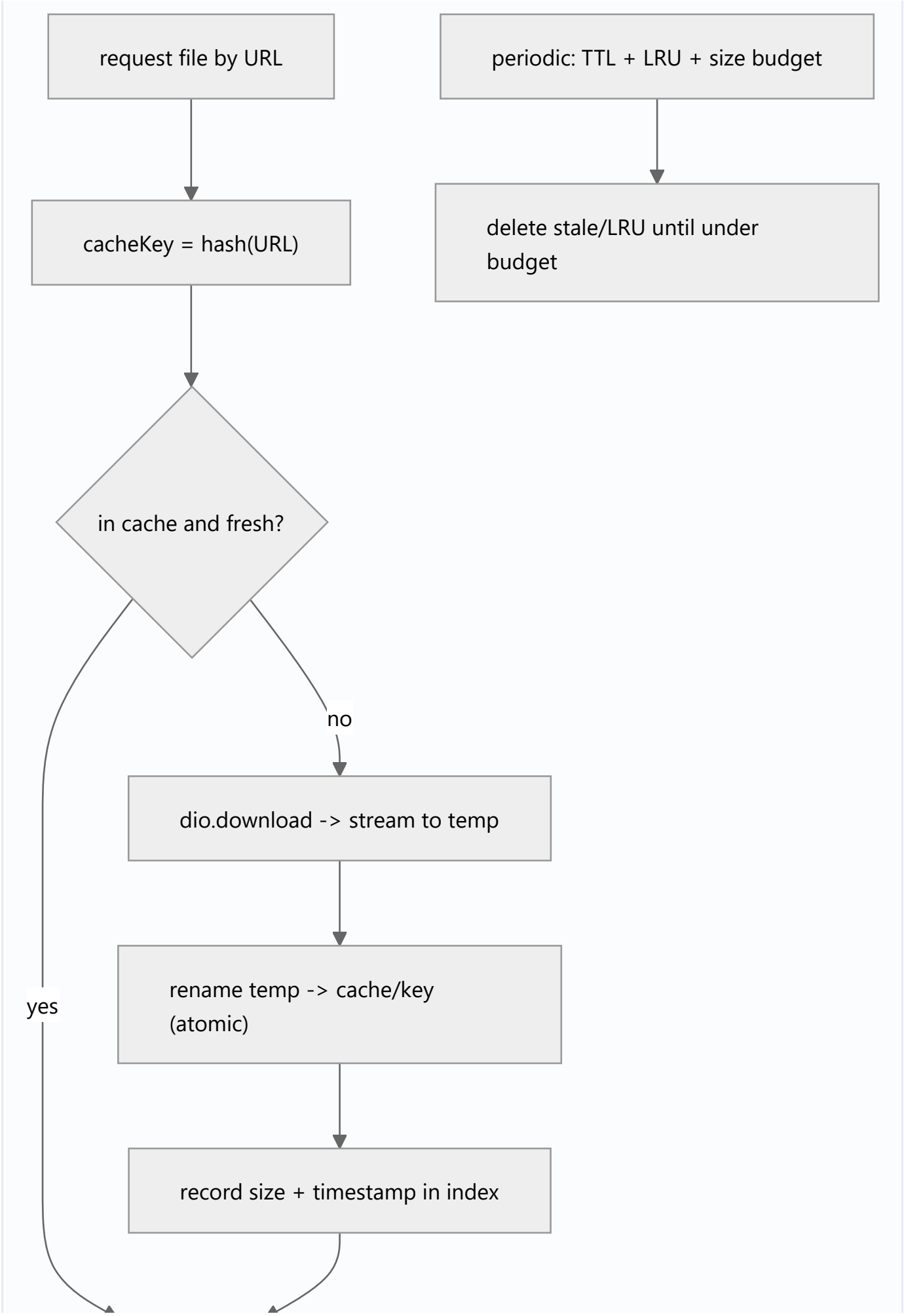
Real-world analogy

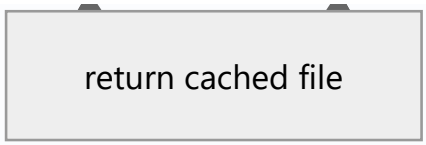
A file cache is a **local pantry** stocking items from the store (server): you check the pantry first (cache-first), only shop when it's empty/expired (fetch), and you **rotate stock** — toss expired items (TTL), remove the least-used when it's full (LRU/size budget). And since a power cut could spoil the pantry (OS clears cache), you never store your **only** copy of anything there.

Problem Statement

Download PDFs/images with a progress bar, serve repeat requests from cache (offline-capable), cap the cache at, say, 200 MB with a 7-day TTL, and clean up stale/oversized entries — without blocking the UI or OOM-ing on big files. You'll stream downloads and implement eviction.

Internal Working


request file by URL
periodic: TTL + LRU + size budget
cacheKey = hash(URL)
delete stale/LRU until under budget
in cache and fresh?
no
yes
dio.download -> stream to temp
rename temp -> cache/key (atomic)
record size + timestamp in index



- **Cache key**: a **stable id** from the URL (e.g., SHA-1 hash) + extension → filename in the cache dir (01_app_directories_and_paths.md). Deterministic lookup, no collisions.
- **Streamed download**: `dio.download(url, savePath, onReceiveProgress: (r, t) => ...)` streams to disk with **progress** (never loads the whole file in memory) (Module 16); write to a **temp file then rename** into place (atomic — 02_reading_writing_files.md) so a partial download isn't served.
- **Cache-first**: check cache (and freshness) before downloading; return the cached file if present + fresh → offline-capable, fast.
- **Freshness/validation**: TTL (age), and optionally HTTP validators (**ETag**/`Last-Modified` + conditional GET / `If-None-Match`) so unchanged files aren't re-downloaded (304).
- **Eviction policies** (combine):
  - **TTL**: delete entries older than N (stale).
  - **LRU**: track last-access; evict least-recently-used first.
  - **Size budget**: keep total under a cap; evict (LRU) until under it.
- **Index/metadata**: keep a small index (JSON/DB) of key → {size, createdAt, lastAccess} to drive eviction without stat-ing every file constantly.
- **OS may clear cache**: entries can vanish → always able to **re-fetch**; never store non-re-fetchable data in cache (01_app_directories_and_paths.md).
- **Concurrency/dedupe**: coalesce concurrent requests for the same URL into one download (avoid duplicate fetches).

Memory Representation

Only a chunk is in memory during streaming. The index (key→metadata) is small and in memory/DB; files live on disk. Total on-disk size is what you bound.

Compiler Behavior

Not applicable.

Runtime Behavior

Cache hits are instant/offline; misses stream from network (progress). Cleanup runs periodically (on launch/idle) to enforce TTL/LRU/budget. The OS may clear the whole cache under storage pressure.

Flutter Engine Behavior

Not applicable; networking/file I/O are `dio` / `dart:io`.

Dart VM Behavior

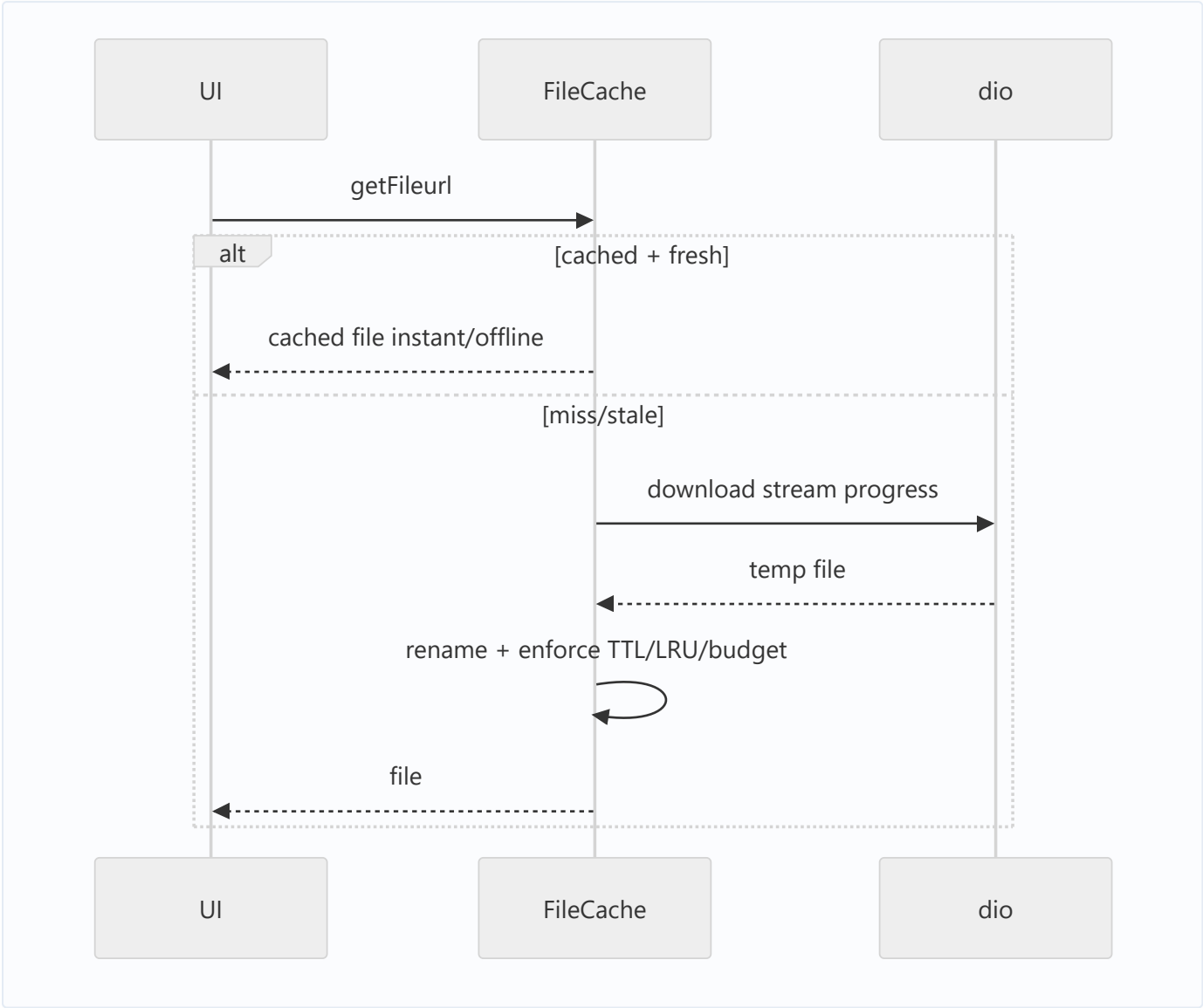
Streaming uses the I/O pool; hashing/large processing can be offloaded to an isolate if heavy.

Examples

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

class FileCache {
  final Dio _dio;
  final Directory _cacheDir;
  final Duration ttl;
  final int maxBytes;
  FileCache(this._dio, this._cacheDir, {this.ttl = const Duration(days: 7), this.maxBytes = 200 << 20});

  String _keyFor(String url) => sha1.convert(utf8.encode(url)).toString();

  // Cache-first: serve fresh cache, else stream-download (atomic), then serve
  Future<File> getFile(String url, {void Function(int, int)? onProgress}) async {
    final file = File(p.join(_cacheDir.path, _keyFor(url)));
    if (await file.exists() && await _isFresh(file)) {
      await file.setLastAccessed(DateTime.now());        // for LRU
      return file;
    }
    final tmp = '${file.path}.tmp';
    await _dio.download(url, tmp, onReceiveProgress: onProgress); // streamed, progress
    await File(tmp).rename(file.path);                    // atomic swap
    await _enforceBudget();
    return file;
  }

  Future<bool> _isFresh(File f) async =>
```

```
      DateTime.now().difference(await f.lastModified()) < ttl;


  // Evict stale (TTL) + LRU until under the size budget
  Future<void> _enforceBudget() async {
    final files = (await _cacheDir.list().toList()).whereType<File>().toList();
    // delete stale
    for (final f in files) {
      if (!await _isFresh(f)) { await f.delete(); }
    }
    var live = (await _cacheDir.list().toList()).whereType<File>().toList();
    var total = 0;
    for (final f in live) { total += await f.length(); }
    if (total <= maxBytes) return;
    live.sort((a, b) => a.statSync().accessed.compareTo(b.statSync().accessed)); // LRU first
    for (final f in live) {
      if (total <= maxBytes) break;
      total -= await f.length();
      await f.delete();
    }
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Buffering whole file in memory	OOM on large downloads	Stream to disk (`dio.download`)
Serving a partial download	Corruption on interrupt	temp + rename (atomic)
Unbounded cache	Fills storage	TTL + LRU + size budget
Storing only copy in cache	OS clears it	Cache must be re-fetchable
No dedupe of concurrent fetches	Duplicate downloads	Coalesce by key
Ignoring ETag/Last-Modified	Re-downloads unchanged files	Conditional GET (304)
Stat-ing all files constantly	Slow cleanup	Maintain a metadata index

Best Practices

- **Stream downloads to disk** (progress via `dio.download`), write **atomically** (temp + rename); key the cache by a **stable URL hash**.
- Serve **cache-first** (offline-capable) with **TTL** freshness (and **ETag/Last-Modified** conditional GETs to skip unchanged files).
- **Bound the cache** (TTL + LRU + size budget) with periodic cleanup; keep a **metadata index**; **dedupe** concurrent fetches.
- Treat cache as **re-fetchable** (never the only copy — OS clears it); wrap behind a repository (05_file_integration.md).

Performance

Streaming caps memory; cache-first turns repeat loads into instant disk reads (and enables offline). Bounded eviction keeps storage/backups sane. Conditional GETs save bandwidth. The whole point is spending disk to save network/battery — within a budget.

Advantages / Disadvantages

- **+** Fast repeat access, offline viewing, bandwidth/battery savings, progress UX, bounded storage.
- **−** Eviction/index complexity, must handle OS cache clearing + partial downloads + dedupe, cache invalidation subtleties.

Interview Questions

1. 🟢 **Why stream downloads instead of loading them into memory?** — Large files would OOM; streaming to disk holds only a chunk at a time (and enables progress).
2. 🟢 **What directory do cached downloads go in and why?** — A cache/temporary directory — they're re-fetchable; the OS may clear it, so never store the only copy there.
3. 🟡 **How do you key and look up cached files?** — By a stable hash of the URL (deterministic filename), checking existence + freshness before downloading (cache-first).
4. 🟡 **What eviction policies keep a file cache bounded?** — TTL (age), LRU (least-recently-used), and a total size budget — combined, with periodic cleanup.
5. 🟡 **How do you avoid re-downloading unchanged files?** — HTTP validators (ETag/`Last-Modified`) via conditional GET → 304 Not Modified.
6. 🔴 **How do you prevent serving a partially downloaded file?** — Download to a temp file and atomically rename into place only on success.
7. 🔴 **How do you handle concurrent requests for the same file?** — Coalesce/dedupe them into a single in-flight download keyed by the cache key.

Senior Engineer Tips

- Always stream + atomic-rename downloads; the two classic bugs are OOM on big files and serving a truncated file after an interrupted download.
- Bound the cache from day one (size budget + TTL + LRU) and keep a metadata index — an unbounded cache is a slow-burning storage/backup problem.
- Lean on `cached_network_image` for image widgets; build this general cache only for non-image files, and dedupe concurrent fetches.

Architect Perspective

A file cache is a bounded, re-fetchable performance layer between the app and the network. Encapsulating streamed download + atomic write + cache-first lookup + TTL/LRU/budget eviction behind a `FileCache`/repository gives fast, offline-capable, storage-safe file access — the same cache-first discipline as data caching, applied to binary files, and a building block for offline-first and media features (Module 19, Module 16, 05_file_integration.md).

Summary

- Stream downloads to disk (progress) with atomic writes; key cache by URL hash; serve cache-first (offline).
- Freshness via TTL (+ ETag/Last-Modified conditional GET); bound with TTL + LRU + size budget + periodic cleanup + metadata index.
- Cache is re-fetchable (OS clears it) — never the only copy; dedupe concurrent fetches; behind a repository.

Revision Notes

- `dio.download(url, tempPath, onReceiveProgress)` → atomic rename; cacheKey = hash(URL); cache-first lookup + TTL freshness (+ ETag/Last-Modified 304).
- Eviction: TTL + LRU + size budget; metadata index; periodic cleanup; dedupe concurrent fetches.
- Cache dir (OS may clear) → always re-fetchable; wrap behind repository; use `cached_network_image` for images.

Practice Questions

1. Why must downloads stream and write atomically?
2. What three signals bound a file cache?
3. How do you skip re-downloading an unchanged file?

Coding Questions

1. Implement cache-first `getFile(url)` with streamed download + atomic write.
2. Implement TTL + LRU + size-budget eviction.
3. Add ETag/Last-Modified conditional GET to skip unchanged files.

Mini Project

Bounded file cache (Flutter): Build a `FileCache` that streams downloads (progress) to a cache dir keyed by URL hash, writes atomically, serves cache-first (offline-capable), and evicts by TTL + LRU + a size budget with a metadata index and concurrent-fetch dedupe. Acceptance: large downloads stream (no OOM) + show progress; partial downloads never served (atomic); repeat/offline requests hit cache; cache stays under budget (TTL/LRU eviction); concurrent same-URL fetches coalesced; behind a repository.

File Integration (Capstone: A File Repository & Lifecycle Strategy)

> The maintainable shape: a single `FileRepository` that hides `path_provider`, `dart:io`, `file_picker`, `share_plus`, and the download cache behind an **intent-based API** (`saveDocument`, `readIndex`, `importAttachment`, `downloadCached`, `shareFile`, `evictCache`), enforcing one **directory & lifecycle strategy** (persistent vs cache), **atomic + async + streamed** I/O, and **cleanup** — so features never touch raw paths or worry about where data lives, backup bloat, corruption, or OOM.

Introduction

This module capstone composes directories, read/write, picking/sharing, and downloads/caching into one architecture. Scattered file code drifts into inconsistent directory choices, unsafe writes, and leaked temp files. A `FileRepository` centralizes the policy so the app expresses *what* it wants stored, not *how/where*. This file shows that design and an end-to-end document-manager flow.

Why this concept exists

File handling has many cross-cutting rules (correct directory per intent, atomic writes, streaming large files, persisting picks, bounded cache, cleanup, platform quirks). Left to each feature, they're applied inconsistently → data loss, jank, bloat. One repository — like data/network repositories — isolates these rules behind a boundary consistent with clean architecture (Module 40).

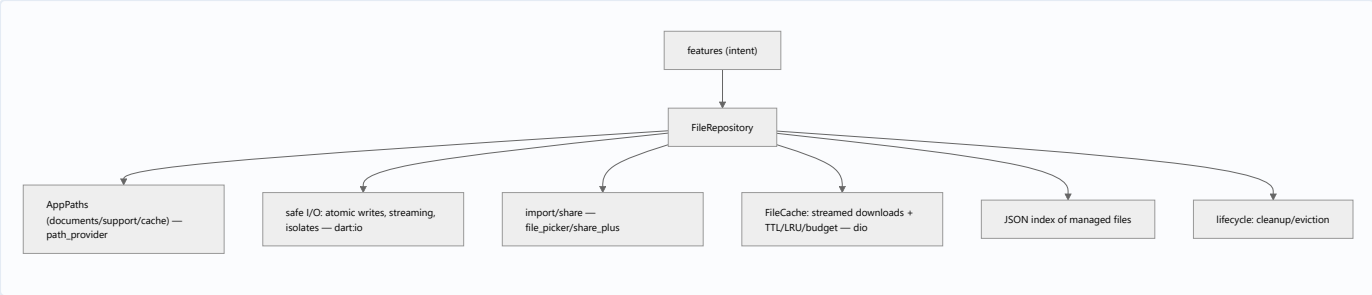
Real-world analogy

`FileRepository` is the **records department**: teams submit requests ("archive this document", "fetch that report", "shred old temp files") and the department applies **filing policy** (which cabinet, backup rules, retention/shredding schedule) uniformly. No employee decides on their own to stuff tax records in the recycling bin — the department enforces the rules.

Problem Statement

Build a document manager: save user documents (persistent), maintain a JSON index (atomic), import picked attachments (persist copies), download report PDFs with progress into a bounded cache, stream/parse large imports off the UI thread, share files, and evict stale cache — all behind one repository with a clear directory & lifecycle strategy. You'll compose every file in this module.

Internal Working



- **Intent-based API**: `saveDocument(name, bytes)`, `readIndex()/writeIndex()`, `importAttachment()`, `downloadCached(url, onProgress)`, `shareFile(id)`, `evictCache()`, `deleteDocument(id)`. Features never see paths.
- **Directory strategy (one place)**: persistent user data → **documents**; internal index/config → **application support**; downloads/thumbnails → **cache**; scratch → **temp** (01_app_directories_and_paths.md). Backup-aware (avoid large blobs in iOS documents).
- **Safe I/O**: **atomic writes** (temp+rename), **async** everywhere, **streaming** for large files, **isolate** offload for heavy parse (02_reading_writing_files.md).
- **Import/share**: pick → **copy into app storage** → index; share via `share_plus` (03_file_picker_and_sharing.md).
- **Cache**: streamed, bounded, cache-first `FileCache` for downloads (04_downloads_and_caching.md); re-fetchable, evictable.
- **Index + lifecycle**: a small JSON/DB index of managed files (id → path/dir/metadata) drives listing, sharing, deletion, and cleanup; run **eviction/cleanup** on launch/idle.
- **Testability**: depend on abstractions (paths, cache, picker, sharer); unit-test the repository with a temp dir + fakes (no device).

Memory Representation

The repository holds cached directories + the file index (small). Files live on disk in their strategy-assigned dirs; streaming/isolates keep memory bounded during large ops.

Compiler Behavior

Compiles against abstractions (paths/cache/picker) — mockable; isolate entry points top-level where needed.

Runtime Behavior

The repository routes each intent to the right directory + safe I/O path; downloads stream + cache; cleanup enforces retention/budget; imports persist copies. Cache/temp may be cleared by the OS (re-fetchable), documents/support persist.

Flutter Engine Behavior

Not applicable beyond the plugins (path_provider/file_picker/share) bridging natively.

Dart VM Behavior

Heavy parsing offloaded to isolates; I/O async on the I/O pool; index kept in the UI isolate.

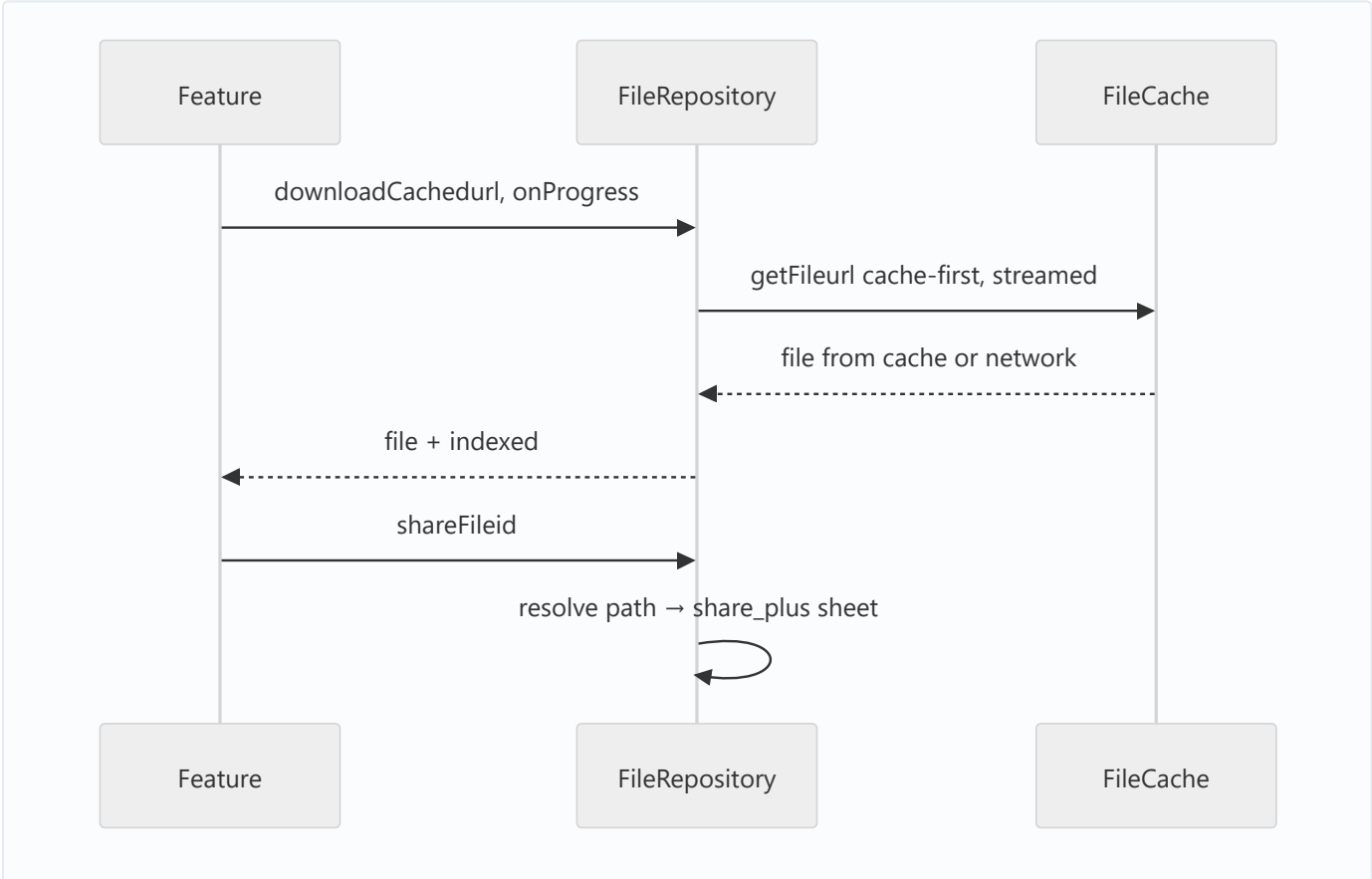
Examples

```
// Intent-based repository — features never touch paths/plugins
abstract class FileRepository {
  Future<ManagedFile> saveDocument(String name, List<int> bytes);   // persistent
  Future<Map<String, dynamic>> readIndex();
  Future<void> writeIndex(Map<String, dynamic> index);              // atomic
  Future<ManagedFile?> importAttachment();                          // pick + persist
  Future<File> downloadCached(String url, {void Function(int,int)? onProgress}); // cached
  Future<void> shareFile(String id);
  Future<void> evictCache();                                        // TTL/LRU/budget
}

class FileRepositoryImpl implements FileRepository {
  final AppPaths paths;        // documents/support/cache (path_provider)
  final FileCache cache;       // streamed downloads + eviction (dio)
  final Picker picker; final Sharer sharer;
  FileRepositoryImpl(this.paths, this.cache, this.picker, this.sharer);

  @override
  Future<ManagedFile> saveDocument(String name, List<int> bytes) async {
    final file = await paths.documentFile(name);       // persistent dir (strategy)
    await _atomicWriteBytes(file, bytes);              // safe write
    return _index(file);                               // record in index
  }

  @override
  Future<ManagedFile?> importAttachment() async {
    final picked = await picker.pickFile();            // system picker
    if (picked == null) return null;
    final dest = await paths.documentFile(p.basename(picked.path));
    await File(picked.path).copy(dest.path);           // persist the pick
```

```
    return _index(dest);
  }

  @override
  Future<File> downloadCached(String url, {void Function(int,int)? onProgress}) =>
      cache.getFile(url, onProgress: onProgress);      // cache-first, streamed, bounded

  // ... readIndex/writeIndex (atomic), shareFile (share_plus), evictCache (cache)
}
```

```
// Unit test with a temp dir + fakes — no device
test('saveDocument persists + indexes', () async {
  final repo = FileRepositoryImpl(FakePaths(tempDir), FakeCache(), FakePicker(), FakeSharer());
  final f = await repo.saveDocument('a.txt', utf8.encode('hi'));
  expect(await File(f.path).readAsString(), 'hi');
  expect((await repo.readIndex()).containsKey(f.id), isTrue);
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Paths/plugins in features	Inconsistent, untestable	Intent-based repository
Inconsistent directory choices	Data loss/backup bloat	One directory strategy
Non-atomic/blocking writes	Corruption/jank	Atomic + async + streaming
Not persisting picked files	Broken references	Copy into app storage + index
Unbounded cache / no cleanup	Storage bloat	TTL/LRU/budget + scheduled eviction
No index of managed files	Can't list/delete/clean	Maintain a file index
Untestable (real device)	Slow/fragile	Temp dir + fakes

Best Practices

- Expose an **intent-based** `FileRepository`; enforce **one directory & lifecycle strategy** (persistent vs cache, backup-aware) inside it.
- Use **atomic + async + streamed** I/O and **isolate** offload; **persist picked files** (copy + index); **share** via the OS sheet.
- Keep a **file index** driving listing/deletion/cleanup; run **eviction** (cache) and **cleanup** (temp) on launch/idle.
- Depend on **abstractions** (paths/cache/picker/sharer) for **testability** (temp dir + fakes); document the strategy.

Performance

The repository centralizes the performance-critical choices (streaming, async, isolates, bounded cache) so they're applied consistently. Correct directory strategy avoids re-fetch/backup costs; cleanup keeps storage bounded. No overhead beyond the underlying I/O.

Advantages / Disadvantages

- **+** Consistent, safe, testable file access; one place for directory/lifecycle policy; features stay simple; no data loss/bloat/corruption.
- **−** Upfront structure/boilerplate, index maintenance, requires discipline to route everything through the repository.

Interview Questions

1. 🟢 **Why wrap file handling in a repository?** — To enforce one directory/lifecycle strategy, safe I/O, and cleanup consistently, keeping paths/plugins out of features and enabling tests.

2. 🟢 **What does an intent-based file API look like?** — Methods like `saveDocument` / `importAttachment` / `downloadCached` / `shareFile` / `evictCache` — features express *what*, not *where/how*.
3. 🟡 **How is the directory strategy applied?** — Persistent user data → documents, internal → support, downloads → cache, scratch → temp; centralized and backup-aware.
4. 🟡 **How do you keep the repository testable?** — Depend on abstractions (paths/cache/picker/sharer) and inject fakes + a temp directory — no device needed.
5. 🟡 **Why maintain a file index?** — To list, share, delete, and clean up managed files without scanning the filesystem, and to store per-file metadata.
6. 🔴 **How do all pieces cooperate for a downloaded, shareable report?** — `downloadCached` streams into the bounded cache (cache-first) → indexed → `shareFile` resolves the path → share sheet; eviction reclaims space later.
7. 🔴 **Where are correctness/performance concerns handled?** — In the repository: atomic writes (correctness), async/streaming/isolates (performance), TTL/LRU/budget (storage) — applied uniformly.

Senior Engineer Tips

- Route *all* file access through the repository and write the directory strategy down; the moment features call `path_provider` directly, consistency erodes.
- Keep an index + scheduled cleanup from day one; retrofitting retention onto an unmanaged pile of files is painful.
- Make the repository testable with a temp dir + fakes so file logic runs in CI — file bugs are otherwise device-only and slow to catch.

Architect Perspective

File integration applies the app's boundary discipline to the filesystem: one repository owns directory policy, safe/efficient I/O, import/share, bounded caching, and cleanup, exposing intent to features. This eliminates the whole class of file bugs (lost data, backup bloat, corruption, OOM, leaks), stays testable in CI, and composes with offline-first, media, and export features — the same clean-architecture seam used for data and networking (Module 40, Module 19, Module 15).

Summary

- One intent-based `FileRepository` enforces the directory & lifecycle strategy and safe/efficient I/O; features never touch paths/plugins.
- Atomic + async + streamed I/O (+ isolates), persist picks, bounded cache, file index, scheduled cleanup.

- Depend on abstractions for testability; document the strategy; composes with offline/media/export.

Revision Notes

- `FileRepository` intent API (`saveDocument` / `importAttachment` / `downloadCached` / `shareFile` / `evictCache`); directory strategy centralized (documents/support/cache/temp), backup-aware.
- Atomic + async + streamed I/O + isolate offload; persist picked files (copy+index); share via OS sheet; bounded `FileCache` .
- File index drives list/delete/cleanup; eviction/cleanup on launch/idle; abstractions + temp dir + fakes for tests.

Practice Questions

1. What belongs in the file repository vs the feature?
2. How is the directory strategy kept consistent?
3. How do you make file logic testable in CI?

Coding Questions

1. Design a `FileRepository` interface + impl over paths/cache/picker/sharer.
2. Implement `saveDocument` (atomic) + `importAttachment` (persist pick) + index.
3. Unit-test the repository with a temp dir + fakes.

Mini Project

Document manager (capstone — Flutter): Build a `FileRepository` composing `AppPaths` (directory strategy), safe I/O (atomic/streamed/isolate), `file_picker` / `share_plus` (import/share), and a bounded `FileCache` (downloads) behind an intent API, maintaining a JSON file index and running eviction/cleanup. Provide abstractions + fakes and unit tests (temp dir). Acceptance: features use intents only; correct directory per file type (backup-aware); atomic/async/streamed I/O; picked files persisted + indexed; downloads cached + bounded; share works; cleanup enforced; unit-tested in CI (no device); runs end-to-end on device.