35 · PDF

Flutter Practice Notes

Contents

35 · PDF

Introduction

This module covers generating, viewing, and printing PDFs in Flutter with the `pdf` package (document/page/widget model — a **separate widget tree** from Flutter's) and the `printing` package (preview, print, share, save). It walks from **generation basics** (text, styling, the `pw.*` widgets), through **layouts/tables/assets** (multi-page tables, images, custom fonts, headers/footers), **viewing & printing** (preview dialog, OS print, share/save), **data-driven documents** (invoices/reports from your models, reusable templates), to a capstone invoice generator. It builds on file handling (Module 34) and sharing.

Why this module exists

Business apps constantly produce documents: invoices, receipts, reports, tickets, certificates. Users expect to preview, print, share, and save them as PDFs. The `pdf` package uses its **own widget system** (`pw.Widget` , not Flutter widgets) rendered to a page-based, paginated document — a distinct mental model. Generating correct, paginated, styled documents from live data (and wiring print/share/save) is a common, high-value skill that's easy to get subtly wrong (pagination, fonts, memory).

Module map

#	File	Topic	Level
1	01_pdf_generation_basics.md	`pdf` package model: `Document` / `Page` / `pw.*` widgets, text/styling	🔵
2	02_layouts_tables_and_assets.md	Multi-page tables, images, custom fonts, headers/footers	🟡
3	03_viewing_and_printing.md	`printing` : preview, OS print, share, save PDFs	🟡
4	04_data_driven_documents.md	Invoices/reports from models, reusable templates, dynamic content	🔴
5	05_pdf_integration.md	Capstone: invoice generator behind a service	🔴

> **Cross-references:** File saving/sharing: Module 34. Isolates (heavy generation): 02 · isolates. Excel/spreadsheets: Module 36. Networking (fetch report data): Module 16. Custom painting (for comparison): Module 23.

Prerequisites

34 File Handling (save/share), basic layout intuition (Flutter widgets help — the `pw.*` API mirrors them), 02 Advanced Dart (async/isolates for heavy docs).

What you'll be able to do after this module

- Generate PDFs with the `pdf` package's document/page/widget model and style text.
- Build multi-page documents with paginated tables, images, custom fonts, and headers/footers.
- Preview, print (OS dialog), share, and save PDFs with the `printing` package.
- Produce data-driven documents (invoices/reports) from your models via reusable templates.
- Wrap PDF generation behind a service, generating heavy docs off the UI thread.

Capstone

Invoice slice: An invoice generator that builds a styled, multi-page PDF from order data (logo, line-item table with pagination, totals, header/footer with page numbers, custom fonts), then lets the user preview, print, share, and save it — generation offloaded to an isolate and wrapped behind a `PdfService`.

Summary

PDF work = the `pdf` package's own paginated widget tree (`pw.*`) for generation + the `printing` package for preview/print/share/save. Build reusable, data-driven templates (invoices/reports), mind pagination/fonts/memory, offload heavy generation, and wrap it behind a service.

PDF Generation Basics (The `pdf` Package Model)

> The `pdf` package builds documents with its **own widget tree** — `pw.Widget` (not Flutter's `Widget`) — laid out onto fixed-size **pages**: you create a `pw.Document`, `addPage()` a `pw.Page` (or `MultiPage`) whose `build` returns `pw.*` widgets (`Text`, `Column`, `Row`, `Container`, `Table`), style with `pw.TextStyle` / `PdfColor`, and finally `doc.save()` to bytes. It *looks* like Flutter but is a **parallel API** rendered to paginated, print-resolution output — so import it prefixed (`as pw`) and don't mix it with `material` / `widgets`.

Introduction

Before layouts or printing, you must grasp the `pdf` package's model: a document of pages, each built from `pw.*` widgets, producing bytes. This file covers the core objects, text/styling, the Flutter-lookalike-but-separate API, and generating a first document — the foundation for everything else.

Why this concept exists

A PDF is a fixed-layout, paginated, print-oriented format — fundamentally different from Flutter's dynamic screen UI. The `pdf` package provides a familiar widget-style API (so Flutter devs feel at home) but targets pages/points/print resolution, with its own layout/pagination rules. The separation avoids conflating on-screen widgets with document widgets.

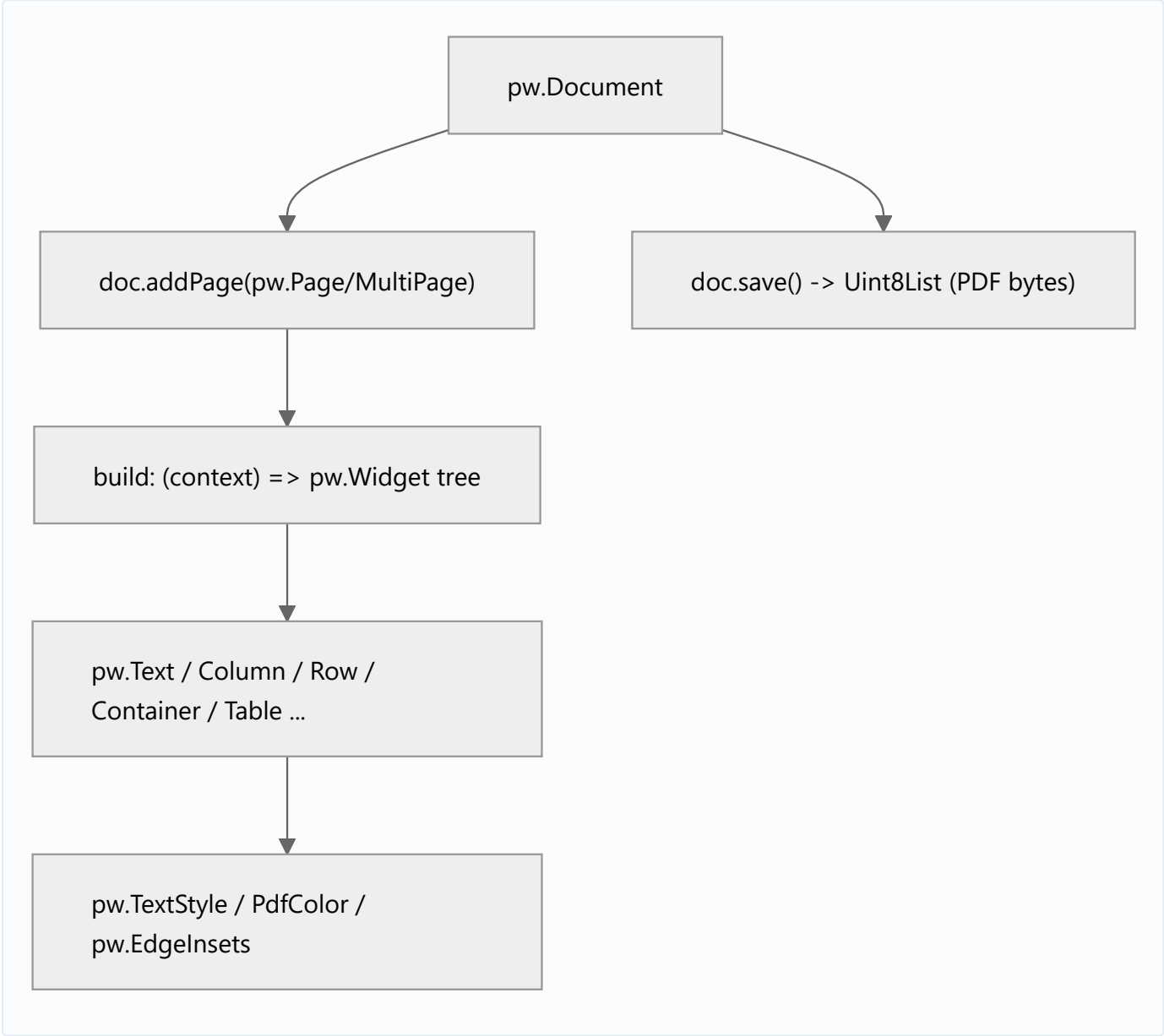
Real-world analogy

Flutter widgets are for a **living, resizable display board**; `pw.*` widgets are for **typesetting a printed page** — same vocabulary (columns, rows, text), different medium with fixed page sizes and margins. It's like using a word processor whose toolbar mirrors your design app's, but the output is paper, not a screen.

Problem Statement

Generate a one-page PDF with a title, a styled paragraph, and a two-column layout, then produce its bytes to save/share. You'll create a `Document`, add a `Page` with `pw.*` widgets, style text, and `save()`.

Internal Working



- `pw.Document`: the container; you `addPage(...)` one or more pages, then `await doc.save()` → `Uint8List` (the PDF bytes) to write/share/print.
- **Pages**: `pw.Page` (single, fixed page) vs `pw.MultiPage` (auto-paginates overflowing content across pages — essential for long content/tables — 02_layouts_tables_and_assets.md). Set `pageFormat` (`PdfPageFormat.a4`, `.letter`), `margin`, orientation.
- `pw.*` **widgets** mirror Flutter: `Text`, `RichText`, `Column`, `Row`, `Container`, `Padding`, `Center`, `Expanded`, `SizedBox`, `Divider`, `Table`, `Wrap`, `Stack`. **Not interchangeable** with `material`/`widgets` — import `package:pdf/widgets.dart as pw`.
- **Styling**: `pw.TextStyle(fontSize, fontWeight, color: PdfColors.blue900, font: ...)`, `PdfColor`/`PdfColors`, `pw.EdgeInsets`, `pw.BoxDecoration`. Units are **PDF points** (72/inch), not logical pixels.
- `build` **callback**: `(pw.Context context) => ...` returns the page's widget tree; `MultiPage` build returns a `List<pw.Widget>` it flows across pages.

- **Fonts**: the default font is limited (often no bold/unicode); load a real font for proper text (02_layouts_tables_and_assets.md).
- **Async + bytes**: `save()` is async; the resulting bytes are what you persist (Module 34) or hand to `printing` (03_viewing_and_printing.md).

Memory Representation

The document builds an in-memory widget tree, then serializes to a `Uint8List`. Large docs (many pages/images) hold significant memory during generation — generate off the UI thread for big ones (04_data_driven_documents.md).

Compiler Behavior

Not applicable; `pw.*` is a normal Dart API.

Runtime Behavior

`save()` lays out all pages and serializes to bytes (can be slow for large/complex docs). `MultiPage` computes pagination at build time.

Flutter Engine Behavior

None directly — the `pdf` package renders to the PDF format, not the Flutter engine. (Viewing a PDF on screen is a separate concern — 03_viewing_and_printing.md.)

Dart VM Behavior

Pure Dart layout/serialization; heavy generation is CPU-bound → offload to an isolate for large docs (02 · isolates).

Examples

```
import 'package:pdf/pdf.dart';
import 'package:pdf/widgets.dart' as pw;      // NOTE: separate widget API, prefixed `pw`
import 'dart:typed_data';

Future<Uint8List> buildSimplePdf() async {
  final doc = pw.Document();

  doc.addPage(
    pw.Page(
```

```
      pageFormat: PdfPageFormat.a4,
      margin: const pw.EdgeInsets.all(32),
      build: (pw.Context context) {
        return pw.Column(
          crossAxisAlignment: pw.CrossAxisAlignment.start,
          children: [
            pw.Text('Report Title',
                style: pw.TextStyle(fontSize: 24, fontWeight: pw.FontWeight.bold,
                    color: PdfColors.blue900)),
            pw.SizedBox(height: 12),
            pw.Text('This is a styled paragraph rendered to a PDF page. '
                'The pw.* widgets mirror Flutter but target print output.'),
            pw.SizedBox(height: 16),
            pw.Row(children: [
              pw.Expanded(child: pw.Text('Left column')),
              pw.Expanded(child: pw.Text('Right column')),
            ]),
          ],
        );
      },
    ),
  );

  return doc.save();                          // async -> PDF bytes (save/share/print)
}
```

Diagrams

Common Mistakes

Mistake	Why	Fix
Mixing Flutter widgets with `pw.*`	Different, incompatible APIs	Use only `pw.*` (import `as pw`)
Single `Page` for long content	Overflow clipped	Use `pw.MultiPage` (auto-paginate)
Relying on the default font	Missing bold/unicode/glyphs	Load a real font
Assuming logical pixels	PDF uses points (72/in)	Size in points; use `PdfPageFormat`
Generating huge docs on UI thread	Jank/freeze	Offload to an isolate
Forgetting `await save()`	Bytes not produced	`await doc.save()`

Best Practices

- Import the PDF API prefixed (`as pw`) and use **only** `pw.*` widgets; never mix with `material` / `widgets`.
- Use `MultiPage` for anything that can overflow; set `pageFormat` / `margin` explicitly; think in **points**.
- **Load real fonts** for correct bold/unicode; keep the widget tree simple/reusable.
- Treat `save()` as **async + potentially heavy** — **offload large docs to an isolate**; hand the bytes to file/print/share layers.

Performance

Generation is CPU/memory-bound and scales with pages/images/complexity. Small docs are instant; large ones (long tables, many images) should be built in an isolate to avoid jank, and images should be sized/compressed (02_layouts_tables_and_assets.md).

Advantages / Disadvantages

- **+** Familiar widget-style API, precise paginated print output, cross-platform, produces standard PDF bytes.
- **−** Separate/parallel API (not Flutter widgets), points-not-pixels, default-font limitations, memory/CPU for large docs, pagination nuances.

Interview Questions

1. 🟢 **Are `pdf` package widgets the same as Flutter widgets?** — No — `pw.*` is a separate, parallel widget API for paginated print output; import it `as pw` and don't mix with `material` / `widgets`.
2. 🟢 **What does `doc.save()` return?** — The PDF as a `Uint8List` (bytes) you save, share, or print — it's async.

3. 🟡 `Page` **vs** `MultiPage`**?** — `Page` is a single fixed page (content can overflow/clip); `MultiPage` auto-paginates overflowing content across pages.
4. 🟡 **Why load a custom font?** — The default font is limited (no bold/unicode/many glyphs); a real font ensures correct text rendering.
5. 🟡 **What units does the** `pdf` **package use?** — PDF points (72 per inch) via `PdfPageFormat`, not logical pixels.
6. 🔴 **When and why offload generation to an isolate?** — For large/complex docs — layout+serialization is CPU-bound and would jank the UI thread.
7. 🔴 **What's the overall generation flow?** — Build a `pw.*` tree → add to `pw.Page` / `MultiPage` in a `Document` → `save()` to bytes → hand to file/print/share.

Senior Engineer Tips

- Keep the mental separation crisp: `pw.*` for documents, Flutter widgets for screens; a `pw` prefix everywhere prevents accidental mixing.
- Default to `MultiPage` and load a font on day one — the two most common "it worked in the demo, broke in prod" PDF bugs are clipped content and missing glyphs.
- Wrap generation so large docs go to an isolate transparently; users notice a frozen UI while a 50-page report renders.

Architect Perspective

The `pdf` package is a document-rendering domain with its own model; treating it as such (separate `pw.*` trees, reusable builders, isolate-offloaded generation, bytes handed to file/print layers) keeps document generation clean and testable, and sets up data-driven templates. It's the generation half; the `printing` package is the presentation half (03_viewing_and_printing.md, 04_data_driven_documents.md).

Summary

- `pdf` package: `pw.Document` + `addPage(pw.Page/MultiPage)` whose `build` returns `pw.*` widgets; style with `pw.TextStyle` / `PdfColor`; `save()` → bytes.
- Separate API from Flutter (import `as pw`), points not pixels, `MultiPage` for overflow, load real fonts.
- Generation is CPU/memory-bound — offload large docs; hand bytes to file/print/share.

Revision Notes

- `import 'package:pdf/widgets.dart' as pw;` — `pw.Document`, `pw.Page` / `pw.MultiPage`, `pw.Text/Column/Row/Container/Table`.

- Style: `pw.TextStyle(fontSize, fontWeight, color)`, `PdfColors`, `pw.EdgeInsets`; `PdfPageFormat.a4/.letter`; units = points.
- `await doc.save()` → `Uint8List`; `MultiPage` for pagination; custom fonts for bold/unicode; isolate for big docs.

Practice Questions

1. Why can't you use Flutter widgets in a PDF `build`?
2. When must you use `MultiPage`?
3. Why load a custom font?

Coding Questions

1. Generate a one-page A4 PDF with a styled title + paragraph, returning bytes.
2. Add a two-column row layout with `Expanded`.
3. Offload generation to an isolate via `compute`.

Mini Project

First PDF (Flutter): Generate a styled one-page A4 document (title, paragraph, two-column layout) with explicit margins/format, returning bytes via `save()`, and offload generation to an isolate. Acceptance: uses only `pw.*` (prefixed); A4 + margins set; styled text; returns `Uint8List`; generation off the UI thread; renders correctly (verify by saving/opening).

Layouts, Tables & Assets (Multi-Page, Fonts, Headers/Footers)

> Real documents need **paginated tables** (`pw.Table` / `TableHelper.fromTextArray` inside a `MultiPage` that flows rows across pages), **custom fonts** (load a `.ttf` as a `pw.Font` — the default font lacks bold/unicode), **images** (`pw.Image` from bytes, sized/compressed), and **repeating headers/footers with page numbers** (`MultiPage` 's `header` / `footer` callbacks give `context.pageNumber` / `pagesCount`). Getting these right is the difference between a demo and a printable report.

Introduction

This file covers the layout building blocks for professional documents: tables that paginate, custom fonts (incl. a fallback/theme), embedding images/logos, and headers/footers with page numbering. These compose into invoices/reports (04_data_driven_documents.md).

Why this concept exists

Business documents are long and structured: line-item tables spanning pages, a logo, branded fonts, and "Page X of Y" footers. `pw.MultiPage` handles pagination; `pw.Table` structures rows/columns; `pw.Font` / `ThemeData` handle typography; `pw.Image` embeds assets. Without these you get clipped content, tofu (□) glyphs, or unpaginated walls of text.

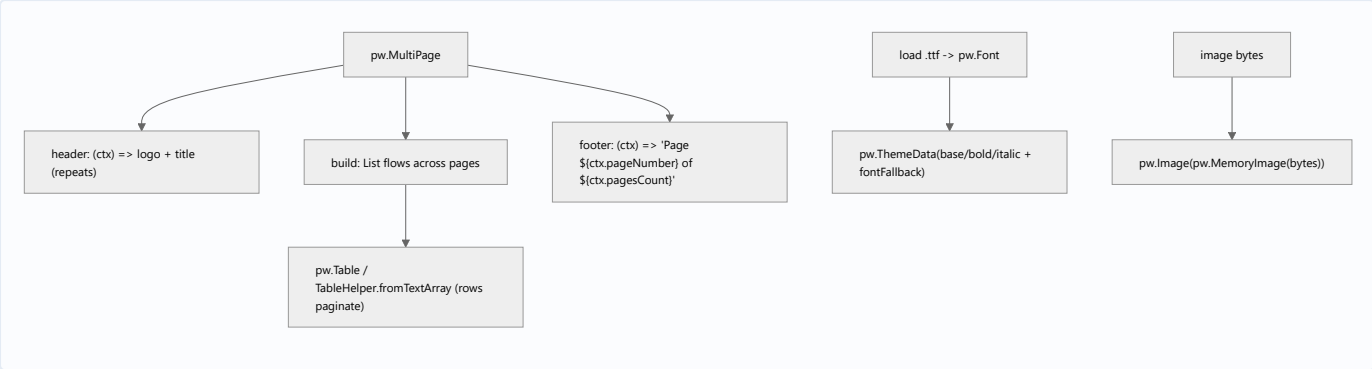
Real-world analogy

This is **typesetting a printed report**: the table is your **spreadsheet grid** that automatically continues onto the next page when it runs off the bottom (pagination), the custom font is your **brand's typeface**, the logo is a **letterhead image**, and the header/footer is the **running title and page number** printed on every sheet.

Problem Statement

Build a multi-page report with a logo header, a long line-item table that paginates with repeating column headers, branded fonts (incl. bold + a rupee/unicode symbol), and a footer showing "Page X of Y." You'll use `MultiPage` , `Table` , `Font` , `Image` , and header/footer callbacks.

Internal Working



- `MultiPage`: `build` returns a `List<pw.Widget>` that **flows across pages** automatically; `header` / `footer` callbacks (`(pw.Context ctx) => pw.Widget`) **repeat on every page**. Set `maxPages`, `pageTheme`.
- **Tables**: `pw.Table` (rows of `pw.TableRow` with `children`) or the convenience `TableHelper.fromTextArray(headers:, data:)` — inside `MultiPage`, table rows **paginate** and you can **repeat the header row** on each page. Control column widths (`columnWidths`, `FixedColumnWidth` / `FlexColumnWidth`), borders, cell alignment/padding.
- **Fonts**: load a `.ttf` (asset/bytes) → `pw.Font.ttf(byteData)`; supply **base/bold/italic** variants and a `fontFallback` list (for unicode/emoji/CJK) via a `pw.ThemeData` on the page theme so all text renders correctly (no tofu). `PdfGoogleFonts` (from `printing`) can fetch fonts.
- **Images**: `pw.Image(pw.MemoryImage(bytes))` (PNG/JPEG bytes) — **size/compress** before embedding (raw high-res images bloat the PDF + memory). SVG via `pw.SvgImage`.
- **Headers/footers + page numbers**: in the callbacks use `ctx.pageNumber` and `ctx.pagesCount` for "Page X of Y"; keep them lightweight (they render per page).
- **Layout widgets**: `pw.Column/Row/Wrap/Container/Padding/Align/Spacer/Divider`, `pw.Table`, `pw.GridView` — compose like Flutter but paginate within `MultiPage`.

Memory Representation

Embedded images live in the document bytes — the biggest size driver; downscale/compress. Fonts are embedded once. Large tables build many widgets — fine, but generation cost scales with rows.

Compiler Behavior

Not applicable.

Runtime Behavior

`MultiPage` computes where content breaks at build time; repeating headers/footers render per page. Oversized images/many rows increase generation time and file size.

Flutter Engine Behavior

None; rendered to PDF, not the engine.

Dart VM Behavior

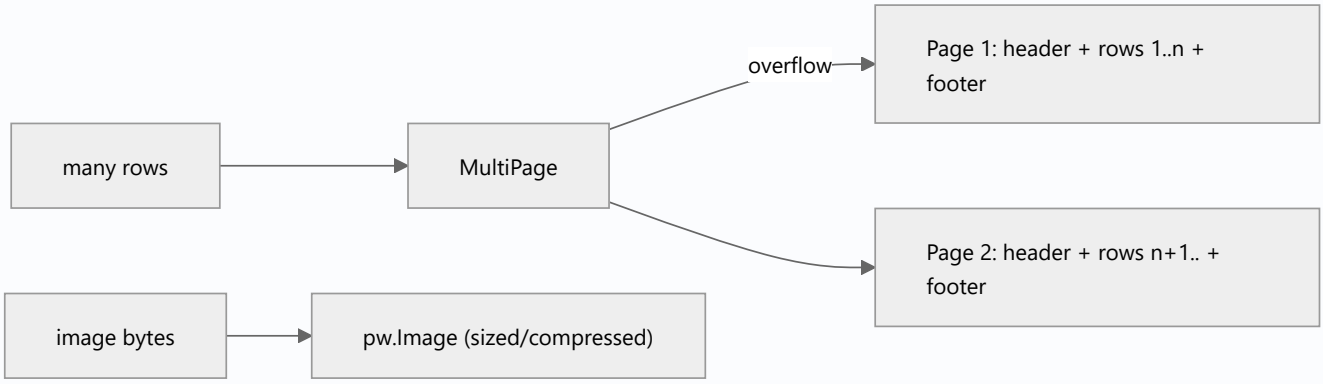
CPU-bound layout scales with content; offload large reports to an isolate (02 · isolates).

Examples

```
import 'package:pdf/pdf.dart';
import 'package:pdf/widgets.dart' as pw;
import 'dart:typed_data';

Future<Uint8List> buildReport(List<List<String>> rows, Uint8List logoBytes,
    Uint8List regular, Uint8List bold) async {
  final theme = pw.ThemeData.withFont(
    base: pw.Font.ttf(regular.buffer.asByteData()),
    bold: pw.Font.ttf(bold.buffer.asByteData()),
    // fontFallback: [pw.Font.ttf(unicodeFont)],  // for ₹/emoji/CJK
  );
  final doc = pw.Document(theme: theme);

  doc.addPage(pw.MultiPage(
    pageFormat: PdfPageFormat.a4,
    margin: const pw.EdgeInsets.all(32),
    header: (ctx) => pw.Row(
      mainAxisAlignment: pw.MainAxisAlignment.spaceBetween,
      children: [
        pw.Text('Sales Report', style: pw.TextStyle(fontSize: 18, fontWeight: pw.FontWeight.bold)),
        pw.Image(pw.MemoryImage(logoBytes), width: 60),          // sized logo
      ],
    ),
    footer: (ctx) => pw.Align(
      alignment: pw.Alignment.centerRight,
      child: pw.Text('Page ${ctx.pageNumber} of ${ctx.pagesCount}',
```

```
          style: const pw.TextStyle(fontSize: 10)),              // page X of Y
    ),
    build: (ctx) => [
      pw.SizedBox(height: 12),
      pw.TableHelper.fromTextArray(                              // paginates + repeats header
        headers: ['#', 'Item', 'Qty', 'Amount'],
        data: rows,
        headerStyle: pw.TextStyle(fontWeight: pw.FontWeight.bold),
        cellAlignments: {2: pw.Alignment.centerRight, 3: pw.Alignment.centerRight},
        columnWidths: {0: const pw.FixedColumnWidth(30)},
      ),
    ],
  ));

  return doc.save();
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Table in a single `Page`	Overflow clipped	`MultiPage` (rows paginate)
No repeating table header	Later pages confusing	Repeat header row in `MultiPage`
Default font for ₹/emoji/CJK	Tofu (□) glyphs	Load font + `fontFallback`
Embedding raw high-res images	Huge file/memory	Downscale/compress first
Heavy header/footer	Slow (renders per page)	Keep them light
No column widths	Ugly/overflowing columns	Set `columnWidths`/alignments
Big report on UI thread	Jank	Isolate

Best Practices

- Use `MultiPage` with **paginating tables** (`TableHelper.fromTextArray`) and **repeating headers**; set **column widths/alignments**.
- Load **base/bold/italic fonts + `fontFallback`** via `ThemeData` (no tofu); **size/compress images** before embedding.
- Add **headers/footers with `pageNumber` / `pagesCount`**; keep them lightweight; think in **points**.
- **Offload large reports** to an isolate; keep builders **reusable/composable** for 04_data_driven_documents.md.

Performance

File size and generation time are dominated by **images** (compress!) and **row count**. Use `MultiPage` (not manual splitting), keep headers/footers cheap, and isolate large reports. Font embedding is a one-time cost.

Advantages / Disadvantages

- **+** Professional paginated documents (tables/logos/fonts/page numbers), automatic pagination, print-quality.
- **−** Font/fallback setup, image-size discipline, pagination/column-width tuning, generation cost for large docs.

Interview Questions

1. 🟢 **How do you make a long table span multiple pages?** — Put it in a `MultiPage`; `pw.Table` / `TableHelper` rows paginate automatically (and you can repeat the header row).

2. 🟢 **How do you add "Page X of Y"?** — In `MultiPage`'s `footer` callback use `context.pageNumber` and `context.pagesCount`.
3. 🟡 **Why load custom fonts + a fallback?** — The default font lacks bold/unicode; base/bold/italic + `fontFallback` prevent missing-glyph "tofu" (₹, emoji, CJK).
4. 🟡 **How do you embed a logo and keep the file small?** — `pw.Image(pw.MemoryImage(bytes))`, sized/compressed before embedding (images dominate file size).
5. 🟡 **How do repeating headers/footers behave?** — They render on every page via the `header` / `footer` callbacks — keep them lightweight.
6. 🔴 **What drives PDF file size and generation time?** — Primarily embedded image resolution and row/content count; compress images and isolate large reports.
7. 🔴 **How do you control table columns?** — `columnWidths` (`Fixed` / `Flex`) and `cellAlignments`/borders on `Table` / `TableHelper`.

Senior Engineer Tips

- Configure a `ThemeData` with proper fonts + fallback once and reuse it across all documents — it eliminates the most common (and embarrassing) PDF bug: missing glyphs.
- Always downscale/compress images before embedding; a single full-res photo can balloon a report to tens of MB.
- Build reusable header/footer/table components; every business document reuses the same letterhead + line-item grid.

Architect Perspective

Layouts/tables/assets are the reusable component layer of document generation: a themed font setup, a standard header/footer, and a paginating table become the toolkit that data-driven templates compose. Keeping images compressed, generation isolate-offloaded, and components reusable makes reports maintainable and performant — feeding invoices/reports and the capstone service (04_data_driven_documents.md, 05_pdf_integration.md).

Summary

- `MultiPage` + paginating `Table` / `TableHelper` (repeating header) for long structured content; column widths/alignments.
- Custom base/bold/italic fonts + `fontFallback` (no tofu); sized/compressed `pw.Image`; header/footer with `pageNumber` / `pagesCount`.
- Images + row count drive size/time — compress + isolate; keep components reusable.

Revision Notes

- `pw.MultiPage(header:, footer:, build: → List<pw.Widget>)`; tables paginate; repeat header row; `columnWidths` / `cellAlignments`.
- Fonts: `pw.Font.ttf` base/bold/italic + `fontFallback` via `pw.ThemeData`; `PdfGoogleFonts` optional.
- Images: `pw.Image(pw.MemoryImage(bytes))` sized/compressed; footer page numbers via `ctx.pageNumber` / `pagesCount`; isolate large docs.

Practice Questions

1. How does a table paginate and keep its header on each page?
2. Why and how do you set up fonts with a fallback?
3. What most affects PDF file size, and how do you control it?

Coding Questions

1. Build a `MultiPage` report with a logo header + "Page X of Y" footer.
2. Render a long paginating table with a repeating header + column widths.
3. Configure a `ThemeData` with base/bold fonts + fallback.

Mini Project

Multi-page report (Flutter): Build a `MultiPage` A4 report: logo header, a long line-item `TableHelper` table that paginates with a repeating header and right-aligned numeric columns, branded fonts (base/bold + fallback for ₹), and a "Page X of Y" footer — with images compressed and generation isolate-offloaded. Acceptance: table spans pages with repeating header; fonts render bold + ₹ correctly; logo embedded + small; page numbers correct; reusable header/footer/table components; generated off the UI thread.

Viewing & Printing (`printing` Package)

> The `printing` package turns PDF bytes into user actions: `Printing.layoutPdf` (opens the OS print/preview dialog — print or save-as-PDF), the `PdfPreview` widget (an in-app preview with print/share/download buttons), `Printing.sharePdf` (native share sheet), and `Printing.convertHtml` (render simple HTML → PDF). It's the presentation half that complements the `pdf` package's generation half — plus it can fetch Google Fonts and raster pages to images.

Introduction

Once you have PDF bytes (01_pdf_generation_basics.md), users want to preview, print, share, or save them. The `printing` package provides the dialogs/widgets for all of these cross-platform. This file covers each entry point, the preview widget, HTML-to-PDF, and when to use which.

Why this concept exists

Generating bytes isn't enough — users expect the familiar OS print dialog (with "Save as PDF"), an in-app preview, and share targets. `printing` wraps the divergent platform print/preview/share APIs (Android print framework, iOS `UIPrintInteractionController`, etc.) into one Dart API so you don't implement each natively.

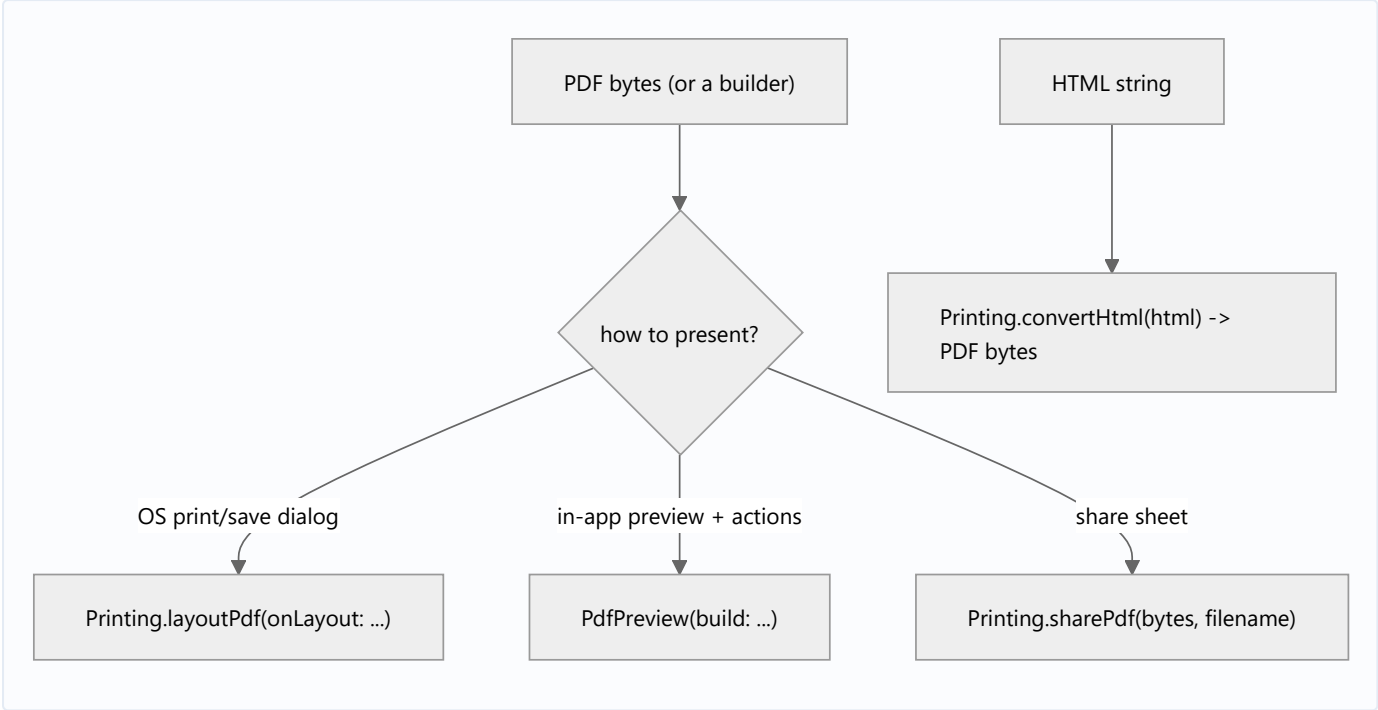
Real-world analogy

If the `pdf` package is the **printing press** (makes the document), `printing` is the **front office**: the **print counter** (send to a printer or save as PDF), the **preview window** on the wall (`PdfPreview`), and the **outgoing mail tray** (share). Same document, different ways to hand it to the user or a device.

Problem Statement

Let the user preview a generated invoice in-app, print it (or save-as-PDF) via the OS dialog, and share it — plus generate a quick PDF from an HTML string for a simple case. You'll use `PdfPreview` , `layoutPdf` , `sharePdf` , and `convertHtml` .

Internal Working



- **`Printing.layoutPdf`**: `Printing.layoutPdf(onLayout: (format) async => bytes)` opens the **OS print dialog** — the user picks a printer or **"Save as PDF"**. The `onLayout` callback receives the target `PdfPageFormat` (so you can regenerate for the chosen paper size). Fully cross-platform.
- **`PdfPreview`** widget: drops an in-app preview into your UI with built-in **print/share/download** actions; `build: (format) => bytes`. Good for a preview screen; customize toolbar/actions.
- **`Printing.sharePdf`**: `sharePdf(bytes: ..., filename: 'invoice.pdf')` opens the native **share sheet** (email, Drive, WhatsApp) — like `share_plus` but PDF-specific (Module 34).
- **`Printing.convertHtml`**: `convertHtml(format:, html:)` renders **simple HTML/CSS → PDF** — handy when you already have HTML, but the supported HTML/CSS subset is **limited** (no full browser); for complex/branded docs, build with `pw.*` instead.
- **Extras**: `Printing.raster(bytes)` rasterizes pages to images (thumbnails/preview images); `PdfGoogleFonts` fetches web fonts for embedding (02_layouts_tables_and_assets.md); check `Printing.info()` for platform capabilities.
- **Regeneration on format**: the `onLayout` / `build` callbacks pass the chosen format so you can generate at the correct page size (e.g., A4 vs Letter).

Memory Representation

You hold the PDF bytes; preview/print may raster pages (extra memory for images). Regenerating per format rebuilds the document.

Compiler Behavior

Not applicable.

Runtime Behavior

`layoutPdf` /preview may call your builder multiple times (e.g., per format change). `sharePdf` / `layoutPdf` present native modals. `convertHtml` runs an HTML→PDF conversion (heavier, limited fidelity).

Flutter Engine Behavior

`PdfPreview` renders rasterized pages within Flutter; print/share dialogs are native UI over Flutter (platform channels — 26).

Dart VM Behavior

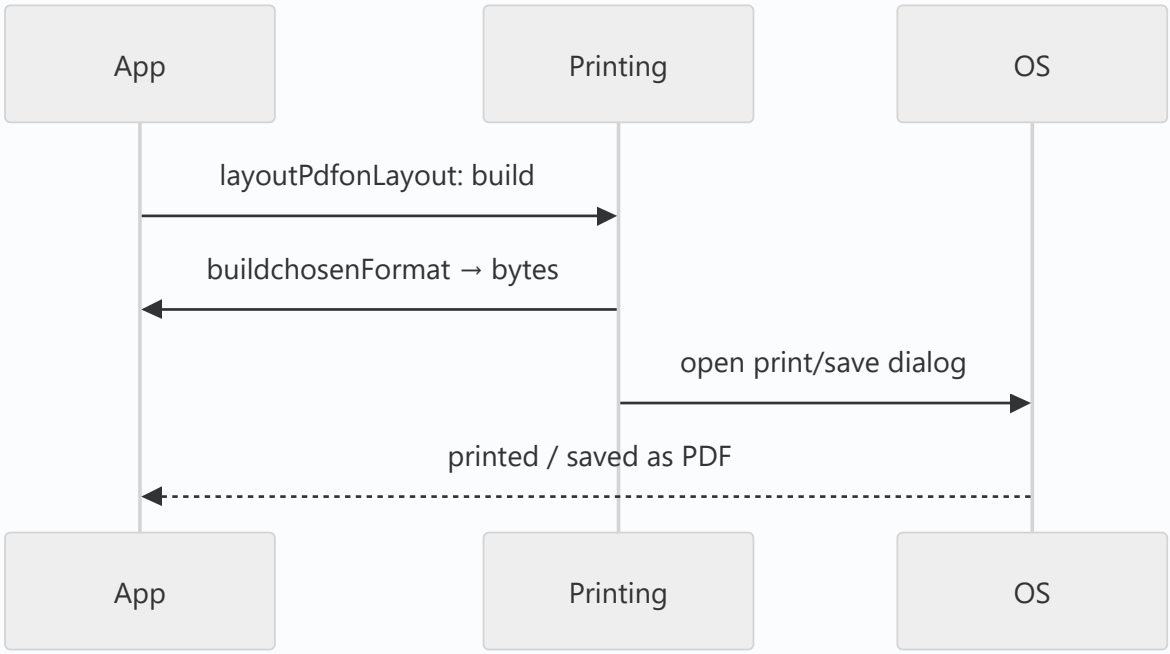
Not applicable beyond generation (offload heavy builds to isolates — 01_pdf_generation_basics.md).

Examples

```
import 'package:printing/printing.dart';
import 'package:pdf/pdf.dart';
import 'dart:typed_data';

// OS print/save-as-PDF dialog (regenerate for the chosen paper format)
Future<void> printInvoice(Future<Uint8List> Function(PdfPageFormat) build) async {
  await Printing.layoutPdf(onLayout: (format) => build(format));
}

// Share the PDF via the native share sheet
Future<void> shareInvoice(Uint8List bytes) async {
  await Printing.sharePdf(bytes: bytes, filename: 'invoice.pdf');
}

// Quick HTML -> PDF (LIMITED HTML/CSS subset; use pw.* for complex docs)
Future<Uint8List> htmlToPdf(String html) =>
    Printing.convertHtml(format: PdfPageFormat.a4, html: html);
```

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

// In-app preview screen with built-in print/share/download actions
class InvoicePreviewScreen extends StatelessWidget {
  final Future<Uint8List> Function(PdfPageFormat) build;
  const InvoicePreviewScreen({super.key, required this.build});
  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: const Text('Invoice')),
        body: PdfPreview(build: (format) => this.build(format)), // preview + actions
      );
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting `convertHtml` to render full HTML/CSS	Limited subset	Use `pw.*` for complex/branded docs
Ignoring the format in `onLayout` / `build`	Wrong paper size	Regenerate for the passed `PdfPageFormat`
Building heavy PDF synchronously in callback	Jank	Offload generation to an isolate
Reimplementing print natively	Wasted effort	Use `printing` (cross-platform)
Using `sharePdf` when print/save is wanted	Wrong UX	Match action to intent
Not checking platform capabilities	Feature unavailable	`Printing.info()`

Best Practices

- Match the action to intent: `layoutPdf` (print/save-as-PDF via OS), `PdfPreview` (in-app preview + actions), `Printing.sharePdf` (share sheet).
- **Regenerate for the passed** `PdfPageFormat` in `onLayout` / `build` (correct paper size); **offload heavy generation** to an isolate.
- Use `convertHtml` only for simple HTML (limited fidelity) — build complex/branded docs with `pw.*` (01_pdf_generation_basics.md).
- Use `PdfGoogleFonts` / `raster` as needed; check `Printing.info()` for capabilities; wrap behind a service (05_pdf_integration.md).

Performance

Preview/print rasterizes pages (memory) — fine for typical docs. The main cost is generation (isolate it). `convertHtml` is heavier and lower fidelity. Regeneration per format is cheap for small docs, notable for large ones (cache if needed).

Advantages / Disadvantages

- + Cross-platform print/preview/share/save with little code, OS-native dialogs, HTML-to-PDF option, font/raster helpers.
- − `convertHtml` limited, regeneration-per-format nuance, native modal behavior, still need `pdf` for real generation.

Interview Questions

1. 🟢 **What does** `Printing.layoutPdf` **do?** — Opens the OS print dialog where the user can print or "Save as PDF"; the `onLayout` callback supplies bytes for the chosen format.

2. 🟢 **How do you show a PDF in-app?** — The `PdfPreview` widget (built-in print/share/download actions) with a `build: (format) => bytes` callback.
3. 🟡 **How do you share a generated PDF?** — `Printing.sharePdf(bytes:, filename:)` opens the native share sheet.
4. 🟡 **When is `convertHtml` appropriate?** — Only for simple HTML/CSS (limited subset); complex/branded documents should be built with `pw.*`.
5. 🟡 **Why do the callbacks pass a `PdfPageFormat`?** — So you regenerate the document at the paper size the user selected (A4/Letter/etc.).
6. 🔴 **How does `printing` relate to the `pdf` package?** — `pdf` generates bytes; `printing` presents them (preview/print/share/save) — generation vs presentation.
7. 🔴 **What extra utilities does `printing` offer?** — `raster` (pages → images), `PdfGoogleFonts` (fetch fonts), `Printing.info()` (capabilities).

Senior Engineer Tips

- Always honor the `PdfPageFormat` in the layout/build callback; hardcoding A4 gives wrong output when a user picks Letter.
- Prefer `pw.*` generation over `convertHtml` for anything branded — HTML-to-PDF fidelity will bite you on real invoices.
- Offload generation inside the callback to an isolate so preview/print dialogs stay responsive on large docs.

Architect Perspective

`printing` is the presentation boundary for documents — the counterpart to `pdf`'s generation. Behind a `PdfService` exposing `preview` / `print` / `share` / `save`, features request an action and the service coordinates generation (isolate) + presentation, keeping paper-format handling and platform quirks in one place. This clean split (generate vs present) makes document features testable and consistent (01_pdf_generation_basics.md, 05_pdf_integration.md, Module 34).

Summary

- `printing`: `layoutPdf` (OS print/save), `PdfPreview` (in-app preview + actions), `sharePdf` (share sheet), `convertHtml` (simple HTML→PDF).
- Regenerate for the passed `PdfPageFormat`; offload heavy generation; use `pw.*` for complex docs (not `convertHtml`).
- Presentation half complementing `pdf`'s generation; behind a service.

Revision Notes

- `Printing.layoutPdf(onLayout: (format)→bytes)` (print/save-as-PDF); `PdfPreview(build: (format)→bytes)`; `Printing.sharePdf(bytes, filename)`; `Printing.convertHtml(format, html)` (limited).
- Honor `PdfPageFormat` in callbacks; isolate generation; `pw.*` for complex/branded; `raster` / `PdfGoogleFonts` / `Printing.info()`.
- Generation (`pdf`) vs presentation (`printing`); wrap behind a `PdfService`.

Practice Questions

1. Which `printing` API for print vs preview vs share?
2. Why do the callbacks provide a `PdfPageFormat`?
3. When should you avoid `convertHtml`?

Coding Questions

1. Wire an OS print/save dialog with `layoutPdf` honoring the format.
2. Build a preview screen with `PdfPreview` + actions.
3. Share a generated PDF via `sharePdf`.

Mini Project

Preview, print & share (Flutter): Given a PDF builder `(PdfPageFormat) => Future<Uint8List>`, build an invoice preview screen (`PdfPreview` with actions), an OS print/save dialog (`layoutPdf` honoring the chosen format), and a share action (`sharePdf`) — with generation offloaded to an isolate. Acceptance: preview shows in-app with print/share/download; OS dialog prints or saves-as-PDF at the correct paper size; share sheet works; heavy generation off the UI thread; behind a service.

Data-Driven Documents (Invoices/Reports from Models & Templates)

> Real documents are **pure functions of your domain data**: `PdfDocument buildInvoice(Invoice data)` maps a typed model → a `pw.*` tree, so the same **reusable template** (header/line-item table/totals/footer components) produces any invoice/report. Keep generation **deterministic and side-effect-free** (data in → bytes out), **format numbers/dates/currency** correctly (locale + `intl`), handle **empty/overflow/long-text** cases, and **offload** heavy generation — then the document layer is testable and every report shares one branded template.

Introduction

This file is where generation meets your app's data: turning `Invoice` / `Report` models into documents via reusable, composable templates. It covers the pure-function design, template composition, formatting/i18n, edge cases, and testing — the pattern that makes document generation maintainable rather than a pile of one-off builders.

Why this concept exists

Businesses generate many documents from structured data, all sharing branding/layout. Hardcoding each is unmaintainable. Modeling generation as `data → pw.* tree` with shared components (letterhead, table, totals block) gives consistency, reuse, and testability — the same separation-of-concerns discipline as widgets-from-state, applied to documents.

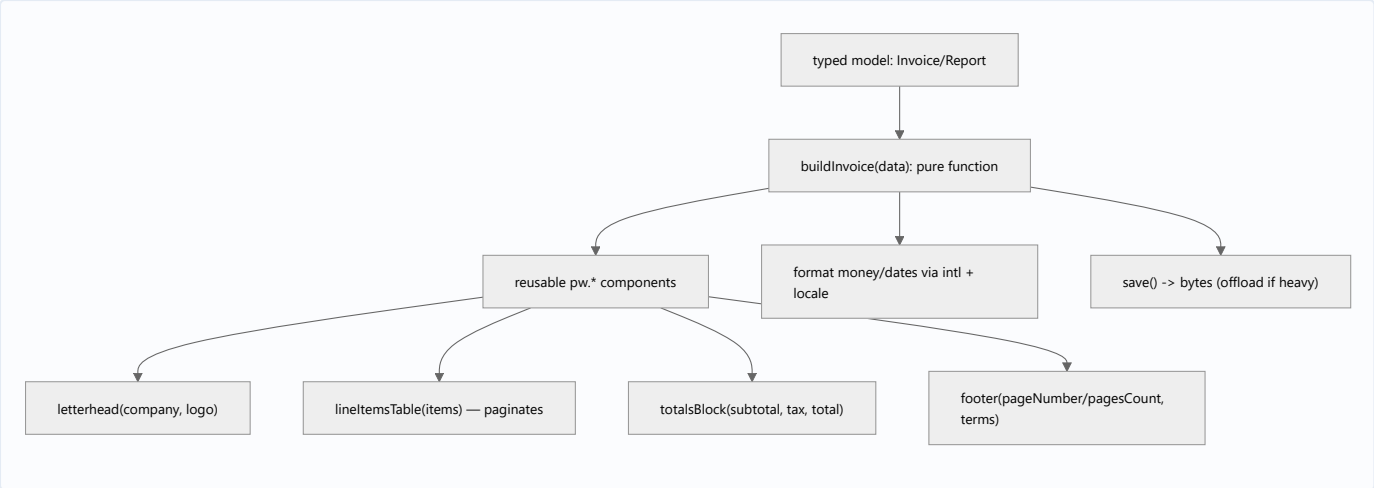
Real-world analogy

A template is a **mail-merge form letter**: the layout (letterhead, table grid, signature block) is fixed and branded; you **pour in the record's data** (customer, line items, totals) and out comes a personalized document. You don't redraw the letterhead for each customer — you reuse the form.

Problem Statement

Generate a branded invoice from an `Invoice` model (customer, line items, tax, totals) that renders correctly whether it has 1 or 500 items, formats currency/dates by locale, handles a long customer name, and reuses the same header/footer/table as your reports. You'll design a pure template + shared components.

Internal Working



- **Pure function generation**: `Future<Uint8List> buildInvoice(Invoice data, {PdfPageFormat format})` — **no I/O, no globals**; same input → same bytes. Makes it testable and isolate-safe.
- **Reusable components**: factor the document into `pw.Widget` builders — `letterhead(company)`, `lineItemsTable(items)`, `totalsBlock(totals)`, `footer(...)` — shared across invoices/receipts/reports for one branded look (02_layouts_tables_and_assets.md).
- **Formatting/i18n**: format **currency** (`NumberFormat.currency(locale, symbol)`), **dates** (`DateFormat`), and numbers with `intl` — never `toString()` raw doubles for money; align numeric columns; respect the document's locale.
- **Data mapping**: map model → rows/cells explicitly; compute derived values (subtotal/tax/total) either in the model/domain (preferred) or a formatter — keep the template presentational.
- **Edge cases**: **empty** list (show "No items"), **very long** text (wrap/truncate), **many items** (MultiPage pagination — 02_layouts_tables_and_assets.md), missing optional fields, huge totals (column width). Design for real data, not the happy demo.
- **Localization/RTL**: support the needed locales/fonts (fallback for scripts) and RTL if required.
- **Offload**: large/complex generation → isolate (02 · isolates); pass only serializable data into the isolate.
- **Testing**: because it's pure, unit-test that `buildInvoice` returns non-empty bytes for representative + edge inputs, and (optionally) golden-test rasterized pages.

Memory Representation

The template builds a `pw.*` tree from the model, serialized to bytes; size scales with items/images. No hidden state — inputs fully determine output.

Compiler Behavior

Not applicable.

Runtime Behavior

Generation cost scales with data size (rows/images). Deterministic output enables caching/regeneration. Edge cases (empty/huge) exercise different layout paths.

Flutter Engine Behavior

None; rendered to PDF.

Dart VM Behavior

CPU-bound; offload large docs to an isolate (pass serializable model data, not live objects with closures).

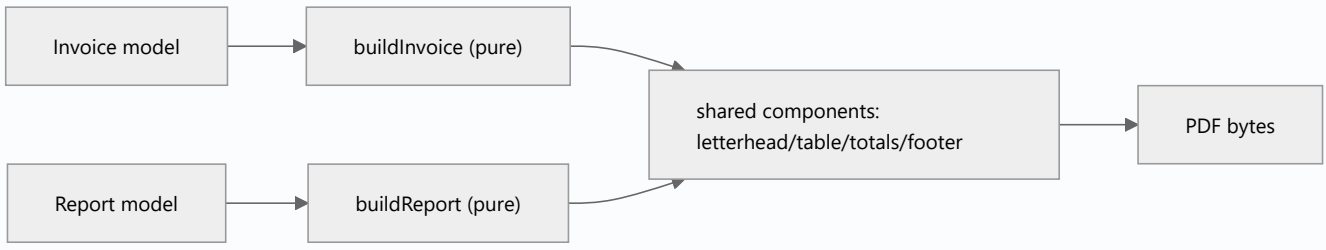
Examples

```
import 'package:pdf/pdf.dart';
import 'package:pdf/widgets.dart' as pw;
import 'package:intl/intl.dart';
import 'dart:typed_data';


// Pure function: Invoice model -> PDF bytes (no I/O, no globals)
Future<Uint8List> buildInvoice(Invoice inv, {PdfPageFormat format = PdfPageFormat.a4}) async {
  final money = NumberFormat.currency(locale: inv.locale, symbol: inv.currencySymbol);
  final doc = pw.Document(theme: _brandTheme);          // shared fonts/theme


  doc.addPage(pw.MultiPage(
    pageFormat: format,
    header: (ctx) => _letterhead(inv.company),          // reusable component
    footer: (ctx) => _footer(ctx, inv.terms),
    build: (ctx) => [
      _billTo(inv.customer, money),
      pw.SizedBox(height: 12),
      inv.items.isEmpty
          ? pw.Text('No items')                          // empty-state edge case
          : _lineItemsTable(inv.items, money),           // paginates
      pw.SizedBox(height: 12),
```

```
      _totalsBlock(inv.subtotal, inv.tax, inv.total, money),
    ],
  ));
  return doc.save();
}

pw.Widget _lineItemsTable(List<LineItem> items, NumberFormat money) =>
    pw.TableHelper.fromTextArray(
      headers: ['Item', 'Qty', 'Unit', 'Amount'],
      data: [
        for (final i in items)
          [i.name, '${i.qty}', money.format(i.unitPrice), money.format(i.amount)]
      ],
      cellAlignments: {1: pw.Alignment.centerRight, 2: pw.Alignment.centerRight, 3:
pw.Alignment.centerRight},
    );
```

```
// Pure -> easy to unit-test (representative + edge inputs)
test('invoice generates non-empty bytes, even when empty', () async {
  expect((await buildInvoice(sampleInvoice)).isNotEmpty, true);
  expect((await buildInvoice(sampleInvoice.copyWith(items: [])).isNotEmpty, true);
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Side effects/globals in generation	Untestable, isolate-unsafe	Pure function: data → bytes
`toString()` for money	Wrong formatting/precision	`intl` `NumberFormat.currency`
Only testing the happy path	Empty/huge/long-text break layout	Handle + test edge cases
Duplicating layout per document	Inconsistent branding, drift	Shared components/template
Computing totals in the template	Mixes domain + presentation	Compute in model/domain
Passing live objects to isolate	Not serializable	Pass plain data
No pagination for big lists	Clipping	`MultiPage` table

Best Practices

- Model generation as a **pure function** (`data → bytes` , no I/O/globals) built from **reusable components/templates** shared across documents.
- Format **money/dates/numbers with** `intl` (locale-aware); keep the template **presentational** (compute derived values in the domain/model).
- Handle **edge cases** (empty, many, long text, missing fields) and **paginate** large lists; support needed **locales/fonts/RTL**.
- **Offload heavy generation** to an isolate (pass serializable data); **unit-test** representative + edge inputs (optionally golden-test).

Performance

Cost scales with data (rows/images) — paginate and compress. Determinism enables caching regenerated docs. Isolate large reports. Formatting is cheap; the win is reuse + testability preventing rework.

Advantages / Disadvantages

- **+** Consistent branding, reuse across document types, testable (pure), locale-correct, handles real data.
- **−** Upfront template/component design, i18n/edge-case handling, isolate serialization discipline.

Interview Questions

1. 🟢 **Why model PDF generation as a pure function?** — `data → bytes` with no side effects is testable, deterministic, cacheable, and safe to run in an isolate.

2. 🟢 **How do you format currency in a document?** — `intl` `NumberFormat.currency(locale, symbol)` — never `toString()` a double for money.
3. 🟡 **How do you keep branding consistent across invoices and reports?** — Factor shared `pw.*` components (letterhead/table/totals/footer) into a reusable template used by all documents.
4. 🟡 **What edge cases must a data-driven document handle?** — Empty lists, very long text, many items (pagination), missing optional fields, large totals — not just the happy path.
5. 🟡 **Where should totals be computed?** — In the domain/model (presentation-free template) — keep the builder presentational.
6. 🔴 **What must you consider when generating in an isolate?** — Pass only serializable data (no live objects/closures); the pure function makes this natural.
7. 🔴 **How do you test document generation?** — Unit-test that representative + edge inputs produce valid non-empty bytes; optionally golden-test rasterized pages for layout regressions.

Senior Engineer Tips

- Keep generation pure and presentational — domain computes totals, the template just lays them out; this is what makes documents testable and reusable.
- Build a small component library (letterhead/table/totals/footer) first; every document type then composes it, and rebranding is one change.
- Always format money/dates through `intl` and test the empty/huge cases — raw `toString()` and happy-path-only are the classic invoice bugs.

Architect Perspective

Data-driven documents apply the "view is a function of state" principle to PDFs: pure builders over domain models, composed from shared branded components, formatted via i18n, isolate-offloaded, and unit-tested. This makes the document layer maintainable, consistent, and CI-testable — the same discipline as UI-from-state, and the foundation the capstone service orchestrates (05_pdf_integration.md, 02_layouts_tables_and_assets.md).

Summary

- Generate documents as pure functions `data → bytes` from reusable, branded components/templates shared across invoices/reports.
- Format money/dates/numbers with `intl`; compute derived values in the domain; handle empty/huge/long-text; paginate; support locales.
- Offload heavy generation (serializable data); unit-test representative + edge inputs.

Revision Notes

- Pure `buildInvoice(data) → bytes` (no I/O/globals) from shared components (letterhead/table/totals/footer); one branded template.
- `intl` `NumberFormat.currency` / `DateFormat`; domain computes totals; handle empty/many/long/missing; `MultiPage` pagination; locales/fonts/RTL.
- Isolate large docs (serializable data); unit-test + optional golden tests.

Practice Questions

1. Why keep generation pure and presentational?
2. How do you ensure consistent branding across document types?
3. Which edge cases must a data-driven template handle?

Coding Questions

1. Write a pure `buildInvoice(Invoice)` from shared components with `intl` formatting.
2. Handle empty-items and long-name edge cases.
3. Unit-test generation for representative + edge inputs.

Mini Project

Invoice template (Flutter): Build a pure `buildInvoice(Invoice, {format})` from reusable branded components (letterhead, paginating line-item table, totals block, footer), formatting money/dates via `intl`, handling empty/many/long-text cases, and offloading to an isolate. Add unit tests for representative + edge inputs. Acceptance: pure (data→bytes, no I/O/globals); shared components reused; correct locale formatting; edge cases handled; paginates large item lists; isolate-offloaded; unit-tested.

PDF Integration (Capstone: A Document Service)

> The maintainable shape: a `PdfService` (or `DocumentService`) that sits between features and both packages — features pass a **domain model + an action** (`generate`, `preview`, `print`, `share`, `save`), and the service composes **pure templates** (`pdf` / `pw.*`) with **presentation** (`printing`) and **persistence** (file repository — Module 34), offloading heavy generation to an **isolate** and honoring the chosen **paper format**. Features never touch `pw.*`, `Printing`, or file paths — they say "give me the invoice, printed."

Introduction

This module capstone unifies generation, layouts, presentation, and data-driven templates into one service. Scattering `pdf` / `printing` /file calls across widgets couples features to document internals and duplicates isolate/format handling. A `PdfService` centralizes it, exposing intent. This file shows the service design and an end-to-end invoice flow.

Why this concept exists

Document features cut across generation (pure templates), presentation (preview/print/share), persistence (save/cache), performance (isolate), and formatting (paper size/locale). Left in widgets, these tangle and drift. One service — like repositories for data — isolates them behind an intent API, consistent with clean architecture (Module 40).

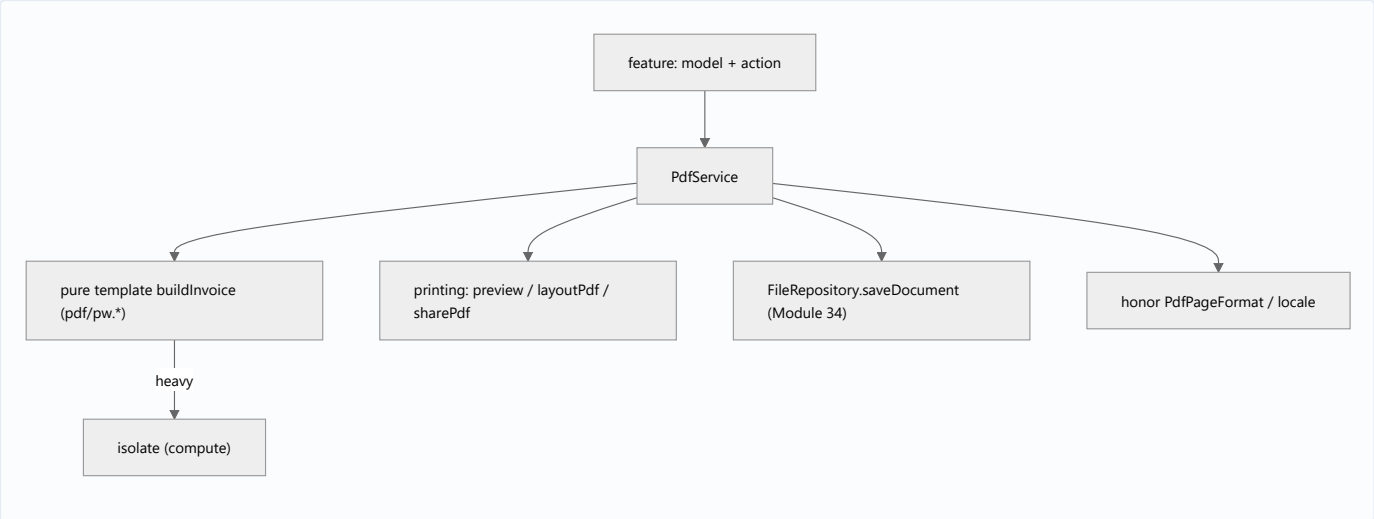
Real-world analogy

`PdfService` is the **print shop**: you bring an order form (domain model) and say what you want ("print two copies", "email me a PDF", "just show me a proof"). The shop runs the press (generation), handles paper size, and routes the output to the counter/mailroom/preview — you don't operate the machinery yourself.

Problem Statement

Deliver an invoice feature: generate a branded, paginated invoice from an `Invoice` model, let the user preview it, print/save-as-PDF via the OS dialog, share it, and persist a copy to the file repository — with heavy generation off the UI thread and correct paper format — all behind one `PdfService`. You'll compose every file in this module.

Internal Working



- **Intent API**: `Future<Uint8List> generate(Invoice, {format})`, `preview(Invoice)`, `print(Invoice)`, `share(Invoice)`, `save(Invoice)` (or an `Action` enum). Features pass a **model**, not `pw.*`.
- **Generation**: delegates to **pure templates** (04_data_driven_documents.md); **offloads** large docs to an isolate (`compute`), passing serializable model data.
- **Presentation**: uses `printing` — `PdfPreview` / `layoutPdf` / `sharePdf` — honoring the passed `PdfPageFormat` (03_viewing_and_printing.md).
- **Persistence**: hands bytes to the `FileRepository` to save into the right directory (documents) and index/share (Module 34); may **cache** generated bytes keyed by model hash to avoid regeneration.
- **Format/locale**: the service owns paper-format + locale handling so templates stay pure and features stay simple.
- **Testability**: depend on abstractions (a template function, a presenter interface, the file repo); unit-test that `generate` returns valid bytes and that actions route correctly with fakes — no device.
- **Separation recap**: `pdf` (generate) + `printing` (present) + file repo (persist), orchestrated by the service.

Memory Representation

The service holds no document state (optionally a small bytes cache keyed by model hash). Bytes flow generate → present/persist. Isolate generation keeps large-doc memory off the UI isolate.

Compiler Behavior

Compiles against template/presenter/file abstractions (mockable); isolate entry points serializable.

Runtime Behavior

`generate` builds (isolate for heavy) → bytes; `preview` / `print` / `share` / `save` route bytes to the right sink honoring format. Regeneration per format handled centrally; optional caching avoids repeat work.

Flutter Engine Behavior

`PdfPreview` rasterizes in-Flutter; print/share dialogs are native; generation is pure Dart (isolate).

Dart VM Behavior

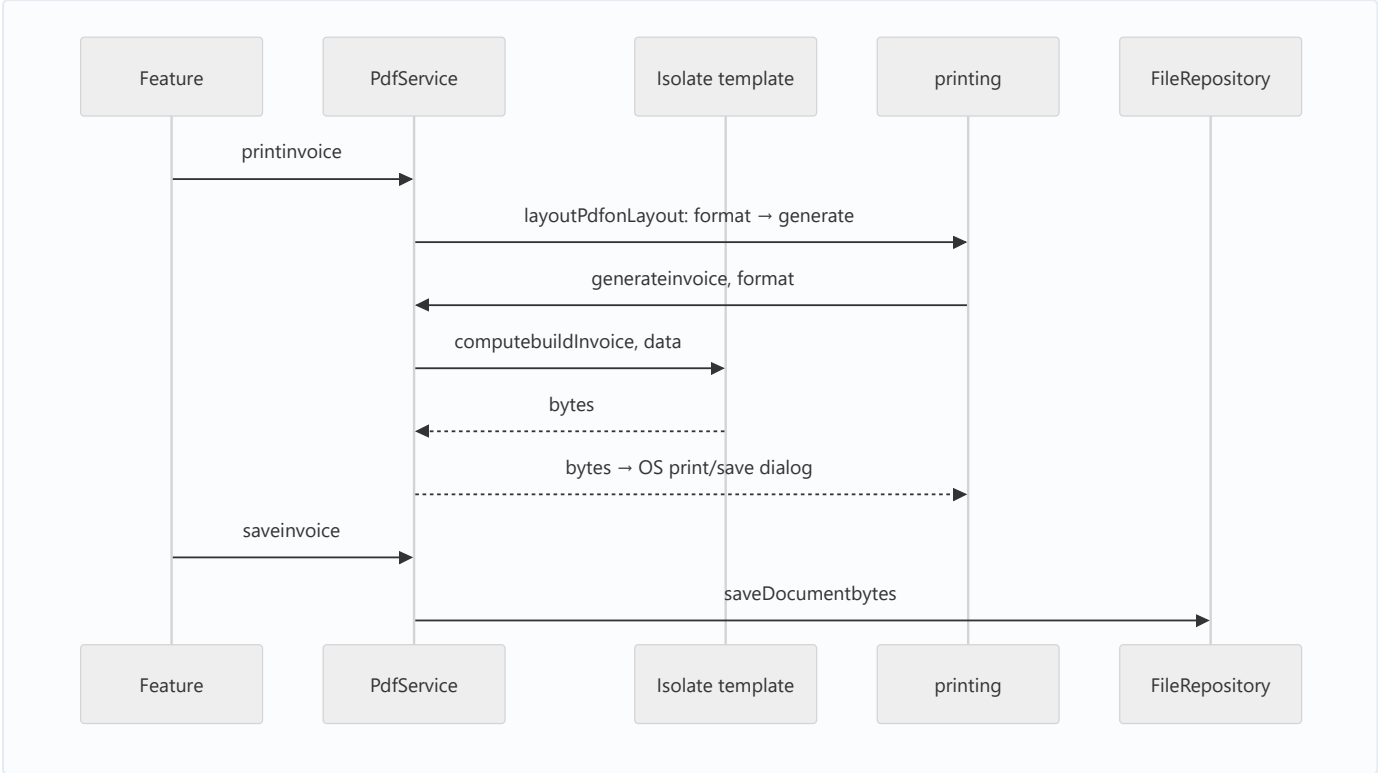
Heavy generation on a background isolate; the service coordinates on the UI isolate.

Examples

```
// Intent-based service — features never touch pw.*, Printing, or file paths
abstract class PdfService {
  Future<Uint8List> generate(Invoice inv, {PdfPageFormat format});
  Future<void> preview(BuildContext context, Invoice inv);
  Future<void> print(Invoice inv);          // OS print/save dialog
  Future<void> share(Invoice inv);          // share sheet
  Future<ManagedFile> save(Invoice inv);    // persist via FileRepository
}

class PdfServiceImpl implements PdfService {
  final FileRepository files;
  PdfServiceImpl(this.files);

  @override
  Future<Uint8List> generate(Invoice inv, {PdfPageFormat format = PdfPageFormat.a4}) =>
      compute(_generateInvoice, _InvoiceArgs(inv, format)); // pure template, isolate

  @override
  Future<void> print(Invoice inv) => Printing.layoutPdf(
      onLayout: (format) => generate(inv, format: format));  // honor chosen format

  @override
  Future<void> share(Invoice inv) async =>
      Printing.sharePdf(bytes: await generate(inv), filename: 'invoice-${inv.id}.pdf');

  @override
  Future<ManagedFile> save(Invoice inv) async =>
      files.saveDocument('invoice-${inv.id}.pdf', await generate(inv)); // Module 34
}

// Top-level, serializable-args isolate entry (pure buildInvoice from data_driven_documents)
Future<Uint8List> _generateInvoice(_InvoiceArgs a) => buildInvoice(a.inv, format: a.format);
```

```
// Feature usage — pure intent
// await pdfService.print(order.toInvoice());
// await pdfService.share(order.toInvoice());
// final saved = await pdfService.save(order.toInvoice());
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`pdf` / `printing` /file calls in widgets	Coupled, duplicated, untestable	One `PdfService` intent API
Generation on UI thread	Jank on large docs	Isolate via the service
Ignoring chosen paper format	Wrong output size	Service honors `PdfPageFormat`
Duplicating isolate/format logic per feature	Drift	Centralize in service
Impure templates (I/O/globals)	Isolate-unsafe/untestable	Pure `data → bytes` templates
No persistence strategy	Lost/misplaced files	Route saves via `FileRepository`
Untestable (real device)	Slow/fragile	Abstractions + fakes

Best Practices

- Expose an **intent-based** `PdfService` (`generate` / `preview` / `print` / `share` / `save`); keep `pdf` / `printing` /file details **inside** it.
- Delegate to **pure templates**, **offload** heavy generation to an isolate, and **honor the paper format/locale** centrally.
- Route **persistence via the** `FileRepository` (Module 34); optionally **cache** bytes by model hash; keep features model-only.

- Depend on **abstractions** for **testability** (template/presenter/file fakes); unit-test generation + action routing.

Performance

The service centralizes the performance levers (isolate generation, format regeneration, optional caching) so they apply uniformly. Features get responsive document actions; large docs never jank the UI; caching avoids redundant regeneration.

Advantages / Disadvantages

- **+** Clean separation (generate/present/persist), testable, consistent, features stay simple, centralized isolate/format/cache handling.
- **−** Upfront service/abstraction design, requires discipline to route all document ops through it.

Interview Questions

1. 🟢 **Why wrap PDF work in a service?** — To hide `pdf` / `printing` /file details behind an intent API, centralize isolate/format/persistence handling, and keep features testable and simple.
2. 🟢 **What does the intent API look like?** — `generate` / `preview` / `print` / `share` / `save` taking a domain model — features pass data + action, not `pw.*`.
3. 🟡 **How does the service keep the UI responsive?** — It offloads heavy generation to an isolate (`compute`) with serializable data.
4. 🟡 **How is the paper format handled?** — Centrally: presentation callbacks pass a `PdfPageFormat`, and the service regenerates at that size.
5. 🟡 **How does persistence work?** — The service routes bytes to the `FileRepository` (correct directory + index), reusing the file layer.
6. 🔴 **How do generation, presentation, and persistence separate?** — `pdf` /pure templates generate bytes; `printing` presents; `FileRepository` persists — the service orchestrates all three.
7. 🔴 **How do you test the service?** — With template/presenter/file fakes: assert `generate` returns valid bytes and that each action routes to the right collaborator — no device.

Senior Engineer Tips

- Route every document operation through the service; the moment a widget calls `Printing` or `pw.*` directly, format/isolate/persistence handling starts drifting.
- Centralize isolate + format handling once so templates stay pure and features stay trivial; add a bytes cache keyed by model hash if the same doc is regenerated often.
- Keep the three concerns cleanly split (generate/present/persist) — it's what makes document features testable in CI and easy to extend (new doc types reuse the pipeline).

Architect Perspective

PDF integration is the orchestration seam for documents: pure templates (generation), `printing` (presentation), and the file repository (persistence), unified behind an intent-based service that owns isolate/format/cache concerns. This mirrors the app's other boundaries (data/network/files), keeps document features simple and CI-testable, and scales to new document types by reusing the pipeline — completing the generation→presentation→persistence story (04_data_driven_documents.md, 03_viewing_and_printing.md, Module 34, Module 40).

Summary

- One intent-based `PdfService` composes pure templates (`pdf`), presentation (`printing`), and persistence (`FileRepository`); features pass model + action.
- Centralizes isolate generation, paper-format handling, and optional caching; keeps `pw.*` / `Printing` /paths out of features.
- Depend on abstractions for testability; clean generate/present/persist separation scales to new document types.

Revision Notes

- `PdfService` : `generate` / `preview` / `print` / `share` / `save` (model); generation via pure templates + `compute` (isolate); honor `PdfPageFormat` .
- Present via `printing` (`PdfPreview` / `layoutPdf` / `sharePdf`); persist via `FileRepository` (Module 34); optional bytes cache by model hash.
- Abstractions + fakes for tests; separation: `pdf` (generate) / `printing` (present) / file repo (persist).

Practice Questions

1. What belongs in the `PdfService` vs the feature/template?
2. How are isolate offloading and paper format handled centrally?
3. How do generation, presentation, and persistence separate?

Coding Questions

1. Design a `PdfService` interface + impl composing template/printing/file repo.
2. Implement `print` honoring the format and `save` via the file repository.
3. Unit-test action routing + generation with fakes.

Mini Project

Invoice generator service (capstone — Flutter): Build a `PdfService` that generates a branded invoice from an `Invoice` model via a pure template (isolate-offloaded), and exposes `preview` (`PdfPreview`), `print` (`layoutPdf`, format-honoring), `share` (`sharePdf`), and `save` (via `FileRepository`, Module 34). Depend on abstractions; add unit tests. Acceptance: features pass model + action only (no `pw.*` / `Printing` /paths); generation off the UI thread; paper format honored; preview/print/share/save all work; saved copy persisted + indexed; unit-tested with fakes; runs end-to-end on device.