36 · Excel

Flutter Practice Notes

Contents

36 · Excel

Introduction

This module covers reading and writing spreadsheets in Flutter: the **package landscape** (`excel` for free read/write of `.xlsx`, Syncfusion XlsIO for richer styling/formulas/charts, `csv` for CSV) and their **workbook → sheet → cell** model; **cells, styling & formulas** (types, number/date formats, styles, merged cells, formulas); **CSV & data interchange** (when CSV beats XLSX, parsing/quoting edge cases); and **import/export of app data at scale** (mapping models ↔ rows, memory/streaming concerns for large datasets), tied together in a capstone. It builds on file handling (Module 34) and pairs with PDF (Module 35).

Why this module exists

Business and data apps constantly import/export spreadsheets: users upload an XLSX/CSV to bulk-load data, or export reports/tables to open in Excel/Sheets. But spreadsheets are deceptively tricky — cell type coercion, date serial numbers, locale/quoting in CSV, formula vs value, and memory blow-ups on large files. Choosing the right package and handling these correctly (off the UI thread) is a common, high-value skill.

Module map

#	File	Topic	Level
1	01_excel_basics_and_packages.md	Package choice (`excel`/Syncfusion/`csv`), workbook/sheet/cell model, read/write	🔵
2	02_cells_styles_and_formulas.md	Cell types, number/date formats, styling, merged cells, formulas	🟡
3	03_csv_and_data_interchange.md	CSV read/write, quoting/locale edge cases, CSV vs XLSX	🟡
4	04_import_export_large_datasets.md	Model↔row mapping, validation, memory/streaming, large files	🔴
5	05_excel_integration.md	Capstone: import/export behind a service, off-UI-thread	🔴

Cross-references: File save/share/pick: Module 34. PDF (the other export format): Module 35. Isolates (heavy parse/generate): 02 · isolates. Networking (fetch export data): Module 16. Models/serialization: 02 · json_serialization.

Prerequisites

34 File Handling (read/write/pick/share), 02 Advanced Dart (async/isolates), basic model/serialization knowledge.

What you'll be able to do after this module

- Choose the right spreadsheet package (`excel` vs Syncfusion vs `csv`) for the job.
- Read and write XLSX workbooks/sheets/cells and understand the cell model.
- Apply cell types, number/date formats, styles, merged cells, and formulas.
- Parse and generate CSV correctly (quoting, delimiters, locale, encoding).
- Import/export app data at scale, mapping models ↔ rows with validation, off the UI thread.

Capstone

Import/export slice: A data manager that imports a user-picked XLSX/CSV (validate + map rows → models, report row errors), exports a filtered dataset to a styled XLSX (headers, number/date formats, totals formula) and to CSV, handles a large file without freezing the UI (isolate), and lets the user share the result — behind a `SpreadsheetService` .

Summary

Excel work = pick the right package (`excel` /Syncfusion/ `csv`) for the workbook→sheet→cell model, handle cell types/formats/formulas and CSV quoting correctly, map models ↔ rows with validation, and do heavy parse/generate off the UI thread — wrapped behind a service, complementing PDF export.

Excel Basics & Package Choice

> Three tools for three needs: `excel` (free, pure-Dart read+write of `.xlsx` — the default for most apps), **Syncfusion XlsIO** (richer styling/formulas/charts/large-file APIs, but commercial-licensed), and `csv` (plain comma-separated text — simplest, universal, but no types/styling). All share a **workbook → sheet → cell (by row/column)** model. Pick by need — and remember XLSX is a **zipped XML** format, so large files are memory-heavy: parse/generate them **off the UI thread**.

Introduction

Before reading or writing anything, choose the right package and understand the shared model. This file compares `excel`, Syncfusion, and `csv`, explains the workbook/sheet/cell structure and coordinate systems (A1 vs row/col indices), and shows a basic read and write — the foundation for the rest of the module.

Why this concept exists

"Excel support" spans free basic read/write, rich formatting/formulas, and trivial CSV — with very different tradeoffs (licensing, features, memory). A single package doesn't fit all cases, so knowing the landscape avoids over-engineering (Syncfusion for a CSV export) or hitting walls (`excel` for advanced charts). The shared workbook/sheet/cell model unifies how you address data.

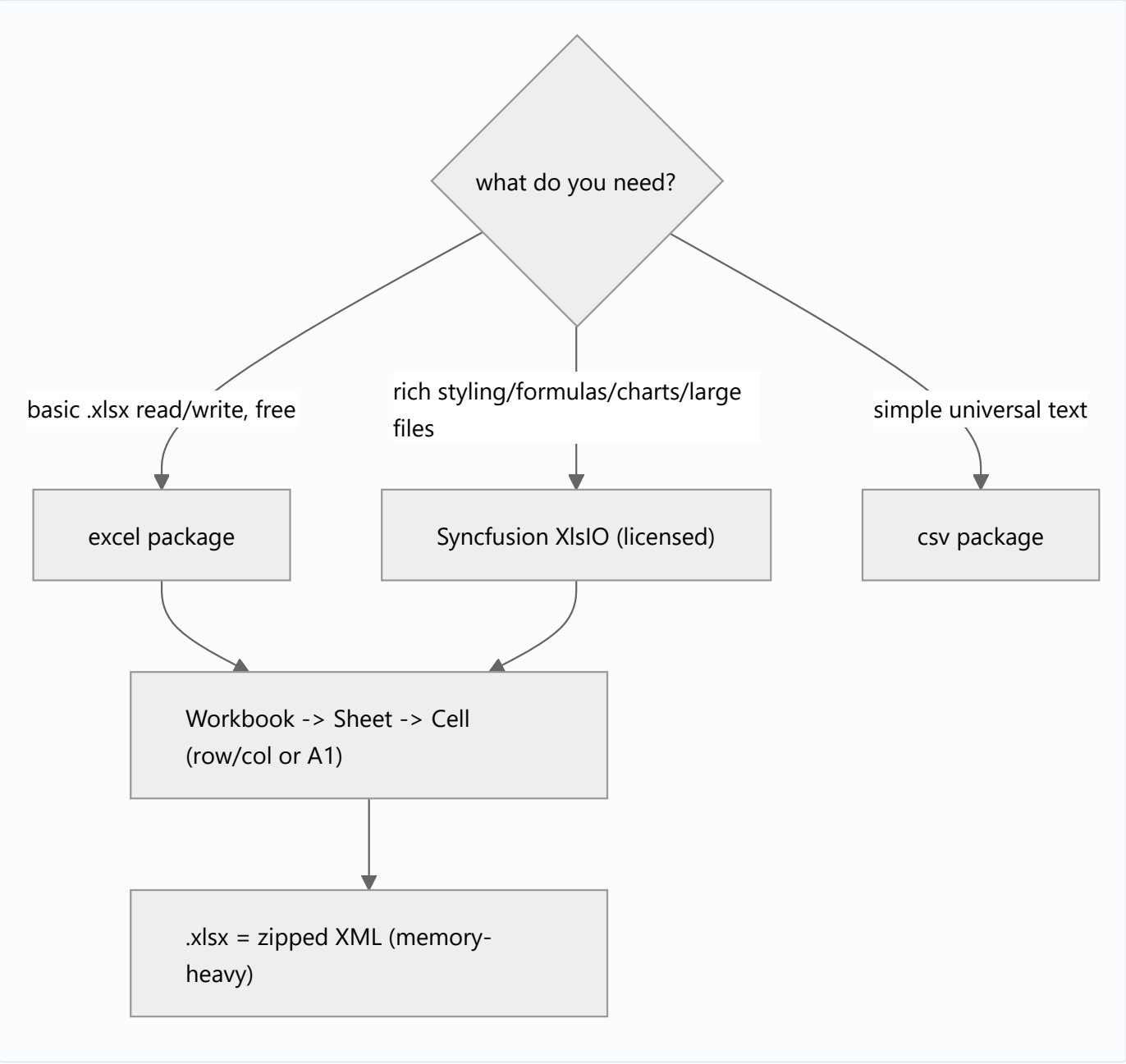
Real-world analogy

Choosing a package is like choosing a **document tool**: `csv` is a **plain text notepad** (universal, no formatting), `excel` is a **free spreadsheet app** (real cells, basic styling), and Syncfusion is the **pro suite** (advanced formatting/charts/formulas — but you pay for the license). A workbook is a **binder**, each **sheet a tabbed page**, each **cell a labeled box** (A1, B2…).

Problem Statement

Decide which package to use for: a simple data export, a styled report with formulas, and a bulk CSV import — then read an uploaded XLSX and write a new one. You'll compare options and do a basic read/write with `excel`.

Internal Working



- `excel` (free, pure Dart): read + write `.xlsx`; `Excel.decodeBytes(bytes)` to read, `excel.save()` / `encode()` to write. Cells addressed by `CellIndex.indexByColumnRow` or A1 (`'A1'`). Good default; limited advanced styling/charts, and large files are memory-heavy (loads the whole workbook).
- **Syncfusion XlsIO** (`syncfusion_flutter_xlsio`, **commercial license**): richer — cell styles, number formats, **formulas**, merged cells, conditional formatting, charts, images, and better large-file handling. Use when you need Excel-grade output; mind the license.
- `csv` (free): encode/decode delimited text (`ListToCsvConverter` / `CsvToListConverter`). No types/styling/multiple sheets — but universal, tiny, streamable (03_csv_and_data_interchange.md).

- **Shared model**: **Workbook** (the file) → one or more **Sheets** (tabs) → **Cells** at (row, column). Coordinates are commonly **0-based indices** in code but **A1** in Excel UI — mind the mapping (A=col 0, row1=index 0).
- **XLSX = zipped XML (OOXML)**: a `.xlsx` is a ZIP of XML parts. Reading/writing decompresses/serializes lots of XML → **memory + CPU heavy** for large files → do it in an **isolate** (02 · isolates); CSV is far lighter.
- **Bytes in/out**: all produce/consume bytes you save/share/pick via the file layer (Module 34).

Memory Representation

`excel`/Syncfusion load the workbook object graph into memory (cells, styles) — scales with size. CSV can be processed row-by-row (streamable). Output is bytes (`Uint8List`) → file layer.

Compiler Behavior

Not applicable; normal Dart packages. Syncfusion requires a license key registration.

Runtime Behavior

Decoding/encoding XLSX parses/serializes XML (slow for large files). CSV parsing is linear and cheap. Cell access by index/A1 is direct.

Flutter Engine Behavior

None; pure Dart processing. UI stays smooth only if heavy work is offloaded.

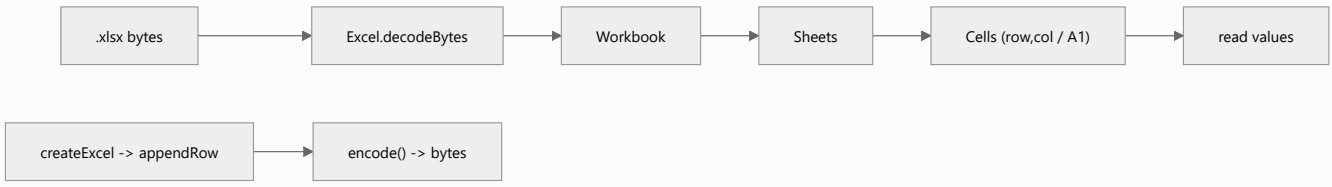
Dart VM Behavior

CPU/memory-bound parsing/serialization → offload large operations to an isolate.

Examples

```
import 'package:excel/excel.dart';
import 'dart:typed_data';

// READ an uploaded .xlsx
List<List<dynamic>> readSheet(Uint8List bytes, {String? sheetName}) {
  final excel = Excel.decodeBytes(bytes);
  final name = sheetName ?? excel.tables.keys.first;   // first sheet by default
  final sheet = excel.tables[name]!;
  return sheet.rows.map((row) => row.map((c) => c?.value).toList()).toList();
}

// WRITE a new .xlsx and return bytes (save/share via Module 34)
Uint8List writeSheet(List<String> headers, List<List<Object?>> rows) {
  final excel = Excel.createExcel();                  // creates a default sheet
  final sheet = excel['Data'];
  sheet.appendRow(headers.map((h) => TextCellValue(h)).toList());
  for (final r in rows) {
    sheet.appendRow(r.map<CellValue?>((v) => v == null
        ? null
        : v is num ? DoubleCellValue(v.toDouble()) : TextCellValue('$v')).toList());
  }
  return Uint8List.fromList(excel.encode()!);          // bytes -> file
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Syncfusion for a simple CSV/export	Overkill + license cost	Use `excel` / `csv`
`excel` for charts/advanced formats	Not supported	Use Syncfusion
Loading huge XLSX on UI thread	Jank/OOM	Isolate; consider CSV/streaming
Confusing A1 vs 0-based indices	Off-by-one/wrong cells	Map A1↔index carefully
Assuming CSV = Excel	CSV has no types/styling/sheets	Choose format by need
Ignoring license (Syncfusion)	Legal/compliance	Register key / pick free option

Best Practices

- **Default to** `excel` for basic `.xlsx` read/write; use **Syncfusion** only when you need rich styling/formulas/charts (mind the **license**); use `csv` for simple/universal interchange.
- Understand the **workbook→sheet→cell** model and the **A1↔0-based index** mapping; access cells explicitly.
- Treat XLSX as **memory/CPU heavy** (zipped XML) — **offload large parse/generate to an isolate**; prefer **CSV/streaming** for very large data.
- Produce/consume **bytes** and route through the **file layer** (Module 34); wrap behind a service.

Performance

XLSX decode/encode is the cost center (XML + zip) — scales with cells/sheets; isolate it. CSV is linear and streamable (cheap). Cell access is O(1). For big datasets, CSV or Syncfusion's streaming beats loading a huge `excel` workbook.

Advantages / Disadvantages

- **+** Options for every need (free basic, rich commercial, universal CSV); familiar workbook model; standard bytes output.
- **−** Package tradeoffs (features vs license vs memory), XLSX heavy for large files, A1/index mapping, CSV lacks types/styling.

Interview Questions

1. 🟢 **Which package for basic free XLSX read/write?** — The `excel` package (pure Dart, read + write).
2. 🟢 **When would you choose Syncfusion or CSV instead?** — Syncfusion for rich styling/formulas/charts/large files (commercial license); CSV for simple, universal, lightweight

interchange.

3. 🟡 **What's the shared spreadsheet model?** — Workbook → sheets → cells addressed by (row, column) or A1 — mind the 0-based-index vs A1 mapping.
4. 🟡 **Why are large XLSX files memory-heavy?** — `.xlsx` is zipped XML (OOXML); decoding/encoding parses/serializes lots of XML and loads the workbook into memory.
5. 🟡 **Why offload spreadsheet work to an isolate?** — Decode/encode is CPU/memory-bound and would jank the UI on large files.
6. 🔴 **CSV vs XLSX — key differences?** — CSV is plain typeless text (no styling/formulas/multiple sheets) but tiny/streamable; XLSX has types/styling/sheets but is heavier.
7. 🔴 **What do these packages produce/consume, and how does that integrate?** — Bytes (`Uint8List`) you read from a picked file and write to save/share via the file layer.

Senior Engineer Tips

- Start with `excel` or `csv`; only reach for Syncfusion when a real requirement (charts, complex formatting, huge files) justifies the license — don't pay for a data dump.
- Assume large files and isolate parse/generate from the start; the "works on 10 rows, freezes on 100k" bug is the classic spreadsheet regression.
- Nail the A1↔index mapping in a tiny helper and reuse it; off-by-one cell bugs are endless otherwise.

Architect Perspective

Package choice is a requirements/licensing/performance decision, not a default. Encapsulating the chosen library behind a `SpreadsheetService` (bytes in/out, isolate-offloaded) lets you swap `excel` ↔Syncfusion↔CSV without touching features, and keeps the memory/CPU concerns in one place. The workbook/sheet/cell model and bytes-based I/O integrate cleanly with the file and import/export layers (05_excel_integration.md, Module 34).

Summary

- `excel` (free basic), Syncfusion (rich, licensed), `csv` (simple/universal) — pick by need; all use workbook→sheet→cell.
- XLSX is zipped XML → memory/CPU heavy → isolate large ops; CSV is light/streamable; mind A1↔0-index.
- Bytes in/out via the file layer; wrap behind a service.

Revision Notes

- `excel` : `Excel.decodeBytes(bytes)` read, `createExcel` / `appendRow` / `encode()` write; cells by `CellIndex` /A1.
- Syncfusion XlsIO (licensed): styles/formulas/charts/large files; `csv` : `CsvToListConverter` / `ListToCsvConverter` (no types/styling).
- Workbook→sheet→cell; A1↔0-based mapping; XLSX = zipped XML (heavy) → isolate; bytes → file layer (Module 34).

Practice Questions

1. Which package for a styled report with formulas and charts?
2. Why isolate XLSX decode/encode for large files?
3. How do A1 references map to code indices?

Coding Questions

1. Read the first sheet of a picked `.xlsx` into rows with `excel`.
2. Write a new `.xlsx` from headers + rows, returning bytes.
3. Offload a large decode to an isolate via `compute`.

Mini Project

Read & write XLSX (Flutter): Using `excel`, read an uploaded `.xlsx` (first sheet → rows) and write a new `.xlsx` from headers + rows returning bytes (saved/shared via Module 34), with large operations offloaded to an isolate. Add a short doc justifying `excel` vs Syncfusion vs `csv` for three scenarios. Acceptance: reads + writes correct cells (A1/index handled); bytes routed to file layer; heavy ops off the UI thread; package-choice rationale documented.

Cells, Styles & Formulas

> Cells carry a **typed value** (text/number/bool/date/formula) — not just a string — and getting the **type + number/date format** right is what makes Excel treat "1,000" as a number and "2026-07-09" as a date rather than text. On top of that you can apply **styles** (font/fill/borders/alignment), **merge cells**, set **column widths**, and write **formulas** (`=SUM(B2:B10)`) that Excel evaluates on open. `excel` handles types + basic styling; **Syncfusion** is the tool when you need real formulas, number formats, and rich formatting.

Introduction

A spreadsheet's value is its structure: numbers you can sum, dates you can sort, currency that displays right, and formulas that compute. This file covers cell types, number/date formatting, styling, merged cells/widths, and formulas — plus which package supports what — the difference between a data dump and a usable Excel file.

Why this concept exists

If everything is text, Excel can't sum, sort by date, or format currency — the export is useless for analysis. Cells therefore have **types** and **number formats**, and formulas let the sheet compute live. Styling makes reports readable. These features distinguish a real spreadsheet from CSV, and require correct typing (a common bug source).

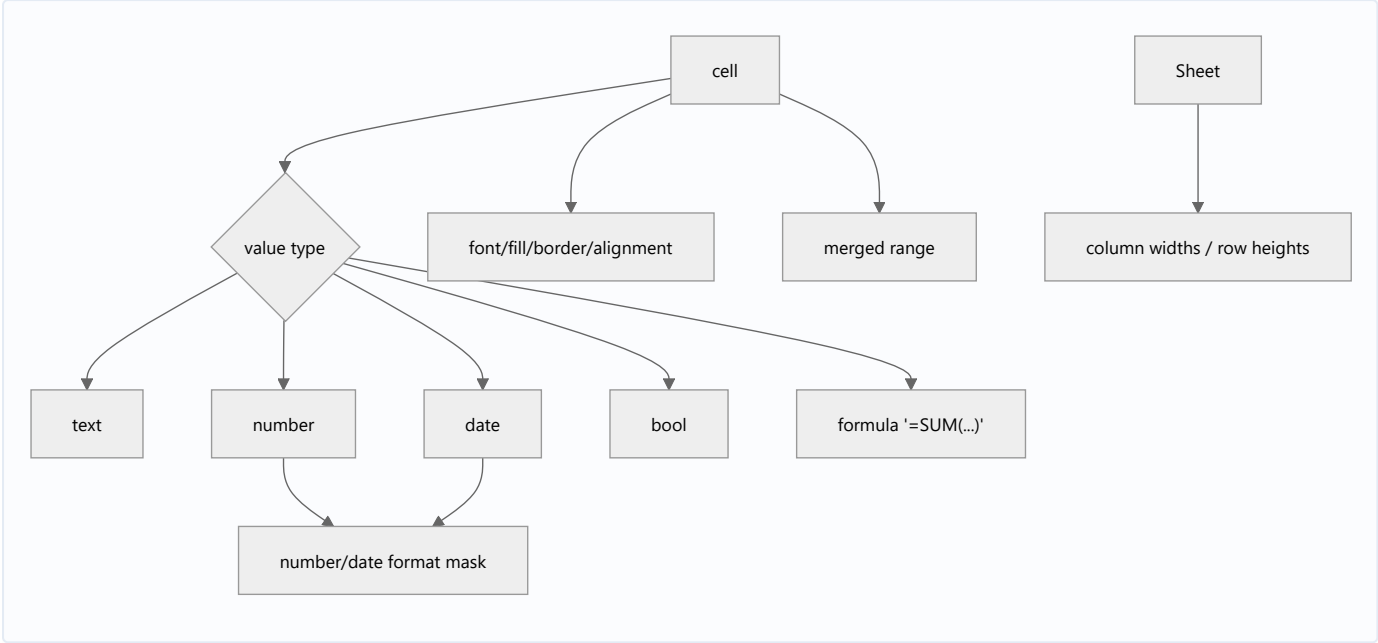
Real-world analogy

A cell isn't just ink on paper — it's a **labeled container that knows what it holds**: a number bin, a date bin, a money bin. Put a number as text and it's like writing "1000" on a sticky note the calculator won't add. **Number formats** are the display mask (₹1,000.00), **styles** are the formatting (bold header, shaded row), and a **formula** is a **live calculator taped into the cell** that recomputes when data changes.

Problem Statement

Export a sales report where amounts are real numbers formatted as currency, dates are real dates, the header row is bold with a fill, a "Total" cell uses `=SUM(...)`, and the title spans merged cells with sensible column widths. You'll set cell types, formats, styles, merges, and a formula.

Internal Working



- **Cell types** (not strings!): text, **number** (int/double), **bool**, **date/time**, and **formula**. In `excel`, use the typed `CellValue`s (`TextCellValue`, `IntCellValue`, `DoubleCellValue`, `DateCellValue`, `BoolCellValue`, `FormulaCellValue`). Wrong type = wrong behavior (numbers as text won't sum).
- **Number/date formats**: a cell's **display format** (e.g., `#,##0.00`, `₹#,##0.00`, `dd-mmm-yyyy`, `0%`) is separate from its value. Dates are stored as **serial numbers** (days since an epoch) with a date format applied — a frequent gotcha when reading (you may get a serial number, not a `DateTime`).
- **Styling**: font (bold/size/color), fill/background, borders, alignment, wrap. `excel` supports **basic** `CellStyle`; **Syncfusion** supports the full range (plus conditional formatting, themes).
- **Merged cells**: merge a range for titles/headers (`sheet.merge(CellIndex..., CellIndex...)`); the value lives in the top-left cell.
- **Column widths / row heights**: set for readability (`setColumnWidth`), auto-fit in Syncfusion.
- **Formulas**: write `=SUM(B2:B10)`, `=AVERAGE(...)`, `=A2*B2` as a **formula cell**; Excel evaluates on open (the file stores the formula, and often a cached result). `excel` has limited formula support; **Syncfusion** evaluates/handles them robustly.
- **Package split**: `excel` → types + basic styling + simple formulas; **Syncfusion** → rich number formats, full styling, reliable formulas, conditional formatting, charts.

Memory Representation

Each cell stores a value + optional style/format reference. Styles are often shared/interned. Formulas store the expression (and possibly a cached value). More styles/formulas = larger file.

Compiler Behavior

Not applicable.

Runtime Behavior

Formulas evaluate when Excel/Sheets opens the file (the app writes the expression, not the computed value, unless it also caches it). Number formats affect display only; the underlying value is what's stored/summed.

Flutter Engine Behavior

None; pure Dart generation.

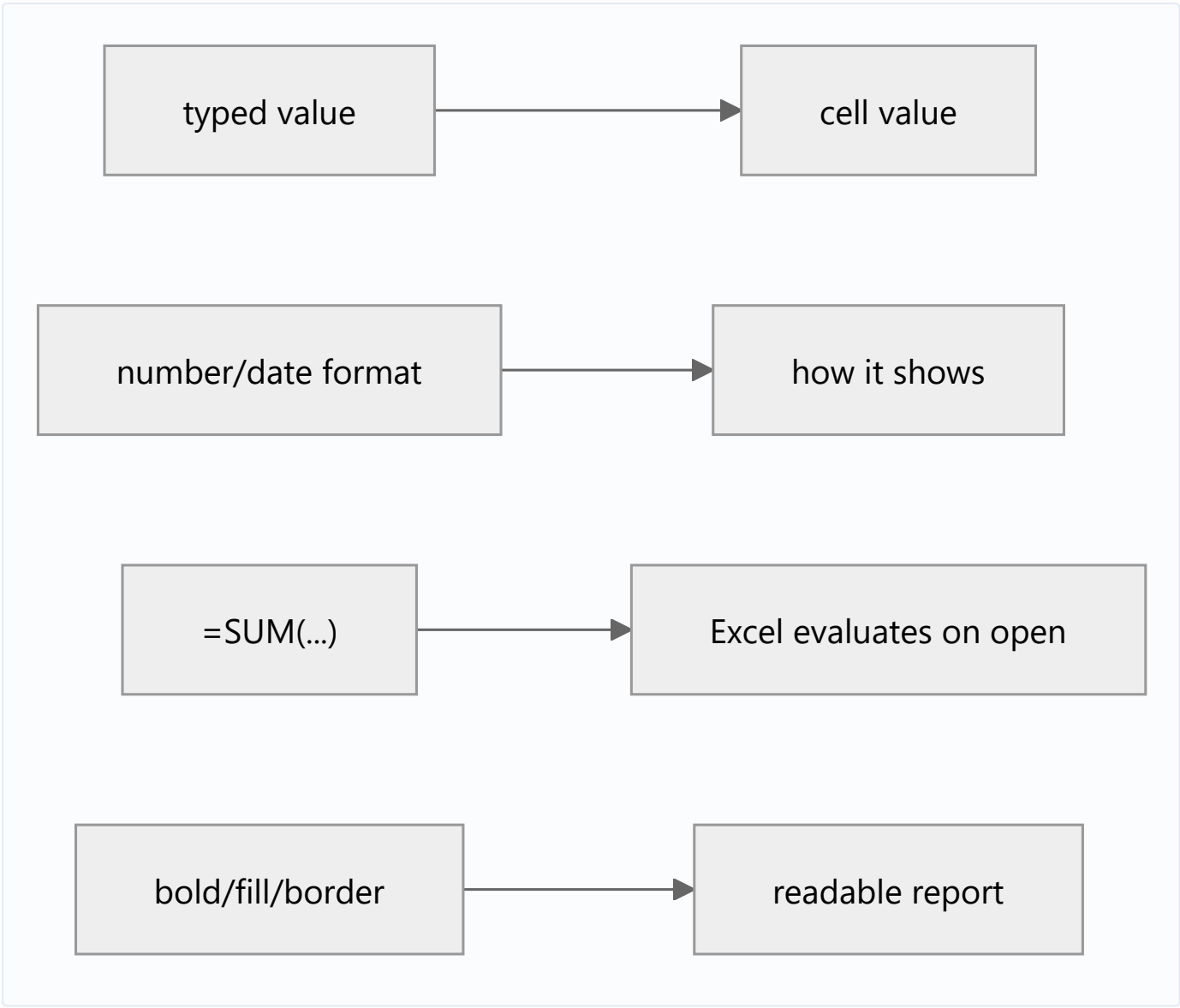
Dart VM Behavior

Not applicable beyond generation cost (isolate large/complex sheets).

Examples

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

void writeStyledReport(Excel excel) {
  final sheet = excel['Sales'];

  // Merged, bold title spanning A1:D1
  sheet.merge(CellIndex.indexByString('A1'), CellIndex.indexByString('D1'));
  final title = sheet.cell(CellIndex.indexByString('A1'));
  title.value = TextCellValue('Sales Report');
  title.cellStyle = CellStyle(bold: true, fontSize: 16,
      horizontalAlign: HorizontalAlign.Center);

  // Bold, filled header row
  final headers = ['Item', 'Qty', 'Date', 'Amount'];
  for (var c = 0; c < headers.length; c++) {
    final cell = sheet.cell(CellIndex.indexByColumnRow(columnIndex: c, rowIndex: 2));
    cell.value = TextCellValue(headers[c]);
    cell.cellStyle = CellStyle(bold: true, backgroundColorHex: ExcelColor.blue100);
  }

  // Typed data: number + date (NOT text) so Excel can sum/sort/format
  sheet.cell(CellIndex.indexByColumnRow(columnIndex: 0, rowIndex: 3)).value = TextCellValue('Widget');
  sheet.cell(CellIndex.indexByColumnRow(columnIndex: 1, rowIndex: 3)).value = IntCellValue(5);
  sheet.cell(CellIndex.indexByColumnRow(columnIndex: 2, rowIndex: 3)).value =
      DateCellValue(year: 2026, month: 7, day: 9);
  sheet.cell(CellIndex.indexByColumnRow(columnIndex: 3, rowIndex: 3)).value = DoubleCellValue(1499.00);

  // Formula: total of the Amount column (evaluated by Excel on open)
  sheet.cell(CellIndex.indexByColumnRow(columnIndex: 3, rowIndex: 10)).value =
      FormulaCellValue('SUM(D4:D9)');
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Numbers/dates as text	Can't sum/sort/format	Use typed cell values
Confusing value vs number format	Wrong display or math	Set value + format separately
Reading a date, getting a serial	Dates are serial numbers	Convert serial → `DateTime`
Expecting `excel` to do rich formulas/charts	Limited support	Use Syncfusion
Writing computed value instead of formula	Not "live" in Excel	Write a formula cell
Over-styling every cell	Bloats file/slow	Reuse styles; style selectively
Not merging for titles	Ugly layout	Merge title/header ranges

Best Practices

- Write **typed cells** (number/date/bool/formula), not strings, so Excel can sum/sort/format; set **number/date formats** for display (separate from value).
- Use **formulas** (`=SUM/AVERAGE/...`) for computed cells (evaluated on open); when **reading dates**, convert **serial numbers** → `DateTime` .
- Apply **styles/merges/widths** for readability but **reuse styles** and style selectively (avoid bloat); use **Syncfusion** for rich formats/formulas/charts.
- Offload complex/large sheet generation to an **isolate**; keep the styling/format logic reusable (05_excel_integration.md).

Performance

More styles/formulas/cells → larger file + slower encode. Interned/reused styles help. Formulas are cheap to write (Excel evaluates on open). Large styled sheets should be isolate-generated. Number formats are free (display metadata).

Advantages / Disadvantages

- **+** Real spreadsheet semantics (types, formats, formulas, styling, merges) — analysis-ready, readable exports.
- **−** Type/format correctness is fiddly (date serials!), `excel` limited for advanced needs, styling bloat, Syncfusion license for full features.

Interview Questions

1. 🟢 **Why must you set cell types instead of writing everything as text?** — So Excel can sum numbers, sort/format dates, and evaluate formulas — text cells break all of that.
2. 🟢 **How are dates stored in a spreadsheet?** — As serial numbers (days since an epoch) with a date format applied; when reading you often get a serial to convert to `DateTime` .
3. 🟡 **What's the difference between a cell's value and its number format?** — The value is the stored data; the number format is a display mask (e.g., `₹#,##0.00`) — the math uses the value, not the display.
4. 🟡 **How do formulas behave in a generated file?** — You write the expression (`=SUM(...)`) as a formula cell; Excel/Sheets evaluates it when the file opens.
5. 🟡 **When do you need Syncfusion over `excel` ?** — For rich number formats, reliable formulas, conditional formatting, charts, and large-file handling.
6. 🔴 **How do you make an exported report readable and analysis-ready?** — Typed cells + number/date formats + a bold/filled header + merged title + column widths + total formulas.

7. 🔴 **Why can over-styling hurt?** — Every distinct style/format bloats the file and slows encoding; reuse styles and style selectively.

Senior Engineer Tips

- Type your cells religiously and handle the date-serial conversion on read — "my sums are zero" and "dates show as 46000" are the two evergreen spreadsheet bugs.
- Write formulas rather than pre-computed values when the user should see live calculations; write cached values when they must match a snapshot exactly.
- Reuse a small set of styles (header/currency/date) instead of styling per cell; it keeps files small and generation fast.

Architect Perspective

Cells/styles/formulas are the semantic layer that makes exports useful. Encapsulating typed-cell mapping, reusable styles, and date/format handling in reusable helpers (behind the spreadsheet service) yields consistent, analysis-ready outputs and confines the fiddly type/serial logic to one tested place — the spreadsheet analog of PDF's reusable components (05_excel_integration.md, Module 35).

Summary

- Cells are typed (text/number/date/bool/formula); set number/date formats (separate from value); dates are serial numbers.
- Use formulas (evaluated on open), styles/merges/widths (reuse styles); `excel` = basic, Syncfusion = rich.
- Type correctness (esp. dates) is the main pitfall; isolate large/complex generation.

Revision Notes

- Typed cells: `TextCellValue` / `IntCellValue` / `DoubleCellValue` / `DateCellValue` / `BoolCellValue` / `FormulaCellValue`; value ≠ number format (display mask).
- Dates = serial numbers (convert on read); formulas (`=SUM(...)`) evaluated on open; styles (`CellStyle`)/merge/ `setColumnWidth` .
- `excel` basic styling/formulas; Syncfusion for rich formats/formulas/charts/conditional; reuse styles; isolate large sheets.

Practice Questions

1. Why write a number as `DoubleCellValue` not `TextCellValue`?
2. How are dates represented, and what must you do when reading them?
3. When do you need Syncfusion instead of `excel`?

Coding Questions

1. Write a styled header row + typed data row (number/date) with `excel`.
2. Add a merged title and a `SUM` total formula.
3. Convert a read date serial into a `DateTime`.

Mini Project

Styled report sheet (Flutter): Generate a sales report XLSX with a merged bold title, a bold/filled header row, typed number/date cells with number/date formats, sensible column widths, and a `=SUM(...)` total — reusing a small set of styles and offloading generation to an isolate.
Acceptance: numbers/dates are typed (sum/sort work in Excel); formats display correctly; header/title styled + merged; total formula evaluates on open; styles reused (no bloat); generated off the UI thread.

CSV & Data Interchange

> CSV is plain delimited text — **universal, tiny, and streamable** — but deceptively error-prone: fields containing the **delimiter, quotes, or newlines must be quoted** (and embedded quotes doubled), the **delimiter isn't always a comma** (locales use `;`), **encoding** matters (UTF-8 + BOM for Excel to read non-ASCII correctly), and everything is a **string** (no types — you parse/coerce). Use the `csv` package (`CsvToListConverter` / `ListToCsvConverter`) rather than naive `split(',')`, which breaks on the first quoted comma.

Introduction

CSV is the lowest-common-denominator interchange format — every tool reads it, and it streams row-by-row without loading a whole workbook. But "just split on commas" corrupts real data. This file covers correct CSV parsing/generation, the quoting/delimiter/encoding/locale edge cases, and when CSV beats XLSX.

Why this concept exists

Systems must exchange tabular data simply and universally; CSV is that format. But its simplicity hides rules (RFC 4180 quoting) that naive code ignores, causing silent corruption (a comma in an address splits a field). The `csv` package implements the rules correctly; understanding the edge cases prevents data loss.

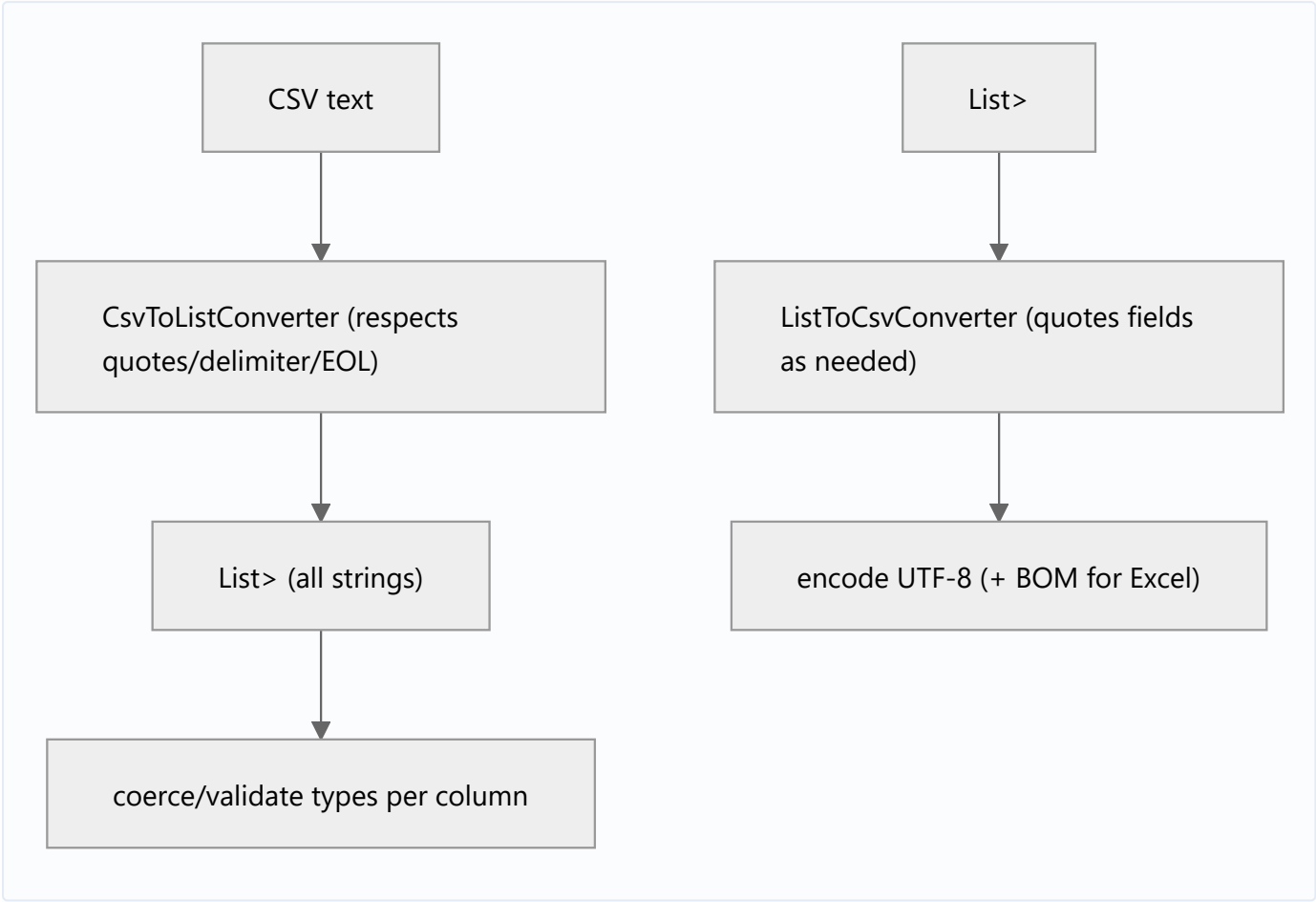
Real-world analogy

CSV is **passing notes with commas between words**: fine until a word itself contains a comma ("Smith, Jr.") — then you must **wrap it in quotes** so the reader doesn't mistake it for two words. Different countries even use a **semicolon** as the separator. And to read accented names, you must agree on the **alphabet (encoding)** first. Naive `split(',')` is a reader who ignores the quotes and mangles the note.

Problem Statement

Import a user CSV whose fields include commas, quotes, and newlines (addresses, notes), possibly semicolon-delimited and with non-ASCII names, and export data as CSV that Excel opens correctly with accents intact. You'll use the `csv` package with correct quoting/delimiter/encoding.

Internal Working



- **Use the `csv` package**, not `split(',')`:
  - **Parse**: `const CsvToListConverter().convert(text)` (or `.convert(text, fieldDelimiter: ';', eol: '\n')`) → `List<List<dynamic>>`. It handles **quoted fields**, **embedded delimiters/newlines**, and **doubled quotes** correctly.
  - **Generate**: `const ListToCsvConverter().convert(rows)` — it **quotes fields** that contain the delimiter/quote/newline and **doubles** embedded quotes automatically.
- **Quoting rules (RFC 4180)**: a field with a comma/quote/newline is wrapped in double quotes; an embedded `"` becomes `""`. The package does this — hand-rolling gets it wrong.
- **Delimiter/locale**: comma is common, but **locale/region** may use `;` (e.g., where `,` is the decimal separator). Let users/config choose the delimiter for import; pick a safe one for export (or match the target).
- **Encoding**: CSV is bytes → text; use **UTF-8**. For **Excel** to read non-ASCII correctly, prepend a **UTF-8 BOM** ( ) or Excel may misread accents. Handle input encoding (some files are Latin-1/UTF-16).
- **Typeless**: every field is a **string** — you must **coerce** (parse int/double/date) and **validate** per column on import (04_import_export_large_datasets.md); numbers/dates have no format, and locale affects decimal separators.

- **Streaming**: large CSVs can be processed **row-by-row** (stream the file, convert incrementally) — far lighter than loading a whole XLSX (Module 34).
- **CSV vs XLSX**: CSV — universal, tiny, streamable, but no types/styling/formulas/sheets. XLSX — typed/styled/multi-sheet, but heavier. Choose by need.

Memory Representation

Parsed CSV is a `List<List<dynamic>>` of strings (memory scales with rows unless streamed). Streaming holds only the current row(s). Output is a string → bytes.

Compiler Behavior

Not applicable.

Runtime Behavior

Parsing is linear; correct quoting handling means fields with commas/newlines stay intact. Wrong delimiter/encoding causes misparsed columns/garbled text.

Flutter Engine Behavior

None; pure Dart.

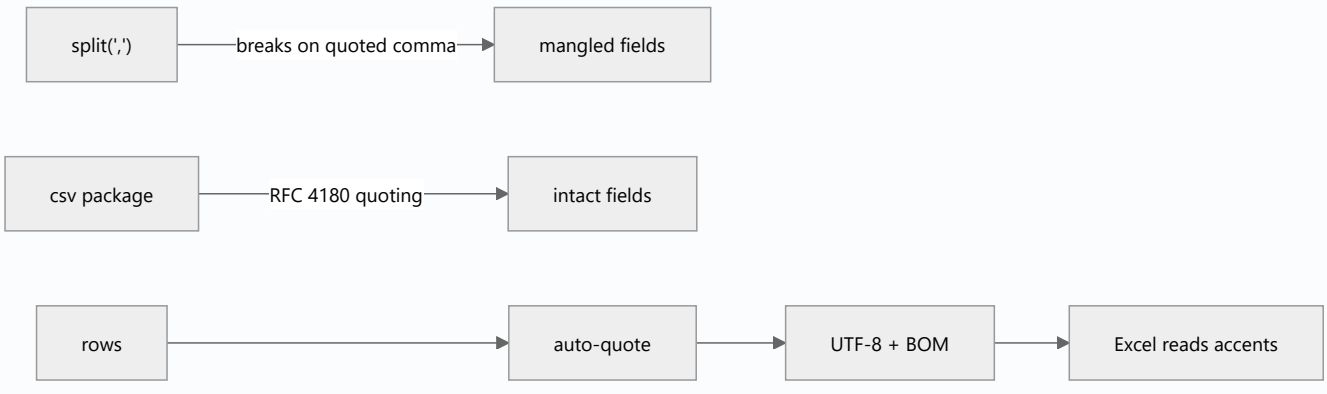
Dart VM Behavior

Cheap linear processing; stream large files to bound memory; heavy per-row coercion can be isolate-offloaded.

Examples

```
import 'package:csv/csv.dart';
import 'dart:convert';
import 'dart:typed_data';

// PARSE (handles quoted commas/newlines; configurable delimiter)
List<List<dynamic>> parseCsv(String text, {String delimiter = ','}) {
  return CsvToListConverter(fieldDelimiter: delimiter, eol: '\n').convert(text);
  // ["Smith, Jr.", "note with ""quotes""", ...] parse correctly — NOT split(',')
}

// GENERATE + encode UTF-8 with BOM so Excel reads accents (café, ₹) correctly
Uint8List toCsvBytes(List<List<Object?>> rows) {
  final csv = const ListToCsvConverter().convert(rows); // auto-quotes fields as needed
  const bom = [0xEF, 0xBB, 0xBF];                       // UTF-8 BOM for Excel
  return Uint8List.fromList([...bom, ...utf8.encode(csv)]);
}

// Typeless -> coerce/validate on import
int parseQty(dynamic cell) => int.tryParse('$cell'.trim()) ?? (throw FormatException('bad qty: $cell'));
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`split(',')` to parse	Breaks on quoted commas/newlines	Use `CsvToListConverter`
Hand-rolling quoting	Wrong for quotes/newlines	Use `ListToCsvConverter`
Assuming comma delimiter	Locales use `;`	Configurable/known delimiter
No UTF-8 BOM for Excel	Accents garbled	Prepend BOM for Excel
Treating fields as typed	CSV is all strings	Coerce + validate per column
Loading huge CSV whole	Memory	Stream row-by-row
Ignoring input encoding	Garbled text	Detect/handle UTF-8/Latin-1/UTF-16

Best Practices

- **Always use the `csv` package** (never `split(',')`); it handles **quoting, embedded delimiters/newlines, doubled quotes** on both read and write.
- Make the **delimiter configurable** (comma/semicolon by locale); **encode UTF-8** and **prepend a BOM** for Excel to read non-ASCII correctly; handle input encoding.
- Treat fields as **strings** → **coerce + validate types per column** on import (04_import_export_large_datasets.md); mind **locale decimal separators**.
- **Stream** large CSVs row-by-row (memory); choose **CSV vs XLSX** by need (universal/light vs typed/styled).

Performance

CSV is linear and streamable — far lighter than XLSX for large datasets (no XML/zip). Stream to bound memory; isolate heavy per-row coercion. Generation is cheap. CSV is the go-to for very large exports/imports.

Advantages / Disadvantages

- **+** Universal, tiny, streamable, human-readable, great for large datasets.
- **−** No types/styling/formulas/sheets, quoting/delimiter/encoding pitfalls, locale issues, manual coercion/validation.

Interview Questions

1. 🟢 **Why not parse CSV with `split(',')`?** — It breaks on fields containing quoted commas, newlines, or quotes; the `csv` package implements the quoting rules correctly.

2. 🟢 **What are CSV's core limitations vs XLSX?** — No types, styling, formulas, or multiple sheets — everything is a string you must coerce.
3. 🟡 **How do you make Excel read a UTF-8 CSV with accents correctly?** — Prepend a UTF-8 BOM (and encode UTF-8); otherwise Excel may garble non-ASCII.
4. 🟡 **Why might the delimiter not be a comma?** — Locales where comma is the decimal separator use `;` — make it configurable/known.
5. 🟡 **How are fields with commas/quotes/newlines represented?** — Quoted with double quotes, embedded quotes doubled (`""`) — RFC 4180; the package handles it.
6. 🔴 **When do you choose CSV over XLSX?** — Large datasets, universal interchange, streaming, or when types/styling aren't needed; XLSX when you need types/formatting/formulas/sheets.
7. 🔴 **How do you process a very large CSV without OOM?** — Stream it row-by-row (incremental convert), coercing/validating as you go, rather than loading it all.

Senior Engineer Tips

- Standardize on the `csv` package everywhere and ban `split(',')` in review — hand-parsing CSV is a guaranteed data-corruption bug on real-world fields.
- Add the UTF-8 BOM for Excel exports and make the delimiter configurable; the two most common "the file looks wrong in Excel" complaints are garbled accents and semicolon locales.
- For big data, prefer streaming CSV over loading XLSX; it's the difference between a snappy export and an OOM.

Architect Perspective

CSV is the interchange boundary: simple bytes/text that every system speaks, at the cost of typelessness and edge-case rules. Encapsulating correct parsing/generation (package-based, encoding/BOM/delimiter-aware, streamable) plus a per-column coercion/validation layer behind the spreadsheet service gives robust, lightweight interchange — complementing typed XLSX and integrating with import/export mapping (04_import_export_large_datasets.md, 05_excel_integration.md).

Summary

- Use the `csv` package (correct quoting/delimiter/newline handling) — never `split(',')`; make delimiter configurable; UTF-8 + BOM for Excel.
- CSV is typeless (coerce/validate per column), locale-sensitive, but universal/tiny/streamable.
- Choose CSV (light/universal/large) vs XLSX (typed/styled) by need; stream large files.

Revision Notes

- `CsvToListConverter` / `ListToCsvConverter` (handle quotes/embedded delimiters/newlines, doubled quotes); never `split(',')`.
- Delimiter configurable (`,` / `;`), UTF-8 encode + BOM for Excel, handle input encoding; fields are strings → coerce + validate; locale decimals.
- Stream large CSVs; CSV (universal/light/large) vs XLSX (typed/styled/sheets); behind the service.

Practice Questions

1. What breaks when you `split(',')` real CSV data?
2. Why add a UTF-8 BOM for Excel, and when?
3. When is CSV preferable to XLSX?

Coding Questions

1. Parse a CSV with quoted commas/newlines using the `csv` package.
2. Generate CSV bytes with a UTF-8 BOM for Excel.
3. Coerce + validate a numeric/date column from string fields.

Mini Project

Robust CSV interchange (Flutter): Build CSV import/export: parse a picked CSV (configurable delimiter, quoted commas/newlines, encoding-aware) into rows and coerce/validate typed columns (reporting bad rows); export data as CSV bytes with a UTF-8 BOM so Excel reads accents/₹ correctly — streaming large files. Acceptance: quoted/edge-case fields parse intact (no `split`); delimiter configurable; export opens correctly in Excel (accents intact); columns coerced + validated; large files streamed; behind the service.

Import/Export & Large Datasets

> Import/export is a **mapping + validation** problem: define an explicit **column ↔ model-field** mapping, **coerce and validate every cell** (collecting per-row errors rather than crashing on the first bad value), and for **large datasets** avoid loading everything into memory — **stream CSV row-by-row** and **offload heavy XLSX parse/generate to an isolate** (or paginate/stream via Syncfusion). The goal: a robust `List<Model> ⇄ bytes` pipeline that survives messy real-world files and doesn't freeze or OOM.

Introduction

This file connects spreadsheets to your app's data: turning uploaded files into validated models and models into exports, at scale. It covers the mapping/coercion/validation layer, error reporting, and the memory/performance techniques for large files — the difference between a toy importer and one that handles a 200k-row upload.

Why this concept exists

Real imports are messy (missing columns, wrong types, blank rows, bad dates); crashing on the first bad cell is unacceptable. And large files OOM or freeze the UI if loaded whole. A deliberate mapping + validation + streaming pipeline makes import/export robust, informative (row errors), and performant.

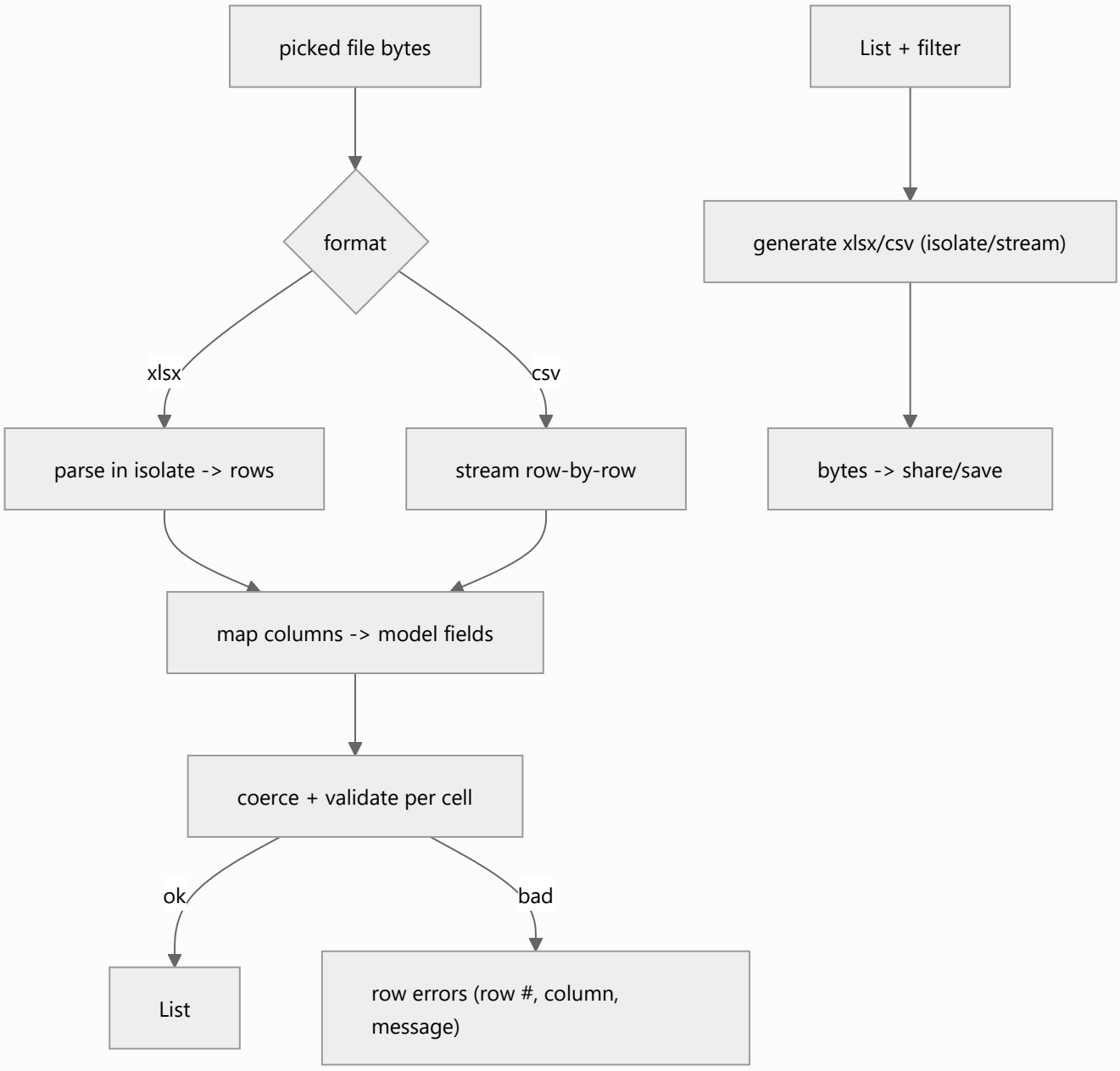
Real-world analogy

Importing is **customs inspection at a port**: each container (row) is checked against a manifest (mapping) — items that don't match (bad types/missing fields) are **flagged with a report**, not thrown overboard, and the rest pass. For a huge shipment you **process containers as they come off the ship** (streaming) rather than piling them all on the dock (loading into memory).

Problem Statement

Import a user-uploaded XLSX/CSV of products (mapping columns → a `Product` model), validating each row and returning both the parsed models **and** a list of row errors (so the user can fix them), then export a filtered dataset to XLSX/CSV — handling a 200k-row file without freezing or OOM. You'll build the mapping/validation pipeline + streaming/isolate handling.

Internal Working



- **Explicit mapping**: define column → field (by **header name**, not fragile position) — build a header→index map from the first row, so column reorder/extra columns don't break it. Document required vs optional columns.
- **Coerce + validate per cell**: parse strings → typed values (int/double/date/bool), trim, handle blanks/defaults, validate ranges/enums/required. Do **not** throw on the first error — **collect** `(rowIndex, column, message)` and continue, returning `{models, errors}` so the user sees all problems.
- **Row-level resilience**: skip/flag blank rows, tolerate extra/missing columns, handle date **serials** (XLSX) and **locale decimals** (CSV) (02_cells_styles_and_formulas.md, 03_csv_and_data_interchange.md).

- **Large files — memory**:
  - **CSV**: **stream** the file and convert **row-by-row** (bounded memory) — ideal for huge datasets (Module 34).
  - **XLSX**: `excel` loads the **whole workbook** (memory-heavy) → **isolate** it, or prefer **CSV** / **Syncfusion streaming** for very large data.
- **Offload**: run parse/validate and generation in an **isolate** (`compute`) so the UI stays responsive; pass **serializable** data (bytes/lists), report **progress** for long jobs.
- **Export**: filter/transform models → rows via the same mapping (inverse), generate XLSX (typed/styled) or CSV, hand bytes to share/save.
- **Idempotent/large export**: for very large exports, stream CSV rather than build a giant workbook.

Memory Representation

Streaming holds one row at a time; whole-workbook parsing holds all cells. Errors are a small list of records. Models are your domain objects. Keep the largest intermediate bounded (stream/isolate).

Compiler Behavior

Not applicable.

Runtime Behavior

Import parses → maps → validates → yields models + errors; large XLSX parse is slow/heavy (isolate). Export generates bytes (isolate for big/styled). Progress reported for long operations.

Flutter Engine Behavior

None; keep the UI isolate free by offloading — otherwise a big import visibly freezes the app.

Dart VM Behavior

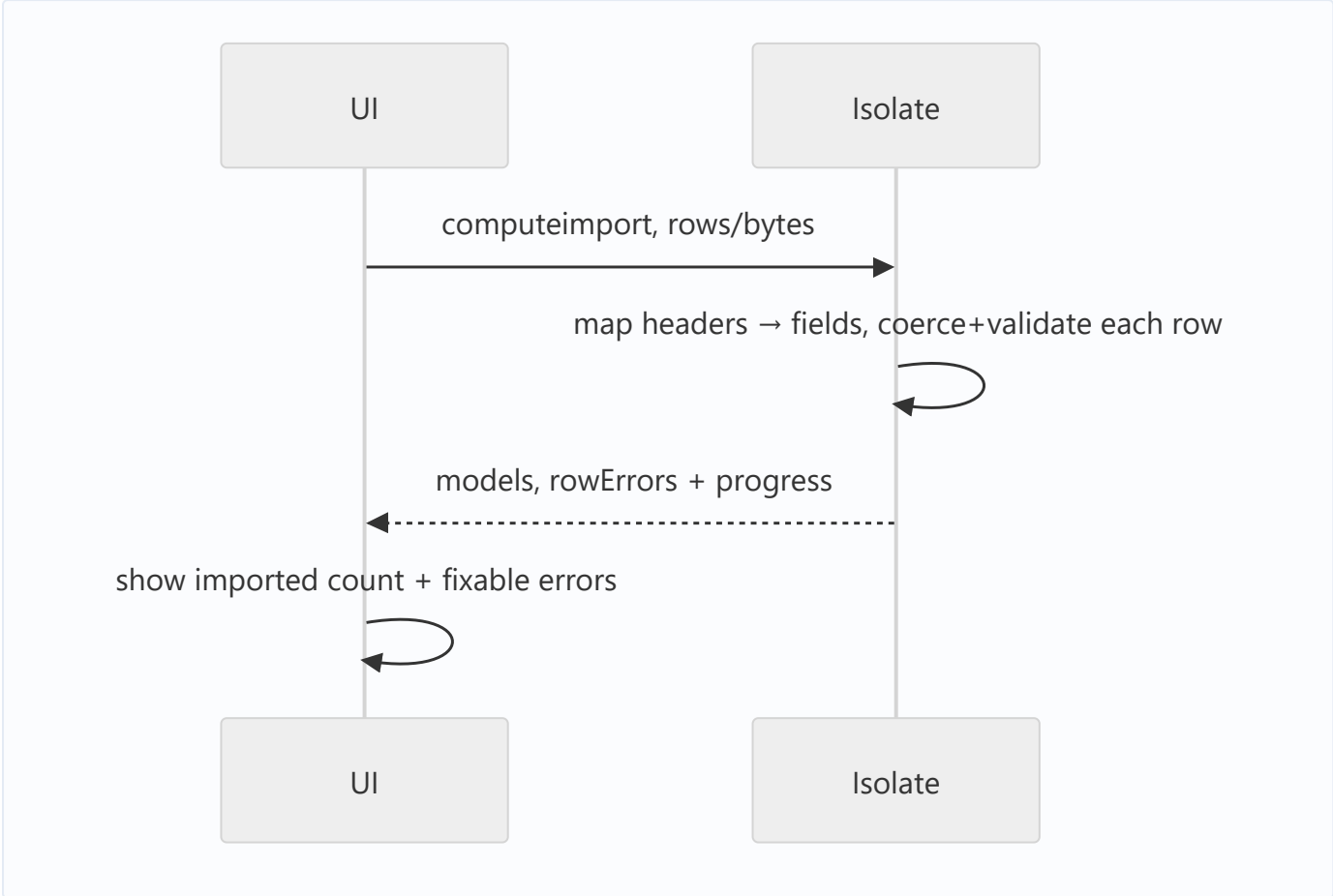
Heavy CPU/memory on the parse/generate isolate; UI isolate coordinates + shows progress. Streaming caps memory; `compute` moves work off the UI isolate.

Examples

```
// Import result: models + collected per-row errors (don't crash on first bad cell)
class ImportResult<T> {
```

```
  final List<T> models;
  final List<RowError> errors;
  ImportResult(this.models, this.errors);
}
class RowError { final int row; final String column, message;
  RowError(this.row, this.column, this.message); }

// Header-based mapping + per-cell validation (runs in an isolate for big files)
ImportResult<Product> importProducts(List<List<dynamic>> rows) {
  final header = {for (var i = 0; i < rows.first.length; i++) '${rows.first[i]}'.trim(): i};
  int col(String name) => header[name] ?? (throw FormatException('missing column: $name'));
  final models = <Product>[]; final errors = <RowError>[];

  for (var r = 1; r < rows.length; r++) {
    final row = rows[r];
    if (row.every((c) => '$c'.trim().isEmpty)) continue;    // skip blank rows
    try {
      final price = double.tryParse('${row[col('Price')]}'.trim());
      if (price == null || price < 0) { errors.add(RowError(r, 'Price', 'invalid price')); continue; }
      models.add(Product(
        name: '${row[col('Name')]}'.trim(),
        price: price,
        // coerce date/qty/etc. with the same guarded pattern...
      ));
    } catch (e) {
      errors.add(RowError(r, '-', '$e'));                    // collect, keep going
    }
  }
  return ImportResult(models, errors);
}

// Offload the heavy work; report both models and errors to the UI
// final result = await compute(importProducts, parsedRows);
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Position-based column mapping	Breaks on reorder/extra cols	Map by header name
Throwing on first bad cell	User can't fix in bulk	Collect per-row errors, continue
Loading huge XLSX on UI thread	Freeze/OOM	Isolate; prefer CSV/streaming
No validation/coercion	Corrupt models	Coerce + validate per cell
Ignoring date serials / locale decimals	Wrong values	Convert serials; handle locale
Building giant workbook for export	OOM	Stream CSV for very large exports
No progress for long jobs	Feels frozen	Report progress from isolate

Best Practices

- Map **by header name** (resilient to reorder/extra columns); **coerce + validate each cell**, **collecting per-row errors** and returning `{models, errors}` (never crash on the first bad value).
- **Stream large CSVs** row-by-row and **isolate** large XLSX parse/generate; prefer **CSV/streaming/Syncfusion** for very large datasets over loading a whole `excel` workbook.

- Handle **date serials, blanks, locale decimals, missing/extra columns**; **report progress** for long jobs.
- Reuse **one mapping** for import (columns→fields) and export (fields→columns); hand bytes to share/save; wrap behind a service.

Performance

Streaming (CSV) bounds memory; isolates keep the UI responsive; header-based mapping avoids re-scans. XLSX is the heavy case — isolate or avoid for huge data. Progress reporting maintains UX during long jobs. The pipeline turns "freezes on big files" into "handles 200k rows smoothly."

Advantages / Disadvantages

- **+** Robust to messy files (row errors), scales to large datasets (streaming/isolate), reusable mapping, responsive UI.
- **−** More code (mapping/validation/error model), memory/isolate discipline, format-specific edge cases (serials/locale).

Interview Questions

1. 🟢 **Why map columns by header name instead of position?** — Position breaks when columns are reordered or extra ones added; header mapping is resilient.
2. 🟢 **How should you handle a bad cell during import?** — Collect a per-row error and continue, returning models + all errors so the user can fix them in bulk — don't crash on the first.
3. 🟡 **How do you import a huge file without freezing/OOM?** — Stream CSV row-by-row and/or run parsing in an isolate; prefer CSV/streaming/Syncfusion over loading a whole `excel` workbook.
4. 🟡 **What format-specific pitfalls must import handle?** — XLSX date serials, blanks/missing/extra columns, and CSV locale decimal separators/quoting.
5. 🟡 **How do you keep the UI responsive during a long import/export?** — Offload to an isolate (`compute`) and report progress back to the UI.
6. 🔴 **How do import and export share logic?** — One column↔field mapping used forward (import) and inverse (export), keeping them consistent.
7. 🔴 **When would you stream a CSV export instead of building a workbook?** — For very large exports, to bound memory (a giant in-memory workbook would OOM).

Senior Engineer Tips

- Return `{models, errors}` from every importer and surface fixable row errors to the user — bulk imports live or die on good error reporting, not on the happy path.
- Map by header and keep one mapping for both directions; it eliminates a whole class of column-drift bugs and keeps round-trips consistent.
- Assume 100k+ rows: stream CSV, isolate XLSX, report progress — the demo works on 20 rows, production sends you a spreadsheet that OOMs.

Architect Perspective

Import/export is a data-boundary pipeline: messy external files ⇄ validated domain models, made robust by explicit mapping + per-cell validation with error collection, and scalable by streaming/isolates. Encapsulating this (one mapping, an `ImportResult`, isolate offloading) behind the spreadsheet service gives features a clean, resilient `List<Model> ⇄ bytes` API — the same validation/boundary discipline as networking/forms, applied to spreadsheets (05_excel_integration.md, 02 · json_serialization).

Summary

- Import/export = explicit header-based mapping + per-cell coercion/validation collecting row errors (never crash on first bad value).
- Large files: stream CSV, isolate XLSX (or use Syncfusion/streaming); handle serials/blanks/locale; report progress.
- Reuse one mapping both directions; return `{models, errors}`; bytes → share/save; behind a service.

Revision Notes

- Map by header name (resilient); coerce+validate each cell → `{models, errors}` (RowError: row/column/message); continue on error.
- Large: stream CSV row-by-row, isolate XLSX parse/generate (or Syncfusion streaming); handle date serials, blanks, missing/extra cols, locale decimals; progress.
- One mapping for import + export; bytes → file layer (Module 34); wrap behind service.

Practice Questions

1. Why collect row errors instead of throwing on the first bad cell?
2. How do you import a 200k-row file without freezing/OOM?
3. Why map by header name rather than column position?

Coding Questions

1. Build a header-mapped `importProducts` returning `{models, errors}`.
2. Stream a large CSV import row-by-row with validation.
3. Export filtered models via the inverse mapping to XLSX/CSV in an isolate.

Mini Project

Robust import/export pipeline (Flutter): Build a `Product` import/export: header-based column mapping, per-cell coercion + validation returning `{models, errors}` (row-level), streaming for large CSV and isolate offloading for XLSX, handling serials/blanks/locale, with progress reporting; export a filtered dataset via the inverse mapping to XLSX + CSV. Acceptance: bad rows reported (not crashing); column reorder tolerated (header mapping); 100k+ rows handled without freeze/OOM (stream/isolate); serials/locale handled; one mapping both directions; progress shown; behind the service.

Excel Integration (Capstone: A Spreadsheet Service)

> The maintainable shape: a `SpreadsheetService` exposing an **intent + model** API (`importProducts(bytes) → {models, errors}`, `exportProducts(models, {format}) → bytes`), hiding the chosen library (`excel`/Syncfusion/`csv`), the **format detection** (xlsx vs csv), the **mapping + validation** pipeline, and **isolate/streaming** offloading — so features pass domain data and get domain data back, never touching workbook APIs, cell types, CSV quoting, or file paths. It complements the PDF service (Module 35) as the app's data-file boundary.

Introduction

This module capstone unifies package choice, cells/formulas, CSV, and import/export-at-scale into one service. Scattering `excel` / `csv` calls across features couples them to library quirks and duplicates mapping/isolate logic. A `SpreadsheetService` centralizes it behind intent. This file shows the design and an end-to-end import→validate→export flow.

Why this concept exists

Spreadsheet features span library choice, format handling, cell typing, CSV edge cases, validation, and performance — all cross-cutting. Left in widgets they tangle and drift. One service — like repositories for data — isolates them behind a `List<Model> ⇄ bytes` API, consistent with clean architecture (Module 40).

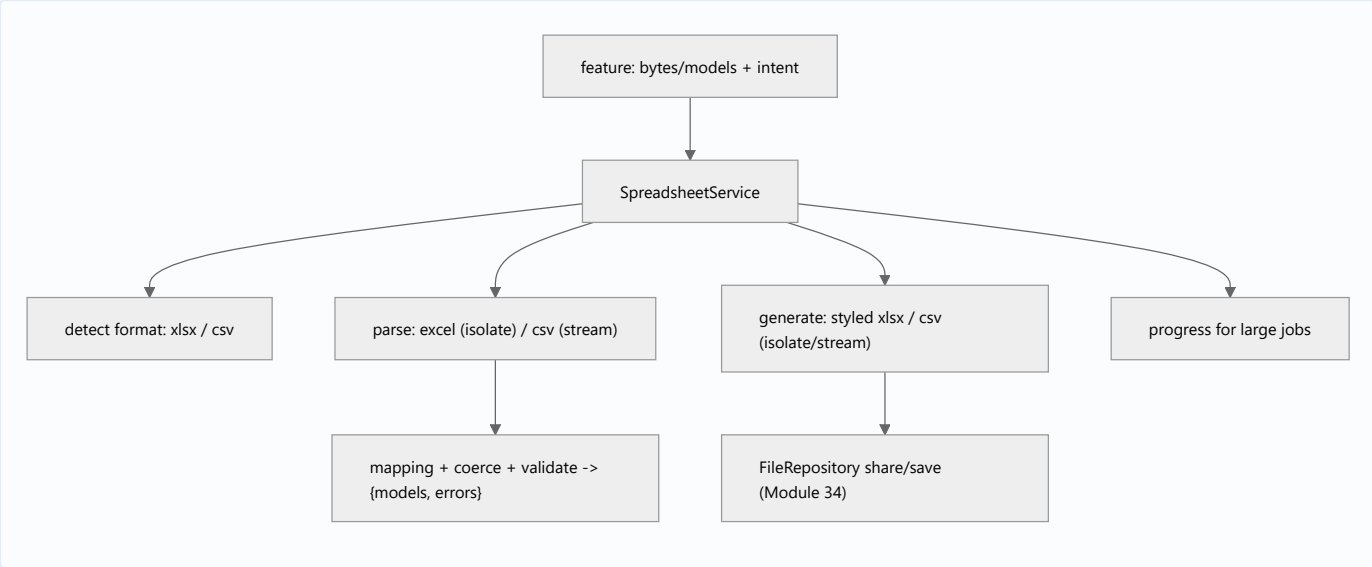
Real-world analogy

`SpreadsheetService` is the **data import/export department**: teams hand it a file ("load these products") or a dataset ("export these to Excel"), and it runs the customs/manifest process (mapping + validation), picks the right machinery (xlsx vs csv), and returns clean records or a finished file — teams never operate the spreadsheet machinery themselves.

Problem Statement

Deliver a product data manager: import a picked XLSX/CSV (detect format, map + validate, return models + row errors), export a filtered dataset to styled XLSX and to CSV, handle large files off the UI thread with progress, and share/save results — all behind one `SpreadsheetService`. You'll compose every file in this module.

Internal Working



- **Intent API**: `Future<ImportResult<T>> import<T>(Uint8List bytes, Mapper<T>)`, `Future<Uint8List> export<T>(List<T>, Mapper<T>, {Format})`, plus `shareExport` / `saveExport`. Features pass **models/bytes + a mapper**, never library types.
- **Format detection**: sniff extension/magic bytes → route to `excel` (xlsx) or `csv`; pick the export format by request.
- **Mapping + validation pipeline**: one `Mapper` (columns↔fields) used both directions; per-cell coerce/validate collecting **row errors** → `{models, errors}` (04_import_export_large_datasets.md).
- **Library encapsulation**: `excel` /Syncfusion/ `csv` live **inside** the service; swapping libraries (e.g., to Syncfusion for charts) doesn't touch features.
- **Performance**: **isolate** XLSX parse/generate, **stream** large CSV, **report progress**; heavy work never on the UI isolate (02 · isolates).
- **Persistence/share**: hand bytes to the `FileRepository` (Module 34) for save/share/pick.
- **Testability**: pure mapping/validation + service over abstractions → unit-test import (valid + messy inputs → models + errors) and export (models → bytes) without a device.

Memory Representation

The service holds no data (optionally progress state); models/bytes flow through. Streaming/isolates bound memory for large files. `ImportResult` carries models + a small errors list.

Compiler Behavior

Compiles against a `Mapper` /library-adapter abstraction (mockable); isolate entry points serializable.

Runtime Behavior

Import: detect → parse (isolate/stream) → map/validate → `{models, errors}`. Export: models → rows → generate (isolate/stream) → bytes → share/save. Progress reported for long jobs.

Flutter Engine Behavior

None; pure Dart. UI stays responsive because heavy work is offloaded.

Dart VM Behavior

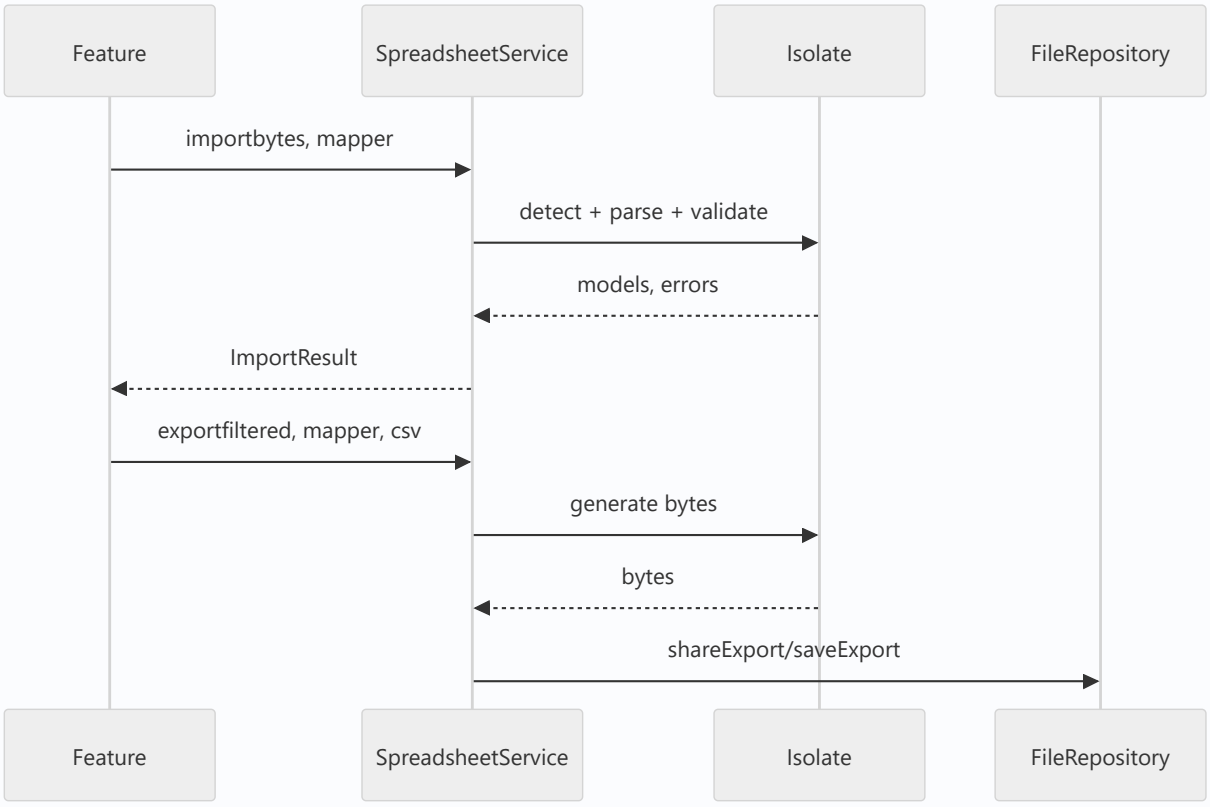
Parse/generate/validate on isolates; UI isolate coordinates + shows progress; streaming caps memory.

Examples

```
// Intent-based service — features pass models/bytes + a mapper, not library types
typedef Mapper<T> = ({T Function(Map<String, dynamic> row) fromRow,
                      Map<String, Object?> Function(T) toRow, List<String> columns});

abstract class SpreadsheetService {
  Future<ImportResult<T>> import<T>(Uint8List bytes, Mapper<T> mapper); // xlsx/csv auto
  Future<Uint8List> export<T>(List<T> items, Mapper<T> mapper, {SheetFormat format});
  Future<void> shareExport(Uint8List bytes, String filename);
  Future<ManagedFile> saveExport(Uint8List bytes, String filename);
}

class SpreadsheetServiceImpl implements SpreadsheetService {
  final FileRepository files;
  SpreadsheetServiceImpl(this.files);

  @override
  Future<ImportResult<T>> import<T>(Uint8List bytes, Mapper<T> m) =>
      compute(_parseAndValidate, _ImportArgs(bytes, m));   // detect + parse + validate (isolate)

  @override
  Future<Uint8List> export<T>(List<T> items, Mapper<T> m, {SheetFormat format = SheetFormat.xlsx}) =>
      compute(_generate, _ExportArgs(items.map(m.toRow).toList(), m.columns, format)); // isolate

  @override
  Future<void> shareExport(Uint8List b, String name) => files.shareBytes(b, name); // Module 34
  @override
  Future<ManagedFile> saveExport(Uint8List b, String name) => files.saveDocument(name, b);
}
```

```
// Feature usage — pure intent, domain in/out
// final result = await sheets.import(pickedBytes, productMapper);
// showErrors(result.errors); save(result.models);
// final bytes = await sheets.export(filtered, productMapper, format: SheetFormat.csv);
// await sheets.shareExport(bytes, 'products.csv');
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Library calls in features	Coupled, duplicated	One `SpreadsheetService` intent API
Duplicated mapping per feature	Drift between import/export	One `Mapper` both directions
Parse/generate on UI thread	Freeze on large files	Isolate/stream in the service
No error reporting	Users can't fix bad files	Return `{models, errors}`
Format handling in widgets	Scattered detection	Centralize detection in service
No persistence route	Lost files	Route via `FileRepository`
Untestable (device-only)	Slow/fragile	Pure pipeline + abstractions

Best Practices

- Expose an **intent + model** `SpreadsheetService` (`import` / `export` / `shareExport` / `saveExport`); keep library/format/CSV details **inside** it.
- Use **one** `Mapper` (columns↔fields) for both directions; return `{models, errors}` ; **isolate/stream** heavy work with **progress**.

- **Encapsulate the library** (swap `excel` ↔Syncfusion↔ `csv` without touching features); route **persistence via** `FileRepository`.
- Keep the **mapping/validation pure** and **unit-test** import/export (valid + messy inputs) without a device.

Performance

The service centralizes the levers (isolate, streaming, progress) so they apply uniformly; features get responsive import/export at any scale. Library encapsulation lets you switch to a faster/streaming backend (Syncfusion/CSV) without feature changes.

Advantages / Disadvantages

- **+** Clean `List<Model> ⇄ bytes` boundary, library-swappable, robust (row errors), scalable (isolate/stream), testable, consistent with PDF service.
- **−** Upfront service/mapper/abstraction design, discipline to route all spreadsheet ops through it.

Interview Questions

1. 🟢 **Why wrap spreadsheet work in a service?** — To hide library/format/CSV details behind a `List<Model> ⇄ bytes` intent API, centralize mapping/validation/isolate handling, and keep features testable.
2. 🟢 **What does the intent API look like?** — `import(bytes, mapper) → {models, errors}` and `export(models, mapper, {format}) → bytes`, plus share/save — features pass domain data + a mapper.
3. 🟡 **Why one `Mapper` for both directions?** — To keep import (columns→fields) and export (fields→columns) consistent and avoid drift.
4. 🟡 **How does the service stay responsive on large files?** — It isolates XLSX parse/generate, streams large CSV, and reports progress — off the UI isolate.
5. 🟡 **How does library encapsulation help?** — Features are decoupled from `excel`/Syncfusion/`csv`, so you can swap backends (e.g., Syncfusion for charts) without touching them.
6. 🔴 **How does import surface data-quality problems?** — It returns `{models, errors}` with per-row errors so the UI can show fixable issues rather than crashing.
7. 🔴 **How does this relate to the PDF service?** — Both are data-file boundaries behind intent services (generate/present/persist for PDF; import/export for spreadsheets), reusing the `FileRepository`.

Senior Engineer Tips

- Route every spreadsheet operation through the service and keep the library behind it; the day you need Syncfusion (or to drop it), only the service changes.
- Share one `Mapper` and return `{models, errors}` everywhere; consistent round-trips and good error reporting are what make import/export trustworthy.
- Bake in isolate/stream/progress from the start; spreadsheet features are the classic "fine in demo, freezes on the customer's real file" trap.

Architect Perspective

Excel integration is the app's tabular data-file boundary, mirroring the PDF service: one intent-based service encapsulates library choice, format handling, a shared mapping/validation pipeline, and isolate/streaming performance, exposing `List<Model> ⇄ bytes`. This keeps features simple, backends swappable, imports robust, and everything CI-testable — the same clean-architecture seam used for data, files, and documents (Module 35, Module 34, Module 40).

Summary

- One intent + model `SpreadsheetService` (`import` / `export` / `share` / `save`) hides library/format/CSV and mapping/validation/isolate details.
- One `Mapper` both directions; `{models, errors}`; isolate/stream + progress; library encapsulated; persistence via `FileRepository`.
- Pure, testable pipeline; complements the PDF service as the data-file boundary.

Revision Notes

- `SpreadsheetService`: `import(bytes, mapper)→{models,errors}`, `export(models, mapper, {format})→bytes`, share/save via `FileRepository`.
- One `Mapper` (columns↔fields); format detection + library (`excel` /Syncfusion/ `csv`) inside; isolate XLSX, stream CSV, progress.
- Pure mapping/validation + abstractions → unit-test; complements PDF service (Module 35); clean-architecture boundary.

Practice Questions

1. What belongs in the service vs the feature/mapper?
2. How does one mapper serve both import and export?
3. How is large-file performance handled centrally?

Coding Questions

1. Design a `SpreadsheetService` interface + impl with a `Mapper` and format detection.
2. Implement `import` (isolate, `{models, errors}`) and `export` (isolate/stream).
3. Unit-test import/export with valid + messy inputs (no device).

Mini Project

Product data manager (capstone — Flutter): Build a `SpreadsheetService` with `import(bytes, mapper) → {models, errors}` (auto-detect xlsx/csv, isolate/stream, row errors), `export(models, mapper, {format})` to styled XLSX + CSV (isolate/stream), and share/save via `FileRepository` — using one `Mapper` both directions and reporting progress. Depend on abstractions; add unit tests. Acceptance: features pass models/bytes + mapper only (no library types); import returns models + fixable row errors; export produces styled XLSX + Excel-correct CSV; large files handled off the UI thread with progress; library encapsulated (swappable); unit-tested; runs end-to-end on device.