Facet — Developer Guide

The files on disk, and how to work with them outside the app

> *A guide to where Facet keeps its data and exactly what each file and folder contains — so you can back it up, sync it, script it, or build your own tools around it with confidence. Where the User Guide explains the app and the Cookbook combines its features, this guide looks underneath both, at the files themselves.*

Facet · getfacet.co.uk · 2026

Everything Facet stores is a plain file you can read, copy, and (carefully) edit with ordinary tools. There is no hidden database engine and no proprietary binary format: a Facet database is a single human-readable JSON file, and the things that travel with it — captured inbox items, bundled images, backups — are all visible folders sitting beside it.

This guide is for anyone who wants to work with those files *from outside the app*: to script a backup, sync a database between machines, feed items in from another program, or simply understand what is on disk. It assumes you are comfortable with files, folders, and a little JSON. You do **not** need to build Facet from source — everything here is about the files a normal installation writes.

> ***This document is written to be extended.*** *Facet's on-disk layout is deliberately additive: new fields and new sibling folders are added without breaking old files. Where a structure is designed to grow, this guide says so, and points at the part of the layout you would extend.*

---

§ 1. The one rule that matters

Before anything else, internalise this:

> ***A Facet database file has exactly one writer.*** *While the app is open, it owns the database file and saves the whole thing from memory on a timer. If another program writes to that same* `.json` *file at the same time, the next in-app save will silently overwrite your change — and two programs writing one JSON file is a corruption risk regardless.*

So the safe rules of engagement are:

- **Reading is always safe.** Copy, parse, grep, or back up any Facet file at any time, app open or closed. Nothing here ever locks a file against readers.
- **Editing the database file by hand is only safe when Facet is closed.** Close the app, make your edit, then reopen. (Keep a copy first — see *Backups*.)
- **To add items from another program, never write the database — use the Inbox** (section 6). The Inbox exists precisely so an outside process can hand Facet a new item without touching the database file. It is the supported, race-free way in.

Everything that follows respects that single-writer rule, and so should anything you build.

---

§ 2. Where everything lives

Facet runs in one of two modes, which decides where its files sit.

Normal (installed) mode

The default. Two locations, both under your Windows user profile:

What	Where	Example
Databases (default folder)	`%AppData%\Facet\`	`C:\Users\you\AppData\Roaming\Facet\`
App settings	`%LocalAppData%\Facet\settings.json`	`C:\Users\you\AppData\Local\Facet\settings.json`

The database folder is the *default* — you can create or open a database anywhere you like (Documents, OneDrive, a network share), and many people do. The settings file always lives in Local AppData and records where your databases are (see section 8).

> *Note the split: databases default to **Roaming** AppData, settings live in **Local** AppData. They are separate folders.*

Portable mode

Drop an empty file named `Facet.portable` next to `Facet.exe`, and Facet keeps everything inside its own application folder instead — settings in `settings.json` beside the exe, databases under a `Data\` subfolder. Nothing is written to your user profile, so the whole thing runs from a USB stick and leaves no trace on the host PC. Remove the marker file to go back to normal mode.

(Defined in `Settings.cs` — `IsPortable`, `SettingsPath`, `DefaultDatabasePath`.)

---

§ 3. The database file

What it is

One database = one `.json` file. The name is up to you (`Work.json`, `Radio Log.json`, …); a freshly-initialised default database in the data folder is called `db.json`. When Facet is pointed at a *folder* rather than a file, it opens `db.json` if present, otherwise the first `*.json` it finds there.

The format is **indented, UTF-8 JSON with camelCase property names** — readable and diff-friendly. You can open it in any text editor.

How saves work

While the app is open, Facet holds the entire database in memory and writes it back on a **debounce timer (≈1.5 seconds after the last change)**. **Ctrl+S** / **File → Save Now** forces an immediate write, and the app saves on exit and when you switch databases.

Saves are **atomic**: Facet writes to a `<name>.json.tmp` file first, then replaces the real file in one filesystem operation. You will therefore never catch the database half-written — but you *may* see a short-lived `.tmp` file beside it during a save. Leave it alone; it is cleaned up as part of the write.

(Defined in `Storage.cs` — `Save`; debounce in `Workspace.cs`.)

Top-level shape

The root JSON object is the database. Its main properties:

```
{
  "name": "Activities Planner",   // display name, shown in the title bar
  "schemaVersion": 27,            // on-disk format version (see below)

  "items":       [ /* the data — see §3.1 */ ],
  "categories":  [ /* the category tree — see §3.2 */ ],
  "rules":       [ /* auto-tagging rules */ ],
  "automations": [ /* named action sequences */ ],
  "views":       [ /* saved views + their window placement */ ],

  "palettes":      [ /* named colour palettes */ ],
  "activePalette": "Standard",

  "printingProfiles": [ /* saved print-formatting sets */ ],
  "printTemplates":   [ /* composite print documents */ ],

  "detail": { "left": null, "top": null, "width": null,
              "height": null, "zOrder": null, "isOpen": true },

  // per-database settings (the Properties dialog)
  "undoDepth": 50,
  "desktopBackground": "",
  "autoRefreshOnCategoryEdit": false,
  "remindersEnabled": true,
  "weekStartDay": "Monday",
  "showCompletion": true,
  "bundleImages": false,
  "prettifyCategoryNames": false
}
```

Everything that belongs to *this* database is in this one file — its items, its structure, its colours, its window layout, and its per-database settings. The only things stored *outside* it are app-level preferences (section 8) and the sibling folders described later.

(Root model: `Database` in `Models.cs`.)

3.1 An item

Each entry in `items` is one captured thought. The fields you are most likely to read or edit:

```
{
  "id": "9f2c…",                  // stable GUID (hex, no dashes) — the item's identity
  "text": "Call Fred next Tuesday about pricing",
  "notesMd": "Long-form **Markdown** notes…",

  "categories": ["people/fred", "actions/calls"],  // assigned tags (paths)
  "manualCategories": ["people/fred"],             // of those, the ones you added by hand

  "due": "2026-06-09T00:00:00",   // ISO date/time, or null
  "dueEnd": null,                  // optional end of a range
  "isComplete": false,
  "completedOn": null,

  "links": [                       // first-class file / URL attachments
    { "label": "", "target": "C:\\docs\\report.pdf" }
  ],

  "numbers":         { "Pledges": 250.00 },   // named numeric columns
  "categoryNumbers": { "frequency": 14250.5 },// typed-category values, keyed by category path
  "categoryDates":   { "heard": "2026-06-01T00:00:00" },
  "categoryTexts":   { "sio": "599" },

  "created":  "2026-06-07T09:14:00",
  "modified": "2026-06-07T09:14:00",
  "source":   ""                   // "" = typed in the app; "Explorer" = captured from outside
}
```

A few things worth knowing if you plan to generate or rewrite items:

- **`id`** is the item's identity. **Import** matches on it (duplicates are skipped), so if you clone an item by copying its JSON, give the copy a fresh GUID or the two will be treated as the same item on merge.
- **`categories`** holds category *paths* (`parent/child/grandchild`), not display names. A tag only resolves to a real category if that path exists in `categories` — otherwise it shows up as an *orphan tag* in the app.
- **`source`** is provenance. Leave it `""` for hand-made items; the Inbox sets it for you (section 6).

(Item model: `Item` in `Models.cs`.)

3.2 A category

Each entry in `categories` is one node of the category tree:

```
{
  "id": "1a2b…",
  "path": "people/fred",      // slash-separated; the tree is implied by the paths
  "matchRule": { /* condition that auto-assigns this category, or null */ },
  "mutexGroup": null,          // shared name ⇒ pick-one (mutually exclusive)
  "chipColor": "#E0EAFF",
  "suppressCheckbox": false,   // items here are notes, not tasks
  "valueType": "None",         // None | Number | Date | Text (typed categories)
  "showTotal": true            // Number categories only
}
```

The tree is **defined by the paths**: every ancestor of a real category must itself be a category (so `people/fred` requires a `people` entry too). If you add categories by hand, add the parents as well.

(Category model: `Category` in `Models.cs`.)

Schema version and forward compatibility

`schemaVersion` (currently **27**) marks the on-disk format. When a newer build opens an older database it runs one-time, additive **migrations** to bring it up to date — and writes a one-off backup first (section 4). The design is deliberately tolerant:

- **Older Facet opening a newer file** keeps the fields it understands and, for anything genuinely unrecognisable in a *rule* or *automation*, drops just that one rule/automation rather than failing the whole load (you get a one-line warning). Unknown plain *fields* are simply ignored.
- **Adding a field** in a future version is safe by construction: missing fields fall back to a sensible default when an old file is loaded.

The practical upshot for you: a database file is robust to round-tripping through tools that don't know every field, but **if you let an older Facet save a newer database, newer-only data it didn't load will be lost.** When in doubt, use the latest version to edit, and keep a backup.

(Migrations: `Migration.cs`. The forward-compat load path: `Storage.LoadFromJson`.)

---

§ 4. Backups

Every time Facet saves, it first ensures a **daily backup** exists — a plain copy of the database file, taken once per calendar day:

```
<database folder>\Facetbackups\<name>-YYYY-MM-DD.json
```

So `Work.json` gets backed up to `Facetbackups\Work-2026-06-07.json` the first time it is saved today, and not again until tomorrow. Before a schema migration, Facet also writes a one-off, labelled copy (`<name>-pre-rules-migration.json`, etc.) as an extra safety net.

Things to know:

- **Backups are never rotated or deleted automatically.** They accumulate one per day. If you want a retention policy, prune the `Facetbackups` folder yourself (a scheduled script deleting files older than *N* days is a fine approach — these are ordinary `.json` copies).
- **The folder location is configurable.** By default it sits beside the database; you can redirect *all* backups to one folder from **File → Properties** (it is recorded as `backupFolderPath` in app settings).
- **A backup is just a database file.** To restore one, close Facet, copy it over your live `.json` (or open it directly with **File → Open Database…**).
- *(Older databases may have a legacy `backups\` folder; Facet renames it to `Facetbackups\` automatically the first time it opens such a database.)*

(Defined in `Storage.cs` — `EnsureDailyBackup`, `WriteOneOffBackup`, `BackupDir`.)

---

§ 5. The sibling folders, at a glance

A database is more than its `.json` file. Depending on what you've used, up to three siblings sit beside it, all named after the database file:

```
C:\Users\you\Documents\
    Work.json                ← the database
    Work.json.inbox\         ← capture queue (hidden)   — section 6
    Work-images\             ← bundled note images       — section 7
    Facetbackups\            ← daily backups             — section 4
```

The first two are described next. All of them are plain files and folders you can copy — but to move a database *and its companions* as a unit, prefer **Publish as Facet package** (section 7), which bundles them correctly and fixes up the internal references for you.

---

§ 6. The Inbox — adding items from outside the app

The Inbox is the **supported way for any outside program to add an item to a Facet database** without breaking the single-writer rule. It is what the Windows Explorer *"Add to Facet"* right-click command is built on, and it is designed to back further producers in future — a browser clip, a share target, a command-line tool, an email drop.

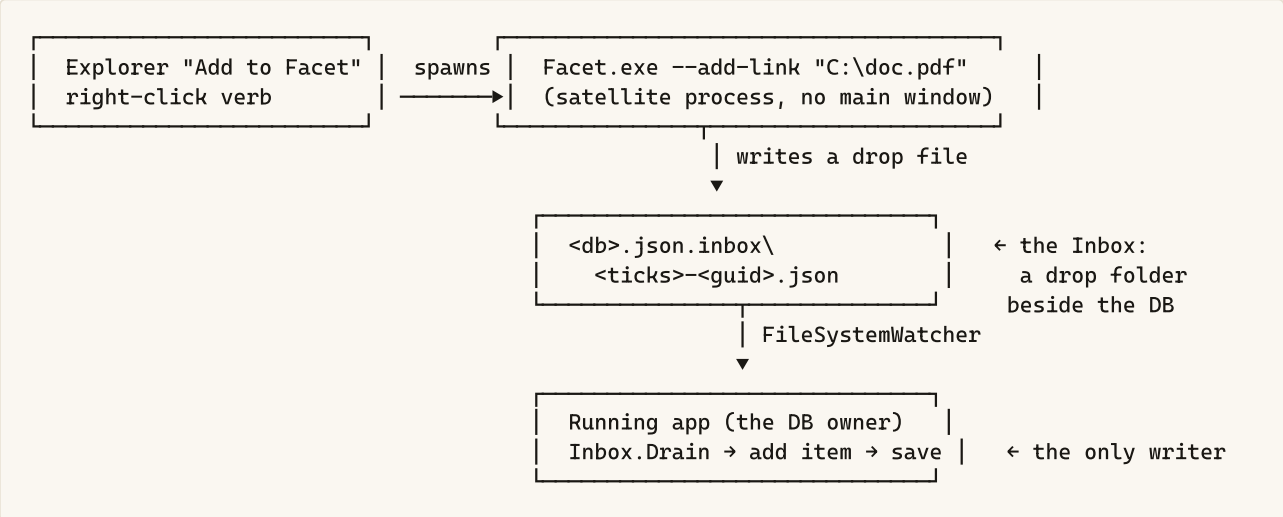
This section is the full reference for that seam: the problem it solves, the file protocol and its guarantees, how to build your own producer, and — as a worked example — exactly how the Explorer command is wired on top of it. If you are adding a new way to get items into Facet from outside the app, sections 6.4–6.5 are the contract you code against; everything in 6.6 is one consumer of it.

6.1 The problem it solves

A Facet database is a single JSON file, and (section 1) it has exactly one writer. The running app keeps the whole database in memory and writes it back on a debounce timer. If an outside process also wrote to that file, the next in-memory save would silently clobber the external edit — and two writers racing on one JSON file is a corruption risk regardless.

So the rule is firm: **a database file has exactly one writer — its owner.** The owner is the running app, or (when the app is closed) the next session that opens the file.

A capture has to add an item while the database may be open. It resolves the tension by *never writing the database file*. Instead the producer writes a small, separate message — an **inbox entry** — into a sibling drop folder. The database's single owner picks the message up and applies it. The producer and the owner never touch the same file, so there is no race and nothing to lock.



If the app is closed when capture happens, the drop files simply accumulate; the next session drains them on open. Nothing is lost and nothing needs to be running.

6.2 Layout

Each database has an inbox folder, a **hidden** sibling named after the database file (`Inbox.DirFor(dbPath)` computes the path):

```
C:\Users\you\Documents\
    Work.json                              ← the database
    Work.json.inbox\                       ← its inbox (hidden)
        <ticks>-<guid>.json                ← a pending drop
        <ticks>-<guid>.json.processing.json  ← a claimed drop (mid-ingest)
        <ticks>-<guid>.json.bad            ← a quarantined malformed drop
```

One inbox per database keeps captures routed correctly even when several databases share a folder. The folder is marked Hidden (best-effort) to keep it out of your way in Explorer. A producer writes a small JSON **drop file** into it; it never touches the database. The running app watches the folder and ingests new drops within a second (and also drains the folder on every database open, so anything left while the app was closed is picked up next time). **One drop file becomes exactly one item.**

6.3 The drop-file format

A drop file is one `InboxEntry`, serialized as indented camelCase JSON:

```
{
  "itemText":   "Quarterly numbers",   // the item's heading; "" ⇒ derive one from the first targ
  "notes":      "Optional **Markdown** notes",
  "targets":    [ "C:\\docs\\q3.pdf", "https://example.com" ],  // file/folder paths and/or http(s
  "createdUtc": "2026-06-07T14:30:00Z",
  "source":     "Explorer"             // who captured it; stamped onto the item's "source"
}
```

Each target becomes a clickable **link** on the created item; multiple targets become multiple links on that one item. The item is run through the rules engine for auto-tagging, just like a typed one.

Provenance (`source`). A producer stamps `source` with its identity; on ingest the consumer copies it onto the created item's `source` field. Items typed directly into the app have an empty `source`. The well-known values live in one place — the `ItemSource` class in `Inbox.cs` — so the set stays discoverable as producers are added:

```
public static class ItemSource
{
    public const string Explorer = "Explorer";   // the Windows Explorer "Add to Facet" verb
    // …a new producer adds its constant here.
}
```

A new producer should add a constant to `ItemSource` and set it on every entry it deposits, rather than hard-coding a string at the deposit site.

6.4 The three operations, and the guarantees

The protocol is a small static class — `Inbox.cs` — with three operations and a strict file protocol. The guarantees live in the file *moves*, so understanding the protocol matters more than the method signatures.

Method	Who calls it	Contract
`Deposit(dbPath, entry) → string`	The **producer** (satellite / extension)	Writes a uniquely-named drop file; returns its path. Creates the inbox if needed. Never touches the database.
`Drain(dbPath) → List<(ClaimPath, Entry)>`	The **owner** (running app / next session)	Atomically claims and parses every pending drop, oldest first. Malformed drops are quarantined and skipped.
`Complete(claimPath)`	The **owner**, after a successful save	Deletes the claimed drop. Best-effort.

These are the properties any consumer can rely on — and the reasons the file dance looks the way it does. Don't "simplify" them away:

- **A producer never corrupts the database.** It only ever writes files *inside the inbox folder*. The database has one writer.
- **No reader ever sees a half-written drop.** `Deposit` writes to a `.tmp` name and renames it into the final `.json` name (an atomic rename on NTFS). The temp name is deliberately **not** `*.json.tmp` — Windows globbing and `FileSystemWatcher` can mis-match `*.json.tmp` as `*.json`, which would surface the partial file.
- **Capture order is preserved.** The file name is `{UTC ticks:D19}-{guid:N}`, so a plain ordinal filename sort in `Drain` yields deposit order. The Guid is a tie-breaker so concurrent deposits in the same tick never collide.
- **An entry is ingested at most once, even with two drainers.** `Drain` claims each drop by renaming it to `<name>.processing.json` *before* reading it. The rename is atomic and exclusive: whoever wins owns the entry; a loser's move throws and it skips that drop. This makes draining safe against a second app instance or a re-entrant watcher tick.
- **A crash mid-ingest re-ingests rather than loses.** The claim file is deleted only by `Complete`, which the owner calls *after* the database save succeeds. Crash before that and the `.processing.json` file remains (see the caveat in 6.7).
- **A malformed or empty drop cannot wedge the queue.** Unparseable JSON, or an entry with no non-blank target, is moved to `<name>.bad` and skipped. The `.bad` file is left for inspection; it is never read again.

6.5 Writing your own producer

To add a new capture source you generally need **a producer** and, usually, nothing on the consumer side — the running app already drains any inbox for the database it has open, and the next session drains on open.

If your extension can determine the target database path, capturing an item is two lines through the supported API:

```
Inbox.Deposit(dbPath, new InboxEntry
{
    ItemText   = "Quarterly numbers",          // or "" to derive a title from the target
    Targets    = new List<string> { url },      // file/folder paths and/or http(s) URLs
    CreatedUtc = DateTime.UtcNow,
    Source     = ItemSource.MyProducer,         // add a constant to ItemSource first
});
```

If you are writing the drop file **directly** (from a non-.NET tool, say), follow the same file protocol the app uses — the guarantees in 6.4 live in the file names, so don't improvise:

1. **Name it** `<ticks>-<guid>.json` — 19-digit zero-padded UTC ticks, a dash, then a GUID (hex, no dashes). The tick prefix makes a plain filename sort equal capture order; the GUID prevents collisions.
2. **Write to a temp name, then rename into place.** Write the bytes to a file ending `.tmp` (deliberately *not* `*.json.tmp`), then move it to the final `.json` name. Keep the temp file on the **same NTFS volume** as the inbox so the rename stays atomic — the inbox is always a sibling of the database, so keep it that way (don't redirect the inbox to a temp drive).
3. **Declare your provenance.** Add a constant to `ItemSource` and set `source` on every entry, so items you create are distinguishable from hand-typed ones (which have `source: ""`) and from other producers' captures.
4. **Resolve the target database read-only** from app settings — `databasePath` is the current one, `recentDatabasePaths` the picker list, and `Settings.DefaultDatabasePath` the fallback (section 8). Reading settings never disturbs a running app.
5. **Deposit one entry at a time.** Don't batch into one giant entry unless you genuinely want a single item with many links.
6. **Consider idempotency** only if your producer can fire repeatedly for the same source — see 6.7.

That is the whole contract. The item appears in the app within a watcher tick if the database is open, or on next open otherwise — and **the app need not be running** when you drop the file.

If your launch model spawns multiple processes (a share target invoked once per item, like Explorer's verb), reuse the coalescing pattern in 6.6 — stage paths, elect one collector via a session mutex, show a single dialog — rather than depositing N entries from N dialogs. A single long-lived process (a browser native-messaging host, a CLI run once) needs none of that; just `Deposit`.

> ***What you should not do:*** *don't open the database "just to append one item" (that reintroduces the two-writer problem the Inbox exists to avoid); don't write into the inbox folder by hand when you could go through* `Inbox.Deposit` *and get the temp→move atomicity and correct file name for free; and don't change the* `<ticks>-<guid>` / `.tmp` / `.processing.json` / `.bad` *naming conventions — the at-most-once, in-order, never-corrupt guarantees are encoded in them.*

6.6 Worked example — the Explorer "Add to Facet" command

The Windows Explorer right-click *"Add to Facet"* command is the reference producer. It is more involved than a minimal producer because Explorer launches a verb **once per selected file**, so it has to coalesce a burst of processes into a single dialog. The pieces:

The verb (`ShellIntegration.cs`). The command is a **per-user registry verb**, not an in-process COM handler:

```
HKCU\Software\Classes\*\shell\Facet.AddLink
    (default)      = "Add to Facet"
    Icon           = "<exe>,0"
    \command
        (default)  = "<exe>" --add-link "%1"
```

This is deliberate: it lives under `HKCU`, so install/uninstall needs no admin rights; nothing of Facet's is loaded into `explorer.exe`, so a Facet bug can never lock or crash the shell — the verb just launches an `.exe`; and `Install()` is idempotent, rewriting the command line each launch so the path stays correct after the app is moved or updated. On Windows 11 the verb appears under "**Show more options**" (the classic context menu). Install/uninstall in-app via **File ▸ "Add to Facet in Explorer menu"**, or headless via `Facet.exe --install-shell` / `--uninstall-shell` (handled in `App.xaml.cs`).

Satellite mode (`App.xaml.cs`). `OnStartup` checks the command line *before* building the `Workspace` or showing a window. If `--add-link` is present it handles that and returns — the full app never spins up. A satellite sets `ShutdownMode.OnExplicitShutdown` so it survives the modal dialog, stages its path, and races for the collector mutex.

Coalescing the launch burst (`AddLinkCoordinator.cs`). Select five files and Windows launches `Facet.exe --add-link` five times, near-simultaneously. The coordinator turns N launches into **one dialog and N captures**:

- **Staging.** Every process atomically drops its path into a per-user staging folder (`<settings dir>\addlink-staging\<ticks>-<guid>.path`), written to a `.writing` temp name then moved into place so a reader never sees a partial file.
- **Election.** Each process tries `WaitOne(0)` on a session-local named mutex (`Local\Facet.AddLink.Collector.v1`). Exactly one wins and becomes the **collector**; the rest have already staged their path, so they just exit.
- **Collection.** The collector sleeps `CollectionWindowMs` (350 ms — longer than Explorer's launch spread, short enough not to feel laggy on a single add), then drains the staging folder in selection order.
- **Staleness.** A staged path whose tick-stamp is older than 10 s is treated as an orphan from a crashed collector and discarded, so it can never contaminate an unrelated capture made minutes later.

The capture dialog (`AddToFacetDialog.xaml.cs`). Lets you pick a **target database** (defaulting to the one in use, then the recent list, de-duplicated; read-only access to `Settings` so the running app is never disturbed) and type **one line of item text** plus optional Notes. For a multi-file selection it offers **one item per file** (one `InboxEntry` per path) or **one item, many links** (a single entry whose `Targets` carries every path). The dialog **never touches a database** — it only produces the `InboxEntry` list via `BuildEntries()`; the caller deposits them through `Inbox`.

The consumer side (`Workspace.cs`). The running app owns the database and so owns draining:

- `SetupInboxWatcher()` points a `FileSystemWatcher` at the current database's inbox folder, re-pointed on every database switch. It watches `*.json` for `Created` / `Renamed` (covering the satellite's temp→final move) and creates the folder so there is always something to watch.
- Watcher callbacks fire on a threadpool thread, so `ScheduleInboxDrain()` marshals the work onto the UI dispatcher and coalesces the burst of events one deposit can raise into a single pass.
- `DrainInboxNow()` claims and ingests every pending entry; it is also called once on every database open to pick up whatever accumulated while the app was closed.
- A low-frequency **poll backstop** (a 30 s `DispatcherTimer`) runs for the life of the app alongside the watcher. The watcher delivers captures live in the common case, but it has blind spots — cloud-synced folders (OneDrive/Dropbox), network shares, internal-buffer overflow under a flood, or a silently-dropped subscription. The poll guarantees those captures still arrive within one interval. It is cheap: `Drain` is a no-op on an empty inbox, and the claim-once rename can't double-add even when a tick races a watcher pass.
- `AddItemFromInbox()` builds the item: the captured text (or a title derived from the file/folder name / URL when blank), run through categorisation for auto-tagging, with each target attached as a link, and the entry's `source` stamped onto the item for provenance. It is **one undo step** and **deliberately never honours a discard rule** — a file you explicitly captured must not silently vanish — and it raises `ItemCreated` so automations see it.

6.7 Known caveats when extending

- **Orphaned `.processing.json` files.** If the owner crashes *between* claiming a drop and `Complete`, the `.processing.json` file is stranded — `Drain` skips `*.processing.json`, so it is never retried. This is an accepted trade-off (it favours never double-adding over never losing). If a future high-volume producer needs it, add a sweep that re-activates `.processing.json` files older than some threshold (mirror the staleness logic in `AddLinkCoordinator`).
- **No dedupe in the Inbox itself.** Re-ingest safety relies on the *consumer* tolerating a re-added item, or on `Complete` having run. `AddItemFromInbox` does not dedupe; this is fine for human-paced capture. A high-volume producer should carry its own idempotency key (in `itemText` or a target) and have the consumer check it.
- **Same-volume requirement.** The atomic-rename guarantees hold within one NTFS volume. The inbox is always a sibling of the database, so this is satisfied by construction — keep it that way.

6.8 Testing a producer

The Inbox needs no app instance to test. Mirror the existing tests:

- `Facet.Tests/InboxTests.cs` — deposit→drain→ complete round-trip, multi-target entries, claim-once (a second drain sees nothing), oldest-first ordering, malformed-drop quarantine, empty-target rejection, missing-inbox. All run against a temp folder.
- `Facet.Tests/AddLinkCoordinatorTests.cs` — staging round-trip in selection order and claim-once, via `StagingDirOverride`.
- **Manual end-to-end:** install the verb, select one and several files in Explorer, *Add to Facet*, and confirm a single dialog and the right number of items — once with the app open (live pickup) and once with it closed (pickup on next open).

6.9 Source map

File	Role
`Inbox.cs`	The drop-folder queue: `Deposit` / `Drain` / `Complete`, plus `InboxEntry` and `ItemSource`. **The reusable core.**
`ShellIntegration.cs`	Registers / removes the Explorer right-click verb (per-user registry).
`AddLinkCoordinator.cs`	Coalesces the per-file process burst Explorer spawns for a multi-file selection.
`AddToFacetDialog.xaml.cs`	The capture dialog; builds the `InboxEntry` list to deposit.
`App.xaml.cs`	Satellite-mode entry points: `--add-link`, `--install-shell`, `--uninstall-shell`.
`Workspace.cs`	The consumer side: `FileSystemWatcher`, `DrainInboxNow`, `AddItemFromInbox`.
`Facet.Tests/InboxTests.cs`	Round-trip, claim-once, ordering, quarantine.
`Facet.Tests/AddLinkCoordinatorTests.cs`	Staging round-trip and claim semantics.

§ 7. The images folder, and Facet packages

Bundled images

By default, when you insert an image into an item's notes, Facet references the original file wherever it happens to live (an absolute `file:///` path). Turn on **Bundle images with the database** (File → Properties) and from then on each inserted image is **copied** into a sibling folder and referenced *relatively*, so the database and its pictures stay together.

- **Folder name:** the database file's stem plus `-images` — `Work.json` → `Work-images\`, beside the database.
- **Reference in notes:** a relative Markdown image link into that folder, with spaces percent-encoded, e.g.

  ```
  ![chart](Work-images/q3%20chart.png)
  ```

- **File types:** any image the OS can render (PNG, JPG, GIF, WEBP, …) — there is no whitelist. The original is copied, not moved; a name clash with *different* content gets a `(2)`, `(3)` suffix, while an identical re-insert reuses the existing copy.

If you are hand-editing notes or moving files, the rule is simply: a relative `<stem>-images/...` reference resolves against the folder beside the database. Keep the folder's name in step with the database file's stem and the images keep showing.

(Defined in `Workspace.cs` — `ImagesFolderName`, `ImagesDir`, `ResolveImageDestination`.)

Facet packages (`.facpub`) — moving a database as one unit

Because a database can have an inbox and an images folder beside it, copying just the `.json` can leave pictures or queued captures behind. **File → Publish as Facet package…** packs the lot into one `.facpub` file, and **File → Create from package…** rebuilds it under a new name.

A `.facpub` is **an ordinary zip with a Facet-specific extension** — the extension is what lets the OS open it in a Facet app rather than a generic archive tool. (Packages published by older builds carry a `.zip` extension and still open; the loader keys on the contents, not the name.) The archive uses **fixed internal names** so it is independent of the source file name:

```
Work.facpub
    database.json          ← the database (always this name inside the package)
    facet-package.json     ← manifest: original name + images-folder name
    images/…               ← contents of the <stem>-images folder (if any)
    inbox/…                ← contents of the .inbox folder (if any)
```

On **Create from package**, Facet writes `<newname>.json`, extracts `images/` to `<newname>-images\` and `inbox/` to `<newname>.json.inbox\`, and **rewrites the relative image references** in every note from the old folder name to the new one so the bundled pictures still resolve. It refuses to overwrite an existing database of that name.

This is the format to target if you want to *produce* a portable Facet bundle from another tool: write `database.json`, a `facet-package.json` manifest, and optional `images/` and `inbox/` folders into a zip, and name it `.facpub`. The manifest's `imagesFolderName` is what the rename step keys off.

Facet Reader opens a `.facpub` read-only — it unpacks the package to a throwaway working copy and never modifies the source file. That read path is `DatabasePackage.OpenReadOnly`, the no-rename, no-persist sibling of `CreateFrom`.

(Defined in `DatabasePackage.cs` — `Publish`, `CreateFrom`, `OpenReadOnly`, and the `Manifest` shape.)

---

§ 8. The settings file

App-level preferences — the things that belong to *you* rather than to any one database — live in a single JSON file, separate from every database:

```
%LocalAppData%\Facet\settings.json     (or beside the exe, in portable mode)
```

It is small, camelCase JSON:

```
{
  "databasePath": "C:\\Users\\you\\Documents\\Work.json",  // current/last database (file or folde
  "recentDatabasePaths": [                                  // Open-Recent list, newest first, max
    "C:\\Users\\you\\Documents\\Work.json",
    "C:\\Users\\you\\Documents\\Home.json"
  ],
  "backupFolderPath": "",          // "" = back up beside each database
  "quickCaptureEnabled": true,     // the Ctrl+Alt+N global hotkey
  "dateDisplay": "Iso",            // Iso | Usa | Eur | Custom — a display preference
  "mainWindowLeft": 100.0,         // last main-window placement
  "mainWindowTop": 50.0,
  "mainWindowWidth": 1280.0,
  "mainWindowHeight": 800.0,
  "mainWindowMaximized": false
}
```

Useful to know:

- `databasePath` / `recentDatabasePaths` are the read-only source of truth for "which database(s)" — exactly what a custom Inbox producer reads to find where to drop a capture (section 6).
- **Settings are per-machine, not per-database.** They do not travel in a Publish-as-Zip bundle; the recipient's own settings apply.
- This file is safe to read at any time. As with any file, only hand-edit it while Facet is closed.

Beside `settings.json` you may also see an `addlink-staging\` folder — a short-lived working area the Explorer "Add to Facet" command uses to coalesce a multi-file selection. Its `<ticks>-<guid>.path` files are transient and clean themselves up; you can ignore it.

(Defined in `Settings.cs` — the `Settings` class.)

---

§ 9. The complete on-disk map

Putting it together, a typical installed setup:

```
%AppData%\Facet\                          (default database folder — Roaming)
    db.json                               the default database
    db.json.inbox\                        its capture queue (hidden)
    db-images\                            its bundled images (if enabled)
    Facetbackups\
        db-2026-06-07.json                a daily backup
        db-pre-rules-migration.json       a one-off pre-migration backup

%LocalAppData%\Facet\                     (app settings — Local)
    settings.json                         your preferences + recent databases
    addlink-staging\                      transient Explorer-capture scratch

<anywhere you keep one>\                  (databases live wherever you put them)
    Work.json
    Work.json.inbox\
    Work-images\
    Facetbackups\
        Work-2026-06-07.json
```

File / folder	Pattern	Holds
Database	`<name>.json`	The whole database (items, structure, settings)
Daily backup	`Facetbackups\<name>-YYYY-MM-DD.json`	One copy per calendar day
One-off backup	`Facetbackups\<name>-<label>.json`	Pre-migration safety copy
Inbox	`<name>.json.inbox\` (hidden)	Queued items from outside producers
Inbox drop	`<ticks>-<guid>.json`	One pending item
Images	`<name>-images\`	Note images, when *Bundle images* is on
App settings	`%LocalAppData%\Facet\settings.json`	Per-user preferences + recent list
Portable marker	`Facet.portable` (beside the exe)	Switches to self-contained mode
Facet package	`<name>.facpub`	Database + inbox + images, for transport (a zip inside; opens in Facet or Facet Reader)

§ 10. A checklist for working with the files safely

1. **Reading is always fine.** Back up, parse, or copy any file at any time.
2. **Edit the database only with Facet closed**, and copy it first. The `Facetbackups` folder is full of ready-made copies if you need one.
3. **To add items from another program, write a drop file to the Inbox** — never write the database. Follow the temp-then-rename, `<ticks>-<guid>.json` protocol in section 6.
4. **Find the target database** read-only from `settings.json` (`databasePath` , `recentDatabasePaths`).
5. **Generating items? Give each a fresh** `id` , use category *paths* (and add any missing ancestor categories), and set a `source` so they're traceable.
6. **Moving a database with its companions? Use Publish as Facet package**, or copy the `.json` , its `.inbox\` , and its `-images\` folder together — and keep the images folder's name in step with the database stem.
7. **Edit with the current Facet version.** Letting an older build save a newer database can drop fields it didn't understand.

---

§ 11. Extending the layout

Facet's on-disk design is meant to grow without breaking older files. Two places are explicitly built as extension points:

- **The database schema is additive.** New versions add fields with safe defaults and bump `schemaVersion`; old files load unchanged, and unknown fields are tolerated rather than fatal. New data rides alongside the old.
- **The Inbox is the integration seam for new capture sources.** Any new way of getting an item into Facet from outside — a browser extension, a share target, a CLI, an email gateway — is a *producer* that drops `InboxEntry` files. It needs nothing on the app side: the running app already drains any inbox for the database it has open, and the next session drains on open. Declare a new `source` value, follow the drop-file protocol, and you've added a capture channel without touching the database or the app.

Section 6 is the full reference for that seam — the claim/quarantine protocol, the guarantees and why they're shaped the way they are, the Explorer command as a worked example, and a checklist for building a new producer. Start there if you want to write one.

This guide describes Facet's on-disk layout as of schema version 27. The shapes shown are stable and additive, but treat field-level detail as a snapshot — the source files named throughout are the authority.